Week 3.2

Authentication & Databases

In this focused lecture, Harkirat covers crucial `async concepts` —Fetch, callbacks, promises, and async/await—as well as delves into `Authentication` strategies and `Databases`, specifically the `MongoDB` database.

fetch() Method

The `fetch()` method in JavaScript is a modern API that allows you to make network requests, typically to retrieve data from a server. It is commonly used to interact with web APIs and fetch data asynchronously. Here's a breakdown of what the `fetch()` method is and why it's used:

What is the `fetch()` Method?

The `fetch()` method is a built-in JavaScript function that simplifies making HTTP requests. It returns a Promise that resolves to the `Response` to that request, whether it is successful or not.

Why is it Used?

1. **Asynchronous Data Retrieval:**
   - The primary use of the `fetch()` method is to asynchronously retrieve data from a server. Asynchronous means that the code doesn't wait for the data to arrive before moving on. This is crucial for creating responsive and dynamic web applications.
2. **Web API Interaction:**
   - Many web applications interact with external services or APIs to fetch data. The `fetch()` method simplifies the process of making HTTP requests to these APIs.
3. **Promise-Based:**
   - The `fetch()` method returns a Promise, making it easy to work with asynchronous operations using the `.then()` and `.catch()` methods. This promotes cleaner and more readable code.
4. **Flexible and Powerful:**
   - `fetch()` is more flexible and powerful compared to older methods like `XMLHttpRequest`. It supports a wide range of options, including headers, request methods, and handling different types of responses (JSON, text, etc.).

Basic Example:

Here's a basic example of using the `fetch()` method to retrieve data from a server:

```
fetch('<https://api.example.com/data>')
  .then(response => {
    if (!response.ok) {
      throw new Error(`HTTP error! Status: ${response.status}`);
    }
```

```
    return response.json();
  })
  .then(data => {
    console.log('Data from server:', data);
  })
  .catch(error => {
    console.error('Fetch error:', error);
  });
```

In this example, we use `fetch()` to make a GET request to 'https://api.example.com/data', handle the response, and then parse the JSON data. The `.then()` and `.catch()` methods allow us to handle the asynchronous flow and potential errors.

Some Asynchronous Concepts

1. Callback Functions:

Definition:
A callback function is a function that is passed as an argument to another function and is executed after the completion of that function.

Example:

```
function fetchData(callback) {
  // Simulating an asynchronous operation
  setTimeout(() => {
    const data = 'Hello, callback!';
    callback(data);
  }, 1000);
}

// Using the callback function
fetchData(result => {
  console.log(result);
});
```

Relation to `fetch()`:
In older JavaScript code or libraries, callbacks were extensively used for handling asynchronous operations, such as handling the response in the
`.then()` block of `fetch()`.

2. Promises:

Definition:
A Promise is an object representing the eventual completion or failure of an asynchronous operation. It is a more structured and readable way to handle asynchronous code compared to callbacks.

Example:

```
function fetchData() {
  return new Promise((resolve, reject) => {
    // Simulating an asynchronous operation
    setTimeout(() => {
      const success = true;
      if (success) {
        const data = 'Hello, Promise!';
```

```
        resolve(data);
      } else {
        reject('Oops! Something went wrong.');
      }
    }, 1000);
  });
}

// Using the Promise
fetchData()
  .then(result => {
    console.log(result);
  })
  .catch(error => {
    console.error(error);
  });
```

Relation to `fetch()`:
The `fetch()` method returns a Promise. We use `.then()` to handle the resolved value (successful response) and `.catch()` for handling errors.

3. Async/Await:

Definition: `async/await` is a syntactic sugar built on top of Promises, making asynchronous code more readable and easier to write.

Example:

```
async function fetchData() {
  return new Promise(resolve => {
    // Simulating an asynchronous operation
    setTimeout(() => {
      const data = 'Hello, Async/Await!';
      resolve(data);
    }, 1000);
  });
}

// Using async/await
async function fetchDataAndPrint() {
  try {
    const result = await fetchData();
    console.log(result);
  } catch (error) {
    console.error(error);
  }
}

// Invoking the async function
fetchDataAndPrint();
```

Relation to `fetch()`:
In the context of `fetch()`, `async/await` provides a more synchronous-looking code structure when dealing with asynchronous operations, especially when handling responses.

Overall Relationship:

- Callbacks were the traditional way of handling asynchronous code.

- Promises introduced a more structured and readable way to handle async operations.
- `async/await` builds on top of Promises, offering a more synchronous coding style, making asynchronous code look similar to synchronous code.

Example Combining All:

```
function fetchData(callback) {
  setTimeout(() => {
    const data = 'Hello, Callback!';
    callback(data);
  }, 1000);
}

function fetchDataPromise() {
  return new Promise(resolve => {
    setTimeout(() => {
      const data = 'Hello, Promise!';
      resolve(data);
    }, 1000);
  });
}

async function fetchDataAsyncAwait() {
  return new Promise(resolve => {
    setTimeout(() => {
      const data = 'Hello, Async/Await!';
      resolve(data);
    }, 1000);
  });
}

// Using callback
fetchData(result => {
  console.log(result);

  // Using Promise
  fetchDataPromise()
    .then(result => {
      console.log(result);

      // Using Async/Await
      fetchDataAsyncAwait()
        .then(result => {
          console.log(result);
        })
        .catch(error => {
          console.error(error);
        });
    })
    .catch(error => {
      console.error(error);
    });
});
```

In this example, we've shown the use of callback, Promise, and Async/Await together. Async/Await provides a cleaner and more readable way to structure asynchronous code, especially when dealing with multiple async operations.

4. Try Catch Blocks

In JavaScript and many other programming languages, a `try-catch` block is a mechanism for handling exceptions or errors in a structured way. This construct is crucial for writing robust and fault-tolerant code.

Purpose:

The primary purpose of a `try-catch` block is to gracefully handle runtime errors or exceptions that may occur during the execution of a program. It allows developers to anticipate potential issues and implement a fallback strategy, preventing abrupt program termination.

Syntax:

The basic syntax of a `try-catch` block is as follows:

```
try {
  // Code that may throw an exception
} catch (error) {
  // Code to handle the exception
}
```

- The `try` block encloses the code that might generate an exception.
- If an exception occurs, the control is transferred to the `catch` block, where the `error` parameter holds information about the exception.

How It Works:

1. **Execution in the Try Block:**
   - Code inside the `try` block is executed sequentially.
   - If an exception occurs at any point, the normal flow of execution is interrupted.
2. **Control Transfer to Catch Block:**
   - When an exception is thrown, control is transferred to the corresponding `catch` block.
   - The `catch` block is responsible for handling the exception.
3. **Exception Handling:**
   - Inside the `catch` block, developers can implement error-handling logic.
   - They can log the error, display a user-friendly message, or take alternative actions to recover from the error.

Example:

```
try {
  // Code that may throw an exception
  const result = 10 / 0; // Division by zero, will throw an exception
  console.log(result); // This line won't be executed
} catch (error) {
  // Code to handle the exception
  console.error('An error occurred:', error.message); // Output: An error occurred: Cannot divide by zero
} finally {
  // Code inside the finally block will execute regardless of whether an exception occurred or not
  console.log('Finally block executed');
}
```

- In this example, a division by zero operation inside the `try` block will throw an exception.
- The control is then transferred to the `catch` block, where an error message is logged.
- The `finally` block, if present, will always execute, providing an opportunity for cleanup or finalization tasks.

Authentication

In a programming context, authentication refers to the process of validating the identity of a user, system, or application

attempting to access a computer system, network, or online service. The primary goal is to ensure that the entity requesting access is indeed who it claims to be. Authentication is a crucial aspect of software development, especially in scenarios where user access to sensitive data or functionalities needs to be controlled. Here's how authentication is typically implemented in programming:

1. Hashing:

Purpose:

- Hashing is a one-way process that converts a password or any data into a fixed-size string of characters, which is typically a hash value. The primary purpose of hashing passwords before storing them in a database is to enhance security.

How it Works:

- When a user signs up and provides a password, the application hashes the password using a cryptographic hash function (e.g., bcrypt, SHA-256).
- The resulting hash is a fixed-length string unique to the input, making it difficult to reverse engineer the original password.

Why Hash Passwords:

- **Security:** Hashing prevents storing plaintext passwords in the database, reducing the risk of data breaches. Even if the database is compromised, attackers only obtain hashed values, which are challenging to convert back to the original passwords.

Example in Node.js using bcrypt:

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

// Hashing a password
const plainPassword = 'user123';
bcrypt.hash(plainPassword, 10, (err, hash) => {
  if (err) throw err;
  console.log('Hashed Password:', hash);

  // Verify a password
  bcrypt.compare('user123', hash, (err, result) => {
    if (err) throw err;
    console.log('Password Match:', result);
  });
});
```

2. Encryption:

Purpose:

- Unlike hashing, encryption is a two-way process that involves converting data into a format that can be easily reversed using a decryption key. Encryption is used to protect the confidentiality of data.

How it Works:

- Users' sensitive information (e.g., credit card details) may be encrypted before storing it in a database.
- To view or use the original data, a decryption key is required.

Why Use Encryption:

- **Confidentiality:** Encrypting sensitive data adds an extra layer of security. Even if unauthorized access occurs, the data remains unreadable without the decryption key.

Example in Node.js using crypto:

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

// Encryption
const dataToEncrypt = 'Sensitive information';
const encryptionKey = 'secretKey';
const cipher = crypto.createCipher('aes-256-cbc', encryptionKey);
let encryptedData = cipher.update(dataToEncrypt, 'utf-8', 'hex');
encryptedData += cipher.final('hex');
console.log('Encrypted Data:', encryptedData);

// Decryption
const decipher = crypto.createDecipher('aes-256-cbc', encryptionKey);
let decryptedData = decipher.update(encryptedData, 'hex', 'utf-8');
decryptedData += decipher.final('utf-8');
console.log('Decrypted Data:', decryptedData);
```

3. JSON Web Tokens (JWT)

A JSON Web Token, or JWT, is like a digital passport for information. It's a special kind of code that carries details about a user or some data. Imagine you have a passport when you travel to different countries – the passport holds your information and proves who you are. Similarly, a JWT carries information and proves certain things about you or the data it holds.

How Does JWT Look?

A JWT is made up of three parts, and they are separated by dots:

1. **Header:** This part says how the JWT is encoded (like secret coding instructions).
2. **Payload:** This part holds the actual information or claims. For example, it might say who you are and when the JWT was created.
3. **Signature:** This part ensures that the JWT hasn't been tampered with. It's like a seal that shows the information is genuine.

When you put these parts together, you get a long string that looks like a secret code.

How Do JWTs Work?

1. **Getting the JWT:**
   - Imagine you log in to a website. After you enter your username and password, the website creates a JWT just for you.
2. **Using the JWT:**
   - Now, instead of asking you for your username and password every time you click on something, the website sends your JWT with each request. It's like having a special pass – once you show it, the website knows it's you.
3. **Checking the JWT:**
   - The website has a special key to check if the JWT is real. If everything is okay, the website knows the information in the JWT is trustworthy.

4. Local Storage

Local Storage is a client-side web storage mechanism that allows websites to store key-value pairs persistently on a user's device. In the realm of authentication, Local Storage often plays a crucial role in maintaining user sessions and preserving authentication tokens.

Technical Implementation:

1. **Token Storage:**
   - After a successful authentication, the server generates an authentication token (e.g., JWT) for the user.
   - This token is securely stored in the Local Storage of the user's browser.
2. **Session Persistence:**
   - Local Storage provides a means to persistently store this token across browser sessions. This persistence ensures that

the user remains authenticated even if they close the browser and return later.

3. **Reducing Authentication Overhead:**
    - Instead of requiring users to authenticate themselves on every interaction, the stored token allows the server to recognize and validate the user swiftly, enhancing the user experience.

Benefits of Local Storage in Authentication:

1. **Efficient Session Management:**
    - Local Storage facilitates efficient session management by enabling the storage of authentication tokens client-side. This reduces the need for frequent server-side authentication checks.
2. **Improved Performance:**
    - Since authentication tokens are readily available locally, the authentication process becomes faster, contributing to an improved overall performance of the application.
3. **Enhanced User Experience:**
    - Users experience the convenience of being automatically recognized and authenticated without the hassle of repeated logins, contributing to a seamless and user-friendly interface.

> Local Storage serves as a valuable tool in the authentication landscape, contributing to efficient session management and enhanced user experiences. However, its use should be tempered with a keen awareness of security considerations, adherence to best practices, and a strategic approach to token management.

Authorization Header

Authorization header is a crucial component of HTTP requests that plays a key role in authenticating and authorizing users or clients to access certain resources on a server.

The Authorization header is used to transmit credentials (such as tokens or API keys) from the client to the server. These credentials are then verified by the server to determine whether the client has the necessary permissions to access the requested resource.

The Authorization header typically follows this basic structure:

```
Authorization: <type> <credentials>
```

- **Type:** Specifies the type of credentials being sent. Common types include "Bearer" for token-based authentication and "Basic" for basic authentication.
- **Credentials:** The actual credentials, which could be a token, username and password combination, or other relevant information, depending on the chosen authentication type.

Bearer Token Authentication Type:

- Example: `Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...`
- Used in token-based authentication (e.g., JSON Web Tokens or OAuth), where a token represents the user's identity and permissions.

How it Works:

1. **Client Request:**
    - The client includes the Authorization header in an HTTP request when accessing a secured resource.

2. **Server Verification:**
   - The server receives the request and extracts the credentials from the Authorization header.
3. **Credential Verification:**
   - The server verifies the credentials, usually by checking against a user database, validating a token, or using other authentication mechanisms.
4. **Access Decision:**
   - Based on the verification result, the server decides whether to grant or deny access to the requested resource.

Example Code (Node.js using Axios):

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

const url = '<https://api.example.com/resource>';
const token = 'your-access-token';

axios.get(url, {
  headers: {
    'Authorization': `Bearer ${token}`
  }
})
  .then(response => console.log(response.data))
  .catch(error => console.error('Error:', error));
```

In this example, the Bearer token is included in the Authorization header of an Axios HTTP GET request.

Cookies vs. Local Storage for Storing JWT Tokens:

When it comes to storing JWT (JSON Web Tokens), both cookies and local storage are commonly used, but they have distinct characteristics and use cases. Here's a comparison to help you choose the appropriate option for your specific scenario:

1. Cookies:

- **Security:**

  Cookies can be more secure than local storage because they have an additional security feature called HTTP-only. When a cookie is marked as HTTP-only, it cannot be accessed by JavaScript, reducing the risk of cross-site scripting (XSS) attacks.

- **Automatic Handling:**

  Cookies are automatically sent with every HTTP request to the domain, including requests for images, stylesheets, and scripts. This automatic handling can be advantageous for authenticating API requests made by the browser.

- **Expiration:**

  Cookies can have an expiration date, allowing the server to set a specific duration for which the token is valid. After expiration, the browser automatically removes the cookie.

- **Domain Restriction:**

  Cookies can be set to be domain-restricted, meaning they are only sent to the server from the same domain that set the cookie. This provides a level of security.

2. Local Storage:

- **Ease of Use:**

  Local storage is easier to use from a JavaScript perspective. You can set, get, and remove items directly using JavaScript without additional HTTP requests.

- **Capacity:**

  Local storage generally has a larger storage capacity compared to cookies.

- **No Automatic Handling:**

  Unlike cookies, local storage data is not automatically sent with every HTTP request. This means you need to manually attach the token to the headers of your API requests if you're using it for authentication.

- **No Expiration Handling:**

  Local storage does not provide built-in expiration handling. If you want to implement token expiration, you need to manage it manually in your code.

Choosing Between Cookies and Local Storage:

- **For Authentication:**

  Use cookies with HTTP-only flag for enhanced security, especially if you need to make authenticated API requests directly from the browser.

- **For Client-Side Interactions:**

  Use local storage if you primarily need to access the token on the client side and manage API requests manually.

- **Considerations:**

  Consider factors like security, automatic handling, and token expiration requirements when making your decision.

> In many cases, a combination of both cookies and local storage might be used. Cookies can be employed for secure, HTTP-only storage, while local storage can be used for easy client-side access.

fetch() vs axios()

Fetch API

1. **Native Browser API:**
   - `fetch` is a native JavaScript function built into modern browsers for making HTTP requests.
2. **Promise-Based:**
   - It returns a Promise, allowing for a more modern asynchronous coding style with `async/await` or using `.then()`.
3. **Lightweight:**
   - `fetch` is lightweight and comes bundled with browsers, reducing the need for external dependencies.

Example Usage:

```
fetch('<https://api.example.com/data>')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error('Error:', error));
```

Axios

1. **External Library:**
   - Axios is a standalone JavaScript library designed to work in both browsers and Node.js environments.
2. **Promise-Based:**
   - Similar to `fetch`, Axios also returns a Promise, providing a consistent interface for handling asynchronous operations.
3. **HTTP Request and Response Interceptors:**
   - Axios allows the use of interceptors, enabling the modification of requests or responses globally before they are handled by `then` or `catch`.
4. **Automatic JSON Parsing:**
   - Axios automatically parses JSON responses, simplifying the process compared to `fetch`.

Example Usage:

```
import axios from 'axios';

axios.get('<https://api.example.com/data>')
  .then(response => console.log(response.data))
  .catch(error => console.error('Error:', error));
```

Comparison Points

1. **Syntax:**
   - `fetch` uses a chain of `.then()` to handle responses, which might lead to a more verbose syntax. Axios, on the other hand, provides a concise syntax with `.then()` directly on the Axios instance.
2. **Handling HTTP Errors:**
   - Both `fetch` and Axios allow error handling using `.catch()` or `.finally()`, but Axios may provide more detailed error information by default.
3. **Interceptors:**
   - Axios provides a powerful feature with interceptors for both requests and responses, allowing global modifications. `fetch` lacks built-in support for interceptors.
4. **Request Configuration:**
   - Axios allows detailed configuration of requests through a variety of options. `fetch` may require more manual setup for headers, methods, and other configurations.
5. **JSON Parsing:**
   - Axios automatically parses JSON responses, while with `fetch`, you need to manually call `.json()` on the response.
6. **Browser Support:**
   - `fetch` is natively supported in modern browsers, but if you need to support older browsers, you might need a polyfill. Axios has consistent behavior across various browsers and does not rely on native implementations.
7. **Size:**
   - `fetch` is generally considered lightweight, being a part of the browser. Axios, being a separate library, introduces an additional file size to your project.

Conclusion

- **Use `fetch` when:**
  - Working on a modern project without the need for additional features.
  - Prefer a lightweight solution and have no concerns about polyfills.
- **Use Axios when:**
  - Dealing with more complex scenarios such as interceptors.
  - Needing consistent behavior across different browsers.
  - Desiring a library with built-in features like automatic JSON parsing.

> In summary, both `fetch` and Axios have their strengths, and the choice depends on the specific requirements and preferences of the project. `fetch` is excellent for simplicity and lightweight projects, while Axios provides additional features and consistent behavior across different environments.

Databases

Until now, we've been storing data in memory. This is bad for a few reasons:

1. Data can't be dynamic:

- When data is stored in memory, it becomes volatile. Any updates or changes made to in-memory objects are temporary and get lost if the process restarts. In a real-world application, this limitation is significant because processes may restart due to various reasons, such as server maintenance, deployments, or unexpected crashes. As a result, any dynamically updated information will be lost, leading to inconsistencies and potential data loss.

Example:

- Imagine an application that keeps track of user sessions or preferences. If this data is only stored in memory and the server restarts, all the user-related information would be reset, impacting the user experience.

2. There are multiple servers in the real world:

- In a real-world application, especially those handling a significant user load or operating across multiple servers, relying solely on in-memory storage is impractical. Multiple servers may be used to distribute the load, enhance performance, and ensure high availability. When data is confined to the memory of a single server, it becomes challenging to maintain consistency and share data across the entire application infrastructure.

Example:

- Consider an e-commerce platform with multiple servers handling user requests. If each server maintains its own set of in-memory data, it becomes challenging to synchronize and share information seamlessly across all servers, leading to potential discrepancies in the displayed data.

> Using databases, not just memory, is vital for strong, scalable apps. They ensure lasting, shared data across servers, maintaining consistency even after restarts.

Types Of Databases

1. **Graph Databases:**

   Graph databases specialize in representing and navigating relationships between entities, making them ideal for applications emphasizing connected data. *Example:* Neo4j

2. **Vector Databases:**

   Vector databases efficiently handle multidimensional and spatial relationships, particularly suited for applications dealing with spatial data. *Example:* InfluxDB

3. **SQL Databases:**

   SQL databases follow a structured query language, maintaining a tabular structure for organized data storage, retrieval, and manipulation. *Example:* PostgreSQL

4. **NoSQL Databases:**

   NoSQL databases offer flexibility in data modeling and are suitable for applications with evolving and diverse data needs. *Example:* MongoDB

Today's class will delve into MongoDB, a famous NoSQL database, exploring its features and applications in the world of data management.

MongoDB

MongoDB is a powerful and versatile NoSQL database that revolutionizes data management with its flexible and scalable design. Here's a breakdown of its key features:

1. **Database Creation:**
   - MongoDB allows users to create multiple databases, acting as distinct containers for organizing and storing data.
2. **Collection Creation:**
   - Within each database, collections serve as the equivalent of tables in relational databases. Collections provide a structured way to group and manage related documents.
3. **JSON Data Storage:**
   - MongoDB adopts a document-oriented data model, storing information in BSON (Binary JSON) format. This facilitates the storage of JSON-like documents in a flexible and readable manner.
4. **Schemaless Design:**
   - Unlike traditional relational databases, MongoDB is schemaless. This means documents within a collection can have varying structures, enabling easy adaptation to changing data requirements without rigid schema constraints.
5. **Scalability:**
   - MongoDB is designed for horizontal scalability, allowing for the distribution of data across multiple servers or clusters. This horizontal scaling ensures optimal performance as data volumes and user loads increase.
6. **Versatility for Most Use Cases:**
   - MongoDB's adaptability makes it a reliable choice for a diverse range of use cases. Whether handling complex data structures or large datasets, MongoDB can efficiently meet the demands of various applications.

> In essence, MongoDB provides a dynamic and scalable solution for modern data storage needs. Its schemaless design, JSON-based documents, and horizontal scalability make it well-suited for applications where flexibility, scalability, and diverse data types are crucial.

Creating a free MongoDB instance

Creating a free MongoDB instance typically involves using MongoDB Atlas, the official cloud-based database service provided by MongoDB. Follow these step-by-step instructions to create a free MongoDB instance using MongoDB Atlas:

1. **Visit MongoDB Atlas:**
   - Open your web browser and go to the MongoDB Atlas website: MongoDB Atlas.
2. **Sign Up or Log In:**
   - If you don't have an account, click on "Sign Up" to create a new account. If you already have an account, log in using your credentials.
3. **Choose a Plan:**
   - Once logged in, click on the "Get Started Free" button to initiate the process of creating a free MongoDB instance.
4. **Fill in the Form:**
   - Provide the required information in the sign-up form. This includes your email, username, and password.
5. **Create an Organization:**
   - After filling in your information, you'll be prompted to create an organization. Enter a name for your organization, and click "Next."
6. **Create a Project:**
   - Inside your organization, you'll create a project. Choose a name for your project, and click "Next."
7. **Create a Cluster:**
   - In the next step, you'll create a cluster. A cluster is a set of servers that will host your MongoDB databases. Choose the free tier (M0 Sandbox), and select your preferred cloud provider and region.
8. **Configure Cluster Settings:**
   - Configure additional settings for your cluster, such as the cluster name, additional features, and whether you want to enable backups. You can stick with the default settings for now.

9. **Create Cluster:**
    - Click the "Create Cluster" button. MongoDB Atlas will start creating your cluster, and this process may take a few minutes.
10. **Wait for Cluster to Deploy:**
    - Once the cluster is created, you'll see it in the MongoDB Atlas dashboard. Wait for the cluster to be deployed and become available.
11. **Access Your Cluster:**
    - Once your cluster is ready, click on the "CONNECT" button. You can then choose to connect using MongoDB Compass (a GUI tool) or connect using your application.
12. **Whitelist Your IP Address:**
    - Before connecting, you need to whitelist your IP address to ensure secure access. Click on the "Add Your Current IP Address" button.
13. **Create a MongoDB User:**
    - Create a MongoDB user by entering a username and password. This user will be used to connect to your MongoDB instance.
14. **Connect to Your Cluster:**
    - After creating the user, click on the "Choose a Connection Method" button and follow the instructions to connect to your MongoDB cluster.

You can now start using MongoDB for your applications.

How does the backend connect to the database?

It does so by Express, JWT and Mongoose Libraries. Let's break down how the backend connects to the database using these libraries:

1. **Express:**
    - **Role:** Creates an HTTP server to handle requests and responses.
    - **Connection to Database:** While Express itself doesn't directly connect to the database, it provides a framework for building the server. Endpoints/routes within Express handle requests, and these routes may involve interactions with the database using other libraries like Mongoose.
2. **Jsonwebtokens (JWT) Library:**
    - **Role:** Allows the creation and verification of JSON Web Tokens (JWT).
    - **Connection to Database:** Typically, JWTs are used for authentication. Once a user is authenticated, the backend can include a JWT in the response. This token can be sent by the client in subsequent requests, allowing the backend to identify and authorize the user without the need to store session information on the server.
3. **Mongoose:**
    - **Role:** An Object Data Modeling (ODM) library for MongoDB and Node.js. It provides a structured way to interact with MongoDB.
    - **Connection to Database:** Mongoose simplifies the process of connecting to MongoDB. It allows defining models, schemas, and provides methods for CRUD (Create, Read, Update, Delete) operations. The connection to the MongoDB database is established using Mongoose, providing a higher-level abstraction for working with MongoDB.

In summary, while Express sets up the server, JWT helps with user authentication, and Mongoose facilitates interaction with the MongoDB database. Together, these libraries form a robust backend infrastructure for handling HTTP requests, securing routes, and managing data in the database.

Mongoose

Mongoose is an Object Data Modeling (ODM) library for MongoDB and Node.js. It provides a higher-level, schema-based abstraction over the MongoDB JavaScript driver. Mongoose acts as a powerful bridge between Node.js applications and MongoDB databases. It streamlines the data modeling process, simplifies interactions with the database, and enhances the overall

development experience when working with MongoDB in a Node.js environment.