BACKEND

ONE LOVE. ONE FUTURE.

Learning Outcomes

By the end of this lesson, students will be able to:

- Understand the role of backend in a web application.
- Build a server using Node.JS and Express.JS
- Build RESTful APIs using Node.JS and Express.JS
- Apply clean architecture and best practices using MVC and Service Layer structure.
- Work with MongoDB and Mongoose to model and manage data.

Content

1. Introduction
2. Node.JS and NPM
3. Express.JS and Best Practices
4. MongoDB and Mongoose

HUST

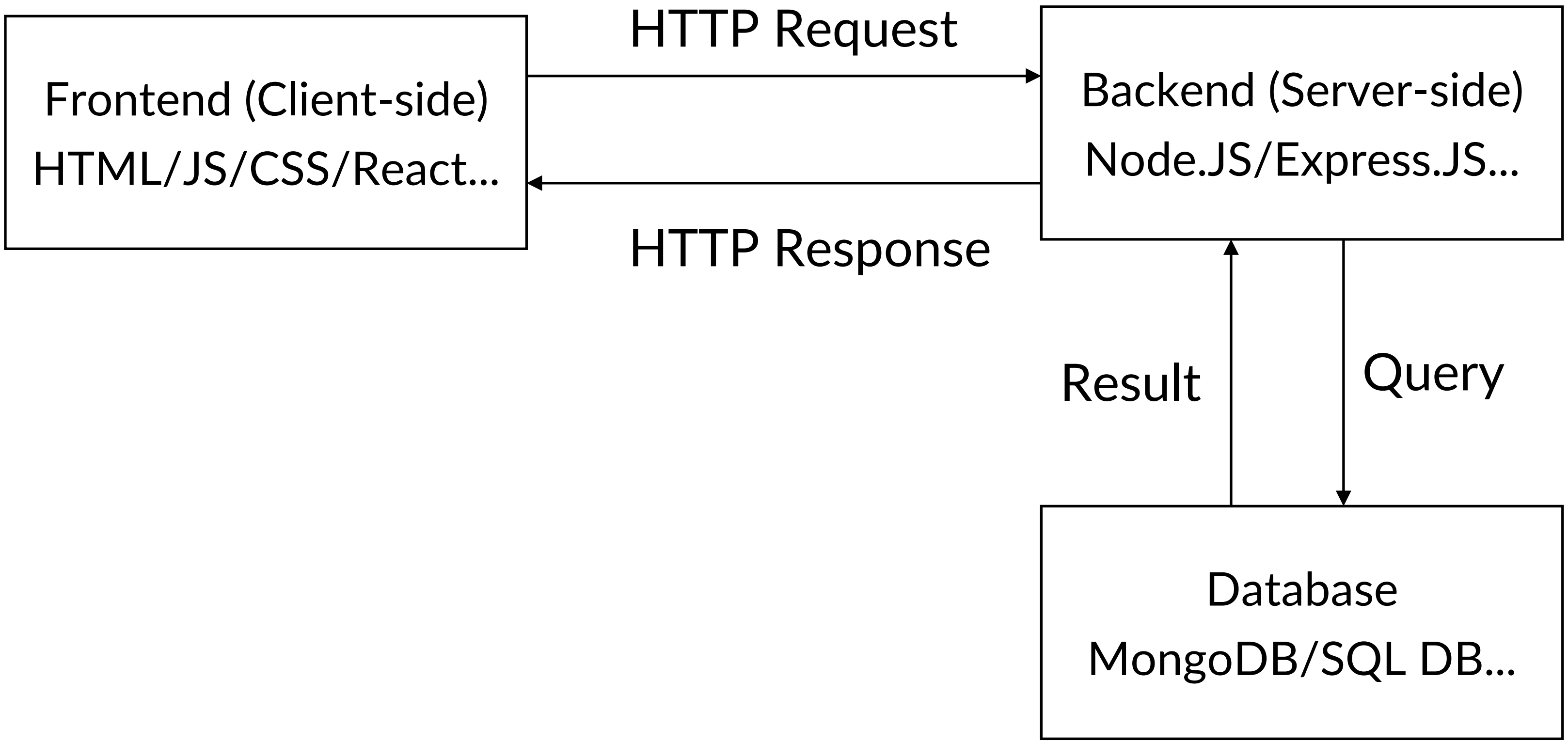
1. Introduction

Questions

- What can’t Frontend do?
  - Access database directly
  - Perform secure logic
  - Manage users & permissions

- Why do we need a backend?
  - Communication between client and database
  - Data storage
  - Business logic
  - Authentication & authorization
  - Result returned: JSON responses (REST API)

Frontend vs Backend



Frontend vs Backend

- Frontend (Client-Side)
  - Runs in the browser (HTML, CSS, JavaScript, React).
  - Handles user interface and interactions.
  - Sends requests to the server via HTTP/HTTPS.
- Backend (Server-Side)
  - Runs on the server (Node.js, Express.js).
  - Processes business logic, authentication
  - Communicates with database and returns responses
- How they work together
  - Frontend makes requests → Backend processes them → Returns responses.
  - JSON is the common format for data exchange.

Backend Technology Landscape

- Languages: JavaScript, TypeScript, Python, Go, Java
- Frameworks:
  - Node.js + Express.js (common)
  - NestJS (enterprise)
  - Fastify (performance)
- Databases:
  - MongoDB (NoSQL)
  - PostgreSQL / MySQL (SQL)
- Infrastructure:
  - Docker, CI/CD, Cloud, Serverless

Why JavaScript for Backend?

- One language for the entire stack
    - Use JavaScript on both Frontend and Backend
    - Easier learning and unified workflow for teams
- Fast and efficient
    - Built on Google's V8 Engine
    - Non-blocking handles thousands of requests concurrently
- Huge ecosystem (NPM)
    - Large package ecosystem
    - Libraries for everything: APIs, auth, databases, testing..
    - Strong community support
- Modern backend-friendly features
    - Async/Await
    - TypeScript compatibility
    - Microservices & Serverless support

2. Node.JS and NPM

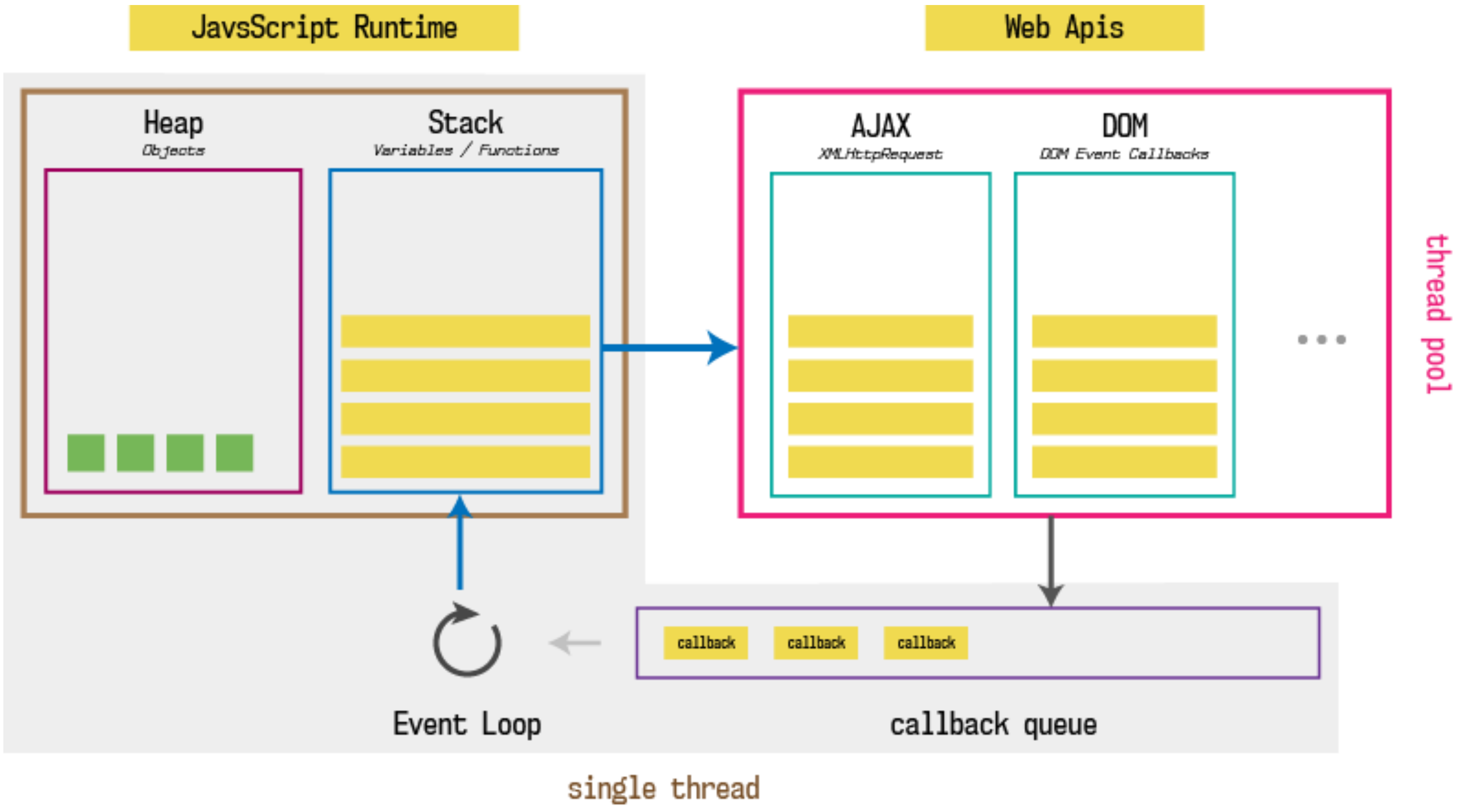
What is Node.js?

- Node.js is a JavaScript runtime environment
  - Allows JavaScript to run outside the browser
  - Built on Google Chrome's V8 JavaScript engine
  - Designed for building fast, scalable, network applications
- Key characteristics
  - Event-driven architecture
  - Non-blocking I/O model
- Common use cases
  - Realtime applications (chat, notifications)
  - Microservices architecture
  - Backend for web & mobile apps

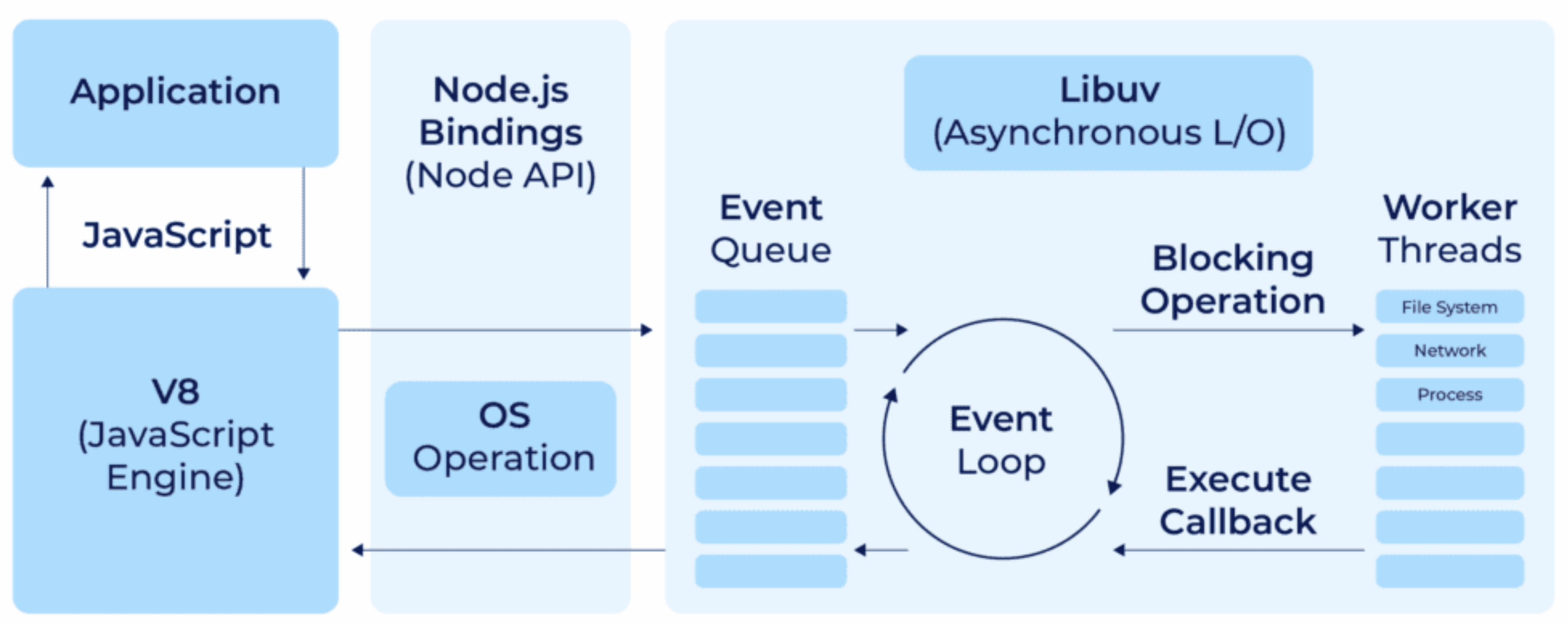
V8 Engine & libuv

- V8 JavaScript engine
  - Google's high-performance JS engine (used in Chrome)
  - Compiles JavaScript directly into machine code
  - Provides extremely fast execution
- libuv
  - A library used internally by Node.js
  - Enables Node.js to perform async operations outside the main thread
  - Foundation of Node's event loop
- How they work together
  - V8 runs your JavaScript code
  - libuv performs async I/O tasks in the background

JavaScript Engine Architecture



Node.js Architecture



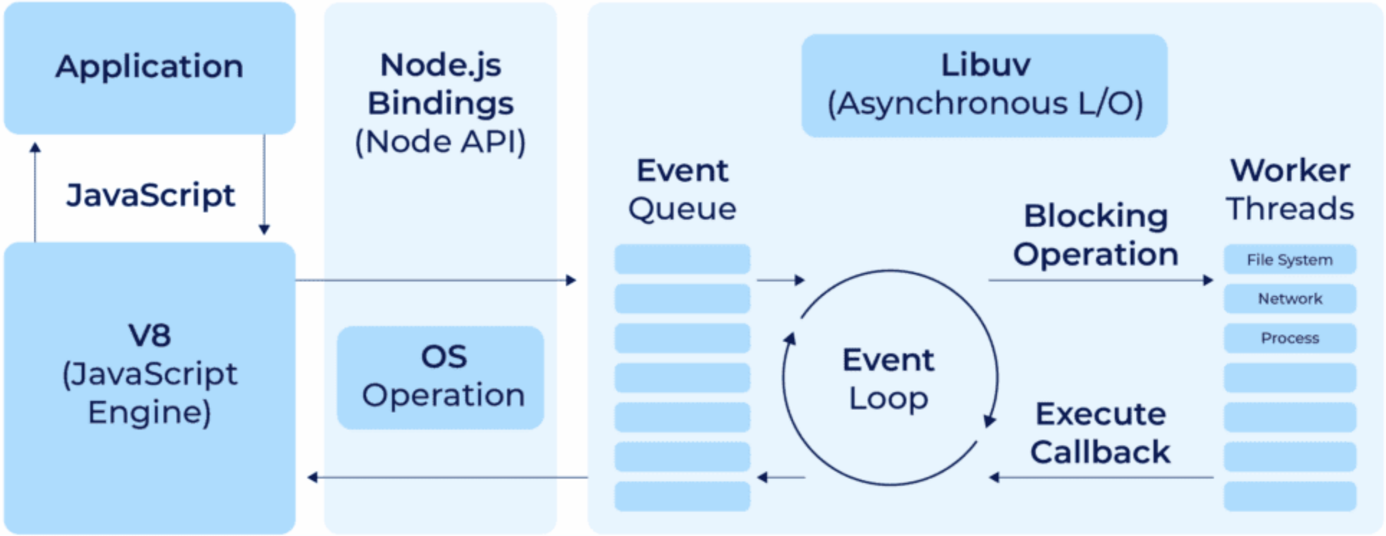
https://litslink.com/blog/node-js-architecture-from-a-to-z

Single-thread & Non-blocking I/O

- Single-thread
  - Node.js runs JavaScript in a single main thread.
  - No multiple threads executing JS in parallel.
  - Avoids issues like data races and deadlocks.
- Non-blocking I/O
  - I/O operations (file, DB, network) are not executed in the main thread.
  - Node delegates them to libuv’s thread pool or the OS, then continues.
  - When I/O finishes, an event is emitted and callback/Promise runs.
- Why this model works
  - Single thread handles logic → lightweight.
  - Non-blocking I/O handles expensive tasks → scalable.
  - Together they allow Node.js to handle thousands of concurrent connections

Example

- fs.readFile("data.txt", **callback**)
  - **JavaScript** calls function → **V8** processes
  - **Node.js bindings** translate to C++
  - **Libuv** sends request to **OS Operation**
  - **OS Operation** reads file in background
  - When done → **OS Operation** tells **Libuv**
  - **Libuv** puts event into **Event Loop**
  - **Event Loop** triggers **callback** in **JavaScript**



Questions

- Does every function go into Libuv?
    - No
    - Only async operations require Libuv
    - Simple synchronous functions run directly in V8’s Call Stack.
- Does every function after being put into Event Queue also be put into Worker Threads?
    - No
    - OS operations
    - Worker Threads
    - Timers

Installing Node.js & Running Your First Script

- Install: https://nodejs.org
- Verify: `node -v` and `npm -v`
- Create a JavaScript file
  - app.js: `console.log("Hello from Node.js");`
  - run: `node app.js`
- Actions
  - Node loads app.js into the **V8 engine**
  - V8 compiles JavaScript into **machine code**
  - V8 runs the code inside the **Call Stack (synchronous)**
  - `console.log()` calls `process.stdout.write()` via Node Bindings
  - OS prints the output (no callback)

Global Environment in Node.js

- Node.js does NOT have window or document
    - Node.js is not a browser environment
    - No DOM, No BOM (Browser APIs like alert())
- Node.js provides its own global objects
    - global: equivalent to window in browsers
    - process: runtime information, e.g., process.env,
    - module /exports: common JS module system

Feature	Browser	Node.js
Global object	window	global
DOM support	✓ Yes	✗ No
File system	✗	✓ fs module
Process info	✗	✓ process

ES Modules (ESM)

- **Official JavaScript module system** used to organize code into reusable and maintainable files
- All imports and exports in ESM are known before the code runs => V8 can optimize loading
- Syntax
  - config:

    ```
    {
      "type": "module"
    }
    ```
  - //math.js

    ```
    export function sum(a, b) {return a + b;}
    ```
  - // app.js

    ```
    import { sum } from './math.js';
    ```

Built-in Modules: fs & path

- Node.js provides core modules that work **without installation.**
- fs - File System Module: used to read, write files

  -
    ```
    import { readFileSync } from 'fs’;
     const data = readFileSync('data.txt', 'utf-8’);
    ```

  -
    ```
    import { writeFileSync } from 'fs’;
     writeFileSync('log.txt', 'Hello!’);
    ```
- path - Path Utilities Module: work with file and directory path
  - aviod OS-specific issues (e.g., \ vs /)

  -
    ```
    import path from 'path’;
     const fullPath = path.join(__dirname, 'uploads', 'photo.jpg’);
    ```

package.json

- package.json
  - The manifest file of a Node.js project.
  - Stores metadata, dependencies, scripts, and project configuration.
  - Automatically created via: npm init npm init -y

```
{
  "name": "my-app",
  "version": "1.0.0",
  "type": "module",
  "scripts": {
    "start": "node index.js",
    "dev": "nodemon index.js"
  },
  "dependencies": {},
  "devDependencies": {}
}
```

NPM (Node Package Manager)

- The default **package manager** for Node.js.
- Enables you to install, update, and manage libraries.
- Why NPM is important?
  - Access to the ecosystem of open-source packages
  - Helps developers reuse existing code
- Common NPM Commands
  - npm init – initialize a new project
  - npm install <package> – install dependencies
  - npm install -D <package> – install devDependencies
  - npm update – update installed packages
  - npm run <script> – run custom scripts from package.json

node_modules

- A directory automatically created by NPM.
- Stores **all installed dependencies** for your project.
- Contains both your direct dependencies and all **nested dependencies** they require.
- Flow
  - NPM reads package.json → checks dependencies.
  - NPM downloads the package from the registry (npmjs.org).
  - NPM installs: The package itself (express)
  - All its required sub-dependencies
  - All packages are placed inside node_modules/.

Semantic Version

- Format: MAJOR.MINOR.PATCH
  - 1.4.2

```
1.4.2
| | |
| | └──── PATCH: bug fixes, no breaking changes
| └────── MINOR: new features, backward compatible
└─────────MAJOR: breaking changes
```

  - Prefix
    - ^1.4.2: allows updates to MINOR + PATCH
    - ~1.4.2: allow updates to PATH only
- Prevents unexpected breaking changes.
- Keeps the project stable across development and production.

Dependencies vs DevDependencies

- Dependencies: packages required in production.
  - Used for handling API requests (Express)
  - Database connections
  - Authentication libraries

```
"dependencies": {
  "express": "^4.18.2",
  "mongoose": "^7.0.3"
}
```

- DevDependencies: packages needed only during development, not required in production.
  - Nodemon (auto-restart server)
  - Testing frameworks (Jest, Mocha)
  - Formatters (ESLint, Prettier)
  - …

```
"devDependencies": {
  "nodemon": "^3.0.0",
  "eslint": "^8.0.0"
}
```

TypeScript in Node.js Backend

- A strongly typed programming language that builds on JavaScript
- Compiles to plain JavaScript
- Makes backend code more reliable, safer, and easier to maintain.

```
hello.ts → (tsc) → hello.js → node hello.js
```

- When use TypeScript?
  - Medium/large backend projects
  - Preparing for enterprise / industry jobs
  - Team collaboration

HUST

hust.edu.vn fb.com/dhbkhn

3. Express.JS and Best Practices

3.1. Express

3.2. Best Practices

What is Express.JS?

- A **web framework** built on top of Node.JS,
- Provides tools to build HTTP servers, REST APIs, and web applications.
- Makes server development easier and cleaner.
- Features
  - Simple & Minimal: focuses on the essentials: routing, middleware, request & response.
  - Flexible architecture: choose patterns (MVC, services, etc.).
  - Huge ecosystem: thousands of packages for authentication, logging, security, rate limiting, validation, etc.
  - Industry adoption: used by Uber, PayPal, IBM.

Node.JS vs Express.JS

Feature	Node.JS	Express.JS
Type	Runtime environment	Web framework
Purpose	Execute JS on server	Build web servers
Built-in routing	❌ No	✓ Yes
Middleware	❌ No	✓ Yes
Developer experience	Low-level, manual	High-level, easy

Hello World with Express

1. **Install Express:** `npm install express`
2. **Create server.js**

```
import express from 'express';
const app = express();
app.get('/', (req, res) => {
  res.send('Hello World!');
});
app.listen(3000, () => {
 console.log('Server running on http://localhost:3000');
});
```

3. Run the server: `node server.js`

4.Open the browser: `http://localhost/3000`

Basic Routing in Express

- Routing = mapping URLs to functions.
- app.METHOD(PATH, HANDLER)
- Where:
  - METHOD → HTTP verb (GET, POST, PUT, DELETE)
  - PATH → URL route (/, /users, /products/:id)
  - HANDLER → function (req, res) => { ... }

- Example

```
app.post('/users', (req, res) => {
    res.send('User created’);
});
```

HTTP Methods (GET, POST, PUT, DELETE)

- HTTP methods define **the type of action** the client wants to perform on the server.
- They are the foundation of **RESTful API design**.

Method	Purpose	Typical Use
GET	Read data	Fetch resources
POST	Create data	Add new resource
PUT	Update data (replace)	Update full resource
DELETE	Remove data	Delete resource

Request Object (params, query, body)

- **req object** contains all information sent **from the client**

Source	Example URL	Access via	Used For
params	/users/123	req.params.id	path variables
query	/users?page=2	req.query.page	filters & search
body	POST {name:"A"}	req.body.name	form/payload

Response Object

- **res object** is used to **send data back to the client**

Method	Purpose
res.json()	Send JSON data
res.status()	Set HTTP status
res.set() / res.header()	Add or modify response headers

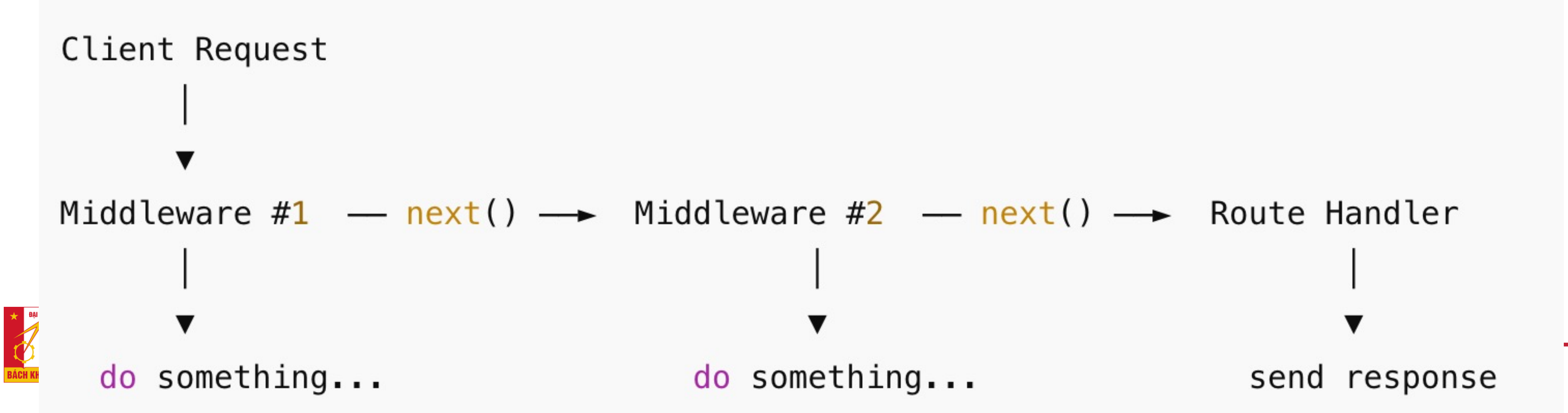
```
app.get('/users', (req, res) => {
  res
    .status(200)
    .set('Content-Type', 'application/json')
    .json({ users: [] });
});
```

What is Middleware?

- Middleware = steps the request must pass through.

```
function middleware(req, res, next) {
  // do something...
  next(); // pass control to the next middleware
}

app.use((req, res, next) => {
  console.log('Time:', Date.now());
  next();// without this, request will stop here
});
```



```
Client Request
      |
      ▼
Middleware #1  ── next() ──►  Middleware #2  ── next() ──►  Route Handler
      |                             |                             |
      ▼                             ▼                             ▼
  do something...               do something...              send response
```

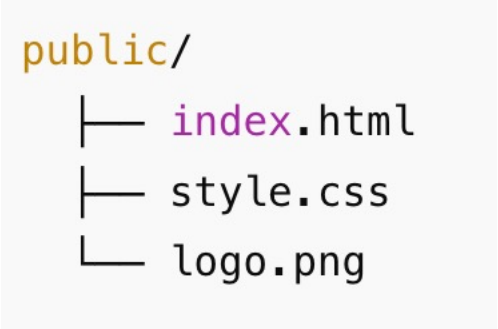
Middleware

Middleware	Purpose	Example Use
express.json()	Parse JSON body	POST/PUT API requests
cors()	Allow cross-origin requests	Frontend ↔ Backend
morgan()	Log requests	Debugging API

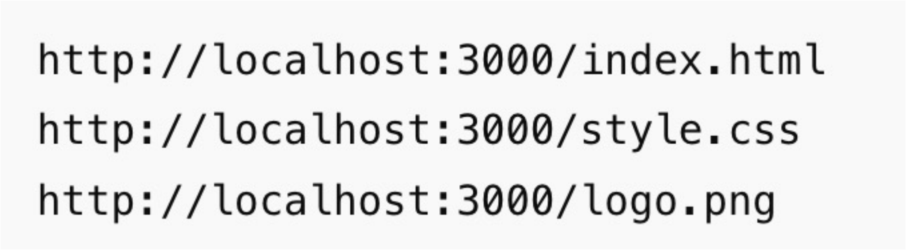
Static Files (express.static)

- express.static() is built-in middleware
- allows direct access to files in a directory and returns the exact contents of that file.

```
app.use(express.static('public'));
```



```
public/
    ├── index.html
    ├── style.css
    └── logo.png
```

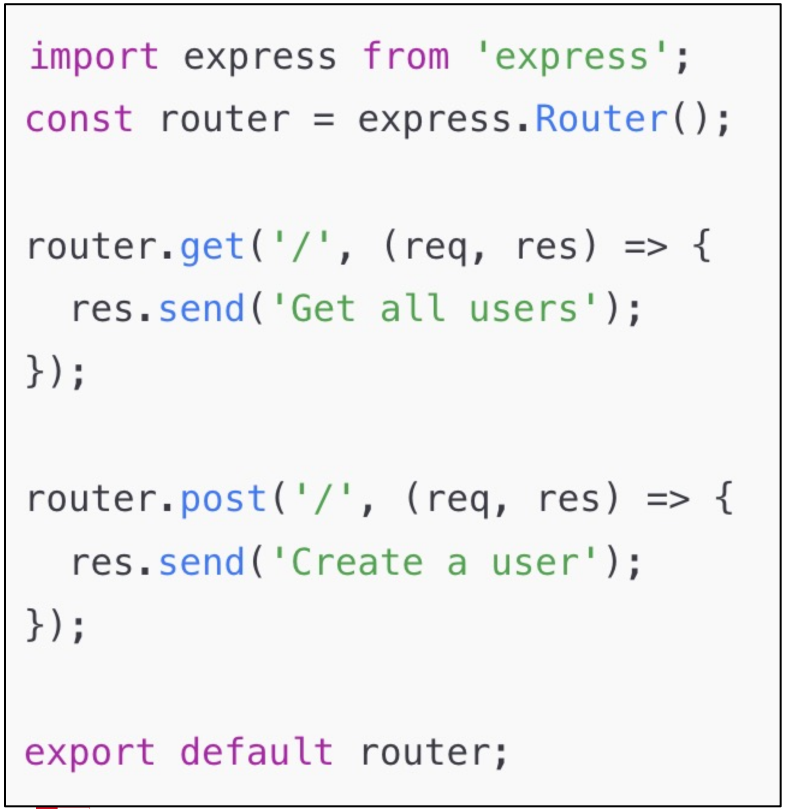


```
http://localhost:3000/index.html
http://localhost:3000/style.css
http://localhost:3000/logo.png
```

Routing

- Split routes into separate **modules**

routes/userRoutes.js



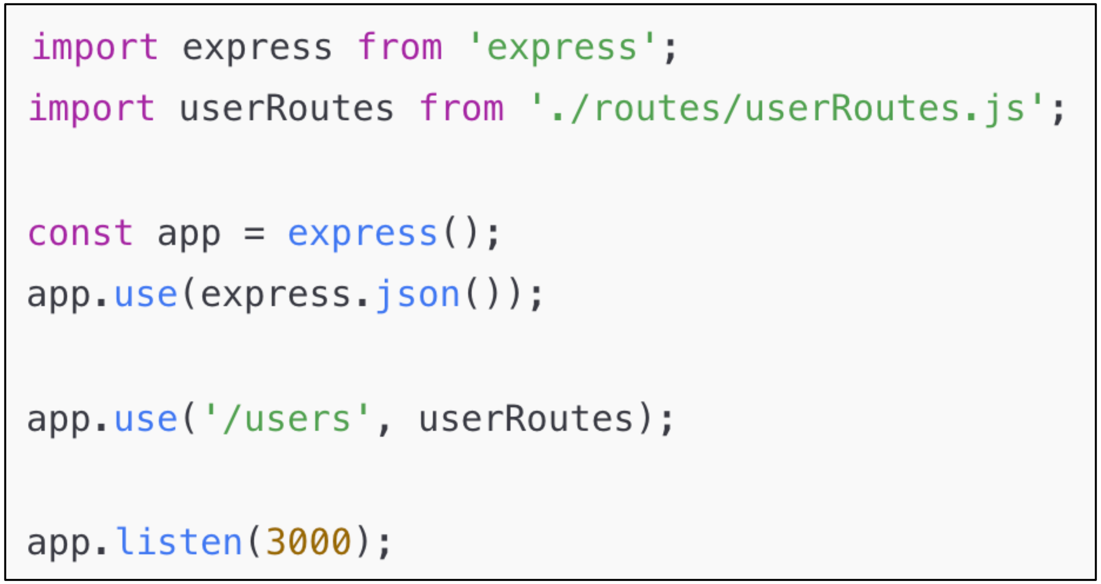
```
import express from 'express';
const router = express.Router();

router.get('/', (req, res) => {
  res.send('Get all users');
});

router.post('/', (req, res) => {
  res.send('Create a user');
});

export default router;
```

server.js



```
import express from 'express';
import userRoutes from './routes/userRoutes.js';

const app = express();
app.use(express.json());

app.use('/users', userRoutes);

app.listen(3000);
```

MVC Architecture

- **MVC (Model – View – Controller)** is a design pattern that separates an application into 3 clear layers:
  - **Model** → Data & database logic
  - **View** → Display (HTML, JSON)
  - **Controller** → Request handling & business logic
- Flow (RESTful)
  - Client→Controller→Model→Database→Model→Controller→Response (JSON)

Controller Layer: Role & Limitations

- Controller is the entry point for every HTTP request.
- Responsibilities
  - Receive and parse HTTP requests
  - Validate input (params, query, body)
  - Call the Service/Model layer
  - Handle errors and return the correct status code
  - Format the final JSON response
- Controller Should *Not* Do
  - Should NOT contain business logic
  - Query the database directly

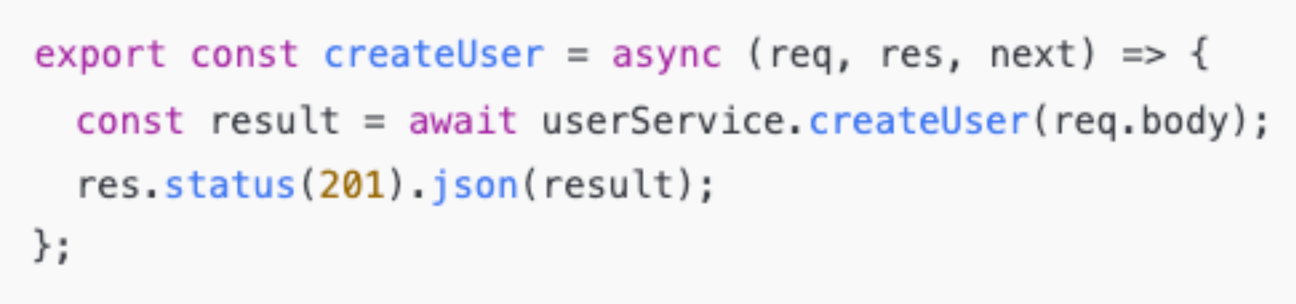
Controller should be thin, clean, and focus only on handling requests.

Where do we put business logic?

Service Layer: Avoiding Fat Controller

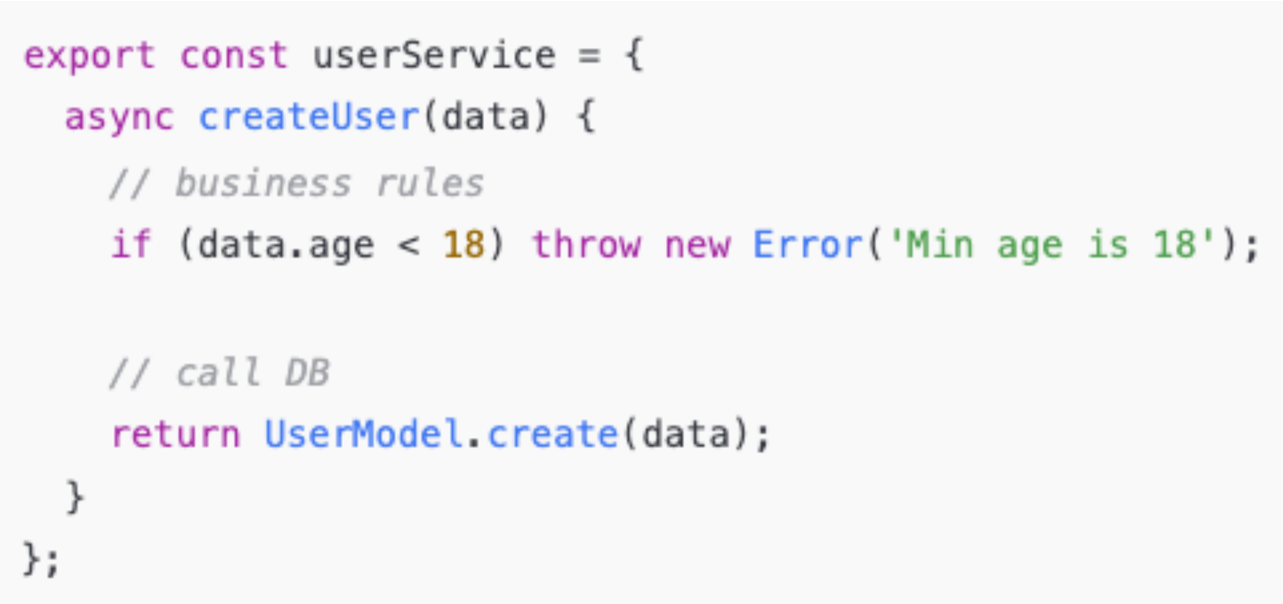
- **Service Layer** contains: Business logic & Data processing

Controller



```
export const createUser = async (req, res, next) => {
  const result = await userService.createUser(req.body);
  res.status(201).json(result);
};
```

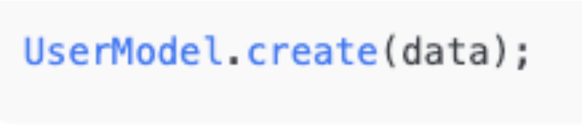
Service



```
export const userService = {
  async createUser(data) {
    // business rules
    if (data.age < 18) throw new Error('Min age is 18');

    // call DB
    return UserModel.create(data);
  }
};
```

Model



```
UserModel.create(data);
```

Error-Handling Middleware

- A special type of middleware that **catches errors** in your application and prevents the server from crashing.
- It must have **4 parameters** → (err, req, res, next)
- Structure

```
app.use((err, req, res, next) => {
  console.error(err.message);

  res.status(err.status || 500).json({
    success: false,
    message: err.message || "Internal Server Error"
  });
});
```

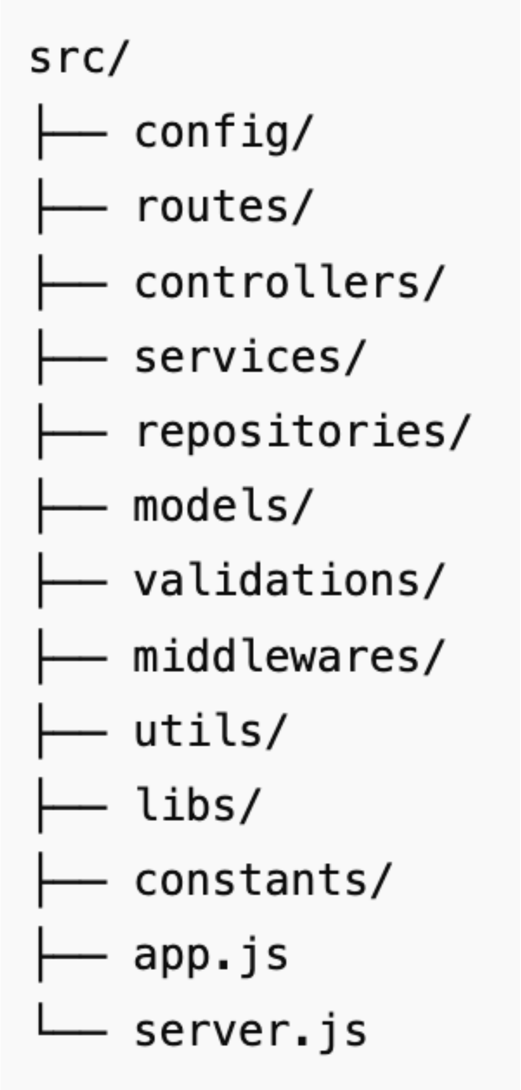
Environment Variables (.env & dotenv)

- Environment variables store **configuration values** that should NOT be hardcoded in your application.
- **.env file**
  - A plain text file that contains key-value pairs:
  - PORT=3000
  - DB_URI=mongodb+srv://username:password@cluster.mongodb.net/app

  (Always place .env in .gitignore)
- **dotenv** in Express
  - npm install dotenv
  - import dotenv from 'dotenv';
  - dotenv.config();
  - const port = process.env.PORT;

Backend Folder Structure

- config: system-level configuration
- routes: API endpoints
- controllers: request/response handling, no business logic
- services: business logics
- repositories: data access layer, handle queries to database
- models: database schema
- validations: request validation
- middlewares: reusable request functions
- utils: shared helper functions
- libs: external service integrations
- constants: store constant values
- app.js: initializes express app
- server.js: starts the server



```
src/
├── config/
├── routes/
├── controllers/
├── services/
├── repositories/
├── models/
├── validations/
├── middlewares/
├── utils/
├── libs/
├── constants/
├── app.js
└── server.js
```

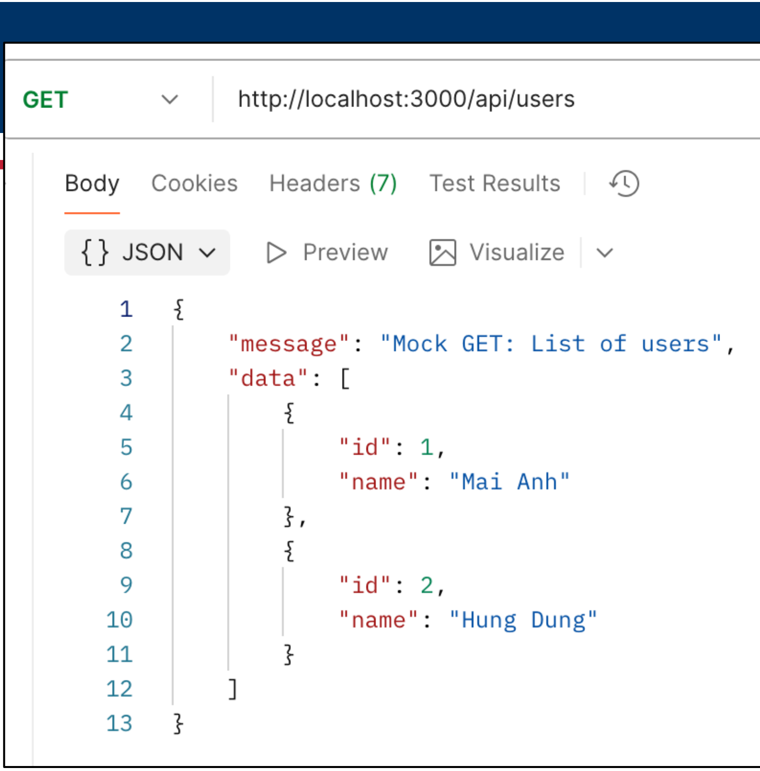
GET/POST example

- Install
  - npm init -y
  - npm install express
  - node index.js

- Postman
  - GET http://localhost:3000/api/users
  - POST http://localhost:3000/api/users

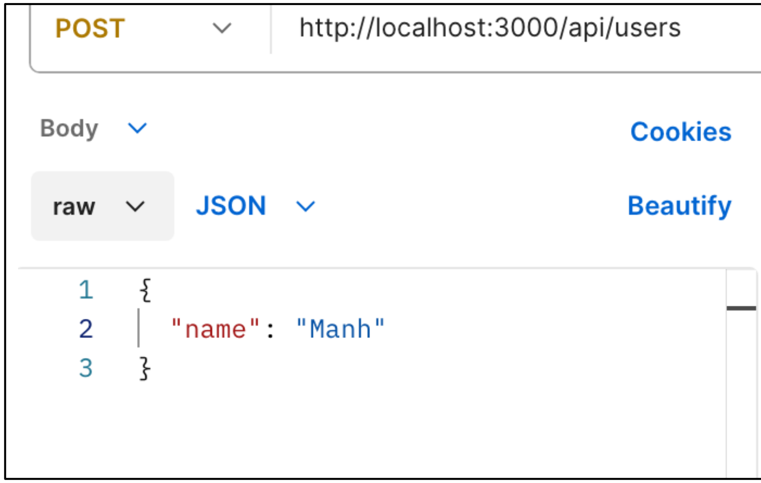
    Content-Type: application/json

    {
        "name": ”Manh"
    }



GET http://localhost:3000/api/users

```
{
    "message": "Mock GET: List of users",
    "data": [
        {
            "id": 1,
            "name": "Mai Anh"
        },
        {
            "id": 2,
            "name": "Hung Dung"
        }
    ]
}
```



POST http://localhost:3000/api/users

```
{
  "name": "Manh"
}
```

index.js

```
const express =
require("express");
const app = express();
app.use(express.json());

// Mock database
let users = [
  { id: 1, name: ”Mai Anh" },
  { id: 2, name: ”Hung Dung" }
];

app.get("/api/users", (req,
res) => {
  res.json({
    message: "Mock GET: List
of users",
    data: users
  });
});
```

```
app.post("/api/users", (req, res) =>
{
  const { name } = req.body;
  const newUser = {
    id: users.length + 1,
    name
  };

  users.push(newUser);
  res.status(201).json({
    message: "Mock POST: User
created",
    data: newUser
  });
});

// Start server
app.listen(3000, () =>
console.log("Mock API running "));
```

4. MongoDB and Mongoose

SQL vs NoSQL

- SQL (Relational Databases)
  - Examples: MySQL, PostgreSQL, SQL Server, Oracle
  - Structured, table-based data

- NoSQL (Non-Relational Databases)
  - Examples: MongoDB, Redis, Cassandra, Neo4j
  - Flexible schema

- Key Differences
  - Schema: Fixed (SQL) vs Flexible (NoSQL)
  - Structure: Tables vs Documents/Key-Value/Graphs
  - Use Cases: Complex relations (SQL) vs High volume & dynamic data (NoSQL)

Document & BSON

- Document (MongoDB)
  - Stored as key–value pairs, similar to JSON
  - Dynamic schema → fields can be added or removed anytime
- BSON (Binary JSON)
  - Binary representation of JSON
  - Efficient storage and better performance
- MongoDB stores data as BSON, but drivers return it as Documents

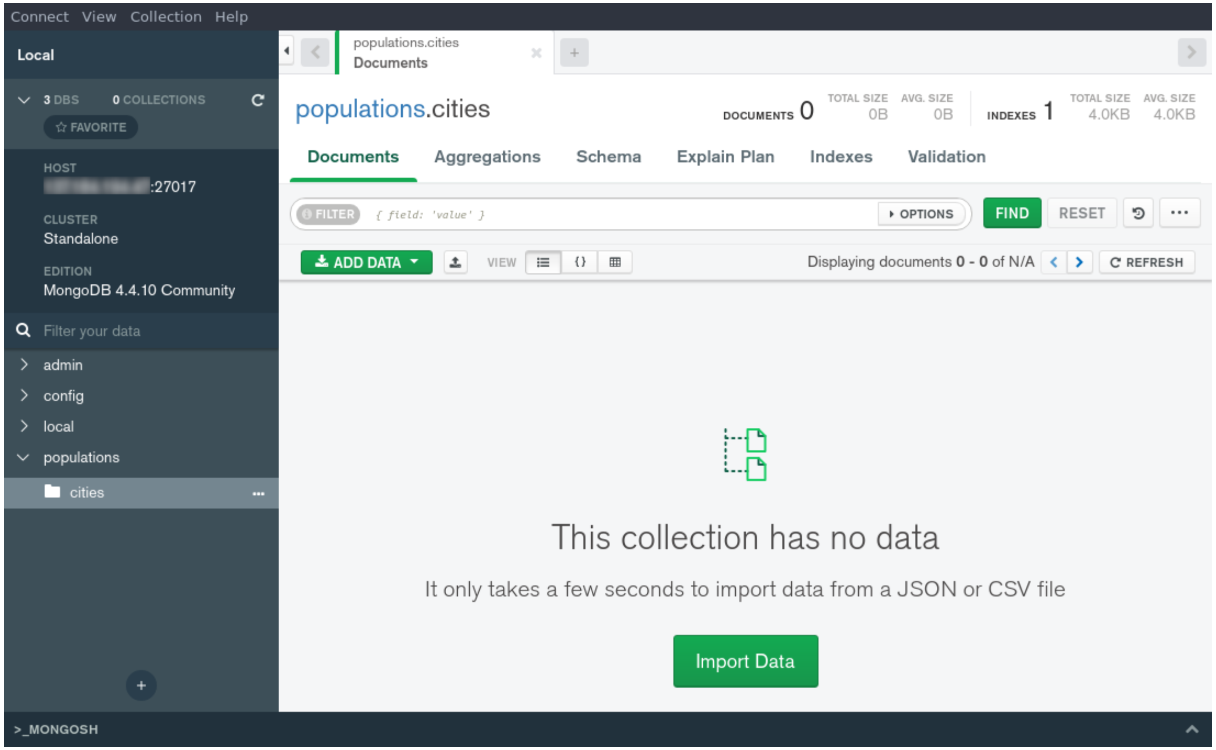
```
{
  "name": ″Minh",
  "age": 25,
  "skills": [″JS", ″C"],
  "address": {
    "city": "Hanoi",
    "zip": "100000"
  }
}
```

MongoDB Atlas

- Cloud database service for MongoDB
- Key Features
  - Automated Scaling
  - Global Clusters
  - Built-in Security:
  - Automated Backups with point-in-time recovery
  - Performance Monitoring dashboards
- Flow: Create Cluster → Get Connection String → Connect via Driver/Mongoose → Build App
  - Cluster = a MongoDB database environment
  - Contains many databases
  - Ensures performance, scaling, security, etc.

MongoDB Compass

- A GUI tool (Graphical User Interface) for MongoDB
- Official desktop application from MongoDB
- Allows you to visually explore, query, and manage data



Local

3 DBS 0 COLLECTIONS

HOST
[redacted]:27017

CLUSTER
Standalone

EDITION
MongoDB 4.4.10 Community

populations.cities

DOCUMENTS 0 TOTAL SIZE 0B AVG. SIZE 0B

INDEXES 1 TOTAL SIZE 4.0KB AVG. SIZE 4.0KB

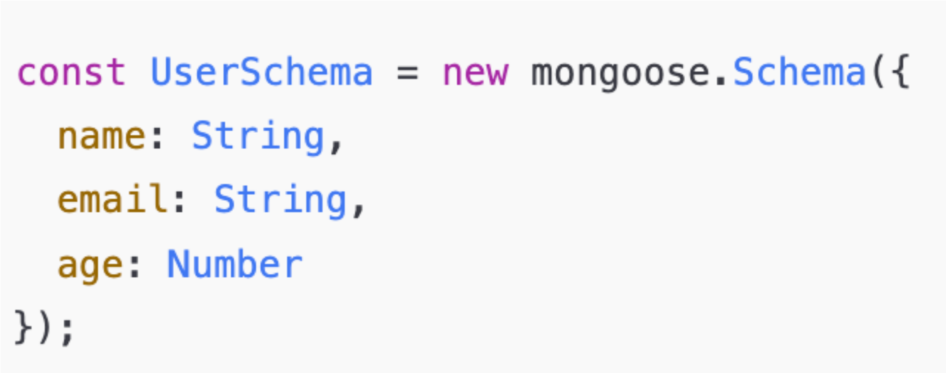
Displaying documents 0 - 0 of N/A

This collection has no data

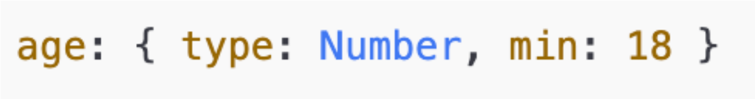
It only takes a few seconds to import data from a JSON or CSV file

What is Mongoose?

- An ODM (Object Data Modeling) library for MongoDB
  - Provides a structured way to interact with MongoDB
  - Mongoose adds **structure, validation, and consistency** to MongoDB collections.



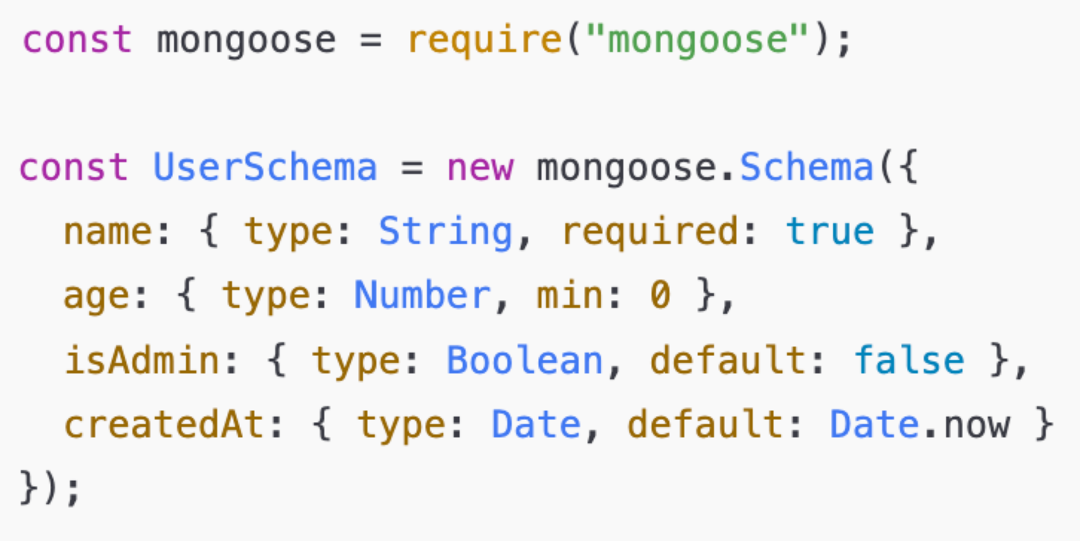
```
const UserSchema = new mongoose.Schema({
  name: String,
  email: String,
  age: Number
});
```



```
age: { type: Number, min: 18 }
```

Mongoose Schema

- A Mongoose Schema defines the structure and data types of documents in a MongoDB collection.
- Common Data Types
  - String – for text fields (name, email, title, etc.)
  - Number – for integers, floats (age, price, score)
  - Boolean – for true/false values (isActive, isAdmin)
  - Date – for timestamps and calendar dates (createdAt, birthday)



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

const UserSchema = new mongoose.Schema({
  name: { type: String, required: true },
  age: { type: Number, min: 0 },
  isAdmin: { type: Boolean, default: false },
  createdAt: { type: Date, default: Date.now }
});
```

Model (mongoose.model)

- Models represent collections in MongoDB
- Function: *mongoose.model(modelName, schema).*

```
// Create Model
const User = mongoose.model("User", UserSchema);
```

- "User": Model name in Mongoose
- Mongoose creates users collection in MongoDB
- User: JS variable
  - User.create({ name: ”Minh" });
  - User.find();
  - User.findById(id);

Connecting to MongoDB

```
• const mongoose = require("mongoose");
• mongoose.connect(process.env.DB_URI)
   .then(() => console.log("Connected to MongoDB"))
   .catch((err) => console.error("Connection Error:", err));
```

Create (Model.create)

- Model.create() is a Mongoose method used to:
  - Create a new document
  - Validate it against the schema
  - Save it directly to the database

```
try {
  const user = await User.create({
    name: ”Hung",
    email: ”hung@sis.hust.edu.vn"
  });
  console.log("User created:", user);
} catch (err) {
  console.error("Create error:", err.message);
}
```

Read – find(), findById()

- **Model.find()**
  - Returns an array of **matching documents**
  - Accepts a filter query (optional)
- **Model.findById()**
  - Finds a document **by its _id**
  - Returns a single document (or null if not found)

```
// Filter: get users older than 18
const adults = await User.find({age: {$gt: 18}});

const user = await User.findById("65f0a2c8...");
```

Update - findByIdAndUpdate()

- Finds a document by _id
- Updates it with new values
- Saves the change to the database

```
const updatedUser = await
User.findByIdAndUpdate(
  userId,
  { name: "New Name" }
);
```

Delete: findByIdAndDelete()

- Finds a document by _id
- Deletes it from the database
- Returns the deleted document

```
const deletedUser = await
User.findByIdAndDelete(userId);
```

Exercise 28.11.2025

Build APIs to manage users

- Note:
  - operations: CRUD
  - MongoAtlas to store data
  - Postman to test APIs
  - errors handlings