My Recipe App

General considerations

The link to the repository is: https://github.com/hpocchiola/MyRecipeApp. The info needed to launch the solution is also on the repository README.

Guide to starting the app

1. Clone the repository (I usually use Visual Studio option when cloning, since I'll be using Visual Studio anyway)
2. If you used VS to clone, just double-click on the solution named My Recipe App. Otherwise, use VS to open the solution from where you cloned the repository.
3. Hit the start button. IIS Express default configuration works great.
4. Both Frontend and Backend should be running now on https://localhost:44391/ .
5. https://localhost:44391/recipes has all the required features. You can also view the api through swagger on https://localhost:44391/swagger.
6. Note: the first time you start the solution VS will automatically execute npm install. It will also create and seed a SQLite DB called MyRecipeAppDb.db

Note: the first time you start the solution VS will automatically execute npm install. It will also create and seed a SQLite DB called MyRecipeAppDb.db

I used .Net Core 3.1 to develop the API, React to develop the web app, and SQLite to store the data. I also used some extra libraries like Entity Framework, Automapper, Swagger, Axios, Material-UI.
I first built the solution using Visual Studio boilerplate for .net Core API with React frontend which automatically created all the SPA static files serving that enables the website to run on the same port as the API.

API

I opted for Entity Framework because it makes handling the db extremely easy while remaining quite simple to install. I used swagger in order to test the API endpoints on the flight. (I could have used Postman but in this case I thought swagger was quicker to test

with). Lastly I decided to use AutoMapper to map from DB entities to DTOs because it's really easy to install and configure.
I opted for a custom exception handling middleware to catch exceptions instead of the more traditional try-catch block approach. Even though the former is a little bit more difficult to install than the latter, I find the former to be way more clearer and practical than having to surround every endpoint with a try-catch block.
I didn't need to use a CORS policy because of the way I served the SPA static files on the same port.
I decided to implement a Repository Pattern in order to access the data. The structure is used Controller => Service => Repository => DB. I used
I used dependency injection to make my services and repositories available across the solution. For that I implemented interfaces for every Repository and Service class.
I used class library type projects to separate the different components of the API. I used 5 of these projects: Services, DTOs, Entities, Services and Context.

DB

decided to use SQLite because the complexity of the data that I needed to store was relatively low. For the data model I went with two entities, one for Recipes and another for Ingredients, and went with a many-to-many relationship between them through an entity called RecipeIngredients. I decided to use this approach because it's way more scalable and realistic than using a string ingredient column on a Recipe entity.

Web App

I used the axios library for the communication with the API. I set up an axios interceptor to process every request and response and do something if any error occurs (In this case it just logs on the console).
I used Material-UI in order to build the necessary components for the views and the layout. To be completely honest, styles and CSS is where I put the least time and effort. I sometimes used inline styling which is far from ideal, but I decided to keep it simple and focus more on other aspects. If I wanted to go with a more complex approach I would probably use scss and its powerful features like mixins.
I uploaded the .env file to the repository in order to make it easier for the reviewer to use the app. This should not be done outside this context.

Features

The features are located on the view Recipes in the /recipe route. On loading the route all the recipes are displayed in a card component, which has the recipe title on the top and the list of ingredients below it.

Above the recipes you can find both filters: one text field input that filters the recipe by title according to the field's input. The second one is a multiple Select of ingredients. The select works like this: it will only return recipes that contain every ingredient checked on the select list. Both filters work at the same time, that is, you can filter recipes by title using the text field and at the same time filter the recipes by ingredients using the select input.

The filtering of recipes only occurs when the search icon button is clicked. There is also a clear input button that resets the filters and is only active when a filter is modified (even if the search button is not yet pressed).

The home view is part of the boilerplate generated by Visual Studio but I decided to keep it in order to have some sort of navigation through the site.

I used React's functional components to build my components. I also used React hooks to implement most of the logic of the components. In particular I used useState to maintain the component's state logic, useEffect in order to fetch data from the api, and useCallback so as to avoid duplicating the fetching data function outside the useEffect (I wanted to use the same function on the useEffect and the onClick handle for the clear filters button).

The structure of the recipes view is as follows:

- ➔ Recipe container component
    - ◆ Filters component
    - ◆ Recipe details component
        - ● Ingredients component

One could argue that it's possible to componentize further but I thought this level of granularity was enough considering the relatively low complexity of the project.

On the same note as the componentization, I used the parent-children communication approach to pass data between them because of its simplicity. It's also possible to use something like React useContext hooks or even Redux in order to have data available across the app. In this case I didn't think it was a great idea given the low number of components and the difficulty that entails setting up such mechanisms.