BACKEND

1) .ENV (Environment Variables)
2) Serve Files From Server
3) Mongoose for MongoDB
4) express.json()
5) CORS(Cross Origin Resource Sharing)
6) JWT (Json Web Token)

Backend Guide : https://github.com/goldbergyoni/nodebestpractices

🧭 **Backend concepts you should absolutely learn (beginner → advanced):**

🔹 1. HTTP & REST basics Methods (GET, POST, PUT, DELETE) Status codes (200, 404, 401, 500) Headers, query params, body	🔹 2. Authentication & Authorization JWT (JSON Web Tokens) Cookie-based sessions Role-based access control (RBAC)	🔹 3. Middleware & Routing (Express, NestJS, etc.) app.use(), req, res Route-level and global middleware
🔹 4. Database integration MongoDB / PostgreSQL / MySQL ORMs like Prisma or Mongoose Query optimization	🔹 5. CORS & Cross-origin requests	🔹 6. Error Handling Centralized error middleware Logging (Winston, Pino) Custom error classes
🔹 7. File Uploads Multer (Express) Cloud storage (S3, Cloudinary)	🔹 8. Security Best Practices Input validation Rate limiting Helmet / CSRF	🔹 9. Testing APIs Postman Jest / Supertest
	🔹 10. API Design Principles Versioning Pagination REST vs GraphQL	

ENV (Environment Variables)

The .env file is a simple text file that stores environment variables for an application. It's commonly used to store sensitive configuration settings, such as API keys, database credentials, and other values that might vary across different environments (development, staging, production). By keeping these settings in a separate file, they can be managed more easily and securely, and the code itself remains cleaner.

1. **Key-Value Pairs**: The file contains key-value pairs, where the key is the name of the environment variable, and the value is its corresponding value.
2. **Security**: .env files are typically excluded from version control systems (e.g., Git) to prevent accidental exposure of sensitive information.
3. **Flexibility**: They allow for easy adaptation to different environments by simply modifying the values in the .env file.

Ex :

```
1. DB_HOST=“localhost”
2. DB_USER=“myuser”
3. DB_PASSWORD=“mypassword”
4. API Keys: API_KEY=“your_api_key”
```

Nodejs Implementation :

1. npm i dotenv
2. in the main server.js file (where the port is running), import dotenv package require('dotenv').config(); This command will help access the env variables in all other files because all files are run through server.js
3. Now whereever you want to use the variables defined in the .env file, just use process.env.VARIABLE

Note : Sometimes, the file doesn't recognize what process.env.VARIABLE is, for that import the dotenv package in that specific file again

What is process ?

The process object is a Node.js object. It provides information about, and control over, the current Node.js process. It is a global object, meaning it is always available in Node.js applications without requiring an import statement. While Node.js uses JavaScript as its language, the process object itself is not a standard JavaScript object available in browser environments. It's specific to the Node.js runtime environment.

more about process object : https://github.com/69JINX/FrontEnd/blob/main/Notes/Notes_Pic/dotenv-and-global-access.png

Serve Files (Assets) From Server

https://github.com/69JINX/FrontEnd/blob/main/Notes/Notes_Pic/serve-files-from-server.png

By default, no one can access files on a server just by using a link to the server. To allow public access to files stored on the server, we can use middleware provided by Express that serves static files via their file paths.

(Note: In real-world applications, files are often stored on cloud platforms like AWS S3 instead of directly on the server.)

Imagine the following file structure on the server:

└── Public/

  └── image.jpg

To give public access to image.jpg via a URL, we can use the following Express middleware:

```
1. app.use('/files', express.static(path.join(process.cwd(), 'Public')))
```

Now, image.jpg will be accessible through this URL: https//your-domain.com/files/image.jpg

Mongoose for MongoDB

```
>> npm i mongoose
```

Mongoose is an Object Data Modeling (ODM) library for MongoDB, a NoSQL database. It provides a higher-level abstraction for working with MongoDB, making it easier for developers to interact with the database using JavaScript. Mongoose simplifies data modeling, schema creation, and validation, allowing developers to work with MongoDB in a more structured and intuitive way.

Here's a more detailed breakdown:

ODM (Object Data Modeling):

Mongoose allows developers to model data using JavaScript objects, which are then mapped to MongoDB documents.

Schema-based:

Mongoose uses schemas to define the structure of documents in MongoDB collections, ensuring data integrity and consistency.

Simplifies database interaction:

Mongoose provides a more developer-friendly way to interact with MongoDB compared to using the native MongoDB driver, which requires more manual query building and schema management.

Benefits of using Mongoose:

Easier data modeling: Mongoose makes it easier to create and manage data structures, especially for complex applications.

Built-in features:

Mongoose includes features like type casting, validation, and query building, reducing the need for manual coding.

Improved code organization: Using Mongoose can lead to cleaner and more organized code, as it handles many of the low-level database operations.

MongoDB v/s Mongoose

Without Mongoose package, we have to use mongodb package(npm i mongodb) to insert data in DB, but mongodb package doesn't provide schemas or validations at the application level.

Mongoose provide more facilities and security then mongodb.

If you don't want to create schema and just want to insert data for small project, use mongodb. But if you want more restriction and validations, use mongoose.

mongodb :

- No schemas or validations at the application level.
- No middleware/hooks (like pre-save, post-remove).
- You must manually handle: Validation, Relationships, Data transformations, Timestamps, defaults, etc.

mongoose :

- Built on top of the mongodb driver.
- Adds schema definitions, model-based data access, and validation.
- Offers middlewares, virtuals, population (joins), plugins, etc.

Guide on how to use Mongoose to connect to MongoDB, define a schema, and perform basic queries :

➢ 1. Install Mongoose

```
npm install mongoose
```

➢ 2. Connect to MongoDB

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

const url = 'mongodb+srv://jubernowawave:JUBERkhan%40123@cluster0.n4tvq.mongodb.net/Eportal'; // connection url from mongodb atlas

mongoose.connect(url, {
  useNewUrlParser: true,
  useUnifiedTopology: true
})
.then(() => console.log('MongoDB connected'))
.catch(err => console.error('MongoDB connection error:', err));
```

➢ 3. Define a Schema & Model

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

const userSchema = new mongoose.Schema({
  first_name: { type: String, required: true },
  last_name : String,
  age: Number,           // Number
  isActive: Boolean,     // Boolean
  birthday: Date,        // Date
  profilePic: Buffer,    // Binary Buffer. Used for storing binary data like files, images, or encrypted content.
  sport: {
    type: mongoose.Schema.Types.ObjectId, // ObjectId reference
    ref: 'Sports' // ObjectId from Sports collection/table
  },
  tags: [String], // Array of Strings
  preferences: mongoose.Schema.Types.Mixed, // Mixed. A flexible type that can hold any kind of value — object, array, string, etc.
  meta: {         // Map of dynamic keys
    type: Map,
    of: String
  },
  price: mongoose.Schema.Types.Decimal128, //  Used for storing high-precision decimal numbers (like currency). eg: 999.99
}, {
  timestamps: true // adds createdAt and updatedAt
});

const User = mongoose.model('User', userSchema);
module.exports = User;
```

4. Use the Model to Perform Queries

```
const User = require('./models/User');

// Create a new user
const newUser = await User.create({ name: 'Alice', age: 25, email: 'alice@example.com' });

```

```
// Find one user
const foundUser = await User.findOne({ name: 'Alice' });

// Update a user
await User.updateOne({ name: 'Alice' }, { age: 26 });

// Delete a user
await User.deleteOne({ name: 'Alice' });

// Find all users
const allUsers = await User.find();
```

More on MongoDB in MongoDB notes :

express.json()

```
app.use(express.json());
```

express.json() is a built-in middleware function in Express.js that parses incoming requests with JSON payloads. It handles JSON data sent from clients and makes it accessible within your server-side application. This allows you to easily work with data sent by clients, especially when building RESTful APIs that use JSON.

Here's a more detailed explanation:

Parsing JSON:

When a client sends data to your server in the form of JSON, express.json() parses this JSON data and attaches it to the req.body property of the request object.

Accessibility:

This means you can access the parsed JSON data within your route handlers using req.body. For example, req.body.name would give you access to the 'name' field in the JSON data sent from the client.

Middleware Function:

express.json() is a middleware function, meaning it's a piece of code that can modify the request object or respond to a request. It's typically used with app.use() to apply it globally or with individual routes for more targeted parsing.

body-parser Dependency:

Historically, Express.js relied on the body-parser library for parsing request bodies. However, in newer versions of Express, express.json() is now a direct part of the framework and essentially replaces the need for body-parser when dealing with JSON data.

```
const express = require('express');
const app = express();

// Use express.json() as middleware
app.use(express.json());

app.post('/api/data', (req, res) => {
  // req.body now contains the parsed JSON data
  const jsonData = req.body;
  console.log(jsonData);  // Output: { name: 'John', age: 30 }
  res.json({ message: 'Data received successfully' });
});

app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

In this example, any request sent to /api/data with a JSON payload will have its data parsed and made available in req.body. This makes it easy to access and work with the data within the route handler.

CORS (Cross Origin Resource Sharing)

It is a security feature built into web browsers that controls how resources on a server can be requested from another domain (or "origin") outside the one the resource originated from.

CORS is a protocol that uses HTTP headers to tell browsers whether a specific web application running at one origin (eg. domain.com) is allowed to access resources from a different origin (eg. domain1.com).

Example scenario:

1. Your frontend is hosted at https://example-frontend.com.
2. Your backend API is hosted at https://api.example-backend.com.
3. The browser blocks API requests from frontend to backend unless the backend includes proper CORS headers (e.g., Access-Control-Allow-Origin).

Why do we need CORS in the backend?

Security: Prevent unauthorized websites from reading sensitive data from your server.
Controlled Sharing: Only allow access to known and trusted frontend domains.
Avoid Same-Origin Policy issues: Browsers enforce the Same-Origin Policy, which blocks cross-origin requests by default for security. CORS allows safe and secure controlled exceptions to this rule.

```
1. app.use(cors());
```

in Express, This line tells your Express.js app to enable CORS (Cross-Origin Resource Sharing) with default settings.

What it does exactly:

Allows requests from any origin (Access-Control-Allow-Origin: *)
Does not allow credentials (like cookies, sessions, or Authorization headers)

Tip : to restrict public usage of your api and only allow defined frontend domain to access your backend APIs :

```
1. app.use(cors({
2.   origin: 'http://your-frontend.com',
3.   methods: ['GET', 'HEAD', 'PUT'], // only these methods will be allowed, all the rest methods PATCH, POST, DELETE will be blocked
4. }));
```

When you use:

```
1. app.use(cors());
```

It's shorthand for:

```
1. app.use(cors({
2.   origin: '*',
3.   methods: ['GET', 'HEAD', 'PUT', 'PATCH', 'POST', 'DELETE'],
4. }));
```

full detailed discussion : https://chatgpt.com/c/6821960c-80d8-8012-8d48-0b5dc3da0684
https://www.geeksforgeeks.org/cross-origin-resource-sharing-cors/
https://www.geeksforgeeks.org/http-access-control-cors/
https://developer.mozilla.org/en-US/docs/Web
https://www.youtube.com/watch?v=E6jgEtj-UjI

JWT (Json Web Token)

JSON Web Tokens (JWTs) are used primarily for authentication and authorization in web applications and APIs. They provide a secure way to share information between two parties, usually a client and a server, in a compact and self-contained format. JWTs allow for stateless authentication, meaning the server doesn't need to store session information, which improves scalability.
JWTs (JSON Web Tokens) can be securely stored on the client side, typically in cookies, allowing user sessions to persist without requiring frequent re-authentication. As long as the JWT remains valid (i.e., until it expires), users can revisit the site without needing to log in again, since the token can be used to verify their identity on subsequent requests.

JWT Authentication Process :

1. user login with his credentials
2. authenticate his credentials(email-password) from backend
3. if authenticated, fetch user data from db and convert that user data to jwt token and send to frontend.
   converting data to jwt token :

   ```
   1. npm i jsonwebtoken
   ```

```
1. const jwt = require('jsonwebtoken')
2. jwt.sign(dataToEncrypt, process.env.JWT_KEY, { expiresIn: '7d' },(error, token)=>{
3.     if(error) return res.status(500).json({message:'try after some time!'});
4.     res.status(200).json({message:'success',token})
5. })
```

ExpiresIn Eg: 60, "2 days", "10h", "7d". A numeric value is interpreted as a seconds count. If you use a string be sure you provide the time units (days, hours, etc), otherwise milliseconds unit is used by default ("120" is equal to "120ms").
https://www.npmjs.com/package/jsonwebtoken#:~:text=numeric%20value

4. frontend get the token and store it in cookies.
5. backend create a verifyToken api to verify the token
6. frontend send token in verifyToken api in the headers

```
1. const res = await axios.post('https://localhost:4000/verifyToken',{},{
2. headers:{
3.   'Authorization':jwtToken
4. }})
```

7. Backend verify the token and send the actual user data.

```
1. jwt.verify(req.headers.authorization, process.env.JWT_KEY,(error, decoded)=>{
2. if(error) res.status(401).json({message:'please login again!'})
3.    res.status(200).json({message:'success',data:decoded})
4. })
5.
```

Error : After receiving the error(invalid token, expired token), the frontend should log out the user and redirect to login page and generate the jwt again (from step 1)

Success : store encrypted userdata in context that was send from verifyToken api

8. The actual user data(decoded) will be saved in the context.
9. The jwt token will stay till its expiry date in the cookie.
10. On Every refresh, the verifyToken api is called with the jwt token (to convert token to userdata)

Check authentication on every protected API route in backend with auth middleware, jwt will be send with header/cookie from frontend :

```
1. // middleware/auth.js
2. const jwt = require('jsonwebtoken');
3.
4. function authenticateToken(req, res, next) {
5.   const authHeader = req.headers['authorization'];
6.   const token = authHeader?.split(' ')[1]; // Bearer <token>
7.
8.   if (!token) return res.status(401).json({ message: 'No token provided' });
9.
10.   jwt.verify(token, process.env.JWT_SECRET, (err, user) => {
11.     if (err) return res.status(403).json({ message: 'Invalid token' });
12.
13.     req.user = user; // Store user info in req for use in routes
14.     next();
15.   });
16. }
17.
18. module.exports = authenticateToken;
```

Use it in routes:

```
1. const authenticateToken = require('./middleware/auth');
2.
3. app.get('/api/profile', authenticateToken, (req, res) => {
4.   res.json({ message: `Hello ${req.user.name}` });
5. });
6.
7. app.post('/api/update', authenticateToken, (req, res) => {
8.   // Only logged-in users can update data
9. });
```