جامعة العين
AL AIN UNIVERSITY

Internship Final Report

{Laravel.
SMART SOLUTIONS | لارافيل للحلول الذكية

Student Name	Mahmoud Husam Abdeen
Student ID	202211625
Program	Bachelor of Science in Software Engineering
Company Name	Laravel Smart Solutions
Company Website	https://laraveluae.com
Internship Start Date	1 June 2026
Internship Finish Date	7 August 2026
Working Hours per Week	40 hours
Company Supervisor	Mr. Obada Alrefai
Academic Supervisor	Dr. Musab Mustafa Hijazi

Executive Summary

This report summarizes my internship experience at **Laravel Smart Solutions** from **1 June 2026 to 7 August 2026**, with a working schedule of **40 hours per week**. Laravel Smart Solutions is an information technology and software-development company located in Al Ain, Abu Dhabi. The company develops business-management systems and web-based solutions for different organizations. Its systems help businesses manage customers, invoices, sales, payments, employees, appointments, expenses, and administrative reports. The company also provides customized software development, website design, and electronic-marketing services [1].

During the internship, I worked with an existing Laravel-based business-management application. The principal technologies and tools used were Laravel 10, PHP 8.1, MySQL, phpMyAdmin, XAMPP, Composer, Visual Studio Code, Blade, JavaScript, AJAX, Bootstrap, Tailwind CSS, and SweetAlert2. At the beginning of the internship, I prepared the local development environment, installed the required dependencies, configured the database connection, and successfully ran the application on localhost. I then studied the Laravel project structure and learned how routes, controllers, models, Blade views, migrations, and database tables work together.

A major part of the internship involved improving the Customer Management module. I practiced CRUD operations by creating, reading, updating, and deleting customer records. I added phone-number validation, birthday information, customer profile pictures, image previews, and default avatars. I also improved the customer dashboard and enhanced the Add Multiple Customers form to process several customer records in one submission.

I also participated in front-end development by improving buttons, forms, tables, cards, alerts, invoice pages, and login and registration interfaces using CSS, Bootstrap, Tailwind CSS,

JavaScript, and supporting front-end libraries. In addition, I reviewed the Sales and Payment Report pages to understand how Laravel retrieves related database records, applies filters, calculates totals, and displays the results to users.

I developed a Task Assignment page and a Task Report to understand and implement the complete flow of data from the database and back-end controller to the front-end interface. The feature allows tasks to be assigned to staff members, stored with a Pending status, marked as completed, filtered by date or status, and displayed with summary totals.

I also examined AJAX and API-related communication through customer operations, invoice submission, and WhatsApp invoice sharing. I worked on generating secure public invoice links that customers could open without accessing the administrative dashboard. I also implemented and tested the Tap Payments workflow and prepared the required structures for the future activation of Tabby and Tamara.

These activities produced practical improvements to the application's customer-management, invoice, task-management, reporting, and online-payment functions. The completed work was tested through the user interface, Laravel back end, and MySQL database to confirm that records were processed and displayed correctly.

Overall, the internship strengthened my technical skills in Laravel, PHP, MySQL, front-end development, AJAX, APIs, payment integration, testing, debugging, and technical documentation. It also improved my problem-solving, time-management, communication, patience, and attention-to-detail skills. Most importantly, the experience allowed me to connect the theoretical knowledge gained at Al Ain University with practical software-development responsibilities in a professional working environment.

Table of Contents

List of tables

List of figures

List of Code Snippets

1. Introduction

Internship training plays a key role in the life of a university student as it helps to transition from academia to actual work practice. Theories, concepts, and technical skills are taught in the classroom, but the internship provides students with the opportunity to put these to the test in a real company setting. It also introduces the students to various aspects of how everyday tasks are achieved, how professional teams operate and how real problems are addressed in real projects.

My internship was with a company called Laravel Smart Solutions, which is a software development and information technology company, based in Al Ain, Abu Dhabi. The company's business-management systems and web-based solutions include customer management, invoicing, sales, payments, appointments, employees, expenses and admin reports. It also offers tailor-made software development, website design and e-marketing services [1].

For the internship, I had the opportunity to collaborate with a pre-built business-management application built with Laravel 10. Laravel structures web applications into related parts, like routes, controllers, models, Blade views, middleware, migrations, and database operations [2]. Initial tasks that I had to do were getting the local development environment set up, installing the project dependencies, configuring the MySQL connection, troubleshooting setup problems and getting the application working on local host.

Learning the processes of working with an established project was also crucial to the internship. In the real world, the developer doesn't always embark on a project from scratch. At times, the developer must understand the old codes, locate the problem and make changes while not impacting other segments of the system. This was helpful for me to get better at reading code, the

structure of the project and understanding how a small change in the database structure could impact the model, controller and page.

The objective of this report is to record the activities that were accomplished in the internship, discuss the technical knowledge and professional skills acquired and assess the link between the University curriculum and the workplace learning. The report also includes an overview of the company, technologies used, the major tasks carried out, the problems faced, the solutions implemented and the results obtained during the training.

This report starts with an overview on Laravel Smart Solutions, its background, industry, services, contact, and organizational structure. It then introduces the working environment of the internship, the working environment of the department, the technology used and the main technical tasks completed. The latter sections are the ones that explain how Al Ain University prepared me for the internship, evaluate the skills and experiences gained, and see the difference between academic theory and practice, as well as the overall conclusion that I drew from the internship.

2. Description of the Company

2.1 Company Name and Overview

The company is known as Laravel Smart Solutions and is called out as Laravel SS on the website. The company is based in Al Ain, Abu Dhabi, UAE and offers software solutions to businesses. It automates business operations for business owners like customer management, sales, invoices, appointments, payments, and reports.

Laravel Smart Solutions is a good internship company for someone majoring in software engineering because they are related to web development, database, system testing, and business software solutions. The internship takes place in a real system that the company uses for their business, which allows practical knowledge of client needs, enhancement of system capabilities, resolution of failures and tests of Web application usage before clients use them.

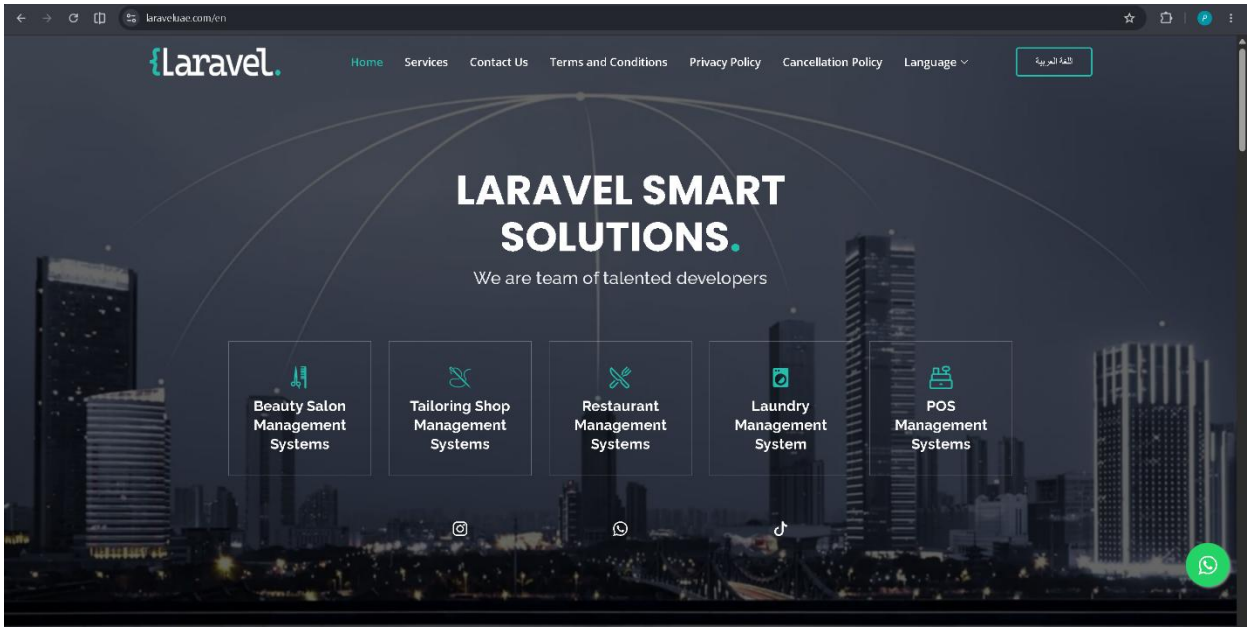

Figure 1 Laravel Smart Solutions Website Homepage

2.2 Brief History and Background

Laravel Smart Solutions has established a strong reputation in the UAE market and has gained valuable experience in the business environment in the UAE and the Arab world. This experience has allowed the company to gain a good grasp of the local market needs and thus this business is able to offer software solutions which are tailored to fit businesses in its local market.

Company Aspect	Details
Market Experience	Dubai, Abu Dhabi, and the rest of the UAE, Gulf, and Arab markets.
Main Strength	Experience in business software solutions
Customer Trust	Over 100 customers trust us.

Table 1 Company Background Summary

2.3 Address and Online Information

Laravel Smart Solutions is in Khalifa Bin Zayed St, Al-Ain, Abu Dhabi, UAE. The company is reachable over the phone, email, WhatsApp, and Instagram.

Type	Details
Address	Khalifa Bin Zayed St, Al-Ain, Abu Dhabi
Email	info@laraveluae.com
Website	laraveluae.com
Mobile	+971508922137 / +971544846201
Landline	+97137373033 ext. 818
Online Channels	Website, WhatsApp, Instagram

Table 2 Company Basic Information

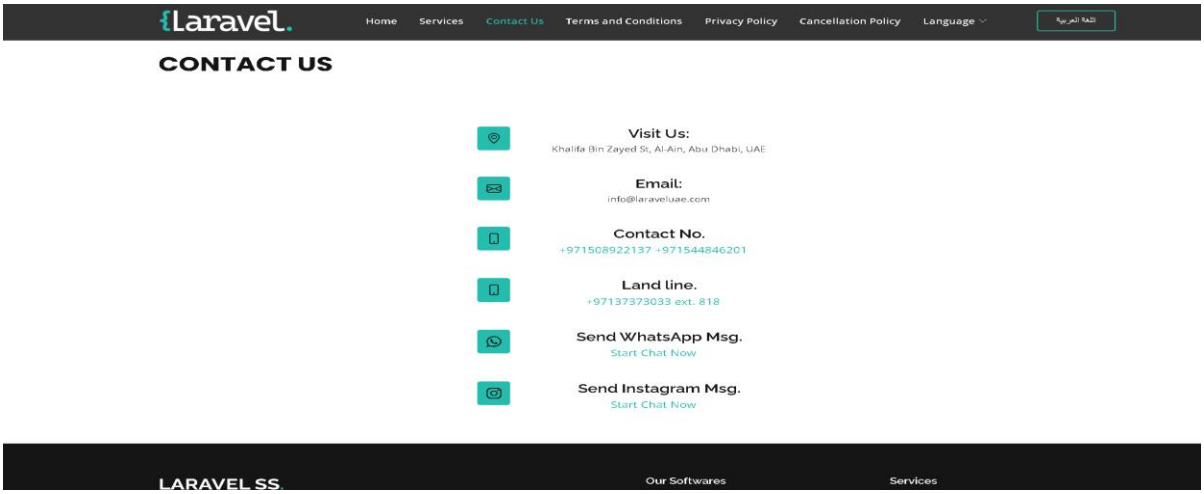

Figure 2 Contact Us Section from the Company Website

2.4 Industry of the Company

Laravel Smart Solutions is an IT and Software Development company. It offers services related to custom software programming, point of sale, business management, web design and e-marketing services. They're directly applicable to software engineering because they have to do with web building, database design, system testing, interface design and technical support.

Industry Field	**Explanation**
Software Development	Developing Business specific systems.
Business Management Systems	Assisting businesses in managing their day-to-day activities.
POS Systems	Assisting with sales, invoice and payments.
Website Design	Creating business and e-commerce web sites.
E-Marketing	Email, WhatsApp and social media marketing.

Table 3 Industry Classification

2.5 Products and Services

The company provides different management systems for businesses, including salon, tailoring, restaurant, laundry, POS, real estate, and car wash systems. These systems can assist business owners to control clients, invoices, sales, expenses, staff, and reports. The website also mentions services such as software development, website design, and e-marketing.

Product / Service	**Purpose**
Salon Management System	Manages salon customers, services, appointments, and payments
Tailoring Shop System	Manages tailoring orders and customer information
Restaurant System	Manages restaurant sales, invoices, and operations
Laundry Management System	Manages laundry orders and customer records
POS Management System	Manage sales, invoices, and payments
Real Estate System	Manage property and customer records
Car Wash Management System	Manages car wash services and customer payments
Website Design	Builds company and e-commerce websites
E-Marketing	Supports marketing through social media, WhatsApp, and email

Table 4 Main Products and Services

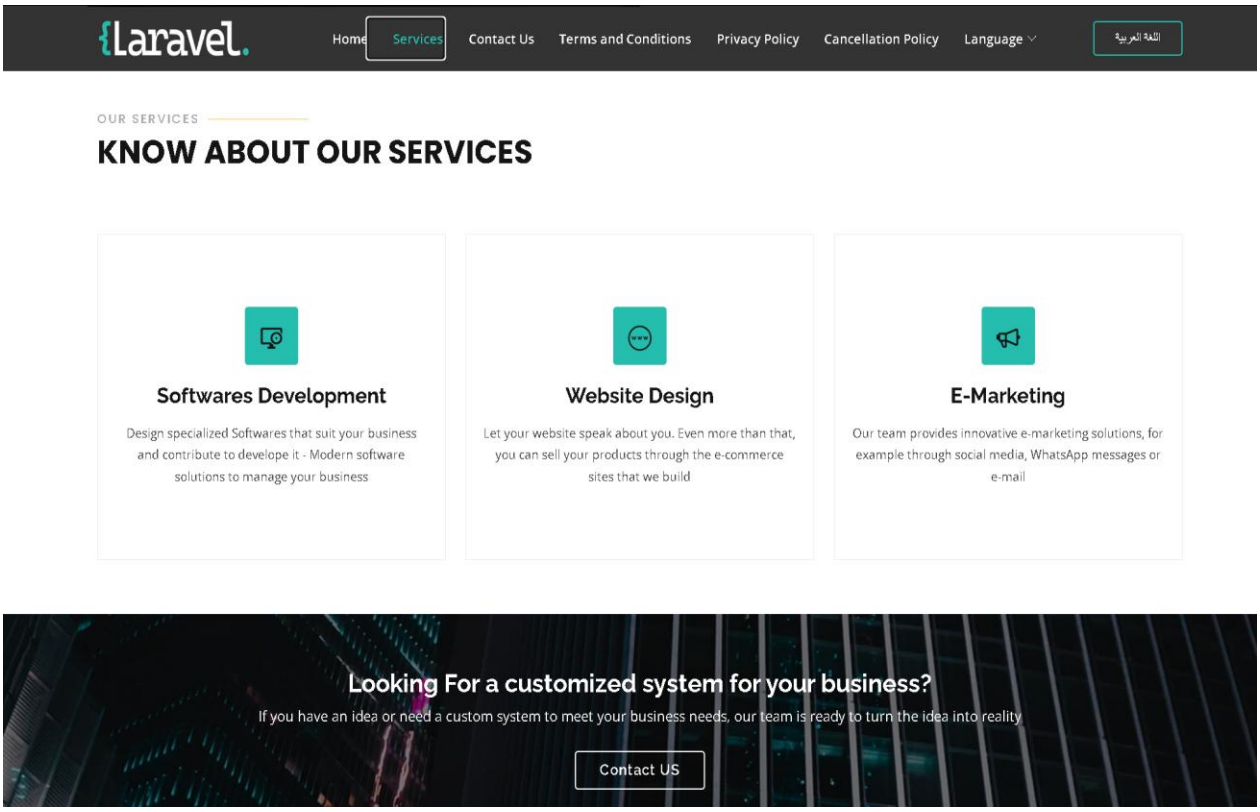

Figure 3 Company Services Section

2.6 Features Offered to Clients

The company's software products offer practical functions that aid their clients in managing day-to-day business activities. These capabilities include customer-record management, appointment scheduling, invoice creation and printing, sales and expense tracking, payment tracking, employee-performance monitoring, custom system features, WhatsApp marketing and administrative reporting[1].

The exact mix of functions will vary according to the kind of business and the customer's needs. Additional features can also be customized if the standard system is not adequate for the client's operation.

Feature	Description
Sales Invoices	Issuing and printing invoices
Sales and Expense Monitoring	Tracking business income and expenses
Employee Performance	Following employee and salesman work
Custom Features	Adding features based on business needs
WhatsApp Marketing	Sending messages and campaigns to clients
Business Reports	Helping owners review business performance

Table 5 Software Features Offered to Clients

2.7 Organizational Structure and Chart

The General Manager/Company Owner is responsible for the management of Laravel Smart Solutions. The company is primarily involved in the software development, web designing, sales and marketing. The Software Development Department will handle the back-end, front-end and database. The Website Design Department is responsible for web design, user interface design, and the Sales and Marketing Department takes care of sales, customer communication and digital marketing.

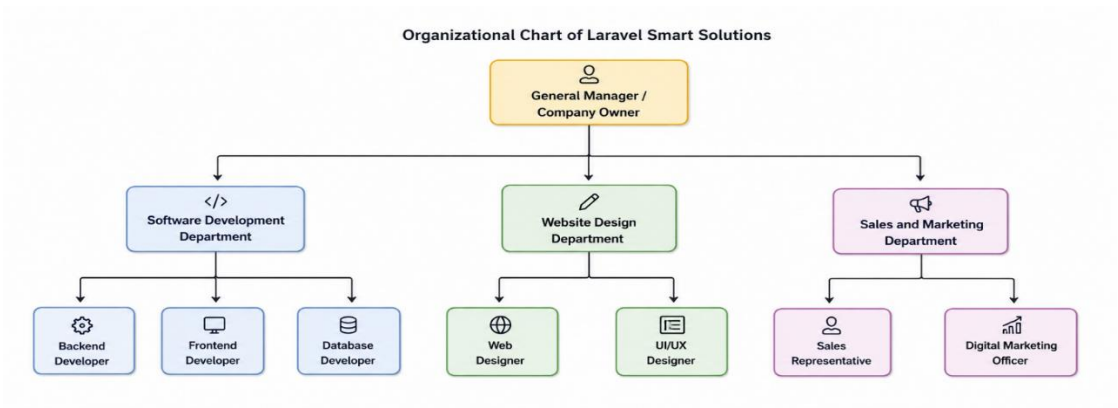

Figure 4 Organizational Chart of Laravel Smart Solutions

Department / Role	Responsibility
General Manager / Company Owner	Manages company operations and decisions
Software Development Department	Develops and maintains software systems
Website Design Department	Designs websites and user interfaces
Sales and Marketing Department	Promote services and communication with customers

Table 6 Organizational Chart Explanation

2.8 Summary

Laravel Smart Solutions is a software development company in Al Ain, Abu Dhabi. Business management systems, POS systems, website design and e-marketing services are offered by the company. It serves a variety of businesses including car-wash businesses, tailoring shops, real estate offices, restaurants, laundromats and salons. The context of their work closely related to Software Engineering, Web Development, Databases, and Real Business System Development, which made the Company appropriate for an internship.

3. Internship Activities

3.1 Introduction

This section outlines the key tasks I have performed during my 400 hours internship with Laravel Smart Solutions. Internship duties centered on gaining insights into, testing, maintaining, and improving an existing business-management application built with Laravel 10 [2]. The activities started from the preparation of the local development environment and the study of the project structure, and progressed into the implementation and testing of new features in the system.

They included Laravel routes, controllers, models, Blade views, migrations, validation, MySQL database manipulation, AJAX communications, reporting functionality, secure sharing of the invoice and online payment processing. The major work was as follows: Practicing Customer CRUD operations, enhancing Customer Management dashboard, adding Birthday and Profile-picture features, enhancing validation and user feedback, reviewing Sales and Payments Reports, adding Task Assignment and Task Report pages, generating secure public link for the invoice, and implementing Tap Payments workflow. In addition, structures for Tabby and Tamara were also constructed and prepared for activation.

The activities were carried out in a systematic way: existing implementation was reviewed, the necessary files and database tables identified, changes were made, successful and unsuccessful scenarios were tested and documented through code snippets, screenshots, tables and workflows. In this section, challenges faced, resolution achieved, skills developed and the gap between academic knowledge and professional software-development practice are also reported.

3.2 Job Duties and Working Environment

My internship work focused on web application development and system enhancement. My job was primarily with a Laravel based project, which was used in order to manage business. The work that I did was to review the system, find out problems, correct errors, test features, enhance the user interface and document what I did.

The tools I used in my working area were XAMPP, MySQL, phpMyAdmin, PHP, Composer, PowerShell/Terminal. I tested the project in the local environment before implementing or documenting project changes. After resolving setup and configuration issues, the web site was tested on localhost.

Supervisor:

My internship supervisor was Obada Alrefai. The internship work was supervised, problems checked up on and directions given regarding the tasks to be performed.

Job Duty	**Description**
Project setup	Preparing the Laravel project to run on the local machine
System review	Understanding the project structure and main modules
Database checking	Reviewing the MySQL database using phpMyAdmin
Error fixing	Solving PHP, Laravel, MySQL, and route/controller errors

Customer module improvement	Improving add, edit, delete, and validation functions
UI improvement	Improving buttons, modal display, and user feedback
Testing	Testing features after each change
Documentation	Writing weekly progress with figures, tables, and code snippets

Table 7 Main Job Duties During the Internship

3.3 Department or Division Layout

The internship work was related mainly to the Software Development Department. This department is tasked with designing, developing and enhancing software systems for clients. The work also relates to technical support – the system has to be tested and corrected when mistakes are encountered.

My main department was Software Development Department, as I have worked on backend fixes, form improvements on the front end, implementation of AJAX, working with databases and system testing.

Department / Division	**Relation to Internship Work**
Software Development	Main department related to coding, testing, and system improvement
Backend Development	Used when working with controllers, routes, models, and database logic

Frontend Development	Used when improving forms, modals, buttons, and user messages
Database Management	Used when checking tables and adding fields such as birthday
System Testing	Used to test the system after each change
Technical Support	Related to troubleshooting errors and fixing system problems

Table 8 Department Layout Explanation

3.4 Development, Database, and Communication Technologies

The company develops and maintains business management systems, leveraging web development technologies. I had an opportunity to collaborate on the Laravel project, database tools, local testing tools and user interface tools during the internship.

Laravel Framework 10 and PHP 8.1 has been used for the development of the project. This system will store the customer and business data in a MySQL database, where I was able to view and edit the database using the software phpMyAdmin [2]. The project was tested locally using XAMPP's localhost [3].

Customer, invoice, payment, task and other business information was recorded in a MySQL database [4] in the application. phpMyAdmin was used for checking the database tables, the fields and relationship, checking the stored records and also matching the records stored in the database with the records displayed in the application [5]. To run and test the application on the development computer it was necessary to provide local Apache and MySQL services, which was done by using XAMPP [6].

Blade, HTML, CSS, JavaScript, Bootstrap, Tailwind CSS, Font Awesome and SweetAlert2 were used for the user interface. Responsive layout and reusability of interface parts were supported by

Bootstrap and Tailwind CSS [8, 9]. The loading, confirmation, success and error messages were displayed using SweetAlert2 [11] and the icons for buttons and interfaces were provided by Font Awesome [12].

Technology	Version	Usage During Internship
Laravel Framework	10.0	Main PHP framework used to build and organize the web application
PHP	8.1	Backend programming language used to run the Laravel project
MySQL	Local MySQL server	Used to store customer records and system data
phpMyAdmin	XAMPP tool	Used to view, check, and manage database tables
Laravel Blade	Laravel Blade	Used to create and display the system interface pages
JavaScript	Standard JS	Used to handle user actions, buttons, modals, and validation
jQuery AJAX	jQuery AJAX	Used to send and receive data without refreshing the page
SweetAlert2	External JS library	Used to display confirmation, loading, success, and error messages
XAMPP	Local server package	Used to run Apache and MySQL locally
Composer	PHP dependency manager	Used to install and manage Laravel/PHP dependencies

Laravel Tinker	2.8	Used as a Laravel development package for testing and interacting with the application
GuzzleHTTP	7.2	Used for handling HTTP requests in PHP/Laravel
Laravel Localization	1.8	Used to support multiple languages in the Laravel project
Intervention Image	2.7	Used for image handling and processing
Laravel Mix	6.0.6	Used to compile frontend assets
Lodash	4.17.19	JavaScript utility library used in frontend development
PostCSS	8.1.14	Used for processing CSS files
PowerShell / Terminal	Windows command line	Used to run setup and project commands

Table 9 Technologies Used During the Internship

3.5 Internship Tasks and Activities

The internship tasks were designed in a sequence corresponding to the technical responsibilities that I was given. Starting from preparing the environment and checking the project structure of the Laravel. Then it went on to CRUD operations for customers, improving the dashboard and front-end, back-end development, AJAX and API communication, task-management functions, reporting and online payment integration. Every task is framed with its aim, how it was done, the files and the database involved, the difficulties faced, and the outcomes of the testing, as well as the skills acquired.

3.5.1 Task 1: Understanding the Basic Structure of Laravel

Task-Description:

The purpose of this task was to understand the structure of the existing Laravel application, prepare the local development environment, and study how routes, controllers, models, Blade views, migrations, validation rules, and database tables work together.

3.5.1.1 Introduction

The first part of my internship was to familiarize myself with the entire framework of the Laravel project before modifying anything in the project. The application had multiple interrelated modules such as customers, invoice, payment methods, reports, and user management; it was therefore essential to understand how the various components of Laravel are able to communicate with each other.

The primary goal for this task was to get a better idea of how a request flows through a Laravel application. A user, for instance, clicks on a button or fills a form, and the request goes to a route. The route is a connection to a controller method. The controller then verifies the information, sends a message to the relevant model and database table and responds back to the user with a response or Blade view [2].

This learning stage gave the technical basis for the activities of the internship later. This allowed me to see where the project's important files were, what parts of Laravel did what, and how to find out what functionality was already implemented before changing it.

Learning Objective	Description
Set-up the development environment.	Set up and configure the tools needed to run the Laravel app locally.
To become familiar with the structure of the project.	Look at the main directories of Laravel and what they do.
Study MVC architecture	Understand about models, views and controllers.
Review application routes	Discover the relationship between URLs and controller methods.
Study controller methods	Know how to deal with requests and business logic.
Review Eloquent models	Know how to communicate with database tables in Laravel.
Study Blade views	Understand how dynamic information can be presented to the user.
Review migrations	Know how Laravel works with creating and altering database structure.
Understand validation	Understand how to verify a user's information before saving.
Investigate the steps in the request lifecycle	Trace a request to the database and back to the front-end page.

Table 10 Main Learning Objectives of Task 1

3.5.1.2 Local Development Environment Preparation

The first practical activity involved setting up the project to be able to run on my local computer. I downloaded the project files and opened them with Visual Studio Code. Set up and configured the main tools the project required including XAMPP, Composer, PHP, MySQL & Apache.

Apache web server and MySQL database server were supplied by XAMPP [6]. Apache was used for the browser hosting of the Laravel application while MySQL hosted the data [4], [6].

phpMyAdmin was used to check the database tables, columns, relationship and store information [5].

The PHP packages needed for this project were installed and managed by composer [7]. After checking the project's composer.json file, I knew that the app needed PHP 8.1 or higher and Laravel Framework 10 [2], [3].

```
</>Bash

composer install
php artisan key:generate
php artisan config:clear
php artisan cache:clear
```

Code Snippet 1 Main Project Setup Commands

The composer installs command installs the packages specified in composer.json [7]. php artisan key:generate is used to generate the application encryption key. The configuration and cache commands clear out old settings that can otherwise cause issues to run the project properly [2].

Then, I changed the database connection in the environment file and ran Apache and MySQL via XAMPP. I managed to fix the startup issues and opened the app on localhost successfully.

Tool or Technology	**Purpose**
Visual Studio Code	Opening, reading, and editing project files.
XAMPP	Apache, MySQL, and phpMyAdmin.
Apache	Running the Laravel application locally.
MySQL	Storing application records
phpMyAdmin	Accessing and working with database tables.

Composer	Installing and managing PHP dependences.
PHP 8.1	Running the Laravel back-end code.
Laravel 10	Main web application framework,.
PowerShell	Executes Composer and Artisan.
Web browser	Opening and testing the local application.

Table 11 Development Tools Used

Challenge Faced	**Solution Applied**
Apache failed to start.	Verified Apache configuration and ports.
MySQL service did not run correctly	Restarted MySQL and checked the database configuration.
Must install Laravel packages.	To install the project dependencies, run the Ran Composer.
Application key was not specified in the Laravel application.	Using the Artisan command, created a new key.
Project configuration was not updated with new values	Cleared the Laravel configuration and application cache.
Database connection failed	Reviewed the database values in the environment file.
Project failed to start on localhost.	Verified Apache, project location and local URL.

Table 12 Environment Setup Challenges and Solutions

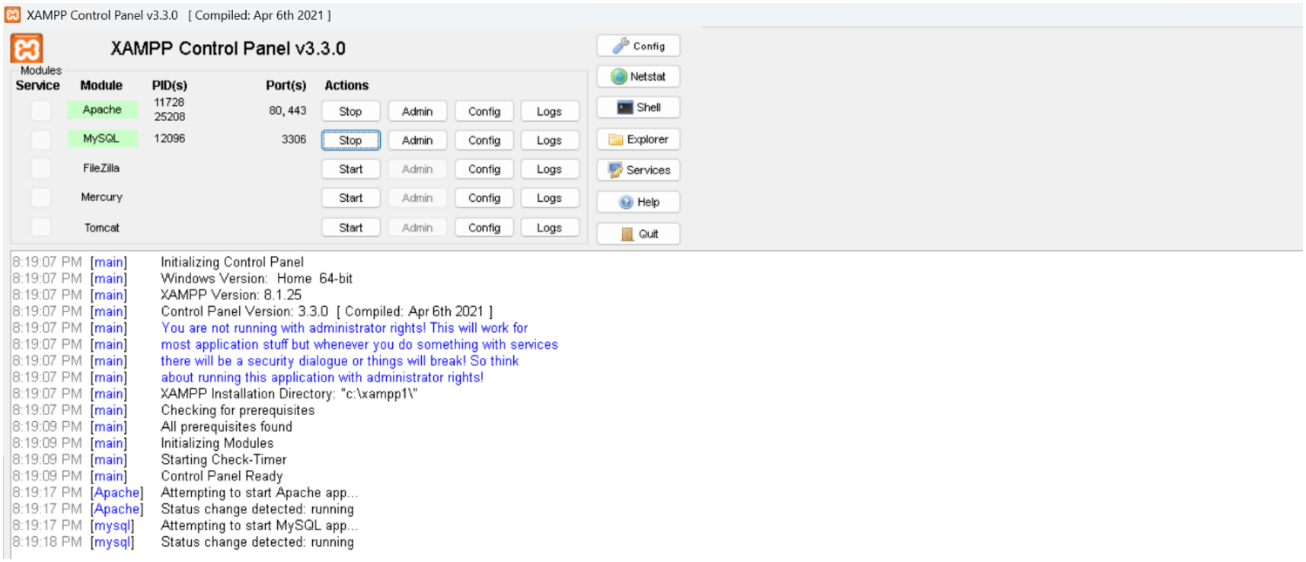

Figure 5 XAMPP Control Panel with Apache and MySQL Services Running

Here the XAMPP Control Panel when the Apache and MySQL services are started. Apache is running on 80, 443 and MySQL is running on 3306. In this manner, it was sure that the local web server and local database server were ready to run and test the Laravel application at localhost.

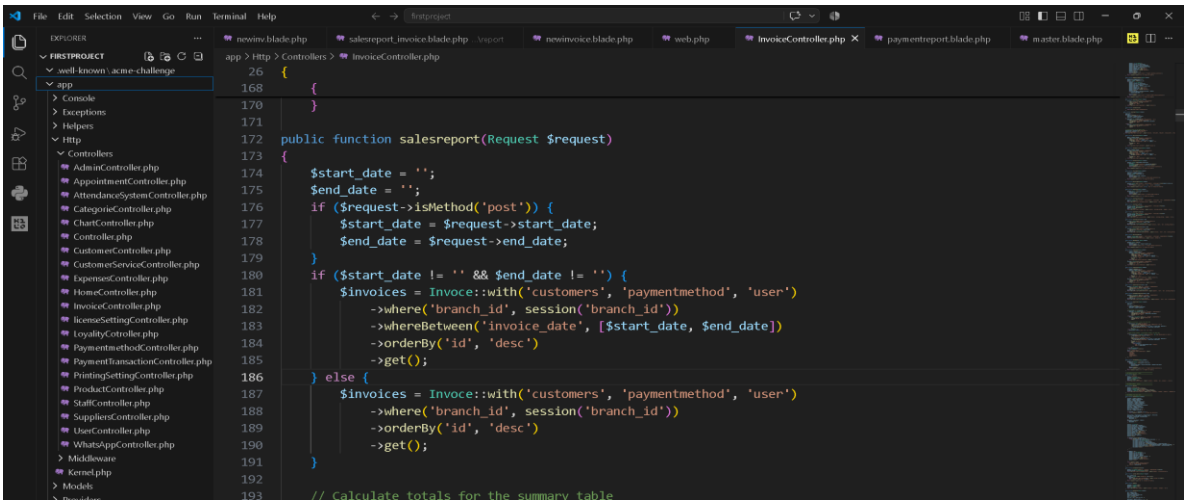

Figure 6 Laravel Project Structure Opened in Visual Studio Code

This figure shows the Laravel project opened in Visual Studio Code. The Explorer panel shows the main project folders such as the app directory and the Http/Controllers folder that contains the

system controller files. The InvoiceController.php file is opened as well, which displays some of the logic on the back-end of the application that is used to pull invoice records and create sales reports. This helped me to understand how Laravel files are organized and where controller methods are set up.

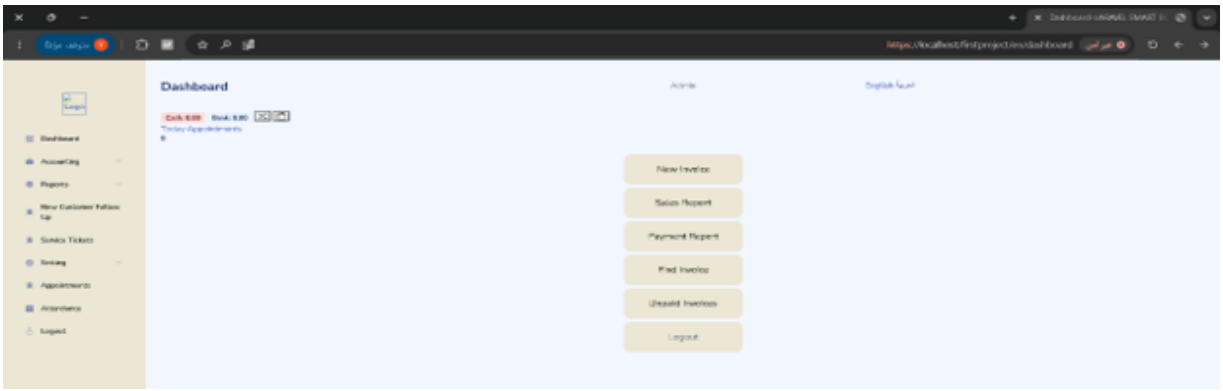

Figure 7 Laravel Application Dashboard Running Successfully on Localhost

This figure represents the Laravel application running successfully with localhost in web browser. Apache, MySQL, the Laravel project, and the connection to the database were configured correctly, as shown in the display of various buttons on the dashboard, in the navigation menu, and in the main system buttons. This also proved that the local development environment was in place for review, test and modify of the application.

3.5.1.3 Understanding the Laravel Project Directory Structure

I've successfully run the application and I am going to review the directory structure of Laravel. This was useful to me for knowing where to find several types of project files, and how Laravel organizes a big web app.

The project had typical Laravel structures, such as app, resources, routes, database, config, public, and storage [2]. An individual director is responsible for a particular function.

The main application logic, including controllers, models, middleware, helpers, and service providers are located in the app directory. The resources directory has Blade templates and other front-end resources. Routes is a directory that contains the application URLs. Migration files are stored in the database directory and application configuration files are stored in the config directory.

The public directory stores files that can be directly accessed via the browser such as CSS, JavaScript, images and uploaded public files. Storage Directory: This is where the logs, cache files, generated files, , will be stored.

3.5.1.4 Understanding the MVC Architecture

Laravel is a Model–View–Controller application [2]. MVC breaks down the application into various parts which makes it easier for anyone to understand, maintain and update the code.

The Model is the representation of the application's data and the interaction with the database. The View is used for displaying information for the user. The Controller receives the request, processes it and uses some application logic, calls the model and returns a view or some other response [2].

When the customer page is opened, for instance, the request is received by a customer route. The index() action is called in CustomerController. The Customer Blade view receives the customer records from the controller, via the Customer model.

MVC Component	**Project File**	**Responsibility**
Model	app/Customer.php	Represents the customer's table.
View	resources/views/admin/customer/customers.blade.php	Shows the customer form and customer table.

Controller	app/Http/Controllers/CustomerController.php	Manages applications and customer requests.
Route	routes/web.php	Relies the browser request to the controller.
Database	customers table	Stores customer records

Table 13 MVC Components in the Customer Module

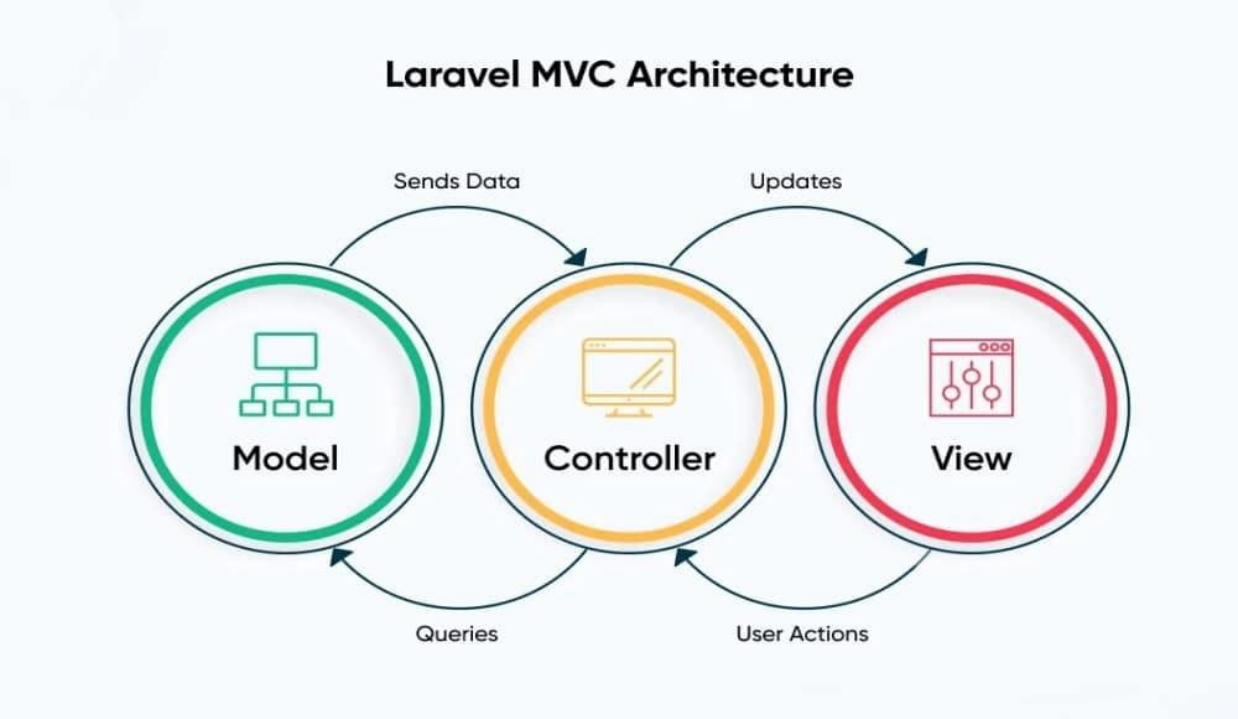

Figure 8 Laravel MVC Architecture and Request Lifecycle

It's a diagram that shows the user request flow in Laravel. A request is initially received by a route that then passes the request to the correct controller. The controller communicates with model to get or set data in a MySQL database. The result is processed, and sent to a Blade view, where the page to be displayed to the user is generated.

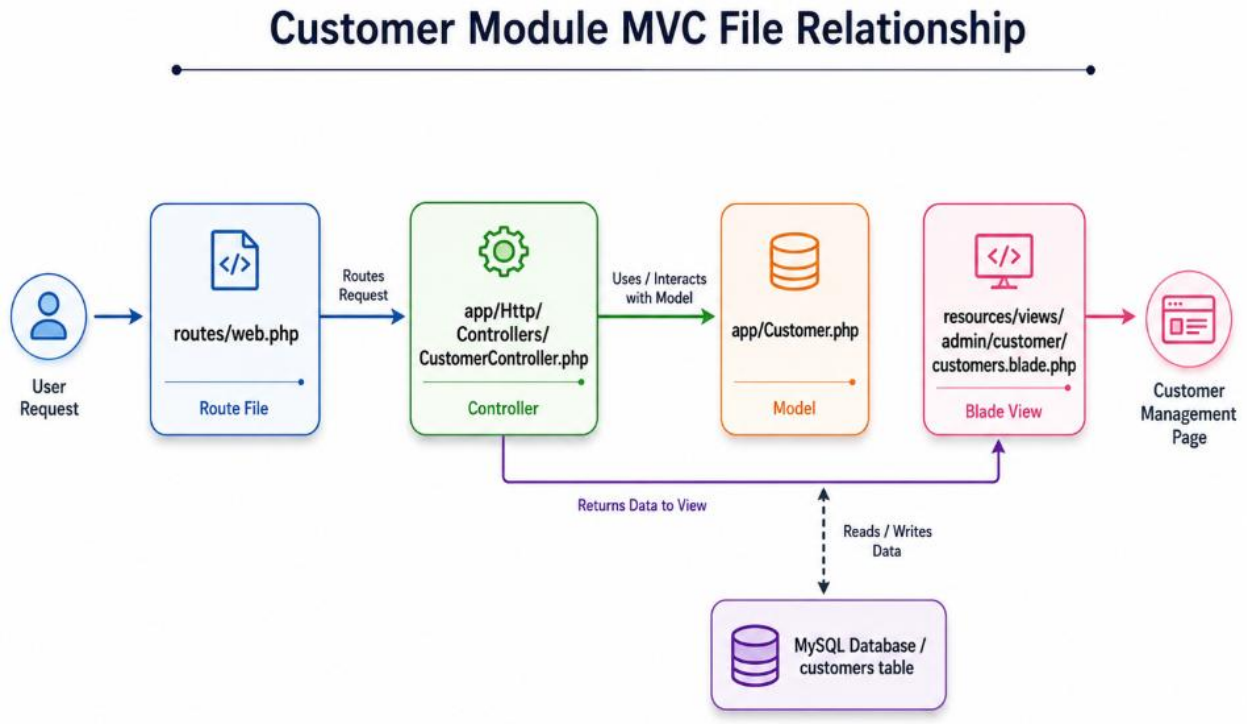

Figure 9 Customer Module MVC File Relationship

This is a diagram of the main files that are associated with the Laravel customer module. Routes/web.php receives the user request and calls the correct method of the CustomerController.php. The Customer.php model is used by the controller to read and/or modify the records in the MySQL customers table. The user requests that the controller show them the page listing the customer management screen: customers.blade.php is sent to the user by the controller after processing the request.

3.5.1.5 Studying the Laravel Routing System

One of the biggest portions of this work was to review the Laravel routing system. Routes are used to set up the URL that the user is able to access, and which controller method should be called to handle such requests [2].

I have read through and learned about the routes/web.Php file and distinct types of HTTP requests. GET is a type of request that is typically used to get or display information. A POST route is used to post or create information. To update a record, use PUT or PATCH routes and use a DELETE route to delete a record [2].

There are multiple customer related routes in the project. These are the routes that link to methods within CustomerController for the Customer Page operations.

```php
</> PHP

Route::get('allcustomer', 'CustomerController@index')
   ->name('allcustomer');

Route::post('storeCustomer', 'CustomerController@storeCustomer')
   ->name('storeCustomer');

Route::get('editcustomer/{id}', 'CustomerController@editcustomerinfo')
   ->name('editcustomer');

Route::post('updatecustomer/{id}', 'CustomerController@updatecustomer')
   ->name('updatecustomer');

Route::get('customerinvoice/{id}', 'CustomerController@customerinvoice')
   ->name('customerinvoice');

Route::delete('deletecustomer/{id}', 'CustomerController@deletecustomer')
   ->name('deletecustomer');
```

Code Snippet 2 Customer Routes from web.php

These routes are showing the correlation of user actions to controller methods in Laravel. For instance, when the customer page is opened, the index() method is executed and when new customer information is submitted, the storeCustomer() method is executed.

I also noticed that customers routes were located within an authentication middleware group. This is so that only logged-in users can access the routes.

```
</>PHP
Route::group(['middleware' => ['authMiddleware']], function (){

  Route::get('dashboard', 'AdminController@index')
    ->name('dashboard');

  Route::get('allcustomer', 'CustomerController@index')
    ->name('allcustomer');
});
```

Code Snippet 3 Authentication Middleware Route Group

This group of routes can be used for securing administrative pages against unauthenticated access[2].

HTTP Method	Route	Controller Method	Purpose
GET	allcustomer	index()	Displays all customers
POST	storeCustomer	storeCustomer()	Saves the new Customer.
GET	editcustomer/{id}	editcustomerinfo()	Fetches customer data to edit.
POST	updatecustomer/{id}	updatecustomer()	Updates an existing customer.
DELETE	deletecustomer/{id}	deletecustomer()	Clears a Customer record.

GET	customerinvoice/{id}	customerinvoice()	Displays invoices related to a customer.
GET	customerrenewals/{id}	customerrenewals()	Displays customer renewals

Table 14 Customer Routes and Their Purposes

Figure 10 Customer Routes Defined in the web.php File

This number represents the routes which have been specified for the user in the Laravel routes/web.php file. Each route maps to a particular customer operation like: show customers, add customers, edit customers, update customers, or delete customers, to the respective method in CustomerController.php.

3.5.1.6 Reviewing Laravel Controllers

Controllers receive requests and arrange the actions to get a response[2]. I have checked out the CustomerController.php to see how the project fetches customer data and passes it on to the customer page.

The index() method creates the page title, retrieves all customers from the database, orders the records by customer ID, and sends the information to the customer Blade view.

```php
</> PHP

public function index()
{
    $title = 'All Customers';

    $customers = Customer::orderBy(
        'customer_id',
        'desc'
    )->get();

    return view(
        'admin.customer.customers',
        compact('title', 'customers')
    );
}
```

Code Snippet 4 CustomerController index() Method

The following approach shows how the controller, model and view are related:

1. The controller defines page title.
2. Customer model is used to fetch customer records.
3. The customers order from newest to oldest.
4. Controller returns customer Blade view.
5. $title and $customers variables are available in the view.

Also, I looked at controller methods to receive data from forms. These are the methods which verify the data that is sent through, manipulate the model, and send back a JSON response to the front-end page.

3.5.1.7 Understanding Eloquent Models and Database Communication

Laravel communicate with database by using Eloquent ORM. Typically, each Eloquent model corresponds to a single table in the database. The developer can use model methods create(), where(), first(), get(), save(), and delete(), instead of writing SQL queries for each operation [2].

I checked the customer model to figure out how does Laravel link with the customer table. The project uses customer_id as the primary key instead of Laravel's default id column.

```php
</> PHP

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Customer extends Model
{
  protected $primaryKey = 'customer_id';

  protected $fillable = [
    'customer_name',
    'phone',
    'email',
    'notes'
  ];
}
```

Code Snippet 5 Customer Eloquent Model

The $primaryKey property tells Laravel that customer_id is the primary key of the table. Defines the fields that can be mass-assigned during the creation or update of a customer.

The $fillable property is also a security feature, as it doesn't allow unknown request fields to be automatically added.

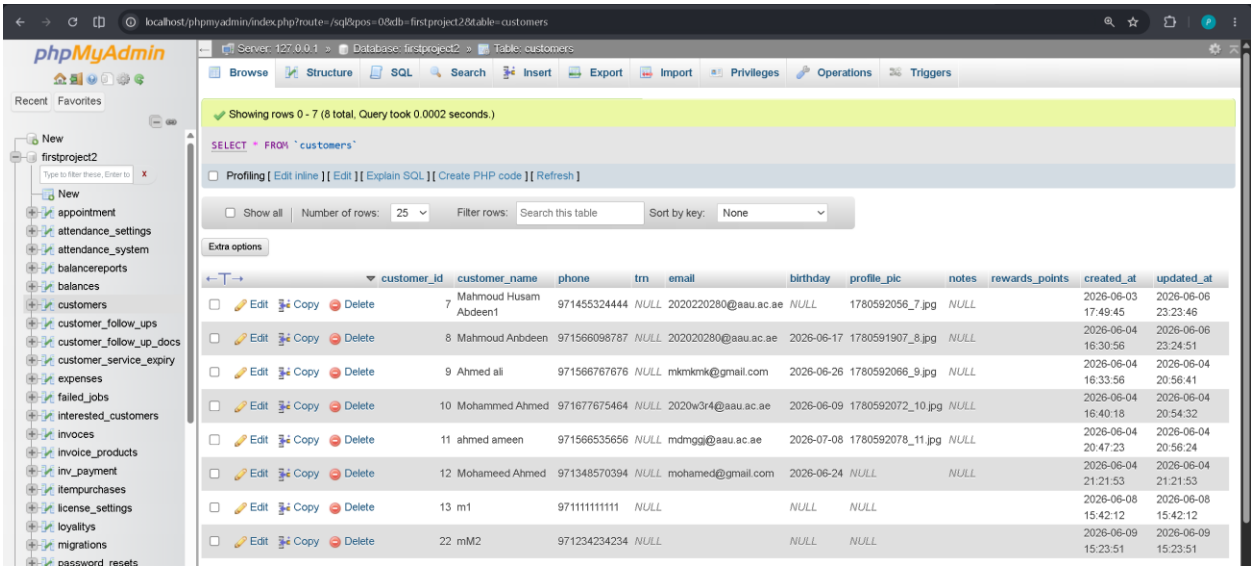

Figure 11 Customers Database Table Displayed in phpMyAdmin

This figure shows the customers table in the MySQL database using phpMyAdmin. The table has the following fields that are related to customers: customer_id, customer_name, phone, email, birthday, profile_pic, notes, rewards_points, created_at, and updated_at. As seen in the stored records, the information entered by the customers in the Laravel application is stored in the database successfully.

3.5.1.8 Understanding Blade Views

Blade is the template engine that is used in Laravel. Blade files are the ones that have normal HTML and some Laravel directives that enable dynamic display of data [2].

I went through the customer Blade page and read directives like:

- @extends
- @section
- @foreach
- @if
- @csrf
- {{ }}

The @extends directive is used to link a page to a main layout. The @section directive will be used to specify the content of the page. The foreach statement will render HTML for each record passed from the controller. These are conditionally displayed with the @if directive and variables are safely printed with {{ }} [2].

```
</> blade
@extends('master')
@section('title', 'Dashboard')
@section('main_content')
   <!-- Customer page content -->
@endsection
```

Code Snippet 6 Blade Page Layout Structure

This way, the customer page can share the common layout of the master page and prevent the navigation bar, header, scripts and page structure from being duplicated.

Blade Directive	Purpose
@extends	Applies a master layout to a page.
@section	Specifies the content in a section of a layout.
@foreach	Reiterates the information for each record.
@if	Outputs content if a condition is true.
@csrf	Forces security token to be added to forms.
@include	Inserts another Blade file.
{{ $variable }}	Prints unescaped content as necessary.
{!! $variable !!}	Displays unescaped content when required
@error	Shows field validation errors.

Table 15 Blade Directives Reviewed

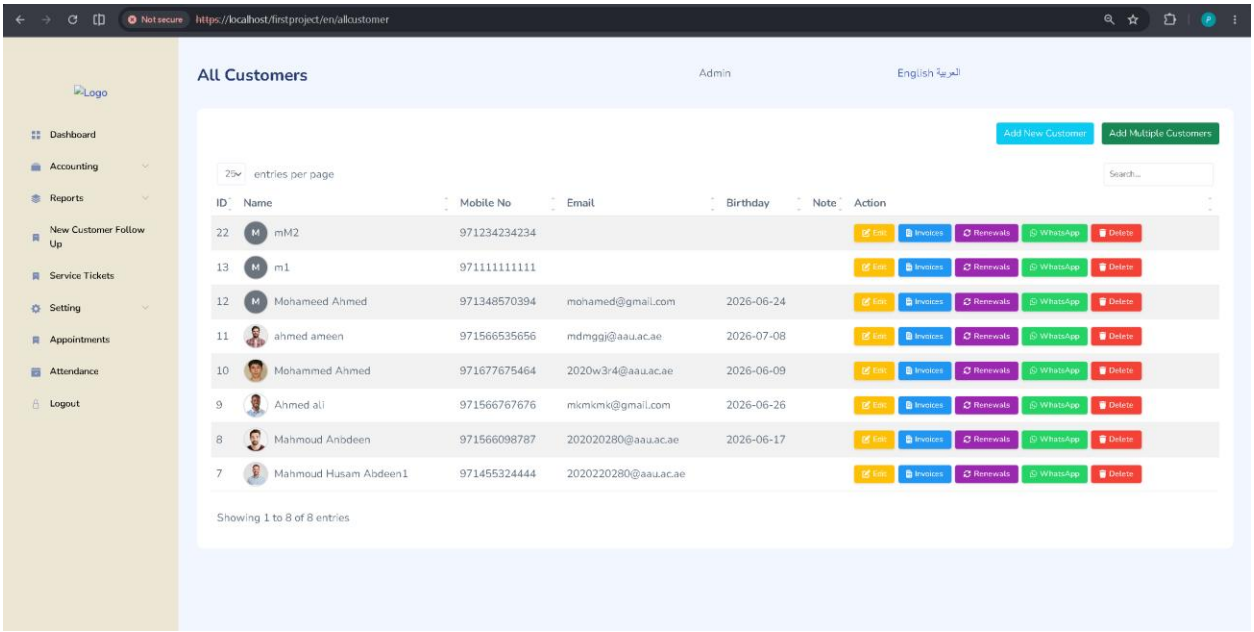

Figure 12 Dynamic Customer Records Displayed on the Customer Management Page

This figure is a display of the customer records on the customer-management page dynamically. The Customer model is used to get data from MySQL Customers table, passed to the Controller and then to the Blade view. This displays customer details including their Customer ID, name, mobile number, email address, date of birth, profile image and the actions they can take, including Edit, Invoices, Renewals, WhatsApp and Delete.

3.5.1.9 Understanding Database Migrations

A migration defines a database table using PHP code and is used to create and update the table. They offer a safe means of performing database structure modifications without manually altering each database via SQL [2].

Typically, each migration will have an up() and a down() method. Up() is for applying the database change and Down() is for reverting the change [2].

I have read the files for the migration of projects and have used this knowledge when new fields were needed later. Adding the birthday column into the customer's table was one example of the migration that was used.

```bash
</> Bash

php artisan migrate
php artisan migrate:status
php artisan migrate:rollback
```

Code Snippet 7 Additional Migration Commands Reviewed

These commands are used to apply migrations, to check the status of migrations, and to roll back the last migration group.

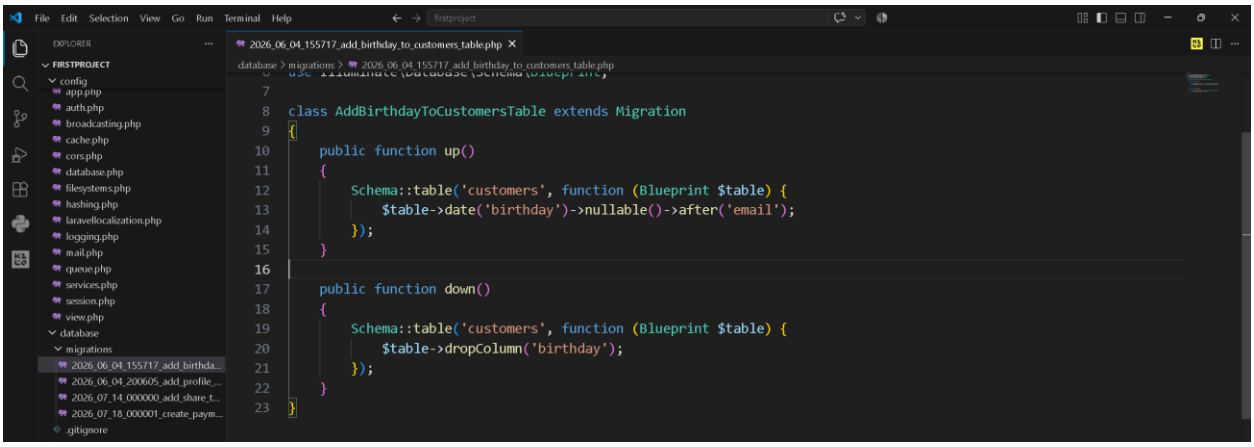

Figure 13 Migration File Used to Add the Birthday Field

The figure displays the Laravel migration file: 2026_06_04_155717_add_birthday_to_customers_table.php The up() method appends a nullable date column for the Birthday to the customers table behind the email column. The down() method will execute when the migration is rolled back, it will delete the birthday column if it has been removed from the database.

	customer_id	customer_name	phone	trn	email	birthday	profile_pic	notes	rewards_points	created_at	updated_at
Edit Copy Delete	7	Mahmoud Husam Abdeen1	971455324444	NULL	2020220280@aau.ac.ae	NULL	1780592056_7.jpg	NULL		2026-06-03 17:49:45	2026-06-06 23:23:46
Edit Copy Delete	8	Mahmoud Anbdeen	971566098787	NULL	202020280@aau.ac.ae	2026-06-17	1780591907_8.jpg	NULL		2026-06-04 16:30:56	2026-06-06 23:24:51
Edit Copy Delete	9	Ahmed ali	971566767676	NULL	mkmkmk@gmail.com	2026-06-26	1780592066_9.jpg	NULL		2026-06-04 16:33:56	2026-06-04 20:56:41
Edit Copy Delete	10	Mohammed Ahmed	971677675464	NULL	2020w3r4@aau.ac.ae	2026-06-09	1780592072_10.jpg	NULL		2026-06-04 16:40:18	2026-06-04 20:54:32
Edit Copy Delete	11	ahmed ameen	971566535656	NULL	mdmggj@aau.ac.ae	2026-07-08	1780592078_11.jpg	NULL		2026-06-04 20:47:23	2026-06-04 20:56:24
Edit Copy Delete	12	Mohameed Ahmed	971348570394	NULL	mohamed@gmail.com	2026-06-24	NULL	NULL		2026-06-04 21:21:53	2026-06-04 21:21:53
Edit Copy Delete	13	m1	971111111111	NULL		NULL	NULL			2026-06-08 15:42:12	2026-06-08 15:42:12
Edit Copy Delete	22	mM2	971234234234	NULL		NULL	NULL			2026-06-09 15:23:51	2026-06-09 15:23:51

Figure 14 Birthday Column Added to the Customers Table in phpMyAdmin

This is what the birthday column will look like if it was added by a Laravel migration. The newly created field is now listed in the highlighted column, indicating that the field has been successfully added to the database and is ready for use to store customer birth dates. Some of the records have already got the birthday value, some of them are still NULL which is OK because the field is nullable.

3.5.1.10 Understanding Laravel Request Validation

Prior to entering the information into the database, it must be validated. Laravel offers a validation system which validates required fields, data types, maximum length of data, email format, number lengths and custom rules [2].

The validation logic in CustomerController was studied. The project validates the customer's name, telephone, email and notes prior to saving customer information.

```
</> PHP

$validator = Validator::make($request->all(), [
    'customer_name' => 'required|string|max:255',
    'phone' => 'required|string|digits:12|starts_with:971',
    'email' => 'nullable|email|max:255',
    'notes' => 'nullable|string|max:1000',
]);
```

Code Snippet 8 Customer Validation Rules

These rules involve the phone number and name of the customer. The phone number must be 12 digits long and start with 971. Email and notes are optional but should be entered according to the correct format.

Field	Validation Rules	Purpose
Customer name	The data type is required, string, and max length is 255 characters.	Checks for a customer name to be entered must be validated.
Phone number	Required, string, 12 characters, beginning with 971	Ensure that the UAE-format phone number is correct.
Email	Valid optional e-mail address (max length 255 characters)	Checks and rejects incorrect email addresses.

Notes	Dedicated to the manufacturer's use.Dedicated to the use of the manufacturer, optional, string up to 1000 characters.	Restricts the note size of customers.

Table 16 Customer Validation Rules

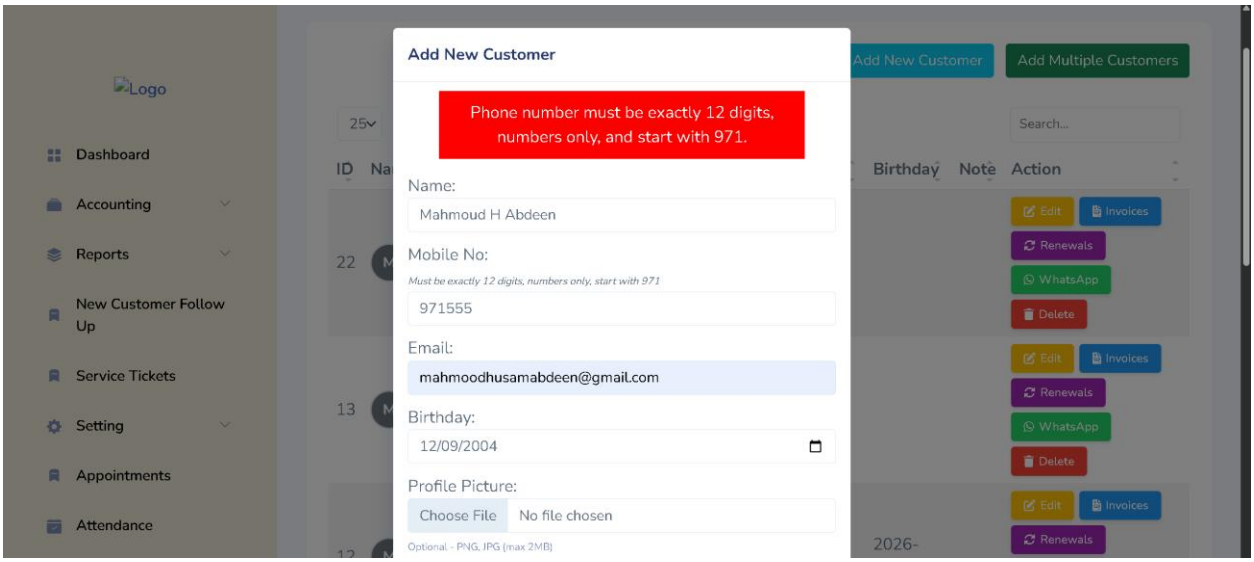

Figure 15 Validation Error Message Displayed for an Invalid Phone Number

This is the screen that will be shown on the customer page when an invalid phone number is entered. The message tells the user that they need to enter a phone number that must be 12 digits long, consist of numbers only, and must begin with 971. This is to be sure that the form validation for the customer is successful before entering the customer into the database.

3.5.1.11 Tracing the Complete Laravel Request Lifecycle

I have followed the entire flow of the customer page request so that I could get the concepts together.

The user makes a GET request to the allcustomer route when accessing the customer page. The route is invoking the index() method of CustomerController. In the above, Customer records are

retrieved from MySQL by the controller using the Customer model. That information is then transferred to the customer Blade view. Finally, Blade creates the HTML table and sends it back to the browser.

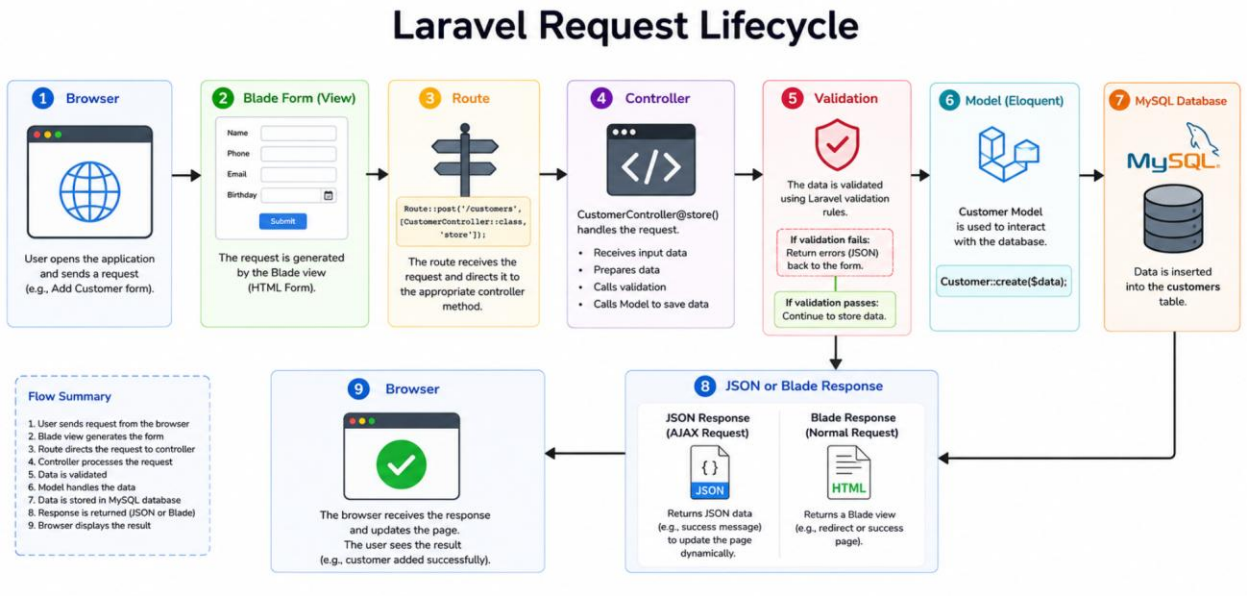

Figure 16 Complete Laravel Request Lifecycle

This figure shows the end-to-end request life cycle of the Laravel application. This is initiated by the user filling out a form in the browser. The Blade view creates the request, this is picked up by the route handler, and passed to the correct controller action. The submitted data is then handled by the controller which applies Laravel validation rules. In case of validation failure, the system responds with error messages in Json format. On successful validation, the controller updates or adds data into the MySQL database using Eloquent model. Lastly, Laravel will return a JSON if the request is made via AJAX, or a Blade response if it is a normal request, which the browser will show to the user [2], [4], [10].

3.5.1.12 Challenges Faced and Solutions Applied

Challenge Faced	Solution Applied
There were numerous folders and files in the project.	Learned the Laravel files and folders.
Sometimes unclear how the pages are related to the PHP code.	Followed the path of each page in the Blade view to its path and controller.
The application was not initially able to run	Reviewed the XAMPP, Composer packages, PHP configuration and environment settings.
Records in databases were not readily apparent	Analyzed tables, columns and data stored in them using phpMyAdmin.
Routes names/Controller methods were not easily understandable	As the namespaces for each controller changed, used web.php to redirect each URL to its controller method.
The responsibility of controller(s) and model(s) was not defined	Followed Eloquent queries in methods of controllers.
It was not easy to understand the dynamic data in Blade	Looked at the variables sent on compact() and rendered using Blade directives.
Change in the database had to be comprehended	Added more comments to the migration up() and down() methods.
There were validation errors for tracing.	Rechecked rules for the controller validation and JSON responses.
The entire lifecycle of requests was complicated	Developed a framework of data flow from browser to database and vice versa.

Table 17 Task 1 Challenges and Solutions

Skill	Learning Outcome
Laravel project setup	Be able to set up and develop a local Laravel project.
Directory navigation	Knows how to find controllers, models, views and routes and migrations.
MVC understanding	Understands how to communicate within components in Laravel.
Route analysis	Capable of figuring out the method that would process a given page request.
Controller analysis	Can read and comprehend requests and responses.
Eloquent ORM	Knows how to work with a database using models.
Blade templates	Capable of comprehending the dynamic generation of the front-end of a page.
Database migrations	Be able to comprehend managed changes to databases.
Request validation	Able to do input checking prior to database storage.
Debugging	Capable of tracking down errors in starting and running programs.

Table 18 Skills Developed During Task 1

3.5.1.13 Task 1 Conclusion

In this task, I set up the Laravel project and was able to run it on my development machine. Set up and configured XAMPP, Apache, MySQL, Composer, PHP 8.1 and Laravel project requirements. I also got rid of some initial setup issues with the server, database, missing packages and configuration of the application and of the Laravel cache.

Once the project was done, I went through the Laravel directory structure and found out where controllers, models, Blade views, routes, migrations, configuration files, front end assets, logs and generated files are stored.

I researched the MVC design of Laravel and how the requests made by the browser flow from the routes, controllers, Eloquent models, MySQL database and finally to the Blade views. I investigated real customer routes, controller strategies, Customer model, different templates of Customer Blades, migration files and validation rules of the internship project.

I also got to know the concepts of Mindware, Artisan commands, Application Configuration, CSRF Protection, Named Routes, JSON Response and Laravel Logs. These concepts of support enabled me to comprehend the organization and securing of the project.

I got a good understanding of the architecture of a Laravel App. I was able to find the right files, know that each piece of code is supposed to do, track the entire process of a request, and discern how the front-end pages are related to back-end logic and the database. This knowledge equipped me to start performing CRUD operations, enhance customer-management operations, modify invoice pages and utilize more advanced features of the system with the tasks I was handed later in the internship experience.

3.5.2 Task 2: Practice CRUD Operations for Customer Records

Task Description:

Practice CRUD operations by creating, reading, updating, and deleting records in the system, especially for customer-related data.

3.5.2.1 Introduction

In this task, I applied the four basic operations of databases that are utilized in web applications: Create, Read, Update and Delete (CRUD). The area of my work was the Customer Management part of the Laravel application [2], [4].

The goal was to not just add customer buttons and form fields but also to see how each of the customer operations flow through the Laravel application. I traced each of the customer Blade pages to the corresponding Laravel routes, controllers, Eloquent models and MySQL table [2], [4].

I also used AJAX to allow customer information to be sent and received without the need to refresh the entire page [10]. Loading indicators, success messages, error notifications and deletion confirmations are shown by using SweetAlert2 [11].

New features added to the uploaded task document include customer deletion, support for birthdays, AJAX operations, validation and customer profile-picture support. In the broader task notes, you will learn about the Create, Read, Update and Delete (CRUD) workflow and how it ties in with the Laravel routes, controllers, models, Blade views and database [2], [4], [10].

.3.5.2.2 Task Objectives

Objective	Description
Create customer records	Add new customer data to the database.
Read customer records	Read and present the details of customers.
Update customer records	Modify existing customer information.
Delete customer records	Securely take off customer information.
Validate input	Don't store invalid customer information.
Use AJAX	Communicate data with out the full page refresh.
Improve feedback	Show loading, success, error and confirmation messages.
Verify database changes	Compare Web page with records in MySQL.
Handle related data	When invoices are linked, make sure that they cannot be deleted unsafely.

Table 19 Main Objectives of Task 2

3.5.2.3 Main Files and Components

Several connected Laravel files with the customer CRUD functionality. These forms, table, buttons, modals and AJAX code are all included on the Blade page. The route file is used to associate each customer action to a controller method. The controller validates and processes the request and the Customer model interacts with the database [2], [4].

File or Component	Purpose
resources/views/admin/customer/customers.blade.php	Shows customer forms, records, buttons, modals, validation messages as well as AJAX code.
routes/web.php	Maps customer operations to CustomerController operations.
app/Http/Controllers/CustomerController.php	Handles Create, Read, Update and Delete requests from the customers.
app/Customer.php	Represents a customers table with Eloquent.
customers table	Stores customer information
customerinvoice.blade.php	Shows all of the invoices for a selected customer.
Birthday migration	Includes Birthday (optional) field.
Profile-picture migration	Includes the optional profile-picture file name.
jQuery and AJAX	From the customer page, sends background requests.
SweetAlert2	Shows loading, confirm, success and error messages.
phpMyAdmin	Validates customers data in MySQL.

Table 20 Main Files and Components Used

Sources: Laravel documentation [2], MySQL documentation [4], phpMyAdmin documentation [5], jQuery documentation [10], and SweetAlert2 documentation [11].

3.5.2.4 Customer CRUD Architecture

The overall architecture of each customer action is the same as the general Laravel architecture: Customer Blade Page/ Laravel Route/ CustomerController/ Customer Model/ MySQL Database/ JSON/ Blade Response [2], [4], [10].

For instance, upon submission of the Add Customer form, AJAX will submit to the storeCustomer route. Laravel runs the storeCustomer() method inside of CustomerController. This method checks the input data and then it inserts the new data to the database by the Customer model [2], [4], [10].

Once operation is done the controller returns the JSON. The response from this action is read by JavaScript and a success or error message is posted on the customer page [2], [10].

Component	**Action**
Customer page	An operation is selected by the administrator.
JavaScript or form	Customer data are gathered.
AJAX	Laravel is requested to do something.
Route	The request is related to a controller method.
Controller	Request is accepted and executed.
Customer model	The action of the database is performed.
MySQL	Customer information is added, accessed, modified or removed.
Controller	It returns a JSON/Blade response.
Customer page	The result is displayed to the administrator.

Table 21 CRUD Request Flow

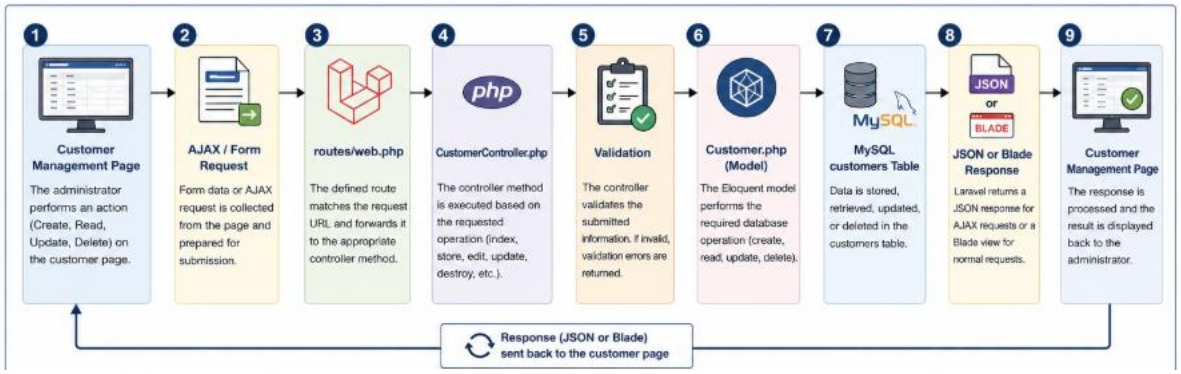

Figure 17 Customer CRUD Operation Workflow in Laravel

This figure shows the entire process of Customer CRUD Operations in the Laravel application. It starts when the administrator acts on the customer-management page (for adding/seeing/editing/deleting a customer). The form or AJAX request is directed to the corresponding route in routes/web.php which invokes the corresponding method in CustomerController.php. Now the controller validates the data and will create, retrieve, update or delete the customers' records from the MySQL customers table using the Customer.php Eloquent model. Laravel will return the Json response if it's an AJAX request, or a Blade view if it's a normal page request and the results will be displayed on the customer-management page.

3.5.2.5 Customer Eloquent Model

The Customer.php model is the entity that models the customer's table within the Laravel app [2], [4]. The table does not have an id primary key, but rather customer_id.

```php
</> PHP

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Customer extends Model
{
  protected $primaryKey = 'customer_id';

  protected $fillable = [
    'customer_name',
    'phone',
    'email',
    'notes'
  ];
}
```

Code Snippet 9 Customer Model

Laravel is configured by the $primaryKey property to look for customer records by using customer_id. The $fillable property defines which fields are allowed to be filled in with Customer:create() [2].

Property	**Purpose**
$primaryKey	Set customer_id as the primary key of the table.
$fillable	Describes the fields that can be mass-assigned.
Customer::create()	Creates the new customer record.

Customer::where()	Searches for customers, based on a condition.
$customer->save()	Saves changes to an existing customer.
$customer->delete()	Deletes a customer record.

Table 22 Customer Model Properties

Source: Laravel Eloquent documentation [2].

3.5.2.6 Create Operation: Adding Customer Records

The first CRUD operation that I worked with was creating customer records. The customer page has an Add New Customer button which will trigger a model form. The form has space for:

- Customer name
- Mobile number
- Email address
- Birthday
- Profile picture
- Notes

The phone field is mandatory and has a max length of 12 and is assigned a HTML pattern with 9 digits after country code UAE.

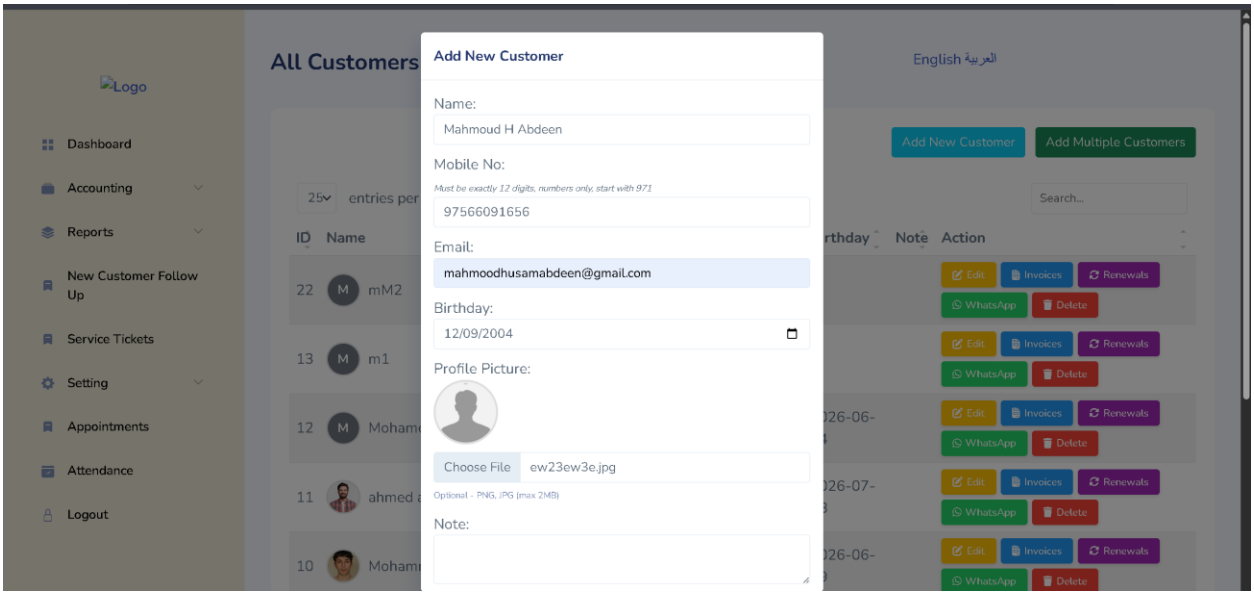

Figure 18 Add New Customer Modal

This is the modal form that is used when creating a new customer record. The form consists of customer name, customer mobile number, customer email address, customer birthday, optional picture of customer and customer notes. The form is then submitted to the admin, the info is validated and sent to the Laravel back-end for storage in the MySQL customers table.

3.5.2.7 AJAX-Based Customer Creation

The Add Customer operation is a FormData / AJAX operation. This is helpful because it contains text information as well as an optional file input.

The data submitted is passed to the /storeCustomer endpoint. The page will not automatically be refreshed during processing. A loading indicator is displayed and the result of the operation is displayed afterwards in SweetAlert2 [10], [11].

```
</> JavaScript

var formData = new FormData();

formData.append('  token', '{{csrf  token() }}');
formData.append('name', $('#name').val());
formData.append('email', $('#email').val());
formData.append('phone', $('#mobile_numner').val());
formData.append('birthday', $('#birthday').val());
formData.append('note', $('#note').val());

if ($('#add_profile_pic')[0].files[0]) {
    formData.append(
        'profile_pic',
        $('#add_profile_pic')[0].files[0]
    );
}

$.ajax({
    url: "{{ url('/storeCustomer') }}",
    type: "POST",
    data: formData,
    processData: false,
    contentType: false,
    dataType: "json",

    success: function (response) {
        if (response.success) {
            Swal.fire({
                icon: 'success',
                title: 'Success!',
                text: response.success,
                timer: 2000,
                showConfirmButton: false
            }).then(function () {
                location.reload();
            });
        }
    }
});
```

Code Snippet 10 Sending Customer Information Through AJAX

The process Data: false option will not cause jQuery to convert the FormData object to a query string. The contentType: false option will make the browser use the proper multipart form-data content type [10].

3.5.2.8 Back-End Customer Creation

The storeCustomer() method gets the information that is submitted. Firstly, it maps the form field names to the names expected by the Customer model.

For instance, the field name on the Blade form is named name and its name in the database is customer_name. These names are combined in the controller prior to validation..

```php
</> PHP
public function storeCustomer(Request $request)
{
  $request->merge([
    'customer_name' => $this->customerName($request),
    'notes' => $this->customerNotes($request),
  ]);
  $validator = Validator::make(
    $request->all(),
    [
      'customer_name' => 'required|string|max:255',
      'phone' => $this->phoneRules(),
      'email' => 'nullable|email|max:255',
      'notes' => 'nullable|string|max:1000',
    ],
    $this->phoneMessage()
  );
  if ($validator->fails()){
    return response()->json([
      'errors' => $validator->errors()
    ], 422);
  }
  $customer = Customer::create([
    'customer_name' => $request->customer_name,
    'phone' => $request->phone,
    'email' => $request->email,
    'notes' => $request->notes,
  ]);
  return response()->json([
    'success' => 'Customer added successfully.',
    'customer' => $customer,
  ]);
}
```

Code Snippet 11 Customer Creation in CustomerController

The method does the following four operations:

1. Changes the form field names.
2. Validates information submitted.
3. Makes a customer record using Eloquent.
4. Makes a JSON success response.

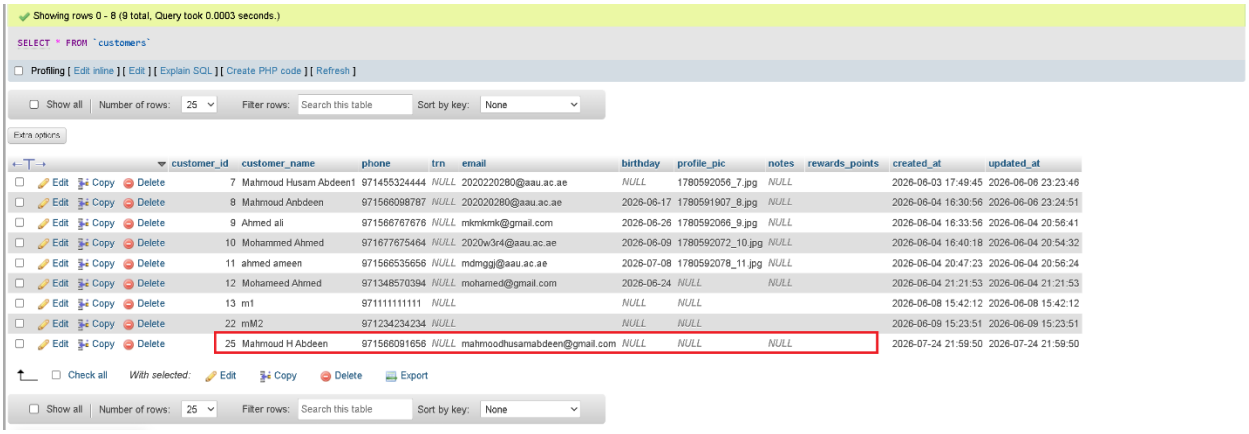

Figure 19 Newly Created Customer Record Stored in phpMyAdmin

This is the number of the newly created customer record that is stored in the customers table in the MySQL database when the Add New Customer form is submitted. The row will include the generated customer id with the customer information (customer name, phone number, email, birthday, notes, profile picture file name.) that was submitted. This is just proof that the information has been successfully transferred from the Customer form to the database via the Laravel controller and Customer model using the Create operation.

3.5.2.9 Customer Input Validation

Customer validation was crucial, as it could impact customer communication, invoicing, reports, and other system processes [2].

Phone-number rules are encapsulated in a private method that can be re-used.

```php
</> PHP

private function phoneRules()
{
  return 'required|string|digits:12|starts_with:971';
}
```

Code Snippet 12 Phone-Number Validation Rules

The validation needs a phone number to:

- Be entered.
- Considered a string.
- Contain exactly 12 digits.
- Start with 971.

```php
</> PHP

private function phoneMessage()
{
  return [
    'customer_name.required'
      => 'Customer name is required.',

    'phone.required'
      => 'Phone number is required.',

    'phone.digits'
      => 'Phone number must be exactly 12 digits.',

    'phone.starts_with'
      => 'Phone number must start with 971.',

    'email.email'
      => 'Please enter a valid email address.',
  ];
}    'success' => 'Customer added successfully.',
    'customer' => $customer,
  ]);
}
```

Code Snippet 13 Custom validation messages were also defined.

Field	Rule	Purpose
Customer name	The length is 255 characters max, required, string.	Does not allow to save customer if no name is entered.
Phone	Required, string, 12 digits in length and only digits	Makes sure that the number is of the correct length.
Phone	Starts with 971	Uses the country-code format in the UAE country.

Email	A valid e-mail address, not longer than 255 characters (optional)	Does not allow invalid email formats.
Notes	The string is optional, has a maximum length of 1000 characters.	Restricts the length of the notes.

Table 23 Verified Back-End Validation Rules

Front-end validation is included in the customer Blade page. The following regular expression is used in JavaScript:

```
</> JavaScript
function isValidPhoneNumber(phone) {
  phone = phone.trim();

  var phonePattern = /^971\d{9}$/;

  return phonePattern.test(phone);
}
```

Code Snippet 14 Front-End Phone Validation

This front-end validation is helpful as it gives immediate feedback, but the Laravel back-end validation will still be in place if the front-end validation is skipped.

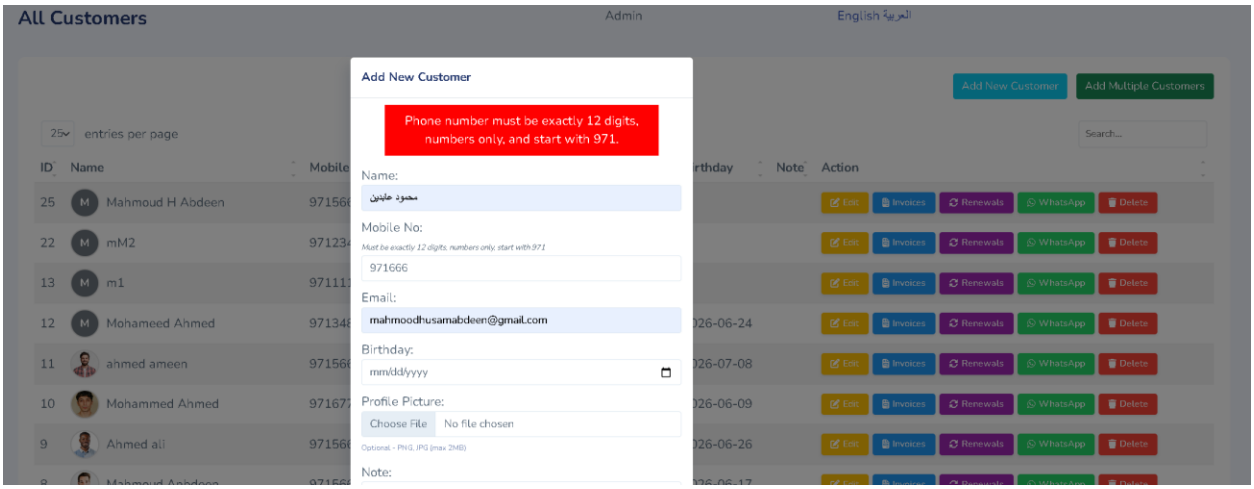

Figure 20 Invalid Phone Number Validation in the Add New Customer Form

Here you can see the validation message when the administrator inputs an invalid mobile number on the Add New Customer form. The system will only accept phone numbers with 12 digits starting with 971. The validation will ensure that no wrong customer information is given and stored in the database..

3.5.2.10 Adding Multiple Customer Records

An Add Multiple Customers modal is present in the customer page. The administrator can add or delete dynamic rows, and type in data for multiple customers in a single action.

Each row contains:

- Customer name
- Phone number
- Email address
- Birthday
- Note

The rows are captured in a customer's array and passed to AJAX via JavaScript.

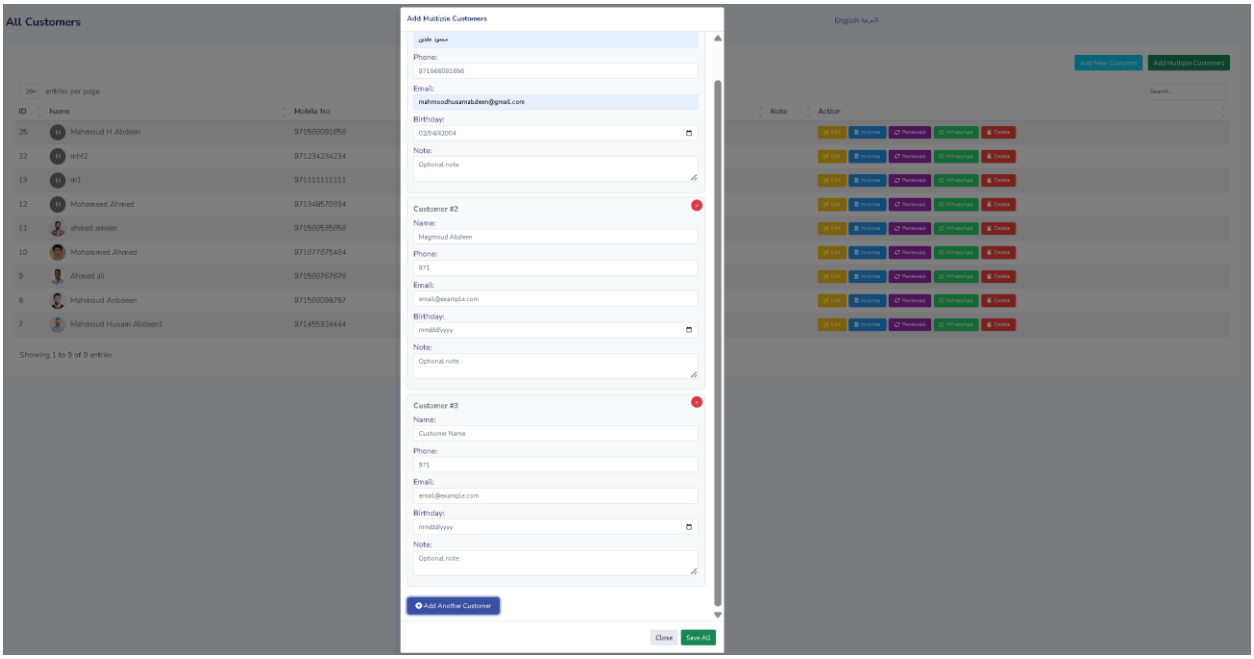

Figure 21 Add Multiple Customers Modal with Dynamic Customer Rows

This figure displays Add Multiple Customers modal that is used to add and save multiple customer records in one go. Dynamic rows include their own customer name, phone number, email address, birthday and optional notes fields. With the help of the Add Another Customer button, the administrator can add rows and with the red delete button remove the rows that are no longer required. Once the details are filled, click on Save All, which sends customer details to the Laravel backend for validation and storing it in the database.

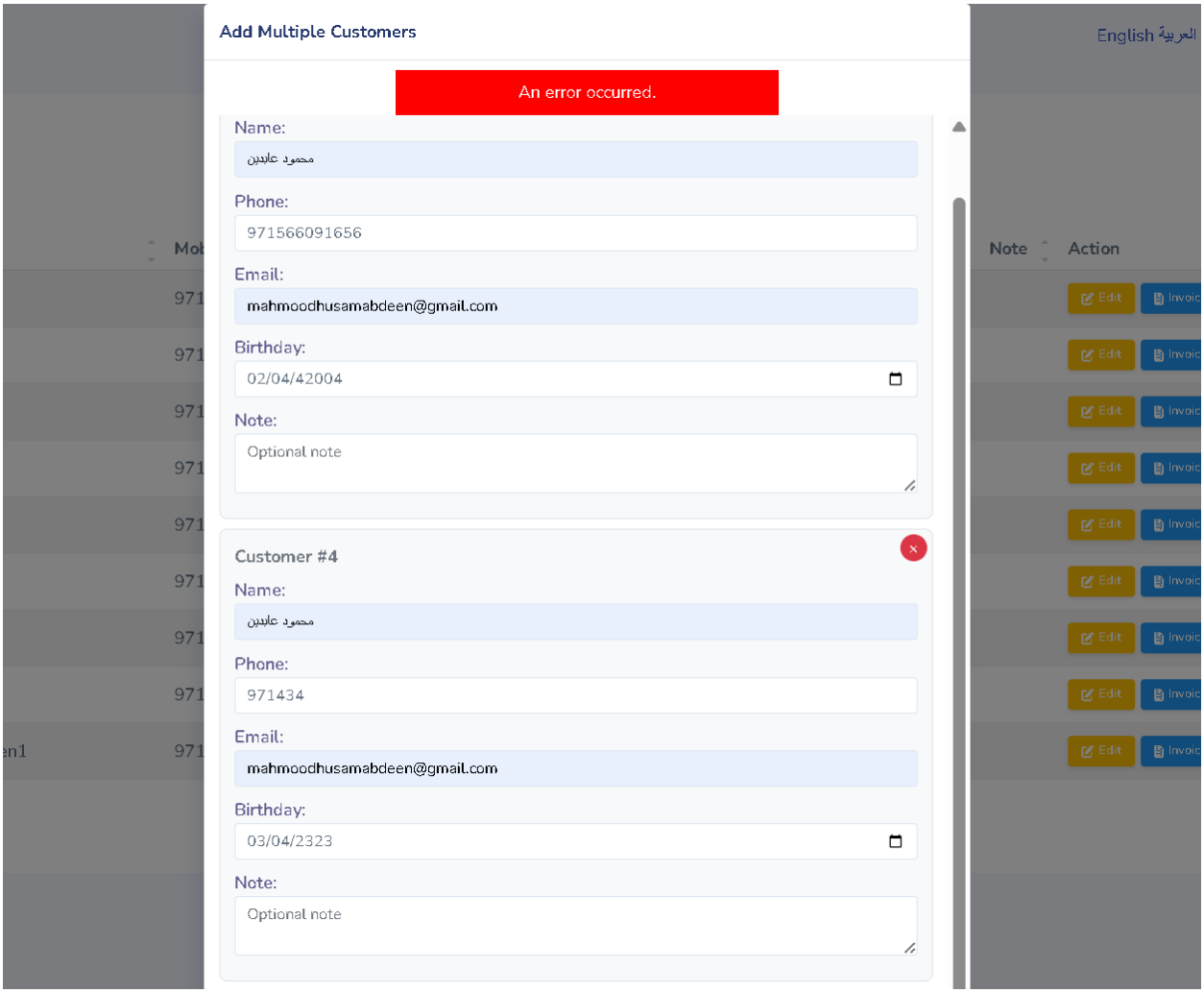

Figure 22 Validation Error During Multiple Customer Submission

This figure illustrates the validation message that appears if the information on one or more records on the Add Multiple Customers form is incorrect. In this instance, Customer #4 has an incorrect phone number because it is missing a digit and is not in the proper format (must start with 971 and have 12 digits). This means that no records are saved in the system until the invalid information is corrected, ensuring accurate and consistent customer data in the system.

3.5.2.11 Read Operation: Displaying Customer Records

To display Customer Records, select 3.5.2.11 Read Operation. The second CRUD operation was retrieving customer information from the database [2], [4].

The index() controller method is used to fetch all customer records, sorting them in the order of customer_id in descending order. This means that the newest customers added are listed first [2], [4].

```php
</> PHP

public function index()
{
  $title = 'All Customers';

  $customers = Customer::orderBy(
    'customer_id',
    'desc'
  )->get();

  return view(
    'admin.customer.customers',
    compact('title', 'customers')
  );
}
```

Code Snippet 15 Retrieving Customer Records

The $customers collection is sent to the Blade page by the controller. The view is using @foreach to produce a row in the table for each customer [2], [10].

Table Column	Information Displayed
ID	Customer primary key
Name	Customer name
Profile	First-letter icon, upload image or default.
Mobile number	Customer phone number
Email	Customer email address
Birthday	Optional birth date
Note	Additional customer information
Action	Edit, Invoices, Renewals, WhatsApp and Delete.

Table 24 Customer Information Displayed

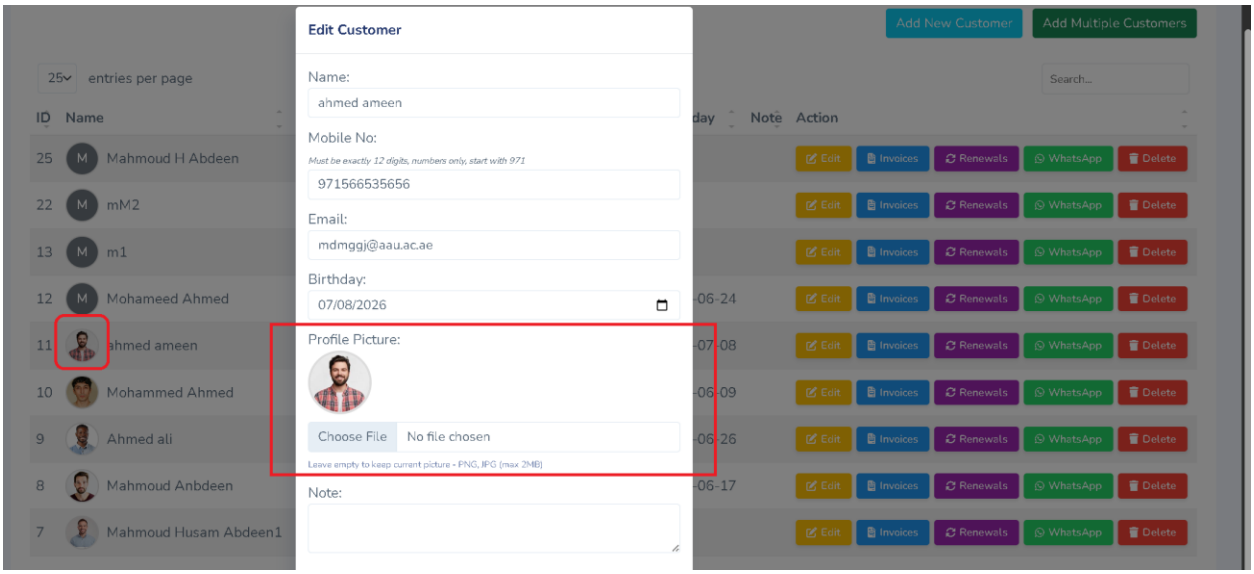

Figure 23 Customer Profile Picture and Default Initial Placeholder

In this figure, it is shown how customer profile information is displayed on the customer-management page. If the customer has an uploaded picture, this will show up next to their name and as a preview in the Edit Customer modal. If customer has no profile picture, then the initial of

their name is used as default image. Also, the administrator can change the image while editing the customer or he/she can untick the file field if they want to keep the existing image.

3.5.2.12 Loading an Individual Customer for Editing

When the administrator opens the Edit Customer modal, information will also need to be read.

The Edit button has the customer ID. Clicking this invokes AJAX to do a GET request of /editcustomer/{id} [2], [10].

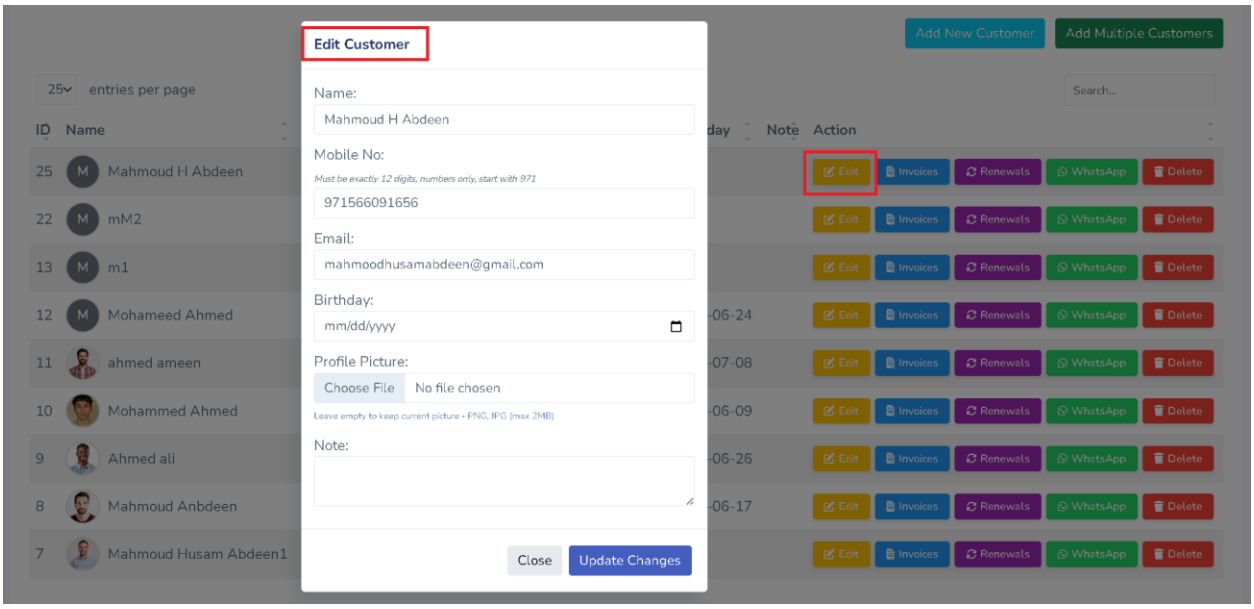

Figure 24 Edit Customer Button and Customer Editing Modal

The Edit button as displayed next to each customer record on the customer-management page is shown in this figure. On click of the button, the selected customer information is retrieved and shown within the Edit Customer modal. The administrator can then edit fields like the customer name, cellphone number, email, birthday, profile pic and notes, before sending them off with the Update changes button.

```php
public function editcustomerinfo($id)
{
  $customer = Customer::where(
    'customer_id',
    $id
  )->first();

  if (!$customer){
    return response()->json([
      'error' => 'Customer not found.'
    ], 404);
  }

  return response()->json([
    'success' => 1,
    'customer' => $customer
  ]);
}
```

Code Snippet 16 Retrieving One Customer

When the requested customer does not exist, the 404 response will prevent the page from continuing.

3.5.2.13 Reading Customer Invoice History

On the customer table there is an Invoices button. It will open a page which will have a listing of all invoices pertaining to the selected customer.

The selected customer_id and the active branch stored in the session are given to the controller where the invoice record is retrieved.

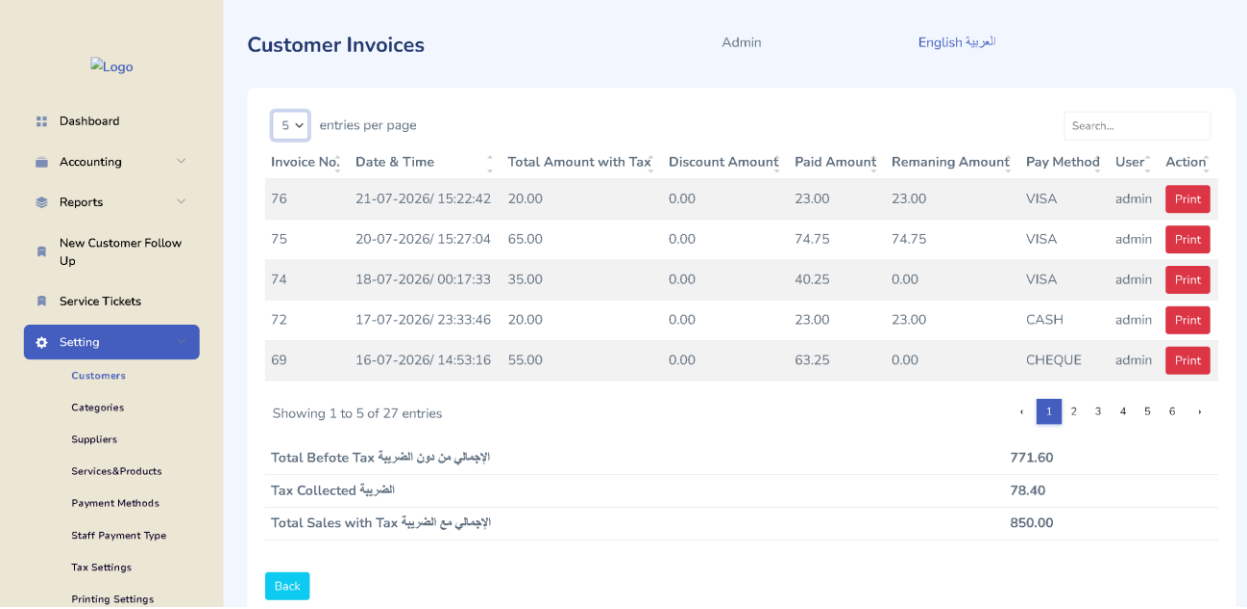

Figure 25 Invoice History for a Selected Customer

This is the invoice history that's being returned for a selected customer. Each related invoice is shown with the invoice number, date and time, total with tax amount, discount amount, paid amount, balance amount, payment method, responsible user and printing action. The total before tax, collected tax and total with tax are also provided in the summary section. Now we have invoice records associated with a customer, we can verify that the system is able to return and display those invoice records with the Customer ID.

3.5.2.14 Update Operation: Editing Customer Information

The third CRUD operation was to update an existing customer.

Once the Edit Customer modal is set with the current record the administrator can change it. The data is then validated using JavaScript and passed on to /updatecustomer/{id} with the new values [10].

3.5.2.15 Updating the Database Record

The updatecustomer() controller method validates the information and looks up the customer by ID [2].

Once it has found the record it alters the properties of the model and triggers save() [2].

Step	Description
1	Administrator clicks the Edit button, which opens the Edit modal.
2	Customer data is retrieved from existing data.
3	Administrator edits one or more fields.
4	The values submitted are checked by JavaScript.
5	An update request is sent by AJAX.
6	The information is validated using Laravel.
7	Customer location is determined by the controller.
8	Properties of the model are modified.
9	save() will update database.
10	Success message is shown using SweetAlert2.

Table 25 Update Operation Process

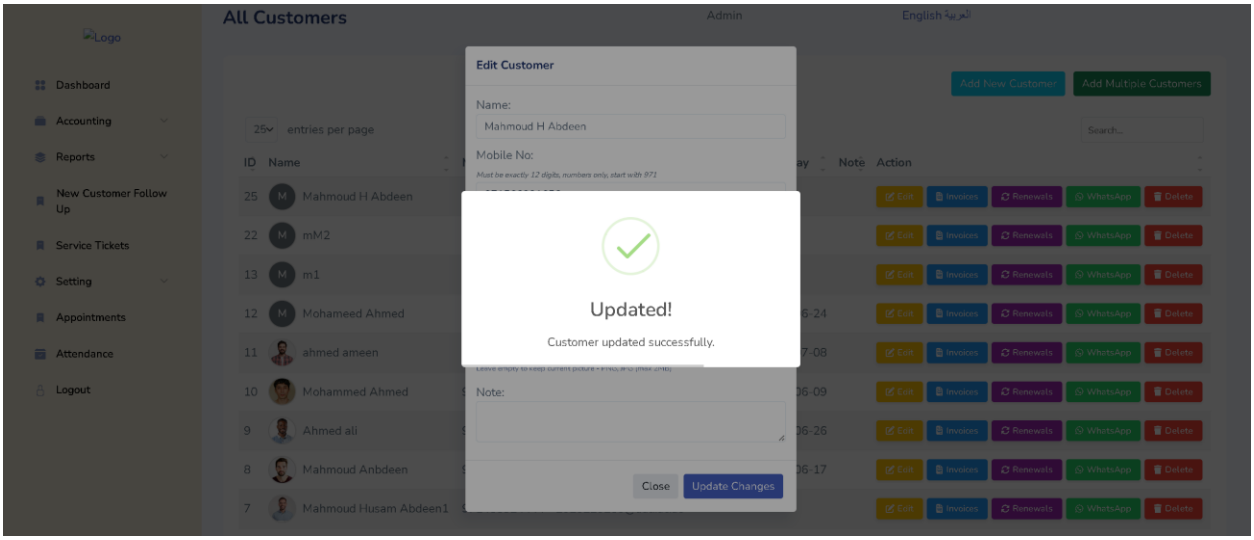

Figure 26 Successful Customer Update Confirmation Using SweetAlert2

This is the success message when the administrator updates an existing customer record. After the modification of the information, the Laravel controller responds with a successful JSON response, once the information is validated and saved in the MySQL database. An alert message “Customer updated successfully” is then shown on SweetAlert2 indicating that the Update operation is done correctly [2], [4], [10], [11].

3.5.2.16 Delete Operation: Removing Customer Records

The four CRUD operations included in this example was deletion of customer records.

Each customer will have a Delete button. The button contains the customer ID and name for selecting the correct customer record and for the confirmation message to identify the customer.

SweetAlert2 prompts the administrator to confirm or cancel deletion prior to request [11].

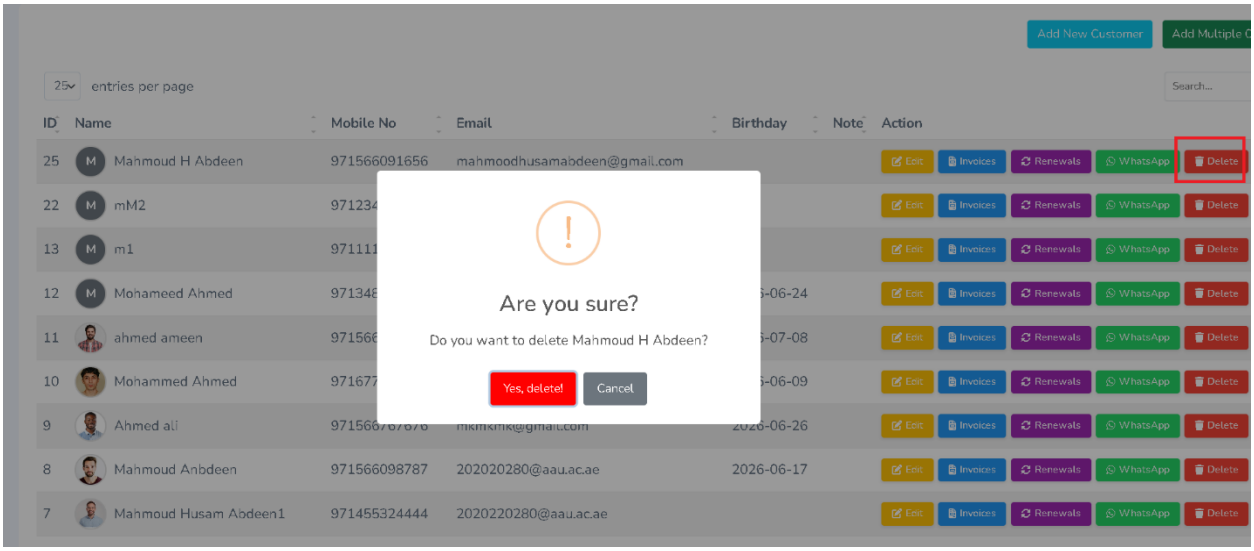

Figure 27 Delete Customer Confirmation Dialog Using SweetAlert2

This is a SweetAlert2 confirmation dialog that appears before deleting a customer. The message will display the selected customer, and will prompt the administrator to confirm the operation. Clicking on Yes, delete! will continue the deletion request, clicking on Cancel will cancel the dialog and revert to a previous state without changes. This confirmation process is to prevent accidental deletion of customer records [11]

3.5.2.17 Safe Back-End Customer Deletion

The deletion method is important with a big protection. When a customer record is deleted, it verifies if the customer has invoice records [2], [4].

If an invoice is associated to the customer, the deletion is aborted and an error message is displayed to the administrator. This helps maintain referential integrity and stops disengaging invoices from their customer [4].

```php
</> PHP

public function deletecustomer($id)
{
    $customer = Customer::where(
        'customer_id',
        $id
    )->first();
    if (!$customer) {
        return response()->json([
            'error' => 'Customer not found.'
        ], 404);
    }
    $invoiceCount = Invoce::where(
        'customer_id',
        $id
    )->count();
    if ($invoiceCount > 0) {
        return response()->json([
            'error'
                => 'Cannot delete customer. This customer has '
                . $invoiceCount
                . ' invoice(s) linked. Delete the invoices first.'
        ], 422);
    }
    try {
        $customer->delete();

        return response()->json([
            'success'
                => 'Customer deleted successfully.'
        ]);
    } catch (\Exception $e) {
        return response()->json([
            'error'
                => 'Cannot delete customer. It is linked to other records.'
        ], 422);
    }
}
```

Code Snippet 17 Customer Deletion with Invoice Check

Check	Purpose
Customer exists	Does not allow the deletion of an invalid ID.
The number of invoices is 0	Does not allow a customer to be deleted if it is associated with invoice(s).
Confirmation dialog	Avoids unintended user actions.
CSRF token	Suppresses the delete request.
Exception handling	Safely deals with other database relationships.
JSON error response	Provides reasons for a failure in the operation.

Table 26 Delete Operation Safety Checks

3.5.2.18 Using SweetAlert2 During CRUD Operations

In order to better communicate with the system and the administrator, SweetAlert2 was used[11].

The loading function shows a pop-up when the system is working on an operation [11].

```
</> JavaScript

function showSwalLoading() {
  Swal.fire({
    allowOutsideClick: false,
    showConfirmButton: false,

    didOpen: function () {
      Swal.showLoading();
    }
  });
}
```

Code Snippet 18 SweetAlert2 Loading Function

Message Type	Use
Loading	Indicates that a request is being processed.
Success	Confirms that a customer was added or updated.
Delete confirmation	Asks permission before deleting a customer.
Deleted message	Confirms successful deletion
Validation error	Describes information that is not valid.
Database error	Describes the reason that an operation was not able to be completed.

Table 27 SweetAlert2 Uses

3.5.2.19 Database Extensions for Customer Records

During the inspection, two migration files were discovered for the customer CRUD interface [2]:

- Birthday migration
- Profile-picture migration

```php
</> PHP

public function up()
{
   Schema::table(
      'customers',
      function (Blueprint $table) {

         $table->date('birthday')
            ->nullable()
            ->after('email');
      }
   );
}

public function down()
{
   Schema::table(
      'customers',
      function (Blueprint $table) {
         $table->dropColumn('birthday');
      }
   );
}
```

Code Snippet 19 Birthday Migration

There is a possibility that the customer is created without having a birthday [2], [4].

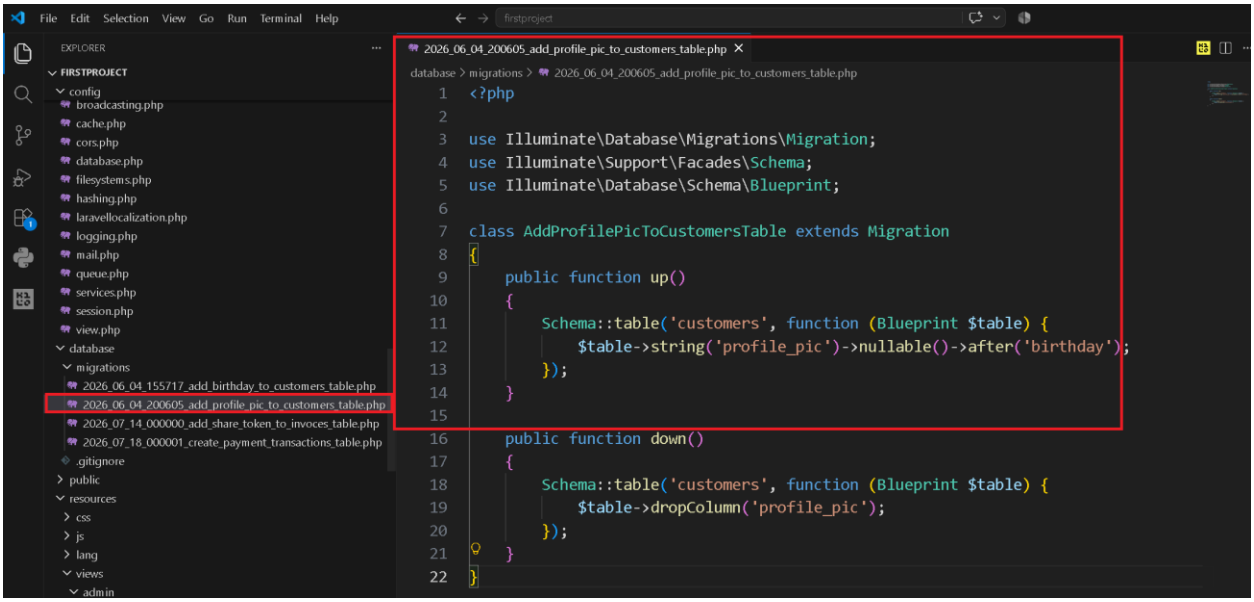

Figure 28 Migration File Used to Add the Customer Profile-Picture Field

You can see the migration file of Laravel 2026_06_04_200605_add_profile_pic_to_customers_table.php in the image. up() adds a column profile_pic to the customers table, which can be null. This column contains the name of the customer image that was uploaded. If the migration is rolled back, then the column is removed by the down() method [2], [4].

3.5.2.20 Implementation Verification Note

Task documentation includes birthday saving, uploading profile pictures, validating a duplicated phone number, giving day information, and creation of multiple customers as complete customer-management features.

The inspected snapshot of firstproject(26).zip, however, has a number of differences, which should be addressed prior to the report stating that all features are fully connected.

3.5.2.21 CRUD Testing

Each operation should be tested individually after implementing and reviewing the customer functions.

Test Case	Expected Result
Input proper Customer Data	Created Customer successfully.
Customer Name: Not specified	Required Name (type mismatch) is shown.
Type in phone number excluding 971	An error message appears that states that the information is not valid.
Write less than 12 digits	A validation error is shown.
Please input over 12 digits	The input is restricted or not accepted.
Fill in letters in telephone field	The letters are removed or rejected.
Enter invalid email	An error message is shown when you try to use email.
Optional e-mail field is left blank	Though still, the customer can be created.
Provided an incorrect customer ID	A proper error response is provided.
Add a number of valid customers	Valid rows are added.
Wiring in an illegal multiple-customer row.	The corresponding row is marked with a highlighter.

Table 28 Create Operation Testing

Test Case	Expected Result
Click Open All Customers page.	Customer records will be shown.
With the exception of page, compare the other pages with phpMyAdmin.	The records shown reflect the records in the database.
Click Edit	The correct customer information is loaded.
Request a missing customer ID	404 responses are returned.
Customer has a picture for their profile.	The user's profile picture appears.
No picture of Customer.	A first letter place holder appears.
Click Invoices	The related customer invoices are displayed.

Table 29 Read Operation Testing

Test Case	Expected Result
Change customer name	The new name is listed in the table.
Change phone number	A new telephone number is saved.
Enter invalid phone	This update is refused.
Enter invalid email	Rejection of the update.
Update notes	The new notes are shown.
Request missing customer	A 404 response is returned.
Don't lose any data already available.	The record is still valid.

Table 30 Update Operation Testing

Test Case	Expected Result
Click Delete	The following dialog will appear to ask for confirmation.
Click Cancel	No record is removed
Confirm the customer without any invoices.	The record is deleted.
The Customer is confirmed using Invoices.	The deletion is not allowed.
Remove customer ID not found in the list	A 404 response is returned.
After removing, check in phpMyAdmin.	There is no longer a record.
Deletion is prevented due to database relationship	Controlled error message is displayed

Table 31 Delete Operation Testing

3.5.2.22 Testing Results

CRUD Operation	Result
Create	Basic customer data can be verified and inputted via AJAX.
Read	Customer records are loaded, shown and loaded in the Edit modal.
Update	The Customer name, phone, email and notes can be changed.
Delete	If a customer does not have any linked invoices, they can be deleted without any worries.
Validation	Names, phone formats and email formats which are not valid are rejected.
User feedback	Clear processing and result messages are shown with SweetAlert2.
Related records	Customer Invoices can be accessed.
Database verification	Records can be accessed using phpMyAdmin.
Optional UI fields	There are birthday and profile-picture controls.
Additional completion	There are some optional fields that need to be synced in the snapshot being inspected.

Table 32 CRUD Results

3.5.2.23 Challenges Faced and Solutions Applied

Challenge Faced	Solution Applied
It was impossible to submit the full page as normal form.	Implemented AJAX to submit customer data in the background.
Note that invalid phone formats might be entered	Added front-end and Laravel Phone validation.
Customer name may be left as is	Added required-name validation
Invalid email addresses could be entered	Added optional email-format validation
Some data was required prior to editing.	Fetched the selected customer using AJAX GET.
Not all fields are required, and thus may be left empty.	Implemented Nullable Fields and safe Blade output.
Undefined array-key errors were reported due to missing values.	Used the ?? The customer table contains a column called " operator.
Users were able to accidentally delete records.	Improved with a SweetAlert2 confirmation dialog.
It would be possible to reference customers to the invoice.	Verified the number of invoices that were being deleted.
Customer IDs may not be included leading to errors	Used the ?? The customer table contains a column called " operator.
Deletions may be denied due to database relationships.	Added exception handling

The number of rows validated required separate validation.	Returned row-specific errors
Profile pictures may not be present	Showed the customer-name initial when one is not available.
The new Database fields required complete integration.	Looked at migration, model, controller, form and DB connection.
Customer files were not fully synchronized	Identified missing route and persistence logic during code review

Table 33 Task 2 Challenges and Solutions

3.5.2.24 Skills and Knowledge Developed

Skill	**Learning Outcome**
CRUD implementation	Learned how to perform Create, Read, Update and Delete operations.
Eloquent ORM	Used models to manage database records.
AJAX	Made requests in the background without a full page refresh.
Form Data	Text and file input submitted.
Laravel validation	Checked information before saving –
JSON responses	Returned success and error information.
Blade templates	Displayed dynamic customer records.
Modal forms	Made additions and edits to the same page.
SweetAlert2	Showed professional feedback from users.
Database verification	Compared application data with phpMyAdmin.
Error handling	Handled validation, missing records and database relationships.
Data integrity	Blocked deletion of customer associated to Invoices.
Debugging	Followed customer functions throughout the entire Laravel workflow.

Table 34 Skills Developed During Task 2

3.5.2.25 Task Summary

In this task, I got to experience the full CRUD operations in the Customer Management section of the Laravel application.

In the Create operation I analyzed and adopted the customer form submission function using AJAX. Customer information was first checked in the Laravel model, and then added to the MySQL database via the Customer model. I also peeked at the dynamic interface for multiple customers and their row-based validation [2], [4], [10].

On the Read, I got all the customer records and then displayed them using a blade table. I've also populated the information for individual customers via AJAX into an Edit modal and looked at the customer invoice-history [2], [10].

With the Update action, I was passing updated customer data via AJAX, verifying the changes, finding the right record in the database and saving those changes back with Eloquent [2], [4], [10].

I added a SweetAlert2 confirmation dialog for the Delete operation and reviewed the deletion logic of the controller. When invoice records are linked to the customer, the controller will validate the existence of the customer and not allow deletion of the customer record [2], [4], [11].

Validation rules, JSON responses, CSRF protection and modal forms, optional fields in the database, migrations, displaying profile pictures and SweetAlert2 messages have been involved with as well [2], [11].

3.5.2.26 Conclusion

This activity helped me get experience in performing the most critical database operations that are utilized in Laravel web applications. I find out that CRUD functionality can not be done in just one file. All the operations depend on the interaction between Blade view, JavaScript, AJAX, route definition, controller methods, validation rules, Eloquent model and MySQL database [2], [4], [10].

Another thing I learned is the need to validate the information at both the front end and back end. Front end validation enhances the user experience and Laravel validation guarantees that the invalid requests don't skip the app's rules [2], [10].

The delete function highlighted the need for the integrity of associated records in a database to be protected. This ensured the data integrity and prevented incomplete invoice information getting lost due to the deletion of customers with invoices [4].

Another thing that the code review highlighted is the need to maintain all the project files in sync. A field in a database is not usable if it's defined in a migration or form. It also needs to be part of the model, validation, controller save logic, update logic and the test process [2], [4].

The end result of this task was an increased awareness of customer-data management, Laravel Eloquent operations, AJAX communication, form validation, safe record deletion, database relationships, testing and debugging. It gave me the ability to dive into more complex customer-dashboard, invoice, reporting and payment functions for the tasks I was given the following internship.

3.5.3 Task 3: Improve the Customer Dashboard and Customer Record Management

Task Description

Work on the customer dashboard by improving the display of customer information and supporting basic functions such as adding, editing, viewing, and managing customer records.

3.5.3.1 Introduction

In this task, I focused on developing the Customer Dashboard to make it more user-friendly and streamlined for customer management. The most important feature that was worked on was the customer table itself, that is, customer profile pictures, redesigning the action buttons, enhancing the Add Customer and Edit Customer forms and creating an option for multiple-customer entry with a link to all the associated features like invoice, renewals, WhatsApp communication, editing and deletion of customer.

The aim of these enhancements was to make the customer page more than just a table of customers, in order to make it a dedicated customer-management workspace. The administrator can now view the customer information, view the customers and open related records, contact the customers and conduct management actions all on the one page.

So prior to the changes, I looked at how the customer records were pulled from the MySQL database via CustomerController.php, and then passed to customers.blade.php. I also looked at customer routes, Eloquent model, database migrations, Bootstrap modals, JS functions, AJAX requests, and other customer pages.

Objective	Description
Enhance customer information presentation	Display customer data in a table that is easy to read and understand.
Include a visual ID of customers	Show profile pictures or default initial avatars.
Improve customer forms	Fill in full customer details using Bootstrap modals.
Support customer editing	Add/Update customer, no page flipping.
Improve dashboard actions	Make Edit, Invoices, Renewals, WhatsApp and Delete buttons visible.
Support multiple-customer entry	Prepare a number of customer records in one modal.
Improve user feedback	Show validation, loading, success and error message.
Connect related records	Open Invoices and Renewals of a particular Customer.
Improve responsive behavior	Don't allow buttons/content to overflow a table.
Maintain record safety	Verify destructive operations and preserve data.

Table 35 Main Objectives of Task 3

3.5.3.2 Reviewing the Existing Customer Dashboard

The first step was to look at the existing customer-management page and see how the customer information was retrieved and displayed [2], [4].

This is the general flow of the customer dashboard:

Customer Model → CustomerController → Customer Blade View → Customer Table [2], [4]

The index() method fetches all the customers from the database and sorts them in descending order of customer_id. The records are then passed to the customer Blade page [2], [4].

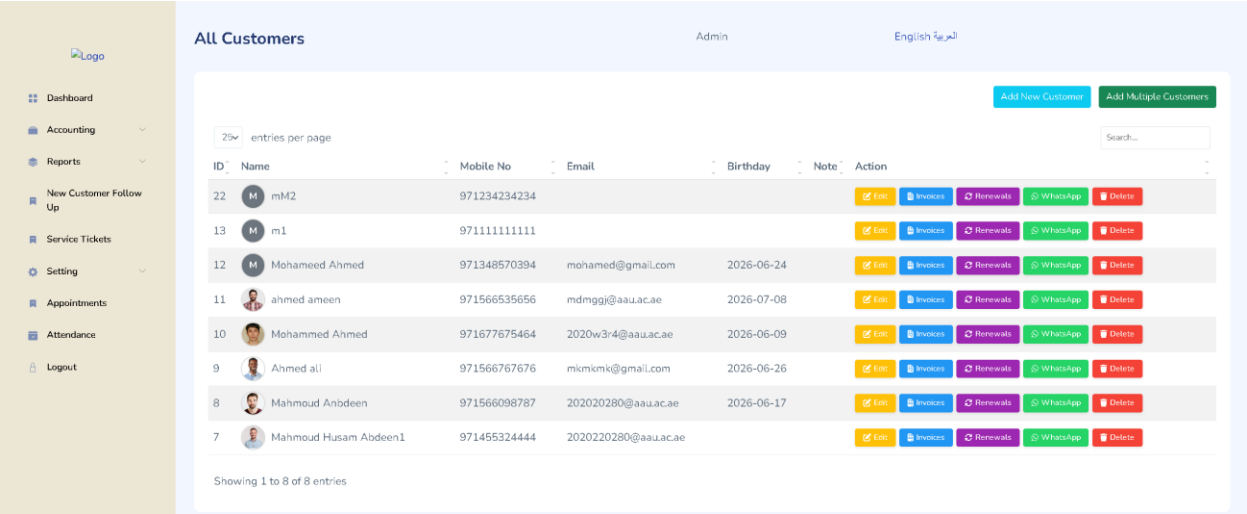

Figure 29 Complete Customer Management Dashboard

This is the entire Customer Management Dashboard with the interface enhancements. Customer records are shown in a well-structured table, with customer ID, name, profile picture/avatar, phone number, email, birthday, notes and action that can be performed on them. The dashboard also contains the controls to add one customer, add multiple customers, search for existing customers, and to change the number of customers to show on the dashboard page. Direct actions are provided

in each Customer row for editing, viewing your Invoices, viewing your renewals, opening WhatsApp communication and deletion.

3.5.3.3 Improving the Customer Information Table

This section encourages you to enhance the Customer Information Table. I made some changes to the customer table, to enable important information to be displayed on one page. One customer appears in a row and both customer data and management actions are included in the table.

Customer Information	**Description**
Customer ID	Unique identification number
Customer name	Customer Record Name.
Profile image	A picture that is uploaded or a default picture as an initial avatar.
Mobile number	Customer contact number
Email address	Optional customer email
Birthday	Customer date of birth (optional).
Notes	Additional customer information
Actions	Edit, Invoices, Renewals, WhatsApp, Delete.

Table 36 Customer Information Displayed on the Dashboard

The customer name and the customer's avatar are placed in the same table cell. This makes it easier to visualize customers while maintaining the written customer name.

Optional values are also safely output in the Blade code. The ?? The " operator is used to avoid mistakes when the customer field is blank [2], [3].

```
</> blade

<td>{{ $customer['phone'] ?? '' }}</td>
<td>{{ $customer['email'] ?? '' }}</td>
<td>{{ $customer['birthday'] ?? '' }}</td>
<td>{{ $customer['notes'] ?? '' }}</td>
```

Code Snippet 20 Displaying Customer Information Safely

This code displays an empty value when optional customer information is unavailable instead of producing an undefined-key error.

3.5.3.4 Adding Customer Profile Pictures

A major Customer Dashboard enhancement was the addition of support for profile pictures. This capability was created to enable the administrator to attach a customer picture to every record.

File Input added to Add Customer and Edit Customer forms. PNG, JPEG and JPG images are accepted.

Component	Change
Database migration	Introduced a new column profile_pic that can be null.
Add Customer form	Added input for a profile-picture.
Edit Customer form	Added an optional replacement-image input.
Customer table	Displays the customer image.
Default avatar	Shows initial of the first name if an image is not available.
Image preview	Displays the chosen image prior to submission.
Edit preview	Displays the current customer image.

AJAX submission	Uses Form Data to include the selected file.

Table 37 Profile-Picture Feature Components

3.5.3.5 Profile-Picture Preview in the Customer Forms

The image-preview function implemented to enable the administrator to check the selected image before going to the Add or Edit Customer form.

The FileReader object is used to read the image locally and show it within the modal in JavaScript.

```
</> blade

$('#add_profile_pic').on('change', function () {
  var file = this.files[0];
  if (file) {
    var reader = new FileReader();
    reader.onload = function (e) {
      $('#add_preview_container').html(
        '<img src="' +
        e.target.result +
        '" class="profile-preview" ' +
        'alt="Profile preview">'
      );
    };
    reader.readAsDataURL(file);
  }
});
```

Code Snippet 21 Profile-Picture Preview

This preview will not save the image as soon as it is viewed. It will give the administrator the opportunity to check that the right file was selected before submitting the form.

3.5.3.6 Improving the Add and Edit Customer Modals

Task 2 explained in detail the Create and Update processes. The primary enhancement made in this task was to make these functions part of a neat Bootstrap modal [8].

The Add Customer modal includes the following fields: Name, Mobile Number, Email, Birthday, Profile Picture, Notes. The Edit Customer modal will show the existing data of the selected customer.

The Edit modal will even show the present customer picture if readily available. This enables the administrator to view what image is associated with the customer prior to choosing an alternative image.

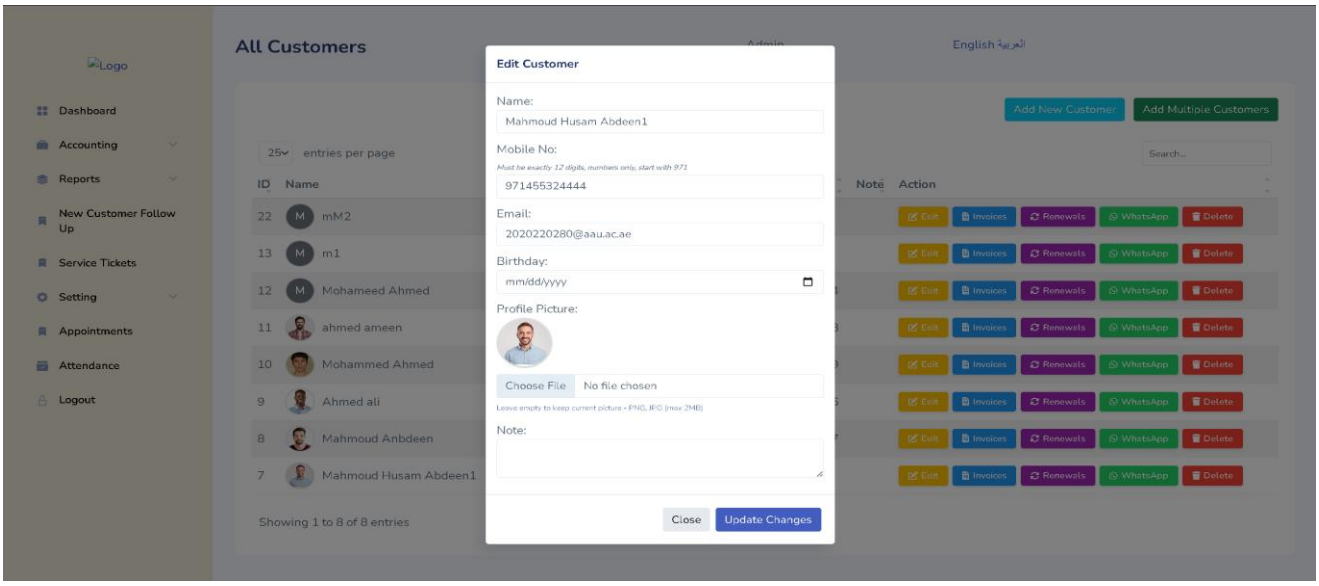

Figure 30 Customer Information Loaded into the Edit Customer Modal

This is the Edit Customer modal with the selected customer information retrieved and displayed. The current customer name, mobile number, email address, birthday, image, and Customer notes are prefilled in the form fields for the administrator to review and add or edit the required information without even leaving the Customer Dashboard [8], [10].

3.5.3.7 Redesigning the Customer Action Buttons

Redesigned Customer action buttons for better clarity and usability. A different colour and icon was used for each of the functions.

Button	Colour	Purpose
Edit	Yellow	Opens the customer information to be edited.
Invoices	Blue	Shows the invoices of the customer.
Renewals	Purple	Displays service-renewal records
WhatsApp	Green	Initiates direct communication with customers.
Delete	Red	Starts the protected deletion process.

Table 38 Customer Action Buttons

Different colors can help the administrator to quickly differentiate between various actions. The addition of Font Awesome icons was used to provide an additional visual indication of each function [12].

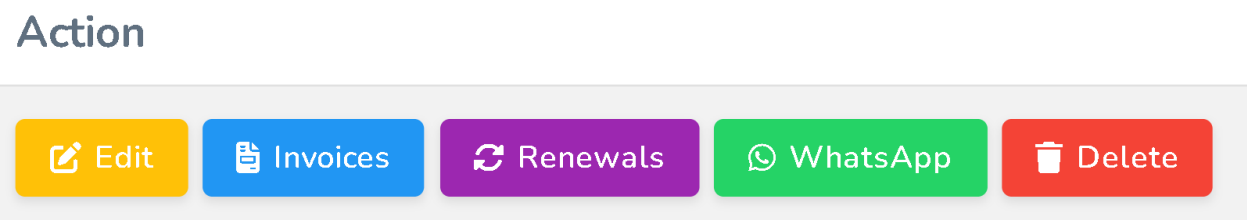

Figure 31 Redesigned Customer Action Buttons

This is the updated version of the action buttons in the Customer Dashboard. Each button has a different function with respect to customer-management, e.g. Edit, Invoices, Renewals, WhatsApp, Delete. Each action was represented by a different colour and icon for easy identification, page appearance and for easy and quick execution of customer related actions by the administrator.

3.5.3.8 Responsive Organization of Action Buttons

There were multiple buttons in the customer action area, needing a good amount of horizontal space. The buttons were put into a flexible container so they wouldn't interrupt the table layout.

```
</> CSS
.action-buttons {
  display: flex;
  gap: 6px;
  align-items: center;
  flex-wrap: wrap;
}
```

Code Snippet 22 Responsive Action Button Container

The flex-wrap property lets buttons wrap to the next line if there isn't enough space horizontally.

3.5.3.9 Quick Access to Related Customer Functions

The enhanced dashboard offers a direct access to customer related information.

The detailed retrieval logic was discussed if it is needed in Task 2. Clearly navigating the dashboard was the focus of Task 3.

Function	**Dashboard Benefit**
Invoice History	Opens the financial information for the selected customer.
Renewals	Shows service-expiry/renewal data.
WhatsApp	Allows to initiate a direct call with the saved phone number.
Edit	Opens customer information on the page.
Delete	Offers a safe customer-management activity.

Table 39 Related Customer Functions

```
</>blade

<a
  class="btn-action-whatsapp"
  href="https://wa.me/{{ $customer['phone'] ?? '' }}"
  target="_blank">

  <i class="fab fa-whatsapp"></i>
  WhatsApp
</a>
```

Code Snippet 23 Direct WhatsApp Communication

The customer cellphone number is put inside a wa.me link. Customers are denominated with the international prefix 971, which means that they can be used directly in the WhatsApp URL.

3.5.3.10Add Multiple Customers Interface

Multiple-customer creation detailed processing, validation and AJAX workflow was already described under Task 2. For this task the emphasis was on the dashboard interface.

The Add Multiple Customers modal enables the admin to:

- Create multiple customer records in the same form.
- Include additional customer rows as needed.
- Remove unnecessary rows.
- Review separate fields for each customer.
- Use Save All to submit the completed records.

3.5.3.11 Challenges Faced and Solutions Applied

Here only the challenges related to dashboard display and usability are included. Task 2: already covered validation, AJAX errors and deletion safety.

Challenge Faced	**Solution Applied**
Customer data was predominantly presented in textual format	Incorporated profile pictures next to customers names.
Several customers didn't have profile pictures	Showed first letter of customer's name.
Different picture sizes of the images were uploaded.	Utilized fixed round measurement and object-fit: cover.
The image selected could not be read.	Improved adding a JavaScript image preview.
The image during editing was not clear.	Presented it in the Edit Customer modal.

Not able to see the image clearly while editing.	Incorporated additional colors, icons and labels.
It was hard to differentiate actions of customers.	Implementing a Flexbox container with wrapping.
There was not enough horizontal space for the buttons.	Implemented the Flexbox container with wrapping.
Several navigation steps to related customer information was needed.	Incorporated new Invoices and Renewals buttons.
Manual searches to communicate with customers were required.	Introduced a direct button for WhatsApp.
One customer was added at a time and that wasn't efficient.	Implemented Multiple Customer modal.
The number of customer rows was not determined	Implemented addition and removal of dynamic controls.
Poorly designed customer information to be scanned	Structured it in uniform table column format.

Table 40 Task 3 Challenges and Solutions

3.5.3.12 Skills and Knowledge Developed

Skill	Learning Outcome
Customer-dashboard design	Structured data into full data management interface.
Laravel Blade	Showed the conditionally displayed information for customers.
CSS	Custom avatars, buttons, previews and layouts.
Bootstrap modals	Presented forms without opening of separate pages.
JavaScript FileReader	Viewed image files.
Conditional display	Used images or default initials (U or I).
Flexbox	Organized buttons responsively
Font Awesome	Made addition of understandable action icons.
User-interface design	Enhancements to colour, spacing and feedback.
Related-page navigation	Linked customers with the invoices and renewals.
WhatsApp linking	Imported phone numbers from a list and started conversations with those numbers.
Dynamic forms	Allows for variable number of rows entered by customers.
Interface testing	Tested and confirmed presentation, navigation and responsiveness.

Table 41 Skills Developed During Task 3

3.5.3.13 Task Summary

In this task, worked on the Customer Dashboard to enhance the look and feel of the dashboard and make it more user friendly. Structured my Customer information in a table and added a picture or default first letter representation of the Customer. I also put images inputs and JavaScript images previews on the add and edit customer forms.

The Edit Customer function is to pass selected customer via AJAX and show the current customer details in a Bootstrap modal. This enables the administrator to view and modify customer data without having to open up a different page [8], [10].

Redesigned customer action buttons with clear colors, icons, spacing, shadows & hover effects. The buttons link to the editing, invoice history, renewal history, WhatsApp communication and deletion of the customer record [12].

I have also helped in working with Add Multiple Customers. This modal include dynamic rows which can be added or removed. There are specific customer fields for each row, and the system checks the data that is being entered before passing on the customer array to the Laravel back-end [2], [10].

Customer records related to the transaction were made available in a convenient manner. The Invoices button will open the selected customer's list of invoices and the Renewals button will show the customer's service-expiry records. This WhatsApp button will open up a direct WhatsApp conversation with the stored phone number. These enhancements resulted in a more organized, visual, responsive and helpful customer dashboard for day-to-day customer-management tasks.

3.5.3.14 Conclusion

This helped me to get a better understanding of how the front end design and back end functionality work in a Laravel customer-management system.

I found that it is not only important to show information from the database on a professional dashboard. It should also be easy to comprehend, easy to follow, have related workflows, be able to deal with optional data, be responsive to screen size, and provide immediate feedback to the user.

The function profile-picture and default-avatar resulted in a better identification of the customer. The cleaned up action buttons made the navigation and use of the device easier. Bootstraps modals enabled a customer's data to be added and edited on a single page. The multiple-customer form enabled to prepare several records in one operation, which was more efficient. The invoice, renewal, WhatsApp, and deletion actions have turned the customer table into a full management workspace and not just a simple data display table [8].

In sum, I had the opportunity to build on skills in Laravel Blade, CSS, Bootstrap, JavaScript, AJAX, dynamic forms, responsive layouts, file previews, database migrations, related-record retrieval, validation, and testing the user-interface [2], [8], [10].

3.5.4 Task 4: Front-End Development Using CSS, Bootstrap, Tailwind CSS

Task Description

Assist in front-end development using pure CSS, Bootstrap, Tailwind CSS, and other front-end libraries to improve page design, layout, buttons, forms, and user-interface responsiveness.

3.5.4.1 Introduction

In accomplishing this task, I worked on developing and applying both front-end skills which enhanced the appearance, organization, usability and responsiveness of web pages. This work comprised of changes to the Laravel invoice and authentication interfaces as well as a set of separate practice pages to explore Bootstrap, Tailwind CSS, JS, HTML forms, dashboard layouts, and responsiveness [8], [9].

Mostly the Bootstrap, custom CSS, Bootstrap Icons, Font Awesome, SweetAlert2, JavaScript, and Laravel Blade were used in the Laravel project [2], [8], [11], [12].

Tailwind CSS was explored and used on individual HTML practice pages, such as a responsive login page, dashboard statistic cards, a customer table and a multi page website that is connected [9].

3.5.4.2 Task Objectives

The primary goal was to gain an understanding of the interworking of the front-end technologies used to create a clear, polished and responsive UI.

Objective	**Description**
Improve page design	Utilize the use of colour, spacing, shadows, borders and alignment effectively.

Improve page layout	Use containers, rows, columns, Flexbox and Grid for organizing content.
Improve buttons	Use clear colors, sizes, icons, hover and labels.
Improve forms	Develop well-structured input boxes, labels, error messages and actions.
Improve user feedback	Implement Alerts, Icons, Loading indicators and confirmation messages.
Study Bootstrap	Apply to ready generated components, responsive layout classes.
Study Tailwind CSS	Create unique designs from utility classes.
Improve responsiveness	Make sure pages can be used on small, medium and large screens.
Practice JavaScript interaction	Manipulate the elements of a page without making a separate page.
Explore templates	Know how to see and use the structure of professional administration dashboards.
Build connected pages	Develop and connect Home Page, Login Page, Registration Page, Dashboard Page and Customer Page.
Test interfaces	Test their appearance, interaction, navigation and responsiveness.

Table 42 Main Objectives of Task 4

3.5.4.3 Front-End Technologies Used

Lots of front end technologies were utilized within the Laravel project or on its own practice pages[8], [9], [10], [11], [12].

Technology	**Purpose**
HTML	Developed page structure, pages forms, tables, navigation, cards, buttons and links.
Pure CSS	Personalized color, spacing, background, borders, shadows, hover effects and layout.
Bootstrap 5	Supplied ready-made forms, buttons, cards, tables and alert and modal boxes and responsive classes.
Tailwind CSS	Designed and created customized designs using utility classes.
JavaScript	Manage the visibility of forms and other interactions with pages.
jQuery	Simplified event handling and operations on the front-end of the Laravel project.
Bootstrap Icons	Incorporated elements such as icons in input fields, alert boxes, buttons and navigation.
Font Awesome	Made additions to invoice, What's App, receipt, edit, print and delete icons.
SweetAlert2	Shows loading, successful, warning, confirmation and error messages.

Mazer Admin Template	Showed examples of how to set up a dashboard as a professional.
Laravel Blade	Brought front-end component together with dynamic Laravel information.
Browser Developer Tools	Tested out the layout, styling, screen widths, responsive behavior.

Table 43 Front-End Technologies and Their Purposes

Sources: Bootstrap documentation [8], Tailwind CSS documentation [9], jQuery documentation [10], SweetAlert2 documentation [11], and Font Awesome documentation [12].

it is verified that the Bootstrap, Bootstrap Icons, SweetAlert2, jQuery, custom CSS, responsive style and other interface libraries are loaded in the main Blade layout from the inspected Laravel project [8], [10], [11].

Most of the Tailwind activities were done on their own practice front-end pages, instead of working entirely on the existing Laravel project snapshot [9].

3.5.4.4 Improving the New Invoice Page

The first front-end work done has been the enhancement of the New Invoice page which is part of the Laravel project.

I've fixed some interface issues and took some time to look into ways to make the actions on the invoice more prominent and understandable using Bootstrap classes [8]. Rather than having to create a separate CSS rule for each simple button, I leveraged Bootstrap's already built-in button classes to achieve the desired look, when they were available to do so[8]. On the New Invoice page, there's a green Save button and outlined Cancel button [8].

```
</>blade

<div class="text-center">
  <button
    type="submit"
    class="btn btn-success"
    id="save_invoice">

    {{ __('msg.Save') }}
  </button>
  <a
    href="{{ url('/dashboard') }}"
    class="btn btn-outline-danger">

    {{ __('msg.Cancel') }}
  </a>
</div>
```

Code Snippet 24 New Invoice Save and Cancel Buttons

The Save button is in green btn-success class, whereas it is a positive operation. The btn-outline-danger class is used to make the Cancel button red with outline, it's obvious on how the Cancel button will be separated from the save operation.

Bootstrap Class	**Purpose**
btn	Uses the basic structure of a Bootstrap Button.
btn-primary	Produces a blue Primary Action.
btn-success	Produces a successful or saving green.
btn-warning	Produces a yellow WARNING or SECOND ACTION.
btn-danger	Creates a destructive action that is red.

btn-outline-secondary	Produces an outlined action that has a neutral effect.
btn-outline-danger	Produces an action with a red outline.
btn-lg	Increases button size
btn-sm	Reduces button size
w-100	Expands the button to occupy the entire width of the button.
shadow-lg	Increases the size of the shadow.

Table 44 Bootstrap Button Classes Practiced

Source: Bootstrap documentation [8].

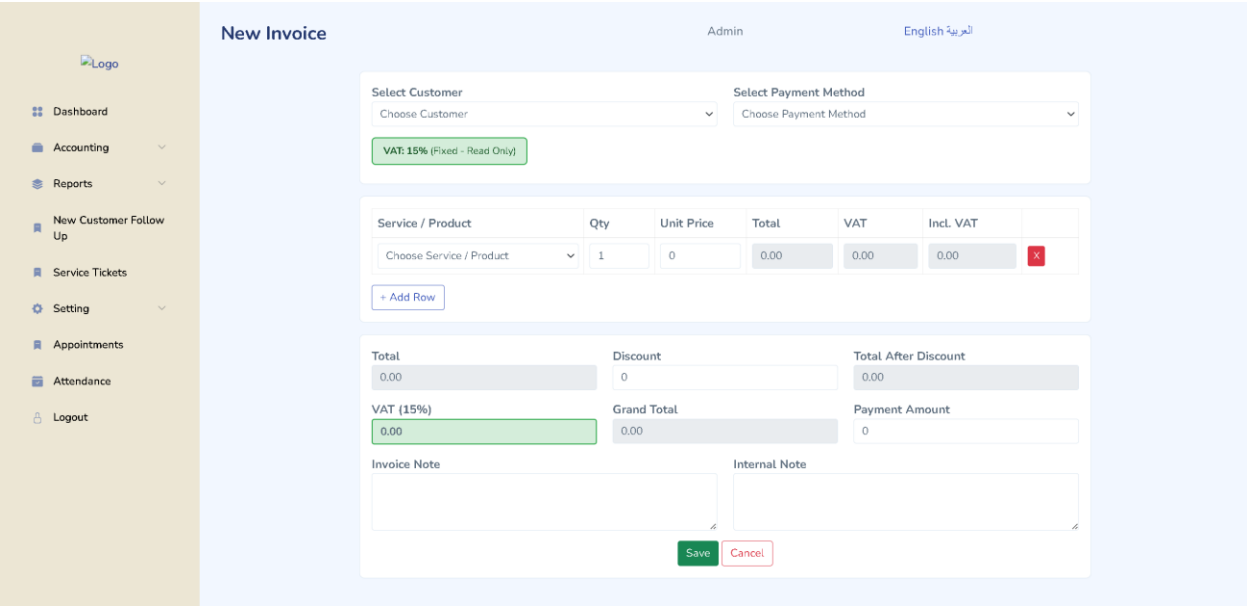

Figure 32 Improved New Invoice Page Using Bootstrap Components

Here is the enhanced New Invoice page, which is organized by using Bootstrap components and custom styling of the form's front end. The page contains fields for customer, payment-method, service/product, quantity, unit price, VAT calculation, discount, grand total, payment-amount and

note. The Save button is green in color and the Cancel button is outlined, this will make the invoice form easier to use and understand.

3.5.4.5 Improving the Invoice Preview Action Buttons

The Invoice Preview page has a number of important actions:

- Print PDF
- Print Receipt
- Send by WhatsApp
- Back

All these actions were organized under a single action section. All the buttons were given distinct colors and icons to enable users to easily identify the purpose of the button.

Button	**Purpose**	**Visual Design**
Print PDF	Opens a printable PDF invoice.	Yellow, with pdf icon.
Print Receipt	Starts the receipt/thermal format.	Green, with a receipt icon.
Send by WhatsApp	Provides the receipt/invoice to the customer.	WhatsApp green
Back	Goes back to the previous page.	Neutral outlined design

Table 45 Invoice Preview Actions

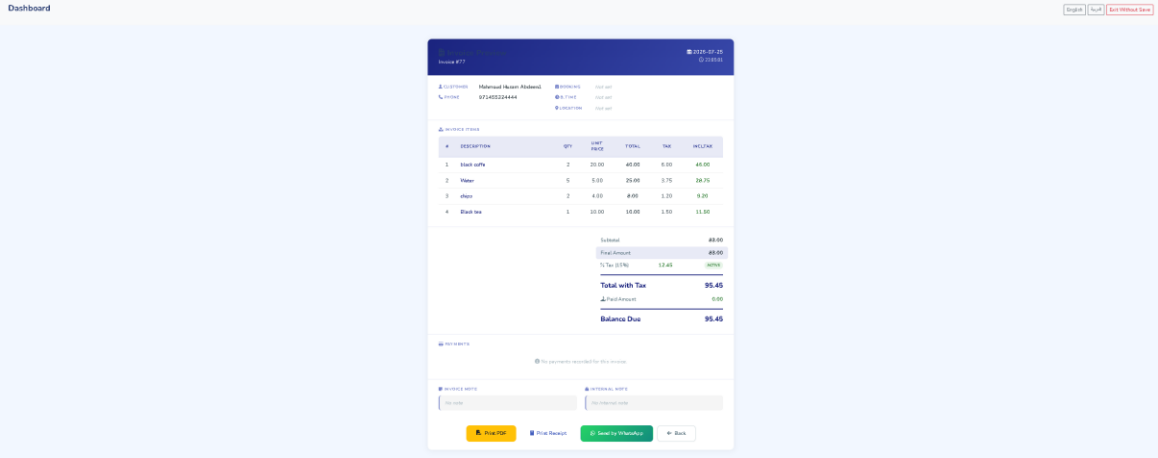

Figure 33 Invoice Preview Action Buttons

This is the main action buttons on the Invoice Preview page. The buttons provide the ability to print the invoice as PDF, print the receipt version of the invoice, send the invoice via WhatsApp or go back to the previous page. Each action was clearly and easily identified using different colors, icons and spacing.

3.5.4.6 Applying Custom CSS to the Invoice Interface

The basic structure of the buttons was based on Bootstrap and custom CSS was applied to increase the specificity of the designs.

Consistent padding, font weight, spacing, rounded corners and transitions were added to the invoice buttons.

CSS Property	**Purpose**
background-color	Changes the background colour.
color	Sets the colour of text.
padding	Adds internal spacing
margin	Adds external spacing

border-radius	Rounds element corners
box-shadow	Applies depth around buttons, cards and forms.
display: flex	Arranges items in rows or columns.
display: grid	Uses a grid to organize elements.
gap	Includes regular spacing between elements.
width	Sets the width of the element.
max-width	Prevents an element from becoming too wide.
transition	Produces seamless transitions between styles.
transform	Moves or scales an element.
object-fit	Controlling how an image fits inside of another container.
overflow-x	Allows horizontal scrolling
@media	Applies responsive rules to certain screen sizes.

Table 46 CSS Properties Practiced

3.5.4.7 Responsive Invoice Preview Design

Added Responsive CSS to avoid the Invoice Preview page being unusable on smaller screens.

```
</>CSS

@media (max-width: 768px) {
  .inv-preview-header {
    flex-direction: column;
    gap: 12px;
    padding: 20px;
  }
  .customer-info-section {
    flex-direction: column;
    gap: 16px;
    padding: 16px 20px;
  }
  .totals-box {
    width: 100%;
  }
  .notes-section {
    flex-direction: column;
  }
  .action-section {
    flex-direction: column;
  }
  .btn-action {
    justify-content: center;
  }
}
```

Code Snippet 25 Responsive Invoice Preview Rules

At smaller screen sizes horizontal sections become vertical sections. Totals box becomes full width and action buttons are stacked and centered. This helps to keep the page from becoming too "wide" and ensures important actions are visible.

3.5.4.8 Creating and Improving the Login Page

Worked on HTML, Bootstrap and custom CSS to create and redesign a Login page. This activity was about form structure, input fields, spacing, icons, card design and user feedback.

Laravel login page is based on:

- Bootstrap Container and Grid classes.
- A centered authentication card.
- Username and password boxes.
- Bootstrap Icons.
- Error alerts.
- A large Login button.
- Loading spinner while submitting.
- Responsive column sizes.

Responsive column classes adjust widths of the card based on the screen size. Form-control classes give uniformity to the inputs, and Bootstrap Icons help determine the username and password fields [8].

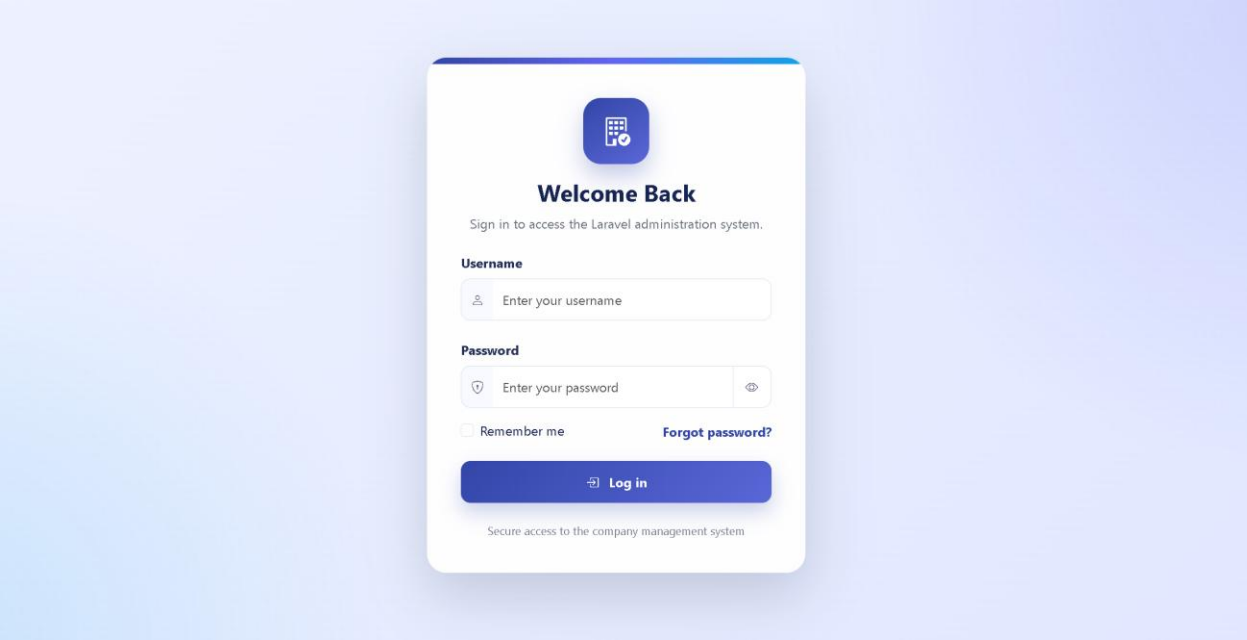

Figure 34 Login Page Using Bootstrap and Custom CSS

This is the enhanced Login page that have been developed using Bootstrap elements and personalized CSS. There's also a login card in the center, a system logo, a username and password input, icons for input, a visibility button for the password input, a Remember Me button, a Forgot Password link, and a full-width Login button. Using rounded corners, spacing, a gradient background and a card shadow, a neat, responsive and professional authentication interface has been created [8].

3.5.4.9 Creating a Login and Registration Practice Page

I also checked out the Laravel login interface and have developed an independent Login and Registration Practice Page with:

- HTML
- Pure CSS
- Bootstrap 5
- JavaScript

The goal was to get some experience with how to structure pages and interact with the user without using similar concepts in Laravel Blade forms.

```
</>HTML

<input
  type="email"
  class="form-control"
  placeholder="Enter your email">

<input
  type="password"
  class="form-control"
  placeholder="Enter your password">

<button
  class="btn btn-primary w-100">
  Login
</button>
```

Code Snippet 26 Bootstrap Login Form Elements

The form-control class is used for a uniform look of the inputs. The button is the main button with the btn-primary class and w-100 makes it 100% wide of the form.

3.5.4.10 Switching Between Login and Registration Forms

The Login form and the Registration form were positioned on the same page and alternated between with JavaScript.

```
</>JavaScript

function showRegister(){
  document
    .getElementById("loginForm")
    .style.display = "none";

  document
    .getElementById("registerForm")
    .style.display = "block";
}

function showLogin(){
  document
    .getElementById("registerForm")
    .style.display = "none";

  document
    .getElementById("loginForm")
    .style.display = "block";
}
```

Code Snippet 27 JavaScript Form Switching

On clicking Register, the Login form is hidden and Registration form is shown. Click on Login to undo this.

This activity illustrated that the content on a page can be manipulated using JavaScript and interaction can be enhanced without loading a new page. This is exactly what is documented in the task report that shows how the task was created, redesigned and tested.

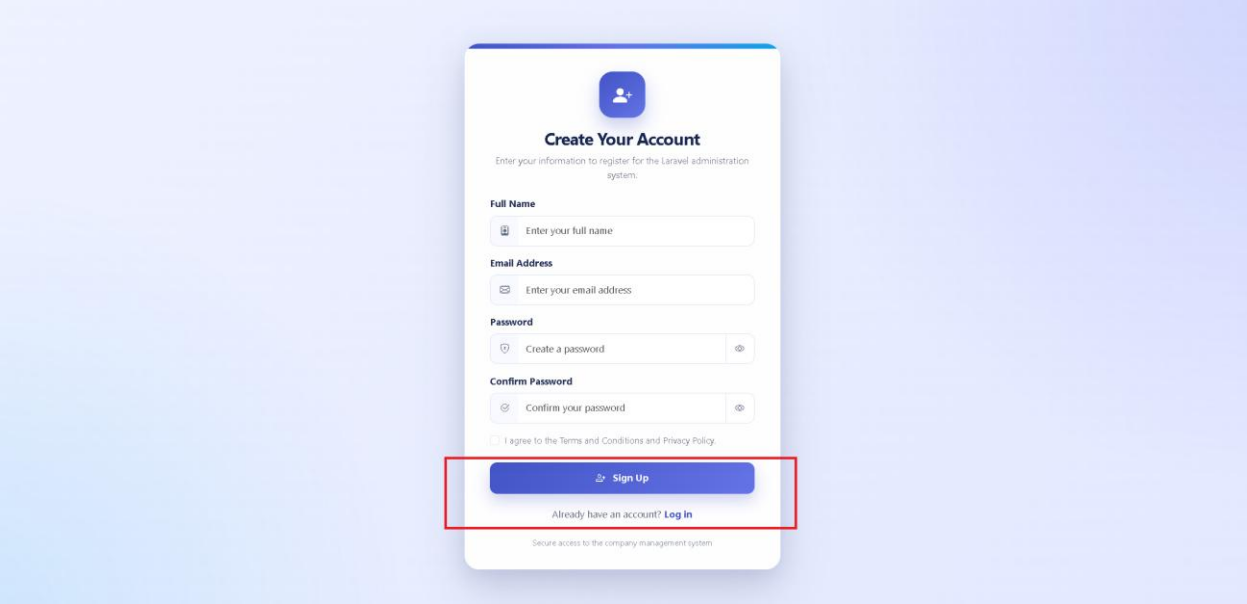

Figure 35 Switching Between the Login and Registration Forms

This is the authentication page with both a Login and Registration form. Registration form is used to create new user with their full name, email address, password, confirmation password and login form for existing users to enter their username and password. The form visibility is handled by using JavaScript, such that when a user clicks on Sign Up, he/she will see the Registration form, and when a user clicks on Log in, he/she will go back to the Login form without opening a new page.

3.5.4.11 Practicing HTML Form Elements

I made up some practice problems that have typical HTML form elements. This activity was helpful in understanding the information that is collected prior to sending it off to a Laravel controller.

HTML Element	Main Purpose	Important Attributes
input	Collects short information	Type, name, id, value, placeholder and required.
select	Produces a drop-down list.	Name, id, class, required.
option	Makes a drop down selection.	value, selected
button	Defines an action that can be invoked by clicking on it.	type, class, id
a	Creates a link	href, class, target
checkbox	Allows multiple selections	Type, name, value, checked are the attributes.
radio	Allows one choice in a group.	Type, name, value, checked are the attributes.
textarea	Collects longer information	Name, Id, Rows, Placeholder.
label	Describes a form control.	for, class

Table 47 HTML Form Elements Practiced

The value of the label's for attribute is equal to the id of the input. This makes them easier to access as the label will be used to focus the appropriate form control.

3.5.4.12 Exploring the Mazer Bootstrap Admin Template

I looked around Mazer Bootstrap Admin Template and learned about the professional admin interface organization.

The template included samples of:

- Sidebars
- Headers
- Navigation menus
- Dashboard cards
- Forms
- Tables
- Alerts
- Icons
- Buttons
- Responsive page layouts

I learned how to choose an appropriate piece from the template structure and edit the content and/or the Bootstrap classes.

Mazer Component	How the activity might be applied in the project.
Sidebar	Displays system navigation
Header	Shows the title of the page and the user information.
Card	Provides dashboard information or summary.
Table	Shows customer, invoice and transaction details.
Form	Collects customer or invoice information.
Modal	Adds and/or alters information, but remains on the page.
Alert	Shows an error message or a status message.
Badge	Identifies status values
Grid	Organizes dashboard in a responsive way.

Table 48 Mazer Components Reviewed

Sources: Bootstrap documentation [8] and SweetAlert2 documentation [11].

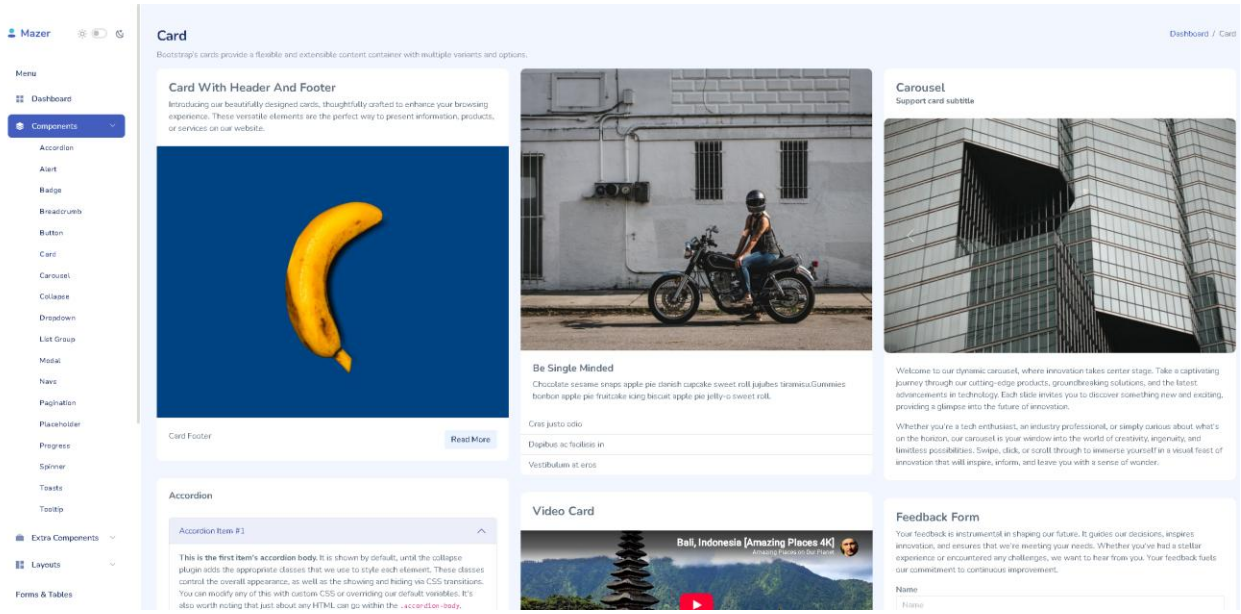

Figure 36 Mazer Bootstrap Components Practice Page

This is the Mazer Bootstrap Admin Template for professional dashboard organization and ready-made components to interface. Responsive sidebar, navigation menu, content cards, image carousel, accordion, video card and feedback form are included. These components helped me to see how the Bootstrap templates can be utilized and modified to develop organized administration dashboards, without having to create all the interface elements from scratch.

3.5.4.13 Using Bootstrap Icons and Font Awesome

To enhance the visual understanding, icons were added to buttons and input fields, as well as to alerts and navigation elements.

```HTML
<button class="btn btn-primary">

  <i class="bi bi-pencil-square"></i>
  Edit
</button>

<a class="btn btn-success">

  <i class="fab fa-whatsapp"></i>
  WhatsApp
</a>
```

Code Snippet 28 Buttons with Icons

The icons were not used exclusively, but in conjunction with the written labels. This helped the interface to be understandable and yet, maintaining a professional look.

3.5.4.14 Using Bootstrap Alerts and SweetAlert2

I found the opportunity to use Bootstrap alerts for permanent and inline messages, and SweetAlert2 for interactive popup messages.

Alert Type	Purpose
Success	Ensures that an operation has been completed.
Warning	Indicates potential issues to the user.
Danger	Shows an error with an operation or validation.
Information	Provides general information
Primary	Displays an important message related to the system.

Table 49 Alert Types and Their Uses

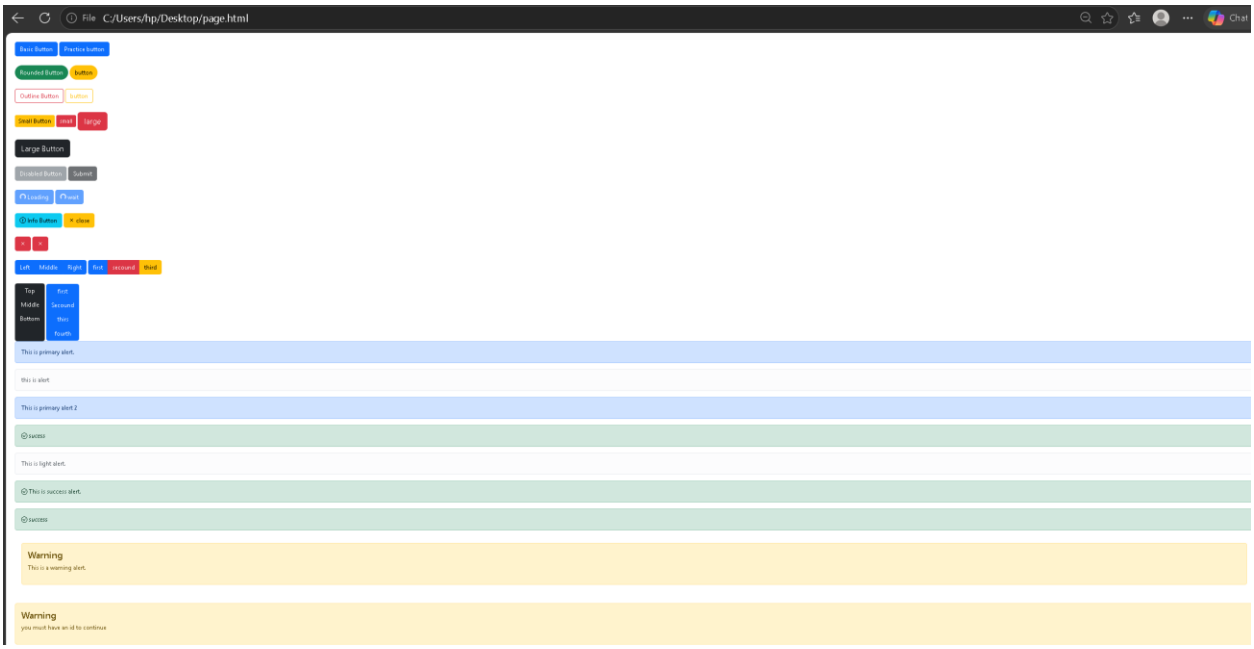

Figure 37 Bootstrap Buttons, Icons, and Alert Components Practice Page

This is a Bootstrap practice page with various styles, color, icon, size and outline/loading/disabled buttons and button groups. Combining bootstrap classes to define clear actions and various types of user feedback, such as primary messages, success, warning and informational ones.

3.5.4.15 Introduction to Tailwind CSS

I have practiced Bootstrap and then studied Tailwind CSS. Tailwind takes a "utility-first" approach, which means that each class is grouped directly within HTML elements [9].

The CDN was used to add Tailwind to the practice page [9].

Design Area	**Example Class**	**Purpose**
Background	bg-blue-600	Uses blue as a background colour.
Text colour	text-white	Sets text to be white.
Padding	px-4 py-2	Adds horizontal and vertical spacing.
Margin	mt-4	Increases the amount of room above an element.
Rounded corners	rounded-lg	Rounds element corners
Shadow	shadow-md	Adds a medium shadow.
Hover effect	hover:bg-blue-700	Modifies the background when you hover over it.
Font weight	font-semibold	Improves the text.

Width	w-full	Fills in the maximum possible width.
Maximum width	max-w-md	Specifies the width of an element.
Alignment	text-center	Centers text
Flexbox	flex	Enables the Flexbox layout.
Grid	grid	Turns the Grid layout on.
Horizontal overflow	overflow-x-auto	Enables horizontal scrolling for a large table.

Table 50 Tailwind Utility Classes Practiced

Source: Tailwind CSS documentation [9].

All the Tailwind activities like utility buttons, responsive login design, statistic cards and tables were recorded as front-end practice work.

3.5.4.16 Creating Tailwind Button Styles

To create a few of the many Tailwind buttons, I used a combination of utilities.

Figure 38 Tailwind CSS Button Styles

This is the figure of five different styles of buttons using Tailwind CSS utility classes. The Primary button is the main page action, Success button is used for positive operations like save, Danger button is used for destructive operations like delete, Outline button is used for secondary navigations on the page and Disabled button is used for a function that is not available. Tailwind classes were used to set the background colors, text colors, padding, rounded corners, border, shadows, icons, hover and disabled appearance.

3.5.4.17 Creating Responsive Dashboard Statistic Cards

I designed dashboard cards that provide some key system information:

- Customer count
- Invoice count
- Total revenue
- Pending payments

Class	**Behavior**
grid	Turns on the Grid layout.
grid-cols-1	When displayed on small screens, shows 1 column.
md:grid-cols-2	On medium screens, will show two columns.
lg:grid-cols-4	Shows 4 columns on a big screen.
gap-6	Adds a space between the cards.
p-6	Adds internal spacing between cards.
rounded-xl	Smooths the corners of the card.

shadow-md	Adds visual depth

Table 51 Responsive Tailwind Grid Classes

Sources: Bootstrap documentation [8] and Tailwind CSS documentation [9].

3.5.4.18 Comparing Bootstrap and Tailwind CSS

I have tried to figure out the pros and cons of Bootstrap and Tailwind CSS and when to use which.

Area	**Bootstrap**	**Tailwind CSS**
Main approach	Ready-made components	Utility-first classes
Buttons	Prepared button classes	A new utility that has been assembled from other utilities.
Forms	Prepared form-control styles	Customized through utilities
Cards	Complete card structure	Developed using layout and styling classes.
Alerts	Ready-made alert components	Designed manually
Tables	Prepared table classes	It is customized using utility classes.
Layout	Bootstrap Grid	Breakpoints, Grid and Flexbox utilities.
Responsiveness	Responsive component & grid classes.	Sm: prefix, md: prefix and lg: prefix.
Customization	Requires additional CSS frequently.	In-class detailed control.

Development speed	Observe a fast for regular interfaces.	Flexible and adaptable for custom designs.
Learning approach	Understand parts in components and class names.	Know utilities and how to use them in conjunction.
Suitable use	Business systems and normal dashboards.	Highly customized interfaces

Table 52 Bootstrap and Tailwind CSS Comparison

When I needed standard components and was in a hurry, Bootstrap came in very handy. I found Tailwind helpful when I wanted full control over colors, spacing, sizing. and responsiveness [8], [9].

A task report has been uploaded and the comparison is detailed, as well as how the individual frameworks were applied in the individual front-end activities.

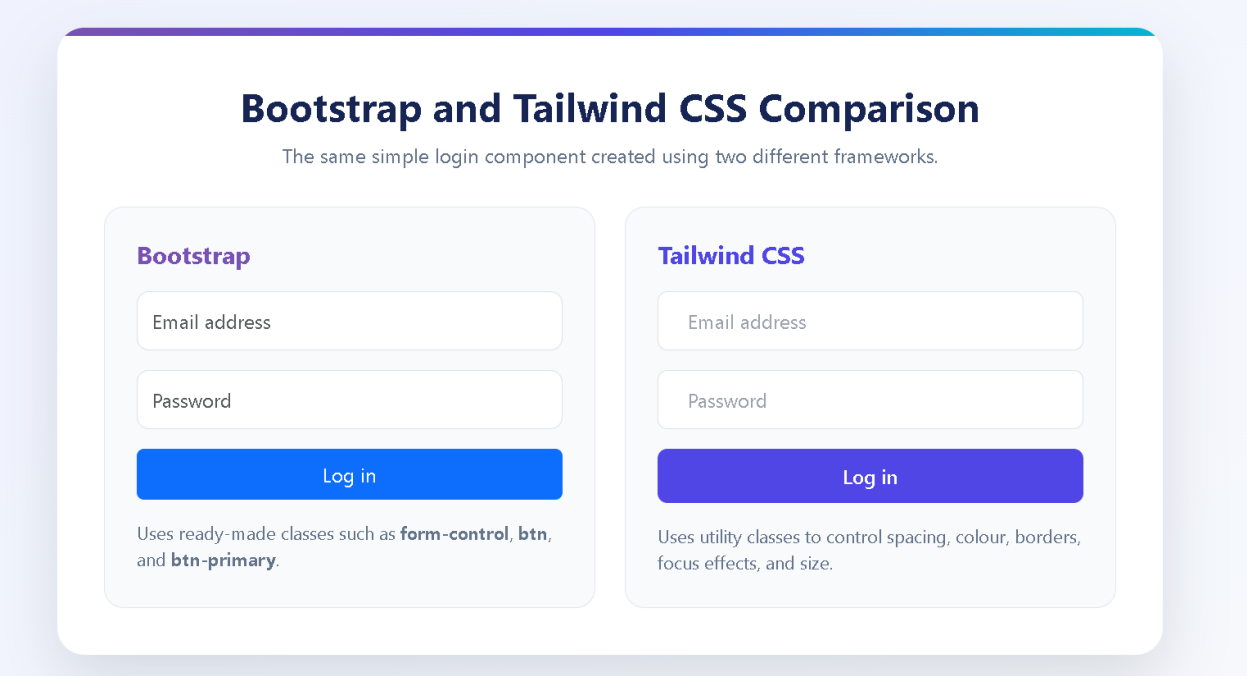

Figure 39 Bootstrap and Tailwind CSS Login Component Comparison

This number is for the same interface of Logins built with Bootstrap and Tailwind. The Bootstrap version is based on classes that are pre-stylized and provided as services, this includes form-control, btn, btn-primary., and the Tailwind version requires classes to be combined in order to handle spacing, colors, borders, focus effects, sizing and buttons design. The example demonstrates that Bootstrap is ideal for rapid development of typical elements, and Tailwind CSS offers more flexibility for styling unique interfaces [8], [9].

3.5.4.19 Challenges Faced and Solutions Applied

Challenge Faced	**Solution Applied**
It was hard to discern the actions of the invoice	Made use of different colors, labels and icons.
Some buttons were too small – not all are accessible.	Made use of bigger Bootstrap classes and extra padding.
The invoice interface was not as neat as possible.	Combination of Bootstrap classes and custom CSS.
Feedback for buttons was required.	Implemented hover effects, animations and shadows.
The sections of an invoice were not small screen friendly.	Added responsive media-query rules.
The login page was very simple in appearance.	Added card, rounded corners, spacing, and shadow effect in the middle of the card.
Purpose for input was not clearly demonstrated	Added Bootstrap Icons
From submission was allowed to be repeated	Made the spinner appear and the Login button was disabled.
Login and registration should be on the same page.Login and Registration should be combined in one page.	Made use of JavaScript for visibility control.
Both form can occur at the same time.	Made use of display: none and display: block.
Elements and attributes of the form were not known.	Developed a Practice page for HTML form.

No clear professional structure of the dashboard	Analyzed a template called Mazer Admin Template.
There were opportunities to improve upon user feedback.	Practiced Bootstrap alerts and SweetAlert2.
Bootstrap was lacking customization	Added pure CSS
Wanted increased design options.	Learnt some Tailwind Utility classes.
At first, the classes with Tailwind were challenging.	Conducted one design property by one design property.
Required to know about dependency of Tailwind.	Tested using and without using the CDN.
Responsive behavior was required for dashboard cards.	Implemented Tailwind Grid breakpoints.
Customer tables were wide, which made for poor fit on small screens.	Added overflow-x-auto
It was challenging to identify status values	Added colour-coded badges
There were a number of pages that required a consistent design	Used re-used headings, colors, spacing, buttons.

Table 53 Task 4 Challenges and Solutions

3.5.4.20 Skills and Knowledge Developed

Skill	**Learning Outcome**
HTML structure	Semantically and normally organized pages.
CSS styling	Manage colors, spacing, shadows, borders and hover effects.
Bootstrap 5	Applied ready-made forms, buttons, cards, alerts, Grid and responsive classes.
Tailwind CSS	Created customized interfaces using utility classes.
JavaScript	Adds interactivity, client-side validation, dynamic content updates, and event handling to web pages.
jQuery	Handled the form submission and loading of the feedback.
Bootstrap Icons	Added some useful icons on the interface.
Font Awesome	Added action-specific icons
SweetAlert2	Professionally showed up a pop message.
Mazer template	Comprehended organization of a professional's dashboard.
Laravel Blade	Implemented front-end elements on dynamic pages of Laravel.
Responsive design	Resized forms, cards, buttons and tables to fit screen size

Flexbox and Grid	Worked effectively with organized interface elements.
Form design	Developed Login, Registration, customer and practice forms.
Data-table design	Provided information in a responsive structure with status badges.
Navigation	Incorporated several pages in links.
Browser testing	Tested layout, interaction, responsiveness and dependency of the framework.
Framework selection	Knows when Bootstrap or Tailwind is more appropriate

Table 54 Skills Developed During Task 4

3.5.4.21 Task Summary

In this task, I worked on and utilized my front-end skills, tools, and technologies such as HTML, pure CSS, Bootstrap, Tailwind CSS, JavaScript, jQuery, Bootstrap Icons, Font Awesome, SweetAlert2, Mazer Admin Template, and Laravel Blade [2], [8], [9], [10], [11], [12].

New Invoice and Invoice Preview pages were enhanced by rationalizing the main actions and provided with clear colors, icons, spacing, rounded corners, hover effect and responsive styles. The print pdf, print receipt, share whats app and go back to previous page actions were more easily understood [8], [12].

I have updated and enhanced the Laravel Login screen to make it responsive with Bootstrap Grid, columnar responsive elements, form controls, icons, card designs, alerts and a loading spinner [8]. I've also made a separate Login and Registration practice page and applied a JavaScript that lets users switch between both of the forms without going to a new page [8]. I used common HTML forms elements and attributes such as labels, inputs, dropdowns, checkboxes, radio buttons, text areas and links. I also tried Mazer Bootstrap Admin Template and made Revision Page with Bootstrap button, card, alert, form, icons and layouts [8].

I took the Bootstrap practice and then learned Tailwind CSS. Developed a responsive 'Login' page, statistic cards, dashboard, customer table and colour coded 'Status' badges using Tailwind buttons. I learned how breakpoint classes can respond to the width of the screen and create an alternative layout [9]. I also did a Bootstrap vs Tailwind comparison. Bootstrap proved to be more effective when it came to quickly developing common business interfaces [8], [9] , but Tailwind allowed for more flexibility when it came to building custom interfaces.

Finally, I developed the multi-page front-end website with Home, Login, Registration, Dashboard, Customer pages with connectivity. Did some browser testing of navigation links and responsiveness.

3.5.4.22 Conclusion

I gained more knowledge about the design, structure, testing and integration of a professional front-end interface with a web application.

I was instructed that a fantastic user interface depends not simply on colour. It also demands proper organization of information, spacing, readability, size of buttons, recognisable icons, responsive layout, feedback for the user and predictability of navigation.

Using classes that were pre-made, Bootstrap was the perfect way to quickly create some standard components to add to websites. When Bootstrap's default appearance wasn't enough, I was able to customize it with Pure CSS. Tailwind CSS showed that it was possible to build very responsive and detailed designs with utility classes without having to write out a lot of individual CSS rules [8], [9].

To improve loading and feedback, I used jQuery, and SweetAlert2 library for system messages, and added user interaction using JavaScript, switching between various forms [10], [11]. Mazer gave examples of how sidebars and cards, tables and forms, sections of content are organized on professional administration dashboards.

Responsive Design Activities helped me learn how to make the cards, tables, forms and buttons work on different screen sizes. Having a multi-page website also helped me to appreciate how a consistent design should be sustained as part of the connected pages.

This project helped me improve my HTML, CSS, Bootstrap, Tailwind CSS, JavaScript, jQuery, Laravel Blade, responsive design, selection of components, dashboard layout, form design, user feedback, interface testing, and organization of a front-end project [2], [8], [9], [10].

3.5.5 Task 5: Supporting Back-End Development Using Laravel

Task Description

Support back-end development tasks by connecting forms with controllers, validating user input, handling database operations, and fixing basic system errors.

3.5.5.1 Introduction

In this task, I assisted with a number of back-end development tasks in the Laravel application. The primary goal was to gain a better understanding of how information flows through a real-world web-based application from the user interface, all the way to the Laravel routes, controllers, models, database, and Blade views [2], [4].

This was not a job about just the look and feel of the pages, but more about the logic of the application. I learned about how a form passes data to a controller, how Laravel validates the data submitted, how it gets inserted or updated in MySQL, and how the controller responds with a success message, a validation error or a report page [2], [4].

Firstly, I was looking at two admin reports that were available: Sales Report, and Payment Report. The reports were helpful in demonstrating examples of how to get data from multiple related tables in the database and format for display, using Laravel. The Sales Report fetches data from the invoices and associates them with the customers, modes of payment, the users, tax rates, amounts paid and balances. The Payment Report is based on payment records and include other related information such as invoice, customer, and payment-method [2], [4].

Having reviewed these reports I replicated the same back-end architecture; developing and testing a new Task Assignment page and Task Report page. On the Task Assignment page the administrator can choose a staff member, type in a task that is requested, and then submit the

information. Before saving the task in the database, the values entered into the form are validated by Laravel [2].

The Task Report undoes what the Task Report does. No information is sent from the page to the database, rather it is getting information from the database and sending it to a Blade view. This report includes the staff member assigned to the task, requested task, creation date, task status and completion date. It also displays summary information which displays total, Pending and finished tasks.

3.5.5.2 Understanding the Laravel Back-End Workflow

Normally, a Laravel page is not directly associated with the database. All requests go through an orderly process with well defined roles for each aspect of the application [2].

The Blade page will be used to present and present the form to the user and to gather information. The user submits the form which calls a Laravel route. The route is used to specify the controller method. The controller then processes the data, validates it, executes the necessary business rules and interacts with MySQL via an Eloquent model [2], [4].

Once the operation is done, the controller will return a response. Depending on the function, the response could be a Blade page, redirect, a success response in JSON format or validation errors [2].

It enabled me to see the whole picture and determine where potential issues could lie. For instance, if you have a page with an incorrect route, or maybe a controller method is missing, perhaps a database column is incorrect, perhaps the relationship is unavailable or a blade variable is missing.

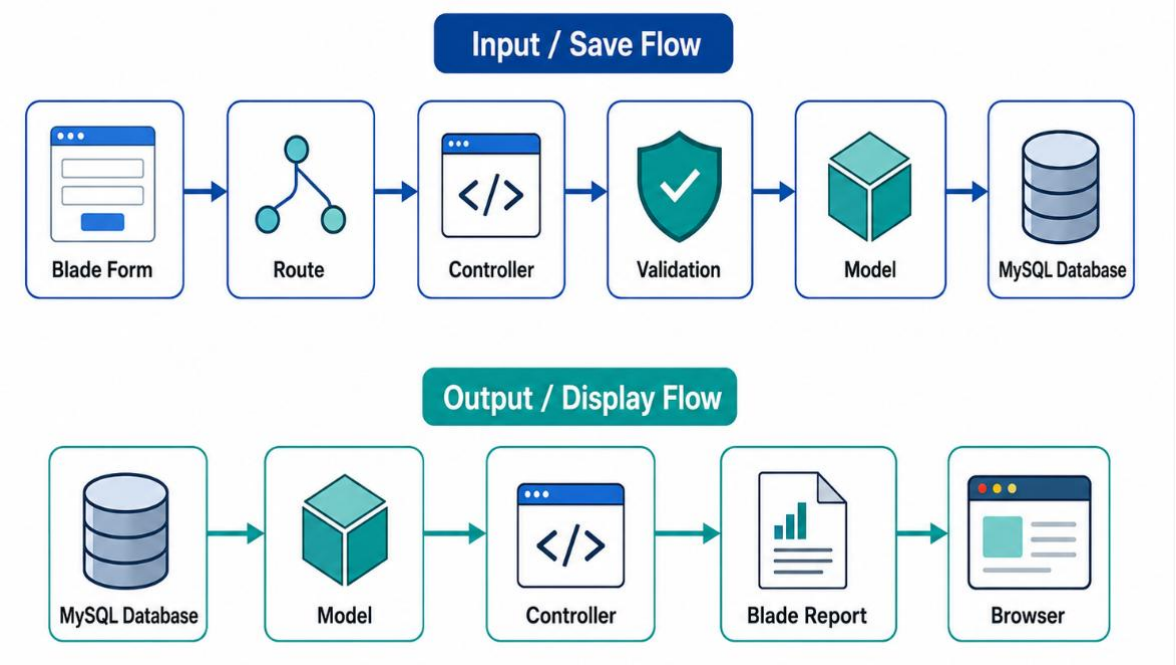

Figure 40 Laravel Back-End Request and Data Flow

This figure shows two major data gateways which are used in the Laravel system. The input & saving flow comes when the user enters information via the Blade form and that information is passed to a route within the Laravel application, which will then route the request to a controller. The controller validates the data submitted and with the help of the corresponding model stores the data in the MySQL database. For the output and display flow, the model will fetch the necessary data from the database, the controller will format the data and finally the Blade report will render the output in the user's browser [2], [4].

3.5.5.3 Reviewing the Sales Report Back End

The first back-end page that I checked out is a Sales Report. The intent of this activity was to delve into the concept of how Laravel gets a collection of invoices and joins to each one some data from another portion of the system.

So I began by searching for the Sales Report route in routes/web.php. The route is capable of handling GET and POST requests. A GET request normally displays the report; a POST request is used when the report parameters, like date-filter values, are filled in by the administrator [2].

```
</> PHP

Route::match(
    ['GET', 'POST'],
    'salesreport',
    'InvoiceController@salesreport'
);
```

Code Snippet 29 Sales Report Route

This is the path to the Sales Report URL to the salesreport() method in InvoiceController.php.

In the controller method, it is verified if the administrator has inputted start date and end date. If both dates are specified, then the range of dates provided in the invoice query is used to get the information created in that range of dates. If no date is provided, then the report fetches all the Invoices of the current Branch.

```php
</> PHP

$invoices = Invoce::with(
   'customers',
   'paymentmethod',
   'user'
)
->where('branch_id', session('branch_id'))
->whereBetween(
   'invoice_date',
   [$start_date, $end_date]
)
->get();
```

Code Snippet 30 Retrieving Invoice Records with Related Data and Date Filtering

This code accesses these relationships required for the report using with() function. The Customer details are provided in the Customer Relationship. Payment-method relationship is to identify how the invoice is paid and user relationship is to identify the administrator/employee who is associated to the invoice.

The condition of the branches is also a factor. Some information about different branches is stored in the system and the report should only show the details of the invoices for the branch that is currently stored in the session.

The controller then fetches the invoices and computes the amount of the invoice, the remaining amount, in the report. Along with the invoice records and selected date values, these calculated values also are passed to the Blade view [2], [4].

Then the Blade page will render an @foreach loop that will display an individual invoice in each row. You can add the invoice number, date, customer name, mobile number, total amount, discount, tax, paid amount, remaining amount, payment method and user to the table [2].

When I read this report, I learned that the controller, besides fetching records, does other things. It applies filters, constructs the query, loads relationships, adds the totals, applies sorting and prepares all variables necessary for the Blade page [2].

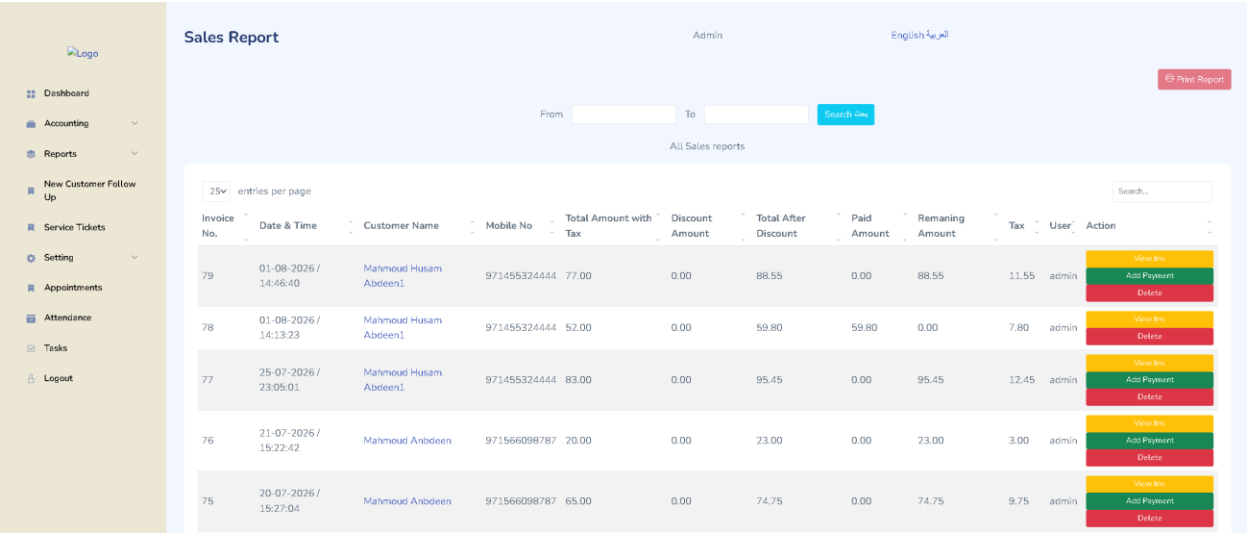

Figure 41 Sales Report Page

This is the administrative Sales Report page to view information related to the sales and invoices. Filters for the date range, a general search field, print-report action and detailed table of the invoice. This shows the following information for each row: invoice number, date and time, customer name, customer mobile number, total amount (with tax), discount, total amount (without discount), paid amount, remaining balance, tax value and the user responsible for that row. The action buttons available also enable the administrator to view the invoice, add a payment if a balance is outstanding or delete the record.

3.5.5.4 Reviewing the Payment Report Back End

Once I looked into the Sales Report, I then looked at the Payment Report. Both reports have financial data but start with different data from the database.

The Sales Report begins with records of sales invoice. The Payment Report is based on the rows in the inv_payment table. The foreign-key values which are recorded in each payment row include the invoice ID, customer ID, payment-method ID and branch ID [4].

To the administrator, only showing these numeric values won't be helpful. Therefore, the Inv_payment model is related to the models: invoice, customer and payment-method [2], [4].

```php
</> PHP

public function inv()
{
   return $this->belongsTo(
      Invoce::class,
      'invoice_id',
      'id'
   );
}
```

Code Snippet 31 Main Payment Relationships

There are relationships between the same model Customer and Paymentmethod. Through these relationships, Laravel can show the invoice number, name of the customer, or name of the payment-method, instead of just the IDs [2], [4].

The Payment Report controller gets the payment information and loads these relationships before returning this information to the Blade page [2].

```php
</> PHP

$payments = Inv_payment::with(
   'inv',
   'customers',
   'paymentmethod'
)
->where('branch_id', session('branch_id'))
->orderBy('pay_id', 'desc')
->get();
```

Code Snippet 32 Retrieving Payment Report Records

If you specify a Date Range, then a whereBetween() condition is used for the payment_date column. This will enable the administrator to only view payment(s) that were created within the selected time period [2], [4].

The controller also summarizes the information of the detailed payment rows. This can be the entire paid amount, collected tax and totals for each payment method [2], [4].

This report helped me to understand the relationship with databases better. Foreign keys are stored in the inv_payment table and Eloquent gets the foreign information from other tables [2], [4].

You don't have to manually search each database table with the Blade page. It will be given the ready relationships by the controller and will show the necessary information [2], [4].

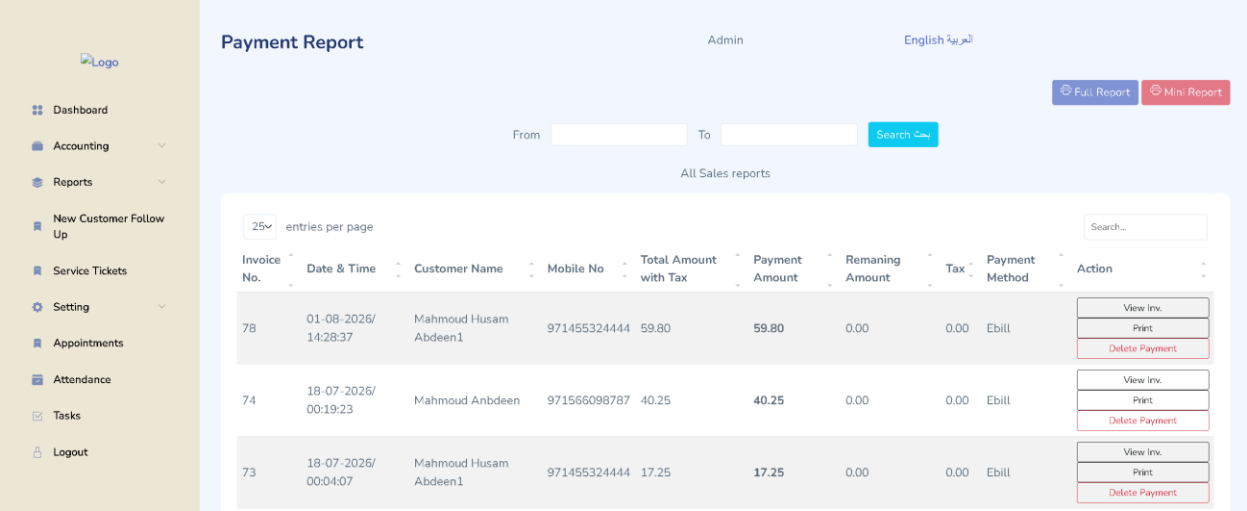

Figure 42 Payment Report Page

This is the Administrative Payment Report page that is used to view payments made in the system. The page features details on the payment table, full report and mini report printing, date-range filters and a search field. Every record will have the following details: invoice number, payment date and time, customer name, customer mobile number, total amount (including tax), payment amount, balance remaining, tax value, and payment method. The action buttons provide the administrator with the ability to view the associated invoice, print the payment details or remove the payment record.

3.5.5.5 Creating the Task Assignment Page

Reviewed the current report structures and worked on Task Assignment page. The goal was that the new feature should share the same architecture as Laravel, which enables the admin to assign tasks to staff [2].

There are tasks displayed on the page and an Add New Task button. Clicking on this button will open a Modal Form. In the form there is a dropdown for staff and a place for requested task [8].

The staff dropdown is not written manually inside the Blade page. The controller retrieves the Staff dropdown is not hand crafted inside the Blade page. The controller calls the database to get the list of available staff records to display in the view. Blade then uses the collection retrieved to create the dropdown options [2], [4].

This will help the admin to choose a genuine staff member in the system. The chosen one passes the staff_id which is given as a foreign key in staff_tasks [4].

```php
</> PHP

Route::post(
    'storenewtask',
    'StaffController@storenewtask'
);

Route::match(
    ['GET', 'POST'],
    'taskreport',
    'StaffController@taskreport'
);
```

Code Snippet 33 Task Management Routes

The first one is a Task Assignment form. The second will launch the Task Report and also can take submitted date filters [2].

Upon opening the Task page by the administrator, the controller starts preparing staff records and currently worked tasks. These variables are passed to Blade, and it will show them on the page [2], [4].

A new task will have a default setting of 1 (Pending). This means that it doesn't need to be manually set to "Not Complete" by the administrator as all newly assigned tasks are created as "Not Complete" status.

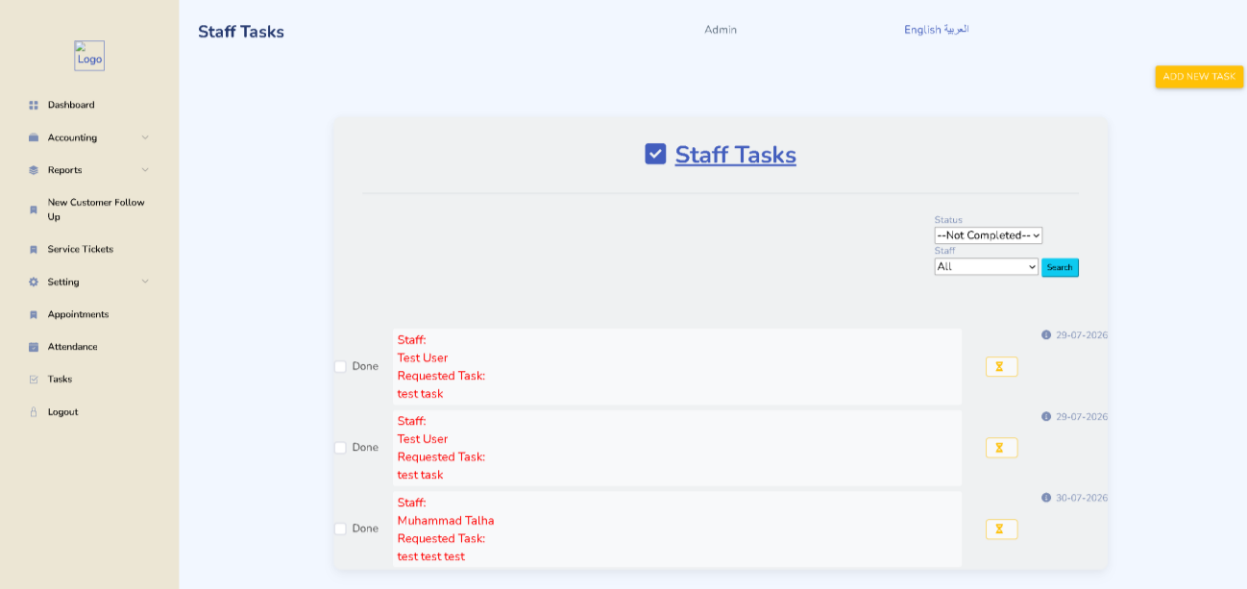

Figure 43 Task Assignment Page

3.5.5.6 Connecting the Task Form with the Controller

The Task Assignment form is AJAX that passes on to the Laravel back end. AJAX enables the page to process the form and get the results back without going to a different processing page.

Upon the administrator's "Save" selection, JavaScript reads the selected staff ID and text for requested task. It then passes on these values to the storenewtask route.

```
</>JavaScript

data: {
   _token: "{{ csrf_token() }}",
   staff_id: $('#staff_id').val(),
   task_text: $('#task_text').val()
}
```

Code Snippet 34 Task Information Sent Through AJAX

The value of the token is Laravel's CSRF token. It validates the origin of the request (from the application form) and guards the route against any posts that originate from an unauthorized source.

The staff_id is used to identify the staff member who the task is to be given. The task_text is the work the administrator wants the student(s) to do.

Once a request is sent AJAX waits for the JSON that is returned by the controller. If it is successful, it will show a success message on the page, close the modal, clear the fields and update the task list.

Upon returning validation errors to the user, the page will show the appropriate message and not save the record.

Particularly, AJAX helped me to understand how to communicate between JavaScript and Laravel. The data is captured in JavaScript and the route, controller and result returned in JSON determines what is displayed on the page.

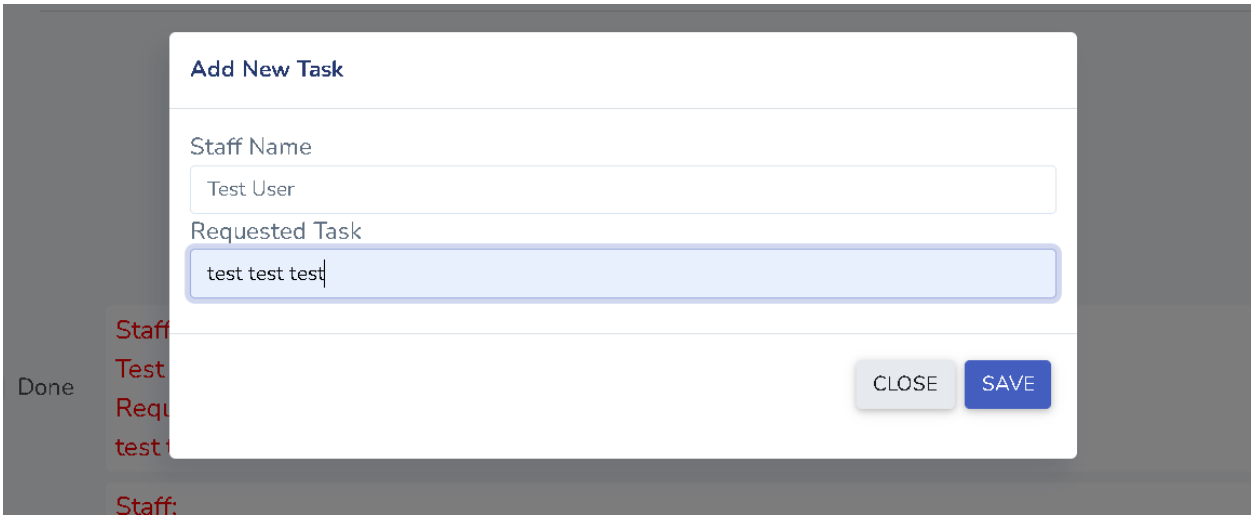

Figure 44 Add New Task Modal

3.5.5.7 Validating the Submitted Task

Validation of data on the back end is crucial as the form in the browser is not enough to ensure the correctness of the data.

The storenewtask() method checks the user input (the staff ID and the requested-task text) before storing it in the database.

```
</>PHP

$validator = Validator::make(
    $request->all(),
    [
        'staff_id' => 'required',
        'task_text' => 'required'
    ]
);
```

Code Snippet 35 Required Task Validation

Firstly, a member of the staff needs to be appointed. If this validation is not completed, the task could be saved, but not to an employee [2].

Second, Requested task must have information in the field. This helps to avoid the empty task records in the database and report [2].

If validation fails, then the controller returns the validation errors as JSON. Task record will not be created. It is important to note that the user does not leave the form, and he can modify the missing data [2], [10].

If the validation succeeds, the controller goes on to the DB operation [2], [4].

This process helps to ensure the accuracy of the Task Report. It is only when the Staff and Task value in the underlying records in the database are valid that a report can give useful information.

3.5.5.8 Saving the Task in the Database

Once validation is successful, the controller creates an instance of Staff_task model and sets the values which is passed [2], [4].

```
</>PHP

$task = new Staff_task;

$task->staff_id = $request->staff_id;
$task->text = $request->task_text;
$task->status = 1;

$task->save();
```

Code Snippet 36 Saving a New Task

The staff ID that was chosen is in staff_id and the desired job is in the text field. Status will automatically become 1 which will mean Pending [2], [4].

Because the Eloquent timestamps are enabled for the model, they are automatically stored when the model is created and updated [2].

Once the task is saved, the controller will return a Json success response. It is with that response that AJAX alerts the administrator that the task was successfully added [2], [10].

I also looked for the submitted info in the staff_tasks table via phpMyAdmin and confirmed that it was all identical. I checked that the right staff ID, task and initial status text and times were saved [5].

This database verification was crucial since a successful message is not an indication of proper value being inserted. A comparison of form, controller and database record showed that the entire back end operation was working correctly.

3.5.5.9 Connecting Tasks with Staff Records

The task table does not contain the employee name in each task record, but a staff ID. The Staff_task model will have an Eloquent relationship to get details of the staff member involved [2], [4].

```
</>PHP

public function staffs()
{
   return $this->belongsTo(
      Staff::class,
      'staff_id',
      'staff_id'
   );
}
```

Code Snippet 37 Staff Relationship in the Task Model

It's a 1:M relationship. Many tasks can be assigned to one staff member and each task is assigned to one of the selected staff members [2], [4].

The employee's name should be shown on the Task Report, however, not the number. A relationship between the staffs creates access between the controller and Blade page to the related staff record [2].

This design does not incur any unnecessary duplication. When a staff member's name changes, the name can only be changed in the staff table. All related tasks are still able to access the updated staff information via the relationship [4].

This understanding reinforced my understanding of the database normalization and Eloquent model connections.

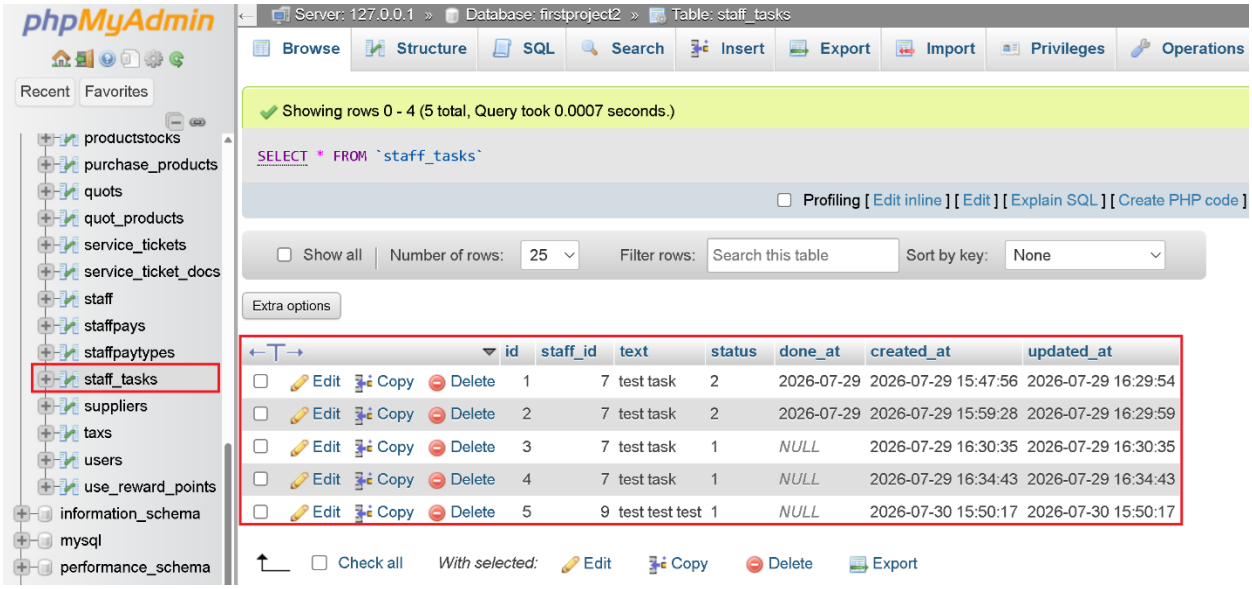

Figure 45 Task Record Stored in the staff_tasks Table

This is a task record in the staff_tasks table in phpMyAdmin. The highlighted row provides information about the task ID, staff ID, task description, status, completion date and created and updated times of the task. The status can be 1 (not completed) or 2 (completed). The done_at field is initially blank and is only filled in when the task is done by the administrator.

3.5.5.10 Marking a Task as Completed

Completed tasks can be marked as such on the Task page. On checking the task checkbox by the administrator, AJAX submits the task ID to an updatetask() controller method [2], [10].

```
</>PHP

Staff_task::where(
   'id',
   $request->catid
)->update([
   'status' => 2,
   'done_at' => date('Y-m-d')
]);
```

Code Snippet 38 Updating Task Completion

The status is changed from 1 to 2. Status 1 is Pending, status 2 is Done.

The done_at field is a date field that holds the date that the task was done. This will enable the report to display the time elapsed between task creation and completion.

The page will be refreshed with the task information after the update. The administrator can easily see the situation of the tasks and pending tasks are separated from completed tasks.

This was a great feature to have hands-on experience with an Update database activity. No deletion or replacement of the task. Laravel checks if the row exists, based on the row's id, and updates only the necessary fields.

3.5.5.11 Creating the Task Report Page

Once the task-creation and status-updated features were done, I began working on the Task Report page.

The initial goal of this page was to learn about how to access data in the database and present it in an organized manner on the front end.

On opening the Task Report, the route calls the taskreport() method in StaffController.php.

The controller first makes sure that the administrator has provided a start date and end date. If both are entered, then it gets tasks created in the selected period. If a date range is not provided, then all task records will be retrieved.

3.5.5.12 Displaying Task Statuses in Blade

The status badge is displayed using a conditional statement that correlates the value in the database with the display of a specific badge [2].

```
</>blade

@if($task->status == 1)
  <span class="badge badge-pending">
    Pending
  </span>
@else
  <span class="badge badge-done">
    Done
  </span>
@endif
```

Code Snippet 39 Displaying Pending and Done Statuses

This code is not used to determine the status of a task. The status is already stored and processed in back end. Blade only determines how it is displayed to the administrator [2].

A yellow badge is normally used for tasks that are still pending, and a green badge is used for tasks which have been completed.

The completion date is also conditionally determined. If the task is still Pending, the done_at will be empty and it will show as a dash on the report. Once the task is done, the date will be put in the format and displayed [2], [4].

This activity was very useful to me in terms of the proper separation of the controller logic from the Blade presentation. The controller fetches and formats the information and Blade determines the appearance of the information on the page [2].

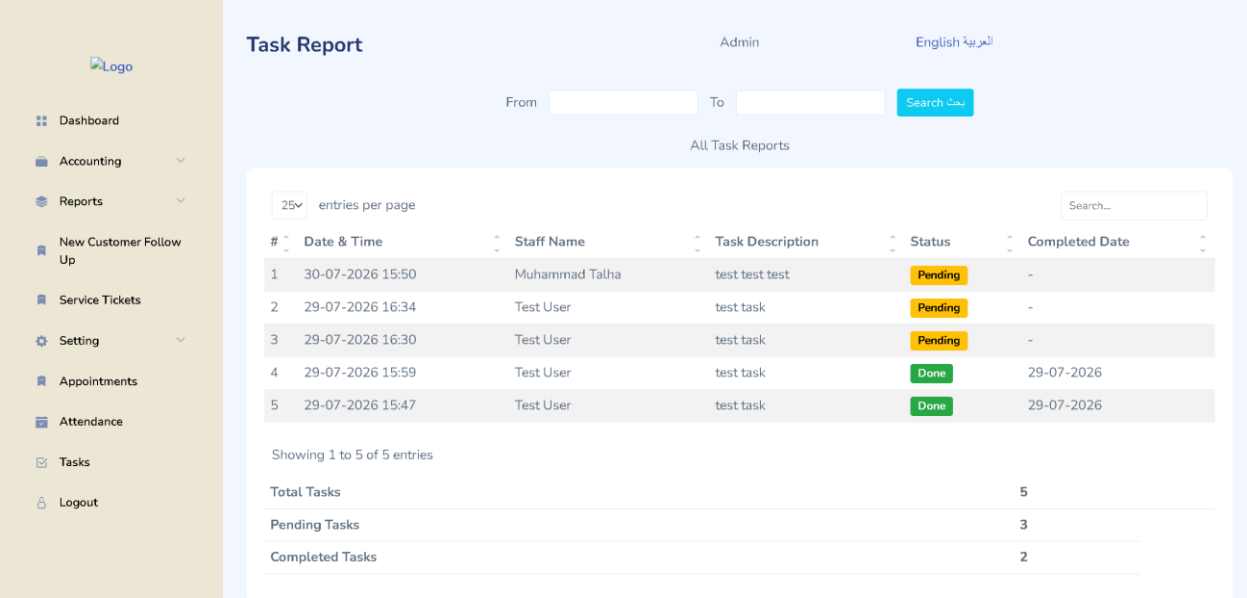

Figure 46 Task Report Page

This is the Task Report page created for displaying the information of the task that was retrieved from the database in an organized report. The page consists of start date, end date, filter button, filter text box and a table of tasks with task id, date and time created, created by, task description, task status and completed date. The report also features colored status badges to differentiate Pending and Done tasks, and show the task status. To facilitate the administrator to see the overall

progress of the tasks, summary information is shown at the bottom of the page, displaying the total number of tasks, pending tasks and completed tasks.

3.5.5.13 Fixing Basic System Errors

In Task 5, I also corrected and reviewed simple mistakes that were present in relations between the forms, routes, controler methods, database queries as well as the Blade pages.

At the time of the error, I didn't go and edit a couple of files. I followed the instructions for requesting and determined the specific component that was the issue.

Challenge Faced	**Solution Applied**
Had to get up to speed on existing reports structure	Followed the route, controller, query, model relationships and Blade view.
There were multiple tables that needed to be joined and used to provide the information needed for sales records.	Customer, payment-method and user relationships with data.
Payment records were found with foreign-key IDs	Incorporated Eloquent relationships to display values that make sense
Task form required a staff list to be provided	Displaying Form opening up, in the same way as the manual version did.
Partial work may be handed in	Validated the staff ID and task text
Task form which required to communicate with Laravel.	Identified the route and controller method to call and connected it with the AJAX request.

Task status required an initial value that was the same for all tasks.	Automatically created new tasks as Pending
There was a need to record completion information.	Added the status and done_at when the task is completed
Task Report required staff members' names.	Built up the staff relationship to each task
Generate reports with desired date filtering.	To the query, added date-range condition.
To ensure the accuracy of the report, the summary totals had to tally.	Determined them from the same collection filtered.
The same applies to empty relationship data which may impact Blade	Implement safe fallback values, where needed
Had to confirm that the data was in fact being stored	Compared page results with the records in phpMyAdmin.

Table 55 Challenges and Solutions

3.5.5.14 Testing the Complete Back-End Process

Task functions were implemented and I then tested the whole workflow. To check if the form, route, controller, validation, model, database table, report and status update operation were all linked up correctly.

Test Case	**Expected Result**
Open the Task page	A list of staff and tasks are displayed
Click on Add New Task modal	Staff dropdown and task field will appear.
Submit without a selection of staff	Validation message appears
Submit (without task text)	Validation message appears
Provide valid task information	Successfully saved task(s).

Look in the staff_tasks table	New record is created and has a status of 1.
Refresh the Task page to see the most up to date information.	New task will be listed in Pending status.
Find tasks according to staff members	Only the tasks for the employee selected are displayed
Show tasks based on their statuses.	Fixed Dividing up tasks to mark them as pending or done.
Set the status of a task to "Completed.	The status is set to 2 and done_at is set.
Click on the Task Report.	Task records stored will be shown.
Apply a date range	Matching task records only appear
Check staff names	Information about staff is correct and is displayed
Check summary totals	Total value, Pending and Done value are equal to the table.
Test - No matching records	Report is well organized and free of errors
Open Sales Report	Invoices and other information is shown.
Report on Filter Sales by date.	Correct invoices appear
Open Payment Report	The payment and related information is displayed
Records can be compared with phpMyAdmin.	These values displayed are the same as the values stored in the database.

Table 56 Task 5 Testing

It was important to test the successful and unsuccessful scenarios. Just because a valid record can be saved, doesn't mean that that function is fully tested. It should also be able to successfully handle incomplete data, process null or empty data sets, modify data sets and provide accurate report summaries.

3.5.5.15 Skills and Knowledge Developed

In this task, I have used a wide range of Laravel skills, such as routes, controller methods, Eloquent models, relationships, AJAX, JSON response, CSRF protection, Laravel validation, database insertion, database updates, date filtering, calculations on the report, Blade loops, Blade conditions, and Database verification [2], [4], [5], [10].

Also enhanced my knowledge of one to many relationships. Each task is assigned to one staff member and each staff member can be linked to a number of tasks [2], [4].

As I used the Sales and Payment Reports I was able to see that many business reports need to include data from multiple tables. The controller has to structure the information prior to it being displayed in the Blade view [2], [4].

Testing activities helped me to develop an understanding of the correlation of the values shown in the browser and the records on the server-side phpMyAdmin. This is significant as it is possible that a page might present a success message when the incorrect value has been entered into the database [5].

3.5.5.16 Task Summary

As part of Task 5, I assisted with back-end work using the Laravel framework; evaluated the current administrative reporting tools and created new task-management functionality that followed the same structure [2].

ran the Sales Report to see how the invoice records, customers, payment types, users, total and date range were handled. I also went through the Payment Report to see how payments are related to invoices, customers and payment methods with Eloquent relationships [2], [4].

I then looked at these examples and then did the Task Assignment page. This page fetches and lists the employees from the database as a drop down. The administrator chooses a staff member and requests an action to be performed and submits the form, using AJAX [2], [4], [10].

The controller checks the staff ID and requested-task text for validity [2]. If they are valid, then Laravel will create a new Staff_task record with a Pending status. If the information is not complete validation errors are returned and the task is not saved.

I also worked on the task-completion process as well. When a Pending task is selected, the ID of the task is sent to the controller, that updates its status to Done, and adds the date it was completed.

Lastly, I created and checked out the Task Report. The controller pulls the task records and staff associated with them, optionally filters based on a date range, counts the total, Pending and completed tasks and passes the information to the Blade page.

3.5.5.17 Conclusion

This task 5 was the most effective and educational to enhance my knowledge and skills in the Laravel back-end development.

I was taught that a working form is much more than HTML inputs and a Save button! Form must be associated with the proper route, the route must invoke the proper controller method, the controller method must validate the form inputs, the model must have the proper fields associated with the proper database and the result must be communicated to the user in a clear manner.

The Sales & Payment Reports gave me an insight into how information of an existing business is retrieved and organized. The Task Assignment page gave me the chance to use Database Insertion and Database validation, whereas the completion function enabled me to use update of an existing record in the database. Task Report contained records and other staff details that are retrieved and displayed with filter and calculated summaries.

This task also enhanced my skill of solving some primitive errors that occur in the system. I was able to track the request through each of Laravel's components and find out what the real issue was rather than making "random" changes.

Overall, Task 5 helped me to improve my knowledge of Laravel MVC architecture, how to pass data from forms to controller, AJAX, CSRF protection, validation, Eloquent relationships, how to perform operations in MySQL, how to provide task status updates, how to filter dates, how to develop a report, how to present info with Blade and testing and systematic debugging.

3.5.6 Task 6: Learning and Assisting with API-Related Data Communication

Task Description

Learn and assist in API-related tasks, including understanding how data is sent and received between the front end, back end, and database in a web application.

3.5.6.1 Introduction

In this assignment, I concentrated on grasping the interactions between various sections of a web app. The aim was to understand how data can pass from the front end page to the Laravel back end, how the controller interacts with the MySQL database, and how this data gets back to the browser [2], [4].

Some of the customer create/update functions have already been described in detail in Task 2 and general form, controller, validation and database operations were covered in Task 5. Hence this section does not detail those implementations. Rather, it focuses on the concepts of APIs illustrated by those functions, such as using AJAX, communicating with a HTTP endpoint, receiving JSON response, response status codes, asynchronous callback, public access with a token, and communicating with WhatsApp via a generated URL [2], [10].

This was not an entire build of a standalone REST API for another application. Rather, I dealt with inside Laravel endpoints that carried out API-like communication between the front and back end. These endpoints enabled the front end to send and receive structured responses without downloading the entire page, and also send values for further processing [2], [10].

The key use case was the WhatsApp invoicing sharing feature. The front end makes an AJAX call with an invoice's ID. Laravel will find the invoice, create or find a secure sharing token, save it in

the MySQL database, produce a public invoice URL, prepare a WhatsApp message, and return a JSON object of the generated details. The customer can then access this invoice by public read-only route without any login [2], [4], [10].

The uploaded materials show how AJAX is used for customer operations, for the test of invoice submission, for checking the database IDs, for sharing the invoice by WhatsApp, for securing tokens, for JSON responses and for retrieval of public invoices. Additionally, examples of AJAX loading, feedback from SweetAlert2, migration of sharing-token, public routes and public invoice controller method were added in the supporting report..

3.5.6.2 Laravel Routes as Application Endpoints

A route can act as an endpoint of your application. The front end issues a request to a particular URL, and the route tells the controller method to which to route [2].

Examples reviewed during this task included endpoints for:

- Filling in the necessary data for selected customers.
- Submitting customer changes.
- Saving invoice information.
- Creating a link to share an invoice.
- Opening an invoice which is public with a token.

These endpoints differ primarily in terms of the kind of request and response that they support.

A get request made by a customer will be a GET request as it is retrieving a record. Customer and invoice forms are of the POST type as they pass values to the database which can be changed. The

invoice-sharing request is a GET request to get a generated link and the public invoice route returns a full Blade page [2].

```
</>PHP

Route::get(
    'editcustomer/{id}',
    'CustomerController@editcustomerinfo'
);

Route::get(
    'invoice/share/{id}',
    'InvoiceController@generateShareLink'
);

Route::get(
    '/invoice/view/{token}',
    'InvoiceController@publicInvoiceView'
);
```

Code Snippet 40 Example Laravel Endpoints

These are examples of three different requests shown:

1. Retrieve a Customer by using an ID.
2. Creating an invoice-sharing ‘reply'.
3. Get a full public invoice with a secure token.

3.5.6.3 Understanding AJAX Communication

AJAX makes it possible to request something from the browser and get a response with AJAX without refreshing the whole page [10].

This made the interface more efficient as the administrator would not have to move back and forward between pages when customer information was being retrieved, updated or inserted into the database [10].

As mentioned in Task 2, the operations on the customer web services are CRUD operations, and these operations are already explained. They were explored in Task 6 as an instance of request-and-response communication [10].

```
</>PHP

$.ajax({
   url:
      "{{ url('/editcustomer') }}"
      + '/' + customerId,

   type: "GET",
   dataType: "json"
});
```

Code Snippet 41 Sending a Customer Retrieval Request

The key points with respect to API are:

- The URL designates the endpoint that is sought.
- The customer ID is the ID of a requested record.
- GET means that you are looking for information.
- datatype: "json" indicates to JavaScript that the data being returned is in the form of JSON.

The information that has to be entered into the customer form, the validation rules and the information being stored in the database has been defined earlier and is not repeated here.

3.5.6.4 HTTP Response Status Codes

The back end returns different HTTP status codes according to the result of a request [2].

The basic rule is that a successful request will have a status code of 200. The controller returns 404 if the requested customer/invoice doesn't exist. In general, Laravel responds with 422 if the validation fails. Occasionally, an unexpected problem with the server could result in the return of 500 [2].

Status Code	Meaning
200	The request was carried out successfully
404	The customer/invoice was not found.
422	Information provided was not valid.
500	There was an unexpected error on the server.

Code Snippet 42 Main HTTP Response Codes

3.5.6.5 Asynchronous User Feedback

User should be given a clear indication that a request is going on in the background via AJAX [10], [11].

It was displayed by using SweetAlert2:

- The request was being processed – a loading message.
- A message that is displayed when an operation is successful.
- Validation / error message if request fails.
- Options related to invoice sharing.

The loading message is particularly useful when it takes time to get information from the back end, such as the database or for creating a sharing link. If no feedback is given the user might click on the same button many times, as the page does not seem to be responding [10], [11].

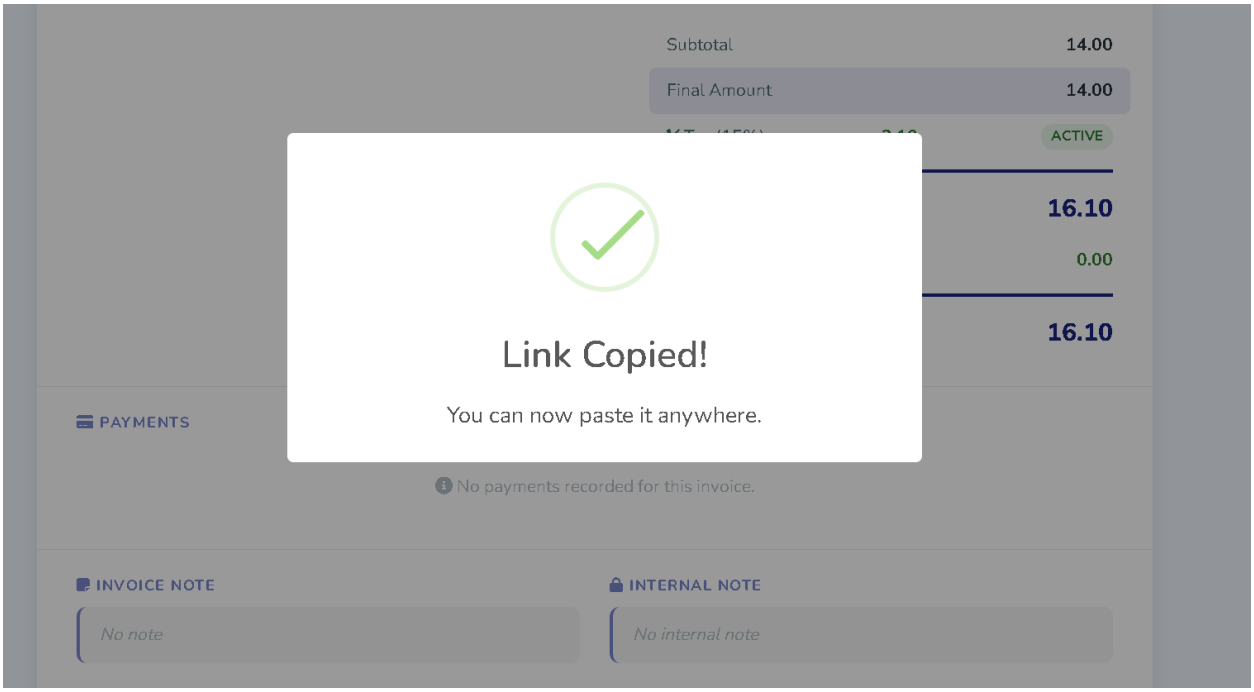

Figure 47 Asynchronous Loading and Success Feedback

This is the amount of feedback messages that are displayed during the processing of an asynchronized AJAX request. The loading indicator lets users know that the system is producing or interpreting the information they are requesting, but it doesn't need to refresh the entire web page. Following a successful operation, SweetAlert2 will pop up a confirmation message (e.g. Link Copied!) that the public invoice link has been generated and that it was successfully copied [10], [11].

3.5.6.6 Invoice Submission and Returned Identifiers

The New Invoice page was also analyzed in terms of an example related to API. Customer, product, payment-method, tax, discount and payment information is submitted to Laravel through the form.

The detailed calculations of the invoice, as well as the invoice page design, was discussed with other tasks. The key to this section is how the front end is sending the form, and how it is utilizing the response returned from the back end [2], [10].

```
</>PHP

$.ajax({
   url: "{{ url('/saveinv') }}",
   type: "POST",
   data: $('#main_form').serialize(),
   dataType: "json"
});
```

Code Snippet 43 Submitting the Invoice in the Background

The details that are submitted are processed by Laravel and the invoice is created. If successful, then the response will have the new invoice ID.

```
</>PHP

if (response.success) {
    window.location.href =
        "{{ url('salesreport/invoice') }}"
        + '/' + response.invoice;
}
```

Code Snippet 44 Using the Returned Invoice ID

This demonstrated that the data returned in the JSON, could be used to trigger a subsequent action on the front-end. The server generates the invoice and JavaScript provides the ID of the invoice to open the proper Invoice Preview page [10].

Testing was performed to ensure customer, product, payment-method and invoice Ids were linked to each other using phpMyAdmin [5]. The complete steps that will save you money on your invoices are not discussed here as they are more involved in the back-end and invoice-development duties.

3.5.6.7 Developing the WhatsApp Invoice-Sharing Endpoint

The key unique feature in Task 6 was the WhatsApp 'Invoice sharing'.

On the Invoice Preview page, the admin clicks Send by WhatsApp. InvID and the payee ID to the server.JavaScript then makes an AJAX call to the server with the invoice ID (InvID) and payee ID [10].

```
</>PHP

$.ajax({
   url:
      "{{ url('/invoice/share') }}/"
      + {{ $invoice->id }},

   method: "GET",
   dataType: "json"
});
```

Code Snippet 45 Requesting an Invoice-Sharing Link

The related route is the one that relates the request to the controller method to generate the share link.

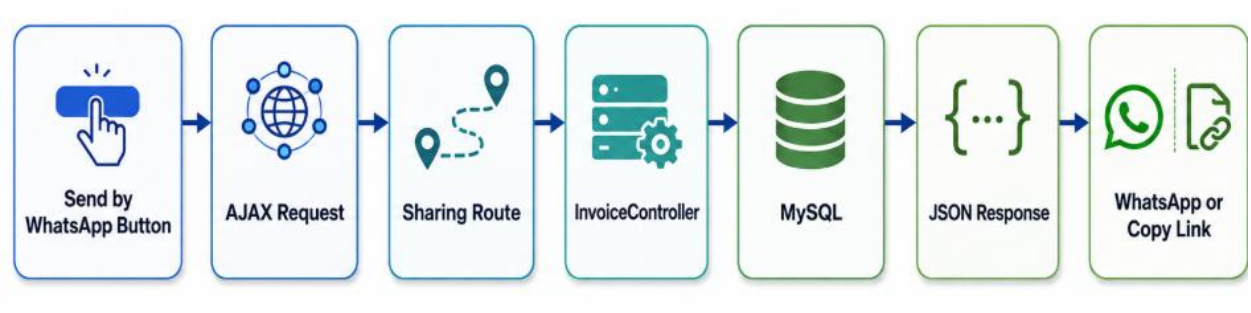

Figure 48 WhatsApp Invoice-Sharing Request Flow

This figure shows how a public link of an invoice is created and shared asynchronously with WhatsApp. On click of Send by WhatsApp button, an AJAX call is made to the Laravel sharing route. The route calls the InvoiceController that will fetch the invoice details and then either create the sharing token or read the existing one and save the necessary data to the MySQL database. The controller then responds with a JSON data with the public invoice URL and WhatsApp link. Lastly, the administrator can open WhatsApp and paste the invoice or copy the link - no need to refresh! [2], [4], [10].

3.5.6.8 Generating a Secure Invoice Token

The controller is first looking for the invoice based on its ID. If this invoice is not found, then Laravel will return a 404 JSON response [2] .

If the invoice is there, but does not have a token, then Laravel will generate a random token and save it to the invoice record [2], [3], [4].

```php
</>PHP

$invoice = Invoce::find($id);

if (!$invoice->share_token) {
    $invoice->share_token =
        Str::random(32);

    $invoice->save();
}
```

Code Snippet 46 Generating or Reusing the Token

The token is saved, so that the same invoice can be re-used with the same public URL. It is not required to create a new link each time the administrator clicks the WhatsApp button [4].

The public URL is composed from public route and token [2] .

```php
</>PHP

$publicUrl = url(
    '/invoice/view/'
    . $invoice->share_token
);
```

Code Snippet 47 Generating the Secure Public Invoice URL

The use of a token is more secure than revealing the predictable invoice ID. A user cannot simply modify the simple number in the address to ask for another invoice.

3.5.6.9 Adding the Token to the Database

A share_token attribute was added to the invoice table via a migration [2].

The token generated is added to the permanent record of the invoice and can be checked in phpMyAdmin [5].

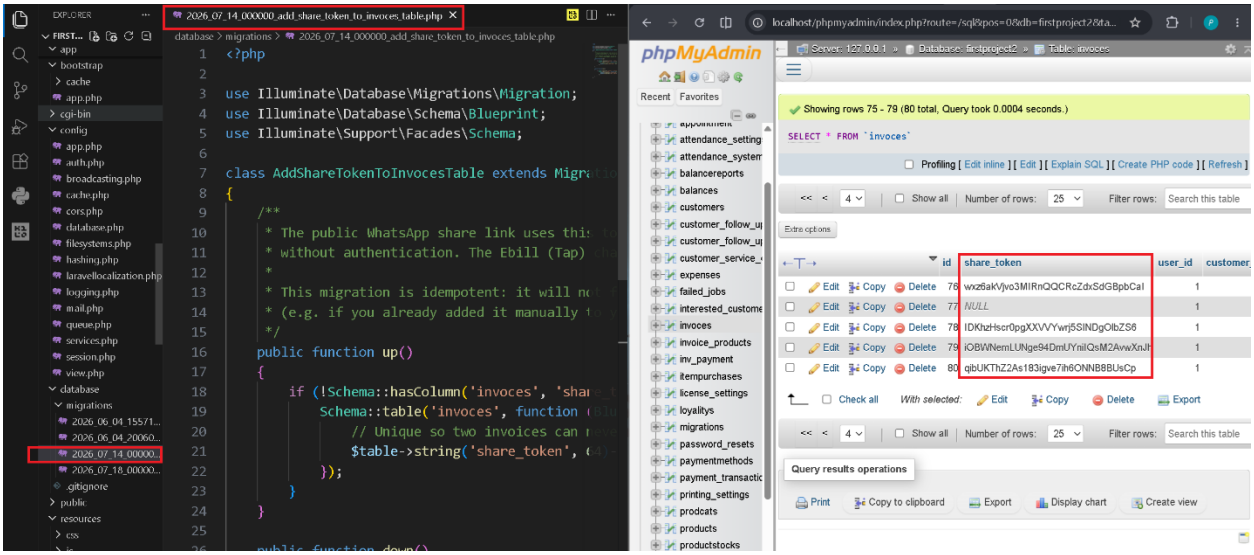

Figure 49 Share-Token Migration and Database Field

This represents the progress of the share_token functionality of the public invoice-sharing feature. On the left side, you can see the migration added by Laravel to add the share_token column to the invoices table, which is both nullable and unique. On the right side, you can see the token values generated and stored in phpMyAdmin for invoices that are shared. If an invoice has not been shared, then it can have a NULL value. A token is generated with each invoice, and can be used to create a unique public invoice link, which a customer can view without having to log in [2], [4], [5].

3.5.6.10 Preparing the WhatsApp Message

Laravel fetches the customer associated with the invoice after creating the public URL [2], [4].

The Customer Number is cleaned before being added to the WhatsApp address. Spaces, symbols and unnecessary characters are removed to make it an international, valid phone number.

The invoice message includes information that can be helpful, including:

- Customer name.
- Invoice number.
- Invoice date.
- Invoice amount.
- Public invoice link.

Only the main message structure is shown below.

```
</>PHP

$message =
   "Hello "
   . $customerName;

$message .=
   "\nInvoice #"
   . $invoice->id;

$message .=
   "\nView your invoice:\n"
   . $publicUrl;
```

Code Snippet 48 Preparing the Message

The full message also contains other information regarding the invoice.

It is important to note that the message is passed via urlencode(), prior to being added into the WhatsApp URL. This way, spaces, line breaks, and symbols can be used safely [3].

```php
$whatsappUrl =
    "https://wa.me/"
    . $customerPhone
    . "?text="
    . urlencode($message);
```

Code Snippet 49 Generating the WhatsApp Invoice Sharing URL

The system takes care of the preparation of the message, but will not automatically send the message via a separate WhatsApp Business API. It opens WhatsApp with the ready-to-go recipient and message for the administrator to review and send.

3.5.6.11 Returning Multiple Values in One Response

The invoice-sharing approach brings back multiple values in a single JSON object that are related to each other.

```php
return response()->json([
    'success' => true,
    'url' => $publicUrl,
    'whatsapp_url' => $whatsappUrl,
    'phone' => $customerPhone
]);
```

Code Snippet 50 Returning the Public Invoice and WhatsApp URLs as a JSON Response

The front end may employ:

- response.url to show or copy the public link of the invoice [10].
- response.whatsapp_url to open a WhatsApp [10].
- response.phone to show the customer number [10].

This was a good illustration of multiple related return values from a single endpoint.

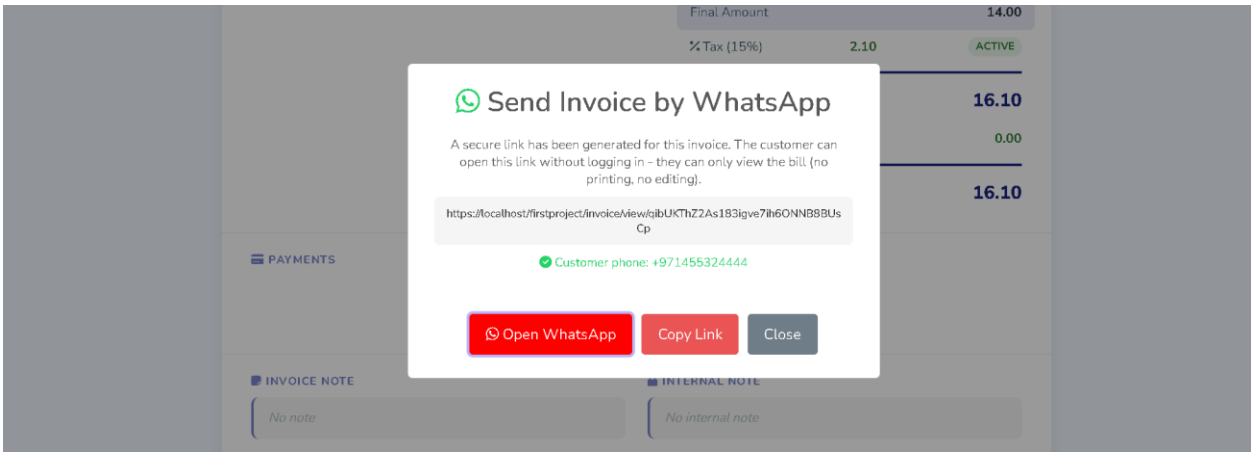

Figure 50 Generated Public Invoice Link

This is the SweetAlert2 dialog box that appears when the administrator clicks on the Send by WhatsApp button. The dialog includes the generated public invoice URL, the customer phone number, a Copy Link button, a Close button and an Open WhatsApp button. This enables the admin to simply send the invoice to the customer on WhatsApp or copy the safe public link to manually send the invoice [11].

3.5.6.12 Public and Authenticated Endpoints

There are two different routes to share the sharing function. The first route is authenticated: invoice/share/{id} and it is the one that the administrator can use to request or generate the secure link. The second is a public route: invoice/view/{token} This is viewed by the customer without the admin username/password [2].

This separation secures the application by sending to the customer just the read-only invoice page. The customer is not able to create, edit, delete or manage invoice records via the public link [2].

```php
</>PHP

Route::get(
    '/invoice/view/{token}',
    'InvoiceController@publicInvoiceView'
);
```

Code Snippet 51 Public Invoice Route

The route is given a token instead of a numerical invoice ID [2].

3.5.6.13 Retrieving the Invoice Through the Token

The public controller method will look for the invoice based on the value in share_token [2], [4].

It also loads the corresponding customer, products, payments and payment method [2], [4].

```php
</>PHP

$invoice = Invoce::with([
    'public_customers',
    'public_invoice_product',
    'public_invoice_pay.paymentmethod'
])
->where('share_token', $token)
->first();
```

Code Snippet 52 Public Invoice Query

If the token is not matched with any record, then Laravel aborts the request [2].

3.5.6.14 Public Invoice Relationships

Since the customer is not logged in, the public invoice will not require administrator's session to display [2].

Instead, Eloquent relationships were used to load the invoice information without relying on admin filtered branch-sessions [2], [4].

```
</>PHP

public function public_customers()
{
   return $this->belongsTo(
      Customer::class,
      'customer_id',
      'customer_id'
   );
}
```

Code Snippet 53 Public Invoice Relationship

Other relationships are used to find the invoice products and payment records [2], [4].

This enables the public page to show:

- Customer information.
- Products and services.
- Quantities and prices.
- Tax.
- Previous payments.
- Remaining balance.

This information can be viewed by the customer but not used for administrative activities.

3.5.6.15 Public Read-Only Invoice Page

The public invoice page was only meant to be viewed.

It displays:

- The number and date of the invoice.
- Customer information.
- Products or services.
- Quantities and prices.
- ax and total amount.
- Paid amount.
- Remaining balance.
- Payment history.
- Invoice notes.
- Company information.

There are no controls on the public page to edit, delete or manage the invoice.

This enabled the created link to be sent to the customer by WhatsApp, as the customer just gets information they need to check the invoice.

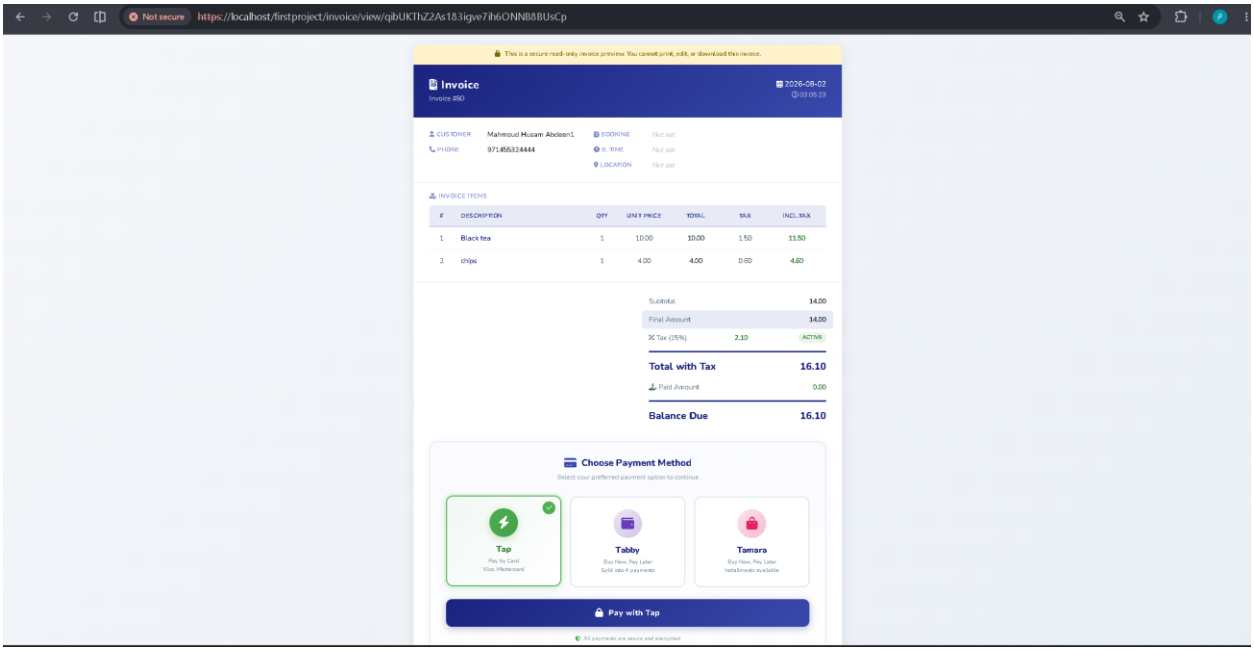

Figure 51 Public Read-Only Invoice

This is a secure public page for display of an invoice, without the need for the customer to log in, and using a unique sharing token. The invoice number and date, customer details, items purchased, quantity, price, tax value, total amount, Paid Amount, and Balance Due appear on the page. It also offers Tap, Tabby and Tamara payment options. The page is restricted for only reading; the customer can only see and pay the invoice, and not edit, delete or manage it.

3.5.6.16 Challenges and Solutions

Challenge Faced	**Solution Applied**
Complete page refreshes are required in front-end actions	Used AJAX requests
It is imperative that JavaScript get back end results in a structured manner.	Returned JSON responses
There was a need for different results of requests to be identified	The various status codes used by the HTTP response.
User required feedback while processing.	Implemented loading and result SweetAlert2 messages
Submission of invoice was reliant on accurate database IDs.	Checked ID's in phpMyAdmin
No matter how you look at it, there is no need for administrator to leave the page to generate a link.	Implemented an AJAX sharing endpoint.
It was necessary for public invoice links to be unpredictable.	Created a random sharing token.
The same link of the invoice should be used again.	Use of MySQL for storing the token
Customer started out without a superior account and had to access it.	Added a public token route.New public Token route.
There were tokens that were not valid and required safe handling.	Returned a 404 error

Public pages were not able to rely on an administrator session.	Public Eloquent relationships that have been used.
WhatsApp called for a proper international phone number format	Cleansweeped the value of the customer phone
In the front end I needed multiple generated values	Returned them as a single JSON object.
It was necessary to convey a message to the customer.	Added the invoice number and the amount, date and link.
The management functions should not be exposed to public access.	Created a Read Only Blade page

Table 57 Task 6 Challenges and Solutions

3.5.6.17 Results and Knowledge Developed

In doing so, I was able to get a hands-on experience with API related communications in Laravel. I learned about how to use AJAX in JavaScript to send a GET or POST request, how to match routes to controller functions, and how to process data and return a JSON response from controllers [2], [10].

I also learned how the use of HTTP status codes indicate if a request was successful, the data was invalid, or the request failed. The invoice-sharing feature was very useful for understanding how to generate tokens, how to store them in the database, how to make public routes, how to create URLs, how to format phone numbers and how to send multiple values in a single JSON response [2], [4], [10].

Also I knew that Laravel can respond with various types of responses. The administrator will get a JSON back with the invoice link, and the customer will open a full read only Blade invoice page.

This task helped me to get a better understanding of more complex API integration such as payment gateways [2], [10].

3.5.6.18 Task Summary

In Task 6, I got the understanding of the communication of information between the front end and the back end (Laravel) and the databases (MySQL) [2], [4].

Customer AJAX functions were examined and how the front end calls and returns data in JSON format. There was no need for repetition of the detailed customer CRUD operations, since it was covered under Task 2 [2], [10].

The process of submitting the invoice was closely examined to see the way the IDs and form values are passed to Laravel and the way the invoice ID is used to control the subsequent action on the front-end [2], [10].

The major feature was the invoice sharing feature on WhatsApp. The Invoice Preview page makes an AJAX call with the ID of the invoice. Laravel tracks down the invoice, produces or gets the safe token, stores it in MySQL, generates the public URL of the invoice, formats the client telephone number, generates the WhatsApp message, and returns the created values as JSON [2], [4], [10].

The administrator can access WhatsApp or copy the public link. The customer clicks on the link via a public link, which looks up the invoice using the token. Laravel then loads customer, products, payments, tax and company data and returns a complete read-only Blade page [2], [4].

Firstly, AJAX communication, HTTP endpoints, GET / POST requests, JSON response, HTTP status codes, token generation, URL construction, public routes, Blade response, database-relationships, security-checks and full request-flow-testing were proven [2], [3], [4], [10].

3.5.6.19 Conclusion

Task 6 gave me an increased awareness of how information flow in and out of a Laravel application can be achieved.

Out of that I learned that communication with API is not just an AJAX function. The request needs to be with the right endpoint and the right HTTP method. The controller needs to make sure the data is correct and process it, the model must know how to retrieve or change the right record in the database, and the response needs to be easily understood and a clear structure [2], [4], [10].

Customer examples showed how to communicate between Laravel and javascript using Json. The invoice-sharing feature showcased a full workflow with the generation of tokens, storing them in a database, building a URL, communicating with WhatsApp and allowing public access [2], [4], [10].

It was also the difference between authenticated and public endpoints that mattered. The safe connection is created by the administrator on the protected system and the customer accesses the protected system with the public read-only route [2].

This overall task helped me to learn more about AJAX, HTTP Requests, Laravel Endpoints, JSON, Response Codes, Asynchronous feedback, Database IDs, Secure Tokens, URL encoding, WhatsApp links, public routes, Blade responses, Eloquent relationships, Error handling, security and end-to-end data communication [2], [3], [4], [10], [11].

3.5.7 Task 7: Payment Gateway Integration with Tap, Tabby, and Tamara

Task Description

Example of payment integration with shipping companies (Tap ,Tabby and Tamara).

3.5.7.1 Introduction

In this task, I was able to work on the online payment module in the Laravel invoice system. The primary goal was to gain a better understanding of the integration of an external payment gateway with a web application and how the entire payment process from the moment the customer chooses a payment option to the receipt of the payment and storage and display was managed.

Practical and working example has been implemented using Ebill Payment Function with tap payments. This Tap integration involved building a payment request, redirecting the customer to a hosted checkout page, receiving the payment request back, checking the status of the payment and storing the successful payment, updating the invoice balance and record the payment in a separate admin report [13].

In the section where the public can pay for the invoice, I've made some changes to the Tap Payments example, and added three payment options [13], [14], [15]:

- Tap
- Tabby
- Tamara

The Tap feature was linked with the full functioning payment process. Tabby and Tamara were created via payment cards, routes, configuration, controller methods, validation on gateway,

request structures, callback routes and unavailable-service messages. They were not, however, fully activated or tested as there did not exist merchant accounts or API keys [13], [14], [15].

It's a crucial separation. Tap shows the full practical integration, Tabby and Tamara show how other payment providers are ready to go in the same Laravel integration [13], [14], [15].

Payment Gateway	Implementation Status	Work Completed
Tap Payments	Working integration	Payment Request, Hosted Payment, Callback Verification, Database record, Invoice Update, and Payment Report.
Tabby	Prepared for future activation	Payment card, route, structure of call, check of credentials, structure of call back.
Tamara	Prepared for future activation	Payment card, route, structure of call, check of credentials, structure of call back.
Online Payment Report	Implemented	Statistics, transaction table, filters, badges, details and pagination.

Table 58 Payment Gateway Implementation Status

3.5.7.2 Main Objectives

This task's primary goals were to gain insight into the entire online payment integration process and to set up the Laravel system for integration with various payment gateways [2], [13], [14], [15].

Objective	**Description**
Offer on-line means of payment for Invoices.	Enable customers to pay the balance outstanding on an invoice via online.
Ensure public access to invoices	Do not use a simple invoice ID as a sharing token, use a unique sharing token.
Adding an external API to connect with your Laravel application.	Send Customer & invoice data to Payment provider.
Use hosted checkout	Redirect customers to a safe payment page which is in the control of the provider
Verify payment results	Get and validate the status of the transactions
Store successful payments	Enter payments made into the invoice-payment table
Update invoice balances	Re-calculate Paid Amount/Balance Due
Prevent duplicate transactions	Try not to have the same callback saved multiple times.
Support multiple gateways	Show the payment options Tap, Tabby and Tamara
Protect credentials	Use the .env and configuration file in Laravel to store API setting.

Write an administrative report	Enable administrators to keep track of online-payment transactions
Improve responsiveness	Make payment cards still work on various sizes of screens

Table 59 Main Objectives of Task 7

Sources: Laravel routing documentation [2], Tap Payments documentation [13], Tabby documentation [14], and Tamara documentation [15].

3.5.7.3 Online Payment Architecture

The entire online payment flow starts by an administrator creating an invoice and providing the customer with a public link to the invoice [2], [4].

The customers opens the common invoice without logging in to the administration dashboard. The public invoice will be read-only, so the customer will be able to see the information on the invoice, but not change any of the products, customer details, payment information or invoice settings [2].

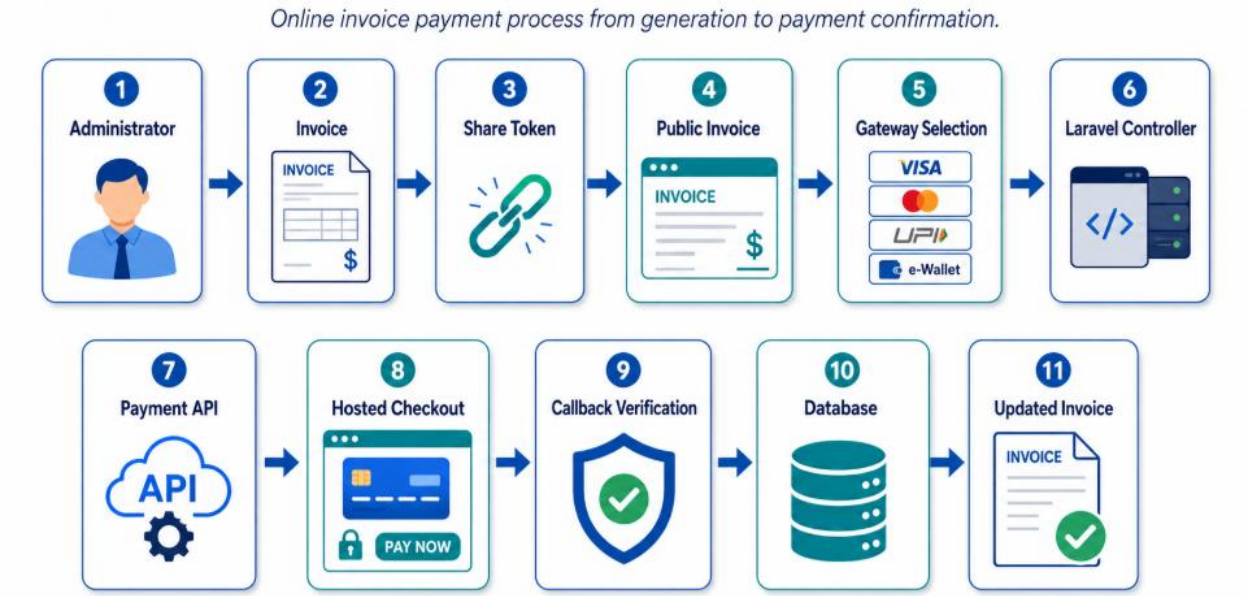

Figure 52 Complete Online Payment Integration Workflow

The entire online payment process is shown in this figure, which is done in the Laravel invoice system. It starts by the administrator creating an invoice and creating a secure share token. This token will be used by the customer to open the public invoice page where he/she can select one of the payment gateways that is available. Laravel then formats the customer and invoice details, and passes them on to the payment API used. The customer is redirected to the hosted checkout page and then he/she completes the payment. It will then return the result to the Laravel callback where the callback is verified. Upon successful payment, the payment is saved in the database, the payment amount and the remaining amount of the invoice is recalculated and the invoice is shown to the customer [2], [4], [13].

3.5.7.4 Loading the Public Invoice Through a Secure Token

The unique share_token is used to identify the public invoice. This is more secure and unpredictable than having a simple numeric identifier in the common URL that is an invoice ID.

```php
</> PHP

Route::get(
    '/invoice/view/{token}',
    'InvoiceController@publicInvoiceView'
)->name('invoice.public_view');
```

Code Snippet 54 Public Invoice Route

The route gets the token and passes it on to the publicInvoiceView() method.

```php
public function publicInvoiceView($token)
{
    $invoice = Invoce::with([
        'public_customers',
        'public_invoice_product',
        'public_invoice_pay.paymentmethod',
        'public_paymentmethod',
    ])->where('share_token', $token)->first();

    if (!$invoice) {
        abort(404, 'Invoice not found or link is invalid.');
    }

    $tax = Tax::first();

    $company = null;

    try {
        $company = Helper::getcurrency();
    } catch (\Exception $e) {
        $company = null;
    }

    return view(
        'admin.report.public_invoice',
        compact('invoice', 'tax', 'company')
    );
}
```

Code Snippet 55 Retrieving the Public Invoice

The controller gets the invoice along with its customer, products, payments and payment method. When the token isn't match with any invoice, then Laravel will return a 404 error.

The public relationships can't use the default session based branch filter because when a customer opens the shared URL they don't have an administrator session.

A sharing token also enhances the customer experience as the customer can open the invoice directly, without having to sign up or log in as an administrator. Meanwhile, the token is more secure than just a number for an invoice that increases sequentially. A unique token would make it more difficult for someone else to guess the address of an invoice, a numerical ID may be more

predictable. Even if the controller validates the token, it doesn't show anything until the validity of the token has been checked and if it is not, it will return 404. This will make the public page convenient, but have an important level of access control.

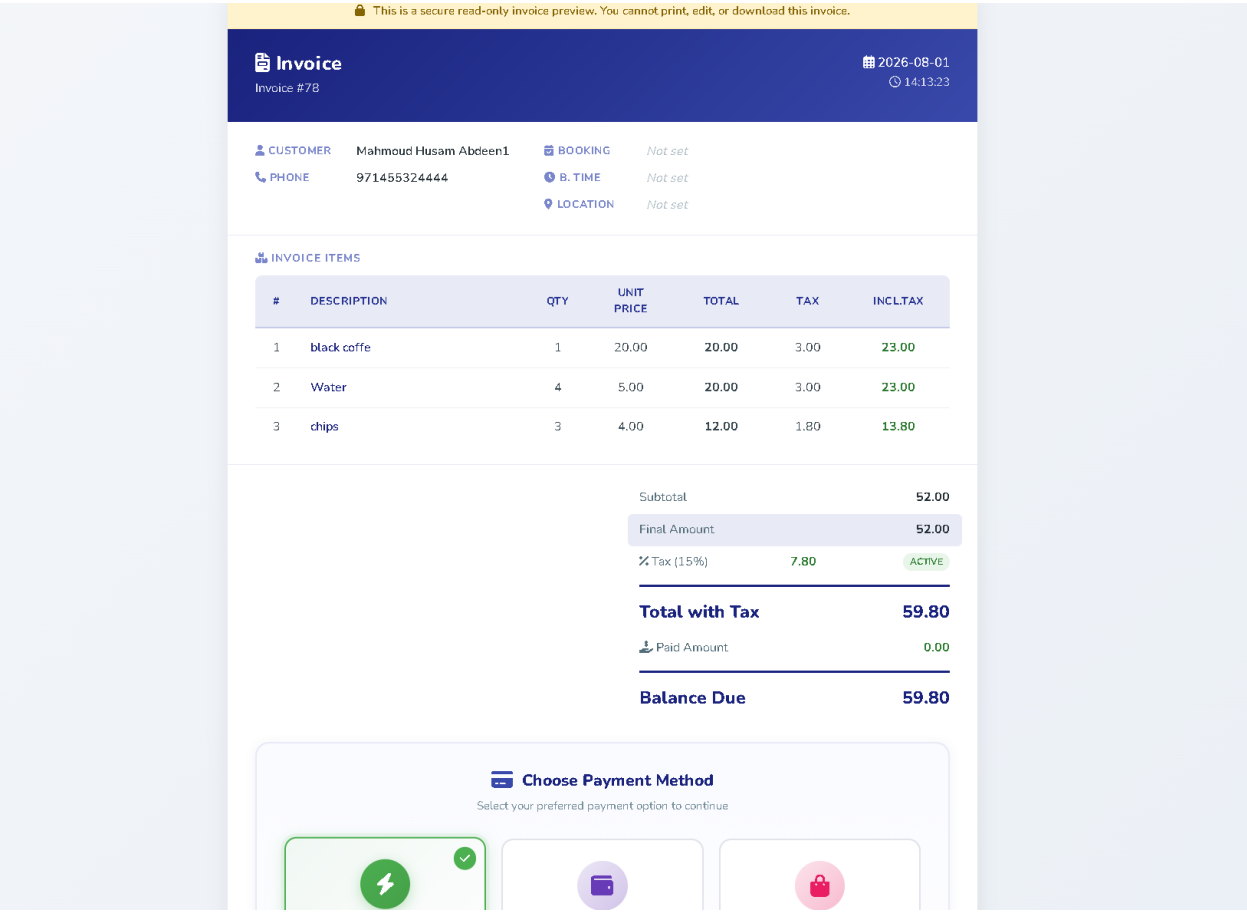

Figure 53 Secure Public Invoice Page

This is the public invoice page (read-only) which is secured and accessible by the customer via a unique sharing token. Shows the invoice number, customer information, purchased items, quantity, quantity price, tax, Total with Tax, Paid Amount and Balance Due. There are also options to pay via Tap, Tabby or Tamara. The customers are able to view the invoice with a gateway selection, but they can't edit, print or modify the invoice.

3.5.7.5 Payment Routes

Outside of the authenticated administration route group the public payment routes are placed. This is required as customers will need to be able to access them without admin login [2].

Route	HTTP Method	Purpose
/invoice/view/{token}	GET	Prints out the public invoice
/invoice/ebill/{token}	POST	Adds a Tap fee charge.
/invoice/ebill/callback	GET/POST	Receives result of the Tap payment (as a field in the Tap transaction)
/invoice/tabby/pay/{token}	GET	Initiates Tabby checkout process, set up in advance
/invoice/tabby/callback	GET/POST	The Tabby is prepared and given to him/her.
/invoice/tamara/pay/{token}	GET	Starts the checkout process for Tamara which is set-up beforehand
/invoice/tamara/callback	GET/POST	Takes the ready Tamara outcome

Table 60 Payment Route Responsibilities

Clearly, it was important to organise these routes as every payment provider needs to have a starting route, plus a callback route. The starting route gets customer's request and initiates the checkout process and the callback route gets the result of the transaction once the customer completes or cancels the payment process. This separation of the public invoice route is also done

because the primary function of the public invoice route is not to carry out any payment action, but just to display the invoice. Descriptive route names will make the controller and Blade code more readable as the application will be able to reference a route by the name rather than continually typing its full URL [2], [13], [14], [15].

GET and POST is supported for callbacks to allow for browser redirection and server to server notification [13], [14], [15].

3.5.7.6 Configuring the Payment Services

The settings for payment were added into the file config/services.php. The real private values are NOT written in the Blade templates, but are read from the file .env [2].

```php
</> PHP

'tap' => [
    'secret'   => env('TAP_SECRET', ''),
    'currency' => env('TAP_CURRENCY', 'AED'),
    'lang'     => env('TAP_LANG', 'en'),
    'base_url' => env(
        'TAP_BASE_URL',
        'https://api.tap.company/v2'
    ),
],
```

Code Snippet 56 Tap Configuration [13]

```php
</> PHP

'tabby' => [
    'secret_key' => env('TABBY_SECRET_KEY', ''),
    'public_key' => env('TABBY_PUBLIC_KEY', ''),
    'currency'   => env('TABBY_CURRENCY', 'AED'),
    'env'        => env('TABBY_ENV', 'test'),
    'base_url'   => env(
        'TABBY_BASE_URL',
        'https://api.tabby.ai/api/v2'
    ),
],
```

Code Snippet 57 Tabby Configuration [14]

```php
</> PHP

'tamara' => [
    'api_url' => env(
        'TAMARA_API_URL',
        'https://api-sandbox.tamara.co'
    ),
    'merchant_token' => env(
        'TAMARA_MERCHANT_TOKEN',
        ''
    ),
    'notification_webhook_url' => env(
        'TAMARA_WEBHOOK_URL',
        ''
    ),
    'currency'   => env('TAMARA_CURRENCY', 'AED'),
    'is_sandbox' => env('TAMARA_IS_SANDBOX', true),
],
```

Code Snippet 58 Tamara Configuration [15]

Configuration	**Purpose**
TAP_SECRET	Approves requests that are sent to Tap
TAP_CURRENCY	Changes the currency to AED
TAP_LANG	The language that the hosted checkout is in.
TAP_BASE_URL	The address of the Tap API to store
TABBY_SECRET_KEY	If the user has prepared Tabby, the app will allow him/her to make requests.
TABBY_PUBLIC_KEY	The key used to encrypt messages for Tabby. Public key for Tabby encryption.
TABBY_ENV	Chooses which environment to test in (test vs production)
TABBY_BASE_URL	Stores the Tabby API address which is the target to send results.

TAMARA_MERCHANT_TOKEN	Takes permission to ask for Tamara that has been prepared.
TAMARA_API_URL	Sandbox / Production API address to store
TAMARA_WEBHOOK_URL	Holds the notification call back URL
TAMARA_IS_SANDBOX	Chooses Sandbox / Production Mode

Table 61 Main Configuration Values

Environment variables can be used to change credentials, without changing the application logic. It also minimizes the chances of credentials being exposed in views, screenshots of source code, and so on.

This configuration structure also facilitates the migrations between the development, testing and production environments for the Laravel project. Different credential and any API address can be used for each environment, without the need to change the controller code. For instance, the testing environment could have merchant credentials in the sandbox and then at a later time, when the environment is deemed production ready, it could be replaced with live merchant credentials. The settings are also global in config/services.php which means that you don't have to duplicate settings in several controller methods. Consequently, this allows changes to be made at one place and the logic for payments remains tidier and simpler to maintain.

3.5.7.7 Payment Gateway Selection Interface

The public invoice payment area was updated to display each of the payment cards, Tap, Tabby and Tamara as individual payment cards.

The payment section will only show if:

- Balance of an invoice is not zero.
- The invoice hasn't been cancelled.

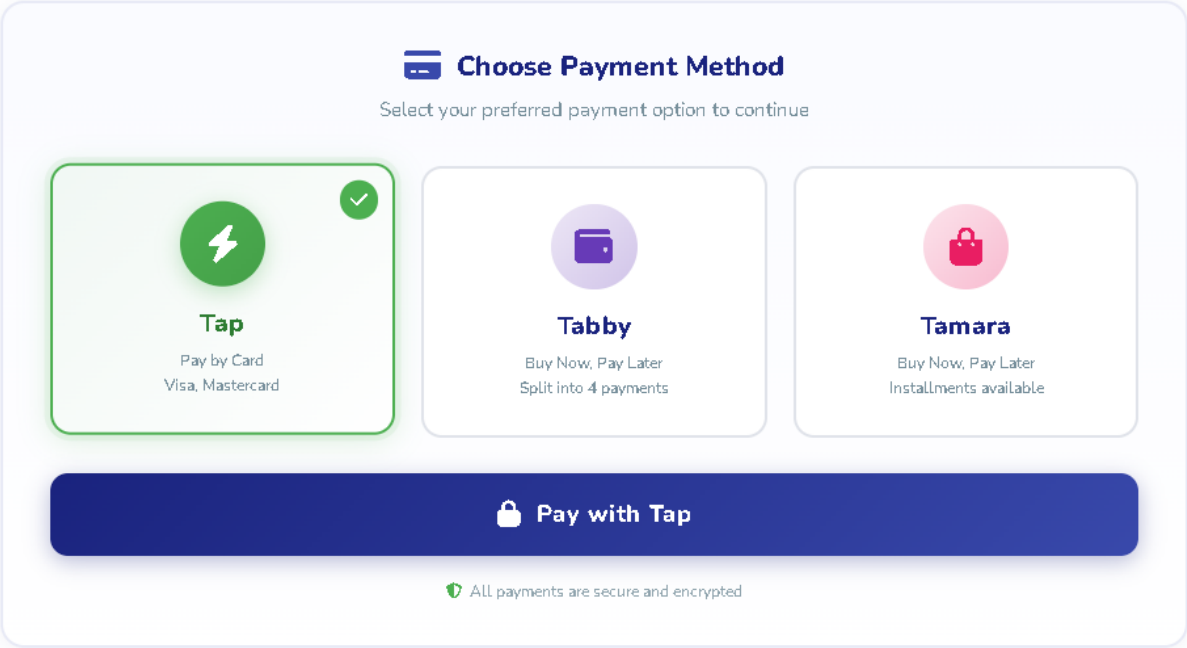

Figure 54 Tap, Tabby, and Tamara Payment Cards

This indicates the Payment Selection section of the public invoice page. It shows three payment gateway choices - Tap, Tabby and Tamara. The figure also displays the state of the selected payment option, the Pay with Tap button to proceed with online payment and the security notice that payments are secure and encrypted. This interface was created to facilitate the customer's selection of his payment method, in an intelligible and easy-to-use manner.

The design of the payment-card was designed around the idea of conveying the options to the customer without their needing to be tech-savvy about payment gates. The name of the provider and a brief description of the service the provider provides is noted on each card. The selected state provides immediate visual feedback and the disabled state means that the customer does not start an unavailable payment process. This is preferable to not showing Tabby and Tamara at all as it shows the interface that it is ready to support them in the future. It also prevents the customer from being confused by being initiated in an incomplete checkout process if the customer is missing any of the required credentials.

3.5.7.8 Payment Gateway Selection Logic

Selected card is controlled in JavaScript and based on this, which payment route should be opened is determined.

Improvement	**Benefit**
Separate cards	Helps to identify providers
Icons and descriptions	Give reasons for each answer
Selected state	Displays the current payment option
Disabled state	Does not use any providers that are unavailable
Responsive grid	Provides a way to wrap cards on smaller screens
Dynamic Pay button	Shows the provider that has been chosen
Loading state	Displays information that the request is being processed.
Supported-gateway validation	Rejects any provider names that were not anticipated

Security message	Helps to reassure the customer that the payment is secure

Table 62 Payment Card Interface Improvements

The JavaScript code enhances the usability and interaction of payment cards and payment button. The selected provider will deactivate the previous selected style and activate the new card when the customer chooses a provider. The payment-button text is then changed to make it obvious which provider has been selected. If the customer keeps going, the function either submits the Tap form or is rerouted to a prepared Tabby/Tamara route [10] The JavaScript code enhances the usability and interaction of payment cards and payment button. The selected provider will deactivate the previous selected style and activate the new card when the customer chooses a provider. This logic is then stored in a single controlled function, which will help decrease the amount of repeated code and make it easier to implement additional payment providers later. But, front-end validation is complemented by back-end validation as the values are still subject to manual changes on the client's side [10].

3.5.7.9 Creating the Working Tap Payment Request

The createEbillCharge method takes care of the working payment process [2], [13].

Prior to formulation of the charge the controller verifies that:

- The token is from a valid invoice.
- The invoice hasn't been cancelled.
- Balance of the invoice is not paid.
- Information regarding customers and invoice can be prepared [2], [13].

Separator between first and last name of the Customer. The phone number is cleaned and divided into a country code and local number. If the e-mail address is not available or is not a valid e-mail address, the code sets the "place-holder" value that is needed by the gateway [13].

Tap will be sent in accordance with the amount of invoice remaining.

Information	**Purpose**
Remaining amount	Does not allow charging the original total amount, if they have paid only part of it
Currency	Sets up the currency of the transactions as AED.
Invoice description	Locates an invoice that is associated with the purchase order
Invoice ID	Relates the transaction to the invoice.
Share token	Incorporates a call-back to the public invoice.
Customer name	Identifies the payer
Customer email	Supports transaction communication
Customer phone	Is able to relate the sale to the customer
Callback URL	Provides a payment result to Laravel.
Redirect URL	Allows the customer to return to the invoice.
threeDSecure	Allows to perform more card validation
save_card	Prevents application to store cards in their storage.

Table 63 Information Sent in the Tap Request

Source: Tap Payments API documentation [13].

Having the accurate payment information is a crucial step as it may result in the provider's rejection of the payment or the charging of the wrong amount. Hence the controller will verify the status of the invoice before generating the charge. It can't rely solely on the invoice amount since the customer might have already paid part of the invoice amount [13]. It, instead, mails the amount presently outstanding. The customer information is also formatted as per Tap's requirements. These validation and preparation steps minimize payment mistakes and keep the out-of-system payment in sync with the appropriate invoice, customer and public sharing token.

3.5.7.10 Sending the Tap API Request

Laravel will be making a POST request to the Tap charges endpoint. The payment payload is provided in JSON and a secret key is provided in the Authorization header [13].

```php
</> PHP

$baseUrl = rtrim(
    config(
        'services.tap.base_url',
        'https://api.tap.company/v2'
    ), '/');
$payload = json_encode($chargeData);

$curl = curl_init();
curl_setopt_array($curl, [
    CURLOPT_URL =>
        $baseUrl . '/charges',
    CURLOPT_RETURNTRANSFER => true,
    CURLOPT_TIMEOUT => 30,
    CURLOPT_CUSTOMREQUEST => 'POST',
    CURLOPT_POSTFIELDS => $payload,
    CURLOPT_HTTPHEADER => [
        'authorization: Bearer ' .
            $tapSecret,
        'content-type: application/json',
    ],
]);
$response = curl_exec($curl);
$err = curl_error($curl);
$httpCode = curl_getinfo(
    $curl,
    CURLINFO_HTTP_CODE
);
curl_close($curl);
$data = json_decode($response);
if (isset($data->transaction->url)) {
    return redirect(
        $data->transaction->url
    );}
```

Code Snippet 59 Sending the Tap Request and Redirecting the Customer

Customer is sent to the hosted Tap checkout page. The provider's page has the card information stored and not within a typical Laravel form.

This decreases the quantity of delicate card details worked with in the program.

Another advantage of using a hosted checkout page is that there's a separation of the Laravel system and the card-payment form. The charge is created and Laravel sends the invoice

information but the customer has to type in the card information in the provider controlled page. This means that the application will not have to design, store or process a typical card entry field. The API response is inspected for the transaction URL after the API call is complete; the customer is then redirected to this URL. When the expected URL is not present, it should not be considered successful and an appropriate error should be returned rather than continue with an incomplete transaction.

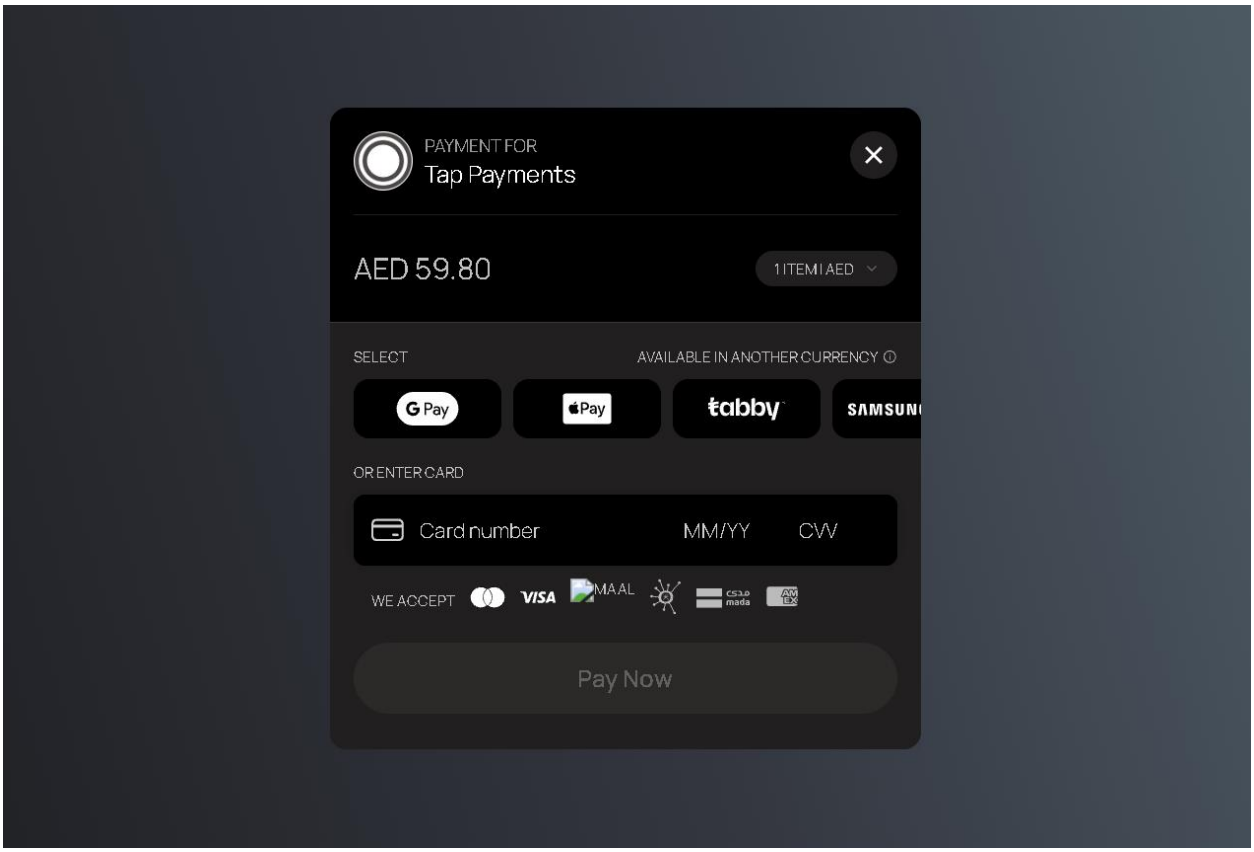

Figure 55 Tap Hosted Checkout Page

This is the secure Tap Payments hosted checkout page which opens up following the customer selecting Tap on the public invoice page. At the checkout it will show the amount that is on the invoice, payment options, card number, card expiry date, CVV fields, and the Pay Now button. Tap has its own hosted interface where the customer can input the payment details directly, instead of the Laravel app. The card information and secret API keys are not shown in the screen shot.

3.5.7.11 Receiving and Verifying the Tap Callback

Tap redirects the customer (or calls back to Laravel) after the payment attempt. The token of the invoice and Tap transaction ID are provided in the callback.

The browser redirect is not enough to be trusted by Laravel. Uses the transactionID to get the official charge details from the Tap Api.

Only payments with official status: will be used as a successful payment: CAPTURED

```php
</> PHP

if (($charge->status ?? '') === 'CAPTURED') {

    $invoice = Invoce::where('share_token', $token)->first();

    $existingPayment = Inv_payment::where('invoice_id', $invoice->id)
        ->where('pay_amount', $charge->amount)
        ->where('payment_date', date('Y-m-d'))
        ->first();

    if ($existingPayment) {
        return redirect()
            ->route('invoice.public_view', $token)
            ->with('ebill_success', 'Payment already recorded.');
    }
}
```

Code Snippet 60 Preventing Duplicate Payments

It's crucial to have the duplication check since the same payment can result in a redirect in the browser and a callback from the server. If this wouldn't be available, a single payment could be stored two times.

Status	System Response
CAPTURED	If the payment was performed - it is considered successful and banked.
INITIATED	Notifies Customer that Payment was not completed.
CANCELLED	When the customer is notified about the cancellation of payment.
FAILED	Another payment method is suggested to Customer.
TIMEOUT	Customer is requested to make another attempt
Unknown status	The payment is not considered as a successful payment.

Table 64 Tap Payment Status Handling

Callback verification is one of the crucial payment processes. A return to the invoice page is not necessarily an indication of successful payment as the return link could be manually clicked or the browser may redirect prior to the final result so as to confirm successful payment. For that reason, Laravel leverages the Tap transaction ID to get the proper charge data from the provider. Completed payments are only those that have a STATUS of CAPTURED. Other statuses will be

processed as failed or pending transactions. This way, the invoice won't be marked as paid solely on the info that is sent back via the customer's browser [13].

3.5.7.12 Allowing External Callbacks Through CSRF Protection

Laravel safeguards the POST requests with CSRF tokens. The local application's callback routes were excluded as external providers are not able to include the token generated by the local application [2].

```php
</> PHP

protected $except = [
    'whatsapp/webhook',

    'invoice/ebill/callback',
    'invoice/ebill/callback/*',

    'invoice/tabby/callback',
    'invoice/tabby/callback/*',

    'invoice/tamara/callback',
    'invoice/tamara/callback/*',
];
```

Code Snippet 61 Payment Callback CSRF Exceptions

All but the callback URLs were removed. CSRF protection is still in effect for the rest of the application routes [2].

This restricted exception is more secure than turning off CSRF for the whole payment feature or for all public routes. Requests from the Laravel app still have their generated CSRF token, and only external callback addresses are able to receive requests without CSRF token. Thus, the callback methods will have to take their own measures to verify the transaction identity, and then get the official status from the payment provider. CSRF exclusion means that the external request

is able to get into the controller, but not mean that the transaction information that is returned as a result should be trusted automatically [2], [13].

3.5.7.13 Saving the Successful Payment

Once the Tap status is confirmed and the duplicated check is passed, then Laravel will save the successful payment in the inv_payment table [2], [4], [13].

Field	Purpose
invoice_id	Is associated with the invoice
cus_id	Associates the payment with the customer
pay_amount	Saves the certain amount
pay_method	Stores the ID of the payment method
payment_date	Saves the date of payment in the database
payment_time	Stores the payment time as a number
branch_id	Associates the payment to the appropriate branch

Table 65 Main Fields Stored in inv_payment

The saving of the payment corresponds to the updating of the invoice's balance as the public invoice should reflect the payment. The payment record maintains the history of the amount, date, time, customer, payment method and branch. That is then deducted from the invoice balance. This enables the system to accept full and part payments. If the payment is complete (for the remaining balance), the due amount will become zero. If it's a lesser amount, the amount that remains unpaid is displayed on the invoice. Separate payment record will also enable you to view the full payment record later [2], [4].

3.5.7.14 Correcting the Payment Method Model

The paymentmethods table has a p_id primary key instead of the default id that is created by Laravel. The model had been set up to make sure Eloquent was able to access the created payment-method ID [2], [4].

```php
</> PHP

protected $table =
   'paymentmethods';

public $timestamps = true;

protected $primaryKey =
   'p_id';

public $incrementing =
   true;

protected $keyType =
   'int';

protected $fillable = [
   'type'
];
```

Code Snippet 62 Payment Method Primary-Key Configuration

If this is not done, the payment-method ID might not be populated when adding the Ebill payment method and thus the payment record would not be saved properly.

This was a good reminder that it is important to make sure that an Eloquent model is in line with the real database-table structure. Laravel will assume that the primary-key column's name will be id. But, the paymentmethods table has the p_id. Eloquent will not necessarily know about this difference unless told so and thus may not identify the inserted or retrieved record [2]. By specifying this primary key, data type, and incrementing behavior, the model can be used to work

with the current table without altering the DB design. This fix made it possible to use the Ebill payment method with the new payment record with the proper identifier.

3.5.7.15 Recording Detailed Online Transactions

Apart from the invoice-payment record, the system also keeps a detailed record of gateway information in the payment_transactions table [2], [4].

Field	**Purpose**
invoice_id	Locates invoice that is related to the item
customer_id	Recognizes the customer associated with the issue
gateway	Knows what Tap, Tabby or Tamara is
transaction_id	The provider transaction reference (1-100 characters)
order_reference	Records the order reference inside the store
amount	Stores the amount of the transaction.
currency	Saves AED or other currency that is supported
status	Holds for stores that are either awaiting, completed, or failed/cancelled.
customer_name	Contains the name of the person making the payment.
customer_email	Email address of the person who will pay for the item
customer_phone	Contains the payers phone number

card_last4	Only retains the last 4 digits of card number if available
charge_response	For debugging purposes, stores the provider response
callback_received_at	The time at which Laravel got the callback
paid_at	Sales at the point of payment.Sales that were booked on payment.
branch_id	Associates the transaction with the branch

Table 66 Main payment_transactions Fields

Detailed transaction table is not used for the same purpose as the basic inv_payment table. The basic table is primarily used to calculate the amount paid against an invoice and the payment_transactions table is used for administration, tracking and technical review. It has the payment provider, transaction reference, status, callback time, currency and stored provider response [4]. This separation ensures that the normal invoice-payment table doesn't get too much "Gateway" data. It also simplifies the process of generating a separate online transaction report, without having to alter the calculations in the invoice-payment method.

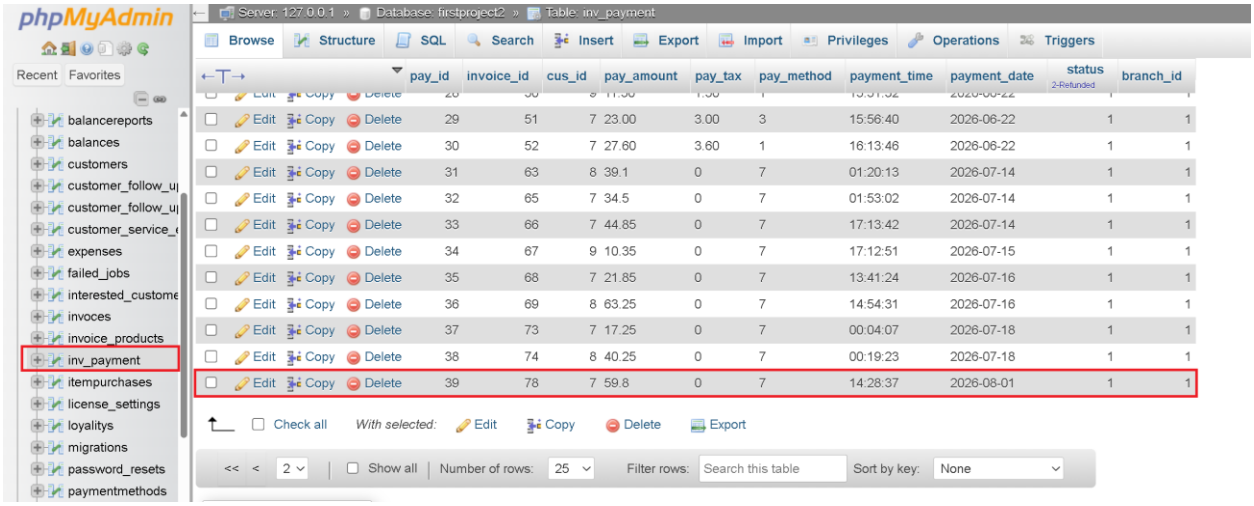

Figure 56 Successful Payment Records in phpMyAdmin

This is the figure of records made from the database after a successful payment to the Tap sandbox was made for an invoice No. 78. In part (a) we can see the new record added to the inv_payment table which contain the customer id, paid amount (AED 59.80), payment-method id, payment date, payment time, status and branch id. Part (b) shows the transaction in gateway table payment_transactions. The two records are proof that the payment is verified, it is linked to the appropriate invoice and that it was successfully saved in the database [5].

3.5.7.16 Correcting Paid Amount and Balance Due

The Paid Amount may show up as a negative during the testing process as it was based on the invoice amount (not the actual payment received).

It was changed to sum the real payment rows in inv_payment.

Value	**Calculation**
Subtotal	Original invoice amount
Discount	Amount removed from the subtotal
Final Amount	Subtotal minus discount
Tax	On sale price - including the tax.
Total with Tax	Amount after adding tax
Paid Amount	Actual records in inv_payment is the sum of the actual records.
Balance Due	Net amount due: Total with Tax - Paid Amount

Table 67 Correcting Paid Amount and Balance Due

The payment section will automatically be hidden when balance is zero.

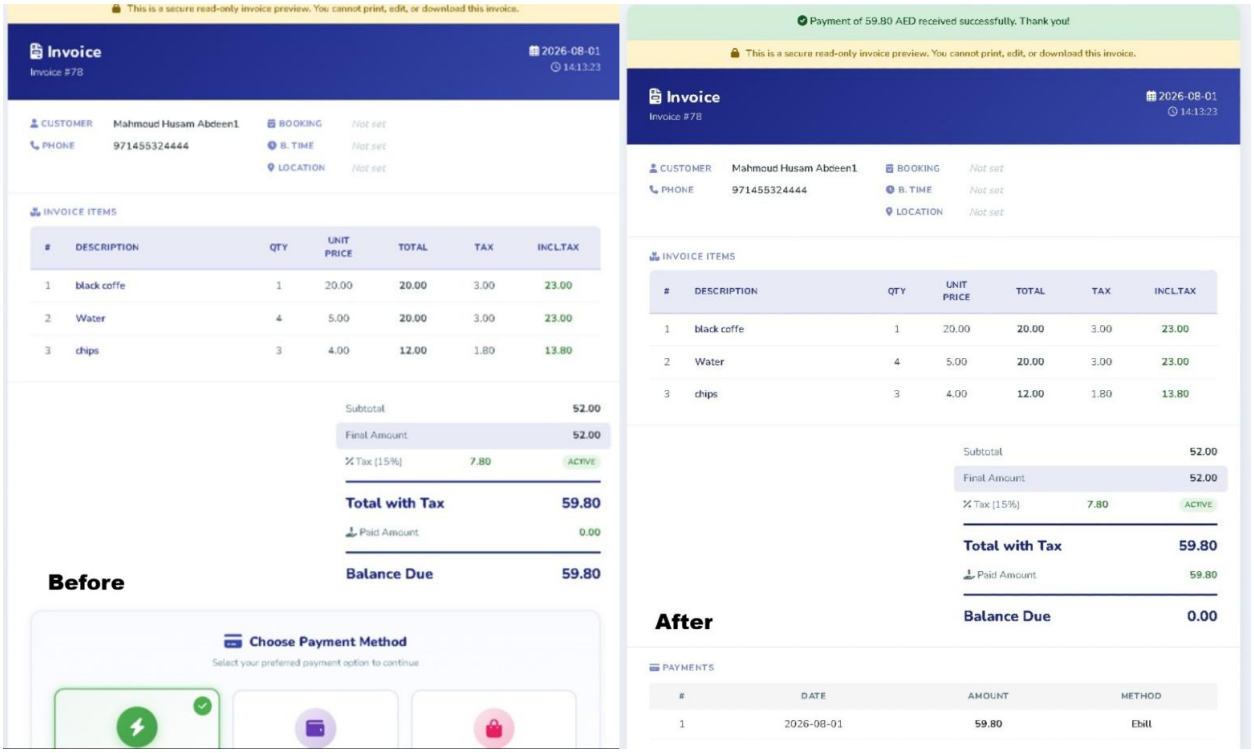

Figure 57 Public Invoice Before and After Payment

This is the figure of public invoice before and after successful Tap payment. The Paid Amount is AED 0.00, Balance Due is AED 59.80 and the payment options, Tap, Tabby and Tamara, are shown before the payment. Once the payment is verified and stored, the Paid Amount changes to AED 59.80, the Balance Due changes to AED 0.00 and the payment-selection section is automatically hidden. The successful transaction is also added to the invoice payment history and the payment was recorded and linked to the appropriate invoice.

The balance shown is particularly relevant when it comes to financial calculations; if they are correct, the customer can make another payment, if they are not, they can't. The Paid Amount may be incorrect resulting in an invalid Outstanding Balance, unnecessary second payment, or not showing the payment options as early as needed. The corrected calculation is based on the actual payment records – this serves as the source of truth. All payments made are added to the Paid

Amount - the Balance Due is the total amount of the invoice minus the Paid Amount. The max(0, ...) ensures that even if there is an unexpected value in your data or rounding makes the value negative, it will not be displayed.

3.5.7.17 Preparing the Tabby Integration

The Tabby structure had been set up in the Laravel project, but was unable to be activated due to the absence of a merchant secret key.

The controller verifies the invoice's validity, and then attempts to determine if TABBY_SECRET_KEY is set.

The ready-made Tabby payload contains:

- Outstanding invoice amount.
- Currency.
- Customer name.
- Email.
- Phone number.
- Billing & shipping address.
- Invoice reference.
- Order item.
- Success URL.
- Cancellation URL.

- Failure URL.

Component	Status
Payment card	Completed
Selection logic	Completed
Public route	Added
Controller method	Added
Credential validation	Added
Request payload	Prepared
Checkout redirection structure	Prepared
Callback route	Added
Merchant secret key	Not available
Sandbox testing	Not completed
Official payment verification	Requires completion
Successful database recording	Requires completion

Table 68 Tabby Preparation Status

Overall, the Tabby integration was not completely enabled, but the work was significant in terms of development activities to prepare its structure. It involved learning how another payment provider could fit in the already existing public invoice workflow. The prepared method will validate the invoice, check the credential, set up the needed customer and order info and specify the potential success, cancel and failure URLs [14]. The steps set the groundwork for subsequent testing. Once the merchant credentials are verified as valid, the rest of the work involves creating an actual checkout session, checking the actual result of the transaction, preventing merchant from calling back, saving the payment and adjusting the invoice balance [14].

3.5.7.18 Preparing the Tamara Integration

The Tamara structure was set up in a similar manner. The controller verifies the invoice and verifies if a merchant token is set-up [15].

The ready Tamara request includes [15]:

- Order reference.
- Total amount.
- AED currency.
- Country code.
- Instalment payment type.
- Consumer details.
- Invoice item.
- Success callback.
- Failure callback.
- Cancellation callback.
- Notification webhook.

Component	Status
Payment card	Completed
Selection logic	Completed
Public route	Added
Controller method	Added
Credential validation	Added
Checkout request structure	Prepared
Sandbox API URL	Configured
Callback route	Added
Merchant token	Not available
Sandbox testing	Not completed
Official transaction verification	Requires completion
Successful database recording	Requires completion

Table 69 Tamara Preparation Status

Source: Tamara API documentation [15].

The current Tamara callback gets the status returned. It needs to confirm the official transaction with Tamara or via a signed webhook before it is used in production [15].

The prepared Tamara implementation has the same general architecture as the other providers but the information and verification requirements are provider-specific. Order reference, amount, customer information, invoice item, payment type and callback addresses are prepared by the controller. Also validates that a merchant token exists before even trying to contact the service [15].

This prevents leaving the request unfinished, and provides a definite unavailable-service message to the customer. Complete implementation would also consider sandbox testing, official status verification, callback security checks and duplicate protection, payment storage and updating of invoice balance [15] .

3.5.7.19 Validating Supported Gateways

The controller has a gateway-validation method permitting only Tap, Tabby and Tamara [2].

```php
</> PHP

private function validateGateway($gateway)
{
    $allowedGateways = [
        'tap',
        'tabby',
        'tamara'
    ];

    if (
        !in_array(
            strtolower($gateway),
            $allowedGateways
        )
    ) {
        return false;
    }

    return true;
}
```

Code Snippet 63 Supported Gateway Validation

The payment options in the list are controlled references that are the same list as the provider list. This decreases the chance of random values coming into the payment-processing logic. A user might try to change the request using browser tools or manually change the URL; the interface normally only sends Tap, Tabby or Tamara. In order to avoid this, the controller validates itself.

This illustrates an important principle of web development: The browser is not a trusted environment, so security and business rules need to be implemented on the back end, as well as the front end to ensure good usability.

3.5.7.20 Online Payment Transactions Report

The Online Payment Transactions Report is an administrative report that offers a single page that shows payment-gateway activity. It includes the transaction id, invoice number, customer name, payment gateway, amount, currency, transaction status, transaction date and a View Details action. They also have summary cards indicating total, successful, pending, failed transactions, total successful amount and transactions via Tap, Tabby, and Tamara.

Filter	**Purpose**
Invoice number	Looks for transactions relating to an invoice.
Customer name	Looks up transactions for a customer
Payment gateway	Tap, Tabby, or Tamara, are all filters.
Status	When the status is successful, pending, failed or cancelled it will be filtered
From date	Specifies the start date for the date range.
To date	Sets the end date of the date range.
Branch	Shows the current branch's log

Table 70 Transaction Report Filters

Pagination takes the records and splits them into groups of 15 records, and keeps the active filters on each page.

The report gives an administrative overview that is not restricted to the parameters of a single invoice page or a single record in the database - online payments are included. They can see in

total how many transactions they have, how many succeeded and how many failed, they can compare the payment providers, and open up a detailed transaction if they need to see more. The search filters may help in narrowing down the search for a particular record particularly if there are lots of transactions in the table. Keeping the chosen filters across the pages during paging also makes it more user friendly as the administrator won't have to type in the same search details when switching to different pages of results.

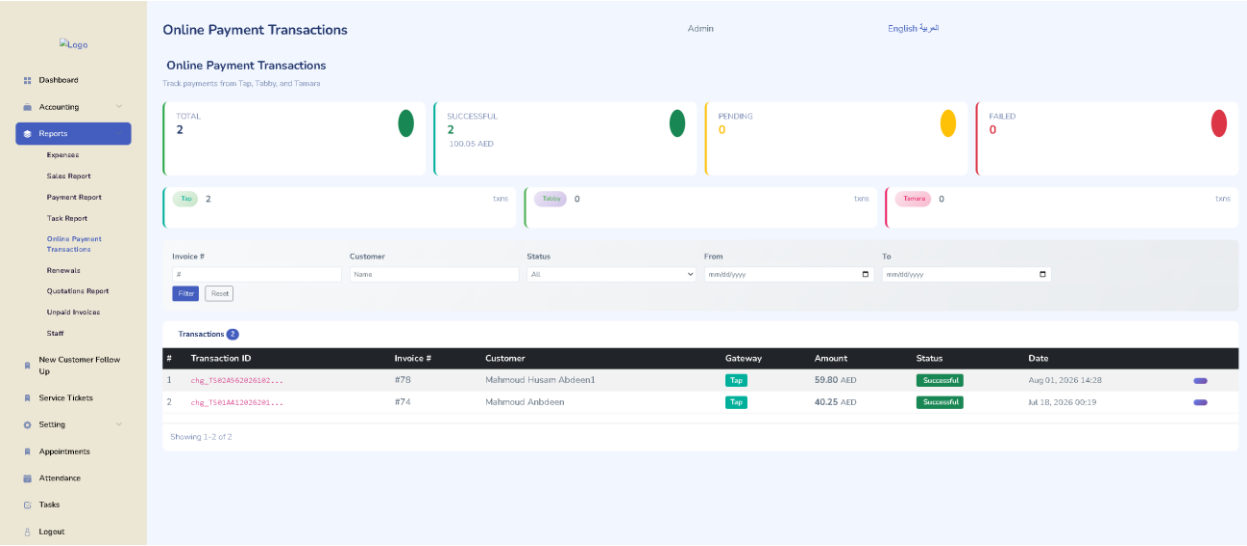

Figure 58 Online Payment Transactions Report

The summary cards give the administrator an overview of the data prior to reading through the individual cards. When, for instance, the total transaction count reveals the total number of payments made, the successful, pending and failed counts provide details of the distribution of the payment results. Successful amount is the amount confirmed via online payments. A separate Tap, Tabby and Tamara count is also included in the report, which is ready to compare the provider usage when all the gateways are active. This summary information along with detailed records allows for efficient monitoring as well as more in-depth transaction analysis.

3.5.7.21 Transaction Status and Gateway Badges

The PaymentTransaction model has the following ready-made labels and colors for their payment statuses and gateway.

Status	Badge Colour	Meaning
Pending	Yellow	Takes a wait for final result to be paid
Successful	Green	Payment was completed
Failed	Red	An error occurred with the payment.
Cancelled	Grey	Payment was cancelled

Table 71 Transaction Status Badges

Badges make the transaction report easier to read, as you can visually find out the status and who provides information. Without having to read each value in the table, an administrator can easily find the failed or pending payments. The badge text should be used in conjunction with colour as a way of communicating the result; it is still important. A combination of clear label and visual style lets the information be more easily understood. The gateway badges will also be more helpful once all three gateways are in play and more records show up in the report, to distinguish between Tap, Tabby and Tamara transactions.

3.5.7.22 Challenges Faced and Solutions Applied

Challenge Faced	**Solution Applied**
Customers were asking for an online option for paying their invoice.	Improved the way the Tap/Ebill payment function is added.
There were no administrator accounts for the public customers.	Used a secure public invoice token.Utilized a secure public invoice token.
Cancelled invoices do not need to be paid.	Verified whether a check was cancelled or not
If the invoice is paid in full, there should not be another payment initiated.	Consulted outstanding balance
In order to connect to Tap, Laravel required some changes.	Created a jsonapi request(s) via the controller
API credentials were required to be protected	Utilised Laravel setup and .env file.
Click on the required Customer Data (Must be valid)	Presented names, emails and phone numbers
Phone numbers were not in a consistent format	Eliminated unneeded characters and split up country code
The needed amount of charge was identified	Sent the outstanding amount of an invoice
Customer wanted a card page that was safe for their customers.	This means that the customer is redirected to hosted checkout.
Laravel was in dire need of confirmation.	The status of the transaction has been retrieved from Tap.
The transaction may be recorded incorrectly in case of a failure.	Successful: Accepted only CAPTURED

May receive Callback more than once	Implemented a functionality to check for any duplicate payments.
Empty Payment-method ID.	Setting p_id as primary key in model
There was a negative value in Paid Amount	Any record that has an actual payment made is summed.
Balance Due may be in the negative	Used max(0, ...)
After a complete payment, payment button still remains.	When the Balance Due was at zero, hid the section.
The customer required multiple payment options	Added Tap, Tabby and Tamara cards
Could not get tabby credentials	Developed and documented its interface and API design
It was not possible to obtain Tamara credentials.	Designed its interface & API design
Gateways other than the ones supported may be submitted.	Implemented front and back end validations.
External callbacks did not pass CSRF check.	Does not include any callback routes
There was a need for administrators to have centralized transaction monitoring.	Created the payment transaction report
It was challenging to find payment records	Support for adding an invoice, customer, status, date and gateway query.

Browsing through large transaction reports was not easy.	Added pagination
It was hard to determine payment statuses	Added colour-coded badges

Table 72 Task 7 Challenges and Solutions

3.5.7.23 Skills and Knowledge Developed

Skill	**Learning Outcome**
Laravel routing	Tethered payment actions to controller's methods
API integration	Wrote and made JSON calls to other providers
Hosted checkout	Instructed patrons to a safe provider Web page
Environment configuration	Settings for stored gateway, which are stored outside the source code of the main application.
Token-based access	Securely loaded all public invoices
Customer-data preparation	Clear and verified cellphone numbers, verified e-mail data
Callback handling	The payment responses of the browser and the server were received.
Transaction verification	Obtained the official provider status
Idempotency	Avoided double counting payment.
Eloquent models	Payment information and transactions history
Database relationships	Invoices and customers, as well as payment methods and transactions are connected.

Financial calculations	The amount has been adjusted to reflect paid amount and balance due.
Middleware configuration	Made external callbacks allowed while full CSRF protection is still in place.
JavaScript	Controlled gateway card for payments and selection.
Responsive UI	Optimized Payment Options for various Screen sizes
Error handling	Shows successful and un-successful messages.
Reporting	Create statistics, filters and paging for transactions
Technical verification	Differentiate between working functions and future integration that is prepared or not
Payment security	Prevents picking up card information from forms in Laravel.

Table 73 Skills Developed During Task 7

3.5.7.24 Task Summary

In this task, I got to work with the online payment section of the Laravel invoice system, and learned about the entire payment-gateway integration process.

Implementation of the working example was done with Tap Payments via Ebill function. The customer clicks on a secure public invoice link, checks the amount of the invoice and clicks Tap, which takes them to a hosted checkout page. Laravel pre-renders the customer information, invoice

number, outstanding amount, currency, callback URL and so on information for the payment before sending it [13].

Tap returns a callback with the transaction ID after the attempt to make a payment. Laravel gets the charge status directly from the charge service, and only records the transaction when the status is CAPTURED. If a payment check is duplicated, a duplicate payment check will not create multiple records [13].

When a payment is successful, the records are kept in the inv_payment table with the payment transactions table holding detailed information about the transactions. There is an actual payment record and the invoice Paid Amount and Balance Due are recalculated. If the entire invoice is paid, the Balance Due will be zero and the payment area will be automatically hidden [2], [4].

I also remade the part of the public invoice payment section which involves Tap, Tabby and Tamara cards. The tabby/tamara routes, configuration structures, controller methods, request payloads, callback routes, credential checks and interface logic were set up. They were not called working live integrations, however, since they did not have valid merchant credentials yet and the steps needed to verify transactions and record them in their database have yet to be done [14], [15].

Finally, I used the Online Payment Transactions Report. The report includes statistics, transaction history, status badges, search functionality, date filters, transaction details and pagination. That way, administrators can track online-payment activity all within a single page.

3.5.7.25 Conclusion

This allowed me to gain an understanding of how all these pieces fit together in an online payment system – front end, back end, API, security, database and reporting.

I discovered that there's more to payment integration than just a Pay button on a page. The reliable implementation should validate the invoice, safeguard credentials, and, of course, prepare accurate customer and amount data, securely redirect the customer to the official provider, receive the callback, verify the official provider position, prevent double entries, update the invoice accordingly, and give clear feedback [2], [4], [13].

The Tap Payments implementation was a full-fledged practical scenario of the same. It demonstrated how to integrate Laravel with an external gateway, do a hosted checkout and verify a captured payment, store the result, and update the invoice calculations [13].

The construction of the Tabby and Tamara structures also gave me insight into how to add more providers to the same structure of Laravel. Once they have their merchant accounts and their test credentials and official callback verification has been made and they have been recorded in the providers' database, their interface, routes, configuration, and request structures can be completed.

Monitoring and reporting were also shown to be important and to have been well captured in the administrative transaction report. The ability to view payment information is essential for not only customers, but also administrators who will want to view transaction ID, gateway, amount, status, date and related invoice.

In summary, this task helped me improve my Laravel API integration, secure configuration, token-based access, callbacks, payment verification, Eloquent database usage, idempotency, financial calculations, payment-interface responsiveness, transaction reporting, and debugging skills, as well as technical testing.

3.6 Summary of the 400 Training Hours

The internship was completed over ten working weeks from 1 June 2026 to 7 August 2026. Each week consisted of 40 working hours, resulting in a total of 400 training hours. Table 10 summarizes the principal activities completed during each week.

Week	Date	Activities	Hours
Week 1	1–7 June 2026	Install the Laravel project environment with XAMPP, Composer, PHP and MySQL. Enhanced Customer Management page with validation, AJAX functions, customer deletion, customer's birthday, uploading the profile-picture, and default customer avatars.	40
Week 2	8–14 June 2026	Enhanced the Multiple Customer form and created the New Invoice page. Addressed invoice printing issues, enhanced invoice history, introduced customisable tax calculations, and tested invoice, customer, product, supplier and payment method relationships.	40
Week 3	15–21 June 2026	Rewrote New Invoice page with Bootstrap button styles and made some minor fixes to the UI. Set up a safe read only public invoice page, linked to WhatsApp invoice sharing and tested all changes and database operations.	40
Week 4	22–28 June 2026	Made the WhatsApp invoice message more correct with the customer and invoice info. Created and enhanced login and registration pages using HTML, CSS, Bootstrap, and JavaScript.	40
Week 5	29 June–5 July 2026	Studied Tailwind CSS and Bootstrap components, practiced form elements, and learned how to use the Mazer Bootstrap	40

		template. Designed a practice page with buttons, cards, alerts, icons and customized UI elements.	
Week 6	6–12 July 2026	Designed responsive login pages, dashboard cards, customer tables and multi-page interfaces using Tailwind CSS and Bootstrap. Compared the two frameworks and tested the layouts on different screen sizes.	40
Week 7	13–19 July 2026	Deployed and tested Tap Payments and Ebill workflow, fixed the payment calculation on the invoice and enhanced the payment interface. Implemented payment options of Tap, Tabby and Tamara and created Online Payment Transactions Report.	40
Week 8	20–26 July 2026	Scrutineered and reviewed the entire data flow for Customer, Sales, and Payment Report. Traced through routes, controllers, models, database queries and Blade views and debugged report filters, calculations and responses.	40
Week 9	27 July–2 August 2026	Looked at the Sales and Payment Report pages and how the information is retrieved and displayed. Developed Task Assignment & Task Report pages and integrated them with the database and back end, with testing on task creation, updating the status of tasks, filtering, and results on the Task Report page.	40
Week 10	3–7 August 2026	Completed internship tasks and system improvements, revised final internship report and technical documentation, and conducted final testing of key system features to ensure they functioned properly.	40
Total	**1 June–7 August 2026**	**Ten weeks of internship training**	**400**

Table 74 Summary of the 400 Training Hours

4. How Al Ain University Prepared Me for the Internship

4.1 General Preparation

Al Ain University gave me the necessary background in programming, databases, web development, software maintenance, software testing, user-interface design, and object-oriented analysis and design which prepared me for the internship. These subjects were helpful to understand the basic concepts needed to work on an existing software project, instead of entering the internship with no technical background.

In my internship experience with Laravel Smart Solutions, I had the opportunity to put this academic knowledge to use by working with a business-management system developed with Laravel. The concepts of programming and web-development like routes, controllers, models, Blade views, forms and AJAX requests were understood. Knowledge of Databases enabled me to utilize MySQL tables, fields, primary keys, foreign keys and relationships. The ideas of software maintenance and testing assisted me in pinpointing errors, implementing changes with caution, and testing the system after every change.

The university has also helped me to grow in general professional skills. I was able to enhance my problem solving, communication, technical-writing, teamwork, independent-learning, and time-management skills through assignments, lab work, presentations, and group projects. I used these skills when looking at new code, writing new documentation for finished work, talking to my supervisor about potential issues, and doing internship work on time.

4.2 Academic Courses That Supported My Internship Experience

My internship performance was directly related to a number of subjects studied at Al Ain University. Theory was given for each subject and then put into practice for work in the Laravel business-management system.

4.2.1 Web Development

The Web Development course allowed me to understand the working of web development from front-end and back-end perspective. I have used a lot of tools that I have learnt in this course and also used throughout my internship like XAMPP, and phpMyAdmin.

At the internship I had installed XAMPP on my local machine and had started Apache and MySQL [6], I had used phpMyAdmin to check the project database. This really helped me to resolve a problem where I need to run my project on Laravel and understand how the data I want to pass to the back end from my customer page [5].

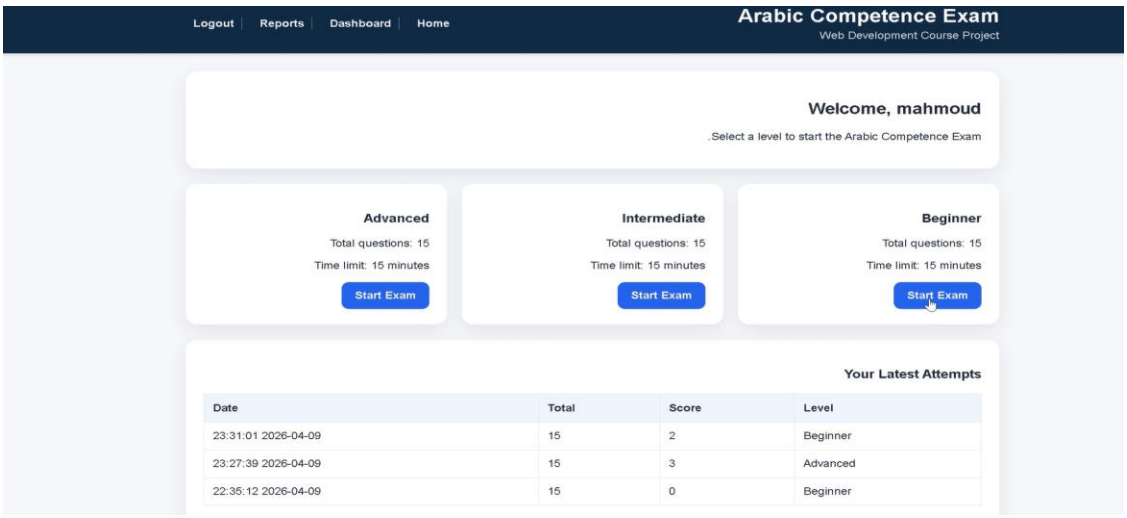

Figure 59 Web development project using XAMPP and PHPMYADMIN

4.2.2 Database

Working through the Database course was an excellent way of grasping how data is manipulated and stored by utilizing tables, columns, records and SQL statements. It provided an elementary concept of dealing with SQL (Insert, update, delete and retrieve data from the Database).

This came in handy during my internship when I was required to use the table with customers in MySQL and look at the data in the table using phpMyAdmin. I also gained an understanding of how new fields as birthday & day were added to the database and linked up with the Laravel model, controller & form.

updated_at	created_at	rewards_points	notes	profile_pic	birthday	email	trn	phone	customer_name	customer_id	
01:39:12 2026-06-05	17:49:45 2026-06-03		NULL	jpg.7_1780592056	NULL	aau.ac.ae@2020220280	NULL	971455324538	Mahmoud Husam Abdeen1	7	تعديل نسخ حذف
20:51:47 2026-06-04	16:30:56 2026-06-04		NULL	jpg.8_1780591907	2026-06-17	aau.ac.ae@2020220280	NULL	971566098787	Mahmoud Anbdeen 2	8	تعديل نسخ حذف
20:56:41 2026-06-04	16:33:56 2026-06-04		NULL	jpg.9_1780592066	2026-06-26	mkmkmk@gmail.com	NULL	971566767676	Ahmed ali	9	تعديل نسخ حذف
20:54:32 2026-06-04	16:40:18 2026-06-04		NULL	jpg.10_1780592072	2026-06-09	2020w3r4@aau.ac.ae	NULL	971677675464	Mohammed Ahmed	10	تعديل نسخ حذف
20:56:24 2026-06-04	20:47:23 2026-06-04		NULL	jpg.11_1780592078	2026-07-08	mdmggj@aau.ac.ae	NULL	971566535656	ahmed ameen	11	تعديل نسخ حذف
21:21:53 2026-06-04	21:21:53 2026-06-04		NULL	NULL	2026-06-24	mohamed@gmail.com	NULL	971348570394	Mohameed Ahmed	12	تعديل نسخ حذف

Figure 60 database table for the customer in Laravel training project

4.2.3 Software Evolution and Maintenance

I learned that software systems should be maintained, improved, and fixed after their creation from Software Evolution and Maintenance. It was very helpful as the project was not a new project, but a project I was working on Laravel.

This knowledge proved to be useful during the internship to correct the errors and incorporate new features. I need to verify the form, model, controller and database when I add a new column to the database. This enabled me to see the impact of one variable or field being passed to more than one variable or field in the system.

```php
public function customerinvoice($id)
{
    $data = [];
    $data['title'] = 'Customer Invoice History';
    $data['customer'] = Customer::find($id);
    $invoces = Invoce::where('customer_id', $id)->orderBy('id', 'desc')->get();
    $data['invoces'] = $invoces;
    $data['total_amount'] = $invoces->sum('total_amount');
    $data['tax'] = $invoces->sum('tax_value');
    return view('admin.customer.customerinvoice', $data);
}
```

Figure 61 Fixing Invoice History Method Error in CustomerController

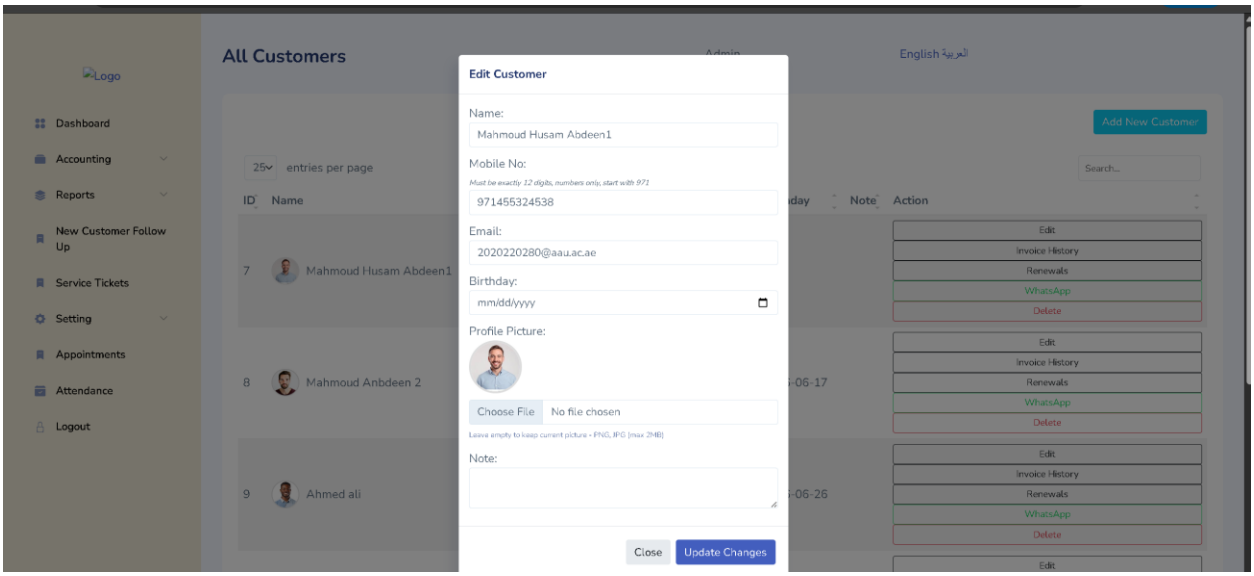

Figure 62 Invoice History Error Fix in CustomerController

4.2.4 Software Measurement and Testing

I find it very important that I use testing as an early as possible after implementing a change in the software, with the Software Measurement and Testing course I have learned this. I tested adding, editing, deleting customers, Open invoice history and saving information about birthday throughout this internship.

This course also provided an inside look at testing different scenarios instead of the right scenario. For instance, you can try out what will occur if the phone number is not correct, empty, duplicated,

or does not begin with 971. It also left me with an understanding of the need to perform input validation to minimize mistakes and problems with SQL injection, and other issues.

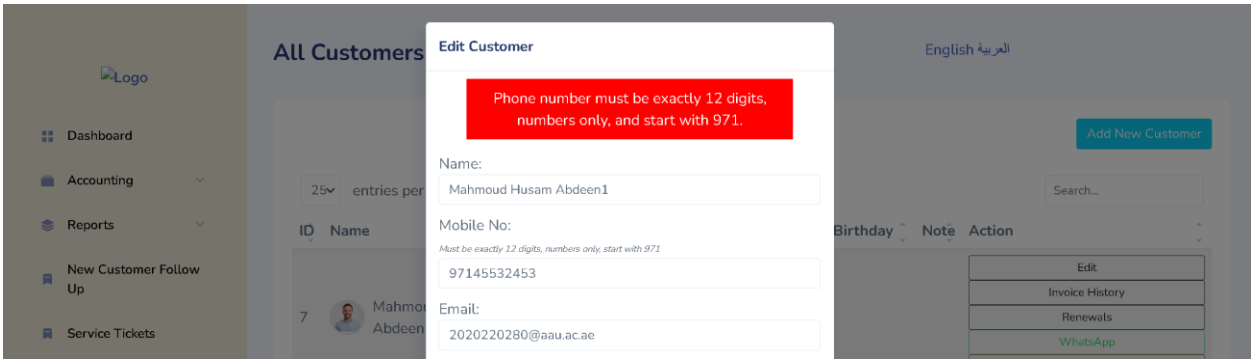

Figure 63 Phone Number Validation Test

4.2.5 User Interface

The User Interface course was an eye opener for me on the system should be clear, simple and easy to use for the user. A bad user interface such as a long form, disorganized page, confusing buttons or strange messages will lead to confusion among users.

I used this when making improvements to modal forms, button colors, validation messages, and SweetAlert2 messages during the internship. For example, I've added a loading message, so that the user will know that it's doing some processing now, so they shouldn't hit submit again. I also added another piece of code prior to deleting a customer, in order to display some confirmation message.

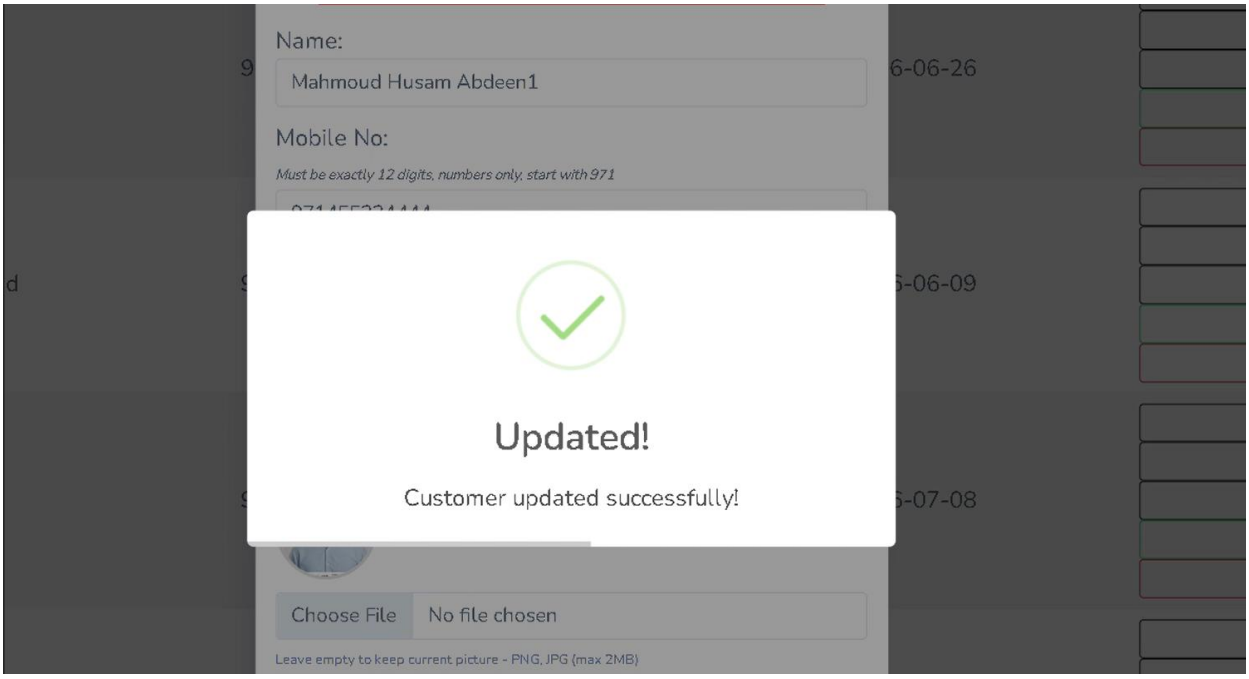

Figure 64 SweetAlert2 Loading

This message is sent to indicate that the customer information was successfully updated on submission of the edit form.

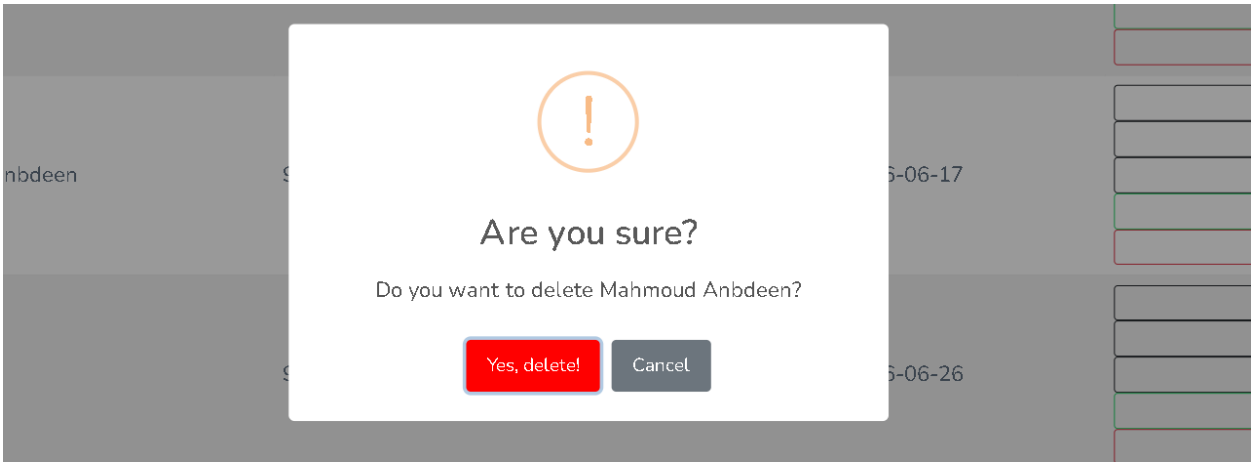

Figure 65 Delete Confirmation Messages

4.2.6 Object-Oriented Analysis and Design

The concepts of OOA and OOD helped me to understand the planning of the system by creating diagrams like class diagrams, use case diagrams, sequence diagrams.

It helped me with the understanding of the relationship between Customer model, CustomerController, customer forms and the database table as part of my internship. The "architecture" of the system can be represented using a class diagram, how the user can interact with the system can be shown using a use case diagram, and the sequence for an interaction such as editing a customer can be shown using a sequence diagram.

4.3 Summary

In general, Al Ain University gave me the academic background that I have needed to be an active participant in the internship. Programming, web development, databases, software maintenance, software testing, user-interface design, and object-oriented analysis and design gave me insights into the structure and behavior of the Laravel application.

The internship helped me to connect subjects that have been studied separately in the university. To add a customer field, for instance, necessities presented to designers included writing code logic, database changes, model and controller changes, interface changes, validation, and testing. This illustrated the interaction of various aspects of software engineering in a working system.

The University also prepared me through laboratory exercises, assignments, presentations and group projects. These activities helped me to develop my problem solving, communication, technical documentation, team working, independent learning and time management skills. This gave me a greater understanding of the code, allowed me to work on assigned projects, troubleshoot, test system changes and document results of my efforts.

5. Assessment of the Internship

The internship with Laravel Smart Solutions provided me with a priceless experience in the business world, helping me evaluate my technical expertise, hands-on skills, and professional readiness for the software engineering industry. It allowed me to test the concepts of the University in a real life Business Management application, to complete a number of development tasks set by the company, to troubleshoot technical issues and to assess my own skills which I still need to improve.

5.1 Skills and Knowledge Acquired

My Laravel Smart Solutions internship experience helped me to hone my web-development, database-management, software-maintenance, testing, and debugging abilities. I enhanced my practical skills with Laravel, PHP, MySQL, PhpMyAdmin, Blade, JavaScript, AJAX, Bootstrap, Tailwind CSS and SweetAlert2. I also got a better understanding of how all the various parts of a full web application—routes, controllers, models, Blade views, validation rules, migrations, and database tables—are interdependent [2], [4].

Here I learned how to perform CRUD, form validation, database relationships, secure tokens, public routes, admin reports and online payment integration. This meant that AJAX enabled data to be sent and retrieved, without reloading the entire page [10]. I also got to know how to read or understand existing code, find the origin of an error, make changes on connected components and test the application without degrading any existing functions.

A professional certificate was not awarded during the internship. But, it helped me gain hands-on experience and technical skills that enhanced my academic abilities and made me more prepared to apply for an entry-level software-development job.

5.2 Duties Completed During the Internship

The first thing I was asked to do was to set up the Laravel project, install the necessary packages, setup the database connection, understand the project structure, troubleshoot setup issues, and test the application on localhost.

The Customer Management module was a large contribution of my work. Created and retrieved customer records, updated and deleted customer records, enhanced phone-number validation, improved birthday storage, customer profile pictures, image previews, default avatars and Add Multiple Customers function. I also updated the Customer Dashboard; rearranged the information shown and redesigned the edit, invoice, renewal, WhatsApp and deletion actions.

Front-end work entailed enhancements to forms, buttons, cards, tables, alerts, invoice pages, login and registration pages. Also took the opportunity to look at the Sales and Payment Reports, created the Task Assignment and Task Report pages, and followed the information flow from the database and controller to the Blade interface.

Also, reviewed AJAX / JSON communications and did work on secure public sharing of invoice via WhatsApp and Tap Payments workflow. The Tap integration involved creating the payment request, redirecting the customer to the checkout page hosted by the Tap service, validating the transaction returned by the Tap service, saving successful transactions, and updating invoice balance [13]. Tabby and Tamara structures were made ready for activation, but were not activated due to the lack of valid merchant credentials [14, 15].

5.3 Effect on My Future Professional Goals

This internship helped me in solidifying my passion for software engineering particularly, web development and back-end programming.

This experience has further motivated me to develop my skills in Laravel, API, application security, automated testing, version control, and deployment to the cloud. In the future, I would like to become a full stack or back end developer, and be able to help with business-management systems and other useful web applications.

5.4 Comparison with University Learning

I had a general knowledge of the subjects I studied at Al Ain University, which helped me during the internship. The programming courses gave me insights into functions, conditions, loops, classes and objects. Database courses were helpful in handling tables, primary key, foreign key, relationship, and CRUD operations.

It took me awhile to grasp the front-end and back-end communication with web-development classes, and to check out existing code, find errors, and test changes with software testing and software maintenance classes.

During the internship I learned how these subjects can be related to a real system. For instance, when adding a new customer field, both database, model, controller and form had to be modified, validation rules had to be added, and the page display had to be updated. This was more comprehensive than working on each subject separately at University.

5.5 Differences Between Theory and Professional Experience

In university, exercises are typically well-defined, the programs are small and the desired outcomes are specified. In a professional project, there are numerous files and modules which are interconnected. I needed to study the code and verify if there were any other functions that relied on the code I wanted to change.

There was a need for further testing and focus on business needs in the area of professional development. For instance, if a customer had invoices associated with them, then he/her could not be deleted. One of the public invoice had to be available without revealing administration functions. Callback verification and duplicate-payment prevention was also needed for online payments.

The internship was an educational experience in itself because I learned that making code work one time is not a feature, it's a feature in progress. It also needs to be able to safely deal with incorrect, incomplete, database failures, and various user actions.

5.6 Hard Skills Acquired

The primary hard skills that I acquired:

- Set up and install a Laravel application with XAMPP and Composer.
- Laravel routes, controllers, models, migrations and Blade views.
- Creating CRUD operations using MySQL and Eloquent.
- Sending/receiving information without reloading the page using AJAX.
- Returning and processing JSON responses.

- Validating and dealing with HTTP status codes.
- Creating responsive interfaces using Bootstrap and Tailwind CSS.
- Creating reports that contain filters, totals and relationships with databases.
- Creating safe and secure public invoice links with tokens.
- Knowing what is a payment request, callback and transaction verification.

These skills have provided me with a more in-depth knowledge of developing and maintaining a full web application.

5.7 Soft Skills Acquired

This internship also helped me in enhancing my soft skills. My problem-solving skills grew out of my need to figure out what was really going wrong by examining the page, route, controller, model, and database.

I learned about fine attention to detail since minor errors in a field name, route, ID or database column would prevent a function from functioning. Another thing I learned was to be more patient in looking at lengthy files and trying out various solutions.

I was able to communicate with the supervisor, so I could ask questions about what to do and explain what I had done. Working on the internship report helped me to enhance my technical documentation skills. Another thing I learned is time management, how to break down big things into small and do the implementation, testing and documentation in an organized manner.

5.8 Difficulties Encountered and How They Were Addressed

The primary challenge was comprehending an extensive pre-existing Laravel application with numerous interconnected files and modules. To deal with this challenge, I tracked back each function to the route, controller, model, and database table and blade view prior to making any changes.

There were also some challenges related to local environment and database-connections. I was checking all the services of XAMPP, PHP requirements, Composer requirements, .env, application key, cached settings and MySQL connection until it worked on localhost.

It was also difficult to test connected functions because it was possible to change one component which could affect another component. I did this by trying out some successful and unsuccessful cases and verifying the result on display, the records saved in the database and then doing the test again after every correction.

One of the more complicated tasks was integration of payment: external communication, transaction-status verification, callback, duplicate-payment prevention, correct invoice calculation. I finished the Tap test workflow and set up Tabby and Tamara structures, knowing that they wouldn't be considered fully activated without merchant credentials and provider verification.

All these struggles helped me become more patient, learn to approach the problem of debugging, learn how to break down a large problem into smaller steps, and learn to do technical research.

5.9 Overall Assessment

The internship overall was beneficial and pertinent to my Software Engineering studies. It gave me the chance to use the knowledge I gained in university in a actual working environment and see how software systems are developed, tested and improved in the work environment.

So the internship made me more confident in developing using Laravel, managing the databases, communication between front-end and back-end, debugging, reporting, and API integration. It also enabled me to realize the areas I need to grow on like automated testing, application security, deployment and greater level of external API integration.

The experience gave me a taste of what it's like to work professionally by showing me the process of figuring out a project that already exists, the importance of resolving technical issues, and how to test effectively and make meaningful additions to a real business application.

6. Conclusions

The internship at Laravel Smart Solutions was a great experience and allowed me to gain practical skills in software engineering and web-application development. It enabled me to use the knowledge acquired in the classroom at Al Ain University, and work on an already existing application for business-management. The primary takeaway from this is that software development is not just the development of new software, but it's a profession. It is also used for comprehending the current code, pinpointing defects, dealing with databases, making improvements, and testing changes while ensuring that related segments of the application remain functional.

Through the internship I found that I have a better understanding of how a model–view–controller works in Laravel and how different components such as routes, controllers, models, Blade views, migrations, validation rules and MySQL database tables are related to each other [2,4]. I implemented this knowledge during the customer CRUD, Customer Dashboard, invoice pages, Sales and Payment Reports, the Task Assignment page, the Task Report, secure invoice sharing and online payment functions.

An additional key issue found was the importance of systematic evaluation and debugging. Multiple project files may be impacted by a change to a form field or column in a database. So for each of the modifications, it was necessary to trace from the front-end interface to the route, to the controller, to the model and finally to the database. Completed functions also had to robustly manage such issues as invalid information, database records not found, incorrect links, linked database records, and repeat requests. The work on payment showed very well the significance of transaction verification and the prevention of duplicate payment.

In addition, during this internship, the importance of communication among different application technologies was also observed. The back-end functionality was handled using Laravel and PHP, the content was stored in the MySQL database, dynamic pages were rendered using Blade and communication was done using JavaScript and AJAX, so that the page would not have to be completely reloaded [2], [4], [10]. This was an eye opener to me and made me realize that all the three components of the web app: front-end, back-end and database must work hand-in-hand to deliver a complete and reliable web app.

The Tap Payments implementation showed how the Laravel business system can interact with an external payment provider, send a customer to an external payment checkout page, determine the status of a payment, store the successful payment and update the associated invoice [13]. Tabby and Tamara structures were readied for future activation, but did not get to be completely activated or tested due to unobtainable merchant credentials [14], [15].

The software development sector has been evolving and developers need to upskill themselves with new technologies, frameworks, security and development methods. Business-management systems, point-of-sale solutions, customized software development, website design and e-marketing solutions are offered to different organizations by companies like Laravel Smart Solutions [1]. Technical knowledge and understanding of business requirements are essential to this industry as software systems need to be secure, reliable, usable, responsive and suitable for the daily operations of their clients.

Also noticed that software development is always a process of Maintenance and enhancements. Business demands might alter, errors might emerge, and consumers may want extra capabilities. Therefore, the developers need to be able to read the existing code, understand the

interdependencies of the system parts, make changes in a safe manner, test the changes and document what has been done. While technical programming skills are essential, the ability to communicate, problem solve, pay attention to detail and be adaptable, patient and manage time are also needed in the professional working environment.

In general, the internship was a good success in bridging the gap between the theory and practice. It gave me an opportunity to brush up my Laravel, PHP, MySQL, front end, AJAX, APIs, reporting, testing, debugging, and payment integration abilities and boost my confidence in those areas. It has also helped me to recognize other areas that needed further development such as automated testing, application security, version control, deployment and integration to production level API. Most significant, is that I had a better understanding of the software-development industry, and an even greater interest in a future career as a back-end or full-stack developer.

Appendices

Appendix A: Summary of System Changes

This appendix provides a summary of the final verification conducted with the main functions improved, reviewed or implemented during the internship. Valid and invalid input, database verification, connected records, system calculations, public access to invoice, task functions and online-payment processing were all tested.

Test Area	Verification Performed	Final Result
Local project environment	Reviewed Apache, MySQL, Composer, Laravel setup and database connection	Application ran successfully on localhost
Customer creation	Supplied valid and invalid customer data	Valid records were kept and invalid values were discarded.
Customer editing	Loaded an existing record, modified it and verified in MySQL.	Correctly recorded the updated values
Customer deletion	Tested customers that have connected and non-connected invoices.	Accidental deletion of protected records was not a problem
Profile-picture upload	Tested both images uploaded by tests and customers who don't have images.	The image and default picture were shown correctly.

Multiple-customer form	Passed multiple rows of data in different values	Valid (processed) rows and row error were detected
Invoice saving	Verified customer ID, product ID, payment ID and invoice ID.	Correct records were connected in the database
Task assignment	Got a task and queried the staff_tasks table.	Task is stored in Pending status.
Task completion	Completed an existing task	Status and Completion date were updated
Task Report	Applied date filters and summarized totals	The record and total displayed were the same.
Public invoice link	Opened a valid and invalid sharing token	The valid token displayed the invoice.
WhatsApp sharing	Created link and tested the prepared message	As expected, the correct customer and invoice information was displayed
Tap payment	Verified checkout, Call back and Payment recording	Successful payments were correctly updating the invoice.
Duplicate payment handling	Repeated callback verification	The duplication of payment records was avoided

Table 75 Summary of Main System Changes Implemented During the Internship

Appendix B: Main Database Changes

The following is a breakdown of the key fields added, modified, or utilized in the database for the customer-management, task-management, public invoice-sharing, and online-payment features implemented as part of the internship.

Database Table	Field	Purpose
customers	birthday	Records customer's birthday
customers	profile_pic	This field stores the path of the profile-image of the customer.
invoices	share_token	Holding a unique token for accessing public invoice.
staff_tasks	staff_id	Matches each task with a member of staff
staff_tasks	text	Saves the Task Description that was assigned
staff_tasks	status	Determines if the task is Pending or Done
staff_tasks	done_at	Records the date the task was completed

Table 76 Summary of Database Changes Introduced During the Internship

These alterations were tested with Laravel migrations, Eloquent models, controller operations and via phpMyAdmin.

References

[1] Laravel Smart Solutions, “Custom Software Development and Business Management Systems,” *Laravel Smart Solutions*. [Online]. Available: https://laraveluae.com/en. [Accessed: Aug. 8, 2026].

[2] Laravel, “Laravel 10.x Documentation,” *Laravel Documentation*. [Online]. Available: https://laravel.com/docs/10.x. [Accessed: Aug. 8, 2026].

[3] The PHP Group, “PHP Manual,” *PHP Documentation*. [Online]. Available: https://www.php.net/manual/en/. [Accessed: Aug. 8, 2026].

[4] Oracle Corporation, “MySQL 8.0 Reference Manual,” *MySQL Documentation*. [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/. [Accessed: Aug. 8, 2026].

[5] phpMyAdmin Contributors, “phpMyAdmin Documentation,” *phpMyAdmin*. [Online]. Available: https://docs.phpmyadmin.net/en/latest/. [Accessed: Aug. 8, 2026].

[6] Apache Friends, “XAMPP,” *Apache Friends*. [Online]. Available: https://www.apachefriends.org/download.html. [Accessed: Aug. 8, 2026].

[7] Composer, “Composer Documentation,” *Composer*. [Online]. Available: https://getcomposer.org/doc/. [Accessed: Aug. 8, 2026].

[8] Bootstrap Team, “Get Started with Bootstrap,” *Bootstrap v5.3 Documentation*. [Online]. Available: https://getbootstrap.com/docs/5.3/getting-started/introduction/. [Accessed: Aug. 8, 2026].

[9] Tailwind Labs, “Installing Tailwind CSS,” *Tailwind CSS v3 Documentation*. [Online]. Available: https://v3.tailwindcss.com/docs/installation. [Accessed: Aug. 8, 2026].

[10] OpenJS Foundation, “jQuery.ajax(),” *jQuery API Documentation*. [Online]. Available: https://api.jquery.com/jQuery.ajax/. [Accessed: Aug. 8, 2026].

[11] SweetAlert2, “SweetAlert2 Documentation,” *SweetAlert2*. [Online]. Available: https://sweetalert2.github.io/. [Accessed: Aug. 8, 2026].

[12] Fonticons, Inc., “Font Awesome Documentation,” *Font Awesome*. [Online]. Available: https://docs.fontawesome.com/web/. [Accessed: Aug. 8, 2026].

[13] Tap Payments, “Charges API,” *Tap Payments Developer Documentation*. [Online]. Available: https://developers.tap.company/reference/charges. [Accessed: Aug. 8, 2026].

[14] Tabby, “API Reference Overview,” *Tabby Developer Documentation*. [Online]. Available: https://docs.tabby.ai/api-reference/overview. [Accessed: Aug. 8, 2026].

[15] Tamara, “Tamara API Reference Documentation,” *Tamara Developer Documentation*. [Online]. Available: https://docs.tamara.co/reference/tamara-api-reference-documentation. [Accessed: Aug. 8, 2026].