CHAPTER 1

ABSTRACT

The Invoice Generation Application is a robust and user-friendly mobile application designed to simplify and automate the process of creating, managing, and tracking invoices. Developed using Android Studio with Java/Kotlin for the frontend, the app provides an intuitive interface for users to generate detailed invoices efficiently. The backend is powered by the Spring Boot framework, ensuring a scalable and secure system that supports seamless communication through RESTful APIs.

Data management is handled using MySQL or PostgreSQL databases, enabling reliable and structured storage of client, product, and invoice data. Postman is utilized for rigorous API testing to ensure the application's reliability and functionality. Git facilitates version control for effective team collaboration, while Maven/Gradle manages dependencies for smooth development and deployment. This application is tailored for businesses of all sizes, offering features like invoice customization, real-time status updates, and secure data handling.

INTRODUCTION

The Invoice Generation Application is a mobile-based solution developed to streamline the process of creating and managing invoices for businesses. Designed using Android Studio with Java/Kotlin, the app offers an intuitive and user-friendly interface for businesses to generate accurate and professional invoices effortlessly. Its responsive design ensures accessibility across various devices, making it suitable for small and medium-sized enterprises.

The backend leverages the Spring Boot framework, providing a robust and scalable infrastructure. RESTful APIs facilitate seamless interaction between the frontend and backend, enabling real-time data synchronization. The application integrates MySQL or PostgreSQL databases to securely store and manage client information, invoice details, and transaction records.

Development tools like Postman ensure comprehensive testing of APIs, while Git supports effective version control and collaboration among developers. Dependency management is handled using Maven or Gradle, simplifying project configuration and build processes. With features such as customizable templates, automated calculations, and payment tracking, the Invoice Generation Application is a complete solution for modern invoicing needs.

CHAPTER 2

SYSTEM ANALYSIS

2.1 EXISTING SYSTEM

The existing systems for invoice generation typically involve manual or semi-automated processes. Businesses often rely on spreadsheet software, generic accounting tools, or pre-built templates for creating and managing invoices. While these methods may suffice for small-scale operations, they lack the efficiency, flexibility, and integration capabilities required by growing businesses.

Manual processes are time-consuming, prone to errors, and lack features such as automated reminders, real-time tracking, and secure cloud storage. Moreover, existing software solutions may require extensive customization or additional costs for integrating advanced functionalities. They also lack the mobile accessibility and user-friendly interface offered by a dedicated invoice generation application.

2.1.1 DRAWBACKS

The existing system has following disadvantages,

I. **Time-Consuming and Error-Prone:** Manual data entry and calculations increase the likelihood of errors and consume significant time.
II. **Limited Accessibility:** Existing systems often lack mobile support, restricting access and updates on the go.
III. **Integration Challenges:** They do not seamlessly integrate with other tools like payment gateways or CRM systems, limiting functionality.
IV. **Lack of Scalability:** Most existing systems are not scalable, making them unsuitable for businesses with expanding operations or complex invoicing needs.

2.2 PROPOSED SYSTEM

The proposed Invoice Generation Application is a mobile-based, automated solution designed to address the inefficiencies of existing systems. Built using Android Studio with Java/Kotlin for the frontend and Spring Boot for the backend, this application provides an intuitive, scalable, and secure platform for creating, managing, and tracking invoices. It supports customizable templates, automated calculations, and integration with payment gateways, offering a comprehensive invoicing solution.

The application ensures real-time data synchronization via RESTful APIs, while a robust database using MySQL or PostgreSQL securely stores client, invoice, and transaction data. Mobile accessibility allows users to generate and manage invoices on the go. Advanced features such as automated reminders, multi-user access, and analytics reporting are included to enhance operational efficiency. The use of Postman for API testing and Git for version control ensures a reliable and collaborative development process.

2.2.1 FEATURES

The proposed system has following advantages,

I. **Enhanced Efficiency:** Automation reduces manual effort, minimizing errors and saving time.
II. **Mobile Accessibility:** Enables invoice generation and management from anywhere, enhancing flexibility.
III. **Integration Support:** Seamlessly integrates with payment gateways and other business tools.
IV. **Scalable and Secure:** Adapts to growing business needs while ensuring secure data handling.

CHAPTER 3

SYSTEM SPECIFICATION

3.1 HARDWARE CONFIGURATION

This section gives the details and specification of the hardware on which the system is expected to work.

- Processor : Intel(R) Core(TM) i5
- RAM : 8 GB minimum
- Hard Disk : 512 GB minimum
- Operating System : Windows 10

3.2 SOFTWARE SPECIFICATION

This section gives the details of the software that are used for the development

- Programming Language : Python, HTML, CSS, Js Bootstrap, Django
- IDE/Workbench : PyCharm, visual Studio code.

CHAPTER 4

SOFTWARE DESCRIPTION

4.1 FRONT END

Java

Java is a class-based, object-oriented programming language that has been one of the most popular languages for Android app development. It is known for its portability, security features, and scalability, making it suitable for building robust and large-scale applications. Java is platform-independent due to its "Write Once, Run Anywhere" (WORA) philosophy, enabled by the Java Virtual Machine (JVM). While Kotlin has emerged as the preferred language for Android, Java remains widely used in Android development due to its extensive documentation, community support, and compatibility with a wide range of libraries.

Java Features:

Object-Oriented: Supports core OOP concepts like inheritance, polymorphism, and encapsulation, which make code reusable and maintainable.

Platform Independence: Java code can run on any platform with a JVM, ensuring portability across devices.

Robust: Java includes strong memory management features, automatic garbage collection, and built-in exception handling.

Multi-threading: Java provides built-in support for multi-threaded programming, allowing for efficient concurrent execution of tasks.

Rich Standard Library: Includes a vast library of built-in APIs for various tasks like networking, file handling, and UI development.

Security Features: Java includes extensive security mechanisms such as access control, cryptography, and authentication.

Automatic Memory Management: Uses garbage collection to manage memory automatically, reducing the risk of memory leaks.

Performance: Java offers high performance with Just-In-Time (JIT) compilation, which compiles bytecode into machine code at runtime.

Community Support: A vast, active developer community offers extensive resources for troubleshooting and learning.

Scalability: Java is highly scalable and widely used in enterprise-level applications, making it suitable for large-scale projects.

Kotlin

Kotlin is a modern, statically typed programming language developed by JetBrains. It is fully interoperable with Java, meaning you can use both languages in the same project without any issues. Kotlin was officially supported for Android app development by Google in 2017, and it has quickly become the preferred language for Android developers due to its concise syntax, null safety, and enhanced features. Kotlin reduces the amount of boilerplate code, making it more readable and easier to maintain, while still maintaining the power and flexibility that Java offers.

Kotlin Features:

Concise Syntax: Kotlin's syntax is more compact than Java, which reduces boilerplate code and makes code easier to read.

Null Safety: Kotlin introduces nullable and non-nullable types, preventing null pointer exceptions at compile time.

Interoperability with Java: Kotlin is fully interoperable with Java, enabling developers to use both languages in the same codebase seamlessly.

Smart Casts: Kotlin automatically casts objects to their appropriate types when safe, reducing the need for manual casting.

Extension Functions: Allows adding new functionality to existing classes without modifying their source code.

Coroutines: Kotlin provides built-in support for coroutines, making asynchronous programming easier and more efficient.

Data Classes: Kotlin offers a simple way to create classes used for storing data, automatically generating boilerplate methods like `toString()`, `equals()`, and `hashCode()`.

Higher-Order Functions: Supports functional programming paradigms like lambda expressions and higher-order functions for cleaner code.

Default Arguments and Named Parameters: Reduces the need for method overloading by allowing functions to have default arguments.

Seamless Android Integration: Kotlin integrates well with Android Studio and provides Android-specific extensions, such as Kotlin Android Extensions for view binding and easier UI handling.

4.2 BACKEND

Spring Boot Framework

Spring Boot is a Java-based framework that simplifies the development of stand-alone, production-grade applications. It is a part of the larger Spring Framework, but Spring Boot makes it easier to set up and deploy Java-based applications by offering features like embedded servers, auto-configuration, and minimal configuration requirements. Spring Boot is widely used for building RESTful web services and microservices due to its rapid development capabilities, robust architecture, and ease of use. It is highly scalable and integrates seamlessly with various tools for security, database access, and testing.

Spring Boot Features:

Auto-Configuration: Automatically configures components based on the application's needs, reducing the need for manual configuration.

Embedded Servers: Includes embedded support for Tomcat, Jetty, and Undertow servers, eliminating the need for external application servers.

Microservices Ready: Ideal for building microservices architectures, with seamless integration with Spring Cloud.

Production-Ready: Provides features like health checks, metrics, and application monitoring, making it suitable for production environments.

Rapid Development: Simplifies the development process by reducing boilerplate code, making it faster to develop applications.

Built-in REST Support: Easily create RESTful web services with built-in support for HTTP methods and status codes.

Spring Data Integration: Works well with Spring Data, enabling simple and easy access to databases like MySQL, PostgreSQL, MongoDB, etc.

Security Integration: Provides easy integration with Spring Security for securing APIs and user authentication.

Testing Support: Has built-in tools to create unit tests, integration tests, and mock services to verify API functionality.

Minimal Configuration: Requires minimal setup and configuration, allowing developers to focus more on business logic rather than application setup.

RESTful API

RESTful APIs (Representational State Transfer) are a set of rules and conventions for building web services that allow communication between client and server over HTTP. RESTful API development tools help developers design, implement, test, and manage APIs. Some popular tools for REST API development include Postman, Swagger, and Insomnia. These tools streamline tasks such as defining API endpoints, testing API functionality, validating responses, and generating documentation. They support essential HTTP methods (GET, POST, PUT, DELETE) and offer features to test and mock APIs for better development and debugging.

RESTful API Development Tools Features:

API Testing: Allows developers to easily send requests to APIs and validate the responses, ensuring the API behaves as expected.

Mocking APIs: Supports mocking API responses to simulate the behavior of an API before it is fully implemented, improving development speed.

Interactive Documentation: Tools like Swagger and Postman allow you to generate interactive, user-friendly documentation for APIs, making it easier for users to understand and use them.

Request History: Provides a history of API requests for debugging and identifying issues.

Environment Variables: Supports different environments (e.g., development, testing, production) and allows dynamic use of environment-specific variables for API testing.

Authentication Support: Tools like Postman allow testing APIs with authentication methods such as OAuth, API keys, and Basic Auth, ensuring secure API interactions.

Automated Testing: Tools like Postman support automated testing of APIs using scripts, making it easier to validate functionality during development and integration.

API Documentation Generation: Automatically generates API documentation based on defined endpoints, making it easier to maintain and share with others.

Version Control Integration: Integrates with version control systems (e.g., Git) to keep track of changes in API development and updates.

Collaboration Features: Offers team collaboration, allowing multiple developers to work on the same API project and share collections, requests, and environments.

Database: MySQL/PostgreSQL

MySQL and PostgreSQL are two of the most widely used relational database management systems (RDBMS). MySQL is known for its speed, reliability, and ease of use, making it ideal for web applications, while PostgreSQL is recognized for its advanced features, scalability, and support for complex queries. Both databases are open-source, highly customizable, and used in a variety of applications, ranging from small projects to large-scale enterprise systems. They store data in tables with rows and columns and support SQL (Structured Query Language) for querying and manipulating data.

MySQL Features:

ACID Compliance: Ensures data consistency and integrity with transactions that are Atomic, Consistent, Isolated, and Durable.

High Performance: Optimized for high read and write operations, making it suitable for web applications with a large number of users.

Replication: Supports master-slave replication for data redundancy and failover support.

Scalability: Handles large-scale applications, with support for horizontal scaling and sharding.

Cross-Platform Support: Runs on various operating systems like Windows, Linux, and macOS, making it highly portable.

Data Security: Offers SSL encryption, user authentication, and role-based access control.

Full-Text Search: Allows fast searching of textual data stored within the database, improving query performance for text-heavy applications.

PostgreSQL Features:

ACID Compliance: Guarantees reliable transactions and data consistency.

Extensibility: Supports custom data types, operators, and functions, making it highly flexible for specialized use cases.

Advanced Querying: Includes support for complex queries, including joins, subqueries, window functions, and common table expressions (CTEs).

JSON Support: PostgreSQL provides robust support for storing and querying JSON data, enabling NoSQL-like capabilities.

Concurrency Control: Uses Multi-Version Concurrency Control (MVCC) for handling high transaction volumes without locking issues.

High Availability: Built-in support for replication, clustering, and failover mechanisms to ensure system uptime.

Security Features: Comprehensive access control features such as SSL, authentication, and row-level security.

Postman (API Testing Tool)

Postman is a powerful tool used for testing APIs by sending requests to an endpoint and analyzing the responses. It simplifies the process of developing, testing, and debugging RESTful APIs. Developers can make different types of HTTP requests (e.g., GET, POST, PUT, DELETE), inspect responses, and validate their structure. Postman also allows for the automation of tests, the creation of reusable collections, and the sharing of API-related workflows with teams.

Postman Features:

Request Creation: Allows the creation of various HTTP requests such as GET, POST, PUT, DELETE, and more for testing APIs.

Response Analysis: Inspect API responses with detailed information such as headers, body content, and status codes.

Environment Variables: Supports different environments (e.g., development, staging, production) and dynamic variables for versatile API testing.

Automated Testing: Enables automated tests with scripting, which can verify API behavior and ensure endpoints are working as expected.

Mock Servers: Create mock API responses to simulate server behavior before implementation, aiding in the development process.

API Documentation: Generates interactive and user-friendly API documentation automatically from request collections.

Collaboration: Allows teams to share and collaborate on collections, environments, and tests, improving teamwork.

Git (Version Control)

Git is a distributed version control system that helps track changes in source code during software development. It allows multiple developers to work on a project simultaneously without interfering with each other's work. Git stores a complete history of changes, making it easy to revert to previous versions and track bugs. Popular platforms like GitHub, GitLab, and Bitbucket offer hosting for Git repositories, enabling collaboration on large projects.

Git Features:

Distributed Version Control: Every developer has a local copy of the full project history, which allows working offline and fast access to version control.

Branching and Merging: Developers can create branches to work on features independently, and Git provides robust tools for merging branches.

History Tracking: Git tracks every change made to the codebase, including commit messages, timestamps, and changes.

Collaboration: Facilitates teamwork with features like pull requests, code reviews, and conflict resolution.

Commit System: Allows granular control over changes, with commits representing small, logical changes that can be easily reverted.

Security: Git uses SHA-1 hashes to ensure the integrity of the repository and protect against data corruption.

Integration with CI/CD Tools: Git integrates with continuous integration (CI) and continuous deployment (CD) tools, making it part of automated workflows.

Maven/Gradle (Dependency Management)

Maven and Gradle are build automation and dependency management tools used in Java development. Maven uses an XML-based configuration to define project dependencies, build processes, and plugins. Gradle, on the other hand, uses a Groovy-based domain-specific language (DSL) to define tasks and dependencies. Both tools simplify the management of project dependencies, automate build processes, and integrate well with continuous integration (CI) systems.

Maven Features:

Dependency Management: Automatically downloads and manages required libraries and frameworks for your project.

Centralized Repository: Maven uses a central repository to store and fetch dependencies, making it easy to maintain libraries.

Project Object Model (POM): Uses a POM file to manage project configuration, dependencies, and build settings.

Build Automation: Automatically compiles source code, runs tests, and packages projects into JAR/WAR files.

Plugin Ecosystem: Extends functionality with various plugins, such as for testing, documentation, and deployment.

Multi-Module Projects: Supports complex, multi-module projects by defining parent-child relationships in the POM file.

Integration with CI/CD: Easily integrates with CI/CD tools for automated testing and deployment pipelines.

Gradle Features:

Declarative DSL: Uses a flexible Groovy-based DSL to define project dependencies and tasks, which can be customized as needed.

Performance Optimization: Gradle optimizes the build process using features like incremental builds and parallel execution.

Multi-Project Builds: Handles large-scale projects with multiple modules efficiently, allowing for shared configurations and dependencies.

Dependency Management: Integrates with Maven and Ivy repositories to manage project dependencies, similar to Maven.

Build Automation: Automates the entire project build process, including compiling, testing, and packaging code.

Customizable Builds: Supports custom build tasks, allowing developers to extend the build process with their own logic.

Integration with CI/CD Tools: Gradle works seamlessly with CI/CD systems for automated testing, building, and deploying applications.

CHAPTER 5

PROJECT DESCRIPTION

5.1 OVERVIEW OF THE PROJECT

The Invoice Generation Application is designed to streamline and automate the process of creating, managing, and sharing invoices for businesses of all scales. This robust application is built using modern technologies to ensure efficiency, scalability, and user-friendly operation. The project comprises an intuitive frontend, a powerful backend, a reliable database system, and supporting tools to ensure seamless functionality.

The frontend of the application is developed using Android Studio, leveraging Java or Kotlin programming languages to create an engaging user interface optimized for Android devices. It provides a smooth user experience with features like invoice creation, viewing, editing, and exporting.

The backend is powered by the Spring Boot framework, which ensures a lightweight, high-performing, and secure architecture. RESTful APIs are implemented to handle communication between the frontend and backend, enabling features such as data synchronization, user authentication, and invoice storage.

Data storage and management are handled by relational database systems such as MySQL or PostgreSQL. These databases are chosen for their reliability, scalability, and support for complex queries, ensuring efficient data handling and retrieval.

Development tools such as Postman are used to test APIs, while Git ensures version control and collaboration among team members. Maven or Gradle is utilized for dependency management, simplifying the build process and maintaining consistent configurations.

This project aims to enhance productivity for businesses by automating invoice-related tasks, reducing manual errors, and ensuring timely delivery of invoices. Its modular architecture ensures adaptability for future enhancements and scalability to meet growing business needs.

5.2 MODULES

The Invoice Generation Application is divided into several functional modules to ensure streamlined operations, better maintainability, and efficient performance. Below are the key modules with their respective functionalities

- User Registration
- Invoice Management
- Dashboard
- Invoice Details View
- Invoice Editing
- Invoice Deletion

5.2.1 MODULE DESCRIPTION

Invoice Management

The Invoice Management module is the core of the application, enabling users to create, view, edit, and delete invoices. It allows users to organize and track all invoices with features like sorting, filtering, and status updates (e.g., Paid, Pending, Overdue). It ensures seamless invoice handling with real-time calculations and error checking.

Dashboard

The dashboard provides an overview of the application's key metrics and activities. Users can view statistics such as total invoices generated, payment statuses, outstanding balances, and revenue trends. The dashboard is designed to give actionable insights at a glance, making it easier to manage business finances efficiently.

Invoice Details View

This module displays the detailed information of a specific invoice. Users can view itemized breakdowns, taxes, discounts, payment terms, and client details. It ensures transparency and allows users to verify or share invoice details easily with clients or stakeholders.

Invoice Editing

Invoice Editing enables users to update existing invoices with new or corrected information. Users can modify client details, item descriptions, quantities, prices, or taxes. This module ensures that businesses can rectify errors or make updates without needing to generate new invoices.

Invoice Deletion

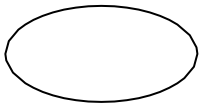
This module provides the functionality to delete unwanted or erroneous invoices. It includes confirmation prompts to avoid accidental deletions and ensures that deleted invoices are properly logged for audit purposes, maintaining data integrity and compliance.

5.3 DATA FLOW DIAGRAM

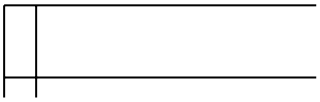
A data flow diagram (DFD) is a graphical representation of the "flow" of data through an information system, modeling its process aspects. A DFD is often used as a preliminary step to create an overview of the system without going into great detail, which can later be elaborated. DFDs can also be used for the visualization of data processing (structured design). A DFD shows what kind of information will be input to and output from the system, how the data will advance through the system, and where the data will be stored. It does not show information about the timing of the process or information about whether processes will operate in sequence or parallel, unlike a flowchart which also shows this information. They based it on the “data flow graph” computation models by David Martin and Gerald Estrin.

Basic DFD Notations

- Destination



- Process

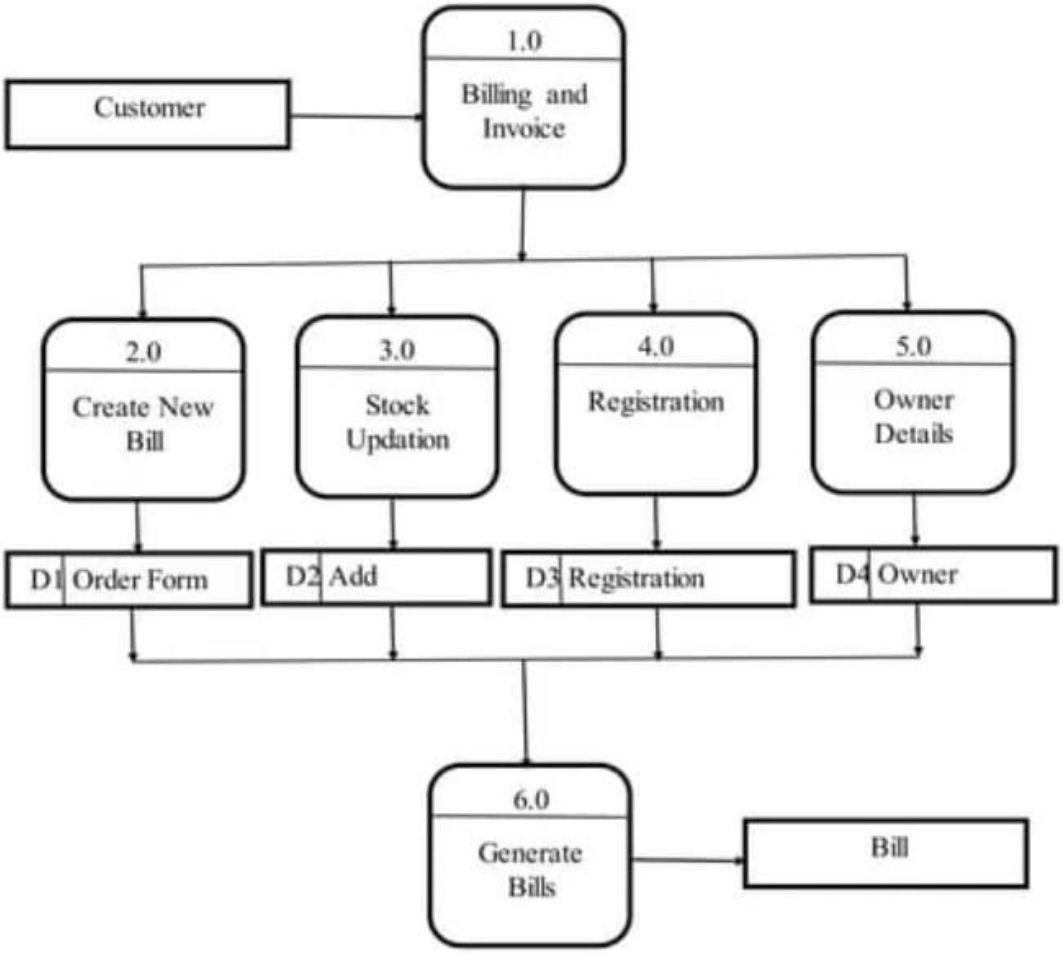


- Data Storage

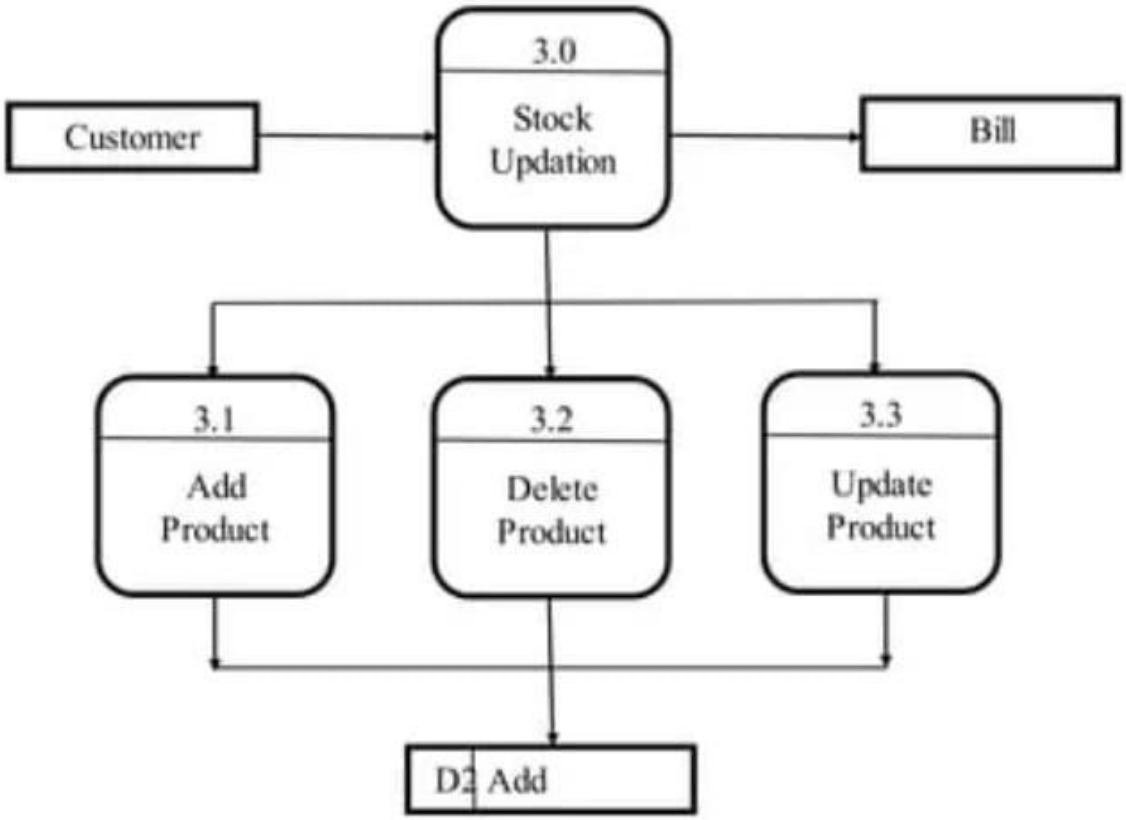
- Data Flow

Level 0:

Level 1:


1.0
Billing and Invoice
Customer
2.0
Create New Bill
3.0
Stock Updation
4.0
Registration
5.0
Owner Details
D1 Order Form
D2 Add
D3 Registration
D4 Owner
6.0
Generate Bills
Bill

Level 2:


3.0
Stock Updation
Customer
Bill
3.1
Add Product
3.2
Delete Product
3.3
Update Product
D2 Add

ENTITY RELATIONSHIP DIAGRAM

The relation upon the system is structured through a conceptual ER-Diagram, which not only specifies the existential entities but also the standard relations through which the system exists and the cardinalities that are necessary for the system state to continue. The Entity Relationship Diagram (ERD) depicts the relationship between the data objects. The ERD is the notation that is used to conduct the data modeling activity The attributes of each data object noted in the ERD can be described resign a data object description.

The set of primary components that are identified by the ERD are

i. Data object

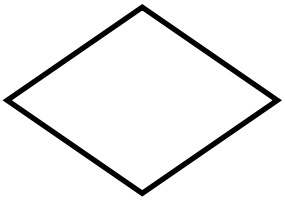
ii. Relationships

iii. Attributes

iv. Various types of indicators

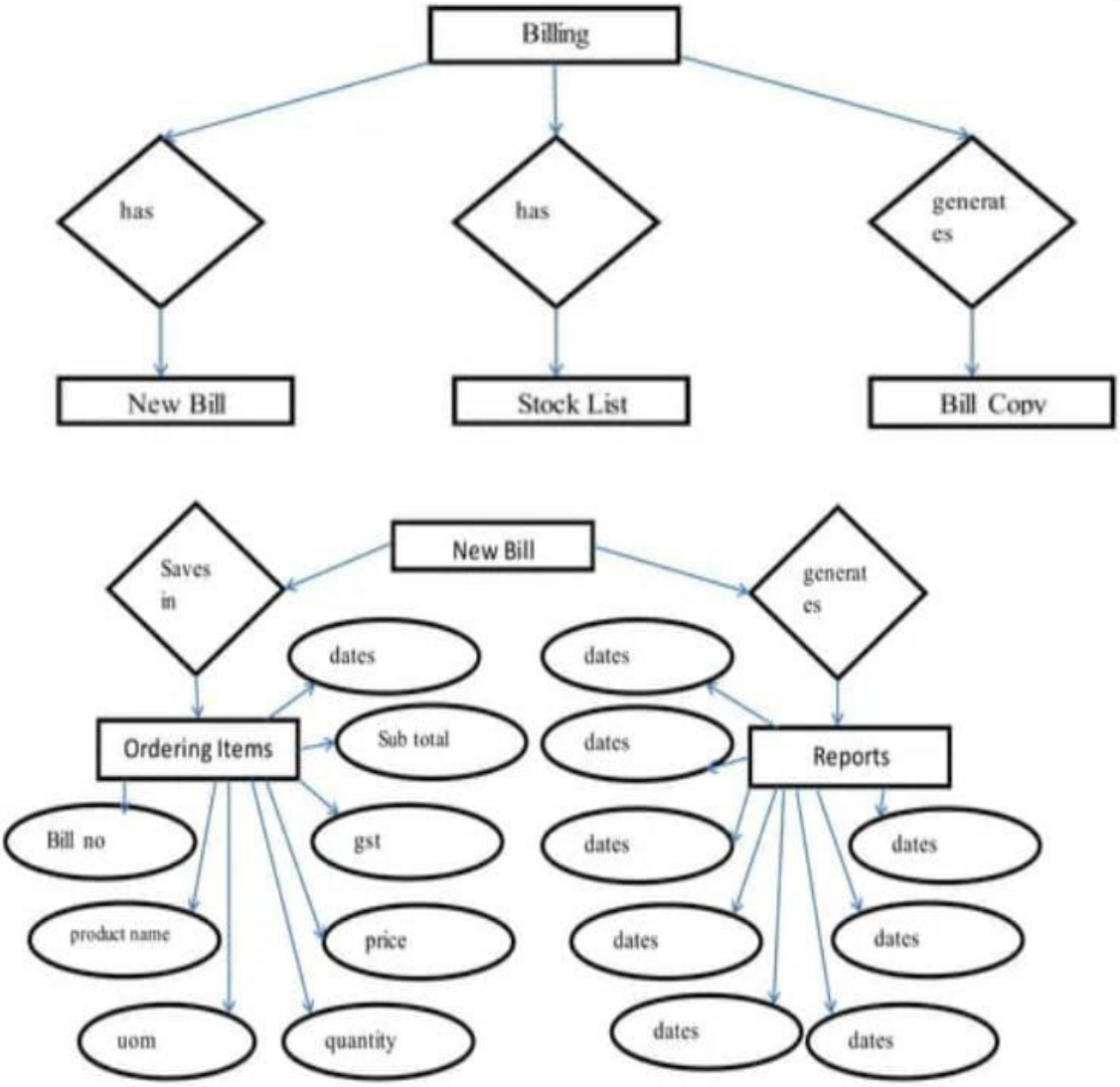
ER-DIAGRAM SYMBOL

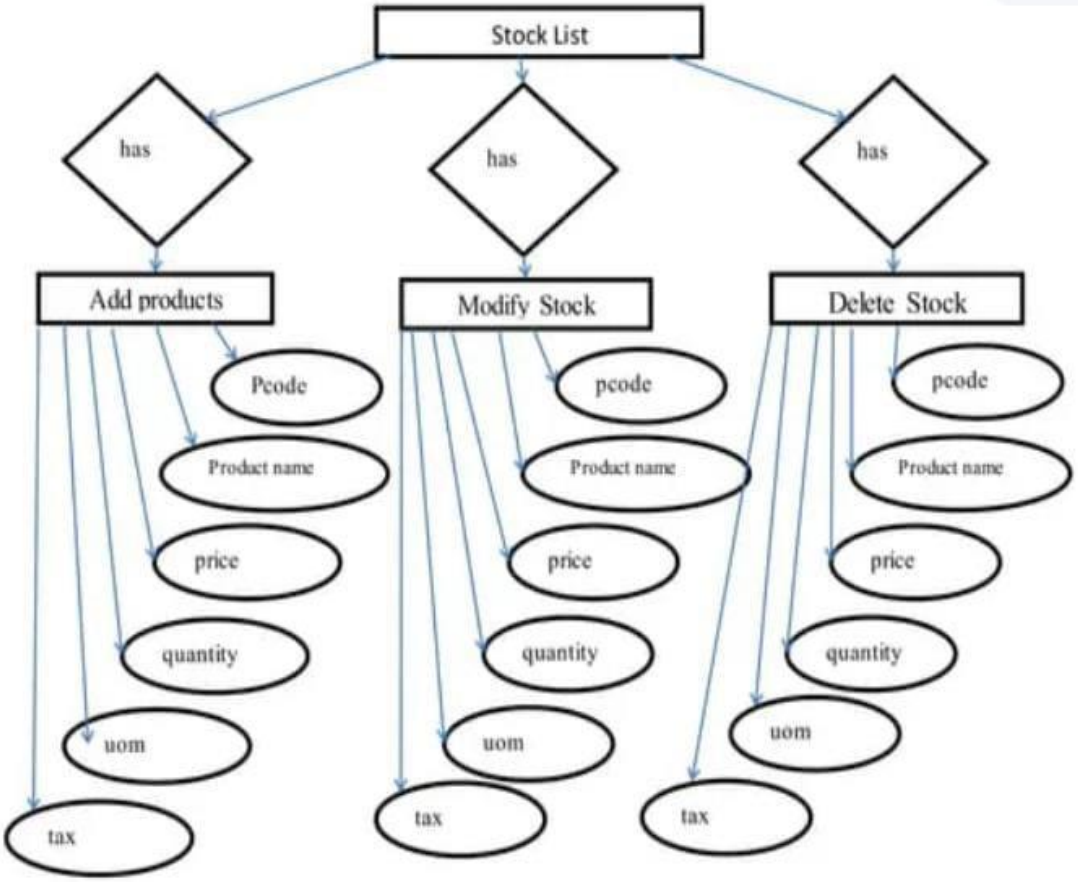
Entity 

Relationship 

Flow

ER DIAGRAM:


Billing
has
has
generat
es
New Bill
Stock List
Bill Copy
Saves
in
New Bill
generat
es
dates
dates
Ordering Items
Sub total
dates
Reports
Bill no
gst
dates
dates
product name
price
dates
dates
uom
quantity
dates
dates


Stock List
has
has
has
Add products
Modify Stock
Delete Stock
Pcode
Product name
price
quantity
uom
tax
pcode
Product name
price
quantity
uom
tax
pcode
Product name
price
quantity
uom
tax

5.4 DATABASE DESIGN

The design phase is the period in the software life cycle during which the designs for architecture, software components, interfaces, and data are created, documented, and verified to satisfy requirements.

Details on computer programming language and environments, application packages, layering, memory size, platform, algorithms, data structure, interfaces, and many others. The design may include the usage of existing components. The components are as follows;

- System Design
- Input design
- Output design
- Code design

SYSTEM DESIGN

The degree of interest in each concept has varied over the year, each has stood the test of time. Each provides the software designer with a foundation from which more sophisticated design methods can be applied. Fundamental design concepts provide the necessary framework for “getting it right”.

During the design process the software requirements model is transformed into design models that describe the details of the data structures, system architecture, interface, and components. Each design product is reviewed for quality before moving to the next phase of software development.

INPUT DESIGN

The design of input focus on controlling the amount of dataset as input required, avoiding delay and keeping the process simple. The input is designed in such a way to provide security. Input design will consider the following steps:

- The dataset should be given as input.
- The dataset should be arranged.
- Methods for preparing input validations.

OUTPUT DESIGN

A quality output is one, which meets the requirement of the user and presents the information clearly. In output design, it is determined how the information is to be displayed for immediate need.

Designing computer output should proceed in an organized, well thought out manner the right output must be developed while ensuring that each output element is designed so that the user will find the system can be used easily and effectively.

This phase contains the attributes of the dataset which are maintained in the database table.

The dataset collection can be of two types namely train dataset and test dataset.

5.5 TABLE DESIGN

1.Registeration_form

Fieldname	Data type
shop_name	Varchar(50)
Gstin	Varchar(20)
tin_no	Varchar(15)
Address	Varchar(150)
Phone	Varchar(20)
State	Varchar(25)
City	Varchar(15)
Pincode	Int(11)
Email	Varchar(20)

2.Owners_details

Fieldname	Data type
Name	Varchar(20)
Password	Varchar(6)
Securityquestion1	Varchar(100)
Ans 1	Varchar(20)
Securityquestion2	Varchar(100)
Ans 2	Varchar(20)

3.Orderform

Fieldname	Data type
productname	Varchar(100)
Uom	Varchar(15)
Quantity	Int(20)
Price	float
Gst	float
Subtotal	float
Dates	float
Billno	Int(11)

4.Addproduct

Fieldname	Data type
Productname	Varchar(100)
Uom	Varchar(15)
Quantity	Int(10)
Price	Float
Pcode	Int(3)
Tax	Int(11)
Dates	date

CHAPTER 6

SYSTEM TESTING

System testing was done by giving different training and testing datasets. This test was done to evaluate whether the system was predicting accurate result or not. During the phase of the development of the system our system was tested time and again.
The series of testing conducted are as follows:

I. Unit testing

II. Integration testing

III. Alpha testing

IV. Beta testing

6.1 UNIT TESTING

In unit testing, we designed the whole system in modularized pattern and each module was tested. Till we get the accurate output from the individual module we worked on the same module.

6.2 INTEGRATION TESTING

After constructing individual modules all the modules were merged and a complete system was made. Then the system was tested whether the prediction given by training dataset to testing set was correct or not. We tried to meet the accuracy as higher as much as we can get. After spending a couple of days in integration testing the average accuracy of our system was 91%.

6.3 ALPHA TESTING

Alpha testing is the first stage of software engineering which is considered as a simulated or actual operational testing done by the individual member of the project. Alpha testing is conducted by the project developers, in context of our project.

6.4 BETA TESTING

Beta testing comes continuously after alpha testing which is considered as a form of external user acceptance testing. The beta version of the program is developed to and provided to limited audience. This is the final test process in the case of this project. In this system the beta-testing is done by our colleagues and the project supervisor.

CHAPTER 7

SYSTEM IMPLEMENTATION

Implementation is the stage in the project where the theoretical design is turned into a working system. The most crucial stage is achieving a successful new system & and giving the user confidence that the new system will work efficiently & and effectively in the implementation stage.

The stage consists of

- Testing the developed program with simple data.
- Detections and correction of errors.
- Creating whether the system meets user requirements.
- Testing whether the system.
- Making necessary changes as desired by the user.
- Training user personnel.

Implementation Procedures

The implementation phase is less creative than the system design. A system project may be dropped at any time before implementation, although it becomes more difficult when it goes to the design phase.

The final report to the implementation phase includes procedural flowcharts, record layouts, report layouts, and a workable plan for implementing the candidate system design into an operational one. Conversion is one aspect of implementation

System Maintenance

Maintenance is the implementation of the review plan. As important as it is, many programmers and analysts are to perform or identify themselves with the maintenance effort. There are psychological, personality, and professional reasons for this. Analysts and programmers spend far more time maintaining programs than they do writing them. Maintenance accounts for 50-80 percent of total system development.

Maintenance is expensive. One way to reduce maintenance costs is through maintenance management and software modification audits.

- Maintenance is not as rewarding or exciting as developing systems. It is perceived as requiring neither skill nor experience.
- Users are not fully cognizant of the maintenance problem or its high cost.
- Few tools and techniques are available for maintenance.
- A good test plan is lacking.
- Standards, procedures, and guidelines are poorly defined and enforced.
- Programs are often maintained without care for structure and documentation.
- There are minimal standards for maintenance.
- Programmers expect that they will not be in their current commitment by the time their programs go into the maintenance cycle.

CHAPTER 8

CONCLUSION

In conclusion, The Invoice Generation Application is a comprehensive solution designed to simplify and automate the invoicing process for businesses of all sizes. By integrating modern technologies such as Android Studio, Spring Boot, and robust database systems like MySQL or PostgreSQL, the application ensures a seamless user experience and reliable performance. Its intuitive interface, coupled with powerful backend support, enables efficient invoice creation, management, and sharing.

This application reduces manual errors, enhances productivity, and ensures timely invoicing, fostering better client relationships. Features such as payment tracking, notifications, and detailed reporting further add value by providing businesses with essential tools to monitor their financial activities. With its modular design and scalability, the application is adaptable to evolving business needs, making it a future-proof solution for invoice management. Ultimately, the Invoice Generation Application empowers businesses to streamline operations and focus on growth.

FUTURE ENHANCEMENTS

The Invoice Generation Application can be enhanced with various advanced features to increase functionality, improve user experience, and cater to diverse business needs.

One key enhancement could be cloud integration, allowing users to securely store and access invoices and client data from any device, ensuring better data management and collaboration. AI-driven insights could provide predictive analytics to help businesses understand financial trends and make informed decisions.

Multi-currency and multilingual support would enable global businesses to manage invoices for international clients effortlessly. Integration with accounting software like QuickBooks or Tally could streamline financial data synchronization, reducing manual effort.

Payment gateway integration would facilitate secure online payments directly from the application. Adding support for subscription-based billing could help businesses with recurring payment models.

Mobile app enhancements could include voice commands for creating invoices and offline functionality to ensure uninterrupted access. Custom dashboards could provide a personalized overview of financial metrics for quick insights.

The addition of advanced security features, like biometric login and data encryption, would enhance data safety. Finally, AI-powered error detection could identify anomalies in invoices, ensuring accuracy. These future enhancements would make the application even more robust, user-friendly, and versatile for businesses.

CHAPTER 9

APPENDIX

9.1 SOURCE CODE

InvoiceGenerationApplication.Java

```
package com.example.InvoiceGeneration;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class InvoiceGenerationApplication {

    public static void main(String[] args) {

        SpringApplication.run(InvoiceGenerationApplication.class, args);

    }

}
```

User entity

```
 private String username;

  private String password;

    private String email;
    private LocalDateTime creationDate;
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
```

```
    }
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public LocalDateTime getCreationDate() {
        return creationDate;
    }
    public void setCreationDate(LocalDateTime creationDate) {
        this.creationDate = creationDate;
    }
}
```

Invoice entity class

```
package com.example.InvoiceGeneration.entity;
import 35ombok35.persistence.Entity;
import 35ombok35.persistence.GeneratedValue;
import 35ombok35.persistence.GenerationType;
import 35ombok35.persistence.Id;
import 35ombok.Builder;
import 35ombok.Data;
```

```
@Entity
@Data
@Builder
public class Invoice {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private Long userId;
    private String clientName;
    private double amount;
    private String date;
    private String description;
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public Long getUserId() {
        return userId;
    }
    public void setUserId(Long userId) {
        this.userId = userId;
    }
    public String getClientName() {
        return clientName;
    }
    public void setClientName(String clientName) {
        this.clientName = clientName;
    }
    public double getAmount() return amount;
    }
    public void setAmount(double amount) {
        this.amount = amount;
```

```
    }
    public String getDate() {
        return date;
    }

    public void setDate(String date) {
        this.date = date;
    }
    public String getDescription() {
        return description;
    }
    public void setDescription(String description) {
        this.description = description;
    }
    public static InvoiceBuilder builder() {
        return new InvoiceBuilder();
    }
}
public class Invoice {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private Long userId;
    private String clientName;
    private double amount;
    private String date;
    private String description;
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public Long getUserId() {
```

```
        return userId;
    }
    public void setUserId(Long userId) {
        this.userId = userId;
    }
    public String getClientName() {
        return clientName;
    }
    public void setClientName(String clientName) {
        this.clientName = clientName;
    }
    public double getAmount() {
        return amount;
    }
    public void setAmount(double amount) {
        this.amount = amount;
    }
    public String getDate() {
        return date;
    }
    public void setDate(String date) {
        this.date = date;
    }
    public String getDescription() {
        return description;
    }
    public void setDescription(String description) {
        this.description = description;
    }
    public static InvoiceBuilder builder() {
        return new InvoiceBuilder();
    }
}
  private Long userId;
```

```
private String clientName;
private double amount;
private String date;
private String description;
public Long getId() {
   return id;
}
public void setId(Long id) {
   this.id = id;
}
public Long getUserId() {
   return userId;
}
public void setUserId(Long userId) {
   this.userId = userId;
}
public String getClientName() {
   return clientName;
}
public void setClientName(String clientName) {
   this.clientName = clientName;
}
public double getAmount() {
   return amount;
}
public void setAmount(double amount) {
   this.amount = amount;
}
public String getDate() {
   return date;
}

public void setDate(String date) {
   this.date = date;
```

```
    }
    public String getDescription() {
        return description;
    }
    public void setDescription(String description) {
        this.description = description;
    }
    public static InvoiceBuilder builder() {
        return new InvoiceBuilder();
    }

}
```

Invoice Builder entity class:

```
package com.example.InvoiceGeneration.entity;
import 40ombok.Builder;
@Builder
public class InvoiceBuilder {
    private Long userId;
    private String clientName;
    private double amount;
    private String date;
    private String description;
    public Invoice build() {
        Invoice invoice = new Invoice();
        invoice.setUserId(userId);
        invoice.setClientName(clientName);
        invoice.setAmount(amount);
        invoice.setDate(date);
        invoice.setDescription(description);
        return invoice;
    }
    public Long getUserId() {
        return userId;
    }
    public InvoiceBuilder userId(Long userId) {
```

```
      this.userId = userId;
        return this;
    }
    public String getClientName() {
        return clientName;
    }
    public InvoiceBuilder clientName(String clientName) {
        this.clientName = clientName;
        return this;
    }
    public double getAmount() {
       return amount;
    }
    public InvoiceBuilder amount(double amount) {
        this.amount = amount;
        return this;
    }
    public String getDate() {
        return date;
    }
    public InvoiceBuilder date(String date) {
        this.date = date;
        return this;
    }
    public String getDescription() {
        return description;
    }
    public InvoiceBuilder description(String description) {
        this.description = description;
        return this;
    }
}
package com.example.InvoiceGeneration.entity;
```

```
import 42ombok.Data;

import java.time.LocalDateTime;
@Data
public class UserBuilder {
    private Long id;
    private String username;
    private String password;
    private String email;
    private LocalDateTime creationDate;
    public static UserBuilder builder() {
        return new UserBuilder();
    }
    public UserBuilder id(Long id) {
        this.id = id;
        return this;
    }
    public UserBuilder username(String username) {
        this.username = username;
        return this;
    }
    public UserBuilder password(String password) {
        this.password = password;
        return this;
    }
    public UserBuilder email(String email) {
        this.email = email;
        return this;
    }
    public UserBuilder creationDate(LocalDateTime creationDate) {
        this.creationDate = creationDate;
        return this;
    }
    public User build() {
```

```
        User user = new User();
        user.setId(this.id);
        user.setUsername(this.username);
        user.setPassword(this.password);
        user.setEmail(this.email);
        user.setCreationDate(this.creationDate); // Set creation date
        return user;
    }
package com.example.InvoiceGeneration.controller;

import com.example.InvoiceGeneration.entity.User;
import com.example.InvoiceGeneration.service.AuthenticationService;
import com.example.InvoiceGeneration.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import java.util.List;
import java.util.Optional;
@RestController
@RequestMapping("/api/users")
public class UserController {
    @Autowired
    private UserService userService;
    @Autowired
    private AuthenticationService authenticationService;
    // Register a new user
    @PostMapping("/register")
    public ResponseEntity<User> registerUser(@RequestBody User user) {
        // Call the UserService to register the user
        User newUser = userService.registerUser(user);
        // Return the registered user with HTTP status CREATED

        return new ResponseEntity<>(newUser, HttpStatus.CREATED);
```

```
    }
    // Login user
    @PostMapping("/login")
    public ResponseEntity<User> loginUser(@RequestBody User user) {
        // Authenticate user credentials using AuthenticationService
        Optional<User> authenticatedUser =
authenticationService.authenticateUser(user.getUsername(), user.getPassword());
        // If authentication is successful, return the user with HTTP status OK
        // Otherwise, return HTTP status UNAUTHORIZED
        return authenticatedUser.map(value -> new ResponseEntity<>(value, HttpStatus.OK))
                .orElseGet(() -> new ResponseEntity<>(HttpStatus.UNAUTHORIZED));
    }
    // Get user by ID
    @GetMapping("/{id}")
    public ResponseEntity<User> getUserById(@PathVariable Long id) {
        // Call UserService to get user by ID
        Optional<User> user = userService.getUserById(id);
        // If user is found, return the user with HTTP status OK
        // Otherwise, return HTTP status NOT_FOUND
        return user.map(value -> new ResponseEntity<>(value, HttpStatus.OK))
                .orElseGet(() -> new ResponseEntity<>(HttpStatus.NOT_FOUND));
    }
    // Get all users
    @GetMapping
    public ResponseEntity<List<User>> getAllUsers() {
        // Call UserService to get all users
        List<User> users = userService.getAllUsers();
        // Return the list of users with HTTP status OK
        return new ResponseEntity<>(users, HttpStatus.OK);
}
```

9.2 SCREENSHOTS

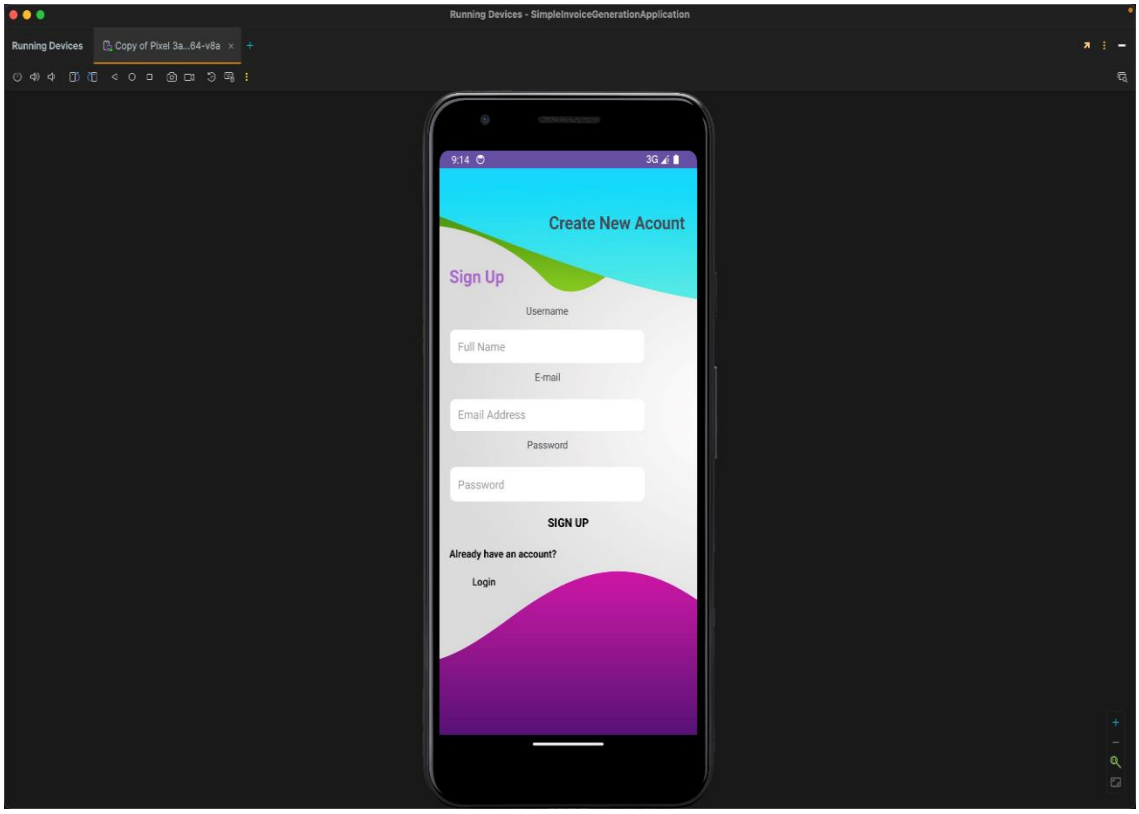


Figure 9.2.1. Signup Page

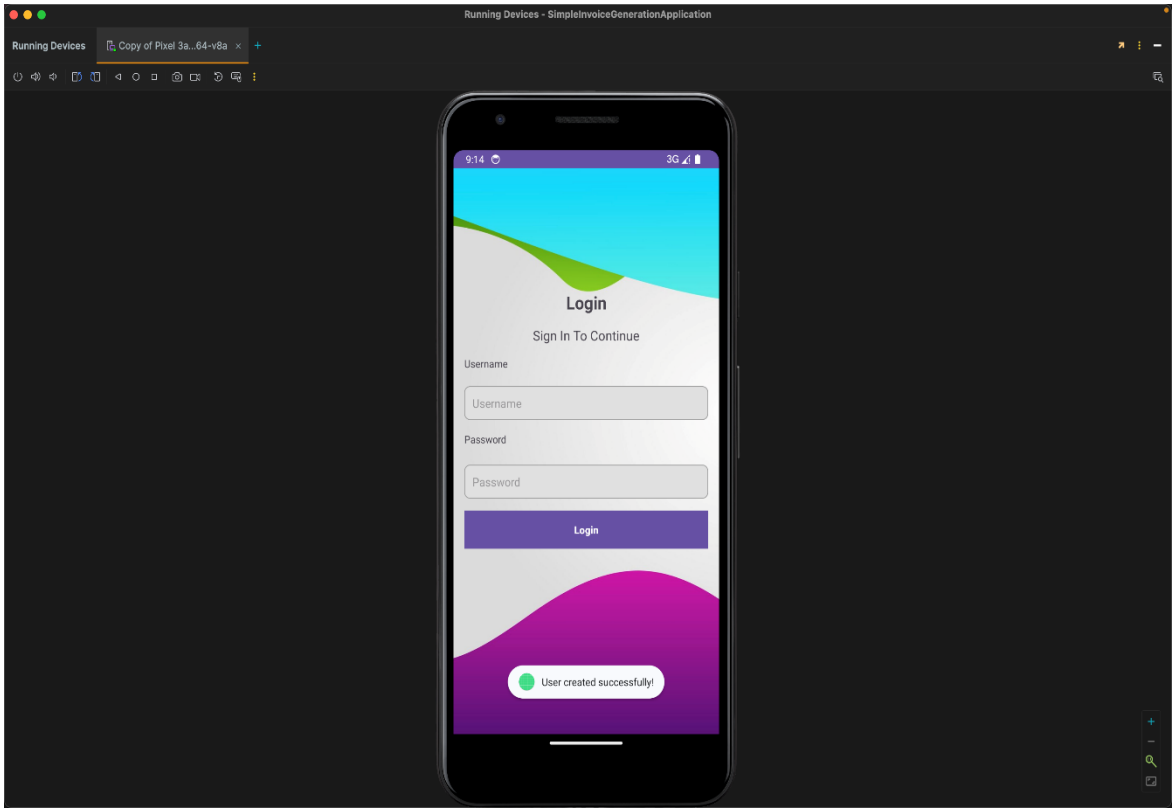


Figure 9.2.2 Login Page

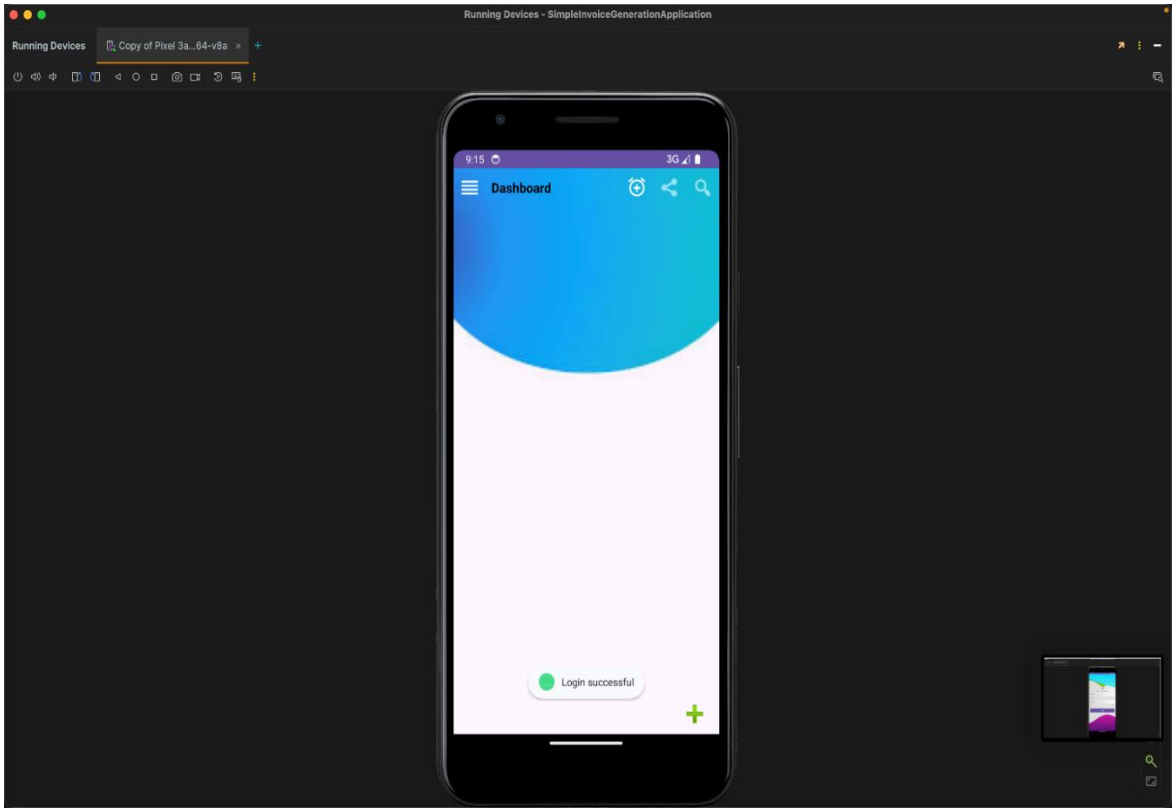


Figure 9.2.3. Dashboard Page

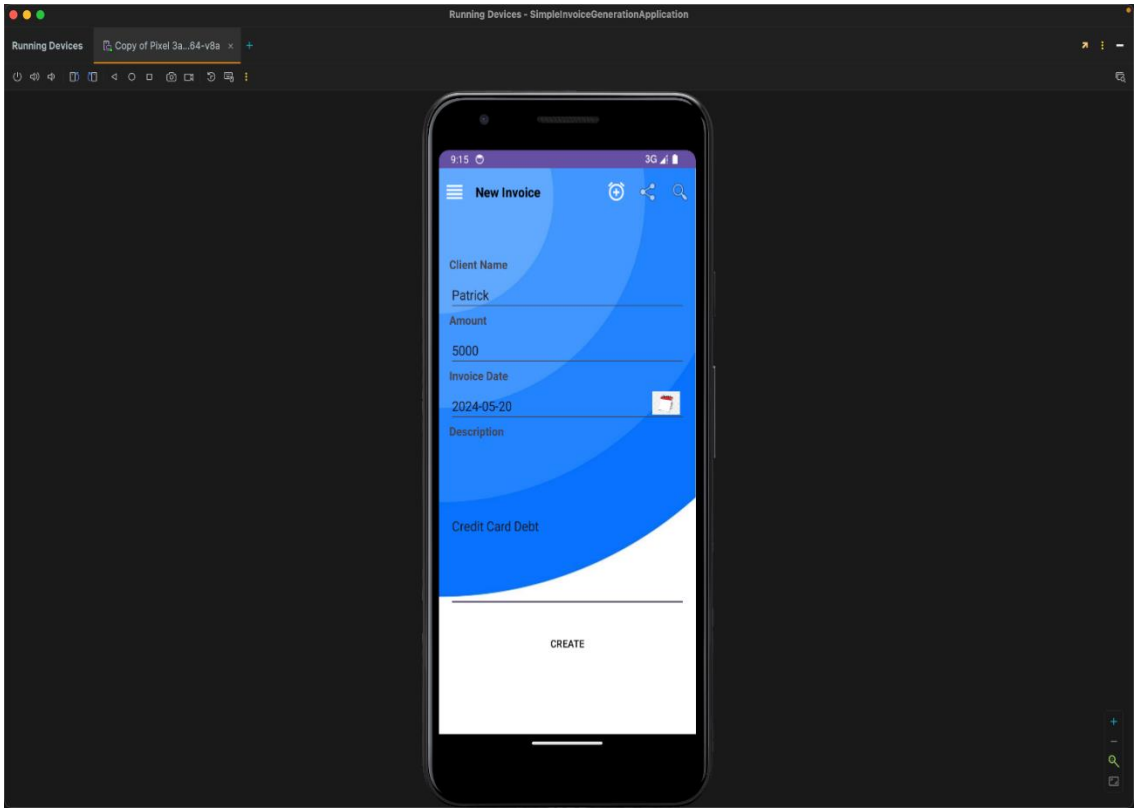


Figure 9.2.4 Create New Invoice Page

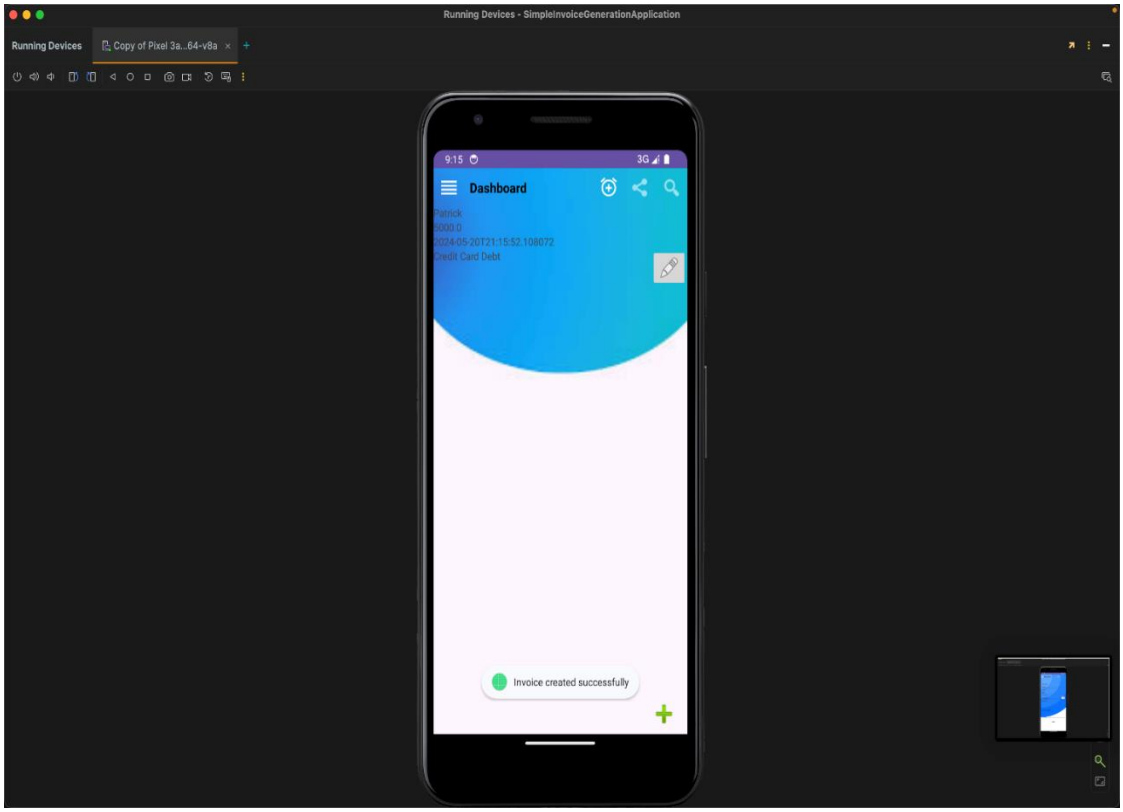


Figure 9.2.5. Invoice created Dashboard Page

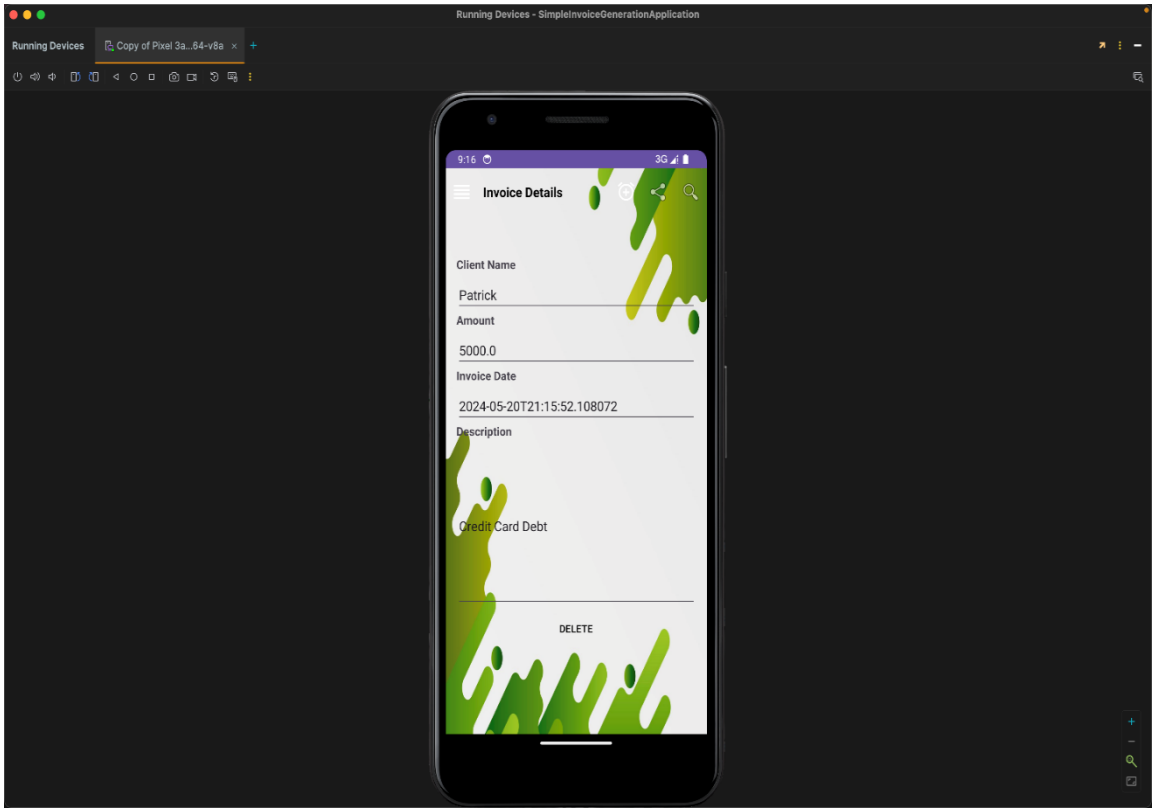


Figure 9.2.6. Invoice Details Page

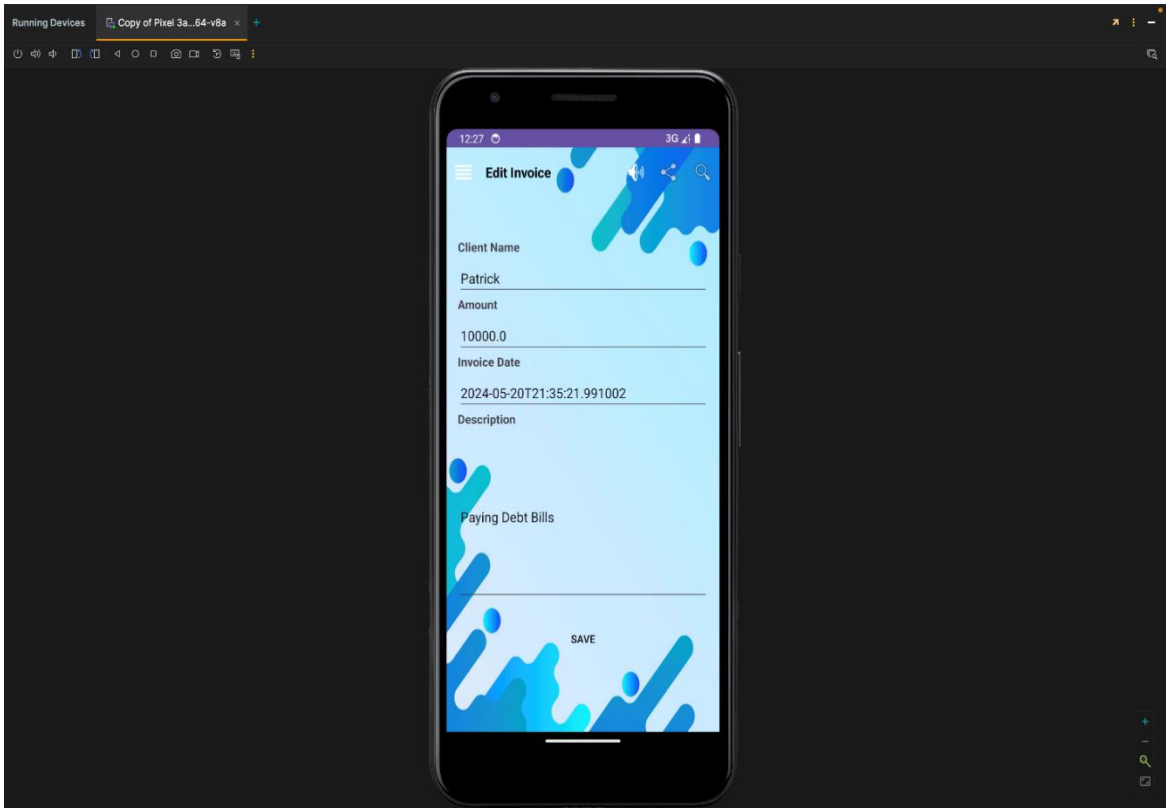


Figure 9.2.7. Edit Invoice Page

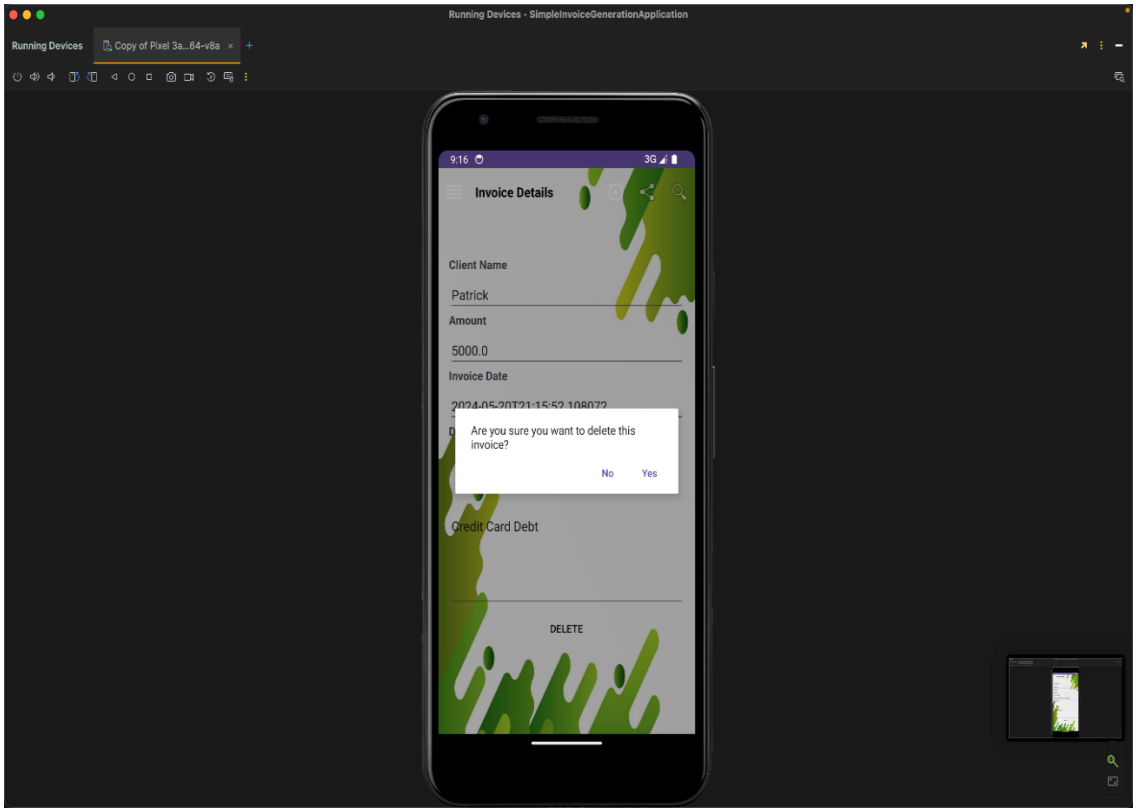


Figure 9.2.8. Invoice Deletion Page

CHAPTER 10

REFERENCES

BOOK REFERENCES

- Spring in Action by Craig Walls (2018)
- RESTful Web Services by Leonard Richardson and Sam Ruby (2007)
- MySQL Cookbook by Paul DuBois (2009)
- Pro Git by Scott Chacon and Ben Straub (2014)
- Maven: The Definitive Guide by Sonatype (2011)

REFERRED WEBSITES

- https://spring.io/projects/spring-boot
- https://www.postman.com/
- https://www.mysql.com/
- https://www.git-scm.com/
- https://maven.apache.org/