* Introducing .NET Health Checks
  + We are going to use the Microsoft.AspNetCore.Diagnostics.HealthChecks package to add health checks to our application
  + This will allow our front-end to monitor the status of our back-end server
  + This is added easy enough by adding the HealthChecks to our services then adding it to the app (before MapControllerRoute so that the middleware doesn’t look for the health endpoint as a general controller)
  + The default check here simply reports Healthy since we have not added any checks to tell it otherwise
* Adding an ICMP Check
  + Internet Control Message Protocol (ICMP) request check is also known as a PING
  + This checks the presence and availability of a server that should be available in a LAN or WAN
  + One machine sends the ICMP echo request packets to the target host and waits for a reply, reporting the round-trip time
  + We are going to add a health check using these messages with statuses of Healthy, Degraded, and Unhealthy
  + Creating an ICMPHealthCheck Class
    - This is just a class that implements IHealthCheck and its method CheckHealthAsync()
    - This method returns a HealthCheckResult, in our case based on the result of a Ping.SendPingAsync() method call
  + Adding ICMPHealthCheck to the Pipeline
    - To add our own health check to the pipeline, we need to register our class with the HealthChecks middleware
    - This is done with the AddCheck<HealthCheckClass>(“GoodName”) method on the service registration AddHealthChecks() method
  + Improving the ICMPHealthCheck Class
    - We make the host and timeout parameters properties and have them added via ctor
    - We added some better messages that are reported as part of the health checks
  + Updating Middleware Setup
    - Now we need to add these new required parameters to the ICMPHealthCheck constructor
    - We can do this by using the services AddHealthChecks().AddCheck(“GoodName”, new MyHealthCheck())
    - Note that the default messaging behavior will still be to display one message based on the sum of all the health checks
    - We need to change this default behavior
  + Implementing a Custom Output Message
    - To do this we add a new type inheriting from HealthCheckOptions
    - The base type includes a ResponseWriter property that we will set in the ctor to implement our desired behavior
    - We supply an anonymous async method that takes an HttpContext and HealthReport as params and returns a Task
    - We hard-code the status code of the HttpContext to a 200 as the default behavior is to return a 200 if all checks are Healthy and a 503 otherwise
      * We don’t need this now as we are returning a more detailed JSON report
    - We also set the result as a serialization of an anonymous object with a checks element populated from the HealthReport entries
    - We configure the app to use these options by adding it to the application UseHealthChecks(“route”, options) method
* Health Checks in Angular
  + We are now going to build an Angular component that can fetch and display the health check JSON data that we have made available
  + This will require creating
    - The **Component** (\*.ts) file containing the Component class and the module references, functions, variables, etc.
    - The **Template** (\*.html) file written in HTML extended with the Angular Template Syntax defining the UI layout architecture
    - The **Style** (\*.css) file containing the cascading style sheet rules and definitions for drawing the UI
  + Here the template represents the view part of MVVM/MVC and the component represents the controller/view model
* Creating the Angular Component
  + health-check.component.ts
    - At the top of the file, we import all the Angular directives, pipes, services, and Components (i.e., all the Modules) that are needed in the class
    - We define a Component decorator
    - We create a constructor for the component, with an HttpClient instance and an injected base URL
    - We also define two interfaces that are used to map the JSON response to a strongly typed object
    - Imports and Modules
      * The import statement is used to import bindings that are exported by other JS modules
      * Modules have been around since ECMAScript 2015 and have been adopted by TS and therefore Angular
      * They are basically a collection of variables, functions, classes, etc. grouped in a single class
      * A module is executed in its own scope (as opposed to global scope)
      * So, elements in the module are not available to outside the module unless they are explicitly exported
      * Note that these JS modules should not be confused with Angular’s modularity via NgModules (they are different concepts)
      * These two systems use similar vocab unfortunately (import vs imports, export vs exports), but they are used in different contexts
    - Dependency Injection
      * A dependency is a service or object that a class needs to instantiate into a variable or property
      * In classic patterns, this is done by creating an instance of the object in the class itself
      * DI works by having a Dependency Injector handle creating injectable objects and injecting them into classes that ask for them
      * This pattern is based on the concept of Inversion of Control (IoC) where a class requiring an object does not create the object directly, but finds a way to inject an instance of the object
      * This is done in Angular via the @Inject decorator
      * Also note that we use variable scoping here to declare both the variables in our constructor as private
        + By default, variables in a ctor are only available in the ctor itself
        + Variable scoping with public or private makes the variables usable throughout the component
    - ngOnInit (and Other Lifecycle Hooks)
      * ngOnInit is one example of an Angular lifecycle hook method
      * When a user visits a view in our app, Angular creates and renders the required Components and directives (and their children)
      * As the user interacts with the components they change, and they are eventually destroyed when the user navigates away
      * All of these key moments trigger various lifecycle hook methods that Angular exposes to the dev
      * Devs can use these so that their components can do things when they fire; these are similar to C# event handlers
      * These include
        + **ngOnChanges**: responds when (re)sets data-bound input properties; called before ngOnInit() and whenever one or more data-bound properties changes
        + **ngOnInit**: initializes directive/Component after Angular first displays data-bound properties and sets the input properties; called once after the first ngOnChanges()
        + **ngDoCheck**: detects and acts on changes that Angular cannot or will not detect otherwise; called during every change detection run, after ngOnChanges and ngOnInit
        + **ngAfterContentInit**: responds after Angular projects external content into the Component’s view or the view that the directive is in; called once after ngDoCheck
        + **ngAfterContentChecked**: responds after Angular checks the content projected into the view; called after ngAfterContentInit and after every ngDoCheck
        + **ngAfterViewInit**: responds after Angular initializes the view and child views; called once after the first ngAfterContentChecked
        + **ngAfterViewChecked**: responds after Angular checks the view and child views; called after ngAfterViewInit and after every ngAfterContentChecked
        + **ngOnDestroy**: cleans up right before Angular destroys the directive/Component; unsubscribes Observables and detaches the event
        + **Additional Handlers to Avoid Memory Leaks**: called before Angular destroys the directive/Component
      * All of these methods are available to all Angular components and directives
      * To call these methods we add them to the Component class