Assessment One

Rebel Cycles

WEB601

By Travis Byrman

Table of Contents

Milestone One

Introduction

I plan to create an online ecommerce website for a small/midsize enterprise company (SME). My project will be designed and created based upon the needs of small independent motorcycle business (Rebel Cycles). Rebel Cycles has a small team of people (one or two) and provides motorcycles services and repairs to customers. They have a small clientele and are not a typical mainstream business. Rebel Cycles main source of income is mainly through 'word of mouth' motorcycle repairs, although they do receive contracted work from time to time.

Issue

Due the Covid-19 outbreak and the alert levels in New Zealand, Rebel Cycles has been heavily impacted, due to being deemed as a non-essential service Rebel Cycles business does not have the sufficient requirements to operate during the alert levels, which is causing them to operate at a loss. The owner has decided to look into other ways of providing their services.

After some research the owner has noticed that online shopping has become very popular. The owner has decided that he wants to change his business focus and start selling motorcycle products online. The products he wants to sell are parts, accessories, motorcycles, and any other related equipment that he could refurbish and sell for a profit.

Requirements

The owner of Rebel Cycles would like the following features on the website:

- Administrators can create and delete products, categories, and accounts.
- Administrators can view full customer records and orders.
- Customers need to create an account and login to add and purchase products.
- Customer must checkout to complete the order
- Customers can browse product categories, view products, based on search or filter.
- A shopping cart must be used to display the contents of the products in detail. Customers must also have the option to change the quantity of a product in the shopping cart.
- A count of the records found must be shown.
- Customers must be able to search for products and have the ability to navigate through web pages.

Solution

I am going to create an online ecommerce website for Rebel Cycles where the owner can sell his products online. To do this I will be using HTML/CSS/JavaScript/React and the following modules.

Node.js
Node.js is a back-end JavaScript runtime environment that runs on the V8 engine. This allows me to create backend/server that hosts my websites. I will be using Node.js to create the server side of my application.

Express

Express is a Web API framework for Java Script. This allows me to create dynamic website pages that will be hosted through Node.js. I will be using Express to process data via GET and POST methods.

EJS

EJS is a templating module used to layout webpages. I will be laying out my webpages using EJS. This module is a very easy and effective way of rendering data and laying out my content.

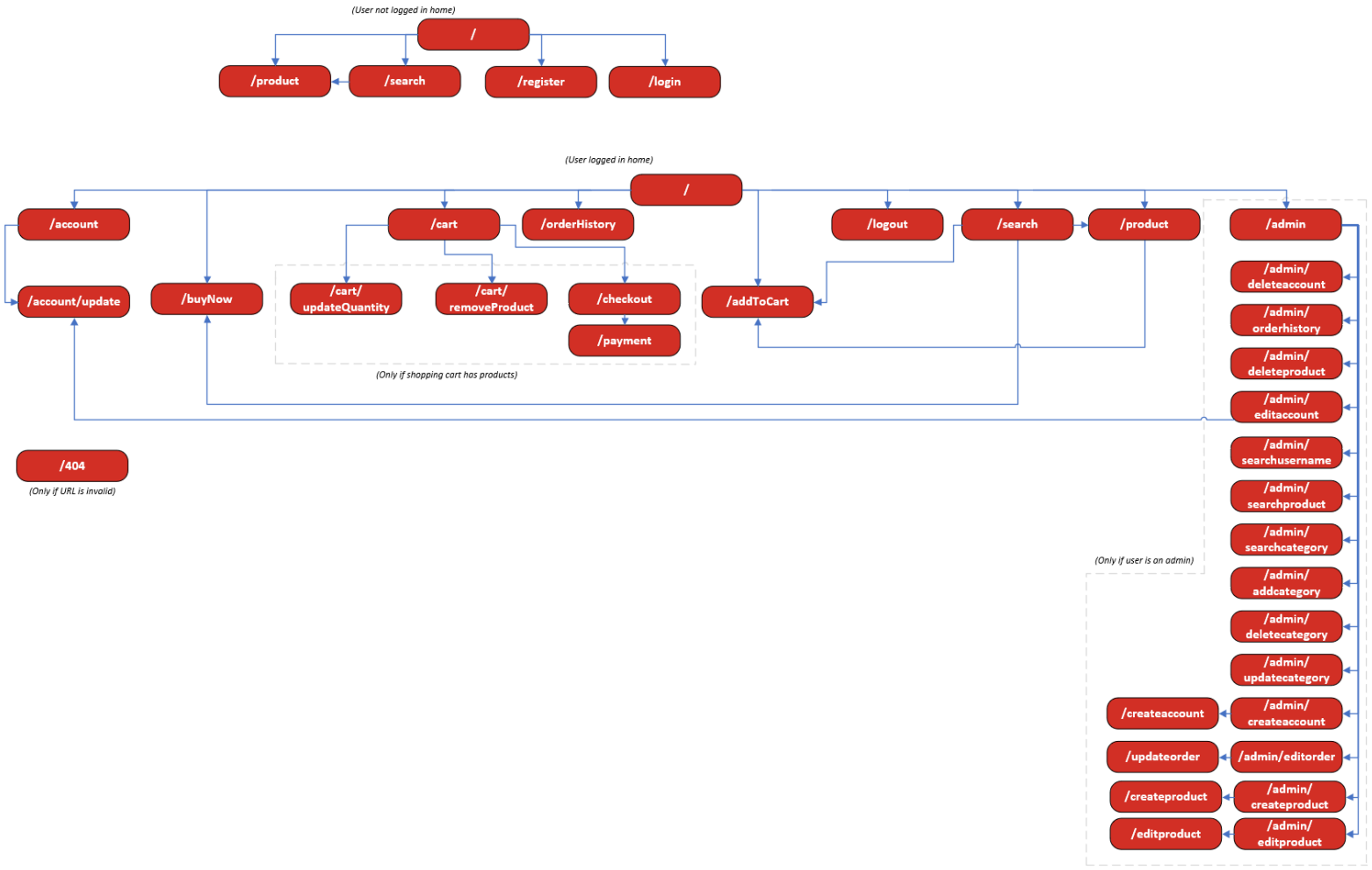
MySQL

MySQL is a database management system. This allows me store and request data from a storage location. I chosen to use MySQL in my project instead of other alternatives (like Mongodb) due its features and known reliability. I am also very familiar with MySQL which will allow me to make effective queries. I will be implementing MySQL into my project and will be using it alongside with Express and EJS.

For this project, I am using an Azure MySQL Database. This database is hosted remotely and does not require any prior configuration.

Hierarchy Diagram

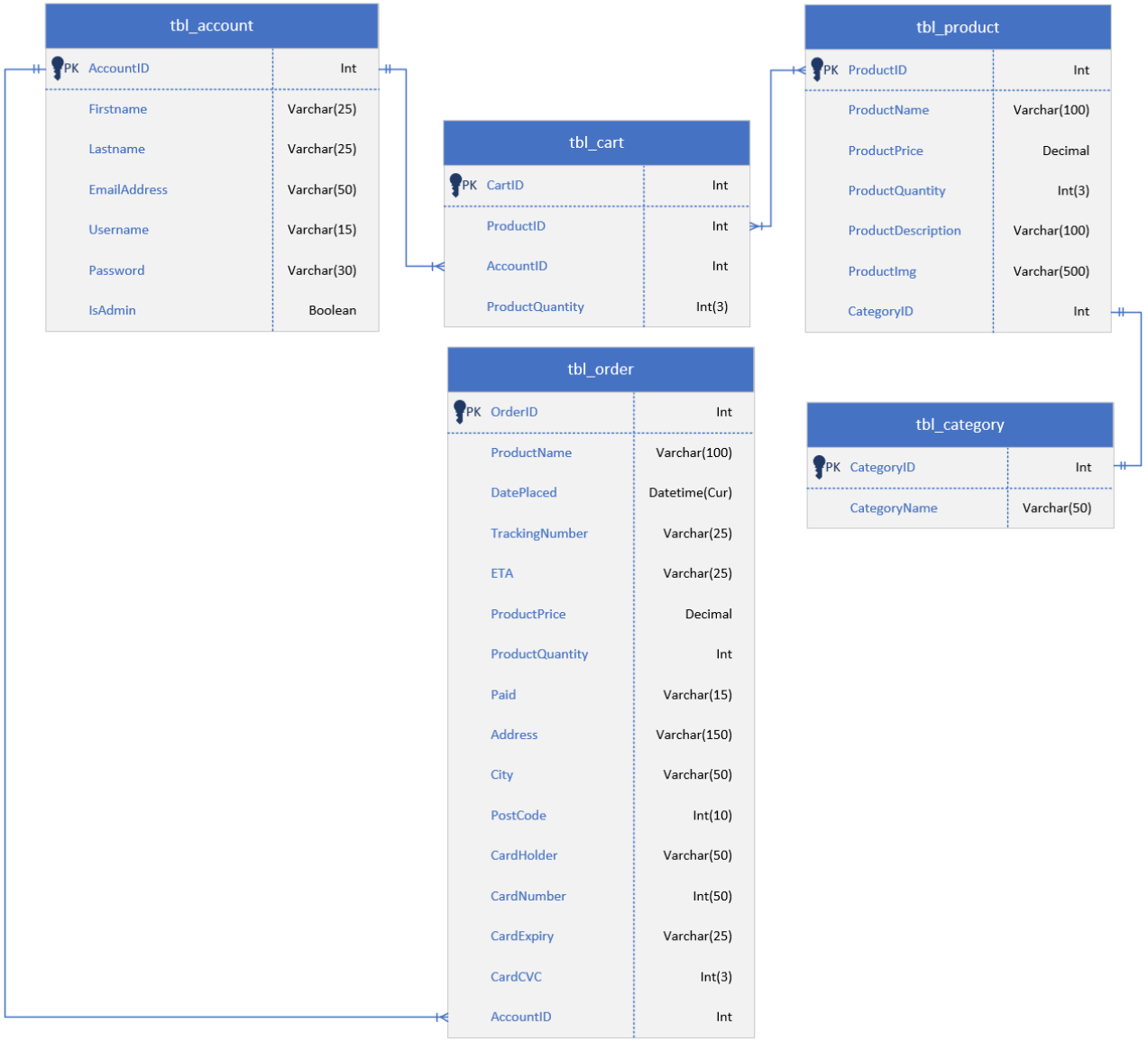
This shows the arrangement of the routes and how a user is directed.



The top part of the diagram shows the sitemap/structure of the routes for when a user is not logged in.

The bottom part of the diagram shows when a user is logged in. They will be using the same home page, but with more options/content.

ERD Diagram



Above shows my database layout for the MySQL database.

CRUD Diagram

Table:	Process:	Login	Register	Product search	Customer updates account	Admin updates user account	Admin creates product	Admin updates product	Customer views cart	Admin views all orders	Customer removes item from cart	Customer adds item to cart	Customer completes payment	View product	View all categories	Admin deletes user account	Admin deletes product	Admin deletes category	Customer updates cart quantity	Admin views all users	Admin views all categories	Admin views all products	Admin updates order	Admin updates product order	Admin updates category	Admin creates an account
	tbl_category																									
	CategoryID																	D			R				R	
	CategoryName			R										R	R			D			R				U	
	tbl_account																									
	AccountID		C		R	R										D				R						C
	Firstname		C		U	U										D				R						C
	Lastname		C		U	U										D				R						C
	Username	R	C		U	U					R					D				R						C
	EmailAddress		C		U	U										D				R						C
	Password	R	C		U	U										D				R						C
	IsAdmin	R	C			U										D				R						C
	tbl_product																									
	ProductID						C	R	R					R			D					R		R		
	ProductName			R			C	U	R					R			D					R		U		
	ProductPrice			R			C	U	R					R			D					R		U		
	ProductQuantity			R			C	U	R				U	R			D					R		U		
	ProductDescription			R			C	U	R					R			D					R		U		
	ProductImg			R			C	U	R					R			D					R		U		
	CategoryID						C	U									D					R		U		
	tbl_cart																									
	CartID								R		RD	C	D						R							
	ProductQuantity								R		D	C	D						U							
	ProductID								R		D	C	D													
	AccountID								R		D	C	D													
	tbl_order																									
	OrderID									R			C										R			
	ProductName									R			C										U			
	DatePlaced									R			C										U			
	TrackingNumber									R			C										U			
	ETA									R			C										U			
	ProductPrice									R			C										U			
	ProductQuantity									R			C										U			
	Paid									R			C										U			
	Address									R			C										U			
	City									R			C										U			
	PostCode									R			C										U			
	CardHolder									R			C										U			
	CardNumber									R			C										U			
	CardExpiry									R			C										U			
	CardCVC									R			C										U			
	AccountID									R			C													

Project accounts/access

I have created two accounts that can be used, one with administrator powers and the other has regular customer powers

Administrator

Username: admin

Password: admin

Customer

Username: user

Password: user

Please see project README for module requirements.

User Manual

Routes

Index

```
app.get('/', (req, res) => {
    getListings = [];
    connection.query(
        `CALL getIndexItems()`, // Call stored procedure
        function(err, results, fields) {
            var x = [];
            for (let i = 0; i < results[0].length; i++) {    // Create an object for each record
                var obj = { id:results[0][i].ProductID,      // Each object contains the record details
                            name:results[0][i].ProductName,  // Push object to array
                            price:parseInt(results[0][i].ProductPrice).toFixed(2),
                            quantity:(results[0][i].ProductQuantity),
                            description:results[0][i].ProductDescription,
                            image:results[0][i].ProductImg,
                            category:results[0][i].CategoryName};
                x.push(obj);
            }
            getListings.push(...x);
        });
    setTimeout(() => {
        res.render('index', {title: 'Home', admin: admin, search: search, loginMessage: loginMessage,  // Render page as html
                            username: aUsername, loginBtnText: loginBtnText, rResult: registrationResult,
                            lResult: loginResult, getListings});
    }, 3000);
});
```

This is the home page, when a user loads this route, it calls a stored procedure on the database. This procedure returns product records that match a category. Each item in the record is added to an object, when completed, that object is pushed to an array, this process repeats as many times as records. These products are then rendered on an html page.

Order history

```
app.get('/orderHistory/:username', (req, res) => {
    const { username } = req.params;
    connection.query(
        `CALL viewOrderHistory("${username}")`, // Call stored procedure
        function(err, results, fields) {
            res.send(results) // Display json object
        }
    );
});
```

This route shows the order history of a user. It gets the username from the URL via the GET method and is used as a parameter for the store procedure when called. The results are displayed as a json object, using res.send.

Add to cart

```
app.get('/addToCart/:id', (req, res) => {
    const { username } = {"username": aUsername}; // Get parameters
    const { id } = req.params;
    const { quantity } = {"quantity": "1"}; // Create an quantity object
    connection.query(
        `CALL addToCart("${username}","${id}","${quantity}")`, // Call stored procedure
        function(err, results, fields) {
            res.redirect('/'); // Redirect user to home
        });
});
```

This route gets the product id from the URL using the GET method. The username is gained locally, until I have created user sessions in Milestone 2. The quantity is always 1 as the route will be linked to a add to cart button. The procedure is called, and the user is redirected to home.

Login and logout

```
app.post('/login', (req, res) => { // Get 'post' body data
    const { password } = req.body;
    const { username } = req.body;
    connection.query(
        `CALL login("${username}","${password}")`,
        Complexity is 3 Everything is cool!
        function(err, results, fields) {
        if (results[0] == undefined) {
            loginResult = "Invalid login credentials."; // Change variable values for html page render
            registrationResult = "";
            res.redirect('/#sec5');
        } else {
            admin = results[0][0].IsAdmin;
            aUsername = results[0][0].Username;
            registrationResult = "";
            loginResult = "";
            loginBtnText = "Logout";
            loginURL = "logout/" + aUsername;
            res.redirect('/#sec5');
            loginMessage = "";
        }
    });
});

Complexity is 3 Everything is cool!
app.get('/logout/:username', (req, res) => {
    const { username } = req.params;
        if (username == aUsername) {
            aUsername = "";
            loginBtnText = "Login";
            loginURL = "#sec5";
            loginMessage = "Please create an account to purchase items.";
            res.redirect('/');
        } else {
            res.status(404).render('404', {title: '404', username: aUsername, admin: admin, loginBtnText: loginBtnText}); // If username is incorrect render 404 page
        }
});
```

The login procedure is called using the body arguments from a POST form. If the login credentials are correct the procedure will not return a statement, triggering my test front end to update variables. If the credentials are correct, the user is logged, the username is stored locally, due to not having any sessions created.

The logout route only sets the username to null, as I haven't created any sessions yet as its not needed for this milestone.

Cart

```
app.get('/cart/:username', (req, res) => {
    const { username } = req.params;
    if (username == aUsername) {
    connection.query(
        `CALL viewShoppingCart("${username}")`,
        function(err, results, fields) {
            var p = [];
            var totalPrice = 0;
            for (let i = 0; i < results[0].length; i++) {
                var obj = { name:results[0][i].ProductName,
                            id:results[0][i].ProductID,
                            total:parseInt(results[0][i].ProductTotal).toFixed(2),
                            price:parseInt(results[0][i].ProductPrice).toFixed(2),
                            totalQuantity:results[0][i].totalProductQuantity,
                            quantity:results[0][i].ProductQuantity,
                            image:results[0][i].ProductImg};
                p.push(obj)
                totalPrice = totalPrice + parseInt(results[0][i].ProductTotal);
            }
            totalPrice = totalPrice + 100;
            totalPrice = totalPrice.toFixed(2);
            getCartItems = p;
            res.render('cart', {title: 'Shopping Cart', username: aUsername, admin: admin, loginBtnText: loginBtnText, getCartItems, totalPrice: totalPrice});
        });
    } else {
        res.status(404).render('404', {title: '404', username: aUsername, admin: admin, loginBtnText: loginBtnText});
    }
});
```

Following the same method of getting items to the index page, each record is represented as an object, and is then added to an array. The array is then rendered on the index page.

Checkout

```
app.post('/checkout', (req, res) => {
    const { address } = req.body;
    const { city } = req.body;
    const { postcode } = req.body;
    const { cardholder } = req.body;
    const { cardnumber } = req.body;
    const { cvc } = req.body;
    const { expiry } = req.body;
    connection.query(
        `CALL customerCheckout("${aUsername}","${address}","${city}","${postcode}","${cardholder}","${cardnumber}","${cvc}","${expiry}")`,
        function(err, results, fields) {
            res.redirect('/');
        }
    );
});
```

This procedure gets all the arguments from a POST method form. The procedure is then called with these as parameters, and an order is created, and the cart item is removed.

Account

```
app.get('/account', (req, res) => {
    details = [];
    const username = aUsername;
    connection.query(
        `CALL adminListUsers("${username}")`,
        function(err, results, fields) {
            var x = [];
            for (let i = 0; i < results[0].length; i++) {
                var obj = { id:results[0][i].AccountID,
                            fname:results[0][i].Firstname,
                            lname:(results[0][i].Lastname),
                            email:results[0][i].EmailAddress,
                            username:results[0][i].Username,
                            password:results[0][i].Password,
                            admin:results[0][i].IsAdmin};
                x.push(obj);
            }
            details.push(...x);
        });
        setTimeout(() => {
        res.render('account', {title: 'Account', details, search: search, username: aUsername, admin: admin, loginBtnText: loginBtnText});
    }, 1500);
});
```

These procedures get the record of the account and an object is created from it, which is used to eb rendered on a webpage.

Payment

```
app.get('/payment', (req, res) => {
    var dt = new Date();
    today =  dt.toISOString().slice(0, 10)
        setTimeout(() => {
            res.render('payment', {title: 'Payment', today, search: search, username: aUsername, admin: admin, loginBtnText: loginBtnText});
        }, 1500);
});
```

The payment page is created. I have used setTimeouts for some rendering to allow enough time for functions to process.

Admin

```
app.get('/admin', (req, res) => {
    if (admin == 1) {
        setTimeout(() => {
            res.render('admin', {title: 'Administration', search: search, username: aUsername, admin: admin, loginBtnText: loginBtnText});
        }, 1500);
    } else {
        res.status(404).render('404', {title: '404', username: aUsername, admin: admin, loginBtnText: loginBtnText});
    }
});
```

If a user is not an admin, then they will not get access to this page, instead they will be redirected to a 404 page.

Search

```
app.get('/search', (req, res) => {
    getListCategories();
    getListings = [];
    const search = req.query.result;
    const category = req.query.category;
    var lookup = "%"+ search +"%"
    connection.query(
        `CALL search("${lookup}","${category}")`,
        function(err, results, fields) {
            var x = [];
            for (let i = 0; i < results[0].length; i++) {
                var obj = { id:results[0][i].ProductID,
                            name:results[0][i].ProductName,
                            price:parseInt(results[0][i].ProductPrice).toFixed(2),
                            quantity:(results[0][i].ProductQuantity),
                            description:results[0][i].ProductDescription,
                            image:results[0][i].ProductImg,
                            category:results[0][i].CategoryName};
                x.push(obj);
            }
        getListings.push(...x);
        res.render('search', {title: 'Search Results', getCategories:getCategories, username: aUsername, admin: admin, loginBtnText: loginBtnText, getListings, search: search});
    });
});
```

This route gets the parameters from the URL and calls a stored procedure that returns all records similar to the parameters.

Product

```
app.get('/product/:id', (req, res) => {
    if (admin == 1) {
    const { id } = req.params;
    connection.query(
        `CALL viewProduct("${id}")`,
        function(err, results, fields) {
            res.send(results);
        }
    );
    } else {
        res.status(404).render('404', {title: '404', username: aUsername, admin: admin, loginBtnText: loginBtnText});
    }
});
```

The route gets the parameters from the URL and calls a procedure that returns the details of the product selected.

MySQL Statements

Here are the two most complex MySQL statements in my application.

Checkout

```
DELIMITER ;;
CREATE DEFINER=`main`@`%` PROCEDURE `customerCheckout`(pUsername VARCHAR(15), pAddress VARCHAR(150), pCity VARCHAR(50), pPostCode INT(10), pCardHolder VARCHAR(50), pCardNumber INT(50), pCardCVC INT(3), pExpiry DATE)
BEGIN
    INSERT INTO `tbl_order` (`ProductName`,`ProductPrice`,`ProductQuantity`,`AccountID`, `Address`, `City`, `PostCode`, `CardHolder`, `CardNumber`, `CardCVC`, `CardExpiry`)
    SELECT tbl_product.ProductName, tbl_product.ProductPrice * tbl_cart.ProductQuantity, tbl_cart.ProductQuantity, tbl_cart.AccountID, pAddress, pCity, pPostCode, pCardHolder, pCardNumber, pCardCVC, pExpiry
    FROM ((tbl_cart
    INNER JOIN tbl_account ON tbl_cart.AccountID = tbl_account.AccountID)
    INNER JOIN tbl_product ON tbl_cart.ProductID = tbl_product.ProductID)
    WHERE tbl_cart.AccountID = (SELECT AccountID FROM tbl_account WHERE Username = pUsername);

    UPDATE tbl_product
    INNER JOIN tbl_cart ON tbl_product.ProductID = tbl_cart.ProductID
    SET tbl_product.ProductQuantity = tbl_product.ProductQuantity - tbl_cart.ProductQuantity
    WHERE tbl_cart.AccountID = (SELECT AccountID FROM tbl_account WHERE Username = pUsername);

    DELETE FROM tbl_cart WHERE AccountID = (SELECT AccountID FROM tbl_account WHERE Username = pUsername);
    SELECT 'Order has been submitted' AS MESSAGE;
END ;;
DELIMITER ;
```

When a user has completed the payment for their products and has ‘checked out’, this procedure is called. This procedure takes the parameters from the call in JavaScript and creates an order record. The order record is created using a join statement, so that the order record can contain details such as foreign keys by using the parameters as where clauses (such as username). The order record is created, the cart record is deleted and the product quantity of the products that were purchased were changed. A select statement was then called to notify the user that the order has been created.

Add product to cart

```
DELIMITER ;;
CREATE DEFINER=`main`@`%` PROCEDURE `addToCart`(pUsername VARCHAR(15), pProductID INT, pProductQuantity INT(3))
BEGIN
SET @CPQ = 0;
SET @PQ = (SELECT ProductQuantity FROM tbl_Product WHERE ProductID = pProductID);

IF EXISTS (SELECT ProductQuantity FROM tbl_cart WHERE ProductID = pProductID) THEN
    SET @CPQ = (SELECT ProductQuantity FROM tbl_cart WHERE ProductID = pProductID);
END IF;

IF ( @CPQ < @PQ) THEN
    IF (SELECT ProductQuantity FROM tbl_Product WHERE ProductID = pProductID) = 0 THEN SELECT 'Product is out of stock' AS MESSAGE;
        ELSE IF EXISTS (SELECT ProductID FROM tbl_cart WHERE ProductID = pProductID AND AccountID = (SELECT AccountID FROM tbl_account WHERE Username = pUsername)) THEN
            UPDATE tbl_cart SET ProductQuantity = ProductQuantity + 1 WHERE ProductID = pProductID AND AccountID = (SELECT AccountID FROM tbl_account WHERE Username = pUsername);
            SELECT 'Product added to cart' AS MESSAGE;
        ELSE
            INSERT INTO `tbl_cart` (`ProductQuantity`,`ProductID`,`AccountID`)
            VALUES(
                (pProductQuantity),
                (pProductID),
                (SELECT AccountID FROM tbl_account WHERE Username = pUsername));
            SELECT 'Product added to cart' AS MESSAGE;
        END IF;
    END IF;
    END IF;
END ;;
DELIMITER ;
```

When a user adds a product to their shopping cart this stored procedure is called. When called it creates two select statements that are later compared. ‘@CPQ’ is the product quantity that the user is requesting to purchase. ‘@PQ’ is the quantity of product available, from the product that is requested to be purchased. If the current product quantity is less than the product quantity, then the requested amount of product to be purchased can be bought. Otherwise, the user is notified. If the product quantity is 0, then the product is out of stock, and the user is notified. If the product already exists in a user’s cart, then the product quantity is increased. If the product doesn’t already exist in the user’s cart, then a product record is created in the cart.

References

MySQL :: MySQL 8.0 reference manual :: 13.2 data manipulation statements. (n.d.). MySQL :: Developer Zone. Retrieved September 24, 2021, from https://dev.mysql.com/doc/refman/8.0/en/sql-data-manipulation-statements.html

Milestone Two

For this milestone I create a front-end interface that utilises my back-end. For my client-side application I used the following libraries:

React – To create a static website.

React Routes – To create links between all my webpages.

Auth0-React – Login authentication.