Certainly! I understand you're looking for a more intricate demonstration that closely adheres to our design discussion.

Let's structure this better:

Server:

Represents the Payment Gateway.

Accepts authorization requests and communicates with a mock credit card company's server to get an authorization code.

Accepts batch payment requests, processes them, and provides a response for each transaction.

Will use SSL for encrypted communication.

Mock Credit Card Company's Server:

Another server that mimics a credit card company.

Accepts payment details and either authorizes or declines the transaction.

Client:

Represents the merchant.

Sends authorization requests.

Later, sends batch payment requests.

Server (Payment Gateway):

python

Copy code

from flask import Flask, request, jsonify

from werkzeug.serving import run\_simple

import ssl

import requests

app = Flask(\_\_name\_\_)

MOCK\_CC\_SERVER = "https://localhost:5000/authorize"

@app.route('/authorize', methods=['POST'])

def authorize():

data = request.json

# Communicate with the Mock Credit Card Company's server to get authorization

response = requests.post(MOCK\_CC\_SERVER, json=data, verify=False)

return response.json(), 200

@app.route('/batch\_payment', methods=['POST'])

def batch\_payment():

transactions = request.json['transactions']

results = []

for transaction in transactions:

response = requests.post(MOCK\_CC\_SERVER, json=transaction, verify=False)

results.append(response.json())

return jsonify(results), 200

if \_\_name\_\_ == '\_\_main\_\_':

context = ssl.SSLContext(ssl.PROTOCOL\_TLSv1\_2)

context.load\_cert\_chain(certfile='cert.pem', keyfile='key.pem')

run\_simple('127.0.0.1', 4000, app, ssl\_context=context)

Mock Credit Card Company's Server:

python

Copy code

from flask import Flask, request, jsonify

from werkzeug.serving import run\_simple

import ssl

app = Flask(\_\_name\_\_)

@app.route('/authorize', methods=['POST'])

def authorize():

data = request.json

if data['card\_number'].endswith('5'):

return jsonify({'status': 'declined'}), 400

return jsonify({'status': 'approved', 'authorization\_code': '12345'}), 200

if \_\_name\_\_ == '\_\_main\_\_':

context = ssl.SSLContext(ssl.PROTOCOL\_TLSv1\_2)

context.load\_cert\_chain(certfile='cert.pem', keyfile='key.pem')

run\_simple('127.0.0.1', 5000, app, ssl\_context=context)

Client (Merchant):

python

Copy code

import requests

import json

GATEWAY\_URL = "https://localhost:4000"

def authorize\_payment(card\_number, user\_info, amount):

data = {

'card\_number': card\_number,

'user\_info': user\_info,

'payment\_amount': amount

}

response = requests.post(f"{GATEWAY\_URL}/authorize", json=data, verify=False)

return response.json()

def batch\_payments(transactions):

data = {

'transactions': transactions

}

response = requests.post(f"{GATEWAY\_URL}/batch\_payment", json=data, verify=False)

return response.json()

if \_\_name\_\_ == '\_\_main\_\_':

# Authorizing a single payment

print(authorize\_payment('4111111111111115', 'John Doe', 100.50))

# Sending a batch of payments

transactions = [

{'card\_number': '4111111111111111', 'user\_info': 'Jane Smith', 'payment\_amount': 55.00},

{'card\_number': '4111111111111115', 'user\_info': 'Robert Brown', 'payment\_amount': 120.00}

]

print(batch\_payments(transactions))

Here's how it works:

The server (Payment Gateway) communicates with the mock credit card company server for authorizations.

The client (merchant) can send authorization requests and batch payments.

If a card number ends with a '5', the mock credit card company server declines it (this is just a mock rule to demonstrate the mechanics).

To run the code:

Start the Mock Credit Card Company's Server on one terminal.

Start the Payment Gateway (Server) on another terminal.

Run the client script to test the transactions.

This implementation is more detailed and should adhere closely to our design discussion. Let me know if you'd like to add more features or if any further adjustments are required!