ДОДАТОК А

Графічний матеріал кваліфікаційної роботи

Міністерство освіти та науки України
Харківський національний університет радіоелектроніки
Кафедра електронних обчислювальних машин

Кваліфікаційна робота
Перший (бакалаврський) рівень

«Веб-сайт для бронювання готельних номерів»

Виконав:
ст. гр. КІУКІ-21-4
Безродний Є.С.

Керівник:
ст. викл. каф. ЕОМ
Ярошевич Р.О.

Харків 2025

МЕТА ТА АКТУАЛЬНІСТЬ РОБОТИ:

Мета кваліфікаційної роботи полягає в розробці зручного веб-сайту для пошуку та бронювання готельних номерів.

Актуальність сайту полягає в практичній необхідності в пошуку тимчасового житла для людей, які подорожують країною або шукають прихисток на недовготривалі терміни.

ЗАДАЧІ ДЛЯ ВИКОНАННЯ:

- ☐ методи аутентифікації користувачів;
- ☐ списки доступних номерів;
- ☐ особистий кабінет користувача;
- ☐ бронювання номерів за датами;
- ☐ збереження даних у хмарі;
- ☐ сучасний та зрозумілий інтерфейс.

2

АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ

Платформа	Аналіз	
	Переваги	Недоліки
Booking.com	• широкий вибір; • зрозумілий інтерфейс; • знижки.	• висока комісія; • проблеми з підтримкою.
Hotels24.ua	• локальний фокус; • оплата при поселенні.	• застарілий інтерфейс; • відсутність мобільного додатку.
airbnb	• унікальні пропозиції; • захист клієнтів.	• обмежена географія в Україні.

3

ПРОГРАМНА РЕАЛІЗАЦІЯ

Клієнтська частина:

- React – для UI компонентів;
- Redux Toolkit – управління станом;
- HTML/CSS/SCSS – розмітка та стилі.

Серверна частина:

- Node.js – серверне середовище;
- Express.js – веб-фреймворк;
- JWT – аутентифікація;
- Multer – завантаження файлів.

База даних:

- Google Firebase – хмарна БД;
- Firestore – NoSQL документи;
- Firebase Auth – система авторизації.

Інструменти:

- Visual Studio Code – IDE;
- DotEnv – конфігурація;
- Helmet – безпека;
- Nodemon – автоперезапуск.

4

АРХІТЕКТУРА ПРОЄКТУ

Серверна частина:

- Controllers – обробка бронювань, номерів, користувачів
- Middleware – аутентифікація, обробка помилок, завантаження файлів
- Routes – API маршрути до контролерів
- Config – налаштування БД та констант

Клієнтська частина:

- Components – повторювані UI компоненти
- Pages – сторінки додатку та стилі до них
- Features – логіка аутентифікації та бронювання
- Store – глобальний стан з Redux

5

РЕАЛІЗОВАНИЙ ФУНКЦІОНАЛ

Аутентифікація
Реєстрація, вхід, вихід користувачів з використанням JWT-токенів. Запис даних у локальне сховище браузера

Хмарне сховище
Усі дані користувачів, номерів, бронювань записуються до хмарної бази даних

Управління номерами
CRUD операції для готельних номерів

Розмежування ролей
Окремі ролі user та admin. Наявність адміністраторської панелі

Система бронювання
Створення, редагування та видалення бронювань

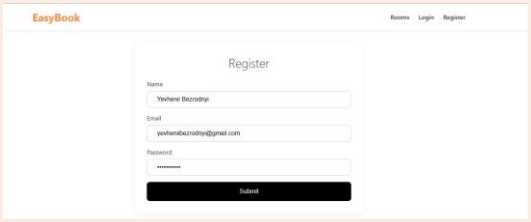
Система відгуків
Можливість залишати відгуки після бронювання

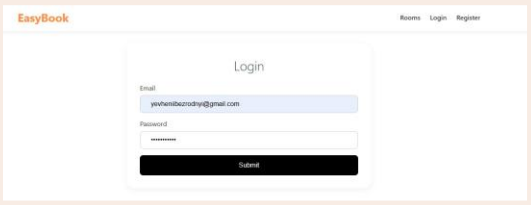
6

АУТЕНТИФІКАЦІЯ

Реєстрація, вхід, вихід користувачів з використанням JWT-токенів. Запис даних у локальне сховище браузера.

```
const loginUser = async (req, res) => {
  try {
    const { email, password } = req.body;

    if (!email || !password) {
      return res.status(400).json({ message: 'Email and password required' });
    }

    const userRecord = await auth.getUserByEmail(email);
    const token = await auth.createCustomToken(userRecord.uid);
    const doc = (await db.collection(usersDoc).doc(userRecord.uid).get()).data();

    res.status(200).json({
      message: 'Login successful',
      userId: userRecord.uid,
      email: userRecord.email,
      role: doc.role,
      token
    });
  } catch (error) {
    console.error('Login error:', error);

    const { status, message } = handleAuthError(error);
    res.status(status).json({
      message,
      error: process.env.NODE_ENV === 'development' ? error.message : undefined
    });
  }
};
```



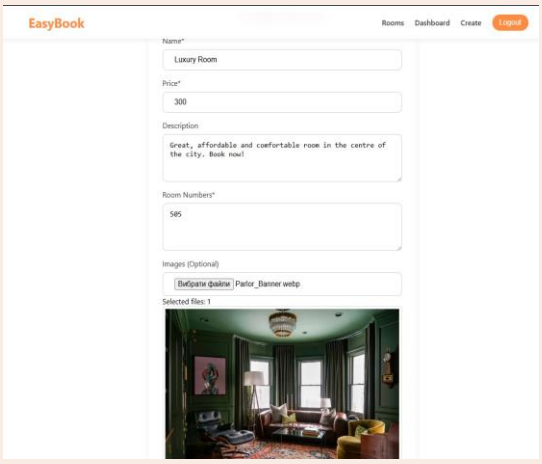


7

УПРАВЛІННЯ НОМЕРАМИ

CRUD операції для готельних номерів

```
const createRoom = async (req, res, next) => {
  try {
    const { name, price, desc, roomNumbers, img } = req.body;

    if (!name || !price || !roomNumbers) {
      return res.status(400).json({ error: "Missing required fields" });
    }
    const roomRef = await db.collection(roomsDoc).add({
      name,
      price,
      desc: desc || "",
      roomNumbers,
      imgUrl: img || "",
      createdAt: new Date(),
      updatedAt: new Date()
    });
    const newRoom = {
      id: roomRef.id,
      name,
      price,
      desc,
      roomNumbers,
      imgUrl: img
    };
    return res.status(201).json(newRoom);
  } catch (error) {
    next(error);
  }
};
```

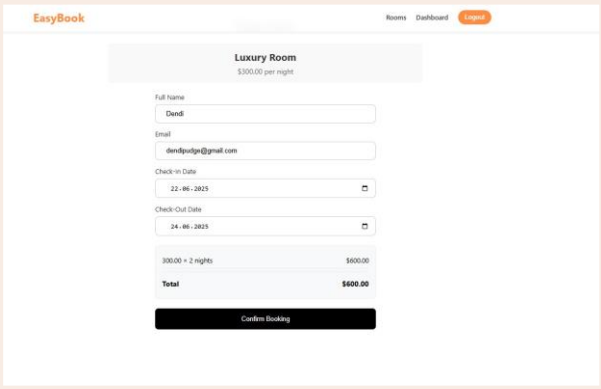


8

СИСТЕМА БРОНЮВАННЯ

Створення, редагування та видалення бронювань

```
const updateBooking = async (req, res, next) => {
  try {
    const bookingId = req.params.id;
    const { name, email, checkInDate, checkOutDate, status } = req.body;
    const bookingRef = db.collection(bookingsDoc).doc(bookingId);
    const doc = await bookingRef.get();
    if (!doc.exists) {
      return res.status(404).json({ error: "Booking not found" });
    }
    const updateData = {};
    if (name) updateData.name = name;
    if (email) updateData.email = email;
    if (checkInDate) updateData.checkInDate = new Date(checkInDate);
    if (checkOutDate) updateData.checkOutDate = new Date(checkOutDate);
    if (status) updateData.status = status;

    if (Object.keys(updateData).length === 0) {
      return res.status(400).json({ error: "No fields to update" });
    }
    updateData.updatedAt = new Date();
    await bookingRef.update(updateData);
    const updatedBooking = await bookingRef.get();
    return res.status(200).json({
      id: updatedBooking.id,
      ...updatedBooking.data()
    });
  } catch (error) {
    next(error);
  }
};
```

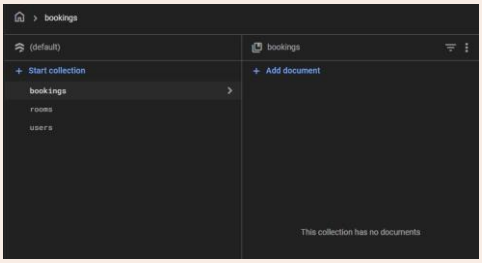


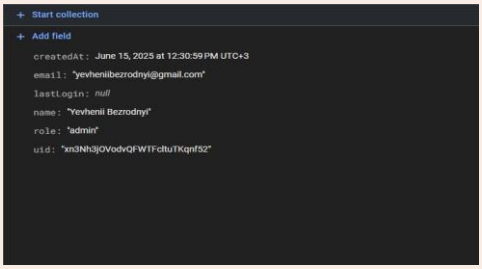
РЕАЛІЗОВАНИЙ ФУНКЦІОНАЛ

Хмарне сховище
Усі дані користувачів, номерів, бронювань записуються до хмарної бази даних

Розмежування ролей
Окремі ролі user та admin. Наявність адміністраторської панелі

Система відгуків
Можливість залишати відгуки після бронювання





ВИСНОВКИ:

Досягнуті результати:

- [ ] Створено повнофункіональний веб-сайт
- [ ] Реалізовано сучасний та зручний інтерфейс
- [ ] Впроваджено надійну систему аутентифікації
- [ ] Забезпечено розмежування ролей користувачів

Перспективи розвитку:

- [ ] Інтеграція інтерактивної мапи
- [ ] Додавання пошуку номерів за критеріями (в розробці)
- [ ] Інтеграція платіжних систем
- [ ] Система чату

11

ДОДАТОК Б

Програмний код

Б.1 Файли директорії back

```
const { bookingsDoc, roomsDoc, usersDoc } =
require("../config/constants");
const { db } = require("../config/fb");
const AppError = require("../utils/AppError");

const getBookings = async (req, res, next) => {
  try {
    const snapshot = await db.collection(bookingsDoc).get();

    if (snapshot.empty) {
      return res.status(200).json([]);
    }

    const bookings = snapshot.docs.map(doc => ({
      id: doc.id,
      ...doc.data()
    }));

    return res.status(200).json(bookings);
  } catch (error) {
    next(error);
  }
};

const createBooking = async (req, res, next) => {
  try {
    const { roomId, name, email, checkInDate, checkOutDate,
totalPrice, nights, roomName, roomPrice } = req.body;
    if (!roomId || !name || !email || !checkInDate ||
!checkOutDate) {
      return res.status(400).json({ error: "Missing required
fields" });
    }
    if (new Date(checkInDate) >= new Date(checkOutDate)) {
      return res.status(400).json({ error: "Check-out date must
be after check-in date" });
    }

    const imgUrl = (await
db.collection(roomsDoc).doc(roomId).get()).data().imgUrl[0];

    const bookingRef = await db.collection(bookingsDoc).add({
```

```
      roomId,
      name,
      email,
      userId: req.user.uid,
      checkInDate: new Date(checkInDate),
      checkOutDate: new Date(checkOutDate),
      totalPrice,
      nights,
      roomName,
      roomPrice,
      createdAt: new Date(),
      status: "confirmed",
      imgUrl,
    });

    const newBooking = {
      id: bookingRef.id,
      roomId,
      name,
      email,
      checkInDate,
      checkOutDate,
      totalPrice,
      nights,
      roomName,
      roomPrice,
    };

    return res.status(201).json(newBooking);
  } catch (error) {
    next(error);
  }
};

const getBooking = async (req, res, next) => {
  try {
    const bookingId = req.params.id;
    const doc = await
db.collection(bookingsDoc).doc(bookingId).get();

    if (!doc.exists) {
      return res.status(404).json({ error: "Booking not found"
});
    }

    const booking = {
      id: doc.id,
      ...doc.data()
    };

    return res.status(200).json(booking);
  } catch (error) {
    next(error);
```

```
  }
};

const updateBooking = async (req, res, next) => {
  try {
    const bookingId = req.params.id;
    const { name, email, checkInDate, checkOutDate, status } =
req.body;
    const bookingRef =
db.collection(bookingsDoc).doc(bookingId);
    const doc = await bookingRef.get();
    if (!doc.exists) {
      return res.status(404).json({ error: "Booking not found"
});
    }
    const updateData = {};
    if (name) updateData.name = name;
    if (email) updateData.email = email;
    if (checkInDate) updateData.checkInDate = new
Date(checkInDate);
    if (checkOutDate) updateData.checkOutDate = new
Date(checkOutDate);
    if (status) updateData.status = status;

    if (Object.keys(updateData).length === 0) {
      return res.status(400).json({ error: "No fields to update"
});
    }
    updateData.updatedAt = new Date();
    await bookingRef.update(updateData);
    const updatedBooking = await bookingRef.get();
    return res.status(200).json({
      id: updatedBooking.id,
      ...updatedBooking.data()
    });
  } catch (error) {
    next(error);
  }
};

const deleteBooking = async (req, res, next) => {
  try {
    const bookingId = req.params.id;
    const bookingRef =
db.collection(bookingsDoc).doc(bookingId);
    const doc = await bookingRef.get();

    if (!doc.exists) {
      return res.status(404).json({ error: "Booking not found"
});
    }

    await bookingRef.delete();
```

```
    return res.status(200).json({
      id: bookingId,
      message: "Booking deleted successfully"
    });
  } catch (error) {
    next(error);
  }
};
const getUserBookings = async (req, res, next) => {
  try {
    const userId = req.params.userId;

    const userRef = db.collection(usersDoc).doc(userId);
    const userDoc = await userRef.get();
    if (!userDoc.exists) {
      throw new AppError('User not found', 404);
    }

    if (req.user.role !== 'admin' && req.user.uid !== userId) {
      throw new AppError('Unauthorized to view these bookings',
403);
    }

    const snapshot = await db.collection(bookingsDoc)
      .where('userId', '==', userId)
      .orderBy('createdAt', 'desc')
      .get();

    if (snapshot.empty) {
      return res.status(200).json([]);
    }

    const bookings = snapshot.docs.map(doc => ({
      id: doc.id,
      ...doc.data()
    }));

    res.status(200).json(bookings);
  } catch (error) {
    next(error);
  }
};

const getMyBookings = async (req, res, next) => {
  try {
    const userId = req.user.uid;

    const snapshot = await db.collection(bookingsDoc)
      .where('userId', '==', userId)
      .orderBy('createdAt', 'desc')
      .get();

    if (snapshot.empty) {
```

```
      return res.status(200).json([]);
    }

    const bookings = snapshot.docs.map(doc => ({
      id: doc.id,
      ...doc.data()
    }));

    res.status(200).json(bookings);
  } catch (error) {
    next(error);
  }
};

const addReviewToBooking = async (req, res, next) => {
  try {
    const bookingId = req.params.id;
    const { rating, comment } = req.body;
    const userId = req.user.uid;

    if (!rating || rating < 1 || rating > 5) {
      throw new AppError('Valid rating (1-5) is required', 400);
    }

    const bookingRef = db.collection(bookingDoc).doc(bookingId);
    const bookingDoc = await bookingRef.get();

    if (!bookingDoc.exists) {
      throw new AppError('Booking not found', 404);
    }

    const booking = bookingDoc.data();

    if (booking.userId !== userId) {
      throw new AppError('You are not authorized to review this
booking', 403);
    }

    if (booking.review) {
      throw new AppError('Review already submitted for this
booking', 400);
    }
    await bookingRef.update({
      review: {
        rating,
        comment,
        createdAt: admin.firestore.FieldValue.serverTimestamp()
      }
    });

    const roomRef = db.collection(roomsDoc).doc(booking.roomId);
    const roomDoc = await roomRef.get();
    const room = roomDoc.data();
```

```
    const newRatingCount = (room.ratingCount || 0) + 1;
    const newRatingTotal = (room.ratingTotal || 0) + rating;
    const newAverageRating = newRatingTotal / newRatingCount;

    await roomRef.update({
      ratingCount: newRatingCount,
      ratingTotal: newRatingTotal,
      averageRating: parseFloat(newAverageRating.toFixed(1))
    });

    res.status(200).json({
      status: 'success',
      message: 'Review added successfully'
    });
  } catch (error) {
    next(error);
  }
};

module.exports = {
  getBookings,
  createBooking,
  getBooking,
  updateBooking,
  deleteBooking,
  addReviewToBooking,
  getMyBookings,
  getUserBookings,
};


const { auth, db } = require("../config/fb");

const authMiddleware = async (req, res, next) => {
  try {

    const authHeader = req.headers.authorization;
    if (!authHeader || !authHeader.startsWith('Bearer ')) {
      return res.status(401).json({
        message: 'Unauthorized - No token provided'
      });
    }

    const idToken = authHeader.split(' ')[1].trim();

    const decodedToken = await auth.verifyIdToken(idToken);
    req.user = decodedToken;

    next();
  } catch (error) {
    console.error('Authentication error:', error.message);
```

```
    let statusCode = 401;
    let message = 'Unauthorized - Invalid token';

    if (error.code === 'auth/id-token-expired') {
      message = 'Token expired';
    } else if (error.code === 'auth/argument-error') {
      message = 'Malformed token';
    } else if (error.code === 'auth/user-disabled') {
      message = 'User account disabled';
      statusCode = 403;
    }

    if (!res.headersSent) {
      return res.status(statusCode).json({
        message,
        error: process.env.NODE_ENV === 'development' ?
error.message : undefined
      });
    }

    console.warn('Authentication failed after headers were
sent');
  }
};

const adminOnlyMiddleware = async (req, res, next) => {
  try {
    if (!req.user || !req.user.uid) {
      return res.status(401).json({ message: 'Unauthorized -
User not authenticated' });
    }

    const userDoc = await
db.collection('users').doc(req.user.uid).get();

    if (!userDoc.exists) {
      return res.status(404).json({ message: 'User not found'
});
    }

    const userData = userDoc.data();

    if (userData.role !== 'admin') {
      return res.status(403).json({
        message: 'Forbidden - Admin privileges required',
        requiredRole: 'admin',
        yourRole: userData.role || 'user'
      });
    }

    req.userData = {
      ...req.user,
      ...userData
```

```
    };

    next();
  } catch (error) {
    console.error('Admin check error:', error.message);

    let status = 500;
    let message = 'Internal server error';

    if (error.code === 'resource-exhausted') {
      status = 429;
      message = 'Too many requests';
    } else if (error.code === 'permission-denied') {
      status = 403;
      message = 'Firestore permission denied';
    }

    if (!res.headersSent) {
      return res.status(status).json({
        message,
        error: process.env.NODE_ENV === 'development' ?
error.message : undefined
      });
    }
  }
};

module.exports = { auth: authMiddleware, adminOnly:
adminOnlyMiddleware };


const { Router } = require("express");
const {
  getRooms,
  createRoom,
  getRoom,
  updateRoom,
  deleteRoom,
} = require("../controllers/roomController");
const { auth, adminOnly } =
require("../middleware/authMiddleware");
const upload = require("../middleware/uploadMiddleware");

const router = Router();

router.get("/", getRooms);
router.get("/:id", getRoom);

router.post("/", auth, createRoom);
router.put("/:id", auth, adminOnly, updateRoom);
router.delete("/:id", auth, adminOnly, deleteRoom);

router.post("/upload-image",
```

```
  auth,
  upload.single('image'),
  async (req, res, next) => {
    try {
      if (!req.file) {
        return res.status(400).json({
          success: false,
          error: 'No file uploaded or invalid file type'
        });
      }

      const relativePath = req.file.path.replace(/^public[\\/]/,
'').replace(/\\/g, '/');

      res.json({
        success: true,
        message: 'File uploaded successfully',
        filePath: relativePath,
        fileUrl:
`${req.protocol}://${req.get('host')}/${relativePath}`
      });
    } catch (error) {
      console.error('Upload error:', error);
      next(error);
    }
  }
);

module.exports = router;
```

Б.2 Файли директорії front

```
import { createSlice, createAsyncThunk } from
"@reduxjs/toolkit";
import { auth } from "../../config/fb";
import {
  createUserWithEmailAndPassword,
  signInWithEmailAndPassword,
  getAuth,
  signOut,
} from "firebase/auth";

const storeUser = (data) => localStorage.setItem("user",
JSON.stringify(data));
const removeUser = () => localStorage.removeItem("user");

const user = JSON.parse(localStorage.getItem("user"));

export const registerUser = createAsyncThunk(
  "auth/register",
```

```
  async ({ email, password, name }, thunkApi) => {
    try {
      const res = await
fetch("http://localhost:5000/api/users/register", {
        method: "POST",
        headers: {
          "Content-Type": "application/json",
        },
        body: JSON.stringify({ email, password, name }),
      });

      const data = await res.json();

      return data;
    } catch (error) {
      return thunkApi.rejectWithValue(error.message);
    }
  }
);

export const loginUser = createAsyncThunk(
  "auth/login",
  async ({ email, password }, thunkApi) => {
    try {
      // 🔐 Verify password and log in with Firebase
      const userCredential = await
signInWithEmailAndPassword(auth, email, password);

      // 🚀 Get the ID token
      const idToken = await userCredential.user.getIdToken();

      // 📩 Send ID token to backend (for auth-protected
requests)
      const res = await
fetch("http://localhost:5000/api/users/login", {
        method: "POST",
        headers: {
          "Content-Type": "application/json",
        },
        body: JSON.stringify({ email, password }),
      });

      const data = await res.json();
      localStorage.setItem("user", JSON.stringify({ ...data,
idToken }));
      return data;
    } catch (error) {
      return thunkApi.rejectWithValue(error.message);
    }
  }
);
```

```
export const logoutUser = createAsyncThunk(
  "auth/logout",
  async (_, thunkApi) => {
    try {
      await signOut(auth);
      removeUser();
      return true;
    } catch (error) {
      return thunkApi.rejectWithValue(error.message);
    }
  }
);

const initialState = {
  user: user ? user : null,
  isLoading: false,
  isSuccess: false,
  isError: false,
  message: "",
};

const authSlice = createSlice({
  name: "auth",
  initialState,
  reducers: {
    reset: (state) => {
      state.isLoading = false;
      state.isSuccess = false;
      state.isError = false;
      state.message = "";
    },
  },
  extraReducers: (builder) => {
    builder
      .addCase(registerUser.pending, (state) => {
        state.isLoading = true;
      })
      .addCase(registerUser.fulfilled, (state, action) => {
        state.isLoading = false;
        state.isSuccess = true;
        state.user = action.payload;
      })
      .addCase(registerUser.rejected, (state, action) => {
        state.isLoading = false;
        state.isError = true;
        state.message = action.payload;
      })
      .addCase(loginUser.pending, (state) => {
        state.isLoading = true;
      })
      .addCase(loginUser.fulfilled, (state, action) => {
        state.isLoading = false;
        state.isSuccess = true;
```

```
        state.user = action.payload;
      })
      .addCase(loginUser.rejected, (state, action) => {
        state.isLoading = false;
        state.isError = true;
        state.message = action.payload;
      })
      .addCase(logoutUser.fulfilled, (state) => {
        state.user = null;
        state.isSuccess = true;
        state.isLoading = false;
      })
      .addCase(logoutUser.rejected, (state, action) => {
        state.isError = true;
        state.message = action.payload;
        state.isLoading = false;
      });
  },
});

export const { reset } = authSlice.actions;
export default authSlice.reducer;


import { createSlice, createAsyncThunk } from
"@reduxjs/toolkit";
import { auth } from "../../config/fb";

const initialState = {
  rooms: [],
  isLoading: false,
  isSuccess: false,
  isError: false,
  message: "",
};

export const createRoom = createAsyncThunk(
  "room/create",
  async (roomData, thunkApi) => {

    try {
      const currentUser = auth.currentUser;
      const token = await currentUser.getIdToken();
      const res = await fetch("http://localhost:5000/api/rooms",
{
        headers: {
          "Content-Type": "application/json",
          "Authorization": `Bearer ${token}`,
        },
        method: "POST",
        body: JSON.stringify(roomData),
      });
```

```
      if (!res.ok) {
        const error = await res.json();
        return thunkApi.rejectWithValue(error);
      }

      const data = await res.json();
      return data;
    } catch (error) {
      console.log(error.message);
      return thunkApi.rejectWithValue(error.message);
    }
  }
);

export const getRooms = createAsyncThunk("room/getall", async
(_, thunkApi) => {
  try {
    const res = await fetch("http://localhost:5000/api/rooms");
    if (!res.ok) {
      const error = await res.json();
      return thunkApi.rejectWithValue(error);
    }

    const data = await res.json();
    return data;
  } catch (error) {
    console.log(error.message);
    return thunkApi.rejectWithValue(error.message);
  }
});

export const updateRoom = createAsyncThunk(
  "/room/update",
  async (roomData, thunkApi) => {
    try {
      const currentUser = auth.currentUser;
      const token = await currentUser.getIdToken();
      const { roomId, ...rest } = roomData;
      const res = await
fetch(`http://localhost:5000/api/rooms/${roomId}`, {
        headers: {
          "Content-type": "application/json",
          "Authorization": `Bearer ${token}`,
        },
        method: "PUT",
        body: JSON.stringify(rest),
      });
      const data = await res.json();
      if (!res.ok) {
        return thunkApi.rejectWithValue(data);
      }

      return data;
```

```
    } catch (error) {
      console.log(error.message);
      return thunkApi.rejectWithValue(error.message);
    }
  }
);

export const deleteRoom = createAsyncThunk(
  "room/delete",
  async (roomId, thunkApi) => {
    try {
      const user = JSON.parse(localStorage.getItem("user"));
      const res = await
fetch(`http://localhost:5000/api/rooms/${roomId}`, {
        headers: {
          "Authorization": `Bearer ${user.idToken}`,
        },
        method: "DELETE",

      });
      const data = await res.json();
      if (!res.ok) {
        return thunkApi.rejectWithValue(data);
      }
      return data;
    } catch (error) {
      return thunkApi.rejectWithValue(error.message);
    }
  }
);

export const roomSlice = createSlice({
  name: "room",
  initialState,
  reducers: {
    reset: (state) => {
      state.isLoading = false;
      state.isError = false;
      state.isSuccess = false;
      state.message = "";
    },
  },
  extraReducers: (builder) => {
    builder
      .addCase(createRoom.pending, (state) => {
        state.isLoading = true;
      })
      .addCase(createRoom.fulfilled, (state, action) => {
        state.isLoading = false;
        state.isSuccess = true;
        state.rooms = action.payload;
      })
      .addCase(createRoom.rejected, (state, action) => {
```

```
        state.isLoading = false;
        state.isError = true;
        state.message = action.payload;
      })
      .addCase(getRooms.pending, (state) => {
        state.isLoading = true;
      })
      .addCase(getRooms.fulfilled, (state, action) => {
        state.isLoading = false;
        state.isSuccess = true;
        state.rooms = action.payload;
      })
      .addCase(getRooms.rejected, (state, action) => {
        state.isLoading = false;
        state.isError = true;
        state.message = action.payload;
      })
      .addCase(updateRoom.pending, (state) => {
        state.isLoading = true;
      })
      .addCase(updateRoom.fulfilled, (state, action) => {
        state.isSuccess = true;
        state.isLoading = false;
        state.rooms = action.payload;
      })
      .addCase(updateRoom.rejected, (state, action) => {
        state.isLoading = false;
        state.isError = true;
        state.message = action.payload;
      })
      .addCase(deleteRoom.pending, (state) => {
        state.isLoading = true;
      })
      .addCase(deleteRoom.fulfilled, (state, action) => {
        state.isLoading = false;
        state.isSuccess = true;
        state.rooms = state.rooms.filter(
          (room) => room._id != action.payload.id
        );
      })
      .addCase(deleteRoom.rejected, (state, action) => {
        state.isLoading = false;
        state.isError = true;
        state.message = action.payload;
      });
  },
});

export const { reset } = roomSlice.actions;
export default roomSlice.reducer;
```