Frontend

The frontend architecture consists of React components using React Router for navigation, Redux for state management, and Tailwind CSS for styling.

Note: Only The most important and major functionality in fronted are covered.

Pages

1. Home Page (`Home.jsx`)

The landing page of the application that showcases different categories of patients.

Key Features:

- Hero section with welcoming message and healthcare-focused content
- Image slider using Swiper for featured insurance listings
- Three sections of patient listings:
    - Insurance Patients
    - Non-Critical Patients
    - Critical Patients
- Each section displays up to 4 listings with options to view more

Data Fetching:

- Fetches three types of listings on component mount:
    - Insurance listings (`/api/listing/get?insurance=true`)
    - Non-critical listings (`/api/listing/get?type=notcritical`)
    - Critical listings (`/api/listing/get?type=critical`)

2. Search Page (`Search.jsx`)

A comprehensive search interface for finding and filtering patient listings.

Key Features:

- Advanced filtering options:
    - Free text search

- ○ Patient type filter (Critical/Non-Critical)
- ○ Insurance status
- ○ Medical terms (Diagnosed/Lab Report)
- ○ Sorting options (Age, Creation Date)
- Responsive layout with sidebar filters
- Pagination with "Show More" functionality
- Real-time URL updates with search parameters

Search Parameters:

- `searchTerm`: Free text search
- `type`: all/notcritical/critical
- `diagnosed`: boolean
- `labreport`: boolean
- `insurance`: boolean
- `sort`: created_at/age
- `order`: asc/desc

3. Sign Up Page (`SignUp.jsx`)

User registration page with multiple authentication options.

Key Features:

- Form fields for:
    - ○ Username
    - ○ Email
    - ○ Password
    - ○ Occupation (doctor/patient)
- OAuth integration for alternative sign-up methods
- Form validation and error handling
- Navigation link to Sign In page
- Loading states during form submission

API Integration:

- POST request to `/api/auth/signup` for user registration
- Supports both traditional and OAuth authentication methods

Common Components Used

- `ListingItem`: Reusable component for displaying patient listings
- `OAuth`: Component for handling OAuth authentication
- `Swiper`: Third-party component for image slider functionality

State Management

- Local state management using React hooks (`useState`, `useEffect`)
- URL-based state management for search parameters
- Form state handling for user input

Navigation

- Uses `react-router-dom` for routing
- `useNavigate` hook for programmatic navigation
- `Link` component for declarative navigation

Styling

- Tailwind CSS for styling
- **Responsive design with mobile-first approac**h
- Custom classes for specific components
- Flex and grid layouts for responsive content organization

Error Handling

- Form validation errors
- API error handling
- Loading states for better user experience

Components

1. Header Component

`Header.jsx` - Main navigation component

Features:

- Responsive navigation bar with gradient background
- Search functionality with URL parameter synchronization
- Dynamic user authentication state display

- Brand logo and navigation links

Props: None

State Management:

- `searchTerm`: Manages search input
- `currentUser`: Retrieved from Redux store

Key Functions:

javascript
Copy

```javascript
handleSubmit(e): // Handles search form submission
```

Styling:

- Gradient background: teal to blue
- Responsive design with mobile considerations
- Hover effects on navigation items

---

2. ListingItem Component

`ListingItem.jsx` - Card component for displaying medical case listings

Props:

javascript

```javascript
{
  listing: {
    _id: string,
    imageUrls: array,
    name: string,
    past_treatment: string,
    disease_description: string,
    insurance: boolean,
    insurance_no: number,
    type: string,
    heartrate_bpm: number,
    bloodpressure_mm_Hg: number
  }
}
```

3. OAuth Component

`OAuth.jsx` - Google authentication implementation

Features:

- Google Sign-In integration
- Redux state management
- Navigation after successful authentication
- Error handling

Dependencies:

- Firebase Authentication
- Redux dispatch
- React Router navigation

Functions:

javascript

```
handleGoogleClick(): // Manages Google authentication flow
```

Backend

Authentication API Documentation

Base URL

`/api/auth`

Endpoints

Sign Up

Creates a new user account.

Route: POST /signup

Request Body:

Json

```json
{
  "username": "string",
  "email": "string",
  "password": "string",
  "occupation": "string"
}
```

Response:

- Status: 201 Created
- Content: "User created successfully!"

Error Responses:

- Various error status codes depending on validation failures or database errors

Sign In

Authenticates a user and creates a session.

Route: POST /signin

Request Body:

json

```json
{
  "email": "string",
  "password": "string"
}
```

Response:

- Status: 200 OK
- Content: User object (excluding password)

- Sets HTTP-only cookie `access_token` containing JWT

Error Responses:

- 404: User not found
- 401: Wrong credentials

Google Authentication

Handles Google OAuth authentication.

Route: POST /google

Request Body:

json

```
{
  "email": "string",
  "name": "string",
  "photo": "string (optional)"
}
```

Response:

- Status: 200 OK
- Content: User object (excluding password)
- Sets HTTP-only cookie `access_token` containing JWT

Notes:

- If user doesn't exist, creates new account with:
  - Random generated password
  - Username derived from Google name
  - Default occupation set to "doctor"
  - Google profile photo (if provided)

Sign Out

Logs out the current user.

Route: GET /signout

Response:

- Status: 200 OK
- Content: `"User has been logged out!"`
- Clears `access_token` cookie

Security Features

1. **Password Hashing:**
   - Uses bcryptjs with salt round of 10
   - Passwords are never stored in plain text
2. **JWT Authentication:**
   - Tokens are signed with environment-specific secret
   - Delivered via HTTP-only cookies for XSS protection
3. **Error Handling:**
   - Custom error handling middleware
   - Sanitized error responses

Technical Implementation Details

- Uses Express.js for routing
- MongoDB for user storage (via Mongoose)
- JWT for session management
- bcryptjs for password hashing
- Cookie-based token storage

Listing API Documentation

Base URL

`/api/listing`

Authentication

Most endpoints require authentication using a JWT token (marked with 🔒). Token must be provided via HTTP-only cookie named `access_token`.

Endpoints

Create Listing

Creates a new listing.

Route: `POST /create` 🔒

Authentication Required: Yes

Request Body:

json
```
{
  "name": "string",
  "insurance": "boolean",
  "labreport": "boolean",
  "diagnosed": "boolean",
  "type": "string (critical/notcritical)",
  // other listing fields
}
```

Response:

- Status: 201 Created
- Content: Created listing object

Delete Listing

Deletes a specific listing.

Route: `DELETE /delete/:id` 🔒

Authentication Required: Yes

URL Parameters:

- `id`: Listing ID

Authorization:

- User can only delete their own listings

Response:

- Status: 200 OK

- Content: "Listing has been deleted!"

Error Responses:

- 404: Listing not found
- 401: Unauthorized (not the listing owner)

Update Listing

Updates a specific listing.

Route: POST /update/:id 🔒

Authentication Required: Yes

URL Parameters:

- id: Listing ID

Authorization:

- User can only update their own listings

Request Body:

json
```
{
  // Any listing fields that need to be updated
}
```

Response:

- Status: 200 OK
- Content: Updated listing object

Error Responses:

- 404: Listing not found
- 401: Unauthorized (not the listing owner)

Get Single Listing

Retrieves a specific listing.

Route: `GET /get/:id`

Authentication Required: No

URL Parameters:

- `id`: Listing ID

Response:

- Status: 200 OK
- Content: Listing object

Error Responses:

- 404: Listing not found

Get Listings

Retrieves multiple listings with filtering, sorting, and pagination.

Route: `GET /get`

Authentication Required: No

Query Parameters:

- `limit`: Number of listings to return (default: 9)
- `startIndex`: Starting index for pagination (default: 0)
- `insurance`: Filter by insurance status (boolean)
- `labreport`: Filter by lab report availability (boolean)
- `diagnosed`: Filter by diagnosis status (boolean)
- `type`: Filter by type ("critical"/"notcritical"/"all")
- `searchTerm`: Search listings by name (string)
- `sort`: Field to sort by (default: "createdAt")
- `order`: Sort order ("desc"/"asc", default: "desc")

Response:

- Status: 200 OK
- Content: Array of listing objects

Filter Behavior:

- When insurance, labreport, or diagnosed filters are undefined or false, returns both true and false values
- When type is undefined or "all", returns both "critical" and "notcritical" listings
- Search term uses case-insensitive regex matching on the name field

Query Examples

Basic Search

```
GET /get?searchTerm=cardiology&limit=10
```

Filtered Search

```
GET /get?type=critical&insurance=true&diagnosed=true&limit=20&startIndex=0
```

Sorted Search

```
GET /get?sort=createdAt&order=asc&limit=15
```

Technical Implementation Details

- MongoDB queries with Mongoose
- Case-insensitive search using regex
- Pagination using limit and skip
- Dynamic filter construction
- Error handling middleware
- Token-based authentication checks

Error Handling

All endpoints use a centralized error handling middleware that:

- Catches and processes all errors
- Returns appropriate HTTP status codes
- Provides meaningful error messages

User API Documentation

Base URL

`/api/user`

Authentication

Most endpoints require authentication using a JWT token (marked with 🔒). Token must be provided via HTTP-only cookie named `access_token`.

Endpoints

Test Route

Simple route to verify API functionality.

Route: `GET /test`

Authentication Required: No

Response:

- Status: 200 OK
- Content:

json
```
{
  "message": "Api route is working!"
}
```

Update User

Updates a user's profile information.

Route: `POST /update/:id` 🔒

Authentication Required: Yes

URL Parameters:

- `id`: User ID

Authorization:

- Users can only update their own profile

Request Body:

json

```
{
  "username": "string (optional)",
  "email": "string (optional)",
  "password": "string (optional)",
  "avatar": "string (optional)"
}
```

Response:

- Status: 200 OK
- Content: Updated user object (excluding password)

Security Features:

- Automatically hashes new passwords
- Excludes password from response
- Validates user authorization

Error Responses:

- 401: Unauthorized (not the account owner)

Delete User

Deletes a user's account.

Route: `DELETE /delete/:id` 🔒

Authentication Required: Yes

URL Parameters:

- `id`: User ID

Authorization:

- Users can only delete their own account

Response:

- Status: 200 OK
- Content: `"User has been deleted!"`
- Clears `access_token` cookie

Error Responses:

- 401: Unauthorized (not the account owner)

Get User Listings

Retrieves all listings created by a specific user.

Route: `GET /listings/:id` 🔒

Authentication Required: Yes

URL Parameters:

- `id`: User ID

Authorization:

- Users can only view their own listings

Response:

- Status: 200 OK
- Content: Array of listing objects

Error Responses:

- 401: Unauthorized (not the account owner)

Get User

Retrieves a user's profile information.

Route: `GET /:id` 🔒

Authentication Required: Yes

URL Parameters:

- `id`: User ID

Response:

- Status: 200 OK
- Content: User object (excluding password)

Error Responses:

- 404: User not found

Data Security

1. **Password Protection:**
   - Passwords are hashed using bcryptjs (salt round: 10)
   - Passwords are never returned in responses
   - Password updates automatically trigger rehashing
2. **Authorization:**
   - All endpoints verify the requesting user's identity
   - Users can only modify their own data
   - Token-based authentication required for sensitive operations
3. **Data Privacy:**
   - Sensitive data is stripped from responses
   - User data is compartmentalized
   - Cookie-based token storage for enhanced security

Technical Implementation Details

- MongoDB/Mongoose for data persistence
- JWT-based authentication
- bcryptjs for password hashing
- Express.js routing
- Middleware-based authorization checks

Example Requests

Update Profile

http

```
POST /api/user/update/123456
Content-Type: application/json
```

```json
{
  "username": "newusername",
  "email": "newemail@example.com",
  "avatar": "https://example.com/avatar.jpg"
}
```

Get User Listings

http
```
GET /api/user/listings/123456
```

Get User Profile

http
```
GET /api/user/123456
```

Error Handling

The API uses a centralized error handling system that:

- Provides consistent error formats
- Includes appropriate HTTP status codes
- Returns descriptive error messages
- Handles both operational and programming errors

Database Models Documentation

User Model

Schema Overview

The User model represents user accounts in the system, storing essential profile information and authentication details.

Collection Name

```
users
```

Fields

Field	Type	Required	Unique	Default	Description
username	String	Yes	Yes	-	Unique username for the account
email	String	Yes	Yes	-	User's email address, used for authentication
password	String	Yes	No	-	Hashed password string
occupation	String	Yes	No	-	User's professional occupation(doctor/patient)
avatar	String	No	No	✓	URL to user's profile picture
timestamps	Object	Auto	-	-	Contains createdAt and updatedAt dates

Default Values

- **avatar**: "https://cdn.pixabay.com/photo/2015/10/05/22/37/blank-profile-picture-973460_1280.png"

Special Properties

- Automatically manages `createdAt` and `updatedAt` timestamps
- Username and email must be unique across all users
- Password is stored in hashed format (hashing handled by authentication controller)

Listing Model

Schema Overview

The Listing model represents medical case listings, storing patient information and medical details.

Collection Name

`listings`

Fields

Field	Type	Required	Description
name	String	Yes	Patient/case name
disease_description	String	Yes	Detailed description of the medical condition
past_treatment	String	Yes	History of treatments received
age	Number	Yes	Patient's age
insurance_no	Number	Yes	Insurance policy number
bloodpressure_mm_Hg	Number	Yes	Blood pressure measurement in mm Hg
heartrate_bpm	Number	Yes	Heart rate measurement in beats per minute
labreport	Boolean	Yes	Indicates if lab reports are available
diagnosed	Boolean	Yes	Indicates if condition has been diagnosed
type	String	Yes	Case type classification
insurance	Boolean	Yes	Indicates if patient has insurance coverage
imageUrls	Array	Yes	Collection of related medical image URLs
userRef	String	Yes	Reference to the creating user's ID
timestamps	Object	Auto	Contains createdAt and updatedAt dates

Type Field Values

- `"critical"`: Indicates urgent medical attention needed
- `"notcritical"`: Indicates routine medical case

Special Properties

- Automatically manages `createdAt` and `updatedAt` timestamps
- `userRef` creates a relationship with the User model

- `imageUrls` stores an array of strings representing image URLs

Relationships

User to Listings (One-to-Many)

- One user can create multiple listings
- Each listing must belong to exactly one user
- Relationship maintained through `userRef` field in Listing model
- `userRef` stores the `_id` of the creating user

Indexing Considerations

User Model Indexes

- Automatic index on `_id` (MongoDB default)
- Unique indexes on:
  - `username`
  - `email`

Listing Model Indexes

- Automatic index on `_id` (MongoDB default)
- Recommended indexes:
  - `userRef` (for quick user-based queries)
  - `type` (for filtering by case type)
  - `diagnosed` (for filtering by diagnosis status)

Data Validation

User Model Validation

- Username must be unique
- Email must be unique and valid format
- Password must be provided (hashed before storage)
- Occupation must be provided

Listing Model Validation

- All numeric fields must be positive numbers
- Boolean fields must be true/false

- Type must be either "critical" or "notcritical"
- ImageUrls must contain at least one entry
- UserRef must be a valid user ID

Usage Examples

Creating a New User

javascript

```javascript
const user = new User({
  username: "doctorsmith",
  email: "smith@hospital.com",
  password: "hashedPassword",
  occupation: "doctor"
});
```

Creating a New Listing

javascript

```javascript
const listing = new Listing({
  name: "John Doe Case",
  disease_description: "Chronic heart condition",
  past_treatment: "Beta blockers",
  age: 45,
  insurance_no: 123456,
  bloodpressure_mm_Hg: 140,
  heartrate_bpm: 75,
  labreport: true,
  diagnosed: true,
  type: "critical",
  insurance: true,
  imageUrls: ["url1", "url2"],
  userRef: "user_id_here"
});
```