Authentication is possible without JWT tokens as well but ..

Why we should use jwt tokens for authentication?

Session-based Authentication: User session data is stored on the server-side (in a database, cache like Redis, or in-memory storage). This data must be synced and accessed by all servers in a distributed system.

The server must maintain session state for each user.
In distributed systems, all servers must share session data, typically requiring a central session store or a caching mechanism, which can become a bottleneck as the number of users grows.

JWT-based Authentication: Tokens are stored on the client-side (in localStorage, sessionStorage, or cookies). The server doesn't need to store or sync token data—it simply verifies the token when it's received.

The server doesn't need to manage or store any user state. Each request carries all necessary authentication data (the token).

This eliminates the need for server-side session synchronization, reducing server load and improving scalability.

Each server independently validates the token using a secret key or public key.

No server needs to query a centralized session store, so there's no dependency or potential bottleneck.

In session-based authentication, scaling infrastructure requires the session store to handle the load.

In token-based authentication, the client-side storage is entirely independent, and scaling the infrastructure doesn't affect token management.

How the JWT Tokens work ?

1. You log in (Client Sends Login Request):

- You (the **client**) provide your credentials (like email and password) and send them to the server.
- The **server** checks if your credentials are correct by looking them up in its database.

---

2. Server Generates a JWT Token:

- If your login is successful, the server creates a **JWT token** containing your user information, such as:
  - Your **user ID**
  - Your **role** (e.g., admin or regular user)
  - An **expiration time** (to ensure the token is valid only for a limited period)
- The server **digitally signs** the token using a **secret key** (known only to the server). This ensures the token cannot be tampered with.

---

3. Token Sent to the Client:

- The generated JWT token is sent back to the client.
- You (the client) store the token in `localStorage`, `sessionStorage`, or cookies.

---

4. You Send the Token with Every Request:

For every subsequent request to the server (e.g., fetching your profile, accessing secure data), you include the token in the `Authorization` header of the HTTP request, like this:

```
Authorization: Bearer <your-JWT-token>
```

-

---

5. Server Verifies the Token:

When the server receives your request with the token:

1. **Decode the Token**:
   - The server extracts the token from the request and decodes it.
   - It checks the data inside the token, such as user ID, role, etc.
2. **Verify the Signature**:
   - The server uses its **secret key** (or a public key if asymmetric encryption is used) to validate the token's signature.

    - This ensures the token hasn't been altered by anyone after it was generated.
3. **Check Expiration**:
    - The server verifies whether the token has expired (by checking the `exp` field in the token).
4. **Trust the Token**:
    - If the token is valid, the server trusts the data inside it and processes your request without looking up any additional information in its database.

---

6. Server Processes the Request:

- Once the token is verified, the server allows you to access the secure resources or data.

Authentication using JWT (step-wise)

JWT (JSON Web Token) authentication is a common method for securing APIs and implementing user authentication. Below is a guide for creating a **signup** and **login** API using JWT.

---

1. Prerequisites

Ensure the following libraries are installed:

```
npm install express jsonwebtoken bcryptjs dotenv mongoose
```

---

2. User Model

Create a model with fields for user details, including a hashed password.

`models/userModel.js`

```
import mongoose from 'mongoose';
import {compareSync,hashSync} from 'bcrypt';

const userSchema = new mongoose.Schema({
  firstname: { type: String, required: true },
  lastname: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  password: { type: String, required: true },
});

// Hash the password before saving the user
userSchema.pre('save', async function (next) {
  if (!this.isModified('password')) return next();
  const salt = await bcrypt.genSalt(10);
  this.password = await bcrypt.hash(this.password, salt);
  next();
});
```

```
const User = mongoose.model('User', userSchema);
export default User;
```

---

3. Auth Controller

Implement logic for signup, login, and generating JWT.

`controllers/authController.js`

```
import User from '../models/userModel.js';
import jwt from 'jsonwebtoken';
import bcrypt from 'bcryptjs';

// Generate JWT token
const generateToken = (id) => {
  return jwt.sign({ id }, process.env.JWT_SECRET, { expiresIn: '1h' });
};

// Signup
export const signup = async (req, res) => {
  try {
    const { firstname, lastname, email, password } = req.body;

    // Check if the user already exists
    const existingUser = await User.findOne({ email });
    if (existingUser) {
      return res.status(400).json({ message: 'User already exists' });
    }

    // Create a new user
    const newUser = await User.create({ firstname, lastname, email, password });

    // Generate a token
    const token = generateToken(newUser._id);

    res.status(201).json({ user: { id: newUser._id, email: newUser.email }, token });
  } catch (error) {
    res.status(500).json({ message: 'Error creating user', error });
  }
};

// Login
```

```
export const login = async (req, res) => {
  try {
    const { email, password } = req.body;

    // Check if the user exists
    const user = await User.findOne({ email });
    if (!user) {
      return res.status(404).json({ message: 'User not found' });
    }

    // Validate password
    const isPasswordValid = await bcrypt.compare(password, user.password);
    if (!isPasswordValid) {
      return res.status(400).json({ message: 'Invalid credentials' });
    }

    // Generate a token
    const token = generateToken(user._id);

    res.status(200).json({ user: { id: user._id, email: user.email }, token });
  } catch (error) {
    res.status(500).json({ message: 'Error logging in', error });
  }
};
```

---

4. Auth Routes

Set up routes for signup and login.

`routes/authRoutes.js`

```
import express from 'express';
import { signup, login } from '../controllers/authController.js';

const router = express.Router();

router.post('/signup', signup); // Route for user signup
router.post('/login', login);   // Route for user login

export default router;
```

---

5. Middleware to Protect Routes

Create middleware to validate JWT and protect routes.

`middlewares/authMiddleware.js`

```
import jwt from 'jsonwebtoken';
import User from '../models/userModel.js';

export const protect = async (req, res, next) => {
  try {
    const token = req.headers.authorization?.split(' ')[1];

    if (!token) {
      return res.status(401).json({ message: 'Not authorized, no token' });
    }

    // Verify token
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.user = await User.findById(decoded.id).select('-password'); // Attach user info to request
    next();
  } catch (error) {
    res.status(401).json({ message: 'Not authorized, token failed' });
  }
};
```

---

6. Integrate in Server

Register the routes and middleware in the main `server.js`.

`server.js`

```
import express from 'express';
import dotenv from 'dotenv';
import cors from 'cors';
import mongoose from 'mongoose';
import authRouter from './routes/authRoutes.js';
import { protect } from './middlewares/authMiddleware.js';

dotenv.config();
```

```
const server = express();

// Middlewares
server.use(cors());
server.use(express.json());

// Connect to DB
mongoose.connect(process.env.MONGO_URL, { useNewUrlParser: true, useUnifiedTopology:
true })
  .then(() => console.log('Database connected'))
  .catch(err => console.log('DB connection error:', err));

// Routes
server.use('/auth', authRouter); // Auth routes

// Example protected route
server.get('/protected', protect, (req, res) => {
  res.status(200).json({ message: 'Welcome to the protected route', user: req.user });
});

// Start the server
server.listen(process.env.PORT, () => {
  console.log(`Server running on <http://localhost>:${process.env.PORT}`);
});
```

---

7. Test the API

Use **Postman** or similar tools to test the endpoints:

1. **Signup**: `POST /auth/signup`

Body:

```
 {
  "firstname": "John",
  "lastname": "Doe",
  "email": "john.doe@example.com",
  "password": "password123"
}
```

    - Response: `{ user, token }`

2. **Login**: `POST /auth/login`

Body:

```
{
  "email": "john.doe@example.com",
  "password": "password123"
}
```

    - Response: `{ user, token }`

3. **Protected Route**: `GET /protected`
    - Add `Authorization` header: `Bearer <token>`
    - Response: `{ message, user }`

Key Points

- **JWT Token**: Securely identifies users for subsequent requests.
- **bcrypt**: Ensures passwords are hashed securely.
- **Protect Middleware**: Validates tokens to secure sensitive routes.

Let me know if you need further assistance! 😊

Understanding AUTH middleware :

Code:

```
export const auth = async (req, res, next) => {

    try {

        const token = req.headers.authorization?.split("Bearer")[1];

        if (!token) {

            return res.status(401).json({ message: 'Not authorized, no token' });

        }

        // Verify token

        const decoded = jwt.verify(token, process.env.JWT_SECRET);

        req.user = await User.findById(decoded.id).select('-password'); // Attach user info to request

        next();

    } catch (error) {

        res.status(401).json({ message: 'Not authorized, token failed' });

    }

};
```

This middleware, named `Auth`, is an Express.js function designed to authenticate requests using a JWT (JSON Web Token). Here's a detailed line-by-line explanation:

---

1. Function Declaration

```
const Auth = (req, res, next) => {
```

- `Auth` is defined as a middleware function.
- It takes three arguments:
  - `req`: Represents the incoming request.
  - `res`: Represents the response to be sent.
  - `next`: A function that allows the request to continue to the next middleware or route handler if authentication succeeds.

---

2. Error Handling with `try-catch`

```
try {
```

- A `try-catch` block is used to handle any runtime errors that occur during the authentication process.

---

3. Retrieving the Authorization Header

```
const authHeader = req.get("Authorization");
```

- This retrieves the `Authorization` header from the incoming HTTP request.
- If the header is not provided, `authHeader` will be `undefined`.

---

4. Checking for Missing Authorization Header

```
if (!authHeader) {
  return res.status(401).json({ error: "Authorization header missing" });
}
```

- If the `Authorization` header is missing (`authHeader` is `undefined` or `null`), the server responds with a `401 Unauthorized` status and an error message.
- The function exits early, preventing further processing.

---

5. Extracting the Token

```
const token = authHeader.split("Bearer ")[1];
```

- The `Authorization` header is expected to contain a string like `Bearer <token>`.
- This splits the string by `"Bearer "` and retrieves the second part (`<token>`), which is the actual JWT.
- If the format is incorrect, `token` will be `undefined`.

---

6. Checking for Missing or Malformed Token

```
if (!token) {
  return res.status(401).json({ error: "Token missing or malformed" });
}
```

- If `token` is `undefined` or `null` (indicating the token is missing or improperly formatted), the server responds with a `401 Unauthorized` status and an appropriate error message.

---

7. Verifying the Token

```
const decoded = jwt.verify(token, process.env.SECRET);
```

- The `jwt.verify()` function is used to decode and verify the token against a secret key stored in `process.env.SECRET`.
- If the token is valid:
  - `decoded` will contain the payload of the JWT (e.g., user data or permissions).
  - The payload is then used to authenticate the user.
- If the token is invalid or expired, an error will be thrown.

---

8. Storing Decoded Data in the Request Object

```
req.user = decoded;
```

- The decoded token payload is attached to the `req.user` property.

- This makes the authenticated user information available to subsequent middleware and route handlers.

---

9. Proceeding to the Next Middleware

next();

- If everything checks out, the `next()` function is called.
- This allows the request to move on to the next middleware or route handler.

---

10. Handling Errors in the `catch` Block

catch (err) {

- Any errors encountered in the `try` block are caught here.

Handling Invalid Token

```
if (err.name === "JsonWebTokenError") {
  return res.status(401).json({ error: "Invalid token" });
}
```

- If the error is due to an invalid token, the server responds with a `401 Unauthorized` status and an "Invalid token" error message.

Handling Expired Token

```
else if (err.name === "TokenExpiredError") {
  return res.status(401).json({ error: "Token expired" });
}
```

- If the error is due to an expired token, the server responds with a `401 Unauthorized` status and a "Token expired" error message.

Handling Other Errors

```
return res.status(500).json({ error: "Internal server error" });
```

- For all other errors, the server responds with a `500 Internal Server Error` status and a generic error message.

Summary

This middleware ensures that:

1. Requests include a properly formatted `Authorization` header with a valid JWT.
2. The token is verified using a secret key.
3. Any errors (e.g., missing header, malformed token, invalid/expired token) are handled appropriately.
4. If authentication is successful, the request proceeds with the user's details stored in `req.user`.