HTTP : TITLE

Introduction to HTTP Request & Response

To set up a JSON server for your Angular projects, follow these steps:

Step 1: Install JSON Server

1. Open your terminal in VS Code.
2. Run the following command to install JSON Server globally:

bash

```
npm install -g json-server
```

Step 2: Create a JSON File

1. In your Angular project directory, create a new folder (e.g., `db`).
2. Inside this folder, create a file called `db.json`.
3. Populate `db.json` with your data. Here's a simple example:

```
{
  "posts": [
    { "id": 1, "title": "Post 1", "content": "Content of Post 1" },
    { "id": 2, "title": "Post 2", "content": "Content of Post 2" }
  ],
  "comments": [
    { "id": 1, "postId": 1, "body": "Comment for Post 1" },
    { "id": 2, "postId": 2, "body": "Comment for Post 2" }
  ]
}
```

Step 3: Run JSON Server

1. In your terminal, navigate to the directory where your `db.json` file is located.
2. Run the following command:

```bash
json-server --watch db.json
```

3. By default, the server will run at `http://localhost:3000`.

Step 4: Make API Calls in Angular

You can now use Angular's `HttpClient` to make API calls to your JSON server. For example:

typescript

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

@Injectable({
  providedIn: 'root'
})
export class ApiService {
  private apiUrl = 'http://localhost:3000/posts'; // Adjust URL as needed

  constructor(private http: HttpClient) {}

  getPosts() {
    return this.http.get(this.apiUrl);
  }

  // Other API methods (e.g., create, update, delete) can be added here
}
```

Step 5: Use the Service in a Component

Inject the service into a component and use it to fetch data:

typescript

```typescript
import { Component, OnInit } from '@angular/core';
import { ApiService } from './api.service';
```

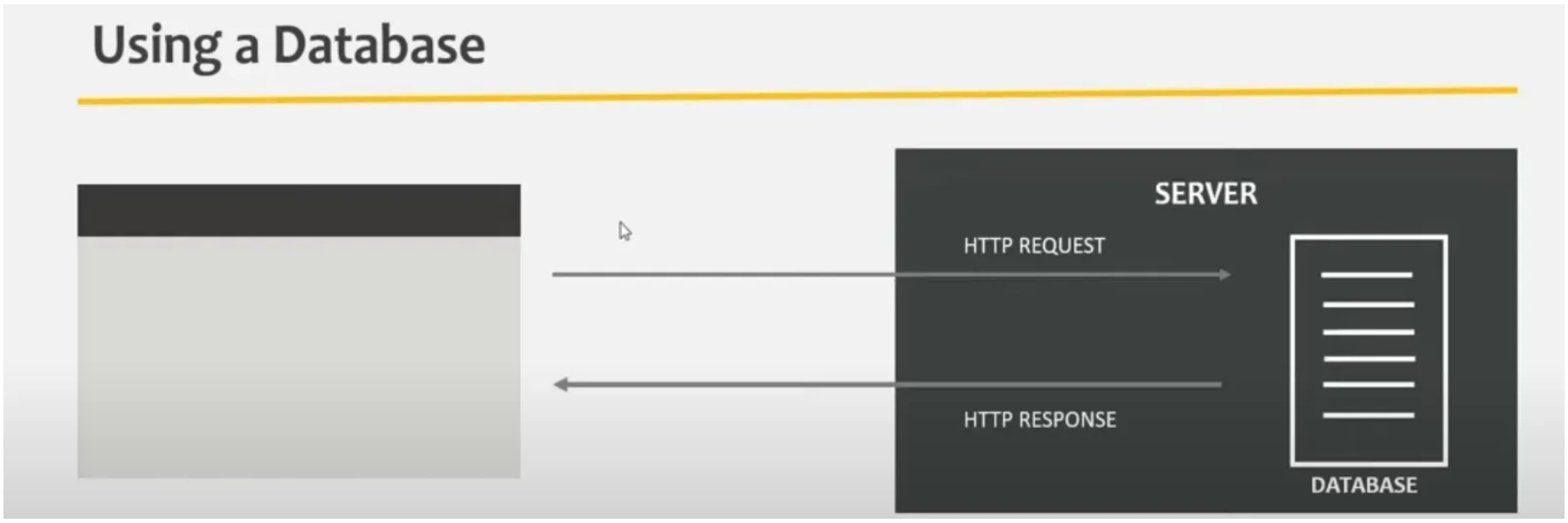
```
@Component({
  selector: 'app-posts',
  template: `
    <ul>
      <li *ngFor="let post of posts">{{ post.title }}</li>
    </ul>
  `
})
export class PostsComponent implements OnInit {
  posts: any[] = [];

  constructor(private apiService: ApiService) {}

  ngOnInit() {
    this.apiService.getPosts().subscribe(data => {
      this.posts = data;
    });
  }
}
```

Now you can run your Angular app, and it will fetch data from your JSON server!

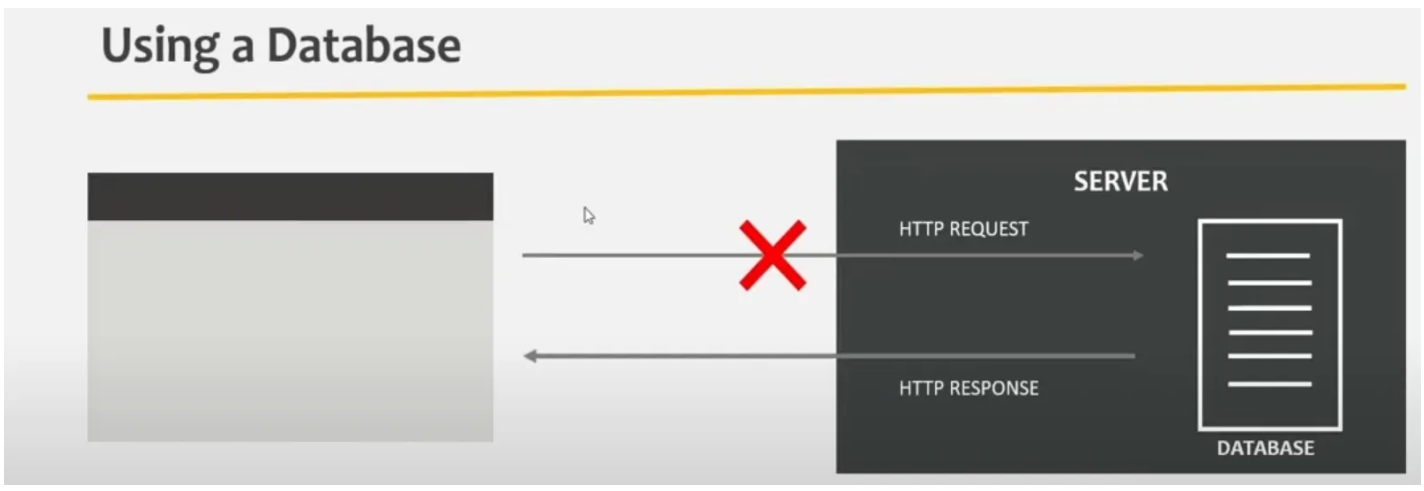
Introduction to HTTP Request & Response

Saving Data in Browsers Memory

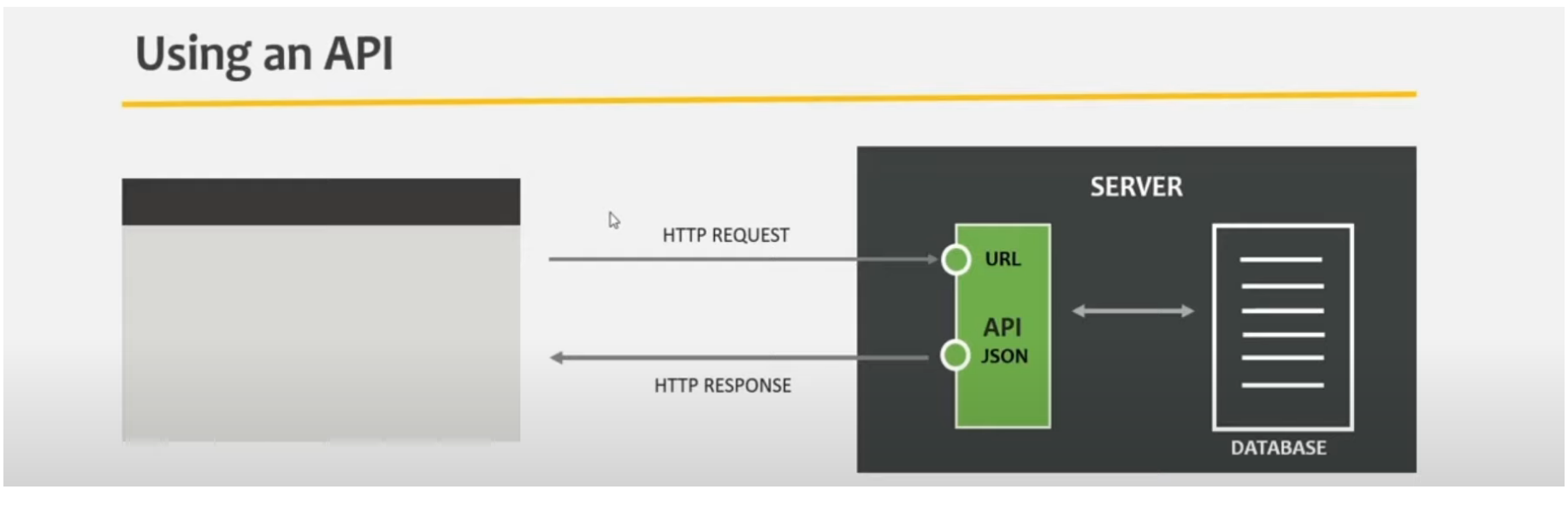
```
users: string[] = ['john', 'mark']

OnSaveClick(newUser: string){
    this.users.push(newUser);
}
```



Directly connecting to the database is not recommended



Use API’s Instead



HTTP Request Core Parts

1. URL

The most important part of an HTTP request is the URL to which we are sending the request.

If we are making an HTTP request to an API, in that case that URL is also called as API endpoint.

For example, an API URL looks something like below:

`yourdomain.com/api/products/id`

HTTP Request Core Parts

2. HTTP Verbs

When communicating with the RESTful APIs, its not just the URL which matters, but also the HTTP verbs you are using.

HTTP verbs define, which type of request you want to send to the API endpoint. We have different types of HTTP verbs for doing different things – ***GET***, ***POST***, ***PUT***, ***DELETE***, ***PATCH*** etc.

HTTP Request Core Parts

3. HTTP Request headers

When making HTTP request to the server, along with API endpoint and HTTP verbs, we also need to send some additional metadata to the server with the request we are sending.

HTTP Request headers are optional and client set some default headers for an HTTP request.

However, it also possible to set HTTP request headers of our own when sending the request to the server.

HTTP Request Core Parts

4. Request Body

For some of the HTTP verbs, we also need to specify the body i.e., the data which we want to send with the request.

Not all the request needs to have a body. HTTP requests like POST, PUT, PATCH etc. need to have a body. And request like GET need not to have a body.

Relational vs NoSQL Database

Relation Database

ID	Name	Gender	Type
101	John Smith	Male	Yearly
102	Mark Vought	Male	Monthly
103	Sarah King	Female	Quarterly

NoSQL Database

```
{
    "id": 101,
    "name": 'John Smith',
    "gender": 'male',
    "type": 'yearly'
},
{
    "id": 101,
    "name": 'Mark Vought',
    "gender": 'male',
    "type": 'Monthly'
}
```

POST ⇒

Don't forgot to add ngModel for listening to the input changes on Forms

crud.component.html U

AllConcepts > src > app > crud > crud.component.html > ...

Go to component

```html
<div class="container">
  <form #postValues="ngForm" (ngSubmit)="SubmitData(postValues.value)">
    FirstName:
    <input type="text" name="firstname" ngModel><br><br>
    LastName:
    <input type="text" name="lastname" ngModel><br><br>
    Gender
    <input type="text" name="gender" ngModel><br><br>

    <input type="submit" value="Submit">
  </form>
</div>
```

```ts
import { Component, inject } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Component({
  selector: 'app-crud',
  templateUrl: './crud.component.html',
  styleUrls: ['./crud.component.css']
})
export class CRUDComponent {

  http = inject(HttpClient);

  SubmitData(Data: any) {

    this.http.post("http://localhost:3000/posts", Data).subscribe((res) => {
      console.log(res)
    });

  }
}
```

```ts
@Component({
  selector: 'app-crud',
  templateUrl: './crud.component.html',
  styleUrls: ['./crud.component.css']
})
export class CRUDComponent {

  http = inject(HttpClient);

  SubmitData(Data: { fisratname: string, lastname: string, gender: string }) {

    this.http.post("http://localhost:3000/posts", Data).subscribe((res) => {
      console.log(res)
    });

  }
```

```ts
@Component({
  selector: 'app-crud',
  templateUrl: './crud.component.html',
  styleUrls: ['./crud.component.css']
})
export class CRUDComponent {

  http = inject(HttpClient);

  SubmitData(Data: Employee) {

    this.http.post("http://localhost:3000/posts", Data).subscribe((res) => {
      console.log(res)
    });

  }
```

Collecting Form Data to Insert

crud.component.ts U

AllConcepts > src > app > crud > crud.component.ts > ...

```ts
import { Component, inject } from '@angular/core';
import { HttpClient } from '@angular/common/http';

interface Employee {
  firstname: string,
  lastname: string,
  gender: string
}
```

crud.component.ts U

AllConcepts > src > app > crud > crud.component.ts > ...

```ts
export class CRUDComponent {

  http = inject(HttpClient);

  EmployeeData: Employee[] = [];

  FetchData() {
    this.http.get<Employee[]>("http://localhost:3000/posts").subscribe((response) => {
      console.log(response)
      this.EmployeeData = response;
    })
  }

  ngOnInit() {
    this.FetchData();
    console.log(this.EmployeeData);
  }
}
```

crud.component.html U

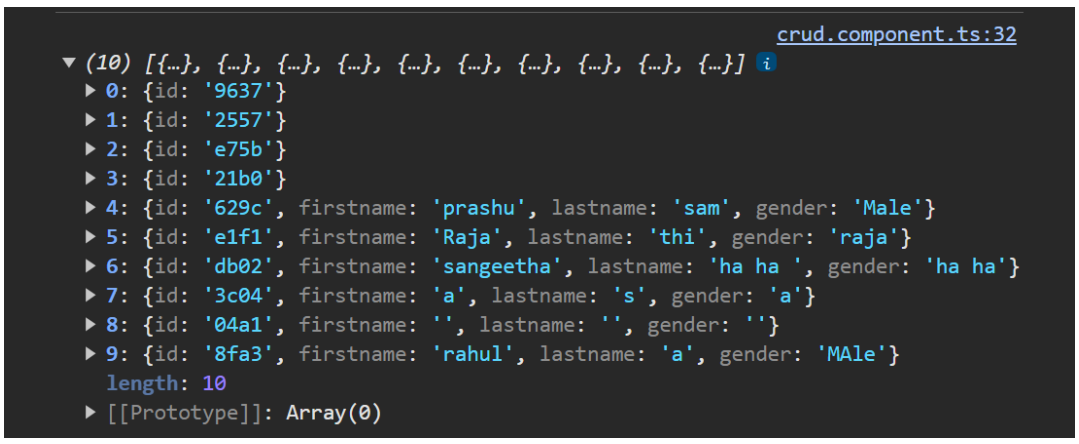
AllConcepts > src > app > crud > crud.component.html > ...

Go to component

```html
<div class="Display">
  <table>
    <thead>
      <th>firstName</th>
      <th>lastName</th>
      <th>gender</th>

    </thead>

    <tbody>
      <tr *ngFor="let data of EmployeeData">
        <td>{{data.firstname}}</td>
        <td>{{data.lastname}}</td>
        <td>{{data.gender}}</td>
      </tr>
    </tbody>
  </table>
</div>
```

Submit

firstName	lastName	gender
prashu	sam	Male
Raja	thi	raja
sangeetha	ha ha	ha ha
a	s	a
rahul	a	MAle



DELETING ALL THE DATA USING ID

The `DeleteAll()` method you've written is only clearing the local array `EmployeeData` but is **not sending a DELETE request to the server** to remove the records from the database (in this case, your JSON server).

To delete all data on the server, you would need to iterate over each employee in `EmployeeData` and make a DELETE request for each record on the server.

Here's how you can update `DeleteAll()` to delete all records from the server:

Copy code

```
DeleteAll() {
  // Iterate over each employee and send a DELETE request for each
  this.EmployeeData.forEach(employee => {
    this.http.delete(`http://localhost:3000/Post/${employee.id}`).subscribe(() => {
      console.log(`Deleted employee with id: ${employee.id}`);
    }, (error) => {
      console.error('Error deleting employee:', error);
    });
  });

  // Clear the local array after deleting all data
  this.EmployeeData = [];
}
```

Explanation:

- `forEach()` **loop**: This loops through all employees in the `EmployeeData` array, and for each employee, it sends an HTTP DELETE request to the server using the `id`.
- **Server-side deletion**: Each employee is deleted from the server, one by one.
- **Clear the local data**: After all the delete requests are sent, the local array `EmployeeData` is cleared.

Alternative: Bulk DELETE (if supported by API)

If your server supports bulk deletion (i.e., sending multiple `id`s in one request), you could handle it more efficiently, but JSON Server doesn't support bulk deletion natively, so the above approach (deleting one by one) is the standard method.

This way, your data is deleted both locally and on the server.

```
DeleteAll() {

  this.EmployeeData.forEach((employee) => {
    let url = `http://localhost:3000/Post/${employee.id}`;

    this.http.delete(url).subscribe(res => console.log(`Deleted employee with ${employee.id}`))
  })

  this.EmployeeData = [];
}
```

Certainly! Let's break down the logic of this code:

typescript Copy code

```
this.EmployeeData.forEach(employee => {
  this.http.delete(`http://localhost:3000/Post/${employee.id}`).subscribe(() => {
    console.log(`Deleted employee with id: ${employee.id}`);
  }, (error) => {
    console.error('Error deleting employee:', error);
  });
});
```

Step-by-Step Explanation:

1. `this.EmployeeData.forEach(employee => {...})`:
   - This line uses the `forEach` method to loop through every item (which is an `employee`) in the `EmployeeData` array.
   - The `employee` object represents each individual employee in the array during each iteration.

2. **Inside the loop**:
   - For each `employee`, it makes an HTTP DELETE request to the server using the `HttpClient` service's `delete` method.
   - The URL `http://localhost:3000/Post/${employee.id}` dynamically inserts the `id` of the employee into the URL.
   - This is the endpoint on your JSON server where each employee is stored, and the request is asking the server to delete the employee whose `id` matches the one in the URL.

3. **Making the DELETE request:**

```typescript
this.http.delete(`http://localhost:3000/Post/${employee.id}`).subscribe(...);
```

- This is an asynchronous HTTP request that sends a DELETE request to the server, asking it to delete the resource (employee) identified by `employee.id`.
- The `subscribe()` method is used to handle the response from the HTTP request. When the request completes successfully or fails, we can perform certain actions inside `subscribe()`.

4. **Handling the Success Response:**

```typescript
subscribe(() => {
  console.log(`Deleted employee with id: ${employee.id}`);
}, (error) => {
  console.error('Error deleting employee:', error);
});
```

- `subscribe(() => {...})`: The first argument inside `subscribe()` is a function that runs when the DELETE request is successful. In this case, it logs a message to the console that the employee has been deleted using `console.log()`.
- `subscribe((error) => {...})`: The second argument is a function that runs if the request fails. If there's an error deleting the employee, it logs the error to the console using `console.error()`.

Key Points of the Logic:

- **Asynchronous deletion**: The `forEach()` loops through all employees in the `EmployeeData` array, and for each one, an individual HTTP DELETE request is sent to the server.
- **Concurrent requests**: Each HTTP request runs asynchronously, meaning they all start almost at the same time without waiting for the previous one to finish. Once the request for a specific employee is successful, the success function logs the deletion.
- **Error handling**: If there's any problem during the deletion of a particular employee, an error message is logged.

Summary:

This logic loops through all employees in `EmployeeData` and sends a DELETE request for each one to the server. Once an employee is successfully deleted on the server, it logs a message. If something goes wrong during the deletion, it logs an error message. Each HTTP DELETE request happens independently and asynchronously.

crud.component.ts U

AllConcepts > src > app > crud > crud.component.ts > ...

```ts
import { Component, inject } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { map } from 'rxjs';

interface Employee {
  id: string,
  firstname: string,
  lastname: string,
  gender: string
}
```

crud.component.ts U

AllConcepts > src > app > crud > crud.component.ts > ...

```ts
  deleteAll() {
    this.EmployeeData.forEach((employee) => {
      const url = `http://localhost:3000/posts/${employee.id}`;

      this.http.delete(url).subscribe(() => {
        console.log(`Deleted employee data with ${employee.id}`);
      });
    });

    this.EmployeeData = [];
  }
}

```

crud.component.html U

AllConcepts > src > app > crud > crud.component.html > ...

```html

<div class="Display">
  <table>
    <thead>
      <th>firstName</th>
      <th>lastName</th>
      <th>gender</th>

    </thead>

    <tbody>
      <tr *ngFor="let data of EmployeeData">
        <td>{{data.firstname}}</td>
        <td>{{data.lastname}}</td>
        <td>{{data.gender}}</td>
      </tr>

      <button (click)="deleteAll()">Delete</button>
    </tbody>
  </table>

</div>
```

DELETING Particular DATA One at a Time USING ID

How it is possible
JSon is returning some id property randomly i haven't declared in interface before but when i include id in interface i can access that property from json How

```
[
 {
   "id": "1fbc",
   "prodId": "1",
   "prodName": "raju",
   "prodDesc": "as",
   "prodPrice": "2000"
 },
```

Great question! Here's what's happening:

1. **JSON Server Auto-Generates an `id` Field**: When you're using JSON Server, it automatically generates an `id` field for each object if it's not provided. This `id` is unique for each entry and helps JSON Server perform operations like `GET`, `DELETE`, and `PUT` by referencing the unique resource.
2. **TypeScript Doesn't Automatically Recognize Fields**: In TypeScript, interfaces define the shape of your objects at compile time. If a property (like `id`) is not declared in your interface, TypeScript will throw an error when you try to access it. It assumes the object only has the properties you declare in the interface.

3. **Including `id` in the Interface**: By adding `id` to the `Product` interface, you're telling TypeScript that objects of this type will also have an `id` property. This allows you to access the `id` field, which is returned by the JSON Server, without TypeScript complaining.

Key Point:

- **JSON Server** returns an `id` field because it auto-generates it.
- **TypeScript** needs that field to be declared in the `Product` interface, so you can access it without errors.

So, when you include the `id` in the interface, you're essentially making TypeScript aware of that property, even though JSON Server already returns it. This is why you need to adjust the interface to match the actual data structure returned by the server.

delete.component.ts U

CRUD > src > app > delete > delete.component.ts > DeleteComponent

```ts
import { CommonModule } from '@angular/common';
import { HttpClient, HttpClientModule } from '@angular/common/http';
import { Component, inject } from '@angular/core';
import { FormsModule } from '@angular/forms';

interface Product {
  id: string,
  prodId: string,
  prodName: string,
  prodDesc: string,
  prodPrice: string
}
```

```ts
http = inject(HttpClient);
ProductData: Product[] = [];

getData() {
  this.http.get<Product[]>("http://localhost:3000/Post").subscribe((res) => {
    this.ProductData = res;
  },
    (error) => {
      console.error('Error fetching data:', error);
    })
}

ngOnInit() {
  this.getData();
}

// DELETE
deleteData(id: string) {
  this.http.delete(`http://localhost:3000/Post/${id}`).subscribe((emp) => {
    this.ProductData = this.ProductData.filter((data) => data.id !== id);
    console.log(`Deleted product with id: ${id}`);
  });
}
```

```html
<h1>DELETE Data</h1>

<div class="container">

  <table>
    <thead>
      <th>Id</th>
      <th>Name</th>
      <th>Desc</th>
      <th>Price</th>
    </thead>

    <tbody>
      <tr *ngFor="let data of ProductData">
        <td>{{data.prodId}}</td>
        <td>{{data.prodName}}</td>
        <td>{{data.prodDesc}}</td>
        <td>{{data.prodPrice}}</td>
        <td><button (click)="deleteData(data.id)">Delete</button></td>
      </tr>
    </tbody>
  </table>
</div>
```

The Properties inside interface and server should match

localhost:3000/Post

Pretty-print


```
[
  {
    "id": "6c54",
    "prodId": "1",
    "prodName": "Naruto",
    "prodDesc": "desc",
    "prodPrice": "2000"
  },
  {
    "id": "0f3b",
    "prodId": "2",
    "prodName": "raju",
    "prodDesc": "a",
    "prodPrice": "11"
  },
  {
    "id": "bd71",
    "prodId": "45",
    "prodName": "prash",
    "prodDesc": "asaks",
    "prodPrice": "2000"
  }
]
```

crud.component.ts U

AllConcepts > src > app > crud > crud.component.ts > CRUDComponent > deleteSingle

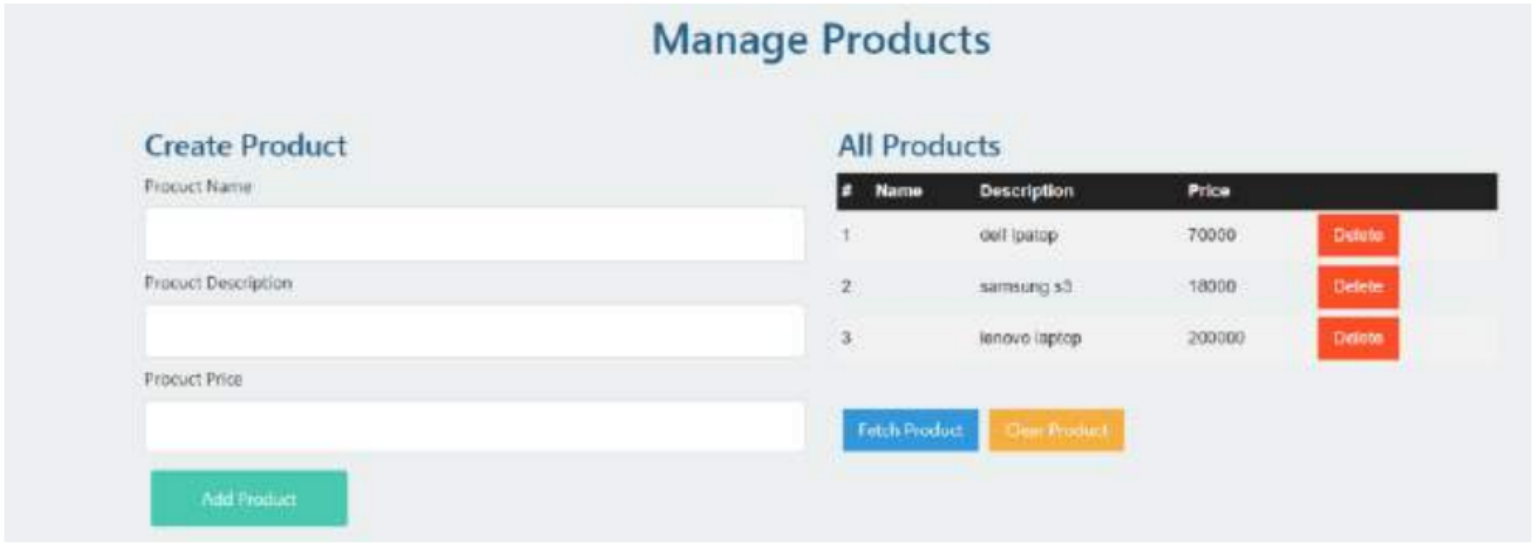
```ts
53
54    deleteSingle(id: string) {
55
56      const url = `http://localhost:3000/posts/${id}`;
57
58      this.http.delete(url).subscribe(() => {
59        this.EmployeeData = this.EmployeeData.filter((empVal)
          => empVal.id !== id);
60        console.log(`deleted employee is ${id}`)
61      })
62
63    }
64  }
65
```

crud.component.html U

AllConcepts > src > app > crud > crud.component.html > div.Display

```html
13
14  <div class="Display">
15    <table>
16      <thead>
17        <th>firstName</th>
18        <th>lastName</th>
19        <th>gender</th>
20
21      </thead>
22
23      <tbody>
24        <tr *ngFor="let data of EmployeeData">
25          <td>{{data.firstname}}</td>
26          <td>{{data.lastname}}</td>
27          <td>{{data.gender}}</td>
28          <td><button (click)="deleteSingle(data.id)">Delete</
            button></td>
29        </tr>
30
31        <button (click)="deleteAll()">Delete</button>
32      </tbody>
33    </table>
```

Try to build this



UPDATE

Goku x vegeta AMV EDIT #anime #sasukefanart #naruto #animeartgallery #sa...