Angular Quick Reference

# Module (pp 98)

## Decription

Modules are logical containers which groups Components, Directives, Pipes and Services.

An Angular application typically has one main Module which can have dependencies to other modules.

To create a module decorate a class with the @NgModule() decorator.

NgModule has the following metadata:

* declarations. Specify the **Components**, **Directives** and **Pipes** that belong to this module
* exports. Specify which **Components**, **Directives** and **Pipes** are publicly accessible from other modules
* imports. What other Modules is this module dependent on
* providers. Which **Services** does this module add to the global collection of injectable services
* bootstrap: Which **Component** serves as the startup (root) Component (main function). Typically only used in the root module

## Example:

@NgModule({

declarations: [ Component, Pipe, Directive ],

exports: [Component, Pipe, Directive ]

imports: [ DependentModule1, DependentModule2 ],

providers: [Service1, Service2],

bootstrap: [RootComponent]

})

export class SomeModule { }

## CLI command:

ng generate module <module name>

or

ng g -m <module name>

# Component (pp 71)

## Description

A Component represents a visual part of the screen. Therefor a Component must have a **View** (template).

A View is nothing more than a html fragment and related stylesheets.

Components can have their own data. They provide their own **Model** which can be bound to in the **View**

In a sense, components can be seen as custom html elements. In fact, since Angular 5 the Angular components can be used in non-angular web sites. This is referred to as **Angular Elements**

Components can accept arguments from outside by using **@Input()** decorators. They translate to custom attributes on the html element.

Components can communicate to the outside world by using **@Output()** decorators. They translate to custom events on the html element.

Components have a lifecycle. You can intercept some phases of the lifecyle

* OnInit (ngOnInit()). The componentent is fully initialized and data is bound.
* OnDestroy (ngOnDestroy()). The component is about to be released.
* OnChanges (ngOnChanges()). Fired when one or more data-bound properties changes

For a complete list see documentation on angular.io

To create a Component decorate a class the @Component() decorator

@Component has the following metadata:

* selector. The element name of the component. Typically, this will be the html tag name but can also be an attribute name. To discriminate the usage Angular follows the css selector namings: tag-name = tag, [attribute-name] = attribute
* template. Inline visual representation of the component. (Inline view)
* templateUrl. External visual representation of the component. (External View). Reference to an html file
* styles. List of Inline css styles
* styleUrls. List of external stylesheets

## Example

@Component({

selector: 'element-name',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

}

## CLI command

ng generate component <component-name>

or

ng g -c <component-name>

# Directive (pp 105)

## Description

Use directives if you want to add custom behavior to existing components.

It’s like a Component without visual elements (templates)

To create a Directive use the @Directive() decorator.

@Directive() has the following metadata

* selector. The name of the directive in an html document. Typically an attribute. Names follows the css selector syntax
* inputs. Input data
* outputs. Events this Directive can fire

Most of the time you use already defined Directives

* \*ngIf=’<boolCondition>’ hides or shows the element.

\*ngIf=’<boolCondition>; then <thenBlock>; else <elseBlock>’ hides or shows <thenBlock> or <elseBlock>

* \*ngFor=’let item of collection’ repeats this element for each item in collection.

\*ngFor=’let item of collection; **index** as idx; **odd** as uneven; **even** as notodd; **first** as no1; **async**

* [ngClass] = “angular expression”. Angular expression can be
  + literal: ‘className’
  + variable: varName
  + list: [‘className1’, ‘className2’]
  + object: {‘className1’:true, ‘className2’: false}
* [ngStyle]=”angular expression”. Angular expression can be
  + object. {‘color’: ‘red’, ‘font-style’: expr}

## Examples

@Directive({

selector: 'my-directive',

inputs:[‘name1’, ‘name2’]

})

export class MyDirective {

name1: string;

name2:string;

}

## CLI command

ng generate directive <directive-name>

Or

ng g -d <directive-name>

# Pipe ()

## Description

Pipes are used to transform data. They appear behind a | sign in an Angular expression. For example,

“Hello” | uppercase results in “HELLO”

Pipes can have additional arguments which are specified in colons (:) behind the pipe. For example,

today | date : short results in 1/8/2019

(today is a random variable containing a date structure, date is a pipe, short is a format)

Many pipes are predefined like async, decimal, lowercase, titlecase, currency, date, i18nSelect

Additionally, you can define your own pipe.

Just define a class that implements the PipeTransform interface and decorate it with @Pipe()

@Pipe() can have the following metadata:

* name. The name of the pipe
* pure. The transform() method is invoked only when its input arguments change. Pipes are pure by default.

## Example

@Pipe({name: 'someName'})

export class SomePipe implements PipeTransform

{

transform(value: any, arg: string): number

{

Let transformedValue = value + arg;

return transformedValue;

}

}

## CLI command

ng generate pipe <pipe-name>

or

ng g -p <pipe-name>

# Service (pp 151, 237)

## Description

Services do something for you like retrieving data or do some calculations.

While your Directives, Pipes and Components should deal with presentation, services deal with data.

Services are accessed through dependency Injection. To make a service known to an angular application you have to register the service in a dependency provider. This is typically done in the providers metadata in modules or components.

To create a service you decorate a service class with the @Injectable() decorator.

@Injectable() has the following metadata

* providedIn. Where this service should be registered. Usually ‘root’ which means the root application injector so it is accessible throughout the entire application. Additionally you could specify an module name instead. In fact ‘root’ is an alias for AppModule, the root module

To register a service there are two approaches

1. via providers metadata on
   1. Modules. The service will be registered on the root injector and behaves like a singleton. It will be available everywhere in angular application
   2. Modules lazy loaded. The service will be registered on the module’s child injector which will be merged with the global injector once the module is loaded
   3. Component/Directive. The service is registered on the components injector and is recreated every time the component is loaded. It’s scope limited to the component and all it’s child components
2. From Angular 6 on Tree-shakable providers: providedIn where you specify the module where this service should be registered. Cannot be used for Components/Directives

## Example

@Injectable()

export class MyService {

doSomething() {}

}

@NgModule({

providers:[MyService]

// or

providers: [{provide:MyService, useClass: MyService}]

// provide is DI token, useClass the Actual class to be instantiated

})

export class TheModule{}

or from Angular 6 on

@Injectable({

providedIn: TheModule

})

export class MyService {

doSomething() {}

}

## CLI command

ng generate service <service-name> (ng g -s <service-name>)

# Routing (pp 181)

## Description

Routing forms the core of a SPA. Without routing no SPA.

The routing module will try to intercept requests that would normally end on the webserver. By intercepting them Routing can take alternative actions like changing the view without information from the server. This requires all the resources to be available on the client (except data)

First thing to do is defining a route table (Routes). This is a list of Route elements which have the following properties.

* path: Describes what the route looks like. Can contain parameters recognized by a colon (:) in front of them. The routing engine will try to match a navigation against this path. First complete match wins (which is different from the **best match**\*)
* component: What component to use when the path matches.
* redirectTo: The path that should be used instead.
* pathMatch: Tells the routing engine how to match. Can be
  + prefix. First part of url matches. Then it will look in the match children etc etc
  + Full. Equal to what’s left. Handy for empty paths
* children: Child Routes
* loadChildren: Use this property to enable lazy loading

The matching component will be inserted in in a placeholder **<router-outlet></router-outlet>**

\*note: complete match vs best match

Say we have the following routes:

1. {path:”:idx”, children:[{path:”b”, component:X}]}
2. {path:”a”, children:[{path:”b”, component:Y}]}

When navigating to “/a/b” the first route (X) will be taken because it’s the first **complete match** (idx=a). Even though the second one is the **best** **match**

## Example

const routes: Routes = [

{path:’a’, component:A},

{path:’b’, component:B, children:[{path:’c’, component:C}]},

{path:’a’, component: A, children:[path:’d’, component:D}]},

{path:’\*\*’, component: NoMatch}

]

@NgModule({

Imports:[RouterModule.forRoot(routes)]

})

export class Dummy()

## How the Routing Engine (RE) works

See example route above

Navigate to ‘/a’

* RE finds a match for path ‘a’ and load component A

Navigate to ‘/b’

* RE finds a match and loads component B

Navigate to ‘/a/b’

* RE finds first route. It has no children to matches ‘b’. It backtracks to root again
* RE finds third route.
  + RE checks the children. Finds no match for ‘b’ It backtracks to root.
* RE finally finds the ‘\*\*’ with means matching any route and uses component NoMatch

Navigate to ‘/a/d’

* RE matches ‘a’ with component A
* RE then tries to look for children to find a match for ‘d’ which aren’t there. It backtracks to root
* RE finds a match in the third route (with component A with children)
  + RE checks the children for ‘d’ which he finds. Component D is used.

## Capturing route parameters

Route parameters are captured via the ActivatedRoute service. This service has a property params which is a dictionary holding all route parameters for the currently activated route. The catch is that this property is an Obervable. You need to subscribe to this observable. It looks something like this.

Assume you have a route ‘/a/:t1/:t2’ which is activated as ‘/a/3/hello’ and the ActivatedRoute service is injected via a field aroutes. The variables :t1 and :t2 can be retrieved via

let vart1: number;

let vart2: string;

aroutes.params.subscribe(params=>{ vart1 = +params.t1; vart2=params.t2});

## Lazy Loading

Starts downloading a module only when the navigation matches.

Use loadChildren in Route

{ path:’x’, loadChildren:’path-to-module#ModuleName’ }

# Guards and Resolvers

Guards can be used to perform some action before or after a routing transition starts.

Examples are check if a user is authenticated before navigation or check if data needs to be saved before navigating away.

Angular offers a few interfaces for guards

* CanActivateFn
* CanActivateChildFn
* CanDeactivateFn
* CanLoad deprecated.
* ResolveFn

Register the guard in the route table

## Example Guard

export const MyGuard =( next: ActivatedRouteSnapshot, state: RouterStateSnapshot) =>

{

// Do some checks here. Return true if activation is allowed. False if activation is not allowed

return true;

}

}

And in Routes table:

{ path:’a’, component:A, canActivate:[MyGuard]}

## Example Resolver

@Injectable()

export const MyResolveGuard = (route: ActivatedRouteSnapshot, state: RouterStateSnapshot) =>

{

// retrieve the data here

return “Hello from resolver”;

}

}

And in Routes table

{ path:’a’, component:A, resolve:{ **someVar**: MyResolveGuard } }

The results can be obtained via ActivatedRoute services (aroute)

aroute.data.subscribe((vals:any)=>{

var data = vals.**someVar**;

});

# Redux

Centralized store for state. It prevents components from being littered with fields containing (shared) state or an avalanche of services containing state.

Described as “one single store for truth”.

Basically the store is one single javascript object with one or more sub-objects which can have sub-objects as well (store tree). These sub objects can be considered as sub stores.

Data in the store is **immutable**. It can only be **replaced**.

The following concepts exists in Redux

## Store

The central store where the state of the application is (Store<> from StoreModule)

## Reducer

Function which creates new state objects for the store. Associated to a specific part of the store (sub store)

function(currentState, action) { … }

## Action

A message object specifying a state change. The action is processed by the reduce. Typically looks like this:

{

type:string

payload: Object

}

## Selector

A function used to select a specific part of the store. Has one argument, the store object and return the object at a specific path

function(root) { return root.substore }

## Effect

A special field (Observable) responsible for side effects (fetching data, storing data etc). They are triggered by actions but don’t manipulate the state tree. You dispatch an action to trigger the effect (eg fetching of data).

The observable returns one or more continuation actions. Usually for updating the store

@effect() someField$ : Observable<Action> = this.action.pipe(ofType(“action type”), …..)