Software Engineering - Major Work

Software Engineering - Major Work....................1
- Identifying and Defining the problem.................... 2
- Client Needs and Requirements....................2
  - Functional Requirements....................2
  - Non-Functional Requirements.................... 2
- Constraints....................3
- Meeting Minutes.................... 3
- Approach.................... 3
- Project Management Plan.................... 5
- GUI Design.................... 9
- Data Dictionary.................... 14
- Process Log....................19
- Client Interactions....................20
- Software Solution....................21
- Testing and Maintenance....................22
- Security Features....................23
- Code Snippets.................... 24
- Pseudocode....................32
- Social and Ethical Issues....................36
- Client Evaluation....................37
- Self-Evaluation....................38

Client: Small Tuition Business Owner

Identifying and Defining the problem

My client runs a small-scale tutoring business that requires software to help them manage and create invoices for their clients. This software needs to be user-friendly and have access to a variety of features so that they can manage all the financial aspects of the business processes in one place.

The client currently manages all student information, session scheduling, and invoicing manually. This process is time-consuming, susceptible to human error in calculations, and makes tracking payments inefficient. Key challenges identified include:

- Difficulty in quickly generating accurate invoices for each student.
- No centralised system for viewing payment statuses, leading to potential cash flow issues.
- Significant administrative time spent on paperwork rather than on core business activities.

Client Needs and Requirements

Functional Requirements

- A secure login system using a username and password for the client
- Functionality to create, read, update, and delete student records
- Automated generation of invoices from the logged session data
- A central dashboard to display an overview of all invoices, clearly distinguishing between paid and outstanding amounts
- The ability to export or save invoices as documents for emailing or printing
- A secure, multi-user registration and login system using email and password.
- Mandatory email verification for new user accounts to ensure user authenticity.
- A secure password reset functionality for users who have forgotten their password.
- A system to record partial or full payments against invoices, which automatically updates the invoice status to "Paid," "Unpaid," or "Partially Paid."
- Functionality to generate and download a PDF summary of all unpaid invoices.
- The ability to email a specific invoice directly to a client as a PDF attachment.
- An integrated calendar to visually track invoice due dates and other custom events.

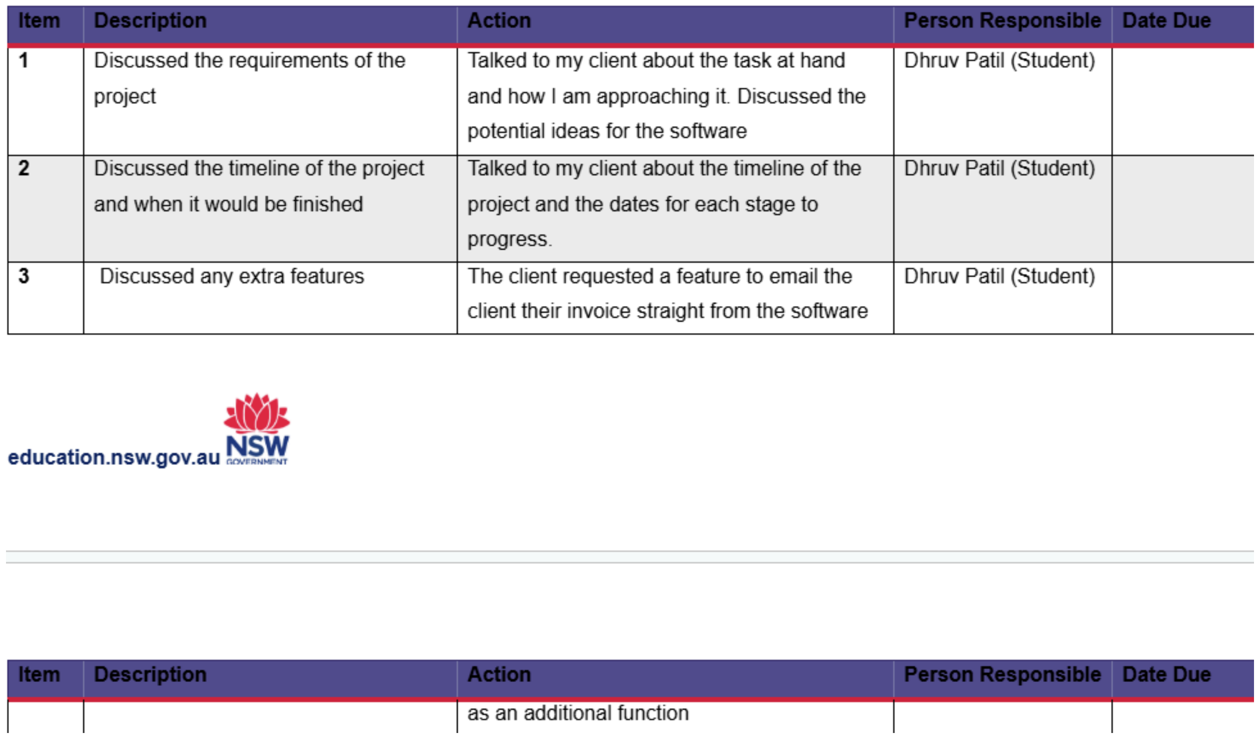
Non-Functional Requirements

- The software must have a clean, intuitive, and user-friendly interface
- All sensitive client and financial data must be stored securely
- The system must be reliable and perform all calculations accurately

Constraints

1. Time
    - Project must be completed by the given due date, Tuesday 24th of June 2025
2. Building
    - The project must be built using HTML, CSS, Python, and SQLite
3. Security
    - Must implement secure coding practices, including:
        - Appropriate password encryption
        - Authorisation on functions/pages for users
        - 2 factor authentication upon creation of account
        - Prevention of Cross-Site Scripting (XSS)

Meeting Minutes



Item	Description	Action	Person Responsible	Date Due
1	Discussed the requirements of the project	Talked to my client about the task at hand and how I am approaching it. Discussed the potential ideas for the software	Dhruv Patil (Student)	
2	Discussed the timeline of the project and when it would be finished	Talked to my client about the timeline of the project and the dates for each stage to progress.	Dhruv Patil (Student)	
3	Discussed any extra features	The client requested a feature to email the client their invoice straight from the software as an additional function	Dhruv Patil (Student)	

Approach

I utilised the SDLC Approach to focus on the different phases of creating a software solution, from planning, to design, to development and implementation, and finally to testing.

1. Planning
   This initial phase focused on defining the project's vision, scope, and core functionality.

- Objective: To create a secure, user-friendly web application for small business owners to manage clients, create invoices, track payments, and generate reports.
- Secure user registration and login system.
- Full CRUD (Create, Read, Update, Delete) functionality for clients.
- Full CRUD functionality for invoices linked to clients.
- Ability to record partial or full payments against invoices.
- PDF generation for individual invoices and summary reports.
- A calendar view for tracking due dates.
- An analytics dashboard for financial insights.
- AI-assisted drafting of reminder emails

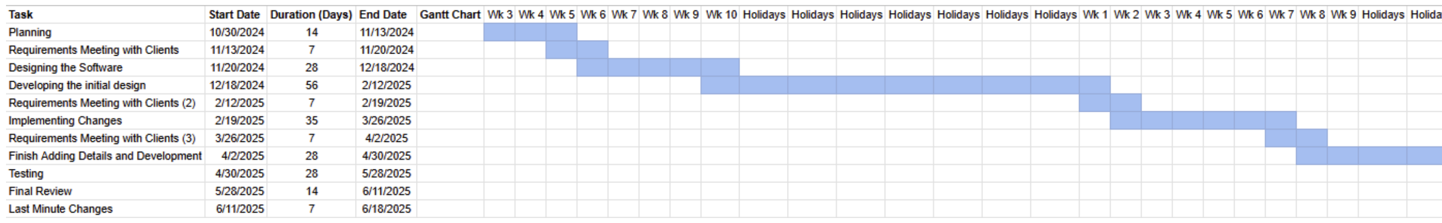
2. Design
   Once the requirements were clear, I designed the application's architecture and data structure.
   - Technology Stack: I chose Python with the Flask micro-framework for its simplicity and scalability, SQLite for its lightweight, file-based database ideal for this scale, and Firebase Authentication to handle the complexities of user management securely.
   - I created simple wireframes for the main pages: the dashboard, the invoice list, the client management screen, and the data entry forms.

3. Development and Implementation
   Development was broken down into several chunks, allowing me to build and test the application in manageable chunks.
   - Chunk 1
     - Set up the basic Flask application, database initialization (init_db), and user session management.
     - Integrated Firebase for user registration and login. It was during this stage that Bug #3 (Verification code wouldn't send) was encountered. The initial fix involved debugging the payload sent to the Brevo email API to ensure it was correctly formatted.
   - Chunk 2
     - Developed the PDF generation feature using the ReportLab library. This is where Bug #5 (PDF generation wouldn't differentiate) was found when testing the download buttons, leading to a fix in the frontend HTML href links.
     - Implemented the calendar view. The initial version did not show invoice due dates, which led to the discovery of Bug #4 (Calendar events not linked). The fix involved adding the specific logic to the /get_events endpoint to query the invoices table.
   - Chunk 3
     - Integrated the Google Generative AI for drafting reminder emails. This is when Bug #6 (AI for emails was locked) occurred due to configuration

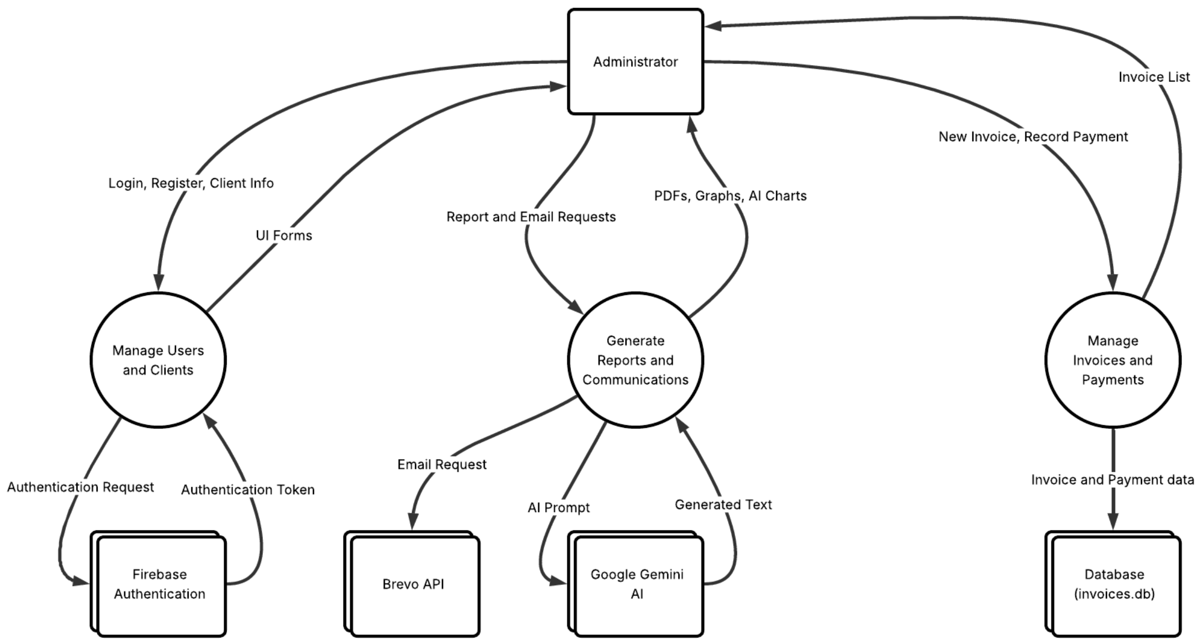
issues, which was fixed by properly setting up environment variables for the API key and enabling the service in the Google Cloud dashboard.

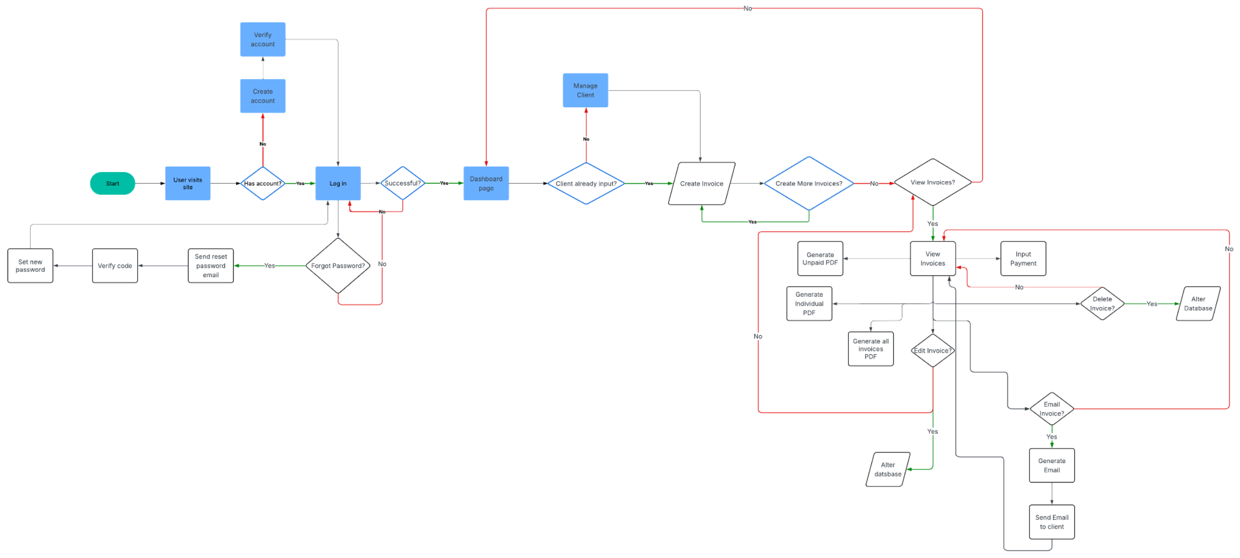
4. Testing
   Testing was not a single phase but a continuous activity integrated into each development sprint.
   - I performed informal unit tests on backend functions to ensure they produced the expected output.
   - I tested how different parts of the system worked together. For example, after creating a client, I would test if I could then create an invoice for them and subsequently record a payment.
   - Bug #1 (Lack of user differentiation) was found when two test users reported they could see and even delete each other's invoices. This was a critical finding that led to a security-focused fix across all database queries.
   - Bug #2 (Database Locking) was discovered during simulated UAT where I used scripts to mimic multiple users creating invoices at the same time, causing the application to crash.

Project Management Plan

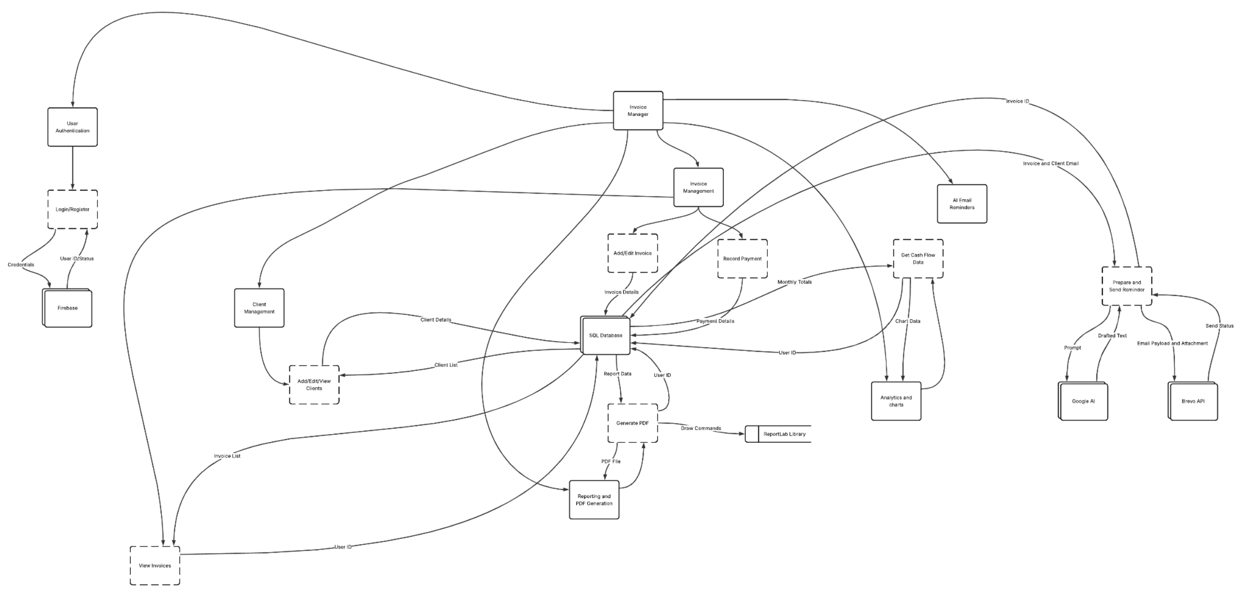


Task	Start Date	Duration (Days)	End Date	Gantt Chart	Wk 3	Wk 4	Wk 5	Wk 6	Wk 7	Wk 8	Wk 9	Wk 10	Holidays	Holidays	Holidays	Holidays	Holidays	Holidays	Holidays	Wk 1	Wk 2	Wk 3	Wk 4	Wk 5	Wk 6	Wk 7	Wk 8	Wk 9	Holidays	Holida
Planning	10/30/2024	14	11/13/2024		🟦	🟦	🟦																							
Requirements Meeting with Clients	11/13/2024	7	11/20/2024				🟦	🟦																						
Designing the Software	11/20/2024	28	12/18/2024					🟦	🟦	🟦	🟦	🟦																		
Developing the initial design	12/18/2024	56	2/12/2025									🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦										
Requirements Meeting with Clients (2)	2/12/2025	7	2/19/2025																	🟦	🟦									
Implementing Changes	2/19/2025	35	3/26/2025																		🟦	🟦	🟦	🟦	🟦	🟦				
Requirements Meeting with Clients (3)	3/26/2025	7	4/2/2025																							🟦	🟦			
Finish Adding Details and Development	4/2/2025	28	4/30/2025																								🟦	🟦	🟦	🟦
Testing	4/30/2025	28	5/28/2025																											
Final Review	5/28/2025	14	6/11/2025																											
Last Minute Changes	6/11/2025	7	6/18/2025																											

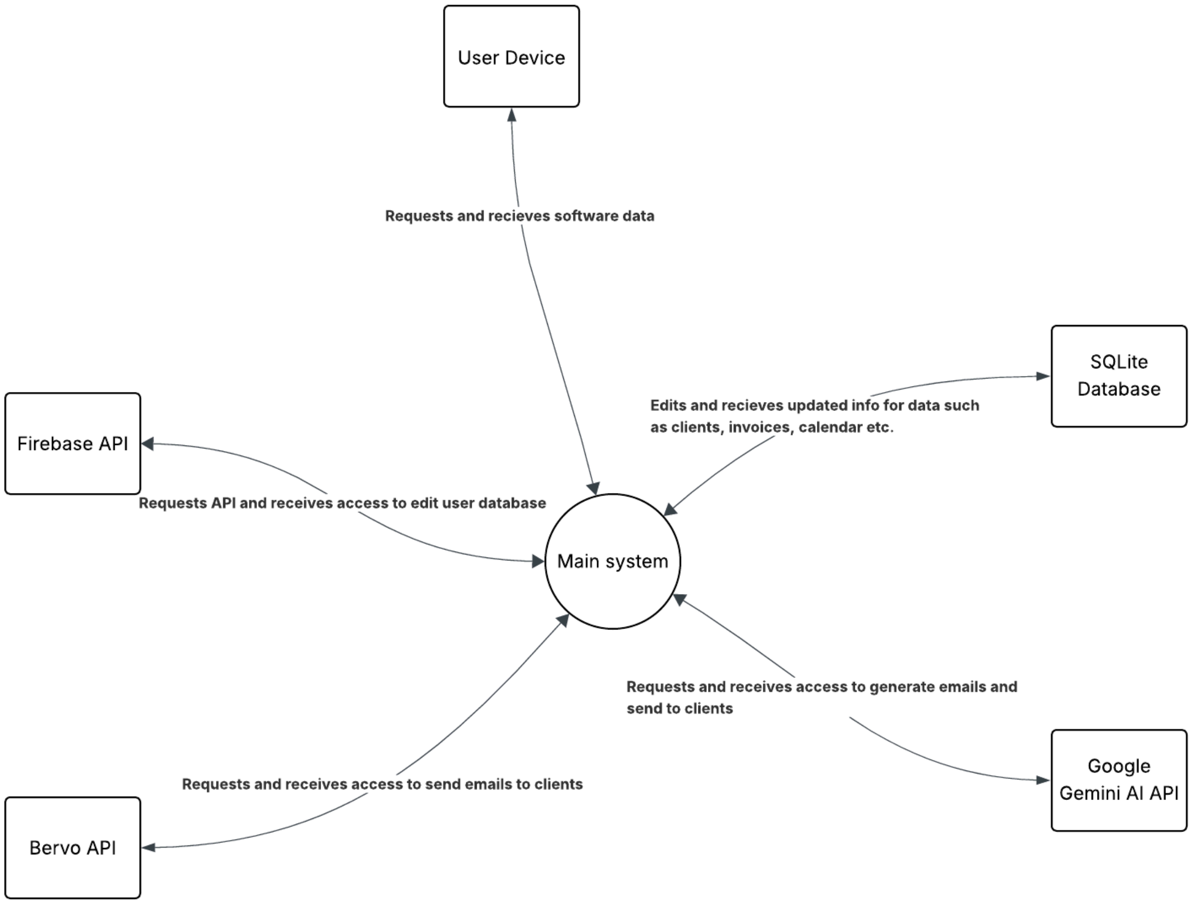




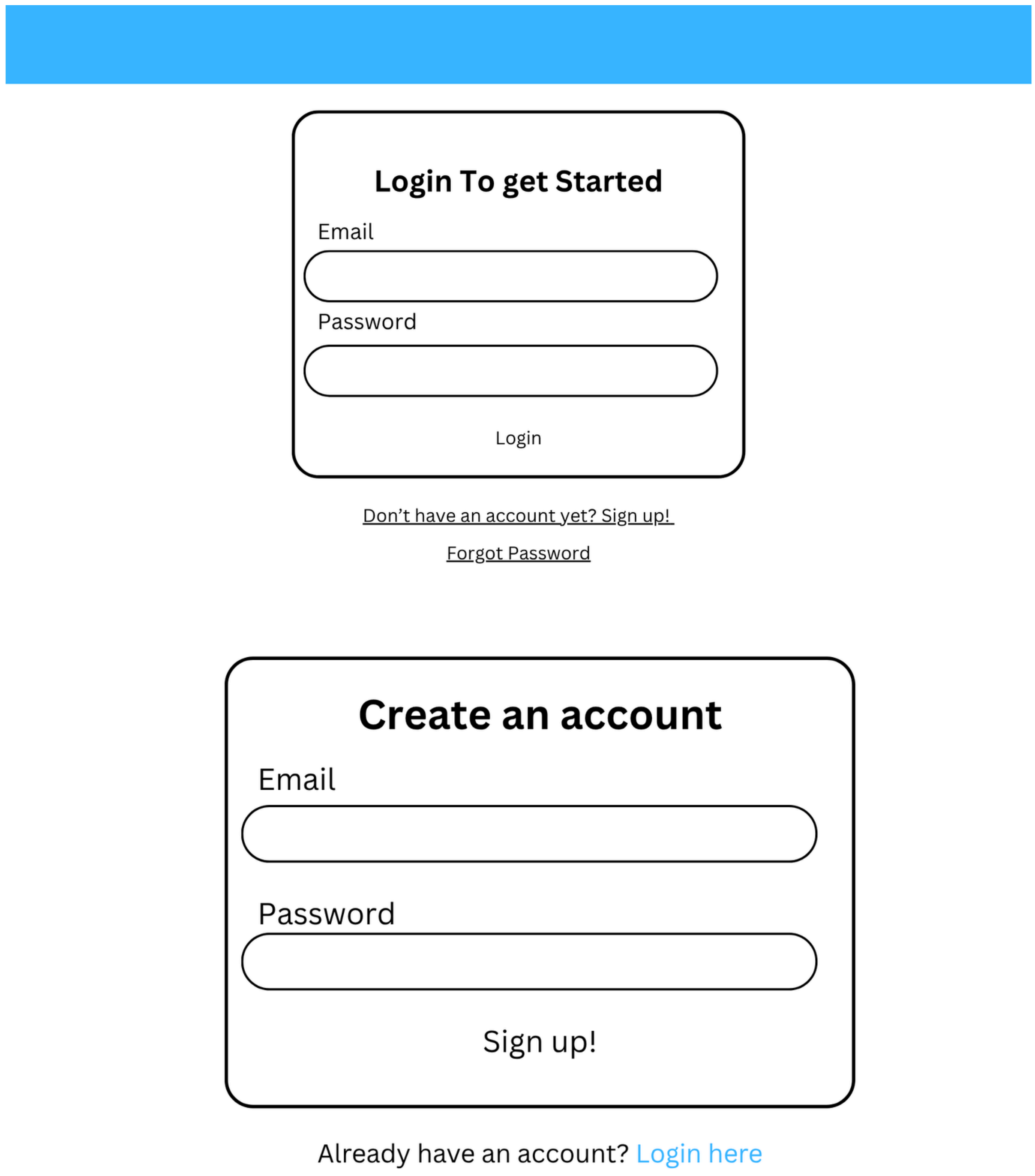
https://lucid.app/lucidchart/b56be2f7-cca5-417a-8463-7edcf3be62ab/edit?viewport_loc=-150%2C-74%2C3326%2C1642%2C0_0&invitationId=inv_a50d4da7-7699-4fab-a8f4-f463ab7fef3a

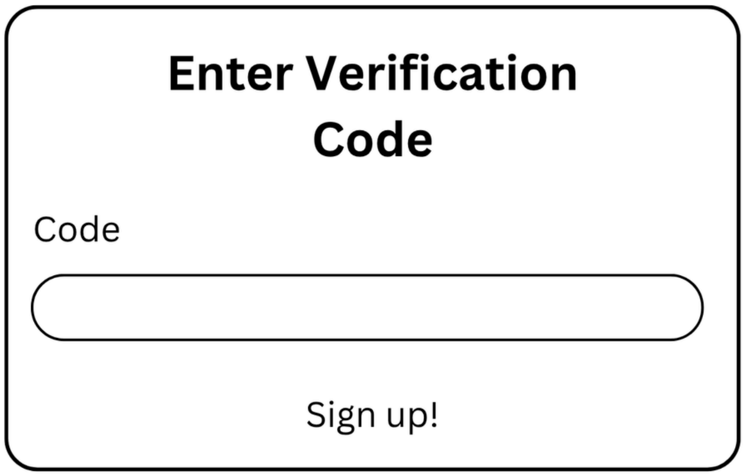


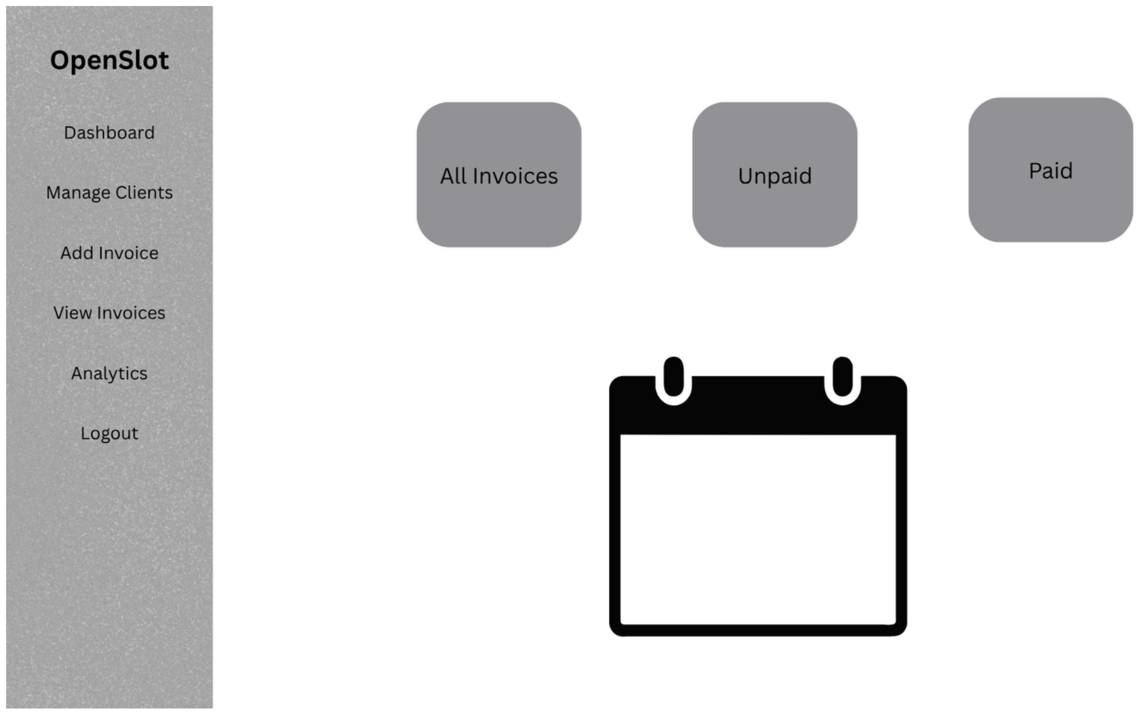
https://lucid.app/lucidchart/51e104d4-b6f4-411e-95f8-69bac6df656a/edit?viewport_loc=-4019%2C-937%2C10394%2C5131%2C0_0&invitationId=inv_706e994a-c2e0-4fc9-b2f7-1ae687a14f8a
(Couldn’t find a function to add the lines with dots, so showcased the data flow through the arrows)

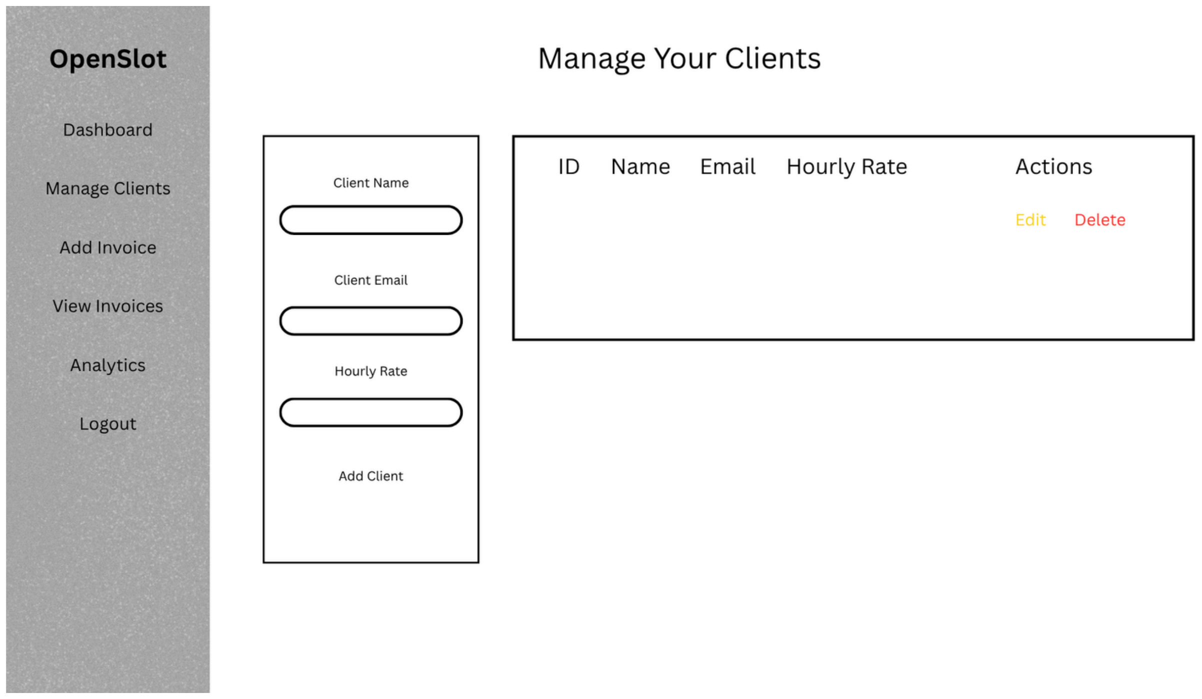

User Device
Requests and recieves software data
SQLite Database
Edits and recieves updated info for data such as clients, invoices, calendar etc.
Firebase API
Requests API and receives access to edit user database
Main system
Requests and receives access to generate emails and send to clients
Google Gemini AI API
Requests and receives access to send emails to clients
Bervo API

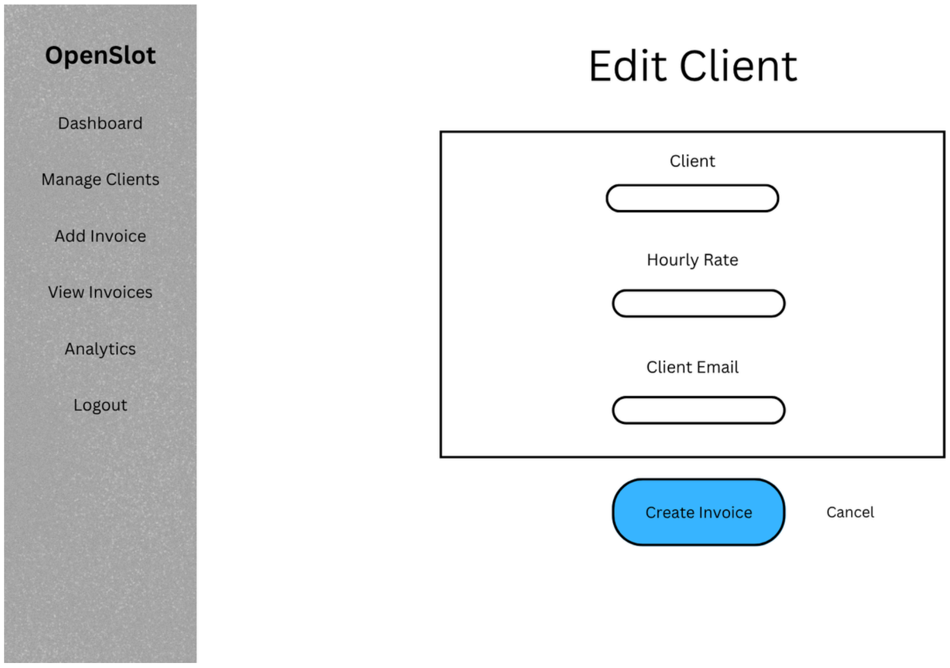
GUI Design

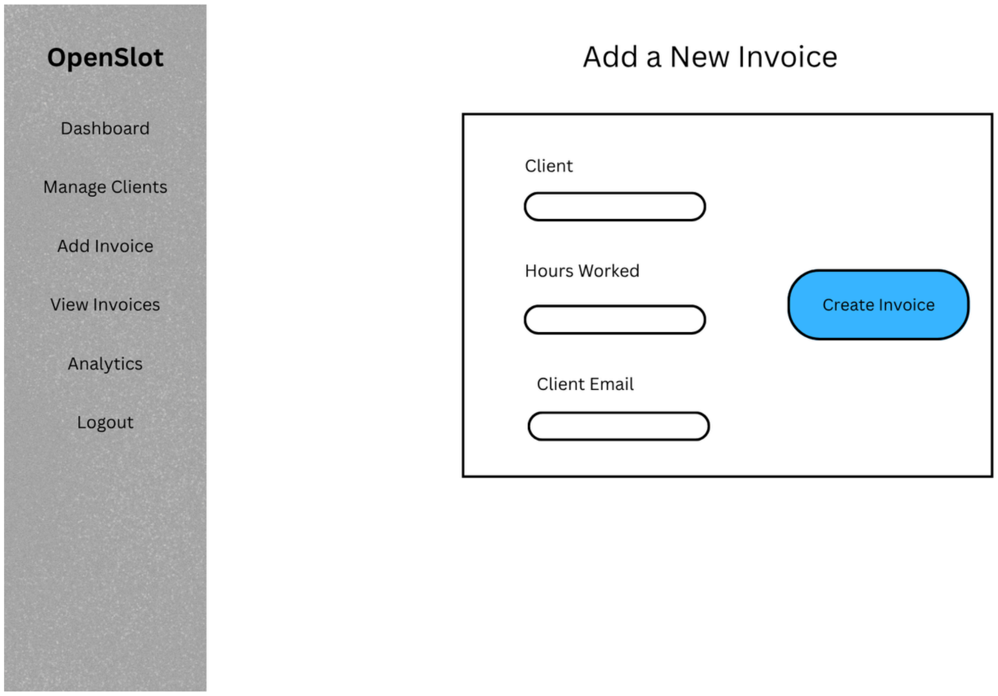

Login To get Started
Email
Password
Login
Don't have an account yet? Sign up!
Forgot Password
Create an account
Email
Password
Sign up!
Already have an account? Login here

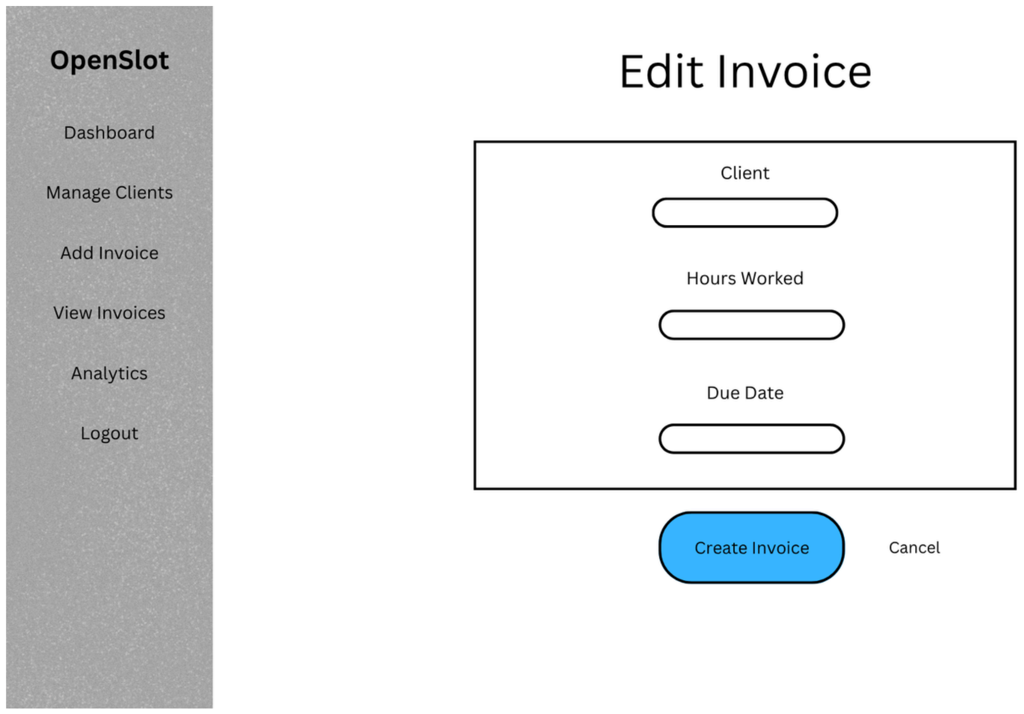

Enter Verification Code
Code
Sign up!

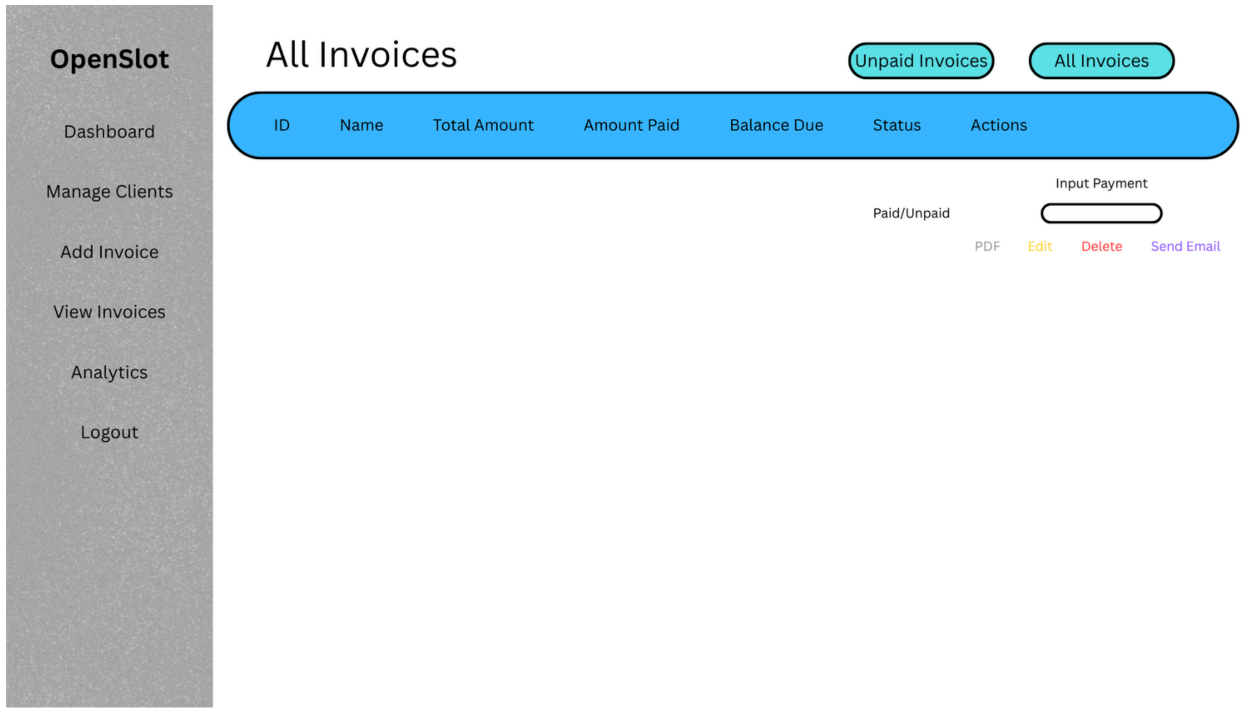

OpenSlot
Dashboard
Manage Clients
Add Invoice
View Invoices
Analytics
Logout
All Invoices
Unpaid
Paid

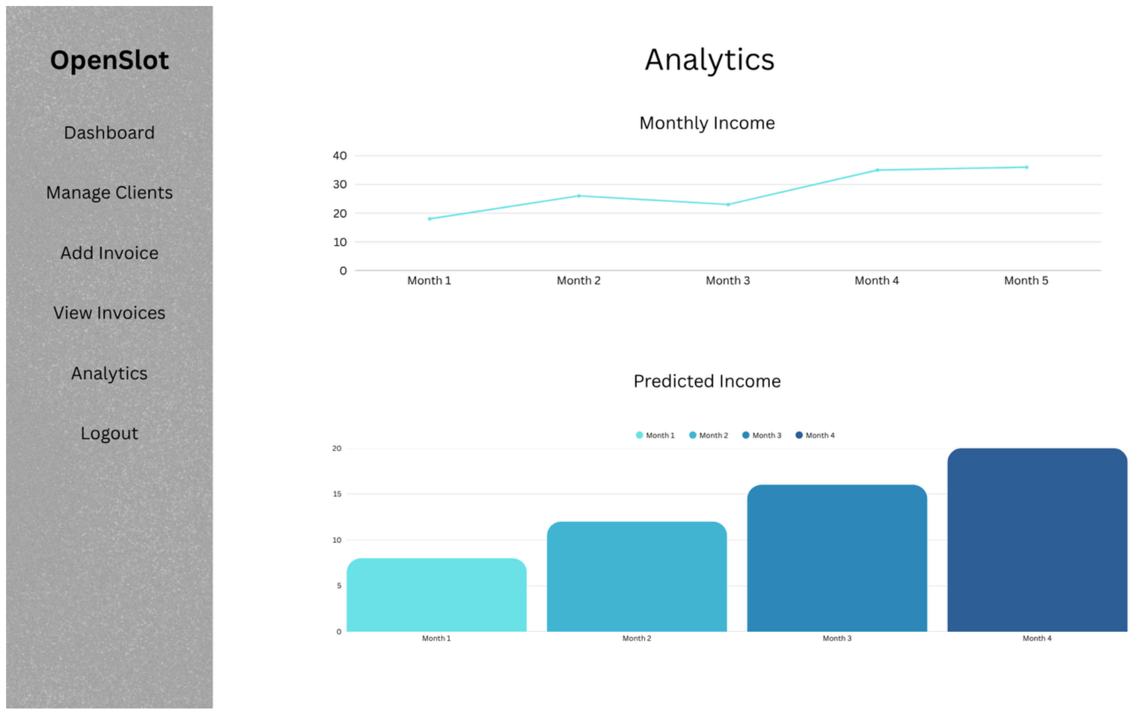

OpenSlot
Dashboard
Manage Clients
Add Invoice
View Invoices
Analytics
Logout
Manage Your Clients
Client Name
Client Email
Hourly Rate
Add Client
ID
Name
Email
Hourly Rate
Actions
Edit
Delete


OpenSlot
Dashboard
Manage Clients
Add Invoice
View Invoices
Analytics
Logout
Edit Client
Client
Hourly Rate
Client Email
Create Invoice
Cancel


OpenSlot
Dashboard
Manage Clients
Add Invoice
View Invoices
Analytics
Logout
Add a New Invoice
Client
Hours Worked
Client Email
Create Invoice


OpenSlot
Dashboard
Manage Clients
Add Invoice
View Invoices
Analytics
Logout
Edit Invoice
Client
Hours Worked
Due Date
Create Invoice
Cancel


OpenSlot
Dashboard
Manage Clients
Add Invoice
View Invoices
Analytics
Logout
All Invoices
Unpaid Invoices
All Invoices
ID
Name
Total Amount
Amount Paid
Balance Due
Status
Actions
Input Payment
Paid/Unpaid
PDF
Edit
Delete
Send Email



Data Dictionary

Function / Route	HTTP Methods	Description
/ (index)	GET	Displays the main dashboard, showing summary statistics (total, paid, unpaid invoices) and the calendar.
/settings	GET, POST	Displays the admin's profile/settings page and handles updates to business information (name, address, etc.).
/register	GET, POST	Displays the user registration form and handles the creation of new admin accounts.
/login	GET	Displays the FirebaseUI login interface for the

		administrator.
/sessionLogin	POST	API endpoint that validates the Firebase token and creates a user session upon successful login.
/verify	GET, POST	Displays the email verification form and validates the user-submitted code.
/forgot_password	GET, POST	Displays a form for the user to enter their email to start the password reset process.
/verify_reset_code	GET, POST	Displays a form for the user to enter the password reset code sent to their email.
/reset_password	GET, POST	Displays the final form for the user to set a new password after successful verification.
/client_login	GET, POST	(Client Portal) Displays the login page for clients and handles their authentication.
/client/dashboard	GET	(Client Portal) The main dashboard for a logged-in client, showing a list of their invoices.
/clients	GET	Displays the client management page with a list of all clients and a form to add new ones.
/add_client	POST	Processes the form submission from the /clients page to create a new client in the database.

/edit_client/<id>	GET, POST	Displays a form pre-filled with a specific client's data and handles updates.
/delete_client/<id>	GET	Deletes a specific client and their associated records from the database.
/invoices	GET	Displays the main table of all invoices. Can be filtered by status (e.g., /invoices?status=Paid).
/add_invoice	GET, POST	Displays the form to create a new invoice and processes its submission.
/edit_invoice/<id>	GET, POST	Displays a form to edit an existing invoice's details (hours, due date) and handles updates.
/delete_invoice/<id>	GET	Deletes a specific invoice and its associated payment records from the database.
/record_payment/<id>	POST	Records a new payment against a specific invoice and updates the invoice's status accordingly.
/recurring	GET	Displays the page for managing recurring invoice schedules.
/recurring/add	GET, POST	Displays a form to create a new recurring invoice template and schedule.
/recurring/run	POST	A system endpoint to check all schedules and automatically generate any invoices that are due.

/get_events	GET	API endpoint that returns all calendar events (invoice due dates and custom events) as JSON.
/add_event	POST	API endpoint that adds a new custom event to the calendar from user input.
/delete_event/<id>	DELETE	API endpoint that deletes a custom event from the calendar.
/analytics	GET	Renders the analytics dashboard page, which then uses JavaScript to fetch data for the graphs.
/api/cash_flow_data	GET	API endpoint that returns historical payment data, grouped by month, used for the income graph.
/api/predicted_growth_data	GET	API endpoint that calculates and returns a linear regression trend line and forecast.
/reports/revenue_by_client	GET	Displays a new report page showing a breakdown of revenue earned from each client.
/api/reports/revenue_by_client_data	GET	API endpoint that provides the data needed to generate the revenue-by-client chart.
/draft_reminder_email/<id>	GET	AI-powered endpoint that takes an invoice ID and returns a professionally drafted reminder email.
/api/send_reminder	POST	Takes a final email body

		and invoice ID, generates a PDF, and sends the email with the attachment.
send_verification_email	Helper	(Not a route) Internal function that sends registration or password reset emails via Brevo.
/download_invoice/<id>	GET	Generates and serves a detailed PDF for a single, specific invoice.
/download_unpaid_invoices_pdf	GET	Generates and serves a summary PDF report of all unpaid invoices.
/download_full_report	GET	Generates and serves a comprehensive PDF report of all invoices for archival or tax purposes.
generate_invoice_pdf	Helper	(Not a route) Internal function that uses the ReportLab library to create a detailed PDF from invoice data.
generate_full_tax_report_pdf	Helper	(Not a route) A dedicated function for creating the comprehensive tax report PDF.
calculate_linear_regression	Helper	(Not a route) A mathematical helper function to calculate trend lines for analytics.
/logout	GET	Clears the user's session data and redirects them to the login page.

Process Log

Date	Event
10/04/2025	Started work on my index Page
15/04/2025	Continued work
18/04/2025	Continued Index Page
19/04/2025	Continued work on index page
21/04/2025	Started Add Invoice Page
22/04/2025	Continued Add Invoice Page
23/04/2025	Finished add invoice functionality
24/04/2025	Started View Invoice Page
29/04/2025	Continued View Invoice Page and linked back to index
03/05/2025	Started Login Page
06/05/2025	Finished Login and started on Registration Page
08/05/2025	Finished Registration Page and added verification codes
11/05/2025	Imported Calendar library and created calendar.html
13/05/2025	Fixed bugs in my add invoice functionality
15/05/2025	Fixed bugs that arose from fixing the previous bug
20/05/2025	Started working on CSS to make the index page look better
25/05/2025	Moved the calendar tab to the index page to integrate the calendar
29/05/2025	Started Manage Clients Page
3/06/2025	Continued Manage Clients Page
5/06/2025	Continued Manage Clients Page

Date	**Event**
10/04/2025	Started work on my index Page
15/04/2025	Continued work
6/06/2025	Worked through bugs that arose when trying to add clients to the SQL database
7/06/2025	Fixed the Manage Clients Page
9/06/2025	Linked the client database to the Add Invoice Page
11/06/2025	Worked on CSS for the Index Page
12/06/2025	Worked on CSS for the Index and Add Invoice Page
14/06/2025	Worked on CSS for Add Invoice and View Invoice Page
15/06/2025	Added PDF functionality to View Invoices Page
16/06/2025	Continued working on PDF functionality
17/06/2025	Started work on Analytics Page
19/06/2025	Continued to work through bugs and errors to get the correct graphs to show
20/06/2025	Started adding AI functionality that allows the user to send emails straight to clients
21/06/2025	Finished AI functionality and cleaned up CSS
22/06/2025	Final Bug checks
23/06/2025	Corrected leftover bugs from the 22/06/2025
24/06/2025	Final Checks and Submission

Client Interactions

1. Discussed the requirements with the client after starting the design process so I could create a sensible GUI design that involved all necessary functionality

2. After building the basic functionality with the client, talked again to ensure I had included all functionality that was requested, and this allowed me to catch one that I had missed (Email invoice functionality) and ensure that was included.
3. Sent the client pictures of the almost finished product to ensure it was up to their standards and this allowed me to edit the aesthetics of the site to suit their requests.
4. Sent the project to the client and discussed on how the project progressed overall and how I handled issues that arose as the client wanted to know how long it took to make a project of this calibre and how stressful it was.

Software Solution

The solution was developed using a combination of Python, HTML, and CSS to meet the client's requirements for a browser-based application.

Programming Language: Python.

- Web Framework: Flask, a lightweight Python framework for building the web application's routes and logic.
- Database: SQLite, a serverless SQL database for storing all application data, including user, client, and invoice information.
- Authentication Service: Google Firebase Authentication for handling user registration, login, and secure password management.
- External APIs: Brevo API for programmatically sending verification and invoice emails, and Gemini API for drafting and sending emails

The program incorporates a combination of control structures, modules, functions, and file handling to deliver a robust solution

- Security: A multi-layered security model was implemented.
    - Authentication: User identity is managed by Google Firebase, a secure, industry-standard service.
    - Authorization: The custom @login_required decorator protects all sensitive routes, redirecting unauthenticated users to the login page.
    - Data Segregation: All database queries are filtered by the logged-in user's ID (user_id). This ensures a user can only ever access or modify their own clients and invoices.
    - Email Verification: The /register and /verify routes work together with the Brevo API to ensure a user controls the email address they sign up with, preventing account misuse.
- Payment Status Logic: The /record_payment function implements a crucial business rule. It retrieves the total invoice amount and compares it to the sum of all payments for that invoice. Based on the result, it programmatically updates the invoice status to 'Paid' or 'Partially Paid', ensuring data integrity.

- Data Relationships: Complex data is managed through SQL JOIN statements. For instance, the /view_invoices route uses a LEFT JOIN to efficiently retrieve each invoice along with the total amount paid from the payments table in a single database query.
- The /generate_receipt and /email_invoice routes demonstrate advanced file handling. The reportlab library creates a PDF document in an in-memory binary buffer (io.BytesIO). This buffer is then used to either send the file directly to the user for download (send_file) or is encoded into base64 to be attached to an email via the Brevo API payload.

Testing and Maintenance

Bug #	**Date Found**	**What is the bug?**	**How I found it?**	**How I fixed it?**
Bug #1	12/4/2025	Lack of user differentiation	Different users were able to access each others' invoices and interact with them	I modified SQL queries that fetch or modify data to include a WHERE user_id = ? clause. I then passed the user_id from the current user's session as a parameter to these queries.
Bug #2	19/4/2025	Database Locking	Testing with simulated users and reports simultaneously	Added a timeout parameter to the SQL code, allowing it to try and form a connection before locking the database
Bug #3	8/5/2025	Verification code wouldn't send	The verification code for new users in the registration process wouldn't send, resulting in them being stuck on the page eternally	I debugged the send_verification_email function and discovered the JSON payload being sent to the Brevo email API was corrupted. I corrected the dictionary's

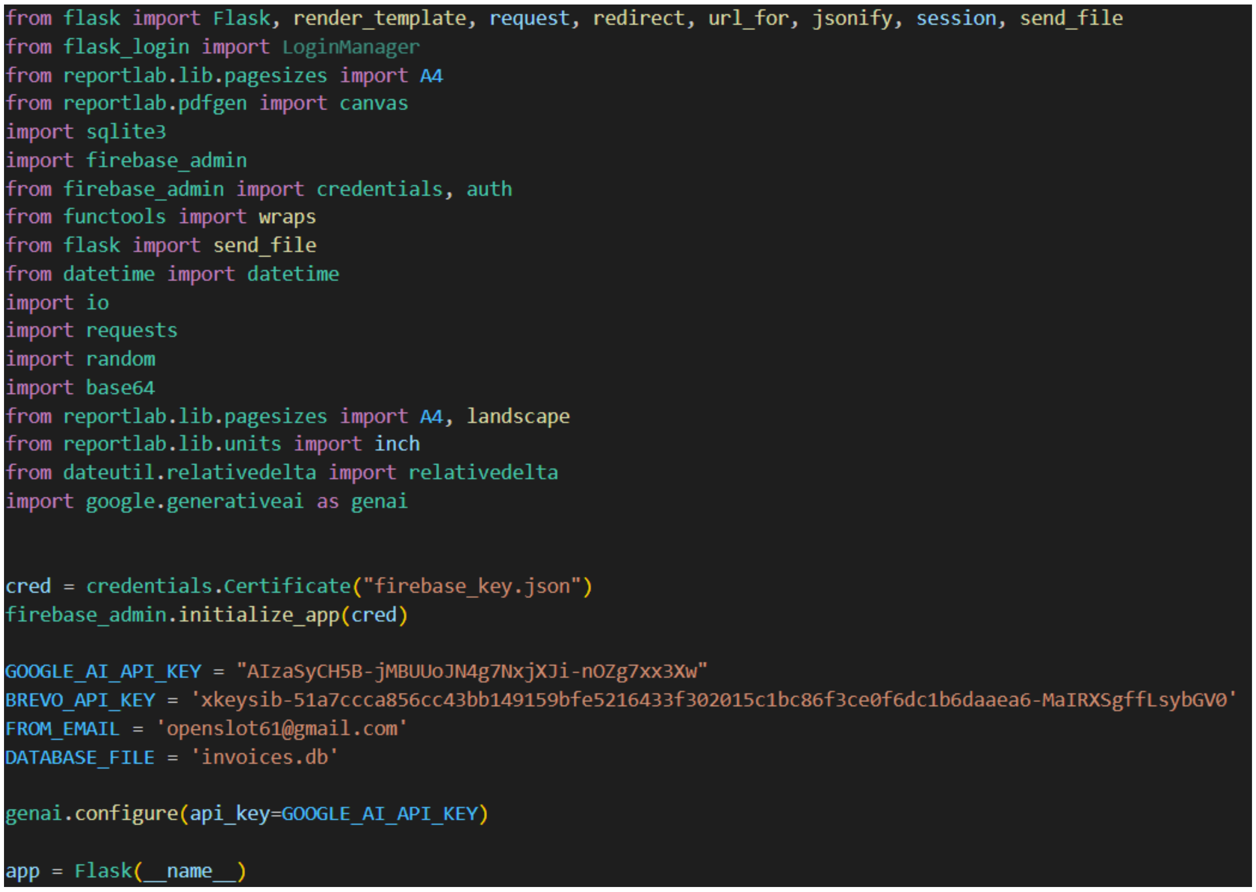
				structure to exactly match the API's documentation.
Bug #4	17/5/2025	Calendar events weren’t linked to invoice due dates	I tested the due date functionality and came across the calendar not linking correctly, leading to no events being added	I added a specific SQL query to select all invoices for the current user that have a valid due date and are not yet paid.
Bug #5	15/6/2025	PDF generation wouldn’t differentiate between unpaid and paid invoices	The “Unpaid” and “All” invoices buttons would produce the same pdf rather than 2 different ones	The button for downloading "Unpaid Invoices" was incorrectly pointing to the wrong URL I fixed this by correcting the href attribute in the HTML <a> tag to point to the correct route
Bug #6	21/6/2025	AI for emails was locked	The AI kept producing an error message when prompted for an email	I had to properly integrate the AI API key into my code and set up the AI on the Google dashboard

Security Features

A critical security feature of this application is its inherent protection against Cross-Site Scripting (XSS) attacks, which is provided by default through the Jinja2 templating engine used by Flask. XSS vulnerabilities occur when an application allows malicious user-submitted code, typically JavaScript, to be rendered directly on a webpage, where it can then execute in other users' browsers. Jinja2 prevents this with a feature called automatic HTML escaping. By default, any data rendered using the {{ ... }} syntax is automatically sanitized before being sent to the browser. Dangerous characters like < and > are converted into their harmless HTML entity equivalents (&lt; and &gt;). This means that if a user attempts to save a client name as <script>alert('attack');</script>, Jinja2 ensures it is displayed only as plain text on the screen, rather than being executed as a script. This default-on security posture is a fundamental strength, as it requires no extra code to implement and can only be disabled by explicitly using the ‘safe’ filter, thus making the application secure by design against this common and serious vulnerability.

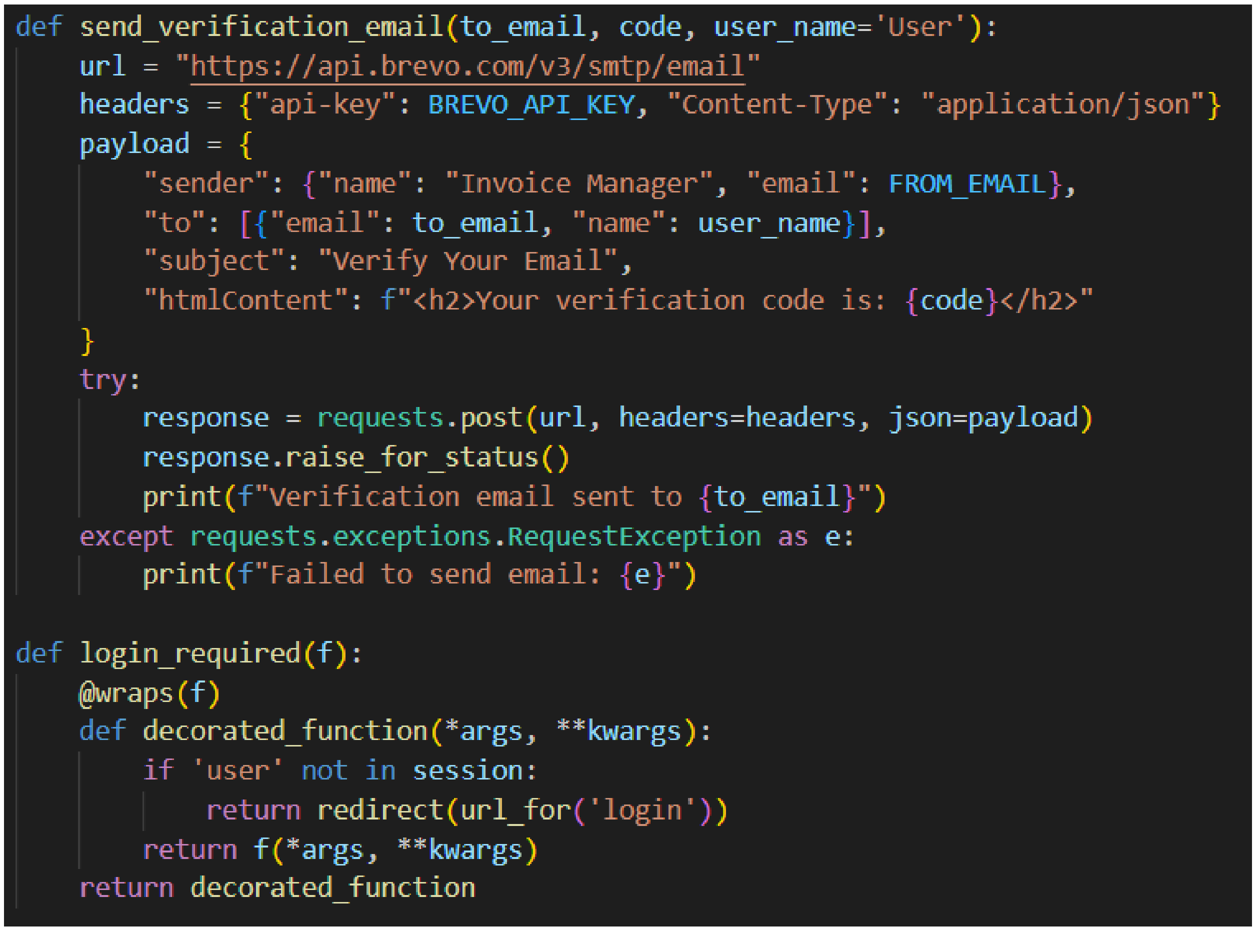
Code Snippets

Setting up libraries and codes for APIs

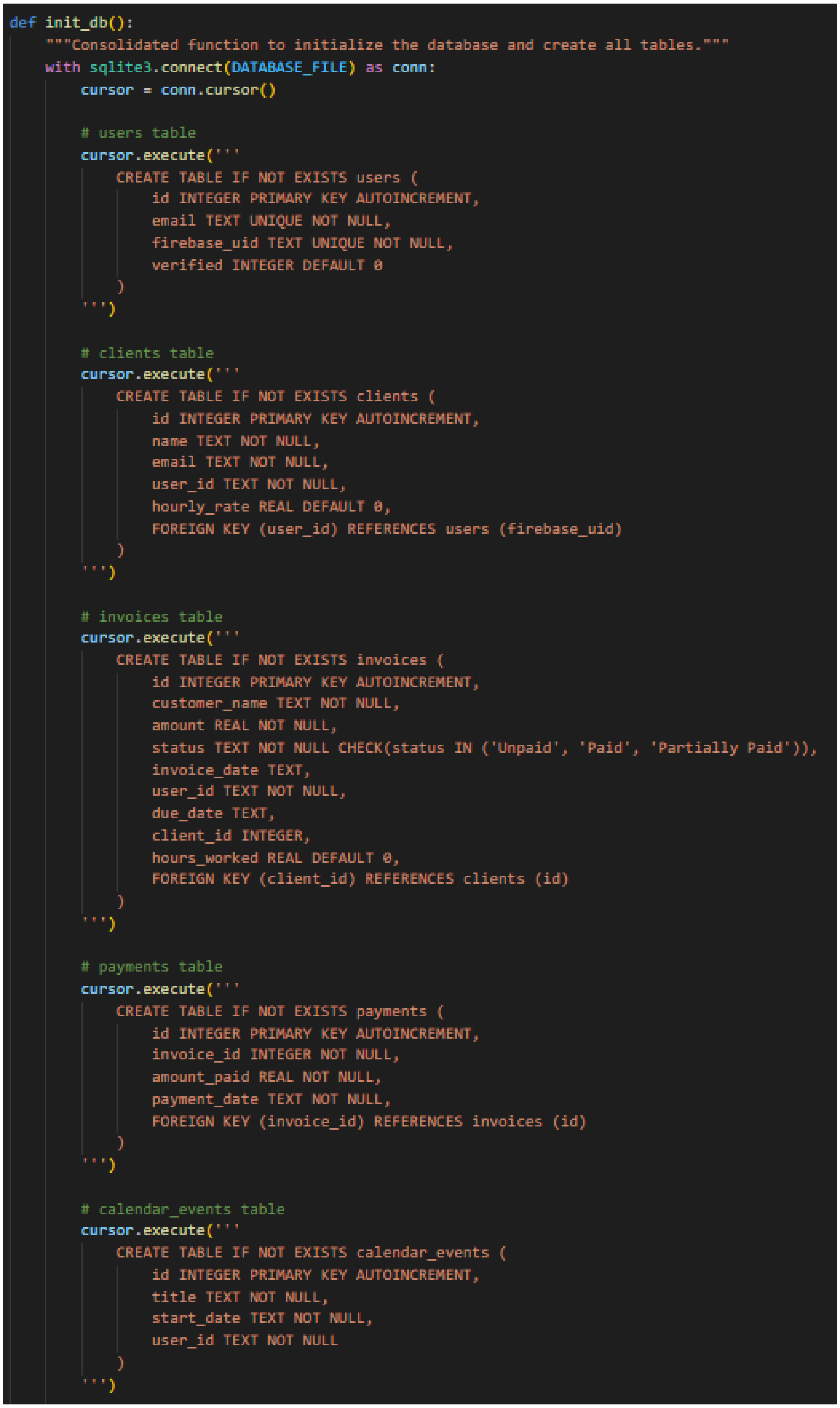


```python
from flask import Flask, render_template, request, redirect, url_for, jsonify, session, send_file
from flask_login import LoginManager
from reportlab.lib.pagesizes import A4
from reportlab.pdfgen import canvas
import sqlite3
import firebase_admin
from firebase_admin import credentials, auth
from functools import wraps
from flask import send_file
from datetime import datetime
import io
import requests
import random
import base64
from reportlab.lib.pagesizes import A4, landscape
from reportlab.lib.units import inch
from dateutil.relativedelta import relativedelta
import google.generativeai as genai


cred = credentials.Certificate("firebase_key.json")
firebase_admin.initialize_app(cred)

GOOGLE_AI_API_KEY = "AIzaSyCH5B-jMBUUoJN4g7NxjXJi-nOZg7xx3Xw"
BREVO_API_KEY = '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'
FROM_EMAIL = 'openslot61@gmail.com'
DATABASE_FILE = 'invoices.db'

genai.configure(api_key=GOOGLE_AI_API_KEY)

app = Flask(__name__)
```

Defining helpers

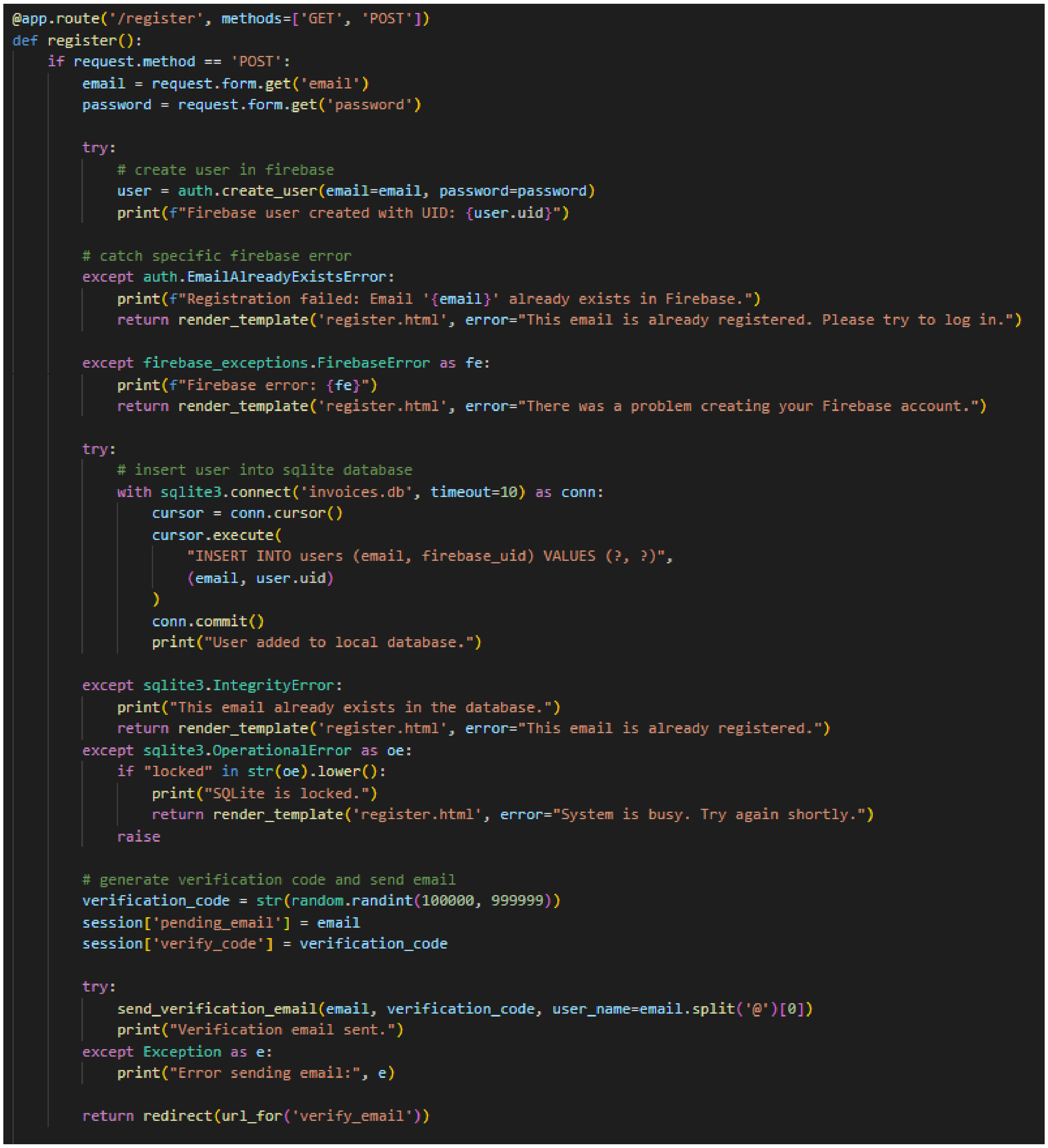

```python
def send_verification_email(to_email, code, user_name='User'):
    url = "https://api.brevo.com/v3/smtp/email"
    headers = {"api-key": BREVO_API_KEY, "Content-Type": "application/json"}
    payload = {
        "sender": {"name": "Invoice Manager", "email": FROM_EMAIL},
        "to": [{"email": to_email, "name": user_name}],
        "subject": "Verify Your Email",
        "htmlContent": f"<h2>Your verification code is: {code}</h2>"
    }
    try:
        response = requests.post(url, headers=headers, json=payload)
        response.raise_for_status()
        print(f"Verification email sent to {to_email}")
    except requests.exceptions.RequestException as e:
        print(f"Failed to send email: {e}")

def login_required(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):
        if 'user' not in session:
            return redirect(url_for('login'))
        return f(*args, **kwargs)
    return decorated_function
```

Setting up SQL tables



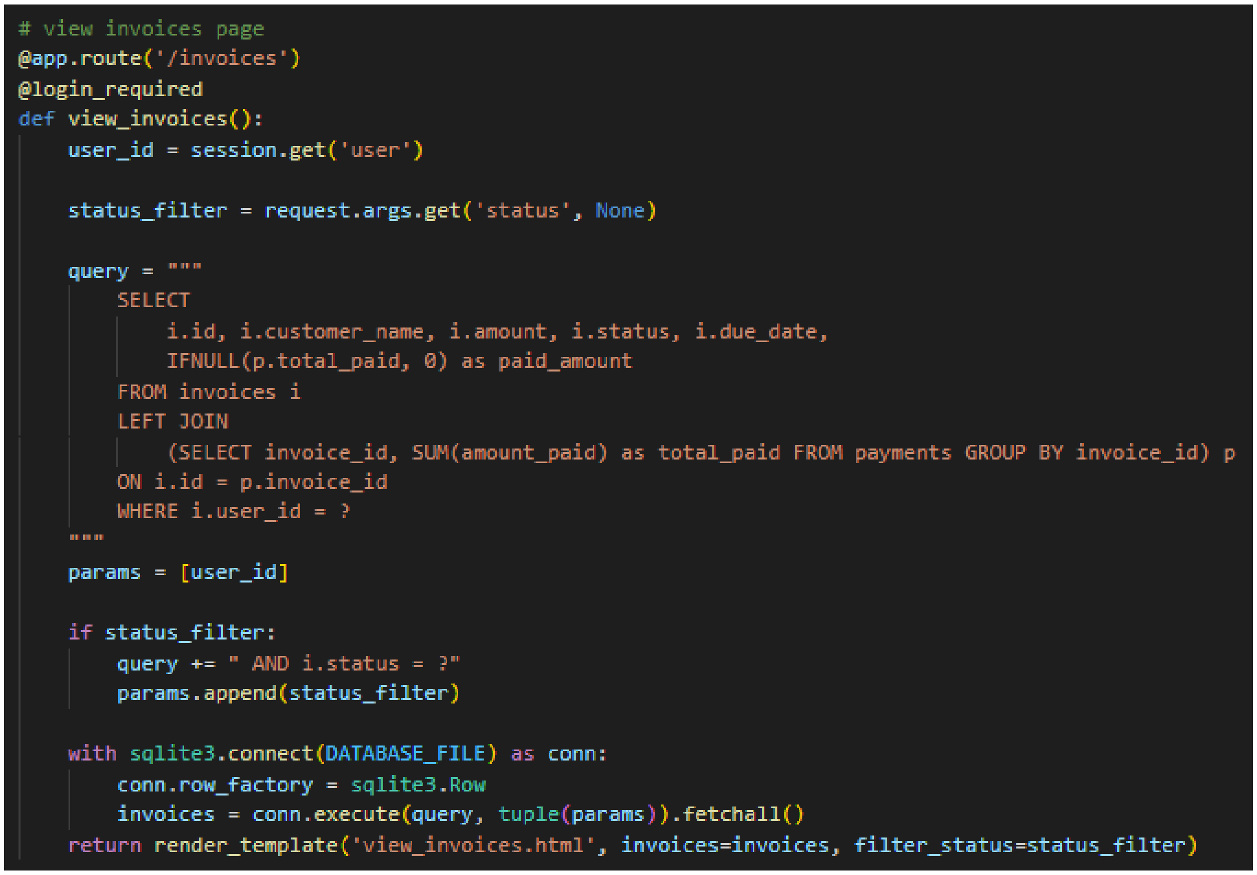
```python
def init_db():
    """Consolidated function to initialize the database and create all tables."""
    with sqlite3.connect(DATABASE_FILE) as conn:
        cursor = conn.cursor()

        # users table
        cursor.execute('''
            CREATE TABLE IF NOT EXISTS users (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                email TEXT UNIQUE NOT NULL,
                firebase_uid TEXT UNIQUE NOT NULL,
                verified INTEGER DEFAULT 0
            )
        ''')

        # clients table
        cursor.execute('''
            CREATE TABLE IF NOT EXISTS clients (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                name TEXT NOT NULL,
                email TEXT NOT NULL,
                user_id TEXT NOT NULL,
                hourly_rate REAL DEFAULT 0,
                FOREIGN KEY (user_id) REFERENCES users (firebase_uid)
            )
        ''')

        # invoices table
        cursor.execute('''
            CREATE TABLE IF NOT EXISTS invoices (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                customer_name TEXT NOT NULL,
                amount REAL NOT NULL,
                status TEXT NOT NULL CHECK(status IN ('Unpaid', 'Paid', 'Partially Paid')),
                invoice_date TEXT,
                user_id TEXT NOT NULL,
                due_date TEXT,
                client_id INTEGER,
                hours_worked REAL DEFAULT 0,
                FOREIGN KEY (client_id) REFERENCES clients (id)
            )
        ''')

        # payments table
        cursor.execute('''
            CREATE TABLE IF NOT EXISTS payments (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                invoice_id INTEGER NOT NULL,
                amount_paid REAL NOT NULL,
                payment_date TEXT NOT NULL,
                FOREIGN KEY (invoice_id) REFERENCES invoices (id)
            )
        ''')

        # calendar_events table
        cursor.execute('''
            CREATE TABLE IF NOT EXISTS calendar_events (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                title TEXT NOT NULL,
                start_date TEXT NOT NULL,
                user_id TEXT NOT NULL
            )
        ''')
```

Register function



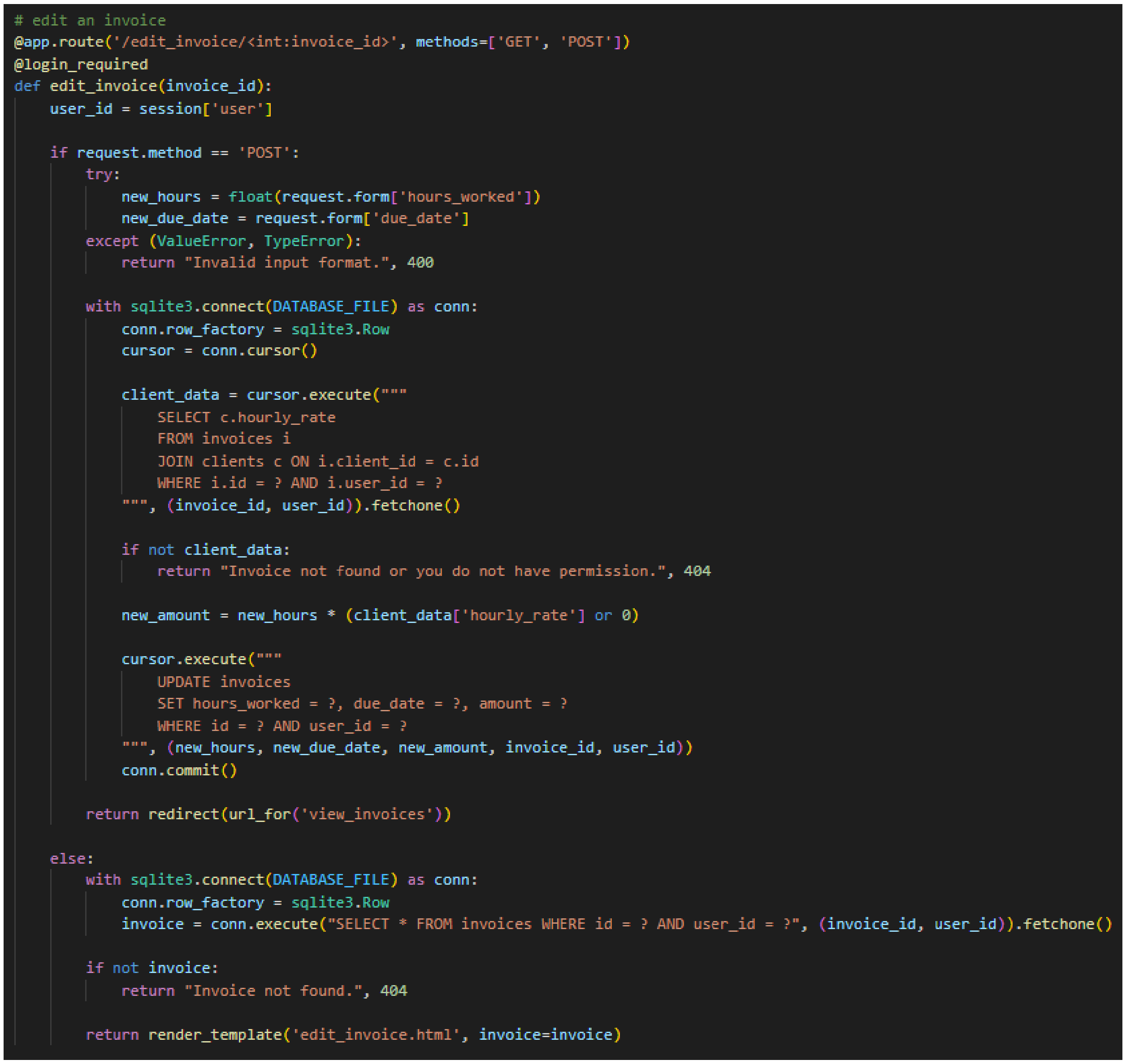
```
@app.route('/register', methods=['GET', 'POST'])
def register():
    if request.method == 'POST':
        email = request.form.get('email')
        password = request.form.get('password')

        try:
            # create user in firebase
            user = auth.create_user(email=email, password=password)
            print(f"Firebase user created with UID: {user.uid}")

        # catch specific firebase error
        except auth.EmailAlreadyExistsError:
            print(f"Registration failed: Email '{email}' already exists in Firebase.")
            return render_template('register.html', error="This email is already registered. Please try to log in.")

        except firebase_exceptions.FirebaseError as fe:
            print(f"Firebase error: {fe}")
            return render_template('register.html', error="There was a problem creating your Firebase account.")

        try:
            # insert user into sqlite database
            with sqlite3.connect('invoices.db', timeout=10) as conn:
                cursor = conn.cursor()
                cursor.execute(
                    "INSERT INTO users (email, firebase_uid) VALUES (?, ?)",
                    (email, user.uid)
                )
                conn.commit()
                print("User added to local database.")

        except sqlite3.IntegrityError:
            print("This email already exists in the database.")
            return render_template('register.html', error="This email is already registered.")
        except sqlite3.OperationalError as oe:
            if "locked" in str(oe).lower():
                print("SQLite is locked.")
                return render_template('register.html', error="System is busy. Try again shortly.")
            raise

        # generate verification code and send email
        verification_code = str(random.randint(100000, 999999))
        session['pending_email'] = email
        session['verify_code'] = verification_code

        try:
            send_verification_email(email, verification_code, user_name=email.split('@')[0])
            print("Verification email sent.")
        except Exception as e:
            print("Error sending email:", e)

        return redirect(url_for('verify_email'))
```

View invoices page



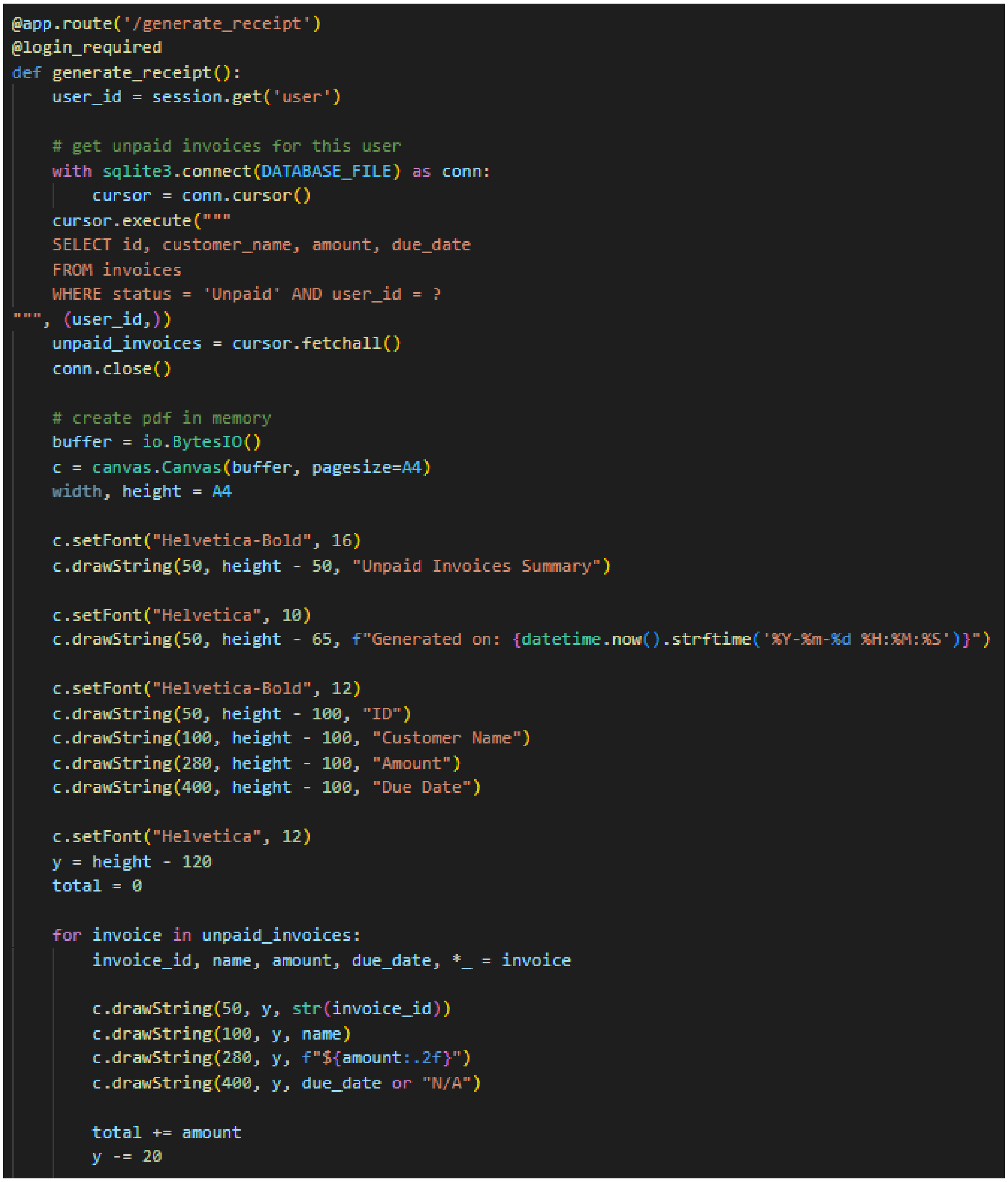
```
# view invoices page
@app.route('/invoices')
@login_required
def view_invoices():
    user_id = session.get('user')

    status_filter = request.args.get('status', None)

    query = """
        SELECT
            i.id, i.customer_name, i.amount, i.status, i.due_date,
            IFNULL(p.total_paid, 0) as paid_amount
        FROM invoices i
        LEFT JOIN
            (SELECT invoice_id, SUM(amount_paid) as total_paid FROM payments GROUP BY invoice_id) p
        ON i.id = p.invoice_id
        WHERE i.user_id = ?
    """
    params = [user_id]

    if status_filter:
        query += " AND i.status = ?"
        params.append(status_filter)

    with sqlite3.connect(DATABASE_FILE) as conn:
        conn.row_factory = sqlite3.Row
        invoices = conn.execute(query, tuple(params)).fetchall()
    return render_template('view_invoices.html', invoices=invoices, filter_status=status_filter)
```

Edit invoice function



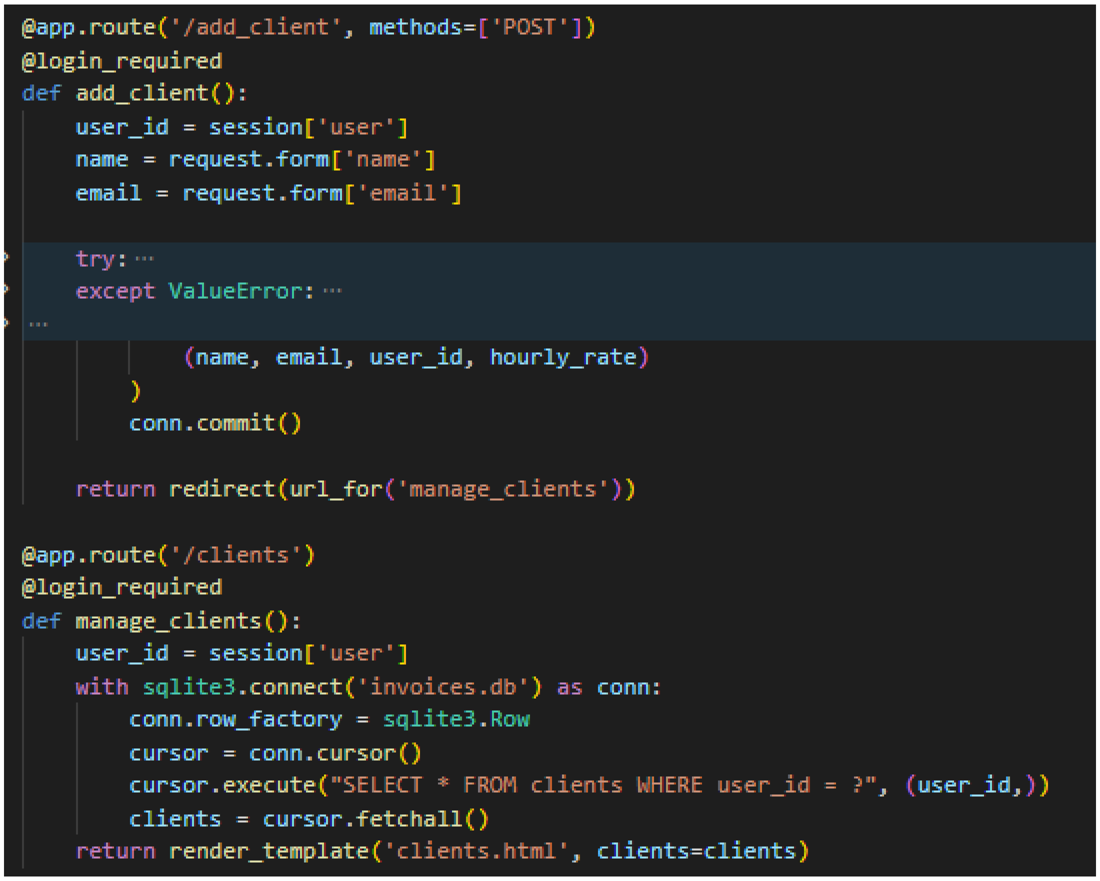
```python
# edit an invoice
@app.route('/edit_invoice/<int:invoice_id>', methods=['GET', 'POST'])
@login_required
def edit_invoice(invoice_id):
    user_id = session['user']

    if request.method == 'POST':
        try:
            new_hours = float(request.form['hours_worked'])
            new_due_date = request.form['due_date']
        except (ValueError, TypeError):
            return "Invalid input format.", 400

        with sqlite3.connect(DATABASE_FILE) as conn:
            conn.row_factory = sqlite3.Row
            cursor = conn.cursor()

            client_data = cursor.execute("""
                SELECT c.hourly_rate
                FROM invoices i
                JOIN clients c ON i.client_id = c.id
                WHERE i.id = ? AND i.user_id = ?
            """, (invoice_id, user_id)).fetchone()

            if not client_data:
                return "Invoice not found or you do not have permission.", 404

            new_amount = new_hours * (client_data['hourly_rate'] or 0)

            cursor.execute("""
                UPDATE invoices
                SET hours_worked = ?, due_date = ?, amount = ?
                WHERE id = ? AND user_id = ?
            """, (new_hours, new_due_date, new_amount, invoice_id, user_id))
            conn.commit()

        return redirect(url_for('view_invoices'))

    else:
        with sqlite3.connect(DATABASE_FILE) as conn:
            conn.row_factory = sqlite3.Row
            invoice = conn.execute("SELECT * FROM invoices WHERE id = ? AND user_id = ?", (invoice_id, user_id)).fetchone()

        if not invoice:
            return "Invoice not found.", 404

        return render_template('edit_invoice.html', invoice=invoice)
```

Invoice PDF generation route



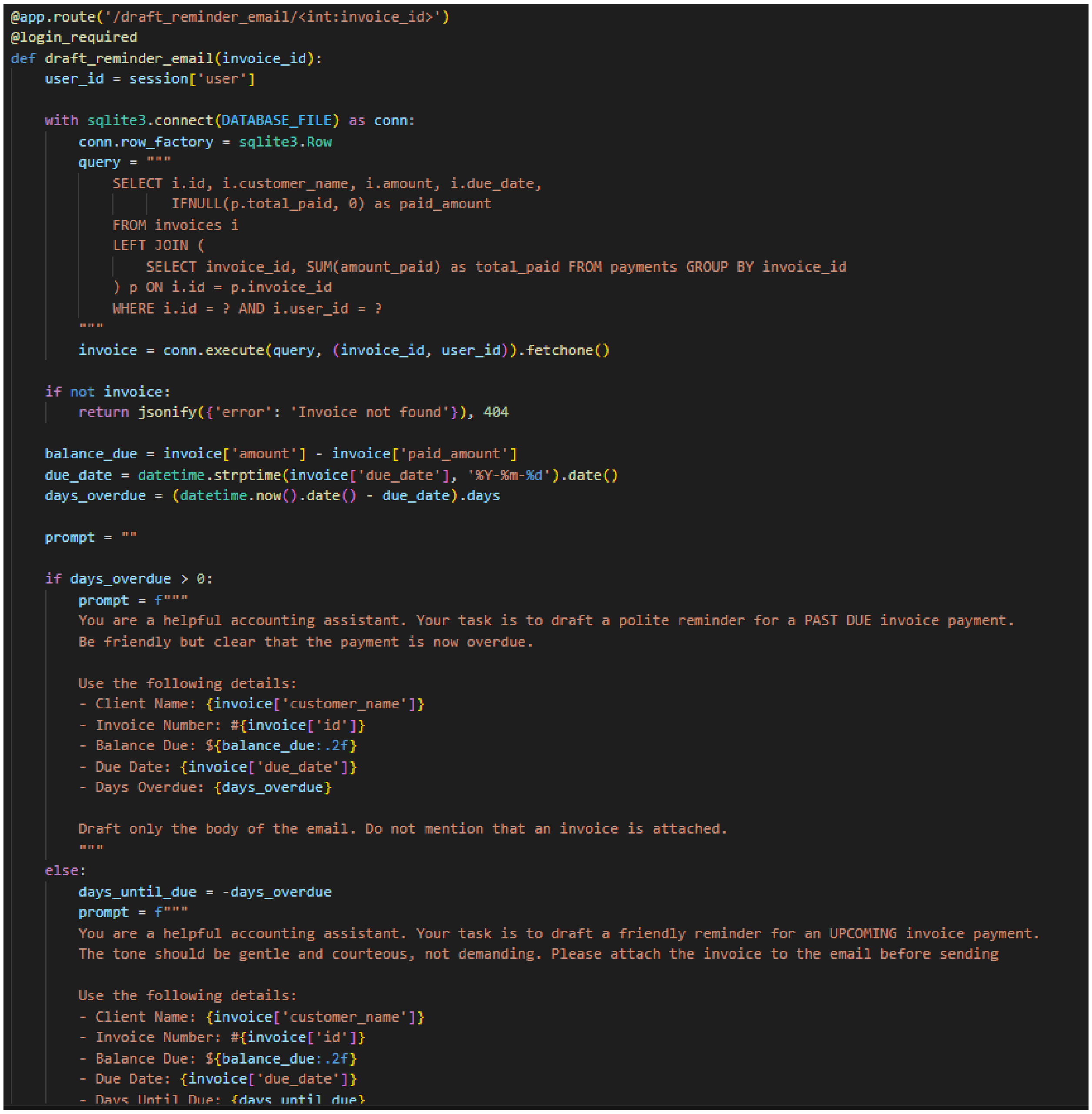
```
@app.route('/generate_receipt')
@login_required
def generate_receipt():
    user_id = session.get('user')

    # get unpaid invoices for this user
    with sqlite3.connect(DATABASE_FILE) as conn:
        cursor = conn.cursor()
    cursor.execute("""
    SELECT id, customer_name, amount, due_date
    FROM invoices
    WHERE status = 'Unpaid' AND user_id = ?
""", (user_id,))
    unpaid_invoices = cursor.fetchall()
    conn.close()

    # create pdf in memory
    buffer = io.BytesIO()
    c = canvas.Canvas(buffer, pagesize=A4)
    width, height = A4

    c.setFont("Helvetica-Bold", 16)
    c.drawString(50, height - 50, "Unpaid Invoices Summary")

    c.setFont("Helvetica", 10)
    c.drawString(50, height - 65, f"Generated on: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")

    c.setFont("Helvetica-Bold", 12)
    c.drawString(50, height - 100, "ID")
    c.drawString(100, height - 100, "Customer Name")
    c.drawString(280, height - 100, "Amount")
    c.drawString(400, height - 100, "Due Date")

    c.setFont("Helvetica", 12)
    y = height - 120
    total = 0

    for invoice in unpaid_invoices:
        invoice_id, name, amount, due_date, *_ = invoice

        c.drawString(50, y, str(invoice_id))
        c.drawString(100, y, name)
        c.drawString(280, y, f"${amount:.2f}")
        c.drawString(400, y, due_date or "N/A")

        total += amount
        y -= 20
```

Add and Manage clients



```python
@app.route('/add_client', methods=['POST'])
@login_required
def add_client():
    user_id = session['user']
    name = request.form['name']
    email = request.form['email']

    try: ...
    except ValueError: ...
...
            (name, email, user_id, hourly_rate)
        )
        conn.commit()

    return redirect(url_for('manage_clients'))

@app.route('/clients')
@login_required
def manage_clients():
    user_id = session['user']
    with sqlite3.connect('invoices.db') as conn:
        conn.row_factory = sqlite3.Row
        cursor = conn.cursor()
        cursor.execute("SELECT * FROM clients WHERE user_id = ?", (user_id,))
        clients = cursor.fetchall()
    return render_template('clients.html', clients=clients)
```

AI email functionality



```python
@app.route('/draft_reminder_email/<int:invoice_id>')
@login_required
def draft_reminder_email(invoice_id):
    user_id = session['user']

    with sqlite3.connect(DATABASE_FILE) as conn:
        conn.row_factory = sqlite3.Row
        query = """
            SELECT i.id, i.customer_name, i.amount, i.due_date,
                   IFNULL(p.total_paid, 0) as paid_amount
            FROM invoices i
            LEFT JOIN (
                SELECT invoice_id, SUM(amount_paid) as total_paid FROM payments GROUP BY invoice_id
            ) p ON i.id = p.invoice_id
            WHERE i.id = ? AND i.user_id = ?
        """
        invoice = conn.execute(query, (invoice_id, user_id)).fetchone()

    if not invoice:
        return jsonify({'error': 'Invoice not found'}), 404

    balance_due = invoice['amount'] - invoice['paid_amount']
    due_date = datetime.strptime(invoice['due_date'], '%Y-%m-%d').date()
    days_overdue = (datetime.now().date() - due_date).days

    prompt = ""

    if days_overdue > 0:
        prompt = f"""
        You are a helpful accounting assistant. Your task is to draft a polite reminder for a PAST DUE invoice payment.
        Be friendly but clear that the payment is now overdue.

        Use the following details:
        - Client Name: {invoice['customer_name']}
        - Invoice Number: #{invoice['id']}
        - Balance Due: ${balance_due:.2f}
        - Due Date: {invoice['due_date']}
        - Days Overdue: {days_overdue}

        Draft only the body of the email. Do not mention that an invoice is attached.
        """
    else:
        days_until_due = -days_overdue
        prompt = f"""
        You are a helpful accounting assistant. Your task is to draft a friendly reminder for an UPCOMING invoice payment.
        The tone should be gentle and courteous, not demanding. Please attach the invoice to the email before sending

        Use the following details:
        - Client Name: {invoice['customer_name']}
        - Invoice Number: #{invoice['id']}
        - Balance Due: ${balance_due:.2f}
        - Due Date: {invoice['due_date']}
        - Days Until Due: {days_until_due}
```

Pseudocode

Login:

```
PROCEDURE SessionLogin(login_token)
  BEGIN
1. Verify the user's identity.
     VALIDATE the provided login_token.
     IF token is invalid:
         RETURN "Login failed" error.
```

```
2. Fetch the user's profile.
     user = GET user profile from database.
     IF user is not found:
         RETURN "User not found" error.
3. Check if their email is verified.
     IF user.isVerified IS FALSE:
         SEND a new verification email.
         RETURN "Email verification required" message.
4. Create a session for the verified user.
     CREATE session for the user.
     RETURN success.
END
```

Forgot Password:

```
PROCEDURE ForgotPassword(email)
  BEGIN
1. Check if the user exists.
     IF a user with the provided email exists:
2. Send a reset code if they do.
         SEND password reset code to their email.
         REDIRECT to the page for entering the reset code.
     ELSE:
3. Show an error if they don't.
         SHOW "Email not found" error.
     END IF
END
```

Add Invoice:

```
PROCEDURE AddInvoice(client_id, hours_worked, due_date)
  BEGIN
1. Get required data.
     current_user = GET user from session.
     client = GET client details for the current_user.
     IF client is not found:
         RETURN "Client not found" error.

2. Calculate the total amount.
     total_amount = hours_worked * client.hourly_rate.

3. Create and save the new invoice.
     CREATE a new 'Unpaid' invoice record with the client info, amount, and dates.
     SAVE the invoice to the database.

4. Show the result.
```

```
      REDIRECT to the main invoices page.
END
```

Record Payment:

```
PROCEDURE RecordPayment(invoice_id, payment_amount)
  BEGIN
1. Record the incoming payment.
      SAVE the payment details to the database, linked to the invoice.

2. Update the invoice status.
      total_paid = CALCULATE total payments for the invoice.
      invoice_total = GET the invoice's original amount.

      IF total_paid >= invoice_total:
          UPDATE invoice status to 'Paid'.
      ELSE:
          UPDATE invoice status to 'Partially Paid'.
      END IF

3. Show the result.
      REDIRECT to the main invoices page.
END
```

Generate Invoice:

```
PROCEDURE GenerateInvoicePDF(invoice_data)
  BEGIN
1. Create a new PDF document.
      CREATE a blank PDF.

2. Add content to the PDF.
      ADD company and client details.
      ADD invoice header (ID, dates, title).
      ADD table of line items (description, hours, rate, amount).
      ADD final totals and footer notes.

3. Return the generated file.
      RETURN the completed PDF for download.
END
```

Generate Unpaid Invoices:

```
PROCEDURE GenerateUnpaidInvoicesReport(user_id)
  BEGIN
1. Fetch the data.
      unpaid_invoices = GET all 'Unpaid' and 'Partially Paid' invoices for the user.
```

```
2. Create the PDF document.
     CREATE a new landscape PDF.
     ADD the report title: "Summary of Unpaid Invoices".

3. Build the invoice table.
     PREPARE a table with columns: "ID", "Customer", "Due Date", "Total", "Paid", "Balance
Due".
     FOR EACH invoice in unpaid_invoices:
        CALCULATE its balance_due.
        ADD a row to the table with the invoice details.

4. Add the final summary.
     CALCULATE the grand_total_due from all balances.
     ADD the grand total to the bottom of the report.

5. Return the generated file.
     RETURN the completed PDF for download.
END
```

Index Page:

```
PROCEDURE ShowDashboard()
  BEGIN
1. Get the current logged-in user.
     current_user = GET user from session.

2. Fetch invoice statistics for that user.
     total_invoices = COUNT all invoices for the user.
     paid_invoices = COUNT 'Paid' invoices for the user.
     unpaid_invoices = COUNT 'Unpaid' invoices for the user.

3. Display the statistics on the dashboard.
     RENDER the main page with the fetched counts.
END
```

Manage Client:

```
FUNCTION manage_clients:
  // Check if the user is logged in
  IF user is not in session THEN
    REDIRECT to login page
  END IF

  // Get the user's ID from the current session
  user_id = GET user_id from session
  // Establish a connection to the database
  CONNECT to database 'invoices.db'

  // Prepare a query to select all clients for the current user
  QUERY = "SELECT all columns FROM clients WHERE user_id = ?"

  // Execute the query with the user's ID
  EXECUTE QUERY with user_id

  // Fetch all the results from the query
  clients = FETCH all results

  // Close the database connection
  CLOSE database connection

  // Display the 'clients.html' page, passing the list of clients to it
  RENDER template 'clients.html' with clients
END FUNCTION
```

Social and Ethical Issues

1. Data Privacy and Security
   As the application manages sensitive client information (names, emails) and financial records (invoices, payments, hourly rates), there is a significant ethical obligation to protect this data from unauthorized access, breaches, or misuse. Failure to do so could lead to financial loss, reputational damage, and a breach of trust for both the administrator and their clients.

   This was combatted through robust authentication, strict authorisation, and XSS attack prevention. The system uses Google's Firebase Authentication, an industry-standard service, for user login. This outsources the complexity of password management and security to a trusted, robust platform, rather than attempting to build a less secure custom solution. The “login_required” decorator is applied to all sensitive backend routes, ensuring that no page can be accessed without a valid, active session. Furthermore, all database queries are filtered by the logged-in “user_id”, creating a hard separation of data. This design makes it impossible for one administrator to accidentally or maliciously view the data of another. To protect data integrity, the application leverages the default security features of the Jinja2 templating engine. Jinja2 employs automatic HTML escaping on all data rendered with {{ ... }} syntax. This converts potentially malicious user-submitted code (e.g., <script> tags) into harmless plain text, effectively neutralizing the most common vector for Cross-Site Scripting (XSS) attacks.

2. Accuracy and Integrity of Financial Data
   The core function of the application is to act as a system of record for financial transactions. Bugs or inaccuracies in calculations could lead to incorrect billing, damaging the professional relationship between the administrator and their clients, and potentially causing financial harm.

   This is prevented through automated calculations, controlled status updates, and data validation. Invoice amounts are calculated automatically by the system (hours_worked * hourly_rate). This removes the risk of manual user error that could occur with manual data entry, ensuring consistency and accuracy in billing. The status of an invoice ('Unpaid', 'Partially Paid', 'Paid') is not a field the user can easily change. It is updated automatically by the system based on the payments recorded against the invoice. This ensures the status is always a true and verifiable reflection of the payment history. The backend code includes ‘try-except’ blocks to validate and handle user input, ensuring that data entered into forms (like hours or payment amounts) is of the correct type and format before being saved to the database.

3. Ethical use of Artificial Intelligence
   The integration of a generative AI to draft emails introduces new ethical considerations, primarily concerning accountability and transparency. If an AI were to generate an inaccurate, inappropriate, or manipulative email, it could damage client relationships.

   This was countered through the implementation of draft stages before sending the email to the client. The AI does not send emails automatically. Its role is strictly limited to generating a draft. The administrator is presented with this draft in an editable text area. They are required to review the content and must perform the final, explicit action of clicking "Send Email." This design ensures the human user is always in control and remains fully accountable for the final message sent to the client.

Client Evaluation

The application has not only met but has significantly exceeded my expectations, fundamentally transforming a once disjointed and error-prone invoicing process into a model of efficiency. Previously, my workflow was a chaotic mix of spreadsheets and manual reminders; today, I have a centralized, streamlined solution. From the moment I first logged in, the system's intuitive and clean user interface made managing clients and creating invoices remarkably simple. The ability to instantly view and filter all invoices by status has given me a powerful and immediate overview of my financial position, bringing a level of clarity I never had before. The software is consistently fast, reliable, and secure, with every feature working flawlessly.

The true value, however, lies in the advanced features that have had a direct and positive impact on my business operations. The professional PDF generation has elevated my company's image and saved me countless hours in manual report compilation for accounting. The AI-powered reminder assistant is, without a doubt, the most transformative feature, automating the delicate task of drafting follow-up emails and preserving positive client relationships. In conclusion, this software has ticked every requirement box and then some, replacing administrative friction with streamlined professionalism. I would not hesitate to recommend this exceptional product to any business owner seeking to solve their invoicing challenges.

- Tanish Patil (Client)

Self-Evaluation

Looking back, I think my progress with this project was evenly-spaced and well planned before attempting it. I think I most enjoyed the problem solving that was required when bugs arose in the code, but it would sometimes be a nuisance to find minute errors in the code.

1. Rating: 4
   The software is well coded and I am very pleased with the outcome, but I feel as though I could have done more with the CSS with a little more time and some more testing to see what is the most aesthetically pleasing.
2. My biggest issue is the use of 2 different databases (SQLite and Firebase), for different reasons. I only learnt about the existence of Firebase mid-way through my project, but I saw the potential with improving my user experience, therefore decided to integrate it. This caused a couple issues but was solvable, however it just annoyed me that I couldn't stick to one set database.
3. I see a future use for my software and am thinking of continuing to maintain and improve on it in the future as it can be readily used for a variety of small businesses. This future has given me a reason to be even more proud of the outcome as it is worthy of future use.
4. Overall, I'm happy with the outcome of the software but would still like to improve on it in the future. Overall, a 4/5.