INITIAL NOTES/COMMENTS:

After an order is processed, I am **NOT** updating it's status (processed, failed) at the remote repository. I am however updating it's status in the local service' data store. This is also intentional as I prefer to leave this data in its original state.

I've used various programming languages, technologies, and architectural patterns (snippets). There's a lot of details and areas not worth mentioning, but hopefully this will help drive discussions around various concepts, answers to questions, etc;

I've scraped the raw data from the underline warehouse and order API's into a spreadsheet and performed the required validation calculations in order to compare the processed order results of my solution:
https://docs.google.com/spreadsheets/d/1ol-VMBKpvjA4M363UXF-RX3xFCuMLWr-hZHv71P74eQ/edit?usp=sharing

Services and Technologies Used:

DOCKER- I've broken each of the services into individual docker containers. This is really for ease of development locally, and not designed in an ideal production-ready state.

SQLITE- Within the Product Service, I use SQLITE as the local data store. It's lightweight and easy to use without needing to run a dedicated data store service.

CLIENT [React & Typescript]- This houses the react front-end service accessed via a browser, which interacts with the backend services via the api-gateway.

API-GATEWAY [Node] - This service is what a browser client sends REST requests to and is the only public facing service. It manages the routing of all REST requests to the corresponding internal services. It serves all responses from an internal service back to the client.

PRODUCT SERVICE [PHP] - This is the primary service which manages 2 bounded contexts within it: Order and Inventory.

Architecture Theory:

From a DDD perspective, DDD is a concept where instead of visualizing and modeling around the structure of DB schemas and data (more of a traditional approach), the behavior and rules of a domain make up the concept of the models.

PRODUCT SERVICE:

Domain events are used allowing subscribers to listen and take action. Event sourcing is not used. The events and their respective listeners are executed synchronously instead of being put into a data store or queue and handled asynchronously - this is simply for demo purposes but adding them to a pub/sub queue is simple :)

Inventory Service

- A Product is considered the aggregate root of this context. An inventory item composed of available quantity, warehouse id, etc is one of the building blocks used to create a Product.

- I wanted to demonstrate a basic <u>eventual consistency</u> data concept. What I'm doing is fetching all warehouse data from an external service, and updating/inserting it in a local db. This allows any reads for warehouse data to be read from this DB (instead of directly from the external warehouse API). This means that if there are no current writes, eventually any reads to the data will return the last write, hence eventual consistency..
- A cronjob is running every 5 minutes to fetch new data from the external warehouse, and store this data in the local DB. Again, not an ideal way to maintain quantities from a ledger/historical perspective, but a way to demonstrate a mitigation strategy around perceived data availability, service degradation, request throttling, etc;
- I used some pieces of the CQRS pattern to separate the reads from the writes (yes, it's ok to read from the same DB you write to).

<u>Order Service</u>
- An Order is considered the aggregate root of this context. Order Item entities are composed of inventory id, and quantity ordered.
- Following DDD, in order to enforce invariants, various business rules were applied to an aggregate either directly or through the entities comprising the aggregate. These are the various building blocks used to build an aggregate root (or rather, the aggregate is built using various entities, value objects, etc;).

FLOWCHARTS:

I've created a few flow charts and diagrams which help illustrate the architecture at a high level, and various use cases and workflows, which can be viewed here (5 Pages):
https://lucid.app/lucidchart/a3630951-de62-496a-9a94-01ad6e108258/view

SCREENSHOTS OF ORDERS STATUS:

Reset State | Process All

Order ID	Status	Items	Total Qty
#2	NEW	2	2
#3	NEW	1	1
#4	NEW	1	4
#5	NEW	1	2
#6	NEW	1	2
#7	NEW	1	1
#8	NEW	1	2
#9	NEW	1	1
#10	NEW	1	1
#11	NEW	1	1
#12	NEW	1	1
#13	NEW	1	1
#14	NEW	1	1
#15	NEW	2	4
#16	NEW	1	1

Reset State | Process All

Order ID	Status	Items	Total Qty
#2	FAILED	2	2
#3	FAILED	1	1
#4	FAILED	1	4
#5	FAILED	1	2
#6	PROCESSED	1	2
#7	FAILED	1	1
#8	PROCESSED	1	2
#9	PROCESSED	1	1
#10	PROCESSED	1	1
#11	FAILED	1	1
#12	PROCESSED	1	1
#13	FAILED	1	1
#14	FAILED	1	1
#15	FAILED	2	4
#16	PROCESSED	1	1