Angular

Set Up

- install Node.js
- npm install -g @angular/cli
    - run ng version to check if installation was successfull
- ng new ProjectName
    - cd Project Name
- ng serve → start the app in Dev Mode
- download Angular Language Service VS Code extension

Basics

- it is also component driven
- your app.component.ts has, by default, an export class AppComponent with a title variable
    - you can import that component and access those variables: {{title}} to use it
    - selector → tag-name
    - templateUrl → where the template for that component is

    - styleUrls → where the style for that component is
- index.html has: <app-root> → the selector for the app component
    - this means that angular will load that component there (where <selector-name> is used)

Structure

- create a shared folder on the level of your app folder

Shared

Models

- basically your domain/entities

Services

- for services

Items and Arrays

Model

- create an item.ts file and an Item class inside it, export it and import it in your app.component.ts
    - with this, you can use your Item class
- go to Module section

Array

- declare your array in your app.component.ts
- to parse through all items of an array:
    - <div *ngFor="let item of items">{{item.getName()}}
    - MAKE SURE YOUR COMPONENT IMPORTS CommonModule

- make sure you specify the object type you have in your array
- check if array is empty: nglf
  - <div *nglf=”array.length ===0”> length is 0
  - <div *ngFor... *nglf=”array.length>0”>...

Structural Directories

nglf

- in a div (or whatever) you can use *nglf=”condition” and if that is true than that component is rendered, if not than it is not
- you can use it this way:
  <div *nglf=”items.length ===0; then noItems else showItems”>
  <ng-template #noItems>
  this is displayed if items.length===0 according to the div above!!!
  </ng-template>
  <ng-template #showItems> ... </ng-template>
  - on short, you can do
    <div *nglf=”...; else showItems”>
    this is displayed if condition is true

    <ng-template #showItems>
    ...
    <ng-template>

ngFor

- <div *ngFor=”let item of items”>{{item.getName()}}

ngOnInit

- this does smth as a component is created

Binding Data (to properties and attributes)

Property Binding

Checkbox

- input element with a type checkbox have a checked attribute
- we want to bind a boolean data to that checked js attribute on an input element
- how to?:
  [checked] = “wish.getIsComplete()” → this bind that element
  (change) = “wish.changeIsComplete()” → this calls wish.changeIsComplete” whenever the checkbox is used

Binding To Attributes

- [attr.data-index]=”ii” ⇒ binds an item to his index in the array

Event Binding

- (clicked)= “function (defined in) appcomponent.ts(parameter)”
  it will trigger an event whenever you click that with the given parameters

Binding for Forms

Input

- you can bind an input value to a field declared in that component class, for example:
  export class AppComponent(){...
  itemName = ‘’; addItem(...){...} }
  ——-
  in app.component.html
  <input type=”...” class=”...” name=”must-have-a-name”
  [(ngModule)]=”itemName”>
  <button (click) = “addItem(itemName)”

⇒ this will call the addItem function with the value written in the input as parameter

Select

- you can bind data to a select element like this: → basically you can specify the default value this way
  <select ... [(ngModel)] =”listValue”>
  <option value=”...”>All</option>
  ...
  </select>
- whenever a select item is chosen, that listValue changes

Filter

- you can use (ngModelChange) =”functionName($event) as parameter” and that event has the new value (the value of the new option chosen) and do smth with it

Getter

- you can create a getter instead of a copy of the main array and filter that one;
- you do not use ngModelChange anymore in your select filter
- get visibleItems() : Item[] {
  if(listValue = ...) { return allItems.filter(...) }
  else { return allItems.filter(...) }
  return allItems;
  }
- this getter is from what I can tell automatically called and rerendering is done automatically

Two-Way Binding

- you can pass data down as much as u want:
  (outputEvent)=”inParentComponent = $event” → inParentComponent variable gets the value of the event emitted by the output event in the childComponent

- [()]→ is both a property and an event binding ⇒ 2way binding
- WHEN you have two way binding, you have:
  @Input() myInput ...
  @Output() myInputChange ...
  and you call it [(myInput)] =”input/this will be changed”
- you can set it to different things:
  [thisIsForInputInChild] =”where he gets that impurt from parent”
  (thisIsForInputInChildChange)=”where the output to go = $event” or smth

Custom Components

- you can create custom components and re-use them (react type shit)
- ng generate component componentName - to automatically create a new component

Pass data to a Custom Component:

- @Input() fieldName: string/number/...;
- <app-my-component [parameter1]=”getter/actualValue/$event/...”></app-my-component>

Form Component

- you need to create an event within that component:
  (addItem) = “do-smth-in-parent-component”
- you create an Output value:
  @Output() addItem = new EventEmitter<Item>()
  addItem(){
  this.addItem.emit(new Item(...)
  }
  - basically you output data from a component
- whenever addItem function is called in the child component, call the other function in the parent component

Styling Components

- you can use [ngClass] to give them a class
  [ngClass] =" 'strikeout' " → .strikeout{...}
- you can do smth like:
  [ngClass] = " isCompete ? strikeout : classIfNotComplete "
- GETTER for class:
  get cssCasses(){
  if(...)
  return ...;
  else
  return ['class1', 'class2', ...]; // that are defined in your .scss / css
  }

How to pass Input/Output through more components (basically context)

- You create an EVENTBUS: you have an observer that each component subscribes to so they can communicate with it directly, not by passing 'props' throughout 1000 components.

Create a service

- Go to Create service from JSON
- export class EventService {
  private subject = new Subject();
  //emit → this will emit
  emit(eventName: string, payload: any){
  this.subject.next({eventName, payload});
  // this basically passes the object you want the subscribers to work with: an object with an eventName and a payload
  }
  //listen →

```
    listen( eventName: string, callback: (event: any) {
        this.subject.asObservable().subscribe( (nextObj: any) => {
            if( eventName === nextObj.eventName ){
                callback(nextObj.payload);
            } // if the eventName is in the nextObj, then you will call the function
of the nextObj
with the required parameters
        }
    }
}
```

Injecting Dependencies

- for your SERVICES
  @Injectable({
  providedIn: 'root' → application level injection,
  })
  export class EventService{...}
- in your app.config.ts you need to include EventService in providers
- you need to specify in constructor for lists/items/app/ whatever component u use:
  constructor(events: EventService){...}

HTTP Requests

- can be done with the FETCH api
- you can use HttpClientModule
  - add to your imports in your app.component → available throughout application

How to fetch from JSON

- items! : Item[]
- create an items.json file in "assets" folder

- {
  {"itemName": "...",
  "isComplete": false
  },
  {"itemName2": "...",
  "isComplete": false
  }
  }

Create Service

- ng generate service Item⇒ ItemService is created in a file: wish.service.ts
  - it is automatically injectable
  - import it in your app.component.ts
  - add it to your constructor as private
- CODE TO FETCH JSON:
  - import HttpClient from '@angular/common/http'
  - constructor(private http: HttpClient){}
    getItem(){
    //callin a method (get/post/...) doesn't actually call the request, you need to use .subscribe()
    return this.http.get('assets/items.json').subscribe();
    }
- in your app component:
  implements OnInit
  - ngOnInit(): void {
    this.itemService.getItems().subscribe((dataFromJson) ⇒ this.items=dataFromJson)
    }
- POST:
  this.http.post(url, body, options)

Option for HTTP Requests

- header: new HttpHeaders(
  ‘Cotent-Type’: ‘application/json’
)
  ⇒
  you can use those options for your requests:
  http.get(’assets/items.json, header)
- to add a new header-field:
  header.set(’Authorization’, ‘value-for-auth’)
- to give parameters:
  header.params = new HttpParams({
  fromObj={
  format:’json’}
  })

HTTP Errors

- import { catchError }
- to your get/post/... request:
  this.http.get(...).pipe(catchError(this.handleError);
- private handleError(error: HttpErrorResponse){
  if(error.status === 0){
  console.error(`Issue with the client/network: ‘, erorr.error)
  } else {
  console.error(’Server-side error: ‘, error.error }
  }
  return throwError(() ⇒ new Error(’Cannot get from the server’)
  }
- when you call the subscribe message:
  .subscribe((data) ⇒ {},
  (error:any) ⇒ {do smth}
)

Modules

(not the best-practice)

- ng generate module item ⇒ item directory with an item.module.ts created
- item-filter, item-list, actual-item etc. (whatever uses that Item class) add them to the folder called “item”
- item service should also be here
- create an item component inside that item module
- add imports in the app.config.ts
- add other exports/imports/declarations as needed
  declarations: item-list, item-filter
  exports: ItemModule

Forms

Reactive Forms

- ng generate module contact
- ng generate component contact -m contact
- this should create a contact component and a contact module inside a contact folder
- in contact.component:
  import formControl
  create fields:
    - add [formControl] = “name” inside your html component
- create a submit function in your contanct.component.ts
  and add it to your ngSubmit in your form field
- in contact.component:
  import { formControl, formGroup }
  create a contactForm = new FormGroup({

name = FormControl()
email=FormControl()...
})

- <form [formGroup] = “contactForm” (ngSubmit)=$event.preventDefault(); submitForm()>
  <input formControlName = “name”>
  ...
  <input type=”email” formControlName =”email> ...

Validate Form Inputs

import {Validators} from ‘@angular/forms’

contactForm = new FormGroup({
name = FormControl(’’, [Validators.required,Validators.minLength(10), Validators.maxL(50)]) ⇒ ⇒this makes that field required, between 10 and 50 characters;
email = FormControl(’’, [Validators.required, Validators.email] ⇒ check to be email

- DISABLE button if form INVALID:
  [disabled]=”!contactForm.valid”/”contactForm.invalid”
- add <div *ngIf=”contactForm.get(’email’)?.invalid && (contactForm.get(’email’).dirty || contactForm.get(’email’)?.touched)>
  <small *ngIf=”contactForm.get(email)?.hasError(’required’)> This field is required </small>
  <small *ngIf=” same.hasError(’email’)> Please enter a valid email address </small>
   // this will show a message wherever u add the div if the email field has been clicked // and moved away from (touched), but not completed with a valid email address (can do this // for any other input ; you can add other hasError(’minLength’/’maxLength’) etc.

Create custom validator

- invalidEmailDomain.ts
  export function invalidEmailDomain(control: AbstractControl) : ValidationErrors

```
| null {
    const value = control.value?.toLowerCase();
    const hosts = ['gmail.com', 'yahoo.com']

   if(!value) return null

   const matches = hosts.some(host ⇒ value.indexOf('@${host}'} >0);
// this check that for each "host" in hosts, if value.indexOf(hosts[0/1/.../n] >0 )
= meaning that it exists; it will return true ⇔ ValidationError otherwise it will
return null
// this means that gmail and yahoo are not ok for the form
   return matches ? { invalidEmailDomain: true} : null;
}
```

- contactForm = new ..{
  emai = new FormControl('', [Validators.required, ..., invalidEmailDomain

Angular Router

- ng new my-application
  say that you want to use AngularRouter
- app.routes.ts →
  const routes: Routes = [
  {path: 'home', component: HomeComponent},
  {path: 'shop', component: ShopComponent}
  ] //
  https://localhost:4200/home ⇒ HomeComponent
  //
  https://localhost:4200/shop ⇒ ShopComponent
- {path: '', component: HomeComponent ⇒
  ⇒
  localhost:4200 will load the HomeComponent
- Add a NOT FOUND: {path:'**', component:NotFoundComponent}
  - this must be the last route in the routes constant

Navigate to Routes

- add a HTML link
- <a routerLink="">Home
- <a routerLink="shop">Shop
- <a routerLink="#" (click)="$event.preventDefault(); goToPage()">Page
  in your.component.ts:
  constructor(private router: Router){}
  goToPage(){
  this.router.navigate(['contact', 'us']) ⇒ goes to /contact/us
  }

Provide Data to Routes

- { path: 'potions', component: PotionslistComponent }
  { path: 'potions/:id', component: PotionComponent }
- in your PotionComponent:
  potion: any = {}
  constructor(private store: PotionsService private router: ActivatedRouter) {
  }
  ngOnInit(): void {
  this.route.paramMap.subscribe((params: paramMap) ⇒ {
  let id = params.get('id);
  if(id){
  this.store.getPotion(id).subscribe(potion ⇒ this.potion = potion);}
  }
  }
  - getPotion(id: number){
    return of(this.potions.find(p ⇒
    p.id === id)
    } //returns observable
- to create a link to that specific id
  <a [routerLink]='"['/potions', potion.id]"> ⇒ will make potions/id link

Pagination

- import { PaginatorModule } from ‘primeng/paginator’
- add to app.component.ts imports
-