Angular Forms 15

Section 15 Forms

Template Drive form (Session_15_Forms/session_15_forms-td-start**)**
171 . Creating form + Registering Controls with NgModule
172. Submit a Form
173. Form object properties
174. Form with @ViewChild
175. Input validation
176. Links for Validators(Reactive approach) + directive validators (Td approach)
177. Using Form State(Submit disable, wrong input message)
178. Validation Error Messages
179. Default values with ngModel property binding
180. 2 way Binding form element
181. Group Form Controls
182. Handling Radio Buttons
183. Set & Patch Form Values
184-185 User the Form data + Reset Form
Reactive Form (Session_15_Forms/session_15_forms-reactive-start)
 187. Reactive Form Setup
188. Creating a form in code
189, Sync the created form with html.
190. Submitting the Form
191. Validating reactive form
192. Get Access to the Form Controls in the html
193. Grouping Controls
194. From Controls Array
195. Creating Custom Validators
196, Using Error codes
197. Creating Custom Async Validator
198. Reacting to status or value changes in the form
199. Setting/Patching/Reset

Template Driven Approach:
Angular infers the Form object from the DOM . For every functionality we modify the template.

Reactive:
Form is created programatically and synchronized with the DOM.

Important angular form should not get submitted:

```
<form action="" method = ""> </form> //Regular form

<form> </form> // Do not have the action to point to some route +  no action method.
```

171 . Creating form + Registering Controls with NgModule

1. First the `FormsModule` should be imported
app.module.ts

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

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

import { FormsModule } from '@angular/forms';

import { HttpModule } from '@angular/http';


import { AppComponent } from './app.component';


@NgModule({

  declarations: [

    AppComponent

  ],

  imports: [

    BrowserModule,

    FormsModule,

    HttpModule

  ],

  providers: [],

  bootstrap: [AppComponent]

})

export class AppModule { }
```

- When FormsModule is imported the <form> tag serve as a selector to Angular to create a Javascript object from it.
- But the forms has to be registered, including all of the desired controlls

2. Adding `ngModel` to the Form input to sign Angular that we would register this control (name attribute is important as well since with this we can select this control)

```
<input

 type="text"

 id="username"

 class="form-control"
```

```
ngModel

name='username'

>
```

172. Submit a Form
- Submit button will start a default javascript submit event, which angular uses.
- This event will be triggered upon clicking submit and call call custom functions onSubmit from the **app.component.ts**

app.component.html

```
<form (ngSubmit)="onSubmit">
```

Place local reference to form html element (#f) and pass it as an argument. (HTMLFormElement)

```
<form (ngSubmit)="onSubmit(f)" #f>
```

Tells angular to give access the create javascript object of this created form.

```
<form (ngSubmit)="onSubmit(f)" #f="ngForm">
```

app.component.ts
- Import `NgForm` as identify the recieved argument as an `NgForm`

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

import { NgForm } from '@angular/forms';      //It has to be imported!

@Component({

  selector: 'app-root',

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

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

})

export class AppComponent {

  suggestUserName() {

    const suggestedName = 'Superuser';

  }

  onSubmit(form: NgForm){
    console.log('form');   // will console log out the actual form object with all of the prope
rties
  }
}
```

173. Form object properties

1. dirty:	The control has been changed
2. touched:	The Control has been touched
3. invalid:	The Control passed through validaiton
4. disabled:	The control is disabled or not

174. Form with `@ViewChild`

- Access a local reference and get as an object
- Another way of accessing the form object, but useful when you want to validate the form before the submit event.

app.component.ts

```
import { Component, ViewChild } from '@angular/core';
import { NgForm } from '@angular/forms';      //It has to be imported!
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  @ViewChild('f') signupForm: NgForm;       //Will help us
  suggestUserName() {
    const suggestedName = 'Superuser';
  }
  onSubmit(){
    console.log(this.signupForm);   // will console log out the actual form object with all of
 the properties
  }
}
```

175. Input validation

- Due to TD approach we have to include all of our validation in the html form. **required** is basic html tag, **email** is a directive which is recognized by Angular
- Will check the validity of the control, with the **valid property** of the control object + adding **ng-valid/ng-invalid** tag to the html
- **pattern="^[1-9]+[0-9]*$**" means greater than zero

app.component.html

```
<input

      type="email"

      id="email"

      class="form-control"

      ngModel

      name="email"

      required

      email

      pattern="^[1-9]+[0-9]*$"

    >
```

176. Links for Validators(Reactive approach) + directive validators (Td approach)

Reactive:	https://angular.io/docs/ts/latest/api/forms/index/Validators-class.html
Td:	https://angular.io/api/forms/Validators

177. Using Form State(Submit disable, wrong input message)

- Disable submit button upon invalid user input
- Remember, we have a local reference `#f` on the html form.

app.component.html

```
          <button

          class="btn btn-primary"

          type="submit"

          [disabled]="f.valid"

          >Submit</button>
```

- We can customize css of the invalid fiels int the using the before mentioned ng-*** added classes by angular.

app.component.css

```
input.ng-invalid.ng-touched{

border: 1px solid red;

}
```

178. Validation Error Messages
- Setting up a local reference with ngModel so we can check the states of the local reference to use it with an *ngIf directive

app.component.html

```
          <div class="form-group">

            <label for="email">Mail</label>

            <input

            type="email"

            id="email"

            class="form-control"

            ngModel

            name="email"

            required

            email

            #email="ngModel"                        //Setting up a local reference so we can qu
ickly check it's stats for the *ngIf directive

            >

            <span class="help-block" *ngIf="!email.valid && email.touched">Please enter a valid
 email</span>

          </div>
```

179. Default values with ngModel property binding

- **no** databinding can be used to set default values to the form **[ngModel]='pet'   --> Single quitation mark ' ... '**
- 1 way property binding can be used as well when we are seting up a **variable (defaultQuestion)** in the **app.component.ts -- > Double quitation mark "..."**

app.component.html

```
        <div class="form-group">

          <label for="secret">Secret Questions</label>

          <select

          id="secret"

          class="form-control"

          [ngModel]='pet'                    // --> Single quitation mark ' ... '

          [ngModel]="defaultQuestion"        //-- > Double quitation mark "..."

          name="secret"

          >

            <option value="pet">Your first Pet?</option>

            <option value="teacher">Your first teacher?</option>

          </select>

        </div>
```

app.component.ts
- 1 way property binding can be used as well when we are seting up a variable (`defaultQuestion`) in the app.component.ts -- > Double quitation mark "..."

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

import { NgForm } from '@angular/forms';

@Component({

  selector: 'app-root',

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

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

})

export class AppComponent {

  @ViewChild('f') signupForm: NgForm;       //Will help us

  defaultQuestion = 'pet';                 //Valid html tag "value"

  suggestUserName() {

    const suggestedName = 'Superuser';

  }

   onSubmit(){
    console.log(this.signupForm);   // will console log out the actual form object with all of
 the properties
  }
}
```

180. 2 way Binding form element
- With two way databining we can get the form's input + us it immediately

app.component.html

```
  <div class="form-group">
     <textarea
     name="questionAnswer"
     rows="3"
     class="form-control"
     [(ngModel)]="answer"></textarea>       //Two way binding using ngModel
  </div>
  <p>Your reply: {{answer}}</p>              // Component variable displayed and updated at every
   keystrike
```

app.component.ts
- Only the answer variable has to be added to the ts file of the component

```
  export class AppComponent {
    answer = '';
  }
```

181. Group Form Controls

- It is possible to group the form controls to have additional information regarding a group of components.
- Just include `ngModelGroup` **="groupName"**

app.component.html

```
  <div
```

```
      id="user-data"

      ngModelGroup="userData"

      #userData = "ngModelGroup">
```

- Has additional form controls - Grouped values

- Also it is possible to pass the grouped controls to a local reference

```
( <span class="help-block" *ngIf="!userData.valid && userData.touched">UserData sections is not
 valid</span> )
```

182. Handling Radio Buttons

- We are making a an array in the app.component.ts, which we will use when we loop through with `*ngFor` on the radio creation. We use property binding to pass the component's variable to the string interpolation with ngModel directive.
app.component.ts

```
export class AppComponent {

  genders = ['male','female'];               //Array to loop through the radio buttons

}
```

app.component.html

```
<div class="radio" *ngFor="let gender of genders">

  <label>

    <input

    type="radio"

    name="gender"

    ngModel

    [value]="gender">

       {{gender}}

  </label>
```

183. Set & Patch Form Values

- Add an click event listener to the button to call `suggestUserName()`
- Since we got an object reference to our form object from angular as **signupForm** we can use the following two data modificaiton techniques
   - 1. `setValue` () ---> Expects an object with values {}   -> downside, it will modify the whole form no matter if the form has been previously filled or not
   - 2. `patchValue` () ---> Expects an object with values {}   -> It will modify only the given form value

```
    <button

    class="btn btn-default"

    type="button"

    (click)="suggestUserName()"

    >Suggest an Username</button>
```

```
  export class AppComponent {

    @ViewChild('f') signupForm: NgForm;       //Will help us

    defaultQuestion = 'pet';                  //Valid html tag "value"

    answer = '';                              //Storing the two way binding value

    genders = ['male','female'];               //Array to loop through the radio buttons

    suggestUserName() {

      const suggestedName = 'Superuser';

      // this.signupForm.setValue({                //1.

      //   userData:{

      //     username: suggestedName,

      //     email:''

      //   },

      //   secret:'pet',

      //   questionAnswer: '',

      //   gender:'male',

      //   })

      this.signupForm.form.patchValue({           //2.

        userData:{

          username: suggestedName
      }

    });

  }
```

184-185 User the Form data + Reset Form
- 1. Create a user object in the app.component.ts to store the values,
- 2 Track if the form has been submitted, so we can use ***ngIf** directive
- 3 onSubmit() gather the data from the created javascript object `signupForm`
- 4 **this.signupForm.reset();** will reset the form values + the ng classes made by angular + if you pass the same object as to setValue you can se

```
export class AppComponent {

  @ViewChild('f') signupForm: NgForm;

  defaultQuestion = 'teacher';

  answer = '';

  genders = ['male', 'female'];

  user = { username: '', email: '', secretQuestion: '', answer: '', gender: '' };        //1 Cr
eating an empty user object

  submitted = false;                                                                   //2 Ta
cking if the form has been submitted or not

  suggestUserName() {

    this.signupForm.form.patchValue({userData: { username: suggestedName}  });

  }

  onSubmit() {

    this.submitted = true;                                                             //2

    this.user.username = this.signupForm.value.userData.username;      //3 userdata is a differ
ent group of values so a diffferent object as well

    this.user.email = this.signupForm.value.userData.email;

    this.user.secretQuestion = this.signupForm.value.secret;

    this.user.answer = this.signupForm.value.questionAnswer;

    this.user.gender = this.signupForm.value.gender;

    this.signupForm.reset();

  }

}
```

app.component.ts (display the result if submitted ="true" with string interpolation)

```
  <div class="row" *ngIf="submitted">

    <div class="col-xs-12">

      <h3>Your Data</h3>

      <p>Username: {{ user.username }}</p>

      <p>Mail: {{ user.email }}</p>

      <p>Secret Question: Your first {{ user.secretQuestion }}</p>

      <p>Answer: {{ user.answer }}</p>

      <p>Gender: {{ user.gender }}</p>

    </div>

  </div>
```

Reactive Form

187. Reactive Form Setup

In the **app.component.ts** we create a wrapper, for the form as a `FormGroup`

```


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

import { FormGroup} from '@angular/forms';


@Component({

  selector: 'app-root',

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

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

})

export class AppComponent {

  genders = ['male', 'female'];

  signupForm: FormGroup;


}
```

Also at the app.module.ts we have to include the new library for "ReactiveFormsModule" from **'@angular/forms'**.

- FormsModule --> Template Driven Approach
- ReactiveFormsModule --> Reactive Approach

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

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

// import { FormsModule } from '@angular/forms';

import {ReactiveFormsModule} from '@angular/forms';

import { HttpModule } from '@angular/http';


import { AppComponent } from './app.component';


@NgModule({

  declarations: [

    AppComponent

  ],

  imports: [

    BrowserModule,

    // FormsModule,

    ReactiveFormsModule

    HttpModule

  ],

  providers: [],

  bootstrap: [AppComponent]

})

export class AppModule { }
```

188. Creating a form in code

App.component.ts
- Importing OnInit lifecycle hook from angular core, so at the initialization of the component the form creation will run
- Import FormControl as well since it is a part of the FormGroup object, with default values, properties and methods.
- **FormGroup** is an angular form objects which holds **key value form Control pairs**

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

import { FormGroup, FormControl } from '@angular/forms';

@Component({

  selector: 'app-root',

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

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

})

export class AppComponent implements OnInit {

  genders = ['male', 'female'];

  signupForm: FormGroup;


  ngOnInit(){

    this.signupForm = new FormGroup({        //FormGroup is an angular form objects which holds
 key value form Control pairs

      'username': new FormControl(null),      //Inital state is null now

      'email': new FormControl(null),

      'gender': new FormControl('male')

    })

  }

}
```

189, Sync the created form with html.
- So here we use propertybinding to determine the Form **formGroup** property, tell angular that it should be connected to the created form in the **app.component.ts**
- //Connecting the created `formControl` to the html, with the defined names in the app.component.ts

```
<div class="container">

  <div class="row">

    <div class="col-xs-12 col-sm-10 col-md-8 col-sm-offset-1 col-md-offset-2">

      <form [formGroup]="signupForm">                        //  Connecting the created form in
 app.component.ts to the html

        <div class="form-group">

          <label for="username">Username</label>

          <input
```

```
            type="text"

            id="username"

            formControlName = "username"                        //Connecting the created formCon

trol to the html

            class="form-control">

        </div>

        <div class="form-group">

          <label for="email">email</label>

          <input

            type="text"

            id="email"

            formControlName  = "email"

            class="form-control">

        </div>

        <div class="radio" *ngFor="let gender of genders">

          <label>

            <input

              type="radio"

              formControlName = "gender"

              [value]="gender">{{ gender }}

          </label>

        </div>

        <button class="btn btn-primary" type="submit">Submit</button>

      </form>

    </div>

  </div>

</div>
```

190. Submitting the Form
- We just has to call the onSubmit method o the submit event, and since we created the form programatically, we can reference it's controls, the form itself really easily just by typing this.signupForm

app.component.html

```
<form [formGroup]="signupForm" (ngSubmit)="onSubmit()">

</form>
```

app.component.ts

```
 onSubmit(){

     console.log(this.signupForm);                  //Since we created this form as new FormGr

oup we can reference it as a component'S variable

 }
```

191. Validating reactive form
- In the template driven approach you just have to include the requried + email keyword into the input
- But in the reactive driven you have to pass in a validator or an array of validators in the `FormControl` **('default value', [Validators])**
html

```
  <input   type="email"   id="email"  class="form-control" gModel name="email"   required   ema

il        #email="ngModel"       >
```

component

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

import { FormGroup,FormControl,Validators} from '@angular/forms';



@Component({

  selector: 'app-root',

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

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

})

export class AppComponent implements OnInit {

  genders = ['male', 'female'];

  signupForm: FormGroup;



  ngOnInit(){

    this.signupForm = new FormGroup({
      'username': new FormControl(null, Validators.required),      //Inital state is null now

      'email': new FormControl(null,[Validators.required,Validators.email]),

      'gender': new FormControl('male')

    });

  }

  onSubmit(){

  console.log(this.signupForm);

  }

}
```

192. Get Access to the Form Controls in the html
(FormName.get('**FormControlName**').**property)**

- signupForm.get('username').valid
- signupForm.get('username').touched

```
   <span class="help-block" *ngIf="!signupForm.get('username').valid && signupForm.get('usernam

  e').touched">Please enter a valid usernamel</span>
```

+ since angular adding the following classes to the html, specificy css can be used as well

- ng-valid
- ng-touched

```
  input.ng-invalid.ng-touched{

  border: 1px solid red;

  }
```

193. Grouping Controls
- new `FormGroups` can be created in the typscript file, which also has to be reflected in the html, both in the strucutre + with the **`signupForm.get()`**
- With `formGroupName` you can synch it with the html

app,component.ts

```
    ngOnInit(){

      this.signupForm = new FormGroup({
```

```
      'userData': new FormGroup({
          'username': new FormControl(null, Validators.required),      //Inital state is null n
ow
          'email': new FormControl(null,[Validators.required,Validators.email]),
      }),
      'gender': new FormControl('male')
    });
  }
```

app.component.html
- Using formGroupName to synch ts code with html, also the division is grouped as it is in the ts file
- The access control route has to be changed as well. **signupForm.get('userData.username').valid**

```
<div formGroupName="userData">
          <div class="form-group">
            <label for="username">Username</label>
            <input
              type="text"
              id="username"
              formControlName = "username"
              class="form-control">
          </div>
          <div class="form-group">
             <span class="help-block" *ngIf="!signupForm.get('userData.username').valid && sign
upForm.get('userData.username').touched">Please enter a valid usernamel</span>
            <label for="email">email</label>
            <input
              type="text"
              id="email"
              formControlName = "email"
              class="form-control">
          </div>
       </div>
```

194. From Controls Array, create elements + delete them

- So we are importing the `FormArray` control, with which we can create dynamic array of controls.
-// with (<FormArray>) we have to tell angular that this is an array of controls named as hobbies
- Adding an click event listener to the html we can combine the event to the form creation

app.component.ts

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

import { FormGroup,FormControl,Validators,FormArray} from '@angular/forms';



@Component({

  selector: 'app-root',

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

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

})

export class AppComponent implements OnInit {

  genders = ['male', 'female'];

  signupForm: FormGroup;

  ngOnInit(){

    this.signupForm = new FormGroup({

      'userData': new FormGroup({

      'username': new FormControl(null, Validators.required),      //Inital state is null now

      'email': new FormControl(null,[Validators.required,Validators.email]),

      }),

      'gender': new FormControl('male'),

      'hobbies': new FormArray([])                    //We create an empty array of controls

    });

  }

  onSubmit(){

  console.log(this.signupForm);

  }


  onAddHobby(){
    const control = new FormControl(null, Validators.required);                //The users crea
tes the hobby
    (<FormArray> this.signupForm.get('hobbies')).push(control);                 // with (<FormA
rray>) we have to tell angular that this is an array of controls named as hobbies
    //(<FormArray> this.signupForm.get('hobbies')).removeAt(index);
  }


}
```

app.component.html
- `formArrayName` tells angular that there is a connection between the component's control + the html
- With the *ngFor we can loop through the array of controls, by creating indexes
- This indexes will be the names for the **formControlName**s

```
 <div formArrayName = "hobbies">
          <h4>YourHobbies</h4>
          <button
          class="btn btn-default"
          type="button"
          (click)="onAddHobby()">Add a hobby</button>
          <div
          class="form-group"
          *ngFor="let hobbyControl of signupForm.get('hobbies').controls; let i=index">
            <input class="form-control" type="text"  [formControlName]="i">
            // <!--We are using propert binding since we are not passing a string, but creating
 a local variable in the ngFor loop-->
          </div>
```

Value object what we get.

195. Creating Custom Validators

- Custom validator is just a function, which is added to the Validators array and get executed once an input is touched
- We want to recieve a **FormControl object** and output an object we key value pairs, which value should be a boolean,
- **If the validator** is valid then you should **return** nothing or **null**!
- **Be careful using `this` with Validators!**

```
forbiddenUserNames = ['Chris','Anna'];

    ngOnInit(){

    this.signupForm = new FormGroup({

      'userData': new FormGroup({

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


      //since we are calling the FormControl with the new keyword, this is reffering to the fom
rControl, but if we bind it ot the 'this, we can refere to the component'

      'email': new FormControl(null,[Validators.required,Validators.email]),

      }),

      'gender': new FormControl('male'),

      'hobbies': new FormArray([])

    });

  }

  //Custom validator is just a function

  // We want to recieve a FormControl object and output an object we key value pairs, which val
ue should be a boolean

  forbiddenNames(control:FormControl):{[s:string]:boolean}{

    if(this.forbiddenUserNames.indexOf(control.value) !==-1){

    //If we do not find the item, it is returning -1, so the inout is valid

      return {'forbiddenname':true}           //If the validator is valid then you should retur
n nothing or null!

    }

    return null;

  }

}
```

196, Using Error codes
- We can use error codes to specify the error messages

app.component.ts:
- This way if we wrap the error message ngIfs into the outer validation, the customized message based on error codes will work

```
  <span
    *ngIf="!signupForm.get('userData.username').valid && signupForm.get('userData.username').to
uched" class="help-block">
    <span *ngIf="signupForm.get('userData.username').errors['nameIsForbidden']">This name is in
valid!</span>
    <span *ngIf="signupForm.get('userData.username').errors['required']">This field is require
d!</span>
  </span>
```

- If we do not wrap them, they will fail all the time!

```
<span *ngIf="!signupForm.get('userData.username').valid && signupForm.get('userData.username').
touched" class="help-block">    </span>           <span *ngIf="signupForm.get('userData.usernam
e').errors['nameIsForbidden']">This name is invalid!</span>
<span *ngIf="signupForm.get('userData.username').errors['required']">This field is required!</s
pan>
```

197. Creating Custom Async Validator
(Waiting for serverinput, Promise)
- Observable has to be imported
- We are using the third argument of the FormControl --> // FormControl(default value, validators, **async Validators**)
- We created a new validator function called `forbiddenEmails` which takes a **FormControl** as an argument and will return a **promise** or an **observable** since the wrapped inner function is an async function
app.component.ts

```
import { Component,OnInit } from '@angular/core';
import { FormGroup,FormControl,Validators,FormArray} from '@angular/forms';
import {Observable} from 'rxjs/Observable';


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

```
export class AppComponent implements OnInit {
  genders = ['male', 'female'];
  signupForm: FormGroup;
  forbiddenUsernames = ['Chris','Anna'];

  ngOnInit(){
    this.signupForm = new FormGroup({
      'userData': new FormGroup({
      'username': new FormControl(null, [Validators.required,this.forbiddenNames.bind(this)]),
      //since we are calling the FormControl with the new keyword, this is reffering to the fomrControl, but if we bind it ot the 'this, we can refere to the component'
      // FormControl(default value, validators, async Validators)
      'email': new FormControl(null,[Validators.required,Validators.email],this.forbiddenEmails.bind(this)),
      }),
      'gender': new FormControl('male'),
      'hobbies': new FormArray([])
    });
  }
  onSubmit(){
  console.log(this.signupForm);
  }
  onAddHobby(){
    const control = new FormControl(null, Validators.required);                //The users creates the hobby
    (<FormArray> this.signupForm.get('hobbies')).push(control);
  }
    forbiddenNames(control: FormControl): {[s: string]: boolean} {
    if (this.forbiddenUsernames.indexOf(control.value) !== -1) {
      return {'nameIsForbidden': true};
    }
    return null;
  }
  //Takes the control as an argument, it will be a promise or an observable, which wraps anythi
ng,will handle async data

  // We are wrapping the setTimeout in a promise, which will resolve an object {} or a null if
 the functions is successfully executed

  //Since the setTimeout function wil never fail, it will never call the reject

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

     const promise = new Promise<any>((resolve,reject)=>{

       setTimeout(()=>{

         if(control.value === "test@test.com"){

           resolve({ 'emailIsForbidden':true})

         } else{

           resolve(null)

         }

       },1500)

     });

     return promise

   }

}
```

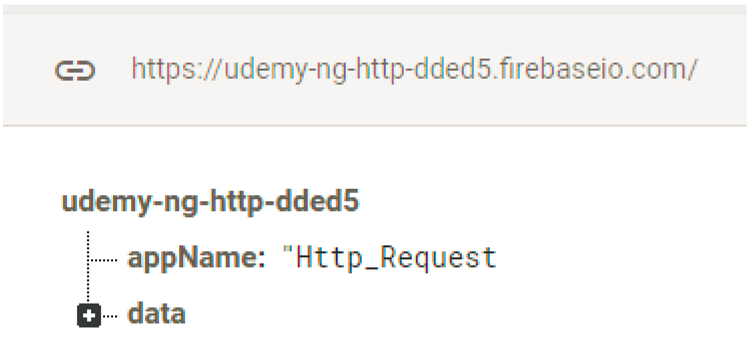
198. Reacting to `status` or `value` changes in the form

- //Whenever valuechanges, the whole form.value will be printed out as object
- //WheneveR the status will change it will be **VALID/INVALID/PENDING**

```
    //These are observables, for which we can subscribe, listen to them

    //Whenever valuechanges, the whole form.value will be printed out as object

      this.signupForm.valueChanges.subscribe(

       (value)=>{console.log(value)}

      );

  //Wheneve the status will change it will be VALID/INVALID/PENDING

    this.signupForm.statusChanges.subscribe(
      (value)=>console.log(value)

      );
```

Status

Value


https://udemy-ng-http-dded5.firebaseio.com/

udemy-ng-http-dded5
appName: "Http_Request
data

199. Setting/Patching/Reset
-We can prepopulate data with setValue, or we can prepopulate a given formControl
- We can reset the whole form and we can pass an object to which we would like to reset the form.

```
//Prepopulate every fomrControl

this.signupForm.setValue({

      'userData': {

        'username': 'Max',

        'email': 'max@test.com'

      },

      'gender': 'male',

      'hobbies': []

    });

//Prepopulate only selected formControl

    this.signupForm.patchValue({

      'userData': {

        'username': 'Anna',

      }

    });

    //- We can reset the whole form and we can pass an object to which we would like to reset t
he form.
```

```
    this.signupForm.reset()
```