https://indepth.dev/

Angular JS vs Angular > 2 - the first version of Angular without TypeScript vs Angular

----

Typescript

Typescript is a superset of JS, uses the .ts file extension:

```
let isDone: boolean = false;
let hex: number = 0xf00d;
let binary: number = 0b1010;
let octal: number = 0o744;

let message: string = ''
let messages: string[] = ['Test 1','Test 2']
let messages: ['Test 1', number] = [Test 1', 2.3]
```

https://www.typescriptlang.org/docs/handbook/intro.html

tsc index.ts –target es6 - convert Typescript to JS based on a ECMA standard, транспилация

!!! you can dynamically change the type of a variable, but you can assign the value of a variable from different type to the variable, for which you want its type to be changed !!!

Namespace in Angular:

```
namespace Vehicle {
  export enum Car {
    Peugeout = 'Message 1',
    Citroen = 1
  }

  export enum Truck {
    Ford = 'Ford',
    RangeRover = 2
  }

}

let carMessages: [number, Vehicle.Car] = [1, Vehicle.Car.Citroen]
```

constructor property promotion is available in Typescript as well:

```
class Suv {

  public model: string;

  constructor(model: string) {
    this.model = model;
  }

}
```

= >

```
class Suv {

  constructor(public model: string) {

  }

}
```

method overriding

```
class Suv {

  constructor(protected model: string) {

  }

  drive():string {
    return `I am driving this ${this.model}`;
  }

}

export class SmallerSuv extends Suv {

  constructor(model: string, private size: number) {
    super(model);
  }
  override drive(): string {
    return super.drive() + ' called from child!'
  }

  park(): void {
    console.log(`I am parking this ${this.model} in size ${this.size}`)
  }

}
```

we can set a custom type

```
// creating a custom type
function printLabel(labelledObj: {label: string}){
  console.log(labelledObj.label);
}
```

Generics - primary used for data structures

```
function identity<T>(arg: T): T {
  return arg;
}

let output = identity<string>('myString');
```

!isNan(+arg) - check if arg is number

!!! interface are possible in Typescript with the interface keyword and we can create objects of set interface as well as classes, however, we can not check if an object is instance of interface and interfaces are converted to basic JavaScript classes !!!

!!! native Javascript feature is # to define private or static properties and it is different from the Typescript:

https://stackoverflow.com/questions/59641564/what-are-the-differences-between-the-private-keyword-and-private-fields-in-types

The private keyword in TypeScript is a compile time annotation. It tells the compiler that a property should only be accessible inside that class

vs

Private fields ensure that properties are kept private at runtime: !!!

Install Angular:

```
npm install -g @angular/cli
ng new some-app
cd some-app
//starting the server:
ng serve
ng build
```

Start Angular project:

```
[15:18:32] donetianpetkov@Donetian-Petkov-NEWs-MacBook-Pro [
~/WebstormProjects/angular_first/soft-uni-day-one ] $ ng serve

✔ Browser application bundle generation complete.

Initial Chunk Files    | Names         |  Raw Size
vendor.js              | vendor        |   1.70 MB |
polyfills.js           | polyfills     | 294.81 kB |
styles.css, styles.js  | styles        | 173.24 kB |
main.js                | main          |  47.69 kB |
runtime.js             | runtime       |   6.53 kB |

                       | Initial Total |   2.21 MB

Build at: 2022-04-03T12:18:54.282Z - Hash: 19f21a7f6345c537 - Time:
11344ms

** Angular Live Development Server is listening on localhost:4200, open
your browser on http://localhost:4200/ **


✔ Compiled successfully.
```

Build:

```
[15:01:27] donetianpetkov@Donetian-Petkov-NEWs-MacBook-Pro [
~/WebstormProjects/angular_first/soft-uni-day-one ] $ ng build

✔ Browser application bundle generation complete.
✔ Copying assets complete.
✔ Index html generation complete.

Initial Chunk Files            | Names         |  Raw Size | Estimated
Transfer Size
main.20a8ef08287cd70f.js       | main          | 116.45 kB |
34.88 kB
polyfills.bd73d25360343036.js | polyfills     |  33.03 kB |
10.64 kB
runtime.20c563868d71cc52.js    | runtime       |   1.06 kB |
606 bytes
styles.ef46db3751d8e999.css    | styles        |   0 bytes |
-

                               | Initial Total | 150.54 kB |
46.12 kB

Build at: 2022-04-03T12:24:06.435Z - Hash: 06132fb6ef214ebb - Time:
17633ms

[15:25:47] donetianpetkov@Donetian-Petkov-NEWs-MacBook-Pro [
~/WebstormProjects/angular_first/soft-uni-day-one ] $ cd dist/
[15:25:55] donetianpetkov@Donetian-Petkov-NEWs-MacBook-Pro [
~/WebstormProjects/angular_first/soft-uni-day-one/dist ] $ ls
soft-uni-day-one
[15:25:55] donetianpetkov@Donetian-Petkov-NEWs-MacBook-Pro [
~/WebstormProjects/angular_first/soft-uni-day-one/dist ] $ cd
soft-uni-day-one/
[15:25:58] donetianpetkov@Donetian-Petkov-NEWs-MacBook-Pro [
~/WebstormProjects/angular_first/soft-uni-day-one/dist/soft-uni-day-one
] $ ls
3rdpartylicenses.txt          index.html
polyfills.bd73d25360343036.js styles.ef46db3751d8e999.css
favicon.ico             main.20a8ef08287cd70f.js
runtime.20c563868d71cc52.js
```

config file: angular.json - how the project will be built https://stackblitz.com/ - online IDE for JS, Angular, React and etc

https://plnkr.co/ - online IDE for JS, Angular, React and etc

tsconfig.json - how the Typescript and the project will be compiled

Component in Angular:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'soft-uni-day-one';
}
```

Manually Creating Components in Angular:

----

1. Create component ts in the src/app folder -> define component inside it:

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

@Component({
  selector: 'app-test',
  template: `<div>Test</div>`,
  styles: [`div {color:blue}`]
})
export class TestComponent {
  title = 'testing';
}
```

2. then include the component in the app.module.ts:

```
import {TestComponent} from "./test.component";

@NgModule({
  declarations: [
    AppComponent,
    TestComponent
```

3. and finally set the selector in the app.component.html or the template file where you want to use the component:

```
<app-test></app-test>
```

Automatically Creating Components:

```
[19:21:59] donetianpetkov@Donetian-Petkov-NEWs-MacBook-Pro [
~/WebstormProjects/angular_first/soft-uni-day-one ] $ ng generate
component example

CREATE src/app/example/example.component.css (0 bytes)
CREATE src/app/example/example.component.html (22 bytes)
CREATE src/app/example/example.component.spec.ts (633 bytes)
CREATE src/app/example/example.component.ts (279 bytes)
UPDATE src/app/app.module.ts (467 bytes)
```

ng g c example - shorter syntax

Angular Specifics

!!! the ng g c finds the closest module to the newly created component and adds the component in the Declaration !!!

!!! Every Angular project has a module file (App.Module) where the Angular app is launched:

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

import { AppComponent } from './app.component';
import {TestComponent} from "./test.component";
import { ExampleComponent } from './example/example.component';

@NgModule({
  declarations: [
    AppComponent,
    TestComponent,
    ExampleComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

in the bootstrap: we set which components will be launched when the app is executed. We need to also specify in the index.html the selector for these components:

app.component.ts:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'soft-uni-day-one';
}
```

->

index.html:

```
<body>
  <app-root></app-root>
</body>
```

!!!!

interpolation is only for showing text in HTML - {{}}

if we need to use a value from the class we use square brackets - in other words we bind:

in the template / html file for the component:

`<input [value]="title">` - title will be the title property in the class

---

events in Angular:

---

in the template / html file for the component:

`<button (click)="nameOfFunction()">Click Me</button>` - when we click the button the nameOfFunction will be executed

in the TS file for the component:

```
export class AppComponent {

nameOfFunction(): returnType {

// some functionality
   }
}
```

Initial Introduction to Directives in Angular

----

special commands for working within the HTML template - *ngFor, *ngIf

in the component ts file:

```
users = [
    {
      name: 'Ivan',
      age: 20
    },
    {
      name: 'Naiden',
      age: 30
    },
    {
      name: 'Ivana',
      age: 50
    }
  ]
```

in the template / html file:

```
<ul>
  <li *ngFor="let user of users">
    {{user.name}}
  </li>
</ul>
```

in the template / html file:

```
<div *ngIf="showText">VISIBLE</div>

<button (click)="toggleText()">{{showText ? 'HIDE TEXT' : 'SHOW
TEXT'}}</button>
```

in the component ts file:

```
showText = true;

toggleText(): void {
    this.showText = !this.showText;
  }
```

---

the event in the callback functions is $event

```
<button (click)="toggleText($event)">{{showText ? 'HIDE TEXT' : 'SHOW
TEXT'}}</button>
```

->

```
toggleText(event: MouseEvent): void {
    event.preventDefault();
    this.showText = !this.showText;
  }
```

---

binding / adding CSS classes to elements:

---

in html:

```
<div *ngIf="showText" [class]="classes">VISIBLE</div>
```

in component:

```
export class ExampleComponent implements OnInit {

  classes=['test', 'test-1'];

in component css:

.test {
  color:red;
}
```

---

conditional class - > `[class.special]="isSpecial"`

---

if isSpecial in the class is true, the element will have CSS class special

---

adding styles inline:

```
[style.color]="isSpecial ? 'red' : 'blue'"
```

Templates variables

---

calling HTML elements in other elements - this is how we can take the values of input fields by clicking a button without a form:

```
<input #inputElement type="text">

<button (click)="changeTitleHandler(inputElement.value)">Change
Title</button>

<p>{{title}}</p>
```

nullsafe operators

```
{{game?.title}}
```

!!!! not recommended to use two-way data binding - [(ngModel)] !!!

Lifecycle Hooks

---

ngOnInit - after constructor before content
ngAfterViewInIt
ngOnDestroy
ngOnChanges - when Input changes, we can call ngOnChanges(simpleChanges: SimpleChanges) { console.log(simpleChanges} to see the change, need for the reference to be changed
ngDoCheck

Презентационни компоненти vs Container компоненти - display data vs process data

!!! we can set a property to a class without initializing it with the ! after the prop - user!: IUser

Another alternative: user: IUser | undefined; !!!

Passing input to child components:

<app-user-list-item>:

ts:

```
@Input() user!: IUser;
```

html:

```
<span>{{user?.name}}</span>
<span>{{user?.age}}</span>
```

->

<app-user-list>:

ts:

```
@Input() userArray: {name: string, age: number}[] = [];
```

html:

```
<div id="container">

  <app-user-list-item *ngFor="let users of userArray"
[user]="users"></app-user-list-item>
</div>
```

->

<app-root>:

html:

```
<app-user-list [userArray]="users"></app-user-list>
```

ts:

```
users = [
    {
      name: 'Ivan',
      age: 20
    },
    {
      name: 'Naiden',
      age: 30
    },
    {
      name: 'Ivana',
      age: 50
    }
  ]
```

Passing Output to Parent Components:

---

same example:

<app-user-list>:

ts:

```
@Output() addUser = new EventEmitter<IUser>();

addNewUser(userNameInput: HTMLInputElement, userAgeInput:
HTMLInputElement): void {

    const {value: name} = userNameInput;
    const {valueAsNumber: age} = userAgeInput;

    this.addUser.emit({name, age}); // triggering the addUser and
emitting the name and age to the parent component - addUser is an event
and in the parent component we have addNewUserHandler, which handles
this event

    userAgeInput.value = '';
    userNameInput.value = '';

  }
```

html:

```
<div>
    <input #userNameInput type="text">
    <input #userAgeInput type="number">
    <button (click)="addNewUser(userNameInput, userAgeInput)">Add New
User</button>
  </div>
```

<app-root>:

ts:

```
addNewUserHandler(newUser: IUser){
    this.users.push(newUser);
  }
```

html:

```
 <app-user-list [userArray]="users"
(addUser)="addNewUserHandler($event)"></app-user-list>
```

Debugging in Angular - setting a debugger after the line we want to be executed:

```
constructor() {
    console.log("I am in the constructor")
  debugger;

  }
```

Reloading the page it will spawn the debugger on the browser

Other Angular Specifics

----

zone.js - gathers all of the async functions from the browser and monitors them - provides a mechanism, called zones, for encapsulating and intercepting asynchronous activities in the browser (e.g. setTimeout, , promises); set in the polyfills.tsl ; uses monkey patching

changeDetection - set in the @Component - changes between Default rendering and OnPush rendering (disables rendering on reference changes) ->

```
  changeDetection: ChangeDetectionStrategy.OnPush
```

SOLID principles -

S - Single-responsiblity Principle: A class should have one and only one reason to change, meaning that a class should have only one job.

O - Open-closed Principle: Objects or entities should be open for extension but closed for modification.

L - Liskov Substitution Principle: every subclass or derived class should be substitutable for their base or parent class.

I - Interface Segregation Principle: A client should never be forced to implement an interface that it doesn’t use, or clients shouldn’t be forced to depend on methods they do not use.

D - Dependency Inversion Principle: Entities must depend on abstractions, not on concretions. It states that the high-level module must not depend on the low-level module, but they should depend on abstractions.

https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design

----

Dependency Injection

Dependency Injection - implementation of the Dependency Inversion; ensure that instead of initializing manually dependencies for a class, instead we tell the class which are its dependencies and Angular initiates them in the constructor.

Instead of:

```
public class Laptop {

  public battery: Battery;

  public videoCard: VideoCard;

  constructor() {

    this.battery = new Battery('Acer battery');

    this.videoCard = new VideoCard('Nvidia 960 GTX');

  }

}
```

we can use

```
public class Laptop {

constructor(public battery: Battery, public videoCard: VideoCard) {}
}
```

!!! We need to include the Battery and the VideoCard in the providers array in the module TS file like:

```
providers: [{
    provide: Battery,
    useClass: Battery
  }]
```
!!!

useValue - we can set a custom string, object, function and etc

----

Services

!!! **Services:** Components should not fetch or save data directly - this is why we use services. Components should focus on presenting data and delegate data access to a service. Services are a great way to share information among classes that don't know about each other and avoid code duplication !!!

!!! we can also include providers in the Component's declaration and in the main.ts !!!

@Inject - we can inject primitive types and etc in the dependency injector through an InjectionToken:

in the module:

```
export const myStringInjectionToken = new InjectionToken('myString');

{
      provide: myStringInjectionToken,
      useValue: 'this is the value'
    }
```

in the class where we want to use the injectiontoken:

```
constructor(@Inject(myStringInjectionToken) myString: string) {

    console.log(myString);

  }
```

@Injectable - we can use this declaration in the class, which we want to provide in the app, without including it in the providers array in the module:

```
@Injectable({
  providedIn: 'root'
})
export class UserService
```

!!! the recommended way to declare dependencies it is to add them to the providers array - we may use the following shorter syntax:

instead of

```
providers: [{
   provide: Battery,
   useClass: Battery
 }]
```

we can use:

`providers: [Battery]` !!!

Observables

Observables - similar to promises, which can be sync or async, an observable represents a stream, or source of data that can arrive over time. You can create an observable from nearly anything, but the most common use case in RxJS is from events. This can be anything from mouse moves, button clicks, input into a text field, or even route changes; can be defined and used with a dollar sign at the end of a stream variable - streamOne$

```
import { of } from 'rxjs';
import { map } from 'rxjs/operators'

of(1000).pipe(map(x => x + 100)).subscribe((x) => {
  console.log(x);
}); // the same thing as:

//Promise.resolve(1000).then(x => x+100).then(console.log);
```

Cold vs Hot Observables - independent subscriptions vs shared subscription

tap vs map - map returns a value to the stream, tap does not and thus, it can be used for a side effect - such as void functions

!!! fake REST api for generating API streams - https://jsonplaceholder.typicode.com/ !!!

!!! imdb API - https://www.omdbapi.com/?i=tt3896198&apikey=3dbf0674 !!!

!!! turn JSON output to object manually - search-replace of the double quotes (ctrl + R):

"(.*)":

to

$1:

!!!

make a HTTP request - >

app.module.ts:

```
import { HttpClientModule} from "@angular/common/http";

imports: [
    HttpClientModule
  ],
```

the component where the HTTP request is made:

```
import {HttpClient, HttpClientModule} from "@angular/common/http";

    loadUsers(search: string = '') {

    const query  = search ? `?email_like=${search}` : '';

    return
this.http.get<IUser[]>(`https://jsonplaceholder.typicode.com/users${quer
y}`)

  }
```

in the component where the data from the HTTP request is processed:

```
this.userService.loadUsers().subscribe(users => this.users = users);
```

Modules

Modules - NgModules are containers for a cohesive block of code dedicated to an application domain, a workflow, or a closely related set of capabilities They can contain components, service providers, and other code files whose scope is defined by the containing NgModule. They can import functionality that is exported from other NgModules, and export selected functionality for use by other NgModules.

ng g m core - generating a module

we must include the module in the root / app module:

```
imports: [
    BrowserModule,
    HttpClientModule,
    CoreModule,
    SharedModule,
    UserModule
  ]
```

!!! If we want to use components, services, etc from one module in another module we must include the components in the Exports array in the first module and include only the module in the Imports array in the second module !!!

Routing:

Example:

1) app-routing.module.ts:

```
import { Routes, RouterModule } from '@angular/router';
import { AboutComponent } from './about/about.component';
import { UserListComponent } from
'./user/user-list/user-list.component';

const routes: Routes = [
  {
    path: '',
    pathMatch: 'full',
    redirectTo: '/user-list'
  },
  {
    path: 'user-list',
    component: UserListComponent
  },
  {
    path: 'about',
    component: AboutComponent
  }
]

export const AppRoutingModule = RouterModule.forRoot(routes);
```

2) app.module.ts:

```
 imports: [
AppRoutingModule
```

3) app.component.html:

```
<div id="content">

  <router-outlet></router-outlet>

</div>
```

4) header.component.html:

```
<nav>
    <a routerLink="/user-list">User List</a>
    <a routerLink="/about">About Us</a>
  </nav>
```

5) module, where the header component has been set:

```
imports: [
RouterModule
```

1) We first define the routes in a custom Routing module
2) We include the Routing module in the root / app module
3) We set router-outlet in the component where the content of the routes will be displayed / rendered
4) In the header component we set a tags with routerLink to build navigation buttons to the routes
5) we import the default RouterModule in the module where the header component has been set.

Not Found Page:

----

1) ng g c not-found

2) app-routing.module.ts:

```
{
    path: '**',
    component: NotFoundComponent
  }
```

making a tag active:

html:

```
<a routerLink="/user-list" routerLinkActive="active">User List</a>
 <a routerLink="/about" routerLinkActive="active">About Us</a>
```

css file:

```
a.active {
  color:red;
}
```

!!!! If we have components with custom routing modules, we should set the root routing module at the end of the imports array in the root / app module:

```
 imports: [
    BrowserModule,
    CoreModule,
    SharedModule,
    UserModule,
    FontAwesomeModule,
    ThemeModule,
    AppRoutingModule,
```

SwitchMap

SwitchMap - The main difference between switchMap and other flattening operators is the cancelling effect. On each emission the previous inner observable (the result of the function you supplied) is cancelled and the new observable is subscribed. You can remember this by the phrase switch to a new observable.

!!! **mergeMap** vs **switchMap** - switchMap cancels previous HTTP requests that are still in progress, while mergeMap lets all of them finish !!!

switchMap is one of the most useful RxJS operators because it can compose Observables from an initial value that is unknown or that change. Conceptually, it is similar to chaining then functions with Promises, but operates on streams (Promises resolve once). Example: You have an Observable of a userID, then use it to *switch* to an HTTP request of all posts owned by that user.

The switchMap operator returns an observable that emits items based on applying a function that you supply to each item emitted by the source observable where that function returns an inner observable. Each time it observes one of these inner observables, the output observable begins emitting the items emitted by that inner observable.

!!!! Does not return an observable, but subscribes to itself and returns the items; thus, can be used to combine two HTTP requests !!!!

```
this.activatedRoute.params.pipe(
      switchMap(({id}) => this.userService.loadUser(id))
    ).subscribe(user => {
      this.user = user;
    })
```

Another variant with additional action in pipe:

```
this.activatedRoute.params.pipe(
      tap(() => this.user = undefined),
      switchMap(({id}) => this.userService.loadUser(id))
    ).subscribe(user => {
      this.user = user;
    })
```

!!! switchMap will cancel any previous requests upon new request !!!

--

subRouting

--

/test
/test/one
/test/two

->

```
const routes: Routes = [
  {
    path: 'test',
    component: MainComponent,
    children: [
      {
        path: '',
        pathMatch: 'full',
        redirectTo: '/test/one'
      },
      {
        path: 'one',
        component: OneComponent
      },
      {
        path: 'two',
        component: TwoComponent
      }
    ]
  }
]
```

!!! Ensure that a module is only initialized once (through Singleton):

```
constructor(@Optional() @SkipSelf() coreModule: CoreModule) {

    if (coreModule) {
      throw new Error('Core module should be imported once!');
    }
```

```
  }
```

Guards

Guards - Angular route guards are interfaces provided by angular which when implemented allow us to control the accessibility of a route based on condition provided in class implementation of that interface.

canActivate - Interface that a class can implement to be a guard deciding if a route can be activated. If all guards return true, navigation continues. If any guard returns false, navigation is cancelled. If any guard returns a UrlTree, the current navigation is cancelled and a new navigation begins to the UrlTree returned from the guard.

Example: checking if id has been provided to /user-detail/ URI

1) Routing Module:

```
{
    path: 'user-detail/:id',
    component: UserDetailComponent,
    canActivate: [ParamsActivate],
    data: {
      paramsActivate: ['id'],
      paramsActivateRedirectURL: '/user-list'
    }
  }
```

2) Component Module:

```
providers: [
    ParamsActivate
  ]
```

3) params.activate.ts

```
import { Injectable } from "@angular/core";
import { ActivatedRouteSnapshot, CanActivate, Router,
RouterStateSnapshot, UrlTree} from "@angular/router";
import {Observable} from "rxjs";

@Injectable()

export class ParamsActivate implements CanActivate {

  constructor(private router: Router) {

  }

  canActivate(route: ActivatedRouteSnapshot, state:
RouterStateSnapshot): boolean | UrlTree | Observable<boolean | UrlTree>
| Promise<boolean | UrlTree> {

    const {data: {paramsActivate, paramsActivateRedirectUrl }} = route;

    if (!paramsActivate || !Array.isArray(paramsActivate) ||
paramsActivate.length === 0) {
      return true;
    }

    const hasAllRouteParams = paramsActivate.reduce((acc, curr) => {
      return !!route.params[curr] && acc
    }, true);

    if (hasAllRouteParams) {
      return true;
    }

    return this.router.parseUrl(paramsActivateRedirectUrl || '/');

  }
}
```

https://codeburst.io/understanding-angular-guards-347b452e1892

FontAwesome

!!! FontAwesome for Angular:

https://www.npmjs.com/package/@fortawesome/angular-fontawesome

CSS class: ng-fa-icon

!!!

Additional Information

----

turn element to a boolean - !!this.user, !!number, etc

Check what platform you are on (browser, etc) - > PLATFORM_ID:

https://angular.io/api/core/PLATFORM_ID

https://github.com/angular/angular/blob/main/packages/common/src/platform_id.ts

at module.ts - >

```
export class CoreModule {

  constructor(@Inject(PLATFORM_ID) platformId: any) {
    console.log(isPlatformBrowser(platformId));
  }

}
```

Custom Injector which takes into consideration on which platform you are:

```
providers:[
  {
    provide: LocalStorage,
    useFactory: (platformId: Object) => {

      if(isPlatformBrowser(platformId)){
        return window.localStorage;
      }

      if (isPlatformServer(platformId)) {

        return class implements Storage {
          length = 0;
          private data: Record <string, string> = {};

          clear(): void {
            this.data= {};
          }

          getItem(key: string): string | null {
            return this.data[key];
          }

          key(index: number): string | null {
            return null;
          }

          removeItem(key: string): void {
            const { [key]: removedItem, ...others } = this.data;

            this.data = others;
          }

          setItem(key: string, value: string): void {

            this.data[key] = value;
          }
        }
      }
      throw Error('NOT IMPLEMENTED!');
    },
    deps: [PLATFORM_ID]
  }
]
```

Protect before login and Redirect after being logged-in:

----

1) auth.activate.ts:

```
import {ActivatedRouteSnapshot, CanActivate, Router,
RouterStateSnapshot, UrlTree} from "@angular/router";
import {Injectable} from "@angular/core";
import {Observable} from "rxjs";
import {UserService} from "../../user/user.service";

@Injectable()
export class AuthActivate implements CanActivate {

  constructor(private router: Router, private userService: UserService)
{
  }

  canActivate(route: ActivatedRouteSnapshot, state:
RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<boolean |
UrlTree> | boolean | UrlTree {
    const { authenticationRequired, authenticationFailureRedirectUrl } =
route.data;

    if (typeof authenticationRequired === 'boolean' &&
      authenticationRequired === this.userService.isLogged) {
      return true;
    }

    let authRedirectUrl = authenticationFailureRedirectUrl;
    if (authenticationRequired) {
      const loginRedirectUrl = route.url.reduce((acc, s) =>
`${acc}/${s.path}`, '');
      authRedirectUrl += `?redirectUrl=${loginRedirectUrl}`;
    }

   return this.router.parseUrl(authRedirectUrl || '/')

  }

}
```

2) user.service.ts:

```
import {Inject, Injectable} from '@angular/core';
import {IUser} from "../shared/interfaces";
import { LocalStorage} from '../core/injection-tokens'

@Injectable()
export class UserService {

  user: IUser | undefined;

  get isLogged(): boolean {
    return !!this.user;
  }

  constructor(@Inject(LocalStorage) private localStorage:
Window['localStorage']) {

    try {
      const localStorageUser = this.localStorage.getItem('<USER>') ||
'ERROR';
      this.user = JSON.parse(localStorageUser);
    } catch {
      this.user = undefined;
    }

  }

  login(email: string, password: string): void {
    this.user = {
      email,
      firstName: 'John',
      lastName: 'Doe'
    };
    this.localStorage.setItem('<User>', JSON.stringify(this.user));
  }

  logout(): void {
    this.user = undefined;
  }

}
```

3) login.component.ts:

```
constructor(private userService: UserService,
              private router: Router,
              private activatedRoute: ActivatedRoute) { }
```

```
 login(email: string, password: string): void {
    this.userService.login(email, password);
    const redirectUrl =
this.activatedRoute.snapshot.queryParams['redirectUrl'] || '/';
    this.router.navigate([redirectUrl]);
  }
```

4) path which can not be visited without login:

```
{
    path: 'add-theme',
    component: NewThemeComponent,
    canActivate: [AuthActivate],
    data: {
      authenticationRequired: true,
      authenticationFailureRedirectUrl: '/login'
    }
}
```

Directives

Directives - AngularJS lets you extend HTML with new attributes called Directives ; At a high level, directives are markers on a DOM element (such as an attribute, element name, comment or CSS class) that tell AngularJS's HTML compiler ($compile) to attach a specified behavior to that DOM element (e.g. via event listeners), or even to transform the DOM element and its children;

```
ng g d name_of_directive
```

!!! Components are directives with template !!!

Attribute directives - change the appearance or behavior of an element, component or another directive - ngStyle or ngClass

Structural directives - change the DOM layout by adding or removing DOM elements - *ngIf or *ngFor; The asterisk, * , syntax on a structural directive, such as *ngIf , is shorthand that Angular interprets into a longer form. Angular transforms the asterisk in front of a structural directive into an <ng-template> that surrounds the host element and its descendants.

!!! **Attribute vs Structural Directives** - only affect / change the element on which they are added to vs affect a whole area in the DOM !!!

Decorator - An Angular Decorator is a function, using which we attach metadata to a class, method, accessor, property, or parameter. We apply the decorator using the form @expression , where expression is the name of the decorator. The Decorators are Typescript features and still not part of the Javascript.

ngIfElse -

```
<button (click)="toggleActive()">Toggle Active</button>
```

1) Basic:

```
<ng-template #isActiveTemplate>
  IS ACTIVE - TRUE
</ng-template>

<ng-template #isNotActiveTemplate>
  IS ACTIVE - FALSE
</ng-template>

<ng-container *ngTemplateOutlet="isActive ? isActiveTemplate :
isNotActiveTemplate"></ng-container>
```

or

2) ngIfElse:

```
<ng-template #isNotActiveTemplate>
  IS ACTIVE - FALSE
</ng-template>

<ng-container *ngIf="isActive; else isNotActiveTemplate">
  IS ACTIVE - TRUE
</ng-container>
```

Example Attribute Directive -

1) ng g d highlight

2) highlight.directive.ts:

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

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective implements OnChanges {

  @Input() appHighlight!: boolean;

  constructor(private elementRef: ElementRef) {
  }

  ngOnChanges(changes: SimpleChanges): void {
        if (changes.appHighlight.currentValue) {
          this.elementRef.nativeElement.style.color = 'red';
        } else {
          this.elementRef.nativeElement.style.color = 'black';
        }
    }

}
```

3) app.component.ts:

```
 isActive = false;

toggleActive() {
    this.isActive = !this.isActive;
  }
```

4) app.component.html:

```
<button (click)="toggleActive()" [appHighlight]="isActive">Toggle
Active</button>
```

Another alternative of highlight.directive.ts:

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

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective { //implements OnChanges {

  @Input() set appHighlight(isActive:boolean) {
    if (isActive) {
      this.elementRef.nativeElement.style.color = 'red';
    } else {
      this.elementRef.nativeElement.style.color = 'black';
    }
  }

  constructor(private elementRef: ElementRef) {
  }
}
```

Other other alternative of highlight.directive.ts:

```
import { Directive, ElementRef, HostBinding, Input, OnChanges,
SimpleChanges} from '@angular/core';

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {

  @Input() set appHighlight(isActive:boolean) {
    if (isActive) {
      this.color = 'red';
    } else {
      this.color = 'black';
    }
  }

  @HostBinding('style.color') color = 'black';
}
```

Example Structure Directive

1) ng g d my-if

2) my-if.directive.ts:

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

@Directive({
  selector: '[appMyIf]'
})
export class MyIfDirective {

  @Input() set appMyIf(isVisible: boolean) {
    if (isVisible) {
      this.viewContainerRef.createEmbeddedView(this.templateRef, {data:
123, $implicit: 'This is a test'});
    } else {
      this.viewContainerRef.clear();
    }
  }

  constructor(private viewContainerRef: ViewContainerRef,
              private templateRef: TemplateRef<any>) { }

}
```

3) app.component.html:

```
<ng-template [appMyIf]="isActive" let-data="data" let-test>
  <ng-container>
    {{data}}
    {{test}}
  </ng-container>
</ng-template>
```

OR

```
<ng-container *appMyIf="isActive; let data=data; let test2">
  {{data}}
  {{test2}}
</ng-container>
```

Result is the same: `123 This is a test`

Forms

Template Driven Forms - In Template Driven Forms we specify behaviors/validations using directives and attributes in our template and let it work behind the scenes. All things happen in Templates hence very little code is required in the component class.

!!! Every button in a form is by default considered a Submit button !!!

Example Template Driven Login Form, which has an email address, password and repeatPassword input fields and various validations:

1) ng g c login

2) ng g d same-value

3) login.component.html:

```
<form #form="ngForm" (ngSubmit)="loginHandler(form)">
  <div class="form-group">
    <label for="email" #highlight="highlight"
[appHighlight]="true">Email: </label>
    <input type="text" name="email" id="email" #emailInput="ngModel"
[ngModel]="tesingEmail" required email>
    <ul *ngIf="emailInput.invalid && emailInput.touched" class="errors">
      <li *ngIf="emailInput.errors?.required">Email is Required</li>
      <li *ngIf="emailInput.errors?.email">Email is Invalid</li>
    </ul>
  </div>

  <div class="form-group">
    <label for="password">Password: </label>
    <input type="password" name="password" id="password"
#passInput="ngModel" ngModel required [minlength]="6">
    <ul *ngIf="passInput.invalid && passInput.touched" class="errors">
      <li *ngIf="passInput.errors?.required">Password is Required</li>
      <li *ngIf="passInput.errors?.minlength">Password should be minimum
6 characters</li>
    </ul>
  </div>
  <div class="form-group">
    <label for="repeat-password">Repeat Password: </label>
    <input type="password" name="repeat-password" id="repeat-password"
#repeatPassInput="ngModel" appSameValue="password" ngModel required
[minlength]="6">
    <ul *ngIf="repeatPassInput.invalid && repeatPassInput.touched"
class="errors">
      <li *ngIf="repeatPassInput.errors?.required">Password is
Required</li>
      <li *ngIf="repeatPassInput.errors?.minlength">Password should be
minimum 6 characters</li>
      <li *ngIf="repeatPassInput.errors?.sameValue">Password and
repeatPassword should be the same</li>

    </ul>
  </div>
  <button>Login</button>
</form>
```

4) same-value.directive.ts:

```
import { Directive, Input, OnChanges, OnDestroy, SimpleChanges} from
'@angular/core';
import {AbstractControl, NgControl, NgForm, NgModel, NG_VALIDATORS,
ValidationErrors, Validator} from '@angular/forms';
import { Subscription } from 'rxjs';

@Directive({
  selector: '[appSameValue]',
  providers: [
    {
      provide: NG_VALIDATORS,
      useExisting: SameValueDirective,
      multi: true
    }
  ]
})
export class SameValueDirective implements Validator, OnDestroy {

  @Input() appSameValue = '';
  @Input() name!: string;
  otherControl!: AbstractControl;
  subscription!: Subscription;
  constructor(private form: NgForm) {
  }

  ngOnDestroy(): void {
    this.subscription.unsubscribe();
    }

  validate(control: AbstractControl): ValidationErrors | null {

    const otherControl = this.form.controls[this.appSameValue];

    if (this.subscription) {
      this.subscription.unsubscribe();
    }

    this.subscription = otherControl.valueChanges!.subscribe(() => {

      control.updateValueAndValidity( { onlySelf: true});

    });

    return control.value !== otherControl?.value ? {
        sameValue: {
          [this.appSameValue]: otherControl?.value,
          [this.name]: control.value
        }
      }
     : null

    }

}
```

5) login.component.ts:

```
import { Component, OnInit } from '@angular/core';
import { NgForm } from '@angular/forms';

@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.css']
})
```

```
export class LoginComponent implements OnInit {

  tesingEmail = 'test@abv.bg'

  constructor() { }

  ngOnInit(): void {
  }

  loginHandler(form: NgForm): void {

    if (form.invalid) {
      return;
    }

    console.log(form);

  }

}
```

!!! **View in Angular** - the HTML template of the component after it has been initialized / instanced

ViewChild() - Property decorator that configures a view query. The change detector looks for the first element or the directive matching the selector in the view DOM. If the view DOM changes, and a new child matches the selector, the property is updated. ; can be used as an alternative method to get to the form:

```
@ViewChild('form', {read: NgForm}) form!: NgForm;
```

!!!

!!! **Bootstrap in Angular**:

https://ng-bootstrap.github.io/#/home

https://ng-bootstrap.github.io/#/getting-started

https://ng-bootstrap.github.io/#/components/accordion/examples https://www.positronx.io/angular-form-validation-with-template-driven-using-bootstrap/

1)

```
[17:36:32] donetianpetkov@Donetian-Petkov-NEWs-MacBook-Pro [
~/WebstormProjects/angular_first/lessonTwo ] $  ng add
@ng-bootstrap/ng-bootstrap
ℹ Using package manager: npm
✔ Found compatible package version: @ng-bootstrap/ng-bootstrap@12.1.1.
✔ Package information loaded.

The package @ng-bootstrap/ng-bootstrap@12.1.1 will be installed and
executed.
Would you like to proceed? Yes
✔ Package successfully installed.
UPDATE package.json (1219 bytes)
✔ Packages installed successfully.
UPDATE src/app/app.module.ts (1887 bytes)
UPDATE angular.json (3166 bytes)
UPDATE src/polyfills.ts (2567 bytes)
```

2)

!!!!

Reactive Forms - Reactive forms are forms where we define the structure of the form in the component class. I,e we create the form model with Form Groups, Form Controls, and Form Arrays. We also define the validation rules in the component class. Then, we bind it to the HTML form in the template. This is different from the template-driven forms, where we define the logic and controls in the HTML template:

https://angular.io/guide/reactive-forms

Example Reactive Form

1) ng g c register

2) register.component.html:

```
<form [formGroup]="form">
  <div class="form-group">
    <label for="email" #highlight="highlight"
```

```
[appHighlight]="true">Email: </label>
    <input type="text" name="email" id="email" formControlName="email">
    <ul *ngIf="form.get('email')!.invalid && form.get('email')!.touched"
class="errors">
      <li *ngIf="form.get('email')!.errors?.required">Email is
Required</li>
      <li *ngIf="form.get('email')!.errors?.email">Email is Invalid</li>
    </ul>
  </div>

  <div class="form-group">
    <label for="password">Password: </label>
    <input type="password" name="password" id="password"
formControlName="password">
    <ul *ngIf="form.get('password')!.invalid &&
form.get('password')!.touched" class="errors">
      <li *ngIf="form.get('password')!.errors?.required">Password is
Required</li>
      <li *ngIf="form.get('password')!.errors?.minlength">Password
should be minimum 6 characters</li>
    </ul>
  </div>

  <div class="form-group">
    <label for="repeat-password">Repeat Password: </label>
    <input type="password" name="repeat-password" id="repeat-password"
formControlName="repeatPassword">
    <ul *ngIf="form.get('repeatPassword')!.invalid &&
form.get('repeatPassword')!.touched" class="errors">
      <li *ngIf="form.get('repeatPassword')!.errors?.required">Password
is Required</li>
      <li *ngIf="form.get('repeatPassword')!.errors?.minlength">Password
should be minimum 6 characters</li>
      <li *ngIf="form.get('repeatPassword')!.errors?.sameValue">Password
and repeatPassword should be the same</li>
    </ul>
  </div>
  <button>Login</button>
</form>
```

3) register.component.ts:

```
import { Component, OnDestroy, OnInit } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
import { Subscription } from 'rxjs';
import { sameValueValidateFactory } from '../same-value-validate-fn';

@Component({
  selector: 'app-register',
  templateUrl: './register.component.html',
  styleUrls: ['./register.component.css']
})
export class RegisterComponent implements OnDestroy{

  form: FormGroup;

  subscription: Subscription;

  constructor(private fb: FormBuilder) {

    this.form = this.fb.group({
      email: ['', [Validators.required, Validators.email]],
      password: ['', [Validators.required, Validators.minLength(6)]],
      repeatPassword: ['']
    });


    const sameValueValidate = sameValueValidateFactory('repeatPassword',
this.form.get('password')!, 'passsword');

    this.subscription =
this.form.get('password')!.valueChanges!.subscribe(() => {

      this.form.controls.repeatPassword.updateValueAndValidity( {
onlySelf: true});

    });

    this.form.controls.repeatPassword.setValidators([
      sameValueValidate,
      Validators.required,
      Validators.minLength(6)
    ]);

  }

  ngOnDestroy(): void {
    this.subscription?.unsubscribe();
    }

}
```

4) same-value-validate-fn:

```
import { AbstractControl } from '@angular/forms';

export function sameValueValidateFactory
(controlName: string,
 otherControl: AbstractControl,
 otherControlName: string) {
  return function sameValueValidate(
    control: AbstractControl,
    ) {
    return control.value !== otherControl?.value ? {
        sameValue: {
          [otherControlName]: otherControl?.value,
          [controlName]: control.value
        }
      }
      : null
  }
}
```

Pipes

Angular Pipes transform the output. You can think of them as makeup rooms where they beautify the data into a more desirable format. They do not alter the data but change how they appear to the user. ; pure pipes are memoized (in computing, memoization is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again.) - upon render the pure pipe is only executed when the input has been changed;

Example:

```
{{ username | uppercase }}

{{ username | lowercase | titlecase }}

{{ data.creationDate | date: 'fullDate' }}
```

!!! We can create our own Pipes:

1) `ng g p shared/get-prop`

2) shared.module.ts:

```
exports: [
    LoaderComponent,
    GetPropPipe
  ]
```

3) get-prop.pipe.ts:

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

@Pipe({
  name: 'getProp'
})
```

```
export class GetPropPipe implements PipeTransform {

  transform(value: Record<string | number, any> | any[], propName:
string | number): any {

    if (
      typeof value===null ||
      typeof value !== 'object' ||
      (Array.isArray(value) && typeof propName !== 'number')) {
      return null;
    }

    return value[propName as any];
  }

}
```

4) app.component.ts:

```
data = [{test: 'testing pipe'}, {test: 2}, {test: 3}];
```

5) app.component.html:

```
{{data | getProp: 0 | getProp: 'test'}}
```

Result:

testing pipe

!!!

!!! We can use the async pipe to subscribe to a Promise / stream, get its result and unsubscribe:

`{{ text | async }}` !!!

pure pipes vs impure pipes:

By default, pipes are defined as pure so that **Angular executes the pipe only when it detects a pure change to the input value.** A pure change is either a change to a primitive input value (such as String, Number, Boolean, or Symbol), or a changed object reference (such as Date, Array, Function, or Object).

vs

To execute a custom pipe after a change within a composite object, such as a change to an element of an array, you need to define your pipe as impure to detect impure changes. **Angular executes an impure pipe every time it detects a change with every keystroke or mouse movement**.

Define impure pipe:

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

@Pipe({
  name: 'getProp',
  pure: false
})
```

Generate pipe:

```
ng g p pipe-name
```

!!! **Re-use argument definition:**

```
type GetPropArguments = Parameters<typeof function_name>;
type GetPropFirstArgument = GetPropArguments[0];
```

!!!!

JWT (JSON Web Token)

JSON web token (JWT), pronounced "jot", is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. Again, JWT is a standard, meaning that all JWTs are tokens, but not all tokens are JWTs.
Because of its relatively small size, a JWT can be sent through a URL, through a POST parameter, or inside an HTTP header, and it is transmitted quickly. A JWT contains all the required information about an entity to avoid querying a database more than once. The recipient of a JWT also does not need to call a server to validate the token.

Header + Payload + Signature

!!! Where to store the JWT - in a cookie or in a local storage or in a third-party service - it is up to security and preference !!!

Interceptors

Interceptors allow us to intercept incoming or outgoing HTTP requests using the HttpClient. They can handle both HttpRequest as well as HttpResponse.

By intercepting the Http request, we can modify or change the value of the request, and by intercepting the response, we can do some predefined actions on a particular status code or message.

withCredentials: true - passes the cookies to the server

Example Interceptor - replacing the /api URI in the project with URL set in the environment.ts:

1) environments/envrionment.ts:

```
export const environment = {
  production: false,
  apiURL: 'https://jsonplaceholder.typicode.com'
};
```

2) user.service.ts:

```
loadUsers(search: string = '') {

    const query  = search ? `?email_like=${search}` : '';

    return this.http.get<IUser[]>(`/api/users${query}`)
}
```

3)   core/app-interceptor.ts:

```
import { HttpEvent, HttpHandler, HttpInterceptor, HttpRequest,
HTTP_INTERCEPTORS} from "@angular/common/http";
import {Injectable, Provider} from "@angular/core";
import {Observable} from "rxjs";
import { environment } from '../../environments/environment';

const {apiURL} = environment;
```

```
@Injectable()

export class AppInterceptor implements HttpInterceptor {

    intercept(req: HttpRequest<any>, next: HttpHandler):
Observable<HttpEvent<any>> {

      if (req.url.startsWith('/api')) {
        return next.handle(req.clone({
          url: req.url.replace('/api', apiURL),
          withCredentials: true
        }))
      }

       return next.handle(req);
    }

}

export const appInterceptorProvider: Provider = {
  provide: HTTP_INTERCEPTORS,
  useClass: AppInterceptor,
  multi: true
}
```

4) core/core.module.ts:

```
providers: [
    appInterceptorProvider
  ]
```

!!!! You may handle server error responses such as HTTP 401:

```
return next.handle(req)

  .pipe(catchError((err: HttpErrorResponse) => {

     if (err.status === 401) {

        // Log the errors

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

     }
```

```
      return throwError(err);
   }
 ));
```

!!!

Lazy Loading

Lazy loading is the process of loading components, modules, or other assets of a website as they're required. ; Lazy loading is a technique that allows you to load JavaScript components asynchronously when a specific route is activated. ;

in Angular - To implement Lazy Loading in Angular, we need to create modules for each section of the application and load the required modules when the user access the component. Now, these components are accessed by routes usually, therefore, lazy loading is configured in the routing where the required module is loaded when the user accesses the required route. This ensures that only the required modules are used in the application when needed, which helps create a highly responsive application.

!!! instead of **canActivate** we need to use **canLoad** !!!

Example Lazy Loading:

1) The module which will be lazy loaded should not be included / imported in the app module

2) Instead there should be a route to the lazy loaded module in the app routing module:

app-routing.module.ts:

```
{
    path: 'user',
    loadChildren: () => import('./user/user.module').then(m =>
m.UserModule)
  },
```

In the app routing module we need to set the below strategy:

```
export const AppRoutingModule = RouterModule.forRoot(routes, {
  preloadingStrategy: PreloadAllModules
});
```

3) Inside the lazy loaded module the paths should be relative to the module - instead of user-list we should set list:

user-routing.module.ts:

```
{
    path: 'list',
    component: UserListComponent
  },
```

4) The route to the components within the lazy loaded module must be set with the module in the URI - /user/list, /user/detail:

user-list-item.component.html:

```
<button type="button" [routerLink]="['/user/detail/',
user?.id]">Detail</button>
```

header.component.html:

```
 <a routerLink="/user/list" routerLinkActive="active">User List</a>
```

Subjects

A Subject is like an Observable, but can multicast to many Observers. Subjects are like EventEmitters: they maintain a registry of many listeners; A Subject and its subscribers have a one-to-many relationship.;

!!! A Subject is a special kind of Observable from the RxJS library which allows us to multicast values to the components which have subscribed to it. Whenever Subject emits a value, each of its subscribers gets notified about the emitted value. !!!

It has three methods:

1. next(value)- This method is used to emit a new value.
2. error(error) - This method is used to send error notifications.
3. complete()- This method suggests that the Subject has completed itswork. Once complete() method is invoked, calling next() or error() won't have any effect.

BehaviorSubject stores the last emitted value. When a new user subscribes, he will immediately get the stored value. While creating BehaviorSubject, we have to provide initial value.

ReplaySubject can emit multiple old values to new subscribers. While creating the ReplaySubject, you have to specify, how many old values you want to store and for how long you want to keep them.

Live Search + Behavior Subject Example

1) user.service.ts:

```
import {Inject, Injectable} from '@angular/core';
import {IUser} from "./interfaces/user";
import {myStringInjectionToken} from './app.module';
import {HttpClient, HttpClientModule} from "@angular/common/http";
import { BehaviorSubject } from 'rxjs';

@Injectable({
  providedIn: 'root'
})

export class UserService {

  private users = new BehaviorSubject<IUser[] | null>(null);
  users$ = this.users.asObservable();

  private user = new BehaviorSubject<IUser | null>(null);
  user$ = this.user.asObservable();

  constructor(private http: HttpClient) {

  }

  loadUsers = (search: string = '') => {

    const query  = search ? `?email_like=${search}` : '';

    this.users.next(null);

    this.http.get<IUser[]>(`/api/users${query}`, {
      withCredentials: true,
      headers: {
        'Content-type': 'application/json'
      }
    }).subscribe((users) => this.users.next(users));

  }

  loadUser = (id: number) => {

    this.user.next(null);

    this.http.get<IUser>(`/api/users/${id}`).subscribe(user =>
this.user.next(user));

  }

}
```

2) user-list.component.ts:

```
import {AfterViewInit, ChangeDetectionStrategy, Component, ElementRef,
EventEmitter, Input, OnInit, Output, ViewChild} from '@angular/core';
import {debounceTime, distinctUntilChanged, fromEvent, map } from
'rxjs';
import { UserService } from 'src/app/user.service';
import {IUser} from "../../interfaces/user";

@Component({
  selector: 'app-user-list',
  templateUrl: './user-list.component.html',
  styleUrls: ['./user-list.component.css'],

})
export class UserListComponent implements OnInit, AfterViewInit {

  @ViewChild('searchInput') searchInput!: ElementRef<HTMLInputElement>;

  @Input() userArray: IUser[] = [];

  users$ = this.userService.users$;

  constructor(public userService: UserService) { }

  ngOnInit(): void {

    this.loadUsers();

  }

  ngAfterViewInit() {
    fromEvent(this.searchInput.nativeElement, 'input')
      .pipe(
        map((e) => (e.target as HTMLInputElement).value),
        debounceTime(200),
        distinctUntilChanged()
      )
      .subscribe((value) => this.loadUsers(value));
  }

  loadUsers = this.userService.loadUsers;

}
```

3) user-list.component.html:

```
<input #searchInput [class.hidden]="!(users$|async)" type="text">

<div *ngIf="(users$ | async) as users">
  <button (click)="loadUsers()">Reload</button>
  <div id="container">

    <app-user-list-item *ngFor="let user of users"
[user]="user"></app-user-list-item>

  </div>
</div>

<app-loader *ngIf="!(users$ | async)"></app-loader>
```

Custom Error Handling Page with Interceptor

----

1) ng g c error

2) error.component.ts:

```
import { Component, OnInit } from '@angular/core';
import {ActivatedRoute} from "@angular/router";

@Component({
  selector: 'app-error',
  templateUrl: './error.component.html',
  styleUrls: ['./error.component.css']
})
export class ErrorComponent  {

  errorMessage: string;

  constructor(private activatedRoute: ActivatedRoute) {

    this.errorMessage = activatedRoute.snapshot.queryParams.error;
  }

}
```

3) error.component.html:

```
<h1>Error:</h1>
<p>{{errorMessage}}</p>

<a routerLink="/">Click here to go to home page</a>
```

4) core/app-interceptor.ts:

```
import {Injectable, Provider} from "@angular/core";
import {HTTP_INTERCEPTORS, HttpEvent, HttpHandler, HttpInterceptor,
HttpRequest} from "@angular/common/http";
import {catchError, Observable, throwError} from "rxjs";
import { environment } from '../../environments/environment';
import {Router} from "@angular/router";
import {UserService} from "../user/user.service";
```

```
const API_URL = environment.apiURL;

@Injectable()
export class AppInterceptor implements HttpInterceptor{

  constructor(private router: Router,
              private userService: UserService) {
  }

  intercept(req: HttpRequest<any>, next: HttpHandler):
Observable<HttpEvent<any>> {

    let reqStream$ = next.handle(req);

    if (req.url.startsWith('/api')) {
      reqStream$ = next.handle(req.clone({
        url: req.url.replace('/api', API_URL),
        withCredentials: true
      }));
    }

    return reqStream$.pipe(catchError((err) => {
      this.router.navigate(['/error'], { queryParams: { error:
err.message} });
      return throwError(err);
    }));
  }

}

export const appInterceptorProvider: Provider = {
  provide: HTTP_INTERCEPTORS,
  useClass: AppInterceptor,
  multi: true
}
```

5) core/error-handler.ts:

```
import {ErrorHandler, Injectable, Provider} from "@angular/core";
import {Router} from "@angular/router";

@Injectable()
```

```
class GlobalErrorHandler implements ErrorHandler {

  constructor(private router: Router) {

  }

  handleError(error: any): void {

    console.log(error);

    this.router.navigate(['/error'], {queryParams: {error: "Ooops!
Something went wrong!"}});

  }

}

export const globalErrorHandlerProvider: Provider = {
  provide: ErrorHandler,
  useClass: GlobalErrorHandler
}
```

6) core.module.ts:

```
providers:[
    appInterceptorProvider,
    globalErrorHandlerProvider,
```

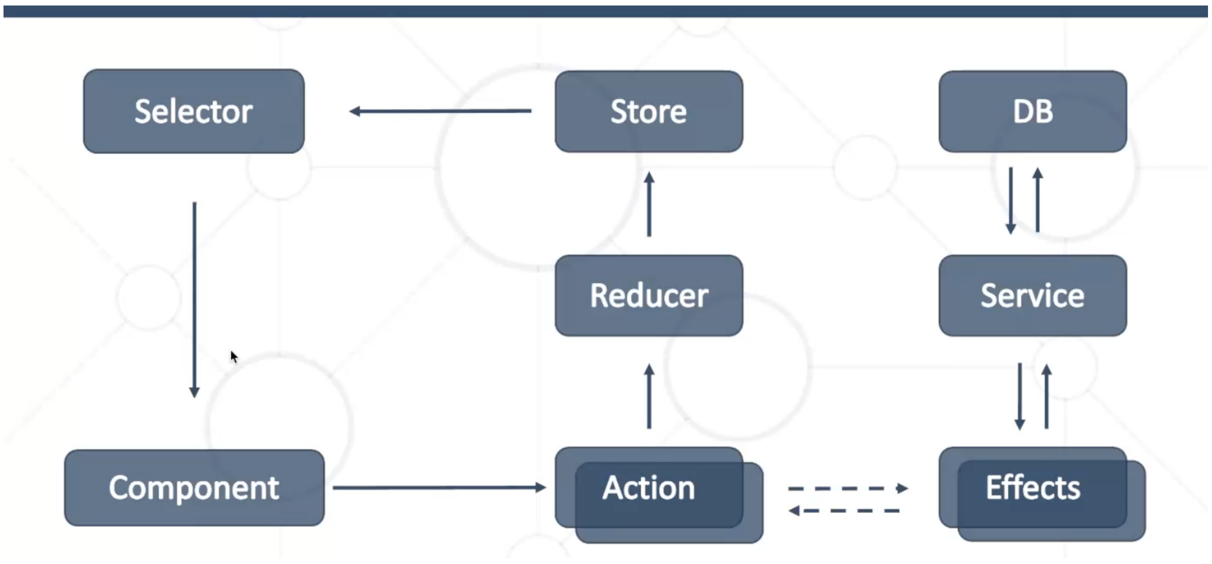
7) user.service.ts:

```
getProfileInfo() {
    return this.http.get<IUser>(`/api/users/profile`, {}).pipe(
      tap((user) => {
        (this as any).user.test = user;
      })
    );
  }
```

Redux and NgRx

Redux is a pattern and library for managing and updating application state, using events called "actions". It serves as a centralized store for state that needs to be used across your entire application, with rules ensuring that the state can only be updated in a predictable fashion.

NgRx is a framework for building reactive applications in Angular. NgRx provides state management, isolation of side effects, entity collection management, router bindings, code generation, and developer tools that enhance developers experience when building many different types of applications.



npm install @ngrx/store --save

or

ng add @ngrx/store (Angular CLI 6+)

!!! Redux DevTools - > Chrome Extensions which detects the states !!!

Basic State Example:

1) ng add @ngrx/store

2) ng add @ngrx/store-devtools

3) app/+store/reducer.ts:

```
export interface IAppState {
  counter: number,
  value: any
}

const initialState: IAppState = {
  counter: 0,
  value: null
}

export function appReducer (state: any = initialState, action: any) {

  if (action.type === 'INC') {
    return { ...state, counter: state.counter + 1};
  }

  if (action.type === 'SET_VALUE') {
    return {...state, value: action.payload};
  }

  return state;
}
```

4) app/+store/actions.ts:

```
export function incrementCounter() {

  return {
    type: 'INC'
  }

}

export function setValue (value: any) {
  return {
    type: 'SET_VALUE',
    payload: value
  }
}
```

5) app/+store/selectors.ts:

```
export function getCounter (s: any) {
  return s.state.counter;
}

export function getValue (s: any) {
  return s.state.value;
}
```

6) app.module.ts:

```
imports: [
StoreModule.forRoot({
      state: appReducer
    }),
    StoreDevtoolsModule.instrument({})
```

7) app.component.ts:

```
import { Component } from '@angular/core';
import {Store} from "@ngrx/store";
import { incrementCounter , setValue } from './+store/actions'
import {getCounter, getValue} from "./+store/selectors";

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {

  counter$ = this.store.select(getCounter);
  value$ = this.store.select(getValue);

  constructor(private store:Store<any>) {

  }

  incrementCounter(): void {
    this.store.dispatch(incrementCounter())
  }

  setValue(valueInput: HTMLInputElement) {
    this.store.dispatch(setValue(valueInput.value))
    valueInput.value = '';
  }

}
```

8) app.component.html:

```
{{counter$ | async}}

<button (click)="incrementCounter()">Increment</button>

<div>Value is: {{value$ | async }}</div>

<input #valueInput >
<button (click)="setValue(valueInput)">Submit</button>
```

The same State Example with NgRx:

1) + 2) - the same

3) +store/actions.ts:

```
import { createAction, props } from '@ngrx/store';

const namespace = '[GLOBAL]';

export const incrementCounter = createAction(
  `${namespace} increment counter`
);

export const setValue = createAction(
  `${namespace} set value`,
  props<{value: string}>()
);
```

4) +store/reducers.ts:

```
import {Action, createReducer, on, State} from "@ngrx/store";
import {incrementCounter, setValue} from "./actions";

export interface IGlobalState {
  counter: number,
  value: any
}

const initialState: IGlobalState = {
  counter: 0,
  value: null
}

export const globalReducer = createReducer(
  initialState,
  on(incrementCounter, (state) => ({...state, counter: state.counter +
1})),
  on(setValue, (state, {value}) => ({...state, value}))
);
```

5) +store/selectors:

```
import {IState} from "./index";
import {createSelector} from "@ngrx/store";

export const selectGlobal = (state: IState) => state.global;

export const selectGlobalCounter = createSelector(
  selectGlobal,
  state => state.counter
);

export const selectGlobalValue = createSelector(
  selectGlobal,
  state => state.value
);
```

6) +store/index.ts:

```
import {globalReducer, IGlobalState} from "./reducers";
import {ActionReducerMap} from "@ngrx/store";

export interface IState {
  readonly global: IGlobalState
}

export const reducers: ActionReducerMap<IState> = {

  global: globalReducer

}
```

6) app.module.ts:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
import {StoreModule} from '@ngrx/store'
import {appReducer} from "./store_demo/reducers";
import { StoreDevtoolsModule } from '@ngrx/store-devtools';
import { environment } from '../environments/environment';
import {reducers} from "./+store";

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    StoreModule.forRoot(reducers),
    StoreDevtoolsModule.instrument({})
  ]
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

7) app.component.ts:

```
import { Component } from '@angular/core';
import {Store} from "@ngrx/store";
import { incrementCounter , setValue} from "./+store/actions";
import {selectGlobalCounter, selectGlobalValue} from
"./+store/selectors"

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {

  counter$ = this.store.select(selectGlobalCounter);
  value$ = this.store.select(selectGlobalValue);

  constructor(private store:Store<any>) {

  }

  incrementCounter(): void {
    this.store.dispatch(incrementCounter())
  }

  setValue(valueInput: HTMLInputElement) {
    this.store.dispatch(setValue({value: valueInput.value}))
    valueInput.value = '';
  }
}
```

8) app.component.html:

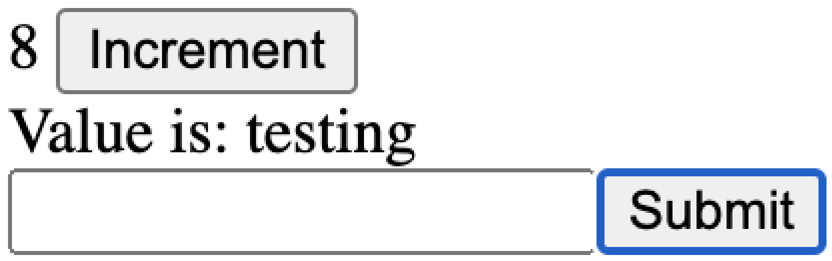
```
{{counter$ | async}}

<button (click)="incrementCounter()">Increment</button>

<div>Value is: {{value$ | async }}</div>

<input #valueInput >
<button (click)="setValue(valueInput)">Submit</button>
```

Result:



NgRx Effects - Effects use streams to provide new sources of actions to reduce state based on external interactions such as network requests, web socket messages and time-based events ; Effects are usually used to perform side effects like fetching data using

HTTP, WebSockets, writing to browser storage, and more.

!!! Most effects are straightforward: they receive a triggering action, perform a side effect, and return an Observable stream of another action which indicates the result is ready. NgRx effects will then automatically dispatch that action to trigger the reducers and perform a state change. !!!

NgRx Effects Example (added to the above State Example):

1) +store/effects.ts:

```
import {Injectable} from "@angular/core";
import {Actions, createEffect, ofType} from "@ngrx/effects";
import {incrementCounter, setValue} from "./actions";
import {map} from 'rxjs/operators'

@Injectable()
export class  GlobalEffects {

  increment = createEffect(() => this.actions$.pipe(
    ofType(setValue),
    map(action => {
      return incrementCounter();
    })
    ));

  constructor(private actions$: Actions) {

  }

}
```

2) app.module.ts:

```
 imports: [
    BrowserModule,
    StoreModule.forRoot(reducers),
    StoreDevtoolsModule.instrument({}),
    EffectsModule.forRoot([
      GlobalEffects
    ])
  ]
```

!!! ofType can be replaced by filter:

ofType(setValue)

- >

filter(action => action.type === setValue.type !!!

!!! We can set up a store within a custom module:

1) custom_module/+store - actions, reducers, effects, index.ts

2) custom_module.module.ts:

```
imports: [
StoreModule.forFeature('user', reducers),
EffectsModule.forFeature([
      UserListEffects,
      UserDetailEffects
    ])
```

!!!

How to clear specific state (in this example: global state) and the app state on the app (all of the states):

1) +store/actions.ts:

```
export const clearGlobalState = createAction(
  `${namespace} clear global state`
);

export const clearAppState = createAction(
  `${namespace} clear app state`
);
```

2) +store/meta-reducers.ts:

```
import {ActionReducer, MetaReducer} from "@ngrx/store";
import {IState} from "./index";
import {clearAppState} from "./actions";

export function clearAppStateMetaReducer(reducer: ActionReducer<any>):
ActionReducer<any> {

  return function(state: IState, action) {

    if (action.type === clearAppState.type) {
      return reducer(undefined, action)
    }
    return reducer(state, action);
  };
}

export const metaReducers: MetaReducer<any>[] =
[clearAppStateMetaReducer];
```

3) +store/reducers.ts:

```
export const globalReducer = createReducer(

.....

  on(clearGlobalState, () => initialState)
```

4) app.module.ts:

```
import {metaReducers} from "./+store/meta-reducers";

imports:
....

    StoreModule.forRoot(reducers, {metaReducers}),
```

We have finished setting up the reset functions - now to set on the component:

5) app.component.ts:

```
 constructor(private store:Store<any>

............

 resetState(): void {
    this.store.dispatch(clearAppState());
  }
```

6) app.component.html:

```
<button (click)="resetState()">RESET APP STATE</button>
```

How to setup basic state store:

actions.ts (setting up types in the state to be monitored) - >
reducers.ts (setting up what will happen when the state changes) - >
index.ts (setting up the access to the state and the reducers from outside) ->
selectors.ts (setting up the access to individual parts of the reducers) ->
(optional) effects.ts (external interactions about which your components doesn't need explicit knowledge)

https://blog.angular-university.io/ngrx-entity/ - Angular NgRx Entity - to store data

https://github.com/IliaIdakiev/ngrx-action-bundles - Angular NgRx - bundle of predefined actions

https://www.youtube.com/c/AngularAirPodcast