C:\dev\VS\ng2-fundamentals\node\_modules\spawn-default-shell\src\get-shell.js

Remove $ sign

const DETECT\_SH\_REGEX = /sh/;

Template reference variables allow you to specify a variable name that points to a component and then you can access any public properties or methods on that component using that variable.

See #thumbnail

1. Interpolation and Expression
2. Event Binding and statements
3. Repeating Data with \*ngFor
4. Removing Elements with \*ngIf and \*ngSwitch
5. Hiding Element
6. Adding Classes and Styles

**Interpolation and Expression:**

*@Component({*

*template: `*

*<h2>{{user.name}}</h2>*

*<img [src]=”user.imageUrl” />*

*<button (click)=”doSomething()”></button>*

*`*

*})*

*export class ProfileComponent {*

*User = {name: ‘John Doe’, imageUrl: ‘https://doe.com/profile.jpg’}*

*doSomething() {*

*}*

*}*

{{user.name}} is interpolation and [src] is property Binding. Both of these are used to bind data from the component to the template. Interpolation is used when you just need to display that data whereas property binding is used when you want to bind the data to the property of a DOM element. In this case, we’re binding the user’s image URL to the source property of the image tag. Interpolation and Property binding both use expression to specify the data from the component to bind to. As you can see, to use interpolation, you enclose an expression in double braces, but to bind to a property, you put the property in square brackets and the expression in quotes. Expression are interpreted by Angular and typically reference a property on the component. In both the cases, we are referring to user property on the profile component. However, expressions are not just restricted to component property binding. You can actually use a JS like expression. For example, instead of using user.name here, we can use 2+3, angular will evaluate and display 5. I could even c all a function on my component like this, but you can’t use any expression here.

{{ 2 + 2 }}, {{ getIt() }}

And there are some recommendations on the types of expression you should use.

1. Expression cannot use Assignment operators (=, +=, ++,..)
2. They cannot use **new** keyword to create new objects
3. Restricted from chaining multiple expressions together in a single expression using semicolons
4. You cant access anything on the global namespace such as console, window etc
5. Expression should have no side effects: this means that calling an expression should not change any data or the state of the application in any way
6. Expression should be fast and simple
7. Expression should be Idempotent, each time we call the expression should return the same result

**Event Binding and statements:**

*<button (click)=”doSomething()”></button>*

The event binding is binding the button’s click event to doSomething function on our profile. Notice that event bindings use parantheses around the element event to bind to versus property bindings that uses square brackets and notice that statements following event binding are put in quotes.

1. Statement allows equals but other assignment operators (+=, ++ etc) are prohibited
2. New Keyword is restricted
3. You cant access anything on the global namespace such as console, window etc
4. Chain Expression are allowed in Statements
5. Statements can have side effects and are not Idempotent
6. Should be simple for fastness of the application

**\*ngFor:**

\* represent that it is a structural directive that means it can change the structure of the DOM

**\*ngIf:**

When the expression is evaluated as false, in the UI html, the false expression is commented as a result the performance will be faster in Angular applications.

**[hidden]:**

We can bind hidden property to a Boolean value then we can hide that DOM element based on the value. Unlike in \*ngIf where the elements are commented out, in hidden angular will add hidden keyword to the css and browser will not display those elements. If we need to frequent hide the DOM elements that hidden will be more advantageous then \*ngIf from performance point of view.

**[ngSwitch]**

<div class="well hoverwell thumbnail">

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

<div>Date: {{eventinput.date}}</div>

<div [ngSwitch]="eventinput?.time">

Time: {{eventinput.time}}

<span \*ngSwitchCase="'8:00 am'"> Early Start</span>

<span \*ngSwitchCase="'10:00 am'"> Late Start</span>

<span \*ngSwitchDefault> Normal start</span>

</div>

<div>Price: ${{eventinput.price}}</div>

<div [hidden]="!eventinput?.location">

<span>Location: {{eventinput.location?.address}}</span>

<span>&nbsp;</span>

<span>{{eventinput.location?.city}} {{eventinput.location?.state}} {{eventinput.location?.zip}}</span>

</div>

<button class="padding-10 btn btn-primary" (click)="handleClickMe()">Register</button>

</div>

**ngClass:**

Class bindings are good if you are wanting to toggle a single class and the ngClass directive is better if you want to toggle multiple classes.

**Class binding:**

It looks lots like a property binding, but there is no property class.green on div, this is a special type of binding called class binding and it is parsed by Angular and its basically saying that if the condition is true then add the green class to the div.

<div [class.green]="eventinput?.time === '8.00 am'" [ngSwitch]="eventinput?.time">

Time: {{eventinput.time}}

<span \*ngSwitchCase="'8:00 am'"> Early Start</span>

<span \*ngSwitchCase="'10:00 am'"> Late Start</span>

<span \*ngSwitchDefault> Normal start</span>

</div>

**ngClass Example: ngClass will inject a Class in the html where as ngStyle will inject a CSS.**

you can return a object like this

{green: isEarlyStart, bold: isNormalStart}

*Or we can return a string with the list of class names separated by spaces or we can return a array of string which represent the classes you want to apply.*

*Return ‘green bold’*

*Return [‘green’, ‘bold’]*

<div [hidden]="!eventinput?.location" [ngClass]="{green: eventinput?.time === '8:00 am', bold: eventinput?.time === '9:00 am'}">

<span>Location: {{eventinput.location?.address}}</span>

<span>&nbsp;</span>

<span>{{eventinput.location?.city}} {{eventinput.location?.state}} {{eventinput.location?.zip}}</span>

</div>

We can call a function and return the class so that we can remove lots of logic on html page.

<div [hidden]="!eventinput?.location" [ngClass]="getStartTimeClass()">

<span>Location: {{eventinput.location?.address}}</span>

<span>&nbsp;</span>

<span>{{eventinput.location?.city}} {{eventinput.location?.state}} {{eventinput.location?.zip}}</span>

</div>

getStartTimeClass() {

const isEarlyStart = this.eventinput && (this.eventinput.time == '8:00 am')

const isNormalStart = this.eventinput && (this.eventinput.time == '9:00 am')

return {green: isEarlyStart, bold: isNormalStart}

}

*getCorrectClass(event:any) {*

*const isOnline = event && event.format=='Online'*

*const isInPerson = event && event.format=='InPerson'*

*if(isOnline) {*

*return {online: isOnline}*

*} else if (isInPerson) {*

*return 'in-person'*

*} else {*

*return ['tbd']*

*}*

*}*

**ngStyle:**

similar to class binding we can do style binding in the code

<div [style.color]="eventinput?.time === '8:00 am' ? '#003300' : '#bbb'"

Using ngStyle in html

<div [hidden]="!eventinput?.location"

**[ngStyle]**="{'color': eventinput?.time === '8:00 am' ? '#003300' : '#bbb', 'font-weight': eventinput?.time === '9:00 am' ? 'bold' : 'normal'}">

<span>Location: {{eventinput.location?.address}}</span>

<span>&nbsp;</span>

<span>{{eventinput.location?.city}} {{eventinput.location?.state}} {{eventinput.location?.zip}}</span>

</div>

Calling a function:

getStartTimeStyle():any {

if(this.eventinput && (this.eventinput.time == '8:00 am')) {

return {color: '#003300', 'font-weight': 'bold'}

} else {

return {}

}

}

**Reusable services:**

Services will have business logic and inject to front end whenever we need it.

Dependency Injection:

Services should be created as Injectables, this is required when we inject a service which also inject other services as its dependencies but it is always a good practice to add it.

import { Injectable } from '@angular/core'

@Injectable()

export class EventService {

getEvents() {

return EVENTS

}

}

We need to have the service call happen when our component first loads. Components have lifecycle hooks that you can hook into and one of those is ngOnInit method. This method is called when the component is created.

**HTTP Service: Observables and RXJS**

Promises:

1. Represent a single value in the future that may or may not come back
2. Promises are asynchronous

Observables:

1. Represent 0 or more values now or in the future
2. Either synchronous or asynchronous
3. They are referred as a stream of observable data or in essence, any value that changes over time
4. Eg: Mouse clicks, where we have x and y position. We can work on them by using map operation where you map an incoming value to a new value. This is a core activity and very easy.
5. Improved error handling
6. Can be closed independently of returning a value
7. Can deal with time
8. Advanced operations
   1. Mathmetical aggregation
   2. Buffering
   3. Debounce
   4. Distinct
   5. Filtering
   6. Combining Observables
   7. build in retry mechanism

By default angular returns a observable for HTTP call. toPromise() will take any observable and convert it to promise. We can also do it after we map the data to the response so that we can get just the data from the response that we want. This can make your handling of HTTP requests benefit from the features of observables and the simplicity of promises.

We need to subscribe if we want to HTTP call to happen.

import { Injectable, EventEmitter } from '@angular/core'

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

import { Subject, Observable } from 'rxjs/Rx';

import { IEvent } from 'app/common/model/IEvents'

@Injectable()

export class EventService {

constructor(private http: Http) {}

getEventsFromAPI():Observable<IEvent[]> {

return this.http.get("/api/events").map((response: Response) => {

return <IEvent[]>response.json();

}).catch(this.handleError)

}

getEvents():any {

return EVENTS

}

private handleError (error: Response) {

return Observable.throw(error.statusText)

}

}

**Toastr: 3rd party service wrapping**

Npm install toastr –save

And then import the JS and CSS sheet in the index file.

Then we can start using it from our program

declare let toastr:any

This means we know about this object and it is part of global scope so that Typescript will not throw an error.

**Routing and Navigation:**

1. Why Routing is necessary
2. Define Routes for pages
3. Link to Routes
4. Navigate from code
5. Route Guards
6. Preload the data using Resolve handling
7. How to load style when the corresponding route is currently active (Route based link styling)
8. Lazy Loading

Index.html is only a full page load, remaining were loaded by JS but they are not full pages, they will only load a portion of the page. To the user, it seems like new pages are being loaded, they even see the URL are changing in the browser, they even see the back and forth button are working as they were loading separate pages

**Adding the route:**

1. Define the base tag in index.html
2. Create a route file
   1. Import routes from @angular/router and put has a data type for the const. it will help to properly write the Routes parameter.
   2. Define all the component where you want to route, remove the **selector** from their definition
   3. Add <router-outlet></router-outlet> where you want to route based on url
   4. Add default route path ‘’ with full or partial match of the url

import { EventDetailsComponent } from './events/event-details/event-details.component';

import { EventListComponent } from './events/events-list.component';

import { Routes } from '@angular/router'

export const appRoutes: Routes = [

{

path: 'events',

component: EventListComponent

},

{

path: 'events/:id',

component: EventDetailsComponent

},

{

path: '',

redirectTo: '/events',

pathMatch: 'full'

}

]

**Reading value from URL:**

1. First import ActivatedRoute from routes

import { ActivatedRoute } from '@angular/router'

Then read the property as shown below, remember the + sign which typecast value to a number.

ngOnInit() {

this.event = this.eventService.getEvent(+this.route.snapshot.params['id'])

}

To link the routes: we are adding the routerLink as shown below

<div [routerLink]="['/events',eventinput.id]" class="well hoverwell thumbnail">

It will navigate to /events and add eventinput.id to the url making it navigate to event/:eventId page

**To navigate from code instead of routerLink:**

Inject Router to the component and call navigate and pass the parameters.

constructor(private route:Router) {

}

cancelForm() {

this.route.navigate(['/events'])

}

**Activated a route based on code:**

Two ways of doing it

1. By adding Route card service
   1. First add the CanActivate controller to route.ts

{

path: 'events/:id',

component: EventDetailsComponent,

canActivate: [EventRouteActivator]

},

* 1. Add EventRouteActivator as a provider in module.ts
  2. Create the eventROuteactivator controller and implement canActivate method

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

import { CanActivate, Router } from '@angular/router'

import { EventService } from '../../services/event.service';

import { ActivatedRouteSnapshot } from '@angular/router/src/router\_state';

@Injectable()

export class EventRouteActivator implements CanActivate {

constructor(private eventService: EventService,private router: Router) {

}

canActivate(route:ActivatedRouteSnapshot) {

const eventExists = !!this.eventService.getEvent(route.params['id'])

if(!eventExists) {

this.router.navigate(['/404'])

}

return eventExists

}

}

1. By function
   1. Add canDeactiveEventCreation function to createEvents controller

{

path: 'events/new',

component: CreateEventComponent,

canDeactivate: ['canDeactivateCreateEvent']

},

* 1. Add this as a provider in our module.ts and write the function

providers:[

EventService,

ToastrService,

EventRouteActivator,

{

provide: 'canDeactivateCreateEvent',

useValue: checkDirtyState

}

],

This function is written outside the AppModule controller {}

function checkDirtyState(component: CreateEventComponent) {

if(component.isDirtyProperty) {

return window.confirm('You have not saved this event, do you really want to cancel?')

}

return true;

}

**Resolve route handler:**

It will allow you to prefetch the necessary data for a component or to do other checks prior to loading a component. This was not used before in our app because data is all being provided synchronously from local variables, but this isn’t very representative of real world. To make eventService as asynchronous we are going to use Rxjs Observable.

getEvents():any {

let subject = new Subject()

setTimeout(() => {subject.next(EVENTS); subject.complete();},100)

return EVENTS

}

Basically subject is a type of observable and we are adding data to the observable (subject.next(EVENTS)) stream and we are doing it inside of setTimeout to simulate asynchrony. So after 100ms, we will add data to the stream. Now we need to update event-list component so that it will read asynchronous observable data. We do this by adding .subscribe()

@Injectable()

export class EventListResolver implements Resolve<any> {

constructor(private eventservice:EventService){}

resolve() {

return this.eventservice.getEvents().map (events => events)

}

}

Subscribe() returns subscription not observable. As we are in resolver and we want to return observable we will use map expression, these events will then get passed along to the component defined in our route.

{

path: 'events',

component: EventListComponent,

resolve: {events: EventListResolver}

},

Event list component

ngOnInit() {

//Below code is no longer needed as we are using resolver in route.ts and created a new provider, now we can directly read events data

// this.eventService.getEvents().subscribe( events => {

// this.events = events

// });

this.events = this.route.snapshot.data['events']

}

To activate a link active where the current page is loaded, use the following css and html code

<li>

<a [routerLink]="['/events']" routerLinkActive="active" [routerLinkActiveOptions]="{exact:true}">All Events</a>

</li>

li > a.active {

color: #F97924;

}

**Feature module:** (Lazy Loadable module)

1. Import common module in feature module where as in app module we will import browser module.
2. We will call Router.forChild() in feature module in app we will use Router.forRoot()

In app main route.ts we need to define the path to user module as shown below

{

path: 'user',

loadChildren:'app/user/user.module#UserModule'

}

In html, we can use

<li>

<a [routerLink]="['user/profile']">Welcome Vijay</a>

</li>

Shared Barrel:

Organizing import in index.ts files

export \* from ‘./events/..’

…

**Angular Forms:**

IEVENTS interface

getEvents():Observable<IEvents[]> {

let subject = new Subject<IEvents[]>()

setTimeout(() => {subject.next(EVENTS); subject.complete();},2000)

return subject

}

Forms:

1. Template based
   1. Allows you to build your form completely in your HTML template. Simple and easy and works great for simple use cases. Cross field validation is difficult. Unit testing is not possible
2. Model based (Reactive forms):
   1. Put the logic in the component

**Template Based Form**

We are going to use few forms directives and all of forms directives are in their own modules, so we need to import the **FormsModule** and since this form is in our user module we will import there.

When the use enters data into those fields, we need to get the data into our component so that we can validate the inputs. We can use databinding. When input value is entered it will update userName field on our component and it is cumbersome so we can simple use ngModel provided by Angular

<input (input)="userName=$event.target.value" id="userName" type="text" class="form-control" placeholder="User Name..." />

Using ngModel

<input [(ngModel)]="userName" id="userName" type="text" class="form-control" placeholder="User Name..." />

**Here [()] represents banana in a box syntax which is two way binding. If we remember the syntax module, parenthesis are used to bind in the HTML to component direction and its typically used for responding to events, whereas the square brackets are used to bind in the component to html direction and it is typically used for displaying data from the component on the page.**

**ngModel requires us to add the name attribute in the model so we will add it.**

<form #loginForm="ngForm" (ngSubmit)="login(loginForm.value)" autocomplete="off">

We are binding the userName and password on the forms, so to access that create a id for #loginForm and on submit the form call the method login on controller. We will pass forms values to the method as inputs

login(loginForm:any) {

console.log(loginForm)

}

Console output

{userName: "vijay", password: "testing"}