**Group Members:**

2019B4A70652P Ayush Agarwal

2020A7PS0116P Ansh Gupta

2020A7PS0024P Nachiket Kotalwar

# Project 1: EBankingSystem (Load Balancer & AutoScaler)

## Introduction:

## Tech Stack

- Docker
- Java Oracle OpenJDK 17
- Jdk-17.0.8
- Intellij Ultimate Text Editer
- MySQL Database
- Maven (mvn)

## Setup / Usage

1 Clone the repository
- git clone https://github.com/CloudComputingBITSProject/EBankingSystem

2 Check your JDK Version and Java Version.
I used Oracle OpenJDK 17 for Java and jdk-17.0.8 as my present JDK for this project.
Also please prefer Intellij to VSCode, as the former was used for creating this code.

To run:

1. Install all the dependencies in pom.xml

2. Run the file src/main/java/com/example/loadbalancer/LoadBalancerApplication.java
OR
2. Run the backend

``` mvn clean package```

```java -jar target/load_balancer.war```

(This is the main file of the program)

3. Run the frontend on a different terminal from

src/main/java/com/example/loadbalancer/service/Runner.java

## Code Structure

File Structure
|-- EBankingSystems1  # Service 1 of the Microservice
|-- EBankingSystems2  # Service 2 of the Microservice
|-- EBankingSystems3  # Service 3 of the Microservice
|-- EBankingSystems4  # Service 4 of the Microservice
|-- EBankingSystems5  # Service 5 of the Microservice
|-- docker  # For database storage of MySQL containers
|-- README.md  # Description of the Setup to Run the Code and File Structure.
|-- Documentation.txt # Complete Analysis of the Code with respect to Cloud Computing.
|-- src/main
  |--java
    |—-com.example.loadbalancer
      |-- controller
        |--Service1Controller  #Controller Routes for Service 1
        |--Service2Controller  #Controller Routes for Service 2
        |--Service3Controller  #Controller Routes for Service 3
        |--Service4Controller  #Controller Routes for Service 4
        |--Service5Controller  #Controller Routes for Service 5
        |--SettingsController  #Controller for Frontend to start/stop services, set properties of LoadBalancer and Autoscaler.

      |-- service
        |--AdminAgent  #Controller Routes for Service 1
        |--DockerAgent  #Controller Routes for Service 2
        |--SQLAgent  #Controller Routes for Service 1
        |--RedirectService  #Controller Routes for Service 2
        |--User  #Controller Routes for Service 1

|--Runner  #Controller Routes for Service 2

|-- pom.xml #Dependencies and other build properties
|-- user_data.txt  #For storing multi-tenant based authentication service.

## API Services

## http://localhost:8080/<service_name>/<api_name>?company=root

## Service name can be anything like service1

## API name: (get-all-users) HTTP GET:

## API name: (add-user/{id}) HTTP POST:

Examples:

## http://localhost:8080/service1/get-all-users?company=root

## http://localhost:8080/service1/add-user/231?company=root

## Theory

Load Balancer

A common instance of this scenario occurs when with one load-balancing instance node.

The algorithm is referred to in the literature as “power of two choices”, because it was first described in Michael Mitzenmacher’s 1996 dissertation, **The Power of Two Choices in Randomized Load Balancing**. it’s implemented as a variation of the Random load-balancing algorithm, so we also refer to it as **Random with Two Choices**.

# IP Hash

The hash algorithm uses the IP address of the client or the value of an HTTP header as the basis for server selection.

With an HTTP header, use the Load Balancer Hash Header property to identify the header to read. When you configure a service without the wizard, this property is available on the Main tab.

With the hash algorithm, the same client is served by the same server. Use this algorithm only when clients access applications that require the storage of server-side state information, such as cookies. Hashing algorithms cannot ensure even distribution.

Here we hash IP of both host and destination server.

**RANDOM LOAD BALANCER**

A random load balancer distributes incoming network traffic randomly among multiple servers to ensure even utilization and prevent any single server from being overwhelmed.

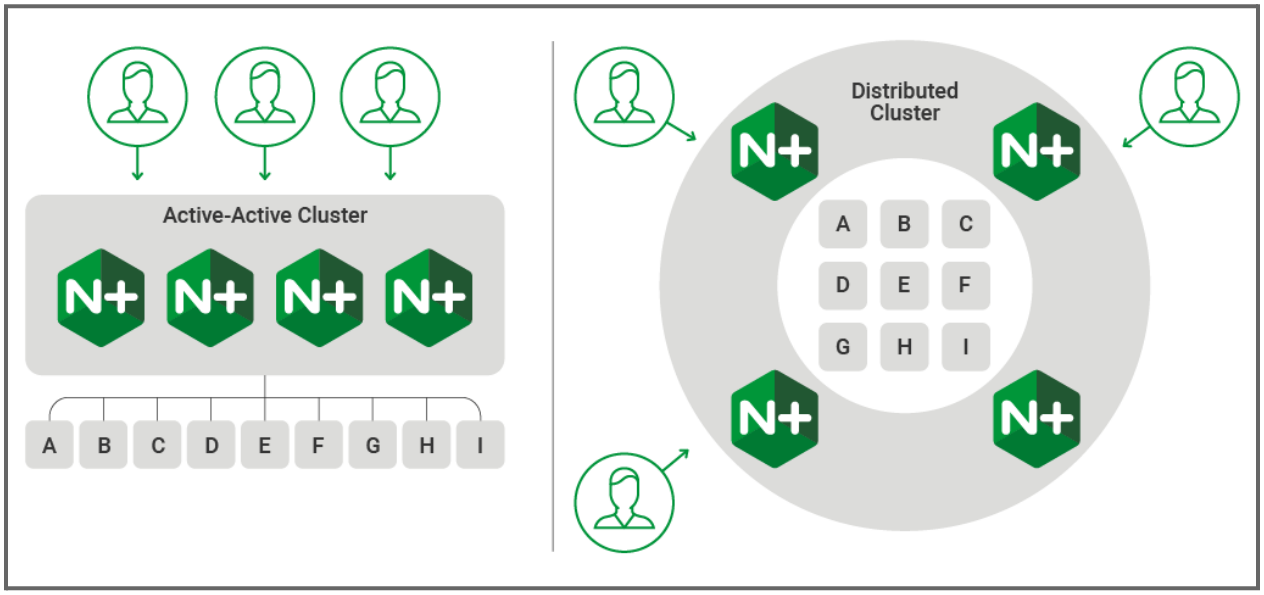
**POWER OF 2 LOAD BALANCER**

A Power of 2 Load Balancer is a type of load balancing algorithm that distributes incoming requests among servers in powers of two. For example, it might route traffic to 2, 4, 8, or 16 servers. This approach simplifies the process and allows for efficient scaling of server resources.

Classic load-balancing methods such as **Least Connections** work very well when you operate a single active load balancer which maintains a complete view of the state of the load-balanced nodes. The “power of two choices” approach is not as effective on a single load balancer, but it deftly avoids the bad-case “herd behavior” that can occur when you scale out to a number of independent load balancers.

This scenario is not just observed when you scale out in high-performance environments;

it's also observed in containerized environments where multiple proxies each load balance traffic to the same set of service instances.




*Cluster topologies using distributed load balancers*

## How Does "Power of Two Choices" Work?

Let's begin with what might be a familiar situation. You've just landed after a long international flight, and along with 400 other travelers, have walked into a busy arrivals hall.

Many airports employ guides in the arrivals hall. Their job is to direct each traveler to join one of the several queues for each immigration desk. If we think of the guides as load balancers:

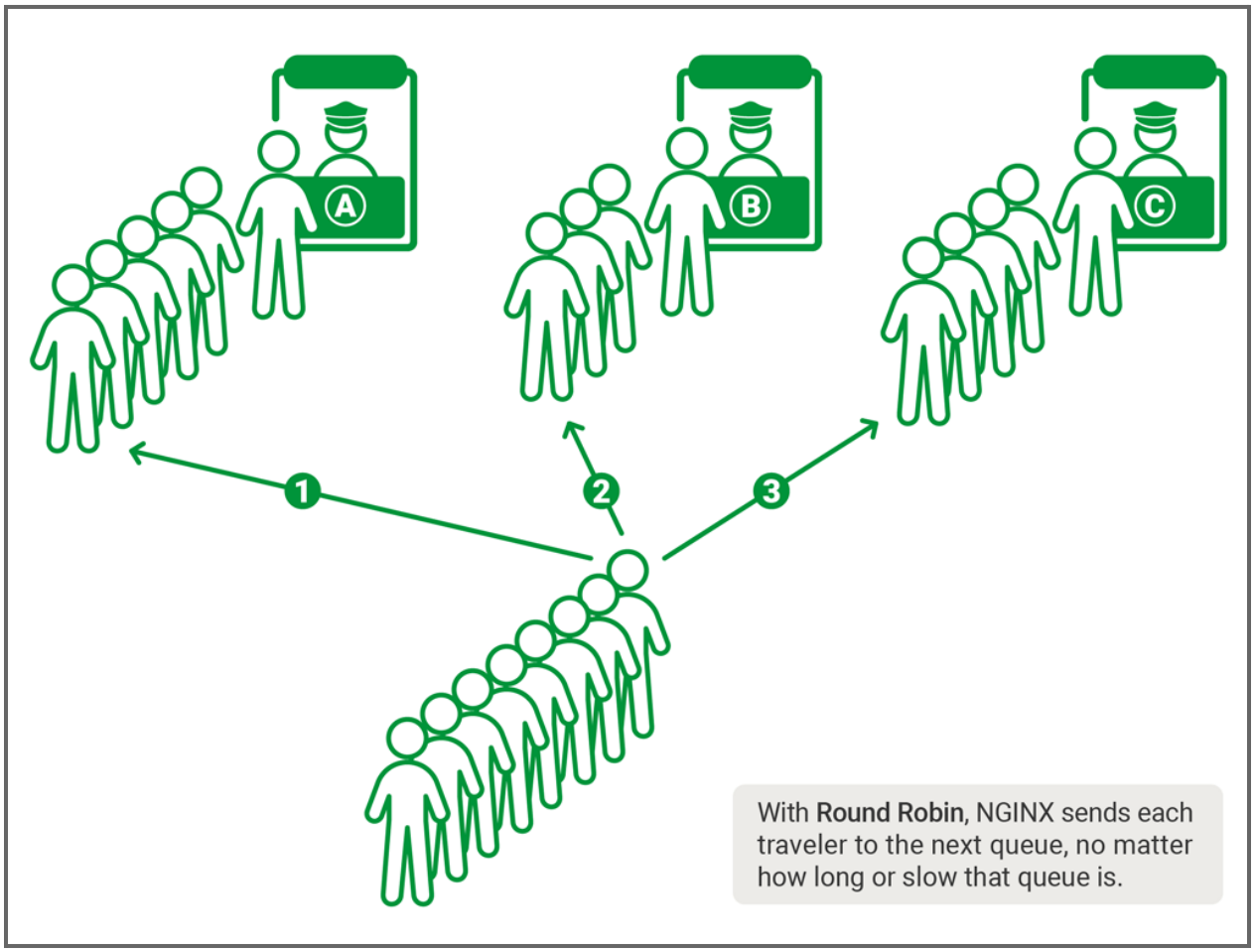
- You and your fellow travelers are *requests*, each hoping to be processed as quickly as possible.
- The immigration desks are (backend) *servers*, each processing a backlog of requests.
- The guides maximize efficency by selecting the best server for each request.

- The methods available to the guides for selecting the best server correspond to load-balancing algorithms.

Let's consider how well some of the possible algorithms work in a distributed load-balancing scenario like the arrivals hall.

### Round-Robin Load Balancing

Round robin is a naive approach to load balancing. In this approach, the guide selects each queue in rotation – the first traveler is directed to queue A, next traveler to queue B, and so on. Once a traveler is directed to the last queue, the process repeats from queue:



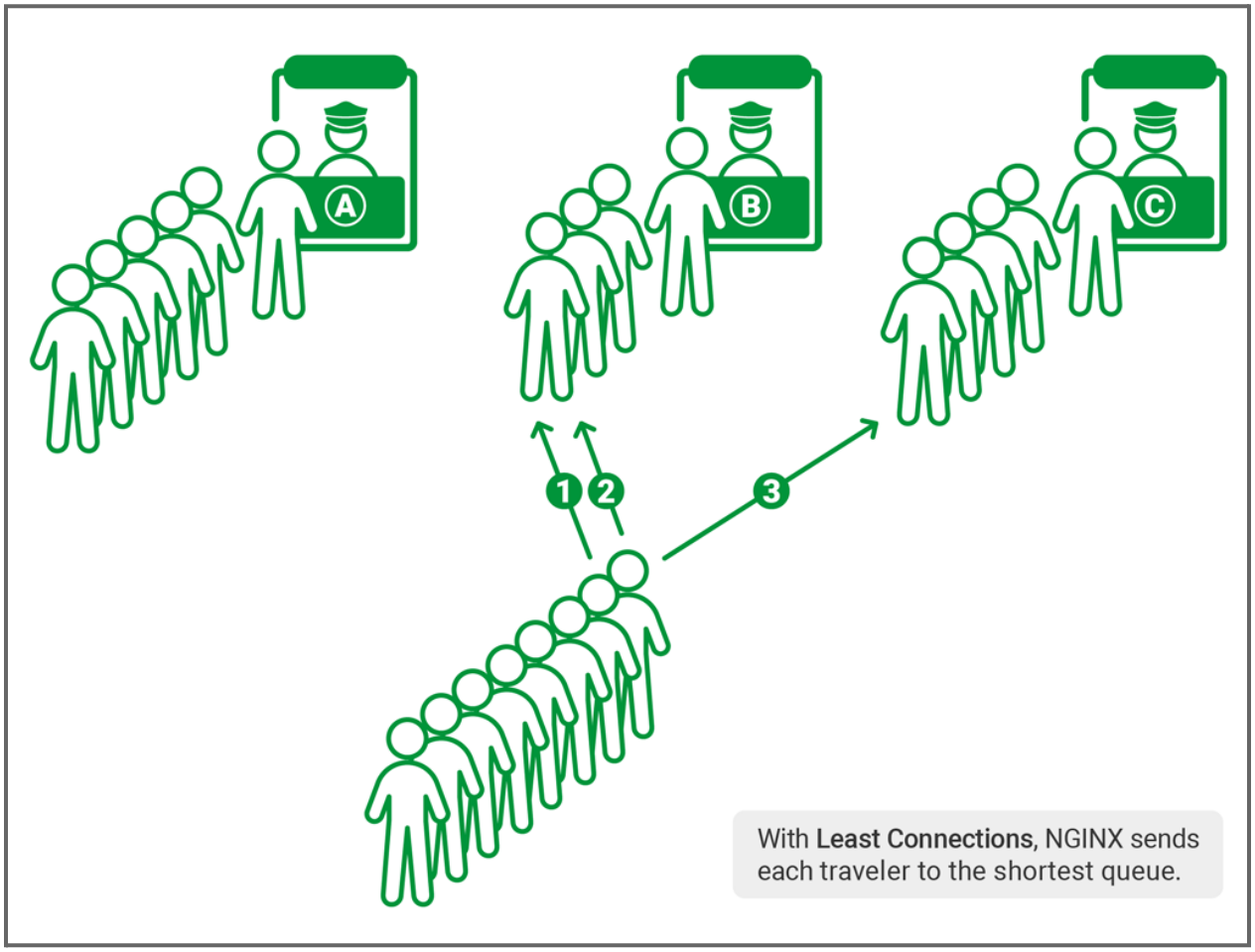

This approach works adequately, until there's a delay in one of the queues. Perhaps one traveler has misplaced his or her documentation, or arouses suspicion in the immigration

officer:

The queue stops moving, yet the guide continues to assign travelers to that queue. The backlog gets longer and longer – that's not going to make the impatient travelers any happier!

### Weighted Least Connections Load Balancing

There's a much better approach. The guide watches each queue, and each time a traveler arrives, he sends that traveler to the shortest queue. This method is analogous to the **Least Connections** load-balancing method in NGINX, which assigns each new request to the server with the fewest outstanding (queued) requests:

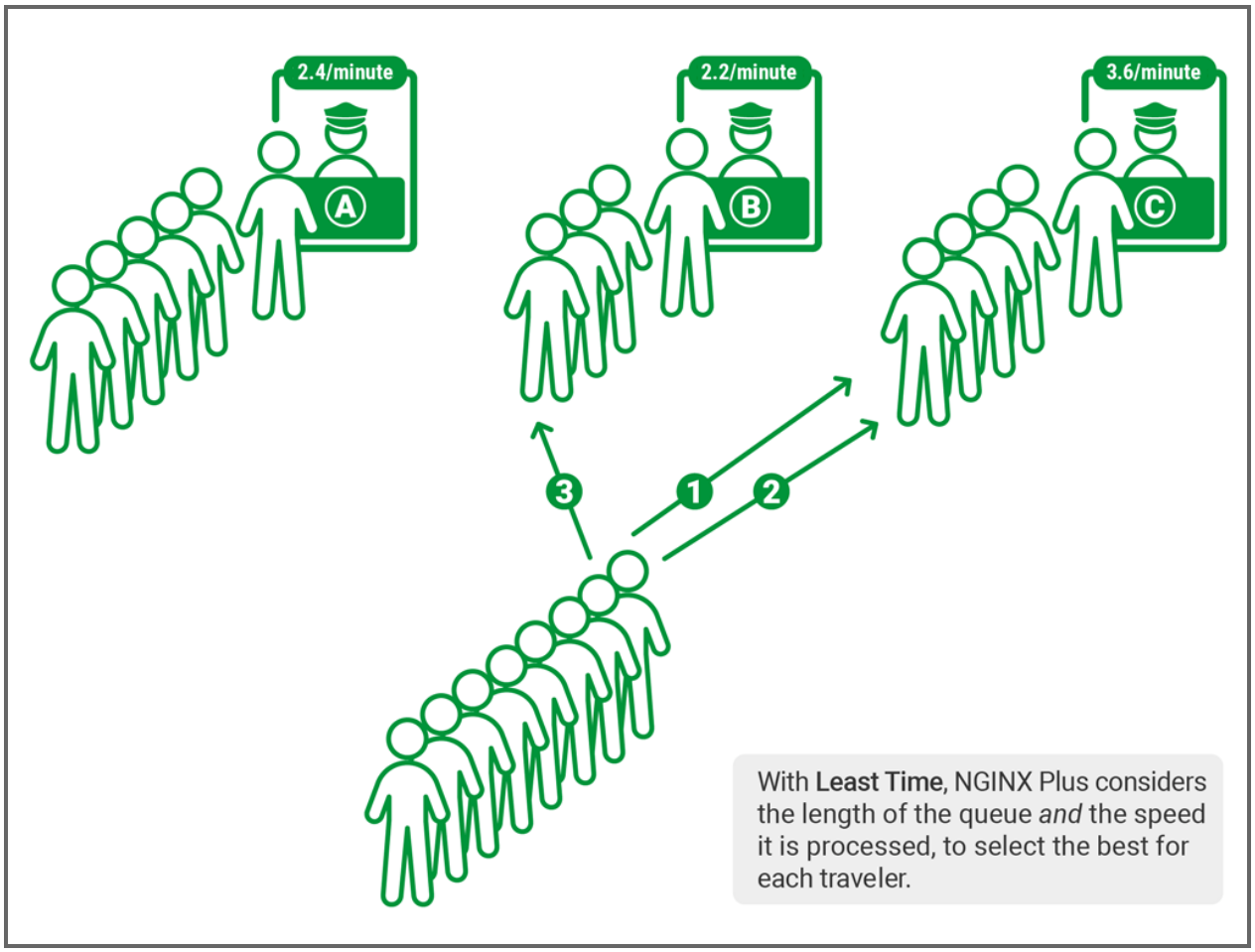


Least Connections load balancing deals quite effectively with travelers who take different amounts of time to process. It seeks to balance the lengths of the queues, and avoids adding more requests to a queue that has stalled.

## Weighted Least Time Load Balancing

We've seen that different passengers take different times to process; in addition, some queues are processed faster or slower than others. For example, one immigration officer might have computer problems which means he processes travelers more slowly; another officer might be a stickler for detail, quizzing travelers very closely. Other officers might be very experienced and able to process travelers more quickly.

What if each immigration booth has a counter above it, indicating how many travelers have been processed in, for example, the last 10 minutes? Then the guide can direct travelers to a queue based on its length and how quickly it is being processed. That's a more effective way to distribute load, and it's what the **Least Time** load-balancing is




This algorithm is specific to NGINX Plus because it relies on additional data collected with NGINX Plus's **Extended Status** metrics. It's particularly effective in cloud or virtual environments where the latency to each server can vary unpredictably, so queue length alone is not sufficient to estimate the delay.

### Autoscaler

### Threshold Based Autoscaler

A threshold-based autoscaler is a component of a system that helps manage the allocation of computational resources dynamically in response to changing workload conditions.

Here's a more detailed explanation:

1. **Metrics Monitoring:** The autoscaler continuously monitors various performance metrics of the system, such as CPU utilization, memory usage, or network traffic. These metrics serve as indicators of the system's current workload and performance.

2. **Thresholds Setting:** Users define specific thresholds for these metrics. For example, if the CPU utilization surpasses 80%, it may trigger a scale-up event. Conversely, if it drops below 30%, it could trigger a scale-down event. These thresholds are critical decision points for the autoscaler.

3. **Scaling Actions:** When the monitored metrics breach the defined thresholds, the autoscaler initiates scaling actions. Scaling actions can include adding more servers or resources (scale-up) when demand is high or reducing resources (scale-down) during periods of low demand.

4. **Efficient Resource Allocation:** The goal of threshold-based autoscaling is to ensure that the system always has sufficient resources to handle the workload effectively without over-provisioning resources when they are not needed. This dynamic adjustment optimizes resource utilization and improves cost efficiency.

5. **Automation:** The process is automated, eliminating the need for manual intervention. This allows the system to adapt to varying workloads in real-time, providing responsiveness and agility to changing conditions.

6. **Flexibility:** Users can tailor the thresholds based on the specific requirements and characteristics of their applications. This flexibility enables customization to accommodate different performance priorities and objectives.

In summary, a threshold-based autoscaler is a crucial component in cloud computing and server management systems. By automatically adjusting resources in response to predefined thresholds, it helps maintain optimal performance, minimize costs, and enhance the overall efficiency of a computing environment.

**Time Series Based Autoscaler**

A Time Series AutoScaler is a system that dynamically adjusts computing resources based on time series data. Time series data, in this context, typically represents the

historical behavior of a metric over time, such as CPU utilization or incoming request rates.

The primary purpose is to efficiently manage resources in response to changing workload patterns, ensuring optimal performance, cost-effectiveness, and scalability.

- Time series autoscaling relies on historical metric data. This could be collected from various sources, including monitoring tools, and it serves as the basis for decision-making.

- **Thresholds:**
  - Upper and lower thresholds are defined based on acceptable performance ranges. These thresholds guide the autoscaler in determining when to scale resources.

- **Scaler Logic:**
  - The autoscaler contains logic to analyze time series data, compare it against thresholds, and trigger scaling actions accordingly.

- **Scaling Actions:**
  - Scaling actions involve dynamically adjusting the number of resources. This could include scaling up (adding more servers or resources) or scaling down (removing excess resources).

**4. Workflow:**
- **Data Analysis:**
  - The autoscaler continuously analyzes time series data to understand the current state of the system. This involves considering trends, seasonality, and patterns in the data.

- **Threshold Evaluation:**
  - The autoscaler evaluates whether the current metric values breach predefined thresholds. For instance, it may scale up if the CPU utilization exceeds a certain percentage.

- **Decision Making:**
  - Decisions are made based on the comparison of current metrics against thresholds. If the system is under heavy load, scaling up may be triggered; if it's underutilized, scaling down may occur.

- **Scaling Actions:**
  - Upon decision-making, the autoscaler initiates scaling actions. These actions interact with the underlying infrastructure to modify the number of resources.

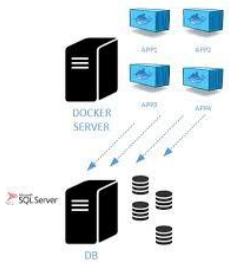
- **Feedback Loop:**
  - Time series autoscaling often operates as a feedback loop. After scaling, the system continues to monitor metrics, and the cycle repeats as needed.

## Work flow

First start the backend
For the frontend, From the Runner.py

1. First we build all the images.
2. Then we run the SQL server for the backend
3. Start a service
4. Select Load Balancer and Autoscaler Strategies

Through postman send a request to




## Assumptions

User is only able to start a server with 2 instances and does not have a choice to scale manually.

# Project 2: Object Storage System

## Tech Stack

- Docker
- Express.js

All of the data is stored locally in the file system of the server.

## API Server:

**Endpoints**:

1. /upload : used for uploading any given files < 1MB in size.
2. /files/{filename} : used to retrieve a particular file given the filename

**Functions**:

This microservice primarily interacts with the user and uses the distributed data servers.

1. Upon receiving the Upload request to retrieve the availability and vector clock information of the given file the microservice sends a POST request to a random data node. The file is converted to base64, and sent to the dataservers.
2. Similarly upon receiving a get File request the request is redirected to a random data node. If the data node returns an array of objects with the same filename but different vector clocks i.e different versions then a vector clock containing the maximal of all versions is taken and an appropriate file is returned.

## Data Servers:

Workflow:

The API server receives the user request and forwards it randomly to on of the dataservers. The dataserver checks if the node belongs in its range using consistent hashing. If yes, it handles the request as the coordinator, otherwise it is forwarded to the first node in the nodeset of the key. Nodeset of the node is the set of nodes which are responsible for that key.

**Endpoints**:

1. /getFile(key)
2. /putFile(key, data, context)

Functions:

1. The getFile API returns the an array of objects. Each object contains the data and the corresponding vector clock. Multiple objects are sent in case of conflicting vector clocks.
2. The putFile API stores the object on the data servers with the given context.
   If context is older than the latest version available on the coordinator server, put does not happen, and user is told that a newer version already exists. User can read that version by doing a getFile.

Features Implemented ->

1. Vector clocks and version conflict resolution
   a. All files stored have vector clocks associated with them. While doing a get, automatically latest version is sent to the user. In case of conflicting versions, all of them are sent in an array to the user. User can then do a put operation to resolve the conflicts.
2. File Replication
   a. All of the files are replicated across N servers.
3. Read Write Quorums
   a. When processing read or writes, the server returns successfully when atleast R or W results are returned.
4. Read Repair which helps in anti-entropy
   a. In case of reads, if it is noticed that any node has an older version of the data, we update their version.
5. Consistent Hashing for nodes
   a. We have implemented consistent hashing on a 64 bit ID space. For each key, the algorithm returns a set of N distinct nodes, which are responsible for the replication.

Configurable paramters -

1. Number of nodes
2. Number of replicas (N)
3. Number of successful reads required to return (R)
4. Number of successful reads required to write (W)

In our default case we have taken 4 Nodes, and N=3, W=2, R=2.

Design choices -

The condition R + W > N ensures that there is an overlap between the nodes that participate in write operations and the nodes that participate in read operations. This overlap guarantees that after a value is written to the database, any subsequent read operation will return the latest written value (or be aware of the write), thereby ensuring strong consistency.

Here's why R + W > N is significant:

1. Avoiding Stale Reads: If R + W > N, any read operation (involving R nodes) is guaranteed to include at least one node that participated in the latest write (involving W nodes), because the sum of participants in read and write operations exceeds the total number of nodes (N).

2. Ensuring Overlap: The overlap means at least one replica node that accepts a write will be read from, ensuring that the most recent write will be seen by a read operation.

Considering your example where N = 3, R = 2, and W = 2:

R + W = 2 + 2 = 4

Since N = 3, we have:

R + W = 4 > 3 = N

This configuration satisfies the condition R + W > N and therefore ensures that for any write operation, a subsequent read operation will intersect with at least one node that contains the newest written data. It is a valid configuration for a quorum-based system, and can help in ensuring strong consistency in your DynamoDB setup.