Payment Processor Communication Implementation

Software Design & Programming

Distributed Computing

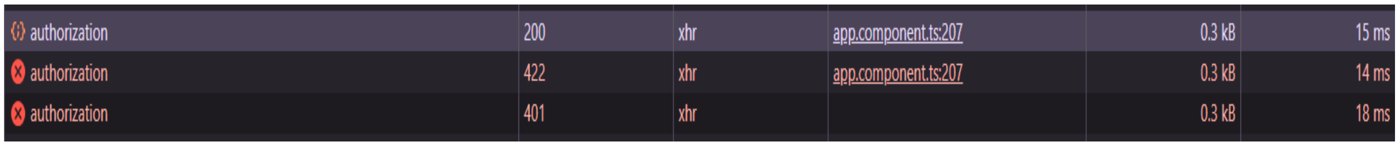
Joseph DiBiasi

University of Denver College of Professional Studies

10/4/2025

Introduction

Our payment processor is still in development, but the basic design has been fleshed out to give an idea of our workflow. The frontend user interface is provided using Angular 19. The design is basic and intended to be a simplified version of the checkout process. Once a user is ready to checkout, a payment authorization request will be sent to the Java 21 backend. This request is sent with headers specifying the content type as JSON and containing the authorization token. Failure to validate the authorization token results in a 401 unauthorized response. This initial mockup of the payment processor then validates the payment authorization details. If validation is successful a 200 okay response is returned; if errors in validation occur they are stored and returned along with a 422 unprocessable entity response. Connecting to the credit card network via a third-party payment gateway along with storing payment authorization information for nightly batch processing is currently only mocked up. The frontend listens for the response and logs the message returned.

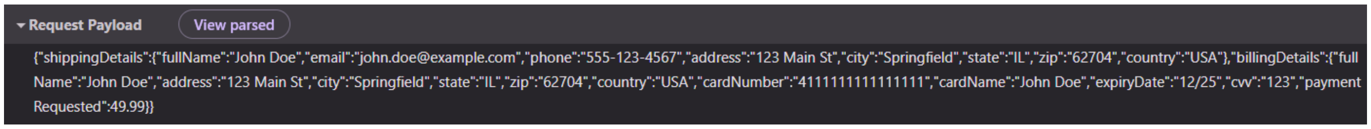


```
authorization   200   xhr   app.component.ts:207   0.3 kB   15 ms
authorization   422   xhr   app.component.ts:207   0.3 kB   14 ms
authorization   401   xhr                          0.3 kB   18 ms
```

Figure 1. Communication Response Examples

Design and Implementation

The initial design of this payment processor is focused on communication from the client side to the payment processor server side and receiving a response. The current implementation demonstrates this through the use of a POST request. All fields displayed on the user interface are able to be modified and, through the two way data binding provided by the ngModel, what the user sees on the interface is the same information that will be sent in the POST request. When the user is satisfied with their checkout information, they submit this information to the payment processor service. Currently, there is no validation enforced on the frontend so this information can be submitted regardless of the content in the fields. This was intentional to better demonstrate the communication process and return not only successful responses but also error responses if fields are left blank. Prior to release, the frontend will be converted to handle all shallow field validation, such as ensuring valid email or credit card number format. Submission will only be allowed if all fields have the required content filled out and match the proper format. A customer can still fill out their payment information incorrectly, but at least they will not be forgetting any crucial details like an address. The backend will then only handle validation of fields the contain incorrect content; instances such as a payment authorization failing due to incorrect billing information. Customers will be informed of the general failures and given the opportunity to resubmit their information once it has been corrected. Once information has been submitted, the data will be sent and received in JSON format.

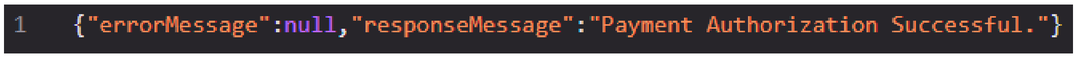


{"shippingDetails":{"fullName":"John Doe","email":"john.doe@example.com","phone":"555-123-4567","address":"123 Main St","city":"Springfield","state":"IL","zip":"62704","country":"USA"},"billingDetails":{"full
Name":"John Doe","address":"123 Main St","city":"Springfield","state":"IL","zip":"62704","country":"USA","cardNumber":"4111111111111111","cardName":"John Doe","expiryDate":"12/25","cvv":"123","payment
Requested":49.99}}

Figure 2. Request Payload in JSON Format

The JSON object being sent is broken down into two main key-value pairs, billing details and shipping details. The amount of payment that is requested is included with billing details; product type and amount were considered superfluous for this demo product and not included. These two main objects of billing and shipping details are then broken down into smaller fields containing everything needed for authorizing payment and shipping the product once payment has been successfully authorized. Billing and shipping details share overlapping fields, this is intentional because while most shipping information is the same as the billing information that is not a guarantee. Customers are more than welcome to pay for an item then ship it to a different location. For the majority of cases where customer shipping and billing addresses are the same, the 'Same as Shipping' checkbox will quickly fill all overlapping billing details with the matching shipping information. Clicking the 'Checkout' button will send the POST request.

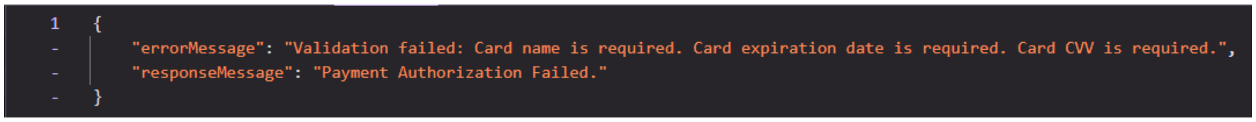
The backend utilizes the Spring Boot framework to make this communication process easier. Spring Boot gives access to restful controller classes designed to receive requests and respond. Currently there is only one controller route, the payment authorization method which utilizes POST mapping to connect with the payment authorization request sent from the frontend. The actual processing of this payment authorization is kept separate from the controllers and instead is located in the payment processor service layer. This service layer will eventually be responsible for the very important tasks of connecting to our third-party payment gateway and storing payment authorization details in our database for later batch processing. These two important tasks are only stubbed out at the moment but will be crucial to the success of our payment processing service. Instead our payment processor service layer validates whether or not the sent request contains data in all fields for billing and shipping details. If validation succeeds, a 200 okay response is returned.



```
{"errorMessage":null,"responseMessage":"Payment Authorization Successful."}
```

Figure 3. Payment Authorization Successful Response

If validation fails a 422 unprocessable content error is returned, signifying that while the content type is correct the content itself contains errors. If any fields fail validation, the overall request will fail but each field is still validated before this so that the response can inform the user of every field that is in error. This logic will eventually be expanded upon to allow customers the chance to re-attempt checkout after attempting to fix the erroneous fields. Backend unit tests covering the controller, model, and service implementation classes were created using AI assistance and validated by developers. Frontend tests are not yet available but will consist of automation testing using Selenium.



```
{
    "errorMessage": "Validation failed: Card name is required. Card expiration date is required. Card CVV is required.",
    "responseMessage": "Payment Authorization Failed."
}
```

Figure 4. Validation Failure Response Containing Multiple Field Errors

Running Demonstration Code

This payment processor is still far from deployment. Any attempt at demonstrating the current payment authorization workflow must be done using a local machine setup for development. Local machine setup includes, but is not limited to, installation of a Java 21 development kit, installation of the front end packages such as Angular 19, and compiling the code prior to running. Additional requirements may vary based on operating system used. Once these needs have been met the backend payment processor service can be started by navigating to the payment-processor-service file location and running the operating system command of mvn spring-boot:run; the payment processor UI can be started using the command npm start. Despite being an Angular project, a simple ng serve command will not suffice for demonstrating the workflow locally as it needs to be started using a proxy config file to navigate the cross-origin resource sharing (CORS) issues. Browsers implement CORS protections to prevent webpages from making a request to a different origin, protecting against cross-site request forgery and other malicious attacks. Users on a windows machine can instead use the provided start_payment_processor batch file. This batch file can be run to start both the payment-processor-service and payment-processor-ui. The payment-processor-service will start first in the background and then the payment-processor-ui should open using the users default browser. Once the browser has opened up the payment-processor-ui, the user can fill out all the shipping and billing details. For easier testing, clicking the ‘Load Default Data’ checkbox immediately populates all fields; unchecking this box clears them. No field validation enforced to submit requests. Clicking ‘Checkout’ submits the request; ‘Cancel’ is not yet implemented.

Dependencies and Assumptions

This project is built on the assumption that more development is forthcoming. Crucial features of the payment processing workflow have not yet been implemented and only stubbed out as methods. Our payment processing service will need to fully implement these stubbed methods before it can be considered complete and one of these methods relies on a third-party connection that we have not yet made. It is possible that we may need a different data structure to what we are anticipating, this would incur additional development costs rewriting the code to adapt.

Demonstrating this code on a machine locally is easy, demonstrating this code on any possible machine locally is another task. There are many different operating systems and factors that can impact environment setup. Overall instructions to set up the local environment have been provided but the exact details will vary. We need to assume that this project can be shared and run successfully with the relevant parties while our next focus is on finding a stable deployed environment to better show off progress.

Summary

Our payment processor still has a lot of development needed before it can be considered the minimum viable product for release, but this small product is able to demonstrate a high level workflow of what we hope our payment processor will achieve. Users can fill out their shipping and billing details needed to validate purchases of our product. The current basic validation checks will be increased in scope and design to provide as seamless a user experience as possible. Our initial security is a bit lacking, but the basic authorization checks are the basis which will be expanded upon to protect against all anticipated security threats. This initial communication structure between client and server is a solid start and can be replicated across our application to meet all our communication needs. All applications start from humble beginnings, this is ours.

References

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

Steen, van Maarten and Andrew Tanenbaum. 2023. Distributed Systems. CreateSpace Independent Publishing Platform. https://www.distributed-systems.net/index.php/books/ds4/