Inhoud

[Some commands 3](#_Toc33597918)

[Routes 3](#_Toc33597919)

[Add links (i.e. in a navigation bar) 4](#_Toc33597920)

[Mark the active link 4](#_Toc33597921)

[Avoid reloading a page: 4](#_Toc33597922)

[Add child route 4](#_Toc33597923)

[Get a param from the route 4](#_Toc33597924)

[Mark the selected (active) item 4](#_Toc33597925)

[Observables 4](#_Toc33597926)

[Cross component data sharing: 4](#_Toc33597927)

[Forms 5](#_Toc33597928)

[Template driven approach 5](#_Toc33597929)

[Setup the template 5](#_Toc33597930)

[Validation 5](#_Toc33597931)

[Add a help text in case of an invalid input: 5](#_Toc33597932)

[Assign default value to an input 5](#_Toc33597933)

[Group controls in a form 5](#_Toc33597934)

[Adding a radio button 6](#_Toc33597935)

[Setting a default value 6](#_Toc33597936)

[Reactive Setup 6](#_Toc33597937)

[FormGroup and FormControl 6](#_Toc33597938)

[Submitting form 6](#_Toc33597939)

[Validation 6](#_Toc33597940)

[Setting a message if invalid 6](#_Toc33597941)

[Grouping controls 7](#_Toc33597942)

[Arrays of formControls 7](#_Toc33597943)

[Custom Validators 8](#_Toc33597944)

[Listen to value / status changes 9](#_Toc33597945)

[Set- and patch value 9](#_Toc33597946)

[Pipes 9](#_Toc33597947)

[HTTP 10](#_Toc33597948)

[Setup 10](#_Toc33597949)

[Post 10](#_Toc33597950)

[Get 10](#_Toc33597951)

[Pipe 10](#_Toc33597952)

[map 10](#_Toc33597953)

[Response type 11](#_Toc33597954)

[ErrorHandlinlg 11](#_Toc33597955)

[catchError 11](#_Toc33597956)

[throwError (observable) 11](#_Toc33597957)

[Define interface for response 11](#_Toc33597958)

[Request headers and params 11](#_Toc33597959)

[Response headers 12](#_Toc33597960)

[Events => pipe.tap 12](#_Toc33597961)

[Interceptors 12](#_Toc33597962)

# Some commands

|  |  |
| --- | --- |
| Opstarten angular server on port 4200 | ng serve |
| Each ZIP file holds code files which you can use to compare your code to it. Or you execute my projects:  1) Extract  2) npm install in the extracted folder  3) ng serve  Got an error related to TypeScript thereafter?  Run npm install --save-dev-exact typescript@3.4.5 in the same project folder and re-try ng serve. | Uit chap 12 |
| Later nog eens goed bekijken | Chapt 14 over AppModule |
| Generate component | ng generate component xxxx or ng g c xxxx |
| Error running script from terminal | Set-ExecutionPolicy RemoteSigned -Scope CurrentUser |
| Create new project | ng new <projectname> |
| Aanpassen navbar navigation menu manage | Zie lesson 50 |
| Directives   * Change content of attributes of an Element * Listen to events on an element | @HostBinding  @HostListener |
| More info about the Renderer | <https://angular.io/api/core/Renderer2> |
| Debugging in dev-tools | Open webpack > **.** > src/app/……..  Or use chrome augury extension |
|  | |
|  |  |

# Routes

To setup routes:

* create an AppRoutingModule
  + Add an array with routes to it, like { path: 'recipes', component: RecipesComponent}
  + Add a root path to redirect to the default path: like: { path: 'recipes', component: RecipesComponent}
  + Define imports and exports in @ngModule: imports: [RouterModule.forRoot(appRoutes)],  
    exports: [RouterModule]
  + And add this new AppRoutingModule to the app
* Add the AppRoutingModule to the app.module.ts imports

## Add links (i.e. in a navigation bar)

* In the <a > element add a routerlink, like so:   
  <a routerLink="recipes">Recipes</a> or   
  <a [routerLink]="['shopping-list', ‘ietsExtras’]">Shopping List</a>

## Mark the active link

* Add routerLinkActive to the <li>, like: <li routerLinkActive="active"…

## Avoid reloading a page:

* Remove href=”#” from the <a….> element

## Add child route

* children:  
   {path: ':id', component: RecipeDetailComponent}

 ]

## Get a param from the route

* const id = +this.route.snapshot.params['id'];   
  let op het plusteken, dat zorgt ervoor dat de waarde naar int gecast word.  
    
  dit geeft alleen de param bij init van component en niet als id wijzigd. Gebruik dan onderstaande oplossing en subscribe op wijzigende params:  
   this.route.params.subscribe(

( params: Params) => {

const id = +params['id'];

console.log('reccipeId: ' + id);

this.recipe = this.recipeService.getRecipe(id);

}

## Mark the selected (active) item

* routerLinkActive="active"

## Observables

* useful documentation:  
  Official Docs: <https://rxjs-dev.firebaseapp.com/>

RxJS Series: <https://academind.com/learn/javascript/understanding-rxjs/>

Updating to RxJS 6: <https://academind.com/learn/javascript/rxjs-6-what-changed/>

* there are three possible functions to implement: next, error and complete  
   anObservable.subscribe(

(data: string) => {

// the observalble was succesful and emitted an event

},

(error: Error) => {

// the observable raised an error which can be handled here

},

() => {

// the observable has completed

}

)

## Cross component data sharing:

* instead of using an EventEmitter.emit(...) use Subject.next(…) which id more efficient.

## Resolver

To perform a specific thing before a route is loaded a Resolver can be used. The resolver will do what it hat to do and then load the route. The below examples shows how recipes are loaded and set on the recipes on the recipeService by using a pipe(tap ( recipes ) => { …. }

export class RescipesResolverService implements Resolve<Recipe[]> {   
 constructor(private recipeService: RecipeService){}   
 resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Recipe[] |   
 Observable<Recipe[]> | Promise<Recipe[]> {  
 console.log('in resolve, aantal recipes is ' + this.recipeService.getRecipes().length) ;  
 if (this.recipeService.getRecipes().length > 0 ) {  
 return this.recipeService.getRecipes();  
 } else {  
 return this.**recipeService.fetchAllRecipes();**  
 }  
 }  
}

**fetchAllRecipes()**{   
 return **this.dataStorageServie.fetchAllRecipes()**  
 .pipe( tap( recipes => {  
 this.recipes = recipes;  
 this.recipesUpdated.next(this.getRecipes());  
 }));  
 }

{path: ':id', component: RecipeDetailComponent, **resolve: [RescipesResolverService]},**  
{path: ':id/edit', component: RecipeEditComponent, **resolve: [RescipesResolverService]},**

# Forms

## Template driven approach

This is one of the two approaches.

### Setup the template

* First give all the controls you want to get access to in the .ts, the attributes ‘ngModel’ and ‘name=”…..”
* On the form tag add an onSubmit listener: (ngSubmit)=”onSubmit()”
* Add a local reference (i.e. ‘f’) to the form (ngSubmit)=”onSubmit()” #f=”ngForm” which will give the .ts access to the form.  
  Instead of parsing the form ‘f’ to onSubmit, add a   
  @ViewChild(‘f’, {static: false} formName: Form to the .ts and remove the parm from onSubmit. This last approach is especially useful if you want to access the controls on the form before submitting the form.

### Validation

* Add directives to the controls to validate it. i.e. ‘required’ ‘email’ pattern=”^[1-9][0-9]\*$”  
  Check out the Validators class: <https://angular.io/api/forms/Validators> or search the Angular docs for ‘Validator’ <https://angular.io/api?type=directive>   
  Additionally, you might also want to enable HTML5 validation (by default, Angular disables it). You can do so by adding the ngNativeValidate  to a control in your template.
* To mark a field as invalid, add some css to it, like so  
  input.ng-invalid.ng-touched{  
   color: red;  
   border: solid;  
  }

### Add a help text in case of an invalid input:

* Add a local reference to the input control: #email=”ngModel”
* Add text: <span class="ng-valid ng- touched" \*ngIf="email.invalid && email.touched">Please enter a valid email adress</span>

### Assign default value to an input

* Bind a field using one way binding with ngModel: with a hardcoded string: [ngModel]="'pet'" or to a field defined in the .ts like so: [ngModel]="defaultSecret"
* Using two way binding by
  + Define field in .ts i.e. answer = ‘’
  + Add ngModel like so: [(ngModel)]="answer"
  + Now use the answer field in another place in the template:   
    <p>your answer was: {{ answer }}</p>

### Group controls in a form

* Add an ngGroup directive to the div which contains the controls you want to group like so:

ngModelGroup="userData"

* Add a local ref to the above ngModelGroup so you can refer to it   
  add: #userData="ngModelGroup"  
  refer: <p \*ngIf="userData.invalid">Invalid group</p>

### Adding a radio button

* Add array with genders to .ts and ad following
* <div class="radio" \*ngFor="let gender of genders">  
              <label for="gender">  
                <input  
                  type="radio"  
                  name="gender"  
                  ngModel  
   [value]="gender"  
   required>  
                  {{ gender }}  
              </label>

### Setting a default value

for field(s) on the form can be done by either:

* this.signUpForm.setValue({…….: ….}), in which case all the fields on the form should be provide by this json
* second way is to use pathchValue which allow you overwrite specific elements of the json, like so: this.signUpForm.**form**.patchValue({userData: {username: suggestedName}});

## Reactive Setup

### FormGroup and FormControl

* Add ReactiveFormsModule to app.module.ts
* A FormGroup is nothing more than a number of controls in a form grouped together.
* Create a field (i.e. signupForm) of type FormGroup
* Initialize the field before the page is rendered, i.e. in ngOnInit() like so:  
  new FormGroup( {‘username‘: new FormControl(‘initValue’)} // that’s enough for now
* Bind the form to the property signupForm in our .html: <form [formGroup]="signupForm">
* Bind the controls now, add a FormControlName to the input elements: FormControlName=”username”`

### Submitting form

* On the form tag add an onSubmit listener: (ngSubmit)=”onSubmit()”, no need to bind the form to the .ts as the signupForm is create in the .ts and then bound via formGroup property

### Validation

* Add validators to the groupControls created in the .ts (NOT in the .html as this only holds the controls which are synchronized to the .ts
* Add validators to control like so:  
  ‘username’: new FormControl(‘null’, Validators.required) // just one… or  
  ‘email’: new FormControl(‘null’, [Validators.required, Validator.email] // array of validators

### Setting a message if invalid

We can’t get the control by using a local ref. However we get the ref to the control with get(‘filedname’) like so:

<span   
 \*ngIf="**signupForm.get('username')**.invalid && **signupForm.get('username').**touched"  
 class="help-block">Please enter a username</span>

### Grouping controls

To group controls just nest the formControls like so:

    this.signupForm = new FormGroup({  
      'userData': new FormGroup({  
        'username': new FormControl(null, Validators.required),  
        'email': new FormControl(null, [Validators.required, Validators.email])  
      }),  
      'gender': new FormControl('male')  
    })  
  }

### Arrays of formControls

To dynamically add controls form an Array/List use the FormArray.

* Add the array to the form

this.signupForm = new FormGroup({  
 'userData': new FormGroup({  
 'username': new FormControl(null, Validators.required),  
 'email': new FormControl(null, [Validators.required, Validators.email]  
 }),  
 'gender': new FormControl('male'),  
 **'hobbies': new FormArray([])**  
 })

* Add a method to add a hobby to the array

onAddHobby(){  
 const hobby = new FormControl(null, Validators.required);  
 (<FormArray>this.signupForm.get('hobbies')).push(hobby);  
 }

* Add a method which returns an array of Controls. **Note**: Cast the result to FormArray!

getControls(){  
 return (<FormArray>this.signupForm.get('hobbies')).controls;  
}

* In the template: Add a new div which contains a button to add a element (control) and an input for every item in the array. Bind the new div with: formArrayName=”…..”.

<div **formArrayName="hobbies"**>  
 <h4>Your hobbies...</h4>  
 <button   
 class="btn btn-primary"  
 type="button" (click)="**onAddHobby()**"  
 >Add hobby</button>  
 <div   
 class="form-group"  
 \*ngFor="let hobbyControl of **getControls()**; let **hobbyIndex** = index" >A  
 <input type="text" class="form-control" [formControlName]=" **hobbyIndex** ">  
 </div>  
 </div>

To delete an entry from the FormArray use

* .remove(index) Note: make sure to cast to FormArray otherwise the .removeAt is not regonized. **(<FormArray>**this.recipeForm.get('ingredients')).removeAt(ingredientIndex);

### Custom Validators

To add your own synchronous validator

* Add a validator method:

forbiddenNameValidator(control: FormControl): {[s: string]: boolean}{  
 if (this.forbiddenNames.indexOf(control.value) >= 0) {  
 return {'This name is not allowd': false}  
 } else {  
 return null;  
 }  
 }

* Add your validator to the field you want to validate it with. **Note the bind(this)**. That’s needed because the validator is not called by “this” object but by angular so you have to bind this to this to tell angular where to find the validator.

'username': new FormControl(null, [Validators.required, this.forbiddenNameValidator.**bind(this)**]),

To add an asynchronous validator

* Add a validator which returns a Promise or an Observable:

emailValidator(control: FormControl): Promise<any> | Observable<any> {

const promise = new Promise((resolve, reject) => {  
 setTimeout(() => {   
 if (control.value === 'test@test.com') {  
 console.log('invalid email: ' + control.value);  
 resolve({'invalidEmail': true})  
 } else {  
 resolve(null);  
 }  
 }, 3000);  
 })  
 return promise;  
}

* Add this validator as third parameter to the field  
  'email': new FormControl(null, [Validators.required, Validators.email], this.emailValidator)

### Listen to value / status changes

Every .ts has two observables available to listen to status- and value changes. You can listen to changes on the whole form or parts of (like below on email only)

* Value changes:

this.signupForm.get(**'userData.email**').**value**Changes.subscribe( (value: FormGroup) => {  
 console.log(value);  
 })

* Status changes

this.signupForm.get('**userData**').**status**Changes.subscribe( (value: FormGroup) => {  
 console.log(value);  
 })

### Set- and patch value

Just like in the [template driven approach](#_Setting_a_default) setValue and patchValue are available to give the controls in the for a (initial value)

## Pipes

Pipes can be used to transform the output with changing the original property. Examples of build in pipes

* {{ value | uppercase}}
* {{ geboorteDatum | ful }}

More info about build in pipes: <https://angular.io/api>

You can also build your own pipe by creating a class which implements PipeTransform and is annotated with @Pipe(name: ‘’nameOfYourPipe”) and @Pipe({ name: 'shorten' })… like:

export class ShortenPipe implements PipeTransform{  
 transform(value: string, lenght: number) {  
 if (value.length > lenght) {  
 return value.substring(0, lenght) + '...';  
 } else {  
 return value;  
 }  
 }   
}

More docs about pips can be found on: [https://angular.io/api](https://angular.io/api?query=pipe) and filter on pipe

# HTTP

## Setup

To be able to use Http we have to add the HttpClient to app.module.ts:  
import { HttpClientModule } from '@angular/common/http'

## Post

Sending a post request

this.http.post('https://expenseclaimtracker-test.firebaseio.com//posts.json',  
 postData  
 ).**subscribe**(responseData => {  
 console.log(responseData);  
 });

## Get

Getting (all) posts (for this example it’s about posts in a firebase db)

this.http.get('https://expenseclaimtracker-test.firebaseio.com/posts.json ')  
 .subscribe( response => {  
 console.log(response);  
 })

## Pipe

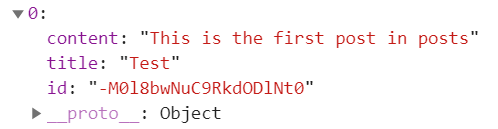
To reformat the response you can add code to the subscribe method. However, it is a good practice to do something with the response before exposing it to .subsribe, using a pipe operator.

### map

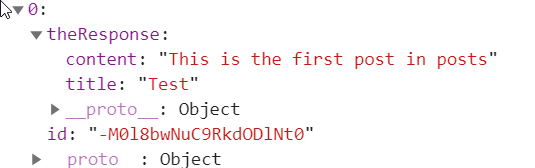
Map is such an operator. map takes the response, reformats it and returns it. After that .subscribe has access to it:

.pipe( map( response => {  
 let postsArray = [];  
 for (const key in response) {  
 postsArray.push({ **...**response[key], id: key})  
 } return postsArray;  
 }))  
Note the spread operator here “**…**”. The spread operator will return all key-value pairs in the respons as single key-value pairs without the parent element.

So:  
this (**with spread**) postsArray.push({ ...response[key], id: key}) returns:



And this postsArray.push({ theResponse: response[key], id: key}) returns:



### Response type

To define the type of the response just add it to the generic method get (or post etc):  
this.http.get**<{ [key: string]: Post }>**

### ErrorHandlinlg

For handling error there are two options

* Adding a second method to .subscribe. As HttpClient is an observalble it supports the second method for errors
* Within the first method of .subscribe, add a pipe with the catchError operator   
  catchError ( errorResponse =>   
   {   
   // do something like logging….  
   throwError(errorResponse); // throwError is an observable  
   }

### catchError

### throwError (observable)

## Define interface for response

To tell angular what the type of the response will be you can add the type to the get, post, put etc. like so:

post<{ name: string }>

## Request headers and params

Request headers can be added to the http request as a second or third param (resp. get or post) within { headers, params }

}

## Response headers

By specifying the part of the response you interested in you can limit the output like so:  
{ observe: 'response' } or { observe: ‘body’ } etc.

### Events => pipe.tap

observe: ‘something’ and then use the pipe(tap something => { } to do something with the response without modifying it. No need to return it. .subscribe will receive the original response anyways.   
In the below example tap is used to determine the type of the response and do something (i.e. inform the user about….\_)

this.http.delete('https://expenseclaimtracker-test.firebaseio.com/posts.json',  
 {  
 observe: **'events'**  
 }).pipe(  
 tap(**events** => {  
 console.log(events);  
 if (events.**type === HttpEventType**.Sent) {  
 console.log('type is Sent');  
 }  
 })  
 );

## Interceptors

To perform some action for (multiple or all) request we use interceptors. They can manipulate the request before sending it (i.e. add a header) or do just do something (i.e. call the log service). It works like this:  
Manupulate request by adding header:

export class AuthInterceptorService implements HttpInterceptor {  
 intercept(req: HttpRequest<any>, next: HttpHandler) {  
 const modifiedRequest = req.clone({  
 headers: req.headers.append('Auth', 'xyz')  
 });  
 return next.handle(modifiedRequest);  
 }  
}

or do some logging using pipe( tap(event => …..

export class LoggingInterceptorService implements HttpInterceptor {  
 intercept(req: HttpRequest<any>, next: HttpHandler) {  
 console.log('Outgoing request');  
 console.log(req.url);  
 console.log(req.headers);  
 return next.handle(req).pipe(  
 tap(event => {  
 if (event.type === HttpEventType.Response) {  
 console.log('Incoming response');  
 console.log(event.body);  
 }  
 })  
 );  
 }

In app.module.ts add the intercepters in the providers section:  
providers: [  
 {  
 provide: HTTP\_INTERCEPTORS,  
 useClass: LoggingInterceptorService,  
 multi: true  
 },  
 {  
 provide: HTTP\_INTERCEPTORS,  
 useClass: AuthInterceptorService,  
 multi: true  
 }  
 ],