Angular - Concepts

# App

## Concept

Angular is a platform and framework for building single-page client applications using HTML and TypeScript, built up from NgModules.

An Angular app is defined by a set of NgModules, with atleast a root module and typically many more feature modules.

* Components define views and modify according to logic and data
* Components use service which provide specific functionality not directly related to views. Services can be injected into components as dependencies, making the code modular, reuseable, and efficient.

Modules, components, and services are classes that use decorators, which mark their type and provide metadata so angular knows how to use them.

* Metadata for a component class assoicates with a template which defines the view
* Metadata for a service class provides the information angular needs to make it available to components through dependency injection

# Modules

## Concept

Angular NgModules declare a compliation context for a set of components which are dedicated to an application domain, workflow, or set of capabilities. The metadata provided to the module describes how the compile the componets template and create an injector at runtime. This meta data allows Angular to convert the modular application into highly performant JavaScript to run on the browser.

Each module can import and export functionality to other modules, allow for modular application.

Every Angular application has a root module called AppModule, which provides the bootstrap mechanism to lanch the application. The app typically contains many other functional components, all of which share the same compilation context, and can also be lazy loaded if required.

## Definition

An NgModule is defined by a class decorated with @NgModule(), which takes a single metadata object. The object contains:

* declarations - declare the declarables which belong to the module: **components**, **directives**, and **pipes**. To use a declarable, it must be defined in exactly one module, to use a declarable more than once, import the modules which has it declared already.
* exports - Subset of delcarations which will be visible and usable in the component templates of oter NgModules
* imports - other modules exported classes which are needed by component templates declared in **this** NgModule
* providers - creates of services that this NgModule contributest to the global collection (become accessible in all parts of the app)
* bootstrap - main application view, called the root component. Only required by root NgModule

### Commonly Used Modules

* BrowserModule - allows app to run on browser
* CommonModule - (imported by BrowserModule) NgIf and NgFor, used by feature modules to import stated directives
* FormsModule - template driven forms NgModel
* ReactiveFormsModule - reactive forms
* RouterModule - RouterLink, .forRoot(), and .forChild()
* HttpClientModule - Request API server

### NgModule Categories

NgModules are typically used to organize an app and keep related code for specific functionality separate from other code. There are various different typical categories for NgModules:

* Domain - Organized around feature or user experience. e.g. placing an order. Only exports one component, which is the feature.
* Routed - Top component of NgModel acts as destination of router route. Routed modules are lazy loaded and don't export anything since their components never appear in the template of an external component.
* Routing - Provides routing configuration for another module. The name of a routing module should parallet the name of its companion NgModule, e.g. ContactModule should have a ContactRoutingModule (contact-routing.module.ts)
* Service - Provides utility serivices such as data access and messaging. Service modules should consist entirely of providers
* Widget - Makes component, directive, or pipe available to other modules, rarely contains any providers.
* Shared - makes set of component, directives, or pipes available to other modules

### Feature Modules

A feature module is a best practice for organising a code base. It delivers a cohesive set of functionalities based on a specific application need. Feature modules are almost the same as root modules, apart from they are imported into a root module and use CommonModule instead of BrowserModule.

Feature modules export their components so they can be used in external module templates.

# Components

## Concept

Each Angular application has atleast one compoent, the root component, which connects the component hierarchy to the DOM. Each component controls a patch of screen called a view, and generally a components job is to enable user experience and nothing more.

### Templates

A template combines HTML with Angular markup whch can modify HTML elements before they are displayed. Template directives contain program logic, and binding markup connect the application data to the dom through:

* Event binding - app responds to user input
* Property binding - interpolate values that are computed from application data into the HTML

Angular two-way binding allows changes in the DOM to be reflected in the program data.



Pipes can be used to improve user experience by transforming values for display.

## Definition

Components are the main building block of Angular applications. The @Compoennt() decorator identifies the class immediatley below it as a component and provides the template and related component-specific metadata. The Component metadata object must have:

* selector - component name, when present in html, component will be rendered
* templateUrl - url to html template (for very simple components it can simply be 'template' and have a string html instead)
* styleUrls - array of urls to stylesheets

Each component is self contained in its own folder under /app and consists of:

* HTML template which declares what is rendered on the page <name>.component.html
* TypeScript class that defines behaviour - <name>.component.ts
* CSS selector that defines how the component is used in a template <name>
* CSS styles applied to the template - <name>.component.css
* test spec - <name>.component.spec.ts

Components can be generated using ng generate component <component-name> or created manually.

## Lifecycle

Each component has a lifecycle that starts when Angular instantiates the component and renders it to the DOM. The application component can implement one or more of the lifecycle hook methods (found in the core library) to run functionality during key events in the component lifecycle, for example responding to updates during change.

### Methods

Some of the hook methods are:

* ngOnInit() - Called once shortly after checking input properties on component for first time
* ngOnChanges() - Called before ngOnInit() and whenever one or more data-bound properties is changed. Method receives SimpleChanges object containg current and previous properties. Called very often, keep logic lean.
* ngDoCheck() - Called after ngOnChanges()
* ngAfterContentInit() - Called once after first ngDoCheck()
* ngAfterContentChecked() - Called after ngAfterContentInit, and every ngDocheck()
* ngAfterViewInit() - Called once after first ngAfterContentChecked() to respond after angular initalized the component and its child views
* ngAfterViewChecked() - Called after ngAfterviewInit() and after every ngAfterContentChecked()
* ngOnDestroy() - Called immediatley before angular destroys component

### Implement

To use a lifecycle method, it must be imported and implemented by the component. For example:

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

@Component({

selector: 'app-hero-detail',

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

styleUrls: ['./hero-detail.component.css']

})

export class HeroDetailComponent implements OnInit {

constructor() { }

ngOnInit() {

}

}

### Change Direction

Like react, Angular uses unidirectional data flow. This means parents pass the data down the tree to their children, and children cannot normally change data in their parents.

Since parents are rendered first, to invert the data flow, a new change detection cycle must be triggered by the child to allow the change to be rendered.

In the parent component a property decorated with @ViewChild, can be accessed in the AfterViewInit() and AfterViewChecked() hooks. The property will be an updated version of the view child and can therefore be used to update the data in the parent component, triggering a new change detection cycle and re-rendering the tree. For example:

@ViewChild(ChildViewComponent) viewChild: ChildViewComponent;

ngAfterViewChecked() {

// viewChild is updated after the view has been checked

if (this.prevHero === this.viewChild.hero) {

this.logIt('AfterViewChecked (no change)');

} else {

this.prevHero = this.viewChild.hero;

this.logIt('AfterViewChecked');

this.doSomething();

}

}

Note: Since AfterViewInit() and AfterViewChecked() hooks fire after the component view has been composed, any immediate modifications to the data-bound properties need to be performed after the current js event cycle. This is so they can trigger a new change detect cycle and be rendered and checked. To do this, set a 0ms timeout.

### Transclusion

Transclusion is content projection in Angular and is the process to insert html from outside the component into a designated spot in the template.

The ng-content tag is used as a placeholder in the component template for the contentChild property, which is decorated with @CotentChild. It follows the same syntax as change @ViewChild, however there is no need to wait for content updates since Angular completes the composition of the projected content before it finishes he composition of the host components view.

## Template Syntax

Angular template syntax combines HTML with Angular syntax.

### Variables

A template variable can refer to the following:

* a DOM element within a template
* a directive
* an element
* TemplateRef
* a web component

A variable is defined by a # symbol in an element:

<input #phone placeholder="phone number" />

The variable can then be refered to by name.

<button (click)="callPhone(phone.value)">Call</button>

### Interpolation

Interpolates the components hero.name properly value into HTML

{{hero.name}}

### Data Binding

### Angular provides three categories of data binding according to the direction of data flow. The target name is to the left of the equal sign contained in different brackets to define data flow direction:

* [] - from the source to view
* () - from view to source
* [()] - two way sequence of view to source to view

The target can be a property, event, or attribute name.

### Property Binding - source to view

HTML can be customized by specifying attributes with string values. Data binding can be used to control things such as the state of a button:

<button [disabled]="isUnchanged">Save</button>

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

Data binding works with js properties of DOM elements, components, and directives, not HTML attributes.

### Event binding - view to source

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

Event binding on an element or component binds the event to a method in the parent component.

It is then possible for the child component to emit this to a parent.

### Two Way Data Binding

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

Two way data binding (mainly used in forms) combines the proper and event binding in a single notation. For example, form input will update the model and therefore the form field and other reliant bindings. It is also used to pass data between parent and child components.

A component can use two way binding in its input properties when it has an @Input and @Output property named <name> and <name>Changed. The two way binding is then referenced in the template as [(<name>)]. NgModel must be used for form elements.

Angular processes all data bindings once for each JavaScript event cycle.

### Attributes and Styles

<https://angular.io/guide/attribute-binding#binding-to-the-style-attribute>

Binding to an element attribute is similar to property binding, but the name of the attribute is preceded by 'attr.':

<p [attr.attribute-you-are-targeting]="expression"></p>

Classes can be added dynamically using the class prefix and a value used conditionally. For example, if onSale is truthy, the class 'style' will be applied:

[class.sale]="onSale"

Multiple classes are added using [class] with the input being either a string, array, or object with the key names as class names and value as conditional expression.

Stlye can be applied usingn the [style] property.

## Styles

CSS styles of each component are encapsulated, so they won't affect the rest of the application or any child components. There are three different view encapsulation modes, which can be set in the component metadata:

* ShadowDom - uses browsers native shadow DOM implementation to attach shadow DOM to component and put view and styles inside it
* Emulated - emulates the behaviour of shawdow DOM by preprocessing and renanming CSS code, scoping it to component
* None - no view encapsulation

Encasulation is set in metadata:

encapsulation: ViewEncapsulation.ShadowDom

### Specific Styles

The component styles have a few special selectors from shadow DOM style scoping:

* :host - target styles in the element that hosts the component. Use brackets to select specific hosts, e.g. :host(.active) {}
* :host-context(<class-name>) - looks for a CSS class in any ancestor of the component host element up to the document root, then if matched it applies it rules to the component. Can be good for themes, since if the root has a theme selector, specific styles can be applied

## Component Interaction

### Parent to Child - Input Binding

Pass data from the parent to a child using input binding.

Child has inputs decorated with @Input:

@Input() hero: Hero;

Parent passes data to child using property binding:

<app-hero-child \*ngFor="let hero of heroes" [hero]="hero"></app-hero-child>

### Child Intercept Input Property Changes

Use getter and setter to modify bound inputs:

export class NameChildComponent {

@Input()

get name(): string { return this.\_name; }

set name(name: string) {

this.\_name = (name && name.trim()) || '<no name set>';

}

private \_name = '';

}

Use ngOnChanges():

ngOnChanges(changes: SimpleChanges) {

const log: string[] = [];

for (const propName in changes) {

const changedProp = changes[propName];

const to = JSON.stringify(changedProp.currentValue);

if (changedProp.isFirstChange()) {

log.push(`Initial value of ${propName} set to ${to}`);

} else {

const from = JSON.stringify(changedProp.previousValue);

log.push(`${propName} changed from ${from} to ${to}`);

}

}

this.changeLog.push(log.join(', '));

}

### Parent Listens For Event Emitter

Child components can expose an EventEmitter property which emit events when something happens, the parent can bind to this and react.

EventEmitter properties are descorated with @Output(),

@Component({

selector: 'app-voter',

template: `

<button (click)="vote(true)" [disabled]="didVote">Agree</button> `

})

export class VoterComponent {

@Output() voted = new EventEmitter<boolean>();

didVote = false;

vote(agreed: boolean) {

this.voted.emit(agreed);

this.didVote = true;

}

}

The parent then binds to the event, the property name must match the child event/eventemitter property, and have the argument set as $event:

@Component({

selector: 'app-vote-taker',

template: `

<app-voter \*ngFor="let voter of voters"

(voted)="onVoted($event)">

</app-voter>

`

})

export class VoteTakerComponent {

voters = ['Narco', 'Celeritas', 'Bombasto'];

onVoted(agreed: boolean) {

console.log(agreed)

}

}

### Reference by Template Variable

If data from a child is only required in the template, a template variable can be used on the child and then used throughout the template.

### Parent Calls @ViewChild

Parent can call @ViewChild to access data, however note the viewchild will not be available until after ngAfterViewInit(), so initial values of properties in the parent cannot reference the view child.

### Service

Parent and Child share service and subscribe to changes. Service uses rxjs subject to annonce new changes and has observable string streams

# Services

## Concept

For data and logic which is not associated with a specific view, and needs to be shared across components, a serivce class is created. Services are typically a class with narrow we define purposes, being commonly used to extract functions such as data fetching, user input validation, or logging, out of the components.

### Injection

Services are injected into components as dependencies. In Angular dependency injection is done using two objects:

* Injector - creates dependencies, maintaining a container of dependency instances that it reuses if possible.
* Provider - an object which tells an injector how to obtain or create a dependency.

To use a dependency in an app, a provider must be registered when the service is defined in the component, injectors are then created automatically by Angular.

### Router

The Router NgModule provides a service which defines navigation paths amoung different application states and view hierarchies in the app. It does this my mapping URL-like paths to views in stead of pages, and intercepts normal link behaviour to show and hide view hierarchies.

## Defintion

Service classes are preceded by the @Injectable() decorator, allowing the metadata required to inject them as dependencies into classes.

A typical service class would be a logger:

export class Logger {

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

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

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

}

Atleast one provider must be registered for a service. Providers can either be part of the services own metadata making it available everywhere or be with specific modules/components. By default, the service is registered a provider with the root injector:

@Injectable({

providedIn: 'root',

})

Angular makes one instance and if it is not used it is removed during tree shaking. Otherwise, register with the specific NgModule or component in the providers array, making it available to be added to all child components:

@NgModule({

providers: [

BackendService,

Logger

],

...

})

Once injected, the service is available to be assigned to a property in the component constructor. Angular checks the constructor for services, if it has already made it, it will use it, otherwise the injector will make one using the registered provider:

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;

}

}

### Teardown

For services which provide subscriptions, in the teardown on the subscriber they need to disconnect to avoid memory leaks. However, if the subsriber is also the provider of the service, then there is no need to disconnect, as the service will also be destroyed on the removal of the providing component.

# Pipes

## Concept

Pipes are a class which define a function which transforms input values to outputs values for display in a view.

Pipes are defined as pure by default, so angular will only execute a pipe on a pure change. This means firstly that the pipe must have no side effects, and secondly that only primitive input value or changed object reference will trigger a pipe (there is no deep checking of objects). It is possible to mark the pipe as inpure in the metadata.

## Definiton

Pipes are a class decorated with the @pipe() decorator. There are various predefined pipes in Angular, for example currency conversion, and custom pipes can also be created.

List of pipes: <https://angular.io/api?type=pipe>

### Template Syntax

Pipes are used in Angular HTML templates by interpolated values and the pipe operator. For example:

{{interpolated\_value | pipe\_name}}

Pipes can be chained, and desired format can be passed to the pipe via string.

<p>The time is {{today | date:'shortTime'}}</p>

### Class Syntax

Pipes are classes decorated with @Pipe and implement the PipeTransform interface. The pipe class must have a transform method, which is invoked by Angular to perform the pipe.

The transform method has the value of the binding as the first argument and any parameters passed as the second argument in list form. For example:

import { Pipe, PipeTransform } from '@angular/core';

@Pipe({name: 'exponentialStrength'})

export class ExponentialStrengthPipe implements PipeTransform {

transform(value: number, exponent?: number): number {

return Math.pow(value, isNaN(exponent) ? 1 : exponent);

}

}

# Directives

## Concept

Angular templates are dynamic, transforming the DOM according to the instructions given by the directives. While a component is technically a directive, they are so distinctive that they have a separate decorator.

There are two types of directive, structural and attribute:

### Structural

Structural directives alter layout by adding remove and replacing elements in the DOM. There are various built in structural directives, all have an asterisk before as convenience notation, such as:

* \*ngFor - repeat a node of each item in a list
* \*ngIf - conditionally create or destroy subview
* [ngSwitch] - switch between views

### Attribute

Attribute directives alter the appearance or behaviour of an exisiting element. They look like regular html attributes on elements. There are various built in attribute directives:

* NgClass - adds or removes CSS classes
* NgStyle - add or removes HTML styles
* NgModel - adds two way databind to forms

## Definition

A directive class is decorated with the @Directive() decorator.

### Structural

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

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

Called in the HTML template, the structural directives perform a function, for example, ngFor is an interative, and ngIf is a conditional

A variable named after 'let' is called a template input variable. The scope of the input variable is limited to the single instance of repeated template.

ng-containers can be used to contain structural directives, this prevents extra levels of html when adding nested directives or directives for elements which can't use them. For example:

<ng-container \*ngIf="hero">

and saw {{hero.name}}. I waved

</ng-container>

ng-template is an angular element for rendering HTML, when rendered it anguar replaces it with its content if a structural directive tells it to, otherwise it is replaced with a comment.

Custom directives are written with a setter, for example:

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

/\*\*

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

\*/

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

export class UnlessDirective {

private hasView = false;

constructor(

private templateRef: TemplateRef<any>,

private viewContainer: ViewContainerRef) { }

@Input() set appUnless(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;

}

}

}

### Attribute

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

Attribute directives are placed in square brackets in a HTML element's attributes list.

Attribute directives can be built and added to elements as attributes in order to apply custom functionality, for example highlight an element on mouse enter etc.

Attribute directives are declared in Angular modules in the same way a components, so ensure it has a selector and is declared in the root or component in which it will be used.

Decorate with @Directive, include ElementRef to access the element, and HostListener to access events on the element. For example:

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

@Directive({

selector: '[appHighlight]'

})

export class HighlightDirective {

constructor(private el: ElementRef) { }

@Input() defaultColor: string;

@Input('appHighlight') highlightColor: string;

@HostListener('mouseenter') onMouseEnter() {

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

}

@HostListener('mouseleave') onMouseLeave() {

this.highlight(null);

}

private highlight(color: string) {

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

}

}

The apply to elements:

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

<p [appHighlight]="color" defaultColor="violet">Highlight me too!</p>

# Routing

## Lazy Loading Modules

By default, all NgModules are eagerly loaded and bundled in the main app bundle, which can lead to large bundles and slow loading times. Lazy loading only loads modules when they are required, commonly when a user changes route and a route specific module will be shown.

### Lazy Load in Routes

<https://stackblitz.com/angular/ymgpgveevoo?file=src%2Fapp%2Fcustomers%2Fcustomers-routing.module.ts>

To lazy load a module in the Angular router, use the loadChildren key in the app routing module:

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

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

const routes: Routes = [

{

path: 'customers',

loadChildren: () => import('./customers/customers.module').then(m => m.CustomersModule)

},

{

path: '',

redirectTo: '',

pathMatch: 'full'

}

];

@NgModule({

imports: [

RouterModule.forRoot(routes)

],

exports: [RouterModule],

providers: []

})

export class AppRoutingModule { }

And simply add the base route to the feature module component in the component routing module since the app router has already set the route:

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

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

import { CustomersComponent } from './customers.component';

const routes: Routes = [

{

path: '',

component: CustomersComponent

}

];

@NgModule({

imports: [RouterModule.forChild(routes)],

exports: [RouterModule]

})

export class CustomersRoutingModule { }

forRoot(routes) in the AppRoutingModule lets Angular know that the module is a routing module. forChild(routes) is added in child routing modules to let Angular know it is a routing module and those routes need to be applied to the routing service

# Observables

Observables provide support for passing message between different parts of the application, such as event handling and async programming. The observer maintains a list of components called observers and notifies them of state changes, similar to publish/subscribe patterns.

Observables are delcarative, so a function is defined for publishing values, but it is not executed until a consumer subscribes to it. Once subscribed a consumer will receive notification until the function completes.

The advantage of observables is the setup and teardown logic is handled by the observable, meaning the subsriber only need to worry about subscribing and unsubscribing.

## Definition

Create an Observable instance which defines a subscriber function called using subscribe(). A consumer calls the subscribe function passing an observer function which handles the notification received from the Observable.

For example:

const locations = new Observable((observer) => {

let watchId: number;

// Simple geolocation API check provides values to publish

if ('geolocation' in navigator) {

watchId = navigator.geolocation.watchPosition((position: Position) => {

observer.next(position);

}, (error: PositionError) => {

observer.error(error);

});

} else {

observer.error('Geolocation not available');

}

// When the consumer unsubscribes, clean up data ready for next subscription.

return {

unsubscribe() {

navigator.geolocation.clearWatch(watchId);

}

};

});

The consumer passes an object with implements the Observer interface. It is an object that defines callback methods to handle the three types of notification:

* next - required to handle each delivered value
* error - optional handling of errors
* complete - optional handling complete notification

So, for example, subscribe with:

// Call subscribe() to start listening for updates.

const locationsSubscription = locations.subscribe({

next(position) {

console.log('Current Position: ', position);

},

error(msg) {

console.log('Error Getting Location: ', msg);

}

});

// Stop listening for location after 10 seconds

setTimeout(() => {

locationsSubscription.unsubscribe();

}, 10000);

With the current design the observable will generate a new stream for each subscriber. It is possible to multicast and only have one stream:

<https://angular.io/guide/observables#multicasting>

## RxJS

Reactive Extensions for JavaScript (RxJS) is a library for reactive programming using observables which makes it easier to compose asynchronous callback based code. RxJS provides an implemenation of Observable and provides utility functions for creating and working with it.

Observables are commonly named with a $ sign at the end of their variable name. This allows for quick scanning, and the ability to use the same name for the return value just without the $.

### from

The 'from' function will create a observable out of a promise, for example an ajax request:

import { from } from 'rxjs';

const data$ = from(fetch('/api/endpoint'));

data$.subscribe({

next(response) { console.log(response); },

error(err) { console.error('Error: ' + err); },

complete() { console.log('Completed'); }

});

### interval

The 'interval' function will create an observable counter, which will run the next function each x period:

import { interval } from 'rxjs';

const secondsCounter$ = interval(1000);

const subscription = secondsCounter$.subscribe(n =>

console.log(`It's been ${n + 1} seconds since subscribing!`));

### fromEvent

Create an observable from an event:

import { fromEvent } from 'rxjs';

const el = document.getElementById('my-element');

const mouseMoves$ = fromEvent(el, 'mousemove');

const subscription = mouseMoves$.subscribe((evt: MouseEvent) => {

console.log(`Coords: ${evt.clientX} X ${evt.clientY}`);

// When the mouse is over the upper-left of the screen,

// unsubscribe to stop listening for mouse movements

if (evt.clientX < 40 && evt.clientY < 40) {

subscription.unsubscribe();

}

});

### ajax

Create an observable which performs an ajax request:

import { ajax } from 'rxjs/ajax';

const apiData$ = ajax('/api/data');

apiData$.subscribe(res => this.apiData = res.response);

### Operators

Operators are functions which build on obserables, enabling manipulation of collections. There are four main operators:

* map()
* filter()
* concat()
* flatMap()

Operators accept configuration options and return a function which accepts the source observable. The operator then returns a new observable with the stream modified by the configuration.

For example:

import { of } from 'rxjs';

import { map } from 'rxjs/operators';

const nums = of(1, 2, 3);

const squareValues = map((val: number) => val \* val);

const squaredNums$ = squareValues(nums);

squaredNums$.subscribe(x => console.log(x));

The pipe function allows multiple operators to be used together, eg:

const squareOddVals$ = pipe(

filter((n: number) => n % 2 !== 0),

map(n => n \* n)

);

Error catching can be added to the end of a pipe to ensure errors are handled. The retry operator retries requests mutliple times before failing. It is also possible to use the .pipe method of a rxjs function:

import { map, retry, catchError } from 'rxjs/operators';

const apiData$ = ajax('/api/data').pipe(

map((res: any) => {

if (!res.response) {

throw new Error('Value expected!');

}

return res.response;

}),

retry(3),

catchError(err => of([]))

);

## Angular and RxJS

### EventEmitter

Angular's EventEmitter class extends RxJS Subject, adding an emit() method to arbitrary values can be sent.

### HttpClient

Angular's HttpClient returns observables from HTTPs method calls. This gives advantages over promised based HTTP APIs:

* Observables will not mutate server response, since operators transform data
* HTTP requests are cancellable through unsubscribe()
* Request can be configured to get proggress event updates
* Failed request can be retried easily

### Async Pipe

AsyncPipe subscribes to an observable and returns the latestest emitted value. E.g.

@Component({

selector: 'async-observable-pipe',

template: `<div><code>observable|async</code>:

Time: {{ time | async }}</div>`

})

export class AsyncObservablePipeComponent {

time = new Observable<string>(observer => {

setInterval(() => observer.next(new Date().toString()), 1000);

});

}

### Reactive Forms

Reactive forms have properties which raise change events on valueChanges and statusChanges. This allows for application logic to occur on the stream from a form input.

### Type Ahead Suggestions

Type ahead suggestions need to:

* Listen for data from an input.
* Trim the value (remove whitespace) and make sure it’s a minimum length.
* Debounce (so as not to send off API requests for every keystroke, but instead wait for a break in keystrokes).
* Don’t send a request if the value stays the same (rapidly hit a character, then backspace, for instance).
* Cancel ongoing AJAX requests if their results will be invalidated by the updated results.

Observables can be used to make type ahead suggestions logic much simpler using a series of operators. For example:

const typeahead = fromEvent(searchBox, 'input').pipe(

map((e: KeyboardEvent) => (e.target as HTMLInputElement).value),

filter(text => text.length > 2),

debounceTime(10),

distinctUntilChanged(),

switchMap(searchTerm => ajax(`/api/endpoint?search=${searchTerm}`))

);

# Injectors

Injectors in angular have rules which define the visibility of injectables in apps.

## Hierarchies

There are two injector hierarchies in angular:

* ModuleInjector - configure a ModuleInjector using @NgModule() or @Injectable()
* ElementInjector - created implicity at each DOM element, by default it is empty unless it is configured in the providers property on a @Directive or @Component

### ModuleInjector

Configured using either:

* @Injectable() providedIn property
* @NgModule() providers array

providedIn is preferable as it allows treeshaking when the injectable is not used.

The ModuleInject is a flattening of all the provider array which can be reached by following the NgModule.imports recursivley. Child ModuleInjectors are created when lazy loading other NgModules.

### ElementInjector

Angular create the ElementInject implictiy for each DOM element, providing a service in the @Component() decorator configure this. When providing a service in an element as a provider, a new instance of the service will be created for each component created. This can be useful for situtation where each component would want its own service, as not to mutate and affect other components. For example, edit mutliple files at the same time.

## Service Resolution

When resolving a service token for a component or directive, angular resolves in in two phases. First against the ElementInjector hierarchy for the element, then agaisnt the ModuleInjector hierarchy. If a token is not found, then an error is thrown.

The resolution can be modified with:

@Optional() - If can't be resolved at runtime no error is thrown

export class OptionalComponent {

constructor(@Optional() public optional?: OptionalService) {}

}

@Self() - Only look in self, can be used to inject a service but only if available in the current host element.

@SkipSelf - Opposite of @Self(), Angular starts search in parent.

@Host - Designates a component as the last stop in the injector tree to look for the provider.

## Providing Services in @Component

Services can be provided in a component in two ways. Using the followin key in the @Component() decorator metadata:

* providers - array
* viewProviders - array

### providers

Services provided here will be available for injection to the component and its children, since it will be added to the ElementInjector.

### viewProviders

viewProviders is very similar to providers but it creates services only available to element defined in the template of the component. Therefore, projected content will not be able to access services given in viewProviders they are projected into.

## Defining Providers

Various values can be set in providers when they are defined.

### useValue

The useValue options allows a fixed value to be associated with a DI token. This is used to provide runtime configuration such as website addresses and feature flags. For example:

{ provide: Hero, useValue: someHero }

Proivde values do not have to be classes, they can also be tokens such as 'TITLE' below. It creates an InjectonToken instance:

{ provide: TITLE, useValue: 'Hero of the Month' },

### useClass

The useClass key creates and returns a new instance of the specified class:

{ provide: LoggerService, useClass: DateLoggerService },

### useExisting

Map one token to another, creating an alias for the other service. Often used on logger services to reduce or increase outputs on specific components.

### useFactory

Create dependency from factory function.