SALESFORCE CRM PROJECT

// PROJECT OVERVIEW STARTS //

1. Project Overview

DESCRIPTION

The Salesforce CRM Project is a Customer Relationship Management system designed to manage customer data, sales opportunities, and related operations. It leverages Flask for web services, SQLAlchemy for ORM, and various utility packages to support email notifications, logging, and database interactions.

GOALS

- Manage customer accounts and opportunities.
- Facilitate email notifications for various system events.
- Log application events for monitoring and debugging.
- Execute SQL queries for database management and reporting.

// PROJECT OVERVIEW END //

// PROJECT ARCHITECTURE STARTS //

2. Project Architecture

SALESFORCE CRM PROJECT

```
|
├── apps/
|   └── app.py
|
├── db_connections/
|   └── configurations.py
|
├── email_setup/
|   ├── email_config.py
|   └── email_operations.py
|
├── logging_package/
|   ├── logging_module.py
|   └── logging_utility.py
|
├── sql_scripts/
|   └── queries.sql
|
├── user_models/
|   └── tables.py
```

```
|
├── utilities/
|   └── reusables.py
|
└── requirements.txt
```

APPLICATION STRUCTURE

- **`apps/`**: Contains core application logic.
  - **`app.py`**: Entry point of the application; defines routes and application initialization.
- **`db_connections/`**: Manages database connections and configurations.
  - **`configurations.py`**: Contains database URL and configuration settings.
- **`email_setup/`**: Handles email notifications.
  - **`email_config.py`**: Email configuration and template settings.
  - **`email_operations.py`**: Functions for sending success, failure, and opportunity-related emails.
- **`logging_package/`**: Provides logging functionalities.
  - **`logging_module.py`**: Logging configuration.
  - **`logging_utility.py`**: Utility functions for logging messages.
- **`sql_scripts/`**: Contains SQL queries for database operations.
  - **`queries.sql`**: SQL scripts for performing various queries.
- **`user_models/`**: Defines SQLAlchemy models.
  - **`tables.py`**: Contains models for database tables such as `Account`, `Dealer`, and `Opportunity`.
- **`utilities/`**: Provides utility functions and reusable code.
  - **`reusables.py`**: Includes functions for validation, formatting, and other utility tasks.
- **`requirements.txt`**: Lists the project's dependencies.

// PROJECT OVERVIEW END //

// APPS START //

apps package / :

APP.PY

In this module all the api's which are required for the accounts table, dealers table and opportunity table are created so that users can use them.

Below are the required imports for the proper working of the api's and the imports are classified based on standard libraries, third party, project specific imports.

```python
# Standard library imports
import uuid  # For generating universally unique identifiers (UUIDs)
from datetime import datetime  # For date and time manipulations

# Third-Party/External Library Imports
```

```python
import pytz  # For timezone handling and conversions
from flask import Flask, request, jsonify  # For building web applications and handlir
from sqlalchemy.exc import SQLAlchemyError  # For handling SQLAlchemy database excepti

# # # Project-Specific Imports ********
# Database Session
from db_connections.configurations import session  # For managing database session cor

# Email Operations
from email_setup.email_operations import (  # Email notifications
    notify_success,
    notify_failure,
    notify_opportunity_details,
    notify_opportunity_update_success
)

# Logging Utility
from logging_package.logging_utility import (  # Logging information, errors, and debu
    log_info,
    log_error,
    log_debug
)

# User Models
from user_models.tables import Account, Dealer, Opportunity  # Database models for acc

# Utility Functions
from utilities.reusables import (  # Reusable utility functions for validations and cc
    get_currency_conversion,
    get_opportunity_stage,
    validate_stage,
    validate_probability,
    parse_date,
    validate_positive_number
)
```

This will creates the Flask app instance for our apis.

```python
# Create Flask app instance
app = Flask(__name__)
```

//

```python
# ----------------------------------------------- ACCOUNT TABLE ----------------------
```

//

BELOW ARE THE API'S FOR ACCOUNT TABLE.

ADD ACCOUNT API :

This is used to add new accounts to the account table using POST method.

Method - POST

Request Body:

- **`account_id`** (string) - The unique identifier for the account (mandatory).
- **`account_name`** (string) - The name of the account (mandatory).

```
@app.route('/add-account', methods=['POST'])
def add_account():
    """
    Adds new accounts to account table
    :return: JSON response with email notifications
    """
    log_info("Received request to add new account")
    try:
        payload = request.get_json()
        log_debug(f"Request payload: {payload}")

        if not payload or 'account_id' not in payload or 'account_name' not in payloac
            error_message = "Invalid input data. 'account_id' and 'account_name' are r
            log_error(error_message)
            notify_failure("Add Account Failed", error_message)
            return jsonify({"error": error_message}), 400

        new_account = Account(
            account_id=payload['account_id'],
            account_name=payload['account_name']
        )

        session.add(new_account)
        session.commit()
        log_info(f"Account added successfully: {payload['account_id']}")

        success_message = (f"Account added successfully.\nAccount ID: {payload['accour
                           f"Account Name: {payload['account_name']}")
        notify_success("Add Account Successful", success_message)

        return jsonify({"message": "Account added successfully", "account_id": payloac
                        "account_name": payload['account_name']}), 201

    except SQLAlchemyError as e:
        session.rollback()
        error_message = f"Error inserting account: {str(e)}"
        log_error(error_message)
        notify_failure("Add Account Failed", error_message)
        return jsonify({"error": "Internal server error", "details": error_message}),
    except Exception as e:
        error_message = f"Unexpected error: {str(e)}"
        log_error(error_message)
        notify_failure("Add Account Failed", error_message)
        return jsonify({"error": "Internal server error", "details": error_message}),
    finally:
```

```
        log_info("End of add_account function")
```

GET ALL ACCOUNTS API :

This api is used to fetch all accounts from the database using GET METHOD

Method - GET

Query Parameters:

None. This API fetches all dealers from the account table.

```
@app.route('/get-all-accounts', methods=['GET'])
def get_all_accounts():
    """
    Fetches all accounts from the account table.
    :return: JSON response with account details and total count, and sends an email nc
    """
    log_info("Received request to get all accounts")
    try:
        # Fetch accounts from the database
        accounts = session.query(Account).all()
        total_count = len(accounts)
        log_info(f"Fetched {total_count} accounts")

        accounts_list = []
        for account in accounts:
            account_dict = account.account_serialize_to_dict()

            # Format currency_conversions
            if isinstance(account_dict.get('currency_conversions'), str):
                currency_conversions = {}
                conversions = account_dict['currency_conversions'].strip().split('\n')
                for conversion in conversions:
                    if conversion:
                        currency, value = conversion.split(': ', 1)
                        currency_conversions[currency] = value
                account_dict['currency_conversions'] = currency_conversions

            accounts_list.append(account_dict)

        # Format account details for response
        account_details = "\n".join(
            [f"Account ID: {account['account_id']}\n"
             f"Account Name: {account['account_name']}\n"
             f"Currency Conversions:\n" +
             "\n".join(
                 [f"{currency}: {value}" for currency, value in account.get('currency_
             for account in accounts_list]
        )

        # Construct success message
```

```
        success_message = (
            f"Successfully retrieved Total {total_count} accounts.\n\n"
            f"Account Details:\n****************************************\n{accour
            f"\nTotal count of accounts: {total_count}"
        )

        # Send email notification
        notify_success("Get All Accounts Successful", success_message)

        # Return JSON response
        return jsonify({"Accounts": accounts_list, "Total count of accounts": total_co

    except Exception as e:
        # Handle exception
        error_message = f"Error in fetching accounts: {str(e)}"
        log_error(error_message)
        notify_failure("Get Accounts Failed", error_message)
        return jsonify({"error": "Internal server error", "details": error_message}),
    finally:
        log_info("End of get_all_accounts function")
```

GET SINGLE ACCOUNT API :

This api is used to get single account details from the data base by taking account id as a query parameters in postman.

Method - GET

Query Parameters:

- **`account_id`** (string) - The ID of the account to be fetched (mandatory).

```
@app.route('/get-single-account', methods=['GET'])
def get_single_account():
    """
    Fetched single account details
    :return: JSON response with email notifications
    """
    log_info(f"Received request to get account details with account id {'account_id'}'
    try:
        # Fetch account_id from query parameters
        accountid = request.args.get('account_id')
        log_debug(f"Account ID fetched is: {accountid}")

        # Check if account_id is provided
        if not accountid:
            error_message = "Account ID not provided or invalid. Please provide a vali
            log_error(error_message)
            notify_failure("Get Single Account Failed", error_message)  # Send email f
            return jsonify({"error": error_message}), 400

        # Fetch the account from the database
```

```
        account = session.query(Account).filter_by(account_id=accountid).first()

        if not account:
            error_message = f"Account not found: {accountid}"
            log_error(error_message)
            notify_failure("Get Single Account Failed", error_message)  # Send email f
            return jsonify({"error": "Account not found"}), 404

        log_info(f"Fetched account: {accountid}")

        # Serialize account data to dictionary
        account_details = account.account_serialize_to_dict()

        # Prepare success message
        success_message = (f"Successfully fetched single account details - "
                           f"\n\nAccount ID: {account_details['account_id']}, "
                           f"\nName: {account_details['account_name']}")

        # Send email notification with the account details
        notify_success("Get Single Account Success", success_message)

        # Return response with serialized account details
        return jsonify({"Account": account_details, "Message": "Single Account Details

    except Exception as e:
        error_message = f"Error in fetching account: {str(e)}"
        log_error(error_message)
        notify_failure("Get Single Account Failed", error_message)  # Send email for f
        return jsonify({"error": "Internal server error", "details": error_message}),

    finally:
        log_info("End of get_single_account function")
```

UPDATE ACCOUNT API :

This api is used to update the account table based on account id using PUT METHOD.

Method - PUT

Request Body:

The request body should contain the following fields in JSON format:

- **`account_id`** (string) - The ID of the account to be updated (mandatory).
- **`account_name`** (string) - The new name of the account (mandatory).

```
@app.route('/update-account', methods=['PUT'])
def update_account():
    """
    Updates account name
    :return: JSON response with email notifications
    """
```

```
    log_info("Received request to update account details")
    try:
        data = request.get_json()
        account_id = data.get('account_id')
        new_account_name = data.get('account_name')

        log_debug(f"Account ID to update: {account_id}, New Name: {new_account_name}")

        if not account_id or not new_account_name:
            error_message = "Account ID and new Account Name must be provided."
            log_error(error_message)
            notify_failure("Update Account Failed", error_message)
            return jsonify({"error": error_message}), 400

        account = session.query(Account).filter_by(account_id=account_id).first()

        if not account:
            error_message = f"Account not found: {account_id}"
            log_error(error_message)
            notify_failure("Update Account Failed", error_message)
            return jsonify({"error": "Account not found"}), 404

        account.account_name = new_account_name
        session.commit()

        success_message = (f"Account ID: \n{account_id}\n\n"
                           f"Successfully updated with new name: \n{new_account_name}'
        log_info(success_message)
        notify_success("Update Account Success", success_message)

        return jsonify({
            "message": "Account details updated successfully.",
            "account_id": account_id,
            "new_account_name": new_account_name
        }), 200

    except Exception as e:
        error_message = f"Error in updating account: {str(e)}"
        log_error(error_message)
        notify_failure("Update Account Failed", error_message)
        return jsonify({"error": "Internal server error", "details": error_message}),

    finally:
        log_info("End of update_account function")
```

DELETE ACCOUNT :

This api is used to delete the account based on the given account id using DELETE METHOD.

Method - DELETE

Query Parameters:

- `account_id` (string) - The ID of the account to be deleted (mandatory).

```
@app.route('/delete-account', methods=['DELETE'])
def delete_account():
    """
    Deletes account based on account id
    :return: JSON response with email notifications
    """
    log_info("Received request to delete account")
    try:
        account_id = request.args.get('account_id')
        log_debug(f"Account ID to delete: {account_id}")

        if not account_id:
            error_message = "Account ID must be provided."
            log_error(error_message)
            notify_failure("Delete Account Failed", error_message)
            return jsonify({"error": error_message}), 400

        account = session.query(Account).filter_by(account_id=account_id).first()

        if not account:
            error_message = f"Account not found: {account_id}"
            log_error(error_message)
            notify_failure("Delete Account Failed", error_message)
            return jsonify({"error": "Account not found"}), 404

        session.delete(account)
        session.commit()

        success_message = (f"Account ID: {account_id}\n"
                           f"Successfully account deleted. Details:\n"
                           f"Account ID: {account.account_id}\n"
                           f"Account Name: {account.account_name}")
        log_info(success_message)
        notify_success("Delete Account Success", success_message)

        return jsonify({
            "message": "Account successfully deleted.",
            "deleted_account_id": account_id,
            "account_name": account.account_name
        }), 200

    except Exception as e:
        error_message = f"Error in deleting account: {str(e)}"
        log_error(error_message)
        notify_failure("Delete Account Failed", error_message)
        return jsonify({"error": "Internal server error", "details": error_message}),

    finally:
        log_info("End of delete_account function")
```

//

```
# ---------------------------------------------- DEALER TABLE ----------------------
```

//

BELOW ARE THE API'S FOR DEALER TABLE.

ADD DEALER API:

This will adds new dealer to the dealer table using POST METHOD.

Method - POST

Request Body:

The request body should be a JSON object containing the following fields:

- `dealer_code` (string): The unique code representing the dealer.
- `opportunity_owner` (string): The name of the opportunity owner for the dealer.

```
@app.route('/add-dealer', methods=['POST'])
def add_dealer():
    """
    Adds new Dealer to the dealer table
    :return: JSON response with email notifications
    """
    log_info("Received request to add new dealer")
    try:
        payload = request.get_json()
        log_debug(f"Request payload: {payload}")

        if not payload or 'dealer_code' not in payload or 'opportunity_owner' not in p
            error_message = "Invalid input data. 'dealer_code' and 'opportunity_owner'
            log_error(error_message)
            notify_failure("Add Dealer Failed", error_message)
            return jsonify({"error": error_message}), 400

        new_dealer = Dealer(
            dealer_code=payload['dealer_code'],
            opportunity_owner=payload['opportunity_owner']
        )

        session.add(new_dealer)
        session.commit()
        log_info(f"Dealer added successfully: {new_dealer.dealer_id}")
```

```
        success_message = (f"Dealer added successfully.\n\n"
                           f"Dealer ID: {new_dealer.dealer_id}\n\n"
                           f"Dealer Code: {payload['dealer_code']}\n\n"
                           f"Opportunity Owner: {payload['opportunity_owner']}")
        notify_success("Add Dealer Successful", success_message)

        return jsonify({
            "message": "Dealer added successfully",
            "dealer_id": new_dealer.dealer_id,
            "dealer_code": payload['dealer_code'],
            "opportunity_owner": payload['opportunity_owner']
        }), 201

    except SQLAlchemyError as e:
        session.rollback()
        error_message = f"Error inserting dealer: {str(e)}"
        log_error(error_message)
        notify_failure("Add Dealer Failed", error_message)
        return jsonify({"error": "Internal server error", "details": error_message}),
    except Exception as e:
        error_message = f"Unexpected error: {str(e)}"
        log_error(error_message)
        notify_failure("Add Dealer Failed", error_message)
        return jsonify({"error": "Internal server error", "details": error_message}),
    finally:
        log_info("End of add_dealer function")
```

GET ALL DEALERS API :

This will fetch all the dealers from the dealers table using GET METHOD .

Method - GET

Query Parameters:

None. This API fetches all dealers from the `dealers` table.

```
@app.route('/get-all-dealers', methods=['GET'])
def get_all_dealers():
    log_info("Received request to get all dealers")
    try:
        # Fetch all dealers from the database
        dealers = session.query(Dealer).all()

        if not dealers:
            log_info("No dealers found.")
            return jsonify({"message": "No dealers found"}), 404

        # Serialize dealer data
        dealers_data = [dealer.dealer_serialize_to_dict() for dealer in dealers]
```

```
        # Prepare formatted success message for email
        success_message = f"Dealers retrieved successfully. Count: {len(dealers)}"
        log_info(success_message)
        notify_success("Get All Dealers Successful", success_message)

        # Return the response with dealer data
        return jsonify({
            "message": "Dealers retrieved successfully.",
            "Total count": len(dealers),
            "dealers": dealers_data
        }), 200

    except SQLAlchemyError as e:
        session.rollback()  # Rollback transaction in case of error
        error_message = f"Error retrieving dealers: {str(e)}"
        log_error(error_message)
        notify_failure("Get All Dealers Failed", error_message)
        return jsonify({"error": "Internal server error", "details": error_message}),
    except Exception as e:
        error_message = f"Unexpected error: {str(e)}"
        log_error(error_message)
        notify_failure("Get All Dealers Failed", error_message)
        return jsonify({"error": "Internal server error", "details": error_message}),
    finally:
        log_info("End of get_all_dealers function")
```

GET PARTICULAR DEALERS API :

This will fetch the details of dealer based on dealer id or dealer code or opportunity owner when passed as query paramaters in postman.

Method - GET

Query Parameters:

- **`dealer_id`** (optional): The unique identifier for a dealer.
- **`dealer_code`** (optional): The code associated with the dealer.
- **`opportunity_owner`** (optional): The owner of the opportunity related to the dealer.

At least one of these query parameters must be provided to fetch dealer details.

```
@app.route('/get-particular-dealers', methods=['GET'])
def get_particular_dealers():
    """
    Fetches particular dealers based on dealer id, dealer code, opportunity owner
    :return: JSON response with email notifications
    """
    log_info("Received request to get particular dealers by parameters")
    try:
        dealer_id = request.args.get('dealer_id')
```

```
        dealer_code = request.args.get('dealer_code')
        opportunity_owner = request.args.get('opportunity_owner')

        log_debug(
            f"Search parameters - dealer_id: {dealer_id}, dealer_code: {dealer_code},
            f"opportunity_owner: {opportunity_owner}")

        if not dealer_id and not dealer_code and not opportunity_owner:
            error_message = "At least one of 'dealer_id', 'dealer_code', or 'opportuni
            log_error(error_message)
            notify_failure("Get Dealers Failed", error_message)
            return jsonify({"error": error_message}), 400

        query = session.query(Dealer)

        if dealer_id:
            query = query.filter_by(dealer_id=dealer_id)
        if dealer_code:
            query = query.filter_by(dealer_code=dealer_code)
        if opportunity_owner:
            query = query.filter_by(opportunity_owner=opportunity_owner)

        dealers = query.all()

        if not dealers:
            error_message = "No dealers found with the provided parameters."
            log_info(error_message)
            notify_success("Get Dealers", error_message)
            return jsonify({"message": error_message}), 404

        dealers_data = [dealer.dealer_serialize_to_dict() for dealer in dealers]

        formatted_dealers_info = "\n".join([
            f"Dealer ID: {dealer['dealer_id']}\n"
            f"Dealer Code: {dealer['dealer_code']}\n"
            f"Opportunity Owner: {dealer['opportunity_owner']}\n"
            "-------------------------"
            for dealer in dealers_data
        ])
        success_message = (f"Retrieved {len(dealers)} dealer(s) successfully.\n\n"
                           f"Dealer Details:\n{formatted_dealers_info}")
        log_info(success_message)
        notify_success("Get Dealers Successful", success_message)

        # Return the response with dealer data
        return jsonify({
            "message": f"Retrieved Total {len(dealers)} dealer(s) successfully.",
            "dealers": dealers_data
        }), 200

    except SQLAlchemyError as e:
        session.rollback()
        error_message = f"Error retrieving dealers: {str(e)}"
        log_error(error_message)
        notify_failure("Get Dealers Failed", error_message)
```

```
        return jsonify({"error": "Internal server error", "details": error_message}),
    except Exception as e:
        error_message = f"Unexpected error: {str(e)}"
        log_error(error_message)
        notify_failure("Get Dealers Failed", error_message)
        return jsonify({"error": "Internal server error", "details": error_message}),
    finally:
        log_info("End of get_dealers function")
```

UPDATE DEALER API :

This api is used to update the dealer details using dealer id .

Method : PUT

Request Body:

The request body should contain JSON data with the dealer details that need to be updated. The `dealer_id` is mandatory for identifying the dealer to be updated, while `dealer_code` and `opportunity_owner` are optional fields that can be updated.

Request Parameters:

- `dealer_id` (required): The unique identifier for the dealer to be updated.
- `dealer_code` (optional): The new code for the dealer.
- `opportunity_owner` (optional): The updated opportunity owner for the dealer.

```
@app.route('/update-dealer', methods=['PUT'])
def update_dealer():
    """
    Updates dealer based on dealer id
    :return: JSON response with email notifications
    """
    log_info("Received request to update a dealer")
    try:
        data = request.json
        dealer_id = data.get('dealer_id')
        dealer_code = data.get('dealer_code')
        opportunity_owner = data.get('opportunity_owner')

        log_debug(
            f"Data received for update: dealer_id={dealer_id}, dealer_code={dealer_coc
            f"opportunity_owner={opportunity_owner}")

        if not dealer_id:
            error_message = "Dealer ID must be provided to update a dealer."
            log_error(error_message)
            notify_failure("Update Dealer Failed", error_message)
            return jsonify({"error": error_message}), 400

        dealer = session.query(Dealer).filter_by(dealer_id=dealer_id).first()
```

```
        if not dealer:
            error_message = f"No dealer found with dealer_id: {dealer_id}"
            log_error(error_message)
            notify_failure("Update Dealer Failed", error_message)
            return jsonify({"error": "Dealer not found"}), 404

        if dealer_code:
            dealer.dealer_code = dealer_code
        if opportunity_owner:
            dealer.opportunity_owner = opportunity_owner

        session.commit()

        updated_dealer_data = dealer.dealer_serialize_to_dict()

        success_message = (f"Dealer updated successfully.\n\n"
                           f"Updated Dealer ID: {updated_dealer_data['dealer_id']}\n"
                           f"Updated Dealer Code: {updated_dealer_data['dealer_code']}
                           f"Updated Opportunity Owner: {updated_dealer_data['opportur
        log_info(success_message)
        notify_success("Dealer Updated Successfully", success_message)

        return jsonify({
            "message": "Dealer updated successfully.",
            "dealer": updated_dealer_data
        }), 200

    except SQLAlchemyError as e:
        session.rollback()
        error_message = f"Error updating dealer: {str(e)}"
        log_error(error_message)
        notify_failure("Update Dealer Failed", error_message)
        return jsonify({"error": "Internal server error", "details": error_message}),
    except Exception as e:
        error_message = f"Unexpected error: {str(e)}"
        log_error(error_message)
        notify_failure("Update Dealer Failed", error_message)
        return jsonify({"error": "Internal server error", "details": error_message}),
    finally:
        log_info("End of update_dealer function")
```

DELETE SINGLE DEALER API :

This api is used to delete single/particular dealer based on dealer id or dealer code or opportunity owner.

Method - DELETE

Query Parameters:

- **`dealer_id`** (optional): Unique identifier for the dealer.
- **`dealer_code`** (optional): Code associated with the dealer.

- **`opportunity_owner`** (optional): Opportunity owner associated with the dealer.

CODE :

```python
@app.route('/delete-single-dealer', methods=['DELETE'])
def delete_single_dealer():
    """
    Deletes the single dealer based on dealer id/ dealer code/ opportunity owner
    :return: JSON response with email notifications
    """
    log_info("Received request to delete a dealer")
    try:
        dealer_id = request.args.get('dealer_id')
        dealer_code = request.args.get('dealer_code')
        opportunity_owner = request.args.get('opportunity_owner')
        log_debug(f"Dealer ID: {dealer_id}, Dealer Code: {dealer_code}, Opportunity Ow

        if not dealer_id and not dealer_code and not opportunity_owner:
            error_message = "At least one of 'dealer_id', 'dealer_code', or 'opportuni
            log_error(error_message)
            notify_failure("Delete Dealer Failed", error_message)
            return jsonify({"error": error_message}), 400

        query = session.query(Dealer)
        if dealer_id:
            query = query.filter_by(dealer_id=dealer_id)
        if dealer_code:
            query = query.filter_by(dealer_code=dealer_code)
        if opportunity_owner:
            query = query.filter_by(opportunity_owner=opportunity_owner)

        dealers_to_delete = query.first()

        if not dealers_to_delete:
            error_message = "Dealer not found with the given criteria."
            log_error(error_message)
            notify_failure("Delete Dealer Failed", error_message)
            return jsonify({"error": error_message}), 404

        for dealer in dealers_to_delete:
            session.delete(dealer)

        session.commit()
        success_message = f"Deleted {len(dealers_to_delete)} dealer(s) successfully."
        log_info(success_message)
        notify_success("Delete Dealer Successful", success_message)

        return jsonify({"message": success_message}), 200

    except SQLAlchemyError as e:
        session.rollback()
        error_message = f"Error deleting dealer: {str(e)}"
        log_error(error_message)
        notify_failure("Delete Dealer Failed", error_message)
        return jsonify({"error": "Internal server error", "details": error_message}),
```

```
    except Exception as e:
        error_message = f"Unexpected error: {str(e)}"
        log_error(error_message)
        notify_failure("Delete Dealer Failed", error_message)
        return jsonify({"error": "Internal server error", "details": error_message}),
    finally:
        log_info("End of delete_dealer function")
```

DELETE ALL DEALERS API :

This api is used to delete all dealers based on dealer id/ dealer code or opportunity owner.

Query Parameters:

- **`dealer_id`** (optional): Unique identifier for the dealer.
- **`dealer_code`** (optional): Code associated with the dealer.
- **`opportunity_owner`** (optional): Opportunity owner associated with the dealer.

CODE :

```
@app.route('/delete-all-dealers', methods=['DELETE'])
def delete_all_dealers():
    """
    Deletes all dealers based on dealer id/ dealer code/ opportunity owner
    :return: JSON response with email notifications
    """
    log_info("Received request to delete dealers")
    try:
        dealer_id = request.args.get('dealer_id')
        dealer_code = request.args.get('dealer_code')
        opportunity_owner = request.args.get('opportunity_owner')

        if not dealer_id and not dealer_code and not opportunity_owner:
            error_message = "At least one of 'dealer_id', 'dealer_code', or 'opportuni
            log_error(error_message)
            return jsonify({"error": error_message}), 400

        query = session.query(Dealer)

        if dealer_id:
            query = query.filter(Dealer.dealer_id == dealer_id)
        if dealer_code:
            query = query.filter(Dealer.dealer_code == dealer_code)
        if opportunity_owner:
            query = query.filter(Dealer.opportunity_owner == opportunity_owner)

        dealers_to_delete = query.all()

        if not dealers_to_delete:
            error_message = "No dealers found with the given criteria."
```

```
            log_error(error_message)
            return jsonify({"error": error_message}), 404

        for dealer in dealers_to_delete:
            session.delete(dealer)
        session.commit()

        success_message = f"Deleted {len(dealers_to_delete)} dealer(s) successfully."
        log_info(success_message)
        notify_success("Delete Dealers Successful", success_message)

        return jsonify({"message": success_message}), 200

    except SQLAlchemyError as e:
        session.rollback()
        error_message = f"Error deleting dealers: {str(e)}"
        log_error(error_message)
        notify_failure("Delete Dealers Failed", error_message)
        return jsonify({"error": "Internal server error", "details": error_message}),

    except Exception as e:
        error_message = f"Unexpected error: {str(e)}"
        log_error(error_message)
        notify_failure("Delete Dealers Failed", error_message)
        return jsonify({"error": "Internal server error", "details": error_message}),

    finally:
        log_info("End of delete_dealers function")
```

//

```
# ----------------------------------------------- OPPORTUNITY TABLE ------------------
```

//

BELOW ARE THE API'S FOR OPPORTUNITY TABLE.

API ENDPOINT: `/NEW-CUSTOMER`

METHOD: `POST`

Purpose:

This API creates a new customer in the opportunity table. It generates a unique opportunity ID, validates the request payload, checks for account and dealer existence, handles the currency conversion of the amount, and commits the new customer (opportunity) to the database. It also logs information and sends email notifications upon success or failure.

Request Payload:

The API expects the payload in JSON format. The key fields include:

- `opportunity_name` (string): Name of the opportunity.
- `account_name` (string): Name of the associated account.
- `dealer_id` (string): Dealer ID.
- `dealer_code` (string): Dealer code.
- `opportunity_owner` (string): Name of the opportunity owner.
- `close_date` (string): Expected close date (format: `YYYY-MM-DD HH:MM:SS`).
- `probability` (float): Probability of the deal closing (used to determine the stage).
- `amount` (float): Opportunity amount.
- `description` (string): Description of the opportunity.
- `next_step` (string): Next step in the opportunity process.

Process Flow:

- **Request Logging:**
  - Logs the receipt of the request and the received payload for debugging.
- **UUID Generation:**
  - Generates a unique `opportunity_id` using `uuid.uuid1()`.
- **Created Date:**
  - Adds the current timestamp as `created_date` using the Asia/Kolkata timezone.
- **Account Validation:**
  - Fetches the account from the `Account` table based on `account_name`.
  - If the account doesn't exist, the API logs an error, sends a failure email notification, and returns a `400` status with an appropriate error message.
- **Dealer Validation:**
  - Retrieves the dealer details (`dealer_id`, `dealer_code`, `opportunity_owner`) from the `Dealer` table.
  - If no dealer matches the provided details, the API logs an error, sends a failure email, and returns a `400` status.
- **Close Date Validation:**
  - Converts the `close_date` from string format to a `datetime` object. If the date format is invalid, it logs an error, sends an email notification, and returns a `400` status.
- **Opportunity Stage Calculation:**
  - The API calls a utility function `get_opportunity_stage(probability)` to determine the opportunity stage based on the `probability` value.
- **Currency Conversion:**
  - Calls a utility function `get_currency_conversion(amount)` to convert the opportunity amount to various currencies like USD, EUR, etc.
- **Opportunity Creation:**
  - Creates a new `Opportunity` object with the validated data and adds it to the database. This includes attributes

such as `opportunity_name`, `amount`, `created_date`, and currency conversion fields.

- **Success Response:**
  - If everything is successful, the API commits the transaction, logs success, sends a success email notification with detailed information, and returns a `201` status along with the customer details.
- **Error Handling:**
  - Any exceptions (SQL errors or general errors) are caught, logged, and notified via email, and the API returns a `500` status with an internal server error message.

CODE :

```
@app.route('/new-customer', methods=["POST"])
def create_new_customer():
    """
    Creates a new customer in the opportunity table.
    :return: JSON response with email notifications and detailed customer information.
    """
    log_info("Received request to create new customer")
    try:
        payload = request.get_json()
        log_debug(f"Request payload: {payload}")

        opportunity_id = str(uuid.uuid1())
        log_info(f"Generated opportunity_id: {opportunity_id}")

        created_date = datetime.now(pytz.timezone('Asia/Kolkata'))
        payload.update({'created_date': created_date, 'opportunity_id': opportunity_id

        account_name = payload.get("account_name")
        account = session.query(Account).filter_by(account_name=account_name).first()

        if not account:
            error_message = f"Account does not exist: {account_name}"
            log_error(error_message)
            detailed_error_message = (f"Failed to create customer due to missing accou
                                      f"Account Name: {account_name}")
            notify_failure("Customer Creation Failed", detailed_error_message)
            return jsonify({"error": error_message}), 400
        log_info(f"Account found: {account_name}")

        dealer_id = payload.get("dealer_id")
        dealer_code = payload.get("dealer_code")
        opportunity_owner = payload.get("opportunity_owner")

        dealer = session.query(Dealer).filter_by(
            dealer_id=dealer_id,
            dealer_code=dealer_code,
            opportunity_owner=opportunity_owner
        ).first()

        if not dealer:
            error_message = f"Invalid dealer details: {dealer_id}, {dealer_code}, {opp
```

```
            log_error(error_message)
            detailed_error_message = (f"Failed to create customer due to invalid deale
                                      f"Dealer ID: {dealer_id}\n"
                                      f"Dealer Code: {dealer_code}\n"
                                      f"Opportunity Owner: {opportunity_owner}")
            notify_failure("Customer Creation Failed", detailed_error_message)
            return jsonify({"error": error_message}), 400
        log_info(f"Dealer found: {dealer_id}, {dealer_code}, {opportunity_owner}")

        close_date_str = payload.get("close_date")
        if close_date_str:
            try:
                close_date = datetime.strptime(close_date_str, "%Y-%m-%d %H:%M:%S")
            except ValueError as e:
                error_message = f"Invalid date format for close_date: {str(e)}"
                log_error(error_message)
                detailed_error_message = f"Failed to create customer due to invalid da
                notify_failure("Customer Creation Failed", detailed_error_message)
                return jsonify({"error": error_message}), 400
        else:
            close_date = None

        probability = payload.get("probability")
        if probability is not None:
            try:
                stage = get_opportunity_stage(probability)
            except ValueError as e:
                error_message = f"Invalid probability value: {str(e)}"
                log_error(error_message)
                notify_failure("Customer Creation Failed", error_message)
                return jsonify({"error": error_message}), 400
        else:
            stage = payload.get("stage", "Unknown")

        amount = payload.get("amount")
        if amount:
            currency_conversions = get_currency_conversion(amount)
        else:
            currency_conversions = {}

        new_opportunity = Opportunity(
            opportunity_id=opportunity_id,
            opportunity_name=payload.get("opportunity_name"),
            account_name=account_name,
            close_date=close_date,
            amount=amount,
            description=payload.get("description"),
            dealer_id=dealer_id,
            dealer_code=dealer_code,
            stage=stage,
            probability=probability,
            next_step=payload.get("next_step"),
            created_date=created_date,
            usd=currency_conversions.get("USD"),
            aus=currency_conversions.get("AUD"),
```

```
            cad=currency_conversions.get("CAD"),
            jpy=currency_conversions.get("JPY"),
            eur=currency_conversions.get("EUR"),
            gbp=currency_conversions.get("GBP"),
            cny=currency_conversions.get("CNY"),
            amount_in_words=str(amount)
        )

        session.add(new_opportunity)
        session.commit()
        log_info(f"Opportunity created successfully: {opportunity_id}")

        customer_details = new_opportunity.serialize_to_dict()

        formatted_currency_conversions = "\n".join(f"{key}: {value}" for key, value ir

        success_message = (f"Customer created successfully.\n\n\n"
                           f"Opportunity ID: {opportunity_id}\n\n"
                           f"Opportunity Name: {payload.get('opportunity_name')}\n\n"
                           f"Account Name: {account_name}\n\n"
                           f"Close Date: {close_date.strftime('%Y-%m-%d %H:%M:%S') if
                           f"Amount: {payload.get('amount')}\n\n"
                           f"Stage: {stage}\n\n"
                           f"Probability: {payload.get('probability')}\n\n"
                           f"Currency Conversions:\n{formatted_currency_conversions}\r
                           f"Created Date: {created_date.strftime('%Y-%m-%d %H:%M:%S')
        notify_success("Customer Creation Successful", success_message)

        return jsonify({
            "message": "Created successfully",
            "customer_details": customer_details
        }), 201

    except Exception as e:
        error_message = f"Error in creating customer: {str(e)}"
        log_error(error_message)
        detailed_error_message = f"Failed to create customer due to an internal server
        notify_failure("Customer Creation Failed", detailed_error_message)
        return jsonify({"error": "Internal server error", "details": error_message}),

    finally:
        log_info("End of create_new_customer function")
```

API ENDPOINT: `/GET-OPPORTUNITIES`

This API fetches opportunities from the `opportunity` table based on various query parameters.

Method - GET

Query Parameters:

- **`opportunity_id`** (string) - Filter opportunities by ID.
- **`opportunity_name`** (string) - Filter opportunities by name (partial match).
- **`account_name`** (string) - Filter opportunities by account name (partial match).
- **`stage`** (string) - Filter opportunities by stage.
- **`probability_min`** (integer) - Minimum probability filter (0-100).
- **`probability_max`** (integer) - Maximum probability filter (0-100).
- **`created_date_start`** (string, date format) - Start date for filtering created date.
- **`created_date_end`** (string, date format) - End date for filtering created date.

Fetches the opportunities based on the provided parameters and they are optional, if no parameters are provided then fetches all opportunities.

CODE :

```
@app.route('/get-opportunities', methods=['GET'])
def get_opportunities():
    """
    Fetches opportunities from the opportunity table based on query parameters.
    :return: JSON response with filtered opportunity details and total count.
    """
    log_info("Received request to get opportunities with query parameters")

    try:
        opportunity_id = request.args.get('opportunity_id')
        opportunity_name = request.args.get('opportunity_name')
        account_name = request.args.get('account_name')
        stage = request.args.get('stage')
        probability_min = request.args.get('probability_min', type=int)
        probability_max = request.args.get('probability_max', type=int)
        created_date_start = request.args.get('created_date_start')
        created_date_end = request.args.get('created_date_end')

        if created_date_start:
            created_date_start = parse_date(created_date_start)
        if created_date_end:
            created_date_end = parse_date(created_date_end)

        if probability_min is not None and not validate_probability(probability_min):
            raise ValueError(f"Invalid minimum probability: {probability_min}. Must be

        if probability_max is not None and not validate_probability(probability_max):
            raise ValueError(f"Invalid maximum probability: {probability_max}. Must be

        if probability_min is not None and probability_max is not None and probability
            raise ValueError("Minimum probability cannot be greater than maximum proba

        if stage:
            stage = validate_stage(stage)

        query = session.query(Opportunity)
```

```
        if opportunity_id:
            query = query.filter(Opportunity.opportunity_id == opportunity_id)
        if opportunity_name:
            if len(opportunity_name) > 255:
                raise ValueError("Opportunity name is too long. Maximum length is 255
            query = query.filter(Opportunity.opportunity_name.like(f'%{opportunity_nam
        if account_name:
            if len(account_name) > 255:
                raise ValueError("Account name is too long. Maximum length is 255 char
            query = query.filter(Opportunity.account_name.like(f'%{account_name}%'))
        if stage:
            query = query.filter(Opportunity.stage == stage)
        if probability_min is not None:
            query = query.filter(Opportunity.probability >= probability_min)
        if probability_max is not None:
            query = query.filter(Opportunity.probability <= probability_max)
        if created_date_start:
            query = query.filter(Opportunity.created_date >= created_date_start)
        if created_date_end:
            query = query.filter(Opportunity.created_date <= created_date_end)

        opportunities = query.all()
        total_count = len(opportunities)
        log_info(f"Fetched {total_count} opportunities based on query parameters")

        opportunities_list = [opportunity.serialize_to_dict() for opportunity in oppor

        notify_opportunity_details("Get Opportunities Successful", opportunities_list,

        return jsonify({"Opportunities": opportunities_list, "Total count of opportuni

    except ValueError as ve:
        error_message = f"Validation error: {str(ve)}"
        log_error(error_message)
        notify_failure("Get Opportunities Validation Failed", error_message)
        return jsonify({"error": "Bad Request", "details": error_message}), 400

    except SQLAlchemyError as sae:
        error_message = f"Database error: {str(sae)}"
        log_error(error_message)
        notify_failure("Get Opportunities Database Error", error_message)
        return jsonify({"error": "Internal server error", "details": error_message}),

    except Exception as e:
        error_message = f"Error in fetching opportunities: {str(e)}"
        log_error(error_message)
        notify_failure("Get Opportunities Failed", error_message)
        return jsonify({"error": "Internal server error", "details": error_message}),

    finally:
        log_info("End of get_opportunities function")
```

API ENDPOINT: `/UPDATE-OPPORTUNITY`

This API updates an existing opportunity record based on the provided `opportunity_id` and optional fields.

Method - PUT

Request Body:

The request body should be a JSON object with at least the `opportunity_id`, and can include the following fields:

- **`opportunity_id`** (string) - Required. The ID of the opportunity to update.
- **`opportunity_name`** (string) - The name of the opportunity.
- **`account_name`** (string) - The name of the associated account.
- **`close_date`** (string, date format) - The close date of the opportunity.
- **`amount`** (number) - The amount for the opportunity.
- **`description`** (string) - A description of the opportunity.
- **`dealer_id`** (string) - The ID of the dealer.
- **`dealer_code`** (string) - The code of the dealer.
- **`stage`** (string) - The stage of the opportunity.
- **`probability`** (number) - The probability of closing the opportunity (0-100).
- **`next_step`** (string) - The next step in the opportunity.
- **`amount_in_words`** (string) - The amount written in words.
- **`currency_conversions`** (object) - A dictionary with currency conversions (e.g., `{ "usd": 100, "eur": 90 }`).

CODE :

```
@app.route('/update-opportunity', methods=['PUT'])
def update_opportunity():
    """
    Update an existing Opportunity record.
    :return: JSON response indicating success or failure.
    """
    log_info("Received request to update opportunity")

    try:
        data = request.get_json()

        opportunity_id = data.get('opportunity_id')
        opportunity_name = data.get('opportunity_name')
        account_name = data.get('account_name')
        close_date = data.get('close_date')
        amount = data.get('amount')
        description = data.get('description')
        dealer_id = data.get('dealer_id')
        dealer_code = data.get('dealer_code')
        stage = data.get('stage')
        probability = data.get('probability')
```

```
        next_step = data.get('next_step')
        amount_in_words = data.get('amount_in_words')
        currency_conversions = data.get('currency_conversions', {})

        if not opportunity_id:
            raise ValueError("Opportunity ID is required.")

        if opportunity_name and len(opportunity_name) > 255:
            raise ValueError("Opportunity name is too long. Maximum length is 255 char

        if account_name and len(account_name) > 255:
            raise ValueError("Account name is too long. Maximum length is 255 characte

        if close_date:
            close_date = parse_date(close_date)

        if amount is not None and not validate_positive_number(amount):
            raise ValueError("Amount must be a positive number.")

        if probability is not None and not validate_probability(probability):
            raise ValueError("Probability must be between 0 and 100.")

        if stage:
            stage = validate_stage(stage)

        valid_currencies = ['usd', 'aus', 'cad', 'jpy', 'eur', 'gbp', 'cny']
        for currency in valid_currencies:
            if currency in currency_conversions:
                if not validate_positive_number(currency_conversions[currency]):
                    raise ValueError(f"Invalid value for currency conversion {currency

        opportunity = session.query(Opportunity).filter_by(opportunity_id=opportunity_
        if not opportunity:
            raise ValueError("Opportunity not found.")

        updated_fields = {}

        if opportunity_name:
            opportunity.opportunity_name = opportunity_name
            updated_fields['opportunity_name'] = opportunity_name
        if account_name:
            opportunity.account_name = account_name
            updated_fields['account_name'] = account_name
        if close_date:
            opportunity.close_date = close_date
            updated_fields['close_date'] = close_date
        if amount is not None:
            opportunity.amount = amount
            conversions = get_currency_conversion(amount)
            opportunity.usd = conversions.get('USD')
            opportunity.aus = conversions.get('AUD')
            opportunity.cad = conversions.get('CAD')
            opportunity.jpy = conversions.get('JPY')
            opportunity.eur = conversions.get('EUR')
            opportunity.gbp = conversions.get('GBP')
```

```
            opportunity.cny = conversions.get('CNY')
            updated_fields['amount'] = amount
            updated_fields['currency_conversions'] = conversions
        if description:
            opportunity.description = description
            updated_fields['description'] = description
        if dealer_id:
            opportunity.dealer_id = dealer_id
            updated_fields['dealer_id'] = dealer_id
        if dealer_code:
            opportunity.dealer_code = dealer_code
            updated_fields['dealer_code'] = dealer_code
        if stage:
            opportunity.stage = stage
            updated_fields['stage'] = stage
        if probability is not None:
            opportunity.probability = probability
            updated_fields['probability'] = probability
        if next_step:
            opportunity.next_step = next_step
            updated_fields['next_step'] = next_step
        if amount_in_words:
            opportunity.amount_in_words = amount_in_words
            updated_fields['amount_in_words'] = amount_in_words
        if currency_conversions:
            opportunity.usd = currency_conversions.get('usd')
            opportunity.aus = currency_conversions.get('aus')
            opportunity.cad = currency_conversions.get('cad')
            opportunity.jpy = currency_conversions.get('jpy')
            opportunity.eur = currency_conversions.get('eur')
            opportunity.gbp = currency_conversions.get('gbp')
            opportunity.cny = currency_conversions.get('cny')
            updated_fields['currency_conversions'] = currency_conversions

        session.commit()

        log_info(f"Opportunity with ID {opportunity_id} updated successfully.")

        notify_opportunity_update_success(
            "Update Opportunity Successful",
            {"opportunity_id": opportunity_id, "updated_fields": updated_fields}
        )

        return jsonify({
            "message": "Opportunity updated successfully.",
            "opportunity_id": opportunity_id,
            "updated_fields": updated_fields
        }), 200

    except ValueError as ve:
        error_message = f"Validation error: {str(ve)}"
        log_error(error_message)
        notify_failure("Update Opportunity Validation Failed", error_message)
        return jsonify({"error": "Bad Request", "details": error_message}), 400
```

```
    except SQLAlchemyError as sae:
        error_message = f"Database error: {str(sae)}"
        log_error(error_message)
        notify_failure("Update Opportunity Database Error", error_message)
        return jsonify({"error": "Internal server error", "details": error_message}),

    except Exception as e:
        error_message = f"Error updating opportunity: {str(e)}"
        log_error(error_message)
        notify_failure("Update Opportunity Failed", error_message)
        return jsonify({"error": "Internal server error", "details": error_message}),

    finally:
        log_info("End of update_opportunity function")
```

API ENDPOINT: `/DELETE-CUSTOMER`

This API deletes a customer record from the `Opportunity` table based on the provided query parameters.

Method - DELETE

Query Parameters:

At least one of the following query parameters is required:

- **`opportunity_id`** (string) - The ID of the opportunity to delete.
- **`account_name`** (string) - The name of the associated account.
- **`dealer_id`** (string) - The ID of the dealer.
- **`dealer_code`** (string) - The code of the dealer.
- **`opportunity_name`** (string) - The name of the opportunity.
- **`stage`** (string) - The stage of the opportunity.
- **`probability`** (integer) - The probability of the opportunity (0-100).
- **`close_date`** (string, date format) - The close date of the opportunity (`YYYY-MM-DD HH:MM:SS`).

CODE :

```
@app.route('/delete-customer', methods=["DELETE"])
def delete_customer():
    """
    Deletes a customer from the opportunity table based on query parameters.
    :return: JSON response with email notifications and result of deletion.
    """
    log_info("Received request to delete customer")
```

```
    try:
        opportunity_id = request.args.get("opportunity_id")
        account_name = request.args.get("account_name")
        dealer_id = request.args.get("dealer_id")
        dealer_code = request.args.get("dealer_code")
        opportunity_name = request.args.get("opportunity_name")
        stage = request.args.get("stage")
        probability = request.args.get("probability", type=int)
        close_date = request.args.get("close_date")

        if not any([opportunity_id, account_name, dealer_id, dealer_code, opportunity_
                    close_date]):
            error_message = ("At least one query parameter (opportunity_id, account_na
                             "opportunity_name, stage, probability, or close_date) is
            log_error(error_message)
            detailed_error_message = "Failed to delete customer due to missing query p
            notify_failure("Customer Deletion Failed", detailed_error_message)
            return jsonify({"error": error_message}), 400

        if close_date:
            try:
                close_date = datetime.strptime(close_date, '%Y-%m-%d %H:%M:%S')
            except ValueError:
                error_message = "Invalid close_date format. Use 'YYYY-MM-DD HH:MM:SS'.
                log_error(error_message)
                detailed_error_message = "Failed to delete customer due to invalid clo
                notify_failure("Customer Deletion Failed", detailed_error_message)
                return jsonify({"error": error_message}), 400

        query = session.query(Opportunity)

        if opportunity_id:
            query = query.filter(Opportunity.opportunity_id == opportunity_id)
        if account_name:
            query = query.filter(Opportunity.account_name == account_name)
        if dealer_id:
            query = query.filter(Opportunity.dealer_id == dealer_id)
        if dealer_code:
            query = query.filter(Opportunity.dealer_code == dealer_code)
        if opportunity_name:
            query = query.filter(Opportunity.opportunity_name == opportunity_name)
        if stage:
            query = query.filter(Opportunity.stage == stage)
        if probability is not None:
            query = query.filter(Opportunity.probability == probability)
        if close_date:
            query = query.filter(Opportunity.close_date == close_date)

        customers_to_delete = query.all()

        if not customers_to_delete:
            error_message = "Customer(s) not found based on provided query parameters.
            log_error(error_message)
            detailed_error_message = "Failed to delete customer(s). No matching custom
            notify_failure("Customer Deletion Failed", detailed_error_message)
```

```
            return jsonify({"error": error_message}), 404

        for customer in customers_to_delete:
            session.delete(customer)
        session.commit()

        success_message = (
                f"Customer(s) deleted successfully.\n\n\n"
                f"Deleted Customers:\n" + "\n".join([f"Opportunity ID: {customer.oppor
                                                    f"Opportunity Name: {customer.opp
                                                    f"Account Name: {customer.account
                                                    f"Dealer ID: {customer.dealer_id}
                                                    f"Dealer Code: {customer.dealer_c
                                                    f"Amount: {customer.amount}\n"
                                                    f"Close Date: {customer.close_dat
                                                    f"Created Date: {customer.created
                                                    for customer in customers_to_dele

        )
        notify_success("Customer Deletion Successful", success_message)

        return jsonify({"message": "Deleted successfully"}), 200

    except Exception as e:
        error_message = f"Error in deleting customer: {str(e)}"
        log_error(error_message)
        detailed_error_message = f"Failed to delete customer due to an internal server
        notify_failure("Customer Deletion Failed", detailed_error_message)
        return jsonify({"error": "Internal server error", "details": error_message}),

    finally:
        log_info("End of delete_customer function")
```

```
    if name == "main":
    app.run(debug=True)
```

PURPOSE

This block ensures that the Flask application runs directly when the script is executed, and not when it is imported as a module. It sets the application to run in debug mode, which is useful for development.

// APPS END //

// DB CONNECTIONS START //

db_connections package/

CONFIGURATIONS.PY :

This module establishes the connection to the PostgreSQL database and provides
a session factory to interact with the database.

Imports :

- # SQLAlchemy imports (for database engine creation, session management, and connection pooling)

  from sqlalchemy import create_engine
  from sqlalchemy.orm import sessionmaker
  from sqlalchemy.pool import NullPool

```
DATABASE_URL = "postgresql://postgres:postgrespassword@localhost:localhost code/postgr
```

This will be the connection to database. Provide postgrespassword and localhostcode.

CONFIGURATIONS FOR SETTING UP THE DATABASE CONNECTION AND SESSION MANAGEMENT USING SQLALCHEMY.

Imports:

- **CREATE ENGINE** : SQLAlchemy function used to create a new database engine, which manages connections to the database and handles SQL execution.

- **SESSIONMAKER** : A factory function that creates new SQLAlchemy session objects for interacting with the database. Sessions allow you to perform transactions with the database.

- **NULLPOOL** : A SQLAlchemy pool class that disables connection pooling, meaning each connection will be created and closed as needed, without reusing connections.

- **CONFIGURATION** : DATABASE_URL: The connection URL for the PostgreSQL database, containing the username, password, host, port, and database name.

- **ENGINE** : The SQLAlchemy engine object that is created using the connection URL. It is responsible for handling communication with the database.

- **CONN** : An active connection to the database created by calling `connect()` on the engine. This is useful for raw SQL queries if needed.

- **SESSION** : A session factory created using `sessionmaker`. It generates session instances that allow transactions with the

database. An active session object used to perform operations on the database such as querying, inserting, and updating data.

This Creates a new SQLAlchemy engine with NullPool (no connection pooling)

```
engine = create_engine(url=DATABASE_URL, echo=True, poolclass=NullPool)
```

This will Establish a connection to the database

```
conn = engine.connect()
```

This will Create a session factory bound to the engine

```
Session = sessionmaker(bind=engine)
```

This will Instantiate a session for database operations

```
session = Session()
```

// DB CONNECTIONS END //

// USER MODELS START //

user_models package/

TABLES :

This module defines the SQLAlchemy ORM models for the `Account`, `Dealer`, and `Opportunity` tables, capturing details about customer accounts, dealers, and sales opportunities. Here's a breakdown of the models and their functionalities:

IMPORTS

- **SQLAlchemy Imports**: For defining ORM models and interacting with the database.
- **Date and UUID Imports**: For handling dates, unique identifiers, and timezone management.

`Base`: The base class for all SQLAlchemy ORM models, allowing them to map to database tables.

```
Base = declarative_base()
```

`ACCOUNT` TABLE :

```
class Account(Base):
    """
    SQLAlchemy ORM class representing the 'account' table.

    This table stores customer account details like account_id and account_name.
    """
    __tablename__ = 'account'

    account_id = Column("account_id", String(10), primary_key=True)  # We are using Fi
    # 4 random digits

    account_name = Column("account_name", String, nullable=False)

    def account_serialize_to_dict(self):
        """
        Converts the Account object into a dictionary.
        """
        return {
            'account_id': self.account_id,
            'account_name': self.account_name
        }
```

- **`account_id`**: A unique identifier for the account, using a string of 10 characters.
- **`account_name`**: The name of the account, which cannot be null.
- **`account_serialize_to_dict()`**: Method to convert the `Account` instance into a dictionary format.

`DEALER` TABLE :

```
class Dealer(Base):
    """
    This table stores dealer information like dealer ID, code, and opportunity owner.
    """
    __tablename__ = 'dealer'

    dealer_id = Column("dealer_id", String, primary_key=True, default=lambda: str(uuic
    dealer_code = Column("dealer_code", String, nullable=False)
    opportunity_owner = Column("opportunity_owner", String, nullable=False)

    def dealer_serialize_to_dict(self):
        """
        Converts the Dealer object into a dictionary.
        """
        return {
            'dealer_id': self.dealer_id,
            'dealer_code': self.dealer_code,
            'opportunity_owner': self.opportunity_owner
        }
```

- **`dealer_id`**: A unique identifier for the dealer, automatically generated using UUID.
- **`dealer_code`**: The code associated with the dealer, cannot be null.
- **`opportunity_owner`**: The owner of the opportunity, cannot be null.
- **`dealer_serialize_to_dict()`**: Method to convert the `Dealer` instance into a dictionary format.

`OPPORTUNITY` TABLE :

```
class Opportunity(Base):
    __tablename__ = 'opportunity'

    opportunity_id = Column("opportunity_id", String, primary_key=True, default=lambda
    opportunity_name = Column("opportunity_name", String, nullable=False)
    account_name = Column("account_name", String, nullable=False)
    close_date = Column("close_date", DateTime, nullable=False)
    amount = Column("amount", Float, nullable=False)
    description = Column("description", String)
    dealer_id = Column("dealer_id", String, nullable=False)
    dealer_code = Column("dealer_code", String, nullable=False)
    stage = Column("stage", String, nullable=False)
    probability = Column("probability", Integer)
    next_step = Column("next_step", String)
    created_date = Column("created_date", DateTime, nullable=False,
                          default=lambda: datetime.now(pytz.timezone('Asia/Kolkata')))

    # New fields for currency conversions and amount in words
    amount_in_words = Column("amount_in_words", String)
    usd = Column("usd", Float)  # US Dollars
    aus = Column("aus", Float)  # Australian Dollars
    cad = Column("cad", Float)  # Canadian Dollars
    jpy = Column("jpy", Float)  # Japanese Yen
    eur = Column("eur", Float)  # Euros
    gbp = Column("gbp", Float)  # British Pounds
    cny = Column("cny", Float)  # Chinese Yuan

    def serialize_to_dict(self):
        """
        Serialize the Opportunity instance to a dictionary with formatted dates and cu
        :return: dict
        """

        def format_datetime(dt):
            """Format datetime to 12-hour format with AM/PM"""
            return dt.strftime("%I:%M %p, %B %d, %Y") if dt else None

        def format_currency_conversions():
            """Format currency conversions into a readable string"""
            currencies = {
                'USD': self.usd,
                'AUD': self.aus,
                'CAD': self.cad,
                'JPY': self.jpy,
```

```
                'EUR': self.eur,
                'GBP': self.gbp,
                'CNY': self.cny
            }
            return '\n'.join(
                f"{currency}: {value if value is not None else 'None'}" for currency,

        return {
            'opportunity_id': self.opportunity_id,
            'opportunity_name': self.opportunity_name,
            'account_name': self.account_name,
            'close_date': format_datetime(self.close_date),
            'amount': self.amount,
            'description': self.description,
            'dealer_id': self.dealer_id,
            'dealer_code': self.dealer_code,
            'stage': self.stage,
            'probability': self.probability,
            'next_step': self.next_step,
            'created_date': format_datetime(self.created_date),
            'amount_in_words': self.amount_in_words,
            'currency_conversions': format_currency_conversions()
        }
```

- **`opportunity_id`**: A unique identifier for the opportunity, automatically generated using UUID.
- **`opportunity_name`**, **`account_name`**, **`close_date`**, **`amount`**, **`description`**, **`dealer_id`**, **`dealer_code`**, **`stage`**: Various attributes of the opportunity.
- **`probability`**, **`next_step`**: Optional attributes for the probability and next steps.
- **`created_date`**: The date when the opportunity was created, with a default value in IST.
- **Currency Fields**: Fields for storing amounts in various currencies.
- **`serialize_to_dict()`**: Method to convert the `Opportunity` instance into a dictionary with formatted dates and currency conversions.

// USER MODELS END //

// UTILITIES START //

utilities package /

REUSABLES.PY

This module contains reusable functions that can be utilized throughout `app.py` for various tasks related to sales opportunities and date parsing. Here's a breakdown of each function:

`GET_OPPORTUNITY_STAGE FUNCTION :`

```
def get_opportunity_stage(probability):
```

```
    """
    Determine the sales opportunity stage based on the probability value.

    :param probability: Integer representing the probability percentage (0 to 100).
    :return: String representing the stage name.
    :raises ValueError: If the probability value is out of range (0-100) or invalid.
    """
    if 10 <= probability <= 20:
        return "Prospecting"
    elif 21 <= probability <= 40:
        return "Qualification"
    elif 41 <= probability <= 60:
        return "Needs Analysis"
    elif 61 <= probability <= 70:
        return "Value Proposition"
    elif 71 <= probability <= 80:
        return "Decision Makers"
    elif 81 <= probability <= 85:
        return "Perception Analysis"
    elif 86 <= probability <= 90:
        return "Proposal/Price Quote"
    elif 91 <= probability <= 95:
        return "Negotiation/Review"
    elif probability == 100:
        return "Closed Won"
    elif probability == 0:
        return "Closed Lost"
    else:
        raise ValueError("Invalid probability value")
```

- **Purpose**: Determines the stage of a sales opportunity based on its probability.
- **Parameters**: `probability` - An integer percentage (0 to 100).
- **Returns**: A string representing the stage.
- **Raises**: `ValueError` for out-of-range or invalid probability values.

`GET_CURRENCY_CONVERSION FUNCTION :`

```
def get_currency_conversion(amount):
    """
    Convert a given amount from INR to various other currencies using predefined rates

    :param amount: Amount in INR to be converted.
    :return: Dictionary with the amount converted to various currencies (USD, AUD, CAD
    """
    # Dummy conversion rates (assumed for demonstration)
    usd_rate = 0.013
    aus_rate = 0.019
    cad_rate = 0.017
    jpy_rate = 1.76
    eur_rate = 0.012
    gbp_rate = 0.010
    cny_rate = 0.094
```

```
    # Perform currency conversions
    usd = amount * usd_rate
    aus = amount * aus_rate
    cad = amount * cad_rate
    jpy = amount * jpy_rate
    eur = amount * eur_rate
    gbp = amount * gbp_rate
    cny = amount * cny_rate

    # Return a dictionary with all conversions rounded to 2 decimal places
    return {
        'USD': round(usd, 2),
        'AUD': round(aus, 2),
        'CAD': round(cad, 2),
        'JPY': round(jpy, 2),
        'EUR': round(eur, 2),
        'GBP': round(gbp, 2),
        'CNY': round(cny, 2),
        'INR': round(amount, 2)  # Original amount in INR
    }
```

- **Purpose**: Converts an amount from INR to various other currencies based on predefined conversion rates.
- **Parameters**: `amount` - The amount in INR to be converted.
- **Returns**: A dictionary with amounts in different currencies and the original amount in INR.

`VALIDATE_PROBABILITY FUNCTION :`

```
def validate_probability(prob):
    """
    Validate probability value.
    :param prob: Probability value.
    :return: Boolean indicating validity.
    """
    return isinstance(prob, int) and 0 <= prob <= 100
```

- **Purpose**: Validates if a given probability value is within the valid range (0 to 100).
- **Parameters**: `prob` - The probability value to be validated.
- **Returns**: `True` if valid, `False` otherwise.

`VALIDATE_POSITIVE_NUMBER FUNCTION :`

```
def validate_positive_number(value):
    """
    Validate positive number.
    :param value: Number to validate.
```

```
    :return: Boolean indicating validity.
    """
    return isinstance(value, (int, float)) and value > 0
```

- **Purpose**: Checks if a given value is a positive number (integer or float).
- **Parameters**: `value` - The number to be validated.
- **Returns**: `True` if valid, `False` otherwise.

`VALIDATE_STAGE FUNCTION :`

```
def validate_stage(stage_str):
    """
    Validate stage value.
    :param stage_str: Stage value to validate.
    :return: None if valid, otherwise raises ValueError.
    """
    stage_str = stage_str.strip()  # Remove leading and trailing spaces
    if not stage_str:
        raise ValueError("Stage value cannot be empty or contain only spaces.")
    if not re.match(r'^[A-Za-z\s]+$', stage_str):
        raise ValueError(f"Invalid stage value: '{stage_str}'. Must contain only lette
    if len(stage_str) > 100:  # Assuming a max length of 100 characters
        raise ValueError(f"Stage is too long. Maximum length is 100 characters.")
    return stage_str
```

- **Purpose**: Validates if a given stage string is non-empty, contains only letters and spaces, and does not exceed 100 characters.
- **Parameters**: `stage_str` - The stage string to be validated.
- **Returns**: The validated stage string if valid, raises `ValueError` otherwise.

`PARSE_DATE FUNCTION :`

```
def parse_date(date_str):
    """
    Parse date from string.
    :param date_str: Date string.
    :return: Parsed datetime object.
    """
    try:
        return datetime.strptime(date_str, "%I:%M %p, %B %d, %Y")
    except ValueError:
        raise ValueError(f"Invalid date format: {date_str}. Expected format: '10:00 AM
```

// UTILITIES END //

// LOGGING START //

logging_package /

LOGGING_MODULE.PY :

This `logging_module` sets up a centralized logging configuration for your application, ensuring that all log messages are consistently captured and formatted. Below is a summary of the module's functionality and configuration details:

OVERVIEW

- **Purpose**: To configure the logging behavior for the entire application.
- **Logging Level**: `DEBUG` - Logs all events from `DEBUG` level and above (e.g., INFO, WARNING, ERROR, CRITICAL).
- **Log Format**: Each log entry includes a timestamp, the log level, and the message itself.
- **File Handling**: Log messages are appended to a file named `main.log`, ensuring that logs are preserved across multiple executions.

MODULE SETUP :

```
import logging

# Configure logging
logging.basicConfig(level=logging.DEBUG,
format='%(asctime)s - %(levelname)s - %(message)s',filemode='a', filename='main.log')
```

KEY CONFIGURATION DETAILS:

1. **Log Level**:
    - The log level is set to `DEBUG`, which ensures that all log messages from `DEBUG` and higher severity levels are recorded.
    - This is useful for capturing detailed application events during development, debugging, or production.
1. **Log Format**:
    - `'%(asctime)s - %(levelname)s - %(message)s'`: This format ensures that every log message is accompanied by the timestamp (`asctime`), the log level (`levelname`), and the actual message (`message`).
1. **File Handling**:
    - Log entries are appended to the `main.log` file (`filemode='a'`), allowing the file to grow with additional log messages rather than overwriting it.
    - If `main.log` doesn't exist, it will be created in the current working directory.

LOGGING_UTILITY.PY :

This `Logging Utilities Module` provides a set of utility functions that make it easier to log different types of messages (`INFO`, `ERROR`, `DEBUG`, and `WARNING`) in a centralized and consistent way. These utility functions interface with the `logging` configuration from your main logging setup, allowing for cleaner and more concise logging throughout your application.

FUNCTIONS:

1. **log_info(message)**:
    - **Purpose**: Logs informational messages that describe the general functioning of the application.
    - **Use Case**: To record standard application events like starting services, completing processes, or tracking non-critical updates.

- **code :**

```
 def log_info(message):
    """
    Logs an informational message.

    :param message: Message to log.
    :return: None
    """
    if LOG_SWITCH:
        logging.info(message)
```

2. **log_error(message)**:
    - **Purpose**: Logs error messages, typically when something goes wrong in the application.
    - **Use Case**: To capture issues like failed operations, unexpected exceptions, or critical errors.

- **Code :**

```
def log_error(message):
    """
    Logs an error message.

    :param message: Message to log.
    :return: None
    """
    if LOG_SWITCH:
        logging.error(message)
```

3. **log_debug(message)**:
    - **Purpose**: Logs detailed debug messages that provide insights into the internal state of the application.
    - **Use Case**: Helpful during development and troubleshooting to track variable states, function flows, and more.

- **Code :**

```
def log_debug(message):
    """
    Logs a debug message.

    :param message: Message to log.
```

```
    :return: None
    """
    if LOG_SWITCH:
        logging.debug(message)
```

3. **log_warning(message)**:
    - **Purpose**: Logs warning messages that highlight potentially problematic situations or non-critical issues.
    - **Use Case**: To alert when something unexpected occurs but doesn't require immediate attention or action.

- **Code :**

```
def log_warning(message):
    """
    Logs a warning message.

    :param message: Message to log.
    :return: None
    """
    if LOG_SWITCH:
        logging.warning(message)
```

LOG_SWITCH:

- **Purpose**: Acts as a simple toggle to enable or disable logging throughout the application without removing or modifying logging calls.
- **Usage**: Set `LOG_SWITCH = False` to disable all logging temporarily.

// LOGGING END //

// EMAIL SETUP START //

email_setup package /:

EMAIL_CONFIG.PY :

This `Email Configurations Module` centralizes all email-related constants used in your application for sending notifications. These constants help configure the SMTP settings and manage the sender, recipient, and error-handling emails in a structured way.

CONSTANTS:

1. **`SMTP_PORT` (int)**:
    - **Purpose**: Specifies the port number used for the SMTP server with STARTTLS encryption.
    - **Default Value**: `587` (standard for STARTTLS).

- **Usage**: Used when setting up the SMTP connection for sending emails.

1. **`SMTP_SERVER` (str)**:
   - **Purpose**: Defines the address of the SMTP server.
   - **Default Value**: `"smtp.gmail.com"` (Gmail's SMTP server).
   - **Usage**: Connects to this server when sending emails via Gmail.

1. **`SENDER_EMAIL` (str)**:
   - **Purpose**: The email address from which notifications are sent.
   - **Default Value**: `"senders email @gmail.com"`.
   - **Usage**: Set as the "From" email in email communications.

1. **`RECEIVER_EMAIL` (list)**:
   - **Purpose**: A list of email addresses that will receive general notifications.
   - **Default Value**: `["receivers email @gmail.com"]`.
   - **Usage**: Use this list to specify the recipients for success, info, or general notifications.

1. **`ERROR_HANDLING_GROUP_EMAIL` (list)**:
   - **Purpose**: A list of email addresses that will receive error-related notifications.
   - **Default Value**: `["errors handling groupemail @gmail.com"]`.
   - **Usage**: For critical or error-handling notifications, use this list to alert the relevant group.

1. **`PASSWORD` (str)**:
   - **Purpose**: The password for the sender's email account.
   - **Default Value**: `"provide yours "` (empty for security purposes).
   - **Usage**: This will store the password required for authenticating the sender email address in the SMTP server.
   - **Note**: This should ideally be stored securely in an environment variable or a configuration file, not directly in the code for security reasons.

EMAIL_OPERATIONS.PY :

This module looks well-structured for handling email notifications within your application. The functions are clearly defined for different use cases such as customer creation, opportunity updates, and general success or failure notifications. Here are a few observations and suggestions:

OBSERVATIONS:

1. **`send_email` Function:**
   - It sends emails using `smtplib` and constructs the message using `MIMEMultipart`. It accepts a list of email addresses, subject, and body, which allows flexibility in handling multiple recipients.

1. **`notify_success` and `notify_failure` Functions:**

- These functions handle basic success and failure notifications by formatting the email content and invoking `send_email`. The use of `RECEIVER_EMAIL` and `ERROR_HANDLING_GROUP_EMAIL` ensures emails are directed appropriately.

1. **Formatted Notifications for Opportunities:**
   - The formatting of opportunity details is very clear and structured. You create detailed email content based on the opportunity fields, which will improve readability for recipients.

1. **Dynamic Opportunity and Customer Notification Functions:**
   - Functions like `notify_customer_creation_success` and `notify_opportunity_update_success` dynamically construct email content by looping over the fields of the given dictionaries. This ensures that you can handle a variety of fields without hardcoding.

SUGGESTIONS:

1. **Handle Sensitive Data:**
   - Ensure the `PASSWORD` in the `email_config.py` is either retrieved from a secure environment variable or a secrets manager to avoid hardcoding sensitive information directly into your code.

1. **Handling Empty Fields:**
   - While some fields in the opportunity or customer details may be optional (e.g., `amount_in_words`, `next_step`), you may want to include a default value (e.g., "N/A") or skip these fields if they are empty to keep the emails concise.

1. **Error Handling in Email Sending:**
   - You may want to include exception handling within the `send_email` function to manage potential failures, such as network issues or incorrect email credentials. This way, you can log errors or notify the appropriate team if the email fails to send.

SEND_EMAIL FUNCTION :

This function is used to send emails whenever there are changes in CRUD operations.

```
def send_email(too_email, subject, body):
    """
    This function is used to send emails whenever there are changes in CRUD operations
    :param too_email: list of email addresses needed to be sent
    :param subject: The subject of the email
    :param body: The message which user needs to be notified
    :return: None
    """
    if too_email is None:
        too_email = []

    msg = MIMEMultipart()
    msg['From'] = SENDER_EMAIL
    msg['To'] = ", ".join(too_email)
    msg['Subject'] = subject
```

```
    msg.attach(MIMEText(body, 'plain'))

    with smtplib.SMTP(SMTP_SERVER, SMTP_PORT) as server:
        server.starttls()
        server.login(SENDER_EMAIL, PASSWORD)
        server.sendmail(SENDER_EMAIL, too_email, msg.as_string())
```

NOTIFY_SUCCESS FUNCTION :

Sends a success notification email with detailed information.

```
def notify_success(subject, details):
    """
    Sends a success notification email with detailed information.

    :param subject: Subject of the success email.
    :param details: Detailed information to include in the email body.
    """
    body = f"Successful!\n\nDetails:\n****************************************\n{d
    send_email(RECEIVER_EMAIL, subject, body)
```

NOTIFY_FAILURE FUNCTION :

Sends a failure notification email with detailed information.

```
def notify_failure(subject, details):
    """
    Sends a failure notification email with detailed information.

    :param subject: Subject of the failure email.
    :param details: Detailed information to include in the email body.
    """
    body = f"Failure!\n\nDetails:\n****************************************\n{deta
    send_email(ERROR_HANDLING_GROUP_EMAIL, subject, body)
```

FORMAT_OPPORTUNITIES_FOR_EMAIL FUNCTION :

Format opportunities data for email content.

```
def format_opportunities_for_email(opportunities):
    """
    Format opportunities data for email content.
    :param opportunities: List of opportunities in dictionary format
    :return: str
    """
    email_content = ""
```

```
    for opp in opportunities:
        email_content += (
            f"Opportunity ID: {opp['opportunity_id']}\n"
            f"Name: {opp['opportunity_name']}\n"
            f"Account: {opp['account_name']}\n"
            f"Amount: {opp['amount']}\n"
            f"Amount in Words: {opp['amount_in_words']}\n"
            f"Close Date: {opp['close_date']}\n"
            f"Created Date: {opp['created_date']}\n"
            f"Dealer ID: {opp['dealer_id']}\n"
            f"Dealer Code: {opp['dealer_code']}\n"
            f"Stage: {opp['stage']}\n"
            f"Probability: {opp['probability']}%\n"
            f"Next Step: {opp['next_step']}\n"
            f"Description: {opp['description']}\n"
            f"Currency Conversions:\n{opp['currency_conversions']}\n\n"
        )
    return email_content
```

NOTIFY_OPPORTUNITY_DETAILS FUNCTION :

Sends an email with detailed opportunity information including the total count.

```
def notify_opportunity_details(subject, opportunities, total_count):
    """
    Sends an email with detailed opportunity information including the total count.

    :param subject: Subject of the email.
    :param opportunities: List of opportunities in dictionary format to include in the
    :param total_count: Total number of opportunities.
    """
    body = (
        f"Opportunity Details:\n"
        f"****************************************\n"
        f"Total Count of Opportunities: {total_count}\n\n"
    )
    body += format_opportunities_for_email(opportunities)
    send_email(RECEIVER_EMAIL, subject, body)
```

NOTIFY_CUSTOMER_CREATION_SUCCESS FUNCTION :

Sends a formatted success notification email for a new customer creation.

```
def notify_customer_creation_success(subject, customer_details):
    """
    Sends a formatted success notification email for a new customer creation.
```

```
    :param subject: Subject of the email.
    :param customer_details: Dictionary containing the details of the newly created cu
    """
    opportunity_id = customer_details.get("opportunity_id")
    opportunity_name = customer_details.get("opportunity_name")
    account_name = customer_details.get("account_name")
    close_date = customer_details.get("close_date").strftime("%d %B %Y, %I:%M %p") if
        "close_date") else "N/A"
    amount = customer_details.get("amount", "N/A")
    stage = customer_details.get("stage", "N/A")
    probability = customer_details.get("probability", "N/A")
    created_date = customer_details.get("created_date").strftime("%d %B %Y, %I:%M %p")
    currency_conversions = customer_details.get("currency_conversions", {})

    # Build the email content (email body)
    email_content = f"Dear Team,\n\nA new customer has been successfully created with
    email_content += "****************************************\n"
    email_content += f"Opportunity ID: {opportunity_id}\n"
    email_content += f"Opportunity Name: {opportunity_name}\n"
    email_content += f"Account Name: {account_name}\n"
    email_content += f"Close Date: {close_date}\n"
    email_content += f"Amount: {amount}\n"
    email_content += f"Stage: {stage}\n"
    email_content += f"Probability: {probability}%\n"
    email_content += "Currency Conversions:\n"

    for currency, value in currency_conversions.items():
        email_content += f"   - {currency.upper()}: {value:.2f}\n"

    email_content += f"Created Date: {created_date}\n"
    email_content += "****************************************\n"
    email_content += "\nRegards,\nCustomer Management Team"

    # Send the email
    send_email(RECEIVER_EMAIL, subject, email_content)
```

NOTIFY_OPPORTUNITY_UPDATE_SUCCESS :

Sends a formatted success notification email for opportunity updates.

```
def notify_opportunity_update_success(subject, details):
    """
    Sends a formatted success notification email for opportunity updates.
    :param subject: Subject of the email.
    :param details: Dictionary containing the details of the updated opportunity.
    """
    opportunity_id = details.get("opportunity_id")
    updated_fields = details.get("updated_fields", {})

    # Build the email content (email body)
```

```
    email_content = f"Dear Team,\n\nThe opportunity has been successfully updated with
    email_content += "*****************************************\n"
    email_content += f"Opportunity ID: {opportunity_id}\n\nUpdated Fields:\n"

    # Formatting updated fields
    index = 1
    if "opportunity_name" in updated_fields:
        email_content += f"{index}. Opportunity Name: {updated_fields['opportunity_nam
        index += 1
    if "account_name" in updated_fields:
        email_content += f"{index}. Account Name: {updated_fields['account_name']}\n"
        index += 1
    if "close_date" in updated_fields:
        close_date = updated_fields['close_date'].strftime("%d %B %Y, %I:%M %p")
        email_content += f"{index}. Close Date: {close_date}\n"
        index += 1
    if "amount" in updated_fields:
        email_content += f"{index}. Amount: {updated_fields['amount']:.2f}\n"
        index += 1
    if "currency_conversions" in updated_fields:
        conversions = updated_fields['currency_conversions']
        email_content += f"{index}. Currency Conversions:\n"
        for currency, value in conversions.items():
            email_content += f"   - {currency.upper()}: {value:.2f}\n"
        index += 1
    if "description" in updated_fields:
        email_content += f"{index}. Description: {updated_fields['description']}\n"
        index += 1
    if "dealer_id" in updated_fields:
        email_content += f"{index}. Dealer ID: {updated_fields['dealer_id']}\n"
        index += 1
    if "dealer_code" in updated_fields:
        email_content += f"{index}. Dealer Code: {updated_fields['dealer_code']}\n"
        index += 1
    if "stage" in updated_fields:
        email_content += f"{index}. Stage: {updated_fields['stage']}\n"
        index += 1
    if "probability" in updated_fields:
        email_content += f"{index}. Probability: {updated_fields['probability']}%\n"
        index += 1
    if "next_step" in updated_fields:
        email_content += f"{index}. Next Step: {updated_fields['next_step']}\n"
        index += 1
    if "amount_in_words" in updated_fields:
        email_content += f"{index}. Amount in Words: {updated_fields['amount_in_words'
        index += 1

    email_content += "*****************************************\n"
    email_content += "\nRegards,\nOpportunity Management Team"

    # Send the email with the correct recipient list
    send_email(RECEIVER_EMAIL, subject, email_content)
```

// EMAIL SETUP END //

// SQL SCRIPTS START //

sql_scripts package /

QUERIES FILE :

ACCOUNT TABLE

- **Account Table** holds basic information about the account.

```
CREATE TABLE account (
account_id VARCHAR(7) PRIMARY KEY,
account_name VARCHAR(100) NOT NULL
);

INSERT INTO account (account_id, account_name)
VALUES
('acc1234', 'Accenture'),
('hcl5678', 'HCL');



-- Query to view the contents


SELECT * FROM account;
```

DEALER TABLE

- **Dealer Table** is storing the dealer details with a UUID as the primary key.

```
CREATE TABLE dealer (
dealer_id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
dealer_code VARCHAR(10) NOT NULL,
opportunity_owner VARCHAR(100) NOT NULL
);

INSERT INTO dealer (dealer_code, opportunity_owner)
VALUES
('CH12', 'Komal Sai'),
('CH12', 'Dinesh'),
```

```
('BL04', 'Mahesh'),
('HY01', 'Sainath');

-- Query to view the contents
SELECT * FROM dealer;
```

OPPORTUNITY TABLE

- **Opportunity Table** stores details of the sales opportunities, including the associated account and dealer details. You have added additional fields for currency conversions and amount in words.

```
CREATE TABLE opportunity (
    opportunity_id UUID PRIMARY KEY DEFAULT uuid_generate_v1(),  -- Using UUID for a g
    opportunity_name VARCHAR NOT NULL,
    account_name VARCHAR NOT NULL,
    close_date TIMESTAMP NOT NULL,
    amount FLOAT NOT NULL,
    description VARCHAR,
    dealer_id UUID NOT NULL,  -- Foreign key to dealer table
    dealer_code VARCHAR(10) NOT NULL,  -- Redundant if dealer_id already exists, but k
    stage VARCHAR NOT NULL,
    probability INTEGER,
    next_step VARCHAR,
    created_date TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,

    -- New fields for currency conversions and amount in words
    amount_in_words VARCHAR,
    usd FLOAT,  -- Currency conversion to USD
    aus FLOAT,  -- Currency conversion to AUD
    cad FLOAT,  -- Currency conversion to CAD
    jpy FLOAT,  -- Japanese Yen
    eur FLOAT,  -- Euro
    gbp FLOAT,  -- British Pound
    cny FLOAT   -- Chinese Yuan,

    CONSTRAINT fk_dealer
        FOREIGN KEY (dealer_id)
        REFERENCES dealer (dealer_id)
);

-- Query to view the opportunities
SELECT * FROM opportunity;
```

- **Foreign Keys**: In the `opportunity` table, I added a foreign key constraint linking `dealer_id` to the `dealer` table. This ensures referential integrity between dealers and opportunities.
- **UUID**: UUID is used as the primary key for `dealer_id` and `opportunity_id` since it's more reliable for unique identification across systems.
- **Currency Fields**: I added fields for different currency conversions (`usd`, `aus`, `cad`, etc.) in the `opportunity` table, as per our requirements.

// SQL SCRIPTS END //

// REQUIREMENTS START //

REQUIREMENTS FILE (`REQUIREMENTS.TXT`)

1. **`psycopg2-binary`**: This is the PostgreSQL adapter for Python, typically used with SQLAlchemy for database connections.
2. **`SQLAlchemy~=2.0.34`**: SQLAlchemy is the ORM (Object-Relational Mapper) for managing database interactions.
3. **`Flask~=3.0.3`**: Flask is a micro web framework used for building web applications.
4. **`pytz~=2024.2`**: `pytz` provides accurate time zone support for Python applications.

ADD THE FOLLOWING LINES TO YOUR `REQUIREMENTS.TXT` TO INSTALL THE NECESSARY DEPENDENCIES WITH THE SPECIFIED VERSIONS:

```
psycopg2-binary
SQLAlchemy~=2.0.34
Flask~=3.0.3
pytz~=2024.2
```

INSTALLATION

TO INSTALL THESE DEPENDENCIES, RUN IN TERMINAL :

```
pip install -r requirements.txt
```

// REQUIREMENTS END //