

**Dr.Cook – Interactive Recipe Website**

Lidor Kupershmid – 318187416

Aviv Aytek – 206713604

Lidor Anfinger – 206953895

Dror Hershkovitz – 206891459

**Direct link to the website**[**:**](:%20%20https:/doctor-cook.up.railway.app/recipes)  [**https://doctor-cook.up.railway.app/recipes**](https://doctor-cook.up.railway.app/recipes) **Link to the repository:** [**https://github.com/lidor1997/Doctor-Cook-Web-Project/tree/main**](https://github.com/lidor1997/Doctor-Cook-Web-Project/tree/main)

# Project Description

Doctor Cook Web Project is a web-based recipe management system that allows users to search, add, and manage their custom recipes. The project is built using React, TypeScript, and Vite for the frontend and Node.js, Express, and TypeScript for the backend, with MongoDB as the database. The system supports JWT-based authentication for secure user management, while React Context is used for state management. The frontend provides an interactive and user-friendly interface for browsing and managing recipes, while the backend serves as a REST API, handling authentication, recipe storage, and user data management. The platform enables users to register, log in, search for recipes, mark favorites, and organize their personal recipe collection. The overall goal of Doctor Cook is to provide a seamless experience for food enthusiasts to store and explore their favorite recipes efficiently.

## Features

**User Authentication:** Secure user registration and login functionality, ensuring personalized access to the platform.

**Recipe Management:** Users can add, edit, and delete their own recipes, allowing for personalized recipe collections.

**Recipe Search:** A search feature enabling users to find recipes based on various criteria such as ingredients, cuisine, or dietary preferences.

**Favorites:** Ability for users to mark recipes as favorites for quick access in the future.

**Responsive Design:** The application is designed to be accessible on various devices, providing a seamless experience across desktops, tablets, and smartphones.

**Text-to-Speech (TTS) Support:** Option to listen to recipes for a hands-free experience.

System Components & Architecture

**Client-Side (Frontend) - React + TypeScript**

* **Framework:** React (with TypeScript)
* **State Management:** React Context API
* **API Calls:** Fetch/Axios for interacting with the backend
* **Routing:** React Router
* **UI Frameworks:** Tailwind CSS (or another styling system)
* **Security:** JWT token stored in localStorage or cookies
* **TTS Integration:** Text-to-Speech support for hands-free recipe navigation

**Key Responsibilities:**  
 Rendering the user interface (UI)  
 Handling user interactions  
 Sending requests to the backend  
Displaying recipe data dynamically

**Server-Side (Backend) - Node.js + Express + TypeScript**

* **Framework:** Express.js (with TypeScript)
* **Authentication:** JWT-based authentication
* **Middleware:** Express middleware for request validation & logging
* **Error Handling:** Centralized error handling with catchAsync.ts
* **Routing:** RESTful API endpoints for authentication, recipes, and user   
  management
* **Security:** CORS protection, input sanitization

**Key Responsibilities:** Storing user and recipe data  
 Ensuring fast retrieval and search capabilities  
 Managing user-specific data (e.g., favorite recipes)

**Database - MongoDB (NoSQL)**

* **Database Type:** NoSQL (Document-Oriented)
* **ODM Library:** Mongoose (for structured schema and easy data handling)
* **Collections:**
  + users → Stores user details & authentication data
  + recipes → Stores recipe details, ingredients, steps, and user favorites

**Key Components**

Client-Side Key Components

The **client-side** is responsible for rendering the **UI**, managing user interactions, and communicating with the backend via API requests.

**File:** src/features/auth/api/login/login.api.ts



This file is responsible for handling user authentication by sending a POST request to the /auth/login endpoint using axios. It imports a pre-configured axios instance from the services/axios.ts file and defines the structure of the request and response using TypeScript types. The LoginRequest type specifies that the request must contain a username and password, while LoginResponse is expected to return a User object. The login function takes a LoginRequest object, sends it to the server via axios.post, and returns the response data. If authentication is successful, the function returns the user's data. Using TypeScript generics <LoginResponse> ensures that the API response matches the expected structure.

**File:** src/features/auth/api/loginByToken/loginByToken.api.ts

This file is responsible for handling user authentication via token validation. Instead of requiring the user to enter their username and password, this function sends a GET request to the /auth endpoint using axios to verify if the user is still authenticated. It imports a pre-configured axios instance from the services/axios.ts file and defines the expected response structure using TypeScript types. The LoginByTokenResponse type ensures that the function expects a User object as the response. The loginByToken function does not take any parameters and simply makes a request to the server to check if a valid authentication token is present. If the request is successful, the function returns the user’s data, allowing the application to maintain the user’s session without requiring them to log in again.

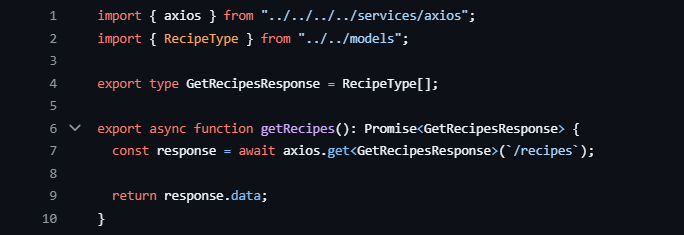
A screenshot of a computer

Description automatically generated

**File:** src/features/auth/api/register/register.api.ts

This file handles user registration by sending a **POST request** to the /auth/register endpoint using axios. It imports a pre-configured axios instance from the services/axios.ts file and defines the expected request and response structures using TypeScript interfaces and types. The RegisterRequest and RegisterRequestApi interfaces specify that the registration request must include a name, username, and password. The RegisterResponse type ensures that the function expects a User object as the response. The register function takes a RegisterRequest object as input, sends the request to the backend using axios.post, and returns the response data. If registration is successful, the function returns the newly created user's details, which can then be used for authentication or session management.

**File:** recipes-client/src/features/recipes/api/getRecipes/getRecipes.api.ts



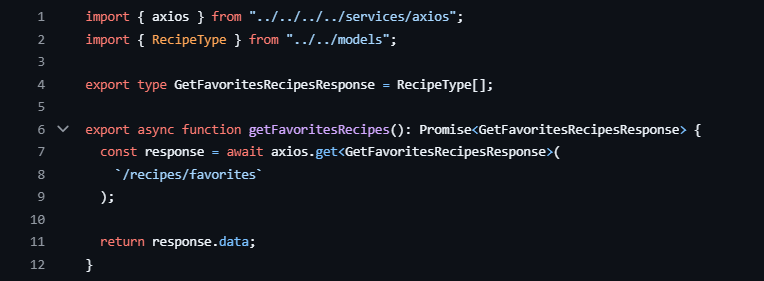
This file is responsible for retrieving a list of recipes by sending a **GET request** to the /recipes endpoint using axios. It imports a pre-configured axios instance from the services/axios.ts file and defines the expected response structure using TypeScript types. The GetRecipesResponse type specifies that the function expects an array of RecipeType objects as the response. The getRecipes function does not take any parameters; it simply makes a request to fetch all available recipes from the backend. If the request is successful, the function returns the list of recipes, which can then be displayed in the user interface.

**File:** recipes-client/src/features/recipes/api/getRecipe/getRecipe.api.ts



This file is responsible for fetching a specific recipe by its ID by sending a **GET request** to the /recipes/{id} endpoint using axios. It imports a pre-configured axios instance from the services/axios.ts file and defines the expected response structure using TypeScript types. The GetRecipeResponse type ensures that the function expects a single RecipeType object as the response. The getRecipe function takes an object containing an id parameter as input, which represents the unique identifier of the recipe to be retrieved. It then makes a request to the backend using axios.get, appending the recipe ID to the request URL. If the request is successful, the function returns the recipe's details, which can then be displayed in the user interface.

**File:** recipes-client/src/features/recipes/api/getFavoritesRecipes/getFavoritesRecipes.api.ts



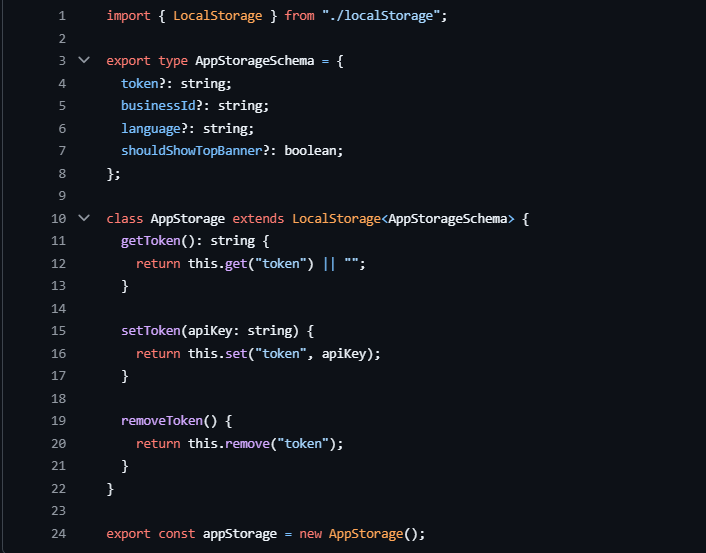
This file is responsible for retrieving the list of favorite recipes by sending a **GET request** to the /recipes/favorites endpoint using axios. It imports a pre-configured axios instance from the services/axios.ts file and defines the expected response structure using TypeScript types. The GetFavoritesRecipesResponse type ensures that the function expects an array of RecipeType objects as the response. The getFavoritesRecipes function does not take any parameters; it simply makes a request to fetch the user's favorite recipes from the backend. If the request is successful, the function returns the list of favorite recipes, which can then be displayed in the user interface for easy access.

**File:** recipes-client/src/features/recipes/api/updateFavorites/updateFavorites.api.ts



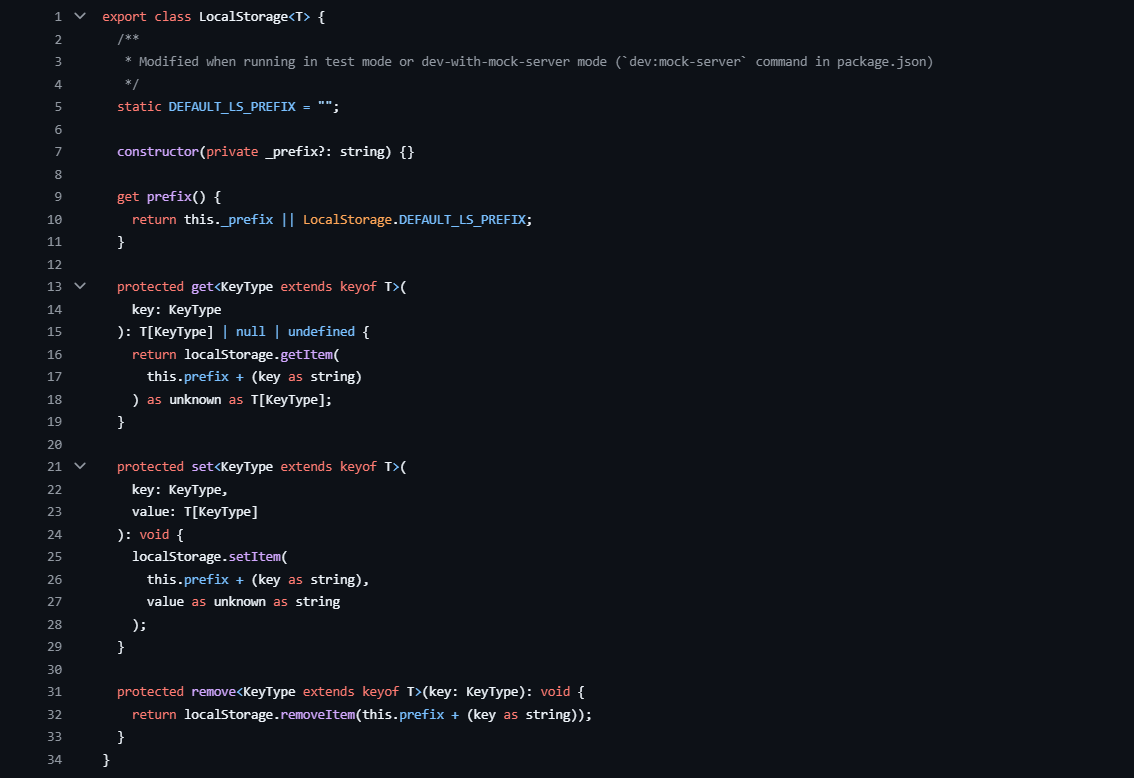
This file is responsible for updating the user's favorite recipes by sending a **PATCH request** to the /recipes/favorites/{recipeId} endpoint using axios. It imports a pre-configured axios instance from the services/axios.ts file and defines the expected request and response structures using TypeScript types. The UpdateFavoritesRequest type specifies that the request must include a recipeId, which represents the unique identifier of the recipe being added or removed from the user's favorites. The UpdateFavoritesResponse type ensures that the function expects a boolean value (true) as the response, indicating a successful update. The updateFavorites function takes an object containing a recipeId parameter as input, makes a request to the backend using axios.patch, and returns the response data. If the request is successful, the function confirms that the favorite list has been updated accordingly.

**File:** recipes-client/src/services/appStorage/appStorage.ts



This file is responsible for managing application-level storage using **local storage**. It defines a class called AppStorage, which extends LocalStorage and provides methods for handling authentication tokens and other stored data. The AppStorageSchema type defines the expected structure of stored data, including token, businessId, language, and shouldShowTopBanner. The class includes methods to **retrieve, set, and remove** the authentication token from local storage. The getToken method retrieves the token, returning an empty string if it is not found. The setToken method stores a given API key in local storage, and the removeToken method deletes the stored token. At the end of the file, an instance of AppStorage is created and exported as appStorage, making it accessible throughout the application for handling authentication-related storage.

**File:** recipes-client/src/services/appStorage/localStorage.ts



This file defines a generic LocalStorage class, which provides a structured way to interact with the browser's local storage. It allows saving, retrieving, and removing data while maintaining a consistent storage prefix. The class supports TypeScript generics, ensuring type safety when storing and retrieving data.

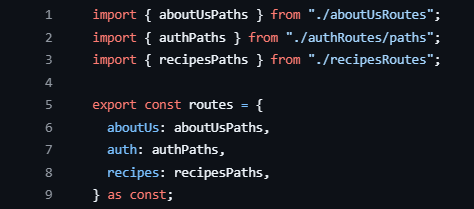
The DEFAULT\_LS\_PREFIX is a static property that modifies storage keys when running in test mode or mock server mode. The class constructor accepts an optional \_prefix parameter, which is used to prepend storage keys to avoid conflicts. The prefix getter ensures that if no custom prefix is provided, the default prefix is used.

The class includes three main methods:

* get(key): Retrieves an item from local storage based on the provided key and returns its stored value, or null if the key does not exist.
* set(key, value): Stores a value in local storage with a key that includes the defined prefix.
* remove(key): Deletes a stored value from local storage based on the key.

By using this generic class, other parts of the application can extend and implement custom storage logic without directly interacting with localStorage, ensuring maintainability and scalability.

**File:** recipes-client/src/routes/routes.ts



This file defines and exports a routes object that consolidates different route path configurations for the application. It imports route path definitions from separate modules (aboutUsRoutes, authRoutes/paths, and recipesRoutes) and organizes them into a structured object.

Each imported module contains predefined route paths related to a specific feature:

* aboutUsPaths holds routes related to the "About Us" section of the application.
* authPaths includes authentication-related routes, such as login and registration.
* recipesPaths manages routes related to the recipe system, such as viewing, creating, or editing recipes.

The routes object is marked as const, ensuring that its structure remains immutable. This approach improves maintainability and allows the application to reference route paths in a centralized manner, reducing the risk of hardcoded URLs scattered across the codebase.

**File:** recipes-client/src/features/recipes/views/RecipeView/RecipeView.tsx

**import** **React**, { useEffect, useState } **from** "react";

**import** { **RecipeType** } **from** "../../models";

**import** { getRecipe } **from** "../../api";

**import** { useLocation } **from** "react-router-dom";

**import** { routes } **from** "../../../../routes";

**export** **const** **RecipeView** = () => {

**const** [recipe, setRecipe] = useState<**RecipeType** | null>(null);

**const** [rating, setRating] = useState<number | null>(null);

**const** [speakingIndex, setSpeakingIndex] = useState<number | null>(null);

**const** { pathname } = **useLocation**();

**useEffect**(() => {

**const** recipeId = pathname.**split**(routes.recipes.recipes)[1];

**const** **getRecipeData** = **async** () => {

**try** {

**const** recipeResponse = **await** **getRecipe**({ id: recipeId });

**setRecipe**(recipeResponse);

} **catch** (error) {

console.**error**(error);

}

};

**getRecipeData**();

}, [pathname]);

**const** **handleRating** = (rating: number) => {

**setRating**(rating);

};

**const** **handleSpeak** = (text: string, index: number) => {

window.speechSynthesis.**cancel**();

**if** (speakingIndex === index) {

**setSpeakingIndex**(null);

**return**;

}

**const** utterance = **new** **SpeechSynthesisUtterance**(text);

// Make the voice more friendly

utterance.pitch = 1.1; // Slightly higher pitch (default is 1)

utterance.rate = 0.9; // Slightly slower rate (default is 1)

utterance.volume = 1; // Maximum volume

// Try to use a friendly voice if available

**const** voices = window.speechSynthesis.**getVoices**();

**const** preferredVoice = voices.**find**(

voice =>

voice.name.**includes**('Samantha') || // iOS/macOS friendly voice

voice.name.**includes**('Google UK Female') || // Chrome friendly voice

voice.name.**includes**('Microsoft Zira') // Windows friendly voice

);

**if** (preferredVoice) {

utterance.voice = preferredVoice;

}

utterance.onend = () => **setSpeakingIndex**(null);

**setSpeakingIndex**(index);

window.speechSynthesis.**speak**(utterance);

};

**if** (!recipe) **return** null;

**return** (

<**div**>

<**div** className="container mx-auto px-4 py-4 sm:py-8 max-w-4xl">

{/\* <!-- Recipe Header --> \*/}

<**div** className="mb-6 text-center">

<**h1** className="dark:text-white text-2xl sm:text-3xl font-bold mb-3 sm:mb-4 text-amber-800">

{recipe.name}

</**h1**>

<**p** className="dark:text-white text-base sm:text-lg mb-4 px-2">

{recipe.description}

</**p**>

{/\* <!-- Time and Servings Info --> \*/}

<**div** className="flex flex-wrap justify-center gap-3 sm:gap-4 text-sm mb-4 sm:mb-6">

<**div** className="flex items-center gap-2">

<**img**

className="dark:hidden h-5 sm:h-6 w-9 sm:w-11 pl-3 sm:pl-5"

src="/images/DarkClock.png"

alt="DarkClock"

/>

<**img**

className="hidden dark:block h-5 sm:h-6 w-9 sm:w-11 pl-3 sm:pl-5"

src="/images/LightClock.png"

alt="LightClock"

/>

<**span** className="dark:text-white text-sm">

Cook time: {recipe.cookTime}

</**span**>

</**div**>

<**div** className="flex items-center gap-2">

<**img**

className="dark:hidden h-5 sm:h-6 w-9 sm:w-11 pl-3 sm:pl-5"

src="/images/WhiteServing.png"

alt="White Serving"

/>

<**img**

className="hidden dark:block h-5 sm:h-6 w-9 sm:w-11 pl-3 sm:pl-5"

src="/images/BlackServing.png"

alt="Black Serving"

/>

<**span** className="dark:text-white text-sm">

Servings: {recipe.servings}

</**span**>

</**div**>

</**div**>

<**div** className="flex justify-center items-center gap-2 mb-4">

<**span** className="dark:text-white text-sm">

By{" "}

<**span** className="dark:text-slate-400 text-amber-800">

{recipe.by}

</**span**>

</**span**>

<**span** className="dark:text-slate-400 ml-2 text-amber-800">

★ {recipe.rating}

</**span**>

</**div**>

</**div**>

{/\* <!-- Recipe Image and nutrients --> \*/}

<**div** className="mb-6 sm:mb-8 flex flex-col sm:flex-row sm:items-start sm:space-x-6 space-y-4 sm:space-y-0">

<**div** className="w-full sm:w-2/3">

<**img**

src={recipe.image}

alt={recipe.name}

className="w-full rounded-lg shadow-lg"

/>

</**div**>

<**div** className="dark:bg-[#2A3236] bg-[#FFF6E0] border border-amber-800 dark:border-white p-4 sm:p-6 rounded-lg w-full sm:w-1/3">

<**h2** className="dark:text-slate-400 text-lg sm:text-xl font-bold mb-3 sm:mb-4 text-amber-800">

Nutrition

</**h2**>

<**ul** className="space-y-1 sm:space-y-2 text-base sm:text-lg">

<**li** className="dark:text-white">

<**strong** className="dark:text-white">Calories: </**strong**>

{recipe.nutrients.calories}

</**li**>

<**li** className="dark:text-white">

<**strong** className="dark:text-white">Protein: </**strong**>

{recipe.nutrients.protein}g

</**li**>

<**li** className="dark:text-white">

<**strong** className="dark:text-white">Fat: </**strong**>

{recipe.nutrients.fat}g

</**li**>

<**li** className="dark:text-white">

<**strong** className="dark:text-white">Carbs: </**strong**>

{recipe.nutrients.carbs}g

</**li**>

</**ul**>

</**div**>

</**div**>

{/\* <!-- Ingredients --> \*/}

<**div** className="dark:bg-[#2A3236] bg-[#FFF6E0] border border-amber-800 dark:border-white p-4 sm:p-6 rounded-lg mb-6 sm:mb-8">

<**h2** className="dark:text-slate-400 text-lg sm:text-xl font-bold mb-3 sm:mb-4 text-amber-800">

Ingredients

</**h2**>

<**ul** className="dark:text-white space-y-2">

{recipe.ingredients.map((ing) => (

<**li**

key={ing.name + ing.value}

className="flex text-base sm:text-lg justify-between"

>

<**span**>{ing.name}</**span**>

<**span**>{ing.value}</**span**>

</**li**>

))}

</**ul**>

</**div**>

{/\* <!-- Instructions --> \*/}

<**div** className="dark:bg-[#2A3236] bg-[#FFF6E0] border border-amber-800 dark:border-white p-4 sm:p-6 rounded-lg mb-6 sm:mb-8">

<**h2** className="dark:text-slate-400 text-lg sm:text-xl font-bold mb-3 sm:mb-4 text-amber-800">

Instructions

</**h2**>

<**ol** className="dark:text-white space-y-3 sm:space-y-4">

{recipe.instructions.map((inst, index) => (

<**li**

key={inst + index}

className="flex gap-3 sm:gap-4 text-base sm:text-lg items-start"

>

<**span** className="dark:text-slate-400 font-bold text-amber-800">

{index + 1}.

</**span**>

<**span**>{inst}</**span**>

<**button**

onClick={() => handleSpeak(inst, index)}

className="ml-2 p-1 hover:opacity-70 transition-opacity"

aria-label={speakingIndex === index ? "Stop speaking" : "Read instruction aloud"}

>

{speakingIndex === index ? (

<**span** className="text-amber-800 dark:text-white">■</**span**>

) : (

<**span** className="text-amber-800 dark:text-white">🔊</**span**>

)}

</**button**>

</**li**>

))}

</**ol**>

</**div**>

{/\* <!-- User Rating --> \*/}

<**div** className="p-4 sm:p-6 text-center">

<**h2** className="dark:text-slate-400 text-lg sm:text-xl font-bold mb-3 sm:mb-4 text-amber-800">

Rate this Recipe

</**h2**>

<**div** className="flex justify-center items-center gap-2">

{[1, 2, 3, 4, 5].map((star) => (

<**span**

key={star}

className="dark:text-white text-2xl sm:text-3xl cursor-pointer"

onClick={() => handleRating(star)}

>

★

</**span**>

))}

</**div**>

</**div**>

{/\* <!-- save space to display the user the rating he voted for the recipe --> \*/}

{rating && (

<**div**

id="rating-feedback"

className="dark:text-white text-amber-800 text-lg sm:text-xl font-bold text-center mb-4 sm:mb-5"

>

Thank you for rating {rating} stars!

</**div**>

)}

</**div**>

</**div**>

);

};

This file defines the RecipeView component, which is responsible for displaying the details of a specific recipe, including its name, description, cooking time, servings, nutritional information, ingredients, and step-by-step instructions. It also includes features such as Text-to-Speech (TTS) for reading instructions aloud and a rating system that allows users to rate the recipe.

The component fetches the recipe data dynamically based on the recipeId extracted from the current route. When the component mounts, it calls the getRecipeData function, which sends a request to the backend using getRecipe({ id: recipeId }). If the request is successful, the recipe details are stored in the component’s state using useState.

The Text-to-Speech (TTS) functionality allows users to listen to step-by-step instructions by clicking on a button next to each instruction. The handleSpeak function utilizes the Web Speech API, setting pitch, rate, and volume for a more natural voice output. The function also selects a preferred voice based on the available voices on the user's device.

The rating system enables users to submit a rating for the recipe by clicking on a star. The selected rating is stored in the component's state using setRating(rating), and a confirmation message is displayed.

The UI is structured using Tailwind CSS classes, ensuring responsiveness across different screen sizes. The component dynamically displays images, ingredient lists, and step-by-step instructions. If no recipe data is found, the component returns null, preventing the UI from rendering empty content.

Overall, this component integrates multiple features, including recipe retrieval, user interaction, accessibility via TTS, and a rating system, providing a user-friendly experience for exploring and interacting with recipes.

Server-Side Key Components

**File:** recipes-api/src/features/auth/controller/authController.ts

**import** bcrypt **from** 'bcryptjs';

**import** { **Response** } **from** 'express';

**import** { catchAsync, generateToken } **from** '../../../utils';

**import** { **User** } **from** '../../user/models';

**import** { **LoginByTokenRequestType**, **LoginRequestType**, **RegisterUserRequestType** } **from** './types';

**export** **const** register = **catchAsync**(**async** (req: **RegisterUserRequestType**, res: **Response**) => {

**const** { username, password, name } = req.body;

**if** (!username || !password || !name) {

**throw** 'Please enter all fields';

}

**const** existingUser = **await** **User**.**findOne**({ username });

**if** (existingUser) {

**throw** 'User already exists';

}

**const** salt = **await** bcrypt.**genSalt**(10);

**const** hashedPassword = **await** bcrypt.**hash**(password, salt);

**const** newUser = **new** **User**({

username,

password: hashedPassword,

name,

});

**const** userRes = **await** newUser.**save**();

**const** token = **generateToken**({ id: userRes.\_id });

**return** res

.**status**(201)

.**json**({ id: userRes.\_id, name: userRes.name, username: userRes.username, favorites: userRes.favorites, token });

});

**export** **const** login = **catchAsync**(**async** (req: **LoginRequestType**, res: **Response**) => {

**const** { username, password } = req.body;

**const** user = **await** **User**.**findOne**({ username });

**if** (!user) {

**throw** 'Invalid credentials';

}

**const** isMatch = **await** bcrypt.**compare**(password, user.password);

**if** (!isMatch) {

**throw** 'Invalid credentials';

}

**const** token = **generateToken**({ id: user.\_id });

**return** res.**status**(200).**json**({

id: user.\_id,

username: user.username,

name: user.name,

favorites: user.favorites,

token,

});

});

**export** **const** loginByToken = **catchAsync**(**async** (req: **LoginByTokenRequestType**, res: **Response**) => {

**const** { user, token } = req;

**if** (!user) {

**throw** 'token is expired';

}

**return** res

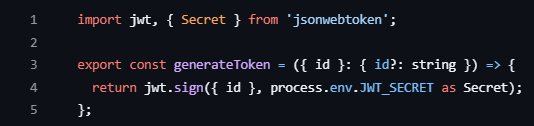
.**status**(200)

.**json**({ id: user.\_id, username: user.username, name: user.name, favorites: user.favorites, token });

});

This file handles user authentication, including registration, login, and token-based authentication. The register function validates input, checks for existing users, hashes passwords with bcrypt, saves new users to the database, generates a JWT token, and returns the user data. The login function verifies credentials by checking the username and comparing the hashed password, generating a JWT token upon success. The loginByToken function authenticates users via an existing token, returning user details if the token is valid; otherwise, it throws an error. All functions use the catchAsync wrapper for error handling.

**File:** recipes-api/src/utils/generateToken.ts



This function generates a **JWT token** for user authentication using jsonwebtoken. It takes an object containing a user ID (id) and signs it with a **secret key** stored in environment variables (process.env.JWT\_SECRET). The generated token can be used to authenticate requests, ensuring secure access to protected routes.

**File:** recipes-api/src/utils/verifyToken.ts

A computer screen with white text

Description automatically generated

This function verifies the validity of a **JWT token** using jsonwebtoken. It takes a token as input and checks it against the **secret key** stored in process.env.JWT\_SECRET. If the token is valid, it returns the decoded payload containing the user ID; otherwise, an error is thrown, indicating that the token is invalid or expired.

**File:** recipes-api/src/middleware/authMiddleware.ts



This middleware function **authenticates requests** by verifying the JWT token. It extracts the Authorization header, retrieves the token, and validates it using verifyToken. If the token is invalid or missing, an error is thrown. Once verified, the middleware fetches the user from the database using the token's decoded user ID. If the user is found, it attaches the user data and token to the request object (req.user and req.token) and proceeds to the next middleware or route handler. This ensures that protected routes can only be accessed by authenticated users.

**File:** recipes-api/src/features/user/models/UserModel.ts

**import** { model, **Schema** } **from** 'mongoose';

**import** { **UserType** } **from** './UserType';

**const** **UserSchema** = **new** **Schema**<**UserType**>(

{

name: {

type: **String**,

required: [true, 'Name is required'],

trim: true,

minlength: [2, 'Name must be at least 2 characters long'],

maxlength: [50, 'Name cannot exceed 50 characters'],

},

username: {

type: **String**,

required: [true, 'Username is required'],

trim: true,

minlength: [2, 'Username must be at least 2 characters long'],

maxlength: [50, 'Username cannot exceed 50 characters'],

},

password: {

type: **String**,

required: [true, 'User password is required'],

},

favorites: {

type: [**String**],

default: [],

},

},

{ timestamps: true },

);

**const** modelName = 'User';

**export** **const** **User** = model<**UserType**>(modelName, **UserSchema**);

This file defines the **User model** using Mongoose, specifying the structure and validation rules for user documents in the database. The schema includes fields for name, username, password, and favorites, with validation rules such as required fields, trimming, and character length constraints. The favorites field stores an array of recipe IDs that the user has marked as favorites. The { timestamps: true } option automatically adds createdAt and updatedAt fields to each document. The model is then exported as User, allowing it to be used for database operations like user creation, authentication, and profile management.

**File:** recipes-api/src/features/recipe/models/RecipeModel.ts

**import** { model, **Schema** } **from** 'mongoose';

**import** { **RecipeType** } **from** './RecipeType';

**const** **RecipeSchema** = **new** **Schema**<**RecipeType**>(

{

name: {

type: **String**,

required: [true, 'Name is required'],

trim: true,

minlength: [2, 'Name must be at least 2 characters long'],

maxlength: [50, 'Name cannot exceed 50 characters'],

},

image: {

type: **String**,

required: [true, 'Image is required'],

trim: true,

},

category: {

type: **String**,

required: [true, 'Category is required'],

trim: true,

minlength: [2, 'Category must be at least 2 characters long'],

maxlength: [50, 'Category cannot exceed 50 characters'],

},

description: {

type: **String**,

required: [true, 'Description is required'],

trim: true,

minlength: [1, 'Description must be at least 10 characters long'],

maxlength: [200, 'Description cannot exceed 200 characters'],

},

cookTime: {

type: **Number**,

required: [true, 'Cook time is required'],

min: [1, 'Cook time must be at least 1 minute'],

max: [300, 'Cook time cannot exceed 300 minutes'],

},

servings: {

type: **Number**,

required: [true, 'Servings is required'],

min: [1, 'Servings must be at least 1'],

max: [20, 'Servings cannot exceed 20'],

},

by: {

type: **String**,

required: [true, 'Author is required'],

trim: true,

minlength: [2, 'Author must be at least 2 characters long'],

maxlength: [50, 'Author cannot exceed 50 characters'],

},

rating: {

type: **Number**,

default: 0,

min: [0, 'Rating must be at least 0'],

max: [5, 'Rating cannot exceed 5'],

},

nutrients: {

calories: {

type: **Number**,

required: [true, 'Calories is required'],

min: [0, 'Calories cannot be negative'],

max: [5000, 'Calories cannot exceed 5000'],

},

protein: {

type: **Number**,

required: [true, 'Protein is required'],

min: [0, 'Protein cannot be negative'],

max: [500, 'Protein cannot exceed 500'],

},

fat: {

type: **Number**,

required: [true, 'Fat is required'],

min: [0, 'Fat cannot be negative'],

max: [500, 'Fat cannot exceed 500'],

},

carbs: {

type: **Number**,

required: [true, 'Carbs is required'],

min: [0, 'Carbs cannot be negative'],

max: [500, 'Carbs cannot exceed 500'],

},

},

ingredients: [

{

name: {

type: **String**,

required: [true, 'Ingredient name is required'],

trim: true,

minlength: [2, 'Ingredient name must be at least 2 characters long'],

maxlength: [50, 'Ingredient name cannot exceed 50 characters'],

},

value: {

type: **String**,

required: [true, 'Ingredient value is required'],

trim: true,

minlength: [1, 'Ingredient value must be at least 1 character long'],

maxlength: [50, 'Ingredient value cannot exceed 50 characters'],

},

},

],

instructions: [

{

type: **String**,

required: [true, 'Instruction is required'],

trim: true,

minlength: [1, 'Instruction must be at least 1 character long'],

maxlength: [200, 'Instruction cannot exceed 200 characters'],

},

],

},

{ timestamps: true },

);

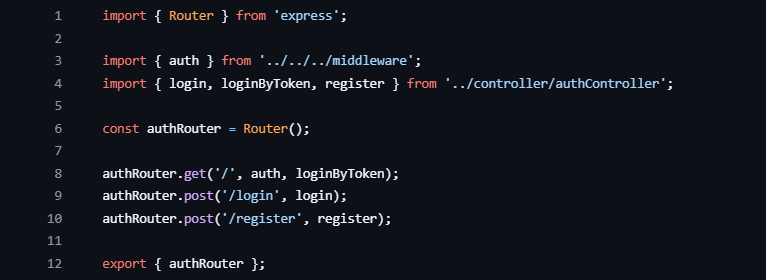
**const** modelName = 'Recipe';

**export** **const** **Recipe** = model<**RecipeType**>(modelName, **RecipeSchema**);

This file defines the Recipe model using Mongoose, specifying the structure and validation rules for recipe documents in the database. The schema includes fields for name, image, category, description, cookTime, servings, by (author), rating, nutrients, ingredients, and instructions, each with specific validation rules for required fields, character limits, and numeric constraints.

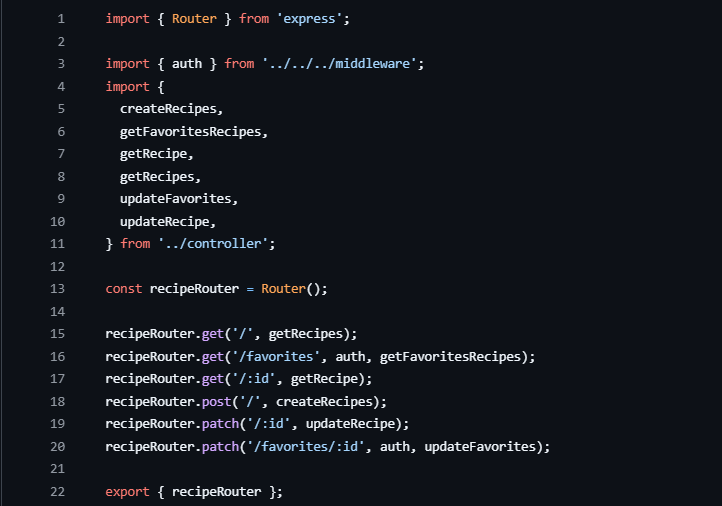
The nutrients object stores nutritional values such as calories, protein, fat, and carbs, ensuring that all values are positive and within reasonable limits. The ingredients array contains objects with name and value fields, while the instructions array stores step-by-step cooking instructions.

The { timestamps: true } option automatically adds createdAt and updatedAt fields to each recipe document. The model is then exported as Recipe, allowing it to be used for database operations such as creating, retrieving, updating, and deleting recipes.

**File:** recipes-api/src/features/auth/router/authRouter.ts

This file defines authentication routes using Express Router. It includes GET / for token-based authentication, POST /login for user login, and POST /register for new user registration. The auth middleware secures token validation, ensuring only authenticated users access protected routes.

**File:** recipes-api/src/features/recipe/router/recipeRouter.ts



This file defines the recipe-related routes using Express Router. It includes endpoints for fetching all recipes (GET /), retrieving a specific recipe (GET /:id), creating a new recipe (POST /), updating a recipe (PATCH /:id), and managing favorites (GET /favorites, PATCH /favorites/:id). The auth middleware protects routes related to user favorites, ensuring only authenticated users can access them.

**File:** recipes-api/src/features/recipe/controller/recipeController.ts

**import** { **Response** } **from** 'express';

**import** { **User** } **from** '../../../features/user/models';

**import** { catchAsync } **from** '../../../utils';

**import** { **Recipe** } **from** '../models';

**import** {

**CreateRecipesRequestType**,

**GetRecipesRequestType**,

**UpdateFavoritesRequestType**,

**UpdateRecipeRequestType**,

} **from** './types';

**export** **const** createRecipes = **catchAsync**(**async** (req: **CreateRecipesRequestType**, res: **Response**) => {

**const** recipes = req.body;

**const** newRecipes = **await** **Recipe**.**insertMany**(recipes);

**return** res.**status**(201).**json**(newRecipes);

});

**export** **const** getRecipes = **catchAsync**(**async** (\_req, res: **Response**) => {

**const** recipes = **await** **Recipe**.**find**();

**return** res.**status**(200).**json**(recipes);

});

**export** **const** getFavoritesRecipes = **catchAsync**(**async** (req: **GetRecipesRequestType**, res: **Response**) => {

**const** user = req.user;

**const** currentUser = **await** **User**.**findById**(user.\_id);

**const** recipes = **await** **Recipe**.**find**({ \_id: { $in: currentUser.favorites } });

**return** res.**status**(200).**json**(recipes);

});

**export** **const** getRecipe = **catchAsync**(**async** (req: any, res: **Response**) => {

**const** { id } = req.params;

**const** recipe = **await** **Recipe**.**findById**(id);

**if** (!recipe) {

**return** res.**status**(404).**json**({ message: 'Recipe not found' });

}

**return** res.**status**(200).**json**(recipe);

});

**export** **const** updateRecipe = **catchAsync**(**async** (req: **UpdateRecipeRequestType**, res: **Response**) => {

**const** { id } = req.params;

**const** updates = req.body;

**const** recipe = **await** **Recipe**.**findByIdAndUpdate**(id, updates, { new: true, runValidators: true });

**if** (!recipe) {

**return** res.**status**(404).**json**({ message: 'Recipe not found' });

}

**return** res.**status**(200).**json**(recipe);

});

**export** **const** updateFavorites = **catchAsync**(**async** (req: **UpdateFavoritesRequestType**, res: **Response**) => {

**const** { id } = req.params;

**const** user = req.user;

console.**log**(user);

**const** isFavorite = user.favorites.**includes**(id);

**const** updatedUser = **await** **User**.**findByIdAndUpdate**(

user.\_id,

{

[isFavorite ? '$pull' : '$addToSet']: { favorites: id },

},

{ new: true },

);

res.**status**(200).**json**({

message: isFavorite ? 'Removed from favorites' : 'Added to favorites',

favorites: updatedUser.favorites,

});

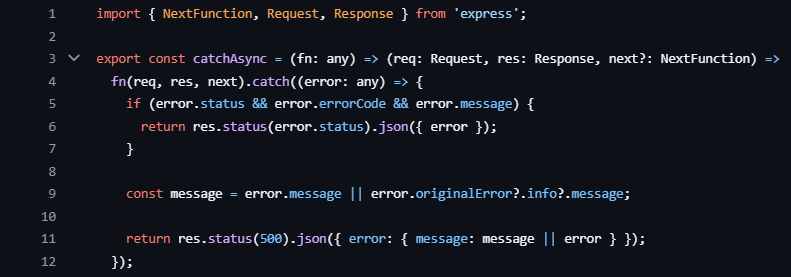
});

This file defines the recipe controller functions for handling CRUD operations and user favorites. It includes:

* createRecipes: Inserts multiple new recipes into the database.
* getRecipes: Retrieves all available recipes.
* getFavoritesRecipes: Fetches the current user's favorite recipes using their stored favorites list.
* getRecipe: Retrieves a single recipe by its ID, returning a 404 error if not found.
* updateRecipe: Updates an existing recipe with new data, validating inputs and returning the updated recipe.
* updateFavorites: Toggles a recipe in the user's favorites list, adding it if not present and removing it if already a favorite.

All functions use catchAsync for error handling and rely on Mongoose models (Recipe and User) for database operations.

**File:** recipes-api/src/utils/catchAsync.ts



This file defines catchAsync, a wrapper function for handling asynchronous errors in Express route handlers. It takes an asynchronous function (fn), executes it, and catches any errors that occur. If the error has a status, errorCode, and message, it returns a structured error response. Otherwise, it defaults to a **500 Internal Server Error** with the error message. This utility ensures cleaner error handling by eliminating the need for repetitive try-catch blocks in controllers.

**File:** recipes-api/index.ts

**import** cors **from** 'cors';

**import** dotenv **from** 'dotenv';

dotenv.**config**();

**import** express **from** 'express';

**import** { connect **as** mongoConnect } **from** 'mongoose';

**import** { authRouter } **from** './src/features/auth';

**import** { recipeRouter } **from** './src/features/recipe';

**const** app = **express**();

**const** port = process.env.port || 3001;

**const** MONGO\_URI = (process.env.MONGO\_URI || '')

.**replace**('<db\_username>', process.env.MONGO\_USERNAME || '')

.**replace**('<db\_password>', process.env.MONGO\_PASSWORD || '');

**const** MONGO\_CONNECTION\_SUCCESS = `server is connected with the MongoDB cluster!`;

**const** MONGO\_CONNECTION\_FAIL = `Connection error - server failed to connect with the MongoDB cluster...`;

**const** SERVER\_SUCCESS = `${process.env.NODE\_ENV} - server is listening on port ${port}`;

**const** SERVER\_FAIL = `${process.env.NODE} = server failed on port ${port}...`;

**const** **mongoConnectWithRetry** = (uri, retries = 5, delay = 3000) => {

**mongoConnect**(uri)

.**then**(() => {

console.**log**(MONGO\_CONNECTION\_SUCCESS);

})

.**catch**((error) => {

console.**log**(`${MONGO\_CONNECTION\_FAIL}: ${error.message}`);

**if** (retries > 0) {

console.**log**(`Retrying in ${delay / 5000} seconds...`);

setTimeout(() => {

**mongoConnectWithRetry**(uri, retries - 1, delay);

}, delay);

} **else** {

console.**log**('Max retries reached. Could not connect to MongoDB.');

}

});

};

// Call the connection function

**mongoConnectWithRetry**(MONGO\_URI);

app.**use**(express.**json**());

app.**use**(**cors**());

//////////////////////////////////////////////////////////

app.**use**('/api/auth', authRouter);

app.**use**('/api/recipes', recipeRouter);

app.**get**('/', (\_req, res) => {

res.**send**('Server api');

});

app

.**listen**(port, (): void => {

console.**log**(SERVER\_SUCCESS);

})

.**on**('error', (e): void => {

console.**log**(SERVER\_FAIL);

console.**error**(e);

});

This file sets up the Express server, configures middleware, and connects to the MongoDB database. It loads environment variables using dotenv, enables CORS for cross-origin requests, and defines API routes for authentication (authRouter) and recipes (recipeRouter). The mongoConnectWithRetry function ensures a reliable database connection, retrying up to five times if the initial connection fails. The server listens on the specified port (process.env.PORT || 3001) and provides a root endpoint (/) that responds with "Server api". If an error occurs while starting the server, it logs the failure.

Sources

**General Development and Documentation**

**• GitHub: For version control and code collaboration.**[**https://github.com**](https://github.com) **• React Documentation: Official guide for building user interfaces with React.**[**https://react.dev/**](https://react.dev/) **• Tailwind CSS Documentation: Utility-first CSS framework for modern styling.**[**https://tailwindcss.com/docs/installation**](https://tailwindcss.com/docs/installation)

**Database Management**

**• MongoDB: Official documentation for working with MongoDB.**[**https://www.mongodb.com/docs/**](https://www.mongodb.com/docs/) **• Mongoose: Simplifies MongoDB interactions in Node.js applications.**[**https://mongoosejs.com/docs/**](https://mongoosejs.com/docs/)