Angular 2

Demo youtube link -https://www.youtube.com/watch?v=-zW1zHqsdyc

Architecture

Angular is a framework for building client applications in HTML and either JavaScript or a language like TypeScript that compiles to JavaScript.

The framework consists of several libraries, some of them core and some optional.

You write Angular applications by

* + 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 you launch the app by bootstrapping the root module.

Angular takes over, presenting your application content in a browser and responding to user interactions according to the instructions you've provided.

Basic building blocks

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

Quickstart

Angular applications are made up of components.

A component is the combination of an HTML template and a component class that controls a portion of the screen.

Eg.

import { Component } from '@angular/core';

@Component({

selector: 'my-app',

template: `<h1>Hello {{name}}</h1>`

})

export class AppComponent {

name = 'Angular';

}

---To diasble proxy

npm config set strict-ssl false

---To enable proxy

npm config set strict-ssl true

Main Files

* main.ts

Compiles the application with the JIT compiler and bootstraps the application's main module (AppModule) to run in the browser.

The JIT compiler is a reasonable choice during the development of most projects and it's the only viable choice for a sample running in a live-coding environment like Plunker. You'll learn about alternative compiling and deployment options later in the documentation.

Eg.

import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';

platformBrowserDynamic().bootstrapModule(AppModule);

* app/app.module.ts

Defines AppModule, the root module that tells Angular how to assemble the application.

Right now it declares only the AppComponent. Soon there will be more components to declare.

Eg.

import { NgModule } from '@angular/core';

import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';

@NgModule({

imports: [ BrowserModule ],

declarations: [ AppComponent ],

bootstrap: [ AppComponent ]

})

export class AppModule { }

* app/app.component.ts

Defines the same AppComponent as the one in the QuickStart playground.

It is the root component of what will become a tree of nested components as the application evolves.

Note.

* 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

Angular apps are modular and Angular has its own modularity system called Angular modules or NgModules.

Every Angular app has at least one Angular module class, the 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, each a cohesive block of code dedicated to an application domain, a workflow, or a closely related set of capabilities.

An Angular module, whether a root or feature, is a class with an @NgModule decorator.

NgModule is a decorator function that takes a single metadata object whose properties describe the module. The most important properties are:

* declarations

The view classes that belong to this module.

Angular has three kinds of view classes: components, directives, and pipes.

* exports

The subset of declarations that should be visible and usable in the component templates of other modules.

* imports

Other modules whose exported classes are needed by component templates declared in this module.

* providers

Creators of services that this module contributes to the global collection of services;

They become accessible in all parts of the app.

* bootstrap

The main application view, called the root component,that hosts all other app views.

Only the root module should set this bootstrap property.

Eg. (src/app/app.module.ts)

import { NgModule } from '@angular/core';

import { BrowserModule } from '@angular/platform-browser';

@NgModule({

imports: [ BrowserModule ],

providers: [ Logger ],

declarations: [ AppComponent ],

exports: [ AppComponent ],

bootstrap: [ AppComponent ]

})

export class AppModule { }

Angular modules vs. JavaScript modules

* The Angular module — a class decorated with @NgModule — is a fundamental feature of Angular.
* JavaScript also has its own module system for managing collections of JavaScript objects.
* It's completely different and unrelated to the Angular module system.
* In JavaScript each file is a module and all objects defined in the file belong to that module.
* The module declares some objects to be public by marking them with the export key word.
* Other JavaScript modules use import statements to access public objects from other modules.

Eg.

import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';

Eg.

export class AppModule { }

Angular libraries

Angular ships as a collection of JavaScript modules. You can think of them as library modules.

Each Angular library name begins with the @angular prefix.

You install them with the npm package manager and import parts of them with JavaScript import statements.

Eg.

import Angular's Component decorator from the @angular/core library like this:

import { Component } from '@angular/core';

You also import Angular modules from Angular libraries using JavaScript import statements:

import { BrowserModule } from '@angular/platform-browser';

In the example of the simple root module above, the application module needs material from within that BrowserModule. To access that material, add it to the @NgModule metadata imports like this.

imports: [ BrowserModule ],

APPMODULE: THE ROOT MODULE

Tell Angular how to construct and bootstrap the app in the root "AppModule".

Every application has at least one Angular module, the root module that you bootstrap to launch the application.

Eg.

import { NgModule } from '@angular/core';

import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';

@NgModule({

imports: [ BrowserModule ],

declarations: [ AppComponent ],

bootstrap: [ AppComponent ]

})

export class AppModule { }

The @NgModule decorator identifies AppModule as an Angular module class (also called an NgModule class).

@NgModule takes a metadata object that tells Angular how to compile and launch the application.

* imports — the BrowserModule that this and every application needs to run in a browser.
* declarations — the application's lone component, which is also ...
* bootstrap — the root component that Angular creates and inserts into the index.html host web page.

* Components

A component controls a patch of screen called a view.

Views are controlled by components.

A component is the combination of an HTML template and a component class that controls a portion of the screen.

Angular creates, updates, and destroys components as the user moves through the application.

Your app can take action at each moment in this lifecycle through optional lifecycle hooks, like ngOnInit() declared above.

Eg.

@Component({

selector: 'hero-list',

templateUrl: './hero-list.component.html',

providers: [ HeroService ]

})

export class HeroListComponent implements OnInit {

heroes: Hero[];

selectedHero: Hero;

constructor(private service: HeroService) { }

ngOnInit() {

this.heroes = this.service.getHeroes();

}

selectHero(hero: Hero) { this.selectedHero = hero; }

}

* Templates

You define a component's view with its companion template.

A template is a form of HTML that tells Angular how to render the component.

Eg.

<h2>Hero List</h2>

<p \*ngIf="heroes.length > 3">There are many heroes!</p>

<p><i>Pick a hero from the list</i></p>

<ul>

<li \*ngFor="let hero of heroes; let i=index" (click)="selectHero(hero)">

{{i}} {{hero.name}}

</li>

</ul>

<hero-detail \*ngIf="selectedHero" [hero]="selectedHero"></hero-detail>

* Metadata

Metadata tells Angular how to process a class.

In fact, HeroListComponent really is just a class. It's not a component until you tell Angular about it.

To tell Angular that HeroListComponent is a component, attach metadata to the class.

In TypeScript, you attach metadata by using a decorator. Here's some metadata for HeroListComponent:

Eg.

@Component({

selector: 'hero-list',

templateUrl: './hero-list.component.html',

providers: [ HeroService ]

})

export class HeroListComponent implements OnInit {

/\* . . . \*/

}

Here is the @Component decorator, which identifies the class immediately below it as a component class.

The @Component decorator takes a required configuration object with the information Angular needs to create and present the component and its view

Here are a few of the most useful @Component configuration options:

* selector:

CSS selector that tells Angular to create and insert an instance of this component where it finds a <hero-list> tag in parent HTML.

For example, if an app's HTML contains <hero-list></hero-list>, then Angular inserts an instance of the HeroListComponent view between those tags.

* templateUrl:

module-relative address of this component's HTML template, shown above.

* providers:

array of dependency injection providers for services that the component requires.

This is one way to tell Angular that the component's constructor requires a HeroService so it can get the list of heroes to display.

The metadata in the @Component tells Angular where to get the major building blocks you specify for the component.

The template, metadata, and component together describe a view.

@Injectable, @Input, and @Output are a few of the more popular decorators.

The architectural takeaway is that you must add metadata to your code so that Angular knows what to do.

* Data binding

Without a framework, you would be responsible for pushing data values into the HTML controls and turning user responses into actions and value updates. Writing such push/pull logic by hand is tedious, error-prone, and a nightmare to read as any experienced jQuery programmer can attest.

Angular supports data binding, a mechanism for coordinating parts of a template with parts of a component.

Add binding markup to the template HTML to tell Angular how to connect both sides.

There are four forms of data binding syntax.

Each form has a direction — to the DOM, from the DOM, or in both directions.

Eg.

<li>{{hero.name}}</li>

<hero-detail [hero]="selectedHero"></hero-detail>

<li (click)="selectHero(hero)"></li>

<input [(ngModel)]="hero.name">

* Interpolation/one way binding

The {{hero.name}} interpolation displays the component's hero.name property value within the <li> element.

* Property binding

The [hero] property binding passes the value of selectedHero from the parent HeroListComponent to the hero property of the child HeroDetailComponent.

* Event binding

The (click) event binding calls the component's selectHero method when the user clicks a hero's name.

* Two-way data binding

Important fourth form that combines property and event binding in a single notation, using the ngModel directive.

In two-way binding, a data property value flows to the input box from the component as with property binding.

The user's changes also flow back to the component, resetting the property to the latest value, as with event binding.

Angular processes all data bindings once per JavaScript event cycle, from the root of the application component tree through all child components.

Data binding plays an important role in communication between a template and its component.

Data binding is also important for communication between parent and child components.

* Directives

Angular templates are dynamic. When Angular renders them, it transforms the DOM according to the instructions given by directives.

A directive is a class with a @Directive decorator.

A component is a directive-with-a-template;

@Component decorator is actually a @Directive decorator extended with template-oriented features.

While a component is technically a directive, components are so distinctive and central to Angular applications that this architectural overview separates components from directives.

Two other kinds of directives exist: structural and attribute directives.

They tend to appear within an element tag as attributes do, sometimes by name but more often as the target of an assignment or a binding.

* Structural directives alter layout by adding, removing, and replacing elements in DOM.

Eg.

<li \*ngFor="let hero of heroes; let i=index"></li>

<hero-detail \*ngIf="selectedHero"></hero-detail>

* Attribute directives alter the appearance or behavior of an existing element. In templates they look like regular HTML attributes, hence the name.

Eg.

<input [(ngModel)]="hero.name">

The ngModel directive, which implements two-way data binding, is an example of an attribute directive.

ngModel modifies the behavior of an existing element (typically an <input>) by setting its display value property and responding to change events.

* Angular has a few more directives that either alter the layout structure (for example, ngSwitch) or modify aspects of DOM elements and components (for example, ngStyle and ngClass).

* Of course, you can also write your own directives. Components such as HeroListComponent are one kind of custom directive.
* Services

Services mainly used for data access...if we want to fecth data from database or backend..put that in a service...so that we can access from all components throughout the application...and also this makes the component lean...

Services are asynchronous..and can return data as a promise or an observable using RxJS(Reacted extensions)

Services Needs @Injectable decorator to make it as a service.

To use in a service in component, import && also add it to providers array in the component decorator...

and also inject into the constructor of the compoment

Service is a broad category encompassing any value, function, or feature that your application needs.

Almost anything can be a service. A service is typically a class with a narrow, well-defined purpose.

It should do something specific and do it well.

Eg.

logging service

data service

message bus

tax calculator

application configuration

There is nothing specifically Angular about services.

Angular has no definition of a service.

There is no service base class, and no place to register a service.

Yet services are fundamental to any Angular application. Components are big consumers of services.

Note.

* Component classes should be lean. They don't fetch data from the server, validate user input, or log directly to the console.
* They delegate such tasks to services.
* A component's job is to enable the user experience and nothing more.
* It mediates between the view (rendered by the template) and the application logic (which often includes some notion of a model).
* A good component presents properties and methods for data binding. It delegates everything nontrivial to services.

Angular does help you follow these principles by making it easy to factor your application logic into services and make those services available to components through dependency injection.

Eg.

import { Injectable } from '@angular/core';

@Injectable()

export class Logger {

log(msg: any) { console.log(msg); }

error(msg: any) { console.error(msg); }

warn(msg: any) { console.warn(msg); }

}

Eg.

import { Injectable } from '@angular/core';

@Injectable()

export class HeroService {

private heroes: Hero[] = [];

constructor(

private backend: BackendService,

private logger: Logger) { }

getHeroes() {

this.backend.getAll(Hero).then( (heroes: Hero[]) => {

this.logger.log(`Fetched ${heroes.length} heroes.`);

this.heroes.push(...heroes); // fill cache

});

return this.heroes;

}

}

* Dependency injection

Dependency injection is a way to supply a new instance of a class with the fully-formed dependencies it requires.

Most dependencies are services. Angular uses dependency injection to provide new components with the services they need.

Eg.

constructor(private service: HeroService) { }

When Angular creates a component, it first asks an injector for the services that the component requires.

An injector maintains a container of service instances that it has previously created.

If a requested service instance is not in the container, the injector makes one and adds it to the container before returning the service to Angular.

When all requested services have been resolved and returned, Angular can call the component's constructor with those services as arguments. This is dependency injection.

If the injector doesn't have a HeroService, how does it know how to make one?

In brief, you must have previously registered a provider of the HeroService with the injector. A provider is something that can create or return a service, typically the service class itself.

You can register providers in modules or in components.

In general, add providers to the root module so that the same instance of a service is available everywhere.

Registering at a component level means you get a new instance of the service with each new instance of that component.

Points to remember about dependency injection:

* Dependency injection is wired into the Angular framework and used everywhere.
* The injector is the main mechanism.
* An injector maintains a container of service instances that it created.
* An injector can create a new service instance from a provider.
* A provider is a recipe for creating a service.
* Register providers with injectors.

* Others
* Animations:

Animate component behavior without deep knowledge of animation techniques or CSS with Angular's animation library.

* Change detection:

The change detection documentation will cover how Angular decides that a component property value has changed, when to update the screen, and how it uses zones to intercept asynchronous activity and run its change detection strategies.

* Events:

The events documentation will cover how to use components and services to raise events with mechanisms for publishing and subscribing to events.

* Forms:

Support complex data entry scenarios with HTML-based validation and dirty checking.

* HTTP:

Communicate with a server to get data, save data, and invoke server-side actions with an HTTP client.

* Lifecycle hooks:

Tap into key moments in the lifetime of a component, from its creation to its destruction, by implementing the lifecycle hook interfaces.

* Pipes:

Use pipes in your templates to improve the user experience by transforming values for display. Consider this currency pipe expression:

Eg.

price | currency:'USD':true

It displays a price of 42.33 as $42.33.

* Router:

Navigate from page to page within the client application and never leave the browser.

* Testing:

Run unit tests on your application parts as they interact with the Angular framework using the Angular Testing Platform.

User Input

User input triggers DOM events.

We listen to those events with event bindings that funnel updated values back into our components and models.

*Binding to user input events*

Eg.

<button (click)="onClickMe()">Click me!</button>

*Get user input from the $event object*

Eg.

template: `

<input (keyup)="onKey($event)">

<p>{{values}}</p>

`

export class KeyUpComponent\_v1 {

values = '';

onKey(event: any) { // without type info

this.values += event.target.value + ' | ';

}

}

*Type the $event*

Eg.

export class KeyUpComponent\_v1 {

values = '';

onKey(event: KeyboardEvent) { // with type info

this.values += (<HTMLInputElement>event.target).value + ' | ';

}

}

The $event is now a specific KeyboardEvent. Not all elements have a value property so it casts target to an input element.

The OnKey method more clearly expresses what it expects from the template and how it interprets the event.

*Passing $event is a dubious practice*

Typing the event object reveals a significant objection to passing the entire DOM event into the method:

The component has too much awareness of the template details.

It can't extract information without knowing more than it should about the HTML implementation.

That breaks the separation of concerns between the template (what the user sees) and the component (how the application processes user data).

The next section shows how to use template reference variables to address this problem.

*Get user input from a template reference variable*

Eg.

@Component({

selector: 'loop-back',

template: `

<input #box (keyup)="0">

<p>{{box.value}}</p> `

})

export class LoopbackComponent { }

Eg.

@Component({

selector: 'key-up2',

template: `

<input #box (keyup)="onKey(box.value)">

<p>{{values}}</p>`

})

export class KeyUpComponent\_v2 {

values = '';

onKey(value: string) {

this.values += value + ' | ';

}

*Key event filtering (with key.enter)*

The (keyup) event handler hears every keystroke.

Sometimes only the Enter key matters, because it signals that the user has finished typing.

One way to reduce the noise would be to examine every $event.keyCode and take action only when the key is Enter.

Eg.

@Component({

selector: 'key-up3',

template: `

<input #box (keyup.enter)="onEnter(box.value)">

<p>{{value}}</p>

`

})

export class KeyUpComponent\_v3 {

value = '';

onEnter(value: string) { this.value = value; }

}

*On blur*

In the previous example, the current state of the input box is lost if the user mouses away and clicks elsewhere on the page without first pressing Enter.

The component's value property is updated only when the user presses Enter.

To fix this issue, listen to both the Enter key and the blur event.

Eg.

@Component({

selector: 'key-up4',

template: `

<input #box

(keyup.enter)="update(box.value)"

(blur)="update(box.value)">

<p>{{value}}</p>

`

})

export class KeyUpComponent\_v4 {

value = '';

update(value: string) { this.value = value; }

}

FORMS

A form creates a cohesive, effective, and compelling data entry experience.

An Angular form coordinates a set of data-bound user controls, tracks changes, validates input, and presents errors.

Need to import FormsModule to AppModule

Eg.

<form (ngSubmit)="onSubmit()" #heroForm="ngForm">

<div class="form-group">

<label for="name">Name</label>

<input type="text" class="form-control" id="name" required [(ngModel)]="model.name" name="name" #name="ngModel">

<div [hidden]="name.valid || name.pristine" class="alert alert-danger">Name is required</div>

</div>

<div class="form-group">

<label for="alterEgo">Alter Ego</label>

<input type="text" class="form-control" id="alterEgo" [(ngModel)]="model.alterEgo name="alterEgo"> </div>

<button type="submit" class="btn btn-success" [disabled]="!heroForm.form.valid">Submit</button>

<button type="button" class="btn btn-default" (click)="newHero(); heroForm.reset()"> New Hero</button>

</form>

.ng-valid[required], .ng-valid.required {

border-left: 5px solid #42A948; /\* green \*/

}

.ng-invalid:not(form) {

border-left: 5px solid #a94442; /\* red \*/

}

Keywords..

* [(ngModel)] - Two way binding... Needs to import FormsModule
* (ngSubmit) - for form submit
* heroForm.reset() - To reset form (pristine will work..pristine means untouched)
* #heroForm="ngForm", #name="ngModel" - Template-reference variables
* #heroForm="ngForm" - To check the form validity and to reset form...
* name.valid - To check element's validity
* #name="ngModel" - To know the field validity
* name.pristine - To check whether it is untouced..
* heroForm.form.valid - To know entire form is valid or not..
* [hidden] - hide & show based on the value given to it
* [disabled] - disable or enable based on the value given to it.
* required - making field as mandatory(HTML property)
* The NgModel directive doesn't just track state; It updates the control with special Angular CSS classes that reflect the state.

You can leverage those class names to change the appearance of the control.

*State Class if true Class if false*

The control has been visited. ng-touched ng-untouched

The control's value has changed. ng-dirty ng-pristine

The control's value is valid. ng-valid ng-invalid

Conclusion

* + - An Angular HTML form template.
    - A form component class with a @Component decorator.
    - Handling form submission by binding to the NgForm.ngSubmit event property.
    - Template-reference variables such as #heroForm and #name.
    - [(ngModel)] syntax for two-way data binding.
    - The use of name attributes for validation and form-element change tracking.
    - The reference variable’s valid property on input controls to check if a control is valid and show/hide error messages.
    - Controlling the Submit button's enabled state by binding to NgForm validity.
    - Custom CSS classes that provide visual feedback to users about invalid controls.

HTTP

The Angular HTTP library simplifies application programming with the XHR and JSONP APIs.

*Providing HTTP services*

Need to import HttpModule, JsonpModule to AppModule

Eg.

import { HttpModule, JsonpModule } from '@angular/http';

@NgModule({

imports: [

HttpModule,

JsonpModule

],

})

The HttpModule is necessary for making HTTP calls.

Though the JsonpModule isn't necessary for plain HTTP,

*Component*

Eg.

getHeroes() {

this.heroService.getHeroes()

.subscribe(

heroes => this.heroes = heroes,

error => this.errorMessage = <any>error);

}

addHero(name: string) {

if (!name) { return; }

this.heroService.create(name)

.subscribe(

hero => this.heroes.push(hero),

error => this.errorMessage = <any>error);

}

Angular injects a HeroService into the constructor and the component calls that service to fetch and save data.

The component does not talk directly to the Angular Http client.

The component doesn't know or care how it gets the data. It delegates to the HeroService.

* *This is a golden rule: always delegate data access to a supporting service class.*

Although at runtime the component requests heroes immediately after creation, you don't call the service's get method in the component's constructor.

Instead, call it inside the ngOnInit lifecycle hook and rely on Angular to call ngOnInit when it instantiates this component.

The service's getHeroes() and create() methods return an Observable of hero data that the Angular Http client fetched from the server.

Think of an Observable as a stream of events published by some source.

To listen for events in this stream, subscribe to the Observable.

These subscriptions specify the actions to take when the web request produces a success event (with the hero data in the event payload) or a fail event (with the error in the payload).

* *Subscribe*

The subscribe() method returns a Subscription.

A Subscription is not another Observable. It's the end of the line for Observables.

You can't call map() on it or call subscribe() again.

The Subscription object has a different purpose, signified by its primary method, unsubscribe.

Fetch data with http.get()

Earlier ones..

Eg.

import { Injectable } from '@angular/core';

import { Hero } from './hero';

import { HEROES } from './mock-heroes';

@Injectable()

export class HeroService {

getHeroes(): Promise<Hero[]> {

return Promise.resolve(HEROES);

}

}

You can revise that HeroService to get the heroes from the server using the Angular Http client service:

Eg.

import { Injectable } from '@angular/core';

import { Http, Response } from '@angular/http';

import { Observable } from 'rxjs/Observable';

import 'rxjs/add/operator/catch';

import 'rxjs/add/operator/map';

import { Hero } from './hero';

@Injectable()

export class HeroService {

private heroesUrl = 'api/heroes'; // URL to web API

constructor (private http: Http) {}

getHeroes(): Observable<Hero[]> {

return this.http.get(this.heroesUrl)

.map(this.extractData)

.catch(this.handleError);

}

private extractData(res: Response) {

let body = res.json();

return body.data || { };

}

private handleError (error: Response | any) {

// In a real world app, you might use a remote logging infrastructure

let errMsg: string;

if (error instanceof Response) {

const body = error.json() || '';

const err = body.error || JSON.stringify(body);

errMsg = `${error.status} - ${error.statusText || ''} ${err}`;

} else {

errMsg = error.message ? error.message : error.toString();

}

console.error(errMsg);

return Observable.throw(errMsg);

}

}

If you are familiar with asynchronous methods in modern JavaScript, you might expect the get method to return a promise.

You'd expect to chain a call to then() and extract the heroes.

Instead you're calling a map() method. Clearly this is not a promise.

In fact, the http.get method returns an Observable of HTTP Responses (Observable<Response>) from the RxJS library

and map() is one of the RxJS operators.

* *RxJS library*

RxJS is a third party library, endorsed by Angular, that implements the asynchronous Observable pattern.

All of the Developer Guide samples have installed the RxJS npm package because Observables are used widely in Angular applications.

This app needs it when working with the HTTP client.

But you must take a critical extra step to make RxJS Observables usable: you must import the RxJS operators individually.

*Enable RxJS operators*

import { Observable } from 'rxjs/Observable';

import 'rxjs/add/operator/catch';

import 'rxjs/add/operator/map';

*Process the response object*

private extractData(res: Response) {

let body = res.json();

return body.data || { };

}

The response object doesn't hold the data in a form the app can use directly.

You must parse the response data into a JSON object.

*Parse to JSON*

The response data are in JSON string form.

The app must parse that string into JavaScript objects by calling response.json().

Don't expect the decoded JSON to be the heroes array directly.

This server always wraps JSON results in an object with a data property. You have to unwrap it to get the heroes

Do not return the response object

The point of a data service is to hide the server interaction details from consumers

*Always handle errors*

The catch() operator passes the error object from http to the handleError() method.

The handleError method transforms the error into a developer-friendly message, logs it to the console, and returns the message in a new, failed Observable via Observable.throw.

*Send data to the server*

create(name: string): Observable<Hero> {

let headers = new Headers({ 'Content-Type': 'application/json' });

let options = new RequestOptions({ headers: headers });

return this.http.post(this.heroesUrl, { name }, options)

.map(this.extractData)

.catch(this.handleError);

}

In the headers object, the Content-Type specifies that the body represents JSON.

*Cross-Origin Requests*

You just learned how to make XMLHttpRequests using the Angular Http service. This is the most common approach to server communication, but it doesn't work in all scenarios.

For security reasons, web browsers block XHR calls to a remote server whose origin is different from the origin of the web page.

The origin is the combination of URI scheme, hostname, and port number. This is called the same-origin policy.

Modern browsers do allow XHR requests to servers from a different origin if the server supports the CORS protocol.

If the server requires user credentials, enable them in the request headers.

Some servers do not support CORS but do support an older, read-only alternative called JSONP. Wikipedia is one such server.

Eg.

import { Injectable } from '@angular/core';

import { Jsonp, URLSearchParams } from '@angular/http';

import 'rxjs/add/operator/map';

@Injectable()

export class WikipediaService {

constructor(private jsonp: Jsonp) {}

search (term: string) {

let wikiUrl = 'http://en.wikipedia.org/w/api.php';

// The Wikipedia "opensearch" API expects four parameters (key/value pairs) to arrive in the request URL's query string.

// The keys are search, action, format, and callback.

let params = new URLSearchParams();

params.set('search', term); // the user's search value

params.set('action', 'opensearch');

params.set('format', 'json');

params.set('callback', 'JSONP\_CALLBACK');

// TODO: Add error handling

return this.jsonp

.get(wikiUrl, { search: params })

.map(response => <string[]> response.json()[1]);

}

}

* Routing

*<base href>*

Most routing applications should add a <base> element to the index.html as the first child in the <head> tag to tell the router how to compose navigation URLs.

If the app folder is the application root, as it is for the sample application, set the href value exactly as shown here.

*src/index.html*

<base href="/">

*Router imports*

import { ModuleWithProviders } from '@angular/core';

import { RouterModule, Routes } from '@angular/router';

*Configuration*

const appRoutes: Routes = [

{

path: 'crisis-center',

component: CrisisListComponent

},

{

path: 'hero/:id',

component: HeroDetailComponent

},

{

path: 'heroes',

component: HeroListComponent,

data: { title: 'Heroes List' }

},

{

path: '',

redirectTo: '/heroes',

pathMatch: 'full'

},

{

path: '\*\*',

component: PageNotFoundComponent

}

];

imports: [

RouterModule.forRoot(appRoutes)

],

//RouterModule.forChild(appRoutes)

Only call RouterModule.forRoot in the root AppRoutingModule (or the AppModule if that's where you register top level application routes).

In any other module, you must call the RouterModule.forChild method to register additional routes.

The :id in the first route is a token for a route parameter. In a URL such as /hero/42, "42" is the value of the id parameter.

The corresponding HeroDetailComponent will use that value to find and present the hero whose id is 42.

The data property in the third route is a place to store arbitrary data associated with this specific route.

The data property is accessible within each activated route.

Use it to store items such as page titles, breadcrumb text, and other read-only, static data.

You'll use the resolve guard to retrieve dynamic data later in the guide.

The empty path in the fourth route represents the default path for the application, the place to go when the path in the URL is empty, as it typically is at the start.

This default route redirects to the route for the /heroes URL and, therefore, will display the HeroesListComponent.

The \*\* path in the last route is a wildcard.

The router will select this route if the requested URL doesn't match any paths for routes defined earlier in the configuration.

This is useful for displaying a "404 - Not Found" page or redirecting to another route.

*Redirecting routes*

A redirect route requires a pathMatch property to tell the router how to match a URL to the path of a route. The router throws an error if you don't.

The other possible pathMatch value is 'prefix' which tells the router to match the redirect route when the remaining URL begins with the redirect route's prefix path.

The order of the routes in the configuration matters and this is by design.

The router uses a first-match wins strategy when matching routes, so more specific routes should be placed above less specific routes.

In the configuration above, routes with a static path are listed first, followed by an empty path route, that matches the default route. The wildcard route comes last because it matches every URL and should be selected only if no other routes are matched first.

*Router outlet*

*In the main component*

<router-outlet></router-outlet>

<!-- Routed views go here -->

*Router links*

<nav>

<a routerLink="/crisis-center" routerLinkActive="active">Crisis Center</a>

<a routerLink="/heroes" routerLinkActive="active">Heroes</a>

</nav>

<router-outlet></router-outlet>

The RouterLink directives on the anchor tags give the router control over those elements.

The RouterLinkActive directive on each anchor tag helps visually distinguish the anchor for the currently selected "active" route.

The router adds the active CSS class to the element when the associated RouterLink becomes active.

You can add this directive to the anchor or to its parent element.

This cascades down through each level of the route tree, so parent and child router links can be active at the same time. To override this behavior, you can bind to the [routerLinkActiveOptions] input binding with the { exact: true } expression. By using { exact: true }, a given RouterLink will only be active if its URL is an exact match to the current URL.

Navigate to hero detail imperatively

<li \*ngFor="let hero of heroes | async" (click)="onSelect(hero)">

<span class="badge">{{ hero.id }}</span> {{ hero.name }}

</li>

onSelect(hero: Hero) {

this.router.navigate(['/hero', hero.id]);

}

*Router state*

After the end of each successful navigation lifecycle, the router builds a tree of ActivatedRoute objects that make up the current state of the router.

You can access the current RouterState from anywhere in the application using the Router service and the routerState property.

*ActivatedRoute*

Each ActivatedRoute in the RouterState provides methods to traverse up and down the route tree to get information from parent, child and sibling routes.

*url:* An Observable of the route path(s), represented as an array of strings for each part of the route path.

*data:* An Observable that contains the data object provided for the route.

Also contains any resolved values from the resolve guard.

*params*: An Observable that contains the required and optional parameters specific to the route.

*queryParams*: An Observable that contains the query parameters available to all routes.

*fragment*: An Observable of the URL fragment available to all routes.

*outlet*: The name of the RouterOutlet used to render the route.

For an unnamed outlet, the outlet name is primary.

*routeConfig*: The route configuration used for the route that contains the origin path.

*parent*: an ActivatedRoute that contains the information from the parent route when using child routes.

*firstChild*: contains the first ActivatedRoute in the list of child routes.

*children*: contains all the child routes activated under the current route.

Import the Router, ActivatedRoute, and Params tokens from the router package.

Eg.

import { Router, ActivatedRoute, Params } from '@angular/router';

Import the switchMap operator because you need it later to process the Observable route parameters.

Eg.

import 'rxjs/add/operator/switchMap';

As usual, you write a constructor that asks Angular to inject services that the component requires and reference them as private variables.

Eg.

constructor(

private route: ActivatedRoute,

private router: Router,

private service: HeroService

) {}

Later, in the ngOnInit method, you use the ActivatedRoute service to retrieve the parameters for the route, pull the hero id from the parameters and retrieve the hero to display.

Eg.

ngOnInit() {

this.route.params

// (+) converts string 'id' to a number

.switchMap((params: Params) => this.service.getHero(+params['id']))

.subscribe((hero: Hero) => this.hero = hero);

}

Note.

* The switchMap operator allows you to perform an action with the current value of the Observable, and map it to a new Observable.
* As with many rxjs operators, switchMap handles an Observable as well as a Promise to retrieve the value they emit.
* The switchMap operator will also cancel any in-flight requests if the user re-navigates to the route while still retrieving a hero.
* Use the subscribe method to detect id changes and to (re)set the retrieved Hero.

*SNAPSHOT: THE NO-OBSERVABLE ALTERNATIVE*

The route.snapshot provides the initial value of the route parameters. You can access the parameters directly without subscribing or adding observable operators. It's much simpler to write and read:

Eg.

ngOnInit() {

// (+) converts string 'id' to a number

let id = +this.route.snapshot.params['id'];

this.service.getHero(id)

.then((hero: Hero) => this.hero = hero);

}

Note.

* you only get the initial value of the parameters with this technique.
* Stick with the observable params approach if there's even a chance that the router could re-use the component.

*Optional Parameters*

this.router.navigate(['/heroes', { id: heroId, foo: 'foo' }]);

Pattern didn’t changed, but added some optional parameters...

Another format for routes...

Eg.

const crisisCenterRoutes: Routes = [

{

path: 'crisis-center',

component: CrisisCenterComponent,

children: [

{

path: '',

component: CrisisListComponent,

children: [

{

path: ':id',

component: CrisisDetailComponent

},

{

path: '',

component: CrisisCenterHomeComponent

}

]

}

]

}

];

@NgModule({

imports: [

RouterModule.forChild(crisisCenterRoutes)

],

exports: [

RouterModule

]

})

export class CrisisCenterRoutingModule { }

Note.

* The router displays the components of these routes in the RouterOutlet of the CrisisCenterComponent, not in the RouterOutlet of the AppComponent shell.

The CrisisListComponent contains the crisis list and a RouterOutlet to display the Crisis Center Home and Crisis Detail route components.

The Crisis Detail route is a child of the Crisis List. Since the router reuses components by default, the Crisis Detail component will be re-used as you select different crises. In contrast, back in the Hero Detail route, the component was recreated each time you selected a different hero.

At the top level, paths that begin with / refer to the root of the application. But child routes extend the path of the parent route. With each step down the route tree, you add a slash followed by the route path, unless the path is empty.

*Relative navigation*

this.router.navigate([crisis.id], { relativeTo: this.route });

*Displaying multiple routes in named outlets*

<router-outlet></router-outlet>

<router-outlet name="popup"></router-outlet>

*SECONDARY ROUTES*

Named outlets are the targets of secondary routes.

*Routing*

Eg.

{

path: 'compose',

component: ComposeMessageComponent,

outlet: 'popup'

},

*RouterLink*

<a [routerLink]="[{ outlets: { popup: ['compose'] } }]">Contact</a>

*SECONDARY ROUTE NAVIGATION: MERGING ROUTES DURING NAVIGATION*

Navigate to the Crisis Center and click "Contact". you should see something like the following URL in the browser address bar.

Eg.

http://.../crisis-center(popup:compose)

The crisis-center is the primary navigation.

Parentheses surround the secondary route.

The secondary route consists of an outlet name (popup), a colon separator, and the secondary route path (compose).

CLEARING SECONDARY ROUTES

this.router.navigate([{ outlets: { popup: null }}]);

Milestone 5: Route guards

At the moment, any user can navigate anywhere in the application anytime. That's not always the right thing to do.

Perhaps the user is not authorized to navigate to the target component.

Maybe the user must login (authenticate) first.

Maybe you should fetch some data before you display the target component.

You might want to save pending changes before leaving a component.

You might ask the user if it's OK to discard pending changes rather than save them.

You can add guards to the route configuration to handle these scenarios.

A guard's return value controls the router's behavior:

If it returns true, the navigation process continues.

If it returns false, the navigation process stops and the user stays put.

The guard might return its boolean answer synchronously. But in many cases, the guard can't produce an answer synchronously.

The guard could ask the user a question, save changes to the server, or fetch fresh data. These are all asynchronous operations.

Accordingly, a routing guard can return an Observable<boolean> or a Promise<boolean> and the router will wait for the observable to resolve to true or false.

The router supports multiple kinds of guards:

CanActivate to mediate navigation to a route.

CanActivateChild() to mediate navigation to a child route.

CanDeactivate to mediate navigation away from the current route.

Resolve to perform route data retrieval before route activation.

CanLoad to mediate navigation to a feature module loaded asynchronously.

You can have multiple guards at every level of a routing hierarchy.

*CanActivate: requiring authentication*

Applications often restrict access to a feature area based on who the user is.

You could permit access only to authenticated users or to users with a specific role.

You might block or limit access until the user's account is activated.

The CanActivate guard is the tool to manage these navigation business rules.

Guards and the service providers they require must be provided at the module-level.

Eg. *Service*

import { Injectable } from '@angular/core';

import { CanActivate } from '@angular/router';

@Injectable()

export class AuthGuard implements CanActivate {

canActivate() {

console.log('AuthGuard#canActivate called');

return true;

}

}

*Router*

const adminRoutes: Routes = [{

path: 'admin',

component: AdminComponent,

canActivate: [AuthGuard],

children: [{

path: '',

children: [

{ path: 'crises', component: ManageCrisesComponent },

{ path: 'heroes', component: ManageHeroesComponent}, { path: '', component: AdminDashboardComponent }

],

}]

}];

*CanActivateChild: guarding child routes*

You can also protect child routes with the CanActivateChild guard. The CanActivateChild guard is similar to the CanActivate guard.

The key difference is that it runs before any child route is activated.

You protected the admin feature module from unauthorized access. You should also protect child routes within the feature module.

Eg. *Service*

import { Injectable } from '@angular/core';

import {

CanActivate, Router,

ActivatedRouteSnapshot,

RouterStateSnapshot,

CanActivateChild

} from '@angular/router';

import { AuthService } from './auth.service';

@Injectable()

export class AuthGuard implements CanActivate, CanActivateChild {

constructor(private authService: AuthService, private router: Router) {}

canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean{

let url: string = state.url;

return this.checkLogin(url);

}

canActivateChild(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean {

return this.canActivate(route, state);

}

/\* . . . \*/

}

*Router*

const adminRoutes: Routes = [{

path: 'admin',

component: AdminComponent,

canActivate: [AuthGuard],

children: [{

path: '',

canActivateChild: [AuthGuard],

children: [

{ path: 'crises', component: ManageCrisesComponent },

{ path: 'heroes', component: ManageHeroesComponent },

{ path: '', component: AdminDashboardComponent }

]

}]

}];

*CanDeactivate: handling unsaved changes*

In the real world, you might have to accumulate the users changes. You might have to validate across fields. You might have to validate on the server. You might have to hold changes in a pending state until the user confirms them as a group or cancels and reverts all changes.

Eg. *Service*

import { Injectable } from '@angular/core';

import { CanDeactivate } from '@angular/router';

import { Observable } from 'rxjs/Observable';

export interface CanComponentDeactivate {

canDeactivate: () => Observable<boolean> | Promise<boolean> | boolean;

}

@Injectable()

export class CanDeactivateGuard implements CanDeactivate<CanComponentDeactivate> {

canDeactivate(component: CanComponentDeactivate) {

return component.canDeactivate ? component.canDeactivate() : true;

}

}

*Router*

{

path: ':id',

component: CrisisDetailComponent,

canDeactivate: [CanDeactivateGuard]

},

*Resolve: pre-fetching component data*

It's preferable to pre-fetch data from the server so it's ready the moment the route is activated.

This also allows you to handle errors before routing to the component.

There's no point in navigating to a crisis detail for an id that doesn't have a record.

It'd be better to send the user back to the Crisis List that shows only valid crisis centers.

Eg. *Service*

import { Injectable } from '@angular/core';

import {

Router, Resolve, RouterStateSnapshot, ActivatedRouteSnapshot

} from '@angular/router';

import { Crisis, CrisisService } from './crisis.service';

@Injectable()

export class CrisisDetailResolver implements Resolve<Crisis> {

constructor(private cs: CrisisService, private router: Router) {}

resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Promise<Crisis> {

let id = route.params['id'];

return this.cs.getCrisis(id).then(crisis => {

if (crisis) {

return crisis;

} else { // id not found

this.router.navigate(['/crisis-center']);

return null;

}

});

}

}

*Router*

{

path: ':id',

component: CrisisDetailComponent,

canDeactivate: [CanDeactivateGuard],

resolve: {

crisis: CrisisDetailResolver

}

},

*Milestone 6: Asynchronous routing*

As you continue to build out feature areas, the overall application size will continue to grow. At some point you'll reach a tipping point where the application takes long time to load.

With asynchronous routing, which loads feature modules lazily, on request. Lazy loading has multiple benefits.

You can load feature areas only when requested by the user.

You can speed up load time for users that only visit certain areas of the application.

You can continue expanding lazy loaded feature areas without increasing the size of the initial load bundle.

Lazy Loading route configuration

{

path: 'admin',

loadChildren: 'app/admin/admin.module#AdminModule',

},

In app.module.ts, remove the AdminModule import statement from the top of the file and remove the AdminModule from the Angular module's imports array.

*CanLoad Guard: guarding unauthorized loading of feature modules*

You're already protecting the AdminModule with a CanActivate guard that prevents unauthorized users from accessing the admin feature area. It redirects to the login page if the user is not authorized.

But the router is still loading the AdminModule even if the user can't visit any of its components. Ideally, you'd only load the AdminModule if the user is logged in.

Add a CanLoad guard that only loads the AdminModule once the user is logged in and attempts to access the admin feature area.

Eg. Service

canLoad(route: Route): boolean {

let url = `/${route.path}`;

return this.checkLogin(url);

}

*Routing*

{

path: 'admin',

loadChildren: 'app/admin/admin.module#AdminModule',

canLoad: [AuthGuard]

},

The existing AuthGuard already has the essential logic in its checkLogin() method to support the CanLoad guard.

*Preloading: background loading of feature areas*

You've learned how to load modules on-demand. You can also load modules asynchronously with preloading.

After each successful navigation, the router looks in its configuration for an unloaded module that it can preload. Whether it preloads a module, and which modules it preloads, depends upon the preload strategy.

The Router offers two preloading strategies out of the box:

* + - No preloading at all which is the default. Lazy loaded feature areas are still loaded on demand.
    - Preloading of all lazy loaded feature areas.

To enable preloading of all lazy loaded modules, import the PreloadAllModules token from the Angular router package.

RouterModule.forRoot(

appRoutes

, { preloadingStrategy: PreloadAllModules }

)

*CANLOAD BLOCKS PRELOAD*

The PreloadAllModules strategy does not load feature areas protected by a CanLoad guard. This is by design.

You added a CanLoad guard to the route in the AdminModule a few steps back to block loading of that module until the user is authorized. That CanLoad guard takes precedence over the preload strategy.

*Custom Preloading Strategy*

Preloading every lazy loaded modules works well in many situations, but it isn't always the right choice, especially on mobile devices and over low bandwidth connections. You may choose to preload only certain feature modules, based on user metrics and other business and technical factors.

You can control what and how the router preloads with a custom preloading strategy.

In this section, you'll add a custom strategy that only preloads routes whose data.preload flag is set to true. Recall that you can add anything to the data property of a route.

Set the data.preload flag in the crisis-center route in the AppRoutingModule.

Eg.

{

path: 'crisis-center',

loadChildren: 'app/crisis-center/crisis-center.module#CrisisCenterModule',

data: { preload: true }

},

Add a new file to the project called selective-preloading-strategy.ts and define a SelectivePreloadingStrategy service class as follows:

*src/app/selective-preloading-strategy.ts (excerpt)*

import 'rxjs/add/observable/of';

import { Injectable } from '@angular/core';

import { PreloadingStrategy, Route } from '@angular/router';

import { Observable } from 'rxjs/Observable';

@Injectable()

export class SelectivePreloadingStrategy implements PreloadingStrategy {

preloadedModules: string[] = [];

preload(route: Route, load: () => Observable<any>): Observable<any> {

if (route.data && route.data['preload']) {

// add the route path to the preloaded module array

this.preloadedModules.push(route.path);

// log the route path to the console

console.log('Preloaded: ' + route.path);

return load();

} else {

return Observable.of(null);

}

}

}

SelectivePreloadingStrategy implements the PreloadingStrategy, which has one method, preload.

* ADVANCED

* Animations

You can build a simple animation that transitions an element between two states driven by a model attribute.

Animations are defined inside @Component metadata.

Before you can add animations, you need to import a few animation-specific imports and functions:

Eg.

import { BrowserModule } from '@angular/platform-browser';

import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

import { Component, Input } from '@angular/core';

import { trigger, state, style, animate, transition } from '@angular/animations';

Eg.

import { Heroes } from './hero.service';

@Component({

selector: 'hero-list-basic',

template: `

<ul>

<li \*ngFor="let hero of heroes"

[@heroState]="hero.state"

(click)="hero.toggleState()">

{{hero.name}}

</li>

</ul>

`,

styleUrls: ['./hero-list.component.css'],

animations: [

trigger('heroState', [

state('inactive', style({

backgroundColor: '#eee',

transform: 'scale(1)'

})),

state('active', style({

backgroundColor: '#cfd8dc',

transform: 'scale(1.1)'

})),

transition('inactive => active', animate('100ms ease-in')),

transition('active => inactive', animate('100ms ease-out'))

])

]

})

export class HeroListBasicComponent {

@Input() heroes: Heroes;

}

*States and transitions*

Angular animations are defined as logical states and transitions between states.

*States*

An animation state is a string value that you define in your application code.

In the example above, the states 'active' and 'inactive' are based on the logical state of hero objects.

The source of the state can be a simple object attribute, as it was in this case, or it can be a value computed in a method.

Eg.

state('inactive', style({

backgroundColor: '#eee',

transform: 'scale(1)'

})),

state('active', style({

backgroundColor: '#cfd8dc',

transform: 'scale(1.1)'

})),

These state definitions specify the end styles of each state.

They are applied to the element once it has transitioned to that state, and stay as long as it remains in that state.

In effect, you're defining what styles the element has in different states.

*Transitions*

After you define states, you can define transitions between the states.

Each transition controls the timing of switching between one set of styles and the next:Each

Eg.

transition('inactive => active', animate('100ms ease-in')),

transition('active => inactive', animate('100ms ease-out'))

Eg.

transition('inactive => active, active => inactive', animate('100ms ease-out'))

Eg.

transition('inactive <=> active', animate('100ms ease-out'))

You can also apply a style during an animation but not keep it around after the animation finishes.

You can define such styles inline, in the transition.

In this example, the element receives one set of styles immediately and is then animated to the next.

When the transition finishes, none of these styles are kept because they're not defined in a state.

Eg.

transition('inactive => active', [

style({

backgroundColor: '#cfd8dc',

transform: 'scale(1.3)'

}),

animate('80ms ease-in', style({

backgroundColor: '#eee',

transform: 'scale(1)'

}))

]),

*The wildcard state \**

The \* ("wildcard") state matches any animation state.

This is useful for defining styles and transitions that apply regardless of which state the animation is in.

Eg.

The active => \* transition applies when the element's state changes from active to anything else.

The \* => \* transition applies when any change between two states takes place.

*The void state*

The special state called void can apply to any animation.

It applies when the element is not attached to a view, perhaps because it has not yet been added or because it has been removed.

The void state is useful for defining enter and leave animations.

Eg.

\* => void transition applies when the element leaves the view, regardless of what state it was in before it left.

*Animation timing*

There are three timing properties you can tune for every animated transition: the duration, the delay, and the easing function.

They are all combined into a single transition timing string.

Eg.

Wait for 100ms and then run for 200ms, with easing: '0.2s 100ms ease-out'

*Multi-step animations with keyframes*

Animation keyframes go beyond a simple transition to a more intricate animation that goes through one or more intermediate styles when transitioning between two sets of styles.

For each keyframe, you specify an offset that defines at which point in the animation that keyframe applies.

The offset is a number between zero, which marks the beginning of the animation, and one, which marks the end.

Eg.

transition('void => \*', [

animate(300, keyframes([

style({opacity: 0, transform: 'translateX(-100%)', offset: 0}),

style({opacity: 1, transform: 'translateX(15px)', offset: 0.3}),

style({opacity: 1, transform: 'translateX(0)', offset: 1.0})

]))

]),

Note that the offsets are not defined in terms of absolute time.

They are relative measures from zero to one.

The final timeline of the animation is based on the combination of keyframe offsets, duration, delay, and easing.

Defining offsets for keyframes is optional. If you omit them, offsets with even spacing are automatically assigned.

For example, three keyframes without predefined offsets receive offsets 0, 0.5, and 1.

*Parallel animation groups*

You've seen how to animate multiple style properties at the same time: just put all of them into the same style() definition.

But you may also want to configure different timings for animations that happen in parallel.

For example, you may want to animate two CSS properties but use a different easing function for each one.

For this you can use animation groups.

In this example, using groups both on enter and leave allows for two different timing configurations.

Both are applied to the same element in parallel, but run independently of each other:

Eg.

transition('void => \*', [

style({width: 10, transform: 'translateX(50px)', opacity: 0}),

group([

animate('0.3s 0.1s ease', style({

transform: 'translateX(0)',

width: 120

})),

animate('0.3s ease', style({

opacity: 1

}))

])

]),

One group animates the element transform and width; the other group animates the opacity.

*Animation callbacks*

A callback is fired when an animation is started and also when it is done.

Eg.

template: `

<ul>

<li \*ngFor="let hero of heroes"

(@flyInOut.start)="animationStarted($event)"

(@flyInOut.done)="animationDone($event)"

[@flyInOut]="'in'">

{{hero.name}}

</li>

</ul>

`,

The callbacks receive an AnimationEvent that contains contains useful properties such as fromState, toState and totalTime.

Those callbacks will fire whether or not an animation is picked up.

* Directives overview

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.

Eg. ngFor, ngIf

* Attribute directives

—change the appearance or behavior of an element, component, or another directive.

Eg. ngStyle

*Build a simple attribute directive*

An attribute directive minimally requires building a controller class annotated with @Directive, which specifies the selector that identifies the attribute.

The controller class implements the desired directive behavior.

Eg.

import { Directive, ElementRef, Input } from '@angular/core';

@Directive({ selector: '[myHighlight]' })

export class HighlightDirective {

constructor(el: ElementRef) {

el.nativeElement.style.backgroundColor = 'yellow';

}

}

<p myHighlight>Highlight me!</p>

The import statement specifies symbols from the Angular core:

Directive provides the functionality of the @Directive decorator.

ElementRef injects into the directive's constructor so the code can access the DOM element.

Input allows data to flow from the binding expression into the directive.

*Respond to user-initiated events*

Currently, myHighlight simply sets an element color.

The directive could be more dynamic.

It could detect when the user mouses into or out of the element and respond by setting or clearing the highlight color.

Begin by adding HostListener to the list of imported symbols; add the Input symbol as well because you'll need it soon.

Eg.

import { Directive, ElementRef, HostListener, Input } from '@angular/core';

Then add two eventhandlers that respond when the mouse enters or leaves, each adorned by the HostListener decorator.

Eg.

@HostListener('mouseenter') onMouseEnter() {

this.highlight('yellow');

}

@HostListener('mouseleave') onMouseLeave() {

this.highlight(null);

}

private highlight(color: string) {

this.el.nativeElement.style.backgroundColor = color;

}

The @HostListener decorator lets you subscribe to events of the DOM element that hosts an attribute directive, the <p> in this case.

Pass values into the directive with an @Input data binding

Currently the highlight color is hard-coded within the directive. That's inflexible.

Start by adding a highlightColor property to the directive class like this:

@Input() highlightColor: string;

Binding to an @Input property

Notice the @Input decorator.

It adds metadata to the class that makes the directive's highlightColor property available for binding.

It's called an input property because data flows from the binding expression into the directive.

Without that input metadata, Angular rejects the binding;

Eg.

<p myHighlight highlightColor="yellow">Highlighted in yellow</p>

<p myHighlight [highlightColor]="'orange'">Highlighted in orange</p>

Bind to an @Input alias

Fortunately you can name the directive property whatever you want and alias it for binding purposes.

Restore the original property name and specify the selector as the alias in the argument to @Input.

Eg.

@Input('myHighlight') highlightColor: string;

<p [myHighlight]="color">Highlight me!</p>

Bind to a second property

@Input() defaultColor: string;

@HostListener('mouseenter') onMouseEnter() {

this.highlight(this.highlightColor || this.defaultColor || 'red');

}

Eg.

<p [myHighlight]="color" defaultColor="violet">

Highlight me too!

</p>

Component's own template may bind to any property of that component, with or without the @Input decorator.

But a component or directive shouldn't blindly trust other components and directives.

The properties of a component or directive are hidden from binding by default.

They are private from an Angular binding perspective.

When adorned with the @Input decorator, the property becomes public from an Angular binding perspective.

Only then can it be bound by some other component or directive.

* What are structural directives?

Structural directives are responsible for HTML layout.

They shape or reshape the DOM's structure, typically by adding, removing, or manipulating elements.

As with other directives, you apply a structural directive to a host element.

The directive then does whatever it's supposed to do with that host element and its descendents.

Structural directives are easy to recognize. An asterisk (\*) precedes the directive attribute name as in this example.

Eg.

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

Eg.

<ul><li \*ngFor="let hero of heroes; let i=index">{{i}} {{hero.name}}</li></ul>

Eg.

<div [ngSwitch]="hero?.emotion">

<happy-hero \*ngSwitchCase="'happy'" [hero]="hero"></happy-hero>

<sad-hero \*ngSwitchCase="'sad'" [hero]="hero"></sad-hero>

<confused-hero \*ngSwitchCase="'confused'" [hero]="hero"></confused-hero>

<unknown-hero \*ngSwitchDefault [hero]="hero"></unknown-hero>

</div>

Note.

* NgSwitch itself is not a structural directive.
* It's an attribute directive that controls the behavior of the other two switch directives.
* That's why you write [ngSwitch], never \*ngSwitch.

The <ng-template>

Group sibling elements with <ng-container>

*Write a structural directive*

you write an UnlessDirective structural directive that does the opposite of NgIf.

NgIf displays the template content when the condition is true.

UnlessDirective displays the content when the condition is false.

Eg.

<p \*myUnless="condition">Show this sentence unless the condition is true.</p>

Creating a directive is similar to creating a component.

Import the Directive decorator (instead of the Component decorator).

Import the Input, TemplateRef, and ViewContainerRef symbols; you'll need them for any structural directive.

Apply the decorator to the directive class.

Set the CSS attribute selector that identifies the directive when applied to an element in a template.

Eg.

import { Directive, Input, TemplateRef, ViewContainerRef } from '@angular/core';

@Directive({ selector: '[myUnless]'})

export class UnlessDirective {

}

TemplateRef and ViewContainerRef

A simple structural directive like this one creates an embedded view from the Angular-generated <ng-template> and inserts that view in a view container adjacent to the directive's original <p> host element.

You'll acquire the <ng-template> contents with a TemplateRef and access the view container through a ViewContainerRef.

You inject both in the directive constructor as private variables of the class.

Eg. HTML

<p \*myUnless="condition" class="unless a">

(A) This paragraph is displayed because the condition is false.

</p>

Directive

import { Directive, Input, TemplateRef, ViewContainerRef } from '@angular/core';

/\*\*

\* Add the template content to the DOM unless the condition is true.

\*/

@Directive({ selector: '[myUnless]'})

export class UnlessDirective {

private hasView = false;

constructor(

private templateRef: TemplateRef<any>,

private viewContainer: ViewContainerRef) { }

@Input() set myUnless(condition: boolean) {

if (!condition && !this.hasView) { this.viewContainer.createEmbeddedView(this.templateRef);

this.hasView = true;

} else if (condition && this.hasView) {

this.viewContainer.clear();

this.hasView = false;

}

}

}

Browser Support

Polyfills

Angular is built on the latest standards of the web platform.

Targeting such a wide range of browsers is challenging because they do not support all features of modern browsers.

You can compensate by loading polyfill scripts ("polyfills") on the host web page (index.html) that implement missing features in JavaScript.

Eg.

<script src="node\_modules/core-js/client/shim.min.js"></script>

A particular browser may require at least one polyfill to run any Angular application.

You may need additional polyfills for specific features.

Chrome, Firefox, Edge, Safari 9+ None

Safari 7 & 8, IE10 & 11, Android 4.1+ ES6

IE9 ES6, classList

COMPONENT STYLES

Angular applications are styled with standard CSS.

That means you can apply everything you know about CSS stylesheets, selectors, rules, and media queries directly to Angular applications.

Additionally, Angular can bundle component styles with components, enabling a more modular design than regular stylesheets.

Loading styles into components

There are several ways to add styles to a component:

By setting styles or styleUrls metadata.

Inline in the template HTML.

With CSS imports.

Styles in metadata

You can add a styles array property to the @Component decorator. Each string in the array (usually just one string) defines the CSS.

Eg.

@Component({

selector: 'hero-app',

template: `

<h1>Tour of Heroes</h1>

<hero-app-main [hero]=hero></hero-app-main>`,

styles: ['h1 { font-weight: normal; }']

})

export class HeroAppComponent {

/\* . . . \*/

}

Style URLs in metadata

You can load styles from external CSS files by adding a styleUrls attribute into a component's @Component decorator:

Eg.

@Component({

selector: 'hero-details',

template: `

<h2>{{hero.name}}</h2>

<hero-team [hero]=hero></hero-team>

<ng-content></ng-content>

`,

styleUrls: ['app/hero-details.component.css']

})

export class HeroDetailsComponent {

/\* . . . \*/

}

Template inline styles

You can embed styles directly into the HTML template by putting them inside <style> tags.

Eg.

@Component({

selector: 'hero-controls',

template: `

<style>

button {

background-color: white;

border: 1px solid #777;

}

</style>

<h3>Controls</h3>

<button (click)="activate()">Activate</button>

`

})

Template link tags

You can also embed <link> tags into the component's HTML template.

As with styleUrls, the link tag's href URL is relative to the application root, not the component file.

Eg.

@Component({

selector: 'hero-team',

template: `

<link rel="stylesheet" href="app/hero-team.component.css">

<h3>Team</h3>

<ul>

<li \*ngFor="let member of hero.team">

{{member}}

</li>

</ul>`

})

CSS @imports

CSS @imports

You can also import CSS files into the CSS files using the standard CSS @import rule.

In this case, the URL is relative to the CSS file into which you're importing.

Eg.

src/app/hero-details.component.css (excerpt)

@import 'hero-details-box.css';

Controlling view encapsulation: native, emulated, and none

As discussed earlier, component CSS styles are encapsulated into the component's view and don't affect the rest of the application.

To control how this encapsulation happens on a per component basis, you can set the view encapsulation mode in the component metadata.

DEPLOYMENT

???????????

HIERARCHICAL DEPENDENCY INJECTORS

????????????

REACTIVE FORMS

????????????