Payment Processor Communication Implementation

Software Design & Programming

Distributed Computing

Joseph DiBiasi

University of Denver College of Professional Studies

11/1/2025

Faculty: Michael Schwartz, MS

Director: Cathie Wilson, MS

Dean: Michael J. McGuire, MLS

Introduction

The initial demonstration of our payment processor featured a simple UI to facilitate communication to our backend processing. This initial foundation has been built on further; the payment details being sent from the UI to the processing server are now persisted in a Postgres relational database. An additional API call has been created to demonstrate the settling of authorized payments. Submitting a form now has guardrails around it preventing the submission when fields are not valid. While these features are themselves still in their beginning stages, they represent an important step in the development of our payment processor as these updates allow for a more complete picture of how our payment processor will function.

Design and Implementation

While the overall design of the payment processor's call to make a payment authorization has not changed, the features surrounding it have been modified. The form field that controls the checkout information has been changed. Previously there was reduced validation around that area so that the error handling could be better demonstrated; now there are required checks on these fields ensuring they are filled out before form submission. Field changes have been made as well; the name used for shipping and billing was initially a combined field of first and last name. Name has now been broken into two fields for first and last name to better store this data accurately. Once all fields have been filled out correctly, the using the 'Checkout' button is no longer disabled and a payment authorization request is able to be sent to the backend.

Once a payment authorization request has been made to the backend payment processor service it proceeds through the DTO object validation then the payment gateway and

data storage methods. There has been a change to both of these methods. The payment gateway is still mostly a mockup to represent our third-party credit card network, but now the billing details object is updated with the results of that process and the date that it occurred. These additional fields have been added to help with the nightly batch process for settling payment claims. The billing and shipping details are then saved in the database. This is planned to happen regardless of the payment authorization's actual results as all data will be stored for auditing and troubleshooting purposes.

The nightly batch processing of authorized payments for settling claims is not yet implemented. The initial framework of this logic has been created for demonstration purposes and is represented through an additional API call. The call is a GET request triggered manually from the UI by clicking the 'Settle Payments' button. Once the request has been sent, all authorized payments that match the current date will be retrieved from the database and passed through our mocked-up settlement method. Once this process succeeds, each will be marked as having been settled through another field added to the billing details object. This is to prevent successful payment authorizations from having multiple settlement attempts as the process can be triggered multiple times in one day. Records previously marked as successfully settled will no longer be retrieved for settlement even if their date is still current. If for some reason the batch clearing process was not triggered for a successfully authorized payment on the day it was created, it will not be pulled in for clearing and settlement processes done on later days. This is intentional as it was assumed that any failure in this process would warrant an individual investigation into the error.

Despite the changes in application design, the methods for running our payment processor have not changed greatly. The backend processing layer is still provided by Java 21. The UI frontend is provided by Angular 19. Provided the local environment is still configured from the last payment processor demonstration there should be no need for additional

environment setup. Persistent data storage is handled via a Postgres 17 relational database, but this is a hosted database through Neon; there should be no need for additional setup to access this through the payment processor application. The proxy configuration file has been reworked and added to the angular.json file. This allows a regular ng serve command to be issued and the proxy configuration will be used automatically.

Challenges

One of the primary challenges faced was how to host a relational database on a limited budget. The solution to this issue was provided by Neon which provided an on-demand serverless Postgres database with limited storage suitable for demonstration purposes. Actual data retrieval was done using JPA repositories. This required a rework of the existing billing and shipping classes to fit with the JPA structure. It was a challenge to set up the initial billing details class so that it persisted in the hosted database, however, once that was done for the billing details class it was easy to replicate the same structure on the shipping class. The payoff for reworking both of these classes to accommodate JPA was creating repositories with simple and flexible queries built into the method naming structure. Find, save, and delete are all useful methods provided stock and when more complex queries are needed, such as adding the payment settlement mockup, all that was required was to structure the method name based on the entities fields we were querying. JPA then automatically created the query to retrieve billing data and all that was needed was to add the parameters.

Another challenge was related to the addition of a second network call. The initial proxy configuration setup was accidentally shadowing the Angular based UI; this only became apparent after the addition of the second network call broke the hosting on the front-end whenever the proxy was invoked. The solution to this was to prefix the actual network calls and restructure the proxy configuration file to match. Once implemented, all additional network calls

made using the prefix are routed via the proxy. This allows for local testing and development of communication between the UI and processing layer without running into cross-origin resource sharing errors. Whether it was the anticipated challenge of moving from a fluid storage system to one actually persisting the data or the unexpected challenge of configuration development for the local environment, these challenges were necessary to solidify the base functionality in the growing payment processor application.

Summary

This development iteration of our payment processor may not have involved changes as drastic as when the initial communication was implemented, but that is because the goal of this iteration was to enhance the existing systems. Persistent data storage is an important part of any application, especially when it pertains to e-commerce. Billing details are used to store all payment information and are essential for generating revenue via the nightly clearing and settlement process. Shipping details are more essential information for our merchant system to store. When a payment authorization is successful, we will need to deliver our product to the customer. Setting up this core of our data storage is the basis from which to expand our data structure. Meanwhile validations on our checkout form are the starting point for more detailed checks involving regex to ensure fields such as email and card numbers are in the proper format. These checks can help ensure that not only is data present before form submission but that the data will also match what the database is expecting in both form and size.

References

Microsoft Copilot. 2025. “Microsoft Copilot.” Microsoft Copilot. Microsoft. 2025. https://copilot.microsoft.com/