

ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
UID LAB



Báo cáo nghiên cứu
IOT WEBSERVER

NGƯỜI THỰC HIỆN: VÕ HỮU DƯ

TP.HCM, Ngày 12 tháng 5 năm 2025

Mục lục

1	CẤU TRÚC TỔNG QUAN	4
1.1	Chức năng tổng quan	4
1.2	Sơ đồ khối	5
2	FRONT END	6
2.1	Giới thiệu	6
2.2	Cài đặt	6
2.3	Bố cục giao diện	6
2.4	Cấu trúc thư mục	7
2.5	Cài đặt thư viện	7
2.6	COMPONENTS	8
2.6.1	Header	8
2.6.2	LoginDialog	13
2.6.3	AvatarMenu	15
2.6.4	ChangePasswordDialog	21
2.6.5	Sidebar	24
2.7	PAGES	28
2.7.1	Status	28
2.7.2	Device	33
2.7.3	Setting	34
2.8	CONTEXTS	36
2.8.1	SnackbarContext	36
2.9	HOOKS	39
2.9.1	useAuth	39
2.10	API	42
2.10.1	loginUser	42
2.10.2	Import	42
2.11	APP	44
2.11.1	Import	44
2.11.2	API URL	45
2.11.3	Component App	45
2.11.4	Kiểm Tra Xác Thực	45
2.11.5	Cấu Trúc JSX	46
2.11.6	Export	47
2.11.7	Chức Năng Chính	48

3	BACK END	49
3.1	Cấu trúc hệ thống	49
3.1.1	Mô hình Client - Server	49
3.1.2	Kiến trúc RESTful API	49
3.1.3	Mô hình MVC	50
3.1.4	Kiến trúc Layered	51
3.2	Cấu trúc thư mục	51
3.3	Cài đặt	52
4	MAIN SERVER	53
4.1	CONTROLLERS	53
4.1.1	deviceController	53
4.1.2	Import	55
4.2	ROUTES	62
4.2.1	deviceRoute	62
4.2.2	userRoute	63
4.3	MIDDLEWARE	65
4.3.1	auth	65
4.3.2	upload	67
4.4	SERVER	69
4.4.1	Import	69
4.4.2	Biên Đường Dẫn	70
4.4.3	Cấu Hình Môi Trường	70
4.4.4	Khởi Tạo Server	70
4.4.5	Cấu Hình Socket.IO	70
4.4.6	Middleware	71
4.4.7	Định Tuyến	72
4.4.8	Tuyến Góc	72
4.4.9	Xử Lý Socket.IO	72
4.4.10	Khởi Động Server	73
4.4.11	Chức Năng Chính	73
5	DATABASE SERVER	74
5.1	CONFIG	74
5.1.1	db	74
5.2	MODEL	75
5.2.1	Device	75
5.2.2	User	77
5.3	SEED	78
5.3.1	userSeed	78
5.4	SERVER	80
5.4.1	Import	80
5.4.2	Biên Đường Dẫn	81
5.4.3	Cấu Hình Môi Trường	81
5.4.4	Kết Nối MongoDB	81
5.4.5	Khởi Tạo Server	81
5.4.6	API Lấy Danh Sách Thiết Bị	82
5.4.7	API Tìm Người Dùng Theo Email	82
5.4.8	API Tìm Người Dùng Theo ID	82



5.4.9	API Tạo Người Dùng Mới	83
5.4.10	API Cập Nhật Người Dùng	83
5.4.11	Khởi Động Server	84
5.4.12	Chức Năng Chính	84
Phụ lục		85
A SOURCE CODE		86

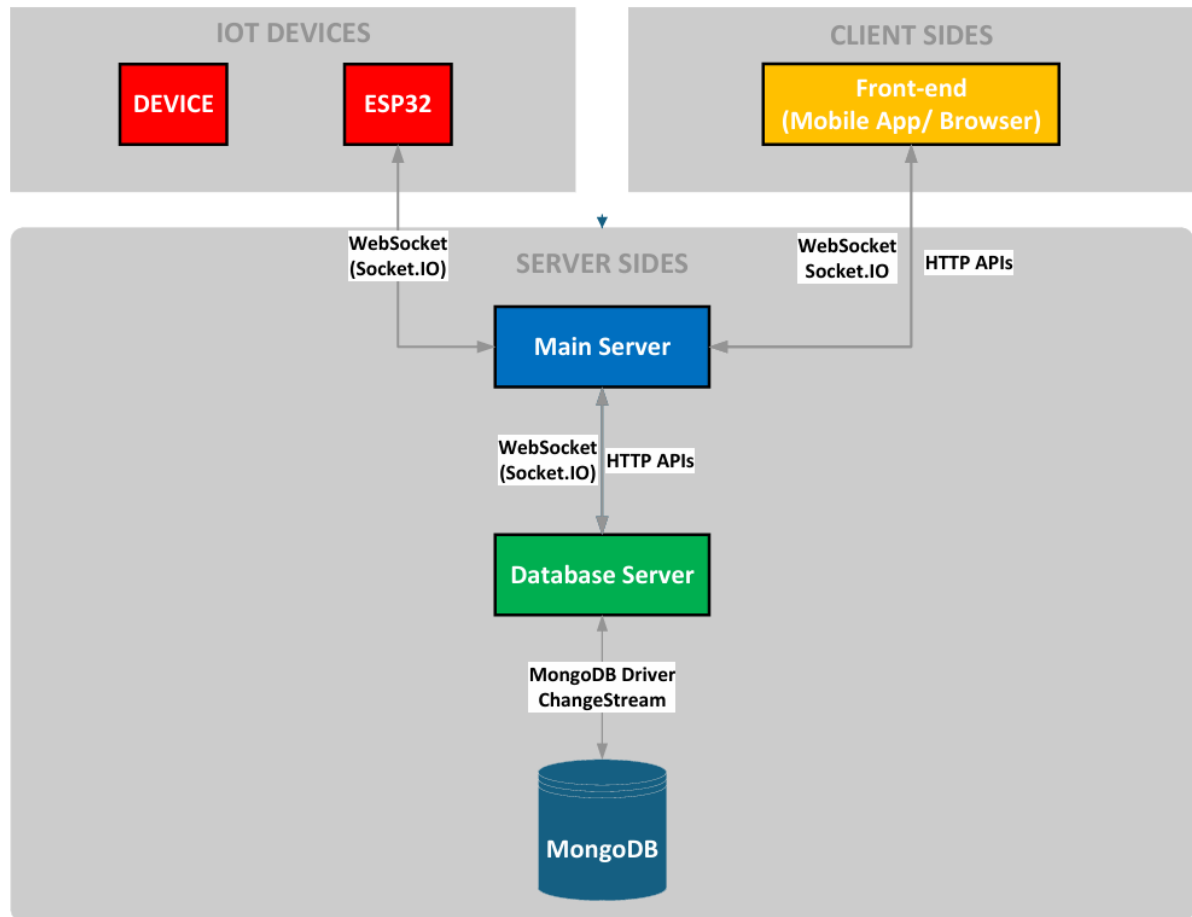
Chương 1

CẤU TRÚC TỔNG QUAN

1.1 Chức năng tổng quan

- Tạo mô hệ thống WebServer giám sát và điều khiển thiết bị IOT từ xa. Cho phép người dùng đăng nhập vào hệ thống và thực hiện các thao tác điều khiển thiết bị IOT.
- Lưu trữ dữ liệu từ thiết bị IOT lên WebServer. Tạo giao diện tương tác với người dùng.
- Đóng gói WebServer thành một ứng dụng có thể chạy trên nhiều nền tảng khác nhau.

1.2 Sơ đồ khối



Hình 1.1: Sơ đồ khối WebServer

- **ESP32**: thiết bị trung gian, nhận dữ liệu từ thiết bị IOT và gửi dữ liệu lên WebServer.
- **Front-end**: gửi POST/GET request đến WebServer để hiển thị giao diện tương tác với người dùng.
- **MainServer**: là máy chủ chính của hệ thống, nhận dữ liệu từ ESP32, xác thực dữ liệu và chuyển tiếp dữ liệu đến DatabaseServer. Nhận các yêu cầu từ Front-end và trả về dữ liệu cho Front-end.
- **DatabaseServer**: là máy chủ cơ sở dữ liệu, lưu trữ dữ liệu của hệ thống thông qua Database. Nhận yêu cầu từ MainServer và trả về dữ liệu cho MainServer.
- **Database**: là nơi lưu trữ dữ liệu của hệ thống, cho phép truy vấn và lưu dữ liệu từ DatabaseServer.

Chương 2

FRONT END

2.1 Giới thiệu

- Front-end là phần giao diện người dùng của hệ thống, cho phép người dùng tương tác với hệ thống thông qua các thao tác trên giao diện.
- Front-end được xây dựng bằng Vite, ReactJS, Material UI, Axios và sử dụng Web-Socket để nhận dữ liệu từ WebServer.
- Front-end sẽ gửi các yêu cầu đến WebServer để lấy dữ liệu và hiển thị lên giao diện.

2.2 Cài đặt

- Cài đặt NodeJS và NPM trên máy tính của bạn. Bạn có thể tải NodeJS tại địa chỉ: <https://nodejs.org/en/download/>

- Cài đặt Vite bằng lệnh sau:

```
1 npm create vite@latest
```

Chọn Framework là React và variant là Javascript.

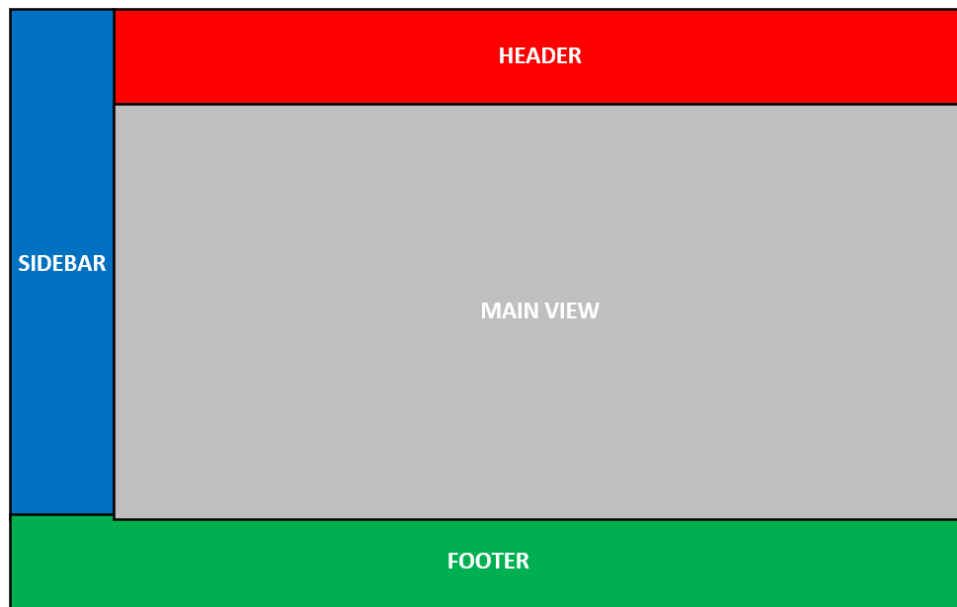
- Chạy ứng dụng bằng lệnh sau:

```
1 npm run dev
```

- Mở trình duyệt và truy cập vào địa chỉ: <http://localhost:5173/>

2.3 Bố cục giao diện

Giao diện có dạng dashboard được phân bố như hình:



Hình 2.1: Giao diện dashboard

2.4 Cấu trúc thư mục

- **node_modules**: Thư mục chứa các thư viện được cài đặt bằng NPM.
- **public**: Thư mục chứa các tệp tĩnh như hình ảnh, biểu tượng, v.v.
- **src**: Thư mục chứa mã nguồn của ứng dụng.
 - **api**: Thư mục chứa các tệp API của ứng dụng.
 - **assets**: Thư mục chứa các tệp tài nguyên như hình ảnh, biểu tượng, v.v.
 - **components**: Thư mục chứa các thành phần giao diện của ứng dụng: Header.jsx, Footer.jsx, Sidebar.jsx.
 - **hooks**: Thư mục chứa các hook tùy chỉnh của ứng dụng, hiển thị ở phần Main view.
 - **pages**: Thư mục chứa các trang của ứng dụng.
 - **router**: Thư mục chứa các tệp định tuyến của ứng dụng.
 - **services**: Thư mục chứa các tệp dịch vụ của ứng dụng.
 - **styles**: Thư mục chứa các tệp CSS của ứng dụng.
 - **App.jsx**: Tệp chính của ứng dụng.
 - **main.jsx**: Tệp khởi động ứng dụng.
 - **theme.js**: Tệp chứa định dạng nền cho ứng dụng.

2.5 Cài đặt thư viện

- Thư viện Material UI: Thư viện giao diện người dùng cho React.

```
1 npm install @mui/material
```


- Thư viện Axios: Thư viện gửi yêu cầu HTTP.

```
1 npm install axios
```

- Thư viện React Router: Thư viện định tuyến cho React.

```
1 npm install react-router-dom
```

- Thư viện Socket Clint: Thư viện WebSocket cho Front

```
1 npm install socket.io-client
```

- Các thư viện khác cài đặt trong quá trình phát triển.

2.6 COMPONENTS

2.6.1 Header

Header là thành phần hiển thị tiêu đề của ứng dụng, thanh tìm kiếm, thông tin người dùng.



Hình 2.2: Giao diện Header trước khi Login



Hình 2.3: Giao diện Header sau khi Login

Kết nối Socket

Socket.IO được sử dụng để giao tiếp thời gian thực với server:

```
1 const API_URL = import.meta.env.VITE_API_URL ||
2   "http://localhost:5000";
3 const socket = io(API_URL, {
4   reconnection: true,
5   reconnectionAttempts: 5,
6   reconnectionDelay: 1000,
7   transports: ["websocket", "polling"],
8   auth: { token: document.cookie.split("; ").find(row =>
9     row.startsWith("authToken="))?.split("=")[1] || null }
10 });
```

- API_URL: Lấy từ biến môi trường hoặc mặc định là `http://localhost:5000`.
- socket: Kết nối với server, ưu tiên WebSocket, dùng polling nếu thất bại. Có cơ chế tự động thử lại 5 lần, cách nhau 1 giây. Token xác thực được lấy từ cookie.

Thành Phần Giao Diện Tùy Chỉnh

Thanh tìm kiếm được định kiểu bằng styled:

```

1  const Search = styled("div")(({ theme }) => ({
2    position: "relative",
3    borderRadius: theme.shape.borderRadius,
4    backgroundColor: alpha(theme.palette.text.primary, 0.05),
5    "&:hover": {
6      backgroundColor: alpha(theme.palette.text.primary, 0.1),
7    },
8    marginLeft: theme.spacing(2),
9    width: "100%",
10   maxWidth: 300,
11  }));
12  const SearchIconWrapper = styled("div")(({ theme }) => ({
13    padding: theme.spacing(0, 2),
14    height: "100%",
15    position: "absolute",
16    display: "flex",
17    alignItems: "center",
18    justifyContent: "center",
19    color: theme.palette.text.secondary,
20  }));
21  const StyledInputBase = styled(InputBase)(({ theme }) => ({
22    color: theme.palette.text.primary,
23    paddingLeft: `calc(1em + ${theme.spacing(4)})`,
24    width: "100%",
25  }));

```

- Search: Container thanh tìm kiếm, nền mờ (5% độ mờ), hover đổi thành 10%, rộng tối đa 300px.
- SearchIconWrapper: Định vị icon tìm kiếm bên trái, căn giữa.
- StyledInputBase: Ô nhập liệu có màu chữ theo theme, thêm padding để tránh che icon.

Component Header

Component nhận các props:

```

1  const Header = ({ onToggleSidebar, user, setUser }) => {}

```

- onToggleSidebar: Hàm mở/đóng sidebar.
- user: Thông tin người dùng (null nếu chưa đăng nhập).
- setUser: Cập nhật trạng thái người dùng.

Trạng Thái và Hook

```
1  const theme = useTheme();
2  const [anchorEl, setAnchorEl] = useState(null);
3  const { email, setEmail, password, setPassword, openLogin,
    setOpenLogin, handleLogin, handleLogout } =
    useAuth(setUser, socket);
```

- theme: Lấy theme để tạo kiểu.
- anchorEl: Lưu vị trí mở AvatarMenu.
- useAuth: Cung cấp trạng thái (email, password, openLogin) và hàm xử lý đăng nhập/dăng xuất.

Xử Lý Sự Kiện

```
1  const handleAvatarClick = (event) => {
2      setAnchorEl(event.currentTarget);
3  };
4  const handleMenuClose = () => {
5      setAnchorEl(null);
6  };
```

- handleAvatarClick: Mở AvatarMenu khi click avatar.
- handleMenuClose: Đóng menu.

URL Ảnh Đại Diện

```
1  const avatarSrc = user?.avatar ? `${API_URL}${user.avatar}` :
    undefined;
2  console.log("Avatar URL in Header:", avatarSrc);
```

Tạo URL ảnh đại diện bằng cách nối API_URL với đường dẫn avatar, in ra để debug.

Cấu Trúc JSX

Thanh điều hướng chính:

```
1  <AppBar
2      position="fixed"
3      sx={{
4          backgroundColor: theme.palette.background.header,
5          color: theme.palette.text.primary,
6          zIndex: theme.zIndex.drawer + 1,
7          boxShadow: "none",
8      }}
9  >
```

- AppBar: Thanh cố định, màu nền và chữ theo theme, z-index cao, không có bóng.

Phần trái của Toolbar:

```

1      <Toolbar sx={{ display: "flex", justifyContent:
2          "space-between", pl: "0px" }}>
3          <Box sx={{ display: "flex", alignItems: "center" }}>
4              <IconButton color="inherit"
5                  onClick={onToggleSidebar} sx={{ mr: 2, p: 0 }}>
6                  <MenuIcon />
7              </button>
8              
10             <Typography variant="h6" fontWeight="bold">
11                 UID LAB
12             </Typography>
13         </Box>

```

- Nút MenuIcon mở/đóng sidebar.
- Logo (60x60px) và tiêu đề "UID LAB" kiểu h6, in đậm.

Phần phải (thanh tìm kiếm và xác thực):

```

1      <Box sx={{ display: "flex", alignItems: "center",
2          gap: 2 }}>
3          <Search>
4              <SearchIconWrapper>
5                  <SearchIcon />
6              </SearchIconWrapper>
7              <StyledInputBase
8                  placeholder="Search..."
9                  inputProps={{ "aria-label": "search" }}
10             />
11          </Search>
12          {!user ? (
13              <Typography
14                  color="inherit"
15                  onClick={() => setOpenLogin(true)}
16                  sx={{ cursor: "pointer" }}
17              >
18                  Dang Nhap
19              </Typography>
20          ) : (
21              <Box sx={{ display: "flex", alignItems:
22                  "center", gap: 1 }}>
23                  <Avatar
24                      src={avatarSrc}
25                      alt={user.username}
26                      onClick={handleAvatarClick}
27                      sx={{ cursor: "pointer", width: 40,
28                          height: 40 }}

```

```

26         />
27         <AvatarMenu
28             anchorEl={anchorEl}
29             onClose={handleMenuClose}
30             user={user}
31             setUser={setUser}
32             onLogout={handleLogout}
33         />
34     </Box>
35     )}
36 </Box>

```

- Thanh tìm kiếm: Ô nhập liệu với icon và placeholder.
- Xác thực: Hiện thị link "Đăng nhập" nếu chưa đăng nhập, hoặc avatar và AvatarMenu nếu đã đăng nhập.

Modal đăng nhập:

```

1         <LoginDialog
2             open={openLogin}
3             onClose={() => setOpenLogin(false)}
4             email={email}
5             setEmail={setEmail}
6             password={password}
7             setPassword={setPassword}
8             handleLogin={handleLogin}
9         />

```

- LoginDialog: Modal đăng nhập, hiển thị khi openLogin là true.

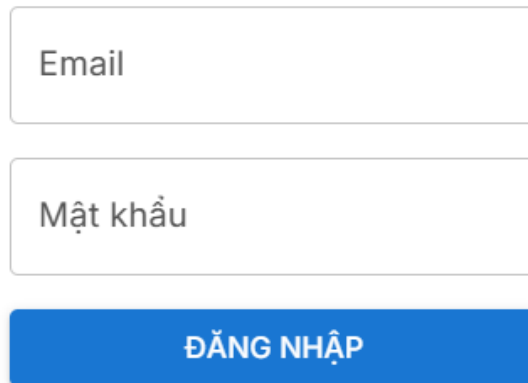
Chức năng chính

- **Mở/đóng Sidebar:** Click MenuItem gọi onToggleSidebar.
- **Thanh Tìm Kiếm:** Ô nhập liệu với icon và kiểu dáng tùy chỉnh.
- **Xác Thực Người Dùng:**
 - Chưa đăng nhập: Link "Đăng nhập" mở LoginDialog.
 - Đã đăng nhập: Avatar mở AvatarMenu với tùy chọn hồ sơ/dăng xuất.
- **Giao Tiếp Thời Gian Thực:** Socket.IO duy trì kết nối với server, hỗ trợ xác thực qua token.

2.6.2 LoginDialog

Là một hộp thoại (modal), sử dụng Material-UI để tạo giao diện đăng nhập người dùng. Nó hiển thị một biểu mẫu với các trường nhập email, mật khẩu và nút đăng nhập.

Đăng nhập



The image shows a login dialog box with a white background and rounded corners. It contains two text input fields: the top one is labeled 'Email' and the bottom one is labeled 'Mật khẩu' (Password). Below these fields is a blue button with the text 'ĐĂNG NHẬP' (Login) in white capital letters.

Hình 2.4: Giao diện LoginDialog

Import

Các thư viện và thành phần Material-UI được nhập để xây dựng giao diện:

```
1 import React from 'react';  
2 import { Dialog, Box, Typography, TextField, Button } from  
  '@mui/material';
```

- React: Thư viện chính để xây dựng component.
- Material-UI: Cung cấp các thành phần:
 - Dialog: Hộp thoại hiển thị biểu mẫu đăng nhập.
 - Box: Container để sắp xếp bố cục.
 - Typography: Hiển thị tiêu đề.
 - TextField: Trường nhập liệu cho email và mật khẩu.
 - Button: Nút thực hiện hành động đăng nhập.

Component LoginDialog

Component nhận các props để quản lý trạng thái và xử lý đăng nhập:

```
1 const LoginDialog = ({ open, onClose, email, setEmail,  
  password, setPassword, handleLogin }) => {
```

- open: Trạng thái hiển thị của hộp thoại (true/false).

- onClose: Hàm đóng hộp thoại.
- email, setEmail: Giá trị và hàm cập nhật email.
- password, setPassword: Giá trị và hàm cập nhật mật khẩu.
- handleLogin: Hàm xử lý logic đăng nhập.

Cấu Trúc JSX

Giao diện của LoginDialog được định nghĩa trong JSX:

```

1  return (
2      <Dialog open={open} onClose={onClose}>
3          <Box sx={{ p: 3, display: 'flex', flexDirection:
4              'column', gap: 2, width: 300 }}>
5              <Typography variant="h6" fontWeight="bold">Dang
6                  nhap</Typography>
7              <TextField
8                  label="Email"
9                  value={email}
10                 onChange={(e) => setEmail(e.target.value)}
11                 fullWidth
12             />
13             <TextField
14                 label="Mat khau"
15                 type="password"
16                 value={password}
17                 onChange={(e) => setPassword(e.target.value)}
18                 fullWidth
19             />
20             <Button variant="contained" onClick={handleLogin}>
21                 Dang nhap
22             </Button>
23         </Box>
24     </Dialog>
25 );

```

- Dialog: Hộp thoại được điều khiển bởi open và onClose.
- Box: Container với:
 - Padding 3 đơn vị (p: 3).
 - Bố cục cột dọc (flexDirection: 'column').
 - Khoảng cách giữa các phần tử (gap: 2).
 - Chiều rộng cố định 300px (width: 300).
- Typography: Tiêu đề "Đăng nhập" với kiểu h6, in đậm.
- TextField (Email): Trường nhập liệu email, liên kết với email và setEmail, chiếm toàn bộ chiều rộng (fullWidth).

- **TextField (Mật khẩu):** Trường nhập liệu mật khẩu, loại `password` để ẩn ký tự, liên kết với `password` và `setPassword`.
- **Button:** Nút "Đăng nhập" với kiểu `contained` (nút nổi), gọi `handleLogin` khi click.

Export

Component được xuất để sử dụng trong các phần khác của ứng dụng:

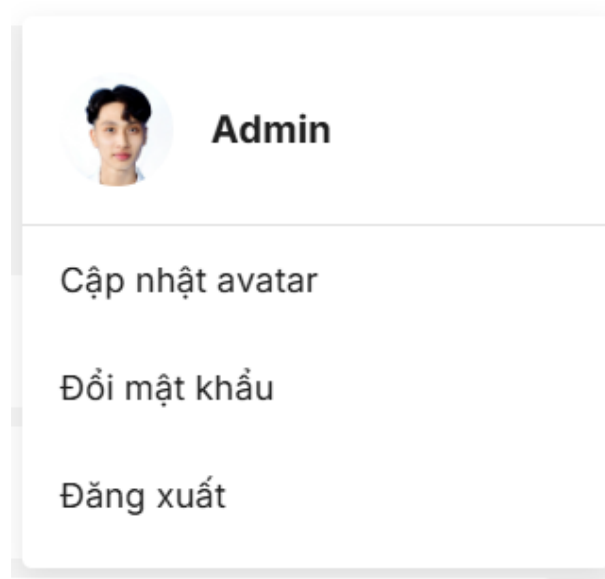
```
1 export default LoginDialog;
```

Chức Năng Chính

- **Hiện thị Hộp Thoại Đăng Nhập:** Hộp thoại xuất hiện khi `open` là `true`, đóng khi gọi `onClose`.
- **Nhập Thông Tin Đăng Nhập:** Người dùng nhập email và mật khẩu vào các trường `TextField`, dữ liệu được cập nhật qua `setEmail` và `setPassword`.
- **Xử Lý Đăng Nhập:** Nút "Đăng nhập" gọi `handleLogin` để thực hiện xác thực (logic cụ thể phụ thuộc vào hàm này).
- **Giao Diện:** Sử dụng `Material-UI` để tạo bố cục rõ ràng, trực quan, với các trường nhập liệu và nút được sắp xếp gọn gàng.

2.6.3 AvatarMenu

Là một menu sử dụng `Material-UI` để hiển thị các tùy chọn cho người dùng đã đăng nhập, bao gồm cập nhật avatar, đổi mật khẩu và đăng xuất. Nó tích hợp với API thông qua `axios` và sử dụng `SnackbarContext` để hiển thị thông báo.



Hình 2.5: Giao diện AvatarMenu

Import

Các thư viện và thành phần được nhập để xây dựng AvatarMenu:

```
1 import React, { useState } from "react";
2 import {
3     Avatar, Menu, MenuItem, Divider, Box, Typography,
4     useTheme,
5 } from "@mui/material";
6 import axios from "axios";
7 import ChangePasswordDialog from "../ChangePasswordDialog";
8 import { useSnackbar } from '../context/SnackbarContext';
```

- React, useState: Quản lý trạng thái cục bộ.
- Material-UI: Cung cấp các thành phần:
 - Avatar: Hiển thị ảnh đại diện.
 - Menu, MenuItem: Tạo menu ngữ cảnh.
 - Divider: Đường phân cách.
 - Box, Typography: Sắp xếp bố cục và hiển thị văn bản.
 - useTheme: Lấy theme ứng dụng.
- axios: Gửi yêu cầu HTTP tới API.
- ChangePasswordDialog: Component để đổi mật khẩu.
- useSnackbar: Hook từ SnackbarContext để hiển thị thông báo.

API URL

Định nghĩa URL API:

```
1 const API_URL = import.meta.env.VITE_API_URL ||
  "http://localhost:5000";
```

- API_URL: Lấy từ biến môi trường hoặc mặc định là `http://localhost:5000`.

Component AvatarMenu

Component nhận các props:

```
1 const AvatarMenu = ({ anchorEl, onClose, user, setUser,
  onLogout }) => {
```

- anchorEl: Vị trí neo menu.
- onClose: Hàm đóng menu.
- user: Thông tin người dùng (username, avatar).
- setUser: Cập nhật trạng thái người dùng.
- onLogout: Hàm xử lý đăng xuất.

Trạng Thái

Quản lý trạng thái cục bộ:

```
1  const theme = useTheme();
2  const [openChangePassword, setOpenChangePassword] =
    useState(false);
3  const [oldPassword, setOldPassword] = useState("");
4  const [newPassword, setNewPassword] = useState("");
5  const { showSnackbar } = useSnackbar();
```

- theme: Lấy theme để tạo kiểu.
- openChangePassword: Điều khiển hiển thị ChangePasswordDialog.
- oldPassword, newPassword: Lưu mật khẩu cũ và mới.
- showSnackbar: Hàm hiển thị thông báo từ SnackbarContext.

Xử Lý Cập Nhật Avatar

Hàm xử lý tải lên avatar:

```
1  const handleAvatarChange = async (event) => {
2      const file = event.target.files[0];
3      if (file) {
4          const formData = new FormData();
5          formData.append("avatar", file);
6          try {
7              const res = await axios.post(
8                  `${import.meta.env.VITE_API_URL}/api/users
9                  /update-avatar`,
10                 formData,
11                 { withCredentials: true }
12             );
13             const updatedAvatar = res.data.avatar;
14             setUser((prev) => ({ ...prev, avatar:
15                 updatedAvatar }));
16             localStorage.setItem("user", JSON.stringify({
17                 username: user.username, avatar: updatedAvatar
18             }));
19             showSnackbar("Update avatar successful",
20                 "success");
21             onClose();
22         } catch (error) {
23             console.error("Error when updating avatar:",
24                 error);
25             showSnackbar("Error when updating avatar",
26                 "error");
27         }
28     }
29 }
```

- Lấy tệp ảnh từ `event.target.files`.
- Tạo `FormData` để gửi tệp lên API `/api/users/update-avatar`.
- Cập nhật trạng thái `user` và `localStorage` nếu thành công.
- Hiển thị thông báo bằng `showSnackBar`.
- Xử lý lỗi với thông báo lỗi.

Xử Lý Đổi Mật Khẩu

Hàm xử lý đổi mật khẩu:

```
1  const handleChangePassword = async () => {
2    try {
3      const res = await axios.post(
4        `${import.meta.env.VITE_API_URL}/api/users
5        /change-password`,
6        { oldPassword, newPassword },
7        { withCredentials: true }
8      );
9      showSnackBar(res.data.message, "success");
10     setOpenChangePassword(false);
11     setOldPassword("");
12     setNewPassword("");
13   } catch (error) {
14     console.error("Error when changing password:", error);
15     showSnackBar(error.response?.data?.message || "Error
16       when changing password", "error");
17   }
18 }
```

- Gửi yêu cầu POST tới `/api/users/change-password` với `oldPassword` và `newPassword`.
- Hiển thị thông báo thành công và reset trạng thái nếu thành công.
- Hiển thị thông báo lỗi nếu thất bại.

Xử Lý Đăng Xuất

Hàm xử lý đăng xuất:

```
1  const handleLogoutClick = () => {
2    onLogout();
3    onClose();
4  }
```

- Gọi `onLogout` và đóng menu.

URL Ảnh Đại Diện

Tạo URL cho avatar:

```
1   const avatarSrc = user.avatar ? `${API_URL}${user.avatar}` :  
    undefined;  
2   console.log("Avatar URL in AvatarMenu:", avatarSrc);
```

- Nối API_URL với đường dẫn avatar, in ra để debug.

Cấu Trúc JSX

Giao diện menu:

```
1   <Menu  
2     anchorEl={anchorEl}  
3     open={Boolean(anchorEl)}  
4     onClose={onClose}  
5     anchorOrigin={{ vertical: "bottom", horizontal: "right" }}  
6     transformOrigin={{ vertical: "top", horizontal: "right" }}  
7     PaperProps={{  
8       sx: {  
9         mt: 1,  
10        width: 250,  
11        bgcolor: theme.palette.background.paper,  
12        color: theme.palette.text.primary,  
13        boxShadow: "0px 4px 12px rgba(0, 0, 0, 0.1)",  
14      },  
15    }}  
16   >  
17     <Box sx={{ display: "flex", alignItems: "center", p: 2 }}>  
18       <Avatar  
19         src={avatarSrc}  
20         alt={user.username}  
21         sx={{ mr: 2, width: 48, height: 48 }}  
22       />  
23       <Typography variant="subtitle1" fontWeight="bold">  
24         {user.username}  
25       </Typography>  
26     </Box>  
27     <Divider />  
28     <MenuItem  
29       component="label"  
30       sx={{ py: 1.5, fontSize: "0.9rem" }}  
31     >  
32       Update avatar  
33       <input  
34         type="file"  
35         accept="image/*"  
36         hidden  
37         onChange={handleAvatarChange}  
38       />
```

```

39     </MenuItem>
40     <MenuItem
41         onClick={() => setOpenChangePassword(true)}
42         sx={{ py: 1.5, fontSize: "0.9rem" }}
43     >
44         Changing Password
45     </MenuItem>
46     <MenuItem
47         onClick={handleLogoutClick}
48         sx={{ py: 1.5, fontSize: "0.9rem" }}
49     >
50         Logout
51     </MenuItem>
52 </Menu>

```

- Menu: Menu ngữ cảnh, neo tại `anchorEl`, hiển thị khi `anchorEl` tồn tại.
- Box: Hiển thị avatar và tên người dùng, căn chỉnh ngang.
- Divider: Đường phân cách.
- MenuItem:
 - "Cập nhật avatar": Chứa input file ẩn để tải ảnh.
 - "Đổi mật khẩu": Mở `ChangePasswordDialog`.
 - "Đăng xuất": Gọi `handleLogoutClick`.

Hộp thoại đổi mật khẩu:

```

1     <ChangePasswordDialog
2         open={openChangePassword}
3         onClose={() => {
4             setOpenChangePassword(false);
5             setOldPassword("");
6             setNewPassword("");
7         }}
8         oldPassword={oldPassword}
9         setOldPassword={setOldPassword}
10        newPassword={newPassword}
11        setNewPassword={setNewPassword}
12        handleChangePassword={handleChangePassword}
13    />

```

- `ChangePasswordDialog`: Hộp thoại hiển thị khi `openChangePassword` là `true`.

Export

```

1     export default AvatarMenu;

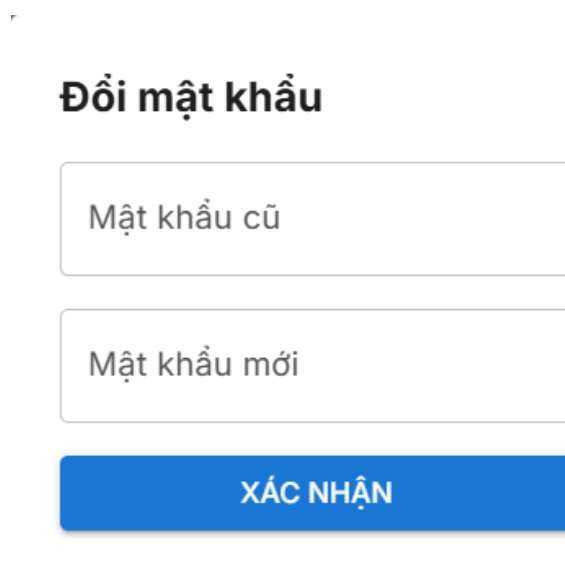
```

Chức Năng Chính

- **Hiển Thị Menu Ngữ Cảnh:** Menu xuất hiện khi click avatar, hiển thị tên người dùng và các tùy chọn.
- **Cập Nhật Avatar:** Cho phép tải lên ảnh mới, gửi tới API và cập nhật trạng thái người dùng.
- **Đổi Mật Khẩu:** Mở hộp thoại để nhập mật khẩu cũ/mới, gửi yêu cầu đổi mật khẩu tới API.
- **Đăng Xuất:** Xử lý đăng xuất và đóng menu.
- **Thông Báo**:** Sử dụng `showSnackBar` để hiển thị thông báo thành công/lỗi.

2.6.4 ChangePasswordDialog

Là một hộp thoại (modal) sử dụng Material-UI để hiển thị biểu mẫu đổi mật khẩu cho người dùng. Nó cho phép người dùng nhập mật khẩu cũ và mật khẩu mới, và thực hiện xác thực khi nhấn nút "Đổi mật khẩu".



Hình 2.6: Giao diện ChangePasswordDialog

Import

Các thư viện và thành phần Material-UI được nhập để xây dựng giao diện:

```
1 import React from "react";  
2 import { Dialog, Box, Typography, TextField, Button } from  
  "@mui/material";
```

- **React:** Thư viện chính để xây dựng component.
- **Material-UI:** Cung cấp các thành phần:
 - **Dialog:** Hộp thoại hiển thị biểu mẫu đổi mật khẩu.

- Box: Container để sắp xếp bố cục.
- Typography: Hiển thị tiêu đề.
- TextField: Trường nhập liệu cho mật khẩu cũ và mới.
- Button: Nút xác nhận hành động đổi mật khẩu.

Component ChangePasswordDialog

Component nhận các props để quản lý trạng thái và xử lý đổi mật khẩu:

```
1  const ChangePasswordDialog = ({
2    open,
3    onClose,
4    oldPassword,
5    setOldPassword,
6    newPassword,
7    setNewPassword,
8    handleChangePassword,
9  }) => {}
```

- open: Trạng thái hiển thị của hộp thoại (true/false).
- onClose: Hàm đóng hộp thoại.
- oldPassword, setOldPassword: Giá trị và hàm cập nhật mật khẩu cũ.
- newPassword, setNewPassword: Giá trị và hàm cập nhật mật khẩu mới.
- handleChangePassword: Hàm xử lý logic đổi mật khẩu.

Cấu Trúc JSX

Giao diện của ChangePasswordDialog được định nghĩa trong JSX:

```
1  return (
2    <Dialog open={open} onClose={onClose}>
3      <Box sx={{ p: 3, display: "flex", flexDirection:
4        "column", gap: 2, width: 300 }}>
5        <Typography variant="h6" fontWeight="bold">Change
6          Password</Typography>
7        <TextField
8          label="Old Password"
9          type="password"
10         value={oldPassword}
11         onChange={(e) =>
12           setOldPassword(e.target.value)}
13         fullWidth
14       />
15       <TextField
16         label="New Password"
17         type="password"
18         value={newPassword}
```

```
16         onChange={ (e) =>
17             setNewPassword(e.target.value)}
18         fullWidth
19     />
20     <Button variant="contained"
21         onClick={handleChangePassword}>
22         Validate
23     </Button>
24 </Box>
25 </Dialog>
26 );
```

- **Dialog:** Hộp thoại được điều khiển bởi `open` và `onClose`.
- **Box:** Container với:
 - Padding 3 đơn vị (`p: 3`).
 - Bố cục cột dọc (`flexDirection: "column"`).
 - Khoảng cách giữa các phần tử (`gap: 2`).
 - Chiều rộng cố định 300px (`width: 300`).
- **Typography:** Tiêu đề "Đổi mật khẩu" với kiểu `h6`, in đậm.
- **TextField (Mật khẩu cũ):** Trường nhập liệu mật khẩu cũ, loại `password` để ẩn ký tự, liên kết với `oldPassword` và `setOldPassword`, chiếm toàn bộ chiều rộng (`fullWidth`).
- **TextField (Mật khẩu mới):** Trường nhập liệu mật khẩu mới, tương tự mật khẩu cũ, liên kết với `newPassword` và `setNewPassword`.
- **Button:** Nút "Xác nhận" với kiểu `contained` (nút nổi), gọi `handleChangePassword` khi click.

Export

Component được xuất để sử dụng trong các phần khác của ứng dụng:

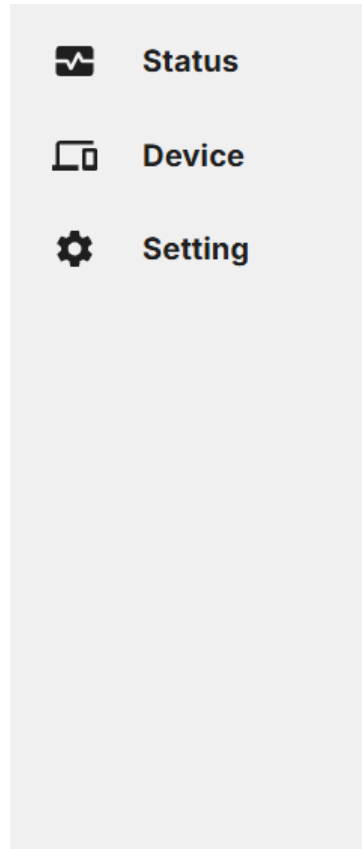
```
1 export default ChangePasswordDialog;
```

Chức Năng Chính

- **Hiển Thị Hộp Thoại Đổi Mật Khẩu:** Hộp thoại xuất hiện khi `open` là `true`, đóng khi gọi `onClose`.
- **Nhập Thông Tin Mật Khẩu:** Người dùng nhập mật khẩu cũ và mới vào các trường `TextField`, dữ liệu được cập nhật qua `setOldPassword` và `setNewPassword`.
- **Xử Lý Đổi Mật Khẩu:** Nút "Xác nhận" gọi `handleChangePassword` để thực hiện logic đổi mật khẩu (được định nghĩa ở component cha, ví dụ: `AvatarMenu`).
- **Giao Diện:** Sử dụng `Material-UI` để tạo bố cục rõ ràng, trực quan, với các trường nhập liệu và nút được sắp xếp gọn gàng.

2.6.5 Sidebar

Là một thanh điều hướng bên (sidebar) sử dụng Material-UI để hiển thị danh sách các mục điều hướng như trạng thái, thiết bị và cài đặt. Nó hỗ trợ trạng thái mở rộng/rút gọn, tích hợp với `react-router-dom` để chuyển trang, và tự động đánh dấu mục đang chọn dựa trên đường dẫn hiện tại.



Hình 2.7: Giao diện Sidebar

Import

Các thư viện và thành phần được nhập để xây dựng Sidebar:

```
1 import React from 'react';
2 import {
3   Drawer, List, ListItem, ListItemIcon, ListItemText,
4   Toolbar, Divider,
5 } from '@mui/material';
6 import StatusIcon from '@mui/icons-material/MonitorHeart';
7 import DevicesIcon from '@mui/icons-material/Devices';
8 import SettingIcon from '@mui/icons-material/Settings';
9 import { Link, useLocation } from 'react-router-dom';
```

- React: Thư viện chính để xây dựng component.
- Material-UI: Cung cấp các thành phần:
 - Drawer: Thanh điều hướng bên.

- `List`, `ListItem`, `ListItemIcon`, `ListItemText`: Tạo danh sách điều hướng.
- `Toolbar`: Khoảng trống trên cùng để căn chỉnh với `AppBar`.
- `Divider`: Đường phân cách.
- `StatusIcon`, `DevicesIcon`, `SettingIcon`: Biểu tượng cho các mục điều hướng.
- `Link`, `useLocation`: Từ `react-router-dom` để điều hướng và lấy đường dẫn hiện tại.

Biến và Cấu Hình

Định nghĩa chiều rộng và danh sách mục điều hướng:

```
1  const expandedWidth = 200;
2  const collapsedWidth = 75;
3
4  const menuItems = [
5    { label: 'Status', path: '/status/', icon: <StatusIcon /> },
6    { label: 'Device', path: '/device/', icon: <DevicesIcon /> },
7    { label: 'Setting', path: '/setting/', icon: <SettingIcon /> },
8  ];
```

- `expandedWidth`: Chiều rộng khi sidebar mở (200px).
- `collapsedWidth`: Chiều rộng khi sidebar rút gọn (75px).
- `menuItems`: Mảng chứa các mục điều hướng, mỗi mục có nhãn (`label`), đường dẫn (`path`), và biểu tượng (`icon`).

Component Sidebar

Component nhận prop để điều khiển trạng thái mở/rút gọn:

```
1  const Sidebar = ({ open }) => {
2    const location = useLocation();
```

- `open`: Trạng thái mở (`true`) hoặc rút gọn (`false`) của sidebar.
- `useLocation`: Hook lấy đường dẫn hiện tại để xác định mục được chọn.

Cấu Trúc JSX

Giao diện của `Sidebar` được định nghĩa trong JSX:

```
1  return (
2    <Drawer
3      variant="permanent"
4      sx={{
5        width: open ? expandedWidth : collapsedWidth,
```

```

6      flexShrink: 0,
7      '& .MuiDrawer-paper': {
8        width: open ? expandedWidth : collapsedWidth,
9        transition: 'width 0.3s',
10       overflowX: 'hidden',
11       boxSizing: 'border-box',
12       backgroundColor: (theme) =>
13         theme.palette.background.sidebar,
14       color: (theme) => theme.palette.text.primary,
15       borderRight: 'none',
16     },
17   }}
18 >
19   <Toolbar />
20   <Divider />
21   <List>
22     {menuItems.map((item) => {
23       const selected = location.pathname === item.path;
24
25       return (
26         <ListItem
27           key={item.path}
28           button
29           component={Link}
30           to={item.path}
31           selected={selected}
32           sx={{(theme) => ({
33             minHeight: 50,
34             px: 2,
35             py: 1,
36             justifyContent: open ? 'initial' :
37               'center',
38             color: theme.palette.text.primary,
39             '&:hover': {
40               backgroundColor:
41                 theme.palette.action.hover,
42             },
43             '&.Mui-selected': {
44               backgroundColor:
45                 theme.palette.action.selected,
46               color: theme.palette.text.primary,
47             },
48           })}
49         >
50         <ListItemIcon
51           sx={{(theme) => ({
52             minWidth: 40,
53             mr: open ? 2 : 'auto',
54             justifyContent: 'center',
55             color: theme.palette.text.primary,
56           })}

```

```

55         >
56             {item.icon}
57         </ListItemIcon>
58         {open && <ListItemText
59             primary={item.label} />}
60     </ListItem>
61 );
62     })}
63 </List>
64 </Drawer>
65 );

```

- **Drawer:** Thanh điều hướng cố định (`variant="permanent"`) với:
 - Chiều rộng động dựa trên `open` (`expandedWidth` hoặc `collapsedWidth`).
 - Hiệu ứng chuyển đổi mượt mà (`transition: 'width 0.3s'`).
 - Màu nền từ `theme.palette.background.sidebar`.
 - Không có viền phải (`borderRight: 'none'`).
- **Toolbar:** Khoảng trống trên cùng để căn chỉnh với `AppBar`.
- **Divider:** Đường phân cách giữa `Toolbar` và danh sách.
- **List:** Danh sách các mục điều hướng, lặp qua `menuItems` để tạo:
 - `ListItem`: Mỗi mục là một nút liên kết (`component=Link`) tới `item.path`.
 - `selected`: Đánh dấu mục đang chọn nếu `location.pathname` khớp `item.path`.
 - Tùy chỉnh giao diện: Căn giữa khi rút gọn, căn trái khi mở; hiệu ứng hover và chọn từ theme.
- **ListItemIcon:** Hiển thị biểu tượng, căn giữa khi rút gọn.
- **ListItemText:** Hiển thị nhãn khi `open` là `true`.

Export

Component được xuất để sử dụng trong ứng dụng:

```
1 export default Sidebar;
```

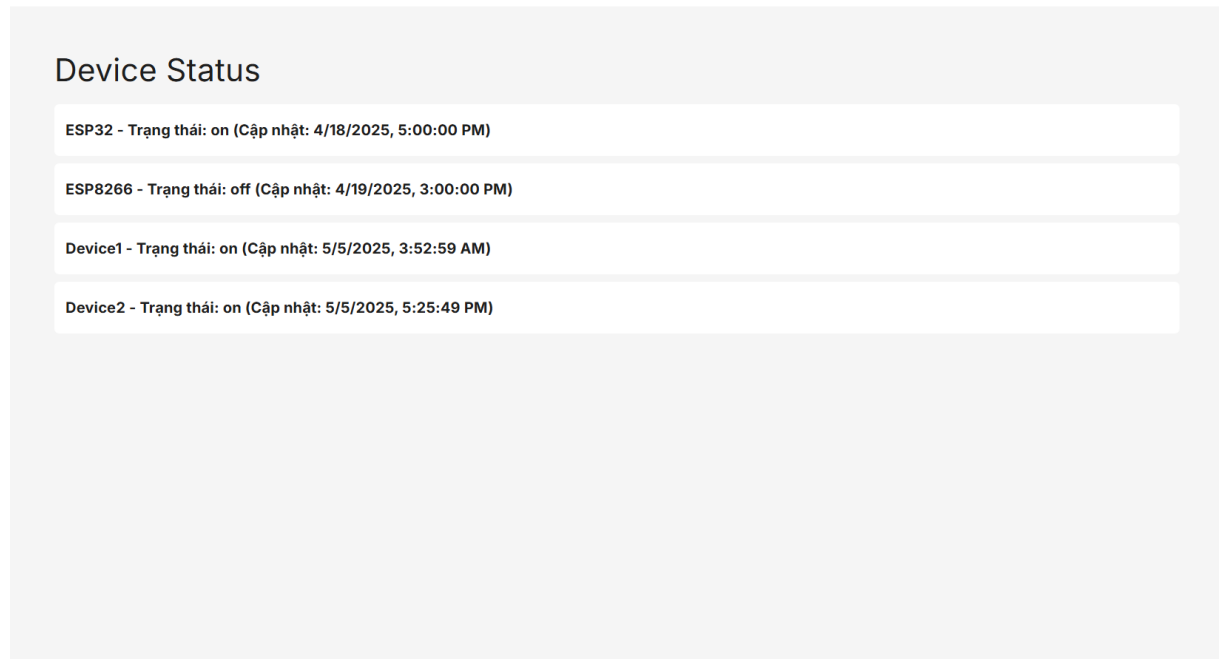
Chức Năng Chính

- **Điều Hướng Ứng Dụng:** Cung cấp các mục điều hướng ("Status", "Device", "Setting") với liên kết tới các trang tương ứng thông qua `react-router-dom`.
- **Trạng Thái Mở/Rút Gọn:** Hiển thị đầy đủ (với nhãn) hoặc rút gọn (chỉ biểu tượng) dựa trên prop `open`.
- **Đánh Dấu Mục Được Chọn:** Tự động làm nổi bật mục tương ứng với đường dẫn hiện tại.
- **Giao Diện:** Sử dụng Material-UI để tạo bố cục mượt mà, hiệu ứng chuyển đổi và giao diện tùy chỉnh theo theme.

2.7 PAGES

2.7.1 Status

Là trang hiển thị trạng thái của hệ thống, sử dụng `axios` để lấy danh sách thiết bị từ API và `Socket.IO` để nhận cập nhật trạng thái thiết bị thời gian thực. Nó hiển thị danh sách thiết bị với tên, trạng thái và thời gian cập nhật, đồng thời xử lý xác thực người dùng và thông báo lỗi qua `SnackbarContext`.



Hình 2.8: Giao diện Status

Import

Các thư viện và thành phần được nhập:

```
1 import { Box, Typography, List, ListItem, ListItemText } from
  "@mui/material";
2 import { useTheme } from "@mui/material/styles";
3 import React, { useEffect, useState } from "react";
4 import axios from "axios";
5 import io from "socket.io-client";
6 import { useNavigate } from "react-router-dom";
7 import { useSnackbar } from '../context/SnackbarContext';
```

- **Material-UI:** Cung cấp `Box`, `Typography`, `List`, `ListItem`, `ListItemText` cho giao diện và `useTheme` để lấy theme.
- **React, `useEffect`, `useState`:** Quản lý trạng thái và vòng đời component.
- **axios:** Gửi yêu cầu HTTP tới API.
- **io:** Kết nối `Socket.IO` thời gian thực.

- `useNavigate`: Điều hướng trang từ `react-router-dom`.
- `useSnackbar`: Hook hiển thị thông báo từ `SnackbarContext`.

API và Socket.IO

Định nghĩa URL API và kết nối Socket.IO:

```
1  const API_URL = import.meta.env.VITE_API_URL ||
   "http://localhost:5000";
2
3  const getToken = () => {
4      return document.cookie.split("; ").find(row =>
5          row.startsWith("authToken="))?.split("=")[1] || null;
6  };
7
8  const socket = io(API_URL, {
9      reconnection: true,
10     reconnectionAttempts: 5,
11     reconnectionDelay: 1000,
12     transports: ["websocket", "polling"],
13     auth: { token: getToken() }
14 });
```

- `API_URL`: Lấy từ biến môi trường hoặc mặc định `http://localhost:5000`.
- `getToken`: Lấy token xác thực từ cookie.
- `socket`: Kết nối Socket.IO với:
 - Tự động thử lại 5 lần, cách nhau 1 giây.
 - Ưu tiên WebSocket, fallback sang polling.
 - Gửi token xác thực qua `auth`.

Component StatusPage

Component nhận prop `user`:

```
1  const StatusPage = ({ user }) => {
```

- `user`: Thông tin người dùng để kiểm tra đăng nhập.

Trạng Thái

Quản lý trạng thái:

```
1  const theme = useTheme();
2  const navigate = useNavigate();
3  const [devices, setDevices] = useState([]);
4  const [error, setError] = useState(null);
5  const [loading, setLoading] = useState(true);
6  const { showSnackbar } = useSnackbar();
```

- theme: Lấy theme từ Material-UI.
- navigate: Hàm điều hướng trang.
- devices: Danh sách thiết bị từ API.
- error: Lưu thông báo lỗi.
- loading: Trạng thái tải dữ liệu.
- showSnackbar: Hiển thị thông báo.

Xử Lý Dữ Liệu và Socket.IO

Sử dụng `useEffect` để lấy dữ liệu và thiết lập Socket.IO:

```
1  useEffect(() => {
2    if (!user) {
3      navigate("/login");
4      return;
5    }
6
7    const fetchDevices = async () => {
8      try {
9        console.log("Fetching devices from API:",
10          `${API_URL}/api/devices`);
11        const response = await
12          axios.get(`${API_URL}/api/devices`, {
13            withCredentials: true,
14          });
15        console.log("Devices fetched:", response.data);
16        setDevices(response.data);
17        setLoading(false);
18      } catch (error) {
19        const errorMsg = error.response?.data?.message ||
20          error.message;
21        console.error("Error fetching devices:",
22          errorMsg);
23        setError(`Error fetching devices: ${errorMsg}`);
24        showSnackbar(`Error fetching devices:
25          ${errorMsg}`, "error");
26        setLoading(false);
27        if (errorMsg.includes("") ||
28          errorMsg.includes("Invalid token")) {
29          navigate("/login");
30        }
31      }
32    };
33
34    fetchDevices();
35
36    socket.on("connect", () => {
37      console.log("Connected to Socket.IO from Frontend!
38        ID:", socket.id);
```

```

31     });
32     socket.on("deviceUpdate", (updatedDevice) => {
33         console.log("Receive deviceUpdate:", updatedDevice);
34         try {
35             if (!updatedDevice || !updatedDevice.name) {
36                 console.warn("Invalid deviceUpdate data",
37                     updatedDevice);
38                 return;
39             }
40             setDevices((prev) => {
41                 const existingDevice = prev.find((device) =>
42                     device.name === updatedDevice.name);
43                 if (existingDevice) {
44                     return prev.map((device) =>
45                         device.name === updatedDevice.name ?
46                             updatedDevice : device
47                     );
48                 }
49                 return [...prev, updatedDevice];
50             });
51             } catch (error) {
52                 console.error("Error processing deviceUpdate:",
53                     error.message);
54                 setError('Error processing deviceUpdate:
55                     ${error.message}');
56                 showSnackbar('Error processing deviceUpdate:
57                     ${error.message}', "error");
58             }
59         });
60     socket.on("connect_error", (err) => {
61         console.error("Socket.IO connection error:",
62             err.message);
63         setError('Socket.IO connection error:
64             ${err.message}');
65         showSnackbar('Socket.IO connection error:
66             ${err.message}', "error");
67         navigate("/login");
68     });
69
70     return () => {
71         socket.off("deviceUpdate");
72         socket.off("connect");
73         socket.off("connect_error");
74         socket.disconnect();
75     };
76 }, [user, navigate]);

```

- Kiểm tra user: Nếu chưa đăng nhập, chuyển hướng tới /login.
- fetchDevices: Gửi yêu cầu GET tới /api/devices để lấy danh sách thiết bị, xử lý lỗi và hiển thị thông báo qua showSnackbar.

- Socket.IO:
 - Lắng nghe connect: Ghi log khi kết nối thành công.
 - Lắng nghe deviceUpdate: Cập nhật hoặc thêm thiết bị vào devices.
 - Lắng nghe connecterror: Hiển thị lỗi và chuyển hướng tới /login.
- Cleanup: Hủy các listener và ngắt kết nối Socket.IO khi component unmount.

Cấu Trúc JSX

Giao diện của StatusPage:

```

1 return (
2   <Box sx={{ p: 3, minHeight: "100vh", bgcolor:
   "background.default" }}>
3     <Typography variant="h4" sx={{ color: "text.primary", mb:
       2 }}>
4       Device Status
5     </Typography>
6     {loading && (
7       <Typography sx={{ color: "text.secondary" }}>
8         Loading devices...
9       </Typography>
10    )}
11    {error && (
12      <Typography sx={{ color: "error.main" }}>
13        {error}
14      </Typography>
15    )}
16    {devices.length === 0 && !loading ? (
17      <Typography sx={{ color: "text.secondary" }}>
18        No devices found.
19      </Typography>
20    ) : (
21      <List sx={{ p: 0 }}>
22        {devices.map((device, index) => (
23          <ListItem
24            key={device.name || index}
25            sx={{
26              my: 1,
27              p: 1.5,
28              bgcolor: "background.paper",
29              borderRadius: "5px",
30              color: "text.primary"
31            }}
32          >
33            <ListItemText
34              primary={` ${device.name} - Status:
35                ${device.status} (Update: ${
36                  device.updatedAt ? new

```

```

37         }) '}'
38       />
39     </ListItem>
40   )})
41 </List>
42 })
43 </Box>
44 );

```

- **Box:** Container chính với padding 3, chiều cao tối thiểu 100vh, màu nền từ theme.
- **Typography** (tiêu đề): "Device Status" với kiểu h4.
- **Trạng thái tải:** Hiển thị "Đang tải thiết bị..." khi loading là true.
- **Lỗi:** Hiển thị thông báo lỗi với màu đỏ nếu error tồn tại.
- **Không có thiết bị:** Hiển thị "Không có thiết bị nào" nếu devices rỗng và không tải.
- **Danh sách thiết bị:** Sử dụng List và ListItem để hiển thị thông tin thiết bị (tên, trạng thái, thời gian cập nhật).

Export

Component được xuất:

```

1 export default StatusPage;

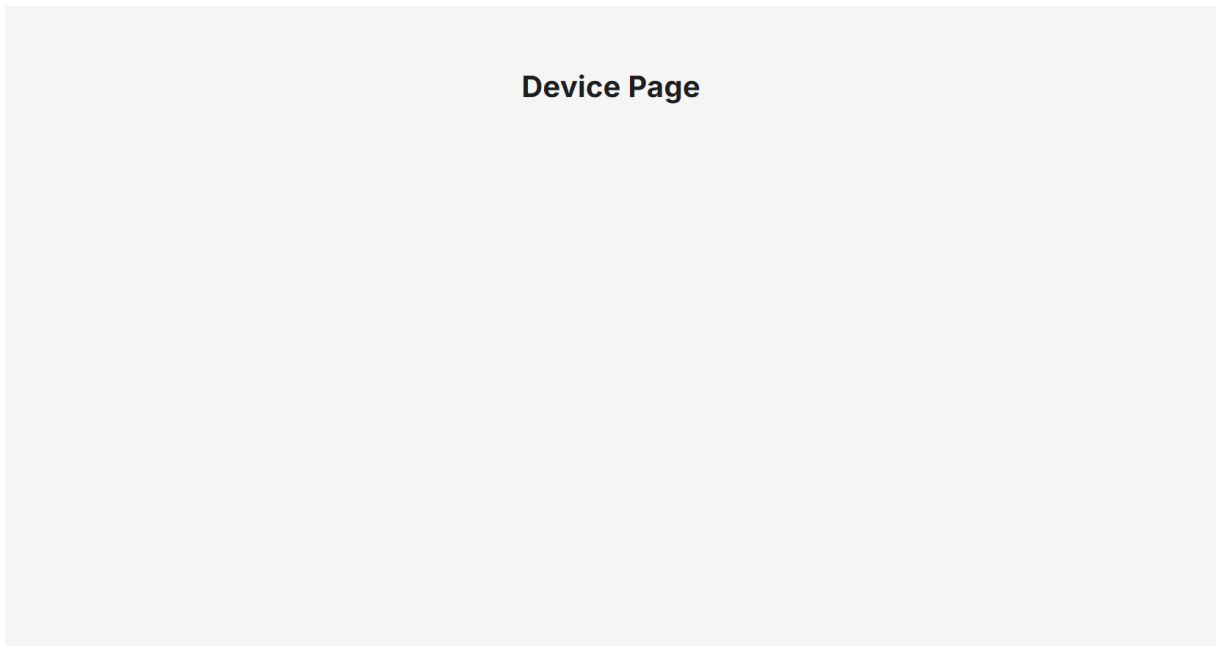
```

Chức Năng Chính

- **Xác Thực Người Dùng:** Chuyển hướng tới /login nếu chưa đăng nhập hoặc token không hợp lệ.
- **Lấy Danh Sách Thiết Bị:** Gửi yêu cầu API để lấy danh sách thiết bị ban đầu.
- **Cập Nhật Thời Gian Thực:** Nhận cập nhật trạng thái thiết bị qua Socket.IO (deviceUpdate).
- **Xử Lý Lỗi:** Hiển thị thông báo lỗi qua showSnackBar và chuyển hướng nếu cần.
- **Giao Diện:** Hiển thị danh sách thiết bị với thông tin rõ ràng, hỗ trợ trạng thái tải và thông báo lỗi.

2.7.2 Device

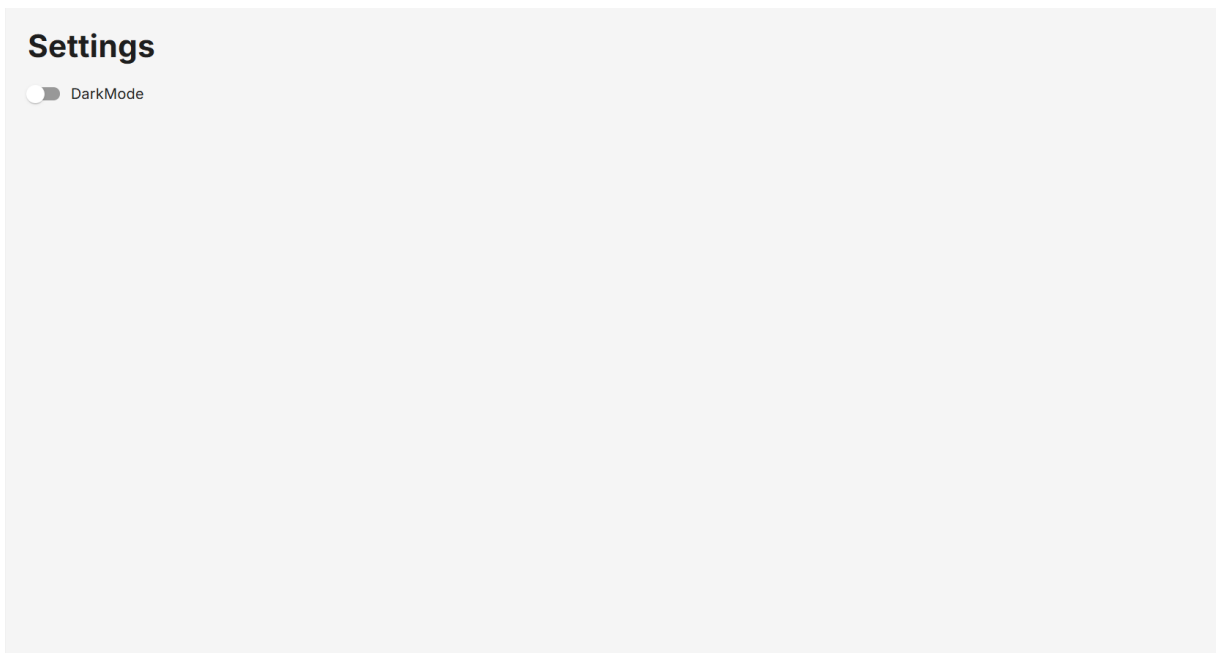
Là trang hiển thị danh sách thiết bị, cho phép người dùng thêm, sửa và xóa thiết bị. Nó sử dụng axios để gửi yêu cầu tới API và Socket.IO để nhận cập nhật thời gian thực. Giao diện bao gồm danh sách thiết bị, các nút hành động và hộp thoại xác nhận.



Hình 2.9: Giao diện Device

2.7.3 Setting

Là trang sử dụng Material-UI để cung cấp giao diện cho phép người dùng thiết lập các tùy chỉnh của trang, đầu tiên là chuyển đổi giữa chế độ sáng (**light**) và chế độ tối (**dark**) thông qua một công tắc (**Switch**)



Hình 2.10: Giao diện Setting

Import

Các thư viện và thành phần được nhập:

```
1      import React from 'react';
2      import { Typography, Switch, FormControlLabel }
        from '@mui/material';
```

- React: Thư viện chính để xây dựng component.
- Material-UI: Cung cấp các thành phần:
 - Typography: Hiển thị tiêu đề.
 - Switch: Công tắc để chuyển đổi chế độ.
 - FormControlLabel: Nhãn cho công tắc.

Component SettingPage

Component nhận các props:

```
1      const SettingPage = ({ mode, setMode }) => {
```

- mode: Chế độ giao diện hiện tại (light hoặc dark).
- setMode: Hàm cập nhật chế độ giao diện.

Xử Lý Sự Kiện

Hàm xử lý chuyển đổi chế độ:

```
1      const handleToggle = () => {
2          setMode(mode === 'light' ? 'dark' : 'light');
3      };
```

- handleToggle: Chuyển đổi mode giữa light và dark bằng cách gọi setMode.

Cấu Trúc JSX

Giao diện của SettingPage:

```
1      return (
2          <div>
3              <Typography variant="h4" gutterBottom
4                  fontWeight="bold">
5                  Settings
6              </Typography>
7
8              <FormControlLabel
9                  control={<Switch checked={mode === 'dark'}
10                      onChange={handleToggle} />}
11                  label={mode === 'dark' ? 'LightMode' : 'DarkMode'}
12          </div>
13      );
```

- `div`: Container chính cho giao diện.
- **Typography**: Tiêu đề "Settings" với kiểu `h4`, in đậm, có khoảng cách dưới (`gutterBottom`).
- **FormControlLabel**: Nhãn và công tắc:
 - `control`: Switch được chọn khi mode là `dark`, gọi `handleToggle` khi thay đổi.
 - `label`: Hiển thị `LightMode` khi mode là `dark`, và `DarkMode` khi mode là `light`.

Export

Component được xuất:

```
1 export default SettingPage;
```

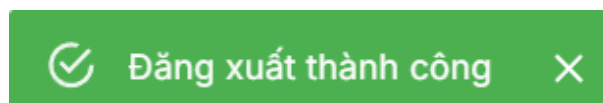
Chức Năng Chính

- **Chuyển Đổi Chế Độ Giao Diện**: Cho phép người dùng chuyển đổi giữa chế độ sáng (`light`) và chế độ tối (`dark`) thông qua công tắc.
- **Giao Diện Trực Quan**: Sử dụng Material-UI để tạo bố cục đơn giản với tiêu đề và công tắc được sắp xếp rõ ràng.
- **Quản Lý Trạng Thái**: Nhận và cập nhật trạng thái `mode` thông qua props `mode` và `setMode`.

2.8 CONTEXTS

2.8.1 SnackbarContext

Dùng để hiển thị các thông báo của ứng dụng sử dụng Snackbar trong MUI thay cho thông báo mặc định dùng Alert của Javascript.



Hình 2.11: Giao diện Snackbar

Import

Các thư viện và thành phần được nhập:

```
1 import React, { createContext, useContext, useState } from  
  'react';  
2 import { Snackbar, Alert, Slide } from '@mui/material';
```

- `React, createContext, useContext, useState`: Quản lý ngữ cảnh và trạng thái.
- **Material-UI**: Cung cấp:
 - `Snackbar`: Hiển thị thông báo tạm thời.

- Alert: Thành phần thông báo với mức độ nghiêm trọng.
- Slide: Hiệu ứng trượt cho thông báo.

Tạo Context

Tạo ngữ cảnh để chia sẻ hàm `showSnackbar`:

```
1  const SnackbarContext = createContext();
```

- `SnackbarContext`: Ngữ cảnh để các component con truy cập `showSnackbar`.

Component SnackbarProvider

Component cung cấp ngữ cảnh và giao diện thông báo:

```
1  export const SnackbarProvider = ({ children }) => {  
2    const [open, setOpen] = useState(false);  
3    const [message, setMessage] = useState('');  
4    const [severity, setSeverity] = useState('success');
```

- `children`: Các component con được bao bọc bởi `SnackbarProvider`.
- `open`: Trạng thái hiển thị của snackbar (`true/false`).
- `message`: Nội dung thông báo.
- `severity`: Mức độ nghiêm trọng (`success, error, v.v.`).

Xử Lý Thông Báo

Hàm kích hoạt và đóng thông báo:

```
1  const showSnackbar = (msg, sev = 'success') => {  
2    setMessage(msg);  
3    setSeverity(sev);  
4    setOpen(true);  
5  };  
6  
7  const handleClose = (event, reason) => {  
8    if (reason === 'clickaway') {  
9      return;  
10   }  
11   setOpen(false);  
12  };
```

- `showSnackbar`: Cập nhật `message`, `severity` và mở snackbar.
- `handleClose`: Đóng snackbar, bỏ qua nếu người dùng click ra ngoài (`clickaway`).

Cấu Trúc JSX

Giao diện và cung cấp ngữ cảnh:

```

1 return (
2   <SnackbarContext.Provider value={{ showSnackbar }}>
3     {children}
4     <Snackbar
5       open={open}
6       autoHideDuration={3000}
7       onClose={handleClose}
8       anchorOrigin={{ vertical: 'top', horizontal: 'right' }}
9       TransitionComponent={Slide}
10      transitionDuration={500}
11    >
12      <Alert
13        onClose={handleClose}
14        severity={severity}
15        sx={{
16          width: '100%',
17          bgcolor: severity === 'success' ? '#4caf50' :
18            '#f44336',
19          color: '#fff',
20          '& .MuiAlert-icon': {
21            color: '#fff',
22          },
23          borderRadius: '8px',
24          boxShadow: '0px 4px 12px rgba(0, 0, 0, 0.1)',
25        }}
26      >
27        {message}
28      </Alert>
29    </Snackbar>
30  </SnackbarContext.Provider>
31 );

```

- **SnackbarContext.Provider**: Cung cấp `showSnackbar` cho các component con.
- **children**: Hiển thị các component con.
- **Snackbar**: Thông báo với:
 - Hiển thị khi `open` là `true`.
 - Tự động ẩn sau 3 giây (`autoHideDuration=3000`).
 - Vị trí góc trên bên phải (`anchorOrigin`).
 - Hiệu ứng trượt (`Slide`) trong 500ms.
- **Alert**: Thành phần thông báo với:
 - Mức độ **severity** (ảnh hưởng màu sắc).
 - Tùy chỉnh giao diện: Màu nền xanh (`#4caf50`) cho `success`, đỏ (`#f44336`) cho `error`; chữ và icon trắng; bo góc và bóng.

Hook useSnackbar

Hook để truy cập showSnackbar:

```
1 export const useSnackbar = () => {  
2   const context = useContext(SnackbarContext);  
3   if (!context) {  
4     throw new Error('useSnackbar must be used within a  
       SnackbarProvider');  
5   }  
6   return context;  
7 };
```

- useSnackbar: Truy cập SnackbarContext, ném lỗi nếu không nằm trong SnackbarProvider.

Chức Năng Chính

- **Hiển Thị Thông Báo:** Cho phép các component con kích hoạt thông báo với nội dung và mức độ nghiêm trọng tùy chỉnh qua showSnackbar.
- **Hiệu Ứng Trượt:** Sử dụng Slide để tạo hiệu ứng mượt mà khi thông báo xuất hiện/biến mất.
- **Tùy Chỉnh Giao Diện:** Tùy chỉnh màu sắc, bo góc và bóng cho thông báo dựa trên severity.
- **Quản Lý Trạng Thái:** Điều khiển hiển thị, nội dung và loại thông báo thông qua trạng thái open, message, severity.

2.9 HOOKS

2.9.1 useAuth

Là một hook tùy chỉnh, cung cấp các chức năng quản lý xác thực người dùng, bao gồm đăng nhập và đăng xuất. Nó tích hợp với API thông qua axios, sử dụng Socket.IO để quản lý kết nối thời gian thực, và hiển thị thông báo qua SnackbarContext. Hook này được sử dụng trong các component như Header để xử lý trạng thái đăng nhập.

Import

Các thư viện và thành phần được nhập:

```
1 import { useState } from "react";  
2 import { useNavigate } from "react-router-dom";  
3 import axios from "axios";  
4 import loginUser from "../api/loginUser";  
5 import { useSnackbar } from '../context/SnackbarContext';
```

- useState: Quản lý trạng thái cục bộ.
- useNavigate: Điều hướng trang từ react-router-dom.

- axios: Gửi yêu cầu HTTP tới API.
- loginUser: Hàm API tùy chỉnh để đăng nhập.
- useSnackbar: Hook hiển thị thông báo từ SnackbarContext.

Hook useAuth

Hook nhận các tham số và quản lý trạng thái:

```
1  const useAuth = (setUser, socket) => {  
2    const [email, setEmail] = useState("");  
3    const [password, setPassword] = useState("");  
4    const [openLogin, setOpenLogin] = useState(false);  
5    const { showSnackbar } = useSnackbar();  
6    const navigate = useNavigate();
```

- setUser: Hàm cập nhật trạng thái người dùng.
- socket: Đối tượng Socket.IO để quản lý kết nối.
- email, setEmail: Quản lý email nhập vào.
- password, setPassword: Quản lý mật khẩu nhập vào.
- openLogin, setOpenLogin: Điều khiển hiển thị hộp thoại đăng nhập.
- showSnackbar: Hiển thị thông báo.
- navigate: Điều hướng trang.

Xử Lý Đăng Nhập

Hàm xử lý đăng nhập:

```
1  const handleLogin = async () => {  
2    const res = await loginUser(email, password);  
3    if (res.success) {  
4      setUser({ username: res.username, avatar: res.avatar  
5        });  
6      localStorage.setItem("user", JSON.stringify({  
7        username: res.username, avatar: res.avatar }));  
8      setOpenLogin(false);  
9      setPassword("");  
10     socket.connect();  
11     showSnackbar("Login successful", "success");  
12     navigate("/status");  
13   } else {  
14     showSnackbar(res.message || "Login failed", "error");  
15   }  
16 }
```

- Gọi loginUser với email và password.

- Nếu thành công:
 - Cập nhật user với username và avatar.
 - Lưu thông tin vào localStorage.
 - Đóng hộp thoại, xóa mật khẩu, kết nối Socket.IO.
 - Hiển thị thông báo thành công và chuyển hướng tới /status.
- Nếu thất bại: Hiển thị thông báo lỗi.

Xử Lý Đăng Xuất

Hàm xử lý đăng xuất:

```
1  const handleLogout = async () => {  
2    try {  
3      await axios.post(  
4        `${import.meta.env.VITE_API_URL}/api/users/logout`,  
5        {},  
6        { withCredentials: true }  
7      );  
8      setUser(null);  
9      localStorage.removeItem("user");  
10     socket.disconnect();  
11     showSnackbar("Logout successful", "success");  
12     navigate("/status");  
13   } catch (error) {  
14     console.error("Error during logout:", error);  
15     showSnackbar("Error during logout", "error");  
16   }  
17 };
```

- Gửi yêu cầu POST tới /api/users/logout.
- Nếu thành công:
 - Xóa user và localStorage.
 - Ngắt kết nối Socket.IO.
 - Hiển thị thông báo thành công và chuyển hướng tới /status.
- Nếu thất bại: Ghi log và hiển thị thông báo lỗi.

Trả Về

Hook trả về các giá trị và hàm:

```
1  return {  
2    email,  
3    setEmail,  
4    password,  
5    setPassword,  
6    openLogin,
```

```
7     setOpenLogin ,  
8     handleLogin ,  
9     handleLogout ,  
10  };
```

- Trả về trạng thái (`email`, `password`, `openLogin`) và các hàm xử lý để sử dụng trong component.

Export

Hook được xuất:

```
1  export default useAuth;
```

Chức Năng Chính

- **Quản Lý Đăng Nhập:** Xử lý đăng nhập thông qua API, cập nhật trạng thái người dùng, lưu trữ cục bộ, và kết nối Socket.IO.
- **Quản Lý Đăng Xuất:** Gửi yêu cầu đăng xuất, xóa dữ liệu người dùng, ngắt Socket.IO, và điều hướng trang.
- **Hiển Thị Thông Báo:** Sử dụng `showSnackBar` để thông báo kết quả đăng nhập/dăng xuất.
- **Điều Hướng Trang:** Chuyển hướng tới `/status` sau khi đăng nhập hoặc đăng xuất.
- **Quản Lý Trạng Thái:** Cung cấp trạng thái và hàm để điều khiển giao diện đăng nhập.

2.10 API

2.10.1 loginUser

Là một hàm API sử dụng `axios` để gửi yêu cầu đăng nhập người dùng tới server. Hàm này nhận email và mật khẩu, gửi yêu cầu POST tới endpoint API, và trả về thông tin người dùng nếu thành công hoặc thông báo lỗi nếu thất bại. Nó được sử dụng trong hook `useAuth` để xử lý logic đăng nhập.

2.10.2 Import

Thư viện được nhập:

```
1  import axios from "axios";
```

- `axios`: Thư viện để gửi yêu cầu HTTP tới API.

Cấu Hình Axios

Cấu hình mặc định cho axios:

```
1 axios.defaults.withCredentials = true;
```

- **withCredentials**: Cho phép gửi cookie trong các yêu cầu HTTP, cần thiết cho xác thực dựa trên cookie.

Hàm loginUser

Hàm xử lý đăng nhập:

```
1  const loginUser = async (email, password) => {  
2    try {  
3      localStorage.removeItem("token");  
4      localStorage.removeItem("authToken");  
5  
6      const res = await  
          axios.post(`${import.meta.env.VITE_API_URL}/api/users/login`,  
              {  
2         email,  
3         password,  
4       });  
5      localStorage.setItem("user", JSON.stringify({  
6        username: res.data.username, avatar:  
7        res.data.avatar }));  
8      return { success: true, username: res.data.username,  
9        avatar: res.data.avatar };  
10     } catch (err) {  
11       return { success: false, message:  
12         err.response?.data?.message || "Login error" };  
13     }  
14   }  
15 }
```

- **email, password**: Tham số đầu vào cho yêu cầu đăng nhập.
- Xóa **token** và **authToken** từ **localStorage** để đảm bảo không sử dụng token cũ.
- Gửi yêu cầu POST tới **/api/users/login** với **email** và **password**.
- Nếu thành công:
 - Lưu thông tin người dùng (**username, avatar**) vào **localStorage**.
 - Trả về đối tượng **{ success: true, username, avatar }**.
- Nếu thất bại:
 - Trả về đối tượng **{ success: false, message }** với thông báo lỗi từ server hoặc mặc định là **"Login error"**.

Export

Hàm được xuất:

```
1 export default loginUser;
```

Chức Năng Chính

- **Gửi Yêu Cầu Đăng Nhập:** Sử dụng `axios` để gửi email và mật khẩu tới endpoint `/api/users/login`.
- **Xử Lý Phản Hồi:** Lưu thông tin người dùng vào `localStorage` nếu thành công, hoặc trả về thông báo lỗi nếu thất bại.
- **Xóa Token Cũ:** Xóa các token cũ từ `localStorage` trước khi đăng nhập để tránh xung đột.
- **Hỗ Trợ Xác Thực Cookie:** Cấu hình `withCredentials` cho phép gửi cookie xác thực.

2.11 APP

Là thành phần chính của ứng dụng, chịu trách nhiệm tổ chức giao diện, quản lý trạng thái toàn cục (người dùng, chế độ giao diện, sidebar), và định tuyến các trang. Nó tích hợp `Material-UI` để tạo giao diện, `react-router-dom` cho định tuyến, và `SnackbarProvider` cho thông báo đồng thời cũng kiểm tra xác thực người dùng khi khởi động.

2.11.1 Import

Các thư viện và thành phần được nhập:

```
1 import React, { useState, useEffect } from 'react';
2 import { Routes, Route, Navigate } from 'react-router-dom';
3 import { Box, Toolbar, CssBaseline, ThemeProvider } from
  '@mui/material';
4 import { getTheme } from './theme';
5 import Header from './components/Header';
6 import Sidebar from './components/Sidebar';
7 import StatusPage from './pages/StatusPage';
8 import DevicePage from './pages/DevicePage';
9 import SettingPage from './pages/SettingPage';
10 import { SnackbarProvider } from './context/SnackbarContext';
11 import axios from 'axios';
```

- `React, useState, useEffect`: Quản lý trạng thái và vòng đời component.
- `react-router-dom`: `Routes, Route, Navigate` cho định tuyến.
- `Material-UI`: `Box, Toolbar, CssBaseline, ThemeProvider` cho giao diện và theme.
- `getTheme`: Hàm tùy chỉnh để tạo theme dựa trên chế độ sáng/tối.

- Header, Sidebar, StatusPage, DevicePage, SettingPage: Các component con.
- SnackbarProvider: Cung cấp ngữ cảnh thông báo.
- axios: Gửi yêu cầu HTTP tới API.

2.11.2 API URL

Định nghĩa URL API:

```
1      const API_URL = import.meta.env.VITE_API_URL ||  
      "http://localhost:5000";
```

- API_URL: Lấy từ biến môi trường hoặc mặc định http://localhost:5000.

2.11.3 Component App

Component quản lý trạng thái toàn cục:

```
1      const App = () => {  
2          const [sidebarOpen, setSidebarOpen] = useState(true);  
3          const [mode, setMode] = useState('light');  
4          const [user, setUser] = useState(() => {  
5              const savedUser = localStorage.getItem('user');  
6              return savedUser ? JSON.parse(savedUser) : null;  
7          });  
8          const [openLogin, setOpenLogin] = useState(false);
```

- sidebarOpen: Trạng thái mở/rút gọn của sidebar.
- mode: Chế độ giao diện (light hoặc dark).
- user: Thông tin người dùng, khởi tạo từ localStorage hoặc null.
- openLogin: Điều khiển hiển thị hộp thoại đăng nhập.

2.11.4 Kiểm Tra Xác Thực

Sử dụng useEffect để kiểm tra token:

```
1      useEffect(() => {  
2          const verifyToken = async () => {  
3              try {  
4                  const response = await  
5                      axios.get(`${API_URL}/api/users/verify-token`,  
6                      {  
7                          withCredentials: true,  
8                      });  
9                  if (!response.data.valid) {  
10                     setUser(null);  
11                     localStorage.removeItem('user');  
12                 }  
13             }  
14         }  
15     }, []);
```

```
11         } catch (error) {
12             console.error("Error verifying token:", error);
13             setUser(null);
14             localStorage.removeItem('user');
15         }
16     };
17
18     if (user) {
19         verifyToken();
20     }
21 }, []);
```

- Gửi yêu cầu GET tới /api/users/verify-token để kiểm tra token.
- Nếu token không hợp lệ hoặc có lỗi, xóa user và localStorage.
- Chỉ chạy khi có user, với mảng phụ thuộc rỗng ([]) để chạy một lần khi mount.

2.11.5 Cấu Trúc JSX

Giao diện và định tuyến của ứng dụng:

```
1 return (
2     <ThemeProvider theme={getTheme(mode)}>
3         <SnackbarProvider>
4             <CssBaseline />
5             <Box sx={{ display: 'flex' }}>
6                 <Sidebar open={sidebarOpen} />
7                 <Box component="main" sx={{ flexGrow: 1,
8                     minHeight: '100vh' }}>
9                     <Header
10                         onToggleSidebar={() =>
11                             setSidebarOpen(prev => !prev)}
12                         user={user}
13                         setUser={setUser}
14                         openLogin={openLogin}
15                         setOpenLogin={setOpenLogin}
16                     />
17                     <Toolbar />
18                     <Box sx={{ p: 3 }}>
19                         <Routes>
20                             <Route
21                                 path="/status"
22                                 element={user ? <StatusPage
23                                     user={user} /> : <Navigate
24                                     to="/" />}
25                             />
26                             <Route
27                                 path="/device"
28                                 element={user ? <DevicePage
29                                     user={user} /> : <Navigate
30                                     to="/" />}
31                             />
32                         </Routes>
33                     </Box>
34                 </Box>
35             </SnackbarProvider>
36         </ThemeProvider>
37     );
```

```

25         />
26         <Route
27             path="/setting"
28             element={<SettingPage mode={mode}
29                 setMode={setMode} />}
30         />
31         <Route path="/" element={<Navigate
32             to="/status" />} />
33         <Route path="*" element={<Navigate
34             to="/status" />} />
35     </Routes>
36 </Box>
37 </Box>
38 </Box>
39 </SnackbarProvider>
40 </ThemeProvider>
41 );

```

- ThemeProvider: Áp dụng theme từ `getTheme(mode)`.
- SnackbarProvider: Bao bọc ứng dụng để cung cấp ngữ cảnh thông báo.
- CssBaseline: Đặt lại kiểu CSS mặc định của Material-UI.
- Box (container chính): Sử dụng flexbox để sắp xếp Sidebar và nội dung chính.
- Sidebar: Thanh điều hướng bên, điều khiển bởi `sidebarOpen`.
- Box (nội dung chính): Chứa Header, Toolbar (khoảng trống), và các trang.
- Header: Thanh điều hướng trên, truyền các prop để quản lý sidebar và người dùng.
- Routes: Định tuyến các trang:
 - `/status`: Hiển thị `StatusPage` nếu có user, ngược lại chuyển hướng tới `/`.
 - `/device`: Hiển thị `DevicePage` nếu có user, ngược lại chuyển hướng tới `/`.
 - `/setting`: Hiển thị `SettingPage` để chuyển đổi chế độ sáng/tối.
 - `/` và `*`: Chuyển hướng tới `/status`.

2.11.6 Export

Component được xuất:

```

1 export default App;

```


2.11.7 Chức Năng Chính

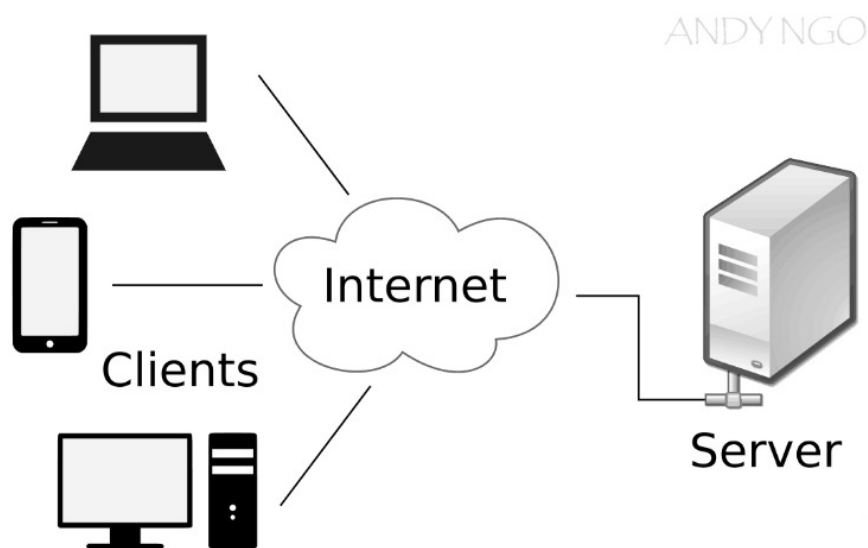
- **Quản Lý Giao Diện:** Sử dụng Material-UI để tổ chức bố cục với sidebar, header và nội dung chính, hỗ trợ chế độ sáng/tối qua `mode`.
- **Xác Thực Người Dùng:** Kiểm tra token khi khởi động, tự động đăng xuất nếu token không hợp lệ.
- **Định Tuyến Trang:** Sử dụng `react-router-dom` để điều hướng giữa các trang `StatusPage`, `DevicePage`, `SettingPage`, với bảo vệ tuyến đường dựa trên trạng thái `user`.
- **Quản Lý Trạng Thái Toàn Cục:** Quản lý `user`, `sidebarOpen`, `mode`, và `openLogin`.
- **Hiển Thị Thông Báo:** Tích hợp `SnackbarProvider` để hiển thị thông báo trong toàn ứng dụng.

Chương 3

BACK END

3.1 Cấu trúc hệ thống

3.1.1 Mô hình Client - Server

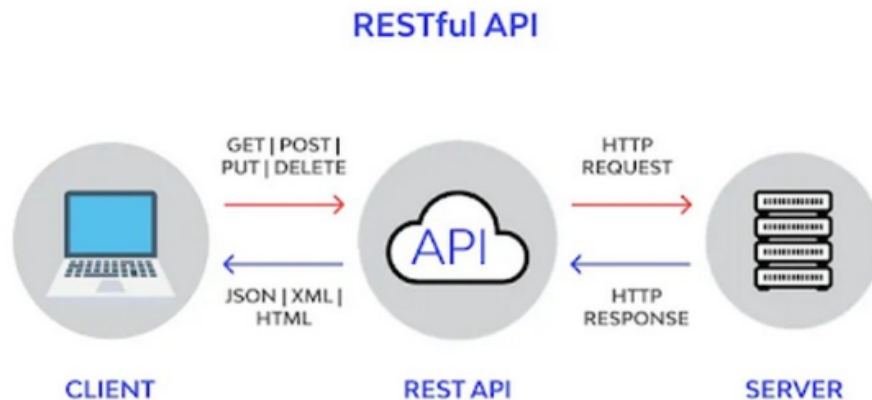


Hình 3.1: Mô hình Client - Server

Mô hình Client - Server là một mô hình kiến trúc mạng trong đó các ứng dụng được chia thành hai phần: client và server. Client là phần mềm chạy trên máy tính của người dùng, trong khi server là phần mềm chạy trên máy chủ. Client gửi yêu cầu đến server và nhận phản hồi từ server.

3.1.2 Kiến trúc RESTful API

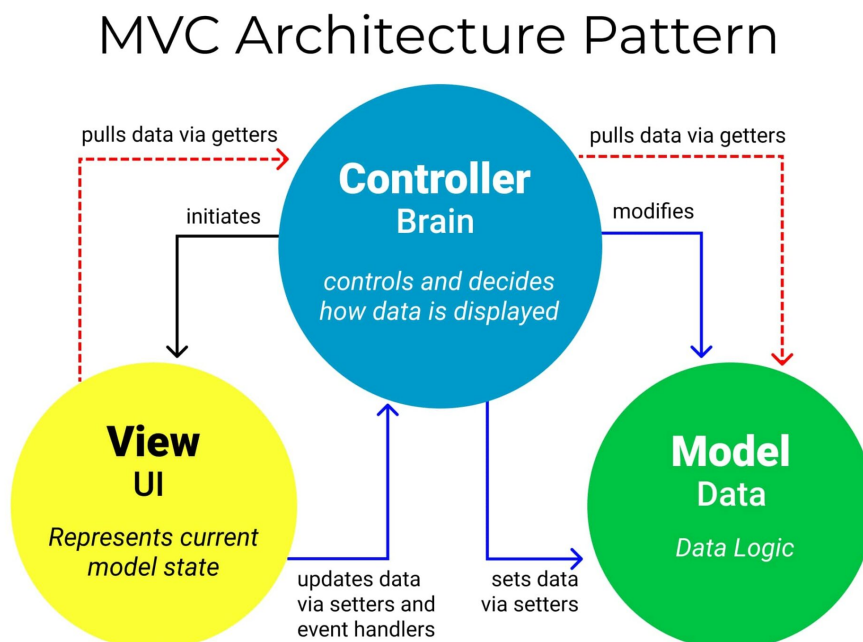
Kiến trúc RESTful API là một kiểu kiến trúc phần mềm cho phép các ứng dụng giao tiếp với nhau thông qua HTTP. RESTful API sử dụng các phương thức HTTP như GET, POST, PUT, DELETE để thực hiện các thao tác CRUD (Create, Read, Update, Delete) trên tài nguyên.



Hình 3.2: Kiến trúc RESTful API

3.1.3 Mô hình MVC

Mô hình MVC (Model-View-Controller) là một mô hình thiết kế phần mềm được sử dụng phổ biến trong phát triển ứng dụng web. Mô hình này chia ứng dụng thành ba phần: Model (mô hình), View (giao diện) và Controller (bộ điều khiển). Mỗi phần có nhiệm vụ riêng và tương tác với nhau để tạo ra ứng dụng hoàn chỉnh.

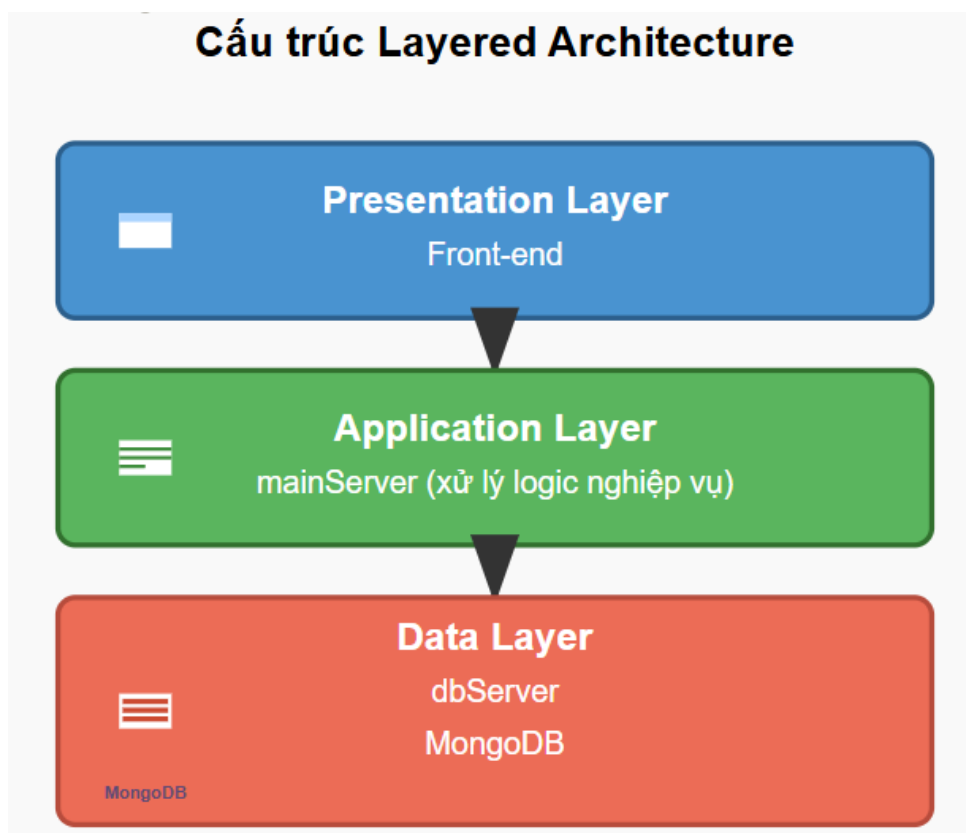


Hình 3.3: Mô hình MVC

3.1.4 Kiến trúc Layered

Kiến trúc Layered là một kiểu kiến trúc phần mềm trong đó ứng dụng được chia thành nhiều lớp. Mỗi lớp có nhiệm vụ riêng và tương tác với các lớp khác thông qua các giao diện. Kiến trúc Layered giúp tách biệt các phần của ứng dụng, dễ dàng bảo trì và mở rộng. Ở đây chúng ta có:

- Presentation Layer: Front-end.
- Application Layer: mainServer
- Data Layer: dbServer và MongoDB.



Hình 3.4: Kiến trúc Layered

3.2 Cấu trúc thư mục

Cấu trúc thư mục của dự án được tổ chức như sau:

- mainServer: Thư mục chứa mã nguồn của Main Server.
 - controllers: Thư mục chứa các controller.
 - middlewares: Thư mục chứa các middleware.
 - routes: routes: Thư mục chứa các route.
 - upload: Thư mục chứa các tệp tải lên.

- server.js: Tập chính của Main Server.
- dbServer: Thư mục chứa mã nguồn của Database Server.
 - config: Thư mục chứa các tệp cấu hình.
 - models: Thư mục chứa các mô hình dữ liệu.
 - seeds: Thư mục chứa các tệp khởi tạo dữ liệu.
 - dbserver.js: Tập chính của Database Server.
- .env: Tệp cấu hình môi trường.

3.3 Cài đặt

Các bước khởi tạo server:

- Khởi tạo NodeJS: `npm init -y`
- Cài đặt dependencies:
 - Express: `npm install express`
 - Mongoose: `npm install mongoose`
 - Multer: `npm install multer`
 - Dotenv: `npm install dotenv`
 - Cors: `npm install cors`
 - Bcrypt: `npm install bcrypt`
 - Jsonwebtoken: `npm install jsonwebtoken`
- Tạo server chính: `touch server.js` và `touch dbserver.js`
- Chạy server bằng lệnh: `npm start`

Các thư viện khác cài đặt trong quá trình phát triển.

Chương 4

MAIN SERVER

4.1 CONTROLLERS

Là nơi chứa logic xử lý cho các endpoint API, thuộc tầng Controller (MVC) hoặc Application Layer (Layered Architecture). Có chức năng:

- Xử lý logic nghiệp vụ: Các hàm trong controller (như loginUser, getDevices) xử lý yêu cầu từ Front-end, gọi API của dbServer để lấy dữ liệu, và trả kết quả.
- Điều phối giữa Route và Model: Kết nối các route (trong routes/) với dữ liệu từ dbServer.

4.1.1 deviceController

Chứa các hàm xử lý liên quan đến thiết bị, như thêm, sửa, xóa thiết bị. Các hàm này sẽ gọi API của dbServer để thực hiện các thao tác trên cơ sở dữ liệu.

Import

Thư viện được nhập:

```
1 import axios from "axios";
```

- axios: Thư viện để gửi yêu cầu HTTP tới server cơ sở dữ liệu.

Biến URL

Định nghĩa URL server cơ sở dữ liệu:

```
1 const DB_SERVER_URL = process.env.DB_SERVER_URL  
  || "http://localhost:5001";
```

- DB_SERVER_URL: Lấy từ biến môi trường hoặc mặc định http://localhost:5001.

Hàm getDevices

Hàm xử lý yêu cầu lấy danh sách thiết bị:

```
1 export const getDevices = async (req, res) => {  
2   try {  
3     const response = await  
4       axios.get(`${DB_SERVER_URL}/db/devices`);  
5     res.json(response.data);  
6   } catch (error) {  
7     res.status(500).json({ message: "Error fetching  
8       device list" });  
9   }  
10 }
```

- **req, res:** Tham số yêu cầu và phản hồi từ framework như Express.js.
- Gửi yêu cầu GET tới `/db/devices` trên `DB_SERVER_URL`.
- Nếu thành công:
 - Trả về dữ liệu JSON từ phản hồi của server (`response.data`).
- Nếu thất bại:
 - Trả về mã trạng thái 500 và thông báo lỗi `{ message: "Error fetching device list" }`.

Export

Hàm được xuất:

```
1 export default getDevices;
```

- Xuất hàm để sử dụng trong các tuyến API của ứng dụng Node.js.

Chức Năng Chính

- **Lấy Danh Sách Thiết Bị:** Sử dụng `axios` để gửi yêu cầu GET tới endpoint `/db/devices` trên server cơ sở dữ liệu.
- **Xử Lý Phản Hồi:** Trả về danh sách thiết bị dưới dạng JSON nếu thành công, hoặc thông báo lỗi nếu thất bại.
- **Xử Lý Lỗi:** Trả về mã trạng thái 500 và thông báo lỗi khi không thể lấy dữ liệu.
- **Tích Hợp API:** Được thiết kế để sử dụng trong các tuyến API, tương thích với Express.js.

UserController

Các hàm API `registerUser`, `loginUser`, `logoutUser`, `updateAvatar`, `changePassword`, cùng các hàm hỗ trợ, là các hàm xử lý yêu cầu liên quan đến quản lý người dùng trong ứng dụng Node.js. Sử dụng `axios` để giao tiếp với server cơ sở dữ liệu, `jwt` để tạo token xác thực, và `fs` để quản lý tệp avatar, các hàm này cung cấp các chức năng đăng ký, đăng nhập, đăng xuất, cập nhật avatar và đổi mật khẩu. Báo cáo này phân tích chi tiết mã nguồn, chức năng và thiết kế của các hàm.

4.1.2 Import

Các thư viện được nhập:

```
1 import axios from "axios";
2 import jwt from "jsonwebtoken";
3 import fs from "fs";
4 import path from "path";
5 import { fileURLToPath } from "url";
```

- `axios`: Gửi yêu cầu HTTP tới server cơ sở dữ liệu.
- `jwt`: Tạo và xác minh JSON Web Token cho xác thực.
- `fs`: Quản lý hệ thống tệp (tệp avatar).
- `path`: Xử lý đường dẫn tệp.
- `fileURLToPath`: Chuyển URL mô-đun ES thành đường dẫn tệp.

Biến và Hằng

Định nghĩa URL và đường dẫn:

```
1 const DB_SERVER_URL = process.env.DB_SERVER_URL ||
2   "http://localhost:5001";
3
4 const __filename = fileURLToPath(import.meta.url);
5 const __dirname = path.dirname(__filename);
```

- `DB_SERVER_URL`: URL server cơ sở dữ liệu, mặc định `http://localhost:5001`.
- `__filename`, `__dirname`: Đường dẫn tệp hiện tại, hỗ trợ mô-đun ES.

Hàm Kiểm Tra Đầu Vào

Các hàm kiểm tra dữ liệu đầu vào:

```
1 const isValidEmail = (email) => {
2   const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
3   return emailRegex.test(email);
4 };
5
6 const validateRegisterInput = (username, email, password) => {
```



```
7      if (!username || username.length < 3) {
8          return "Username must be at least 3 characters";
9      }
10     if (!email || !isValidEmail(email)) {
11         return "Invalid email";
12     }
13     if (!password || password.length < 6) {
14         return "Password must be at least 6 characters";
15     }
16     return null;
17 };
18
19 const validateLoginInput = (email, password) => {
20     if (!email || !isValidEmail(email)) {
21         return "Invalid email";
22     }
23     if (!password || password.length < 6) {
24         return "Password must be at least 6 characters";
25     }
26     return null;
27 };
28
29 const validateChangePasswordInput = (oldPassword,
30     newPassword) => {
31     if (!oldPassword || oldPassword.length < 6) {
32         return "Password must be at least 6 characters";
33     }
34     if (!newPassword || newPassword.length < 6) {
35         return "Password must be at least 6 characters";
36     }
37     if (oldPassword === newPassword) {
38         return "New password must differ from old password";
39     }
40     return null;
41 };
42
```

- isValidEmail: Kiểm tra định dạng email bằng regex.
- validateRegisterInput: Kiểm tra username (ít nhất 3 ký tự, sử dụng ≥ 3), email (hợp lệ), password (ít nhất 6 ký tự, sử dụng ≥ 6).
- validateLoginInput: Kiểm tra email (hợp lệ), password (ít nhất 6 ký tự, sử dụng ≥ 6).
- validateChangePasswordInput: Kiểm tra oldPassword, newPassword (ít nhất 6 ký tự, sử dụng ≥ 6 , và khác nhau).

Hàm registerUser

Hàm đăng ký người dùng:

```
1 export const registerUser = async (req, res) => {
2   const { username, email, password } = req.body;
3   try {
4     const validationError = validateRegisterInput(username,
5       email, password);
6     if (validationError) {
7       return res.status(400).json({ message:
8         validationError });
9     }
10
11     const response = await
12       axios.get(`${DB_SERVER_URL}/db/users/email/${email}`);
13     if (response.status === 200) {
14       return res.status(400).json({ message: "Email already
15         exists" });
16     }
17
18     const newUser = await
19       axios.post(`${DB_SERVER_URL}/db/users`, { username,
20         email, password });
21     res.status(201).json({ message: "Registration successful"
22       });
23   } catch (err) {
24     if (err.response && err.response.status === 404) {
25       try {
26         await axios.post(`${DB_SERVER_URL}/db/users`, {
27           username, email, password });
28         res.status(201).json({ message: "Registration
29           successful" });
30       } catch (createErr) {
31         res.status(500).json({ message: "Error
32           registering user" });
33       }
34     } else {
35       res.status(500).json({ message: "Error registering
36         user" });
37     }
38   }
39 };
```

- Kiểm tra đầu vào với validateRegisterInput.
- Kiểm tra email tồn tại bằng GET /db/users/email/:email.
- Nếu email đã tồn tại, trả về lỗi 400.
- Nếu email không tồn tại (404), tạo người dùng mới bằng POST /db/users.
- Trả về thông báo thành công (201) hoặc lỗi (500).

Hàm loginUser

Hàm đăng nhập người dùng:

```
1 export const loginUser = async (req, res) => {
2   const { email, password } = req.body;
3   try {
4     const validationError = validateLoginInput(email,
5       password);
6     if (validationError) {
7       console.log("Validation error:", validationError);
8       return res.status(400).json({ message:
9         validationError });
10    }
11
12    console.log("Logging in:", email);
13    const response = await
14      axios.get(`${DB_SERVER_URL}/db/users/email/${email}`);
15    const user = response.data;
16
17    console.log("User found:", user.email);
18    const isMatch = await bcryptCompare(password,
19      user.password);
20    if (!isMatch) {
21      console.log("Incorrect password for:", email);
22      return res.status(401).json({ message: "Invalid login
23        credentials" });
24    }
25
26    console.log("Login successful, generating token...");
27    const token = jwt.sign({ id: user._id },
28      process.env.JWT_SECRET, { expiresIn: "1d" });
29    res.cookie("authToken", token, {
30      httpOnly: true,
31      secure: process.env.NODE_ENV === "production",
32      sameSite: "lax",
33      maxAge: 24 * 60 * 60 * 1000,
34    });
35    res.json({ username: user.username, avatar: user.avatar
36      });
37  } catch (err) {
38    if (err.response && err.response.status === 404) {
39      console.log("User not found:", email);
40      return res.status(401).json({ message: "Invalid login
41        credentials" });
42    }
43    console.error("Login error:", err.message);
44    res.status(500).json({ message: "Login error" });
45  }
46};
```

- Kiểm tra đầu vào với validateLoginInput.

- Tìm người dùng bằng GET /db/users/email/:email.
- So sánh mật khẩu với bcryptCompare (yêu cầu bcryptjs).
- Nếu mật khẩu khớp, tạo JWT và lưu vào cookie authToken.
- Trả về username và avatar, hoặc lỗi nếu thất bại.

Hàm logoutUser

Hàm đăng xuất người dùng:

```
1 export const logoutUser = async (req, res) => {
2     res.clearCookie("authToken");
3     res.json({ message: "Logout successful" });
4 }
```

- Xóa cookie authToken.
- Trả về thông báo thành công.

Hàm updateAvatar

Hàm cập nhật avatar:

```
1 export const updateAvatar = async (req, res) => {
2     try {
3         const userId = req.user.id;
4         console.log("User ID:", userId);
5         const response = await
6             axios.get(`${DB_SERVER_URL}/db/users/${userId}`);
7         const user = response.data;
8
9         if (!req.file) {
10             console.log("No file uploaded");
11             return res.status(400).json({ message: "No file
12                 uploaded" });
13         }
14
15         console.log("File uploaded:", req.file);
16
17         const uploadDir = path.join(__dirname,
18             "../upload/avatars");
19         if (!uploadDir) {
20             console.log("Creating upload/avatars directory");
21             fs.mkdirSync(uploadDir, { recursive: true });
22         }
23
24         if (user.avatar) {
25             const oldAvatarFilename = path.basename(user.avatar);
26             const oldAvatarPath = path.join(uploadDir,
27                 oldAvatarFilename);
28             try {
```

```

25         if (fs.existsSync(oldAvatarPath)) {
26             console.log('Deleting old avatar file:
27                 ${oldAvatarPath}');
28             fs.unlinkSync(oldAvatarPath);
29         } else {
30             console.log('Old avatar file not found:
31                 ${oldAvatarPath}');
32         }
33     } catch (err) {
34         console.error("Error deleting old avatar file:",
35             err.message);
36     }
37
38     const avatarPath = `/avatars/${req.file.filename}`;
39     await axios.put(`${DB_SERVER_URL}/db/users/${userId}`, {
40         avatar: avatarPath });
41     console.log("Updated avatar in database:", avatarPath);
42
43     res.json({ avatar: avatarPath });
44 } catch (err) {
45     console.error("Error updating avatar:", err.message);
46     if (req.file) {
47         const newAvatarPath = path.join(__dirname,
48             "../upload/avatars", req.file.filename);
49         try {
50             if (fs.existsSync(newAvatarPath)) {
51                 console.log('Deleting new file due to error:
52                     ${newAvatarPath}');
53                 fs.unlinkSync(newAvatarPath);
54             }
55         } catch (err) {
56             console.error("Error deleting new file:",
57                 err.message);
58         }
59     }
60     res.status(500).json({ message: err.message || "Error
61         updating avatar" });
62 }
63 };

```

- Lấy `userId` từ `req.user.id` (giả định middleware xác thực).
- Kiểm tra và tạo thư mục `upload/avatars` nếu chưa tồn tại.
- Xóa avatar cũ nếu có.
- Lưu đường dẫn avatar mới và cập nhật trong cơ sở dữ liệu bằng PUT `/db/users/:id`.
- Nếu lỗi, xóa tệp mới tải lên và trả về thông báo lỗi.

Hàm changePassword

Hàm đổi mật khẩu:

```
1 export const changePassword = async (req, res) => {
2   const { oldPassword, newPassword } = req.body;
3   try {
4     const validationError =
5       validateChangePasswordInput(oldPassword, newPassword);
6     if (validationError) {
7       return res.status(400).json({ message:
8         validationError });
9     }
10    const userId = req.user.id;
11    const response = await
12      axios.get(`${DB_SERVER_URL}/db/users/${userId}`);
13    const user = response.data;
14
15    const isMatch = await bcryptCompare(oldPassword,
16      user.password);
17    if (!isMatch) {
18      return res.status(401).json({ message: "Incorrect old
19        password" });
20    }
21
22    await axios.put(`${DB_SERVER_URL}/db/users/${userId}`, {
23      password: newPassword });
24
25    res.json({ message: "Password change successful" });
26  } catch (err) {
27    console.error("Error changing password:", err.message);
28    res.status(500).json({ message: "Error changing password"
29      });
30  }
31};
```

- Kiểm tra đầu vào với validateChangePasswordInput.
- So sánh oldPassword với mật khẩu hiện tại bằng bcryptCompare.
- Cập nhật mật khẩu mới bằng PUT /db/users/:id.
- Trả về thông báo thành công hoặc lỗi.

Hàm bcryptCompare

Hàm so sánh mật khẩu:

```
1 const bcryptCompare = async (plainPassword, hashedPassword)
2   => {
3   const bcrypt = await import("bcryptjs");
```

```
3     return await bcrypt.compare(plainPassword,  
4         hashedPassword);  
};
```

- Tạm thời nhập `bcryptjs` và so sánh mật khẩu, yêu cầu cài đặt `bcryptjs` trên server chính.

Chức Năng Chính

- **Đăng Ký Người Dùng:** Kiểm tra đầu vào, kiểm tra email tồn tại, tạo người dùng mới.
- **Đăng Nhập Người Dùng:** Kiểm tra thông tin đăng nhập, tạo JWT, lưu cookie xác thực.
- **Đăng Xuất Người Dùng:** Xóa cookie xác thực.
- **Cập Nhật Avatar:** Xóa avatar cũ, lưu avatar mới, cập nhật cơ sở dữ liệu.
- **Đổi Mật Khẩu:** Kiểm tra mật khẩu cũ, cập nhật mật khẩu mới.
- **Kiểm Tra Đầu Vào:** Đảm bảo dữ liệu hợp lệ cho đăng ký, đăng nhập, đổi mật khẩu.

4.2 ROUTES

Là nơi định tuyến yêu cầu từ Front-end đến controller, thuộc tầng Controller (MVC) hoặc Application Layer (Layered Architecture). Có chức năng:

- Định nghĩa endpoint API: Xác định các URL (như `/api/users/login`, `/api/devices`) và phương thức HTTP (GET, POST, v.v.).
- Điều phối yêu cầu: Chuyển yêu cầu từ Front-end đến các hàm xử lý trong controllers/ (như `loginUser`, `getDevices`).
- Áp dụng middleware: Thêm các middleware (như `authenticateToken`, `upload`) trước khi gọi controller.

4.2.1 deviceRoute

Xác định endpoint như GET `/api/devices` để lấy danh sách thiết bị, gọi hàm `getDevices` trong `deviceController.js` và sử dụng `authenticateToken` để kiểm tra token trước khi xử lý

Import

Các thư viện và thành phần được nhập:

```
1 import express from "express";  
2 import { getDevices } from  
   "../controllers/deviceController.js";  
3 import { authenticateToken } from "../middleware/auth.js";
```

- **express**: Framework để tạo router và xử lý yêu cầu HTTP.
- **getDevices**: Hàm xử lý từ **deviceController** để lấy danh sách thiết bị.
- **authenticateToken**: Middleware xác thực token từ **auth**.

Định Nghĩa Router

Tạo một router Express:

```
1 const router = express.Router();
```

- **router**: Một instance của **express.Router** để định nghĩa các tuyến API.

Tuyến API

Định nghĩa tuyến GET:

```
1 router.get("/", authenticateToken, getDevices);
```

- **router.get("/")**: Định nghĩa tuyến GET cho đường dẫn gốc (/).
- **authenticateToken**: Middleware kiểm tra token xác thực trước khi xử lý yêu cầu.
- **getDevices**: Hàm xử lý trả về danh sách thiết bị.

Export

Xuất router để sử dụng trong ứng dụng:

```
1 export default router;
```

- Xuất **router** để tích hợp vào ứng dụng Express chính.

Chức Năng Chính

Tệp định tuyến cung cấp một tuyến API để lấy danh sách thiết bị:

- **Định Tuyến Yêu Cầu**: Xử lý yêu cầu GET tại / để trả về danh sách thiết bị.
- **Xác Thực Yêu Cầu**: Sử dụng middleware **authenticateToken** để đảm bảo chỉ người dùng đã đăng nhập mới truy cập được.
- **Tích Hợp Hàm Xử Lý**: Gọi **getDevices** để lấy dữ liệu từ server cơ sở dữ liệu.
- **Thiết Kế Mô-đun**: Sử dụng **express.Router** để tổ chức tuyến API một cách độc lập, dễ bảo trì.

4.2.2 userRoute

Tệp định nghĩa route cho người dùng: Xác định các endpoint như POST /api/users/login, POST /api/users/register, POST /api/users/change-password... Gọi các hàm như **loginUser**, **registerUser**, **changePassword** trong **usersController.js** và sử dụng **authenticateToken** (xác thực token) hoặc **upload** (xử lý file upload) cho một số route.

Import

Các thư viện và thành phần được nhập:

```
1 import express from "express";
2 import { registerUser, loginUser, logoutUser, updateAvatar,
   changePassword } from "../controllers/usersController.js";
3 import { authenticateToken } from "../middleware/auth.js";
4 import upload from "../middleware/upload.js";
```

- `express`: Framework để tạo router và xử lý yêu cầu HTTP.
- `registerUser, loginUser, logoutUser, updateAvatar, changePassword`: Các hàm xử lý từ `usersController`.
- `authenticateToken`: Middleware xác thực token từ `auth`.
- `upload`: Middleware xử lý tải tệp từ `upload`.

Định Nghĩa Router

Tạo một router Express:

```
1 const router = express.Router();
```

- `router`: Một instance của `express.Router` để định nghĩa các tuyến API.

Tuyến API

Định nghĩa các tuyến API cho quản lý người dùng:

```
1 router.post("/register", registerUser);
2 router.post("/login", loginUser);
3 router.post("/logout", logoutUser);
4 router.post("/update-avatar", authenticateToken,
   upload.single("avatar"), updateAvatar);
5 router.post("/change-password", authenticateToken,
   changePassword);
6
7 router.get('/verify-token', authenticateToken, (req, res) => {
8   res.json({ valid: true });
9 });
```

- `router.post("/register", registerUser)`: Tuyến POST để đăng ký người dùng mới.
- `router.post("/login", loginUser)`: Tuyến POST để đăng nhập người dùng.
- `router.post("/logout", logoutUser)`: Tuyến POST để đăng xuất người dùng.
- `router.post("/update-avatar", authenticateToken, upload.single("avatar"), updateAvatar)`: Tuyến POST để cập nhật avatar, yêu cầu xác thực và tải tệp.

- `router.post("/change-password", authenticateToken, changePassword)`: Tuyến POST để đổi mật khẩu, yêu cầu xác thực.
- `router.get("/verify-token", authenticateToken, ...)`: Tuyến GET để xác minh token, trả về `{ valid: true }` nếu token hợp lệ.

Export

Xuất router để sử dụng trong ứng dụng:

```
export default router;
```

- Xuất `router` để tích hợp vào ứng dụng Express chính.

Chức Năng Chính

Tệp định tuyến cung cấp các tuyến API để quản lý người dùng:

- **Định Tuyến Yêu Cầu Người Dùng**: Xử lý các yêu cầu POST cho đăng ký, đăng nhập, đăng xuất, cập nhật avatar, và đổi mật khẩu; yêu cầu GET để xác minh token.
- **Bảo Mật Yêu Cầu**: Sử dụng middleware `authenticateToken` để bảo vệ các tuyến `update-avatar`, `change-password`, và `verify-token`.
- **Xử Lý Tệp Tải Lên**: Sử dụng middleware `upload.single("avatar")` để xử lý tệp avatar trong tuyến `update-avatar`.
- **Tích Hợp Hàm Xử Lý**: Gọi các hàm từ `usersController` để thực hiện logic quản lý người dùng.
- **Thiết Kế Mô-đun**: Sử dụng `express.Router` để tổ chức các tuyến API một cách độc lập, dễ bảo trì.

4.3 MIDDLEWARE

Là nơi chứa các middleware dùng để:

- Xử lý trung gian: Thực hiện các tác vụ trước khi yêu cầu đến controller (như xác thực, xử lý file upload).
- Kiểm soát luồng yêu cầu: Kiểm tra, biến đổi hoặc chặn yêu cầu dựa trên điều kiện.

4.3.1 auth

Middleware xác thực token người dùng, kiểm tra tính hợp lệ của token trong cookie và giải mã thông tin người dùng.

Import

Thư viện được nhập:

```
1 import jwt from "jsonwebtoken";
```

- `jwt`: Thư viện để xác minh và giải mã JSON Web Token.

Hàm `authenticateToken`

Middleware xác minh token:

```
1 export const authenticateToken = (req, res, next) => {  
2   const token = req.cookies.authToken;  
3   if (!token) {  
4     return res.status(401).json({ message: "Not logged  
5       in" });  
6   }  
7   jwt.verify(token, process.env.JWT_SECRET, (err, user) => {  
8     if (err) {  
9       return res.status(403).json({ message: "Invalid  
10         token" });  
11     }  
12     req.user = user;  
13     next();  
14   });  
};
```

- `req, res, next`: Tham số chuẩn của middleware Express.
- Lấy token từ cookie `authToken`.
- Nếu không có token, trả về lỗi 401 với thông báo "Not logged in".
- Sử dụng `jwt.verify` để xác minh token với `JWT_SECRET`.
- Nếu token không hợp lệ, trả về lỗi 403 với thông báo "Invalid token".
- Nếu hợp lệ, gán thông tin người dùng vào `req.user` và gọi `next()` để chuyển tiếp yêu cầu.

Export

Xuất middleware để sử dụng trong các tuyến API:

```
1 export default authenticateToken;
```

- Xuất `authenticateToken` để tích hợp vào các tuyến API của ứng dụng Express.

Chức Năng Chính

Middleware `authenticateToken` cung cấp xác thực cho các yêu cầu API:

- **Xác Minh Token:** Kiểm tra sự tồn tại và tính hợp lệ của JWT trong cookie `authToken`.
- **Bảo Vệ Tuyến API:** Từ chối các yêu cầu không có token (401) hoặc token không hợp lệ (403).
- **Gán Thông Tin Người Dùng:** Thêm thông tin người dùng vào `req.user` nếu token hợp lệ.
- **Tích Hợp Express:** Sử dụng cơ chế middleware để dễ dàng áp dụng cho các tuyến API cần xác thực.

4.3.2 upload

Middleware xử lý tải tệp, sử dụng thư viện `multer` để quản lý việc tải lên tệp avatar của người dùng.

Import

Các thư viện được nhập:

```
1 import multer from "multer";
2 import path from "path";
3 import { fileURLToPath } from "url";
4 import fs from "fs";
```

- `multer`: Thư viện để xử lý tải lên tệp trong Express.
- `path`: Xử lý đường dẫn tệp.
- `fileURLToPath`: Chuyển URL mô-đun ES thành đường dẫn tệp.
- `fs`: Quản lý hệ thống tệp (tạo thư mục, kiểm tra tệp).

Biến Đường Dẫn

Định nghĩa đường dẫn tệp hiện tại:

```
1 const __filename = fileURLToPath(import.meta.url);
2 const __dirname = path.dirname(__filename);
```

- `__filename, __dirname`: Đường dẫn tệp hiện tại, hỗ trợ mô-đun ES.

Cấu Hình Lưu Trữ

Cấu hình lưu trữ cho multer:

```
1  const storage = multer.diskStorage({
2    destination: (req, file, cb) => {
3      const uploadDir = path.join(__dirname,
4        "../upload/avatars");
5      if (!fs.existsSync(uploadDir)) {
6        fs.mkdirSync(uploadDir, { recursive: true });
7      }
8      cb(null, uploadDir);
9    },
10   filename: (req, file, cb) => {
11     const uniqueSuffix = Date.now() + "-" +
12       Math.round(Math.random() * 1e9);
13     cb(null, uniqueSuffix +
14       path.extname(file.originalname));
15   },
16 });
```

- **destination:** Tạo thư mục upload/avatars nếu chưa tồn tại, đặt đích lưu trữ là thư mục này.
- **filename:** Tạo tên tệp duy nhất bằng thời gian hiện tại và số ngẫu nhiên, giữ nguyên phần mở rộng của tệp gốc.

Cấu Hình Multer

Tạo middleware upload với multer:

```
1  const upload = multer({
2    storage,
3    limits: { fileSize: 5 * 1024 * 1024 },
4    fileFilter: (req, file, cb) => {
5      const filetypes = /jpeg|jpg|png/;
6      const extname =
7        filetypes.test(path.extname(file.originalname).
8          toLowerCase());
9      const mimetype = filetypes.test(file.mimetype);
10     if (extname && mimetype) {
11       return cb(null, true);
12     }
13     cb(new Error("Only JPEG or PNG files are accepted"));
14   },
15 });
```

- **storage:** Sử dụng cấu hình lưu trữ đã định nghĩa.
- **limits:** Giới hạn kích thước tệp tối đa 5MB (5 * 1024 * 1024 byte).
- **fileFilter:** Chỉ chấp nhận tệp JPEG hoặc PNG dựa trên phần mở rộng và loại MIME, trả về lỗi nếu không hợp lệ.

Export

Xuất middleware để sử dụng trong các tuyến API:

```
1 export default upload;
```

- Xuất upload để tích hợp vào các tuyến API Express, ví dụ: tuyến update-avatar.

Chức Năng Chính

Middleware upload cung cấp xử lý tải lên tệp hình ảnh:

- **Xử Lý Tệp Tải Lên:** Sử dụng multer để nhận và lưu tệp hình ảnh vào thư mục upload/avatars.
- **Tạo Tên Tệp Duy Nhất:** Tạo tên tệp dựa trên thời gian và số ngẫu nhiên để tránh xung đột.
- **Giới Hạn và Lọc Tệp:** Giới hạn kích thước tệp 5MB, chỉ chấp nhận định dạng JPEG hoặc PNG.
- **Quản Lý Thư Mục:** Tự động tạo thư mục upload/avatars nếu chưa tồn tại.
- **Tích Hợp Express:** Middleware dễ dàng áp dụng cho các tuyến API cần tải lên tệp.

4.4 SERVER

Là Application Layer, xử lý logic và điều phối giữa Front-end và dbServer.

4.4.1 Import

Các thư viện và mô-đun được nhập:

```
1 import express from "express";  
2 import dotenv from "dotenv";  
3 import deviceRoutes from "../routes/deviceRoutes.js";  
4 import userRoutes from "../routes/usersRoute.js";  
5 import { createRequire } from "module";  
6 import http from "http";  
7 import cookieParser from "cookie-parser";  
8 import path from "path";  
9 import { fileURLToPath } from "url";
```

- **express:** Framework để tạo server và xử lý yêu cầu HTTP.
- **dotenv:** Tải biến môi trường từ tệp .env.
- **deviceRoutes, userRoutes:** Các tuyến API cho thiết bị và người dùng.
- **createRequire:** Hỗ trợ nhập CommonJS trong mô-đun ES.

- http: Tạo server HTTP cho Socket.IO.
- cookieParser: Xử lý cookie trong yêu cầu.
- path, fileURLToPath: Xử lý đường dẫn tệp.

4.4.2 Biến Đường Dẫn

Định nghĩa đường dẫn tệp hiện tại:

```
1 const __filename = fileURLToPath(import.meta.url);  
2 const __dirname = path.dirname(__filename);
```

- __filename, __dirname: Đường dẫn tệp hiện tại, hỗ trợ mô-đun ES.

4.4.3 Cấu Hình Môi Trường

Tải biến môi trường từ tệp .env:

```
1 dotenv.config({ path: path.join(__dirname, "../.env") });
```

- dotenv.config: Tải các biến môi trường từ tệp .env trong thư mục gốc.

4.4.4 Khởi Tạo Server

Tạo ứng dụng Express và server HTTP:

```
1 const app = express();  
2 const server = http.createServer(app);
```

- app: Instance của express để xử lý yêu cầu HTTP.
- server: Server HTTP để hỗ trợ Socket.IO.

4.4.5 Cấu Hình Socket.IO

Khởi tạo Socket.IO với CORS:

```
1 const require = createRequire(import.meta.url);  
2 const socketIo = require("socket.io");  
3  
4 const allowedOrigins = [  
5     "http://localhost:5173",  
6     "http://192.168.1.10:5173",  
7     "http://localhost:3000",  
8 ];  
9  
10 const io = socketIo(server, {  
11     cors: {  
12         origin: allowedOrigins,  
13         methods: ["GET", "POST", "OPTIONS"],
```

```
14         credentials: true,
15     },
16     pingTimeout: 60000,
17     pingInterval: 25000,
18 });
```

- Nhập `socket.io` qua CommonJS vì mô-đun ES không hỗ trợ trực tiếp.
- `allowedOrigins`: Danh sách các nguồn được phép cho CORS.
- `io`: Instance `Socket.IO` với cấu hình CORS, thời gian chờ 60 giây, và kiểm tra kết nối mỗi 25 giây.

4.4.6 Middleware

Cấu hình middleware cho ứng dụng:

```
1  app.use((req, res, next) => {
2      const origin = req.headers.origin;
3      if (allowedOrigins.includes(origin)) {
4          res.setHeader("Access-Control-Allow-Origin", origin);
5          res.setHeader("Access-Control-Allow-Credentials",
6              "true");
7          res.setHeader("Access-Control-Allow-Methods", "GET,
8              POST, OPTIONS");
9          res.setHeader("Access-Control-Allow-Headers",
10              "Content-Type, Authorization");
11      }
12      next();
13  });
14
15  app.use("/avatars", express.static(path.join(__dirname,
16      "upload/avatars")));
17
18  app.use(cookieParser());
19  app.use(express.json());
20
21  app.use((req, res, next) => {
22      console.log(`Received request: ${req.method} ${req.url}`);
23      next();
24  });
```

- CORS middleware: Cho phép các nguồn trong `allowedOrigins` với phương thức GET, POST, OPTIONS và hỗ trợ cookie.
- Phục vụ tệp tĩnh: Phục vụ tệp từ thư mục `upload/avatars` tại tuyến `/avatars`.
- `cookieParser`: Xử lý cookie trong yêu cầu.
- `express.json`: Phân tích dữ liệu JSON từ yêu cầu.
- Logging middleware: Ghi log phương thức và URL của mỗi yêu cầu.

4.4.7 Định Tuyến

Tích hợp các tuyến API:

```
1 app.use("/api/users", userRoutes);  
2 app.use("/api/devices", deviceRoutes);
```

- `userRoutes`: Xử lý các API liên quan đến người dùng tại `/api/users`.
- `deviceRoutes`: Xử lý các API liên quan đến thiết bị tại `/api/devices`.

4.4.8 Tuyến Gốc

Định nghĩa tuyến gốc:

```
1 app.get("/", (req, res) => {  
2   res.send("Backend is running");  
3 });
```

- Trả về thông báo "Backend is running" cho yêu cầu GET tại `/`. Lưu ý: Emoji đã được giữ nguyên vì báo cáo không yêu cầu thay đổi chuỗi này.

4.4.9 Xử Lý Socket.IO

Xử lý các sự kiện Socket.IO:

```
1 io.on("connection", (socket) => {  
2   console.log("Client connected:", socket.id,  
3     socket.handshake.query);  
4  
5   socket.on("message", (msg) => {  
6     console.log("Received message:", msg);  
7   });  
8  
9   socket.on("Value", (data) => {  
10    console.log("Received value:", data);  
11  });  
12  
13  socket.on("disconnect", (reason) => {  
14    console.log("Client disconnected:", socket.id,  
15      "Reason:", reason);  
16  });  
17 });
```

- Lắng nghe sự kiện `connection`: Ghi log khi client kết nối.
- Lắng nghe `message` và `Value`: Ghi log dữ liệu nhận được.
- Lắng nghe `disconnect`: Ghi log khi client ngắt kết nối và lý do.

4.4.10 Khởi Động Server

Khởi động server trên cổng được chỉ định:

```
1  const PORT = process.env.PORT || 5000;  
2  server.listen(PORT, "0.0.0.0", () => {  
3      console.log('Server running at http://0.0.0.0:${PORT}');  
4  });
```

- **PORT:** Lấy từ biến môi trường hoặc mặc định 5000.
- **server.listen:** Khởi động server trên tất cả các giao diện mạng (0.0.0.0).

4.4.11 Chức Năng Chính

Tệp khởi tạo ứng dụng Express cung cấp nền tảng cho server:

- **Xử Lý Yêu Cầu HTTP:** Tích hợp Express để xử lý các yêu cầu qua các tuyến API `userRoutes` và `deviceRoutes`.
- **Giao Tiếp Thời Gian Thực:** Sử dụng Socket.IO để xử lý kết nối, tin nhắn, và sự kiện giá trị từ client.
- **Phục Vụ Tập Tĩnh:** Phục vụ hình ảnh avatar từ thư mục `upload/avatars`.
- **Cấu Hình CORS:** Cho phép các nguồn được chỉ định truy cập API với hỗ trợ cookie.
- **Middleware:** Áp dụng `cookieParser`, `express.json`, và `logging` để xử lý yêu cầu hiệu quả.

Chương 5

DATABASE SERVER

5.1 CONFIG

5.1.1 db

Sử dụng mongoose để kết nối đến MongoDB thông qua MONGO_URI từ tệp .env. Báo lỗi nếu kết nối thất bại và thoát process.

Import

Thư viện được nhập:

```
1 import mongoose from 'mongoose';
```

- mongoose: Thư viện để kết nối và tương tác với cơ sở dữ liệu MongoDB.

Hàm connectDB

Hàm thiết lập kết nối MongoDB:

```
1  const connectDB = async () => {  
2    try {  
3      const conn = await  
4        mongoose.connect(process.env.MONGO_URI, {  
5      });  
6      console.log('MongoDB connected:  
7        ${conn.connection.host}');  
8    } catch (error) {  
9      console.error('MongoDB connection error:  
10        ${error.message}');  
11      process.exit(1);  
12    }  
13  };
```

- Lấy URI kết nối từ biến môi trường MONGO_URI.
- Sử dụng mongoose.connect để thiết lập kết nối với MongoDB.

- Nếu thành công, ghi log địa chỉ host của kết nối.
- Nếu thất bại, ghi log lỗi và thoát ứng dụng với mã lỗi 1.

Export

Xuất hàm để sử dụng trong ứng dụng:

```
1 export default connectDB;
```

- Xuất connectDB để gọi khi khởi động ứng dụng Node.js.

Chức Năng Chính

Hàm connectDB cung cấp kết nối cơ sở dữ liệu:

- **Kết Nối MongoDB:** Sử dụng mongoose để thiết lập kết nối với cơ sở dữ liệu MongoDB dựa trên MONGO_URI.
- **Xử Lý Lỗi:** Ghi log và thoát ứng dụng nếu kết nối thất bại, đảm bảo ứng dụng không chạy khi thiếu cơ sở dữ liệu.
- **Ghi Log Kết Nối:** Thông báo khi kết nối thành công với địa chỉ host.
- **Tích Hợp Ứng Dụng:** Hàm được xuất để dễ dàng gọi trong tệp khởi tạo server Express.

5.2 MODEL

Chứa các schema và logic dữ liệu, thuộc tầng Data Layer (Layered Architecture) hoặc Model (MVC).

5.2.1 Device

Schema cho thiết bị, định nghĩa cấu trúc dữ liệu và các phương thức liên quan đến thiết bị.

Import

Thư viện được nhập:

```
1 import mongoose from 'mongoose';
```

- mongoose: Thư viện để định nghĩa schema và tương tác với MongoDB.

Định Nghĩa Schema

Tạo schema cho tài liệu thiết bị:

```
1  const deviceSchema = new mongoose.Schema({
2    name: { type: String, required: true },
3    status: { type: String, enum: ['on', 'off'], default: 'off' },
4    updatedAt: { type: Date, default: Date.now }
5  });
```

- **deviceSchema**: Schema xác định cấu trúc tài liệu thiết bị với các trường:
 - **name**: Chuỗi, bắt buộc (**required: true**).
 - **status**: Chuỗi, chỉ nhận giá trị 'on' hoặc 'off', mặc định 'off'.
 - **updatedAt**: Ngày giờ, mặc định là thời gian hiện tại (**Date.now**).

Định Nghĩa Model

Tạo model từ schema:

```
1  const Device = mongoose.model('Device', deviceSchema,
    'GateWay');
```

- **Device**: Model được tạo từ **deviceSchema**, liên kết với bộ sưu tập **GateWay** trong MongoDB.
- Tên model **Device** dùng để truy vấn và thao tác với tài liệu trong bộ sưu tập.

Export

Xuất model để sử dụng trong ứng dụng:

```
1  export default Device;
```

- Xuất **Device** để sử dụng trong các module khác, ví dụ: trong các hàm API để truy vấn hoặc cập nhật thiết bị.

Chức Năng Chính

Schema và model **Device** cung cấp mô hình dữ liệu cho thiết bị:

- **Định Nghĩa Cấu Trúc Dữ Liệu**: Xác định các trường **name**, **status**, và **updatedAt** với ràng buộc và giá trị mặc định.
- **Tương Tác Với MongoDB**: Cung cấp model **Device** để thực hiện các thao tác như tạo, đọc, cập nhật, xóa tài liệu trong bộ sưu tập **GateWay**.
- **Ràng Buộc Dữ Liệu**: Đảm bảo **name** bắt buộc, **status** chỉ nhận 'on' hoặc 'off', và **updatedAt** tự động cập nhật.
- **Tích Hợp Ứng Dụng**: Model dễ dàng sử dụng trong các API để quản lý thiết bị.

5.2.2 User

Schema cho người dùng, định nghĩa cấu trúc dữ liệu và các phương thức liên quan đến người dùng.

Import

Các thư viện được nhập:

```
1 import mongoose from 'mongoose';  
2 import bcrypt from 'bcryptjs';
```

- **mongoose**: Thư viện để định nghĩa schema và tương tác với MongoDB.
- **bcryptjs**: Thư viện để mã hóa và so sánh mật khẩu.

Định Nghĩa Schema

Tạo schema cho tài liệu người dùng:

```
1 const userSchema = new mongoose.Schema({  
2   username: { type: String, required: true, unique: true },  
3   email:    { type: String, required: true, unique: true },  
4   password: { type: String, required: true },  
5   avatar:   { type: String, default: null },  
6 });
```

- **userSchema**: Schema xác định cấu trúc tài liệu người dùng với các trường:
 - **username**: Chuỗi, bắt buộc, duy nhất.
 - **email**: Chuỗi, bắt buộc, duy nhất.
 - **password**: Chuỗi, bắt buộc.
 - **avatar**: Chuỗi, mặc định null.

Mã Hóa Mật Khẩu

Middleware mã hóa mật khẩu trước khi lưu:

```
1 userSchema.pre("save", async function (next) {  
2   if (!this.isModified("password")) return next();  
3   this.password = await bcrypt.hash(this.password, 10);  
4   next();  
5 });
```

- **pre("save")**: Middleware chạy trước khi lưu tài liệu.
- Kiểm tra nếu **password** không thay đổi, bỏ qua bằng **next()**.
- Mã hóa **password** bằng **bcrypt.hash** với độ mạnh 10, lưu vào **this.password**.

Phương Thức So Sánh Mật Khẩu

Thêm phương thức để so sánh mật khẩu:

```
1   userSchema.methods.matchPassword = async function  
    (enteredPassword) {  
2       return await bcrypt.compare(enteredPassword,  
        this.password);  
3   };
```

- **matchPassword:** Phương thức so sánh mật khẩu nhập vào với mật khẩu đã mã hóa bằng `bcrypt.compare`.
- Trả về `true` nếu khớp, `false` nếu không.

Định Nghĩa Model

Tạo model từ schema:

```
1   const User = mongoose.model("User", userSchema);
```

- **User:** Model được tạo từ `userSchema`, liên kết với bộ sưu tập `users` (mặc định) trong MongoDB.
- Tên model `User` dùng để truy vấn và thao tác với tài liệu người dùng.

Chức Năng Chính

Schema và model `User` cung cấp mô hình dữ liệu cho người dùng:

- **Định Nghĩa Cấu Trúc Dữ Liệu:** Xác định các trường `username`, `email`, `password`, và `avatar` với ràng buộc và giá trị mặc định.
- **Mã Hóa Mật Khẩu:** Tự động mã hóa `password` trước khi lưu bằng `bcrypt`.
- **So Sánh Mật Khẩu:** Cung cấp phương thức `matchPassword` để xác minh mật khẩu khi đăng nhập.
- **Tương Tác Với MongoDB:** Model `User` hỗ trợ tạo, đọc, cập nhật, xóa tài liệu trong bộ sưu tập `users`.
- **Tích Hợp Ứng Dụng:** Model dễ dàng sử dụng trong các API để quản lý người dùng.

5.3 SEED

Tạo dữ liệu mẫu cho cơ sở dữ liệu, giúp kiểm tra và phát triển ứng dụng mà không cần nhập liệu thủ công.

5.3.1 userSeed

Tạo dữ liệu mẫu cho người dùng, bao gồm tên người dùng, email, mật khẩu và avatar.

Import

Các thư viện và mô-đun được nhập:

```
1 import mongoose from 'mongoose';
2 import dotenv from 'dotenv';
3 import bcrypt from 'bcryptjs';
4 import User from '../models/User.js';
5 import path from 'path';
6 import { fileURLToPath } from 'url';
```

- `mongoose`: Thư viện để kết nối và tương tác với MongoDB.
- `dotenv`: Tải biến môi trường từ tệp `.env`.
- `bcryptjs`: Thư viện để mã hóa mật khẩu (dùng trong model `User`).
- `User`: Model người dùng từ `../models/User.js`.
- `path`, `fileURLToPath`: Xử lý đường dẫn tệp.

Biến Đường Dẫn

Định nghĩa đường dẫn tệp hiện tại:

```
1 const __filename = fileURLToPath(import.meta.url);
2 const __dirname = path.dirname(__filename);
```

- `__filename`, `__dirname`: Đường dẫn tệp hiện tại, hỗ trợ mô-đun ES.

Cấu Hình Môi Trường

Tải biến môi trường từ tệp `.env`:

```
1 dotenv.config({ path: path.join(__dirname, '../.env') });
```

- `dotenv.config`: Tải các biến môi trường từ tệp `.env` trong thư mục gốc, cách thư mục hiện tại hai cấp.

Kết Nối MongoDB

Thiết lập kết nối với MongoDB:

```
1 await mongoose.connect(process.env.MONGO_URI);
```

- Sử dụng `mongoose.connect` để kết nối với MongoDB, lấy URI từ biến môi trường `MONGO_URI`.

Khởi Tạo Dữ Liệu

Xóa dữ liệu cũ và tạo người dùng mẫu:

```
1  await User.deleteMany();
2
3  await User.create({
4      username: 'Admin',
5      email: 'du.vohuudu@gmail.com',
6      password: '123456',
7      avatar: null,
8  });
9
10 console.log('Seeded user successfully!');
11 process.exit();
```

- `User.deleteMany()`: Xóa tất cả tài liệu trong bộ sưu tập `users`.
- `User.create`: Tạo người dùng mẫu với thông tin `username`, `email`, `password`, và `avatar`.
- Ghi log thông báo thành công và thoát ứng dụng bằng `process.exit()`.

Chức Năng Chính

Tập khởi tạo dữ liệu cung cấp cơ chế thiết lập dữ liệu ban đầu:

- **Kết Nối MongoDB**: Thiết lập kết nối với cơ sở dữ liệu MongoDB bằng `mongoose` và `MONGO_URI`.
- **Xóa Dữ Liệu Cũ**: Xóa toàn bộ tài liệu người dùng hiện có trong bộ sưu tập `users`.
- **Tạo Người Dùng Mẫu**: Thêm một người dùng `Admin` với thông tin mẫu để khởi tạo dữ liệu.
- **Quản Lý Môi Trường**: Tải biến môi trường từ tệp `.env` để cấu hình kết nối.
- **Ghi Log và Thoát**: Thông báo khi hoàn tất và thoát ứng dụng.

5.4 SERVER

Data Layer, xử lý dữ liệu và cung cấp API cho `mainServer`. Kết nối MongoDB, định nghĩa schema.

5.4.1 Import

Các thư viện và mô-đun được nhập:

```
1  import express from "express";
2  import dotenv from "dotenv";
3  import connectDB from "../config/db.js";
4  import User from "../models/User.js";
```

```
5 import Device from "./models/Device.js";  
6 import path from "path";  
7 import { fileURLToPath } from "url";
```

- **express**: Framework để tạo server và xử lý yêu cầu HTTP.
- **dotenv**: Tải biến môi trường từ tệp `.env`.
- **connectDB**: Hàm kết nối MongoDB từ `./config/db.js`.
- **User, Device**: Model MongoDB cho người dùng và thiết bị.
- **path, fileURLToPath**: Xử lý đường dẫn tệp.

5.4.2 Biến Đường Dẫn

Định nghĩa đường dẫn tệp hiện tại:

```
1 // Define __filename and __dirname  
2 const __filename = fileURLToPath(import.meta.url);  
3 const __dirname = path.dirname(__filename);
```

- **__filename, __dirname**: Đường dẫn tệp hiện tại, hỗ trợ mô-đun ES.

5.4.3 Cấu Hình Môi Trường

Tải biến môi trường từ tệp `.env`:

```
1 // Read .env file from parent directory (Back-end)  
2 dotenv.config({ path: path.join(__dirname, "../.env") });
```

- **dotenv.config**: Tải các biến môi trường từ tệp `.env` trong thư mục gốc, cách thư mục hiện tại một cấp.

5.4.4 Kết Nối MongoDB

Gọi hàm kết nối cơ sở dữ liệu:

```
1 // Call connectDB function  
2 connectDB();
```

- **connectDB()**: Kết nối ứng dụng với MongoDB bằng cách gọi hàm từ `./config/db.js`.

5.4.5 Khởi Tạo Server

Tạo và cấu hình server Express:

```
1 const app = express();  
2 app.use(express.json());
```

- **app**: Instance của **express** để xử lý yêu cầu HTTP.
- **express.json**: Middleware phân tích dữ liệu JSON từ yêu cầu.

5.4.6 API Lấy Danh Sách Thiết Bị

Định nghĩa API GET để lấy tất cả thiết bị:

```
1 app.get("/db/devices", async (req, res) => {
2   try {
3     const devices = await Device.find();
4     res.json(devices);
5   } catch (error) {
6     res.status(500).json({ message: "Error fetching
7       device list" });
8   }
9 });
```

- Tuyến /db/devices: Truy vấn tất cả tài liệu thiết bị bằng `Device.find()`.
- Trả về danh sách thiết bị dưới dạng JSON hoặc lỗi 500 nếu thất bại.

5.4.7 API Tìm Người Dùng Theo Email

Định nghĩa API GET để tìm người dùng bằng email:

```
1 app.get("/db/users/email/:email", async (req, res) => {
2   try {
3     const user = await User.findOne({ email:
4       req.params.email });
5     if (!user) {
6       return res.status(404).json({ message: "User not
7         found" });
8     }
9     res.json(user);
10  } catch (error) {
11    res.status(500).json({ message: "Error fetching user"
12      });
13  }
14 });
```

- Tuyến /db/users/email/:email: Tìm người dùng bằng `User.findOne` với email.
- Trả về thông tin người dùng hoặc lỗi 404 nếu không tìm thấy, lỗi 500 nếu thất bại.

5.4.8 API Tìm Người Dùng Theo ID

Định nghĩa API GET để tìm người dùng bằng ID:

```
1 app.get("/db/users/:id", async (req, res) => {
2   try {
3     const user = await User.findById(req.params.id);
4     if (!user) {
5       return res.status(404).json({ message: "User not
6         found" });
7     }
8   }
9 });
```

```
7         res.json(user);
8     } catch (error) {
9         res.status(500).json({ message: "Error fetching user"
10            });
11     }
12 });
```

- Tuyến `/db/users/:id`: Tìm người dùng bằng `User.findById` với ID.
- Trả về thông tin người dùng hoặc lỗi 404 nếu không tìm thấy, lỗi 500 nếu thất bại.

5.4.9 API Tạo Người Dùng Mới

Định nghĩa API POST để tạo người dùng mới:

```
1 app.post("/db/users", async (req, res) => {
2     try {
3         const user = await User.create(req.body);
4         res.status(201).json(user);
5     } catch (error) {
6         res.status(500).json({ message: "Error creating user"
7            });
8     }
9 });
```

- Tuyến `/db/users`: Tạo người dùng mới bằng `User.create` với dữ liệu từ `req.body`.
- Trả về thông tin người dùng mới với mã 201 hoặc lỗi 500 nếu thất bại.

5.4.10 API Cập Nhật Người Dùng

Định nghĩa API PUT để cập nhật người dùng:

```
1 app.put("/db/users/:id", async (req, res) => {
2     try {
3         const user = await
4             User.findByIdAndUpdate(req.params.id, req.body, {
5                 new: true });
6         if (!user) {
7             return res.status(404).json({ message: "User not
8                found" });
9         }
10        res.json(user);
11    } catch (error) {
12        res.status(500).json({ message: "Error updating user"
13            });
14    }
15 });
```

- Tuyến `/db/users/:id`: Cập nhật người dùng bằng `User.findByIdAndUpdate` với ID và dữ liệu từ `req.body`.

- Trả về thông tin người dùng đã cập nhật hoặc lỗi 404 nếu không tìm thấy, lỗi 500 nếu thất bại.

5.4.11 Khởi Động Server

Khởi động server trên cổng được chỉ định:

```
1  const DB_PORT = process.env.DB_PORT || 5001;  
2  app.listen(DB_PORT, "0.0.0.0", () => {  
3    console.log('Database Server running at  
      http://0.0.0.0:${DB_PORT}');  
4  });
```

- **DB_PORT**: Lấy từ biến môi trường hoặc mặc định 5001.
- **app.listen**: Khởi động server trên tất cả các giao diện mạng (0.0.0.0) tại cổng DB_PORT.

5.4.12 Chức Năng Chính

Tập khởi tạo server cơ sở dữ liệu cung cấp các API để quản lý dữ liệu:

- **Kết Nối MongoDB**: Gọi `connectDB` để thiết lập kết nối với cơ sở dữ liệu MongoDB.
- **API Quản Lý Thiết Bị**: Cung cấp tuyến `/db/devices` để lấy tất cả thiết bị.
- **API Quản Lý Người Dùng**: Cung cấp các tuyến để tìm người dùng theo email hoặc ID, tạo và cập nhật người dùng.
- **Xử Lý Yêu Cầu JSON**: Sử dụng `express.json` để phân tích dữ liệu JSON từ yêu cầu.
- **Khởi Động Server**: Chạy server trên cổng DB_PORT để xử lý các yêu cầu API.

Appendices

Phụ lục A

SOURCE CODE

GITHUB: https://github.com/duvohuu/UID_IOT