* Fetching Data
  + The Angular frontend is going to request information via HTTP requests to the .NET Controller backends
  + Requests and Responses
    - JSON Conventions and Defaults
      * Note that by default .NET Core MVC middleware will make several conversions to the output
      * E.g., properties will be converted from PascalCase to camalCase, and there will be no CRLF (single line)
  + A Very Long List
    - We are going to create a new component that fetches all the cities from the database
    - Note that in the component template (the HTML) attribute we use a new **[hidden] attribute** on the table of cities
    - The [hidden] Attribute
      * The **hidden** attribute is a valid HTML-5 sttribute that can be used to hide content from the user (similar to CSS **display: none**)
      * The square brackets around the attribute are Angular syntax for a property binding
      * This defines an HTML property or attribute that gets its value from the Component
      * Note this binding **only flows one way** (from the Component to the Template)
      * But, couldn’t we have instead used the \*ngIf directive to achieve the same thing?
      * Sort of, the difference is that \*ngIf will add or remove the element from the DOM; this means that the element and its children will be initialized and disposed of every time the conditional changes
      * The hidden attribute will only hide an element from the user; it will still be available in the DOM for manipulation (e.g. by JS)
      * We use the hidden attribute here because the table we have created will soon be a much more complex object
* Serving Data with Angular Material
  + We need to make the UI in the cities table better
  + We are going to do this via the Angular Material UI Component library that implements Material Design
  + Material Design is a UI design language created by Google and used throughout their software offerings
  + Angular (being a Google product) offers an npm package with native UI elements, the Component Dev Kit (CDK), some animations, etc.
  + This can be installed by running **ng add @angular/material** from the client root folder, which will get a compatible version of the npm package and update several files for you
  + MatTableModule
    - So we start by importing the MatTableModule from the material package in a separate, new ng module
    - We do this, then import the module in the app.module.ts to import the necessary MatTableModule globally
    - We then modify the template for our city table to use the material design
    - This is done by creating a mat-table type of table and pointing to the cities property as the dataSource for the table
    - We then define an ng-container for each column we want to represent in the table
    - The matColumnDef identifies which column in the overall column definition this container is pointing to
    - The matHeaderCellDef defines the name of the material design column
    - The matCellDef uses a let statement to define the object being passed to it, then can use the data in that object in the actual data cell
    - At the bottom of the table definition, we have a mat-header-row with a matHeaderRowDef pointing to a list of strings that defines the order of the columns based on their matColumnDef names
    - Finally, the mat-row row has a matRowDef that defines how we will loop over the data source for rows
  + MatPaginatorModule
    - Client-Side Paging
      * We want to add pagination to this project to help view the mass crap stack of data we have
      * This can be added to the MatTableModule using the MatPaginatorModule
      * From this module we import the MatPaginator and we import MatTableDataSource, and we also import ViewChild for use in a minute
      * This data source class will be used for the table data source directly, and this also has a paginator property that we will set
      * We create a new Paginator instance in the component, and instantiate is as a view child from the paginator that we define in the template
      * This component paginator is used to set the data source paginator after instantiating the data source from the cities JSON
      * But this doesn’t actually reduce the amount of data that we need to sift through, since the API still returns all the things
    - Server-Side Paging
      * This requires updating the controller to support the paging responses, updating the client to support making multiple requests, and adding a new ApiResult class to improve controller response performance
      * API Controller
        + Adding the actual pagination in the server is actually pretty easy
        + We just add optional variables to the controller GET method for a page index and page size
        + Then we use the DbSet.Skip() method to skip a number of pages, and use the DbSet.Take() method to only take a small number
        + This is useful, but the client will not know the other information required to properly paginate this data

How many total pages there are

How many results are on each page

What page we are currently looking at

* + - * + To provide this information, we are going to implement our own ApiResult type
        + The ApiResult contains a static method used to construct a new instance
        + This type contains a Data property that contains the paged data, a TotalCount with the total number of items, a TotalPages with the number or pages (TotalCount / PageSize), and other standard properties
        + It also has two basic methods to aid with paging: HasNextPage() and HasPreviousPage()
        + Note that we want this class to be instantiated in an async way, and since ctor’s can’t be async, we use a static factory class to do this
      * Angular Component
        + We now need to update the component logic to utilize the new data available in the ApiResult
        + We update our logic to get data from the controller using information in a PageEvent
        + The PageEvent includes properties on the page size and index that are used to set HttpParams for the new http call
        + We also update the subscription to explicitly set the paginator length, index, and page size, then extract the data and update the component dataSource
        + On initialization, we call this method with some default values (zero index, 10 items)
        + We then hook into the page event on the paginator to call getData for any page size or index change in the paginator
      * Notes on the ViewChild Decorator
        + @ViewChild can be used to get a reference to a DOM template element within an Angular Component
        + This is done with a selector of some kind; this can be a class name (if it has an @Component or @Directive decorator) a template variable, etc.
  + MatSortModule
    - Server-Side
      * We implement this in the ApiResult via some LINQ sorting
      * We have a sort column and a sort order and add these as properties to the ApiResult
      * But, because we are basing our LINQ sorting off of user-provided data, we need to check that the sort column exists
      * This is primarily a security measure to prevent e.g. SQL injection attacks
      * In order to do this we need to utilize the System.Linq.Dynamic.Core nuget package that enables dynamic LINQ (with strings) on an IQueryable
      * This is necessary because standard LINQ works with a strongly-typed approach (i.e. when we query an object, the source type and all the properties must be known by the compiler)
      * So standard LINQ can’t work e.g. with generic objects or generics in general (type <T>’s)
      * The dynamic library allows up to write expressions with literal strings and have them translated to strongly-typed equivalent’s via **reflection**
      * This is good, but is also not less testable, more prone to major crashes due to simple string literal errors, and open to e.g. SQL injection attacks
    - Client-Side
      * We implement sorting in Angular using the MatSort class from the MatSortModule
      * This class uses the active and direction properties to set the sorting on the table elements
      * Active is the element that is actively being sorted and direction is the direction of the sort (asc or desc)
      * We add the matSort attributes to the table element to make this a sorting table
      * We set the default sorting on initialization, then on any sorting change the matSortChanged event fires
      * So we bind our load method to the matSortChanged event which uses the sort class in the template to get the active and direction sort properties and sets the HTML params based on this
      * We also add the mat-sort-header attribute to the mat-header-cells which I assume allows us to actually sort on these headers
  + Adding Filtering
    - Note that there is no Angular Material module that we can use to achieve filtering
    - Instead we will implement the logic ourselves in the controller and front-end
    - Controller and ApiResult
      * We add optional parameters to the ApiResult for a filter column and filter query
      * Because we allow for filtering on any column, we are again using dynamic LINQ to create our LINQ on the fly
      * As such, we are using the same validation logic to ensure that the filter column exists
    - Angular Component
      * We are going to use the MatInput material input box from the MatInputModule because it is better than the standard input
      * We add an optional event to the loadData() function that we will use to pass key argument events for filtering
      * We bind the keyup event in the MatInput to this function, extract the text from this box, and pass as an HTML parameter
      * Note that I want to show the loading icon during these filter requests so that the intermittent results are not confusing
      * Because these events are asynchronous, we are using a count of the requests to determine when all the requests have been processed
      * We only show the results when the count drops to zero
* Countries Component
  + We do basically the same thing for the countries data set