Scalable, Distributed E-commerce Design

Uchenna Kevin Anyanwu

10/22/2018

Background

Monolithic service architecture possesses several advantages such as being simple to deploy, have less moving parts in its design, deterministic memory footprint, and good for simple demonstrations. However, applications like these are pretty common and have many advantages, they are easy to comprehend, manage, develop, test and deploy. You can also scale them by running multiple copies of it behind a load balancer and it works quite well up to a certain level[1]. The drawbacks of the monolithic design are numerous: first, the developer is restricted to one language for the monolith; second, agile development is difficult due to strong coupling; third, when one feature is changed, the entire production cycle is halted; lastly, all features are implemented on one processor and would be difficult to take advantage of a more sophisticated system with improved hardware capabilities. To remedy these shortcomings, system architects choose, whenever possible, the microservice architecture. In this architecture, the monolithic system is split into smaller, independent components or resources. These systems are flexible, maintainable, fault-tolerant, and scalable. However, these benefits are without flaws. Firstly, distributing the software over multiple locations means that there are more moving parts, hard to track down bugs; Second, calls from one system to the next, takes time, and can be costly; third, integration and testing becomes more involved, taking more manpower and resources; fourth, architects have to think more about system availability and robustness; lastly, more time will be spent on designing interfaces between components, and, if care is not taken, will lead to negative ripple effects across the system.

Aim

This paper intends to design and impliment a scalable, distributed ecommerce backend system using a functional monolith ecommorce system.

Method

First, we choose a high-level programming language that is easy to learn, good for rapid prototyping and industry-leading in software development.

Second, we develop an actor system diagram to show the interactions between all actors of an e-commerce system (not shown).

Third, we design the interfaces between the actors of the system, this will help in understanding the set of APIs needed to interact with the system (not shown).

Fourth, we develop an architectural system diagram that encompasses the previous two steps (figures)

Fifth, we develop an API documentation with accompanying simple test cases (POSTMAN).

Sixth, we code CRUD operations to CREATE, READ, UPDATE, and DELETE resources of the system. In addition, we write other APIs listed in the previous step.

Lastly, we test each component in isolation on Windows 10 machine, stubbing external dependencies, and progressively adding real component into the environment, until we have included all of the real components to form the entire system.

Software Requirements

Describe a general architecture for a distributed, scalable system that would serve as a backend for a Describe a general architecture for a distributed, scalable system that would serve as a backend for a hypothetical webstore and shopping cart type application. The requirements would be that individual users would have records of what was purchased and their properties, and be able to return to an existing cart of items if they had a cart that wasn't emptied between sessions by check-out.

Implement in code using a bare-bones version of this shopping backend cart and the purchase history component as a 'Proof of Concept. It must understand that different products have certain fixed properties (name, price), and many specific properties (ie, t-shirt sizes, color of hat, flavor of candy, etc).

The code should be modular enough to easily add new products or integrate with a hypothetical frontend. It should also have the ability for a specified user to return to an existing cart of items, then update the user's purchase history and emptying the cart on check-out.

Design and Developlement Ecommerce Design

Ecommerce Design

There are several basic components of an e-commerce system: User management, which manages each user account, storing information specific to each account ; Products, which specifies each unique product that is available; Category, which provides a grouping for products, which makes searching easy; Order management, which places the order of each product, and also, acts as the accountant of system; Payments, which bills the account and interfaces directly to the outside payment portal; Shopping cart, which is a holding location for soon-to-be checkout-out item; and Search, which is used to quickly browse through available products. The number of components in an e-commerce system can vary immensely. Large e-commerce systems like Amazon uses several hundred different components, all working together, to enable the processing of a single item, end-to-end.

Monolithic Ecommerce Design

Figure 1 (at the bottom) is a monolithic design of a generic backend for an e-commerce system. In a very simple system, there are three components: the database, a complete e-commerce service, and a web server, would be running one machine or a set of machines, but each machine being duplicated. This approach is most favorable for small-scale development.

Distributed Ecommerce Design

Figure 2 (at the end) is a microservice design of a generic backend for an e-commerce system. These independent services could run at different locations, not just on one local host machine, in which case, different IP address would be assigned to each service in the system. This approach is most favorable for agile, modular, and scalable applications.

Scalable, Robust Ecommerce Design

Figure 3 (at the end) add a Registor, load balancer, and service discovery instances to the e-commerce system, we can truly achieve high availability, scalable, and fault-tolerance. The service discovery node, such as Zookeeper node, would track the current network location of a service instance, based on instructions from Registor, which monitors the starting and stopping of containers and updates ZooKeeper about the state changes. Apache Ngnix Plus would be used as a load balancer to

keep requests distributed in a fair manner amongst all the services in the e-commerce system.

Implementation of cart and purchase history

The shopping cart microservice was designed using python, flask, and sqlite3 database. In this service, we wrote CRUD, and accompanying operations that would suit a client or POSTMAN. The response would be in JSON, java script object notation, which is a widely accepted standard technology for interoperability between diverse systems. XML is another standard that was not used because of is lack of adoption in internet technology space.

The following API were developed to add user specified products from the product db to the cartdb : /ecommerce/v1/cart/product: POST

The following API was developed to view the purchase history of the current logged in user : /ecommerce/v1/order: GET

The following sequence of call can be abstracted to the process followed by a user from sign-up to checkout.

Specific implementation code of all other APIs can be found in the repository of the github page.

/ecommerce/v1/account/register - Register new user

/ecommerce/v1/account/login - Login new user

/ecommerce/v1/product/ - retreive all available products

/ecommerce/v1/product/query - query products

/ecommerce/v1/cart/product - add products to user's cart

/ecommerce/v1/cart/user - retreive products from cart

/ecommerce/v1/cart/product/remove - remove an item from the cart

/ecommerce/v1/payment/ - pay for all products in cart

/ecommerce/v1/order - retrieve history of paid products

/ecommerce/v1/account/logout - logout

Also, at any time, users can log out, before checkout, and then log in without all products in the cart cleared. This feature is realized by the cart service requesting products from the product service and storing those products in its cart database, which is persistent. Persistence allows data written from memory to a permanent storage. In a real e-commerce system, this is replicated over several databases. Purchasing history is also stored on a database as well. When users call /ecommerce/v1/payment/, the payment service transfers all items from the cart database to the order database, while requesting cart service to delete those products from the cart database.

Testing Microservices

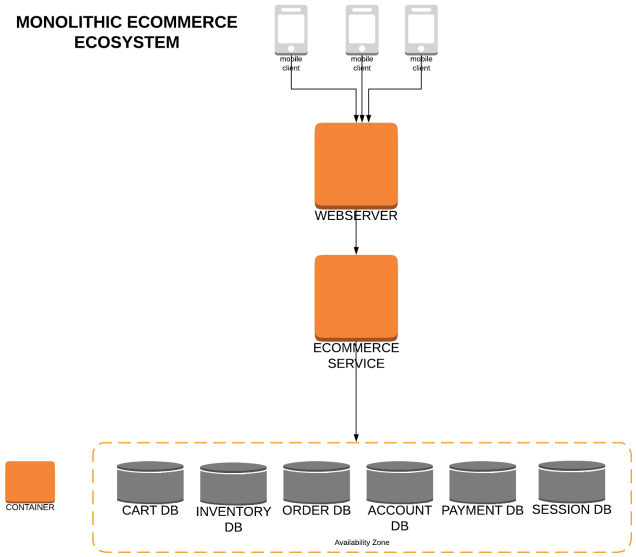
System testing was done on the domain-level, client level. We used POSTMAN to test our design. Given more time, a more robust testing platform and a variety of test case would be utilized.

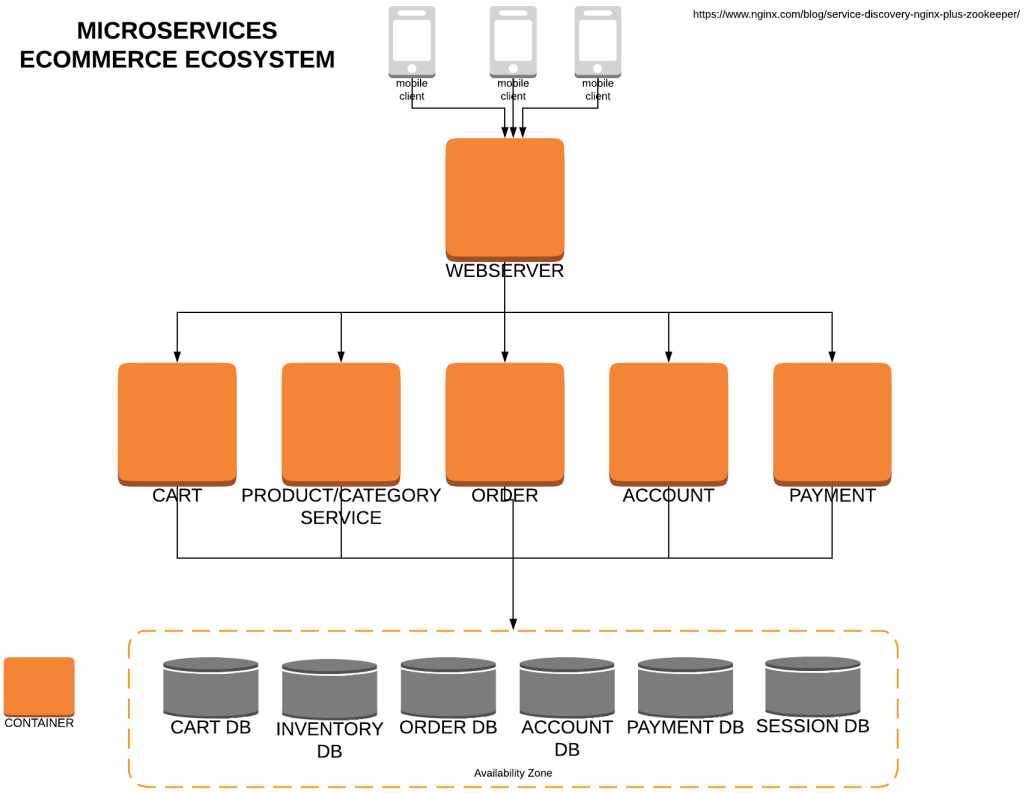
Conclusion

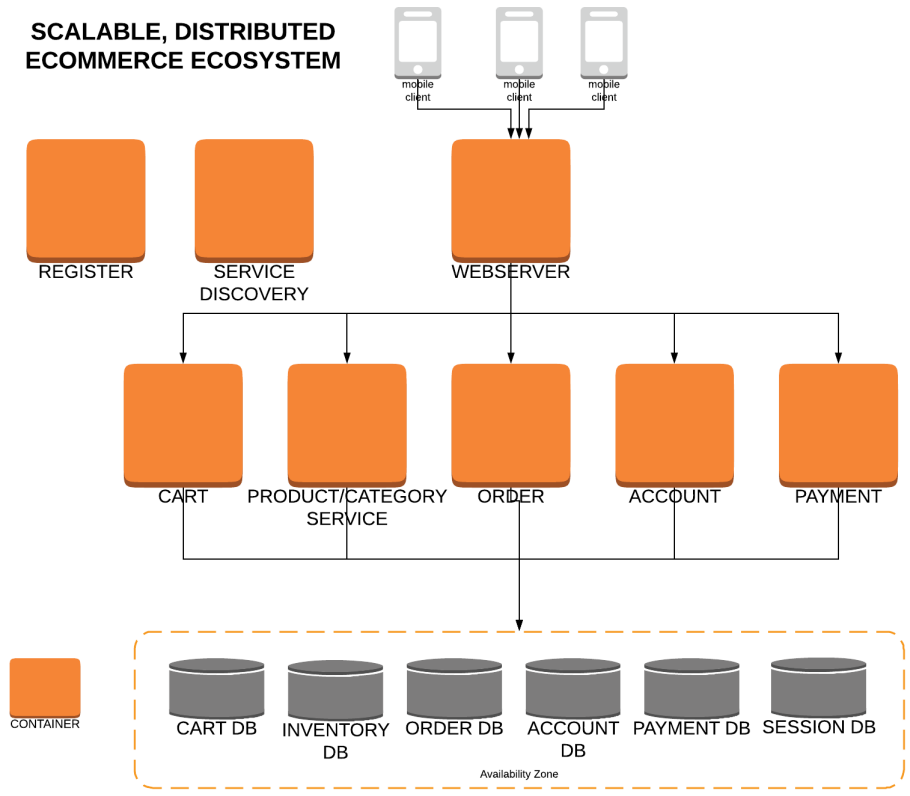
A distributed e-commerce backend architecture was implemented, tested, and deployed locally within a short time and scope. However, not of enough time was provided to deploy on the cloud, setup Registor, Nginx, Zookeeper, and Docker, so all testing was done on a local machine with the bare-bones components that can be found in the GitHub repository. Hypothetically, on a large-scale test, each service would be deployed on independent machines and would still work the same. Difference between our setup and the more advanced one is each service would have a different IP address. In our case, each service has different port numbers. In this hypothetical system, we can test performance parameters, such as latency and other tests, between the two designs and also achieve a deeper understanding of the values of microservices design at scale.

References

[1] https://medium.com/@helloansh/designing-scalable-backend-infrastructures-from-scratch-af80f5767ccc. *The Medium* Anshul Chauhan 2007.




MICROSERVICES
ECOMMERCE ECOSYSTEM
https://www.nginx.com/blog/service-discovery-nginx-plus-zookeeper/
mobile client
mobile client
mobile client
WEBSERVER
CART
PRODUCT/CATEGORY SERVICE
ORDER
ACCOUNT
PAYMENT
CONTAINER
CART DB
INVENTORY DB
ORDER DB
ACCOUNT DB
PAYMENT DB
SESSION DB
Availability Zone


SCALABLE, DISTRIBUTED
ECOMMERCE ECOSYSTEM
mobile client
mobile client
mobile client
REGISTER
SERVICE DISCOVERY
WEBSERVER
CART
PRODUCT/CATEGORY SERVICE
ORDER
ACCOUNT
PAYMENT
CONTAINER
CART DB
INVENTORY DB
ORDER DB
ACCOUNT DB
PAYMENT DB
SESSION DB
Availability Zone