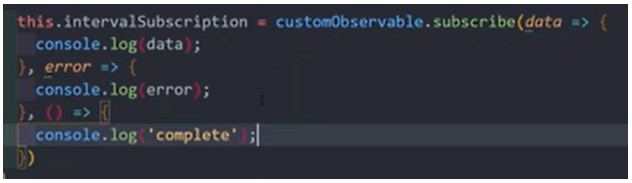
Angular Part - 2

…continuing part-1 ……. Custom observables

Catch Errors & Complete in Rxjs custom observable using observer.error and complete



Understand Rxjs Operators in the observables before sending to the subscribe data

Before processing the data into subscriber, operators will receive data, manipulate data and then forwarded to the subscriber


```
this.mySubscription = customObservable.pipe(map(data => { return `amazing count ${data}`})).subscribe(data => {
  console.log(data);
})
```

Multiple operators can be chained

Understand the Subjects in Rxjs angular. Implement the Subject for cross-component communication

Above is create subject and publish to subject
Below is subscribe for that specific subject and unsubscribe when leaves
Performance friendly instead of events and emits
Allows cross component state management

Form Handling

Angular provides two primary methods for handling forms: Reactive Forms and Template-Driven Forms. A new, experimental Signal Forms approach is also available as of Angular v21.

1. Reactive Forms

Reactive forms follow a model-driven approach, where the form logic and structure are defined primarily within the component class, offering greater control and flexibility. This method is best for complex, dynamic forms with intricate validation logic.

Key classes and methods:

- **FormControl**: Manages the value and validation status of an individual input field (e.g., <input>, <select>).
- **FormGroup**: Collects instances of FormControl or other FormGroup instances into a single object, tracking the status of the entire group.
- **FormArray**: Manages an array of FormControl or FormGroup instances, ideal for dynamic lists of inputs.
- **FormBuilder**: A service that provides convenient factory methods (control(), group(), array()) to generate form controls, reducing repetitive boilerplate code.
- **setValue**() and **patchValue**(): Methods used to programmatically update the value of a form control or group.
- **valueChanges observable**: An observable stream that emits the current value of the form every time it changes, allowing for real-time reactions and data flow management.
- **onSubmit**() method: A function in the component class bound to the form's (ngSubmit) event to process the form data after submission.

2. Template-Driven Forms

Template-driven forms manage most of the form logic directly within the HTML template using directives. This approach is simpler and suitable for basic forms like login or contact forms with straightforward logic.

Key directives and methods:

- **FormsModule**: This module must be imported to use template-driven forms.
- **NgForm**: A directive automatically bound to the <form> tag, creating a top-level FormGroup instance behind the scenes and tracking the form's overall value and validation status.
- **ngModel**: A directive that provides two-way data binding ([(ngModel)]) between the form input element and a property in the component class. It also creates a FormControl instance for the element it's on.
- **ngSubmit** event: Used to bind a component method to the form's submission event.
- **Template reference variables**: Used with the #name="ngModel" syntax to access the form control's state (e.g., valid, invalid, dirty, touched) within the template for validation feedback

Template Driven Forms in Angular. Get NgForm Object from the template to code

Modal file:

Using ngMode and name attribute on inputs
ngSubmit on form element
Create local reference - ngForm for the form element and pass it to the submit method

```
<div class="bg-green-200 rounded-xl border border-gray-400">
    <div class="bg-green-200 rounded-xl">
        <!-- <p class="rounded-t-xl text-center py-3" appRedText>Service portal component</p> -->
        <div class="p-5">
            <div>
                <div class="">
                    <P>Add Product Page</P>
                </div>
                <div class="flex flex-col gap-2 w-[30%] mt-5 text-[18px] rounded-lg p-5 shadow-md bg-white">
                    <form (ngSubmit)="onFormSubmit(addproductform)" #addproductform="ngForm" class="flex flex-col gap-2">
                        <div class="flex flex-row justify-between">
                            <label for="username">User Name</label>
                            <input
                                type="text"
                                class="w-[60%] px-2 border border-gray-800 rounded-md"
                                id="username"
                                name="username"
                                ngModel
                            />
                        </div>

                        <div class="flex flex-row justify-between">
                            <label for="email">Email</label>
                            <input
                                type="text"
                                class="w-[60%] px-2 border border-gray-800 rounded-md"
                                id="email"
                                name="email"
                                ngModel
                            />
                        </div>
                        <div class="flex flex-row justify-between">
                            <label for="gender">Gender</label>
                            <select
                                class="w-[60%] border border-gray-800 rounded-md"
                                id="gender"
                                name="gender"
                                ngModel
                            >
                                <option value="male">Male</option>
                                <option value="female">Female</option>
                            </select>
                        </div>
                        <button
                            class="mt-2 px-8 py-2 rounded-lg cursor-pointer font-semibold text-white bg-linear-65 from-blue-500 to-"
                        >
                            Add Product
                        </button>
                    </form>
                </div>
            </div>
        </div>
    </div>
</div>
```

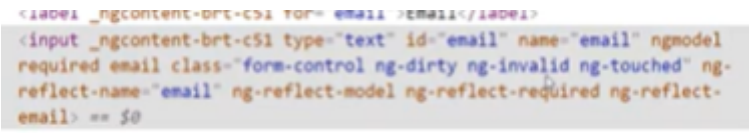
Controller file:

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

@Component({
  selector: 'app-add-product.page',
  imports: [FormsModule],
  templateUrl: './add-product.page.html',
  styleUrl: './add-product.page.css',
})
export class AddProductPage {

  onFormSubmit(form: NgForm) {
    console.log(form);
  }
}
```

Using @viewChild - no need to pass data across on submit method

```
export class AddProductPage {

  @ViewChild("addproductform") adproductForm!: NgForm;

  onFormSubmit() {
    console.log(this.adproductForm);
  }
}
```

This gives us form Object containing set of information of related to the form (ex:- dirty, touched, error, value, valid.. And many more)



Validations

These classes added automatically for the input elements

```
import { clsx, type ClassValue } from 'clsx';
import { twMerge } from 'tailwind-merge';

export function cn(...args: ClassValue[]) {
    return twMerge(clsx(args));
}
//!-- [class]="cn('p-4', isActive ? 'bg-blue-500' : 'bg-gray-200', 'text-lg font-semibold')"
-->
```

```
import { Component, ViewChild } from '@angular/core';
import { FormsModule, NgForm } from "@angular/forms";
import { cn } from '../../../../shared/utils/cn.util';

@Component({
  selector: 'app-add-product.page',
  imports: [FormsModule],
  templateUrl: './add-product.page.html',
  styleUrl: './add-product.page.css',
})
export class AddProductPage {

  @ViewChild("addproductform") adproductForm!: NgForm;

  buttonClasses = cn(
    "mt-2 px-8 py-2 rounded-lg font-semibold text-white transition-all duration-300",
    !this.adproductForm?.valid ? 'cursor-not-allowed bg-slate-300' : 'cursor-pointer
bg-linear-65 from-blue-500 to-blue-900 hover:bg-linear-65 hover:from-blue-700
hover:to-blue-950'
  );

  onFormSubmit() {
    console.log(this.adproductForm.valid);
  }
}
```

```
                    <button
                        [class]="buttonClasses"
                        type="submit"
                        [disabled]="!adproductForm?.valid"
                    >
                        Add Product
                    </button>
```

```
<div class="form-group">
    <label for="email">Email</label>
    <input type="text" class="form-control" id="email" name="email" ngModel required
    <span class="help-text"
        *ngIf="f.controls.email && !f.controls.email.valid && f.controls.email.touche
        valid
        email</span>
```

ngModel=”male” default value for the select input

```
<label for="gender">Gender</label>
<select class="form-control" id="gender" name="gender" ngModel="male">
    <option value="male">Male</option>
    <option value="female">Female</option>
</select>
```

Control this via variable for (data binding) sync selected value

NgModelGroup - Grouping The Form Controls in Template Driven Forms using ngModelGroup

```
<div ngModelGroup="userData">
    <div class="form-group">
        <label for="username">Use
        <input type="text" class=
    </div>

    <div class="form-group">
        <label for="email">Email<
        <input type="text" class=
            #email="ngModel" />
        <span class="help-text" *
            valid
            email</span>
    </div>
</div>
```

“userData” object will be created inside value object of form object

```
<div *ngIf="!userdata.valid && userdata.touched">User Data is Invalid<
<div ngModelGroup="userData" #userdata="ngModelGroup">
    <div class="form-group">
        <label for="username">User Name</label>
        <input type="text" class="form-control" id="username" name="us
    </div>

    <div class="form-group">
        <label for="email">Email</label>
        <input type="text" class="form-control" id="email" name="emai
            #email="ngModel" />
        <span class="help-text" *ngIf="!email.valid && email.touched">
            valid
            email</span>
    </div>
```

Set Value and Patch Value for populating Form Elements in the Template Driven Forms

```
fillValues() {
  this.signUpForm.form.setValue({
    userData: {
      email: 'leela@leela.com',
      name: 'Leela'
    },
    gender: 'male',
    about: 'About Us'
  })
}
```

This method can be bind to button click. It will fill the entire form at once. This object must be same as the structure of formObject value property

```
fillValues() {
  this.signUpForm.form.patchValue({
    userData: {
      email: 'leela@leela.com',
      username: 'Leela'
    },

  })
}
```

filling only the portion of the form

Get and Reset the Form Data controls in the Template Driven Forms

```
this.signUpForm.reset()
```

Introduction to Reactive Forms Approach. Create FormGroup and FormControl with code

ReactiveFormsModule -> import in app module

New form

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

```
<form>
    <div class="form-group">
        <label>UserName</label>
        <input type="text" class="form-control">
    </div>
    <div class="form-group">
        <label>Email</label>
        <input type="text" class="form-control">
    </div>

    <div *ngFor="let gender of genders">
        <label>
            <input type="radio" name="gender" [value]="gender" />{{gender}}
        </label>
    </div>

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

</form>
```

```
@Component({
  selector: 'app-reactive-forms',
  templateUrl: './reactive-forms.component.html',
  styleUrls: ['./reactive-forms.component.css']
})
export class ReactiveFormsComponent implements OnInit {
  genders = ['male', 'female'];
  signUpForm: FormGroup;
  constructor() { }

  ngOnInit(): void {
    this.signUpForm = new FormGroup({
      'username': new FormControl(null),
      'email': new FormControl(null),
      'gender': new FormControl('female')
    })
  }

}
```

```
<input type="text" class="form-control" formControlName="username">
```

This form control name should be same as what we provide in ts file

Validations

when getting errors angular automatically adding some classes Html elements. Can check with inspect

```
<label>UserName</label>
<input type="text" class="form-control" formControlName="username">
<span class="help-block">Please enter valid username</span>
```

```
<div class="form-group">
    <label>UserName</label>
    <input type="text" class="form-control" formControlName="username">
    <span class="help-block"
        *ngIf="!signUpForm.get('username').valid && signUpForm.get('username').touched">Please enter
        valid username</span>
</div>
<div class="form-group">
    <label>Email</label>
    <input type="text" class="form-control" formControlName="email">
    <span class="help-block"
        *ngIf="!signUpForm.get('email').valid && signUpForm.get('email').touched">Please enter valid
        Email</span>
</div>
```

```
export class ReactiveFormsComponent implements OnInit {
  genders = ['male', 'female'];
  signUpForm: FormGroup;
  constructor() { }

  ngOnInit(): void {
    this.signUpForm = new FormGroup({
      'username': new FormControl(null, Validators.required),
      'email': new FormControl(null, [Validators.required, Validators.email]),
      'gender': new FormControl('female')
    })
  }
```

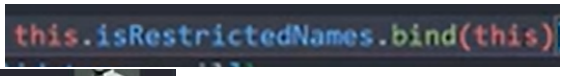
```
input.ng-invalid.ng-touched {
    border: 1px solid red;
}
```

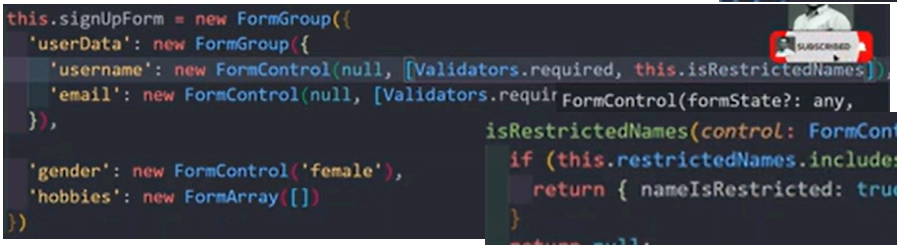
Grouping the Controls in the Reactive Forms using FormGroupName

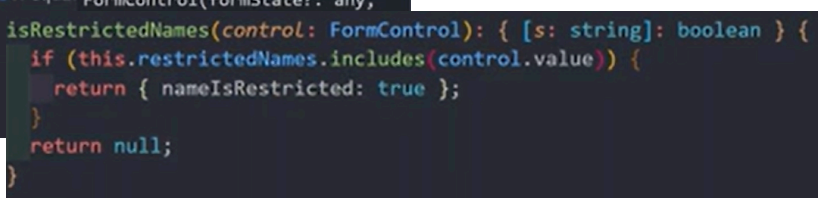
Dynamically Add Form Controls with FormArray FormArrayName in the Reactive Forms

Create Custom Validations for the reactive Forms

this.restrictedNames is a plain array of names







Validator function returning key is use full for error handling, it will treated as the name of raised error

```
<label>UserName</label>
<input type="text" class="form-control" formControlName="username">
<span class="help-block"
    *ngIf="!signUpForm.get('userData.username').valid && signUpForm.get('userData.user
    <span *ngIf="signUpForm.get('userData.username').errors.required">Username is
        required</span>
    <span *ngIf="signUpForm.get('userData.username').errors.required">Username is not
        valid</span>
</span>
```

Create a Custom Asynchronous Validator in the Reactive Forms

Testing a promise instead of an actual network request

```
isRestrictedEmails(control: FormControl): Promise<any> | Observable<any> {
  let promise = new Promise((resolve, reject) => {
    setTimeout(() => {
      if (control.value === 'test@test.com') {        You, a few seconds ago
        resolve({ emailIsRestricted: true });
      } else {
        resolve(null);
      }
    }, 2000)
  });
  return promise;
```

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

Asynchronous validator/validator array should placed as the third parameter for the form control
And must be use bind(this) if validator function using “this” instance

Ng-pending class will be automatically added to the html element during the loading time.

Explore StatusChanges, ValueChanges, SetValue, PatchValue, and reset in Reactive Forms

```
ngOnInit(): void {
  this.signUpForm = new FormGroup({
    'userData': new FormGroup({
      'username': new FormControl(null, [Validators.re
      'email': new FormControl(null, [Validators.requi
    }),

    'gender': new FormControl('female'),
    'hobbies': new FormArray([])
  });

  this.signUpForm.valueChanges.subscribe(value => {
    console.log(value);
  })      You, 2 days ago • dasdasd
}
```

```
this.signUpForm.setValue({
  userData: {
    username: 'Hai Leela',
    email: 'asad@asd.com'
  },
  gender: 'male',
  hobbies: []
})
```

```
this.signUpForm.patchValue({
  userData: {
    username: 'Hai Leela',
  },
  gender: 'male',
  hobbies: []
})
```

```
this.signUpForm.reset()
```

Pipes

In Angular, pipes are functions used in templates to transform data for display without altering the original data source. They are denoted by the pipe (|) symbol within interpolation expressions. Angular offers built-in pipes for common transformations and allows for the creation of custom pipes.

Using Pipes

Pipes are applied within template expressions using the vertical bar (|) character.

```
{{ value | pipeName }}
```

```
<p>{{ "We think you are doing great!" | uppercase }}</p>
<!-- Output: WE THINK YOU ARE DOING GREAT! -->
```

Pipes can accept parameters and be chained together, with the output of one pipe becoming the input for the next.

```
<p>{{ presentDate | date: 'fullDate' | uppercase }}</p>
<!-- Output: TUESDAY, JANUARY 7, 2025 (example date) -->
```

Built-in Pipes

Angular provides a variety of built-in pipes for typical data formatting tasks.

- **AsyncPipe**: Unsubscribes automatically and displays the latest value emitted by a Promise or an Observable, preventing memory leaks.
- **CurrencyPipe**: Formats a number as a currency string (e.g., $100.00).
- **DatePipe**: Formats a date value according to locale rules (e.g., Apr 15, 1988).
- **DecimalPipe**: Transforms a number into a string with a decimal point and specific digit options.
- **JsonPipe**: Converts a JavaScript object to a JSON string representation, useful for debugging.
- **LowerCasePipe**: Transforms text to all lowercase.
- **PercentPipe**: Transforms a number to a percentage string (e.g., 88.9%).
- **SlicePipe**: Creates a subset of an array or string.
- **TitleCasePipe**: Transforms text to title case.
- **UpperCasePipe**: Transforms text to all uppercase.

To use a built-in pipe, you must import it into the relevant component's imports array or an associated NgModule.

Custom Pipes

For transformations beyond the built-in options, you can create custom pipes. A custom pipe is a class decorated with @Pipe and implements the PipeTransform interface, which requires a transform method to define the logic.
You can generate one using the Angular CLI command:
ng generate pipe <pipe-name>

Pure vs. Impure Pipes

Pipes are pure by default. A pure pipe executes only when its input value or object reference changes (e.g., assigning a completely new array to a variable). They are highly performant because Angular avoids unnecessary recalculations.

Impure pipes are executed during every change detection cycle, regardless of whether the input data has changed. This allows detection of changes within mutable objects (like pushing an item into an existing array), but comes with a performance penalty and should be used cautiously. The AsyncPipe is an example of a built-in impure pipe.

Pipes in Angular. Chaining Multiple Pipes, parameterized Pipes

```
<div *ngFor="let user of users" class="my-3">
    <div>Name: {{user.name | uppercase}}</div>
    <div>Joined Date: {{user.joinedDate | date}}</div>
</div>
```

Passing parameters(can multiple parameters also) to pipes

```
<div>Joined Date: {{user.joinedDate | date:'fullDate'}}</div>
```

Hiring multiple pipes

```
<div>Joined Date: {{user.joinedDate | date:'fullDate' | uppercase}}</div>
```

Create a custom Pipe and pass parameters to the Pipe

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

@Pipe({
    name:"truncate"
})
export class TruncatePipe implements PipeTransform{
    transform(value: any) {
        return value.substr(0, 5);
    }
}
Import this pipe in ngModule or usage component ts file
Ow to use in templates :-  <P>{{ pageName | truncate }}</P>
```

```
transform(value: any, limit: number) {
    if (value.length > limit) {
        return value.substr(0, limit) + ' ...';
    }
    return value;
}
```

Accepting params

```
<div>Name: {{user.name | shorten:5}}</div>
```

Creating Filter Pipe. Filter the list of data with search string

```
<div *ngFor="let user of users | filter:filteredString" class="my-3">
    <div>Name: {{user.name | shorten:10}}</div>
    <div>Joined Date: {{user.joinedDate | date:'fullDate' | uppercase}}</div>
</div>
```

```
  name: 'filter'
})
export class FilterPipe implements PipeTransform {

  transform(value: any, filterString: string) {
    if (value.length === 0) {
      return value;
    }

    const users = [];
    for (const user of value) {
      if (user['name'] === filterString) {
        users.push(user);
      }
    }
    return users;
  }
```

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

Pure and Impure pipes

Understanding the async (Asynchronous) Pipe

```
export class FilterPipesComponent implements OnInit {

  appStatus = new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve('Users Data Received');
    }, 3000);
  })
```

```
<div class="my-2">
    {{appStatus | async}}
</div>
```

display only when got the resolved value (promise, observable, subscription any item)

Promise created for simulate asynchronous task

HTTP Request

In Angular, you make HTTP requests using the built-in HttpClient service from the @angular/common/http package. This service utilizes RxJS Observables to handle asynchronous operations and provides features like typed responses, streamlined error handling, and request/response interception.

Step 1: Configure HttpClient

Depending on your Angular version and setup (standalone or module-based), you need to make HttpClient available via dependency injection.

- For Angular v17+ Standalone APIs (recommended): Add provideHttpClient() to the providers array in your app.config.ts or component providers.

```
import { provideHttpClient } from '@angular/common/http';
// In app.config.ts (or component providers array)
providers: [provideHttpClient()]
```

- For Module-based Angular applications (older versions): Import HttpClientModule in your root application module (app.module.ts).

```
import { HttpClientModule } from '@angular/common/http';
// ...
@NgModule({
  imports: [BrowserModule, HttpClientModule],
  // ...
})
export class AppModule { }
```

Step 2: Create a Service (Best Practice)

It is a best practice to centralize data access logic in a service rather than components.

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class DataService {
  private apiUrl = 'https://api.example.com/data'; // Replace with your API endpoint

  constructor(private http: HttpClient) { } // Inject the HttpClient

  // Define methods for different HTTP requests
}
```

Step 3: Make HTTP Requests

Inject your DataService into a component and call its methods. Remember that the methods return Observables, so you must subscribe to them to initiate the request and receive the response.

Common HTTP Methods

- **GET** (Retrieve data): Used to fetch data from the server.

```
// In data.service.ts
getData(): Observable<any[]> {
  return this.http.get<any[]>(this.apiUrl); // Use a generic type for type safety
}

// In your component.ts
ngOnInit(): void {
  this.dataService.getData().subscribe(data => {
    console.log(data);
    // Assign data to a component property to display in the template
  });
}
```

- **POST** (Create data): Used to send data to the server.

```
// In data.service.ts
createData(newData: any): Observable<any> {
  return this.http.post<any>(this.apiUrl, newData);
}

// In your component.ts (e.g., in a form submission handler)
submitData(payload: any): void {
  this.dataService.createData(payload).subscribe(response => {
    console.log('Created:', response);
  });
}
```

- **PUT/PATCH** (Update data): Used to update existing data.

```
// In data.service.ts
updateData(id: number, updatedData: any): Observable<any> {
  return this.http.put<any>(`${this.apiUrl}/${id}`, updatedData);
  // Use .patch() for partial updates
}
```

- **DELETE** (Remove data): Used to delete a record.

```
// In data.service.ts
deleteData(id: number): Observable<any> {
  return this.http.delete<any>(`${this.apiUrl}/${id}`);
}
```

Step 4: Error Handling

Use RxJS operators like catchError and retry within your service methods to handle potential errors gracefully. Error handling can be implemented using catchError to manage errors. Additional features like Interceptors for tasks such as adding authentication headers and the async pipe in templates for subscribing to Observables can also be used. More details on implementing error handling with catchError

Introduction to Http Request. Make a Http Post Request Call through HttpClientModule

```
  constructor(private http: HttpClient) {}

  createProdcut () {
    let productData = this.newProdcut;
    this.http.post("https://dummyjson.com/products/add", productData).subscribe(response => {
      console.log(response);
    })
  }
```

Make Http Get Request and use RxJs Operators to transform the response

```
  getProducts () {
    this.http.get("https://dummyjson.com/products").subscribe(response => {
      console.log(response);
    })
  }
```

```
  getProducts () {
    this.http.get("https://dummyjson.com/products")
    .pipe(map((response: any) => {
      return response?.products?.map((item: any) => {return { id:item?.id, title:item?.title}})
    }))
    .subscribe(response => {
      console.log(response);

    })
  }
```

Recommended to create interface for define data types and import for the type of the response

Using Services for Http requests with HttpClient. Communicate Services and Components

```
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { map } from 'rxjs';

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

    constructor(private http: HttpClient) {}

    createProdcut(productData: any) {
        return this.http.post('https://dummyjson.com/products/add', productData);
    }

    getProducts() {
        return this.http
        .get('https://dummyjson.com/products')
        .pipe(
            map((response: any) => {
                return response?.products?.map((item: any) => {
                    return { id: item?.id, title: item?.title };
                });
            })
        )
    }
}
```

```
import { HttpClient } from '@angular/common/http';
import { Component } from '@angular/core';
import { map } from 'rxjs';
import { ProductService } from '../../service/product.service';

@Component({
  selector: 'app-product-list.page',
  imports: [],
  templateUrl: './product-list.page.html',
  styleUrl: './product-list.page.css',
})
export class ProductListPage {

  newProdcut: {
    name: string,
    price: number
  } = {
    name: "Electronic Chargerk",
    price: 34
  }

  constructor(private http: HttpClient, private productService: ProductService) {}

  createProdcut () {
    let productData = this.newProdcut;
    this.productService.createProdcut(productData).subscribe(res => {console.log(res)});
  }

  getProducts () {
    this.productService.getProducts().subscribe(res => console.log(res));
  }
}
```

Implement HTTP Delete Request

```
clearPosts() {
  this.http
    .delete('https://ng-complete-guide-aad09.firebaseio.com/posts.json')
    .subscribe((response) => {
      console.log(response);
    });
}
```

Error Handling in the Http Request Calls

```
getPosts() {
  this.postService.fetchPosts().subscribe((response) => {
    this.posts = response;
  }, error => {
    console.log(error)
  });
}
```

Sending HTTP Headers in the API Request Call

```
createPost(postData: Post) {
  return this.http.post<{ name: string }>(
    'https://ng-complete-guide-aad09.firebaseio.com/posts.json',
    postData,
    {
      headers: new HttpHeaders({
        'custome-header': 'post Leela',
      }),
    }
  );
```

Adding Query Params for the Url

```
fetchPosts() {
  return this.http
    .get<{ [key: string]: Post }>(
      `https://ng-complete-guide-aad09.firebaseio.com/posts.json`,
      {
        headers: new HttpHeaders({
          'custom-header': 'leela',
        }),
        params: new HttpParams().set('custom', 'hai')
      }
    )
    .pipe(
      map((response) => {
        let posts: Post[] = [];
        for (let key in response) {
          posts.push({ ...response[key], key });
        }
        return posts;
      })
    );
}
```

```
let searchParams = new HttpParams();
searchParams = searchParams.append('custom', 'hai');
searchParams = searchParams.append('name', 'Leela');
return this.http
  .get<{ [key: string]: Post }>(
    `https://ng-complete-guide-aad09.firebaseio.com/p
    {
      headers: new HttpHeaders({
        'custom-header': 'leela',
      }),
      params: searchParams,
    }
  )
```

```
return this.http.post<{ name: string }>(
  'https://ng-complete-guide-aad09.firebaseio.com/posts.json',
  postData,
  {
    headers: new HttpHeaders({
      'custom-header': 'post Leela',
    }),
    observe: 'response',
  }
);
```

Get entire HTTPresponse object with observe

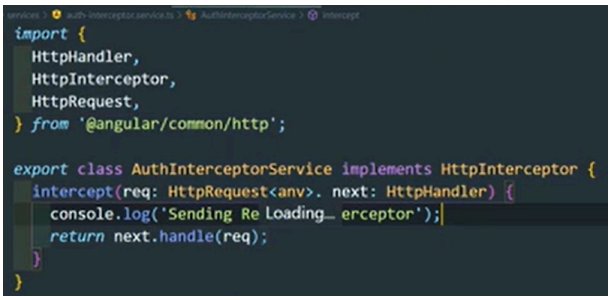
```
this.http
  .delete('https://ng-complete-guide-aad09.firebaseio.com/posts.json', {
    observe: 'events',
  }).pipe(tap(response => {
    console.log(response);
  }))
  .subscribe((response) => {
    console.log(response);
  });
```

```
this.http
  .delete('https://ng-complete-guide-aad09.firebaseio.c
    observe: 'events',
  })
  .pipe(
    tap((response) => {
      if (response.type === 0) {
        console.log('request sent');
      }

      if (response.type === 4) {
        console.log(response.body);
      }
    })
```

tap operator does not modify the data or return, it only perform specified operation

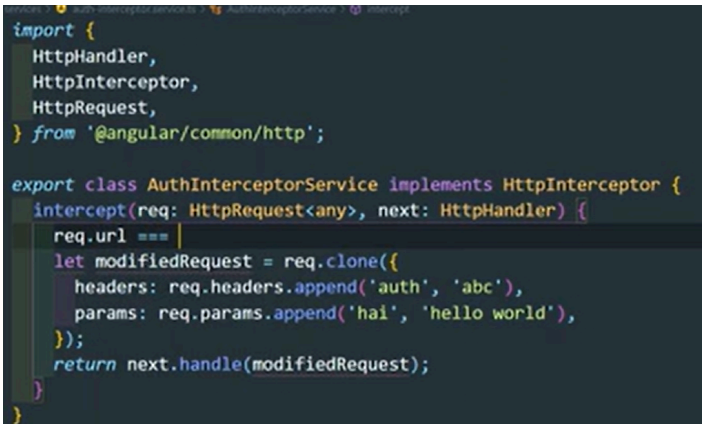
Introducing HTTP Interceptors using HTTP_INTERCEPTORS

auth-interceptor.ts



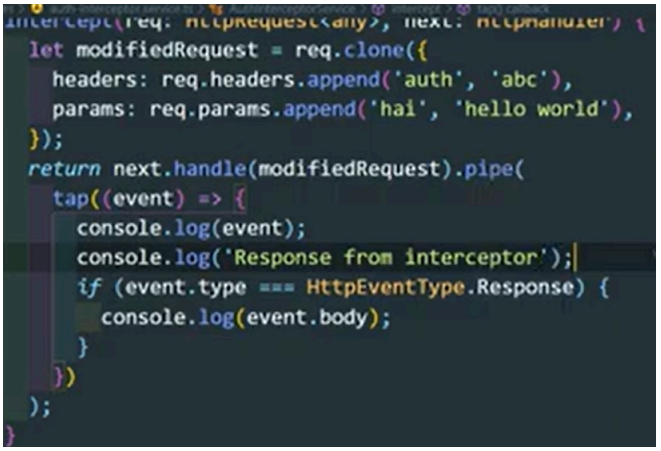
```
import {
  HttpHandler,
  HttpInterceptor,
  HttpRequest,
} from '@angular/common/http';

export class AuthInterceptorService implements HttpInterceptor {
  intercept(req: HttpRequest<anv>. next: HttpHandler) {
    console.log('Sending Re Loading… erceptor');
    return next.handle(req);
  }
}
```

```
providers: [
  {
    provide: HTTP_INTERCEPTORS,
    useClass: AuthInterceptorService,
    multi: true,
  },       You, a few seconds ago • Unco
```



```
import {
  HttpHandler,
  HttpInterceptor,
  HttpRequest,
} from '@angular/common/http';

export class AuthInterceptorService implements HttpInterceptor {
  intercept(req: HttpRequest<any>, next: HttpHandler) {
    req.url ===
    let modifiedRequest = req.clone({
      headers: req.headers.append('auth', 'abc'),
      params: req.params.append('hai', 'hello world'),
    });
    return next.handle(modifiedRequest);
  }
}
```

Response interceptor



```
  let modifiedRequest = req.clone({
    headers: req.headers.append('auth', 'abc'),
    params: req.params.append('hai', 'hello world'),
  });
  return next.handle(modifiedRequest).pipe(
    tap((event) => {
      console.log(event);
      console.log('Response from interceptor');
      if (event.type === HttpEventType.Response) {
        console.log(event.body);
      }
    })
  );
}
```

Adding Multiple Interceptors for the HTTP Request. Interceptors executing order

this executes as this order

Authentication

```
onFormSubmit(authForm: NgForm) {
  if (!authForm.valid) {
    return;
  }

  if (this.isLoginMode) {
    //Perform Login Request Call
  } else {
    this.authService
      .signUp(authForm.value.email, authForm.value.password)
      .subscribe(
        (response) => {
          console.log(response);
        },
        (error) => {
          console.log(error);
        }
      );
  }
}
```

```
onFormSubmit(authForm: NgForm) {
  if (!authForm.valid) {
    return;
  }

  this.isLoading = true;

  if (this.isLoginMode) {
    //Perform Login Request Call
  } else {
    this.authService
      .signUp(authForm.value.email
      .subscribe(
        (response) => {
          console.log(response);
          this.isLoading = false;
        },
        (error) => {
          console.log(error);
          this.isLoading = false;
        }
      );
  }
```

Loading state

Improve Error Messages with CatchError and throwError Rxjs operators

```
signUp(email: string, password: string) {
  return this.http
    .post<AuthResponseData>(
      'https://identitytoolkit.googleapis.com/v1/accounts:
    ',
      { email, password, returnSecureToken: true }
    )
    .pipe(
      catchError((errorRes) => {
        let errorMessage = 'An Error Occurred';
        if (!errorRes.error || !errorRes.error.error) {
          return throwError(errorMessage);
        }
        switch (errorRes.error.error.message) {
          case 'EMAIL_EXISTS':
            errorMessage = 'Email Already Exists';
        }
        return throwError(errorMessage);
      })
    );
}
```

in authentication service

Create and store user and token

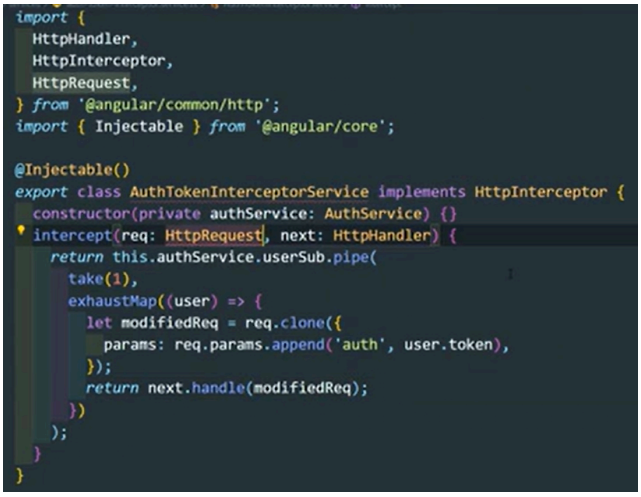
```
login(email: string, password: string) {
  return this.http
    .post<AuthResponseData>(
      `https://identitytoolkit.googleapis.com/v1/accounts:signInWithI
    `,
      { email, password, returnSecureToken: true }
    )
    .pipe(catchError(this.getErrorHandler), tap(this.handleUser));
}

private handleUser(response: AuthResponseData) {
  const expireDate = new Date(
    new Date().getTime() + +response.expiresIn * 1000
  );
  const user = new User(
    response.email,
    response.localId,
    response.idToken,
    expireDate
  );
}
```

```
You, a few seconds ago | 1 author (You)
@Injectable({ providedIn: 'root' })
export class AuthService {
  isLoggedIn = false;
  userSub = new Subject<User>();          You,
```

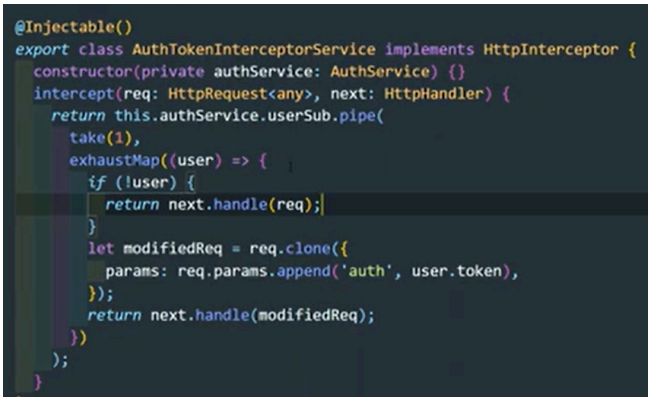
```
private handleUser(response: AuthResponseData) {
  const expireDate = new Date(
    new Date().getTime() + +response.expiresIn * 1000
  );
  const user = new User(
    response.email,
    response.localId,
    response.idToken,
    expireDate
  );
  this.userSub.next(user);
}
```

```
import { AuthService } from './../services/auth.service';
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-navigation',
  templateUrl: './navigation.component.html',
})
export class NavigationComponent implements OnInit {
  isAuthenticated = false;
  constructor(private authService: AuthService) {}

  ngOnInit() {
    this.authService.userSub.subscribe((user) => {
      this.isAuthenticated = user ? true: false;
    });
  }
}
```

Add the auth token as parameter using Interceptors

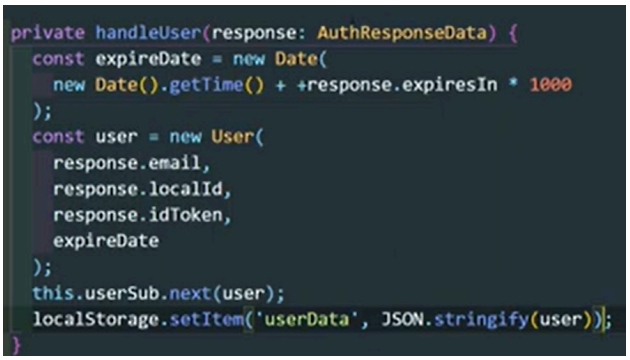


```
import {
  HttpHandler,
  HttpInterceptor,
  HttpRequest,
} from '@angular/common/http';
import { Injectable } from '@angular/core';

@Injectable()
export class AuthTokenInterceptorService implements HttpInterceptor {
  constructor(private authService: AuthService) {}
  intercept(req: HttpRequest, next: HttpHandler) {
    return this.authService.userSub.pipe(
      take(1),
      exhaustMap((user) => {
        let modifiedReq = req.clone({
          params: req.params.append('auth', user.token),
        });
        return next.handle(modifiedReq);
      })
    );
  }
}
```

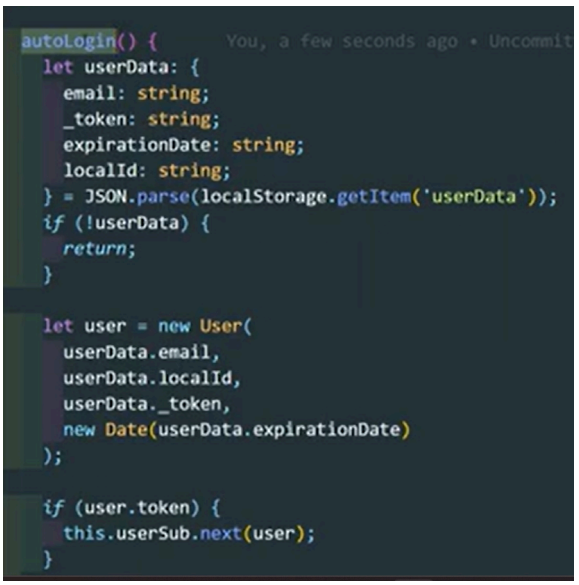


```
@Injectable()
export class AuthTokenInterceptorService implements HttpInterceptor {
  constructor(private authService: AuthService) {}
  intercept(req: HttpRequest<any>, next: HttpHandler) {
    return this.authService.userSub.pipe(
      take(1),
      exhaustMap((user) => {
        if (!user) {
          return next.handle(req);
        }
        let modifiedReq = req.clone({
          params: req.params.append('auth', user.token),
        });
        return next.handle(modifiedReq);
      })
    );
  }
```

Saving Token in LocalStorage for the autologin feature



```
private handleUser(response: AuthResponseData) {
  const expireDate = new Date(
    new Date().getTime() + +response.expiresIn * 1000
  );
  const user = new User(
    response.email,
    response.localId,
    response.idToken,
    expireDate
  );
  this.userSub.next(user);
  localStorage.setItem('userData', JSON.stringify(user));
}
```



```
autoLogin() {      You, a few seconds ago • Uncommit
  let userData: {
    email: string;
    _token: string;
    expirationDate: string;
    localId: string;
  } = JSON.parse(localStorage.getItem('userData'));
  if (!userData) {
    return;
  }

  let user = new User(
    userData.email,
    userData.localId,
    userData._token,
    new Date(userData.expirationDate)
  );

  if (user.token) {
    this.userSub.next(user);
  }
```

```
export class AppComponent implements OnInit, OnDestroy {
  title = 'angularrouting';
  userAdded = false;
  userAddedSubscription: Subscription;
  constructor(
    private authService: AuthService,
    private userService: UserService
  ) {}

  ngOnInit() {
    this.authService.autoLogin();
  }

  onLoginClick() {}

  onLogoutClick() {
    this.authService.logout();
  }

  ngOnDestroy() {
    this.userAddedSubscription.unsubscribe();
  }
```

Auto-Logout the user when the token expired

```
private handleUser(response: AuthResponseData) {
  const expireDate = new Date(
    new Date().getTime() + +response.expiresIn * 1000
  );
  const user = new User(
    response.email,
    response.localId,
    response.idToken,
    expireDate
  );
  this.userSub.next(user);
  localStorage.setItem('user (property) AuthResponseData.expiresIn: string
  this.autoLogout(+response.expiresIn * 1000);
}
```

```
autoLogout(expirationDate: number) {
  setTimeout(() => {
    this.logout();
  }, expirationDate);
}

logout() {
  this.userSub.next(null);
  this.router.navigate(['/auth']);
  localStorage.removeItem('userData');
}
```