Express and RESTful APIs

What is REST?

REST (Representational State Transfer) is an architectural style for designing networked applications. It relies on a stateless, client-server communication model using HTTP methods like GET, POST, PUT, DELETE, and PATCH. RESTful APIs follow a standardized approach, making them scalable, flexible, and easy to integrate.

1. **GET (Retrieve Data)**
    a. Used to fetch data from a server.
    b. Should not modify the resource.
    c. Idempotent (same request returns the same response).
2. **POST (create data)**
    a. Used to create a new resource.
    b. Sends data in the request body.
    c. Not idempotent (creates a new resource every time).
3. **PUT (Update Data - Full Update)**
    a. Used to completely replace an existing resource.
    b. Requires all fields to be provided in the request body.
    c. Idempotent (same request gives the same result).
4. **DELETE (Remove Data)**
    a. Used to delete a resource.
    b. Idempotent (deleting a non-existent resource should return the same response).
5. **PATCH (Update Data - Partial Update)**
    a. Used to partially update an existing resource.
    b. Requires only the fields that need updating.

How to create a simple To-Do Application API using Node.js, Express, and MongoDB (Mongoose). This API allows users to create, read, update, and delete (CRUD) tasks.

1. Initialize a new Node.js project:

```
mkdir todo-api && cd todo-api
npm init -y
```

2. Install dependencies:

```
npm install express mongoose dotenv cors body-parser
```

a. Mongoose :- ODM (Object Data Modeling) library for MongoDB. Helps with database operations.
b. Dotenv :- Loads environment variables from a .env file, keeping sensitive data safe.
c. Cors :- Middleware to enable Cross-Origin Resource Sharing (CORS) for APIs.
d. Body-parser :- Middleware to parse incoming JSON data in requests.

Project Structure

```
todo-api/
|— server.js
|— .env
|— models/
|    └— Task.js
|— routes/
|    └— taskRoutes.js
|— controllers/
|    └— taskController.js
|— config/
|    └— db.js
```

1. **Database Configuration (config/db.js)**

```
const mongoose = require("mongoose");
// Define an asynchronous function to establish a database connection
const connectDB = async () => {
  try {
    await mongoose.connect('mongodb://localhost:27017/mydatabase', {
// Enables the new Server Discovery and Monitoring engine for better stability
      useUnifiedTopology: true
    });
    console.log("MongoDB Connected...");
  } catch (error) {
    console.error("Database Connection Error:", error);
    process.exit(1);
  }
};
```

2. Task Model (models/Task.js)

```
const mongoose = require("mongoose");
// Define a schema (structure) for the "Task" collection in MongoDB
const taskSchema = new mongoose.Schema({
  title: { type: String, required: true },
  completed: { type: Boolean, default: false },
}, { timestamps: true }); // Automatically adds "createdAt" and "updatedAt" fields


module.exports = mongoose.model("Task", taskSchema);
```

3. Routes (routes/taskRoutes.js)

```
const express = require("express");
const {
  getTasks,
  getTaskById,
  createTask,
  updateTask,
  deleteTask,
} = require("../controllers/taskController");
// Create an Express Router instance
const router = express.Router();


router.get("/", getTasks);
router.get("/:id", getTaskById);
router.post("/", createTask);
router.put("/:id", updateTask);
router.delete("/:id", deleteTask);


module.exports = router;
```

4. Task Controller (controllers/taskController.js)

```
const Task = require("../models/Task");

// Get all tasks
exports.getTasks = async (req, res) => {
  try {
    const tasks = await Task.find();
    res.status(200).json(tasks);
  } catch (error) {
    res.status(500).json({ error: "Error fetching tasks" });
  }
};

// Get a single task by ID
exports.getTaskById = async (req, res) => {
  try {
    const task = await Task.findById(req.params.id);
    if (!task) return res.status(404).json({ error: "Task not found" });
    res.status(200).json(task);
  } catch (error) {
    res.status(500).json({ error: "Error fetching task" });
  }
};

// Create a new task
exports.createTask = async (req, res) => {
  try {
    const newTask = new Task(req.body);
    await newTask.save();
    res.status(201).json(newTask);
  } catch (error) {
    res.status(400).json({ error: "Error creating task" });
  }
};
```

```
// Update a task
exports.updateTask = async (req, res) => {
  try {
    const updatedTask = await Task.findByIdAndUpdate(req.params.id, req.body, { new: true });
    if (!updatedTask) return res.status(404).json({ error: "Task not found" });
    res.status(200).json(updatedTask);
  } catch (error) {
    res.status(500).json({ error: "Error updating task" });
  }
};

// Delete a task
exports.deleteTask = async (req, res) => {
  try {
    const deletedTask = await Task.findByIdAndDelete(req.params.id);
    if (!deletedTask) return res.status(404).json({ error: "Task not found" });
    res.status(200).json({ message: "Task deleted successfully" });
  } catch (error) {
    res.status(500).json({ error: "Error deleting task" });
  }
};
```

5. Server Setup (server.js)

```
const express = require("express");              // Express framework for building APIs
const dotenv = require("dotenv");                // dotenv for loading environment variables
const cors = require("cors");                    // CORS middleware for handling cross-origin requests
const connectDB = require("./config/db");   // Import database connection function
const taskRoutes = require("./routes/taskRoutes");   // Import task routes
dotenv.config();    // Load environment variables from .env file
connectDB();  // Connect to MongoDB database
```

```
// Create an Express application instance
const app = express();


// Middleware
app.use(cors());  // Enables CORS (Cross-Origin Resource Sharing) to allow external API requests
app.use(express.json());  // Parses incoming JSON request bodies


// Routes
app.use("/api/tasks", taskRoutes); // Mounts task-related routes under "/api/tasks"
//Start the server
const PORT = process.env.PORT || 5000; // Sets the port from environment variables or defaults to 5000
app.listen(PORT, () => console.log(`Server running on port ${PORT}`)); // Starts the server and logs the port
```

6. **Environment Variables (.env)**

```
MONGO_URI=mongodb://localhost:27017/todoDB
PORT=5000
```

7. **Running the Project**

```
node server.js
```

create an authentication and authorization API using Node.js, Express, MongoDB, and JWT.

1. **Install Required Packages**

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

2. **Create the Project Structure**

```
/auth-api
|— /config
|   |— db.js              # MongoDB connection
|— /models
|   |— User.js            # User model
|— /routes
|   |— authRoutes.js      # Authentication routes
|— /middleware
|   |— authMiddleware.js  # Middleware for authentication & authorization
|— /controllers
|   |— authController.js  # Controller for authentication
|— .env                   # Environment variables
|— server.js              # Main server file
```

3. Setup MongoDB Connection (config/db.js)

```
const mongoose = require("mongoose");

const connectDB = async () => {
  try {
    await mongoose.connect(process.env.MONGO_URI, {
      useNewUrlParser: true,
      useUnifiedTopology: true,
    });
    console.log("MongoDB Connected...");
  } catch (error) {
    console.error("Database Connection Error:", error);
    process.exit(1);
  }
};

module.exports = connectDB;
```

4. **Create the User Model (models/User.js)**

```
const mongoose = require("mongoose");
const bcrypt = require("bcryptjs");

const UserSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  password: { type: String, required: true },
  role: { type: String, enum: ["user", "admin"], default: "user" },
}, { timestamps: true });

// Hash the password before saving
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();
});

module.exports = mongoose.model("User", UserSchema);
```

5. **Create Authentication Controller (controllers/authController.js)**

```
const User = require("../models/User");
const bcrypt = require("bcryptjs");
const jwt = require("jsonwebtoken");
```

```
// Register a new user
exports.register = async (req, res) => {
  try {
    const { name, email, password, role } = req.body;

    let user = await User.findOne({ email });
    if (user) return res.status(400).json({ error: "User already exists" });

    user = new User({ name, email, password, role });
    await user.save();

    res.status(201).json({ message: "User registered successfully" });
  } catch (error) {
    res.status(500).json({ error: "Error registering user" });
  }
};

// Login user and generate JWT token
exports.login = async (req, res) => {
  try {
    const { email, password } = req.body;

    const user = await User.findOne({ email });
    if (!user) return res.status(400).json({ error: "Invalid credentials" });

    const isMatch = await bcrypt.compare(password, user.password);
    if (!isMatch) return res.status(400).json({ error: "Invalid credentials" });
```

```
    // Generate JWT token
    const token = jwt.sign({ id: user._id, role: user.role },
process.env.JWT_SECRET, { expiresIn: "1h" });


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

6. Create Authentication Middleware (middleware/authMiddleware.js)

```
const jwt = require("jsonwebtoken");


// Middleware to protect routes (Authentication)
exports.protect = (req, res, next) => {
  const token = req.header("Authorization");
  if (!token) return res.status(401).json({ error: "Access denied" });


  try {
    const decoded = jwt.verify(token.replace("Bearer ", ""),
process.env.JWT_SECRET);
    req.user = decoded;
    next();
  } catch (error) {
    res.status(401).json({ error: "Invalid token" });
  }
};
```

```
// Middleware to authorize admin users
exports.admin = (req, res, next) => {
  if (req.user.role !== "admin") {
    return res.status(403).json({ error: "Access forbidden: Admins only" });
  }
  next();
};
```

7. Create Routes (routes/authRoutes.js)

```
const express = require("express");
const { register, login } = require("../controllers/authController");
const { protect, admin } = require("../middleware/authMiddleware");

const router = express.Router();

router.post("/register", register);
router.post("/login", login);

// Example: Protected route for Admins only
router.get("/admin", protect, admin, (req, res) => {
  res.json({ message: "Welcome Admin" });
});

module.exports = router;
```

8. **Setup Server (server.js)**

```
require("dotenv").config();
const express = require("express");
const cors = require("cors");
const connectDB = require("./config/db");
const authRoutes = require("./routes/authRoutes");

const app = express();

// Connect to Database
connectDB();

// Middleware
app.use(cors());
app.use(express.json());

// Routes
app.use("/api/auth", authRoutes);

// Start Server
const PORT = process.env.PORT || 5000;
app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

9. **Create .env File**

```
PORT=5000
MONGO_URI=mongodb://localhost:27017/auth-api
JWT_SECRET=your_jwt_secret_key
```

10. How to Test the API

a. **Start MongoDB**:

```
mongod
```

b. Run the server:

```
node server.js
```

c. Test using Postman or cURL:

Register a User:

```
POST /api/auth/register
{
  "name": "John Doe",
  "email": "john@example.com",
  "password": "123456",
  "role": "admin"
}
```

Login to get JWT Token:

```
POST /api/auth/login
{
  "email": "john@example.com",
  "password": "123456"
}
```

Access Admin Route (Requires Token in Headers)

```
GET /api/auth/admin
Authorization: Bearer <your-jwt-token>
```

Authentication vs Authorization

Both Authentication and Authorization are crucial for securing applications, but they serve different purposes.

Authentication (Who are you?)

Authentication is the process of **verifying a user's identity** before granting access to a system.

- **Purpose:** Confirms that users are who they claim to be.
- **How?** Using **usernames, passwords, OTPs, biometrics, etc.**
- **Example:**
    - You enter your **email & password** to log into a website.
    - The system checks if the credentials are valid.

🔹 **Common Authentication Methods:**

✔️ Username & Password
✔️ OTP (One-Time Password)
✔️ Biometric (Fingerprint, Face ID)
✔️ OAuth (Google, Facebook Login)
✔️ JWT (JSON Web Token)

Authorization (What can you do?)

Authorization is the process of **determining what a user is allowed to do** after authentication.

- **Purpose:** Controls access to resources based on user permissions.
- **How?** Using **roles, permissions, access control lists (ACLs), etc.**
- **Example:**
    - A **user** logs into an admin dashboard.
    - If the user is **admin**, they can **delete users**.
    - If the user is **normal**, they can **only view content**.

🔹 **Common Authorization Methods:**

✔️ Role-Based Access Control (**RBAC**)
✔️ Attribute-Based Access Control (**ABAC**)
✔️ OAuth Scopes
✔️ JSON Web Token (**JWT**)

Session in Express

A session in Express.js is used to store user data on the server between HTTP requests. This helps in managing user authentication, shopping carts, and other temporary data.

```
const express = require("express");
const session = require("express-session");


const app = express();


// Configure Session Middleware
app.use(session({
  secret: "mySecretKey",  // Used to sign the session ID cookie
  resave: false,          // Prevents session from being saved back if not modified
  saveUninitialized: true, // Saves uninitialized session (new session)
  cookie: { maxAge: 60000 } // Session expires in 1 minute
}));


// Route to Create Session
app.get("/create", (req, res) => {
  req.session.user = { name: "John Doe", role: "admin" };
  res.send("Session Created!");
});


// Route to Read Session
app.get("/read", (req, res) => {
  res.send(req.session.user || "No active session");
});
```

```
// Route to Check Session
app.get("/check", (req, res) => {
  if (req.session.user) {
    res.send("Session exists");
  } else {
    res.send("No active session");
  }
});

// Route to Destroy Session
app.get("/destroy", (req, res) => {
  req.session.destroy(() => res.send("Session Destroyed!"));
});

// Start Server
app.listen(3000, () => console.log("Server running on http://localhost:3000"));
```

Install Required Packages

```
npm install express express-session
```

What is JWT (JSON Web Token)?

JWT (JSON Web Token) is a secure way to authenticate users and transmit data between parties as a JSON object. It is widely used for authentication and authorization in web applications.

Why Use JWT?

✓ Stateless authentication – No need to store sessions on the server.

✓ Secure – Uses encryption and signatures to prevent tampering.

✓ Compact – Small in size, making it efficient for transmission.

✓ Cross-platform – Works on web, mobile, and API-based systems.

How JWT Works?

1 User logs in → Server verifies credentials.

2 Server generates a JWT and sends it to the client.

3 Client stores the JWT (in localStorage or cookies).

4 Client sends JWT with each request (inside the Authorization header).

5 Server verifies JWT → If valid, the request is processed.