Full Stack Development with MERN

API Development and Integration Report

Date	12-07-2024
Team ID	SWTID1720106020
Project Name	Project – Banking Management App
Maximum Marks	10

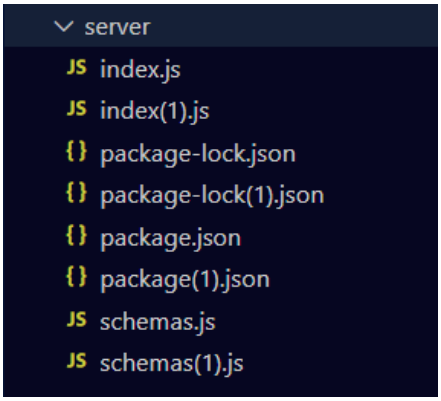
Project Title: Banking Management App
Date: 12-07-2024
Prepared by: Rangaiah Gari Harshavardhan Reddy, Kopparapu Shashank, Tirumala Mukesh Goud and Nimmala Sai Lahari

Objective
The objective of this report is to document the API development progress and key aspects of the backend services implementation for the Banking Application.

Technologies Used

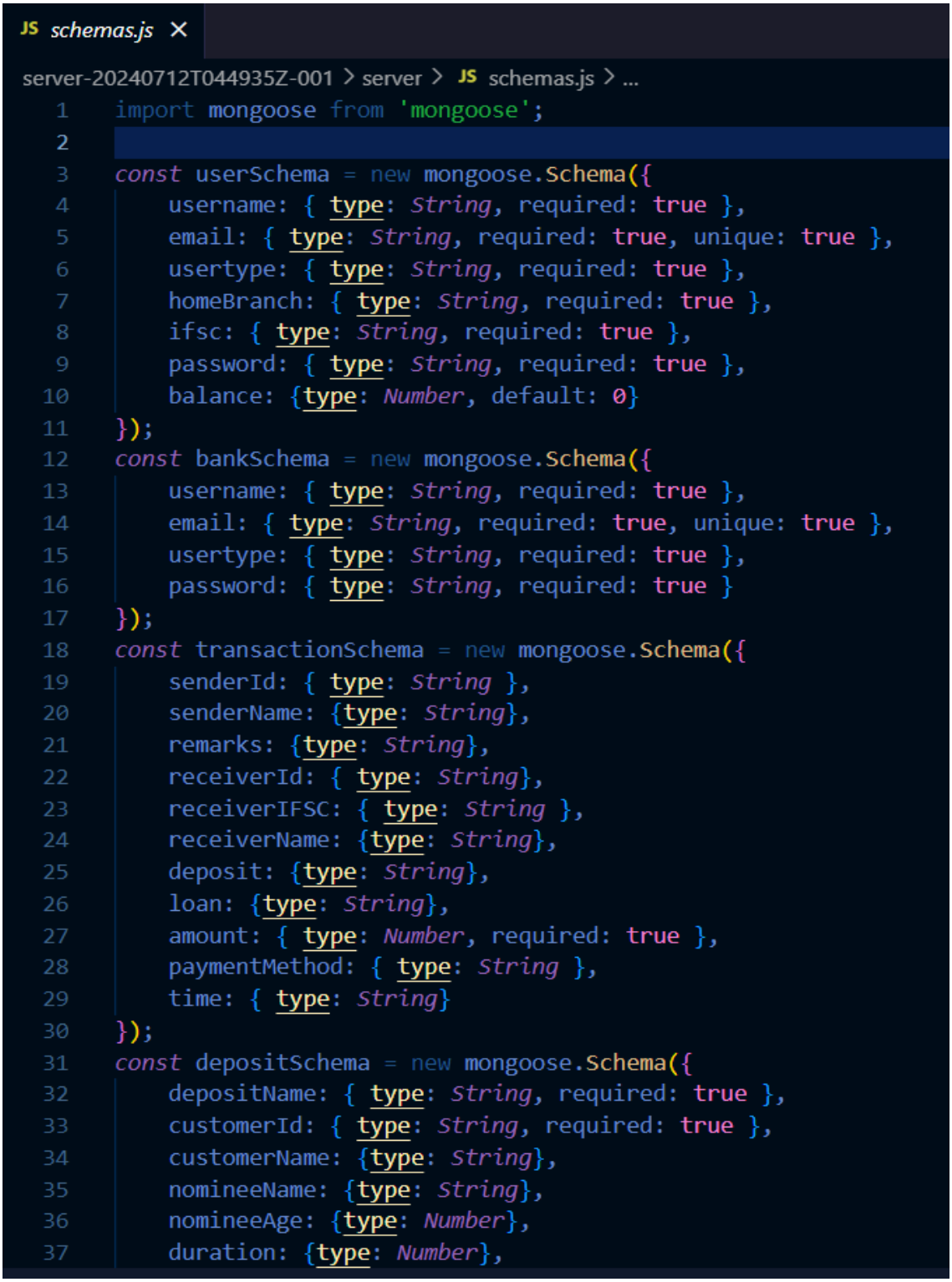
- **Backend Framework:** Node.js with Express.js
- **Database:** MongoDB
- **Authentication:** [e.g., JWT, OAuth]

Project Structure
Provide a screenshot of the backend project structure with explanations for key directories and files.



Key Directories and Files

1. **/schems.js**
    - Includes Mongoose schemas and models for MongoDB collections



```
JS schemas.js ×
server-20240712T044935Z-001 > server > JS schemas.js > ...
1   import mongoose from 'mongoose';
2
3   const userSchema = new mongoose.Schema({
4       username: { type: String, required: true },
5       email: { type: String, required: true, unique: true },
6       usertype: { type: String, required: true },
7       homeBranch: { type: String, required: true },
8       ifsc: { type: String, required: true },
9       password: { type: String, required: true },
10      balance: {type: Number, default: 0}
11  });
12  const bankSchema = new mongoose.Schema({
13      username: { type: String, required: true },
14      email: { type: String, required: true, unique: true },
15      usertype: { type: String, required: true },
16      password: { type: String, required: true }
17  });
18  const transactionSchema = new mongoose.Schema({
19      senderId: { type: String },
20      senderName: {type: String},
21      remarks: {type: String},
22      receiverId: { type: String},
23      receiverIFSC: { type: String },
24      receiverName: {type: String},
25      deposit: {type: String},
26      loan: {type: String},
27      amount: { type: Number, required: true },
28      paymentMethod: { type: String },
29      time: { type: String}
30  });
31  const depositSchema = new mongoose.Schema({
32      depositName: { type: String, required: true },
33      customerId: { type: String, required: true },
34      customerName: {type: String},
35      nomineeName: {type: String},
36      nomineeAge: {type: Number},
37      duration: {type: Number},
```

    - Contains functions to handle requests and responses.

```
    duration: {type: Number},
    amount: { type: Number, required: true },
    createdDate: {type:String},
    matureDate: {type:String}
}
);

const loanSchema = new mongoose.Schema({
    loanType: { type: String },
    customerId: { type: String, required: true },
    customerName: {type: String},
    nomineeName: {type: String},
    nomineeAge: {type: String},
    duration: {type: Number},
    loanAmount: { type: Number },
    balance: { type: Number },
    loanStatus: {type: String, default: 'pending'},
    createdDate: {type: String},
    endDate: {type: String}
}
);

export const User = mongoose.model('users',userSchema);
export const Bank = mongoose.model('bank',bankSchema);
export const Transactions = mongoose.model('transactions',transactionSchema);
export const Deposits = mongoose.model('deposits',depositSchema);
export const Loans = mongoose.model('loans',loanSchema);
```

2. **/index.js**

    1. **/controllers**

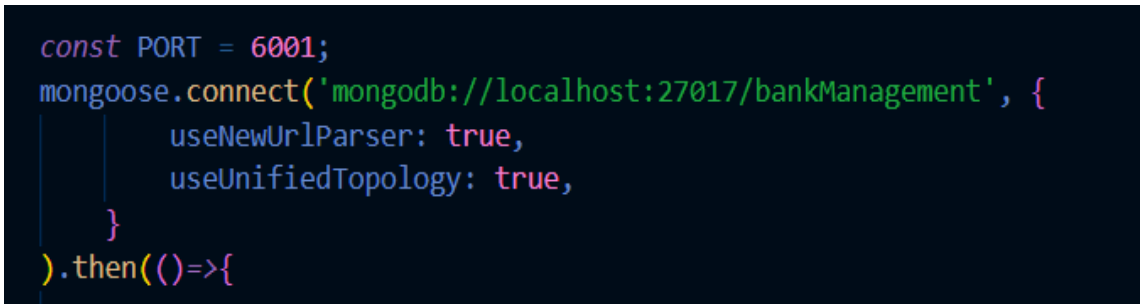
        - User Controller
        - Transaction Controller
        - Loan Controller

    2. **/Middleware**

```
const requestLogger = (req, res, next) => {
    console.log(`${req.method} ${req.url} -${new Date().toISOString()}`);
    next(); }; app.use(requestLogger);
```

3. **/config**

- Configuration files for database connections.



API Endpoints
A summary of the main API endpoints and their purposes:

Authentication and User Management

- POST /register: Registers a new user
- POST /login: Logs in a user .
- GET /user-details/:id: Fetches the details of a user by their ID.

Transactions

- POST /send-money: Sends money from one user to another.
- GET /transactions: Fetches all transactions.

Deposits

- GET /fetch-deposits: Fetches all deposit records.
- POST /new-deposit: Creates a new deposit for a customer.

Loans

- GET /fetch-loans: Fetches all loan records.
- POST /new-loan: Creates a new loan request for a customer.
- PUT /approve-loan: Approves a loan request.
- PUT /decline-loan: Declines a loan request.
- POST /repay-loan: Processes a loan repayment.

Users

- GET /fetch-users: Fetches all users.

User Authentication

- **POST /api/user/register** - Registers a new user.

```
app.post('/register', async (req, res) => {
    const { username, email, usertype, password, homeBranch } = req.body;
    try {
      if (usertype === 'customer'){
        const existingUser = await User.findOne({ email });
        if (existingUser) {
            return res.status(400).json({ message: 'User already exists' });
        }
        const IFSC = {'hyderabad': 'SB007HYD25',
                      'bangalore': 'SB007BLR30',
                      'chennai': 'SB007CNI99',
                      'mumbai': 'SB007MBI12',
                      'tirupati': 'SB007TPTY05',
                      'vizag': 'SB007VZG229',
                      'pune': 'SB007PN77',
                      'delhi': 'SB007DLI09',
                      'kochi': 'SB007KCI540',
                      'Venkatagiri': 'SB007VGR313',  }
        const hashedPassword = await bcrypt.hash(password, 10);
        const newUser = new User({
            username,
            email,
            usertype,
            homeBranch,
            ifsc : IFSC[homeBranch],
            password: hashedPassword
        });
        const userCreated = await newUser.save();
        return res.status(201).json(userCreated);
      }else if (usertype === 'admin'){
        const existingUser = await Bank.findOne({ email });
        if (existingUser) {
            return res.status(400).json({ message: 'User already exists' });
        }
        const hashedPassword = await bcrypt.hash(password, 10);
```

- **POST /api/user/login -** Authenticates a user and returns a token.

```
app.post('/login', async (req, res) => {
    const { email, usertype, password } = req.body;
    try {

      if (usertype === 'customer'){

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

            if (!user) {
                return res.status(401).json({ message: 'Invalid email or password' });
            }
            const isMatch = await bcrypt.compare(password, user.password);
            if (!isMatch) {
                return res.status(401).json({ message: 'Invalid email or password' });
            } else{

                return res.json(user);
            }
      }else if (usertype === 'admin'){
            const user = await Bank.findOne({ email });

            if (!user) {
                return res.status(401).json({ message: 'Invalid email or password' });
            }
            const isMatch = await bcrypt.compare(password, user.password);
            if (!isMatch) {
                return res.status(401).json({ message: 'Invalid email or password' });
            } else{
                return res.json(user);
            }
      }
    } catch (error) {
      console.log(error);
      return res.status(500).json({ message: 'Server Error' });
    }
```

User Management

- **GET /api/user/-** Retrieves user information by ID.

```
app.get('/user-details/:id', async (req, res) => {
    try{

        const user = await User.findOne({_id: req.params.id});
        if(!user){
            return res.status(404).json({ message: 'User not found' });
        }
        res.json(user);
    } catch (error) {
      console.log(error);
      return res.status(500).json({ message: 'Server Error' });
    }
});
```

- **PUT /api/user/-** Updates user information by ID.

```
app.put('/approve-loan', async (req, res)=>{
    const {id} = req.body;
    try{
        console.log(id)
        const loan = await Loans.findOne({_id: id});
        const user = await User.findOne({_id: loan.customerId});
        loan.loanStatus = 'approved';
        user.balance = user.balance + loan.loanAmount;
        await loan.save();
        await user.save();
        res.json({message:"loan approved successfully"});

        const transaction = new Transactions({
            receiverId: user._id, receiverName: user.name, loan: loan.loanType, amount, time: new Date(), remarks: "Loan approval"
        })
        await transaction.save();

    }catch(err){
        res.status(500).json({message: 'error occured'});
    }
})
```

Deposites Management

- **GET /api/fetch-deposits** - Retrieves all deposits.

```
app.get('/transactions', async (req, res)=>{
    try{

        const transactions = await Transactions.find();
        res.json(transactions);

    }catch(err){
        res.status(500).json({message: "Error occured"});
    }
})

// All deposits

app.get('/fetch-deposits', async (req, res)=>{
    try{
        const deposits = await Deposits.find();
        res.json(deposits);
    }catch(err){
        res.status(500).json({message: "Error occured"});
    }
})
```

- **POST /api/new-deposit** - Adds new Deposit.

```
app.post('/new-deposit', async (req, res) =>{
    const {depositName, customerId, customerName, nomineeName, nomineeAge, duration, amount, createdDate} = req.body
    try{
        const date = new Date(createdDate);

        const matureDate = date.getDate() + '-' + (date.getMonth() % 12) + '-' + (date.getFullYear() + Math.floor(duration/12) )
        const user = await User.findOne({_id: customerId});
        const newDeposit = new Deposits({
            depositName, customerId, customerName, nomineeName, nomineeAge, duration, amount, createdDate, matureDate
        });

        const transaction = new Transactions({
            senderId: customerId, senderName: customerName, deposit: depositName, amount, time: new Date(), remarks: "Deposit payment"
        })
        await transaction.save();

        const depo = await newDeposit.save();
        user.balance = user.balance - amount;
        await user.save();
        res.json({message: "deposit created"});
    }catch(err){
        res.status(500).json({message: "Error occured"});
    }
})
```

User Details

- **GET /api/fetchuserdetails** - Retrieves all details of a user.

Loans Data Management

- **GET /api/fetch-loans** - Retrieves all loans.

```
app.get('/fetch-loans', async (req, res)=>{
    try{
        const loans = await Loans.find();
        res.json(loans);
    }catch(err){
        res.status(500).json({message: "Error occured"});
    }
})
```

- **POST /api/new-loan** - Creates a new loan.

```
app.post('/new-loan', async (req, res) =>{

    const {loanType, customerId, customerName, nomineeName, nomineeAge, duration, loanAmount, createdDate} = req.body
    try{
        const date = new Date(createdDate);

        const endDate = date.getDate() + '-' + (date.getMonth() % 12) + '-' + (date.getFullYear() + Math.floor(duration/12) );

        const balance = loanAmount;
        const newLoan = new Loans({
            loanType, customerId, customerName, nomineeName, nomineeAge, duration, loanAmount, balance, createdDate, endDate
        })
        const loan = await newLoan.save();
        res.json({message: "loan request created"});
    }catch(err){
        res.status(500).json({message: "Error occured"});
    }
})
```

- **POST /api/repay -loan** – Repays an existing loan.

```
app.post('/repay-loan', async (req, res)=>{
    const {id, amount} = req.body;
    try{
        const loan = await Loans.findOne({_id: id});
        const user = await User.findOne({_id: loan.customerId});
        loan.balance = loan.balance - amount;
        user.balance = user.balance - amount;
        await loan.save();
        await user.save();

        const transaction = new Transactions({
            senderId: user._id, senderName: user.name, loan: loan.loanType, amount, time: new Date(), remarks: "Loan re-payment"
        })
        await transaction.save();

        res.json({message: 'repayment successful'});
    }catch(err){
        res.status(500).json({message: 'message occured'});
    }
})
```

- **PUT /api/approve -loan** -Approves a new loan.

```
app.put('/approve-loan', async (req, res)=>{
    const {id} = req.body;
    try{
        console.log(id)
        const loan = await Loans.findOne({_id: id});
        const user = await User.findOne({_id: loan.customerId});
        loan.loanStatus = 'approved';
        user.balance = user.balance + loan.loanAmount;
        await loan.save();
        await user.save();
        res.json({message:"loan approved successfully"});

        const transaction = new Transactions({
            receiverId: user._id, receiverName: user.name, loan: loan.loanType, amount, time: new Date(), remarks: "Loan approval"
        })
        await transaction.save();

    }catch(err){
        res.status(500).json({message: 'error occured'});
    }
})
```

- **PUT /api/decline -loan** -Declines a loan if the customer is not eligible.

```
app.put('/decline-loan', async (req, res)=>{
    try{
        const {id} = req.body;
        const loan = await Loans.findOne({_id: id});
        loan.loanStatus = 'declined';
        await loan.save();
        res.json({message:"loan declined successfully"});

    }catch(err){
        res.status(500).json({message: 'error occured'});
    }
})
```

Integration with Frontend

The backend communicates with the frontend via RESTful APIs. Key points of integration include:

- **User Authentication:** Tokens are passed between frontend and backend to handle authentication.

```
app.post('/login', async (req, res) => {
    const { email, usertype, password } = req.body;
    try {

      if (usertype === 'customer'){

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

          if (!user) {
              return res.status(401).json({ message: 'Invalid email or password' });
          }
          const isMatch = await bcrypt.compare(password, user.password);
          if (!isMatch) {
              return res.status(401).json({ message: 'Invalid email or password' });
          } else{

              return res.json(user);
          }
      }else if (usertype === 'admin'){
          const user = await Bank.findOne({ email });

          if (!user) {
              return res.status(401).json({ message: 'Invalid email or password' });
          }
          const isMatch = await bcrypt.compare(password, user.password);
          if (!isMatch) {
              return res.status(401).json({ message: 'Invalid email or password' });
          } else{
              return res.json(user);
          }
      }
    } catch (error) {
      console.log(error);
      return res.status(500).json({ message: 'Server Error' });
    }
```

- **Data Fetching:** Frontend components make API calls to fetch necessary data for display and interaction.

```
app.get('/user-details/:id', async (req, res) => {
    try{

        const user = await User.findOne({_id: req.params.id});
        if(!user){
            return res.status(404).json({ message: 'User not found' });
        }
        res.json(user);
    } catch (error) {
      console.log(error);
      return res.status(500).json({ message: 'Server Error' });
    }
});
```

Error Handling and Validation
Describe the error handling strategy and validation mechanisms:

- **Error Handling:** Centralized error handling using middleware.

```
app.post('/new-deposit', async (req, res) =>{
    const {depositName, customerId, customerName, nomineeName, nomineeAge, duration, amount, createdDate} = req.body
    try{
        const date = new Date(createdDate);

        const matureDate = date.getDate() + '-' + (date.getMonth() % 12) + '-' + (date.getFullYear() + Math.floor(duration/12) )
        const user = await User.findOne({_id: customerId});
        const newDeposit = new Deposits({
            depositName, customerId, customerName, nomineeName, nomineeAge, duration, amount, createdDate, matureDate
        });

        const transaction = new Transactions({
            senderId: customerId, senderName: customerName, deposit: depositName, amount, time: new Date(), remarks: "Deposit payment"
        })
        await transaction.save();

        const depo = await newDeposit.save();
        user.balance = user.balance - amount;
        await user.save();
        res.json({message: "deposit created"});
    }catch(err){
        res.status(500).json({message: "Error occured"});
    }
})
```

- **Validation:** Input validation using libraries like Json.

```
app.post('/send-money', async (req, res) =>{
    const {senderId, senderName, remarks, receiverId, receiverIFSC, amount, paymentMethod, time} = req.body;
    console.log(req.body);
    try{
        const sender = await User.findOne({_id: senderId});
        const receiver = await User.findOne({_id: receiverId});
        if(!receiver){
            return res.status(404).json({message: 'Receiver not exists'})
        }
        if(receiver.ifsc !== receiverIFSC){
            return res.status(500).json({message: 'Transaction failed'})
        }
        const receiverName = receiver.username;
        const transaction = new Transactions({
            senderId,
            senderName,
            receiverId,
            receiverIFSC,
            receiverName,
            amount,
            remarks,
            paymentMethod,
            time
        })
        const newTransaction = await transaction.save();
        sender.balance = parseFloat(sender.balance) - parseFloat(amount);
        receiver.balance = parseFloat(receiver.balance) + parseFloat(amount);
        await sender.save();
        await receiver.save();
        res.json({message: 'Transaction successful'});
    } catch (err){
    }
});
```

Security Considerations
Outline the security measures implemented:

Password Hashing

- Uses bcrypt to hash passwords before storing them in the database, ensuring that passwords are not stored in plain text.

Input Validation and Error Handling

- Basic error handling for database queries and user input is implemented to ensure robustness and to prevent the application from crashing.

CORS

- Uses cors middleware to handle Cross-Origin Resource Sharing, which controls how resources are shared across different domains.

Environment Configuration

- The database connection string and other sensitive information should be stored in environment variables (e.g., using dotenv package) to avoid exposing them in the source code.

- **Authentication:** Secure authentication.

```
app.post('/login', async (req, res) => {
    const { email, usertype, password } = req.body;
    try {

      if (usertype === 'customer'){

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

            if (!user) {
                return res.status(401).json({ message: 'Invalid email or password' });
            }
            const isMatch = await bcrypt.compare(password, user.password);
            if (!isMatch) {
                return res.status(401).json({ message: 'Invalid email or password' });
            } else{

                return res.json(user);
            }
      }else if (usertype === 'admin'){
            const user = await Bank.findOne({ email });

            if (!user) {
                return res.status(401).json({ message: 'Invalid email or password' });
            }
            const isMatch = await bcrypt.compare(password, user.password);
            if (!isMatch) {
                return res.status(401).json({ message: 'Invalid email or password' });
            } else{
                return res.json(user);
            }
      }
```

- **Data Encryption:** Encrypt sensitive data at rest and in transit.

```
        const hashedPassword = await bcrypt.hash(password, 10);
        const newUser = new User({
            username,
            email,
            usertype,
            homeBranch,
            ifsc : IFSC[homeBranch],
            password: hashedPassword
        });
        const userCreated = await newUser.save();
        return res.status(201).json(userCreated);
      }else if (usertype === 'admin'){
        const existingUser = await Bank.findOne({ email });
        if (existingUser) {
            return res.status(400).json({ message: 'User already exists' });
        }
        const hashedPassword = await bcrypt.hash(password, 10);
        const newUser = new Bank({ username, email, usertype, password: hashedPassword });
        const userCreated = await newUser.save();
        return res.status(201).json(user);
      }
    } catch (error) {
      console.log(error);
      return res.status(500).json({ message: 'Server Error' });
    }
});
```