**Lab07: App Development with Oracle Database**

# **Objective**

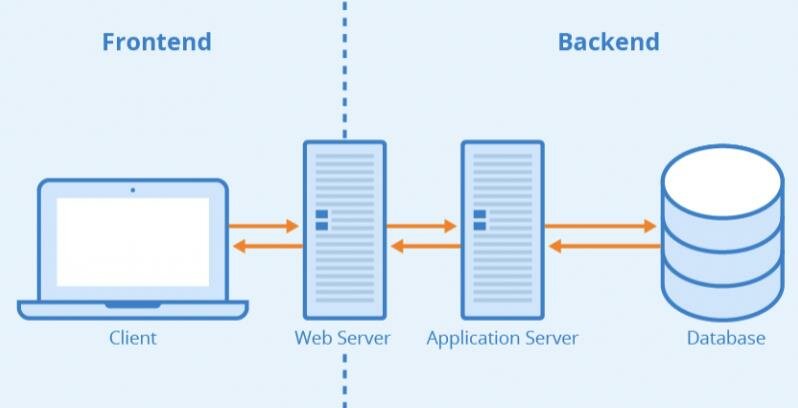
This lab is designed to provide students with hands-on experience of the web (app) development process, focusing on the creation of a compact Online Transaction Processing (OLTP) system that establishes seamless communication with an Oracle Database.

# **Acknowledgements**

This lab has been developed through a collaborative effort with our course teaching assistants, Ashnah Khalid Khan and Muhammad Ali Haider.

# **Introduction to Web Development**

In this activity, we will be connecting to an Oracle database using a NodeJS backend server with a frontend built on ReactJS. NodeJS provides us with packages and libraries that allow us to easily connect with Oracle databases and make queries. ReactJS provides us with a means to create dynamic, responsive webpages for our application, to allow the client to request services from our application. Both of these are commonly used together to create fully functional web apps, and we will be utilizing them to create one as well.



# **Business Scenario**

Consider a small dessert shop which sells a variety of desserts such as scrumptious cupcakes, brownies and more. The shop maintains a dessert management system to store, retrieve and manage the data about their products and sales. The system maintains records for desserts available, persons that order them, and orders places.   
Desserts are described with attributes like name, protein, carbs, fat, and unit price. Persons are registered with their details such as name, mobile number, email, and address. Orders are recorded with details like order date, quantity, and total price, as well as what dessert was ordered and who ordered them. Multiple persons can place multiple orders for multiple desserts.

# Dessert Shop Business Rules:

Entity relationship rules:

* Each person can place multiple orders for multiple dessert items.

Field specific rules:

* Each new dessert/person/order is identified by an ID which must be auto-incremented.
* Each dessert’s nutritional values must always be entered and not null.
* Total is determined as a derived attribute, calculated automatically using unit prices of ordered desserts and their quantity.
* In each order, you will be able to order exactly one type of desert.
* To order multiple deserts, a separate order for each desert has to be placed.
* If a unit price for a desert is updated, all previous orders’ total prices for that desert must be updated.
* If a person’s record is deleted, all orders placed by that person must also be deleted.
* If a dessert is deleted, all orders associated with that dessert must also be deleted.

Please note: This scenario is developed to give you an understanding of web development hence, the entities are kept smaller in number. In actuality, many entities may be required to form a complete database system ensuring all your information is properly modelled. Notice that the total bill of the order is not stored in this schema, in the actual scenario you would break the Orders table into Orders and OrderDetails to keep the complete information of each product and its quantity and complete order information which will .

# **Relational Schema**



# Section A: Back-end development – Connecting with the database.

**Pre-requisites: Downloads & Installations**

1. First, we will set up our backend for the application. Download NodeJS from this link: [**https://nodejs.org/en**](https://nodejs.org/en).
2. Create a folder named **backend** and open it in VS Code or your preferred IDE.
3. Navigating to the terminal, enter the following command to initialize your Node project:

**npm init -y**

1. If the previous command had successfully run, you will see a package.json file in your folder.
2. We will now install the required packages for our application. Enter the following command to install the express, body-parser, cors and oracledb packages:

**npm install express body-parser cors oracledb**

1. If the previous command had successfully run, you will see a bunch of files in a **node\_modules** folder in your folder.

**Creating A Connection To The Database**

1. Create a file named **connection.js** in your backend folder.
2. To use any package (or library) in our Node application, we use the keyword **require** and assign that to some constant value in this format:

**const nameOfConstant = require(‘nameOfPackageOrLibrary’);**

1. To establish a connection, we require the oracledb package in this file. Write the following line of code in your connection.js file:  
   **const oracledb = require(‘oracledb’);**
2. We now create an asynchronous function that establishes our connection to the database. This has the following format:

**async function *nameOfFunction*() {**

**try {**

**//Code to establish connection to datbase**

**}**

**catch (error) {**

**//Code to handle any errors in establishing connection**

**}**

**}**

1. In the try block of our code, we will use the **oracledb.getConnection** method to establish our connection. This method requires our database **user**, **password** and **connectString** (i.e. our host and service name) as parameters. We also print any errors we might receive in using the getConnection method in our catch block.
2. Including all of these into our function, the format will look something like this (ignore the use of **await** and **return** keywords. Make sure you enter the correct user and password credentials according to your system). Make sure to create a new user for the application if you haven’t already.

**async function *getConnection*() {**

**try {**

**const connection = await oracledb.getConnection({**

**user: 'c##alihaider',**

**password: '123',**

**connectString: 'localhost/orcl'**

**});**

**return connection;**

**}**

**catch (error) {**

**console.error(error);**

**}**

**}**

1. In order to use this function in other files of our application, we export it by writing the following line of code:

**module.exports = { getConnection };**

1. The final version of our **connection.js** file should look something like this:

A screen shot of a computer program

Description automatically generated

**Establishing The Connection To The Database**

1. Create another file named **app.js** in your backend folder.
2. We require the **body-parser** (for parsing JSON data in HTTP requests), **cors** (for querying your backend from different base URLs) and **express** (a framework for building apps) packages in this file. Write the following line of code in your app.js file:

**const bodyParser = require('body-parser');**

**const cors = require('cors');**

**const express = require('express');**

1. We also need to assign a port that will ‘listen’ to incoming requests to our backend. In this, we are listening at port 3001 for requests to our backend:

**const port = 3001;**

1. Next, we create an instance of **express** and assign it to a variable. We then also configure it to use **body-parser** and **cors** as part of the application:

**const app = express();**

**app.use(bodyParser.json());**

**app.use(cors());**

1. To establish if we are able to listen to requests coming into our app, we create a function using the **listen** method so that it takes **port** as an argument and prints us a message if our app is listening on that port number:

**app.listen(port, () => {**

**console.log(`Example app listening at http://localhost:${port}`)**

**})**

1. The final version of our **app.js** file after these steps should look something like this:

A computer screen shot of a program code

Description automatically generated

# Section B: Setting up queries to the database

1. Create a folder named **router** in your backend folder. This will include all router files of your application – basically, the URLs on which your queries made to the database (or APIs) are called.
2. Create another folder named **controller** in your backend folder. This will include all controller files of your application – basically, this is where the actual functions (or APIs) where you are querying the database are located.
3. The first route and API we will make will create our tables in our database. In your **controller** folder, create a file called **tableCreationController.js**.
4. In **tableCreationController.js**, we will need to establish a connection to our database to be able to create tables, so we first require our **connection.js** file and **getConnection()** function as below:

**const {getConnection} = require("../connection");**

1. Since we will need to be able to use any functions (APIs) defined in this file in other parts of our application, we will export all of them like below:

**module.exports = {**

**//All functions defined here and separated by commas**

**};**

1. Each function (API) inside this has three parameters: **req** (request – tells us the request type, any values given from the user), **res** (response – sends responses back to the user, including error and success messages)and **next** (calls another function after your function – note: it may not be used sometimes). Most of our functions will follow a format similar to this one:

***nameOfFunction* : async function(req,res,next) {**

***<declare variable for connection>***

**try {**

***<establish connection>***

***<define query – give parameters>***

***<execute query>***

***<(Optional): move on to next function/API>***

**}**

**catch(error) {**

***<log error>***

**}**

**finally {**

***<close connection or print error>***

**}**

**}**

**},**

1. For example, to write an API for creating a Deserts table in your database, we write the following code:

***CreateDesertsTable*: async function(req,res,next) {**

***//declare variable for connection***

***let connection;***

**try {**

***//establish connection***

***connection = await getConnection();***

***//define query – give parameters***

***const query = `CREATE TABLE Deserts (***

***desert\_id NUMBER GENERATED BY DEFAULT ON NULL AS IDENTITY PRIMARY KEY,***

***deserts VARCHAR2(255) NOT NULL,***

***protein NUMBER(10,2) NOT NULL,***

***carbs NUMBER(10,2) NOT NULL,***

***fat NUMBER(10,2) NOT NULL,***

***unit\_price NUMBER NOT NULL***

***)`;***

***//execute query***

***await connection.execute(query);***

***//(Optional): move on to next function/API***

***next();***

**}**

**catch(error) {**

***//log error***

***console.log("Error executing SQL query:" ,error)***

***res.status(500).send('Internal Server Error');***

**}**

**finally {**

***//close connection or print error***

***if(connection){***

***try{***

***await connection.close();***

***}***

***catch(error){***

***console.log("Error closing database connection:", error);***

***}***

***}***

**}**

**},**

1. The final version of our **tableCreationController.js** file after these steps should look something like this:

A screen shot of a computer program

Description automatically generated

1. Now we will define the route for our **CreateDesertsTable** API. You can think of them as pathways or URLs that lead you to your desired API. In the **router** folder, create a **tableCreationRouter.js** file.
2. In this file, we will require an instance of **express** and an instance of **express.Router()** to manage our routes. We also require all our APIs defined in our **tableCreationController.js** file.

**const express = require("express");**

**const router = express.Router();**

**const tableCreationController = require("../controller/tableCreationController");**

1. To define our route, we use the **router** instance and choose a specific request type e.g. POST, GET, PUT, DELETE etc. Then we define the route (path or URL) at which we want this API to be accessed. And finally we define which API of that controller file we want to call or ‘hit’ at that route using that function’s name. This will follow this format:

**router.*typeForExamplePOST*(“*routeOrPath*”, *nameOfControllerFileInstance*.*nameOfAPI*);**

1. For our **CreateDesertsTable** API, we will define the following route like so:

**router.post("/", tableCreationController.CreateDesertsTable);**

1. Finally we will also export these routes using the following line of code so they can be used elsewhere in our application:

**module.exports = router;**

1. The final version of our **tableCreationRouter.js** file after these steps should look something like this:

A screen shot of a computer program

Description automatically generated

1. You will also need to modify your **app.js** file to include these routes to be able to access these APIs. In your **app.js** file, you will have to require the **tableCreationRouter.js** file:

**const tableCreationRouter = require('./router/tableCreationRouter.js');**

1. Then you will need to configure this with your **express app** instance. Suppose you want to call APIs from the **tableCreationController.js** file whenever your route begins with **‘…/TableCreation’**. Then you will configure your route like so:

**app.use('/TableCreation', tableCreationRouter);**

1. Finally, we define a root route which will be called whenever our application is running:

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

**res.json({"message": "DB3 Application WORKING"});**

**})**

1. The final version of our **app.js** file after these steps should look something like this:

A screen shot of a computer program

Description automatically generated

**Final Touches**

1. Download the **desertRouter.js, orderRouter.js, personRouter.js** and **tableCreationRouter.js** files from LMS and place them inside the **router** folder. Note: replace your current **tableCreationRouter.js** file with this one.
2. Download the **desertController.js, orderController.js, personController.js** and **tableCreationController.js** files from LMS and place them inside the **controller** folder. Note: replace your current **tableCreationController.js** file with this one.
3. Require and configure your router files in the **app.js** file using appropriate routes.
4. Enter the following command on the terminal to run your code:

**npm start**

1. We will now test each of your APIs using your frontend and explore the application.

# Section C: Front-end development – Designing the user interface

**Pre-requisites: Downloads & Installations**

1. We will now set up our frontend for the application. Navigate to the parent directory of your **backend** folder and open the terminal, and enter the following command:  
   **npm install -g create-react-app**
2. Next, set up a React project named **frontend** by entering the following command in the terminal:  
   **npx create-react-app frontend**
3. You should now have a **frontend** folder with some files created inside it. Open this folder in VS Code or your preferred IDE.
4. Open the terminal and enter the following command to download some packages we will use later:

**npm i react-router-dom**

1. Enter the following command to run the boilerplate code for your React project:

**npm start**

**Basics & Structure**

1. Navigate to the **public > index.html** file. The **index.html** is the backbone of your website carrying all your basic HTML code for the webpage.
2. Find the line of code that says ***<title>React App</title>.*** Replace ***React App*** with a suitable title for your webpage e.g. ***<title>DBMS Web App Activity</title>.***
3. Since React works with a lot of components, it is good to structure our project into sections for better readability. Navigate to the **src** folder and create four new folders named **components**, **dataController**, **Layouts** and **pages** inside it.
4. Usually, every element in React is used as a component. For example, a button may be a component, a table might be a separate component, perhaps a combination of a button and table. All these components come together to form a webpage. For readability and as best practice, all components are placed in the **components** folder, each in their own individual subfolder with a corresponding CSS file to style them.
5. Your **dataController** folder contains all details related to your APIs.
6. **Layouts** and **pages** will be used to basically structure your webpages and components.
7. Since going into detail on each individual aspect of a React project and building it from scratch is beyond the scope of this lab, we will clone an existing repository from Github.

**Setting-up Git & Cloning A Repository From Github**

1. Download Git from this link: [**https://git-scm.com/downloads**](https://git-scm.com/downloads).
2. Once it has downloaded, open the file and follow the steps as instructed.
3. Check if you’ve installed it correctly by opening the terminal and entering this command:

**git --version**

1. If you’re using VS Code, you will need to enable Git too. Open VS Code.
2. On the top left of your window, navigate to **File > Preferences > Settings** and then type **Git: Enabled.** Make sure this box is ticked.
3. Login to your GitHub account online.
4. Now we will configure Git on our device as well. Navigate to the terminal again and enter the following command replacing it with your username:

**git config --global user.name "yourusername"**

1. Now enter this command replacing it with your email:

**git config --global user.email "email@youremail.com"**

1. To verify the above configuration, enter this command:

**git config --global --list**

1. Now we will clone a repository from Github. Go to this link: [**https://github.com/Muhammad-AliHaider/DB\_Application\_FE**](https://github.com/Muhammad-AliHaider/DB_Application_FE)**.**
2. Click on **<> Code** and copy the given link in the HTTPS tab.
3. Open VS Code from the folder you want to clone your repository in and open a new window.
4. Click on the **Clone Git Repository** option under the Start heading.
5. Click on **Clone From GitHub.**
6. Paste the copied URL into the input and press Enter. You may be prompted to give GitHub credentials so don’t worry about this.
7. Open this cloned repository by clicking Open.
8. This will now give you all code files needed to set up your frontend for your application.
9. Open the terminal and enter the following command to download all the necessary packages used in the application:

**npm i**

1. To run your application, enter the following command:

**npm start**

1. You should now have a webpage open like this:

A screenshot of a computer

Description automatically generated

1. We will now test each of the buttons on your frontend and explore the application. Click on **Create Tables**. This button will call all functions on this API in your **tableCreationRouter.js** file in your Node project:

A screenshot of a computer

Description automatically generated

1. You should get a popup displaying this message:

A screenshot of a computer

Description automatically generated

1. Now click on the **Add Mock Data** button. This will add all dummy records in your **strings.js** file in your **dataController** folder in your React project.
2. You will now receive this pop-up message:

A screenshot of a computer

Description automatically generated

1. You should now be able to view data when you click on your **Desserts**, **Persons** and **Orders** tabs:

A screenshot of a computer

Description automatically generated

1. Use the **Search** button in each category to run your ***“withConditions”*** functions. Any conditions you give for searching are sent as parameters to that function. For example, when you search for a dessert in the **Dessert** tab, you are hitting this API in your **desertRouter.js** file:

A screenshot of a computer

Description automatically generated

This API calls this function in your **desertController.js** file which implements a standard SELECT query using WHERE:

A screen shot of a computer

Description automatically generated

1. Explore the same for the **Persons** and **Orders** tabs.
2. Similarly, the **Edit** buttons correspond to “**update”** functions in each controller file, and each of these functions implements a standard UPDATE query. **Add New** buttons correspond to **“add”** functions implementing INSERT queries and **Delete** buttons correspond to functions implementing DELETE queries.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

1. For example, try out the following tasks to test

* Add a dessert, modify its fats value, and then delete it.
* Add a person, modify its address, and then delete it.
* Add an order from an existing person and desert.
* View all orders of a certain dessert of your choice and observe the total price. Update the unit\_price of that desert. Again view all orders of that same desert. Observe any changes in the total price of those orders.
* Delete this desert you viewed in the previous part. Try to view all orders of that same desert again. Observe any changes in results.

1. Now click on the **Drop Tables** button. This will delete all tables in your database so you can start from scratch.
2. You should receive this pop-up message:

A screenshot of a computer

Description automatically generated

1. Congratulations, you just made a web app and integrated it to database – YAY!