**Qwipo Assignment Reference Document**

**1. Introduction**

The goal of this project is to build a full-stack web application for managing customer and address information. This document will guide you through the process, from setting up your development environment to implementing the required features.

The application will have two main entities:

* **Customer:** Represents a customer with personal details.
* **Multiple Addresses:** A customer can have one or more addresses associated with their profile.

You will build a backend API to handle the business logic and a frontend web interface for users to interact with the data.

**2. Tech Stack Overview**

This project uses a modern JavaScript-based tech stack.

* **Backend: Node.js with Express.js**
  + **Node.js** is a JavaScript runtime that lets you run JavaScript on the server.
  + **Express.js** is a framework for Node.js that simplifies building web applications and APIs. It helps you define routes (URLs) for your application (e.g., /api/customers) and handle requests from the frontend.
* **Frontend: React JS with React Router**
  + **React JS** is a popular library for building user interfaces (UIs). It uses a component-based architecture, meaning you build your UI by creating small, reusable pieces of code.
  + **React Router** is a library for handling navigation in a React application. It allows you to create different "pages" (e.g., a customer list page and a customer details page) and navigate between them without reloading the entire web page.
* **Database: SQLite**
  + **SQLite** is a lightweight, file-based database. It's easy to set up because it doesn't require a separate server process. Your entire database will be stored in a single file within your project, making it perfect for development and smaller applications.

**3. Project Setup**

A common practice for full-stack applications is to have separate folders for the client (frontend) and server (backend).

**Recommended Folder Structure**

customer-management-app/

├── client/ # React JS frontend code

└── server/ # Node.js/Express backend code

**Backend Setup (in the server/ folder)**

1. **Initialize Node.js Project:** Open your terminal in the server/ directory and run npm init -y. This creates a package.json file.
2. **Install Dependencies:** Install Express, SQLite, and cors (for allowing the frontend to communicate with the backend).
3. npm install express sqlite3 cors
4. **Create an index.js file:** This will be the entry point for your server.

**Frontend Setup (in the client/ folder)**

1. **Create React App:** In a separate terminal window, navigate to the root directory (customer-management-app/) and run:
2. npx create-react-app client
3. **Install Dependencies:** After the setup is complete, navigate into the client folder and install axios (for making API calls) and react-router-dom.
4. cd client
5. npm install axios react-router-dom

**4. Database Design (SQLite)**

You need two tables: one for customers and one for their addresses.

**customers Table Schema**

| **Column** | **Type** | **Constraints** | **Description** |
| --- | --- | --- | --- |
| id | INTEGER | PRIMARY KEY AUTOINCREMENT | Unique identifier |
| first\_name | TEXT | NOT NULL | Customer's first name |
| last\_name | TEXT | NOT NULL | Customer's last name |
| phone\_number | TEXT | NOT NULL UNIQUE | Customer's phone number |

**addresses Table Schema**

| **Column** | **Type** | **Constraints** | **Description** |
| --- | --- | --- | --- |
| id | INTEGER | PRIMARY KEY AUTOINCREMENT | Unique identifier |
| customer\_id | INTEGER | FOREIGN KEY (customers) | Links to the customer |
| address\_details | TEXT | NOT NULL | Street, building, etc. |
| city | TEXT | NOT NULL | City name |
| state | TEXT | NOT NULL | State name |
| pin\_code | TEXT | NOT NULL | Postal/ZIP code |

This structure creates a **one-to-many relationship**: one customer can have many addresses.

**5. Backend API Development (Node.js & Express)**

In your server/index.js, you'll set up your Express server and define the API endpoints.

**API Endpoints (Routes)**

Here are the routes you'll need to create. All routes should be prefixed with /api.

**Customer Routes**

* POST /api/customers: Create a new customer.
* GET /api/customers: Get a list of all customers (should support searching, sorting, and pagination).
* GET /api/customers/:id: Get details for a single customer.
* PUT /api/customers/:id: Update a customer's information.
* DELETE /api/customers/:id: Delete a customer.

**Address Routes**

* POST /api/customers/:id/addresses: Add a new address for a specific customer.
* GET /api/customers/:id/addresses: Get all addresses for a specific customer.
* PUT /api/addresses/:addressId: Update a specific address.
* DELETE /api/addresses/:addressId: Delete a specific address.

**Example: Creating the "Get All Customers" Endpoint**

// server/index.js

const express = require('express');

const sqlite3 = require('sqlite3').verbose();

const cors = require('cors');

const app = express();

app.use(cors());

app.use(express.json()); // To parse JSON request bodies

// Connect to SQLite database

const db = new sqlite3.Database('./database.db', (err) => {

if (err) {

console.error(err.message);

}

console.log('Connected to the SQLite database.');

});

// GET all customers

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

const sql = "SELECT \* FROM customers";

db.all(sql, [], (err, rows) => {

if (err) {

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

return;

}

res.json({

"message": "success",

"data": rows

});

});

});

const PORT = 5000;

app.listen(PORT, () => {

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

});

**6. Frontend UI Development (React)**

Break down the UI into reusable components.

**Component Structure**

* **App.js**: Set up your routes using react-router-dom.
* **Pages:**
  + CustomerListPage.js: Displays a list of all customers. Includes search and filter controls.
  + CustomerDetailPage.js: Shows the details of a single customer and their addresses.
  + CustomerFormPage.js: A page containing a form to either create a new customer or edit an existing one.
* **Components:**
  + CustomerList.js: The component that fetches and renders the table/list of customers.
  + CustomerForm.js: The actual form for customer data.
  + AddressList.js: Renders the list of addresses for a customer.
  + AddressForm.js: A form for adding or editing an address.

**Example: Fetching and Displaying Customers**

// client/src/pages/CustomerListPage.js

import React, { useState, useEffect } from 'react';

import axios from 'axios';

function CustomerListPage() {

const [customers, setCustomers] = useState([]);

useEffect(() => {

// Fetch customers from the backend API

axios.get('<http://localhost:5000/api/customers>')

.then(response => {

setCustomers(response.data.data);

})

.catch(error => {

console.error('There was an error fetching the customers!', error);

});

}, []); // Empty array means this effect runs once on component mount

return (

<div>

<h1>Customer List</h1>

<ul>

{customers.map(customer => (

<li key={customer.id}>

{customer.first\_name} {customer.last\_name}

</li>

))}

</ul>

</div>

);

}

export default CustomerListPage;

**7. Key Assignment Features Explained**

* **Validation:**
  + **Client-Side (React):** Use state to check if form fields are empty or invalid before the user clicks submit. This provides instant feedback.
  + **Server-Side (Node.js):** Always re-validate data on the backend before inserting it into the database. This is a crucial security step.
* **Search and Filtering (e.g., by City):**
  + The frontend will send search parameters in the URL (e.g., /api/customers?city=Mumbai).
  + The backend will read these parameters (req.query.city) and modify the SQL query to include a WHERE clause.
* **Pagination:**
  + The frontend sends the desired page number to the backend (e.g., /api/customers?page=2).
  + The backend uses LIMIT and OFFSET in the SQL query to fetch only the data for that page.
* **Page Navigation (React Router):**
  + Use the <Link> component from react-router-dom to navigate between pages without a full page reload.
  + Use dynamic routes like /customers/:id to create pages for specific customers.

**8. Final Steps**

1. **Testing:** Manually test all CRUD operations. Ensure your forms handle user input correctly and that data is saved and displayed as expected.
2. **GitHub:** Create a GitHub repository and commit your code regularly. This shows your development process.
3. **Video Recording:** Use a screen recorder (like Loom or OBS Studio) to create a short video demonstrating all the features of your completed application.

Good luck! This project is an excellent opportunity to build a complete, real-world application and showcase your skills.