Report

platform.

* **mongoose.connect()**: Establishes a connection to MongoDB.
* **.then()**: Logs "MongoDB connected" if the connection is successful.
* **.catch()**: Logs an error message if the connection fails.

It’s used to ensure the backend can interact with the MongoDB database.

email.js

module.exports = {

service: 'Gmail',

auth: {

user: 'akmaralabylkjair@gmail.com',

pass: process.env.GMAIL\_PASS,

},

};

This code exports an email configuration object to be used with an email-sending library (likely **Nodemailer**) in a Node.js application.

authController.js

const bcrypt = require('bcrypt');

const otplib = require('otplib');

const User = require('../models/user');

const { sendWelcomeEmail, sendFailedLoginNotification } = require('../utils/email-sender');

const QRCode = require('qrcode'); // Required for QR code generation

// Register User with hashed password and optional 2FA setup

exports.register = async (req, res) => {

const { username, password, firstName, lastName, age, gender, email } = req.body;

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

if (user) return res.status(404).json({ message: 'User already exists.' });

if (await User.findOne({ email })) {

return res.status(400).json({ message: 'Email already exists' });

}

try {

// Hash the password

const hashedPassword = await bcrypt.hash(password, 10);

// Create new user instance

const newUser = new User({

username,

email,

password: hashedPassword,

firstName,

lastName,

age,

gender,

email,

role: 'editor',

});

// Save user to the database

await newUser.save();

sendWelcomeEmail(email, firstName);

return res.status(201).json({ message: "Registration success" });

} catch (error) {

return res.status(500).json({ message: 'Registration failed.' });

}

};

// Setup 2FA (Generate QR Code)

exports.setup2FA = async (req, res) => {

try {

const user = await User.findOne({ username: req.params.username });

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

if (user.secret) {

return res.status(409).json({ message: '2FA already enabled' });

}

let secret = otplib.authenticator.generateSecret();

user.secret = secret;

await user.save();

// Generate otpauth URL for 2FA setup

const otpauth = otplib.authenticator.keyuri(user.username, 'PortfolioPlatform', secret);

// Generate QR code URL from otpauth URL

QRCode.toDataURL(otpauth, (err, qrCodeUrl) => {

if (err) {

return res.status(500).json({ message: 'Failed to generate QR code.' });

}

// Send QR code URL to the client

res.json({

qrCodeUrl,

});

});

} catch (error) {

res.status(500).json({ message: 'Failed to setup 2FA.' });

}

};

// Login with username and password, and optionally OTP for 2FA

exports.login = async (req, res) => {

const { username, password } = req.body;

try {

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

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

// Initialize failed login attempts if not set

if (!req.session.failedAttempts) {

req.session.failedAttempts = {};

}

const userAttempts = req.session.failedAttempts[username] || { count: 0, lastFailedLogin: null };

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

if (!isPasswordValid) {

userAttempts.count += 1;

userAttempts.lastFailedLogin = new Date();

req.session.failedAttempts[username] = userAttempts; // Update session

// If the user exceeded the max attempts, notify via email

if (userAttempts.count === 3) {

sendFailedLoginNotification(user.email);

}

return res.status(401).json({ message: 'Invalid password.' });

}

// Reset failed attempts on successful login

req.session.failedAttempts[username] = { count: 0, lastFailedLogin: null };

// Proceed with login

req.session.user = {

userId: user.\_id,

username: user.username,

firstName: user.firstName,

lastName: user.lastName,

email: user.email,

age: user.age,

gender: user.gender,

role: user.role,

};

if (!user.secret) {

res.json({ success: "no2Fa", message: "Logged in successfully!" });

} else {

res.json({ success: "2Fa", message: "Logged in successfully!" });

}

} catch (error) {

res.status(500).json({ success: "error", message: 'Login failed.' });

}

};

// Verify 2FA OTP and complete login

exports.verifyOTP = async (req, res) => {

const { token } = req.body;

const { username } = req.params;

try {

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

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

// Initialize failed OTP attempts if not set

if (!req.session.failedOTPAttempts) {

req.session.failedOTPAttempts = {};

}

const userOTPAttempts = req.session.failedOTPAttempts[username] || { count: 0, lastFailedLogin: null };

// Verify the OTP

const isTokenValid = otplib.authenticator.verify({ token, secret: user.secret });

if (!isTokenValid) {

userOTPAttempts.count += 1;

userOTPAttempts.lastFailedLogin = new Date();

req.session.failedOTPAttempts[username] = userOTPAttempts;

// If the user exceeded max attempts, notify via email

if (userOTPAttempts.count === 3) {

sendFailedLoginNotification(user.email);

}

return res.status(401).json({ message: 'Invalid 2FA code.' });

}

// Reset failed attempts on successful OTP verification

req.session.failedOTPAttempts[username] = { count: 0, lastFailedLogin: null };

req.session.user = {

userId: user.\_id,

username: user.username,

firstName: user.firstName,

lastName: user.lastName,

email: user.email,

age: user.age,

gender: user.gender,

role: user.role,

};

res.json({ success: true, message: "OTP verified successfully." });

} catch (error) {

res.status(500).json({ success: false, message: 'Failed to verify OTP.' });

}

};

exports.logout = async (req, res) => {

req.session.destroy((err) => {

if (err) {

return res.status(500).json({ message: 'Failed to log out.' });

}

res.clearCookie('connect.sid'); // Clear the session cookie

res.redirect('/');

});

};

exports.skip = async (req, res) => {

const { username } = req.params;

try {

// Find user by username and update the secret field to be empty

const user = await User.findOneAndUpdate(

{ username: username },

{ $set: { secret: null } },

{ new: true }

);

if (!user) {

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

}

return res.json({ message: '2FA setup skipped', user });

} catch (error) {

return res.status(500).json({ message: 'Internal server error' });

}

};

**Register User (register function)**

* **Purpose**: Registers a new user with a hashed password.
* **Steps**:
  + Checks if the username already exists in the database.
  + Checks if the email is already in use.
  + Hashes the provided password using bcrypt.
  + Creates a new user object and saves it to the database.
  + Sends a welcome email after successful registration.
  + Responds with a success or failure message.

**2. Setup 2FA (setup2FA function)**

* **Purpose**: Generates and sets up two-factor authentication (2FA) for the user.
* **Steps**:
  + Finds the user by username.
  + Checks if 2FA is already set up (if user.secret exists).
  + Generates a new 2FA secret using otplib.
  + Saves the secret to the user's record in the database.
  + Generates a QR code URL for setting up 2FA, using QRCode.toDataURL().
  + Responds with the QR code URL to the client.

**3. Login (login function)**

* **Purpose**: Allows the user to log in with username, password, and optionally 2FA.
* **Steps**:
  + Verifies if the user exists in the database.
  + Compares the password with the stored hashed password using bcrypt.compare().
  + Tracks failed login attempts in the session.
  + Sends an email notification if 3 failed login attempts are reached.
  + If the password is valid, checks if 2FA is enabled. If 2FA is set, the user is redirected for OTP verification.
  + Responds with a success message based on login outcome.

**4. Verify OTP (verifyOTP function)**

* **Purpose**: Verifies the 2FA OTP provided by the user.
* **Steps**:
  + Finds the user by username.
  + Tracks failed OTP attempts in the session.
  + Verifies the OTP using otplib.authenticator.verify().
  + If OTP is valid, logs the user in and clears failed OTP attempts.
  + If OTP is invalid, tracks the failed attempt and sends an email notification after 3 failures.
  + Responds with success or error based on OTP validity.

**5. Logout (logout function)**

* **Purpose**: Logs the user out and clears the session.
* **Steps**:
  + Destroys the session and clears the session cookie.
  + Redirects the user to the homepage.

**6. Skip 2FA (skip function)**

* **Purpose**: Allows the user to skip 2FA setup.
* **Steps**:
  + Finds the user by username and sets the secret field to null, effectively disabling 2FA.
  + Responds with a success message confirming the 2FA setup has been skipped

**Create Portfolio Item**

* **Route:** POST /portfolio-items
* **Description:** Accepts title, description, and images (uploaded files). It stores the image paths and associates the portfolio item with the user based on the session.
* **Error Handling:** Returns a 500 error if it fails to create the portfolio item.

js

Копировать код

exports.createPortfolioItem = async (req, res) => {

const { title, description } = req.body;

const images = req.files.map(file => path.join('uploads', file.filename)); // Save image paths

try {

const newItem = new PortfolioItem({

title,

description,

images,

userId: req.session.user.userId // Assign the userId from session

});

await newItem.save();

res.status(201).json({ message: 'Portfolio item created successfully', item: newItem });

} catch (error) {

console.error('Error creating portfolio item:', error);

res.status(500).json({ error: 'Failed to create portfolio item' });

}

};

**2. Get All Portfolio Items**

* **Route:** GET /portfolio-items
* **Description:** Fetches all portfolio items and adds user details (first name, last name) for each item.
* **Error Handling:** Returns a 500 error if it fails to fetch the portfolio items.

js

exports.getPortfolioItems = async (req, res) => {

try {

const items = await PortfolioItem.find();

const itemsWithUser = await Promise.all(items.map(async (item) => {

const user = await User.findById(item.userId); // Use findById to get user by userId

return {

...item.toObject(),

author: {

firstName: user ? user.firstName : 'Anonymous',

lastName: user ? user.lastName : 'Anonymous'

}

};

}));

res.json(itemsWithUser);

} catch (error) {

console.error('Error fetching portfolio items:', error);

res.status(500).json({ error: 'Failed to fetch portfolio items' });

}

};

**3. Get Portfolio Item by ID**

* **Route:** GET /portfolio-items/:id
* **Description:** Fetches a specific portfolio item by its ID.
* **Error Handling:** Returns a 404 error if the item is not found, and a 500 error if fetching the item fails.

js

exports.getByIdPortfolioItem = async (req, res) => {

try {

const itemId = req.params.id;

const item = await PortfolioItem.findById(itemId);

if (!item) {

return res.status(404).json({ error: 'Portfolio item not found' });

}

res.json(item);

} catch (error) {

console.error('Error fetching portfolio item:', error);

res.status(500).json({ error: 'Failed to fetch portfolio item' });

}

};

**4. Update Portfolio Item**

* **Route:** PUT /portfolio-items/:id
* **Description:** Allows updating portfolio item title, description, and images. It handles image deletions (removes images from the server and the database).
* **Error Handling:** Returns 404 error if item not found, 500 error if update fails.

js

exports.updatePortfolioItem = async (req, res) => {

const { title, description, deletedImages } = req.body;

try {

const portfolioItem = await PortfolioItem.findById(req.params.id);

const currentImages = portfolioItem ? portfolioItem.images : [];

const newImages = req.files ? req.files.map(file => path.join('uploads', file.filename)) : [];

const allImages = [...currentImages, ...newImages];

const deletedImagesArray = Array.isArray(deletedImages) ? deletedImages : [deletedImages];

if (deletedImagesArray && deletedImagesArray.length > 0) {

deletedImagesArray.forEach((imagePath) => {

const imageFilePath = path.join(\_\_dirname, '..', imagePath);

fs.unlink(imageFilePath, (err) => {

if (err) console.error('Error deleting image:', err);

else console.log('Deleted image:', imagePath);

});

const imageIndex = allImages.indexOf(imagePath);

if (imageIndex > -1) {

allImages.splice(imageIndex, 1);

}

});

}

const updatedItem = await PortfolioItem.findByIdAndUpdate(

req.params.id,

{ title, description, images: allImages, updatedAt: Date.now() },

{ new: true }

);

if (!updatedItem) {

return res.status(404).json({ error: 'Item not found' });

}

res.json({ message: 'Portfolio item updated successfully', item: updatedItem });

} catch (error) {

console.error('Error updating portfolio item:', error);

res.status(500).json({ error: 'Failed to update portfolio item' });

}

};

**5. Delete Portfolio Item (Admin Only)**

* **Route:** DELETE /portfolio-items/:id
* **Description:** Deletes a portfolio item, including its associated images from the server.
* **Error Handling:** Returns a 404 error if the item is not found, 500 error if deletion fails.

js

exports.deletePortfolioItem = async (req, res) => {

try {

const deletedItem = await PortfolioItem.findById(req.params.id);

if (!deletedItem) {

return res.status(404).json({ error: 'Item not found' });

}

deletedItem.images.forEach((imagePath) => {

const imageFilePath = path.join(\_\_dirname, '..', imagePath);

fs.unlink(imageFilePath, (err) => {

if (err) console.error('Error deleting image:', err);

else console.log('Deleted image:', imagePath);

});

});

await PortfolioItem.findByIdAndDelete(req.params.id);

res.json({ message: 'Portfolio item deleted successfully' });

} catch (error) {

console.error('Error deleting portfolio item:', error);

res.status(500).json({ error: 'Failed to delete portfolio item' });

}

};

**const express = require('express');**

**const router = express.Router();**

**const { isAuthenticated } = require('../middlewares/auth');**

**router.get('/profile', isAuthenticated, (req, res) => {**

**// If the user is authenticated, this will be executed**

**res.render('profile', { user: req.session.user });**

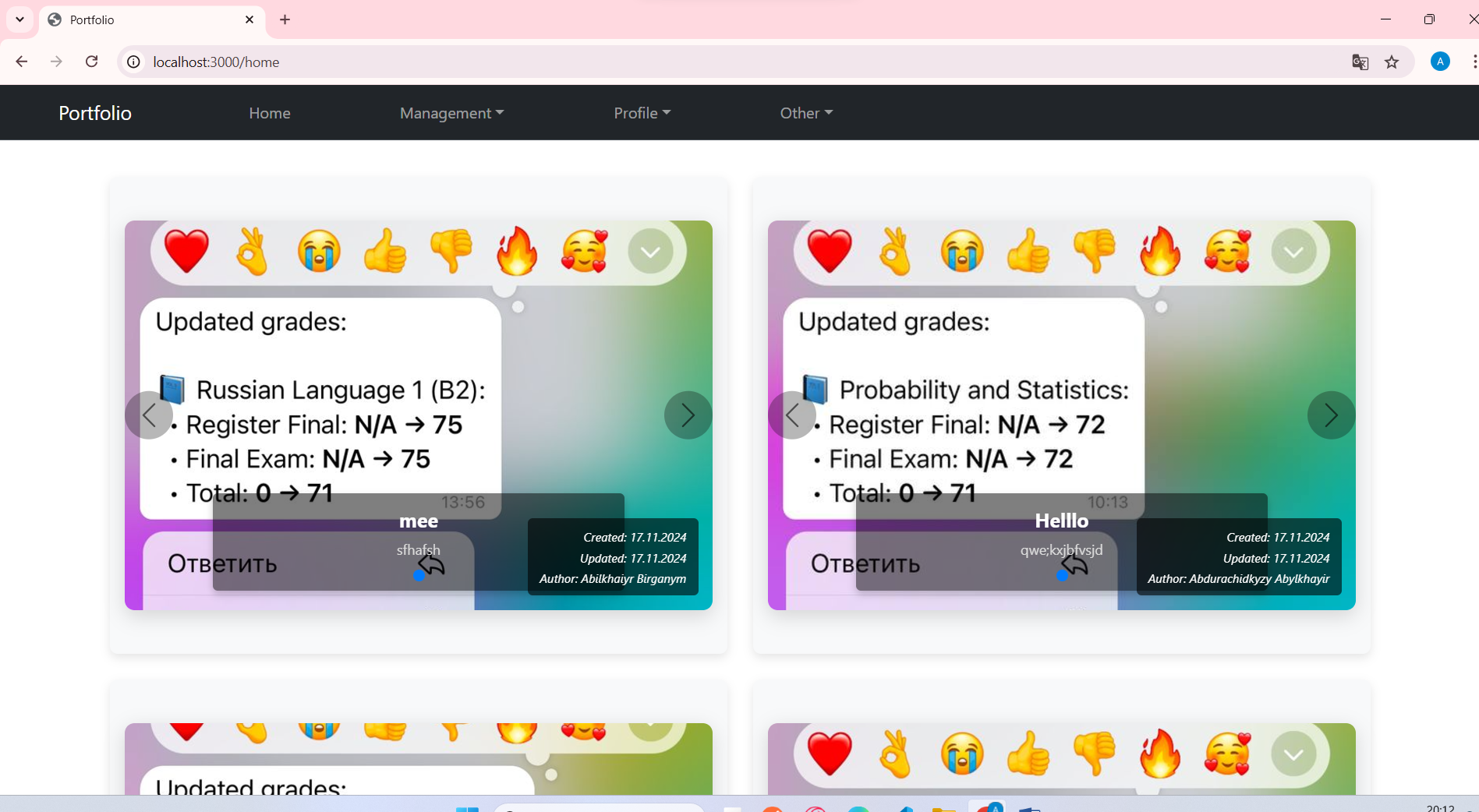
**});**

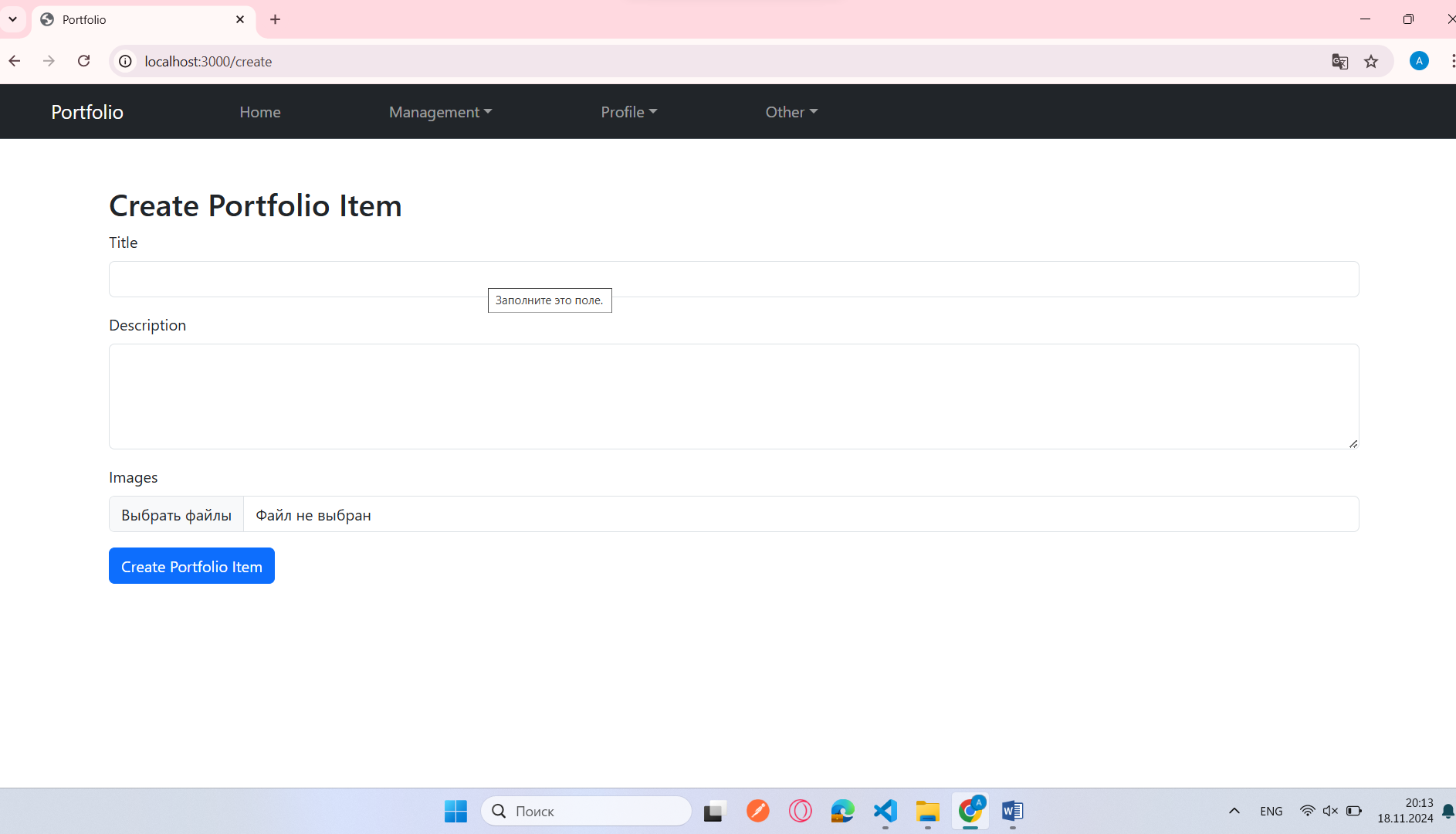
**router.get('/error', (req, res) => {**

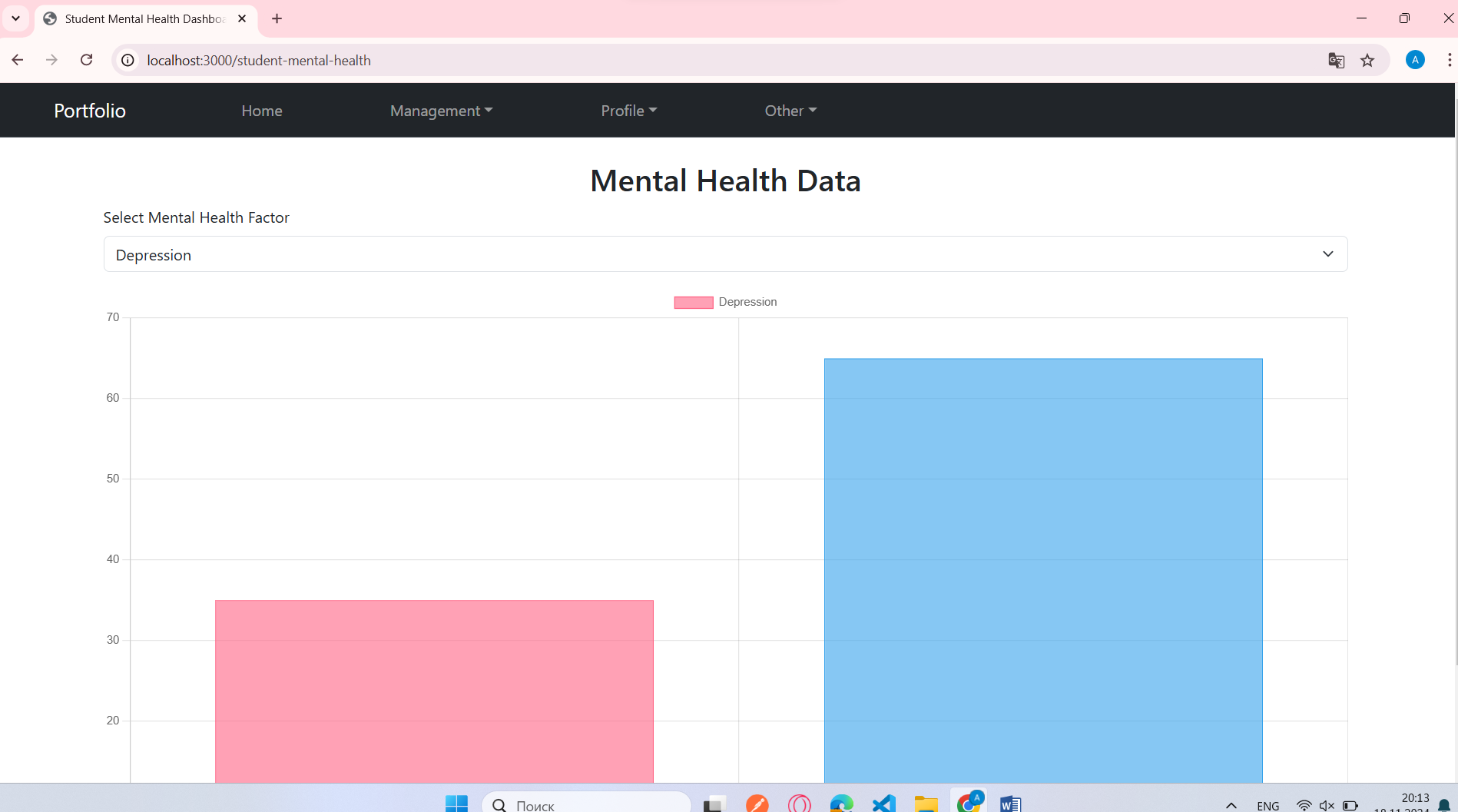
**// Render an error page if user is not authenticated**

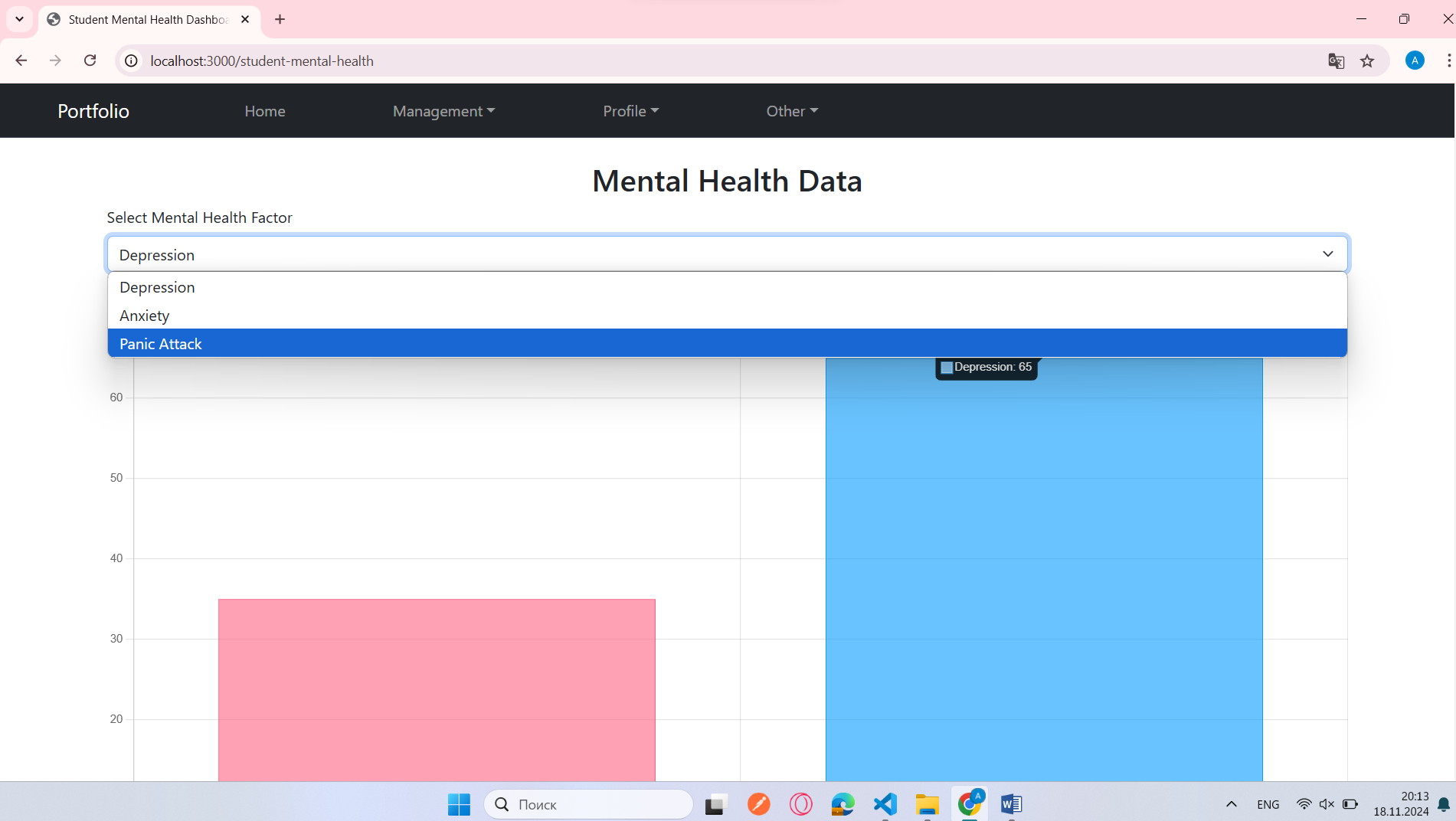
**res.render('error', { message: req.flash('error') });**

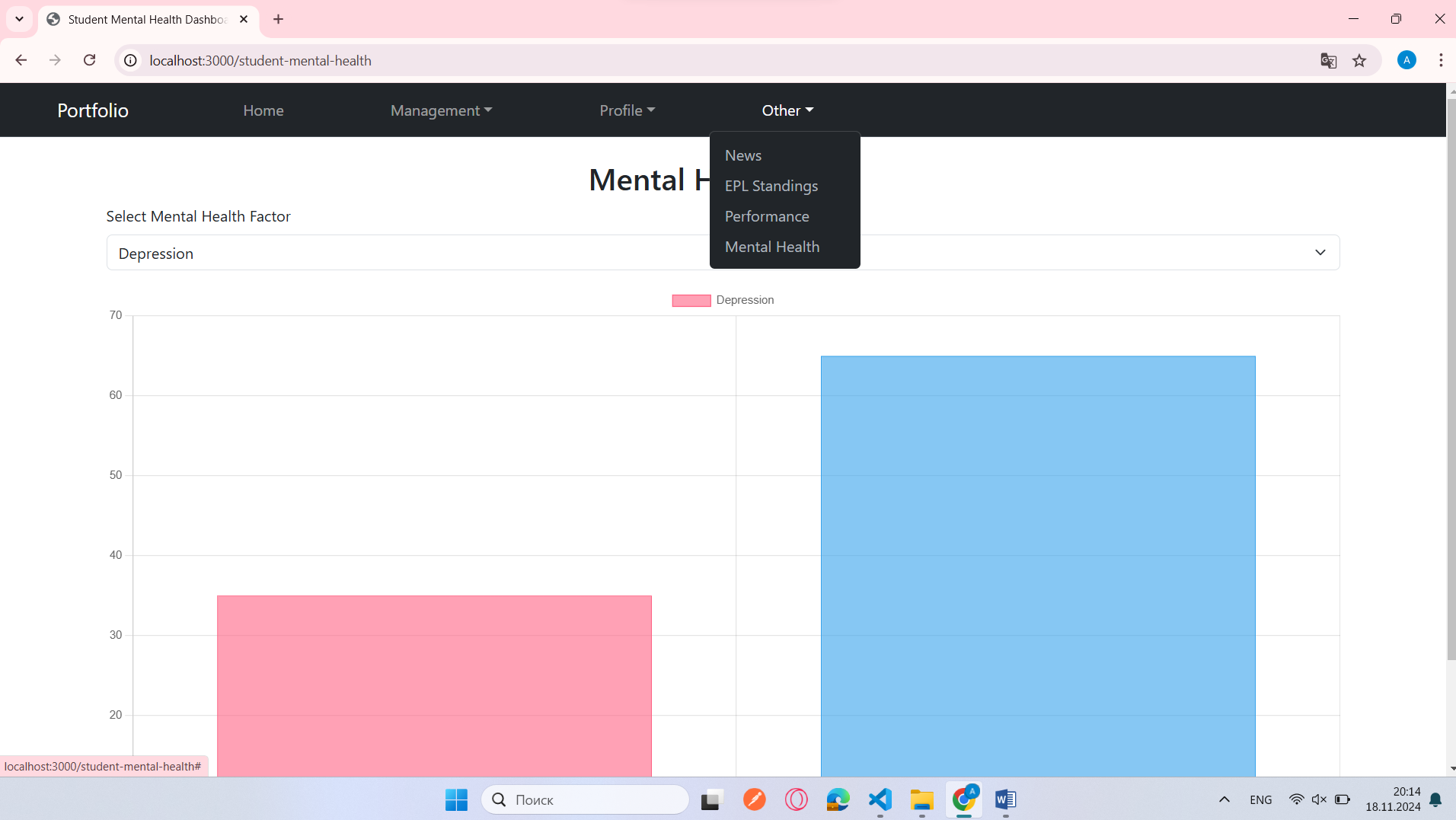
**});**

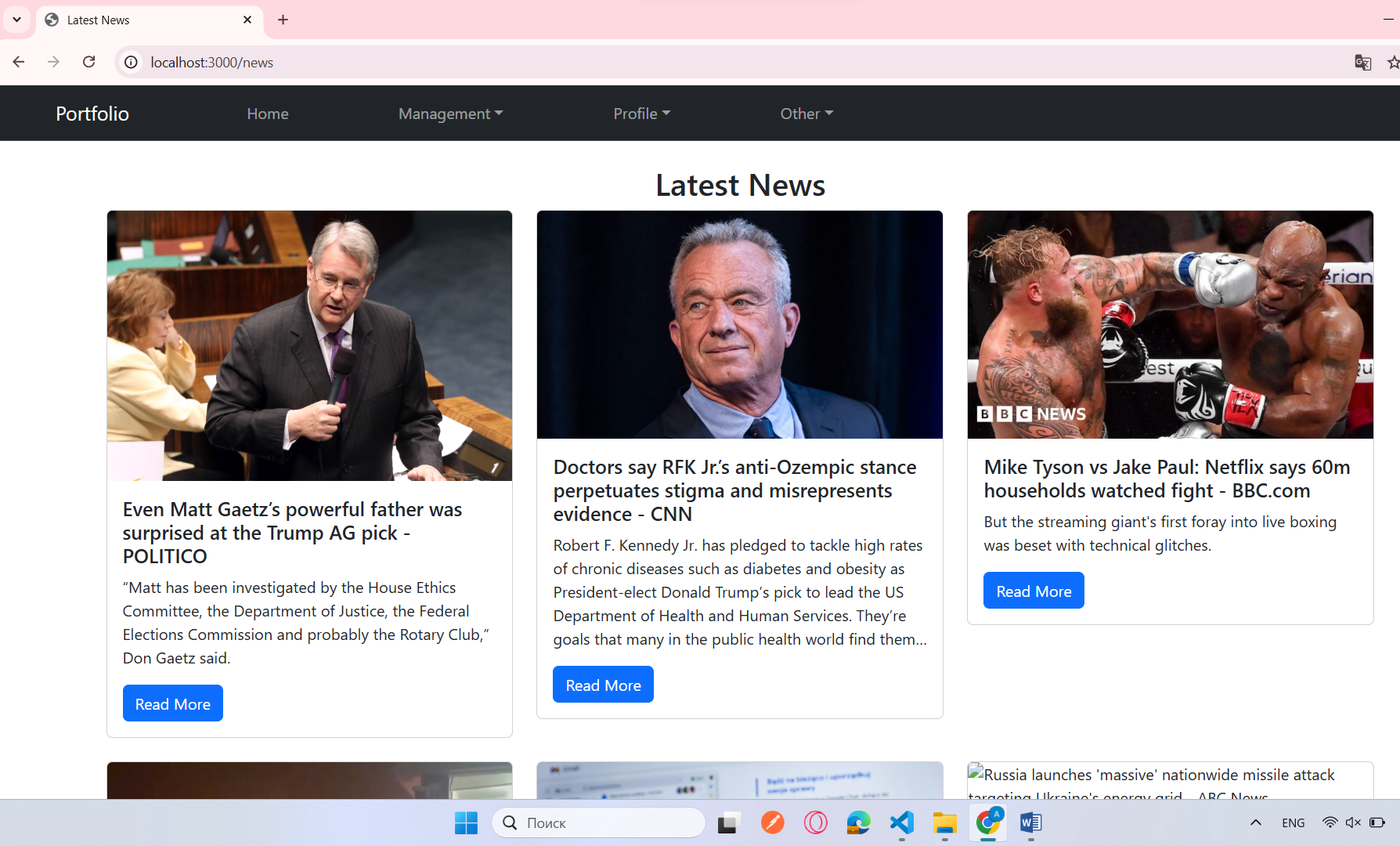








****

****