# **Full Handover Documentation**

## **1. Setting Up Accounts and Services**

### **A. CFIT GitHub Account (Wintec's GitHub Account)**

1. **Access the CFIT GitHub Account:**
   * Ensure you have access to the CFIT GitHub account provided by Wintec.
   * If not, request access from the relevant authorities at Wintec.
2. **Create a New Repository:**
   * Go to [GitHub](https://github.com) and log in to the CFIT GitHub account.
   * Click on "New" to create a new repository.
   * Fill in the repository name, description, and select the appropriate visibility (public/private).
   * Click "Create repository".

### **B. MongoDB Atlas**

1. **Create a MongoDB Atlas Account:**
   * Go to [MongoDB Atlas](https://www.mongodb.com/cloud/atlas) and click on "Try Free".
   * Sign up using your email or a preferred method.
   * Verify your email address.
2. **Set Up Your Cluster:**
   * Once logged in, click on "Build a Cluster".
   * Select the free tier for the cluster.
   * Choose a cloud provider and region (default settings are usually fine).
   * Click "Create Cluster".
3. **Configure Network Access:**
   * Go to "Network Access" and click on "Add IP Address".
   * Add your IP address or allow access from anywhere (0.0.0.0/0) for development purposes.
   * Click "Confirm".
4. **Create a Database User:**
   * Go to "Database Access" and click on "Add New Database User".
   * Create a user with a username and password.
   * Assign the user roles (e.g., readWrite on your database).
   * Click "Add User".
5. **Get Connection String:**
   * Go to "Clusters" and click on "Connect".
   * Select "Connect Your Application".
   * Copy the connection string provided.

### **C. Netlify**

1. **Create a Netlify Account:**
   * Go to [Netlify](https://www.netlify.com) and sign up.
   * Verify your email address.
2. **Connect to GitHub:**
   * Click on "New site from Git" and connect your GitHub account.
   * Select the CFIT GitHub repository you want to deploy.
3. **Configure Build Settings:**
   * Set the build command to npm run build.
   * Set the publish directory to build.
4. **Deploy Site:**
   * Click "Deploy site" and wait for the build process to complete.
   * Once deployed, you will receive a live URL for your site.

### **D. Railway**

1. **Create a Railway Account:**
   * Go to [Railway](https://railway.app) and sign up.
   * Link your GitHub account to Railway.
2. **Create a New Project:**
   * Click on "New Project" and select your GitHub repository.
   * Configure environment variables required by your application (e.g., MongoDB connection string).
3. **Deploy Project:**
   * Click on "Deploy" and monitor the logs to ensure the deployment is successful.

## **2. Cost Breakdown**

* **GitHub (CFIT):** $0 (GitHub Free for private repositories)
* **MongoDB Atlas:**
  + Free Tier: $0 (Shared clusters with limited resources)
  + Paid Tiers: Starting at $9/month for dedicated clusters (price varies with configuration)
* **Netlify:**
  + Free Tier: $0 (Includes basic CI/CD and hosting)
  + Paid Plans: Starting at $19/month for additional features
* **Railway:**
  + Free Tier: $0 (Limited usage and resources)
  + Paid Plans: Usage-based pricing (approximately $5/month for basic usage)

## **3. Step-by-Step Guide**

### **A. Downloading Repo from CFIT GitHub Account**

1. **Clone Repository:**
   * Open GitHub and navigate to the CFIT organisation repository.
   * Click on the repository you want to clone.
   * Click on the "Code" button and copy the HTTPS URL.
2. **Clone in Visual Studio Code:**
   * Open Visual Studio Code.
   * Open the terminal (Ctrl + ).

Run the command:  
sh  
Copy code  
git clone <repository-url>

Navigate into the cloned directory:  
sh  
Copy code  
cd <repository-name>

### **B. Installing NPM Modules**

1. **Install Dependencies:**
   * Open the terminal in Visual Studio Code.

Run:  
sh  
Copy code  
npm install

* + This will install all required dependencies listed in package.json.

### **C. Testing and Running Locally**

1. **Run Local Server:**

After installing dependencies, run:  
sh  
Copy code  
npm start

* + Open your browser and navigate to http://localhost:3000 to see your application running.

### **D. Deploying on Netlify**

1. **Connect Repository:**
   * Log in to Netlify.
   * Click "New site from Git" and select your GitHub repository.
   * Follow the prompts to deploy your site.
2. **Configure Build Settings:**
   * Set the build command to npm run build.
   * Set the publish directory to build.
3. **Deploy Site:**
   * Click "Deploy site" and wait for the build process to complete.
   * Once deployed, you will receive a live URL for your site.

### **E. Configuring MongoDB Atlas**

1. **Create Cluster:**
   * In MongoDB Atlas, create a new cluster (free tier is available).
2. **Create Database User:**
   * Go to the "Database Access" tab and create a new user with readWrite permissions.
3. **Configure Network Access:**
   * Go to the "Network Access" tab and whitelist your IP address.
4. **Connect to Cluster:**
   * Obtain the connection string from the "Clusters" tab.
   * Use this connection string in your application to connect to the MongoDB database.
5. mongodb://mongo:irQdmHfVMmhOibwBtMpVkpyTBHmJmJGh@roundhouse.proxy.rlwy.net:36424

### **F. Deploying Backend on Railway**

1. **Create Project:**
   * Log in to the Railway and create a new project.
   * Link your GitHub repository to the project.
2. **Configure Environment Variables:**
   * Set up the required environment variables for your backend. These typically include:
     + Database connection strings (e.g., MongoDB URI, MySQL host, username, password)
     + API keys or secrets
     + Any other configuration specific to your backend application.
3. **Install Dependencies:**
   * Open your terminal or command prompt.
   * Navigate to your backend project directory.
   * Run npm install or npm i to install all necessary dependencies specified in your package.json file.
4. **Start Backend Locally:**
   * After dependencies are installed, ensure your backend runs locally without issues.
   * Run npm run start to start your backend server locally. This command should start your Node.js server and make it accessible on http://localhost:your\_port.
5. **Deploy Project on Railway:**
   * Once your backend runs locally, commit any changes to your GitHub repository.
   * In your Railway project dashboard, click on the "Deploy" button.
   * Railway will fetch your code from GitHub, build your application, and deploy it automatically.
6. **Monitor Deployment:**
   * Monitor the deployment logs provided by Railway to ensure everything is deploying correctly.
   * Look for any errors or warnings during the deployment process and address them if necessary.
7. **Backend URL from Railway:**
   * After successful deployment, Railway will provide a public URL for your backend application.
   * This URL will be accessible over the internet and can be used to interact with your backend APIs.
8. **Frontend URL from Netlify:**
   * Use the URL of your frontend application deployed on Netlify for user interactions.
   * This is where your users will access the frontend of your application, which communicates with your backend deployed on Railway.
9. **Using Navicat Software (Optional):**
   * If you need to manage databases or perform database-related tasks, use Navicat software.
   * Connect to your database using the appropriate credentials and manage your data as required.

## **4. Handing Over**

### **Documentation**

1. **Repository Access:**
   * Provide the client with access to the CFIT GitHub repository.
   * Share the repository URL and cloning instructions.
2. **Environment Setup:**
   * Provide a detailed README file in the repository with setup instructions.
   * Include steps to install dependencies, run the project locally, and deploy.
3. **Configuration Details:**
   * Share the MongoDB Atlas cluster details and access credentials securely.
   * Provide Netlify and Railway access if necessary.

### **Handing Over**

#### **1. Repository Access**

1. **Provide Access to CFIT GitHub Repository:**
   * Ensure the client has a GitHub account.
   * Go to the repository on the CFIT GitHub account.
   * Click on "Settings" -> "Manage access".
   * Click on "Invite a collaborator".
   * Enter the client’s GitHub username or email and click "Add collaborator".
2. **Share Repository URL:**
   * Copy the repository URL and send it to the client.

Example URL format:  
bash  
Copy code  
https://github.com/CFIT-organization/repository-name

1. **Cloning Instructions:**

Provide the following cloning instructions to the client:  
markdown  
Copy code  
### Cloning the Repository

1. Open your terminal or command prompt.

2. Run the following command to clone the repository:

```sh

git clone https://github.com/CFIT-organization/repository-name

Navigate into the cloned directory:  
sh  
Copy code  
cd repository-name

#### **2. Environment Setup**

1. **Detailed README File:**

Ensure the repository includes a comprehensive README.md file with the following sections:  
markdown  
Copy code  
# Project Name

## Getting Started

### Prerequisites

- Node.js

- npm

### Installation

1. Clone the repository:

```sh

git clone https://github.com/CFIT-organization/repository-name

cd repository-name

Install dependencies:  
sh  
Copy code  
npm install

* + **Running Locally**

Start the development server:  
sh  
Copy code  
npm start

* + 1. Open your browser and navigate to http://localhost:3000.
  + **Deployment  
    Netlify**
    1. Connect your GitHub repository to Netlify.
    2. Set the build command to npm run build.
    3. Set the publish directory to build.
  + **Railway**
    1. Connect your GitHub repository to Railway.
    2. Configure the required environment variables.
    3. Deploy the project.
  + **Configuration  
    MongoDB Atlas**
    1. Obtain the connection string from MongoDB Atlas.
    2. Update the connection string with your credentials.
    3. Set up environment variables in your .env file.
    4. Use environment variables in your application to connect to MongoDB.
  + Copy code

1. **Environment Variables:**

Provide instructions for setting up environment variables:  
markdown  
Copy code  
### Setting Up Environment Variables

1. Create a `.env` file in the root directory of your project.

2. Add the following environment variables to the `.env` file:

```env

MONGODB\_URI=mongodb+srv://<username>:<password>@cluster0.mongodb.net/<dbname>?retryWrites=true&w=majority

* + 1. Ensure the .env file is added to .gitignore to prevent sensitive information from being committed to version control.
  + Copy code

#### **3. Configuration Details**

1. **MongoDB Atlas Credentials:**
   * Provide the client with the MongoDB Atlas cluster details and access credentials securely (e.g., via a secure email or password manager).
2. **Netlify and Railway Access:**
   * If necessary, provide the client with access to Netlify and Railway accounts used for deployment.
   * Ensure the client knows how to manage deployments and configurations on these platforms.

#### **4. Ongoing Support**

1. **Support Period:**
   * Offer a brief period of support (e.g., 1-2 weeks) to help with any issues or questions during the transition.
   * Provide contact details and preferred communication channels (e.g., email, Slack).
2. **Documentation and Tutorials:**
   * Share any additional documentation or tutorials that might help the client understand the project better.
   * Include links to relevant resources and official documentation for the tools and services used.

### **Example README Content**

markdown

Copy code

# Project Name

## Getting Started

### Prerequisites

- Node.js

- npm

### Installation

1. Clone the repository:

```sh

git clone https://github.com/CFIT-organization/repository-name

cd repository-name

Install dependencies:  
sh  
Copy code  
npm install

### **Running Locally**

Start the development server:  
sh  
Copy code  
npm start

1. Open your browser and navigate to http://localhost:3000.

### **Deployment**

#### **Netlify**

1. Connect your GitHub repository to Netlify.
2. Set the build command to npm run build.
3. Set the publish directory to build.

#### **Railway**

1. Connect your GitHub repository to Railway.
2. Configure the required environment variables.
3. Deploy the project.

### **Configuration**

#### **MongoDB Atlas**

1. Obtain the connection string from MongoDB Atlas.
2. Update the connection string with your credentials.

Set up environment variables in your .env file:  
env  
Copy code  
MONGODB\_URI=mongodb+srv://<username>:<password>@cluster0.mongodb.net/<dbname>?retryWrites=true&w=majority

Use environment variables in your application to connect to MongoDB:  
javascript  
Copy code  
const mongoose = require('mongoose');

const dotenv = require('dotenv');

dotenv.config();

mongoose.connect(process.env.MONGODB\_URI, {

useNewUrlParser: true,

useUnifiedTopology: true

})

.then(() => console.log('MongoDB connected...'))

1. .catch(err => console.log(err));
2. **Create a Railway Account**:
   1. Sign up for an account on [Railway.app](https://railway.app/) if you haven't already.
3. **Create a New Project**:
   1. Once logged in, click on the "New Project" button on the Railway dashboard.
4. **Add MongoDB Plugin**:
   1. In your new project, click on the "Provision Database" button.
   2. Select the "MongoDB" plugin from the list of available databases.
   3. Railway will provision a new MongoDB instance for your project.
5. **Retrieve MongoDB Connection URL**:
   1. After the MongoDB instance is created, click on it to view its details.
   2. Copy the MONGODB\_URL connection string provided by Railway. This will be used to connect your application to the MongoDB database.
6. **Set Up Environment Variables**:
   1. In your Railway project, go to the "Settings" tab.
   2. Click on "Variables" to add new environment variables.
   3. Add a new environment variable named MONGODB\_URL and paste the connection string you copied from the MongoDB instance.
7. **Update Your Application to Use the MongoDB URL**:

In your application code, make sure to use the MONGODB\_URL environment variable to connect to MongoDB. For example, in a Node.js application using Mongoose, you might have something like this:  
javascript  
Copy code  
import mongoose from 'mongoose';

const mongoDBUrl = process.env.MONGODB\_URL;

mongoose.connect(mongoDBUrl, {

useNewUrlParser: true,

useUnifiedTopology: true,

}).then(() => {

console.log('Connected to MongoDB');

}).catch((error) => {

console.error('Error connecting to MongoDB:', error.message);

});

1. **Deploy Your Application**:
   1. If your application is not yet deployed on Railway, you can deploy it by connecting your GitHub repository to Railway.
   2. In your project on the Railway dashboard, click on "Deployments" and follow the instructions to connect your GitHub repository.
   3. Push your changes to GitHub, and Railway will automatically deploy your application.

### **Example Deployment with MongoDB in a Node.js Application**

1. **Install Dependencies**:

Make sure you have the necessary dependencies in your package.json:  
json  
Copy code  
{

"dependencies": {

"express": "^4.17.1",

"mongoose": "^5.11.15",

"dotenv": "^8.2.0"

}

}

1. **Use Environment Variables**:

Create a .env file in your project root and add the following line (for local development):  
plaintext  
Copy code  
MONGODB\_URL=mongodb://localhost:27017/yourdbname

1. **Update Your Application Code**:

Ensure your application reads the environment variable:  
javascript  
Copy code  
import mongoose from 'mongoose';

import dotenv from 'dotenv';

dotenv.config();

const mongoDBUrl = process.env.MONGODB\_URL;

mongoose.connect(mongoDBUrl, {

useNewUrlParser: true,

useUnifiedTopology: true,

}).then(() => {

console.log('Connected to MongoDB');

}).catch((error) => {

console.error('Error connecting to MongoDB:', error.message);

});

1. **Deploy on Railway**:
   * Commit your changes to GitHub.
   * Push to your repository.
   * Railway will automatically build and deploy your application.

## **Backend File Structure**

java

Copy code

bwaka-master/

├── .DS\_Store

├── .dockerignore

├── .env

├── .eslintrc.js

├── @types/

├── package.json

├── src/

│ ├── app/

│ │ ├── controllers/

│ │ │ ├── authController.ts

│ │ │ ├── bookingController.ts

│ │ │ ├── userController.ts

│ │ │ └── index.ts

│ │ ├── middleware/

│ │ │ ├── authMiddleware.ts

│ │ │ └── errorMiddleware.ts

│ │ ├── models/

│ │ │ ├── Booking.ts

│ │ │ ├── User.ts

│ │ │ └── index.ts

│ │ ├── routes/

│ │ │ ├── authRoutes.ts

│ │ │ ├── bookingRoutes.ts

│ │ │ └── index.ts

│ │ └── services/

│ │ ├── authService.ts

│ │ ├── bookingService.ts

│ │ ├── userService.ts

│ │ └── index.ts

│ ├── config/

│ │ ├── db.ts

│ │ ├── logger.ts

│ │ └── index.ts

│ ├── utils/

│ │ ├── constants.ts

│ │ ├── helpers.ts

│ │ └── index.ts

│ ├── app.ts

│ └── server.ts

├── tsconfig.json

└── yarn.lock

### **Explanation of Key Directories and Files**

1. **.dockerignore**: Specifies files and directories to be ignored by Docker.
2. **.env**: Contains environment variables such as database connection strings and API keys.
3. **package.json**: Lists the project dependencies and scripts.
4. **src/**: Main source directory containing the application code.
   * **app/**: Contains application-specific modules.
     + **controllers/**: Handles incoming requests and interacts with services.
       - **authController.ts**: Manages authentication-related requests.
       - **bookingController.ts**: Manages booking-related requests.
       - **userController.ts**: Manages user-related requests.
     + **middleware/**: Middleware functions for handling authentication and errors.
     + **models/**: Database models for entities like Booking and User.
     + **routes/**: Defines API routes.
     + **services/**: Business logic and interactions with the database.
   * **config/**: Configuration files for database and logging.
   * **utils/**: Utility functions and constants.
   * **app.ts**: Application setup and middleware configuration.
   * **server.ts**: Entry point for starting the server.
5. **tsconfig.json**: TypeScript configuration file.
6. **yarn.lock**: Ensures consistent installation of dependencies.

## **API Endpoints**

### **Authentication Endpoints**

* **POST** /auth/login: User login.
* **POST** /auth/register: User registration.
* **POST** /auth/logout: User logout.

### **Booking Endpoints**

* **GET** /bookings: Retrieve all bookings.
* **POST** /bookings: Create a new booking.
* **GET** /bookings/:id: Retrieve a specific booking by ID.
* **PUT** /bookings/:id: Update a booking by ID.
* **DELETE** /bookings/:id: Delete a booking by ID.

### **User Endpoints**

* **GET** /users: Retrieve all users.
* **GET** /users/:id: Retrieve a specific user by ID.
* **PUT** /users/:id: Update a user by ID.
* **DELETE** /users/:id: Delete a user by ID.

### **Example Endpoint Implementations**

#### **Authentication Controller (authController.ts)**

typescript

Copy code

import { Request, Response } from 'express';

import authService from '../services/authService';

export const login = async (req: Request, res: Response) => {

const { email, password } = req.body;

const token = await authService.login(email, password);

res.status(200).json({ token });

};

export const register = async (req: Request, res: Response) => {

const { email, password } = req.body;

const user = await authService.register(email, password);

res.status(201).json({ user });

};

// More functions...

#### **Booking Controller (bookingController.ts)**

typescript

Copy code

import { Request, Response } from 'express';

import bookingService from '../services/bookingService';

export const getAllBookings = async (req: Request, res: Response) => {

const bookings = await bookingService.getAllBookings();

res.status(200).json(bookings);

};

export const createBooking = async (req: Request, res: Response) => {

const bookingData = req.body;

const newBooking = await bookingService.createBooking(bookingData);

res.status(201).json(newBooking);

};

// More functions...

### **Example Route Implementations**

#### **Authentication Routes (authRoutes.ts)**

typescript

Copy code

import { Router } from 'express';

import { login, register, logout } from '../controllers/authController';

const router = Router();

router.post('/login', login);

router.post('/register', register);

router.post('/logout', logout);

export default router;

#### **Booking Routes (bookingRoutes.ts)**

typescript

Copy code

import { Router } from 'express';

import { getAllBookings, createBooking, getBookingById, updateBooking, deleteBooking } from '../controllers/bookingController';

const router = Router();

router.get('/', getAllBookings);

router.post('/', createBooking);

router.get('/:id', getBookingById);

router.put('/:id', updateBooking);

router.delete('/:id', deleteBooking);

export default router;

**# Backend Folder Documentation**

## Project Structure

```

bwaka-master/

├── .dockerignore

├── .env

├── .eslintrc.js

├── @types/

│ ├── index.d.ts

├── package.json

├── src/

│ ├── app/

│ │ ├── controllers/

│ │ │ ├── authController.ts

│ │ │ ├── bookingController.ts

│ │ │ ├── userController.ts

│ │ │ └── index.ts

│ │ ├── middleware/

│ │ │ ├── authMiddleware.ts

│ │ │ └── errorMiddleware.ts

│ │ ├── models/

│ │ │ ├── Booking.ts

│ │ │ ├── User.ts

│ │ │ └── index.ts

│ │ ├── routes/

│ │ │ ├── authRoutes.ts

│ │ │ ├── bookingRoutes.ts

│ │ │ └── index.ts

│ │ ├── services/

│ │ │ ├── authService.ts

│ │ │ ├── bookingService.ts

│ │ │ ├── userService.ts

│ │ │ └── index.ts

│ ├── config/

│ │ ├── db.ts

│ │ ├── logger.ts

│ │ └── index.ts

│ ├── utils/

│ │ ├── constants.ts

│ │ ├── helpers.ts

│ │ └── index.ts

│ ├── app.ts

│ └── server.ts

├── tsconfig.json

└── yarn.lock

```

### Explanation of Key Directories and Files

1. \*\*`.dockerignore`\*\*: Specifies files and directories to be ignored by Docker.

2. \*\*`.env`\*\*: Contains environment variables such as database connection strings and API keys.

3. \*\*`.eslintrc.js`\*\*: Configuration file for ESLint, used to enforce code quality and style guidelines.

4. \*\*`@types/`\*\*: Directory containing TypeScript declaration files.

5. \*\*`package.json`\*\*: Lists the project dependencies and scripts.

6. \*\*`src/`\*\*: Main source directory containing the application code.

- \*\*`app/`\*\*: Contains application-specific modules.

- \*\*`controllers/`\*\*: Handles incoming requests and interacts with services.

- \*\*`authController.ts`\*\*: Manages authentication-related requests.

- \*\*`bookingController.ts`\*\*: Manages booking-related requests.

- \*\*`userController.ts`\*\*: Manages user-related requests.

- \*\*`middleware/`\*\*: Middleware functions for handling authentication and errors.

- \*\*`authMiddleware.ts`\*\*: Middleware for handling authentication logic.

- \*\*`errorMiddleware.ts`\*\*: Middleware for handling errors and exceptions.

- \*\*`models/`\*\*: Database models for entities like Booking and User.

- \*\*`Booking.ts`\*\*: Schema definition and model for booking data.

- \*\*`User.ts`\*\*: Schema definition and model for user data.

- \*\*`routes/`\*\*: Defines API routes.

- \*\*`authRoutes.ts`\*\*: Routes for authentication-related endpoints.

- \*\*`bookingRoutes.ts`\*\*: Routes for booking-related endpoints.

- \*\*`services/`\*\*: Business logic and interactions with the database.

- \*\*`authService.ts`\*\*: Contains logic for authentication operations.

- \*\*`bookingService.ts`\*\*: Contains logic for booking operations.

- \*\*`userService.ts`\*\*: Contains logic for user operations.

- \*\*`config/`\*\*: Configuration files for database and logging.

- \*\*`db.ts`\*\*: Database connection setup and configuration.

- \*\*`logger.ts`\*\*: Logger configuration using a logging library.

- \*\*`utils/`\*\*: Utility functions and constants.

- \*\*`constants.ts`\*\*: Common constants used throughout the application.

- \*\*`helpers.ts`\*\*: Helper functions for various tasks.

- \*\*`app.ts`\*\*: Application setup and middleware configuration.

- \*\*`server.ts`\*\*: Entry point for starting the server.

7. \*\*`tsconfig.json`\*\*: TypeScript configuration file.

8. \*\*`yarn.lock`\*\*: Ensures consistent installation of dependencies.

## API Endpoints

### Authentication Endpoints

- \*\*POST\*\* `/auth/login`: User login.

- \*\*POST\*\* `/auth/register`: User registration.

- \*\*POST\*\* `/auth/logout`: User logout.

### Booking Endpoints

- \*\*GET\*\* `/bookings`: Retrieve all bookings.

- \*\*POST\*\* `/bookings`: Create a new booking.

- \*\*GET\*\* `/bookings/:id`: Retrieve a specific booking by ID.

- \*\*PUT\*\* `/bookings/:id`: Update a booking by ID.

- \*\*DELETE\*\* `/bookings/:id`: Delete a booking by ID.

### User Endpoints

- \*\*GET\*\* `/users`: Retrieve all users.

- \*\*GET\*\* `/users/:id`: Retrieve a specific user by ID.

- \*\*PUT\*\* `/users/:id`: Update a user by ID.

- \*\*DELETE\*\* `/users/:id`: Delete a user by ID.

## Example Endpoint Implementations

### Authentication Controller (authController.ts)

```typescript

import { Request, Response } from 'express';

import authService from '../services/authService';

export const login = async (req: Request, res: Response) => {

const { email, password } = req.body;

const token = await authService.login(email, password);

res.status(200).json({ token });

};

export const register = async (req: Request, res: Response) => {

const { email, password } = req.body;

const user = await authService.register(email, password);

res.status(201).json({ user });

};

// More functions...

```

### Booking Controller (bookingController.ts)

```typescript

import { Request, Response } from 'express';

import bookingService from '../services/bookingService';

export const getAllBookings = async (req: Request, res: Response) => {

const bookings = await bookingService.getAllBookings();

res.status(200).json(bookings);

};

export const createBooking = async (req: Request, res: Response) => {

const bookingData = req.body;

const newBooking = await bookingService.createBooking(bookingData);

res.status(201).json(newBooking);

};

// More functions...

```

## Example Route Implementations

### Authentication Routes (authRoutes.ts)

```typescript

import { Router } from 'express';

import { login, register, logout } from '../controllers/authController';

const router = Router();

router.post('/login', login);

router.post('/register', register);

router.post('/logout', logout);

export default router;

```

### Booking Routes (bookingRoutes.ts)

```typescript

import { Router } from 'express';

import { getAllBookings, createBooking, getBookingById, updateBooking, deleteBooking } from '../controllers/bookingController';

const router = Router();

router.get('/', getAllBookings);

router.post('/', createBooking);

router.get('/:id', getBookingById);

router.put('/:id', updateBooking);

router.delete('/:id', deleteBooking);

export default router;

```

This documentation provides a comprehensive overview of the backend folder structure, key files, and example implementations of API endpoints and routes.

**# Frontend Folder Documentation**

## Project Structure

```

frontend-main/

├── .gitignore

├── .idea/

├── README.md

├── netlify.toml

├── package-lock.json

├── package.json

├── public/

│ ├── favicon.ico

│ ├── index.html

│ ├── logo192.png

│ ├── logo512.png

│ ├── manifest.json

│ └── robots.txt

├── src/

│ ├── App.css

│ ├── App.tsx

│ ├── assets/

│ ├── components/

│ ├── index.css

│ ├── index.tsx

│ ├── logo.svg

│ ├── pages/

│ ├── react-app-env.d.ts

│ ├── reportWebVitals.ts

│ └── setupTests.ts

├── tsconfig.json

└── yarn.lock

```

### Explanation of Key Directories and Files

1. \*\*`.gitignore`\*\*: Specifies which files and directories to ignore in version control, preventing sensitive or unnecessary files from being committed to the repository.

2. \*\*`.idea/`\*\*: Directory containing project-specific settings for the IntelliJ IDEA development environment.

3. \*\*`README.md`\*\*: Markdown file providing an overview of the project, setup instructions, and other relevant information.

4. \*\*`netlify.toml`\*\*: Configuration file for deploying the site on Netlify. It specifies the build settings and redirects.

5. \*\*`package-lock.json`\*\*: Automatically generated file that ensures consistent installation of dependencies.

6. \*\*`package.json`\*\*: Lists the project dependencies, scripts, and other metadata about the project.

7. \*\*`public/`\*\*: Directory containing static assets that are publicly accessible.

- \*\*`favicon.ico`\*\*: The favicon for the application.

- \*\*`index.html`\*\*: The main HTML file for the application.

- \*\*`logo192.png`, `logo512.png`\*\*: Logo images used in the application.

- \*\*`manifest.json`\*\*: Configuration file for Progressive Web App (PWA) settings.

- \*\*`robots.txt`\*\*: Instructions for web crawlers on how to index the site.

8. \*\*`src/`\*\*: Main source directory containing the application code.

- \*\*`App.css`\*\*: CSS file for styling the main App component.

- \*\*`App.tsx`\*\*: Main App component of the application.

- \*\*`assets/`\*\*: Directory for storing asset files such as images and fonts.

- \*\*`components/`\*\*: Directory for reusable React components.

- \*\*`index.css`\*\*: Global CSS file for the application.

- \*\*`index.tsx`\*\*: Entry point for the React application.

- \*\*`logo.svg`\*\*: SVG file for the application logo.

- \*\*`pages/`\*\*: Directory for different page components of the application.

- \*\*`react-app-env.d.ts`\*\*: TypeScript declaration file for React app environment.

- \*\*`reportWebVitals.ts`\*\*: File for measuring performance metrics.

- \*\*`setupTests.ts`\*\*: Configuration file for setting up tests.

9. \*\*`tsconfig.json`\*\*: TypeScript configuration file.

10. \*\*`yarn.lock`\*\*: Ensures consistent installation of dependencies when using Yarn.

## Example File Implementations

### `index.html`

This is the main HTML file for your application. It sets up the basic structure and includes links to the favicon, manifest, CSS, and JavaScript files.

```html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<link rel="icon" href="%PUBLIC\_URL%/favicon.ico">

<link rel="manifest" href="%PUBLIC\_URL%/manifest.json">

<link rel="stylesheet" href="%PUBLIC\_URL%/index.css">

<title>Frontend Application</title>

</head>

<body>

<noscript>You need to enable JavaScript to run this app.</noscript>

<div id="root"></div>

<script src="%PUBLIC\_URL%/index.js"></script>

</body>

</html>

```

### `App.tsx`

This is the main App component of your application, which serves as the root component for your React application.

```tsx

import React from 'react';

import logo from './logo.svg';

import './App.css';

function App() {

return (

<div className="App">

<header className="App-header">

<img src={logo} className="App-logo" alt="logo" />

<p>

Edit <code>src/App.tsx</code> and save to reload.

</p>

<a

className="App-link"

href="https://reactjs.org"

target="\_blank"

rel="noopener noreferrer"

>

Learn React

</a>

</header>

</div>

);

}

export default App;

```

### `index.tsx`

This is the entry point for your React application. It renders the App component into the root element in the `index.html` file.

```tsx

import React from 'react';

import ReactDOM from 'react-dom';

import './index.css';

import App from './App';

import reportWebVitals from './reportWebVitals';

ReactDOM.render(

<React.StrictMode>

<App />

</React.StrictMode>,

document.getElementById('root')

);

// If you want to start measuring performance in your app, pass a function

// to log results (for example: reportWebVitals(console.log))

// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals

reportWebVitals();

```

### `App.css`

This CSS file contains styles for the App component, ensuring a consistent look and feel for the application.

```css

.App {

text-align: center;

}

.App-logo {

height: 40vmin;

pointer-events: none;

}

.App-header {

background-color: #282c34;

min-height: 100vh;

display: flex;

flex-direction: column;

align-items: center;

justify-content: center;

font-size: calc(10px + 2vmin);

color: white;

}

.App-link {

color: #61dafb;

}

```

### `manifest.json`

This manifest file configures your PWA, specifying the app's name, icons, start URL, and display properties.

```json

{

"short\_name": "ReactApp",

"name": "Create React App Sample",

"icons": [

{

"src": "favicon.ico",

"sizes": "64x64 32x32 24x24 16x16",

"type": "image/x-icon"

},

{

"src": "logo192.png",

"type": "image/png",

"sizes": "192x192"

},

{

"src": "logo512.png",

"type": "image/png",

"sizes": "512x512"

}

],

"start\_url": ".",

"display": "standalone",

"theme\_color": "#000000",

"background\_color": "#ffffff"

}

```

### `netlify.toml`

This configuration file is used by Netlify to define the build process and redirects. It ensures that your single-page application routes work correctly.

```toml

[build]

publish = "build"

command = "npm run build"

[[redirects]]

from = "/\*"

to = "/index.html"

status = 200

```

### **How They Are Linked ( Backend )**

1. **Controllers**:
   * Controllers are linked to routes. Each route file (e.g., bookings.ts) imports its corresponding controller (e.g., bookings.ts in controllers) and calls its functions when specific endpoints are hit.
2. **Models**:
   * Models are used within controllers to interact with the MongoDB database. Each controller imports the necessary model(s) to perform operations.
3. **Middleware**:
   * Middleware functions are used in routes to process requests before they reach the controllers. For example, auth.ts could be used to authenticate requests before allowing access to certain routes.
4. **Routes**:
   * Routes map HTTP requests to controller functions. For example, a GET request to /api/bookings might call a function in bookings.ts in the controllers folder.
5. **Util**:
   * Utility functions are helper functions used throughout the application to perform common tasks, such as validating environment variables, sending emails, or handling JWTs.
6. **Schedule**:
   * The schedule folder contains tasks that are run at specific times or intervals. For example, emailScheduler.ts might be responsible for sending reminder emails to users at a scheduled time.
7. **App Initialization**:
   * app.ts sets up the Express application with middleware, routes, and error handling.
   * server.ts starts the application, connects to the database using Mongoose, and starts any scheduled tasks.

### **Common Issues When Downloading from GitHub**

The errors you see when directly downloading from GitHub are usually because dependencies are not installed. Here's why:

1. **Missing Dependencies**: When you clone a repository from GitHub, the node\_modules folder (which contains all the dependencies) is not included. You need to install these dependencies using npm install or yarn install.
2. **Environment Variables**: The .env file is typically not included in version control for security reasons. You need to create this file based on the provided .env.example or documentation.
3. **TypeScript Configuration**: Ensure that tsconfig.json is properly set up to compile your TypeScript files.

**Frontend folder**

### **1. src Folder**

Contains the main source code of the application.

#### **components Folder**

Reusable React components:

* **StaffLoginModal.tsx, StaffRegisterModal.tsx**: Modals for staff login and registration.
* **UserLoggedInView.tsx, UserLoggedOutView.tsx**: Views for logged-in and logged-out users.
* **UserLoginModal.tsx, UserSignUpModal.tsx**: Modals for user login and sign-up.
* *View components*\*: Specific views for logged-in staff or users.

#### **errors Folder**

* **http\_errors.ts**: Manages HTTP error types and handling.

#### **models Folder**

TypeScript interfaces for data models:

* **bookings.ts, calendars.ts, registers.ts, rosters.ts, staff.ts, user.ts**: Interfaces for different data models.

#### **network Folder**

Handles API requests:

* **websites\_api.ts**: API functions for backend interaction.

#### **pages Folder**

Main page components:

* **AdminLink.tsx, AdminTableView.tsx**: Admin-related pages.
* **BookingPage.tsx, CalendarPage.tsx, RegisterPage.tsx**: Main functional pages.
* **ForgotPasswordPage.tsx, ResetPasswordPage.tsx**: Password management pages.
* **HomePage.tsx, NotFoundPage.tsx**: Home and 404 error pages.

#### **styles Folder**

CSS and CSS modules for styling:

* Various .module.css files: Specific styles for components and pages.
* **global.css**: Global styles.

#### **utils Folder**

Utility functions and helpers:

* **formatDate.ts**: Date formatting functions.
* **utils.module.css**: Utility CSS classes.

### **2. public Folder**

Static assets and main HTML:

* **index.html**: Main HTML file.
* **favicon.ico, logo.png**: Icons and logos.
* **manifest.json**: PWA configuration.
* **robots.txt**: Web crawler instructions.

### **3. assets/images Folder**

Image assets:

* Various image files (.jpg, .png, .webp): Used throughout the application.

### **Additional Files**

* **index.tsx**: Entry point for the React application.
* **App.tsx**: Main application component.
* **App.test.tsx**: Tests for the App component.
* **setupTests.ts**: Test setup file.
* **tsconfig.json**: TypeScript configuration.
* **package.json**: Project dependencies and scripts.
* **README.md**: Project documentation.

This structure ensures the codebase is organised, modular, and maintainable.

### **Project Structure**

1. **public**:
   * Contains static files like index.html, images, and other assets that need to be served directly by the web server.
2. **src**:
   * This is the source folder where the main application code resides. Let's break down the subfolders and files within src:
   * **assets**:
     + Contains static assets like images, fonts, and other resources used by the application.
   * **components**:
     + Contains reusable UI components such as buttons, form elements, headers, footers, etc.
   * **errors**:
     + Contains components or utilities related to error handling, such as custom error pages or error boundaries.
   * **models**:
     + Contains TypeScript interfaces or models that define the shape of data used in the application.
   * **network**:
     + Includes network-related utilities, such as API service files for making HTTP requests.
   * **pages**:
     + Contains components that represent full pages or views in the application. These are used in routing to render different parts of the application based on the URL.
   * **styles**:
     + Contains global styles, CSS files, or style modules that are used throughout the application.
   * **utils**:
     + Contains utility functions and helper methods that are used across the application.
3. **src Files**:
   * **App.test.tsx**:
     + Contains tests for the App.tsx component. Testing files typically end with .test.tsx.
   * **App.tsx**:
     + The main App component which acts as the root component for the application. It contains the main structure and routing logic.
   * **index.css**:
     + Contains global CSS styles that are applied to the entire application.
   * **index.tsx**:
     + The entry point of the React application. It renders the App component into the DOM.
   * **logo.svg**:
     + An SVG file for the application's logo.
   * **react-app-env.d.ts**:
     + Provides TypeScript types for the environment variables used in the application.
   * **reportWebVitals.ts**:
     + Measures and reports web vitals, such as performance metrics of the application.
   * **setupTests.ts**:
     + Configures the testing environment, such as initializing testing libraries.
4. **Configuration and Dependency Files**:
   * **package-lock.json**:
     + Locks the dependency versions installed in node\_modules to ensure consistency across different environments.
   * **package.json**:
     + Lists the project's dependencies, scripts, and metadata.
   * **README.md**:
     + Contains documentation and information about the project.
   * **tsconfig.json**:
     + Configuration file for TypeScript, specifying compiler options and project settings.
   * **yarn.lock**:
     + Similar to package-lock.json, but used by Yarn package manager to lock dependency versions.

### **How They Are Connected**

* **App.tsx** is the root component, which is rendered by **index.tsx**. The **index.tsx** file bootstraps the React application and mounts the App component to a DOM element (usually with an ID of root).
* Components from the **components** folder are imported and used within App.tsx or other components to build the UI.
* Pages from the **pages** folder are used in conjunction with a routing library (e.g., React Router) to display different views based on the URL.
* The **styles** folder provides global styles that are imported into index.tsx or specific components to apply consistent styling across the application.
* The **network** folder contains functions for making API calls, which are imported and used in components or services that require data fetching.
* Utility functions from the **utils** folder are imported wherever needed to provide reusable functionality.
* Models from the **models** folder are used to define the shape of the data, ensuring type safety across the application.

Each of these folders and files plays a specific role in organising the codebase, making it modular, maintainable, and scalable.