<table>
<tr>
<td>Day 23</td>
<td>Angular Advanced</td>
<td>

Fetch Data Using HTTP

In Angular we have an HTTPClient service that is used to send HP requests to a server and receive HTTP responses asynchronously using observables.
Step 1: Import HPClient module in app.ts.
Step 2: Create a service to fetch data ex dataService
Step 3: use/consume service in component

Features :

- Simple and clean way of handling API for HTTP request
- Supports request/response transformations and interceptors
- Works seamlessly with observables

Error Handling using HTTP

Error handling in an angular HTTP client involves catching errors that occur during HTTP requests, such as network failure, server failure.

```
import { HttpClient, HttpErrorResponse } from '@angular/common/http';
import { Observable, throwError } from 'rxjs';
import { catchError } from 'rxjs/operators';

export class DataService {

  private apiUrl = 'https://jsonplaceholder.typicode.com/posts';

  constructor(private http: HttpClient) {}

  getPosts(): Observable<any> {
    return this.http.get(this.apiUrl).pipe(
      catchError(this.handleError)

private handleError(error: HttpErrorResponse) {
    if (error.error instanceof ErrorEvent) {
      console.error('Client-side error:', error.error.message);
    } else {
      console.error(`Server returned code ${error.status}, body: `, error.error);
    }
    return throwError(() => 'Something went wrong; please try again later.');
  }
```

Feature :

1. Prevents application from breaking due to HTTP errors.
2. Allows showing meaningful error messages to users.
3. Centralizes error logging.
4. Extra boilerplate code for each request if not using interceptors.

</td>
</tr>
</table>

		5. Might hide actual errors if not logged properly. Best practices : ● Always centralize HTP error handling in a service or interceptor. ● Rely on consoles for unit testing else throw errors with user friendly error messages not technical ones. ● Log detailed errors for debugging(avoid exposing sensitive data.) ● Use retry logic(logic operator) for transient network errors.
Day 24	Angular Advanced & Angular Forms	Template Driven Forms
		Reactive Forms

HTTPclient module :
Angular service for making HP requests, returning observables.
Pros & Cons : -
Simplifies HTTP request with builtin JSON parsing.
Works seamlessly with RxJS.

- Requires handling of Observables.

Best practices :
-Create a separate service for API logic .
-Centralize API URLs in one place.

HTTP request (get, put, delete,post)

Created a service, injecting it in components to show asynchronous communication.(API/JSON)
HTTP interceptor : A service that can inspect and transform HTTP requests. Acting like a **Middle ware.**

It can be used for Logging and Error Handling.
Benefits :

- Centralized request/response logging & error handling.
- Can add authentication headers automatically
- Avoids code repetition.

Limitations :

- Can make debugging harder if overused.
- All requests pass through interceptors - even those not needing modification.

Best practices :

- Keep logic minimal to avoid blocking requests.
- Chain multiple interceptors for different concerns(logging, auth etc)

Angular Form
<u>Template Driven forms</u>
- Forms defined mostly HTML using ngModel for two way binding.
Pro:

- **Simple and quick to implement.**
- **Less boilerplate code**

Cons:

- **Less scalable for complex forms**

- **Harder to test programmatically.**

Best practice :

Always use TDF for small/simple requirements.

```
<form #formRef="ngForm" (ngSubmit)="onSubmit(formRef)">
  <input name="username" ngModel required>
  <button type="submit">Submit</button>
</form>
```

Reactive forms

Forms built in **Typescript** using **FormControl** and **formGroup**, offering full programmatic control.

Pros:

- Easy to unit test.
- Better scalability for dynamic/complex forms.

Cons:

- More initial code.

Best practice :

- Ideal for large, dynamic i.e. data driven forms.

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

@Component({
  selector: 'app-root',
  template: `
    <form [formGroup]="myForm" (ngSubmit)="onSubmit()">
      <input formControlName="username">
      <button type="submit">Submit</button>
    </form>
  `
})

export class AppComponent {
  myForm = new FormGroup({
    username: new FormControl('', Validators.required)
  });

  onSubmit() {
    console.log(this.myForm.value);
  }
}
```

Implementing Template driven forms and reactive forms:
Step1: Create or reuse an angular App
Step 2: Generate two components.

```
ng g c template-form
ng g c reactive-form
```

Step 3: Import form modules(v.imp) in app.ts
imports: [BrowserModule, FormsModule, ReactiveFormsModule],

Step 4: Template driven forms (Simple, ngModel)

```
export class TemplateFormComponent {
  // backing model is optional; using formRef.value in submit handler
  onSubmit(formRef: NgForm) {
    console.log('Template Form Submitted:', formRef.value);
    formRef.reset(); // optional
  }
```

```
<h2>Template-Driven Form</h2>

<form #formRef="ngForm" (ngSubmit)="onSubmit(formRef)" novalidate>
  <label>
    Username
    <input
      name="username"
      ngModel
      required
      minlength="3"
      placeholder="e.g. alex"
      />
  </label>

  <!-- tiny validation display -->
  <div *ngIf="formRef.submitted && formRef.controls['username']?.invalid" style="color:#c00">
    Username is required (min 3 chars).
  </div>

  <button type="submit" [disabled]="formRef.invalid">Submit</button>
</form>
```

Step4: Reactive forms (programmatic control)

```
export class ReactiveFormComponent {
  userForm = new FormGroup({
    name:  new FormControl('', [Validators.required, Validators.minLength(3)]),
    email: new FormControl('', [Validators.required, Validators.email])
  });

onSubmit() {
```

```
    if (this.userForm.valid) {
      console.log('Reactive Form Submitted:', this.userForm.value);
      this.userForm.reset();
    } else {
      this.userForm.markAllAsTouched();//Marks every control in the form as if the user has interacted with them
            // so angular validation state touched becomes true for each conrol
    }
  }

 // helpers for template
  get name()  { return this.userForm.get('name'); }
  get email() { return this.userForm.get('email'); }
}
```

```
<h2>Reactive Form</h2>

<form [formGroup]="userForm" (ngSubmit)="onSubmit()" novalidate>
  <label>
    Name
    <input formControlName="name" placeholder="e.g. Jamie" />
  </label>
            <div *ngIf="name?.touched && name?.invalid" style="color:#c00">
            <div *ngIf="name?.errors?.['required']">Name is required.</div>
             <div *ngIf="name?.errors?.['minlength']">Min length is 3.</div>
              </div>

 <label>
    Email
    <input formControlName="email" placeholder="e.g. jamie@mail.com" />
  </label>
             <div *ngIf="email?.touched && email?.invalid" style="color:#c00">
            <div *ngIf="email?.errors?.['required']">Email is required.</div>
           <div *ngIf="email?.errors?.['email']">Must be a valid email.</div>
         </div>
```

Step 5: Showing both form on root page

```
<h1>Angular Forms Demo (Day 24)</h1>

<app-template-form></app-template-form>
<hr />
<app-reactive-form></app-reactive-form>
```

Step 6: Running application
Ng serve

Step 1: Implement an interface HTTPInterceptor in the LoggingInterceptor class.

Step 2: implementing a method intercept(req: httpRequest<any>, next: HTPhandler) : Observable<httpEvent<any>>

Step 3: Registering in app.ts

Importing and adding it in providers

1. Setup Modules

- **Import Angular Essentials**
  - **HttpClientModule → so we can make HTTP calls.**
  - **ReactiveFormsModule → so we can use reactive forms.**
- **Register HTTP Interceptor in AppModule under providers.**

---

2. Create the HTTP Service (user.service.ts)

- **Define an API endpoint (here: https://jsonplaceholder.typicode.com/users).**
- **Inject HttpClient.**
- **Create a method getUsers() returning an Observable from HttpClient.get().**

---

3. Create the Interceptor (log.interceptor.ts)

- **Implement HttpInterceptor.**
- **Intercept every outgoing HTTP request.**
- **Log the request URL before sending.**
- **Pass the request to next.handle() and log the response when received.**

---

4. Create the Component (app.component.ts)

- **OnInit lifecycle:**
  - **Call the userService.getUsers() method.**

- **Subscribe to the returned Observable to store the result in a local users array.**

- **Create a Reactive Form using FormGroup and FormControl.**
- **Handle Form Submission by logging form values to console.**

---

5. Create the Template (app.component.html)

- **Display Data:**
  - **Use *ngFor to loop over the users array and show each user's name and email.**
- **Create Form UI:**
  - **Bind form controls to input fields using formControlName.**
  - **Add a submit button to trigger the onSubmit() method.**

---

6. Run the App

- **Start with ng serve.**
- **Open browser → Angular app runs:**
  1. **On load: ngOnInit() calls the API → Interceptor logs request/response.**
  2. **API data appears in the UI list.**
  3. **User enters data in the form and submits → console logs the submitted object.**

Case Study :

You can build an angular application that allows users to fill out a registration form, submit that data to a mock API, fetch all registered users from the server and handle errors gracefully.

Above case study is based on following topics :

1. Reactive forms in angular
2. HTTP GET and POST request using HTPClient
3. Error Handing in HTP calls
4. Displaying fetch data in a table
5. Form validation & Marking fields as touched.

Step 1: Generate Projects & Modules

```
ng generate module user
ng generate component user/register
ng generate service user/user
ng generate interceptor interceptors/error
```

Step 2: Enable forms and HTP in app.module.ts or app.ts

```
@NgModule({
  imports: [BrowserModule, HttpClientModule, ReactiveFormsModule],
  ...
```

Step 3: Implementing userService with following methods :

```
registerUser(userData: any) {
  return this.http.post(this.apiUrl, userData).pipe(
    catchError(error => {
      return throwError(() => new Error('Registration failed. Please try again.'));
    })
  );
}

getUsers() {
  return this.http.get(this.apiUrl).pipe(
    catchError(error => {
      return throwError(() => new Error('Failed to fetch users.'));
    })
  );
}
```

Step 4: Defining interceptor.ts

```
export class ErrorInterceptor implements HttpInterceptor {

intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
  return next.handle(req).pipe(
    catchError((error: HttpErrorResponse) => {
      console.error('HTTP Error: ', error.message);
      alert('An error occurred: ' + error.message);
      return throwError(() => error);
    })
  );
}
```

Step 5: register component: with user form and validation

```
export class RegisterComponent {
```

```
  users: any[] = [];
  errorMessage = '';

  userForm = this.fb.group({
    name: ['', [Validators.required, Validators.minLength(3)]],
    email: ['', [Validators.required, Validators.email]],
    password: ['', [Validators.required, Validators.minLength(6)]]
  });

constructor(private fb: FormBuilder, private userService: UserService) {}


onSubmit() {
    if (this.userForm.invalid) {
      this.userForm.markAllAsTouched();
      return;
    }
    this.userService.registerUser(this.userForm.value).subscribe({
      next: (res) => {
        alert('User Registered Successfully!');
        this.userForm.reset();
      },
      error: (err) => this.errorMessage = err.message
    });
  }

 fetchUsers() {
    this.userService.getUsers().subscribe({
      next: (data: any) => this.users = data,
      error: (err) => this.errorMessage = err.message
    });
  }
```

Step 6: Formatting HTML for register.component.html

```
<form [formGroup]="userForm" (ngSubmit)="onSubmit()">
  <label>Name:</label>
<input formControlName="name" />
  <div *ngIf="userForm.get('name')?.touched && userForm.get('name')?.invalid">
    Name is required (min 3 chars)
  </div>


 <label>Email:</label>
  <input formControlName="email" />
  <div *ngIf="userForm.get('email')?.touched && userForm.get('email')?.invalid">
    Valid email required
  </div>

<label>Password:</label>
  <input type="password" formControlName="password" />
  <div *ngIf="userForm.get('password')?.touched && userForm.get('password')?.invalid">
    Min length 6 required
  </div>
```

```
  <button type="submit">Register</button>
</form>


<hr />

<button (click)="fetchUsers()">Fetch Users</button>

<table *ngIf="users.length">
  <tr><th>Name</th><th>Email</th></tr>
  <tr *ngFor="let u of users">
    <td>{{u.name}}</td>
    <td>{{u.email}}</td>
  </tr>
</table>
```