Payment API Integration

Software Engineers:

- *Cody Strange – Software Engineer*

Abstract

Payment API Integration aims to develop a backend API that simulates processing payments, logs transaction data to a MySQL database, and is deployable through AWS Lambda. The project involves creating RESTful API endpoints to accept and validate payment data, store transaction history, and retrieve past transactions. It will also demonstrate the ability to package and deploy the application to the cloud using AWS services. This project showcases skills in backend development (using Flask or Node.js), database integration (MySQL), cloud deployment (AWS Lambda and API Gateway), API design, testing, and version control with Git. It aligns well with the requirements for a software developer role at Nexio, highlighting proficiency in building and maintaining scalable, cloud-based payment solutions.

Organization

Coding Standards

Commenting

Functions

- Brief description of function
- Parameters
- Return Values

Classes

- Brief description of class

ChatGPT

- I highly recommend letting ChatGPT do most of the commenting
- Double check any comments by ChatGPT

Naming Conventions

- Classes: CapWords
- Functions: snake_case
- Variables: snake_case
- Constants: ALLCAPS
- Files/Folders: snake_case

Code Formatter

- Run all python files through black

Type Safety

- Every function should have the parameters types listed and the return type of the function listed
-

```
def winrate_race(self, race_one:str, race_two:str = "all") -> float:···
```

Software Requirements

Programming Languages and Frameworks

- Python 3.11.0
- Javascript
- Node.js

Database Management

- MySQL –
  - Root password=SQLrootpassword
  - User credentials password=SQLuserpassword
  - cody@localhoast=yourpassword
- SQLAlchemy

Cloud Deployment and Infrastructure

- AWS CLI
    - Access Key = AKIAX464R6FHHIHZCT4D
    - Secret access key = dV2X+6Frh8g2x9dNLU/DzUWFU5gkmyP9ob4rXieo
- AWS RDS
    - Username = Cstrange
    - Master Password = Ihatethis
    - Account Id = 543239041358
    - Database name = paymentAPI
- AWS Lambda
- AWS API Gateway
- Zappa

Development Environment

- VS Code

Version Control

- Git
- Github

Virtual Environment

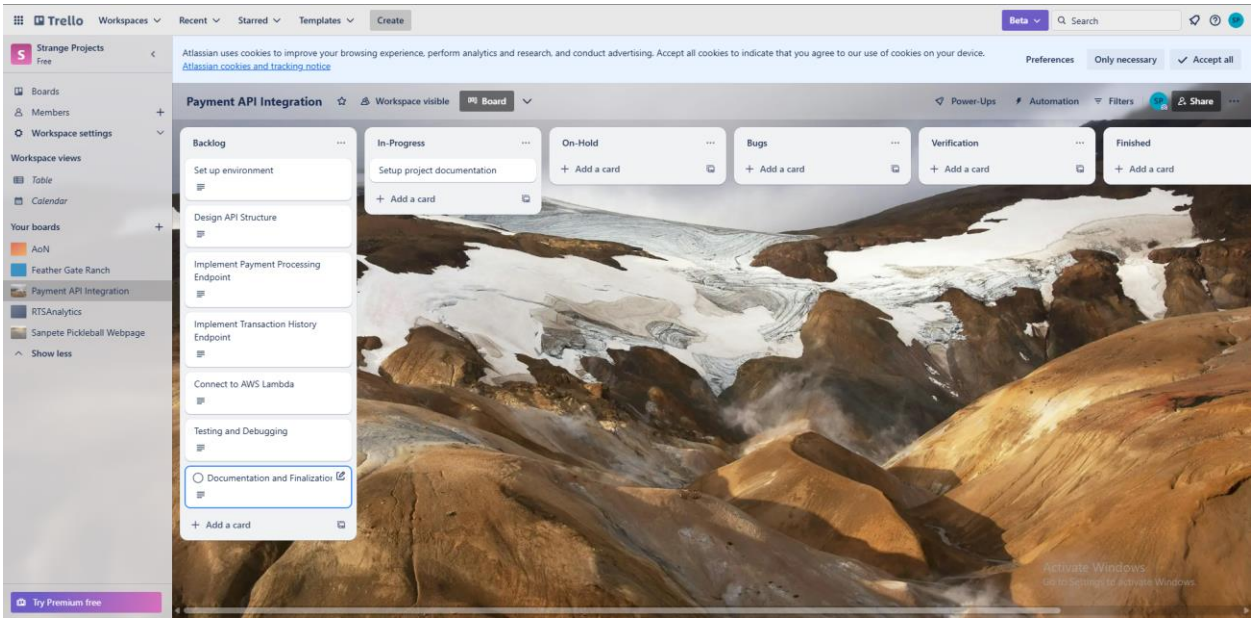
- Venv

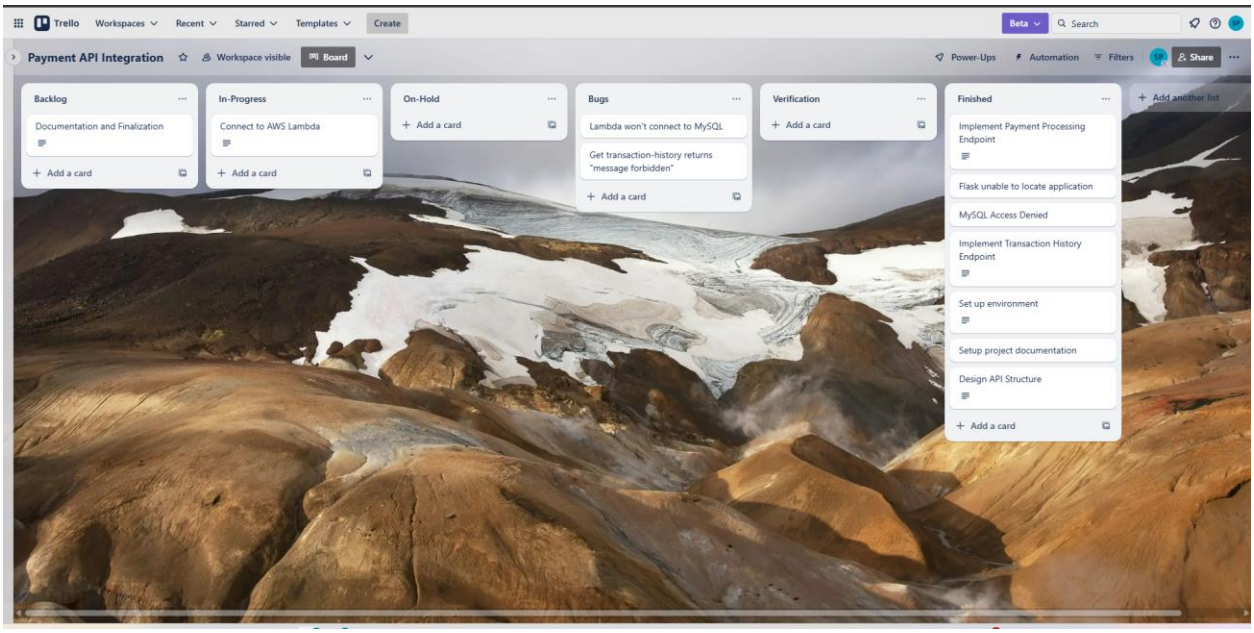
Additional Tools

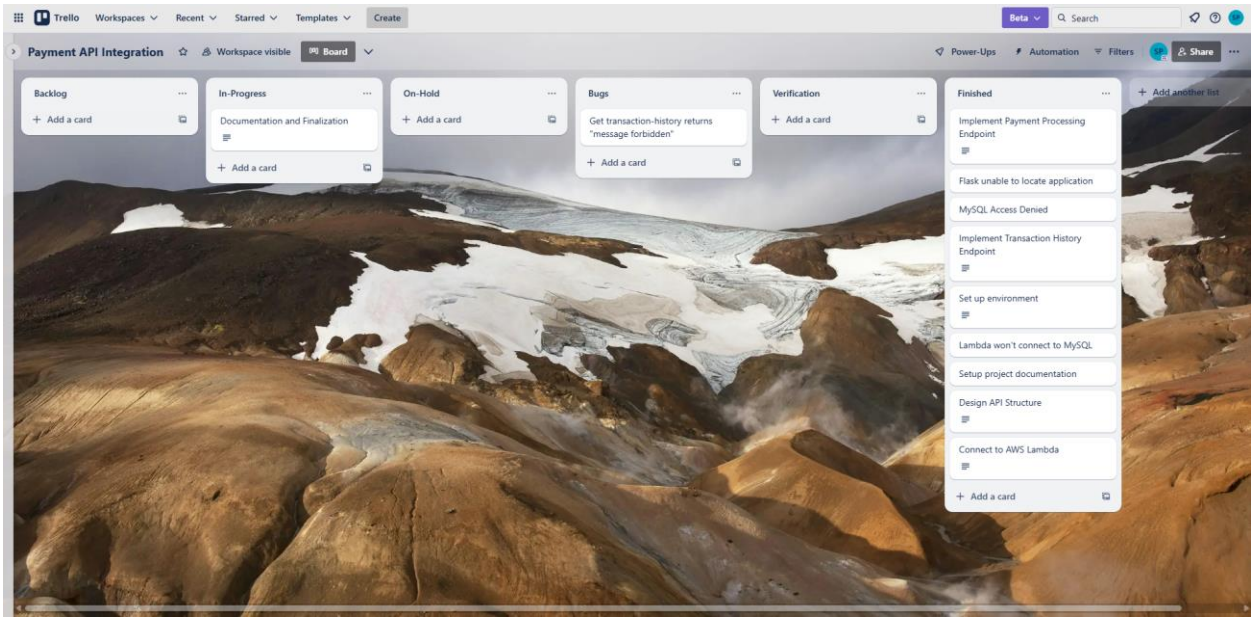
- cURL
- Lucidchart

Backlog

Description: This contains the tasks that we have completed each iteration as well as what we are currently working on, what known bugs exist, and what remains to be done. We add to the backlog as we discover new tasks but ignore them until we finish what we are working on.







Requirements Gathering

Functional Requirements

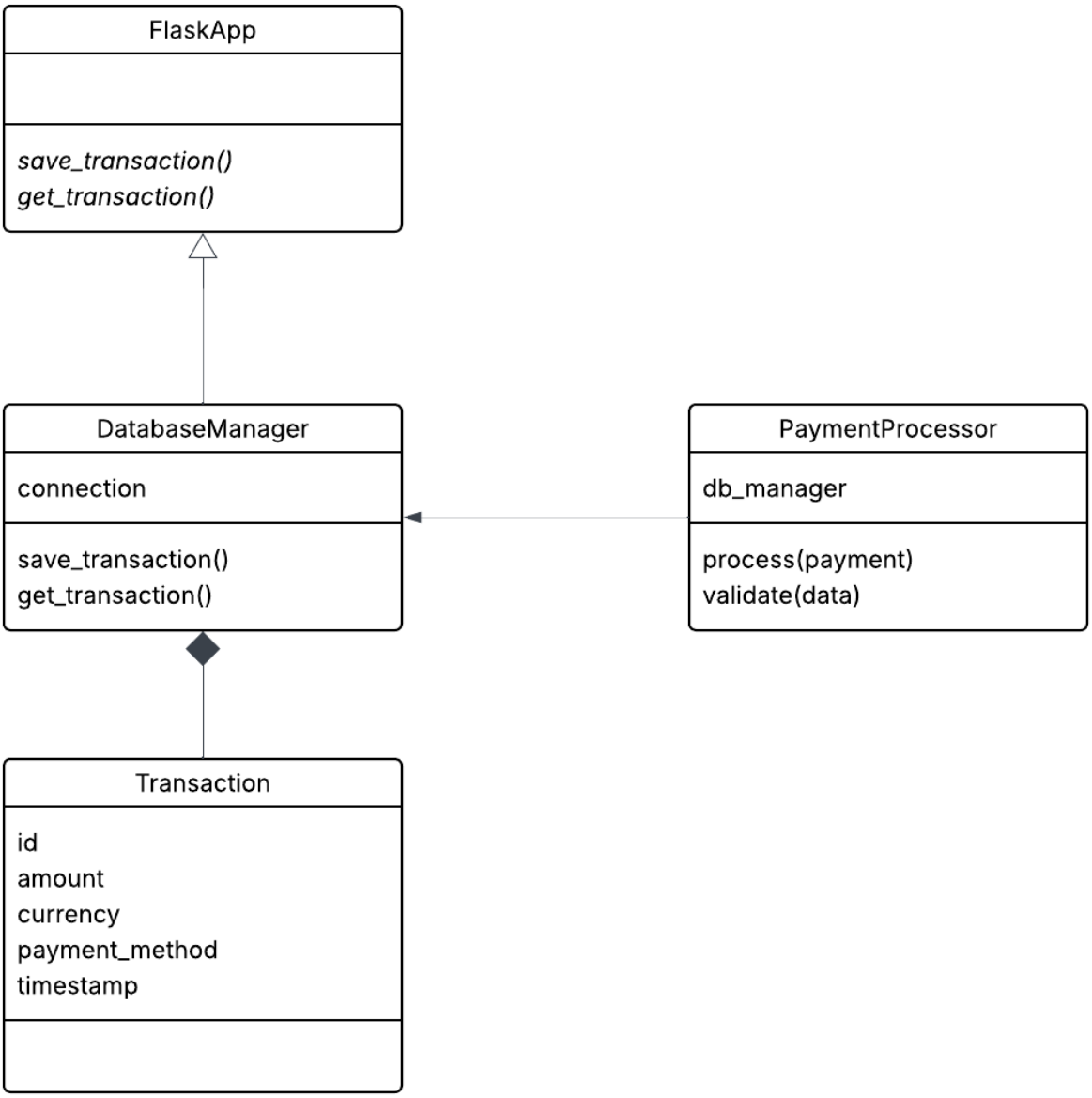
Description: These requirements are the general goals that our application should be able to meet.

- Simulate Payment Processing:
  - Accept payment information through an API endpoint (e.g., /process-payment).
  - Validate the data (amount, currency, payment method) before processing.
  - Simulate storing transaction details in a database (MySQL).
- Store Transaction History:
  - Log each transaction with relevant details (timestamp, amount, status).
  - Save this information securely in a database.
- Retrieve Transaction Data:
  - Provide an endpoint (e.g., /transaction-history) to fetch the list of past transactions.
- Deploy to the Cloud:
  - Package the backend application and deploy it on AWS Lambda.
  - Use AWS API Gateway to expose the API publicly.

Design

UML Class Diagram

Description:



Development

Code Snippets

Process_payment.py

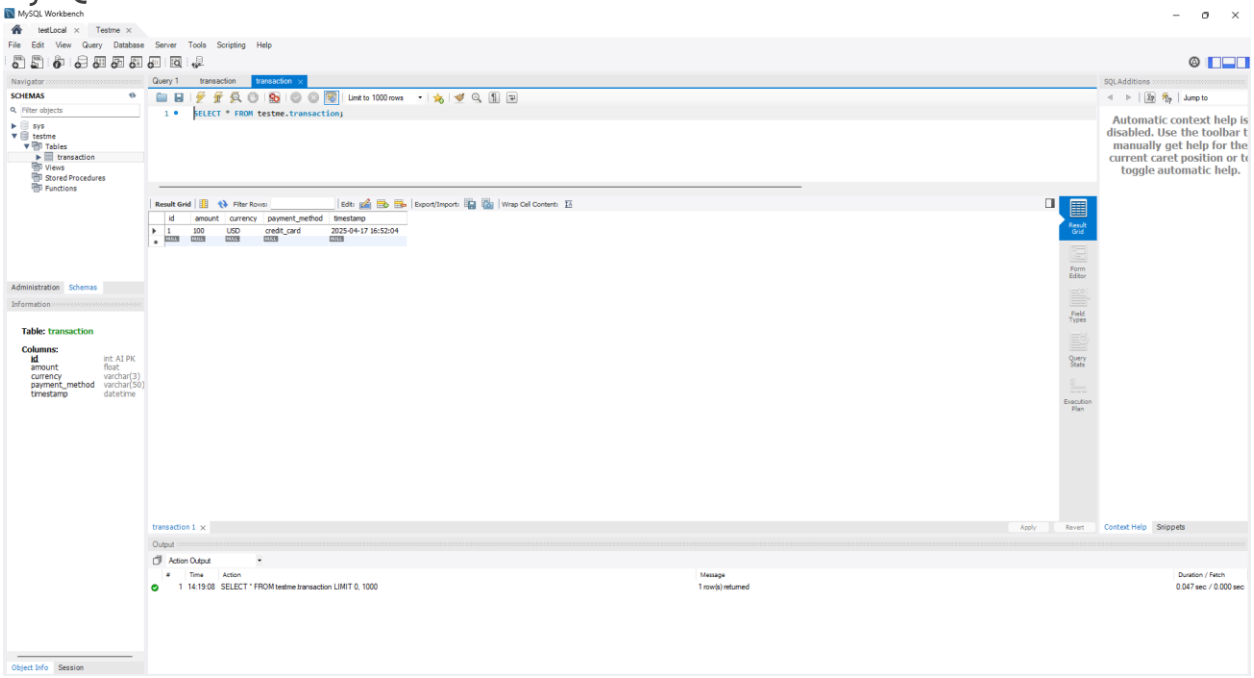
process_payment.py > index

```python
from flask import Flask, request, jsonify, Response  # Flask web framework and tools for JSON handling
from transaction import db, Transaction    # Import the database instance and Transaction model
import sys  # Used for flushing logs (optional)

# Initialize the Flask app
app = Flask(__name__)

# Configure connection to the cloud MySQL database
app.config['SQLALCHEMY_DATABASE_URI'] = 'mysql+pymysql://Clstrange:Ihatethis@database2.crm4oyaco2kc.us-west-2.rds.amazonaws.com:3306/testme'

# Uncomment this line to use a local database instead
# app.config['SQLALCHEMY_DATABASE_URI'] = 'mysql+pymysql://root:SQLrootpassword@localhost:3306/payment_api'

# Disable SQLAlchemy modification tracking (improves performance)
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

# Initialize SQLAlchemy with the app
db.init_app(app)

# Create all tables from models within app context
with app.app_context():
    db.create_all()

# Validate payment input data for required fields and correct types
def validate_payment(data: dict) -> tuple[bool, str]:
    """
    Validates incoming payment data.

    Args:
        data (dict): The payment data to validate.

    Returns:
        tuple: (is_valid: bool, error_message: str)
    """

    required_fields = ['amount', 'currency', 'payment_method']
    for field in required_fields:
        if field not in data:
            return False, f"Missing field: {field}"
    if not isinstance(data['amount'], (int, float)) or data['amount'] <= 0:
        return False, "Amount must be a positive number."
    if not isinstance(data['currency'], str) or len(data['currency']) != 3:
        return False, "Currency must be a 3-letter code (e.g., USD)."
    if not isinstance(data['payment_method'], str):
        return False, "Payment method must be a string."
    return True, ""

```

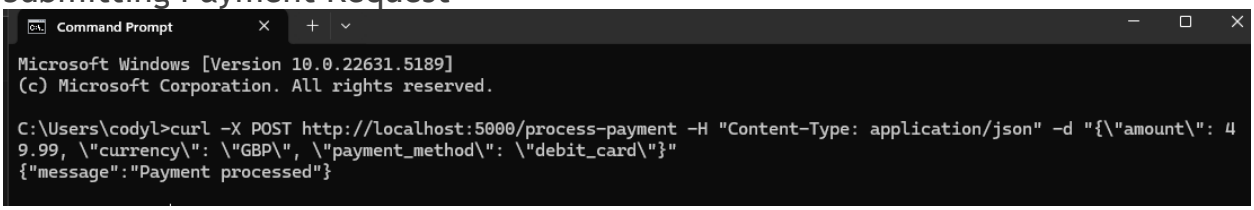
Sc2_extractor.py

```python
class SC2Extractor(Extractor):
    """
    Extracts, filters, and pushes data from replays
    to a database
    """

    def __init__(self) -> None:
        """
        SC2Extractor constructor
        """
        super().__init__()

        self._player_data = PlayerDataStorage()
        self._game_data = GameDataStorage()
        self._play_data = PlayDataStorage()

    def extract(self) -> dict:
        """
        extract data from a group of replays and return a dictionary of replay data
        """
        replay_container = {}
        replay_counter = 0

        # Getting replays from folder
        for filename in os.listdir(self.folder_path):
            file_path = os.path.join(self.folder_path, filename)
            if os.path.isfile(file_path) and file_path.endswith(".SC2Replay"):
                # Filling replay dictionary
                replay_counter += 1

                # Check file loads properly
                try:
                    replay = sc2reader.load_replay(file_path, load_map=True)
                except Exception:
                    logging.warning(f"File: {file_path} - File failed to load")

                replay_container[replay_counter] = replay

            else:
                # Check file opens properly
                logging.warning("File not found or File isn't of type .SC2Replay")

        return replay_container
```

MySQL Workbench



Submitting Payment Request



Review

Overview

This project was a deep dive into full-stack backend development, deploying a serverless payment API using Flask, AWS Lambda (via Zappa), and a MySQL database hosted on AWS RDS. The goal was to create a production-ready, cloud-hosted API capable of processing transactions and exposing them through a paginated history endpoint.

What I Learned

- How to build restful endpoints for handling data insertion and data retrieval
- About defining and validating JSON data inputs
- How to deploy python apps serverless using Zappa and AWS Lambda
- Configured and connected to AWS RDS MySQL from both local and cloud environments
- How to manage security and permissions to users

Struggles I Faced

- Connecting to the AWS RDS, faced problems with:
    - User permissions
    - Security inbound rules
    - URI formats

Successes

- Deployed a fully working cloud API that is both scalable and lightweight.
- Connected local and cloud environments seamlessly.
- Created a clean, well-documented project with a professional README, code comments, and API examples.
- Developed a reusable Flask app with modular organization and proper route handling.
- Built confidence with serverless deployment, cloud permissions, and troubleshooting tools.

Takeaways

This project worked with many layers of backend and cloud architecture: from Flask models to SQL and cloud networking. By working from local dev to AWS Lambda, I've gained a better sense capability over a production-ready Python API.

I'm now more confident in:

- Deploying and managing cloud APIs
- Navigating AWS tools like RDS and IAM permissions

This project is a solid showcase for future roles that value cloud-first API and serverless design.