

# Building a Full-Stack Web Application with Angular and Node.js: A Hands-On Demo

## Prerequisites

Before we get started, you'll need to have the following installed:

- Node.js
- Angular CLI

## Creating the Node.js backend

First, let's create a simple Node.js backend that serves data from a JSON file. Next, create a new directory for your project and open a terminal in that directory.

1. Initialize a new Node.js project by running "**npm init -y**" and following the prompts.
2. Install the modules using the "**npm install body-parser cors express**" command.
3. Create a new file called `server.js` in your project directory.

Add the following code to `server.js`:



```
const express = require("express");
const app = express();
const bodyParser = require("body-parser");
const fs = require("fs");

app.use(bodyParser.json());
```

## 1. End-point to get data.

A code editor window with a dark background and light-colored text. It contains a JavaScript function definition for an Express.js route. The code is as follows:

```
app.get("/api/items", (req, res) => {  
  const items = JSON.parse(fs.readFileSync("data.json", "utf8"));  
  res.send(items);  
});
```

A green circular icon with a white 'G' is located in the bottom right corner of the code editor window.

This code defines a route using the Express.js framework for handling HTTP requests. Specifically, it defines a GET route for the path `/api/items`. When a client sends a GET request to this path, the code inside the function is executed.

The first line of the function uses the built-in `fs` (file system) module in Node.js to read the contents of a file named `data.json`. Specifically, it uses the `readFileSync` method to read the file synchronously and parse its contents using `JSON.parse`. This assumes that `data.json` file exists in the same directory as the server code.

The parsed contents of the file are then sent back to the client using the `res.send` method. In this case, the contents of the `items` array are sent as the response.

So, when a client sends a GET request to the `/api/items` path, the server reads the contents of `data.json` file and sends its contents as the response, which is expected to be an array of items in JSON format.

## 2. End-point to get data by ID.

```
app.get("/api/items/:id", (req, res) => {  
  const id = parseInt(req.params.id);  
  const items = JSON.parse(fs.readFileSync("data.json", "utf8"));  
  const item = items.find((item) => item.id === id);  
  if (item) {  
    res.send(item);  
  } else {  
    res.status(404).send("Item not found");  
  }  
});
```

This code defines another route using the Express.js framework for handling HTTP requests. Specifically, it defines a GET route for the path `/api/items/:id`. The `:id` in the path is a parameter that can be used to match specific items in the `"data.json"` file.

The code inside the function is executed when a client sends a GET request to this path with an `"id"` parameter. The `"id"` parameter is extracted from the request object using `req.params.id` and converted to an integer using `parseInt`.

The code then reads the contents of the `"data.json"` file using the `"fs"` module and parses the contents using `JSON.parse` to create an array of items. The `"find"` method is then used to find an item in the array whose `"id"` property matches the `"id"` parameter from the request.

If the `"item"` is found, the server sends it back to the client using the `"res.send"` method. Otherwise, the server sets the status code to 404 (Not Found) using the `"res.status"` method and sends an error message `"Item not found"` using the `"res.send"` method.

In summary, this route allows clients to retrieve a specific item from the `"data.json"` file by providing its `"id"` as a parameter in the request URL. If the item is found, it is sent back as a response. Otherwise, an error message is sent with a 404 status code.

### 3. End-point to delete data by ID.

```
app.delete("/api/items/:id", (req, res) => {  
  const id = parseInt(req.params.id);  
  const items = JSON.parse(fs.readFileSync("data.json", "utf8"));  
  const index = items.findIndex((item) => item.id === id);  
  if (index !== -1) {  
    items.splice(index, 1);  
    fs.writeFileSync("data.json", JSON.stringify(items));  
  }  
  res.send({});  
});
```

This code defines a DELETE route using the Express.js framework for handling HTTP requests. Specifically, it defines a DELETE route for the path `/api/items/:id`. The `:id` in the path is a parameter that can be used to match specific items in the `"data.json"` file.

When a client sends a DELETE request to this path with an `"id"` parameter, the code inside the function is executed. The `"id"` parameter is extracted from the request object using `req.params.id` and converted to an integer using `parseInt`.

The code then reads the contents of the `"data.json"` file using the `"fs"` module and parses the contents using `JSON.parse` to create an array of items. The `"findIndex"` method is then used to find the index of an item in the array whose `"id"` property matches the `"id"` parameter from the request.

If the `"index"` is not `-1`, indicating that the item is found, the code uses the `"splice"` method to remove the item from the array. The modified array is then written back to the `"data.json"` file using the `"fs"` module and the `"writeFileSync"` method.

Finally, the server sends an empty object as the response using the `"res.send"` method.

In summary, this route allows clients to delete a specific item from the `"data.json"` file by providing its `"id"` as a parameter in the request URL. If the item is found, it is removed from the array and the modified array is written back to the `"data.json"` file. The server sends an empty object as the response.

## 4. End-point to delete data by ID.

```
app.put("/api/items/:id", (req, res) => {  
  const id = parseInt(req.params.id);  
  const items = JSON.parse(fs.readFileSync("data.json", "utf8"));  
  const index = items.findIndex((item) => item.id === id);  
  if (index !== -1) {  
    items[index] = req.body;  
    fs.writeFileSync("data.json", JSON.stringify(items));  
  }  
  res.send({});  
});
```

This code defines a PUT route using the Express.js framework for handling HTTP requests. Specifically, it defines a PUT route for the path `/api/items/:id`. The `:id` in the path is a parameter that can be used to match specific items in the `"data.json"` file.

When a client sends a PUT request to this path with an `"id"` parameter and a request body containing a JSON object representing an item, the code inside the function is executed. The `"id"` parameter is extracted from the request object using `"req.params.id"` and converted to an integer using `"parseInt"`.

The code then reads the contents of the `"data.json"` file using the `"fs"` module and parses the contents using `JSON.parse` to create an array of items. The `"findIndex"` method is then used to find the index of an item in the array whose `"id"` property matches the `"id"` parameter from the request.

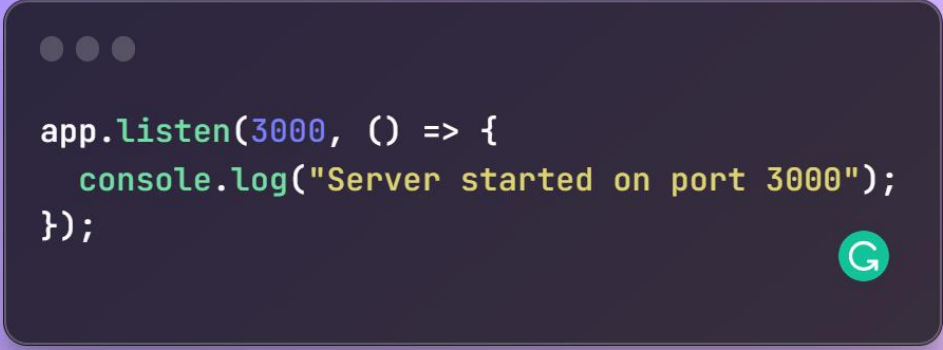
Suppose the `"index"` is not `-1`, indicating that the item is found. In that case, the code replaces the item at the corresponding index in the array with the request body JSON object using the assignment operator.

The modified array is then written back to the `"data.json"` file using the `"fs"` module and the `"writeFileSync"` method.

Finally, the server sends an empty object as the response using the `"res.send"` method.

In summary, this route allows clients to update a specific item in the `"data.json"` file by providing its `"id"` as a parameter in the request URL and sending a request body containing a JSON object

representing the updated item. If the item is found, it is replaced with the updated item in the array and the modified array is written back to the "data.json" file. The server sends an empty object as the response.



```
app.listen(3000, () => {  
  console.log("Server started on port 3000");  
});
```

## Listen on Port: 3000

This code calls the listen method on the Express.js application object to start a web server on port 3000.

The listen method takes two arguments: the port number to listen on and a callback function to execute when the server starts listening for requests. In this case, the port number is set to 3000, and the callback function logs a message to the console indicating that the server has started.

When the server is started, it will listen for incoming HTTP requests on port 3000. Any requests that match the routes defined in the application will trigger the corresponding functions to handle those requests and send responses back to the clients.

**To start the server, use the command `node server.js`**

## Creating the Angular Front-end

1. Open a terminal and navigate to the directory where you want to create your Angular app.
2. Run the following command to create a new Angular app: `ng new my-app`
3. Change into the app directory: Change into the app directory:
4. Run the following command to start the Angular development server: `ng serve`.
5. Open your browser and navigate to `http://localhost:4200`. You should see the default Angular app running.

**Now we will need to add Bootstrap to the application.**

**To add Bootstrap to an Angular project, follow these steps:**

1. Install Bootstrap: Open the terminal and navigate to the root directory of your Angular project. Then run the following command: `npm install Bootstrap`
2. Import Bootstrap: Open the `styles.css` file in the `src` directory of your project and add the following line at the top of the file:  
`@import "~bootstrap/dist/css/bootstrap.min.css";`

This line tells Angular to import the Bootstrap CSS file from the `node_modules` directory.

3. Include Bootstrap JavaScript: If you want to use Bootstrap's JavaScript components (like modals, tooltips, etc.), you must include the Bootstrap JavaScript file. Open the `angular.json` file in the root directory of your project and add the following lines to the `scripts` array:

```
"node_modules/jquery/dist/jquery.min.js",  
"node_modules/bootstrap/dist/js/bootstrap.min.js"
```

This tells Angular to include the jQuery and Bootstrap JavaScript files from the `node_modules` directory.

That's it! You should now be able to use Bootstrap in your Angular project. Note that if you're using a newer version of Angular, you may need to use the `angular.json` file to configure your project instead of the `styles.css` file.

**Tasks:**

1. Create a navigation bar
2. Create a three-column grid page

## Angular service

To create a service in Angular that calls your API to retrieve data and populates it in a table, you can follow these steps:

Generate a new service using the ng generate service item

1. This will create a new item.service.ts file in the src/app directory.
2. In item.service.ts, import HttpClient from @angular/common/http and inject it into the constructor:

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

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

  constructor(private http: HttpClient) { }

}
```



3. Add a method to the ItemService class that calls your API using HttpClient. In this example, the method is named `getItems()`:

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

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

  constructor(private http: HttpClient) { }

  getItems() {
    return this.http.get('/api/items');
  }

}
```

4. In the component where you want to display the table, inject the ItemService and call the getItems() method. In this example, we'll assume the component is named ItemListComponent:

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

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

  private baseUrl = 'http://localhost:3000'; // Add this line

  constructor(private http: HttpClient) { }

  getItems() {
    return this.http.get(`${this.baseUrl}/api/items`); // Update this line
  }
}
```

5. In the ItemListComponent template, use the ngFor directive to iterate over the items array and display the data in a table:

```
<table>
  <thead>
    <tr>
      <th>ID</th>
      <th>Name</th>
      <th>Description</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let item of items">
      <td>{{ item.id }}</td>
      <td>{{ item.name }}</td>
      <td>{{ item.description }}</td>
    </tr>
  </tbody>
</table>
```

