* Exploring Angular Forms
  + Forms in Angular
    - Note that the **mat-form-field > input** element that we use to allow users to filter data is a way for a user to interact
    - But, if you look at the generated HTML, you will only see an **input** element (no **form** element)
    - This is because you don’t need form data to submit this request; it is done via an async JS XMLHttpRequest (XHR) (i.e. AJAX)
    - This doesn’t require a form container to handle data encoding and transmission
  + Reasons to Use Forms
    - Even if we don’t need a form to handle encoding/transmission, there are other advantages
    - Specifically, there are disadvantages to using a single input element and a text string
      * Can’t keep track of global form state
      * Can’t easily display an error message to users if a form is invalid
      * We are not validating the data in any way
    - Note that we could add these features manually in our Components and Angular directives like \*ngIf and \*ngFor, but there are easier ways
    - Specifically, Template-Driven Forms and Client-Driven Forms (aka Reactive Forms)
  + Template-Driven Forms
    - These forms are defined mainly in the Template for your component (big surprise)
    - They involve defining a form element and using the ngForm module
    - You define inputs in the form and define attributes (per normal) and use the [()] two-way property binding syntax to ensure the forms are responsive
    - Note that [(ngModel)]=”city.Name” is really Angular shorthand for [ngModel]=”city.Name” and (ngModelChanged)=”city.Name = $event”
    - Pros
      * These forms are easy to write (especially from an HTML knowledge standpoint)
      * These forms are easy to read and understand if you have a decent background in HTML
    - Cons
      * These require a lot of HTML which is difficult to maintain and more error-prone than pure TS
      * These forms cannot be unit tested; they require E2E tests with a browser to e.g. verify the validation logic
      * Readability will quickly drop as you add more and more validation and logic
    - Generally, Template-Driven is a good option when the forms are small and the validation logic is light
  + Model-Driven/Reactive Forms
    - These types of forms were introduced in Angular 2 to combat the cons of Template-Driven forms
    - The main difference is the amount of HTML required in the template is significantly less
    - In the Component you define a FormGroup element that represents the whole form and encapsulates related controls
    - Each individual data input is represented by a FormControl element within the FormGroup (either the parent or a child)
    - The FormControl tracks data related to the control’s current state and the actual control value
    - The FormGroup tracks the state of each child control; so the group is only valid if all the children are valid
    - The powerful difference here is in where the control lies
      * For template driven, we don’t really have control over the validation or submission logic once it’s out there; it’s written in the HTML and there it will stay
      * For model-driven/reactive, the HTML updates parts of a strongly-typed FormGroup and then we operate on that data directly in the component; giving us much more control
* Building Our First Reactive Form
  + CityEditComponent
    - So this is going to be a form that allows us to edit a city instance
    - To do this we are going to create a new component (new TS, CSS, and HTML files) for a **city-edit** component
    - For this component we will define a parent FormGroup that will be what the use interacts with, and a City property that will hold the actual data
    - Note the FormModel will only represent the data that we want the user to be able to edit (e.g. not the primary ID)
    - When the page is loaded and onNgInit() fires we create the new FormGroup with the 3 FormControl’s for the editable fields
    - We then get the city ID from the activated router and use this to get the city instance we need
    - We set the local City variable based on this result and then use FormGroup.patchValue() to set the form values from the result
    - On form submission we get the values of the form (I assume from the formControlName attributes in the Template), set our local City instance with them, and use those values in a PUT request to the backend
    - After submission we use the router to navigate back to the main cities page
    - Adding the Navigation Link
      * So per usual we import the Component in our app.module.ts file and we need to add a router link to navigate to the page
      * This path for this router link though uses different syntax to ensure the City ID is set in the route parameters
      * Specifically it uses path: ‘city/:id’
    - Then we update the main cities list view to have it so if you click on the ID you navigate to the CityEditComponent
    - This is does via an anchor tag binding routerLink to [‘/city’, city.id]
* Adding a New City
  + Note that a common requirement of a Detail View with editing capabilities is to be able to create a new city from the same view
  + Note that running this currently will result in a foreign key exception since we aren’t setting the ID for the Country
  + When we were updating the city via the edit form this was silently happening in the background
  + During edit, when we grabbed the results and stored in locally, we were STORING THE FULL OBJECT
  + We applied a TS interface to the object, but of course it can have additional data that isn’t exposed by that interface
  + So, when we PUT the value, we are actually sending the object back with its original country ID
  + We need to add a dropdown list for users to select which country they want (for new cities) and if they want to change the country of existing cities
  + HTML Select
    - Note that we can do this pretty easily via our standard practices plus an HTML select element
    - We fetch all the countries from the DB and will show a list of names and set the option values from the country ID’s
    - We just need to create a new div and use a select element instead of an input element in the div
    - Then we loop through the countries and populate the select as needed
  + Angular Material Select (MatSelectModule)
    - The MatSelectModule provides a better way to present an option list to the user
    - This requires only a bit more sophistication, but results in a much nicer UI with scrolling and such
* Understanding Data Validation
  + Obviously front-end client validation is necessary for any app requiring data input from users via forms
  + Template-Driven Validation
    - Note that we aren’t using template-driven forms, but knowing their validation is still helpful
    - Angular uses directives that match HTML validator functions
    - When the form changes, Angular will run these functions and generate a list of validation errors
    - This can be ran using the \*ngIf directive (they are data validation directives here)
    - We two-way bind the property that we want to validate in the input, then we choose to show errors based on the value of this property
  + Model-Driven Validation
    - Pretty much all the validation logic for model-driven forms is handled in the Component logic
    - We can create either async or sync validators that either return a set of validation errors or null
    - They are set up using the second and third arguments when we create the FormControl’s that the validators are meant for
    - Note that async validators are run after sync validators, and only if the sync validators pass
    - This is done for performance reasons
    - On ever change we execute the function in e.g. the \*ngIf directives; we tie these into the logic to check if the field is invalid and which error it has
    - We do this by getting the form field control and checking various stats on the field
  + Server-Side Validation
    - Client-side validation is better in terms of performance when letting users know that their input is incorrect quickly
    - But, we need server-side validation to handle
      * Implementation Errors on the client-side validation
      * Client-Side Hacks that allow malformed data to be sent to the backend despite rules
      * Request Forgery
    - Note that server-side validation is actually the requirement here; client-side validation is just a nice-to-have
    - Server-side validation is typically more complex since you require both the back-end validation logic and front-end logic to display the resulting errors
    - We are going to create a custom validator to ensure we aren’t trying to add a city that already exists
    - This means checking if there is a city present that has the same name, lat, and lon in the DB
    - To do this we are going to create a custom validator that asynchronously calls to the DB to check if the proposed values are valid
    - DupeCityValidator
      * To create this we define an isDupeCity() function that returns an AsyncValidatorFn type
      * This type of function takes an AbstractControl type and returns either an Observable that resolves to a map of string to any, or it returns null
      * Within this function we create a temp City variable based on the name, lat, lon, and countryId currently in the form
      * For the ID we either use the current ID or set it to zero explicitly
      * We do that so that we can check if the city ID we are modifying aligns with a city with parameters that also match what we are looking at
      * This is so that, when we are modifying an existing city we don’t get an erroneous message that the city already exists (which in that case it obviously would)
      * We use this validator to be used by the main FormGroup (instead of the individual FormControl’s)
      * Note that we do not explicitly subscribe to the return from our HTTP call
      * We use a combination of the ReactJS (RxJS) pipe operator and the map function to map the result to the required return type for the AsyncValidatorFn
    - Observables and RxJS Operators
      * Note there is a significant difference between the subscribe() function and the pipe + map approach for handling Observables
      * The subscribe() function will actually execute the Observable and get the result; this will return a Subscription that can be canceled, but cannot be subscribed to again
      * The map() operator should be used when we want to manipulate the result of the Observable without executing it; this allows it to be passed to other async actors that can also manipulate it and eventually execute it
      * The pipe() operator is just an RxJS operator that composes/chains other operators (e.g., map, filter)
      * This is relatable to how the IQueryable<T> interface types in the .NET EF logic works
      * We can manipulate IQueryable<T> interface types without actuallyexecuting the query; eventual calls to ToListAsync(), ToArrayAsync(), etc. will actually execute the query
    - Performance Issues
      * We should consider if there will be performance issues since the duplication validation makes server-side validation calls on each change
      * But, because synchronous validators are called first, we will only call this async validator when the rest of the form is already valid
      * This means that we save a lot of calls that would otherwise be made on the server side
* Introducing the FormBuilder
  + Instead of repeating the same logic to make a country editing form, we will use the FormBuilder tool
  + This this uses a FormGroup element, the main difference is in how we create the FormGroup
  + Instead of manually defining the FormControls, we use the FormBuilder.group() function with an object composed of the names of the controls as the properties and a list of form state, sync validators, and async validators
  + The actual FormGroup is automatically constructed off of this
* CountryEditComponent
  + This is pretty much the same as the CityEditComponent functionally, with just minor differences that are specific to the City/Country properties