UNIVERSITY OF CALIFORNIA
SANTA CRUZ

INHOUSE MICROSERVICES – INFRASTRUCTURE & APPLICATIONS

A project submitted in partial satisfaction
of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

UMANG SARDESAI

June 2018

The Project of Umang Sardesai
is approved:

_____

Professor Dr. Peter Alvaro, Chair

_____

Professor Dr. Charlie McDowell

ABSTRACT

The idea of this project is to create a mock-up of an industry standard microservice application. This application resembles the Cart/Checkout section of any ecommerce website. The microservices can be considered as black boxes and talk to each other only via REST API calls, to perform functions such as “Add to Cart” and “Place Order”. This application has additionally been integrated with tracing to generate call graphs and Netflix’s Hystrix library to achieve resilience and fault tolerance.

ACKNOWLEDGEMENTS

This project which was started in Fall 2017, would not be possible without the help of fellow colleagues and professors. Firstly, I would like to thank Professor Peter Alvaro for his giving me this amazing opportunity to develop this infrastructure from scratch. Building things from ground up has its own set of challenges and the learning experience is huge. I'm also grateful for his guidance and mentorship throughout the two quarters.

I would also like to thank Ashutosh Raina for his suggestions in designing the architecture and planning the steps from development to deployment. A big thank you to Tuan Tran for being together with me in this for the last 4 months and getting the tracing infrastructure ready. I'm grateful for Stephen Pinkerton for his tips on Service Discovery with Docker. Also, huge thanks to Ryan Jacobson for helping me with Hystrix and making sure the infrastructure is fault resistant. Last but not the least, I am appreciative of Professor Charlie McDowell for agreeing to be the reader of my project.

TABLE OF CONTENTS

INTRODUCTION

Distributed Systems lab (Disorderly Labs) at UC Santa Cruz for the past two years had been dependent on external microservice applications when it came to implementing their research and there was uncertainty as to when the results would be back. This school of thought lead to the idea of having an in-house microservice application which would give results immediately and have it designed such that it could suit the Lab's research.

One of the most common microservice applications prevalent in the industry is that of an ecommerce website. A simple checkout involves multiple microservices working together to return a successful response. Such an application was ideal for our lab.

The ecommerce application being implemented consists primarily of 5 main microservices, namely App, Inventory, Cart, Payment Gateway, and Receipt Generator. The App will act as an entry to the client's request. The Inventory microservice will be responsible for maintaining the Inventory database, whereas the Cart microservice as the name suggests deals with the Cart database. Payment Gateway is a service which responds to a payment request. Receipt Generator generates a receipt after the payment is processed.

Each of the above microservices are written in Spring Boot. Spring Boot is an application framework for Java used to write web applications. Each of these applications was containerized by Docker which makes the infrastructure easy to ship and deploy and makes it platform independent. Microservices are integrated with Hystrix, which can be viewed as a complex try-catch mechanism for Distributed Systems.

The microservices have also been integrated with Zipkin Open tracing. The tracing traces every call made from one microservice to another and records how long each microservice took to respond. There is an additional microservice called the config-server which was required primarily for Hystrix and to store the addresses of every microservice. Locally these services talk to each other using a Docker subnet. These services were deployed on Amazon's Elastic Compute Cloud (EC2) machines to imitate a production ready infrastructure and so that it could also be integrated with services like Chaos Monkey.

THE ARCHITECTURE

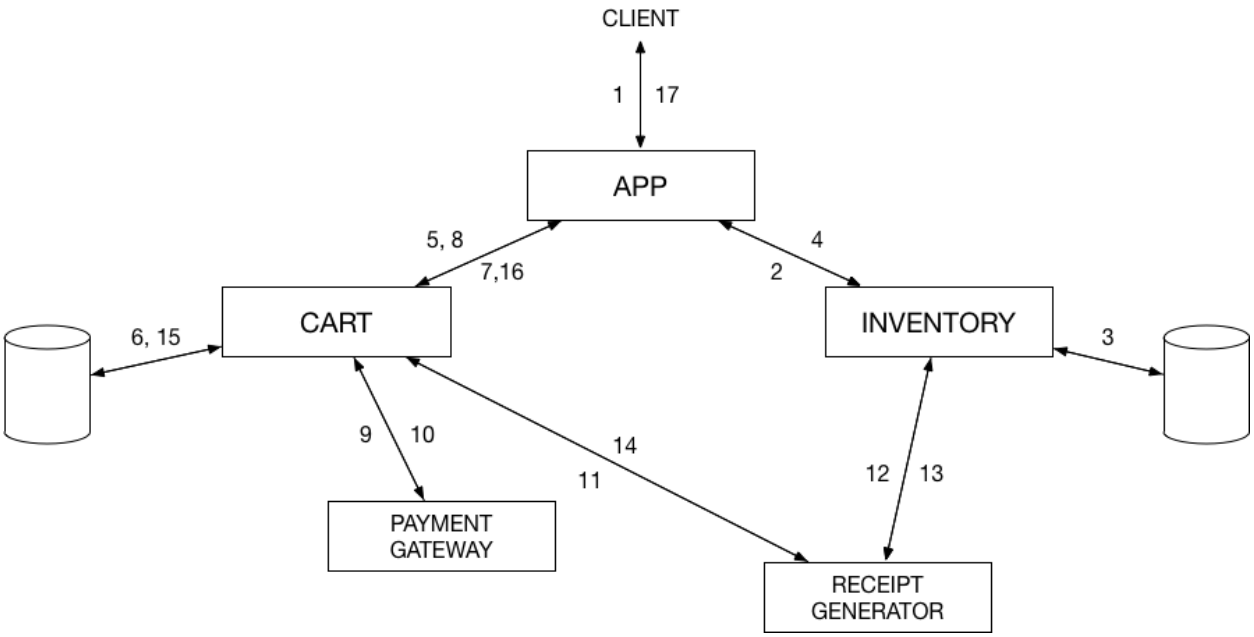

Figure 1

(The numbers are representative of the calls made from one microservice to another. Details regarding these calls are given under Workflow)

WHAT DOES EACH MICROSERVICE DO?

INVENTORY:

The major function of this service is to manage the inventory database. Any functionality that requires accessing the inventory database must do it via the Inventory microservice[1]. Currently, it is assumed that the client knows the names of the items in the inventory. If client sends an unknown name, Inventory would fail with a "Malformed JSON" exception.

API:

a) **GET /inventory/checkAvailibility**
   Query Parameter = Name[2]
   Returns the quantity currently available in the inventory for item name=<Name>

b) **GET /inventory/checkPrice**
   Query Parameter = Name
   Returns the price of the item with name=<Name>

c) **PUT /inventory/TakeFromInventory**
   Query Parameter = Name, Quantity
   First checks if quantity asked for, is less than or equal to currently available. If not, it will return "Not enough in inventory". If yes, it removes the specified quantity from Inventory database for item with name=<Name> and returns the total price for the item(s) removed.

d) **GET /inventory/getName**
   Query Parameter = ItemID
   Returns the name of the item with ItemID=<ItemID>

e) **PUT /inventory/addBackToInventory**
   Query Parameter = ItemID, quantity
   Increases the quantity of item with ItemID=<ItemID> by quantity=<quantity>. Returns success on successful update query.

---

[1] The return type for all microservices is "application/JSON". Whenever I say returns success/failure it is a JSON response having status parameter as success or failure like this: {"status" : "success"} or {"status" : "failure"}

[2] It is assumed that the client sends the correct name which exists in the inventory.

INVENTORY DATABASE:

The inventory database stores information about the items that are provided by the ecommerce app. The schema is as below:

1. ItemID – INT (primary key)
2. name – VARCHAR (50)
3. quantity – INT
4. price – DOUBLE

PAYMENT GATEWAY:

This is a custom payment processing service created, that always returns a success to any payment request made to it.

API:

a) **PUT /pg/makePayment**
Query Parameter = total_price
Returns success with a delay of 100ms

RECEIPT GENERATOR:

The purpose of this microservice is to generate a receipt after the payment is processed. Successful generation of the receipt indicates the order was processed successfully. Although the API endpoints use the word 'invoice', we must consider that it is something that is generated after payment is processed.

API:

a) POST /invoice/<userid>/generateInvoice
This microservice will first fetch the all the items present in cart for user with ID=<userid> by calling "/cart/<userid>/getCartItems". Then using the ItemIDs, it fetches the respective name of the item using "/inventory/getName". All the purchases (Item name and price) are stored in a String variable. You then return success.

b) GET /invoice/<userid>/getInvoice

Return Type: String
Returns the String created in "/invoice/<userid>/generateInvoice"**.** If the invoice String variable is null, then return "Invoice not generated yet"

CART:

This microservice is the core part of this infrastructure since it communicates with all other microservices. It has two important functions. One of the functions, is to maintain the Cart database. Secondly, it communicates with the Payment Gateway and Receipt Generator and holds the logic to process the order.

API:

a) **POST /cart/<userid>/addToCart:**
Query parameter: ItemID, quantity, total_price
Makes an insert query to cart with the data in the query parameters for user with ID=<userid>. Returns success after a successful insertion.

b) **DELETE /cart/<userid>/emptyCart:**
Deletes everything present in cart for user with ID=<userid>. Returns success after successful deletion.

c) **GET /cart/<userid>/getCartItems:**
Return all the items in the cart for user with ID=<userid>. The select query returns a List of "Cart" items which is automatically converted to JSON by the Spring framework.

d) **PUT /cart/<userid>/undoCart:**
This resembles "Remove from cart" function. First it checks if there are any items in Cart for user with ID=<userid>. If not, it returns a "No items in cart" response. If yes, it empties the cart for user with ID=<userid> using "cart/<userid>/emptyCart" request and adds back all the removed items to Inventory using multiple "inventory/addBackToInventory" requests. Once all items are successfully added, it returns a success response.

e) **PUT /cart/<userid>/placeOrder:**
Takes all the items from Cart for user with ID=<userid> and processes them to place the order. First it checks if there are any items in Cart for user with ID=<userid>. If not, it returns a "No items in cart" response. If yes, this microservice first gathers the total price of all items currently present in cart for user with ID=<userid>. With the total price, a request is made to the

payment gateway (“/pg/makePayment”) for processing the payment. Then, it sends a request to get a receipt by making a call to “/invoice/<userid>/generateInvoice”. Lastly, after receipt generation, all the items in the cart for user with ID=<userid> are deleted using “cart/<userid>/emptyCart”. Post successful deletion, a success message is returned.

CART DATABASE:

The cart database stores all the items that have been added to the cart. Catalog’s primary key is the foreign key for Cart. The total price for every entry is Item price multiplied by the quantity. The schema is as below:

1. UserID – VARCHAR (50)
2. ItemID – INT,
3. quantity – INT,
4. total_price – DOUBLE

APP:

This is the gateway to the client’s requests. This microservice serves the function of directing requests to either the Cart or the Inventory microservice as appropriate. For example, to take items from the inventory, the client will talk to the App and the App will then forward the request to the Inventory. As seen in the above diagram, all the major calls start here, move around the application and then get back to the app and finally to the client. Most of the API endpoints given by app can be considered as **end-to-end use cases**.

API:

a) **PUT /app/<userid>/addToCart**
   Query Parameter = Name, Quantity
   Adding to cart consists of two parts. The first (“app/takeFromInventory”) is explained above. If this returns failure, it would return that response back to client. If it is a success, app makes a call to “/cart/<userid>/addToCart”. It returns the response of this call back to client.

b) **PUT /app/<userid>/placeOrder**
   This simply makes a call to “/cart/<userid>/PlaceOrder” and return the response back to client.

c) PUT /app/<userid>/instantPlaceOrder

Query Parameter = Name, Quantity

This call not only adds to cart but also places order at the same time. It first calls "/app/<userid>/addToCart". If this fails, it returns the response to the client. If this returns success, App calls "/app/placeOrder" and returns the response to that request back to the client.

d) PUT /app/<userid>/undoCart

This simply makes a call to "cart/UndoCart" and returns the response to the client.

FUNCTIONALITIES AND WORKFLOW

This following subsection briefs us over the use-cases (functionalities) supported by the application. The last use-case (instant Place Order) involves all the 5 microservices and is explained in detail in end-to-end workflow.

FUNCTIONALITIES:

a) **Adding an item to Cart:**
This starts with client sending a request to App to add an item to Cart, by specifying the item name and the quantity. App first calls Inventory to take out the item specified, out of the inventory. Based on the quantity available in the inventory, the Inventory replies "Success" or "Failure". If successful, App then goes ahead and calls Cart to add the item to its database. Once the item is successfully added, Cart returns "Success" which is routed back to Client.

b) **Placing an order:**
This functionality involves placing an order for all the items currently present in the cart. The user makes the request to place order to App. App forwards the request to Cart. Cart calculates the total cost of the items present in the database. Cart then makes a request to Payment Gateway to process this payment. If the payment was unsuccessful, Cart returns "failure" as a response to App. If payment was successful, Cart sends a request to Receipt Generator to create a receipt of the order. When Receipt generator returns success, Cart deletes all the items present in the database for that user and returns "Success" to App. App returns the

c) **Removing items from Cart:**
This removes all the items present in the cart for the user. This starts by user sending a request to App. App forwards this request to Cart. Cart first removes all the items from the database for that user. On successful deletion, Cart sends multiple requests to Inventory to add the items back to inventory. If any Add back to Cart request fails, the item is inserted by in Cart Database to restore Atomicity.

d) **Instantly placing an order in one request:**
The app supports adding to cart and placing order in one request itself. This call extends to all five microservices and is explained in detail in the next section.

END-TO-END WORKFLOW:

Given below is the workflow of the application using a sample use case of “instant Place order” request, which performs an “Add to Cart” followed by “Place Order”. The numbers are a reference to the calls seen in Figure 1.

1. Client sends an “instantPlaceOrder” request with name of the item and the quantity required.

2. Client sends a “takefromInventory” request to Inventory.

3. Inventory checks with its database to see if the required quantity is available.

4. If unavailable, Inventory replies with “Not enough in inventory”. If available, it deducts the required quantity from its inventory and returns success.

5. App makes an “AddToCart” request to Cart.

6. Cart makes an insert query to Cart database.

7. The response is returned to App.

8. Since this is an instant Place Order, the app makes a “Place Order” once the “AddToCart” returns successfully.

9. Cart then fetches the total price of all items from the cart and sends a payment request to the Payment Gateway.

10. Payment Gateway just responds with a success response.

11. Cart then makes a request to cart to generate a receipt.

12. To generate the receipt, Receipt Generator needs to know item names and hence makes a request to Inventory to fetch the names of items that have

been ordered.

13. Inventory responds back with the name list.

14. Receipt Generator generates the receipt and returns success.

15. On receiving a success from Receipt Generator, Cart deletes all the items present in the Cart database.

16. On successful deletion, Cart returns Receipt Generator’s response back to App.

17. App forwards the response to the client.

TOOLS

Given below is brief explanation of all the tools used to get this infrastructure up and running. The source code for the application can be found here: https://github.com/disorderlylabs/DisorderlyLabs-Microservice-Infrastructure, and can be used as a reference while going through the below tools:

SPRING BOOT:
Spring Boot is an application framework for Java used to write web applications. Spring Boot forms the core part of the project as it is used to develop the individual microservices. The structure used to code every microservice can be divided into three parts as given below:

a) Repositories: This contains the Java class that can be mapped to the database/repository. This class also has the getter setter functions.

b) Mappers: This class implements the RowMapper interface. The JDBC template uses the Controller class, to interact with the database and needs RowMapper to map the data fields of the repository class into the columns in the database.

c) Controller: The class where all the API endpoints are defined. It uses Rest Template to make HTTP requests to other microservices. It uses Java's prominent JDBC template to communicate with its respective database.

H2 DATABASE/MYSQL DATABASE:
H2 is an embedded database which is used by default in Spring Boot. The schema and any initial data that should be loaded into the database, can be specified in "src/main/resources" under "application.properties". To talk to an external MySQL server, you need to give the MySQL URL in the "application.properties" and specify a couple of additional dependencies in the "build.gradle".

GRADLE:
Gradle helps you manage all the package dependencies in Java. All the dependencies are mentioned in a "build.gradle" file.

DOCKER:

a) Why Docker containers?
The documentation page[3] on Docker says – "A container image is a lightweight, stand-alone, executable package of a piece of software that includes everything needed to run it: code, runtime, system tools, system libraries, settings. Available for both Linux and Windows based apps, containerized software will always run the same, regardless of the environment". Containers isolate software from its surroundings, for example you can expect the Docker container to give the same result on a Mac OS and on Windows.

b) Docker's role in the application:
The idea is to wrap the Spring Boot applications using Docker and convert them to containers. Spotify provides a neat open source plugin to containerize any application which has been built using Gradle[4]. Each Docker container would act as microservice sending and responding to API calls.

HYSTRIX:

a) What is Hystrix?
Hystrix has been beautifully described in this blog post[5] by Baeldung - "A typical distributed system consists of many services collaborating together. These services are prone to failure or delayed responses. If a service fails it may impact on other services affecting performance and possibly making other parts of application inaccessible or in the worst case bring down the whole application. Of course, there are solutions available that help make applications resilient and fault tolerant – one such framework is Hystrix."

Hystrix controls the interaction between microservices by providing fault tolerance and latency tolerance. A simple example of what Hystrix can do is, if a certain instance of a microservice is down, it will divert all calls to its backup thus maintaining Fault resilience.

[3] https://www.docker.com/what-container
[4] https://github.com/palantir/gradle-docker
[5] http://www.baeldung.com/introduction-to-hystrix

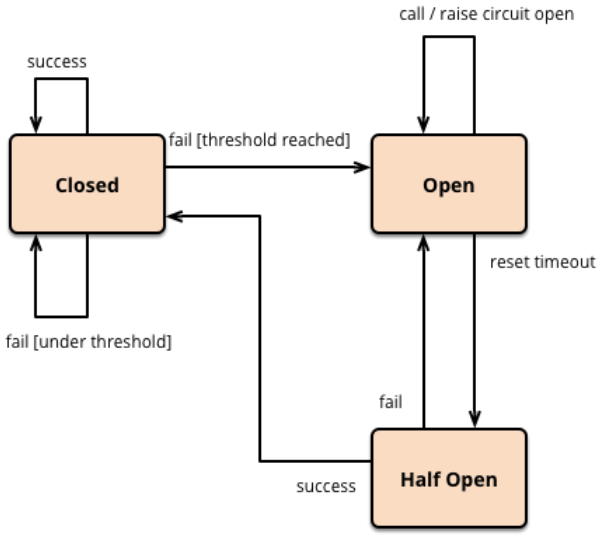

Figure 3[6]

b) Hystrix as a Circuit Breaker:
Figure 3 contains the various stages of the circuit (connection) between two microservices. Consider them to be App and Inventory. If Hystrix wasn't implemented, and if there was a timeout at Inventory, then the App would crash. Here is how Hystrix would help:

1. For starts, the circuit is closed i.e. the connection between the two microservices is working.
2. If there is failure/timeout, Hystrix keeps a tap of the such disconnections happening. However, it will keep the circuit closed since the number of disconnections/timeouts are below the threshold.
3. Hystrix will keep on retrying and once the timeouts cross the threshold, the circuit is opened.
4. When the circuit is open, you can choose what should be done. You can either have a template response saying "Inventory is down" or have Hystrix call the back-up instance in the fall back part of the code.
5. After certain amount of time, Hystrix will again try to call the Inventory to see if the connection is fixed or not.
6. If yes, then it will close the circuit. If not, then it will reset the time limit and keep the circuit open.

[6] https://martinfowler.com/bliki/CircuitBreaker.html

ZIPKIN OPEN TRACING:

In a distributed application where various microservices interact across the network, capturing system behavior to verify correctness is a non-trivial task, a task that can be simplified with the help of a distributed tracing framework. We used OpenZipkin to facilitate both distributed tracing for our application as well to perform fault injection. Zipkin, like many modern distributed tracing frameworks, was inspired by Dapper, and follows the same naming convention and workflow.

Trace collection can be facilitated by launching a Dockerized instance of the Zipkin server locally or at a remote location, and configure the application to report traces to this server. For each request, a span gets constructed per service invoked, and these spans will be sent asynchronously to the Zipkin server as each service finishes. The parent-relationship metadata within these spans that gets propagated through the network, in the form of key-value pairs inside request headers, allows Zipkin to reconstruct the lineage graphs for the requests. One of the goals of this project is to provide tracing and fault injection with minimal changes to the source code for the microservices, and the Spring framework allows us to achieve this goal using interceptors; this goal can also be achieved for languages that allows for function decorators.

DEPLOYMENT

You can have these services running locally or on the cloud (AWS). Describing both methods below:

LOCAL DEPLOYMENT:

1. You create a Docker subnet (using "docker create network" command) so that the containers talk to each other within the subnet.
2. Every Docker container is given an IP.
3. A ENV variable file having the IPs of all the containers is given to all the containers so that they know which IP address to contact when calling a microservice.
4. Alternatively, you can have config server as an additional microservice which stores the IP addresses of the microservices. This config server is itself Dockerized. The services on bootup contact the config server to retrieve the IPs of rest of the microservices.
5. The databases are in embedded form (H2 database) when running the services on a local environment. Here crashing of either the Cart or Inventory would also mean that the Cart and Catalog database respectively, are down.

DEPLOYMENT ON AWS:

1. T2.micro EC2 instances were used to deploy the Docker containers.
2. These were bare Linux images and Docker had to be installed to get the containers running.
3. Since these are EC2 separate from one another, you need not have a subnet for services to communicate with each other.
4. Like Local deployment, you share the IP addresses of the nodes via a ENV variable file which is provided to containers while running them or via an external config server. This external config server would be a container in itself.
5. On EC2, each SQL server is not embedded and is separate from its microservices. Hence, the two database servers (MySQL servers) are installed an EC2 instance each.

RESULTS

If the all installations have been made correctly, the microservice application should be up-and-running. The image below is a successful call graph for instant Place order as shown on the Zipkin dashboard.

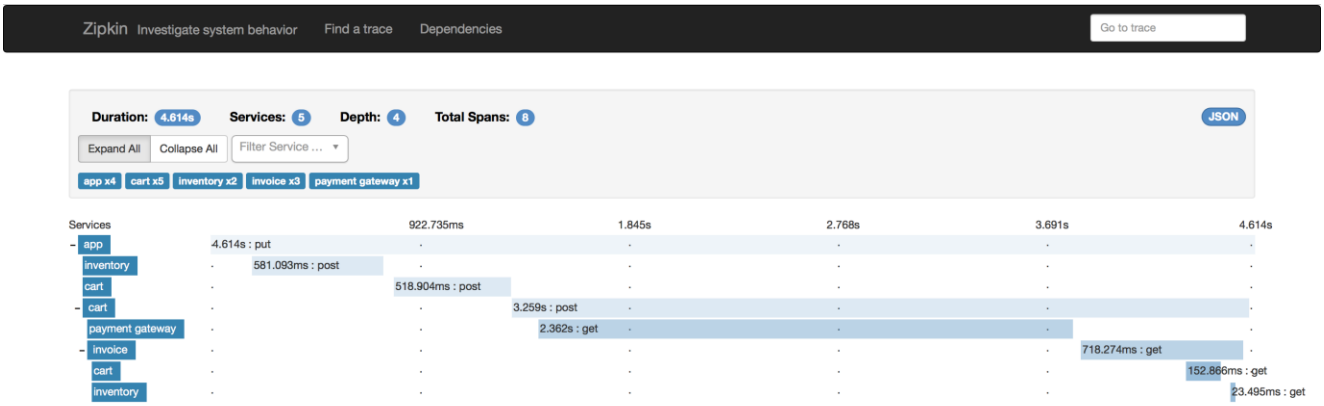

Figure 4

FUTURE WORK

1. As you have noticed, the IP addresses were loaded in a ENV file for every container or into the config server. This was done manually every time, the microservices were booted. The next step would be to implement service discovery where even if the IP addresses get dynamically changed, the microservices can fetch them via another service. The config server mentioned is not service discovery, as the IP addresses must be manually fed into the config server. In Service Discovery, the services are discovered automatically.

2. In our current setup, the App can be a single point of failure. One can implement a load balancer or something like HA proxy to make sure all the requests to the ecommerce app is via the load balancer or HA proxy.