

The MEAN Stack: A starters guide

What this article covers

In this article I'll cover the MEAN technology stack in general, how to set it up and build an example application – which you can code along! Let's dive in.

What's the MEAN stack?

In one sentence: The MEAN stack is a combination of frameworks used to build full stack web applications. It is JavaScript based, so every architecture layer uses it to bring the application to life – which can have many benefits, developer experience wise, but also business wise.

The acronym MEAN stands for the following:

- MongoDB
- Express
- Angular
- Node.js

Each letter represents one layer: MongoDB as the primary database, Express and Node act as the backend and Angular makes up for the frontend. The stack and its variations are very popular, especially with JavaScript developers. Now, let us have a look at these technologies in detail.

The architecture of MEAN

MongoDB as the primary database

MongoDB is a noSQL document database, that stores Data in flexible JSON documents. It has over 85 million downloads and is almost the go-to choice for most developers when it comes to noSQL databases.

A big community and being able to host the database either locally with docker or with your favourite cloud provider makes Mongo a great choice as a database.

Express and Node.js for the backend

Sitting on top of the database, we have our backend server hosted with Node.js and an Express API.

Node.js is a JavaScript runtime environment mostly acting as a server for such applications. The 'feature' making Node outstanding is that it enables JavaScript to run outside of web-browsers.

It's open-source and runs on almost any operating system, if you already have developed some JavaScript on your device, you have it installed too!

Express enables our app to talk with our frontend and the outer world, it's our API – also based on JavaScript. Powering web applications since 2010 hand in hand with Node.js, its service is crucial for the MEAN stack.

Angular as the frontend

Firstly developed and published by Google in 2016 – as a rewrite of AngularJS, it is a TypeScript/JavaScript frontend framework for building small projects and scalable enterprise applications.

Being TypeScript first, safe, and robust development is ensured with strong types – making it extremely popular in enterprise applications.

The framework handles our complete client-side application including styling and functionality.

And something special about this guide: we'll work with **Angular 17**, which has a lot of updates and new features!

Variations of the MEAN stack

The frontend framework is the most opinionated discussion around web developers (JavaScript framework war) and caused some variations of the MEAN stack.

The framework for the frontend is the only thing that gets switched out when working with variations of MEAN.

The most popular variations are:

- MERN – React
- MEVN - Vue

or with any other frontend framework you like, such as:

- MESN – Svelte

But for this article, we'll stick with the original Angular.

Who uses the MEAN stack?

I previously mentioned that MEAN is pretty popular, but which companies do actually use it? Well, here's some examples:

- Google (Gmail, Play Store, G Suite)
- Forbes

- Paypal (for its developer portal)
- UpWork

If you were to include the MERN stack, you'd have even more industry leaders on that list. That is because React gained a lot of popularity in the last 3 years.

Pros of the MEAN stack

Only one language

Having one language for all layers makes it easier to build and maintain the system as only a developer proficient in JavaScript/TypeScript is needed – and not one which knows many technologies, which may be hard and expensive to find for a business.

Writing the code in one language is also good for the developer itself as it creates consistency throughout the project, and you don't have to switch technologies every time you work on different architecture layers. That increases your productivity, and you can become a very advanced JavaScript developer when working a lot with the technology.

Big community and well-supported

Angular, Express and MongoDB have a very big community, and they have been around for a long time which tells that these frameworks have been battle tested – which is great when developing, because you won't have to migrate your system every time requirements change.

Not only the community is big, also the backers. Having one of the biggest tech corporations worldwide maintaining Angular is absolutely great. This ensures high-quality service and security which is crucial for developing large scale applications.

Moreover, MongoDB is being looked after by MongoDB Inc., the company that founded the database platform – a big pro, because the database really matters.

Cons of the MEAN stack

Not everything is perfect, even the MEAN stack! Sadly, it still has some flaws which need to be considered when choosing it for a project.

No types by default

JavaScript itself does not require you to use types which can cause a lot of chaos in your code – imagine using a data object and having to guess in which format the values are, kinda hard isn't it?

Of course, that problem is solved by TypeScript – but it is not the default and there are many projects which don't use it (just because they don't have to). That makes the code hard to maintain, test and extend, especially for developers that don't know your codebase that well.

Logic isolation

Since we have a tight connected business and server logic it is pretty hard to separate them – sometimes causing ☹️(spaghetti)-code, something that we do not really want for scalable, testable and maintainable applications.

How to set up a MEAN application

I think you got the core concepts and ideas of MEAN, so let us now have a look on how to set up a walking-skeleton application! Please note that this is an opinionated guide – you could structure the project differently as well.

What you'll need to follow along

Before starting to read the guide make sure you have the following things ready:

- Your favourite IDE (e.g. VS Code, IntelliJ)
- Installed Docker Desktop
- Downloaded and installed Node.js
- Optional: A git repository to publish your code
- Optional: VS Code MongoDB extension

If you got everything ready, go on reading!

Setup MongoDB Docker container

First of all, we need the database. For that we'll spin up a Docker container with a MongoDB image.

☹️ **Note:** You can also host the database on MongoDB Atlas – but this is not covered in this article.

Start Docker Desktop and then open your terminal. To pull the latest image from mongo, enter the following:

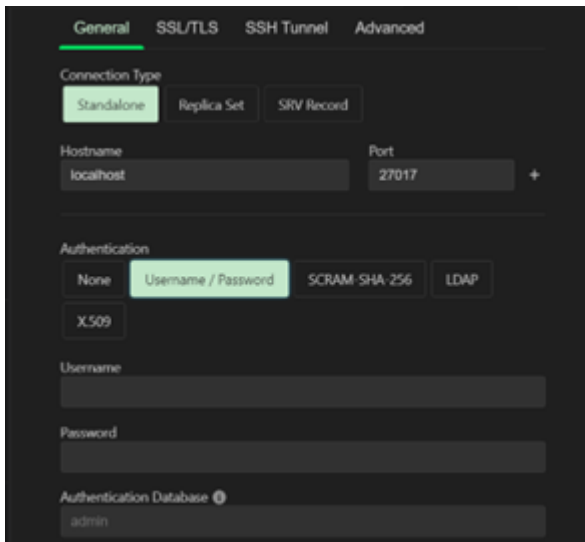
```
docker pull mongo:latest
```

And then run the container with a name you want.

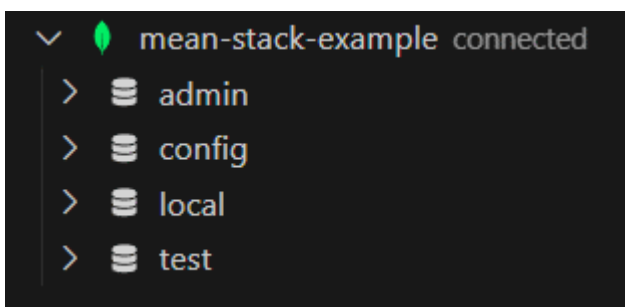
```
docker run -d -p 27017:27017 --name=mean-stack-example-mongodb -e  
MONGO_INITDB_ROOT_USERNAME=user -e MONGO_INITDB_ROOT_PASSWORD=password mongo:latest
```

To bind the container port to the machine port we use the `-p` (port) option. We also want to secure our database with a root admin user, which can be set with `-e` (environment variable) for username and password.

To test if the container works, you can open up VS Code and connect to our database using the MongoDB extension. Just click on the 'Add a connection' button and choose 'Connection Settings' – which will open a form to specify the connection.

The image shows the 'Connection Settings' dialog for MongoDB in VS Code. It has four tabs: 'General', 'SSL/TLS', 'SSH Tunnel', and 'Advanced'. The 'General' tab is active. Under 'Connection Type', 'Standalone' is selected. The 'Hostname' is 'localhost' and the 'Port' is '27017'. Under 'Authentication', 'Username / Password' is selected. There are input fields for 'Username' and 'Password'. The 'Authentication Database' is set to 'admin'.

Enter the username and password from the docker run command and hit connect! Now you should see the connection in the sidebar and its databases when expanding the toggle.



If you want to create a test database with a simple insert, create a new playground and enter the following:

```
use('test');  
db.createCollection('testCollection');  
db.testCollection.insertOne({name:"hey there!"})
```

That's all you need to know for now. Let us go on and set up the application.

Set up the server-side application

For the sake of simplicity, I will use a mono-repository approach for this guide, but that's not a must – feel free to adapt it to your needs.

Open an empty folder/repository in your IDE (the project folder) and create the folders for the backend:

- `server`
- `server/src`

The src folder is where our Express files will go. Now initialize a node project inside the `server` directory.

```
cd server
npm init
```

And since we want to work with safe types, add a `tsconfig.json` and a `.env` file for the connection strings to the folder (Leave it empty for now).

We need some npm packages for our Express.js REST API, so install them in the server directory. And because we are working with typescript, we also need to add the types for the packages.

```
npm install cors dotenv express mongodb
npm install --save-dev typescript @types/cors @types/express @types/node ts-node
```

We need `mongodb` to connect our application to the database, and the `express` package to create our api. `dotenv` and `cors` are for our communication with the frontend over the api.

And lastly, we need to add some configuration to the `tsconfig.json` file – typescript uses this config to compile the code we write.

mean-stack-example-app/server/tsconfig.json

```
{
  "compilerOptions": {
    "module": "commonjs",
    "esModuleInterop": true,
    "target": "es6",
    "noImplicitAny": true,
    "moduleResolution": "node",
    "sourceMap": true,
    "outDir": "dist",
    "baseUrl": ".",
    "allowJs": true,
    "paths": {
```

```
    "": ["node_modules/*"]
  },
  "include": ["src/**/*"]
}
```

That's all for the basic configuration. To see how to connect to the database and add an express route, read on to the example project.

Set up the Angular client

In the root folder (`mean-stack-example-app`) of the project we need to install the Angular cli with the following command:

```
cd ..
npm install -g @angular/cli
```

After successful installation, again in the root directory, create a new Angular application:

```
ng new client --routing --style=css
```

When the initialization is done, we'll have a client folder where the Angular app is located. We need the `--routing` flag, so that a routing module is being generated (for the individual pages of the application). `--style=css` makes css our default preprocessor.

Now you can start the application by entering the following in your terminal:

```
cd client
ng serve
```

That's everything for setting up the application. If you want to figure out on your own how to connect to the database, build express endpoints and create the client pages - feel free to do so now!

For everyone that's working with MEAN for the first time, the following example project is exactly what you need to learn the stack - go on reading!

Let's build an example project!

The best way to learn is practice – that's why we'll build a recipe book with the MEAN stack! It is a simple CRUD-app, here a sneak-peek on what it will look like:

Tasty Recipe List

by Andy

Add a new Recipe +

Spaghetti Bolognese

Edit

Delete

main course

Classic Italian pasta dish with a rich meat sauce.

200 grams Spaghetti
300 grams Ground beef
400 ml Tomato sauce...

Pancakes

Edit

Delete

breakfast

Fluffy and delicious pancakes for a perfect breakfast.

1 cup All-purpose flour
1 cup Milk
1 piece Egg...

Brownies

Edit

Delete

dessert

Indulgent chocolate brownies for a sweet treat.

1 cup Butter
2 cups Granulated sugar
4 pieces Eggs...

Add document to MongoDB

Firstly, we'll need a document in our database where we can store the recipes. For that, open up a playground in your VS Code MongoDB extension or your preferred database manager and add a document.

```
use('test');  
db.createCollection('recipes');
```

That's all we need to do on the database level directly.

Create interfaces on server side for type safety

Now let us move on to the server side application. Firstly, we need to define what a recipe looks like by creating a TypeScript interface for it.

mean-stack-example-app/server/src/recipe.ts

```
import * as mongodb from "mongodb"  
import { Ingredient } from "../ingredient";  
  
export interface Recipe {  
  title: string;  
  description: string;  
  category: "breakfast" | "main course" | "snack" | "dessert";  
  ingredients: Ingredient[];  
  instructions: string;  
  _id?: mongodb.ObjectId;  
}
```

I decided that a recipe has a title, brief description, category, ingredients (array of Ingredient

interface), instructions and an optional `_id` which will be generated by MongoDB automatically.

mean-stack-example-app/server/src/ingredient.ts

```
import * as mongodb from "mongodb"

export interface Ingredient {
  name: string;
  quantity: number;
  unit: number;
  _id?: mongodb.ObjectId;
}
```

An ingredient consists of a name, quantity and unit. And again the optional `_id`. That's it for our data model - not that complex, but a comprehensive model is not the main aim of this guide.

Connect Express to the database

The next step is to connect our backend application to the database.

For that, create a `database.ts` file in the `src/` folder of the server and add the following code:

mean-stack-example-app/server/src/database.ts

```
import * as mongodb from "mongodb";
import { Recipe } from "../recipe";

export const collections: {
  recipes?: mongodb.Collection<Recipe>;
}={};

export async function connectToMongoDb(uri:string) {
  const client = new mongodb.MongoClient(uri);
  await client.connect();

  const db = client.db("test");

  const recipesCollection = db.collection<Recipe>("recipes");
  collections.recipes = recipesCollection;
}
```

In this file, I am exporting a `collections` constant, which contains the recipes collection from the database. If your database model has more collections, you can reference them in this object.

The `connectToMongoDb` function handles the connection to the database with the native `MongoClient`, developed by MongoDB. When calling this method you need to add a `uri` parameter, which is the

connection string to the database.

Because I have my collection in the `test` database I need to reference it in the `db` constant. Lastly, I am pulling the `recipes` collection from the database and set it to the `recipes` constant.

Now we need to call the function in our `server.ts` file and start the express server.

mean-stack-example-app/server/src/server.ts

```
import * as dotenv from "dotenv"
import express from "express"
import { connectToMongoDb } from "../database"
import { error } from "console";
import { recipeRouter } from "../recipe.routes";

dotenv.config();

const {CONNECTION_URI, EXPRESS_PORT} = process.env;

if (!CONNECTION_URI) {
  console.error("Missing connection URI in .env");
  process.exit(1);
}

if (!EXPRESS_PORT) {
  console.error("Missing express port in .env");
  process.exit(1);
}

connectToMongoDb(CONNECTION_URI)
.then(()=>{
  const app = express();
  const cors=require('cors');
  app.use(cors({
    origin: 'http://localhost:4200', // Replace with your Angular app's URL
    methods: 'GET,HEAD,PUT,PATCH,POST,DELETE',
    credentials: true,
  }));
  app.listen(EXPRESS_PORT,()=>{
    console.log(`Server running on localhost:${EXPRESS_PORT}`);
  })
})
.catch(error=> console.error(error));
```

I am referencing two variables from my `.env` file, firstly the connection string to the database and secondly, the port I want to run my express server on. Please adjust the variables according to your credentials.

The `.env` file could look something like this:

```
CONNECTION_URI=mongodb://user:password@localhost:27017/?authSource=admin&readPreference=primary&ssl=false&directConnection=true
EXPRESS_PORT=5200
```

□ **Tip:** You can obtain the connection string of your database by right-clicking on the connection in the VS Code MongoDB extension → **Copy Connection String**.

To prevent basic errors, I am checking if the variables do exist, and then proceed to configure cors. This is a must, otherwise you won't be able to make requests from your Angular client. Then, the `connectToMongoDb` function with the `CONNECTION_URI` as a parameter is called. If the connection was successful, an Express server will be started on the specified port.

Let's see if this works. Enter this command to start the server:

& don't forget to start your docker container □

```
cd server
npx ts-node src/server.ts
```

Which gives the following output (if successful):

```
Server running on localhost:5200
```

Great, that's it for the database connection! Let's move on to the backend implementation.

Create REST Endpoints

Firstly, let us create the endpoints to fetch/create/update/delete data from the database. For that, I'll use the router from express in the `recipe.routes.ts` file.

The endpoints needed:

- `/` → GET all recipes
- `/:id` → GET one recipe by its id
- `/` → POST create a recipe
- `/:id` → PUT update a recipe by its id
- `/:id` → DELETE a recipe by its id

Start by exporting a constant of the router and tell it to use json.

mean-stack-example-app/server/src/recipe.routes.ts

```
import * as express from "express";
import * as mongodb from "mongodb";
import { collections } from "./database";

export const recipeRouter = express.Router();
recipeRouter.use(express.json());
```

GET all/one recipe(s)

To add a new route to the router, you need to call the REST-operation method on the router object and write an arrow function, which contains the logic.

mean-stack-example-app/server/src/recipe.routes.ts

```
recipeRouter.get("/", async (_req, res) => {
  try {
    const recipes = await collections.recipes.find({}).toArray();
    res.status(200).send(recipes);
  } catch (error) {
    res.status(500).send(error.message);
  }
});
```

Getting all recipes is fairly simple. I call the `get()` method on the router with the url and the arrow function to be executed.

In the try block I am using my `collections.recipes` instance from the `database.js` file, to execute a find query on the collection. The query itself is empty, because I want all my recipes.

If everything went fine, we receive all recipes in an array along with a status code of 200.

To receive just one recipe by its id, add the following to your `recipe.routes.ts`

mean-stack-example-app/server/src/recipe.routes.ts

```
recipeRouter.get("/:id", async (req, res) => {
  try {
    const id = req?.params.id;
    const recipe = await collections.recipes.findOne({_id: new
mongodb.ObjectId(id)});

    if (recipe) {
      res.status(200).send(recipe)
    } else {
      res.status(404).send(`No recipe with id: ${id}`);
    }
  } catch (error) {
    res.status(500).send(error.message);
  }
});
```

```
}  
});
```

To find a recipe with a specified id, you need to extract it from the request parameter and then use it in the query. Pay attention that you convert the id to an `objectId`, otherwise the request will fail, e.g. not find anything.

If something is returned from the query the method returns a 200 along with the requested recipe, else a 404 is sent (because no recipe exists with the id). Any other error runs into the catch, which sends a status code of 500.

POST Create a recipe

To create a recipe with a POST request, we need to create a new endpoint calling the `.post()` method on the router.

mean-stack-example-app/server/src/recipe.routes.ts

```
recipeRouter.post("/", async (req, res)=>{  
  try {  
    const recipeToInsert:Recipe = req.body;  
    const insertedRecipe = await collections.recipes.insertOne(recipeToInsert);  
    if(insertedRecipe.acknowledged) {  
      res.status(201).send(insertedRecipe)  
    } else {  
      res.status(500).json({error: "Failed to create recipe."});  
    }  
  } catch (error) {  
    res.status(400).json({error: error.message});  
  }  
})
```

The recipe the client wants to add lies in the body of the request, which we can access with the `.body` function. We then call the database and tell it to insert the requested recipe. If it works, we send back the `insertedRecipe` object, which contains the id of the newly created object. Otherwise, we catch the error and return it with a status code.

DELETE a recipe

Deleting is done by calling the `delete()` method on the router. We can create a method that deletes a recipe by its id:

mean-stack-example-app/server/src/recipe.routes.ts

```
recipeRouter.delete("/:id", async (req,res)=>{  
  try{  
    const id:string = req?.params.id;  
    const deletedRecipe = await collections.recipes.deleteOne({_id:new
```

```

mongodb.ObjectId(id)});

    if(deletedRecipe.deletedCount>0){
        res.status(202).send(deletedRecipe);
    } else if (deletedRecipe.deletedCount==0){
        res.status(404).json({error: "No recipe with id " + id});
    }
} catch (error) {
    res.status(400).json({error: error.message});
}
});

```

Again, we extract the id from the parameter and then call the database operation, which is a `deleteOne()` statement. When the `deletedCount` is greater than 0 we have successfully deleted the recipe and can send a 202 (Accepted).

If the count is 0, which means no recipe with the id exists, we return a 404 error. Any other error will be caught and sent back with a status code of 400, for example when the client sends an invalid id.

PUT Update a recipe

"Whoops there's a typo in my recipe - let me fix that by updating it".

That's what PUT requests are for, updating an existing entity in the datastore. In Express, we can do so by calling the `put()` operation on the router.

The put/update method for the recipes would look like this:

mean-stack-example-app/server/src/recipe.routes.ts

```

recipeRouter.put("/:id", async(req,res)=>{
    try {
        const id:string = req?.params.id;
        const recipeWithChanges:Recipe = req.body;
        const updatedRecipe = await collections.recipes.updateOne({_id:new
mongodb.ObjectId(id)}, {$set: recipeWithChanges});

        if (updatedRecipe.matchedCount>0 && updatedRecipe.modifiedCount>0){
            res.status(200).send(updatedRecipe);
        } else if (updatedRecipe.matchedCount==0){
            res.status(404).json({error: "No recipe with id " + id});
        } else if (updatedRecipe.matchedCount>0 && updatedRecipe.modifiedCount==0){
            res.status(304).send(updatedRecipe);
        }
    } catch (error) {
        res.status(400).json({error: error.message});
    }
});

```

One last time, we extract the id from the parameter and the new recipe version from the body. Then we run an `updateOne()` query on the recipe collection to apply the changes the client wants.

If the update was successful (found a recipe and made some changes), a status code of 200 and the query result is sent. If the query hasn't been able to find a recipe, a 404 is sent, and if the database didn't have to make any changes, because the new and old version are the same, a 304 (Not modified) is sent to the client.

And that's it for the routes, we have all the CRUD operations ready to be consumed by the frontend!

Register the routes

Just one more thing for the backend, then we can start building the Angular app ☐.

Currently, the Express server does not know about the recipe routes because we did not register them, so we need to help him out here.

In the `server.ts`, before the `app.listen()` method call, add:

mean-stack-example-app/server/src/server.ts

```
app.use("/recipes", recipeRouter);
```

Don't forget to restart your server after modifying this file!

Finally, the frontend (Angular)

For all my frontend gurus, now the fun part for you ☐. Leave your Express server and Docker container running and switch over to your client directory - that's where everything will happen now.

Create interfaces on client

To follow our type-safe principle, we need to establish types on the client as well. We already created interfaces for the server, and now we need them on our client too.

You might ask yourself, why don't we make a shared library? - that's because these interfaces defer a bit.

Either create a `recipe.ts` + `ingredient.ts` file in the `src/app/` directory, or generate one with this command:

```
ng generate interface recipe
ng generate interface ingredient
```

Open up the files and add the interface specification to them:

mean-stack-example-app/client/src/app/ingredient.ts

```
export interface Ingredient {  
  name?: string;  
  quantity?: number;  
  unit?: string;  
  _id?: string;  
}
```

mean-stack-example-app/client/src/app/recipe.ts

```
import { Ingredient } from "../ingredient";  
  
export interface Recipe {  
  title?: string;  
  description?: string;  
  category?: "breakfast" | "main course" | "snack" | "dessert" ;  
  ingredients?: Ingredient[];  
  instructions?: string;  
  _id?: string;  
}
```

Notice the difference from the server interfaces in the id attribute (it's just a string!) and all fields have a **?** so typescript doesn't throw any errors because of nullable fields.

Create service to communicate with Express API

Next we'll create an Angular service that handles the communication with our API, it separates the logic from the presentation layer and makes it reusable.

Using the **ng generate service recipe** command, we can automatically generate a boilerplate service class.

Before implementing the service, add the following to your **app.config.ts** file:

```
export const appConfig: ApplicationConfig = {  
  providers: [provideRouter(routes), provideClientHydration(),  
    provideHttpClient(withFetch())]  
};
```

Adding these providers enables the **HttpClient** and the **Router**.

In the service file, we can add the following methods to call our backend endpoints.

mean-stack-example-app/client/src/app/recipe.service.ts


```

import { Injectable } from '@angular/core';
import { Recipe } from './recipe';
import { HttpClient } from '@angular/common/http';
import { AddRecipeRequest } from './addRecipeRequest';

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

  private url:String = "http://localhost:5200/api/recipes";

  constructor(private http:HttpClient) { }

  getAllRecipes(){
    return this.http.get<Recipe[]>(this.url+"/");
  }

  getSingleRecipe(id:String){
    return this.http.get<Recipe>(this.url+"/"+id);
  }

  createRecipe(recipe:AddRecipeRequest){
    return this.http.post<unknown>(this.url+"/", recipe);
  }

  deleteRecipe(id:string){
    return this.http.delete<unknown>(this.url+"/"+id);
  }

  updateRecipe(id:string, recipe:AddRecipeRequest){
    return this.http.put<unknown>(this.url+"/"+id, recipe);
  }

}

```

And because we do not send an actual recipe as a request, you need a **AddRecipeRequest** TypeScript interface:

```

export interface AddRecipeRequest {
  title?: string | undefined;
  description?: string | undefined;
  category?: string | undefined;
  ingredients?: unknown[] | undefined;
  instructions?: string | undefined;
}

```

Since we're writing a service that is used by other components, the class needs to be annotated with

the `@Injectable` annotation. The constructor instantiates the `HttpClient` object, which handles the request logic - this is a feature from Angular directly.

Why do we need to call the module through the constructor? Because of dependency injection, a design pattern that is used by default in Angular. This pattern creates more flexibility and modularity in the app, which is great.

The rest of the service consists of the 5 methods for making calls to the API.

What each method does, should be clear from the function names. To use the HTTP client, you can simply write `this.http.[method]<type>` with the route and needed data for the request.

Compared to other JavaScript frameworks this approach is a super convenient way (and I think the best) to handle http requests equally across the app - no more `await fetch()`... calls spread across the components ☐.

Moreover, it's great that you can tell Angular of which type the response will be, taking care of the `attribute?` headache in TypeScript. For our application we can specify the `<Recipe>` type on the methods - so the application assumes that the response is of type `recipe`.

If you don't have an interface for a third-party API or just want a workaround, you can simply add the `<unknown>` type, which is, as stated in the Angular Docs, a better approach than the `<any>` type.

All of these patterns make it a lot easier to work with the service on a component level, which we'll take a look at now.

List all recipes component

The recipe-list component will handle the presentation, styling and logic for displaying all recipes. Create one by entering the following command in your terminal:

```
ng generate component recipe-list
```

Which should generate a folder inside the `app` directory with the following structure:

```
\---app
  ...
  \---recipe-list
    recipe-list.component.css
    recipe-list.component.html
    recipe-list.component.spec.ts
    recipe-list.component.ts
```

Four files make up the entire component:

- styling (`recipe-list.component.css`)

- presentation (`recipe-list.component.html`)
- test (`recipe-list.component.spec.ts`)
- logic (`recipe-list.component.ts`)

But why not everything in one file like React does? On one hand it's kind of a personal preference - you can do a single-file approach in Angular too, you just need to change a few properties in the annotation.

But if you have seen React components before, you know they can get pretty long and look like some delicious spaghetti(code) ☐.

Splitting the code into separate files makes it look better and is easier to maintain - and we follow an important software design principle: **Single Concern**! Each file is only responsible for one part.

Now we can add the following html to the `recipe-list.component.html` file:

mean-stack-example-app/client/src/app/recipe-list/recipe-list.component.html

```
<div class="p-8">
<header class="my-6 flex items-start gap-6">
  <div>
    <h1 class="text-3xl font-bold">Tasty Recipe List </h1>
    <h2 class="text-sm text-blue-500">by Andy</h2>
  </div>
  <a routerLink="/recipes/add" class="bg-blue-500 text-neutral-50 px-2 py-1 rounded-
lg hover:shadow-xl transition-all">Add a new Recipe +</a>
</header>
<main>
  <div class="recipe-list flex gap-6 flex-wrap">
    @for (recipe of recipes(); track recipe) {
      <div class="recipe flex w-80 flex-col gap-2 shadow-lg p-3 rounded-lg
hover:shadow-xl hover:cursor-pointer transition-all bg-neutral-50">
        <div class="flex items-center justify-between flex-wrap">
          <h2 class="text-xl font-medium"><a
routerLink="/recipes/view/{{recipe._id}}">{{ recipe.title }}</a></h2>
          <div class="flex gap-2 items-center">
            <a routerLink="/recipes/edit/{{recipe._id}}" class="text-neutral-500"
>Edit</a>
            <button class=" text-red-500 rounded-lg bg-red-300 px-2 py-1"
(click)="deleteRecipe(recipe._id || '')" >Delete</button>
          </div>
          <div class="w-min text-nowrap category rounded-xl bg-blue-200 text-blue-
500 px-2 py-1 text-sm ">{{ recipe.category }}</div>
          <p>{{ recipe.description }}</p>
          <div class="ingredients line-clamp-3 p-1 text-neutral-400">
            @for (ingredient of recipe.ingredients; track ingredient) {
              <div>
                <span>{{ingredient.quantity}} </span>
                <span>{{ingredient.unit}} </span>
              </div>
            }
          </div>
        </div>
      </div>
    }
  </div>
</main>
```

```

        <span>{{ingredient.name}} </span>
      </div>
    }
  </div>
</div>
}
</div>
</main>
</div>

```

A big chunk of the code is for styling purposes (Tailwind 🍷), except the data references and the brand-new syntax for html-logic in Angular 17!

In earlier versions of Angular you had to write syntax inside the html tag itself (for example `*ngFor`), which makes it really confusing to use when writing a bit more complex display logic. But thankfully the Angular development team noticed that issue, and you can now use `@for`, `@if`, etc. statements in your components!

This component uses `@for` to render a recipe card for each recipe inside the recipes list, and a loop to render the ingredients. It also contains the html logic for all our features: viewing, editing, adding and updating recipes, we'll build these in the following steps.

To reference a variable of your component.ts file, you need to add two curly brackets `{{variable}}` surrounding the variable name. Of course the variable needs to be defined, which I did in the `recipe-list.component.ts` file:

mean-stack-example-app/client/src/app/recipe-list/recipe-list.component.ts

```

import {Component, OnInit, signal, WritableSignal} from '@angular/core';
import { CommonModule } from '@angular/common';
import { Recipe } from '../recipe';
import { RecipeService } from '../recipe.service';
import { RouterLink, RouterOutlet } from "@angular/router";

@Component({
  selector: 'app-recipe-list',
  standalone: true,
  imports: [CommonModule, RouterLink, RouterOutlet],
  templateUrl: './recipe-list.component.html',
  styleUrls: ['./recipe-list.component.css']
})
export class RecipeListComponent implements OnInit{
  recipes = signal<Recipe[]>([]);

  constructor(private recipeService:RecipeService){}

  ngOnInit(): void {
    this.loadRecipes();
  }
}

```

```

private loadRecipes(){
  this.recipeService.getAllRecipes().subscribe(res => {
    this.recipes.set(res);
  })
}

public deleteRecipe(id: string): void {
  this.recipeService.deleteRecipe(id).subscribe((res:any)=>{
    if(res.deletedCount==1){
      this.recipes.update(arr=>
        arr.filter(recipe => recipe._id !== id));
    }
  })
}
}

```

In a nutshell, this code fetches the data and loads it into a variable, so the html file can use it. For that, we are subscribing to our `getAllRecipes()` function from the `RecipeService` and set the recipes variable to the result of the method.

Since we always want the freshest data from the database we'll use the `OnInit` interface, which whenever we access the `/recipes` route calls the `ngOnInit()` function (similar to `useEffect()` in React). And that function calls our load function to receive and display the newest data.

And one new Syntax feature: Angular Signals! Because we want the recipe list to be reactive (update the UI when we delete one for example) Signals are our help in need. Just define the variable with `signal<Recipe[]>([])` and you're good to go! To update a signal you can call either call the `set()` or `update()` method.

With all that (fine separated!) code, we get something that looks like this:

Tasty Recipe List

by Andy

Add a new Recipe +

Spaghetti Bolognese

Edit

Delete

main course

Classic Italian pasta dish with a rich meat sauce.

200 grams Spaghetti
300 grams Ground beef
400 ml Tomato sauce...

Pancakes

Edit

Delete

breakfast

Fluffy and delicious pancakes for a perfect breakfast.

1 cup All-purpose flour
1 cup Milk
1 piece Egg...

Brownies

Edit

Delete

dessert

Indulgent chocolate brownies for a sweet treat.

1 cup Butter
2 cups Granulated sugar
4 pieces Eggs...

Detail view of a single recipe component

If you looked closely, you may have noticed the link in the title of the recipe. For this link to work, we need another component which displays a single recipe in detail. Again, use the `ng generate component recipe` command for that.

Before implementing anything, add the new route to the `app.routes.ts` file by adding a new object to the array:

mean-stack-example-app/client/src/app/app.routes.ts

```
...
{ path: 'recipes/view/:id', component: RecipeComponent },
...
```

The `:id` is a placeholder for the dynamic id each recipe has.

Great! Now we can work out the component logic for viewing a single recipe.

mean-stack-example-app/client/src/app/recipe/recipe.component.ts

```
import { Component, Input } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RecipeService } from '../recipe.service';
import { Recipe } from '../recipe';
import { RouterLink } from "@angular/router";

@Component({
  selector: 'app-recipe',
  standalone: true,
  imports: [CommonModule, RouterLink],
  templateUrl: './recipe.component.html',
  styleUrls: ['./recipe.component.css']
})
export class RecipeComponent {

  constructor(private recipeService: RecipeService) {

  }
  recipe: Recipe = {};
  @Input()
  set id(recipeId: string) {
    this.recipeService.getSingleRecipe(recipeId).subscribe(res=>{
      this.recipe = res;
    })
  }
}
```

The `set id()` function handles the data fetching using our service. You need to name the function like the parameter you specified in the routes, so Angular knows what you mean.

And that's all! The html is pretty easy too:

mean-stack-example-app/client/src/app/recipe/recipe.component.html

```

<div class="recipe flex flex-col gap-5 p-3 h-screen bg-neutral-50">
  <a routerLink="/recipes" class="text-sm underline">⌕ Back to home</a>
  <h2 class="text-3xl font-medium">{{ recipe.title }}</h2>
  <div class="w-min text-nowrap category rounded-xl bg-blue-200 text-blue-500 px-2
py-1 text-sm ">{{ recipe.category }}</div>
  <p>{{ recipe.description }}</p>
  <div class="ingredients p-1 text-neutral-400">
    @for (ingredient of recipe.ingredients; track ingredient) {
      <div>
        <span>{{ingredient.quantity}} </span>
        <span>{{ingredient.unit}} </span>
        <span>{{ingredient.name}} </span>
      </div>
    }
  </div>
  <p class="max-w-96">{{recipe.instructions}}</p>
</div>

```

Again, we're using the variables from the component and listing them with a for loop - that's it.

Adding a recipe

Adding a recipe needs a separate component as well, because it will live on a separate page. Create a new component with:

```
ng generate component recipe-add
```

And add the route to `app.routes.ts`:

mean-stack-example-app/client/src/app/app.routes.ts

```
{ path: 'recipes/add', component: RecipeAddComponent},
```

Adding a new recipe has a few caveats:

- Category selection
- Adding multiple ingredients

But let's look at how Angular can help us tackle these problems. I'd like to write the logic first and then display it. So in your `recipe-add.component.ts`, write:

mean-stack-example-app/client/src/app/recipe-add/recipe-add.component.ts

```

import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormBuilder, Validators, FormArray, ReactiveFormsModule } from

```

```

"@angular/forms";
import {Router, RouterLink} from "@angular/router";
import {RecipeService} from "../recipe.service";
import {AddRecipeRequest} from "../addRecipeRequest";

@Component({
  selector: 'app-recipe-add',
  standalone: true,
  imports: [CommonModule, ReactiveFormsModule, RouterLink],
  templateUrl: './recipe-add.component.html',
  styleUrls: ['./recipe-add.component.css']
})
export class RecipeAddComponent {

  constructor(private fb: FormBuilder, private recipeService: RecipeService, private
router:Router) {

  }

  categories: string[] = ['breakfast', 'main course', 'snack', 'dessert'];
  requestFailed: boolean = false;

  addRecipeForm =
    this.fb.group({
      title: ['', Validators.required],
      description: ['', Validators.required],
      category: ['', Validators.required],
      ingredients: this.fb.array([]),
      instructions: ['', [Validators.required, Validators.minLength(10)]]
    });

  get ingredients(): FormArray {
    return this.addRecipeForm.get('ingredients') as FormArray;
  }

  addIngredient(): void {
    this.ingredients.push(this.fb.group({
      name: ['', Validators.required],
      quantity: ['', [Validators.required, Validators.min(0)]],
      unit: ['', [Validators.required, Validators.minLength(1)]]
    }))
  }

  removeIngredient(index: number): void {
    this.ingredients.removeAt(index);
  }

  createRecipe(): void {
    if (this.addRecipeForm.valid) {
      this.requestFailed=false;
      const recipeData: any = this.addRecipeForm.value;
    }
  }
}

```



```

const newRecipe: AddRecipeRequest = {
  title: recipeData.title,
  description: recipeData.description,
  category: recipeData.category,
  ingredients: recipeData.ingredients,
  instructions: recipeData.instructions
};

this.recipeService.createRecipe(newRecipe).subscribe((res:any) => {
  if(res.insertedId!=null){
    this.router.navigate(["/recipes"]);
  } else {
    this.requestFailed=true;
  }
});
}
}
}

```

Well that's quite a bit of code - let's go top down to explain everything.

To bind a form to some variables we use Angulars FormBuilder, which is very handy because it is super easy to add validations and group the variables nicely in one object. The validations come from Angular directly as well, there are a lot of prebuilt ones, but you can also write your own - for this guide, the base ones will be enough. So, everything is in the `addRecipeForm` object - the ingredients are an exception.

That's because ingredients are dynamic, one recipe can have a variable number of them. So we need a `FormArray` and some methods to control the elements:

- `addIngredient` pushes a new ingredient object to the ingredients array
- `get ingredients()` returns the current ingredients of the form
- `removeIngredient(index: number)` removes the ingredient with the given index

You'll see in a second how we use these methods in the html - I must say I am incredibly astonished how well forms in Angular work, especially with complex form logic, and must say the best I have seen in my experience.

The `createRecipe()` method puts everything together and calls the service method. If the request was successful, the user is being redirected to the `/recipes` page - where he can see his delicious new recipe! Else, the `requestFailed` variable is set to `true`, which triggers an error message to be displayed in the html.

That's it for the logic, now the presentation:

mean-stack-example-app/client/src/app/recipe-add/recipe-add.component.html

```

<div class="p-8">
  <header class="my-6 flex flex-col items-start gap-6">

```

```

<a routerLink="/recipes" class="text-sm underline">⏪ Back to home</a>
<h1 class="text-3xl font-bold">Create a new Recipe</h1>
</header>
<main>
  <form [formGroup]="addRecipeForm" class="w-1/3" (submit)="createRecipe()">
    <div class="flex flex-col gap-1 pb-4">
      <label for="title" class="text-neutral-400">Title </label>
      <input id="title" type="text" formControlName="title" class="border-2 rounded-
md">
    </div>

    <div class="flex flex-col gap-1 pb-4">
      <label for="description" class="text-neutral-400">Description </label>
      <textarea id="description" type="text" formControlName="description"
class="border-2 rounded-md"></textarea>
    </div>

    <div class="flex flex-col gap-1 pb-4">
      <label for="category" class="text-neutral-400">Category </label>
      <select id="category" formControlName="category" class="border-2 rounded-md">
        @for(category of categories; track category) {
          <option [value]="category">{{category}}</option>
        }
      </select>
    </div>

    <div class="flex flex-col gap-1 pb-4" formArrayName="ingredients">
      <h3 class="text-base text-neutral-400 mb-3">Ingredients</h3>

      @for(ingredient of ingredients.controls; track ingredient; let index =
$index){
        <div class="flex flex-col gap-3 mb-4" [formGroupName]="index">
          <div class="flex flex-col gap-1">
            <label for="ingredient-name-{{index}}" class="text-neutral-
400">Name</label>
            <input id="ingredient-name-{{index}}" type="text" formControlName="name"
placeholder="Name" class="border-2 rounded-md">
          </div>
          <div class="flex flex-row justify-between items-end">
            <div class="flex flex-col gap-1">
              <label for="ingredient-quantity-{{index}}" class="text-neutral-
400">Quantity</label>
              <input id="ingredient-quantity-{{index}}" type="number"
formControlName="quantity" placeholder="1" class="border-2 rounded-md">
            </div>
            <div class="flex flex-col gap-1">
              <label for="ingredient-unit-{{index}}" class="text-neutral-
400">Unit</label>
              <input id="ingredient-unit-{{index}}" type="text" formControlName="unit"
placeholder="ml" class="border-2 rounded-md">
            </div>
          </div>
        </div>
      }
    </div>
  </form>

```

```

        <button type="button" (click)="removeIngredient(index)" class="text-red-500
rounded-lg bg-red-300 px-2 py-1">Delete</button>
      </div>
    </div>
  }

  <button type="button" (click)="addIngredient()" class="bg-blue-500 text-
neutral-50 px-2 py-1 rounded-lg hover:shadow-xl transition-all">Add ingredient
+</button>
</div>

<div class="flex flex-col gap-1 pb-4">
  <label for="instructions" class="text-neutral-400">Instructions </label>
  <textarea id="instructions" type="text" formControlName="instructions"
class="border-2 rounded-md"></textarea>
</div>

  <button type="submit" [disabled]="!addRecipeForm.valid" class="bg-blue-500 text-
neutral-50 px-2 py-1 rounded-lg hover:shadow-xl transition-all disabled:bg-blue-200
disabled:hover:shadow-none">Create Recipe</button>
  @if(requestFailed) {
    <p class="text-red-500 my-2">Adding recipe failed - Please try again.</p>
  }
</form>
</main>
</div>

```

The biggest part of the page is, of course, the form. To bind the form variable we need the `[formGroup]="addRecipeForm"` attribute. On submit, the function inside the submit attribute (`submit)="createRecipe()"` will be called, for this case, to create a recipe.

For an input to be mapped to its corresponding variable, you need to set the `formControlName` attribute on the `<input>` to the variable name.

That's the same for a select field, too (categories), but we also need to loop over the options - which is an array in the component file.

Now the tricky part - the dynamic ingredients. Since these are an array the parent div needs to be addressed with the `formArrayName` attribute. Then you can loop over all the ingredients currently in the list (which will be zero at the beginning).

For Angular not to throw any errors, you need to set the `[formGroupName]="index"` to the index of the array - if you have an id or something else unique, you can use that as well. But then, same as before, add the `formControlName` to the inputs, and you're good to go!


And don't forget to make the `input` id unique, for example like I did, by adding the index to it `ingredient-name-{{index}}`.

Since the user may want to add more ingredients, we need a button that adds more input fields. We have already written the method which pushes an object to our array, so only the binding is

missing - just do so by setting the `(click)="addIngredient()"` directive.

Lastly, we need a button to submit the form, which has one special ability - being disabled if the form is not valid. Because we already have our validations in place, we just need to call `addRecipeForm.valid` on the form, which returns us a boolean whether all checks have passed or not. Just set the result on the `[disabled]` attribute and the user won't be able to submit a falsy form!

And that's the result ☐:

 [Back to home](#)

Create a new Recipe

Title

Description

Category

Ingredients

Name

Quantity

Unit

Delete

Add ingredient +

Instructions

Create Recipe

Delete a recipe

Something easy in-between: deleting a recipe. HTML wise we've got everything ready - if you forgot

to add the delete button, here again:

mean-stack-example-app/client/src/app/recipe-add/recipe.component.html

```
<button class=" text-red-500 rounded-lg bg-red-300 px-2 py-1"
(click)="deleteRecipe(recipe._id)" >Delete</button>
```

When the button is clicked, the function `deleteRecipe(recipe._id)` with the recipe id is called, so let's look at the implementation for that.

```
public deleteRecipe(id: string): void {
  this.recipeService.deleteRecipe(id).subscribe((res:any)=>{
    if(res.deletedCount==1){
      this.recipes.update(arr=>
        arr.filter(recipe => recipe._id !== id));
    }
  })
}
```

In a nutshell, the `deleteRecipe()` service function is called and when it was successful, the array of recipes will be updated/replaced with a filtered one.

And because we're using Angular Signals, we can call the `update()` function on the recipes which updates the data without needing to refresh the page!

[Delete Recipe GIF] | [../images/mean-delete-recipe.gif](#)

Edit a recipe

If you need to change something in your existing recipes, you want to be able to edit them right? For that, we need a new page/component which you can generate with the `ng generate component recipe-edit` command.

Then, add the route to your `app.routes.ts`:

```
{ path: 'recipes/edit/:id', component:RecipeEditComponent}
```

Editing a recipe is a bit similar to creating a new recipe, because we use the same form. But let's look at the logic first - in your `recipe-edit.component.ts` file, write:

```
import {Component, Input} from '@angular/core';
import { CommonModule } from '@angular/common';
import {RecipeService} from "../recipe.service";
import {FormArray, FormBuilder, FormControl, FormGroup, ReactiveFormsModule,
Validators} from "@angular/forms";
import {Router, RouterLink} from "@angular/router";
import {Recipe} from "../recipe";
```

```

import {AddRecipeRequest} from "../addRecipeRequest";
import {Ingredient} from "../ingredient";

@Component({
  selector: 'app-recipe-edit',
  standalone: true,
  imports: [CommonModule, ReactiveFormsModule, RouterLink],
  templateUrl: './recipe-edit.component.html',
  styleUrls: ['./recipe-edit.component.css']
})
export class RecipeEditComponent {

  constructor(private recipeService:RecipeService, private fb: FormBuilder, private
router:Router) {

  }

  categories: string[] = ['breakfast', 'main course', 'snack', 'dessert'];
  requestFailed: boolean = false;
  recipe:Recipe = {};
  editRecipeForm = this.fb.group({
    title: ['', Validators.required],
    description: ['', Validators.required],
    category: ['', Validators.required],
    ingredients: this.fb.array([]),
    instructions: ['', [Validators.required, Validators.minLength(10)]]
  });

  @Input()
  set id(recipeId: string) {
    this.recipeService.getSingleRecipe(recipeId).subscribe(res=>{
      this.recipe = res;
      this.editRecipeForm.patchValue({
        title: this.recipe.title || '',
        description: this.recipe.description || '',
        category: this.recipe.category || '',
        instructions: this.recipe.instructions || '',
      })
      this.recipe.ingredients?.forEach(ingredient => {
        this.addExistingIngredient(ingredient);
      })
    })
  }

  get ingredients(): FormArray {
    return this.editRecipeForm.get('ingredients') as FormArray;
  }

  addIngredient(): void {
    this.ingredients.push(this.fb.group({
      name: ['', Validators.required],
      quantity: ['', [Validators.required, Validators.min(0)]],
    }));
  }

```

```

        unit: ['', [Validators.required, Validators.minLength(1)]]
      )))
    }

    addExistingIngredient(ingredient:Ingredient): void {
      this.ingredients.push(this.fb.group({
        name: [ingredient.name, Validators.required],
        quantity: [ingredient.quantity, [Validators.required, Validators.min(0)]],
        unit: [ingredient.unit, [Validators.required, Validators.minLength(1)]]
      })))
    }

    removeIngredient(index: number): void {
      this.ingredients.removeAt(index);
    }

    updateRecipe(): void {
      if (this.editRecipeForm.valid) {
        this.requestFailed = false;
        const recipeData: any = this.editRecipeForm.value;

        const updatedRecipe: AddRecipeRequest = {
          title: recipeData.title,
          description: recipeData.description,
          category: recipeData.category,
          ingredients: recipeData.ingredients,
          instructions: recipeData.instructions
        };

        this.recipeService.updateRecipe(this.recipe._id ||
'',updatedRecipe).subscribe((res: any) => {
          if (res.modifiedCount == 1) {
            this.router.navigate(["/recipes"]);
          } else {
            this.requestFailed = true;
          }
        });
      }
    }
  }
}

```

Some of the code is almost the same as in the `recipe-add` component. Because we already have existing data in this form, it needs to be loaded properly, which happens in the `set id(recipeId: string)` function.

It fetches the data for the recipe using the id url parameter and updates the form group to the data of the fetched recipe. For that, we can use the inbuilt function `patchValue()`, which takes in an object of the values the `FormGroup` has. For the ingredients (a `FormArray`), we need to loop over the ingredients on the recipe and add them to the form - that happens in the `addExistingIngredient()` function.

The `updateRecipe()` method calls the recipe service to update the recipe and redirects the user back to the list view upon a successful request.

Now for the presentation part:

```
<div class="p-8">
  <header class="my-6 flex flex-col items-start gap-6">
    <a routerLink="/recipes" class="text-sm underline">⏪ Back to home</a>
    <h1 class="text-3xl font-bold">Edit Recipe
  {{editRecipeForm.get('title')?.value}}</h1>
  </header>
  <main>
    <form [formGroup]="editRecipeForm" (submit)="updateRecipe()">
      <div class="flex flex-col gap-1 pb-4">
        <label for="title" class="text-neutral-400">Title </label>
        <input id="title" type="text" formControlName="title" class="border-2 rounded-
md">
      </div>
      <div class="flex flex-col gap-1 pb-4">
        <label for="description" class="text-neutral-400">Description </label>
        <textarea id="description" type="text" formControlName="description"
class="border-2 rounded-md"></textarea>
      </div>
      <div class="flex flex-col gap-1 pb-4">
        <label for="category" class="text-neutral-400">Category </label>
        <select id="category" formControlName="category" class="border-2 rounded-md">
          @for(category of categories; track category) {
            <option [value]="category">{{category}}</option>
          }
        </select>
      </div>
      <div class="flex flex-col gap-1 pb-4" formArrayName="ingredients">
        <h3 class="text-base text-neutral-400 mb-3">Ingredients</h3>

        @for(ingredient of ingredients.controls; track ingredient; let index =
$index){
          <div class="flex flex-col gap-3 mb-4" [formGroupName]="index">
            <div class="flex flex-col gap-1">
              <label for="ingredient-name-{{index}}" class="text-neutral-
400">Name</label>
              <input id="ingredient-name-{{index}}" type="text" formControlName="name"
placeholder="Name" class="border-2 rounded-md">
            </div>
            <div class="flex flex-row justify-between items-end">
              <div class="flex flex-col gap-1">
                <label for="ingredient-quantity-{{index}}" class="text-neutral-
400">Quantity</label>
                <input id="ingredient-quantity-{{index}}" type="number"
formControlName="quantity" placeholder="1" class="border-2 rounded-md">
              </div>
            </div>
          </div>
        }
```



```

        <div class="flex flex-col gap-1">
          <label for="ingredient-unit-{{index}}" class="text-neutral-400">Unit</label>
          <input id="ingredient-unit-{{index}}" type="text" formControlName="unit"
placeholder="ml" class="border-2 rounded-md">
        </div>
        <button type="button" (click)="removeIngredient(index)" class="text-red-500
rounded-lg bg-red-300 px-2 py-1">Delete</button>
      </div>
    </div>
  }

  <button type="button" (click)="addIngredient()" class="bg-blue-500 text-
neutral-50 px-2 py-1 rounded-lg hover:shadow-xl transition-all">Add ingredient
+</button>
</div>
<div class="flex flex-col gap-1 pb-4">
  <label for="instructions" class="text-neutral-400">Instructions </label>
  <textarea id="instructions" type="text" formControlName="instructions"
class="border-2 rounded-md"></textarea>
</div>
<button type="submit" [disabled]="!editRecipeForm.valid" class="bg-blue-500
text-neutral-50 px-2 py-1 rounded-lg hover:shadow-xl transition-all disabled:bg-blue-
200 disabled:hover:shadow-none">Create Recipe</button>
  @if(requestFailed) {
    <p class="text-red-500 my-2">Editing recipe failed - Please try again.</p>
  }
</form>
</main>

</div>

```

The html is almost the same as in the `recipe-add` component. But I've added one special thing, well not that special, but something to show off reactivity. It's the reference to the form variable of the recipe title, which updates as you change the name in the input field!

And that's it - we created a simple recipe book CRUD application using the MEAN-Stack! If you followed my guide until here, you're all set to build out your own ideas - the possibilities are endless!

How you could extend the project

Because of the limited time-frame I had to create this project there are a lot of things left to add, so feel free to clone the project and implement it!

End-to-End Tests

Cypress, Selenium or a different testing framework - end-to-end testing is super important, especially in big applications. So why not give it a try and increase the test coverage of the recipe

book?

Searching and Filtering

Filter recipes by their categories, ingredients or even search them based on the content - maybe try out Elasticsearch for that? A topic with a lot of potential.

Shopping Lists

Add the ingredients of recipes to a shopping list so you don't need to think of what to buy - would be a great feature, eh? Especially great to further advance your REST and data modeling knowledge.

Of course, AI

What about an AI that could give you a weekly food plan based on your specific needs? Or gives you recommendations, helps you grind towards your weight goals, etc. - you see, there are endless possibilities, build what you think is worth and challenges you the most!

Images

Recipes with an image of what the food will look like are way nicer to look at and help the user. Maybe spin up an S3 bucket and get hands-on cloud experience? Great opportunity to advance on this important topic.

Languages (i18n)

Make the application multi-language and allow users to browse recipes in different languages - use Angular or a third party provider to achieve this!

I think I've given you enough ideas - or maybe you come up with something else, feel free to do so! Now I have nothing more to say than happy coding!

Conclusion

Well actually there is one or more things I have to say - the conclusion.

What do I think of the MEAN stack after building a project?

Angular

I must say that I am really happy with Angular 17 - the new syntax is so much more convenient and not confusing anymore. Working with forms was the biggest pro I noticed - having it bind to variables, organizing them in groups, directly adding validations, pre-written methods to use, that's just great, big props to Angular!

One thing - If you are troubleshooting an issue it's really difficult, because neither ChatGPT, StackOverflow or your IDE IntelliSense can help you. It's mostly grinding through the official

documentation and trying to find the one helping code snippet ☐. But Angular 17 is still in Beta, so I don't want to be picky, it will be better in the future.

Express

Building the app was super easy and there was almost no structure overhead, which can be good, but pretty bad if you don't know what you're doing.

Since it doesn't require you to abstract things into modules or layers, the risk of ending with some illegal looking code is not that small. So please keep that in mind if you're not that advanced in backend-development and educate yourself on software architecture.

And a recommendation: Use Express with TypeScript, because without would be a bit of a chaos when the application grows.

MongoDB

Same as Express, super easy to set up and almost no structure overhead.

But same here, if you don't really know what you are doing it can end in total chaos. MongoDB won't stop you because there are no constraints like you have in a relational database.

The connector interfaces to use in the code are working fine and the docs are great. MongoDB has a really neat blog page that is well maintained and easy to understand.

MEAN stack conclusion

I have explained my views on each layer, so just simple question to ask if I am satisfied with the stack:

Would I use MEAN to build projects in the future? Yes!

But please don't be mad at me, because I may also use React (MERN) ☐.

Sources

- <https://seclgroup.com/10-best-examples-of-websites-and-apps-built-with-angular/>
- <https://www.trio.dev/blog/companies-use-angular>
- <https://blog.hubspot.com/website/angularjs-website-examples>
- <https://www.monocubed.com/blog/websites-built-with-angular/>
- <https://www.mongodb.com/languages/mean-stack-tutorial>

Written w ☐ by Andreas Krenn

Have a look at the complete project at: <https://github.com/AndyTrendygh/mean-stack-example-app>