Cheat Sheet:

OTHER//

Git Basics

Initialize a repo: git init

Clone a Repo: git clone <repo-url>

Check status: git status

Stage changes: git add <file> or git add .

Commit Changes: git commit –m “Your Message”

View Logs: git log

Push changes: git push origin <branch name>

Pull Changes git pull origin <branch name>

Create a branch : git branch <branch-name>

Switch branch: git checkout <branch-name>

Merge Branches: git merge <branch-name>

Undo changes: git restore <file> (unstage file)

Git reset –hard (reset to last commit):

Common Errors

//BACKEND

PACKAGE.JSON

"scripts": {

"start": "node backend/server.js",

"server": "nodemon backend/server.js",

"client": "npm run dev --prefix frontend",

"dev": "concurrently \"npm run server\" \"npm run client\""

}

Explanation:

This section of code is part of a package.json file, typically used in a **full-stack application** setup with both a backend (Node.js) and frontend (e.g., React). Here's what each script does:

**Scripts Explained**

**start**:

"start": "node backend/server.js"

Runs the backend server using **Node.js**.

This is the production-ready script that directly executes the server file.

Commonly used by hosting platforms like Heroku to start the backend server.

**server**:

"server": "nodemon backend/server.js"

Runs the backend server using **Nodemon**.

Nodemon automatically restarts the server whenever you make changes to the code.

This is typically used during development to improve productivity.

**client**:

"client": "npm run dev --prefix frontend"

Runs the **frontend development server**.

The --prefix frontend flag tells npm to look for the package.json in the frontend directory.

The frontend directory contains the frontend code (e.g., React, Angular, or Vue).

**dev**:

"dev": "concurrently \"npm run server\" \"npm run client\""

Runs both the backend (npm run server) and frontend (npm run client) simultaneously.

Uses the **concurrently** package, which allows multiple scripts to run concurrently in one terminal.

This is useful during development, as it eliminates the need to open two terminals to start both the backend and frontend servers.

**When and How to Use These Scripts**

**Development**:

Run npm run dev to start both servers (backend and frontend).

Example:

npm run dev

You’ll see both the backend and frontend logs in the terminal.

**Production**:

Run npm start to start only the backend (e.g., on a production server).

Example:

npm start

**Individual Testing**:

Run npm run server to work on the backend alone.

Run npm run client to work on the frontend alone.

**Prerequisites for This Setup**

The concurrently package should be installed:

npm install concurrently

A frontend folder with its own package.json and dev script must exist.

Example frontend package.json:

{

"scripts": {

"dev": "react-scripts start"

}

}

Node.js Server Basics

(commonjs (require) vs es6 modules(import))

Setup a basic server in Node.js

1. NPM Install express

ES6 Module

Import express from ‘express’

const app = express();

app.use(express.json()); //Parse Json bodies

//Basic Routes

App.get(‘/’, (req, res) => {

Res.send(‘Hello world’);

});

app.listen(3000, () => {

console.log()`Server running on ${port}`});

Simple Mongoose Connect

Npm I mongoose

Import mongoose from ‘mongoose’

Const connectDB = async () => {  
 try {

Const conn = await mongoose.connect(process.env.MONGO\_URI);

Console.log(`Mongo DB Connected: ${conn.connection.host`});

} catch(error) {

Console.error(`Errpr: ${error.message}`);

process.exit(1);

}

};

export default connectDB;

CRUD (CREATE, READ, UPDATE, DELETE) ARE THE BASIC SET OF OPERATIONS THAT ALLOW USERS TO INTERACT WITH THE MONGODB SERVER

CREATE or Insert are used to insert or add new documents in the collection. If a collection does not exist, then it will create a new collection in the database.

|  |  |
| --- | --- |
| Method | Description |
| Db.collection.insertOne() | Inserts single document in the collection |
| Db.collection.insertMany() | Insert multiple documents in the collection |
| Db.createCollection() | Used to create an empty collection |

READ Operations is used to retrieve documents from the collection or query a collection for a document.

|  |  |
| --- | --- |
| Method | Description |
| Db.collection.find() | Retrieve documents from the collection |

Pretty() method is used to decorate the result such that it is easy to read.

Update operations are used to update or modify the existing document in the collection

|  |  |
| --- | --- |
| **Method** | **Description** |
| **db.collection.updateOne()** | It is used to update a single document in the collection that satisfy the given criteria. |
| **db.collection.updateMany()** | It is used to update multiple documents in the collection that satisfy the given criteria. |
| **db.collection.replaceOne()** | It is used to replace single document in the collection that satisfy the given criteria. |

Delete operation are used to delete or remove the documents from a collection.

| **Method** | **Description** |
| --- | --- |
| **db.collection.deleteOne()** | It is used to delete a single document from the collection that satisfy the given criteria. |
| **db.collection.deleteMany()** | It is used to delete multiple documents from the collection that satisfy the given criteria. |

Example

const Item = mongoose.model('Item', { name: String });

// Create

const newItem = new Item({ name: 'Sample' });

await newItem.save();

// Read

const items = await Item.find();

// Update

await Item.updateOne({ \_id: 'id' }, { name: 'Updated Name' });

// Delete

await Item.deleteOne({ \_id: 'id' });

Example (TODO APP)

1. Create model or schema for tasks

Create CRUD ROUTES

// Create a new task

router.post('/', async (req, res) => {

try {

const task = new Task({ title: req.body.title });

const savedTask = await task.save();

res.status(201).json(savedTask);

} catch (err) {

res.status(400).json({ error: err.message });

}

});

// Read all tasks

router.get('/', async (req, res) => {

try {

const tasks = await Task.find();

res.json(tasks);

} catch (err) {

res.status(500).json({ error: err.message });

}

});

// Update a task

router.put('/:id', async (req, res) => {

try {

const updatedTask = await Task.findByIdAndUpdate(

req.params.id,

{ completed: req.body.completed },

{ new: true }

);

res.json(updatedTask);

} catch (err) {

res.status(400).json({ error: err.message });

}

});

// Delete a task

router.delete('/:id', async (req, res) => {

try {

await Task.findByIdAndDelete(req.params.id);

res.json({ message: 'Task deleted successfully' });

} catch (err) {

res.status(500).json({ error: err.message });

}

});

module.exports = router;

\*\*import routes into main server file\*\*

MODELS/ SCHEMAS (MONGOOSE)

In **Mongoose**, a **schema** and a **model** serve different purposes, but they work together to define and interact with data in a MongoDB collection. Here's a detailed breakdown of their differences:

**Schema**

* **Definition**: A schema is a blueprint or structure that defines the shape of documents in a MongoDB collection.
* **Purpose**: It specifies the fields, their types, and any validation or default values for a document.
* **Usage**: It's used to enforce a structure for the data, ensuring consistency and allowing validation.
* **Example**:
* const mongoose = require('mongoose');
* // Define a schema
* const userSchema = new mongoose.Schema({
* name: { type: String, required: true },
* email: { type: String, required: true, unique: true },
* age: { type: Number, default: 18 }
* });

Here, userSchema defines the structure of the data in the collection (e.g., name, email, age).

**Model**

* **Definition**: A model is a wrapper for the schema that provides the interface for interacting with the MongoDB database.
* **Purpose**: It allows you to create, read, update, and delete (CRUD) documents in the database based on the schema.
* **Usage**: Models represent collections and provide methods to query and manipulate the data.
* **Example**:
* // Create a model
* const User = mongoose.model('User', userSchema);

Here, User is the model, and it connects the schema to the users collection in MongoDB. It provides methods like find, create, and update.

**Key Differences**

| **Aspect** | **Schema** | **Model** |
| --- | --- | --- |
| **Definition** | Blueprint of the document's structure and rules. | Wrapper for the schema to interact with the database. |
| **Purpose** | Defines fields, types, validation, defaults, etc. | Provides methods to perform database operations. |
| **Relationship** | Used to create a model. | Built on top of a schema. |
| **Usage** | For defining the data structure and constraints. | For querying and modifying the database. |

**Putting It Together**

const mongoose = require('mongoose');

// Step 1: Define a Schema

const blogSchema = new mongoose.Schema({

title: { type: String, required: true },

content: { type: String, required: true },

author: { type: String },

publishedAt: { type: Date, default: Date.now }

});

// Step 2: Create a Model

const Blog = mongoose.model('Blog', blogSchema);

// Step 3: Use the Model

async function createBlog() {

const blog = new Blog({

title: "My First Blog",

content: "This is the content of the blog.",

author: "Toni King"

});

await blog.save(); // Saves the blog document to the `blogs` collection

}

createBlog();

In summary: The **schema** is what the data is, and the **model** is how you work with that data.

TLDR: Schema defines what the data looks like (structures and rules) like a blueprint. *“Each user has a name (string), email (string), and age (number).”* While a model provides the tools to interact with that data. It acts as the bridge between the code and the database. *"Using the model, you can create a user, find all users, update a user, or delete a user."*

SCHEMAS ADVANCED

Absolutely! In **Mongoose**, schemas are incredibly flexible and allow you to define various data types, validation rules, default values, and even custom options. Here's a detailed guide:

**Data Types**

Mongoose supports several data types you can use in your schemas:

| **Data Type** | **Usage Example** |
| --- | --- |
| String | name: { type: String } |
| Number | age: { type: Number } |
| Boolean | isAdmin: { type: Boolean } |
| Date | createdAt: { type: Date } |
| Array | tags: { type: [String] } |
| ObjectId | user: { type: mongoose.Schema.Types.ObjectId, ref: 'User' } |
| Buffer | file: { type: Buffer } |
| Mixed (any type) | any: { type: mongoose.Schema.Types.Mixed } |
| Map | meta: { type: Map, of: String } |

**Schema Options**

Each field can have additional options for more control:

**Common Field Options**

| **Option** | **Description** | **Example** |
| --- | --- | --- |
| type | Specifies the data type. | name: { type: String } |
| required | Makes the field mandatory. | email: { type: String, required: true } |
| unique | Ensures the value is unique in the collection. | email: { type: String, unique: true } |
| default | Sets a default value. | createdAt: { type: Date, default: Date.now } |
| enum | Restricts the field to specific values. | status: { type: String, enum: ['active', 'inactive'] } |
| min / max | Sets numeric boundaries for Number fields. | age: { type: Number, min: 18, max: 65 } |
| match | Validates a String with a regular expression. | email: { type: String, match: /.+\@.+\..+/ } |
| validate | Custom validator function for any field. | age: { type: Number, validate: value => value > 0 } |
| trim | Removes leading/trailing whitespace (for Strings). | name: { type: String, trim: true } |
| lowercase/uppercase | Converts a String to lower/upper case. | email: { type: String, lowercase: true } |

**Advanced Field Options**

**Embedded Objects**: You can embed objects or sub-documents directly in a schema.

const userSchema = new mongoose.Schema({

name: { type: String, required: true },

address: {

street: { type: String },

city: { type: String },

zip: { type: String }

}

});

**Arrays of Sub-Documents**: You can store an array of sub-documents.

const blogSchema = new mongoose.Schema({

title: { type: String, required: true },

comments: [

{

body: { type: String },

date: { type: Date, default: Date.now }

}

]

});

**Referencing Other Documents**: You can reference documents in other collections using ObjectId.

const postSchema = new mongoose.Schema({

title: { type: String },

author: { type: mongoose.Schema.Types.ObjectId, ref: 'User' }

});

**Custom Validation**

You can add your own validation logic to a schema field:

const productSchema = new mongoose.Schema({

price: {

type: Number,

validate: {

validator: function (value) {

return value >= 0; // Price must be non-negative

},

message: props => `${props.value} is not a valid price!`

}

}

});

**Schema-Level Options**

You can pass additional options when defining the schema:

const schemaOptions = {

timestamps: true, // Adds `createdAt` and `updatedAt` fields

toJSON: { virtuals: true }, // Include virtual fields in JSON output

toObject: { virtuals: true } // Include virtual fields in object output

};

const userSchema = new mongoose.Schema({

name: { type: String, required: true }

}, schemaOptions);

**Virtual Fields**

Virtuals allow you to define computed properties that are not stored in the database:

userSchema.virtual('fullName').get(function () {

return `${this.firstName} ${this.lastName}`;

});

**Middleware**

Schemas can define middleware to hook into lifecycle events like save, find, etc.:

userSchema.pre('save', function (next) {

console.log('A user is being saved:', this);

next();

});

Etc:

ROUTES NOTES

// @desc Auth user & get token

// @route POST /api/users/auth

// @access Public

MIDDLE WARE 101

**Middleware**: Functions that have access to req, res, and next. They can modify the request/response objects or handle errors.

Exactly! Middleware is a function that acts as a **bridge** or **layer** between the backend and frontend (or other parts of the backend). It's part of the **request-response cycle** in an Express application and allows you to modify, process, or handle incoming requests and outgoing responses.

**How Middleware Works**

1. **Request Comes In**: The client (frontend) sends a request to the server (backend).
2. **Middleware Processes the Request**: Middleware functions can:
   * Inspect or modify the request (e.g., add authentication data, parse JSON).
   * Perform actions (e.g., log data, check permissions, or validate inputs).
3. **Pass to Next Middleware or Route Handler**: After processing, middleware can pass the request to:
   * The next middleware function in the chain.
   * The route handler (if it's the last middleware).
4. **Response is Sent**: Middleware or the route handler sends a response back to the client.

**Key Features of Middleware**

* **Access to Request and Response Objects**: Middleware functions can read and modify req (request) and res (response) objects.
* **Control the Flow**: Middleware uses the next() function to move to the next middleware or route handler. If next() isn’t called, the request-response cycle is halted.
* **Can Handle Errors**: Middleware can catch and process errors in the application.

**Types of Middleware**

1. **Built-in Middleware**: Provided by Express to handle common tasks.
   * Example: express.json() parses JSON data in the request body.
   * app.use(express.json());
2. **Third-party Middleware**: Libraries or packages that add functionality.
   * Example: cors for Cross-Origin Resource Sharing.
   * import cors from 'cors';
   * app.use(cors());
3. **Custom Middleware**: Your own functions to handle specific tasks.
   * Example: Logging requests.
   * const logger = (req, res, next) => {
   * console.log(`${req.method} ${req.url}`);
   * next(); // Pass to the next middleware or route handler
   * };
   * app.use(logger);
4. **Error-Handling Middleware**: Middleware specifically for processing errors. Recognized by having four arguments: (err, req, res, next).
   * Example:
   * const errorHandler = (err, req, res, next) => {
   * res.status(500).json({ message: err.message });
   * };
   * app.use(errorHandler);

**Middleware Example**

import express from 'express';

const app = express();

// Built-in middleware to parse JSON

app.use(express.json());

// Custom middleware to log requests

app.use((req, res, next) => {

console.log(`${req.method} request to ${req.url}`);

next(); // Pass control to the next middleware or route

});

// Route handler

app.get('/', (req, res) => {

res.send('Hello, Middleware!');

});

// Error-handling middleware

app.use((err, req, res, next) => {

console.error(err.stack);

res.status(500).send('Something broke!');

});

app.listen(3000, () => {

console.log('Server is running on port 3000');

});

**Middleware in Action**

* **Frontend Request**: A user makes a request to GET /.
* **Logging Middleware**: Logs the method (GET) and URL (/).
* **Route Handler**: Responds with Hello, Middleware!.
* **Error Handling**: If an error occurs, the error-handling middleware sends a 500 status.

**Why Use Middleware?**

Middleware is essential for:

* **Organizing Code**: Breaking logic into reusable, modular functions.
* **Pre-Processing Requests**: Parsing, validating, or modifying incoming data.
* **Adding Functionality**: Authentication, logging, CORS, etc.
* **Error Handling**: Managing and responding to application errors.

Middleware ensures your backend processes requests systematically and efficiently. If you'd like to dive deeper into any part, let me know!

MIDDLE WARE ERROR HANDLER:

const notFound = (req, res, next) => {

const error = new Error(`Not Found - ${req.originalUrl}`);

res.status(404);

next(error);

};

const errorHandler = (err, req, res, next) => {

let statusCode = res.statusCode === 200 ? 500 : res.statusCode;

let message = err.message;

// If Mongoose not found error, set to 404 and change message

if (err.name === 'CastError' && err.kind === 'ObjectId') {

statusCode = 404;

message = 'Resource not found';

}

res.status(statusCode).json({

message: message,

stack: process.env.NODE\_ENV === 'production' ? null : err.stack,

});

};

export { notFound, errorHandler };

code breakdown:

notFound: designed to catch routes or endpoints that don’t match any defined routes in your application, if no route handler is found it triggers an error. Handles requests to unknown routes by setting a 404 status and creating an error object with information about the missing resource.

 **req.originalUrl**: This captures the URL that the client tried to access.

 **new Error(message)**: Creates a new error object with a custom message, indicating the resource wasn’t found.

 **res.status(404)**: Sets the HTTP status code to 404 (Not Found) for the response.

 **next(error)**: Passes the error to the next middleware (typically the errorHandler) for further processing.

errorHandler: The errorHandler function is the central middleware for handling all errors in the application. It processes errors passed via next() or those thrown during request handling. Provides a centralized way to handle errors consistently across the application. Ensures meaningful error responses are sent to the client.

 **Determine the Status Code**:

* res.statusCode === 200 ? 500 : res.statusCode: If the current status code is 200 (default for successful responses), change it to 500 (internal server error). Otherwise, keep the existing status code.

 **Mongoose-Specific Error Handling**:

* Checks if the error is related to a Mongoose CastError (e.g., trying to query a database with an invalid ObjectId).
* If so, sets the status code to 404 and modifies the error message to 'Resource not found'.

 **Respond with JSON**:

* Sends a JSON response containing:
  + **message**: The error message (customized if needed).
  + **stack**: The stack trace of the error (useful for debugging). Hidden in production (NODE\_ENV === 'production') for security reasons.

To use these middleware functions, you would typically include them at the end of your route definitions in an Express app:

// Not Found Middleware (catch undefined routes)

app.use(notFound);

// Error Handler Middleware (process all errors)

app.use(errorHandler);

DATABASE DESIGN

**Database Design 101**

Database design is the process of organizing and structuring data to efficiently store, manage, and retrieve it. Whether you’re using a relational database (like MySQL or PostgreSQL) or a NoSQL database (like MongoDB), good design is key to scalability, performance, and maintainability.

Here’s a step-by-step guide:

**1. Understand Your Requirements**

Before designing the database, gather details about the application's requirements:

* What entities (things) need to be stored?
* What relationships exist between those entities?
* What queries will be performed frequently?
* What constraints or rules apply to the data?

**2. Choose a Database Type**

* **Relational Databases** (SQL):
  + Store data in tables with rows and columns.
  + Use **SQL** for queries.
  + Best for structured data with relationships.
  + Examples: MySQL, PostgreSQL, SQLite.
* **NoSQL Databases**:
  + Flexible schemas, designed for large-scale and unstructured data.
  + Examples:
    - **Document-based** (e.g., MongoDB): Data stored in JSON-like documents.
    - **Key-Value** (e.g., Redis): Fast lookups for key-value pairs.
    - **Graph** (e.g., Neo4j): Focused on relationships between data.
    - **Wide-Column** (e.g., Cassandra): Optimized for large-scale analytical queries.

**3. Identify Entities**

Entities represent objects or things in your application. For example:

* In an e-commerce app: **Users**, **Products**, **Orders**.
* In a library system: **Books**, **Members**, **Loans**.

**4. Define Attributes**

Attributes are the properties of each entity. For example:

* **User**:
  + id: Unique identifier
  + name: String
  + email: String
  + created\_at: Date
* **Order**:
  + id: Unique identifier
  + user\_id: Foreign key to the User
  + total\_price: Number
  + status: String

**5. Determine Relationships**

Entities often relate to each other. There are three main types of relationships:

1. **One-to-One**:
   * Example: A user has one profile.
   * Relational DB: Use foreign keys or a separate table.
   * NoSQL: Embed related data in the same document.
2. **One-to-Many**:
   * Example: A user can place many orders.
   * Relational DB: Use foreign keys in the "many" side (e.g., orders table).
   * NoSQL: Embed orders in the user document or use references.
3. **Many-to-Many**:
   * Example: Products and Categories (a product can belong to many categories, and a category can have many products).
   * Relational DB: Use a join table.
   * NoSQL: Use arrays or separate collections.

**6. Normalize Your Data**

Normalization organizes data to minimize redundancy and ensure consistency:

* **1NF (First Normal Form)**: Eliminate duplicate columns. Each column should have atomic values (one value per cell).
* **2NF (Second Normal Form)**: Ensure all non-primary attributes depend on the primary key.
* **3NF (Third Normal Form)**: Ensure no transitive dependencies (non-key attributes shouldn't depend on other non-key attributes).

**Example**: Instead of storing a user's address in every order, store it in a Users table and reference it from the Orders table.

**7. Denormalize for Performance (If Necessary)**

While normalization reduces redundancy, it can make queries slower due to multiple joins. In read-heavy applications, you can denormalize:

* Example: Store frequently accessed data (e.g., user's name) in multiple places to avoid joins.

**8. Design Primary and Foreign Keys**

* **Primary Key**: A unique identifier for each row in a table (e.g., id).
* **Foreign Key**: A field in one table that references the primary key in another.

**9. Indexing for Speed**

Indexes improve query performance by allowing the database to find rows faster:

* Example: Add an index on email for quick user lookups.
* Avoid over-indexing—it can slow down writes.

**10. Handle Constraints**

Constraints enforce data integrity:

* **Unique**: Ensure no duplicate values (e.g., emails).
* **Not Null**: Prevent empty values.
* **Check**: Validate data with rules (e.g., price > 0).
* **Foreign Key**: Enforce valid relationships.

**11. Backup and Scalability**

* Plan for backups to avoid data loss.
* Ensure scalability:
  + **Vertical Scaling**: Add resources to the same server.
  + **Horizontal Scaling**: Distribute the database across multiple servers.

**12. Sample Relational Database Design**

For a blogging application:

**Entities:**

1. **Users**:
   * id (Primary Key)
   * name
   * email (Unique)
2. **Posts**:
   * id (Primary Key)
   * title
   * content
   * user\_id (Foreign Key referencing Users)
3. **Comments**:
   * id (Primary Key)
   * content
   * post\_id (Foreign Key referencing Posts)
   * user\_id (Foreign Key referencing Users)

**Relationships:**

* A **User** can have many **Posts** (One-to-Many).
* A **Post** can have many **Comments** (One-to-Many).
* A **Comment** belongs to one **User** and one **Post**.

**13. Sample NoSQL Database Design**

For the same blogging application in MongoDB:

{

"users": [

{ "\_id": 1, "name": "Alice", "email": "alice@example.com" }

],

"posts": [

{

"\_id": 1,

"title": "My First Post",

"content": "This is a post.",

"user\_id": 1,

"comments": [

{ "content": "Great post!", "user\_id": 2 }

]

}

]

}

Here:

* Comments are embedded within posts for easier reads.
* The user\_id still references the user.

EXAMPLE: Design a database for a Library Management System using a relation database

**1. Understand Requirements**

We want to build a system to manage books in a library. The system should:

* Store details about books, authors, and library members.
* Track who has borrowed which book and when it is due.
* Manage genres for books (e.g., fiction, science fiction).
* Provide functionality to add, update, and retrieve information about books, authors, and members.

**2. Choose a Database Type**

We’ll use a **relational database** (e.g., MySQL or PostgreSQL) since:

* The data is structured.
* Relationships between entities (e.g., books and authors) are important.

**3. Identify Entities**

Based on the requirements, we identify the main entities:

1. **Books**: General details about each book.
2. **Authors**: Information about book authors.
3. **Members**: Library users who can borrow books.
4. **Loans**: Tracks which member has borrowed which book and the due date.
5. **Genres**: Categories to classify books.

**4. Define Attributes**

For each entity, define its attributes (fields):

* **Books**:
  + id: Primary Key, unique identifier for each book.
  + title: Title of the book.
  + summary: Short description of the book.
  + isbn: Unique ISBN number for the book.
  + genre\_id: Foreign Key, links to the Genres table.
  + author\_id: Foreign Key, links to the Authors table.
* **Authors**:
  + id: Primary Key, unique identifier.
  + name: Full name of the author.
  + bio: Short biography of the author (optional).
* **Members**:
  + id: Primary Key, unique identifier.
  + name: Full name of the member.
  + email: Contact email for the member.
  + membership\_start: Date the membership started.
* **Loans**:
  + id: Primary Key, unique identifier.
  + book\_id: Foreign Key, links to the Books table.
  + member\_id: Foreign Key, links to the Members table.
  + borrow\_date: Date the book was borrowed.
  + due\_date: Date the book is due for return.
* **Genres**:
  + id: Primary Key, unique identifier.
  + name: Name of the genre (e.g., Fiction, Non-fiction).

**5. Determine Relationships**

1. **Books and Authors**: One book has one author, but one author can write many books. (One-to-Many)
2. **Books and Genres**: One book can belong to multiple genres, and one genre can apply to multiple books. (Many-to-Many)
3. **Books and Loans**: One book can appear multiple times in the Loans table to track its borrowing history. (One-to-Many)
4. **Loans and Members**: Each loan is associated with one member, but a member can borrow multiple books. (One-to-Many)

**6. Normalize Data**

1. Break down data to minimize redundancy:
   * Store genres in their own table instead of duplicating genre names in the Books table.
   * Keep authors in a separate table instead of storing their names multiple times.
2. Create a **join table** for the many-to-many relationship between Books and Genres:
   * **BooksGenres**:
     + book\_id: Foreign Key (Books table).
     + genre\_id: Foreign Key (Genres table).

**7. Database Schema**

Here’s how the schema looks:

**Books Table**

| **Field** | **Type** | **Constraints** |
| --- | --- | --- |
| id | INT | Primary Key |
| title | VARCHAR(255) | Not Null |
| summary | TEXT |  |
| isbn | VARCHAR(13) | Unique |
| genre\_id | INT | Foreign Key (Genres) |
| author\_id | INT | Foreign Key (Authors) |

**Authors Table**

| **Field** | **Type** | **Constraints** |
| --- | --- | --- |
| id | INT | Primary Key |
| name | VARCHAR(255) | Not Null |
| bio | TEXT |  |

**Members Table**

| **Field** | **Type** | **Constraints** |
| --- | --- | --- |
| id | INT | Primary Key |
| name | VARCHAR(255) | Not Null |
| email | VARCHAR(255) | Unique |
| membership\_start | DATE | Not Null |

**Loans Table**

| **Field** | **Type** | **Constraints** |
| --- | --- | --- |
| id | INT | Primary Key |
| book\_id | INT | Foreign Key (Books) |
| member\_id | INT | Foreign Key (Members) |
| borrow\_date | DATE | Not Null |
| due\_date | DATE | Not Null |

**Genres Table**

| **Field** | **Type** | **Constraints** |
| --- | --- | --- |
| id | INT | Primary Key |
| name | VARCHAR(255) | Unique, Not Null |

**BooksGenres Table (Join Table for Many-to-Many)**

| **Field** | **Type** | **Constraints** |
| --- | --- | --- |
| book\_id | INT | Foreign Key (Books) |
| genre\_id | INT | Foreign Key (Genres) |

**8. Example Queries**

Here are some queries we can run with this design:

**Find all books written by a specific author**:

SELECT title

FROM Books

WHERE author\_id = (SELECT id FROM Authors WHERE name = 'J.K. Rowling');

**Get the borrowing history of a specific member**:

SELECT Books.title, Loans.borrow\_date, Loans.due\_date

FROM Loans

JOIN Books ON Loans.book\_id = Books.id

WHERE Loans.member\_id = (SELECT id FROM Members WHERE name = 'Alice Smith');

**List all genres for a specific book**:

SELECT Genres.name

FROM BooksGenres

JOIN Genres ON BooksGenres.genre\_id = Genres.id

WHERE BooksGenres.book\_id = (SELECT id FROM Books WHERE title = '1984');

**Find overdue loans**:

SELECT Members.name, Books.title, Loans.due\_date

FROM Loans

JOIN Members ON Loans.member\_id = Members.id

JOIN Books ON Loans.book\_id = Books.id

WHERE Loans.due\_date < CURDATE();

**9. Visualizing the Relationships**

A simple UML diagram:

* **Books** (1) ↔ (1) **Authors**
* **Books** (Many) ↔ (Many) **Genres**
* **Books** (1) ↔ (Many) **Loans**
* **Members** (1) ↔ (Many) **Loans**

Example with nonrelational database (MONGO)

Let's design a **Library Management System** using a **non-relational database** like **MongoDB**.

**Scenario: Library Management System**

The system should:

1. Store details about books (e.g., title, summary, author, genre, ISBN).
2. Manage information about library members.
3. Track which members borrowed which books and when they are due.
4. Organize books by genre and allow searching by author, title, or genre.

**Key Differences in NoSQL Design**

In NoSQL, we focus on collections and documents:

* Collections: Groups of related documents (similar to tables in SQL).
* Documents: JSON-like objects that store data.

Instead of rigid schemas and relationships, NoSQL databases prioritize flexibility, often embedding related data within documents to optimize for read operations.

**Step-by-Step Database Design**

**1. Identify Collections**

For the Library Management System, we need:

1. **Books**: Stores details about each book.
2. **Members**: Stores information about library members.
3. **Loans**: Tracks borrowing information.

**2. Design Documents and Relationships**

We'll use **denormalization** (embedding related data) where it improves performance and **normalization** (referencing related data) where it avoids excessive duplication.

**Books Collection**

{

"\_id": "1",

"title": "The Great Gatsby",

"summary": "A classic novel by F. Scott Fitzgerald.",

"isbn": "9780743273565",

"authors": [

{ "id": "101", "name": "F. Scott Fitzgerald" }

],

"genres": ["Fiction", "Classic"],

"copies\_available": 3

}

* Embedded **authors** for fast lookups (denormalized).
* Genres stored as an array.
* copies\_available tracks the count of available copies.

**Members Collection**

{

"\_id": "501",

"name": "Alice Smith",

"email": "alice.smith@example.com",

"membership\_start": "2023-01-01",

"borrowed\_books": [

{

"book\_id": "1",

"borrow\_date": "2024-11-01",

"due\_date": "2024-11-15"

}

]

}

* Borrowed books are embedded for quick access to what each member has borrowed.

**Loans Collection (Optional)**

If detailed loan tracking is required, use a separate collection:

{

"\_id": "301",

"book\_id": "1",

"member\_id": "501",

"borrow\_date": "2024-11-01",

"due\_date": "2024-11-15"

}

* This design is normalized to avoid embedding too much loan data into books or members.

**3. Optimize Relationships**

* **Authors and Genres**: For flexibility, you might store these in their own collections:
* // Authors Collection
* {
* "\_id": "101",
* "name": "F. Scott Fitzgerald",
* "bio": "An American novelist and short story writer."
* }
* // Genres Collection
* {
* "\_id": "201",
* "name": "Fiction"
* }

**4. Example Queries**

**Find All Books by an Author**

db.books.find({ "authors.name": "F. Scott Fitzgerald" });

**Get All Books in a Genre**

db.books.find({ genres: "Fiction" });

**Find Books Borrowed by a Member**

db.members.find({ \_id: "501" }, { borrowed\_books: 1 });

**Track Overdue Loans**

db.loans.find({ due\_date: { $lt: new Date() } });

**5. Advantages of NoSQL for This Case**

1. **Flexible Schema**: Easily add fields without modifying the structure.
2. **Optimized for Reads**: Embedding relationships (e.g., authors in books) makes lookups faster.
3. **Scalable**: Horizontal scaling is easier with NoSQL.

**6. Visualizing the Data Model**

**Collections:**

* **Books**:
  + Title, summary, ISBN, authors (embedded), genres, copies\_available.
* **Members**:
  + Name, email, membership\_start, borrowed\_books (embedded).
* **Loans** (Optional):
  + book\_id, member\_id, borrow\_date, due\_date.

**Relationships:**

* **Books → Authors**: Embedded for quick lookups.
* **Books → Genres**: Array in the Books collection.
* **Members → Borrowed Books**: Embedded to track each member's activity.

**7. Final Thoughts**

* **Embed** relationships when the data is frequently accessed together (e.g., books and authors).
* **Reference** relationships when data is reused across collections (e.g., genres or detailed loan tracking).

FILE STRUCTURES:

my-crud-app/

├── models/ # Database schemas (e.g., Task.js)

├── routes/ # Application routes (e.g., tasks.js)

├── public/ # Static files (optional, for frontend assets)

├── views/ # Frontend templates (if using templating engines)

├── .gitignore # Git ignore file

├── package.json # Node.js project metadata

├── server.js # Main application entry point

├── README.md # Project documentation

 **models/**:

* Store MongoDB schemas using Mongoose.

 **routes/**:

* Keep route definitions (e.g., tasks.js for CRUD routes).

 **public/**:

* Store static files like CSS, JS, and images.

 **views/**:

* If using a templating engine (e.g., EJS, Pug), store view files here.

 **server.js**:

* Entry point of the application where you configure Express.js, middleware, and database connections.

 **README.md**:

* Write project documentation, including how to set up and run the app.

SETTING UP DEVELOPMENT ENVIRONMENT

ENV VARIABLES

Install dotenv. Create .env file (make sure to add to .gitignore file

Examples .env file

PORT=3000

MONGO\_URI=mongodb://localhost:27017/myDatabase

Install nodemon for auto restart then update package.json with dev script

"scripts": { "start": "node server.js", "dev": "nodemon server.js" }

CSS BASICS

Here’s a guide to the \*\*CSS basics\*\* covering common properties, units, and layout techniques. This is a great reference to include in your cheatsheet!

---

### \*\*1. Common CSS Properties\*\*

#### \*\*Text Styling\*\*

- \*\*Font properties\*\*:

```css

font-family: 'Arial', sans-serif;

font-size: 16px;

font-weight: bold; /\* or normal, lighter, bolder \*/

font-style: italic; /\* or normal \*/

text-transform: uppercase; /\* capitalize, lowercase \*/

text-align: center; /\* left, right, justify \*/

line-height: 1.5; /\* Space between lines \*/

```

- \*\*Color\*\*:

```css

color: #333; /\* Text color \*/

background-color: #f4f4f4; /\* Background color \*/

```

#### \*\*Box Model\*\*

- \*\*Spacing\*\*:

```css

margin: 10px; /\* Outer spacing \*/

padding: 15px; /\* Inner spacing \*/

```

- \*\*Sizing\*\*:

```css

width: 50%; /\* or px, em, rem, vw \*/

height: 100px; /\* Height of an element \*/

max-width: 100%; /\* Prevent overflow \*/

```

- \*\*Borders and shadows\*\*:

```css

border: 1px solid #ddd; /\* border-style: solid, dashed, dotted \*/

border-radius: 8px; /\* Rounded corners \*/

box-shadow: 2px 4px 10px rgba(0, 0, 0, 0.2); /\* Shadow effect \*/

```

#### \*\*Positioning\*\*

- \*\*Display\*\*:

```css

display: block; /\* or inline, flex, grid, inline-block, none \*/

visibility: hidden; /\* or visible \*/

```

- \*\*Positioning\*\*:

```css

position: relative; /\* or absolute, fixed, sticky \*/

top: 10px; /\* Moves element down \*/

left: 20px; /\* Moves element to the right \*/

```

- \*\*Overflow\*\*:

```css

overflow: hidden; /\* or scroll, auto, visible \*/

```

---

### \*\*2. Units\*\*

#### \*\*Absolute Units\*\*

- Pixels (`px`): Fixed size, e.g., `width: 300px;`

- Points (`pt`): Used in print styles.

#### \*\*Relative Units\*\*

- \*\*Font size-based\*\*:

- `em`: Relative to the parent’s font size.

```css

font-size: 2em; /\* Twice the size of parent \*/

```

- `rem`: Relative to the root element’s font size.

```css

font-size: 1.5rem; /\* 1.5 times root font size \*/

```

- \*\*Viewport-based\*\*:

- `vw`: Percentage of the viewport width.

```css

width: 50vw; /\* Half the viewport width \*/

```

- `vh`: Percentage of the viewport height.

```css

height: 100vh; /\* Full viewport height \*/

```

---

### \*\*3. Layout Techniques\*\*

#### \*\*Flexbox\*\*

A layout model for aligning items in rows or columns.

```css

.container {

display: flex;

justify-content: center; /\* Align horizontally: center, space-between, etc. \*/

align-items: center; /\* Align vertically: center, flex-start, flex-end \*/

flex-wrap: wrap; /\* Wrap items to the next row if needed \*/

}

.item {

flex: 1; /\* Grow/shrink with the container \*/

}

```

#### \*\*Grid\*\*

A powerful 2D layout system for designing complex layouts.

```css

.container {

display: grid;

grid-template-columns: repeat(3, 1fr); /\* 3 equal columns \*/

grid-gap: 10px; /\* Spacing between items \*/

}

.item {

grid-column: span 2; /\* Span across two columns \*/

}

```

#### \*\*Positioning\*\*

- \*\*Static (default)\*\*:

```css

position: static;

```

- \*\*Relative\*\*: Position relative to its normal position.

```css

position: relative;

top: 10px; /\* Moves down \*/

```

- \*\*Absolute\*\*: Positioned relative to the nearest positioned ancestor.

```css

position: absolute;

top: 0;

left: 0;

```

- \*\*Fixed\*\*: Stays in place relative to the viewport.

```css

position: fixed;

```

---

### \*\*4. Responsive Design\*\*

#### \*\*Media Queries\*\*

Adjust styles based on screen size.

```css

@media (max-width: 768px) {

body {

font-size: 14px;

}

}

```

#### \*\*Fluid Layouts\*\*

Use percentages or viewport units instead of fixed values.

```css

.container {

width: 80%;

}

```

#### \*\*CSS Grid and Flexbox for Responsiveness\*\*

Combine `grid` and `flex` for fluid layouts that adapt to different screen sizes.

---

### \*\*5. Common Patterns\*\*

- \*\*Centering with Flexbox\*\*:

```css

.center {

display: flex;

justify-content: center;

align-items: center;

height: 100vh;

}

```

- \*\*Sticky Header\*\*:

```css

header {

position: sticky;

top: 0;

background-color: #fff;

z-index: 10;

}

```

- \*\*Card Layout\*\* (Flex/Grid example):

```css

.card-container {

display: grid;

grid-template-columns: repeat(auto-fill, minmax(200px, 1fr));

gap: 20px;

}

.card {

padding: 15px;

border: 1px solid #ddd;

border-radius: 5px;

box-shadow: 0 2px 5px rgba(0, 0, 0, 0.1);

}

```

Javascript

Here’s a \*\*JavaScript Basics\*\* guide to help you with syntax reminders, common functions, and debugging tips. This is a great starting point for building out your cheatsheet!

---

### \*\*1. Syntax Reminders\*\*

#### \*\*Variable Declaration\*\*

- `let`: Block-scoped, can be reassigned.

- `const`: Block-scoped, cannot be reassigned.

- `var`: Function-scoped (avoid using in modern JavaScript).

```javascript

let x = 10;

const y = 20;

var z = 30; // Not recommended

```

#### \*\*Data Types\*\*

- \*\*Primitive Types\*\*: `string`, `number`, `boolean`, `undefined`, `null`, `bigint`, `symbol`.

- \*\*Non-Primitive Types\*\*: `object` (includes arrays, functions).

```javascript

let str = "Hello";

let num = 42;

let isActive = true;

let obj = { key: "value" };

let arr = [1, 2, 3];

```

#### \*\*Control Flow\*\*

- \*\*Conditional Statements\*\*:

```javascript

if (x > 10) {

console.log("x is greater than 10");

} else if (x === 10) {

console.log("x is equal to 10");

} else {

console.log("x is less than 10");

}

```

- \*\*Loops\*\*:

```javascript

for (let i = 0; i < 5; i++) {

console.log(i);

}

while (x > 0) {

console.log(x);

x--;

}

for (let num of arr) {

console.log(num); // Iterates over array values

}

for (let key in obj) {

console.log(key); // Iterates over object keys

}

```

---

### \*\*2. Common Functions\*\*

#### \*\*String Methods\*\*

```javascript

const str = "Hello, World!";

console.log(str.toLowerCase()); // "hello, world!"

console.log(str.toUpperCase()); // "HELLO, WORLD!"

console.log(str.includes("World")); // true

console.log(str.split(", ")); // ["Hello", "World!"]

```

#### \*\*Array Methods\*\*

```javascript

const arr = [1, 2, 3, 4];

console.log(arr.push(5)); // [1, 2, 3, 4, 5]

console.log(arr.pop()); // [1, 2, 3, 4]

console.log(arr.map(x => x \* 2)); // [2, 4, 6, 8]

console.log(arr.filter(x => x > 2)); // [3, 4]

```

#### \*\*Object Operations\*\*

```javascript

const person = { name: "Toni", age: 25 };

console.log(Object.keys(person)); // ["name", "age"]

console.log(Object.values(person)); // ["Toni", 25]

```

#### \*\*Math\*\*

```javascript

console.log(Math.max(1, 5, 10)); // 10

console.log(Math.random()); // Random number between 0 and 1

```

#### \*\*Date\*\*

```javascript

const today = new Date();

console.log(today.toDateString()); // e.g., "Mon Nov 27 2024"

console.log(today.getFullYear()); // e.g., 2024

```

---

### \*\*3. Debugging Tips\*\*

#### \*\*Using `console.log()`\*\*

- Print variables, data, or checkpoints to the console:

```javascript

console.log("Variable x:", x);

```

#### \*\*Error Handling\*\*

- \*\*Try/Catch\*\* for managing errors gracefully:

```javascript

try {

const result = riskyFunction();

console.log(result);

} catch (error) {

console.error("An error occurred:", error.message);

}

```

#### \*\*Debugging in the Browser\*\*

1. Open developer tools in your browser (`Ctrl+Shift+I` or `Cmd+Option+I`).

2. Use the \*\*Sources\*\* tab to set breakpoints in your code.

#### \*\*`debugger` Statement\*\*

- Pause execution and debug in developer tools:

```javascript

debugger;

```

#### \*\*Use Linting Tools\*\*

- Tools like \*\*ESLint\*\* can identify potential bugs or code smells.

#### \*\*Check Types\*\*

- Use `typeof` to verify data types:

```javascript

console.log(typeof x); // e.g., "number"

```

---

### \*\*4. Miscellaneous Reminders\*\*

#### \*\*Arrow Functions\*\*

- Concise syntax for writing functions:

```javascript

const add = (a, b) => a + b;

console.log(add(2, 3)); // 5

```

#### \*\*Template Literals\*\*

- Embed variables in strings:

```javascript

const name = "Toni";

console.log(`Hello, ${name}!`); // "Hello, Toni!"

```

#### \*\*Destructuring\*\*

- Simplify object/array extraction:

```javascript

const { name, age } = person;

console.log(name, age);

const [first, second] = arr;

console.log(first, second);

```

#### \*\*Modules\*\*

- Export/Import syntax:

```javascript

// file1.js

export const greet = () => "Hello!";

// file2.js

import { greet } from './file1.js';

console.log(greet()); // "Hello!"

```

---

Workflow notes

**1. Mobile-First Design**

**What is Mobile-First Design?**

* Design your website starting with the smallest screens (mobile).
* Add styles for larger screens using media queries.

**Why Use Mobile-First?**

* Prioritizes essential content for smaller screens.
* Easier to scale up designs for larger devices than scale down.
* Improves performance on mobile devices by loading only necessary resources.

Best Practices

1. **Start with Base Styles**: Write styles for the smallest screen as your default.
2. **Progressively Enhance for Larger Screens**: Use media queries to add styles for larger screens:
3. **Use Relative Units**: Use %, em, or rem for flexible and scalable designs.

**Common Breakpoints**

These breakpoints are typically used, but you can adjust them based on your project:

* **Extra Small (Mobile)**: max-width: 480px
* **Small (Tablet)**: min-width: 481px and max-width: 768px
* **Medium (Desktop)**: min-width: 769px and max-width: 1024px
* **Large (Widescreen)**: min-width: 1025px

COMMAND LINE AND NPM/ YARN BASICS

**2. npm/Yarn Basics**

**Initialization**

* **Initialize a new Node.js project**:
* npm init # Step-by-step prompts
* npm init -y # Skip prompts and use defaults
* yarn init # Yarn equivalent

**Installing Packages**

**Install a package**:

npm install <package-name>

yarn add <package-name>

**Install a specific version**:

npm install <package-name>@<version>

yarn add <package-name>@<version>

**Install globally**:

npm install -g <package-name>

yarn global add <package-name>

**Managing Dependencies**

**Save a package as a dev dependency**:

npm install <package-name> --save-dev

yarn add <package-name> --dev

**Remove a package**:

npm uninstall <package-name>

yarn remove <package-name>

**Running Scripts**

**Run a script defined in package.json**:

npm run <script-name>

yarn <script-name>

**Common scripts**:

start: Start the app.

dev: Start the development server.

test: Run tests.

build: Build the app for production.

**Updating Packages**

**Update a package**:

npm update <package-name>

yarn upgrade <package-name>

**Check outdated packages**:

npm outdated

yarn outdated

**Installing All Dependencies**

**Install dependencies from package.json**:

npm install

yarn install