**OAuth:**

**Client ID: 601162910971-3uc5s50fquo2f925fbspok29qtp1dfaq.apps.googleusercontent.com**

**Client Secret: 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**

**Backend (Node.js + SQL)**

1. **Setup**:
   * Initialize the project with npm init.
   * Install dependencies: express, sequelize, mysql2 (or your chosen SQL database).
2. **Database**:
   * Create a SQL database named blog\_craft.
   * Tables:
     + Users: For storing user data (id, username, password).
     + Posts: For blog posts (id, title, content, userId, createdAt).
   * Define relationships: One-to-many (User → Posts).
3. **APIs**:
   * **User APIs**:
     + POST /register: Create a new user.
     + POST /login: Authenticate user and generate JWT.
   * **Post APIs**:
     + GET /posts: Fetch all posts.
     + GET /posts/:id: Fetch a specific post.
     + POST /posts: Create a post (auth required).
     + PUT /posts/:id: Edit a post (auth required).
     + DELETE /posts/:id: Delete a post (auth required).
4. **Testing**:
   * Use Postman to test each endpoint.

**Frontend (React)**

1. **Setup**:
   * Use create-react-app or Vite to set up the project.
   * Install axios for API calls and react-router-dom for routing.
2. **Components**:
   * **Authentication**: Login and Register forms.
   * **Dashboard**: List of blog posts with options to edit or delete (if authorized).
   * **Post Management**:
     + Create Post: Form for adding new posts.
     + Edit Post: Form pre-filled with existing data.
   * **Post Details**: Display content of a single post.
3. **State Management**:
   * Use React Context or a state management library like Zustand or Redux for global states (auth token, user data).
4. **Styling**:
   * Use CSS frameworks like TailwindCSS or custom styling for a modern look.

**Learning Goals**

1. **SQL**:
   * Learn how to design tables, write queries (SELECT, INSERT, UPDATE, DELETE), and establish relationships.
   * Practice creating database models using Sequelize.
2. **Postman**:
   * Test each endpoint thoroughly with different scenarios (success, failure, edge cases).
3. **Authentication**:
   * Understand JWT for token-based authentication.

**Outcome**

By the end of this project, you will have:

* A fully functional full-stack blog application.
* Hands-on experience with SQL database integration and API development.
* A better understanding of React-based UI and authentication workflows.

**Step 1: Initialize the Backend**

1. Create a new folder for the backend (e.g., blog-craft-backend).
2. Open the terminal, navigate to the folder, and run:

bash

Copy code

npm init -y

This creates a package.json file.

1. Install essential packages:

bash

Copy code

npm install express sequelize mysql2 jsonwebtoken bcrypt dotenv body-parser

* + express: To create the server.
  + sequelize: For SQL ORM.
  + mysql2: To connect to MySQL.
  + jsonwebtoken: For authentication.
  + bcrypt: For hashing passwords.
  + dotenv: To manage environment variables.
  + body-parser: To parse request bodies.

1. Install development dependencies:

bash

Copy code

npm install --save-dev nodemon

Use nodemon for automatic server restarts during development.

**Step 2: Project Structure**

Create the following folder structure:

bash

Copy code

blog-craft-backend/

├── config/ # Database and environment configs

├── models/ # Sequelize models

├── routes/ # API routes

├── controllers/ # Request handlers

├── middleware/ # Middleware for authentication

├── server.js # Entry point

├── .env # Environment variables

**Step 3: Set Up Sequelize and Database**

1. **Initialize Sequelize**:  
   Run the command:

bash

Copy code

npx sequelize init

It will create folders for config, models, and migrations.

1. **Configure Database**:  
   Edit config/config.json to match your MySQL setup:

json

Copy code

{

"development": {

"username": "root",

"password": "your\_password",

"database": "blog\_craft",

"host": "127.0.0.1",

"dialect": "mysql"

}

}

1. **Create Models**:
   * **User Model**:

javascript

Copy code

// models/User.js

module.exports = (sequelize, DataTypes) => {

const User = sequelize.define("User", {

username: { type: DataTypes.STRING, allowNull: false, unique: true },

password: { type: DataTypes.STRING, allowNull: false },

});

return User;

};

* + **Post Model**:

javascript

Copy code

// models/Post.js

module.exports = (sequelize, DataTypes) => {

const Post = sequelize.define("Post", {

title: { type: DataTypes.STRING, allowNull: false },

content: { type: DataTypes.TEXT, allowNull: false },

});

Post.associate = (models) => {

Post.belongsTo(models.User, { foreignKey: "userId", as: "author" });

};

return Post;

};

1. **Run Migrations**:  
   Create the database and apply migrations:

bash

Copy code

npx sequelize db:create

npx sequelize db:migrate

**Step 4: Build REST APIs**

1. **Server Setup**:  
   Create server.js to set up the Express app:

javascript

Copy code

require("dotenv").config();

const express = require("express");

const bodyParser = require("body-parser");

const db = require("./models");

const app = express();

app.use(bodyParser.json());

// Test route

app.get("/", (req, res) => res.send("API is running"));

// Sync database and start server

db.sequelize.sync().then(() => {

app.listen(3000, () => console.log("Server running on port 3000"));

});

1. **User APIs**:  
   Add routes for registration, login, and authentication.
2. **Post APIs**:  
   Add routes for CRUD operations on posts.
3. **Define Routes**:
   * Start adding REST API routes for CRUD operations on blogs and user data.
   * Example: Create routes for adding, retrieving, updating, and deleting blog posts.
4. **Database Setup**:
   * Connect your backend to a SQL database using Sequelize.
   * Define models for users and blog posts.
5. **Authentication**:
   * Implement user authentication using jsonwebtoken and bcrypt.
6. **Environment Variables**:
   * Use the .env file for sensitive information (e.g., database credentials, JWT secrets).

**Step 1: Set Up the Basic Server with Routes**

This will ensure the server responds correctly to API calls.

1. **Edit server.js:** Create and configure a basic Express server with a test route.

javascript

Copy code

const express = require("express");

const bodyParser = require("body-parser");

const dotenv = require("dotenv");

// Initialize app and environment variables

dotenv.config();

const app = express();

// Middleware

app.use(bodyParser.json());

// Test Route

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

res.send("Welcome to Blog Craft Backend!");

});

 **Enhancing Validation**  
Add validation for request data to ensure proper inputs (e.g., validate title length, content presence).

 **Authentication & Authorization**

* Integrate **OAuth** or **JWT** to secure routes (e.g., only authenticated users can create/update/delete posts).
* Create models for User and handle user registration, login, and role-based access control.

 **Additional Features**

* Implement pagination and search for posts.
* Add support for categories/tags for better organization of blog posts.

 **Error Handling Middleware**  
Create a centralized error-handling middleware to catch errors consistently.

 **Unit and Integration Tests**  
Add testing using **Jest** or **Mocha** to validate your APIs.

 **Frontend Integration**  
Start connecting this backend with your frontend using REST API calls.

// Start the server

const PORT = process.env.PORT || 3000;

app.listen(PORT, () => {

console.log(`Server is running on http://localhost:${PORT}`);

});

1. **Restart the Server**: Stop the current server (Ctrl + C), and then restart it:

bash

Copy code

npm run dev

1. **Test on localhost:3000**: Open a browser and navigate to http://localhost:3000. You should see:

css

Copy code

Welcome to Blog Craft Backend!

**Step 2: Create Database Connection**

Now, connect the backend to your MySQL database using Sequelize.

1. **Create the Database**:
   * Open your MySQL client (e.g., MySQL Workbench).
   * Run the following SQL to create a database:

sql

Copy code

CREATE DATABASE blog\_craft;

1. **Add .env File**: In the blog-craft-backend folder, create a .env file and add:

makefile

Copy code

DB\_NAME=blog\_craft

DB\_USER=root

DB\_PASSWORD=yourpassword

DB\_HOST=localhost

DB\_DIALECT=mysql

PORT=3000

JWT\_SECRET=your\_secret\_key

1. **Set Up Sequelize Connection**: Add a config folder in blog-craft-backend and create db.js:

javascript

Copy code

const { Sequelize } = require("sequelize");

const dotenv = require("dotenv");

dotenv.config();

// Sequelize instance

const sequelize = new Sequelize(

process.env.DB\_NAME,

process.env.DB\_USER,

process.env.DB\_PASSWORD,

{

host: process.env.DB\_HOST,

dialect: process.env.DB\_DIALECT,

}

);

// Test the connection

sequelize.authenticate()

.then(() => {

console.log("Connected to the database successfully.");

})

.catch((err) => {

console.error("Unable to connect to the database:", err);

});

module.exports = sequelize;

1. **Integrate db.js in server.js:** Modify server.js to include the database connection:

javascript

Copy code

const sequelize = require("./config/db");

// Sync models with database

sequelize.sync({ force: false })

.then(() => {

console.log("Database synced successfully.");

})

.catch((err) => {

console.error("Error syncing database:", err);

});

**Step 3: Create Models**

Define your data structure for users and blogs using Sequelize.

1. **Create a models folder** in blog-craft-backend.
2. **Add User.js:**

javascript

Copy code

const { DataTypes } = require("sequelize");

const sequelize = require("../config/db");

const User = sequelize.define("User", {

username: {

type: DataTypes.STRING,

allowNull: false,

unique: true,

},

email: {

type: DataTypes.STRING,

allowNull: false,

unique: true,

},

password: {

type: DataTypes.STRING,

allowNull: false,

},

});

module.exports = User;

1. **Add Blog.js:**

javascript

Copy code

const { DataTypes } = require("sequelize");

const sequelize = require("../config/db");

const Blog = sequelize.define("Blog", {

title: {

type: DataTypes.STRING,

allowNull: false,

},

content: {

type: DataTypes.TEXT,

allowNull: false,

},

authorId: {

type: DataTypes.INTEGER,

allowNull: false,

},

});

module.exports = Blog;

1. **Update Associations:** Modify server.js to set up model relationships:

javascript

Copy code

const User = require("./models/User");

const Blog = require("./models/Blog");

// Define relationships

User.hasMany(Blog, { foreignKey: "authorId" });

Blog.belongsTo(User, { foreignKey: "authorId" });

// Sync models with database

sequelize.sync({ force: false })

.then(() => {

console.log("Models synced successfully.");

})

.catch((err) => {

console.error("Error syncing models:", err);

});

**Step 4: Test the Setup**

1. Restart the server:

bash

Copy code

npm run dev

1. Confirm in your MySQL client that the Users and Blogs tables were created.
2. Test the / route again to ensure the server still works.

**1. User Authorization & Role Management:**

* **Define User Roles**: Decide if you want to have different roles for users (e.g., Admin, Author, Reader). This can be useful for controlling who can write, edit, or view posts.
* **Store User Info**: You can store user roles and other profile details in your database.

**2. CRUD Operations for Blogs:**

* **Create Blog Posts**: Allow authenticated users to create new blog posts.
* **Edit Blog Posts**: Give users the ability to edit their own posts.
* **Delete Blog Posts**: Allow users to delete their posts.
* **View Posts**: Create routes for viewing all posts and individual posts.

**3. Frontend Integration:**

* Integrate the Google login into your frontend and make use of the authentication flow to display user-specific content.
* Build out pages like a dashboard, create blog page, edit post page, etc.

**4. User Profiles & Dashboard:**

* After login, users should be redirected to their personal dashboard where they can manage their posts.
* Allow users to update their profile (name, bio, etc.) through a form.

**5. Database Integration:**

* Set up Sequelize models for the **User**, **Blog Posts**, and potentially **Comments**.
* Make sure to handle database relations, such as one-to-many between users and blog posts.

**6. Security Measures:**

* Ensure that your routes are protected by authentication checks (like req.isAuthenticated()) to prevent unauthorized access to sensitive data.
* Use JWT tokens or sessions to manage user sessions securely.

**7. Enhancements:**

* Implement search functionality for blog posts.
* Add tags/categories for better organization of posts.
* Optionally, allow users to leave comments on blog posts.