Jeff Vang

5/4/2021

SEIS630 – 01

Database Management Systems & Design

Database Design: Mechanisms of Transportation

**Overview/Objectives**

This goal of this project is to design and implement a functional database that utilizes advanced SQL concepts to assist in the day-to-day functions of a travel agency booking website. In addition to the design, this project will demonstrate the capabilities of connecting to an Oracle SQL database from an application using a modern web tech stack. The following are key concepts that will be used within the project as they relate to SQL:

* Using stored procedures to increase efficiency of core functionalities.
* Using triggers to implement referential integrity actions.
* Using table constraints to enforce business capabilities.
* Connecting to SQL from an application using NodeJS.

**Designing the Database Schema**

Following the decision to move forward with this domain I needed to identify business processes and transactions that a customer of a travel agency may want to perform. These use cases made it easier for me to identify potential entities for my database schema. The next step was to obtain the bigger picture of the domain.

**Using Oracle Data Modeler: Forward Engineering**

For my first pass in building the schema I decided to use Oracle Data Modeler’s forward engineering capabilities. I designed an ER diagram from chapter 6 to show the different entities and their relationships to each other. When I generated the relational model, as well as the data definition statements, it did not appear to have all of the table constraints that I was hoping would already be created for me. It was missing foreign key constraints and unique constraints. I also noticed duplicated foreign key columns that were auto generated by the data modeler which I found to be a nuisance.

**Logical Model**

**![Diagram

Description automatically generated]()**

**Using Oracle Data Modeler: Backwards Engineering**

As a result, I decided to add those constraints to the relational model and cleaned up some of the duplicate columns. Using the data modeler’s backward engineer capabilities, I was able to get a better logical representation of my database schema. By adding the foreign key constraints, the data modeler was able to accurately portray the relationships between the entities.

**Relational Model**

**![Diagram

Description automatically generated]()**

**Starting Up a Local Oracle Database**

Rather than going the virtualization route for my database I decided to use my local machine, that runs the Windows 10 operating system, to host my database server. I found this to be much more efficient so that I could begin the implementation of my database. I created an admin user in which I could create the infrastructure of the database and run data manipulation statements. One caveat I found with the modeler generated code was that it created constraints using separate *Alter Table* statements rather than within a single *Create Table* statement.

CREATE TABLE ORDERS (

order\_id INTEGER NOT NULL,

customer\_id INTEGER NOT NULL,

invoice\_id INTEGER NULL,

departure\_id INTEGER NOT NULL,

arrival\_id INTEGER NOT NULL,

status\_id INTEGER NOT NULL,

departuredate DATE NOT NULL,

arrivaldate DATE NOT NULL

);

ALTER TABLE ORDERS ADD CONSTRAINT order\_pk PRIMARY KEY ( order\_id );

ALTER TABLE ORDERS ADD CONSTRAINT order\_orderid\_un UNIQUE ( order\_id );

**Populating the Database**

For majority of the entities in my database I found that I could dynamically generate data and insert them into the database as part of a business process. For this reason, I deferred populating my database in full to a later point in time. There were a few entities that I found needed to be prepopulated as it did not make sense for them to be populated as a business process. Two tables, *Status*, which represents the state of an *Order,* and *BookType*, which represents the mode of transportation that a customer may select.

**INSERT INTO Status (Name) VALUES ('PENDING\_PAYMENT')**

**INSERT INTO Status (Name) VALUES ('RECEIVED')**

**INSERT INTO Status (Name) VALUES ('CONFIRMED')**

**INSERT INTO Status (Name) VALUES ('PENDING')**

**INSERT INTO BookType (Name) VALUES ('Flight')**

**INSERT INTO BookType (Name) VALUES ('Car Rental')**

**INSERT INTO BookType (Name) VALUES ('Rail')**

**INSERT INTO BookType (Name) VALUES ('Shuttle')**

**Building the Application**

The purpose of building an application is to demonstrate business interactions of our travel agency This application also serves as the data manipulation point of our database implementation. I decided to use NodeJS as my tech stack because of its versatility of being both a frontend and backend application. Using an open-source library called *socket.io* to create a communication channel between the front and the back end. In order to communicate with SQL, I decided to use another open-source library called *node-oracledb*.

**User Interface**

**![Table

Description automatically generated]()**

Figure 1. Home page. Shows interactions to place travel orders and to display all orders. Data is generated dynamically on each attempt to place an order.Home page. Shows interactions to place travel orders and to display all orders. Data is generated dynamically

**Graphical user interface

Description automatically generated with medium confidence**

Figure 2. Customer page. A detailed page of customer data.

**Initializing the Oracle Client**

Before communicating with the Oracle server, I needed to initialize Oracle client that was packaged with the open-source library. It required a connection string *localhost/orcl* to tell the client where the server was located. I used the password flow protocol to authenticate with the database on my local server.

Figure 3. Code snippet that establishes a connection with the database server.

establishConnection = async () => {

try {

if(!this.connection) {

this.connection = await oracledb.getConnection(this.config);

this.connection ? console.log('Database Connected') : console.log(`Unable to connect to ${this.config.connectionString}`)

}

return true;

}

catch(e: any)

{

return false;

}

}

export const createOrder = () => {

const departureLocation = createLocation();

const arrivalLocation = createLocation();

const departureDate = new Date(2021, random.int(0, 12), random.int(0, 31)).toLocaleString()

const arrivalDate = addHours(new Date(departureDate), random.int(10, 14)).toLocaleString()

const newOrder = {

customer: createCustomer(),

departure: departureLocation,

arrival: arrivalLocation,

departureDate: departureDate,

arrivalDate: arrivalDate,

status: Status.Pending,

books: createBooks(random.int(1, 3), departureLocation, arrivalLocation, departureDate, arrivalDate)

} as Order

return newOrder;

}

Figure 4. Code snippet of randomize the generation of order data.

**Using Stored Procedures**

For the process of getting the data populated into the database I decided to utilize stored procedures from lecture 11 in favor of embedding SQL into my source. From a maintenance perspective, stored procedures allow developments to execute common actions against a database table will having minimal permissions to a table. In some cases, read and write permissions are given only to database administrators. Stored procedures ensure developers are only able to interact with tables in a predefined manner. In the following code snippets, I show examples of how to execute a stored procedure by defining *IN, OUT* and *INOUT* keywords specific to PL/SQL. These keywords define to the stored procedure which variables should be inputs and which variables should be outputted as a result. Immediately following the execution of the stored procedure, I ran into issues where Oracle tables did not accurately reflect the data manipulation of the executed stored procedure. I eventually determined that I needed to *commit* my database changes. This led me to believe that *Transactions* were being used inside the scope of *node-oracledb.*

addOrder = async (data: Order, departureId: number, arrivalId: number, customerId: number) => {

const result = await this.connection.execute(

`BEGIN

AddOrder(:Param\_CustomerId, :Param\_DepartureId, :Param\_ArrivalId, :Param\_StatusId, :Param\_DepartureDate, :Param\_ArrivalDate, :Param\_OrderId);

END;`,

{

Param\_CustomerId: customerId,

Param\_DepartureId: departureId,

Param\_ArrivalId: arrivalId,

Param\_StatusId: { type: oracledb.NUMBER, dir: oracledb.BIND\_IN, val: data.status },

Param\_DepartureDate: data.departureDate,

Param\_ArrivalDate: data.arrivalDate,

Param\_OrderId: { type: oracledb.NUMBER, dir: oracledb.BIND\_OUT }

}

)

this.connection.commit()

return (result.outBinds as any).Param\_OrderId as number;

}

**Using Triggers**

In our *OrdersAudit* table we keep track of all the different actions that a user can perform on an order. Although this can easily be done using a stored procedure or some kind of business logic from within the application, I decided to use a trigger to demonstrate this logic behind the scenes. When a change to an order takes place, changing from *Confirmed* to *Cancelled*, we log this interaction using a SQL trigger.

CREATE

OR REPLACE TRIGGER orders\_audit\_trg AFTER INSERT

OR

UPDATE

ON orders FOR EACH ROW

DECLARE StatusName VARCHAR(10);

BEGIN

SELECT

Name INTO StatusName

FROM

Status

WHERE

status.status\_id = :new.status\_id;

INSERT INTO

ordersaudit (order\_id, invoice\_id, status, occurreddate)

VALUES

(

:new.order\_id,

:new.invoice\_id,

StatusName,

SYSTIMESTAMP

)

;

END

;

**Conclusion**

I found this project to be very rewarding in learning to work with Oracle. My previous experience with SQL has primarily been with Microsoft SQL Server and making queries against existing databases. While working on this project I was able to gain hands on experience with designing a schema using both forward and backends engineering. This enabled me to really think about the relationships between relations. I got to work with stored procedures and triggers and really understand the benefits they provide application developers. From an application standpoint, I was able to get a better understanding of how to establish connections with database servers in a secure manner. This project can be accessed in GitHub via the link, <https://github.com/aliendev3096/SEIS630-FinalProject>.