

React TypeScript CRUD Application with MySQL

Complete Documentation











Table of Contents

1. Project Overview
2. Technologies & Libraries
3. Project Setup Guide
 - 3.1 Prerequisites
 - 3.2 Backend Setup
 - 3.3 Frontend Setup
4. Code Architecture & Logic
5. Issues Faced & Debugging
6. API Documentation
7. Additional Resources

1. Project Overview

This is a full-stack CRUD (Create, Read, Update, Delete) application that demonstrates user management functionality. The application consists of a React TypeScript frontend with Tailwind CSS for styling, and a Node.js Express backend with TypeScript that connects to a MySQL database.

Features

-  **Create** - Add new users with validation
-  **Read** - Display all users in a responsive table
-  **Update** - Edit existing user information
-  **Delete** - Remove users with confirmation dialog
-  **Input validation** and comprehensive error handling
-  **Responsive design** with Tailwind CSS
-  **TypeScript** throughout for type safety
-  **MySQL database** with auto table creation
-  **RESTful API** endpoints
-  **CORS enabled** for cross-origin requests

2. Technologies & Libraries

Frontend Technologies

Technology	Version	Purpose
React	18.x	Frontend framework for building user interfaces
TypeScript	5.x	Type-safe JavaScript with compile-time error checking
Tailwind CSS	3.x	Utility-first CSS framework for rapid styling
Lucide React	Latest	Beautiful, customizable SVG icons

Backend Technologies

Technology	Version	Purpose
Node.js	16+	JavaScript runtime for server-side development
Express	4.x	Web application framework for Node.js
MySQL2	3.x	MySQL client for Node.js with Promise support
CORS	2.x	Enable cross-origin resource sharing
Dotenv	16.x	Load environment variables from .env file

3. Project Setup Guide

3.1 Prerequisites

1. Install Node.js (v16 or higher)

```
# Download from: https://nodejs.org/  
# Verify installation  
node --version  
npm --version
```

Expected Output:

v18.17.0
9.6.7

2. Install MySQL (v5.7 or higher)

```
# Download from: https://dev.mysql.com/downloads/mysql/  
# Verify installation  
mysql --version
```

Expected Output:

mysql Ver 8.0.34 for Win64 on x86_64

3. Start MySQL Service

```
net start mysql80
```

Expected Output:

The MySQL80 service is starting.
The MySQL80 service was started successfully.

3.2 Backend Setup

Step 1: Create Project Structure

```
# Create main project directory
mkdir react-crud-mysql
cd react-crud-mysql

# Create backend structure
mkdir backend
cd backend
mkdir src
cd src
mkdir config controllers models routes
cd ../../
```

Step 2: MySQL Database Setup

```
-- Open MySQL Command Line or MySQL Workbench
CREATE DATABASE crud_app;
USE crud_app;
SHOW DATABASES;
```

Step 3: Initialize Backend Project

```
cd backend
npm init -y

# Install dependencies
npm install express mysql2 cors dotenv
npm install -D @types/node @types/express @types/cors typescript nodemon ts-node
```

Step 4: Create Configuration Files

package.json:

```

{
  "name": "crud-backend",
  "version": "1.0.0",
  "description": "CRUD API with Node.js, Express, and MySQL",
  "main": "dist/server.js",
  "scripts": {
    "dev": "nodemon src/server.ts",
    "build": "tsc",
    "start": "node dist/server.js"
  },
  "dependencies": {
    "express": "^4.18.2",
    "mysql2": "^3.6.5",
    "cors": "^2.8.5",
    "dotenv": "^16.3.1"
  },
  "devDependencies": {
    "@types/node": "^20.10.0",
    "@types/express": "^4.17.21",
    "@types/cors": "^2.8.17",
    "typescript": "^5.3.2",
    "nodemon": "^3.0.2",
    "ts-node": "^10.9.1"
  }
}

```

tsconfig.json:

```

{
  "compilerOptions": {
    "target": "es2020",
    "module": "commonjs",
    "outDir": "./dist",
    "rootDir": "./src",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true,
    "resolveJsonModule": true
  },
  "include": ["src/**/*.ts"],
  "exclude": ["node_modules", "dist"]
}

```

.env:

```

PORT=3001
DB_HOST=localhost
DB_USER=root
DB_PASSWORD=your_mysql_password_here
DB_NAME=crud_app

```

Step 5: Test Backend

```
cd backend
npm run dev
```

Expected Output:

Connected to MySQL database

Database tables initialized

Server running on http://localhost:3001

3.3 Frontend Setup

Step 1: Create React Application

```
# Navigate back to project root
cd ..

# Create React TypeScript app
npx create-react-app@latest frontend --template typescript
cd frontend
```

Step 2: Install Frontend Dependencies

```
# Install additional libraries
npm install lucide-react

# Install Tailwind CSS
npm install -D tailwindcss@latest postcss@latest autoprefixer@latest

# Initialize Tailwind
npx tailwindcss@latest init -p
```

Step 3: Configure Tailwind CSS

tailwind.config.js:

```
/** @type {import('tailwindcss').Config} */
module.exports = {
  content: [
    "./src/**/*..{js,jsx,ts,tsx}",
  ],
  theme: {
    extend: {},
  },
  plugins: [],
}
```

Update src/index.css:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Step 4: Start Frontend

```
npm start
```

Expected Output:

Compiled successfully!
Local: http://localhost:3000

4. Code Architecture & Logic

Backend Architecture

1. MVC Pattern Implementation

The backend follows the Model-View-Controller (MVC) architectural pattern:

- **Models** (`src/models/`): Define TypeScript interfaces and data structures
- **Views**: JSON responses (no traditional views in API)
- **Controllers** (`src/controllers/`): Handle business logic and HTTP requests
- **Routes** (`src/routes/`): Define API endpoints and route handlers

2. Database Layer

```
// Connection Management
export const connection = mysql.createConnection(dbConfig);

// Database Initialization
export const initDatabase = async () => {
  // Creates connection
  // Initializes tables if they don't exist
  // Handles connection errors
};
```

Key Features:

- **Connection Pooling:** Uses mysql2/promise for better performance
- **Auto Table Creation:** Creates users table on first run
- **Error Handling:** Comprehensive error catching and logging
- **Environment Configuration:** Uses .env variables for flexibility

3. Controller Logic

CRUD Operations Implementation:

```
// CREATE - Insert new user
export const createUser = async (req: Request, res: Response) => {
  // 1. Extract data from request body
  // 2. Validate required fields
  // 3. Validate data format (email, age)
  // 4. Insert into database
  // 5. Return created user
  // 6. Handle duplicate email error
};

// READ - Get all users
export const getAllUsers = async (req: Request, res: Response) => {
  // 1. Execute SELECT query
  // 2. Return users array
  // 3. Handle database errors
};
```

Validation Logic:

- **Required Fields:** Checks for name, email, age
- **Email Format:** Uses regex pattern validation
- **Age Range:** Validates age between 1-150
- **Duplicate Prevention:** Handles unique email constraint

Frontend Architecture

1. React Component Structure

Main App Component:

- **State Management:** Uses React hooks (useState, useEffect)
- **CRUD Operations:** Handles all user operations
- **Form Handling:** Manages form state and validation
- **Error Handling:** Displays user-friendly error messages

2. State Management

```
// Application State
const [users, setUsers] = useState<User[]>([]);           // User list
const [isFormVisible, setIsFormVisible] = useState(false); // Form visibility
const [editingUser, setEditingUser] = useState<User | null>(null); // Edit mode
const [formData, setFormData] = useState<UserForm>({...}); // Form inputs
const [loading, setLoading] = useState(false);           // Loading state
const [error, setError] = useState<string>('');          // Error messages
```

3. API Communication Logic

```
// HTTP Request Pattern
const fetchUsers = async () => {
  setLoading(true);
  try {
    const response = await fetch(`${API_BASE_URL}/users`);
    if (!response.ok) throw new Error('Request failed');
    const data = await response.json();
    setUsers(data);
  } catch (err) {
    setError('User-friendly error message');
  } finally {
    setLoading(false);
  }
};
```

HTTP Methods Used:

- **GET:** Fetch users list
- **POST:** Create new user
- **PUT:** Update existing user
- **DELETE:** Remove user

5. Issues Faced & Debugging Techniques

Issue 1: TypeScript Compilation Errors

Problem:

TSError: × Unable to compile TypeScript:

src/routes/userRoutes.ts:7:3 - error TS2305: Module "'../controllers/userController'" has no exported member 'deleteUser'.

Root Cause:

- Missing export statements in controller functions
- Incorrect import/export syntax
- File corruption during copy-paste

Debugging Steps:

```
# 1. Verify exports in userController.ts
cd backend\src\controllers
findstr "export" userController.ts

# 2. TypeScript compilation test
npx tsc --noEmit
```

Solution:

- Ensure all functions have **export** keyword
- Recreate corrupted files
- Verify file paths and naming

Issue 2: Environment Variables Not Loading

Problem:

Database connection failed: Error: Access denied for user 'root'@'localhost' (using password: NO)

Root Cause:

- **.env** file loaded after database module import
- Incorrect .env file location
- Environment variable syntax errors

Debugging Steps:

```
// Add to server.ts
console.log('Environment Variables Debug:');
console.log('DB_HOST:', process.env.DB_HOST);
console.log('DB_USER:', process.env.DB_USER);
console.log('DB_PASSWORD:', process.env.DB_PASSWORD ? '***' : 'UNDEFINED');
```

Solution:

```
// MUST be first imports
import dotenv from 'dotenv';
dotenv.config();

// Then other imports
import express from 'express';
import { initDatabase } from './config/database';
```

Issue 3: Port Configuration Conflicts

Problem:

Server running on `http://localhost:3306`

Failed to fetch users. Make sure the backend is running.

Root Cause:

- Backend running on MySQL port (3306) instead of API port (3001)
- Frontend trying to connect to wrong port
- Port environment variable misconfiguration

Debugging Steps:

```
# Check which ports are in use
netstat -ano | findstr :3001
netstat -ano | findstr :3306

# Check MySQL port
netstat -ano | findstr :3306
```

Solution:

```
# Correct .env Configuration
PORT=3001           # Node.js API server
DB_HOST=localhost  # MySQL runs on 3306 (default)
```

Port Architecture Understanding:

Frontend (React) Backend (Node.js) Database (MySQL) Port 3000 → Port 3001 → Port 3306

General Debugging Techniques

1. Logging Strategies

Backend Logging:

```
// Request logging
app.use((req, res, next) => {
  console.log(`${new Date().toISOString()} - ${req.method} ${req.path}`);
  next();
});

// Error logging
catch (error) {
  console.error('Detailed error:', {
    message: error.message,
    stack: error.stack,
    code: error.code
  });
}
```

Frontend Logging:

```
// Network debugging
console.log('API Request:', {
  url: `${API_BASE_URL}/users`,
  method: 'POST',
  body: userData
});

// State debugging
console.log('Current state:', { users, loading, error });
```

2. Network Debugging

Browser Developer Tools:

- **F12 → Network Tab:** Monitor HTTP requests
- **F12 → Console Tab:** Check JavaScript errors
- **F12 → Application Tab:** Check localStorage/cookies

Command Line Tools:

```
# Test API endpoints
curl -X GET http://localhost:3001/api/health
curl -X GET http://localhost:3001/api/users
curl -X POST http://localhost:3001/api/users -H "Content-Type: application/json" -d '{"name":"John","email":"john"
```

6. API Documentation

Base URL

`http://localhost:3001/api`

Endpoints

1. Health Check

GET `/health`

Description: Check if the API server is running.

Response:

```
{
  "status": "OK",
  "message": "Server is running"
}
```

Status Codes:

- **200 OK** : User updated successfully
- **400 Bad Request** : Invalid input data or no fields to update
- **404 Not Found** : User doesn't exist
- **409 Conflict** : Email already exists (if email updated)
- **500 Internal Server Error** : Database error

5. Delete User

DELETE `/users/:id`

Description: Delete a user by their ID.

Parameters:

- **id** (path parameter): User ID (integer)

Response:

```
{
  "message": "User deleted successfully"
}
```

Status Codes:

- **200 OK** : User deleted successfully
- **404 Not Found** : User doesn't exist
- **500 Internal Server Error** : Database error

Example API Usage

Using curl:

```
# Get all users
curl -X GET http://localhost:3001/api/users

# Get specific user
curl -X GET http://localhost:3001/api/users/1

# Create new user
curl -X POST http://localhost:3001/api/users \
  -H "Content-Type: application/json" \
  -d '{"name":"John Doe","email":"john@example.com","age":25}'

# Update user
curl -X PUT http://localhost:3001/api/users/1 \
  -H "Content-Type: application/json" \
  -d '{"name":"John Smith","age":26}'

# Delete user
curl -X DELETE http://localhost:3001/api/users/1
```

Using JavaScript fetch:

```
// Get all users
const users = await fetch('http://localhost:3001/api/users')
  .then(res => res.json());

// Create user
const newUser = await fetch('http://localhost:3001/api/users', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({
    name: 'John Doe',
    email: 'john@example.com',
    age: 25
  })
}).then(res => res.json());
```

7. Additional Resources

Production Deployment Considerations

1. Environment Configuration

```
# Production .env example
NODE_ENV=production
PORT=8000
DB_HOST=your-mysql-host.com
DB_USER=your-db-user
DB_PASSWORD=your-secure-password
DB_NAME=your-production-db
```

2. Security Enhancements

- Add authentication middleware
- Implement rate limiting
- Use HTTPS in production
- Sanitize input data
- Add request validation
- Implement proper error handling

3. Performance Optimizations

- Use connection pooling for MySQL
- Add caching layer (Redis)
- Implement pagination for large datasets
- Add database indexes
- Optimize SQL queries

4. Build Commands

Backend Build:

```
cd backend
npm run build
npm start
```

Frontend Build:

```
cd frontend
npm run build
# Serve static files with nginx or similar
```

Extending the Application

1. Additional Features

- User authentication and authorization
- User profile pictures
- Pagination and sorting
- Search and filtering
- Data export/import
- Audit logs

2. Database Improvements

- Add user roles table
- Implement soft deletes
- Add database migrations
- Create backup strategies

3. Frontend Enhancements

- Add routing (React Router)
- Implement global state management (Redux/Context)
- Add form validation library (Formik/React Hook Form)
- Implement testing (Jest/React Testing Library)

Learning Resources

1. Documentation

- [React TypeScript Documentation](#)
- [Express.js Documentation](#)
- [MySQL Documentation](#)
- [Tailwind CSS Documentation](#)

2. Best Practices

- [Node.js Best Practices](#)
- [React Best Practices](#)
- [TypeScript Handbook](#)

Final Project Structure

```
react-crud-mysql/ ├── README.md ├── .gitignore ├── backend/ | ├── package.json | ├── tsconfig.json  
| ├── .env | ├── nodemon.json | ├── dist/ (generated after build) | └── src/ | ├── server.ts | ├──  
config/ | | ├── database.ts | ├── models/ | | ├── User.ts | ├── controllers/ | | ├──  
userController.ts | ├── routes/ | ├── userRoutes.ts └── frontend/ ├── package.json ├──
```



```
tsconfig.json |─ tailwind.config.js |─ postcss.config.js |─ public/ | |─ index.html | |─  
favicon.ico | |─ manifest.json |─ src/ | |─ index.tsx | |─ index.css | |─ App.tsx | |─  
App.css |─ build/ (generated after build)
```

Database Schema

Users Table Structure:

```
CREATE TABLE users (  
  id INT AUTO_INCREMENT PRIMARY KEY,  
  name VARCHAR(255) NOT NULL,  
  email VARCHAR(255) UNIQUE NOT NULL,  
  age INT NOT NULL,  
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,  
  updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP  
);
```

Key Constraints:

- **Primary Key:** Auto-incrementing ID
- **Unique Constraint:** Email must be unique
- **Not Null:** Name, email, age are required
- **Timestamps:** Automatic creation and update tracking

Summary

This comprehensive documentation covers all aspects of the React TypeScript CRUD application, from setup to deployment considerations. The application demonstrates modern full-stack development practices with type safety, proper error handling, and clean architecture patterns.

Key Achievements:

- ☒ Complete full-stack application with modern technologies
- ☒ TypeScript implementation throughout for type safety
- ☒ Comprehensive error handling and validation
- ☒ RESTful API design with proper HTTP status codes
- ☒ Responsive and accessible user interface
- ☒ Production-ready code structure and practices

Running the Complete Application

Terminal 1 (Backend):

```
cd react-crud-mysql/backend  
npm run dev
```

Terminal 2 (Frontend):

```
cd react-crud-mysql/frontend
npm start
```

Access Points:

- **Frontend React App:** <http://localhost:3000>
- **Backend API:** <http://localhost:3001/api>
- **Health Check:** <http://localhost:3001/api/health>

React TypeScript CRUD Application with MySQL

Complete Documentation - Generated on 9/1/2025

Author: Claude AI Assistant | Project: Full-Stack CRUD Application

: Server is healthy

2. Get All Users

GET /users

Description: Retrieve a list of all users.

Response:

```
[
  {
    "id": 1,
    "name": "John Doe",
    "email": "john@example.com",
    "age": 25,
    "created_at": "2024-01-15T10:30:00.000Z",
    "updated_at": "2024-01-15T10:30:00.000Z"
  }
]
```

Status Codes:

- 200 OK : Successfully retrieved users
- 500 Internal Server Error : Database error

3. Create User

POST /users

Description: Create a new user.

Request Body:

```
{
  "name": "John Doe",
  "email": "john@example.com",
  "age": 25
}
```

Response:

```
{
  "id": 1,
  "name": "John Doe",
  "email": "john@example.com",
  "age": 25,
  "created_at": "2024-01-15T10:30:00.000Z",
  "updated_at": "2024-01-15T10:30:00.000Z"
}
```

Status Codes:

- **201 Created** : User created successfully
- **400 Bad Request** : Invalid input data
- **409 Conflict** : Email already exists
- **500 Internal Server Error** : Database error

Validation Rules:

- **name** : Required, string, non-empty
- **email** : Required, string, valid email format, unique
- **age** : Required, integer, between 1 and 150

4. Update User

PUT `/users/:id`

Description: Update an existing user's information.

Parameters:

- **id** (path parameter): User ID (integer)

Request Body:

```
{
  "name": "John Smith",
  "email": "johnsmith@example.com",
  "age": 26
}
```

Note: All fields are optional. Only provided fields will be updated.

Status Codes:

- 200 OK