ZAEEM PATEL ST10201991

THASLYN GOVENDER ST10133946

RYLAN NEWMAN ST10190421

LIAM COLE ABRAHAM ST10144656

UBAIDULLAH YUSUF SHAIK ST10232176

1. Using MongoDB Atlas

Why MongoDB Atlas?

MongoDB Atlas is a cloud-hosted version of MongoDB that offers several advantages, particularly in scalability, security, and global distribution. Here's why it's the preferred database for this application:

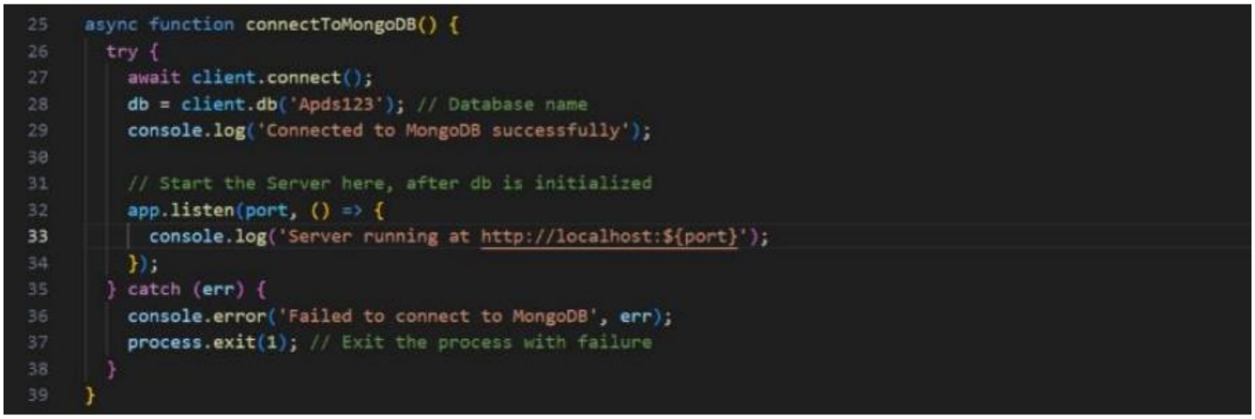
- **Scalability:** Atlas dynamically scales database clusters, making it easy to handle growing demands.
- **Security:** Built-in SSL encryption, IP whitelisting, and integrated user authentication enhance the security of data in transit.
- **Global Distribution:** Atlas allows for global distribution of data, reducing latency by placing databases close to users.

2. Asynchronous Programming in Node.js

Node.js uses an event-driven, non-blocking model that's efficient for asynchronous operations. Here, async/await manages the MongoDB connection process, ensuring that other processes aren't blocked while the connection is being established.

What Makes Something Asynchronous?

- **Non-Blocking I/O:** Asynchronous programming allows the program to continue executing without waiting for I/O-bound tasks (like database connections) to complete.
- **Scalability:** It improves throughput and response time, allowing the application to handle multiple tasks simultaneously.



```javascript
async function connectToMongoDB() {
  try {
    await client.connect();
    db = client.db('Apds123'); // Database name
    console.log('Connected to MongoDB successfully');

    // Start the Server here, after db is initialized
    app.listen(port, () => {
      console.log('Server running at http://localhost:${port}');
    });
  } catch (err) {
    console.error('Failed to connect to MongoDB', err);
    process.exit(1); // Exit the process with failure
  }
}
```

The try-catch block in the code handles connection errors, enabling a graceful shutdown with `process.exit(1)` on connection failure, which is crucial for robust production systems.

3. Error Handling and Resilience

Effective error handling ensures the application handles failures gracefully:

- **Graceful Shutdown:** If the MongoDB connection fails, the application exits gracefully. This prevents running a service without a database connection.
- **Logging:** Errors, such as MongoDB connection failures, are logged to aid diagnosis. Proper logging is essential for troubleshooting issues like credential errors or network outages.

4. Why Mongoose is Preferred Over Native MongoDB Driver

Mongoose provides higher-level abstractions over the native MongoDB driver:

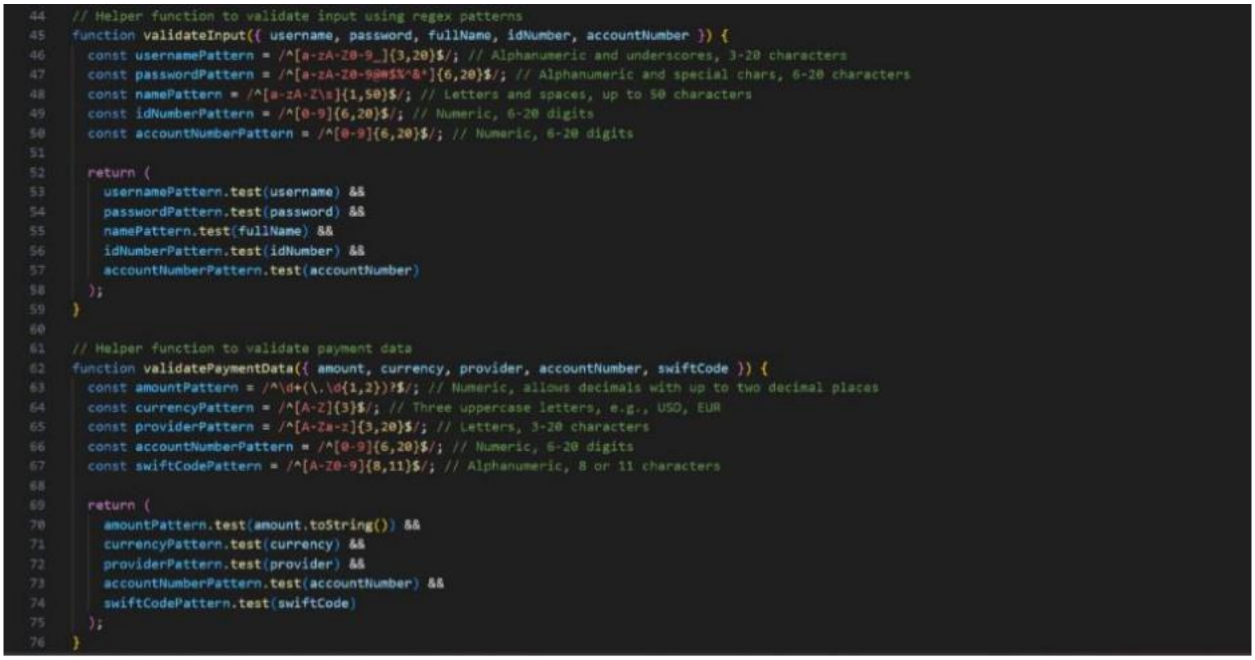
- **Schema Validation:** Validates data according to predefined schemas before saving, preventing invalid documents.
- **Virtuals and Middleware:** Virtual properties allow computed fields that aren't stored in MongoDB, while middleware supports functions to execute actions before or after specific events (e.g., pre-save).

Whitelisting in MongoDB Atlas

Whitelisting allows only specific IP addresses to connect to your MongoDB instance, adding an extra layer of security by limiting database access to trusted IPs.

Why Whitelisting Matters

- **Network Security:** Prevents unauthorized access by only allowing connections from whitelisted IP addresses.
- **Defense Against DDoS:** Mitigates DDoS attacks by restricting connections to trusted sources.



```
// Helper function to validate input using regex patterns
function validateInput({ username, password, fullName, idNumber, accountNumber }) {
  const usernamePattern = /^[a-zA-Z0-9_]{3,20}$/; // Alphanumeric and underscores, 3-20 characters
  const passwordPattern = /^[a-zA-Z0-9@#$%^&*]{6,20}$/; // Alphanumeric and special chars, 6-20 characters
  const namePattern = /^[a-zA-Z\s]{1,50}$/; // Letters and spaces, up to 50 characters
  const idNumberPattern = /^[0-9]{6,20}$/; // Numeric, 6-20 digits
  const accountNumberPattern = /^[0-9]{6,20}$/; // Numeric, 6-20 digits

  return (
    usernamePattern.test(username) &&
    passwordPattern.test(password) &&
    namePattern.test(fullName) &&
    idNumberPattern.test(idNumber) &&
    accountNumberPattern.test(accountNumber)
  );
}

// Helper function to validate payment data
function validatePaymentData({ amount, currency, provider, accountNumber, swiftCode }) {
  const amountPattern = /^\d+(\.\d{1,2})?$/; // Numeric, allows decimals with up to two decimal places
  const currencyPattern = /^[A-Z]{3}$/; // Three uppercase letters, e.g., USD, EUR
  const providerPattern = /^[A-Za-z]{3,20}$/; // Letters, 3-20 characters
  const accountNumberPattern = /^[0-9]{6,20}$/; // Numeric, 6-20 digits
  const swiftCodePattern = /^[A-Z0-9]{8,11}$/; // Alphanumeric, 8 or 11 characters

  return (
    amountPattern.test(amount.toString()) &&
    currencyPattern.test(currency) &&
    providerPattern.test(provider) &&
    accountNumberPattern.test(accountNumber) &&
    swiftCodePattern.test(swiftCode)
  );
}
```

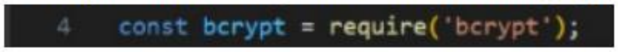
Hashing and Salting

Hashing and salting secure sensitive data, such as passwords, by converting data into fixed-size hash values and adding random data (salt) for uniqueness. In Node.js, bcrypt is commonly used for hashing with salt.

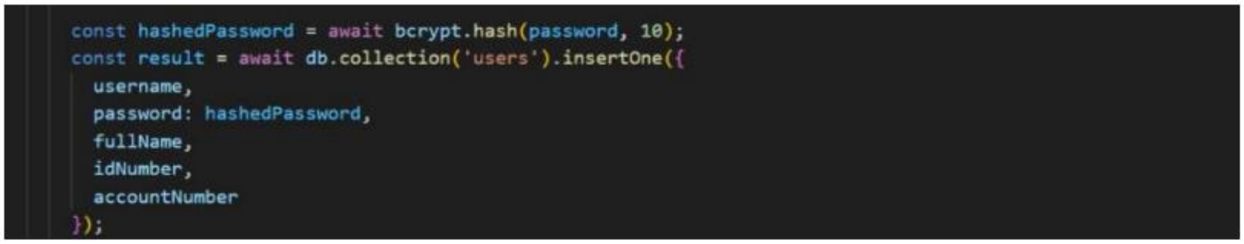
Why Salt Matters

- **Unique Hashes for Same Data:** Even if two users have the same password, salt creates unique hash values.
- **Protection Against Rainbow Tables:** Salt prevents the use of precomputed tables to reverse hashes.

Example of Hashing with Salt in Node.js (using bcrypt):



```
const bcrypt = require('bcrypt');
```



```
const hashedPassword = await bcrypt.hash(password, 10);
const result = await db.collection('users').insertOne({
  username,
  password: hashedPassword,
  fullName,
  idNumber,
  accountNumber
});
```

5. SSL (Secure Sockets Layer)

SSL encrypts data transmitted between the client and the server, protecting it from eavesdropping. In MongoDB Atlas, SSL is enabled by default, ensuring secure data transmission.

Benefits of SSL

- **Data Encryption:** Encrypts data in transit, preventing interception.
- **Server Authentication:** The SSL handshake verifies the server's certificate, ensuring client-server trust.

Brute Force Attack Protection with Express Brute

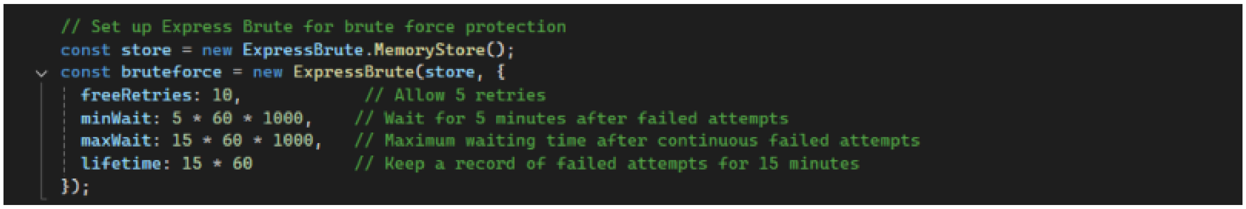
To prevent brute force attacks—automated attempts to guess passwords—Express Brute is implemented as a rate-limiting solution.

Detailed Configuration Breakdown

- **freeRetries: 10** - After 10 failed attempts, restrictions are applied.
- **minWait: 5 minutes** - After exceeding the retry limit, the user must wait at least 5 minutes to try again.
- **maxWait: 15 minutes** - With continued unsuccessful attempts, the wait time increases up to 15 minutes.
- **lifetime: 15 minutes** - Failed attempts are remembered for 15 minutes, resetting the counter after this period.

How It Works in Practice

1. **10 Failed Attempts:** After 10 incorrect logins within 15 minutes, the user is locked out for 5 minutes.
2. **Continued Failure:** If the user fails repeatedly, the wait time escalates up to 15 minutes.
3. **Retry After Timeout:** If the user stops and waits 15 minutes, the counter resets.



```
// Set up Express Brute for brute force protection
const store = new ExpressBrute.MemoryStore();
const bruteforce = new ExpressBrute(store, {
  freeRetries: 10,            // Allow 5 retries
  minWait: 5 * 60 * 1000,    // Wait for 5 minutes after failed attempts
  maxWait: 15 * 60 * 1000,   // Maximum waiting time after continuous failed attempts
  lifetime: 15 * 60          // Keep a record of failed attempts for 15 minutes
});
```

Security Benefits

1. **Slows Down Attackers:** Rate-limiting deters brute force attacks by increasing time costs.
2. **Denial of Service Prevention:** Limits request attempts, preventing denial of service on authentication endpoints.

3. **Configurable Logic:** Allows for adjustable retry limits and waiting times, balancing user experience with security.

References

- MongoDB Documentation. (2023). *Mongoose*. Available at

- Node.js Foundation. (2023). *Node.js Documentation*. Available at
- OWASP Foundation. (2023). *Password Hashing Cheat Sheet*. Available at
- RFC 5246. (2008). *The Transport Layer Security (TLS) Protocol Version 1.2*. Available at
- GDPR. (2023). *General Data Protection Regulation*. Available at
- bcrypt. (2023). *bcrypt Documentation*. Available at
- Kaur, G., & Kaur, A. (2021). *Brute Force Attack and Its Prevention Mechanisms: A Review*. *Journal of Information Technology and Software Engineering, 11*(4), pp. 1-7.