

**ZAEEM PATEL ST10201991**

**THASLYN GOVENDER ST10133946**

**RYLAN NEWMAN ST10190421**

**LIAM COLE ABRAHAM ST10144656**

**UBAIDULLAH YUSEF SHAIK ST10232176**

Why MongoDB Atlas? .....	4
3. Asynchronous Programming in Node.js .....	4
4. Error Handling and Resilience.....	5
5. Why Mongoose is Preferred Over Native MongoDB Driver .....	5
Whitelisting in MongoDB Atlas .....	6
Why Whitelisting Matters:.....	6
How to Configure Whitelisting in MongoDB Atlas: .....	7
2. Hashing and Salting.....	7
Hashing .....	7
Salting .....	8
Why Salt Matters:.....	8
3. SSL (Secure Sockets Layer) .....	9
Benefits of SSL: .....	9
Enabling SSL in MongoDB: .....	9
SSL Certificate Management: .....	10
Brute Force Attack Protection with Express Brute: .....	10
Detailed Configuration Breakdown:.....	10
How It Works in Practice:.....	11
Security Benefits:.....	11

## 1. MongoDB Connection with Mongoose

A NoSQL database called MongoDB uses flexible documents that resemble JSON to store data. To construct schemas and work with this data more conveniently, Mongoose is employed (MongoDB Documentation, 2023).

### Key Features of Mongoose:

- **Schema Definition:** Mongoose offers an application data modeling method based on schemas. Schemas specify the document structure and map to MongoDB collections.
- **Model-Based Interaction:** Models can be developed to engage in structured data interaction after schemas have been established.
- **Middleware and Hooks:** Mongoose supports middleware functions (such validation and saving) that execute at specified times by allowing pre- and post-operation hooks.

By using Mongoose instead of directly interacting with the MongoDB driver (MongoClient), you get built-in validation, casting, query building, and business logic hooks, which significantly simplify application development (MongoDB Documentation, 2023).

```
17
18 // MongoDB connection URI
19 const uri = process.env.MONGO_URI; // Ensure MONGO_URI is set in your .env file
20 const client = new MongoClient(uri);
21
22 // Connect to MongoDB
23 let db;
```

The options `useNewUrlParser` and `useUnifiedTopology` ensure that MongoDB connections are made with the latest URL string parser and unified topology layer, which are more stable and performant.

## 2. Using MongoDB Atlas

The connection URI provided points to **MongoDB Atlas**, a cloud-hosted version of MongoDB. Atlas provides several benefits for developers:

- **Scalability:** Atlas provides flexibility as your application expands by dynamically scaling your database clusters based on use.

- **Security:** SSL encryption, IP whitelisting, and integrated user authentication are just a few of the built-in security features of Atlas. The documentation for MongoDB, 2023.
- **Global Distribution:** With MongoDB Atlas, developers can disperse their databases over the world to give users in various locations low-latency reads and writes.

### ***Why MongoDB Atlas?***

- **Managed Service:** Atlas removes the need for manual database management (e.g., updates, backups, scaling).
- **Multi-Cloud:** Atlas can be deployed across AWS, Azure, or Google Cloud, making it adaptable to different cloud ecosystems.
- **Built-In Monitoring:** Atlas provides extensive monitoring and alerting tools that track performance, replication, and other critical factors.

## **3. Asynchronous Programming in Node.js**

- Because it is event-driven and non-blocking, Node.js performs asynchronous operations efficiently. In this instance, `async/await` is used to manage the MongoDB connection process asynchronously. This makes sure that while the connection is being formed, the application doesn't impede other processes.

### **What Makes Something Asynchronous?**

- **I/O that isn't blocked:** Asynchronous code execution enables the program to carry out additional actions without having to wait for I/O-bound processes, such as database connections, to finish.
- **Scalability:** By allowing the software to manage several tasks at once, asynchronous programming enhances throughput and reaction times. (Foundation for Node.js, 2023).

```

25  async function connectToMongoDB() {
26      try {
27          await client.connect();
28          db = client.db('Apds123'); // Database name
29          console.log('Connected to MongoDB successfully');
30
31          // Start the Server here, after db is initialized
32          app.listen(port, () => {
33              console.log('Server running at http://localhost:${port}');
34          });
35      } catch (err) {
36          console.error('Failed to connect to MongoDB', err);
37          process.exit(1); // Exit the process with failure
38      }
39  }

```

The try-catch block is used to handle errors during the connection attempt. If the connection fails, the process exits with a failure code 1.

## 4. Error Handling and Resilience

The code includes error handling to ensure that the application gracefully handles failures to connect to MongoDB.

- **Graceful Shutdown:** If the program fails to create a database connection, it will depart gracefully by executing `process.exit(1)`. In production systems, when you want to fail quickly and prevent running a service without a database connection, this is a crucial pattern.
- **Logging:** When errors such as MongoDB connection failures are recorded to aid in problem diagnosis. For production-grade apps to troubleshoot problems like credential errors, network outages, or database misconfigurations, proper logging is essential. (Foundation for Node.js, 2023).

## 5. Why Mongoose is Preferred Over Native MongoDB Driver

- Although you could communicate with MongoDB using the built-in `MongoClient`, Mongoose provides higher-level features that make development easier:
- **Schema Validation:** In order to stop invalid documents from being inserted, Mongoose schemas validate data before saving.
- **Getters/Setters and Virtuals:** Computed fields that aren't kept in MongoDB can be used with virtual properties. Custom logic for reading and writing document fields is made possible using getters and setters.

- Mongoose has middleware support that enables you to execute pre- and post-hooks on document activities, such as pre-save and post-update. For jobs like hashing passwords, logging, or sending notifications, this is helpful.

## Whitelisting in MongoDB Atlas

Whitelisting is a security mechanism that controls which IP addresses are allowed to access your MongoDB instance. Only the IPs in your whitelist can connect, providing an additional layer of security. (MongoDB Documentation, 2023).

### Why Whitelisting Matters:

- Network Security: Whitelisting makes sure that the database can only be accessed by trusted IP addresses. As a result, there is less chance of unwanted access or possible attacks from unidentified sources.
- Defense Against DDoS: By denying connections from untrusted IP addresses, it lessens the impact of Distributed Denial of Service (DDoS) attacks. The documentation for MongoDB, 2023.
- IP Management: When working with dynamic cloud infrastructure, it is useful to be able to dynamically manage the list of permitted IPs.

```

44 // Helper function to validate input using regex patterns
45 function validateInput({ username, password, fullName, idNumber, accountNumber }) {
46   const usernamePattern = /^[a-zA-Z0-9_]{3,20}$/; // Alphanumeric and underscores, 3-20 characters
47   const passwordPattern = /^[a-zA-Z0-9@#$%^&*]{6,20}$/; // Alphanumeric and special chars, 6-20 characters
48   const namePattern = /^[a-zA-Z\s]{1,50}$/; // Letters and spaces, up to 50 characters
49   const idNumberPattern = /^[0-9]{6,20}$/; // Numeric, 6-20 digits
50   const accountNumberPattern = /^[0-9]{6,20}$/; // Numeric, 6-20 digits
51
52   return (
53     usernamePattern.test(username) &&
54     passwordPattern.test(password) &&
55     namePattern.test(fullName) &&
56     idNumberPattern.test(idNumber) &&
57     accountNumberPattern.test(accountNumber)
58   );
59 }
60
61 // Helper function to validate payment data
62 function validatePaymentData({ amount, currency, provider, accountNumber, swiftCode }) {
63   const amountPattern = /^\d+(\.\d{1,2})?$/; // Numeric, allows decimals with up to two decimal places
64   const currencyPattern = /^[A-Z]{3}$/; // Three uppercase letters, e.g., USD, EUR
65   const providerPattern = /^[A-Za-z]{3,20}$/; // Letters, 3-20 characters
66   const accountNumberPattern = /^[0-9]{6,20}$/; // Numeric, 6-20 digits
67   const swiftCodePattern = /^[A-Z0-9]{8,11}$/; // Alphanumeric, 8 or 11 characters
68
69   return (
70     amountPattern.test(amount.toString()) &&
71     currencyPattern.test(currency) &&
72     providerPattern.test(provider) &&
73     accountNumberPattern.test(accountNumber) &&
74     swiftCodePattern.test(swiftCode)
75   );
76 }

```

## ***How to Configure Whitelisting in MongoDB Atlas:***

In MongoDB Atlas, you configure the IP whitelist as follows:

1. **Navigate to Security Settings** in your MongoDB Atlas dashboard.
  - Add IP Addresses: CIDR (Classless Inter-Domain Routing) ranges or individual IP addresses can be specified. All IP addresses in the 192.168.1.x range, for instance, are allowed to use 192.168.1.0/24.
  - Permit Access from Anywhere: This is an optional feature that is not advised for production environments. If your IP addresses are dynamic or you're testing locally, you can use 0.0.0.0/0 to grant access from any location. However, because it exposes your database to the internet, this is not secure for production use.

Top Techniques:

- Dynamic IP Allowances: If your IP changes regularly, use dynamic DNS or VPNs to control access.
- Multi-Region Whitelisting: You should set up the whitelist appropriately if your teams or services are dispersed throughout multiple areas.
- Regular Review: Check the IP whitelist from time to time to make sure that no superfluous or out-of-date IPs are still permitted.

## **2. Hashing and Salting**

In applications dealing with user authentication or sensitive data (such as passwords), hashing and salting are fundamental security measures to ensure data integrity and protection against attacks. (OWASP Foundation, 2023).

### ***Hashing***

Hashing is the process of converting data (like a password) into a fixed-size string of characters, which is typically a hash value. Once hashed, the data cannot be reversed (unlike encryption, which is reversible). The hashed value is stored instead of the original data.

**Common Hashing Algorithms:**

- **SHA-256:** One of the most commonly used cryptographic hash functions. It's part of the SHA-2 family and provides strong security.
- **bcrypt:** A slow hash function that includes a built-in salt generation mechanism. It is particularly well-suited for storing passwords due to its computational expense (brute force attacks take much longer). (bcrypt, 2023).

## Salting

A salt is random data added to the original data before hashing to make it unique, even if two users have the same password. This prevents attackers from using precomputed tables (called "rainbow tables") to reverse the hashes. (OWASP Foundation, 2023).

## Why Salt Matters:

- **Unique Hashes for Same Data:** Without salting, identical passwords would always result in identical hashes. With salt, even if two users have the same password, their hashed values will differ.
- **Protection Against Rainbow Table Attacks:** Rainbow tables store precomputed hash values for a large set of potential passwords. Adding salt to the password before hashing thwarts the use of these tables.

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

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

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

```
hashPassword('myPassword123')
  .then(hashedPassword => console.log(`Hashed Password:
${hashedPassword}`));
```

- **bcrypt.genSalt(saltRounds)** generates a salt using the specified number of rounds.
- **bcrypt.hash(password, salt)** hashes the password with the generated salt.



- `bcrypt.compare(password, hashedPassword)` compares a plain password with the hashed version to verify if they match.

#### **Best Practices:**

- Use **bcrypt** with a high number of salt rounds (e.g., 10-12). The more rounds, the slower the hashing function, but it adds security.
- Avoid using fast hash functions (e.g., MD5, SHA1) for passwords, as they are vulnerable to brute force attacks.
- Store both the hash and the salt in your database.

### **3. SSL (Secure Sockets Layer)**

SSL (now known as TLS, Transport Layer Security) is a protocol that encrypts data sent over the internet to ensure that sensitive information is secure between a client and a server. For a MongoDB instance, enabling SSL is crucial for protecting data during transmission, especially when using a cloud service like MongoDB Atlas. (RFC 5246, 2008).

#### ***Benefits of SSL:***

- **Data Encryption:** SSL encrypts all data transmitted between the client (e.g., a web browser or app) and the database, protecting it from eavesdropping and man-in-the-middle attacks.
- **Server Authentication:** The SSL handshake ensures that the client is talking to the legitimate server by verifying the server's certificate. (RFC 5246, 2008).
- **Regulatory Compliance:** Many regulations, such as GDPR or HIPAA, require SSL/TLS encryption to protect personal and sensitive data in transit. (GDPR, 2023).

#### ***Enabling SSL in MongoDB:***

In MongoDB Atlas, SSL is enabled by default. Here's how to ensure you're using it in your connection string:

##### **1. MongoDB Atlas SSL Connection String:**

- a. MongoDB Atlas automatically includes the SSL parameters in your connection URI.
- b. Example:

`mongodb+srv://username:password@cluster.mongodb.net/test?retryWrites=true&w=majority&ssl=true`

2. **Using SSL in Node.js with Mongoose:** When connecting to MongoDB from a Node.js application, you can ensure SSL is being used by adding the `ssl=true` option in the URI string or explicitly setting the SSL option when creating the connection.

### Example:

```
16 // MongoDB connection URI
17 const uri = "mongodb+srv://liamcole0705:Apds12345@cluster0.ezqrt.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0";
18 const client = new MongoClient(uri);
19
```

### SSL Certificate Management:

- **Self-Signed Certificates:** When using a self-hosted MongoDB instance (not MongoDB Atlas), you can create a self-signed certificate for testing purposes. However, in production, always use a certificate from a trusted Certificate Authority (CA).
- **CA-Signed Certificates:** In production environments, MongoDB Atlas uses CA-signed certificates to ensure security and authenticity. This is the standard for web applications requiring secure connections.

### Brute Force Attack Protection with Express Brute:

In a brute force attack, numerous attempts are made to guess a password or get around authentication by experimenting with various combinations. Attackers submit several login attempts quickly by automating this process.

Express Brute employs rate-limiting by monitoring the quantity of requests made within a specific time frame in order to thwart such assaults. A user is momentarily locked out and must wait before attempting again if they attempt more times than is permitted inside the allotted time frame.

### Detailed Configuration Breakdown:

- **freeRetries: 10:**
  - After 10 unsuccessful login attempts, the client will start facing restrictions.
- **minWait: 5 \* 60 \* 1000 (5 minutes):**

- After exceeding the retry limit, the user will need to wait at least 5 minutes before making another attempt.
- **maxWait: 15 \* 60 \* 1000 (15 minutes):**
  - If multiple unsuccessful attempts continue, the waiting time will increase up to a maximum of 15 minutes between login attempts. This prevents continuous retry attempts.
- **lifetime: 15 \* 60 (15 minutes):**
  - Express Brute will remember the failed attempts for 15 minutes. If the user stops trying and waits 15 minutes, the failed attempts counter will be reset.

```
// 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
});
```

## How It Works in Practice:

1. **Scenario 1:**
  - a. A user enters their login credentials incorrectly 10 times in a row within 15 minutes.
  - b. The protection kicks in, and the user is locked out for 5 minutes before they can try again.
2. **Scenario 2:**
  - a. The user continues to fail even after the first lockout. After the 11th or more failed attempt, the wait time increases progressively up to a maximum of 15 minutes.
3. **Scenario 3:**
  - a. The user stops trying after reaching 10 failed attempts and comes back after 15 minutes. Since the record of failures is only kept for 15 minutes, the retry counter is reset, and the user is given another 10 free attempts.

## Security Benefits:

1. **Slows Down Attackers:**
  - a. By increasing wait times after multiple failed attempts, Express Brute dramatically slows down brute force attacks. It becomes computationally

expensive and time-consuming for an attacker to guess passwords or API keys.

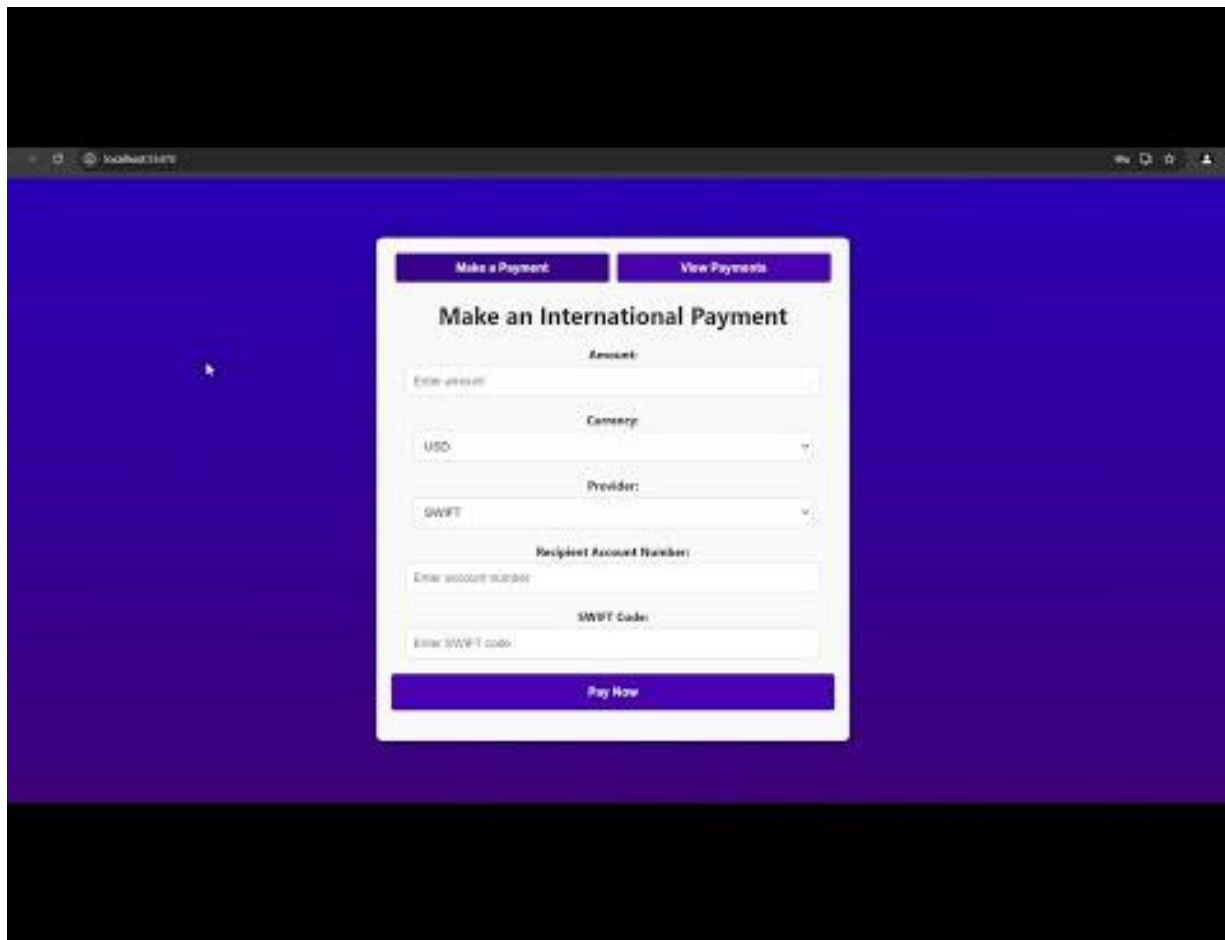
## 2. Prevents Denial of Service:

- a. By limiting the number of attempts within a specific window, Express Brute also mitigates denial of service attempts focused on authentication endpoints.

## 3. Configurable Retry Logic:

- a. The retry limit and waiting times can be adjusted to balance user experience and security. More aggressive settings can be applied for more sensitive operations (e.g., admin logins).

Our Demo: [https://youtu.be/N1\\_YGFrGEEQ](https://youtu.be/N1_YGFrGEEQ)

A screenshot of a web application interface for making an international payment. The interface is displayed within a browser window with a dark theme. The main content area has a dark blue background. A white modal form is centered on the screen. At the top of the modal, there are two buttons: "Make a Payment" and "View Payments". Below these buttons, the title "Make an International Payment" is displayed. The form contains several input fields: "Amount:" with a placeholder "Enter amount", "Currency:" with a dropdown menu showing "USD", "Provider:" with a dropdown menu showing "SWIFT", "Recipient Account Number:" with a placeholder "Enter account number", and "SWIFT Code:" with a placeholder "Enter SWIFT code". At the bottom of the form, there is a large blue button labeled "Pay Now".

Github: <https://github.com/ZAEEM-PATEL/APDS7311-PaymentPortal>

## References

- MongoDB Documentation. (2023). Mongoose. Available at: <https://mongoosejs.com/docs/index.html> (Accessed: 8 October 2024).
- Node.js Foundation. (2023). Node.js Documentation. Available at: <https://nodejs.org/en/docs/> (Accessed: 8 October 2024).
- OWASP Foundation. (2023). Password Hashing Cheat Sheet. Available at: [https://cheatsheetseries.owasp.org/cheatsheets/Password\\_Hashing\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/Password_Hashing_Cheat_Sheet.html) (Accessed: 8 October 2024).
- RFC 5246. (2008). The Transport Layer Security (TLS) Protocol Version 1.2. Available at: <https://tools.ietf.org/html/rfc5246> (Accessed: 8 October 2024).
- GDPR. (2023). General Data Protection Regulation. Available at: <https://gdpr-info.eu/> (Accessed: 8 October 2024).
- bcrypt. (2023). bcrypt Documentation. Available at: <https://www.npmjs.com/package/bcrypt> (Accessed: 8 October 2024).
- Kaur, G. and Kaur, A., 2021. Brute Force Attack and Its Prevention Mechanisms: A Review. Journal of Information Technology and Software Engineering, 11(4), pp.1-7.