CRUD USING HTTP

Aitrich

Contents

What Is HTTP ?

HTTP refers to the built-in module that allows you to make HTTP requests to fetch data from external APIs, servers, or resources.

It provides a set of powerful methods to interact with RESTful APIs and other HTTP-based services.

HTTP module in Angular involves the following components:

- HTTP Client
- HTTP Interceptors
- HTTP Headers
- Observables

HTTP Client

- An HTTP client is a software application that initiates HTTP requests to a server and receives HTTP responses.
- It acts as an intermediary between a user or another application and a web server.
- The Angular framework provides the HttpClient module as part of the @angular/common/http package.
- The HttpClient module simplifies the process of making HTTP requests and handling responses by providing a high-level API.

Some of the common functionalities of an HTTP client

- **Sending HTTP requests:** The client can send various HTTP requests like GET, POST, PUT, and DELETE to the server. Each request specifies the method, the URL of the resource, and optionally, request headers and an entity-body.
- **Receiving HTTP responses:** The server responds to the client's request with an HTTP response, which includes the status code, response headers, and an optional entity body. The client can parse the response and extract the relevant information.
- **Handling different protocols:** Most HTTP clients support both HTTP/1.1 and HTTP/2 protocols.
- **Managing cookies and sessions:** Clients can store and manage cookies and sessions sent by the server to maintain state information across multiple requests.
- **Following redirects:** If the server responds with a redirect status code, the client can automatically follow the redirect and send a new request to the new location.

HTTP Interceptors

- HTTP Interceptors are a powerful tool in web development and server-side programming, typically associated with web frameworks and libraries.
- They allow you to intercept and handle HTTP requests and responses globally within an application, offering several benefits and use cases.
- Classes or functions that "intercept" HTTP requests and responses as they travel between your application and the server.
- Can be implemented on both client-side (e.g., Angular) and server-side (e.g., Express.js) frameworks.

HTTP Headers

- HTTP headers are essential components of the communication between clients and servers, containing metadata that accompanies HTTP requests and responses to convey crucial information.

- These headers play a pivotal role in enhancing the functionality and security of web communication, providing instructions, authentication details, and metadata for efficient data exchange.

- HTTP headers come in various types, including Request Headers, Response Headers, and Entity Headers, each serving distinct purposes in managing the flow of information during HTTP transactions.

Observable

- Observables are a powerful feature in Angular, provided by the RxJS library.
- Observables are used extensively in Angular to handle asynchronous operations, such as HTTP requests, user input events, and other asynchronous tasks.
- They represent a stream of data that can be observed over time.

Aitrich

Observable

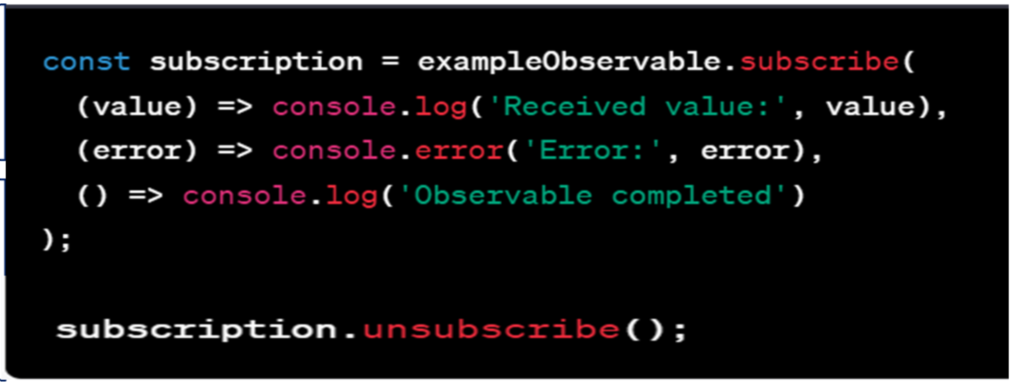
- It managing asynchronous operations related to HTTP requests, providing a streamlined and reactive approach to handle data communication with servers.
- Angular leverages the observable pattern to represent asynchronous data streams, allowing developers to subscribe to these streams and react to events emitted over time, a paradigm well-suited for handling HTTP operations
- It offer advantages such as handling multiple values over time, providing powerful operators for transformation and manipulation, and facilitating cleaner, more readable asynchronous code.
- It simplifies error handling by providing elegant mechanisms to catch and manage errors during HTTP requests,

Observable

Subscribing to an Observable

To consume data emitted by an Observable, we need to subscribe to it.

Subscriptions receive emitted values, handle errors, and completion signals



```
const subscription = exampleObservable.subscribe(
  (value) => console.log('Received value:', value),
  (error) => console.error('Error:', error),
  () => console.log('Observable completed')
);

 subscription.unsubscribe();
```

Observable Operators

Observable operators in Angular are powerful functions that allow for the transformation, combination, filtering, and manipulation of data emitted by Observables, providing a versatile and efficient way to handle asynchronous operations in Angular applications.

Some Observable Operators are

map: Transforms emitted data.

merge: Concatenates emissions from multiple Observables simultaneously

concat: Emits emissions from one Observable and then the next, one after the other.

filter: Only emits values that match a predicate function.

delay: Introduces a delay before emitting values.

Observable

Observables in Angular provide robust mechanisms for handling both errors and completion.

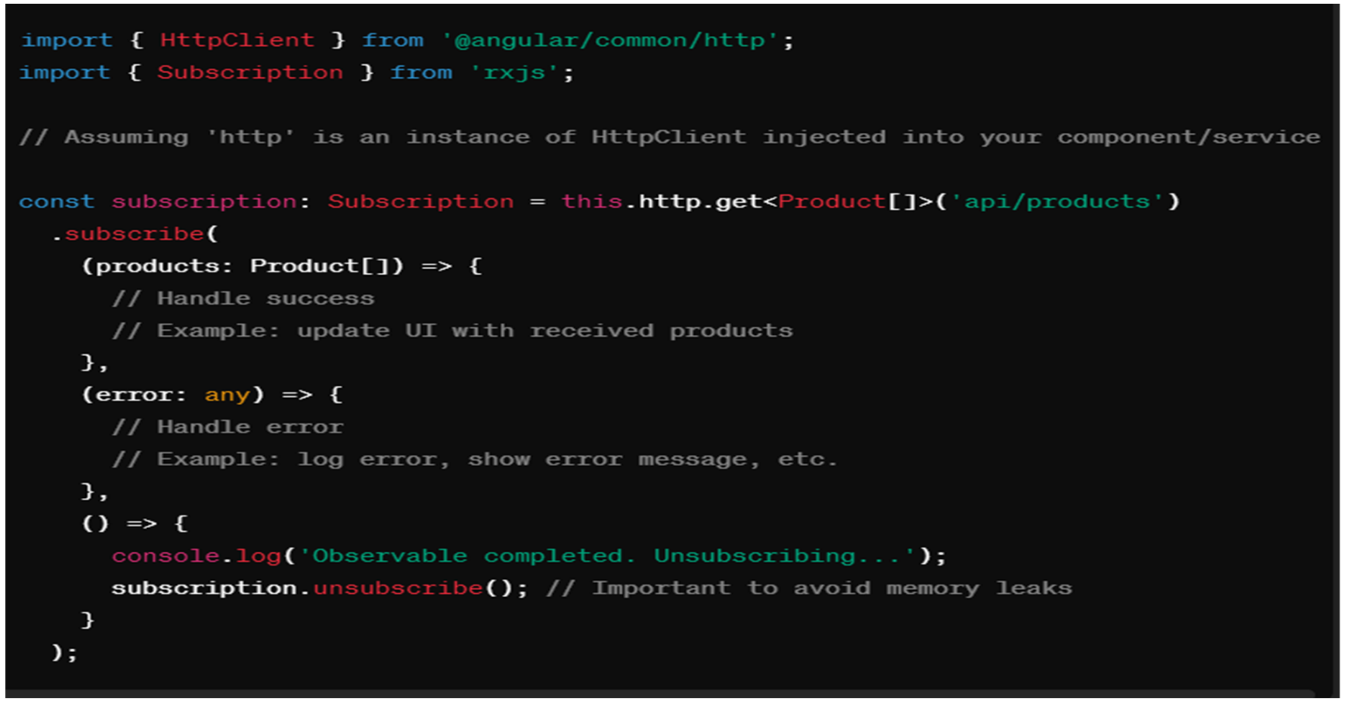
Error Callback:

- Called when an error occurs during the observable execution.
- Provides an opportunity to log the error, notify the user, or take corrective actions.

Complete Callback:

- Called when the observable finishes emitting all its values and reaches its natural end.
- Offers a chance to perform cleanup tasks or execute final logic.
- The complete callback is also crucial for proper resource management and unsubscribing from the observable to prevent memory leaks.

Example



```
import { HttpClient } from '@angular/common/http';
import { Subscription } from 'rxjs';

// Assuming 'http' is an instance of HttpClient injected into your component/service

const subscription: Subscription = this.http.get<Product[]>('api/products')
  .subscribe(
    (products: Product[]) => {
      // Handle success
      // Example: update UI with received products
    },
    (error: any) => {
      // Handle error
      // Example: log error, show error message, etc.
    },
    () => {
      console.log('Observable completed. Unsubscribing...');
      subscription.unsubscribe(); // Important to avoid memory leaks
    }
  );
```

Exception Handling

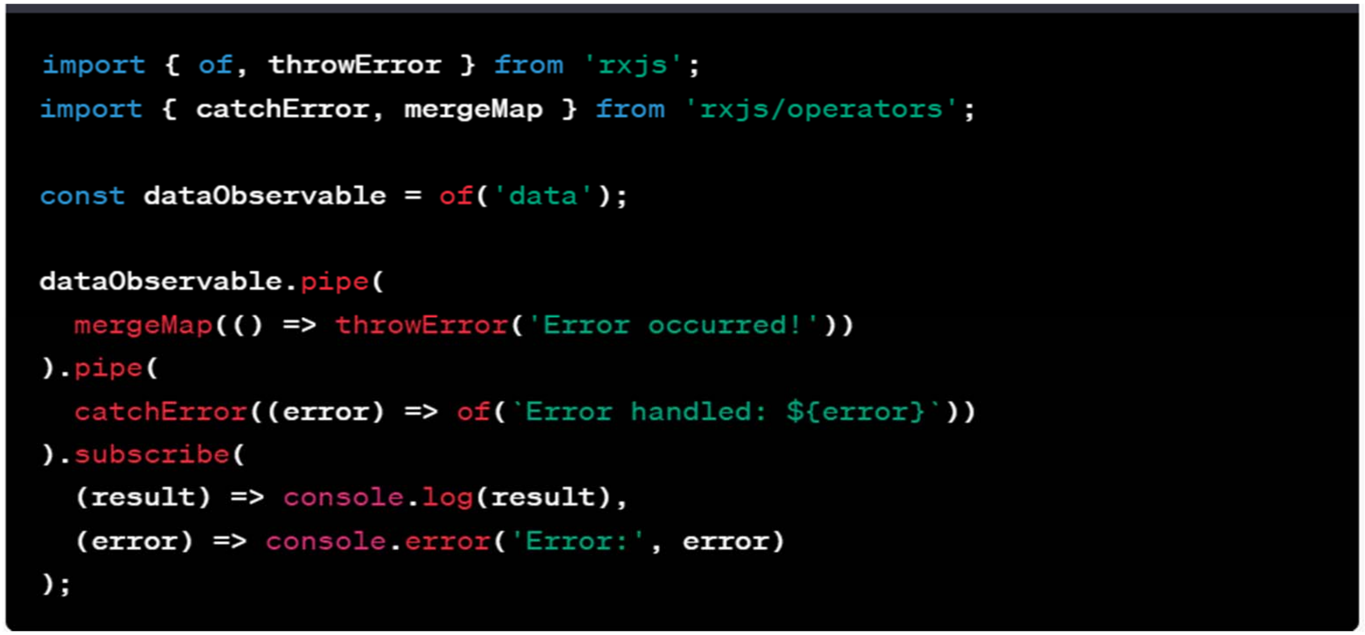
Using catchError to Handle Errors within Observable Streams:

- **The catchError operator is a fundamental tool in handling errors emitted by Observables.**
- **It intercepts errors and allows you to define alternative behavior instead of letting the error terminate the entire stream.**

Aitrich

Exception Handling

Example: catchError Operator



```
import { of, throwError } from 'rxjs';
import { catchError, mergeMap } from 'rxjs/operators';

const dataObservable = of('data');

dataObservable.pipe(
  mergeMap(() => throwError('Error occurred!'))
).pipe(
  catchError((error) => of(`Error handled: ${error}`))
).subscribe(
  (result) => console.log(result),
  (error) => console.error('Error:', error)
);
```

Subscribe To Observable

- Subscribing to an Observable is essential for consuming the data emitted by the Observable.

- The **subscribe()** method allows us to observe and react to the data stream.

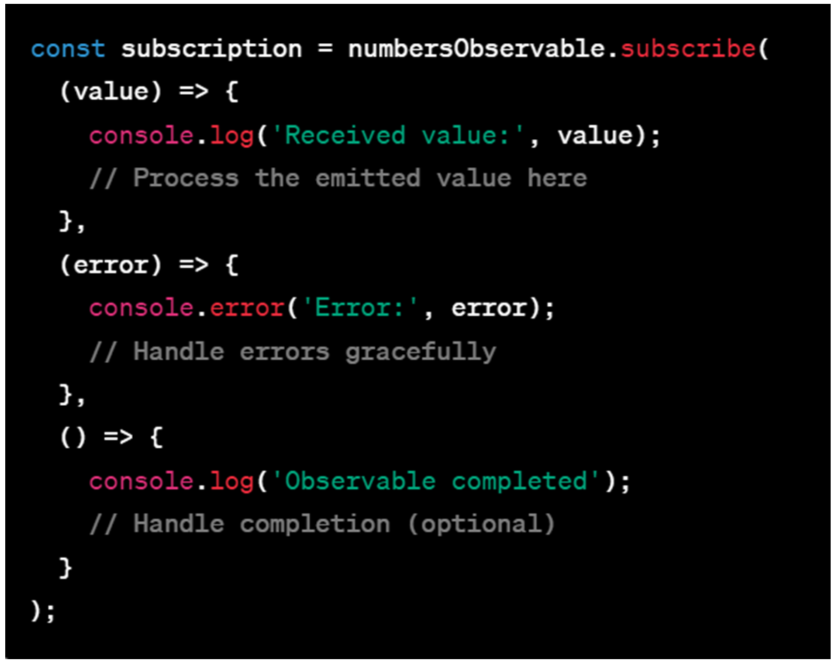
- The subscribe() method takes three optional callbacks: next, error, and compete.

- Properly unsubscribe from Observables to avoid memory leaks.

Aitrich

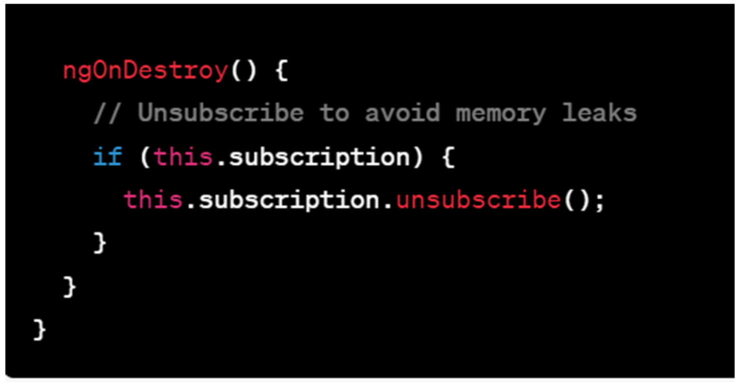
Subscribe To Observable

Subscribe to the Observable



```
const subscription = numbersObservable.subscribe(
  (value) => {
    console.log('Received value:', value);
    // Process the emitted value here
  },
  (error) => {
    console.error('Error:', error);
    // Handle errors gracefully
  },
  () => {
    console.log('Observable completed');
    // Handle completion (optional)
  }
);
```

Unsubscribe from the Observable



```
  ngOnDestroy() {
    // Unsubscribe to avoid memory leaks
    if (this.subscription) {
      this.subscription.unsubscribe();
    }
  }
}
```

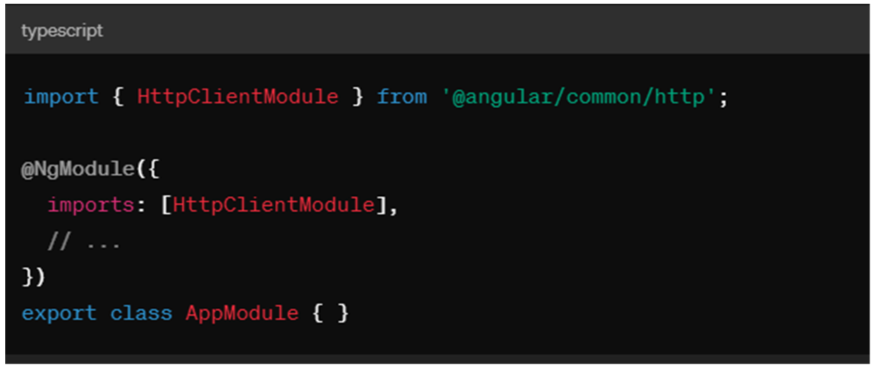
HTTP request

- Make an HTTP request using the HttpClient service.
- The asynchronous method sends an HTTP request, and returns an Observable that emits the requested data when the response is received.
- The return type varies based on the observe and responseType values that pass to the call.

To make an HTTP request in Angular, you generally follow these steps:

Import HttpClientModule:

Ensure that you have imported HttpClientModule in your Angular application. You typically add this to the imports array in your @NgModule decorator in the app.module.ts file.

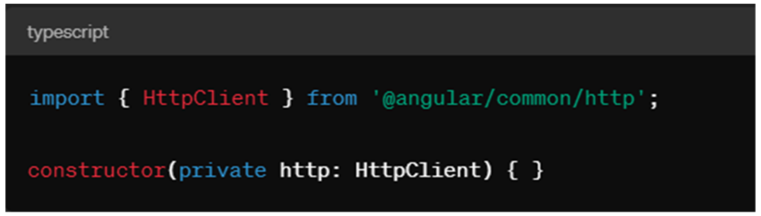


typescript

```typescript
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  imports: [HttpClientModule],
  // ...
})
export class AppModule { }
```

Inject HttpClient:

Inject the HttpClient service into the component or service where you want to make the HTTP request.

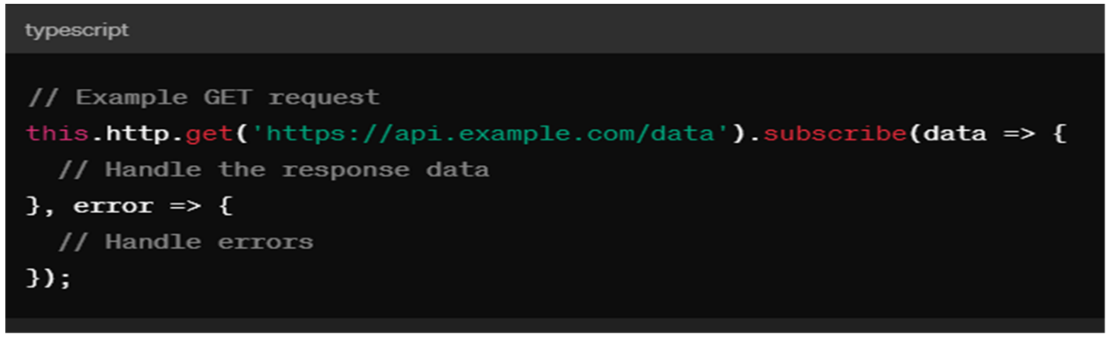


typescript

```typescript
import { HttpClient } from '@angular/common/http';

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

Make the HTTP Request:

Use one of the HTTP methods (get, post, put, delete, etc.) provided by the HttpClient service to make the actual HTTP request. Specify the URL of the API endpoint and, if needed, include request parameters or a request body.



typescript

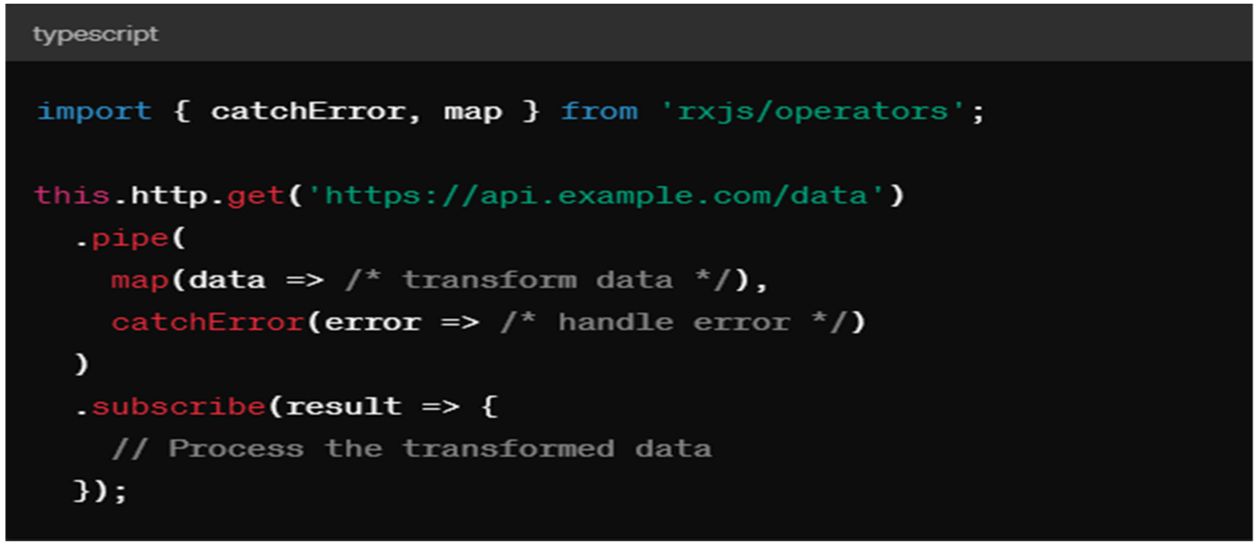
```typescript
// Example GET request
this.http.get('https://api.example.com/data').subscribe(data => {
  // Handle the response data
}, error => {
  // Handle errors
});
```

Handle the Response:

Subscribe to the observable returned by the HTTP method to handle the response. You can use the subscribe method to define what to do with the response data, and optionally, how to handle errors.

Use RxJS Operators(optional):

You can use RxJS operators like map, catchError, and others to transform or handle the data and errors more effectively.



typescript

```typescript
import { catchError, map } from 'rxjs/operators';

this.http.get('https://api.example.com/data')
  .pipe(
    map(data => /* transform data */),
    catchError(error => /* handle error */)
  )
  .subscribe(result => {
    // Process the transformed data
  });
```

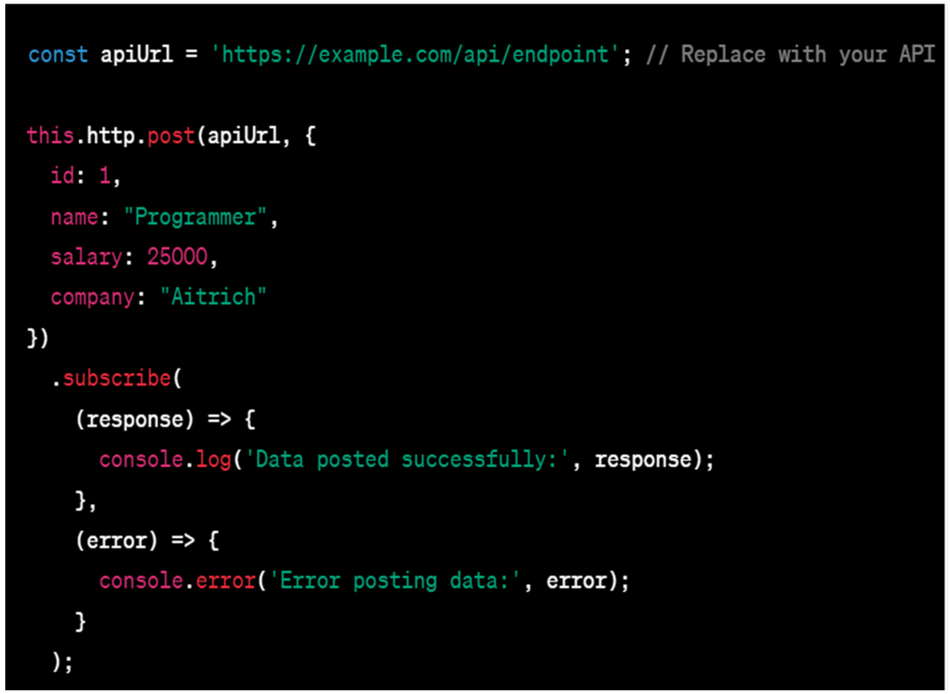
Crud Operations with HTTP

- CRUD (Create, Read, Update, Delete) operations can be performed using HTTP methods
- HTTP (Hypertext Transfer Protocol) is the foundation of data communication on the web, and it provides several methods to interact with resources on a server.
- The primary or most-commonly-used HTTP methods are POST, GET, PUT, and DELETE. These correspond to create, read, update, and delete (or CRUD) operations,

CREATE Operation

HTTP Method : POST

- Used to submit data to be processed to a specified resource.
- Expects a 201 "Created" response with the newly created resource details.



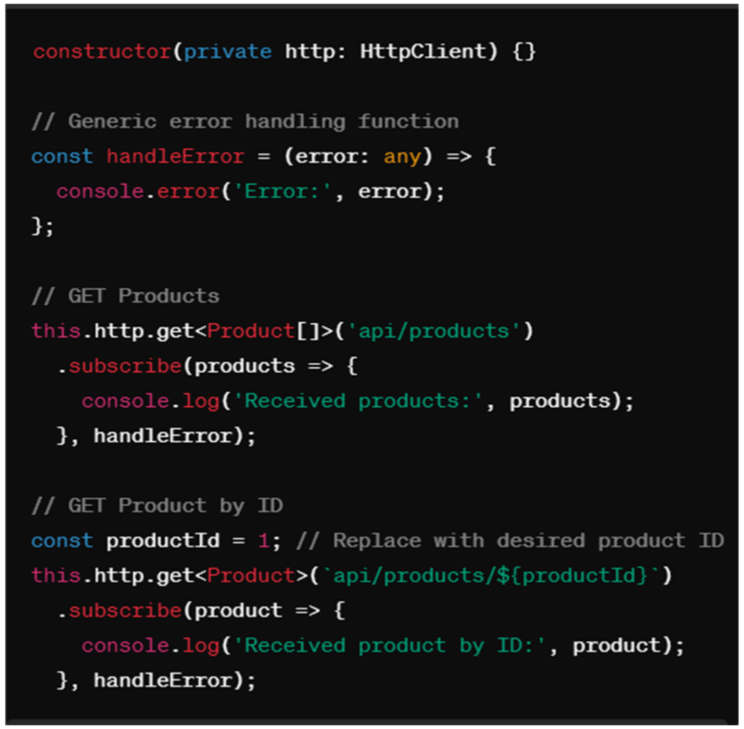
```
const apiUrl = 'https://example.com/api/endpoint'; // Replace with your API

this.http.post(apiUrl, {
  id: 1,
  name: "Programmer",
  salary: 25000,
  company: "Aitrich"
})
  .subscribe(
    (response) => {
      console.log('Data posted successfully:', response);
    },
    (error) => {
      console.error('Error posting data:', error);
    }
  );
```

READ Operation

- **HTTP Method: GET**

- **Description: Used to retrieve data from the server, either specific resources or collections of resources.**

- **Expects a 200 "OK" response with the requested data in the body.**

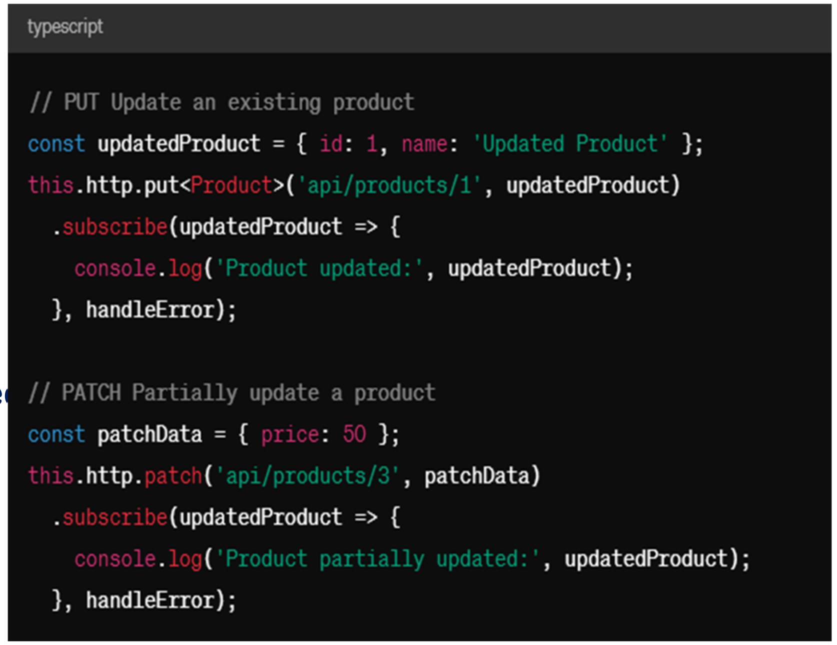


```
constructor(private http: HttpClient) {}

// Generic error handling function
const handleError = (error: any) => {
  console.error('Error:', error);
};

// GET Products
this.http.get<Product[]>('api/products')
  .subscribe(products => {
    console.log('Received products:', products);
  }, handleError);

// GET Product by ID
const productId = 1; // Replace with desired product ID
this.http.get<Product>(`api/products/${productId}`)
  .subscribe(product => {
    console.log('Received product by ID:', product);
  }, handleError);
```

UPDATE Operation

HTTP Method : PUT (for full updates) or PATCH (for partial updates)

- Updates an existing resource on the server.
- Partially updates an existing resource on the server.
- Expects a 200 "OK" response with the update resource details..

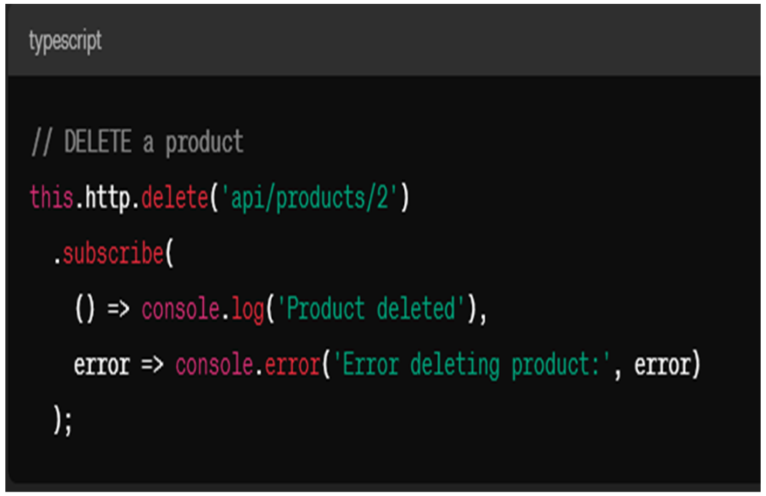


typescript

```typescript
// PUT Update an existing product
const updatedProduct = { id: 1, name: 'Updated Product' };
this.http.put<Product>('api/products/1', updatedProduct)
  .subscribe(updatedProduct => {
    console.log('Product updated:', updatedProduct);
  }, handleError);

// PATCH Partially update a product
const patchData = { price: 50 };
this.http.patch('api/products/3', patchData)
  .subscribe(updatedProduct => {
    console.log('Product partially updated:', updatedProduct);
  }, handleError);
```

DELETE Operation

- **HTTP Method : DELETE**
- **Deletes a resource from the server.**
- **Expects a 204 "No Content" response indicating successful deletion.**



```typescript
// DELETE a product
this.http.delete('api/products/2')
  .subscribe(
    () => console.log('Product deleted'),
    error => console.error('Error deleting product:', error)
  );
```

Questions?

Aitrich