Main.py :-   
  
import sys  
import os  
import pandas as pd  
from PyQt5.QtWidgets import (  
 QApplication, QMainWindow, QWidget, QVBoxLayout, QHBoxLayout, QPushButton,  
 QFileDialog, QMessageBox, QLabel, QTableWidget, QTableWidgetItem, QLineEdit, QComboBox,  
 QDialog, QListWidget, QListWidgetItem, QFormLayout, QDialogButtonBox, QScrollArea, QGraphicsView, QGraphicsScene, QGraphicsRectItem  
)  
from PyQt5.QtCore import Qt, QFileInfo, QStandardPaths  
from PyQt5.QtGui import QPixmap  
from fpdf import FPDF  
import pdfplumber  
import fitz  
from reportlab.lib.styles import getSampleStyleSheet  
from reportlab.platypus import SimpleDocTemplate, Table, TableStyle, Paragraph  
from reportlab.lib import colors  
from docx import Document  
import json  
import logging  
  
*# Set up logging*logging.basicConfig(level=logging.INFO, format="%(levelname)s: %(message)s")  
logger = logging.getLogger(\_\_name\_\_)  
  
  
class MainApp(QMainWindow):  
 def \_\_init\_\_(*self*):  
 *super*().\_\_init\_\_()  
 *self*.setWindowTitle("Advanced Data Search & Export Tool 2.2")  
 *self*.setGeometry(100, 100, 1200, 800)  
  
 *# Initialize variables  
 self*.df = None *# DataFrame to store uploaded data  
 self*.filtered\_df = None *# DataFrame to store filtered data  
 self*.pdf\_path = None *# Path to the uploaded PDF  
 self*.pdf\_document = None *# PDF document object  
 self*.base\_name = "Invoice" *# Default base name for exported files  
 self*.current\_zoom = 1.0 *# Zoom level for PDF preview  
 self*.text\_boxes = [] *# List to store text boxes  
 self*.box\_data = [] *# List to store box data  
 self*.sort\_orders = {} *# Dictionary to track column sorting order  
 self*.docx\_template\_path = None *# Path to the uploaded DOCX template  
 self*.image\_path = None *# Path to the uploaded image  
 self*.box\_column\_map = {} *# Dictionary to map boxes to columns  
  
 # Create the main widget and layout  
 self*.central\_widget = QWidget()  
 *self*.setCentralWidget(*self*.central\_widget)  
 *self*.layout = QVBoxLayout(*self*.central\_widget)  
  
 *# Top Bar Layout  
 self*.top\_bar\_layout = QHBoxLayout()  
 *self*.layout.addLayout(*self*.top\_bar\_layout)  
  
 *# Add buttons to the top bar  
 self*.load\_button = QPushButton("Load Data")  
 *self*.top\_bar\_layout.addWidget(*self*.load\_button)  
 *self*.load\_button.clicked.connect(*self*.load\_data)  
  
 *self*.generate\_all\_invoices\_button = QPushButton("Generate All Invoices")  
 *self*.top\_bar\_layout.addWidget(*self*.generate\_all\_invoices\_button)  
 *self*.generate\_all\_invoices\_button.clicked.connect(*self*.generate\_all\_invoices)  
  
 *self*.invoice\_button = QPushButton("Create Invoice")  
 *self*.top\_bar\_layout.addWidget(*self*.invoice\_button)  
 *self*.invoice\_button.clicked.connect(*self*.create\_invoice\_dialog)  
  
 *self*.load\_image\_button = QPushButton("Load Image")  
 *self*.top\_bar\_layout.addWidget(*self*.load\_image\_button)  
 *self*.load\_image\_button.clicked.connect(*self*.load\_image)  
  
 *self*.add\_box\_button = QPushButton("Add Box")  
 *self*.top\_bar\_layout.addWidget(*self*.add\_box\_button)  
 *self*.add\_box\_button.clicked.connect(*self*.add\_box)  
  
 *self*.save\_structure\_button = QPushButton("Save Structure")  
 *self*.top\_bar\_layout.addWidget(*self*.save\_structure\_button)  
 *self*.save\_structure\_button.clicked.connect(*self*.save\_structure)  
  
 *self*.load\_structure\_button = QPushButton("Load Structure")  
 *self*.top\_bar\_layout.addWidget(*self*.load\_structure\_button)  
 *self*.load\_structure\_button.clicked.connect(*self*.load\_structure)  
  
 *self*.export\_csv\_button = QPushButton("Export as CSV")  
 *self*.top\_bar\_layout.addWidget(*self*.export\_csv\_button)  
 *self*.export\_csv\_button.clicked.connect(lambda: *self*.export\_data("csv"))  
  
 *self*.export\_excel\_button = QPushButton("Export as Excel")  
 *self*.top\_bar\_layout.addWidget(*self*.export\_excel\_button)  
 *self*.export\_excel\_button.clicked.connect(lambda: *self*.export\_data("xlsx"))  
  
 *self*.export\_pdf\_button = QPushButton("Export as PDF")  
 *self*.top\_bar\_layout.addWidget(*self*.export\_pdf\_button)  
 *self*.export\_pdf\_button.clicked.connect(lambda: *self*.export\_data("pdf"))  
  
 *# Add a label for instructions  
 self*.label = QLabel("Load an Excel/CSV file to search, filter, and export data.")  
 *self*.layout.addWidget(*self*.label)  
  
 *# Add search and filter widgets  
 self*.search\_layout = QHBoxLayout()  
 *self*.search\_input = QLineEdit()  
 *self*.search\_input.setPlaceholderText("Search...")  
 *self*.search\_layout.addWidget(*self*.search\_input)  
  
 *self*.filter\_column = QComboBox()  
 *self*.filter\_column.addItem("All Columns")  
 *self*.search\_layout.addWidget(*self*.filter\_column)  
  
 *self*.filter\_type = QComboBox()  
 *self*.filter\_type.addItems(["Contains", "Equals", "Starts with"])  
 *self*.search\_layout.addWidget(*self*.filter\_type)  
  
 *self*.search\_button = QPushButton("Search")  
 *self*.search\_button.clicked.connect(*self*.perform\_search)  
 *self*.search\_layout.addWidget(*self*.search\_button)  
  
 *self*.layout.addLayout(*self*.search\_layout)  
  
 *# Add a table to display the loaded data  
 self*.table = QTableWidget()  
 *self*.layout.addWidget(*self*.table)  
  
 *# DOCX Upload button  
 self*.upload\_docx\_btn = QPushButton("📂 Upload DOCX Template")  
 *self*.upload\_docx\_btn.clicked.connect(*self*.upload\_template)  
 *self*.top\_bar\_layout.addWidget(*self*.upload\_docx\_btn)  
  
 *# Add a button to fill the DOCX template  
 self*.fill\_docx\_btn = QPushButton("📝 Fill DOCX Template")  
 *self*.fill\_docx\_btn.clicked.connect(*self*.fill\_docx\_template)  
 *self*.layout.addWidget(*self*.fill\_docx\_btn)  
  
 *# Initialize PDF preview  
 self*.graphics\_scene = QGraphicsScene()  
 *self*.graphics\_view = QGraphicsView(*self*.graphics\_scene)  
 *self*.layout.addWidget(*self*.graphics\_view)  
  
 def load\_data(*self*):  
 *"""Loads data from an Excel/CSV file."""* file\_path, \_ = QFileDialog.getOpenFileName(  
 *self*, "Open File", "", "Excel/CSV Files (\*.xlsx \*.xls \*.csv)"  
 )  
  
 if not file\_path:  
 QMessageBox.warning(*self*, "Error", "No file selected!")  
 return  
  
 try:  
 if file\_path.endswith(".csv"):  
 *self*.df = pd.read\_csv(file\_path)  
 else:  
 *self*.df = pd.read\_excel(file\_path)  
  
 if *self*.df.empty:  
 QMessageBox.warning(*self*, "Error", "The file is empty!")  
 return  
  
 *# Update the filter column dropdown  
 self*.filter\_column.clear()  
 *self*.filter\_column.addItem("All Columns")  
 *self*.filter\_column.addItems(*self*.df.columns.tolist())  
  
 *# Display the data in the table  
 self*.display\_data\_in\_table(*self*.df)  
  
 QMessageBox.information(*self*, "Success", f"Loaded {*len*(*self*.df)} records!")  
  
 except *Exception* as e:  
 QMessageBox.critical(*self*, "Error", f"Failed to load file: {e}")  
  
 def display\_data\_in\_table(*self*, *data*):  
 *"""Displays the loaded data in the table widget."""  
 self*.table.setRowCount(*data*.shape[0])  
 *self*.table.setColumnCount(*data*.shape[1])  
 *self*.table.setHorizontalHeaderLabels(*data*.columns)  
  
 for i in *range*(*data*.shape[0]):  
 for j in *range*(*data*.shape[1]):  
 *self*.table.setItem(i, j, QTableWidgetItem(*str*(*data*.iat[i, j])))  
  
 def perform\_search(*self*):  
 *"""Performs a search on the loaded data."""* if *self*.df is None:  
 QMessageBox.warning(*self*, "Error", "No data loaded!")  
 return  
  
 search\_query = *self*.search\_input.text().strip()  
 filter\_column = *self*.filter\_column.currentText()  
 filter\_type = *self*.filter\_type.currentText()  
  
 if not search\_query:  
 *self*.display\_data\_in\_table(*self*.df)  
 return  
  
 *# Handle comma-separated sub-queries* sub\_queries = [q.strip() for q in search\_query.split(',')]  
 filtered\_data = *self*.df.copy()  
  
 *# Apply queries in priority order* for q in sub\_queries:  
 if filter\_column == "All Columns":  
 filtered\_data = filtered\_data[  
 filtered\_data.apply(lambda *row*: row.astype(*str*).str.contains(q, case=False, na=False).any(), axis=1)  
 ]  
 else:  
 if filter\_type == "Contains":  
 filtered\_data = filtered\_data[filtered\_data[filter\_column].astype(*str*).str.contains(q, case=False, na=False)]  
 elif filter\_type == "Equals":  
 filtered\_data = filtered\_data[filtered\_data[filter\_column].astype(*str*) == q]  
 elif filter\_type == "Starts with":  
 filtered\_data = filtered\_data[filtered\_data[filter\_column].astype(*str*).str.startswith(q, na=False)]  
  
 *self*.filtered\_df = filtered\_data  
  
 if *self*.filtered\_df.empty:  
 QMessageBox.information(*self*, "No Results", "No matching records found.")  
 return  
  
 *self*.display\_data\_in\_table(*self*.filtered\_df)  
  
 def export\_data(*self*, *format*):  
 *"""Exports the filtered data to the specified format."""* if *self*.filtered\_df is None or *self*.filtered\_df.empty:  
 QMessageBox.warning(*self*, "Error", "No filtered data to export!")  
 return  
  
 file\_path, \_ = QFileDialog.getSaveFileName(  
 *self*, "Save File", "", f"{*format*.upper()} Files (\*.{*format*})"  
 )  
  
 if not file\_path:  
 return  
  
 try:  
 if *format* == "csv":  
 *self*.filtered\_df.to\_csv(file\_path, index=False)  
 elif *format* == "xlsx":  
 *self*.filtered\_df.to\_excel(file\_path, index=False)  
 elif *format* == "pdf":  
 *self*.save\_df\_as\_pdf(*self*.filtered\_df, file\_path)  
  
 QMessageBox.information(*self*, "Success", f"Data exported as {*format*.upper()} successfully!")  
  
 except *Exception* as e:  
 QMessageBox.critical(*self*, "Error", f"Failed to export data: {e}")  
  
 def save\_df\_as\_pdf(*self*, *df*, *save\_path*):  
 *"""Saves the DataFrame as a PDF."""* pdf = FPDF()  
 pdf.set\_auto\_page\_break(auto=True, margin=15)  
 pdf.add\_page()  
  
 *# Add title* pdf.set\_font("Arial", "B", 16)  
 pdf.cell(200, 10, "Exported Data", ln=True, align='C')  
  
 *# Add table headers* pdf.set\_font("Arial", "B", 12)  
 for col in *df*.columns:  
 pdf.cell(40, 10, col, 1)  
 pdf.ln()  
  
 *# Add table data* pdf.set\_font("Arial", size=12)  
 for \_, row in *df*.iterrows():  
 for col in *df*.columns:  
 pdf.cell(40, 10, *str*(row[col]), 1)  
 pdf.ln()  
  
 *# Save the PDF* pdf.output(*save\_path*)  
 *print*(f"✅ PDF saved: {*save\_path*}")  
  
 def create\_invoice\_dialog(*self*):  
 *"""Opens a dialog to create an invoice."""* if *self*.df is None:  
 QMessageBox.warning(*self*, "Error", "No data loaded!")  
 return  
  
 dialog = QDialog(*self*)  
 dialog.setWindowTitle("Create Invoice")  
 layout = QVBoxLayout(dialog)  
  
 *# Select Columns* columns\_label = QLabel("Select Columns:")  
 layout.addWidget(columns\_label)  
 columns\_list = QListWidget()  
 columns\_list.setSelectionMode(QListWidget.MultiSelection)  
 for col in *self*.df.columns:  
 item = QListWidgetItem(col)  
 columns\_list.addItem(item)  
 layout.addWidget(columns\_list)  
  
 *# Invoice Details* details\_label = QLabel("Invoice Details:")  
 layout.addWidget(details\_label)  
 form\_layout = QFormLayout()  
 invoice\_number = QLineEdit()  
 form\_layout.addRow("Invoice Number:", invoice\_number)  
 customer\_name = QLineEdit()  
 form\_layout.addRow("Customer Name:", customer\_name)  
 invoice\_date = QLineEdit()  
 form\_layout.addRow("Invoice Date:", invoice\_date)  
 layout.addLayout(form\_layout)  
  
 *# Buttons* button\_box = QDialogButtonBox(QDialogButtonBox.Ok | QDialogButtonBox.Cancel)  
 button\_box.accepted.connect(dialog.accept)  
 button\_box.rejected.connect(dialog.reject)  
 layout.addWidget(button\_box)  
  
 dialog.exec\_()  
  
 def generate\_all\_invoices(*self*):  
 *"""Generates a separate invoice for each row in the DataFrame."""* if *self*.df is None:  
 QMessageBox.warning(*self*, "Error", "No data loaded!")  
 return  
  
 output\_folder = QFileDialog.getExistingDirectory(*self*, "Select Output Folder")  
 if not output\_folder:  
 return  
  
 for index, row in *self*.df.iterrows():  
 invoice\_number = *str*(row.get("Invoice Number", f"INV-{index + 1}")) *# Get invoice number or create one* customer\_name = *str*(row.get("Customer Name", f"Customer {index + 1}")) *# Get customer name or create one* invoice\_date = *str*(row.get("Invoice Date", "N/A")) *# Get invoice date or set to N/A  
  
 # Create a temporary DataFrame with only the current row* temp\_df = pd.DataFrame([row])  
  
 *# Generate the invoice* pdf = FPDF()  
 pdf.add\_page()  
 pdf.set\_font("Arial", size=12)  
  
 *# Invoice Header* pdf.set\_font("Arial", "B", 16)  
 pdf.cell(0, 10, "Invoice", ln=True, align='C')  
 pdf.set\_font("Arial", size=12)  
  
 *# Customer Information* address\_columns = ["Address", "City", "Zip"] *# Example address columns* if address\_columns:  
 pdf.cell(0, 10, "Customer Information:", ln=True)  
 address\_text = ""  
 for col in address\_columns:  
 if col in temp\_df.columns:  
 address\_text += f"{col}: {temp\_df.iloc[0][col]}\n"  
 pdf.multi\_cell(0, 10, address\_text)  
  
 *# Invoice Details* pdf.cell(0, 10, f"Invoice Number: {invoice\_number}", ln=True)  
 pdf.cell(0, 10, f"Customer Name: {customer\_name}", ln=True)  
 pdf.cell(0, 10, f"Invoice Date: {invoice\_date}", ln=True)  
 pdf.ln(10)  
  
 *# Table Headers* pdf.set\_font("Arial", "B", 12)  
 for col in *self*.df.columns:  
 pdf.cell(40, 10, col, 1)  
 pdf.ln()  
  
 *# Table Data* pdf.set\_font("Arial", size=12)  
 for col in *self*.df.columns:  
 pdf.cell(40, 10, *str*(row[col]), 1)  
 pdf.ln()  
  
 *# Other Sections* other\_sections = [{"title": "Notes", "column": "Notes"}] *# Example other sections* if other\_sections:  
 pdf.ln(10)  
 for section in other\_sections:  
 if section["column"] in temp\_df.columns:  
 pdf.cell(0, 10, f"{section['title']}: {temp\_df.iloc[0][section['column']]}", ln=True)  
  
 *# Save the PDF* file\_path = os.path.join(output\_folder, f"Invoice\_{index + 1}.pdf")  
 pdf.output(file\_path)  
  
 QMessageBox.information(*self*, "Success", f"{*len*(*self*.df)} invoices generated successfully!")  
  
 def load\_image(*self*):  
 *"""Loads an image for the PDF preview."""* file\_path, \_ = QFileDialog.getOpenFileName(*self*, "Open Image", "", "Image Files (\*.png \*.jpg \*.jpeg)")  
 if file\_path:  
 *self*.image\_path = file\_path  
 *self*.create\_pdf\_preview()  
  
 def create\_pdf\_preview(*self*):  
 *"""Creates a PDF preview with the loaded image."""* if *self*.image\_path:  
 pixmap = QPixmap(*self*.image\_path)  
 *self*.pdf\_label = QLabel()  
 *self*.pdf\_label.setPixmap(pixmap)  
 *self*.scroll\_area = QScrollArea()  
 *self*.scroll\_area.setWidget(*self*.pdf\_label)  
 *self*.graphics\_scene = QGraphicsScene()  
 *self*.graphics\_view = QGraphicsView(*self*.graphics\_scene)  
 *self*.graphics\_view.setGeometry(*self*.scroll\_area.geometry())  
 *self*.graphics\_view.setStyleSheet("background: transparent;")  
 *self*.graphics\_view.setAttribute(Qt.WA\_TranslucentBackground)  
 *self*.layout.addWidget(*self*.scroll\_area)  
 *self*.layout.addWidget(*self*.graphics\_view)  
 for rect, dropdown in *self*.box\_column\_map.items():  
 *self*.graphics\_scene.addItem(rect)  
 *self*.layout.addWidget(dropdown)  
  
 def add\_box(*self*):  
 *"""Adds a resizable box and column selection dropdown."""* rect = QGraphicsRectItem(100, 100, 200, 50)  
 *self*.graphics\_scene.addItem(rect)  
 column\_dropdown = QComboBox()  
 column\_dropdown.addItems(*self*.df.columns.tolist())  
 *self*.layout.addWidget(column\_dropdown)  
 *self*.box\_column\_map[rect] = column\_dropdown  
  
 def save\_structure(*self*):  
 *"""Saves the structure of boxes and columns to a JSON file."""* structure = []  
 for rect, dropdown in *self*.box\_column\_map.items():  
 structure.append({  
 "x": rect.rect().x(),  
 "y": rect.rect().y(),  
 "width": rect.rect().width(),  
 "height": rect.rect().height(),  
 "column": dropdown.currentText()  
 })  
 file\_path, \_ = QFileDialog.getSaveFileName(*self*, "Save Structure", "", "JSON Files (\*.json)")  
 if file\_path:  
 with *open*(file\_path, 'w') as f:  
 json.dump(structure, f)  
  
 def load\_structure(*self*):  
 *"""Loads the structure of boxes and columns from a JSON file."""* file\_path, \_ = QFileDialog.getOpenFileName(*self*, "Load Structure", "", "JSON Files (\*.json)")  
 if file\_path:  
 with *open*(file\_path, 'r') as f:  
 structure = json.load(f)  
 *self*.box\_column\_map = {}  
 *self*.graphics\_scene.clear()  
 for item in structure:  
 rect = QGraphicsRectItem(item['x'], item['y'], item['width'], item['height'])  
 *self*.graphics\_scene.addItem(rect)  
 column\_dropdown = QComboBox()  
 column\_dropdown.addItems(*self*.df.columns.tolist())  
 column\_dropdown.setCurrentText(item['column'])  
 *self*.layout.addWidget(column\_dropdown)  
 *self*.box\_column\_map[rect] = column\_dropdown  
 *self*.create\_pdf\_preview()  
  
 def generate\_pdf\_with\_boxes(*self*, *output\_path*):  
 *"""Generates a PDF with boxes and text."""* doc = fitz.open() *# Create empty PDF* page = doc.new\_page()  
 if *self*.image\_path:  
 rect = page.rect  
 page.insert\_image(rect, filename=*self*.image\_path)  
 for rect, column\_dropdown in *self*.box\_column\_map.items():  
 column\_name = column\_dropdown.currentText()  
 text = *str*(*self*.df.iloc[0][column\_name])  
 x = rect.rect().x()  
 y = rect.rect().y()  
 page.insert\_text((x, y), text)  
 doc.save(*output\_path*)  
  
 def upload\_template(*parent*=None):  
 global template\_file  
  
 *# Set default paths* docs\_path = QStandardPaths.writableLocation(QStandardPaths.DocumentsLocation)  
  
 try:  
 *# Get file path with native dialog* file\_path, \_ = QFileDialog.getOpenFileName(  
 *parent*,  
 "Select DOCX Template",  
 docs\_path,  
 "Word Documents (\*.docx);;All Files (\*)"  
 )  
  
 if not file\_path:  
 logging.debug("User cancelled file selection")  
 return False  
  
 *# Validate file* file\_info = QFileInfo(file\_path)  
  
 *# 1. Check file exists* if not file\_info.exists():  
 QMessageBox.critical(  
 *parent*,  
 "File Not Found",  
 "The selected file does not exist."  
 )  
 return False  
  
 *# 2. Check file size (max 20MB)* max\_size\_mb = 20  
 file\_size\_mb = file\_info.size() / (1024 \* 1024)  
 if file\_size\_mb > max\_size\_mb:  
 QMessageBox.critical(  
 *parent*,  
 "File Too Large",  
 f"File exceeds maximum size of {max\_size\_mb}MB"  
 )  
 return False  
  
 *# 3. Check read permissions* if not file\_info.isReadable():  
 QMessageBox.critical(  
 *parent*,  
 "Permission Denied",  
 "You don't have permission to read this file."  
 )  
 return False  
  
 *# 4. Validate DOCX structure* try:  
 doc = Document(file\_path)  
 if not doc.paragraphs and not doc.tables:  
 QMessageBox.warning(  
 *parent*,  
 "Empty Document",  
 "The document appears to be empty or corrupted."  
 )  
 return False  
  
 *# Test saving a dummy version (checks for write permissions)* temp\_path = os.path.join(QStandardPaths.writableLocation(  
 QStandardPaths.TempLocation), "temp\_validation.docx")  
 doc.save(temp\_path)  
 os.remove(temp\_path)  
  
 except PackageNotFoundError:  
 QMessageBox.critical(  
 *parent*,  
 "Invalid DOCX",  
 "This is not a valid Word document (invalid package structure)."  
 )  
 return False  
 except *Exception* as e:  
 QMessageBox.critical(  
 *parent*,  
 "Document Error",  
 f"Failed to process document: {*str*(e)}"  
 )  
 return False  
  
 *# If we get here, file is valid* template\_file = file\_path  
 lbl\_template.setText(f"📄 {file\_info.fileName()} (Loaded)")  
 lbl\_template.setToolTip(file\_path) *# Show full path on hover* logging.info(f"Successfully loaded template: {file\_path}")  
 return True  
  
 except *Exception* as e:  
 logging.error(f"Unexpected error loading template: {*str*(e)}", exc\_info=True)  
 QMessageBox.critical(  
 *parent*,  
 "Unexpected Error",  
 f"An unexpected error occurred:\n{*str*(e)}"  
 )  
 return False  
  
 def fill\_docx\_template(*self*):  
 *"""Fills the DOCX template with data from the DataFrame."""* if not *self*.template\_file:  
 QMessageBox.critical(None, "Error", "No DOCX template uploaded!")  
 return  
  
 if *self*.df is None or *self*.df.empty:  
 QMessageBox.critical(None, "Error", "No data file uploaded!")  
 return  
  
 output\_folder = QFileDialog.getExistingDirectory(None, "Select Output Folder")  
 if not output\_folder:  
 return  
  
 try:  
 *# Validate placeholders in the template* if not *self*.validate\_docx\_template():  
 QMessageBox.critical(None, "Error", "Template validation failed. Please check the placeholders.")  
 return  
  
 *# Process each row in the DataFrame* for idx, row in *self*.df.iterrows():  
 doc = Document(*self*.template\_file)  
  
 *# Replace placeholders with data* for para in doc.paragraphs:  
 for col in *self*.df.columns:  
 placeholder = f"{{{{{col.strip()}}}}}"  
 if placeholder in para.text:  
 *self*.replace\_placeholder\_in\_runs(para, placeholder, *str*(row[col]))  
  
 *# Save the filled DOCX file* output\_path = os.path.join(output\_folder, f"Invoice\_{idx + 1}.docx")  
 doc.save(output\_path)  
 logging.info(f"✅ Saved: {output\_path}")  
  
 QMessageBox.information(None, "Success", f"Filled DOCX files saved in {output\_folder}")  
  
 except *Exception* as e:  
 logging.error(f"❌ Error filling DOCX template: {e}")  
 QMessageBox.critical(None, "Error", f"Failed to fill DOCX template: {e}")  
  
 def validate\_docx\_template(*self*):  
 *"""Validates that the DOCX template contains all required placeholders."""* try:  
 doc = Document(*self*.docx\_template\_path)  
 placeholders\_found = *set*()  
  
 for para in doc.paragraphs:  
 for col in *self*.df.columns:  
 placeholder = f"{{{{{col.strip()}}}}}"  
 if placeholder in para.text:  
 placeholders\_found.add(col)  
  
 *# Check if all placeholders are found* missing\_placeholders = *set*(*self*.df.columns) - placeholders\_found  
 if missing\_placeholders:  
 logger.error(f"❌ Missing placeholders in template: {missing\_placeholders}")  
 return False  
  
 return True  
  
 except *Exception* as e:  
 logger.error(f"❌ Error validating template: {e}")  
 return False  
  
 def replace\_placeholder\_in\_runs(*self*, *paragraph*, *placeholder*, *replacement*):  
 *"""Replaces placeholders in a paragraph while preserving formatting."""* for run in *paragraph*.runs:  
 if *placeholder* in run.text:  
 run.text = run.text.replace(*placeholder*, *replacement*)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 app = QApplication(sys.argv)  
 window = MainApp()  
 window.show()  
 sys.exit(app.exec\_())

Utils/\_\_init\_\_.py :-  
  
from .data\_utils import filter\_data, display\_data  
from .file\_utils import upload\_file, export\_filtered\_data, save\_df\_as\_pdf  
from .pdf\_utils import load\_pdf, add\_text\_to\_pdf *# Added add\_text\_to\_pdf*from .pdf\_generator import generate\_pdfs  
from .data\_mapper import map\_data\_to\_docx  
from .docx\_filler import fill\_docx\_template  
from .invoice\_generator import InvoiceGenerator  
from .theme\_manager import ThemeManager  
from .gui\_utils import create\_table\_widget, display\_data as display\_table\_data  
from .invoice\_utils import generate\_pdf\_invoice

Utils/ data\_mapper.py :-

import os  
import re  
import logging  
from docx import Document  
import pandas as pd  
from typing import List, Optional, Set, Dict  
from datetime import datetime  
from num2words import num2words  
from docx.shared import Pt  
from PyQt5.QtWidgets import QMessageBox  
  
logging.basicConfig(level=logging.INFO, format="%(levelname)s: %(message)s")  
  
class DataMapper:  
 def \_\_init\_\_(*self*, *parent*=None):  
 *self*.parent = *parent  
 self*.column\_mapping = {  
 'invoicenumber': 'INVOICE\_NUMBER',  
 'invoicedate': 'INVOICE\_DATE',  
 'isddistributorgstin': 'ISD\_DISTRIBUTOR\_GSTIN',  
 'isddistributorname': 'ISD\_DISTRIBUTOR\_NAME',  
 'isddistributoraddress': 'ISD\_DISTRIBUTOR\_ADDRESS',  
 'isddistributorstate': 'ISD\_DISTRIBUTOR\_STATE',  
 'isddistributorpincode': 'ISD\_DISTRIBUTOR\_PINCODE',  
 'isddistributorstatecode': 'ISD\_DISTRIBUTOR\_STATE\_CODE',  
 'creditrecipientgstin': 'CREDIT\_RECIPIENT\_GSTIN',  
 'creditrecipientname': 'CREDIT\_RECIPIENT\_NAME',  
 'creditrecipientaddress': 'CREDIT\_RECIPIENT\_ADDRESS',  
 'creditrecipientstate': 'CREDIT\_RECIPIENT\_STATE',  
 'creditrecipientpincode': 'CREDIT\_RECIPIENT\_PINCODE',  
 'creditrecipientstatecode': 'CREDIT\_RECIPIENT\_STATE\_CODE',  
 'cgst': 'CGST',  
 'sgst': 'SGST',  
 'utgst': 'UTGST',  
 'igst': 'IGST',  
 'amount': 'AMOUNT',  
 'regoffice': 'REG\_OFFICE',  
 'cin': 'CIN',  
 'email': 'E\_MAIL',  
 'e-mail': 'E\_MAIL',  
 'website': 'WEBSITE',  
 'amount\_in\_words': 'AMOUNT'  
 }  
  
 def normalize\_column\_names(*self*, *df*: pd.DataFrame) -> pd.DataFrame:  
 *"""Normalize column names to ensure consistent matching"""  
 df*.columns = [  
 col.strip().upper().replace(' ', '\_').replace('-', '\_')  
 for col in *df*.columns  
 ]  
 return *df* def map\_data\_to\_docx(*self*, *template\_path*: *str*, *data*: pd.DataFrame, *output\_folder*: *str*) -> Optional[List[*str*]]:  
 *"""Main function to generate DOCX files from template and data"""* try:  
 if not *self*.validate\_inputs(*template\_path*, *data*, *output\_folder*):  
 return None  
  
 os.makedirs(*output\_folder*, exist\_ok=True)  
 generated\_files = []  
 template\_placeholders = *self*.scan\_template\_placeholders(*template\_path*)  
  
 logging.info(f"Template placeholders: {template\_placeholders}")  
 logging.info(f"Data columns: {*data*.columns.tolist()}")  
  
 for idx, row in *data*.iterrows():  
 try:  
 doc = Document(*template\_path*)  
 row\_data = *self*.prepare\_row\_data(row, template\_placeholders)  
  
 *# Debug output for first row* if idx == 0:  
 *self*.log\_debug\_info(row, template\_placeholders, row\_data)  
  
 if not *self*.replace\_all\_placeholders(doc, row\_data):  
 logging.error(f"Skipping row {idx} due to replacement errors")  
 continue  
  
 output\_path = *self*.generate\_output\_path(*output\_folder*, row\_data, idx)  
 doc.save(output\_path)  
 generated\_files.append(output\_path)  
 logging.info(f"Generated: {output\_path}")  
  
 except *Exception* as e:  
 logging.error(f"Error processing row {idx}: {*str*(e)}", exc\_info=True)  
 continue  
  
 return generated\_files if generated\_files else None  
  
 except *Exception* as e:  
 logging.error(f"Fatal error: {*str*(e)}", exc\_info=True)  
 QMessageBox.critical(*self*.parent, "Error", f"Failed to generate documents: {*str*(e)}")  
 return None  
  
 def validate\_inputs(*self*, *template\_path*: *str*, *data*: pd.DataFrame, *output\_folder*: *str*) -> *bool*:  
 *"""Validate all input parameters"""* if not os.path.exists(*template\_path*):  
 QMessageBox.critical(*self*.parent, "Error", f"Template file not found: {*template\_path*}")  
 return False  
  
 if *data*.empty:  
 QMessageBox.critical(*self*.parent, "Error", "No data provided in DataFrame")  
 return False  
  
 try:  
 os.makedirs(*output\_folder*, exist\_ok=True)  
 return True  
 except *Exception* as e:  
 QMessageBox.critical(*self*.parent, "Error", f"Output folder not writable: {*str*(e)}")  
 return False  
  
 def scan\_template\_placeholders(*self*, *template\_path*: *str*) -> Set[*str*]:  
 *"""Extract all unique placeholders from a DOCX template"""* doc = Document(*template\_path*)  
 placeholders = *set*()  
 placeholder\_pattern = re.compile(r"\{\{\s\*(.\*?)\s\*\}\}") *# Handles whitespace* def extract\_from\_text(*text*: *str*):  
 return {match.strip() for match in placeholder\_pattern.findall(*text*)}  
  
 *# Process all document components* components = [  
 doc.paragraphs,  
 \*[cell.paragraphs for table in doc.tables  
 for row in table.rows  
 for cell in row.cells],  
 \*[section.header.paragraphs for section in doc.sections],  
 \*[section.footer.paragraphs for section in doc.sections]  
 ]  
  
 for paragraphs in components:  
 for paragraph in paragraphs:  
 placeholders.update(extract\_from\_text(paragraph.text))  
 for run in paragraph.runs:  
 placeholders.update(extract\_from\_text(run.text))  
  
 return {ph for ph in placeholders if ph} *# Remove empty strings* def prepare\_row\_data(*self*, *row*: pd.Series, *template\_placeholders*: Set[*str*]) -> Dict[*str*, *str*]:  
 *"""Prepare complete row data with all required fields and proper formatting"""* row\_data = {}  
  
 *# Process all placeholders in template* for ph in *template\_placeholders*:  
 *# Normalize the placeholder name* norm\_ph = ph.lower().replace(' ', '').replace('.', '').replace('-', '')  
  
 *# Special handling for amount\_in\_words* if norm\_ph == 'amount\_in\_words':  
 try:  
 amount = *float*(*row*['AMOUNT'])  
 words = num2words(amount, lang='en\_IN').title()  
 *# Ensure proper formatting* words = words.replace('And', 'and') *# Fix capitalization* row\_data['amount\_in\_words'] = f"{words} Rupees Only"  
 except *Exception* as e:  
 logging.error(f"Amount to words failed: {*str*(e)}")  
 row\_data['amount\_in\_words'] = ""  
 continue  
  
 *# Find matching column using our mapping* data\_key = *self*.column\_mapping.get(norm\_ph)  
  
 if data\_key and data\_key in *row*:  
 value = *row*[data\_key]  
 *# Convert numpy types to native Python* if *hasattr*(value, 'item'):  
 value = value.item()  
 row\_data[ph] = *self*.format\_value(value, ph)  
 else:  
 row\_data[ph] = ""  
 logging.warning(f"No data mapping for placeholder: {ph} (normalized: {norm\_ph})")  
  
 return row\_data  
  
 def replace\_all\_placeholders(*self*, *doc*: Document, *row\_data*: Dict[*str*, *str*]) -> *bool*:  
 *"""Replace placeholders throughout document with formatting preservation"""* try:  
 *# Process all paragraphs in main document* for paragraph in *doc*.paragraphs:  
 *self*.replace\_in\_paragraph(paragraph, *row\_data*)  
  
 *# Process all tables* for table in *doc*.tables:  
 for row in table.rows:  
 for cell in row.cells:  
 for paragraph in cell.paragraphs:  
 *self*.replace\_in\_paragraph(paragraph, *row\_data*)  
  
 *# Process headers and footers* for section in *doc*.sections:  
 for header in [section.header, section.first\_page\_header]:  
 if header:  
 for paragraph in header.paragraphs:  
 *self*.replace\_in\_paragraph(paragraph, *row\_data*)  
  
 for footer in [section.footer, section.first\_page\_footer]:  
 if footer:  
 for paragraph in footer.paragraphs:  
 *self*.replace\_in\_paragraph(paragraph, *row\_data*)  
  
 return True  
  
 except *Exception* as e:  
 logging.error(f"Error replacing placeholders: {*str*(e)}", exc\_info=True)  
 return False  
  
 def replace\_in\_paragraph(*self*, *paragraph*, *row\_data*: Dict[*str*, *str*]):  
 *"""Replace placeholders in a paragraph while preserving formatting"""  
 # First combine all runs to handle split placeholders* full\_text = ''.join(run.text for run in *paragraph*.runs)  
  
 *# Skip if no placeholders* if not *any*(f'{{{{{ph}}}}}' in full\_text for ph in *row\_data*):  
 return  
  
 *# Perform all replacements* modified\_text = full\_text  
 for ph, value in *row\_data*.items():  
 modified\_text = modified\_text.replace(f'{{{{{ph}}}}}', value)  
  
 *# Only update if changes were made* if modified\_text != full\_text:  
 *# Clear existing content  
 paragraph*.clear()  
  
 *# Add new content with preserved formatting* run = *paragraph*.add\_run(modified\_text)  
 run.font.size = Pt(10)  
  
 *# Preserve other formatting from first run if available* if *paragraph*.runs and *paragraph*.runs[0].font.name:  
 run.font.name = *paragraph*.runs[0].font.name  
  
 def format\_value(*self*, *value*, *key*=None) -> *str*:  
 *"""Format values with special handling for certain fields"""* if pd.isna(*value*):  
 return ""  
  
 *# Handle numpy types* if *hasattr*(*value*, 'item'):  
 value = *value*.item()  
  
 *# Special formatting for amounts* if *key* and 'amount' in *key*.lower() and *isinstance*(*value*, (*int*, *float*)):  
 return "{:,.2f}".format(*value*)  
  
 *# Special handling for GSTIN (format with spaces)* if *key* and 'gstin' in *key*.lower() and *isinstance*(*value*, *str*) and *len*(*value*) == 15:  
 return f"{*value*[:2]} {*value*[2:5]} {*value*[5:7]} {*value*[7:12]} {*value*[12:15]}"  
  
 return *str*(*value*).strip()  
  
 def generate\_output\_path(*self*, *output\_folder*: *str*, *row\_data*: *dict*, *idx*: *int*) -> *str*:  
 *"""Generate output path with invoice number if available"""* invoice\_num = *str*(*row\_data*.get('Invoice Number', *idx* + 1)).strip()  
 timestamp = datetime.now().strftime("%Y%m%d\_%H%M%S")  
 return os.path.join(*output\_folder*, f"ISD\_Invoice\_{invoice\_num}\_{timestamp}.docx")  
  
 def log\_debug\_info(*self*, *row*, *template\_placeholders*, *row\_data*):  
 *"""Log debug information for the first row"""* logging.info("\n=== DEBUG INFORMATION ===")  
 logging.info(f"Template placeholders: {*template\_placeholders*}")  
 logging.info(f"Data columns: {*row*.index.tolist()}")  
 logging.info(f"First row data: {*dict*(*row*)}")  
  
 logging.info("\n=== PLACEHOLDER MAPPING ===")  
 for ph in *template\_placeholders*:  
 norm\_ph = ph.lower().replace(' ', '').replace('.', '').replace('-', '')  
 data\_key = *self*.column\_mapping.get(norm\_ph, "NO MATCH")  
 logging.info(f"Template: {ph:25} → Data: {data\_key}")  
  
 logging.info("\n=== MATCHED DATA ===")  
 for ph, value in *row\_data*.items():  
 logging.info(f"{ph:25}: {value}")  
 logging.info("=====================")

Utils/docx\_filler.py :-  
  
import os  
import re  
import logging  
from docx import Document  
import pandas as pd  
from docx.shared import Pt  
from typing import Dict, List, Optional, Set  
from num2words import num2words  
from PyQt5.QtWidgets import QMessageBox  
from datetime import datetime  
  
logging.basicConfig(level=logging.INFO, format="%(levelname)s: %(message)s")  
  
class DocxFiller:  
 def \_\_init\_\_(*self*, *parent*=None):  
 *self*.parent = *parent  
 self*.default\_font\_size = 10 *# Default font size in points* def fill\_template(*self*, *template\_path*: *str*, *data*: pd.DataFrame, *output\_folder*: *str*) -> Optional[List[*str*]]:  
 *"""Main function to fill DOCX templates with data"""* try:  
 if not *self*.validate\_inputs(*template\_path*, *data*, *output\_folder*):  
 return None  
  
 os.makedirs(*output\_folder*, exist\_ok=True)  
 generated\_files = []  
 template\_placeholders = *self*.extract\_placeholders(*template\_path*)  
  
 *# Check for missing placeholders in data* missing\_placeholders = *self*.check\_missing\_placeholders(template\_placeholders, *data*)  
 if missing\_placeholders:  
 QMessageBox.warning(  
 *self*.parent,  
 "Warning",  
 f"Template placeholders not found in data:\n{', '.join(missing\_placeholders)}"  
 )  
  
 for idx, row in *data*.iterrows():  
 try:  
 doc = Document(*template\_path*)  
 row\_data = *self*.prepare\_row\_data(row, template\_placeholders)  
  
 if not *self*.replace\_placeholders\_in\_document(doc, row\_data):  
 continue  
  
 output\_path = *self*.generate\_output\_path(*output\_folder*, row\_data, idx)  
 doc.save(output\_path)  
 generated\_files.append(output\_path)  
 logging.info(f"Generated: {output\_path}")  
  
 except *Exception* as e:  
 logging.error(f"Error processing row {idx + 1}: {*str*(e)}", exc\_info=True)  
 continue  
  
 return generated\_files if generated\_files else None  
  
 except *Exception* as e:  
 logging.error(f"Fatal error: {*str*(e)}", exc\_info=True)  
 QMessageBox.critical(*self*.parent, "Error", f"Failed to fill templates: {*str*(e)}")  
 return None  
  
 def validate\_inputs(*self*, *template\_path*: *str*, *data*: pd.DataFrame, *output\_folder*: *str*) -> *bool*:  
 *"""Validate all input parameters"""* if not os.path.exists(*template\_path*):  
 QMessageBox.critical(*self*.parent, "Error", f"Template file not found: {*template\_path*}")  
 return False  
  
 if *data*.empty:  
 QMessageBox.critical(*self*.parent, "Error", "No data provided in DataFrame")  
 return False  
  
 try:  
 os.makedirs(*output\_folder*, exist\_ok=True)  
 return True  
 except *Exception* as e:  
 QMessageBox.critical(*self*.parent, "Error", f"Output folder not writable: {*str*(e)}")  
 return False  
  
 def check\_missing\_placeholders(*self*, *template\_placeholders*: Set[*str*], *data*: pd.DataFrame) -> Set[*str*]:  
 *"""Check which template placeholders are missing from the data columns"""* data\_columns = {col.lower() for col in *data*.columns}  
 return {  
 ph for ph in *template\_placeholders* if ph.lower() not in data\_columns  
 }  
  
 def prepare\_row\_data(*self*, *row*: pd.Series, *template\_placeholders*: Set[*str*]) -> Dict[*str*, *str*]:  
 *"""Prepare complete row data with all required fields and proper formatting"""* row\_data = {}  
  
 *# First handle special fields* if 'amount\_in\_words' in *template\_placeholders* and 'AMOUNT' in *row*:  
 try:  
 amount = *float*(*row*['AMOUNT'])  
 words = num2words(amount, lang='en\_IN').title()  
 words = words.replace(' And ', ' and ') *# Fix capitalization* row\_data['amount\_in\_words'] = f"{words} Rupees Only"  
 except *Exception* as e:  
 logging.error(f"Amount to words failed: {*str*(e)}")  
 row\_data['amount\_in\_words'] = ""  
  
 *# Process all placeholders in template* for ph in *template\_placeholders*:  
 if ph == 'amount\_in\_words':  
 continue *# Already handled  
  
 # Try to find matching column with flexible matching* matched\_col = None  
 norm\_ph = ph.lower().replace(' ', '\_').replace('-', '\_')  
  
 for col in *row*.index:  
 norm\_col = col.lower().replace(' ', '\_').replace('-', '\_')  
 if norm\_col == norm\_ph:  
 matched\_col = col  
 break  
  
 if matched\_col:  
 value = *row*[matched\_col]  
 *# Convert numpy types to native Python* if *hasattr*(value, 'item'):  
 value = value.item()  
 row\_data[ph] = *self*.format\_value(value, ph)  
 else:  
 row\_data[ph] = ""  
 logging.warning(f"No data mapping for placeholder: {ph}")  
  
 return row\_data  
  
 def replace\_placeholders\_in\_document(*self*, *doc*: Document, *row\_data*: Dict[*str*, *str*]) -> *bool*:  
 *"""Replace placeholders throughout document with formatting preservation"""* try:  
 *# Process all paragraphs in main document* for paragraph in *doc*.paragraphs:  
 *self*.replace\_in\_paragraph(paragraph, *row\_data*)  
  
 *# Process all tables* for table in *doc*.tables:  
 for row in table.rows:  
 for cell in row.cells:  
 for paragraph in cell.paragraphs:  
 *self*.replace\_in\_paragraph(paragraph, *row\_data*)  
  
 *# Process headers and footers* for section in *doc*.sections:  
 for header in [section.header, section.first\_page\_header]:  
 if header:  
 for paragraph in header.paragraphs:  
 *self*.replace\_in\_paragraph(paragraph, *row\_data*)  
  
 for footer in [section.footer, section.first\_page\_footer]:  
 if footer:  
 for paragraph in footer.paragraphs:  
 *self*.replace\_in\_paragraph(paragraph, *row\_data*)  
  
 return True  
  
 except *Exception* as e:  
 logging.error(f"Error replacing placeholders: {*str*(e)}", exc\_info=True)  
 return False  
  
 def replace\_in\_paragraph(*self*, *paragraph*, *row\_data*: Dict[*str*, *str*]):  
 *"""Replace placeholders in a paragraph while preserving formatting"""  
 # First combine all runs to handle split placeholders* full\_text = ''.join(run.text for run in *paragraph*.runs)  
  
 *# Skip if no placeholders* if not *any*(f'{{{{{ph}}}}}' in full\_text for ph in *row\_data*):  
 return  
  
 *# Perform all replacements* modified\_text = full\_text  
 for ph, value in *row\_data*.items():  
 modified\_text = modified\_text.replace(f'{{{{{ph}}}}}', value)  
  
 *# Only update if changes were made* if modified\_text != full\_text:  
 *# Clear existing content  
 paragraph*.clear()  
  
 *# Add new content with preserved formatting* run = *paragraph*.add\_run(modified\_text)  
 run.font.size = Pt(*self*.default\_font\_size)  
  
 *# Preserve other formatting from first run if available* if *paragraph*.runs and *paragraph*.runs[0].font.name:  
 run.font.name = *paragraph*.runs[0].font.name  
  
 def format\_value(*self*, *value*, *key*=None) -> *str*:  
 *"""Format values with special handling for certain fields"""* if pd.isna(*value*):  
 return ""  
  
 *# Handle numpy types* if *hasattr*(*value*, 'item'):  
 value = *value*.item()  
  
 *# Special formatting for amounts* if *key* and 'amount' in *key*.lower() and *isinstance*(*value*, (*int*, *float*)):  
 return "{:,.2f}".format(*value*)  
  
 *# Special handling for GSTIN (format with spaces)* if *key* and 'gstin' in *key*.lower() and *isinstance*(*value*, *str*) and *len*(*value*) == 15:  
 return f"{*value*[:2]} {*value*[2:5]} {*value*[5:7]} {*value*[7:12]} {*value*[12:15]}"  
  
 return *str*(*value*).strip()  
  
 def extract\_placeholders(*self*, *template\_path*: *str*) -> Set[*str*]:  
 *"""Extract all unique placeholders from a DOCX template"""* doc = Document(*template\_path*)  
 placeholders = *set*()  
 placeholder\_pattern = re.compile(r'\{\{\s\*(.\*?)\s\*\}\}') *# Handles whitespace* def extract\_from\_text(*text*: *str*):  
 *"""Inner function to extract placeholders from text"""* return {match.strip() for match in placeholder\_pattern.findall(*text*)}  
  
 *# Process all document components* components = [  
 doc.paragraphs,  
 \*[cell.paragraphs for table in doc.tables  
 for row in table.rows  
 for cell in row.cells],  
 \*[section.header.paragraphs for section in doc.sections],  
 \*[section.footer.paragraphs for section in doc.sections]  
 ]  
  
 for paragraphs in components:  
 for paragraph in paragraphs:  
 placeholders.update(extract\_from\_text(paragraph.text))  
 for run in paragraph.runs:  
 placeholders.update(extract\_from\_text(run.text))  
  
 return {ph for ph in placeholders if ph} *# Remove empty strings* def generate\_output\_path(*self*, *output\_folder*: *str*, *row\_data*: *dict*, *idx*: *int*) -> *str*:  
 *"""Generate output path with invoice number if available"""* invoice\_num = *str*(*row\_data*.get('INVOICE\_NUMBER', *idx* + 1)).strip()  
 timestamp = datetime.now().strftime("%Y%m%d\_%H%M%S")  
 return os.path.join(*output\_folder*, f"Document\_{invoice\_num}\_{timestamp}.docx")

Utils/data\_utils.py :-  
*# (Your original data\_utils.py content)*import pandas as pd  
from PyQt5.QtWidgets import QMessageBox, QTableWidgetItem  
  
def filter\_data(*df*, *search\_query*, *sub\_query*, *main\_column*, *sub\_column*, *filter\_type*):  
 filtered\_data = *df*.copy()  
  
 if *search\_query*:  
 if *main\_column* == "All Columns":  
 filtered\_data = filtered\_data[  
 filtered\_data.apply(lambda *row*: row.astype(*str*).str.contains(*search\_query*, case=False, na=False).any(), axis=1)  
 ]  
 else:  
 if *filter\_type* == "Contains":  
 filtered\_data = filtered\_data[filtered\_data[*main\_column*].astype(*str*).str.contains(*search\_query*, case=False, na=False)]  
 elif *filter\_type* == "Equals":  
 filtered\_data = filtered\_data[filtered\_data[*main\_column*].astype(*str*) == *search\_query*]  
 elif *filter\_type* == "Starts with":  
 filtered\_data = filtered\_data[filtered\_data[*main\_column*].astype(*str*).str.startswith(*search\_query*, na=False)]  
  
 if *sub\_query*:  
 if *sub\_column* == "All Columns":  
 filtered\_data = filtered\_data[  
 filtered\_data.apply(lambda *row*: row.astype(*str*).str.contains(*sub\_query*, case=False, na=False).any(), axis=1)  
 ]  
 else:  
 filtered\_data = filtered\_data[filtered\_data[*sub\_column*].astype(*str*).str.contains(*sub\_query*, case=False, na=False)]  
  
 if filtered\_data.empty:  
 QMessageBox.showinfo("No Results", "No matching records found.")  
 return pd.DataFrame()  
  
 return filtered\_data  
  
def display\_data(*table*, *data*, sort\_orders):  
 *table*.setRowCount(0)  
 *table*.setColumnCount(*len*(*data*.columns))  
 *table*.setHorizontalHeaderLabels(*list*(*data*.columns))  
  
 for i, row in *data*.iterrows():  
 for j, col in *enumerate*(*data*.columns):  
 item = QTableWidgetItem(*str*(row[col]))  
 *table*.setItem(i, j, item)

Utils/ file\_utils.py:-  
  
from PyQt5.QtWidgets import QFileDialog, QMessageBox  
import pandas as pd  
from reportlab.platypus import SimpleDocTemplate, Table, TableStyle  
from reportlab.lib import colors  
from reportlab.lib.pagesizes import A4  
  
def upload\_file():  
 file\_path, \_ = QFileDialog.getOpenFileName(None, "Open File", "", "Excel/CSV Files (\*.xlsx \*.xls \*.csv)")  
 if file\_path:  
 try:  
 df = pd.read\_csv(file\_path, encoding="utf-8", low\_memory=False) if file\_path.endswith(".csv") else pd.read\_excel(file\_path, sheet\_name=0)  
 if df.empty:  
 QMessageBox.showerror("Error", "Loaded file is empty or could not be read.")  
 return None  
 QMessageBox.showinfo("Success", "File uploaded successfully!")  
 return df  
 except *Exception* as e:  
 QMessageBox.showerror("Error", f"Failed to load file: {e}")  
 return None  
 return None  
  
def export\_filtered\_data(*df*, *format*):  
 save\_path, \_ = QFileDialog.getSaveFileName(None, "Save File", "", f"{*format*.upper()} Files (\*.{*format*})")  
 if save\_path:  
 try:  
 if *format* == "xlsx":  
 *df*.to\_excel(save\_path, index=False)  
 elif *format* == "csv":  
 *df*.to\_csv(save\_path, index=False)  
 elif *format* == "pdf":  
 save\_df\_as\_pdf(*df*, save\_path)  
 QMessageBox.showinfo("Success", f"Filtered data saved as {*format*.upper()} successfully!")  
 except *Exception* as e:  
 QMessageBox.showerror("Error", f"Failed to save file: {e}")  
  
def save\_df\_as\_pdf(*df*, *save\_path*):  
 doc = SimpleDocTemplate(*save\_path*, pagesize=A4)  
 elements = []  
 data = [*df*.columns.tolist()] + *df*.astype(*str*).values.tolist()  
 table = Table(data)  
 style = TableStyle([  
 ('BACKGROUND', (0, 0), (-1, 0), colors.grey),  
 ('TEXTCOLOR', (0, 0), (-1, 0), colors.whitesmoke),  
 ('ALIGN', (0, 0), (-1, -1), 'CENTER'),  
 ('FONTNAME', (0, 0), (-1, 0), 'Helvetica-Bold'),  
 ('BOTTOMPADDING', (0, 0), (-1, 0), 8),  
 ('BACKGROUND', (0, 1), (-1, -1), colors.beige),  
 ('GRID', (0, 0), (-1, -1), 1, colors.black),  
 ('FONTSIZE', (0, 0), (-1, -1), 8),  
 ])  
 table.setStyle(style)  
 elements.append(table)  
 doc.build(elements)

Utils/gui\_utils.py:-

from PyQt5.QtWidgets import QTableWidget, QTableWidgetItem, QHeaderView  
from PyQt5.QtCore import Qt  
  
def create\_table\_widget(*parent*):  
 *"""  
 Creates a QTableWidget with scrollbars.  
 """* table = QTableWidget(*parent*)  
 table.setAlternatingRowColors(True)  
 table.setEditTriggers(QTableWidget.NoEditTriggers) *# Make table read-only* table.setSelectionBehavior(QTableWidget.SelectRows) *# Select entire rows* table.setSelectionMode(QTableWidget.SingleSelection) *# Single row selection  
  
 # Configure horizontal header* header = table.horizontalHeader()  
 header.setSectionResizeMode(QHeaderView.ResizeToContents) *# Resize columns to content* return table  
  
def display\_data(*table*, *data*, *sort\_orders*=None):  
 *"""  
 Displays data in the QTableWidget.  
 """* if *sort\_orders* is None:  
 sort\_orders = {}  
  
 *table*.setRowCount(0) *# Clear existing data* if *data*.empty:  
 *table*.setColumnCount(0)  
 return  
  
 *table*.setColumnCount(*len*(*data*.columns))  
 *table*.setHorizontalHeaderLabels(*list*(*data*.columns))  
  
 for i, row in *data*.iterrows():  
 *table*.insertRow(i)  
 for j, col in *enumerate*(*data*.columns):  
 item = QTableWidgetItem(*str*(row[col]))  
 item.setTextAlignment(Qt.AlignCenter) *# Align text to center  
 table*.setItem(i, j, item)  
  
 *# Adjust column widths to content  
  
 table*.resizeColumnsToContents()

Utils/invoice\_generator.py :-  
  
import os  
import pandas as pd  
from fpdf import FPDF  
from PyQt5.QtWidgets import QFileDialog, QMessageBox  
  
  
class InvoiceGenerator:  
 def \_\_init\_\_(*self*, *template\_path*=None):  
 *"""Initialize invoice generator with an optional PDF template."""  
 self*.template\_path = *template\_path  
 self*.df = None *# Store the loaded data* def load\_invoice\_data(*self*, *file\_path*=None):  
 *"""Loads invoice data from an Excel/CSV file."""* if *file\_path* is None:  
 file\_path, \_ = QFileDialog.getOpenFileName(None, "Open File", "", "Excel/CSV Files (\*.xlsx \*.xls \*.csv)") *# Change 1* if not *file\_path*:  
 QMessageBox.showerror("Error", "No file selected!") *# Change 2* return None  
  
 try:  
 if *file\_path*.endswith(".csv"):  
 *self*.df = pd.read\_csv(*file\_path*)  
 else:  
 *self*.df = pd.read\_excel(*file\_path*)  
  
 *# Debug: Print the DataFrame  
 print*("DataFrame loaded:")  
 *print*(*self*.df)  
  
 if *self*.df.empty:  
 QMessageBox.showerror("Error", "The file is empty!") *# Change 2* return None  
  
 QMessageBox.showinfo("Success", f"Loaded {*len*(*self*.df)} invoice records!") *# Change 2* return *self*.df  
  
 except *Exception* as e:  
 QMessageBox.showerror("Error", f"Failed to load file: {e}") *# Change 2* return None  
  
 def generate\_invoices(*self*, *output\_folder*=None):  
 *"""Generates PDF invoices from the dataset."""* if *self*.df is None or *self*.df.empty:  
 QMessageBox.showerror("Error", "No data loaded for invoices!") *# Change 2* return  
  
 if *output\_folder* is None:  
 output\_folder = QFileDialog.getExistingDirectory(None, "Select Output Folder") *# Change 1* if not *output\_folder*:  
 QMessageBox.showerror("Error", "No output folder selected!") *# Change 2* return  
  
 *# Debug: Print the output folder  
 print*(f"Output folder: {*output\_folder*}")  
  
 *# Ensure the output folder exists* if not os.path.exists(*output\_folder*):  
 os.makedirs(*output\_folder*)  
  
 *# Generate a PDF for each row of data (starting from the second row)* for index, row in *self*.df.iloc[1:].iterrows():  
 *self*.create\_invoice(row, *output\_folder*, index)  
  
 QMessageBox.showinfo("Success", f"Invoices saved in {*output\_folder*}!") *# Change 2* def create\_invoice(*self*, *data\_row*, *output\_folder*, *invoice\_number*):  
 *"""Creates a single invoice PDF."""* pdf = FPDF()  
 pdf.set\_auto\_page\_break(auto=True, margin=15)  
 pdf.add\_page()  
  
 *# Add invoice title* pdf.set\_font("Arial", "B", 16)  
 pdf.cell(200, 10, "Invoice", ln=True, align='C')  
  
 *# Add client details* pdf.set\_font("Arial", size=12)  
 pdf.cell(100, 10, f"Client: {*data\_row*.get(*self*.df.columns[0], 'Unknown')}", ln=True)  
 pdf.cell(100, 10, f"Invoice #: {*invoice\_number*}", ln=True)  
  
 *# Add a space* pdf.cell(200, 10, "", ln=True)  
  
 *# Add table headers* pdf.set\_font("Arial", "B", 12)  
 for col in *self*.df.columns:  
 pdf.cell(40, 10, col, 1)  
 pdf.ln()  
  
 *# Add table data* pdf.set\_font("Arial", size=12)  
 for col in *self*.df.columns:  
 pdf.cell(40, 10, *str*(*data\_row*[col]), 1)  
 pdf.ln()  
  
 *# Save the PDF* output\_path = os.path.join(*output\_folder*, f"Invoice\_{*invoice\_number*}.pdf")  
 pdf.output(output\_path)  
  
 *# Debug: Verify the PDF was created* if os.path.exists(output\_path):  
 *print*(f"✅ Invoice saved: {output\_path}")  
 else:  
 *print*(f"❌ Failed to save invoice: {output\_path}")

Utils/ invoice\_utils.py :-  
  
from fpdf import FPDF  
from PyQt5.QtWidgets import QMessageBox, QFileDialog  
  
def generate\_pdf\_invoice(*df*, *selected\_columns*, *invoice\_number*, *customer\_name*, *invoice\_date*, *parent*):  
 *"""Generates a PDF invoice from a DataFrame."""* if not *selected\_columns*:  
 QMessageBox.warning(*parent*, "Error", "Please select at least one column.")  
 return False  
  
 selected\_columns\_names = [item.text() for item in *selected\_columns*]  
 invoice\_df = *df*[selected\_columns\_names].copy()  
  
 *# Create PDF Invoice* pdf = FPDF()  
 pdf.add\_page()  
 pdf.set\_font("Arial", "B", 16)  
 pdf.cell(200, 10, "Invoice", ln=True, align='C')  
  
 pdf.set\_font("Arial", size=12)  
 pdf.cell(200, 10, f"Invoice Number: {*invoice\_number*}", ln=True)  
 pdf.cell(200, 10, f"Customer Name: {*customer\_name*}", ln=True)  
 pdf.cell(200, 10, f"Invoice Date: {*invoice\_date*}", ln=True)  
 pdf.ln(10)  
  
 *# Add table headers* pdf.set\_font("Arial", "B", 12)  
 for col in invoice\_df.columns:  
 pdf.cell(40, 10, col, 1)  
 pdf.ln()  
  
 *# Add table data* pdf.set\_font("Arial", size=12)  
 for \_, row in invoice\_df.iterrows():  
 for col in invoice\_df.columns:  
 pdf.cell(40, 10, *str*(row[col]), 1)  
 pdf.ln()  
  
 *# Save the PDF* file\_path, \_ = QFileDialog.getSaveFileName(*parent*, "Save Invoice", "", "PDF Files (\*.pdf)")  
 if file\_path:  
 pdf.output(file\_path)  
 QMessageBox.information(*parent*, "Success", "Invoice generated successfully!")  
 return True  
 else:  
 QMessageBox.information(*parent*, "Cancelled", "Invoice generation cancelled.")  
 return False

Utils/ pdf\_generator.py :-  
  
from docx2pdf import convert  
import os  
from PyQt5.QtWidgets import QMessageBox  
  
def generate\_pdfs(*docx\_files*, *output\_folder*):  
 *"""Converts a list of DOCX files to PDFs and saves them in the output folder."""* pdf\_files = []  
 for docx\_file in *docx\_files*:  
 if not os.path.exists(docx\_file):  
 *print*(f"ERROR: DOCX file not found: {docx\_file}")  
 continue  
  
 pdf\_file = os.path.join(*output\_folder*, os.path.splitext(os.path.basename(docx\_file))[0] + ".pdf")  
 try:  
 convert(docx\_file, pdf\_file)  
 pdf\_files.append(pdf\_file)  
 except *Exception* as e:  
 *print*(f"ERROR: Failed to convert {docx\_file}: {e}")  
  
 if pdf\_files:  
 *print*(f"INFO: Successfully converted {*len*(pdf\_files)} files to PDF.")  
 return pdf\_files  
 else:  
 *print*("INFO: Successfully converted 0 files to PDF.")  
 return None

Utils/Pdf\_utils.py :-  
  
import fitz  
from PIL import Image  
from PyQt5.QtGui import QPixmap, QImage  
  
def add\_text\_to\_pdf(*pdf\_path*, *boxes\_and\_columns*, *data*, *output\_path*):  
 doc = fitz.open(*pdf\_path*)  
 page = doc[0] *# Assuming first page* for box, column in *boxes\_and\_columns*.items():  
 text = *str*(*data*.iloc[0][column]) *# Get data from the first row* x = box.x()  
 y = box.y()  
  
 page.insert\_text((x, y), text)  
  
 doc.save(*output\_path*)  
  
def load\_pdf(*pdf\_path*):  
 *"""  
 Loads the first page of a PDF as a QPixmap.  
  
 Args:  
 pdf\_path (str): Path to the PDF file.  
  
 Returns:  
 tuple: (QPixmap, fitz.Document) or (None, None) in case of error.  
 """* try:  
 pdf\_document = fitz.open(*pdf\_path*)  
 page = pdf\_document[0]  
 pix = page.get\_pixmap()  
 img = Image.frombytes("RGB", [pix.width, pix.height], pix.samples)  
 img.thumbnail((800, 1000)) *# Resize the image  
  
 # Convert PIL Image to QImage* qimage = QImage(img.tobytes("raw", "RGB"), img.width, img.height, QImage.Format\_RGB888)  
  
 *# Convert QImage to QPixmap* pdf\_pixmap = QPixmap.fromImage(qimage)  
  
 return pdf\_pixmap, pdf\_document  
 except *Exception* as e:  
 *print*(f"Error loading PDF: {e}")  
 return None, None

Utils/Theme\_manager.py :-  
  
from PyQt5.QtWidgets import QApplication, QMainWindow, QMenu, QAction  
from PyQt5.QtCore import Qt  
import qdarkstyle  
import darkdetect  
  
def get\_system\_theme():  
 return "dark" if darkdetect.isDark() else "light"  
  
def apply\_theme(*app*, *theme*):  
 if *theme* == "dark":  
 *app*.setStyleSheet(qdarkstyle.load\_stylesheet(qt\_api='pyqt5'))  
 else:  
 *app*.setStyleSheet("") *# Reset to default light theme*class ThemeManager:  
 def \_\_init\_\_(*self*, *main\_window*):  
 *self*.main\_window = *main\_window  
 self*.app = QApplication.instance() *# Get the application instance  
 self*.current\_theme = get\_system\_theme()  
 apply\_theme(*self*.app, *self*.current\_theme)  
  
 *self*.create\_theme\_menu()  
  
 def create\_theme\_menu(*self*):  
 menu\_bar = *self*.main\_window.menuBar()  
 theme\_menu = QMenu("Theme", *self*.main\_window)  
  
 dark\_action = QAction("🌙 Dark", *self*.main\_window)  
 dark\_action.triggered.connect(lambda: *self*.set\_theme("dark"))  
 theme\_menu.addAction(dark\_action)  
  
 light\_action = QAction("☀️ Light", *self*.main\_window)  
 light\_action.triggered.connect(lambda: *self*.set\_theme("light"))  
 theme\_menu.addAction(light\_action)  
  
 menu\_bar.addMenu(theme\_menu)  
  
 def set\_theme(*self*, *theme*):  
 *self*.current\_theme = *theme* apply\_theme(*self*.app, *theme*)

Views/\_\_init\_\_.py :-  
  
from .pdf\_view import PDFView  
from .main\_view import MainView  
from .invoice\_view import InvoiceView

Views/ invoice\_view.py :-  
  
from PyQt5.QtWidgets import QWidget, QVBoxLayout, QPushButton  
from utils.invoice\_generator import InvoiceGenerator  
  
class InvoiceView(QWidget):  
 def \_\_init\_\_(*self*, *parent*=None):  
 *super*().\_\_init\_\_(*parent*)  
 *self*.invoice\_generator = InvoiceGenerator()  
 *self*.init\_ui()  
  
 def init\_ui(*self*):  
 layout = QVBoxLayout()  
  
 *# Upload Button  
 self*.upload\_btn = QPushButton("📂 Upload Invoice Data")  
 *self*.upload\_btn.clicked.connect(*self*.upload\_file)  
 layout.addWidget(*self*.upload\_btn)  
  
 *# Generate Invoice Button  
 self*.generate\_btn = QPushButton("📄 Generate Invoices")  
 *self*.generate\_btn.clicked.connect(*self*.generate\_invoices)  
 layout.addWidget(*self*.generate\_btn)  
  
 *self*.setLayout(layout)  
  
 def upload\_file(*self*):  
 *"""Load invoice data"""  
 self*.invoice\_generator.load\_invoice\_data()  
  
 def generate\_invoices(*self*):  
 *"""Generate invoices"""  
 self*.invoice\_generator.generate\_invoices()

Views/ main\_view.py :-

from PyQt5.QtWidgets import QWidget, QVBoxLayout, QHBoxLayout, QPushButton, QComboBox, QLineEdit, QMenu, QMenuBar, QAction, QFileDialog, QMessageBox, QTreeWidget, QTreeWidgetItem  
from PyQt5.QtCore import Qt  
import pandas as pd  
import pdfplumber  
import fitz  
import os  
from utils.file\_utils import export\_filtered\_data  
from utils.data\_utils import filter\_data  
from utils.docx\_filler import fill\_docx\_template  
from views.pdf\_view import PDFView  
from utils.gui\_utils import display\_data, create\_table\_widget  
  
class MainView(QWidget):  
 def \_\_init\_\_(*self*, *parent*=None):  
 *super*().\_\_init\_\_(*parent*)  
 *self*.df = None  
 *self*.filtered\_df = None  
 *self*.sort\_orders = {} *# Track sorting order for columns  
 self*.template\_file = None  
 *self*.output\_folder = None  
 *self*.pdf\_path = None  
 *self*.pdf\_document = None  
 *self*.box\_data = None  
 *self*.pdf\_canvas = None  
 *self*.init\_ui()  
  
 def init\_ui(*self*):  
 layout = QVBoxLayout()  
  
 *# Top Frame Layout* top\_frame = QWidget()  
 top\_layout = QHBoxLayout()  
  
 *# Upload Button  
 self*.upload\_btn = QPushButton("📂 Upload File")  
 *self*.upload\_btn.clicked.connect(*self*.upload\_file)  
 top\_layout.addWidget(*self*.upload\_btn)  
  
 *# Search Entry  
 self*.search\_entry = QLineEdit()  
 *self*.search\_entry.setPlaceholderText("Search...")  
 *self*.search\_entry.returnPressed.connect(*self*.search\_and\_generate)  
 top\_layout.addWidget(*self*.search\_entry)  
  
 *# Search Button  
 self*.search\_btn = QPushButton("🔍")  
 *self*.search\_btn.clicked.connect(*self*.search\_and\_generate)  
 top\_layout.addWidget(*self*.search\_btn)  
  
 *# Sub-Search Entry  
 self*.sub\_search\_entry = QLineEdit()  
 *self*.sub\_search\_entry.setPlaceholderText("Sub-Search...")  
 *self*.sub\_search\_entry.returnPressed.connect(*self*.search\_and\_generate)  
 top\_layout.addWidget(*self*.sub\_search\_entry)  
  
 *# Sub-Search Column Dropdown  
 self*.sub\_search\_column\_dropdown = QComboBox()  
 *self*.sub\_search\_column\_dropdown.addItem("All Columns")  
 top\_layout.addWidget(*self*.sub\_search\_column\_dropdown)  
  
 *# Sub-Search Button  
 self*.sub\_search\_btn = QPushButton("🔍 Sub-Search")  
 *self*.sub\_search\_btn.clicked.connect(*self*.search\_and\_generate)  
 top\_layout.addWidget(*self*.sub\_search\_btn)  
  
 *# Column Dropdown  
 self*.column\_dropdown = QComboBox()  
 *self*.column\_dropdown.addItem("All Columns")  
 top\_layout.addWidget(*self*.column\_dropdown)  
  
 *# Filter Type Dropdown  
 self*.filter\_dropdown = QComboBox()  
 *self*.filter\_dropdown.addItems(["Contains", "Equals", "Starts with"])  
 top\_layout.addWidget(*self*.filter\_dropdown)  
  
 *# Clear Button  
 self*.clear\_btn = QPushButton("❌ Clear Filters")  
 *self*.clear\_btn.clicked.connect(*self*.clear\_filters)  
 top\_layout.addWidget(*self*.clear\_btn)  
  
 *# Load PDF Button  
 self*.load\_pdf\_btn = QPushButton("📂 Load PDF")  
 *self*.load\_pdf\_btn.clicked.connect(*self*.load\_pdf)  
 top\_layout.addWidget(*self*.load\_pdf\_btn)  
  
 *# PDF to Excel Button  
 self*.pdf\_to\_excel\_btn = QPushButton("📥 PDF to Excel")  
 *self*.pdf\_to\_excel\_btn.clicked.connect(*self*.convert\_pdf\_to\_excel)  
 top\_layout.addWidget(*self*.pdf\_to\_excel\_btn)  
  
 *# Export Menu Button  
 self*.export\_menu\_btn = QPushButton("📤 Export")  
 *self*.export\_menu = QMenu(*self*)  
 *self*.export\_menu.addAction("📤 Export as CSV", lambda: *self*.export\_filtered\_data("csv"))  
 *self*.export\_menu.addAction("📤 Export as Excel", lambda: *self*.export\_filtered\_data("xlsx"))  
 *self*.export\_menu.addAction("📤 Export Full PDF", lambda: *self*.export\_filtered\_data("pdf"))  
 *self*.export\_menu.addAction("📤 Export Individual PDFs", *self*.export\_each\_row\_as\_pdf)  
 *self*.export\_menu\_btn.setMenu(*self*.export\_menu)  
 top\_layout.addWidget(*self*.export\_menu\_btn)  
  
 *# Export PDFs Button  
 self*.export\_pdfs\_btn = QPushButton("📤 Export PDFs")  
 *self*.export\_pdfs\_btn.clicked.connect(*self*.export\_filled\_pdfs)  
 top\_layout.addWidget(*self*.export\_pdfs\_btn)  
  
 top\_frame.setLayout(top\_layout)  
 layout.addWidget(top\_frame)  
  
 *# Treeview  
 self*.tree = create\_table\_widget(*self*)  
 layout.addWidget(*self*.tree)  
  
 *self*.setLayout(layout)  
  
 def clear\_filters(*self*):  
 *"""Resets all search filters and refreshes the dataset."""* if *self*.df is None:  
 QMessageBox.showerror(*self*, "Error", "No data loaded to clear filters.")  
 return  
  
 *self*.filtered\_df = *self*.df.copy() *# Reset data  
 self*.sort\_orders = {} *# Reset sorting order  
  
 self*.search\_entry.setText("")  
 *self*.sub\_search\_entry.setText("")  
 *self*.column\_dropdown.setCurrentText("All Columns")  
 *self*.sub\_search\_column\_dropdown.setCurrentText("All Columns")  
 *self*.filter\_dropdown.setCurrentText("Contains")  
  
 *self*.display\_data(*self*.search\_entry.text(), *self*.sub\_search\_entry.text(), *self*.column\_dropdown.currentText(),  
 *self*.sub\_search\_column\_dropdown.currentText(), *self*.filter\_dropdown.currentText())  
  
 def upload\_file(*self*):  
 *"""Handles the file upload."""* file\_path, \_ = QFileDialog.getOpenFileName(*self*, "Open File", "", "Excel Files (\*.xlsx);;CSV Files (\*.csv)")  
 if file\_path:  
 try:  
 if file\_path.endswith(".xlsx"):  
 *self*.df = pd.read\_excel(file\_path)  
 elif file\_path.endswith(".csv"):  
 *self*.df = pd.read\_csv(file\_path)  
 *self*.column\_dropdown.clear()  
 *self*.column\_dropdown.addItem("All Columns")  
 *self*.column\_dropdown.addItems(*list*(*self*.df.columns))  
 *self*.sub\_search\_column\_dropdown.clear()  
 *self*.sub\_search\_column\_dropdown.addItem("All Columns")  
 *self*.sub\_search\_column\_dropdown.addItems(*list*(*self*.df.columns))  
 *self*.display\_data()  
 except *Exception* as e:  
 QMessageBox.showerror(*self*, "Error", f"Failed to upload file: {e}")  
 else:  
 *print*("No file selected.")  
  
 def display\_data(*self*, *search\_query*="", *sub\_query*="", *main\_column*="All Columns", *sub\_column*="All Columns", *filter\_type*="Contains"):  
 *"""Filters and updates the Treeview based on search criteria."""* if *self*.df is None:  
 return  
  
 *# Apply filtering* filtered\_df = filter\_data(*self*.df, *search\_query*, *sub\_query*, *main\_column*, *sub\_column*, *filter\_type*)  
  
 *# Update Treeview* display\_data(*self*.tree, filtered\_df, *self*.sort\_orders)  
  
 def open\_pdf\_view(*self*):  
 *"""Opens the PDF view with the loaded PDF."""* if *self*.pdf\_path:  
 pdf\_window = QWidget()  
 pdf\_window.setWindowTitle("PDF Preview")  
 pdf\_view\_instance = PDFView(pdf\_window, *self*.pdf\_path)  
 layout = QVBoxLayout()  
 layout.addWidget(pdf\_view\_instance)  
 pdf\_window.setLayout(layout)  
 pdf\_window.show()  
 else:  
 QMessageBox.showerror(*self*, "Error", "No PDF loaded.")  
  
 def start\_processing(*self*):  
 *"""Handles the processing of data and template files."""* if not *self*.df or not *self*.template\_file or not *self*.output\_folder:  
 QMessageBox.showerror(*self*, "Error", "Please upload all required files!")  
 return  
  
 *# Fill DOCX Templates for Each Row* filled\_files =

View/ pdf\_view.py :-  
  
from PyQt5.QtWidgets import QWidget, QLabel, QScrollArea, QVBoxLayout, QHBoxLayout, QPushButton, QComboBox, QLineEdit  
from PyQt5.QtGui import QPixmap  
from PyQt5.QtCore import Qt  
from utils.pdf\_utils import load\_pdf  
  
class PDFView(QWidget):  
 def \_\_init\_\_(*self*, *parent*, *pdf\_path*):  
 *super*().\_\_init\_\_(*parent*)  
 *self*.pdf\_path = *pdf\_path  
 self*.text\_boxes = []  
 *self*.box\_data = []  
 *self*.pdf\_pixmap = None  
 *self*.pdf\_document = None  
 *self*.init\_ui()  
  
 def init\_ui(*self*):  
 layout = QHBoxLayout(*self*)  
  
 *self*.pdf\_pixmap, *self*.pdf\_document = load\_pdf(*self*.pdf\_path)  
  
 if *self*.pdf\_pixmap:  
 *self*.scroll\_area = QScrollArea()  
 *self*.scroll\_area.setAlignment(Qt.AlignCenter)  
 *self*.scroll\_area.setWidgetResizable(True)  
  
 *self*.pdf\_label = QLabel()  
 *self*.pdf\_label.setPixmap(*self*.pdf\_pixmap)  
 *self*.scroll\_area.setWidget(*self*.pdf\_label)  
  
 layout.addWidget(*self*.scroll\_area, 4)  
  
 *self*.frame\_right = QWidget()  
 *self*.frame\_right.setStyleSheet("background-color: #f0f0f0;")  
 right\_layout = QVBoxLayout(*self*.frame\_right)  
  
 *self*.btn\_add\_box = QPushButton("➕ Add Text Box")  
 *self*.btn\_add\_box.clicked.connect(*self*.add\_text\_box)  
 right\_layout.addWidget(*self*.btn\_add\_box)  
  
 layout.addWidget(*self*.frame\_right, 1)  
 else:  
 error\_label = QLabel("Failed to load PDF.")  
 layout.addWidget(error\_label)  
  
 *self*.setLayout(layout)  
  
 def add\_text\_box(*self*):  
 frame = QWidget()  
 frame\_layout = QHBoxLayout(frame)  
  
 entry = QLineEdit()  
 frame\_layout.addWidget(entry)  
  
 *self*.text\_boxes.append(entry)  
  
 *# Create a placeholder label for positioning* placeholder\_label = QLabel()  
 placeholder\_label.setText("Placeholder")  
 placeholder\_label.setStyleSheet("border: 1px solid red;")  
 placeholder\_label.setFixedSize(100,20)  
  
 *# Add to the scroll area  
 self*.pdf\_label.layout = QVBoxLayout()  
 *self*.pdf\_label.setLayout(*self*.pdf\_label.layout)  
 *self*.pdf\_label.layout.addWidget(placeholder\_label)  
  
 *# Store the data  
 self*.box\_data.append({"entry": entry, "label": placeholder\_label, "x": 50, "y": 50, "column": None})

Widget/ \_\_init\_\_.py :-  
  
from .search\_widgets import SearchWidgets  
from .export\_widgets import ExportWidgets  
from .pdf\_preview\_widget import PDFPreviewWidget

Widget/ export\_widgets.py :-   
  
from PyQt5.QtWidgets import QWidget, QPushButton, QMenu, QHBoxLayout  
  
class ExportWidgets(QWidget):  
 def \_\_init\_\_(*self*, *parent*, *filtered\_df*, *export\_data\_callback*):  
 *super*().\_\_init\_\_(*parent*)  
 *self*.filtered\_df = *filtered\_df  
 self*.export\_data\_callback = *export\_data\_callback  
 self*.create\_widgets()  
  
 def create\_widgets(*self*):  
 layout = QHBoxLayout(*self*)  
  
 export\_menu\_btn = QPushButton("📤 Export")  
 export\_menu = QMenu(*self*)  
 export\_menu.addAction("📤 Export as CSV", lambda: *self*.export\_data\_callback("csv"))  
 export\_menu.addAction("📤 Export as Excel", lambda: *self*.export\_data\_callback("xlsx"))  
 export\_menu.addAction("📤 Export as PDF", lambda: *self*.export\_data\_callback("pdf"))  
 export\_menu\_btn.setMenu(export\_menu)  
  
 layout.addWidget(export\_menu\_btn)  
 *self*.setLayout(layout)

Widget/ pdf\_preview\_widget.py :-

from PyQt5.QtWidgets import QWidget, QLabel, QScrollArea, QVBoxLayout, QHBoxLayout, QPushButton, QLineEdit  
from PyQt5.QtGui import QPixmap  
from PyQt5.QtCore import Qt  
from utils.pdf\_utils import load\_pdf  
  
class PDFPreviewWidget(QWidget):  
 def \_\_init\_\_(*self*, *parent*, *pdf\_path*):  
 *super*().\_\_init\_\_(*parent*)  
 *self*.parent = *parent  
 self*.pdf\_path = *pdf\_path  
 self*.text\_boxes = []  
 *self*.box\_data = []  
 *self*.create\_widgets()  
  
 def create\_widgets(*self*):  
 *# Load PDF  
 self*.pdf\_pixmap, *self*.pdf\_document = load\_pdf(*self*.pdf\_path)  
  
 if *self*.pdf\_pixmap:  
 *# Scroll Area for PDF Preview  
 self*.scroll\_area = QScrollArea()  
 *self*.scroll\_area.setAlignment(Qt.AlignCenter)  
 *self*.scroll\_area.setWidgetResizable(True)  
  
 *self*.pdf\_label = QLabel()  
 *self*.pdf\_label.setPixmap(*self*.pdf\_pixmap)  
 *self*.scroll\_area.setWidget(*self*.pdf\_label)  
  
 *# Frame for Right-Side Controls  
 self*.frame\_right = QWidget()  
 *self*.frame\_right.setStyleSheet("background-color: #f0f0f0;")  
 right\_layout = QVBoxLayout(*self*.frame\_right)  
  
 *# Add Text Box Button  
 self*.btn\_add\_box = QPushButton("➕ Add Text Box")  
 *self*.btn\_add\_box.clicked.connect(*self*.add\_text\_box)  
 right\_layout.addWidget(*self*.btn\_add\_box)  
  
 *# Layout for the entire widget* layout = QHBoxLayout(*self*)  
 layout.addWidget(*self*.scroll\_area, 4) *# PDF preview takes 4/5 of the space* layout.addWidget(*self*.frame\_right, 1) *# Controls take 1/5 of the space  
 self*.setLayout(layout)  
 else:  
 error\_label = QLabel("Failed to load PDF.")  
 layout = QHBoxLayout(*self*)  
 layout.addWidget(error\_label)  
 *self*.setLayout(layout)  
  
 def add\_text\_box(*self*):  
 *"""Adds a text box to the PDF preview."""* frame = QWidget()  
 frame\_layout = QHBoxLayout(frame)  
  
 entry = QLineEdit()  
 frame\_layout.addWidget(entry)  
  
 *self*.text\_boxes.append(entry)  
  
 *# Create a placeholder label for positioning* placeholder\_label = QLabel()  
 placeholder\_label.setText("Placeholder")  
 placeholder\_label.setStyleSheet("border: 1px solid red;")  
 placeholder\_label.setFixedSize(100,20)  
  
 *# Add to the scroll area  
 self*.pdf\_label.layout = QVBoxLayout()  
 *self*.pdf\_label.setLayout(*self*.pdf\_label.layout)  
 *self*.pdf\_label.layout.addWidget(placeholder\_label)  
  
 *# Store the data  
 self*.box\_data.append({"entry": entry, "label": placeholder\_label, "x": 50, "y": 50, "column": None})

Widget/ search\_widgets.py :-  
  
from PyQt5.QtWidgets import QWidget, QLineEdit, QComboBox, QPushButton, QHBoxLayout  
from PyQt5.QtCore import Qt  
  
class SearchWidgets(QWidget):  
 def \_\_init\_\_(*self*, *parent*, *tree*, *display\_data\_callback*):  
 *super*().\_\_init\_\_(*parent*)  
 *self*.tree = *tree  
 self*.display\_data\_callback = *display\_data\_callback  
 self*.create\_widgets()  
  
 def create\_widgets(*self*):  
 layout = QHBoxLayout(*self*)  
  
 *self*.search\_entry = QLineEdit()  
 *self*.search\_entry.setPlaceholderText("Search...")  
 layout.addWidget(*self*.search\_entry)  
  
 *self*.sub\_search\_entry = QLineEdit()  
 *self*.sub\_search\_entry.setPlaceholderText("Sub-Search...")  
 layout.addWidget(*self*.sub\_search\_entry)  
  
 *self*.column\_dropdown = QComboBox()  
 *self*.column\_dropdown.addItem("All Columns")  
 layout.addWidget(*self*.column\_dropdown)  
  
 *self*.sub\_search\_column\_dropdown = QComboBox()  
 *self*.sub\_search\_column\_dropdown.addItem("All Columns")  
 layout.addWidget(*self*.sub\_search\_column\_dropdown)  
  
 *self*.filter\_dropdown = QComboBox()  
 *self*.filter\_dropdown.addItems(["Contains", "Equals", "Starts with"])  
 layout.addWidget(*self*.filter\_dropdown)  
  
 *self*.search\_btn = QPushButton("🔍Search")  
 *self*.search\_btn.clicked.connect(*self*.perform\_search)  
 layout.addWidget(*self*.search\_btn)  
  
 *self*.clear\_btn = QPushButton("Clear")  
 *self*.clear\_btn.clicked.connect(*self*.clear\_filters)  
 layout.addWidget(*self*.clear\_btn)  
  
 *self*.setLayout(layout)  
  
 def perform\_search(*self*):  
 search\_query = *self*.search\_entry.text().strip()  
 sub\_query = *self*.sub\_search\_entry.text().strip()  
 main\_column = *self*.column\_dropdown.currentText()  
 sub\_column = *self*.sub\_search\_column\_dropdown.currentText()  
 filter\_type = *self*.filter\_dropdown.currentText()  
 *self*.display\_data\_callback(search\_query, sub\_query, main\_column, sub\_column, filter\_type)  
  
 def clear\_filters(*self*):  
 *self*.search\_entry.setText("")  
 *self*.sub\_search\_entry.setText("")  
 *self*.column\_dropdown.setCurrentText("All Columns")  
 *self*.sub\_search\_column\_dropdown.setCurrentText("All Columns")  
 *self*.filter\_dropdown.setCurrentText("Contains")  
 *self*.display\_data\_callback(  
 search\_query="",  
 sub\_query="",  
 main\_column="All Columns",  
 sub\_column="All Columns",  
 filter\_type="Contains"  
 )  
  
 def update\_columns(*self*, *columns*):  
 *self*.column\_dropdown.clear()  
 *self*.column\_dropdown.addItem("All Columns")  
 *self*.column\_dropdown.addItems(*columns*)  
 *self*.sub\_search\_column\_dropdown.clear()  
 *self*.sub\_search\_column\_dropdown.addItem("All Columns")  
 *self*.sub\_search\_column\_dropdown.addItems(*columns*)