Angular

Created by :
Sangeeta Joshi

Agenda

- Architectural Overview
- Hello World (Angular CLI)

Angular 1 Vs Angular 2

- No $scopes ,No controllers
- Everything is a component

What is Component?

:a self-contained object:

1. which owns its own presentation logic,
2. view,
3. internal state

Example: Button<button>

Advantages of Component Architecture

- Why Component based Architecture ?
  - A component is ***an independent software unit*** that can be composed with the other components to create a software system.
  - Component based web development : ***future of web development***.
  - Reusability
  - allow segmentation within the app to be written independently.
  - Developers can concentrate on business logic only.
- These things are not just features but the requirement of any thick-client web framework.

MVC

- Traditional MVC (n-tiered) Architecture Tries to be Loosely Coupled
- Separation of concern
- App is divided into layers: Model, View, Controller, Service, Persistence, Networking

Problems with MVC

- But MVC has lots of disadvantages:
  - Complexity
  - Fragility
  - Non-reusable
  - Difficult to extend existing functionality

MVVM

- VM : Component class
- View: Template

- Angular 2 is the next version of Google's massively popular MV* framework :
- for building complex applications in the browser (and beyond).

  a faster

  more powerful

  Cleaner

  easier to use tool

  a tool that embraced future web standards

  brought ES6 to more developers around the world.

- Angular 2 is the next version of Google's massively popular MV* framework :
- for building complex applications in the browser (and beyond).

  a faster

  more powerful

  Cleaner

  easier to use tool

  a tool that embraced future web standards

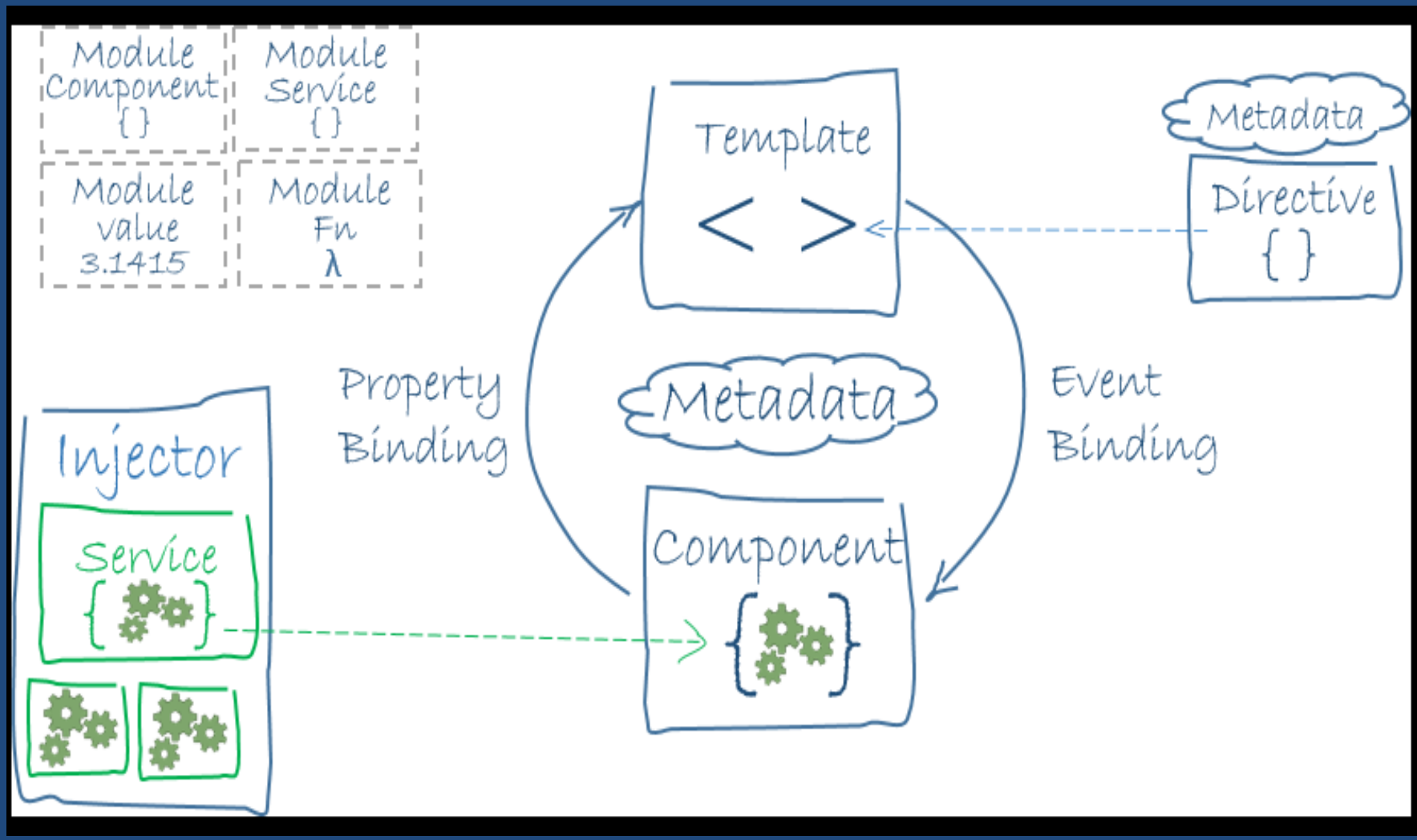
  brought ES6 to more developers around the world.

Architectural Overview

Main building blocks of an Angular application

- Modules
- Components
- Templates
- Metadata
- Data binding
- Directives
- Services
- Dependency injection

Architectural Overview



Overview

- Angular - a framework for building client applications in
  : HTML
  : JavaScript
  or a language like
  : TypeScript that compiles to JavaScript.
- The framework consists of several libraries, core and optional.

- For writing Angular applications involves:
  composing HTML templates with Angularized markup
  writing component classes to manage those templates
  adding application logic in services
  boxing components and services in modules
  Then launching the app by bootstrapping the root module.

Modules

Ngmodules:

- Unit of compilation and distribution of Angular components and pipes.

- the compilation context of its components

- it tells Angular how these components should be compiled.

Modules

- Angular apps are modular
- Angular has its own modularity system called *Angular modules* or *NgModules*
- Every Angular app has at least one module, root module, conventionally named *AppModule*
- While the ***root module*** may be the only module in a small application, most apps have many more ***feature modules***
- An Angular module, whether a root or feature, is a class with an @NgModule decorator

(Decorators are functions that modify JavaScript classes. Angular has many decorators that attach metadata to classes so that it knows what those classes mean and how they should work)

Modules

NgModule is a decorator function that takes a single metadata object whose properties describe the module:

- Declarations
- Imports
- Providers

Bootstrap

Exports

Modules

- Declarations: “View Classes” those belong to this module.

  [view classes: Components,directives,pipes]
- Imports : Other modules needed by components declared in this module
- Providers :
- Bootstrap : It defines the components that are instantiated when a module is bootstrapped
- Exports : Only component mentioned here is added to the compilation context of AppModule

Components

What is a Component?

-A component knows how to interact with its host element.

-A component knows how to interact with its content and view children.

-A component knows how to render itself.

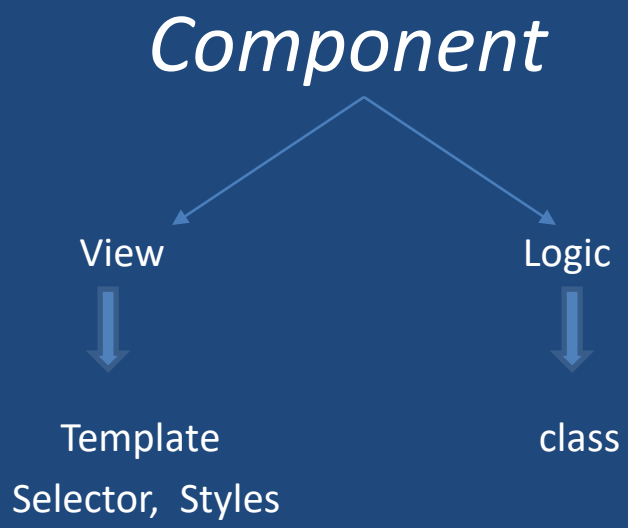
-A component configures dependency injection.

-A component has a well-defined public API of input and output properties.

All of these make components in Angular self-describing, so they contain all the information needed to instantiate them.

Components

- *component* :controls a patch of screen called a *view*



Ex: F:\Demos\Angular2\CLI\MyDemos\my-demo1-app

Components

- A class with component metadata
- Responsible for a piece of the screen referred to as view.
- Template is a form HTML that tells angular how to render the component.
- Metadata tells Angular how to process a class

Components:Templates

Component's template in one of two places:

- *inline using the template property*

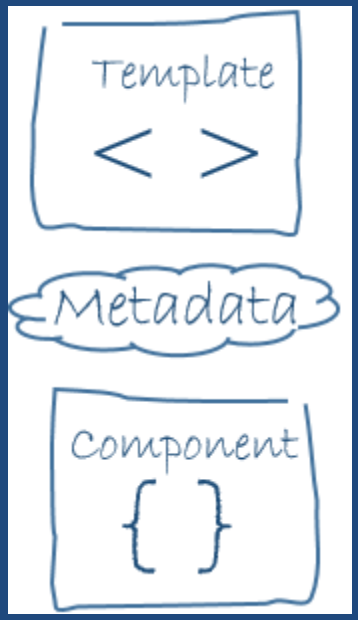
or

- *in a separate HTML file (templateUrl property)*

- The choice between inline and separate HTML is a matter of taste, circumstances, and organization policy.
- In either style, the template data bindings have the same access to component's properties.

@Component

- selector:
- templateUrl:
- providers:



Component

- It is also a type of directive with template,styles and logic part
- In this type of directive you can use other directives whether it is custom or builtin in the @component annotation like following:

```
@Component({
  selector: "my-app"
  directives: [custom_directive_here]
  })
```

- use this directive in your view as:

```
<my-app></my-app>
```

@Input() & @Output

- Input and Output Properties

A component has ***input*** and ***output*** properties, which can be defined:

in the component decorator

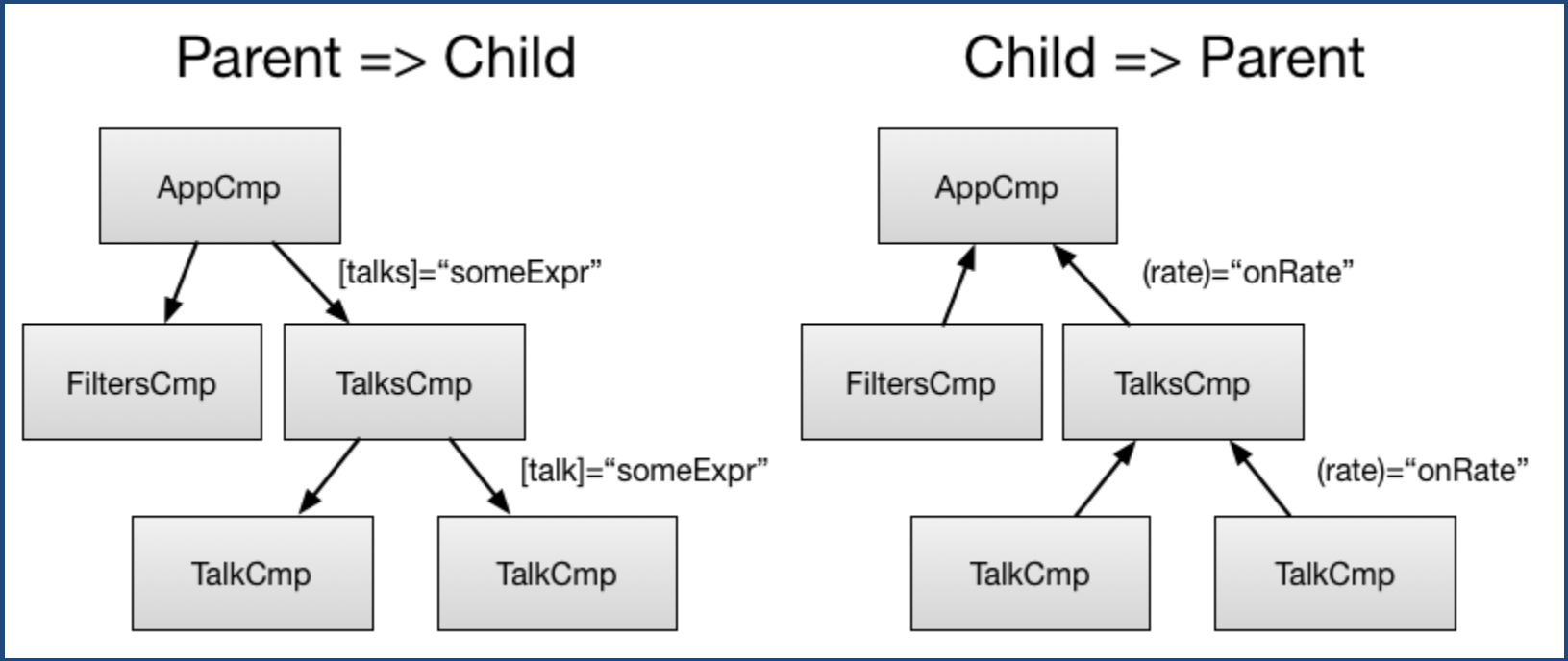
or

using property decorators.

@Component

- Data flows into a component via ***input properties.***
- Data flows out of a component via ***output properties,***
- hence the names: ‘input’ and ‘output’.

@Component


Parent => Child
AppCmp
[talks]="someExpr"
FiltersCmp
TalksCmp
[talk]="someExpr"
TalkCmp
TalkCmp
Child => Parent
AppCmp
(rate)="onRate"
FiltersCmp
TalksCmp
(rate)="onRate"
TalkCmp
TalkCmp

Life Cycle Methods

- Angular calls lifecycle hook methods on directives and components as it creates, changes, and destroys them.
- A component has a lifecycle managed by Angular itself.
- Angular creates it, renders it, creates and renders its children, checks it when its data-bound properties change, and destroys it before removing it from the DOM.
- Angular offers **lifecycle hooks** that provide visibility into these key life moments and the ability to act when they occur.

Component Life Cycle

- A component has a lifecycle managed by Angular.

- Angular :
  creates it, renders it,
  creates and renders its children,
  checks it when its data-bound properties change,
  and destroys it before removing it from the DOM.

- Angular offers lifecycle hooks that provide visibility into these key life moments and the ability to act when they occur.

Life Cycle Methods

- Component lifecycle hooks

- Directive and component instances have a lifecycle as Angular ***creates, updates, and destroys*** them.
- Developers can tap into key moments in that lifecycle by implementing one or more of ***the Lifecycle Hook interfaces*** in the Angular core library.
- Each interface has a single hook method whose name is the interface name prefixed with ng.
- For example, the ***OnInit interface*** has a hook method named ***ngOnInit*** that Angular calls shortly after creating the component:

Life Cycle Methods

- Interface are optional (technically)
- The interfaces are optional for JavaScript and Typescript developers from a purely technical perspective. The JavaScript language doesn't have interfaces. Angular can't see TypeScript interfaces at runtime because they disappear from the transpiled JavaScript.
- Fortunately, they aren't necessary. You don't have to add the lifecycle hook interfaces to directives and components to benefit from the hooks themselves.
- Angular instead inspects directive and component classes and ***calls the hook methods if they are defined***. Angular finds and calls methods like ngOnInit(), with or without the interfaces.
- Nonetheless, it's good practice to add interfaces to TypeScript directive classes in order to benefit from strong typing and editor tooling.

Life Cycle Methods

- Angular calls lifecycle hook methods on directives and components as it creates, changes, and destroys them.
- A component has a lifecycle managed by Angular itself.
- Angular creates it, renders it, creates and renders its children, checks it when its data-bound properties change, and destroys it before removing it from the DOM.
- Angular offers **lifecycle hooks** that provide visibility into these key life moments and the ability to act when they occur.

Life Cycle Methods:Sequence

- First creates a component/directive by calling its constructor,

- Then calls the lifecycle hook methods in the following sequence at specific moments:

Life Cycle Hooks

Hook	Purpose and Timing
ngOnChanges()	Respond when Angular (re)sets data-bound input properties. The method receives a SimpleChanges object of current and previous property values. Called before ngOnInit() and whenever one or more data-bound input properties change.
ngOnInit()	Initialize the directive/component after Angular first displays the data-bound properties and sets the directive/component's input properties. Called *once*, after the *first*ngOnChanges().

Life Cycle Hooks

ngDoCheck()	Detect and act upon changes that Angular can't or won't detect on its own. Called during every change detection run, immediately after ngOnChanges() and ngOnInit().
ngAfterContentInit()	Respond after Angular projects external content into the component's view. Called *once* after the first ngDoCheck(). *A component-only hook*.
ngAfterContentChecked()	Respond after Angular checks the content projected into the component. Called after the ngAfterContentInit() and every subsequent ngDoCheck(). *A component-only hook*.

Life Cycle Hooks

ngAfterViewInit()	Respond after Angular initializes the component's views and child views. Called *once* after the first ngAfterContentChecked(). *A component-only hook*.
ngAfterViewChecked()	Respond after Angular checks the component's views and child views. Called after the ngAfterViewInitand every subsequent ngAfterContentChecked(). *A component-only hook*.
ngOnDestroy	Cleanup just before Angular destroys the directive/component. Unsubscribe Observables and detach event handlers to avoid memory leaks. Called *just before* Angular destroys the directive/component.

Angular 2 Directives

There are three kinds of directives in Angular:

- ***Components***—directives with a template.
- ***Structural directives***—change the DOM layout by adding and removing DOM elements.
- ***Attribute directives***—change the appearance or behavior of an element, component, or another directive.

Angular 2 Directives

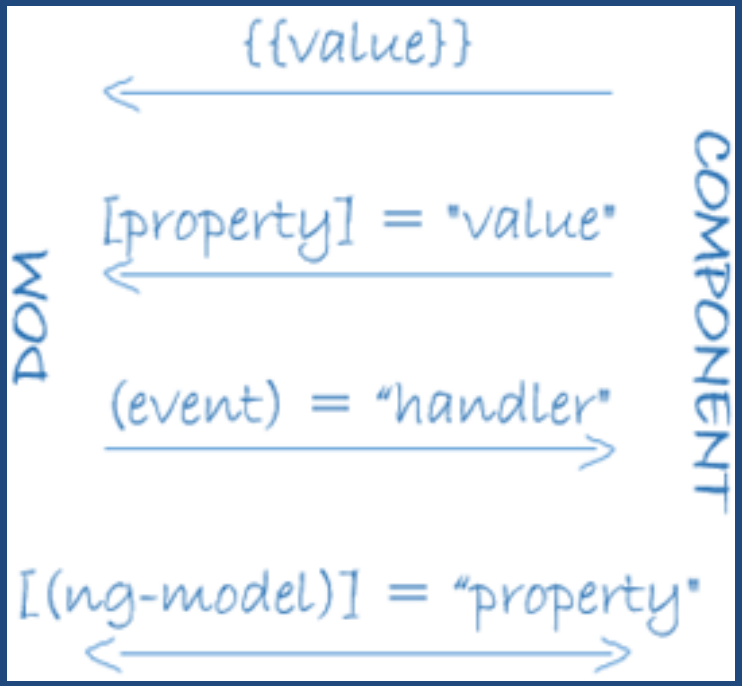
- *Components* are the most common of the three directives.
- Structural Directives change the structure of the view. (Ex:NgFor and NgIf.)
- Attribute directives are used as attributes of elements. (Ex:NgStyle - can change several element styles at the same time.

Structural Directives

- They shape or reshape the DOM's structure, typically by adding, removing, or manipulating elements.
- We apply a structural directive to a host element.
- An asterisk (*) precedes the directive attribute name

  <div *ngIf="hero" >{{hero.name}}

Data Binding



Interpolation ; Property Binding ; Event Binding ; NgModel

Data Binding

- Angular 2 data binding

Angular 2 data binding

C/D	Attribute	Binding type
—>	{{ value }}	one-way
—>	[property] = “value”	property
<—	(event) = “handler”	event
<—>	[(ng-model)] = “property”	two-way

Component Interactions

- Pass data from parent to child with input binding
- Parent listens for child event
- Parent interacts with child via local variable
- Parent calls an @ViewChild()
- Parent and children communicate via a service

Forms

- Angular offers two form-building technologies:
  - reactive forms
  - template-driven forms
- The two technologies belong to the @angular/forms library and share a common set of form control classes.
- they diverge markedly in philosophy, programming style, and technique.
- They even have their own modules:
  ReactiveFormsModule and FormsModule.

Template Driven Forms

- Template-driven forms, introduced take a completely different approach.

- We place HTML form controls (such as <input> and <select>) in ***component template*** and bind them to data model properties in the component, using directives like ***ngModel.***

- We don't create Angular form control objects. Angular directives create them for us, using the information in our data bindings.

Template Driven Forms

- We don't push and pull data values.
- Angular handles that for us with ngModel.
- Angular updates the mutable data model with user changes as they happen.
- For this reason, the ngModel directive is not part of the ReactiveFormsModule.
- While this means less code in the component class***, template-driven forms are asynchronous*** which may complicate development in more advanced scenarios.

Reactive Forms

- Angular reactive forms favour explicit data management
  flowing between a non-UI data model (typically retrieved from a server)
  &
  a UI-oriented form model that retains the states and values of the HTML controls on screen.
- Reactive forms offer the ease of using reactive patterns, testing, and validation.

- With reactive forms, you create ***a tree of Angular form control objects in the component class*** and bind them to native form control elements in the component template

Reactive Forms

- We create and manipulate form control objects directly in the component class.
- As the component class has immediate access to both ***the data model and the form control structure*** :
  - we can push data model values into form controls
  - pull user-changed values back out.
  - The component can observe changes in form control state and react to those changes.

Reactive Forms

- advantage of working with form control objects directly :
  - value and validity updates are always synchronous and under our control.
  - We won't encounter the timing issues that sometimes plague a template-driven form
  - reactive forms can be easier to unit test.

Reactive Forms

- In keeping with the reactive paradigm:
  - the component preserves the immutability of the data model, treating it as a pure source of original values.
  - Rather than update the data model directly, the component extracts user changes and forwards them to an external component or service, (for such as saving them) and returns a new data model to the component that reflects the updated model state.

Form Validation

Template-driven validation

- To add validation to a template-driven form, you add the same validation attributes as with native HTML form validation.
- Angular uses directives to match these attributes with validator functions in the framework.
- Every time the value of a form control changes, Angular runs validation and generates either a list of validation errors, which results in an INVALID status, or null, which results in a VALID status.
- You can then inspect the control's state by exporting ngModel to a local template variable.

Form Validation

- The following example exports NgModel into a variable called name:
- <input id="name" name="name" class="form-control" required minlength="4" forbiddenName="bob" [(ngModel)]="hero.name" #name="ngModel" >
- <div *ngIf="name.invalid && (name.dirty || name.touched>
- <div *ngIf="name.errors.required"> Name is required.
- <div *ngIf="name.errors.minlength"> Name must be at least 4 characters long.
- <div *ngIf="name.errors.forbiddenName"> Name cannot be Bob.

Form Validation

- #name="ngModel" exports NgModel into a local variable called name.
- NgModel mirrors many of the properties of its underlying FormControl instance, so you can use this in the template to check for control states such as valid and dirty

Why check dirty and touched?

- You may not want your application to display errors before the user has a chance to edit the form.
- The checks for dirty and touched prevent errors from showing until the user does one of two things:
  - changes the value, turning the control dirty;
  - or blurs the form control element, setting the control to touched.

Reactive form validation

- In a reactive form, the source of truth is the component class. Instead of adding validators through attributes in the template, you add validator functions directly to the form control model in the component class. Angular then calls these functions whenever the value of the control changes.

Built-in validators

- You can choose to write your own validator functions, or you can use some of Angular's built-in validators.
- The same built-in validators that are available as attributes in template-driven forms, such as required and minlength, are all available to use as functions from the Validators class.
- To update the form to be a reactive form, you can use some of the same built-in validators—this time, in function form.

Services

- As the application grows, it will have multiple components.
- These components may require to work on some common data set.
- Instead of copying and pasting the same code over and over, we'll ***create a single reusable data service*** and inject it into the components that need it.

Dependancy Injection

Why @Injectable()?

@Injectable() marks a class as available to an injector for instantiation.

Generally speaking, an injector will report an error when trying to instantiate a class that is not marked as @Injectable().

:recommended: adding @Injectable() to every service class, even though that don't have dependencies and, therefore, do not technically require it. Here's why:

Future proofing: No need to remember @Injectable() when we add a dependency later.

Consistency: All services follow the same rules, and we don't have to wonder why a decorator is missing.

Configuring Injector

We don't have to create an Angular injector.
Angular creates an application-wide injector for us during the bootstrap process.

Registering providers in an NgModule

```
@NgModule({
  imports: [
    BrowserModule
                ],
  declarations: [
    AppComponent,
    CarComponent,
    HeroesComponent,
    /* . . . */
  ],
  providers: [
    UserService,
    { provide: APP_CONFIG, useValue: HERO_DI_CONFIG }
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

Configuring Injector

- import { Component } from '@angular/core';
- import { HeroService } from './hero.service';
- @Component({
- selector: 'my-heroes',
- providers: [HeroService],
- template: `
- <h2>Heroes</h2>
- <hero-list></hero-list>
- `
- })
- export class HeroesComponent { }

Configuring Injector

- When to use the NgModule and when an application component?
- A provider in an NgModule is registered in the root injector. That means that every provider registered within an NgModule will be accessible in the entire application.

- Aa provider registered in an application component is available only on that component and all its children.

- We want the APP_CONFIG service to be available all across the application, but a HeroService is only used within the Heroes feature area and nowhere else.

Pipes

app/exponential-strength.pipe.ts

```
import { Pipe, PipeTransform } from '@angular/core';
/*
 * Raise the value exponentially
 * Takes an exponent argument that defaults to 1.
 * Usage:
 *   value | exponentialStrength:exponent
 * Example:
 *   {{ 2 |  exponentialStrength:10}}
 *   formats to: 1024
*/
@Pipe({name: 'exponentialStrength'})
export class ExponentialStrengthPipe implements PipeTransform {
  transform(value: number, exponent: string): number {
    let exp = parseFloat(exponent);
    return Math.pow(value, isNaN(exp) ? 1 : exp);
  }
}
```