Forms

Introduction to Angular 6

Form Overview

- Creating and using forms in Angular will involve
  - data binding (both from UI to the code, and vice versa)
  - form state tracking
  - validation and error handling.
- Angular provides two different approaches to handling forms:
  - reactive and template-driven.
- Both capture user input events from the view, validate the user input, create a form model and data model to update, and provide a way to track changes.

Reactive vs Template-driven

- Reactive forms are more scalable, reusable, and testable.
  - If forms are a key part of your application, or you're already using reactive patterns for building your application, use reactive forms.
- Template-driven forms are useful for adding a simple form to an app.
  - They are easy to add to an app, but they do not scale as well as reactive forms.
  - If you have very basic form requirements and logic that can be managed solely in the template, use template-driven forms.

Reactive vs Template-driven

	REACTIVE	TEMPLATE DRIVEN
Setup (form model)	More explicit, created in the component class.	Less explicit, created by the directives
Data model	Structured	Unstructured
Predictability	Synchronous	Asynchronous
Form validation	Functions	Directives
Mutability	Immutable	Mutable
Scalability	Low-level API access	Abstraction on top of APIs

Form building blocks

- Both reactive and template-driven forms share underlying building blocks.

- A FormControl instance that tracks the value and validation status of an individual form control.
- A FormGroup instance that tracks the same values and status for a collection of form controls.
- A FormArray instance that tracks the same values and status for an array of form Controls

Reactive form features

- The form model provides the value and status of the form element at a given point in time.
  - usually through the FormControl instance.
- Updates from the view to model and model to view are synchronous
  - The developer has absolute control over how and when the data is synced from the view to the model and vice versa.
- The entire tree of form control objects is defined in component code
  - Subsequently bound to native form control elements in the template.
  - Accessible immediately without needing to go through Angular's asynchronous life-cycle.

Template driven form features

- Directive NgModel is responsible for creating and managing the form control instance for a given form element.
  - You no longer have direct control over the form model.
- Angular is responsible for the data model sync
  - pushes data to the model and reads and updates values in the UI via directives like ngModel.
- Template-driven forms are nice and declarative, and easy to understand.
- There is less code required in the component class compared to reactive forms.

Distinction between view and model

- Advantage of using reactive forms over template-driven forms
- forces developers to have a separation between what the user interacts with (the form model), and the persisted data model that drives the application.
- This is quite common in most applications, where the presented view is different from what the underlying data model is.

LAB

ngModel

- provides two-way data binding between elements in the UI and properties in the component
  - abstracts away the internals of each and every input type that needs to be bound.
  - needs a name field to the input form element to work.
- We can use a simplified banana-in-a-box syntax to provide two-way data binding:
  - The banana-in-the-box syntax only has the capability to set the data-bound property.
- If you need to do something more complicated, or set it in a different field itself, or do multiple things, then you might want to consider the expanded syntax.

Form validation and control state

- Angular form validation for template-driven forms relies on and extends the native form validation from HTML.
- Angular does the work of integrating these control states and validations with its own internal model
  - This is used in turn to display appropriate message to user.
- There are two aspects to this:
- The state of the form control
  - whether the user has visited it, whether the user has changed it, and finally whether it is in a valid state.
- The validity of the form control
  - whether a form control is valid or not, and the underlying reason for which the form element is invalid.
- The ngModel directive changes and adds CSS classes to the element it is on

Form Control State

Control state	CSS class if True	CSS class if False
Visited	ng-touched	ng-untouched
Changed	ng-dirty	ng-pristine
Valid	ng-valid	ng-invalid

Angular validators

- Internally, Angular has its own set of validators.
  - These mirror the HTML Form validators that are then used to drive a similar behavior
- Once you add any validators to your form elements, Angular will take care of running them every time any of the form control changes.
- This would then be reflected at each control level, as well as at an aggregate level at the form

Template reference variable

- allows us to get a temporary handle on a DOM element, component, or directive directly in the template.
  - denoted by a standard syntax in the HTML, which is a prefix of #.
  - it can also be used to reference the class/value underlying a directive, which is how we use it in conjunction with forms and form fields.
  - can be passed in as arguments to the component class
- Template reference variables can be added at the form level and at each control level
- The form-level template reference variable usually gets the NgForm model object bound to it
  - This allows us to check on things like value of the form model as well validity of individual controls.

Template reference variable

- Most of the default validators will add a corresponding entry on the errors field on the template reference variable
  - you can use this to display a relevant message
  - Need to be aware and handle multiple validators having errors simultaneously, and decide which messages to show conditionally.
- Template reference variables and validators provide flexibility in controlling how and when to show validation messages. You can choose:
  - to do it completely in the template
  - have them in the template, but decide when and how to show them from your component class
  - creating your validation messages and drive it completely from your component class

ngModelGroup

- Use the ngModelGroup directive to group various form elements together under a common name
- When we create forms using the template-driven approach, we declare the form controls in the template, and add directives to it (like ngModel).

Form Control and Form Group

- Form Control directly represents an individual form element in the template.
  - A reactive form is simply a set of grouped FormControls.
- It is at the FormControl level that we assign initial values and validators
  - The first argument to the FormControl constructor is the default value of the form control
  - The second argument to the FormControl constructor can either be a single Validator, or an array of Validators.
- There are a set of built-in validators to ensure that the FormControl has a minimum value.
  - These validators can either be synchronous or asynchronous
- The FormGroup is useful as a way to group relevant form fields under one group
  - Allows us to track the form controls individually, or as a group

Form Builder

- FormBuilder is syntactic sugar to allow us to quickly create FormGroup and FormControl elements
  - The reactive form still relies on those elements under the covers for its functioning
- For any form with more than a few elements, it is better to use the FormBuilder rather than the FormGroup method
  - it reduces both the code and makes it a lot more concise and readable

Reactive form data

- Dealing with form control state and validity is similar to with template-driven forms,
  - base control states and validity are the same
- Good practice to not directly assign the form model to our data model, instead make a copy of it
- For a simple object
  - a simple Object.assign or the spread operator
- for slightly more complicated model
  - do a deep copy