

School of Computer Science
UNIVERSITY OF PETROLEUM AND ENERGY STUDIES
DEHRADUN, UTTARAKHAND



Container Orchestration and Infrastructure Management
ContactHub: Contacts Management System
Project Report
(2023-2024)
for
5th Semester

SUBMITTED

TO:

Ms.Avita Katal
Assistant Professor(SG)
School of Computer Science

SUBMITTED

BY:

Sanaya Bhardwaj
SAP ID: 500091835
Roll no.: R2142210690

Part 1: Design and Implementation

INTRODUCTION:

In an increasingly interconnected world, managing personal and professional contacts efficiently is vital. ContactHub-The Contact Manager Microservices project leverages the power of Spring Boot, a robust Java-based framework, to create a scalable and highly maintainable solution for contact management.

MICROSERVICES DESIGN

The purpose and functionality of each microservices:

1. User Service:

Purpose: The User Service is responsible for managing user-related data and authentication. It handles user registration, login, and user profile management.

Functionalities:

The User Service is responsible for managing user-related data and operations, such as user registration, authentication, and profile management. When a client sends a request to the API Gateway that requires user-related information or actions, the API Gateway forwards the request to the User Service. The User Service processes the request and responds to the API Gateway.

2. Contact Service:

Purpose: The Contact Service is focused on managing contact-related data. It handles the creation, retrieval, modification, and deletion of contact records.

Functionalities:

The Contact Service manages contact-related data and operations, including creating, retrieving, updating, and deleting contact records. When a client needs to perform operations on contacts, the API Gateway routes the request to the Contact Service. The Contact Service processes the request and returns the relevant data or updates.

3. Eureka Server:

Purpose: The Eureka Server serves as the service registry and discovery server. It allows microservices to register themselves and discover other services.

Functionalities:

Eureka is used for service discovery and registration. Each microservice, including the User Service and Contact Service, registers itself with the Eureka server upon startup. Eureka maintains a registry of all available microservices and their network locations (IP addresses and ports).

4. API Gateway Service:

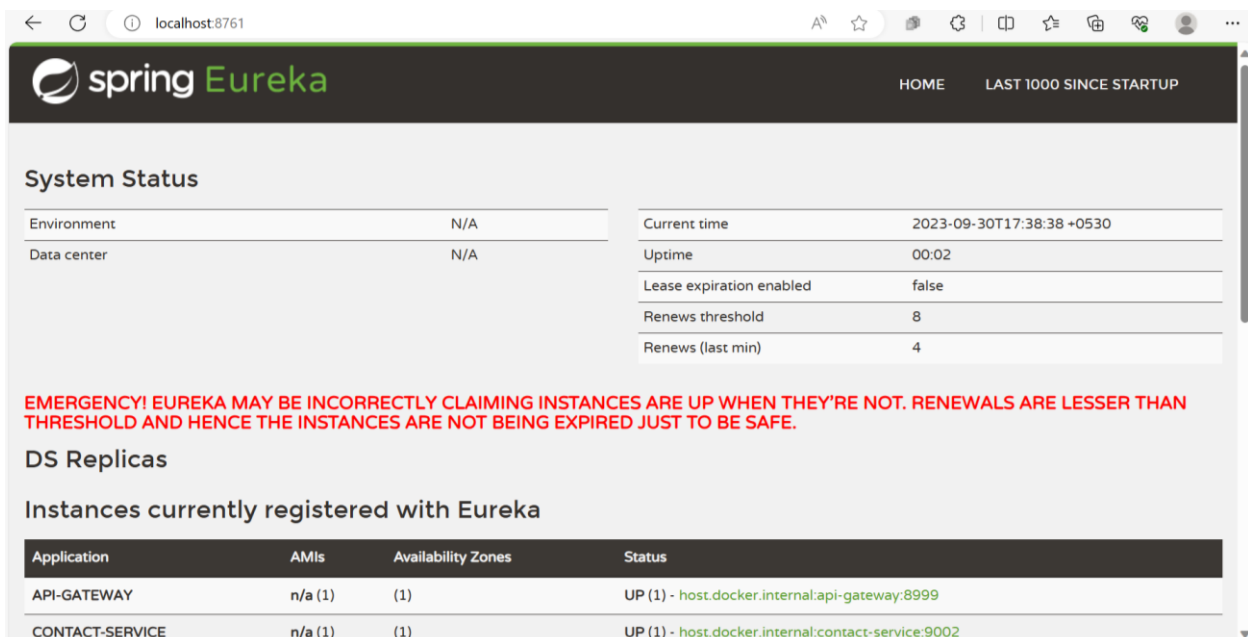
Purpose: The API Gateway Service acts as the entry point for client applications and centralizes requests to various microservices. It can handle authentication, load balancing, and routing.

Functionalities:

The API Gateway serves as the entry point for external clients (e.g., web or mobile applications) to interact with the microservices. It receives incoming requests from clients and routes them to the appropriate microservice based on the request path or URL. The API Gateway may also handle cross-cutting concerns such as authentication, logging, and rate limiting.

These microservices work together to create a modular and scalable architecture. Each microservice has its own specific area of responsibility, making it easier to develop, test, and maintain. The API Gateway Service provides a single-entry point for clients, offering a unified API while handling cross-cutting concerns such as authentication and load balancing.

Additionally, Eureka Server ensures that microservices can dynamically discover and communicate with each other, allowing for flexibility in scaling and maintaining the system. This microservices architecture promotes agility, scalability, and the ability to deploy and update individual components independently, ultimately leading to a more resilient and maintainable system.



The screenshot displays the Spring Eureka web interface in a browser window at localhost:8761. The interface has a dark header with the 'spring Eureka' logo and navigation links for 'HOME' and 'LAST 1000 SINCE STARTUP'. Below the header, the 'System Status' section contains two tables. The left table shows 'Environment' and 'Data center' as 'N/A'. The right table shows 'Current time' as '2023-09-30T17:38:38 +0530', 'Uptime' as '00:02', 'Lease expiration enabled' as 'false', 'Renews threshold' as '8', and 'Renews (last min)' as '4'. A red warning message states: 'EMERGENCY! EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT. RENEWALS ARE LESSER THAN THRESHOLD AND HENCE THE INSTANCES ARE NOT BEING EXPIRED JUST TO BE SAFE.' Below this, the 'DS Replicas' section is followed by 'Instances currently registered with Eureka', which contains a table with two rows of registered instances.

Application	AMIs	Availability Zones	Status
API-GATEWAY	n/a (1)	(1)	UP (1) - host.docker.internal:api-gateway:8999
CONTACT-SERVICE	n/a (1)	(1)	UP (1) - host.docker.internal:contact-service:9002

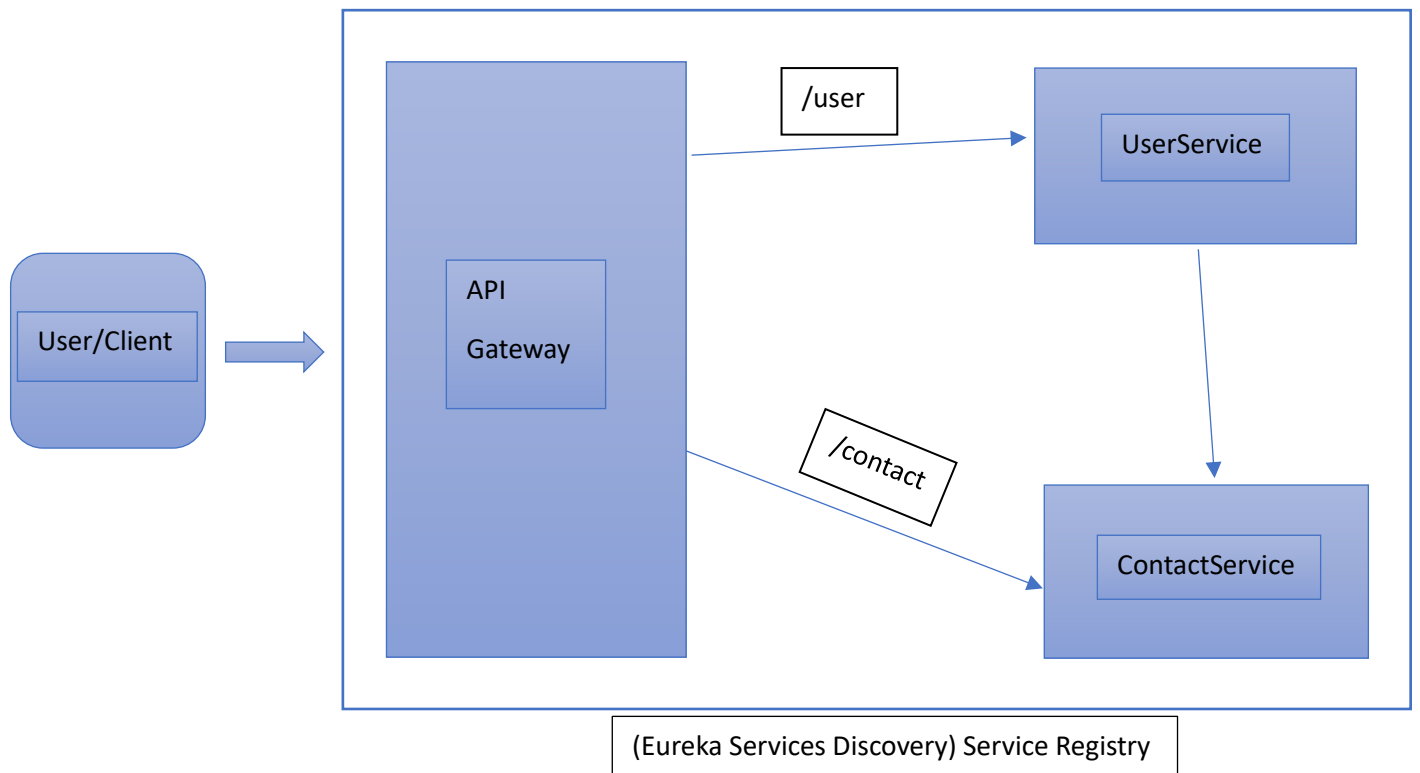
Spring Eureka shows a real-time catalog of available microservices and their network locations for service discovery in a microservices architecture.

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
API-GATEWAY	n/a (1)	(1)	UP (1) - host.docker.internal:api-gateway:8999
CONTACT-SERVICE	n/a (1)	(1)	UP (1) - host.docker.internal:contact-service:9002
USER-SERVICE	n/a (1)	(1)	UP (1) - host.docker.internal:user-service:9010

General Info	
Name	Value
total-avail-memory	88mb
num-of-cpus	8
current-memory-usage	63mb (71%)
server-uptime	00:02
registered-replicas	
unavailable-replicas	
available-replicas	

Instance Info	
Name	Value
ipAddr	192.168.0.103
status	UP

How the microservices will communicate with each other?



- API Gateway is the entry point for client requests.
- Eureka (Service Registry) registers and tracks microservices.
- User Service manages user-related data.
- Contact Service handles contact-related data.
- API Gateway routes client requests to the appropriate service.
- Eureka enables services to discover each other.
- User Service handles user data and operations.
- Contact Service manages contact data.
- Services communicate via HTTP-based RESTful APIs.

Why breaking down the application into specific microservices?

Breaking down the Contact Manager application into specific microservices is justified for the following reasons:

- 1. Scalability:** Microservices allow individual components to scale independently, optimizing resource utilization.
- 2. Modularity:** Microservices promote modular development, making it easier to maintain and update specific functionalities.
- 3. Flexibility:** Each microservice can use the most suitable technology stack for its specific task.
- 4. Resilience:** Isolating services enhances fault tolerance, as a failure in one service does not affect others.
- 5. Specialization:** Microservices are specialized in handling specific tasks, leading to better performance and efficiency.
- 6. Team Autonomy:** Development teams can work independently on microservices, boosting productivity.
- 7. Improved Deployment:** Smaller, focused services are easier to deploy, manage, and scale.
- 8. Easier Testing:** Smaller services are easier to test and debug.

Containerization

Suitable technology for containerization

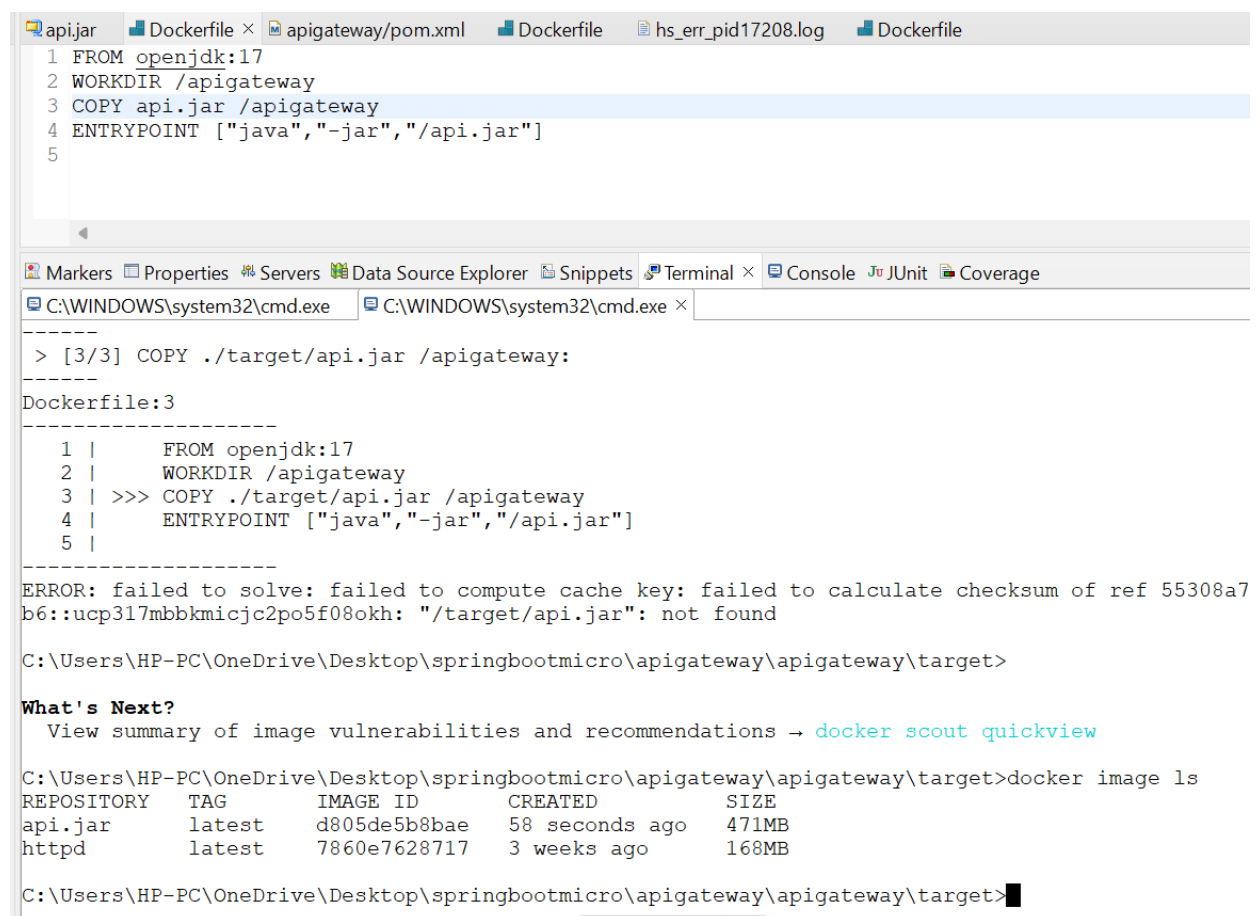
I have chosen Docker for Containerization as it provides a lightweight and consistent way to package, distribute, and run applications and their dependencies, ensuring portability, scalability, and efficient resource utilization.

Its advantages in the context of this project:

- **Isolation:** Docker containers isolate services, preventing conflicts and ensuring stability.
- **Portability:** Containers are portable across environments, simplifying deployment.
- **Efficiency:** Docker optimizes resource utilization and minimizes overhead.
- **Scalability:** Containers can be easily scaled up or down to accommodate varying loads.
- **Consistency:** Docker ensures consistent environments for development, testing, and production.
- **Dependency Management:** Simplifies managing and versioning application dependencies.
- **Security:** Provides a secure runtime environment, reducing security risks.
- **Fast Deployment:** Rapid provisioning and deployment of microservices.

Provide Dockerfiles for each microservice that include necessary dependencies and configurations.

Dockerfile with the Docker image build for API gateway Service:



The screenshot shows an IDE with a Dockerfile editor and a terminal window. The Dockerfile contains the following content:

```
1 FROM openjdk:17
2 WORKDIR /apigateway
3 COPY api.jar /apigateway
4 ENTRYPOINT ["java", "-jar", "/api.jar"]
5
```

The terminal window shows the command `> [3/3] COPY ./target/api.jar /apigateway:` being executed. Below the command, the Dockerfile content is displayed again. An error message is shown: `ERROR: failed to solve: failed to compute cache key: failed to calculate checksum of ref 55308a7b6:ucp317mbbkmicjc2po5f08okh: "/target/api.jar": not found`. The terminal also shows the current directory: `C:\Users\HP-PC\OneDrive\Desktop\springbootmicro\apigateway\apigateway\target>`. At the bottom, there is a section titled "What's Next?" with a link to `docker scout quickview` and a table showing the results of `docker image ls`.

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
api.jar	latest	d805de5b8bae	58 seconds ago	471MB
httpd	latest	7860e7628717	3 weeks ago	168MB

Dockerfile with Docker image build for Contact Service:

The screenshot shows an IDE with a Dockerfile for 'contact_service' and its build output in the console. The Dockerfile contains the following instructions:

```
1 FROM openjdk:17
2 WORKDIR /contact_service
3 COPY contact-service.jar /contact_service
4 ENTRYPOINT ["java", "-jar", "/contact-service.jar"]
5
```

The console output shows the build process, including the transfer of the context, the build of the image, and the naming of the image to 'docker.io/library/contact-service.jar'. The build is successful.

What's Next?
View summary of image vulnerabilities and recommendations → [docker scout quickview](#)

C:\Users\HP-PC\OneDrive\Desktop\springbootmicro\contact_service\contact_service\target>docker images ls

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
contact-service.jar	latest	895a2b9357de	45 seconds ago	471MB
api.jar	latest	d805de5b8bae	5 minutes ago	471MB
httpd	latest	7860e7628717	3 weeks ago	168MB

C:\Users\HP-PC\OneDrive\Desktop\springbootmicro\contact_service\contact_service\target>

Dockerfile with Docker image build for User Service:

The screenshot shows an IDE with a Dockerfile for 'user_service' and its build output in the console. The Dockerfile contains the following instructions:

```
1 FROM openjdk:17
2 WORKDIR /user_service
3 COPY user-service.jar /user_service
4 ENTRYPOINT ["java", "-jar", "/user-service.jar"]
```

The console output shows the build process, including the transfer of the context, the build of the image, and the naming of the image to 'docker.io/library/user-service.jar'. The build is successful.

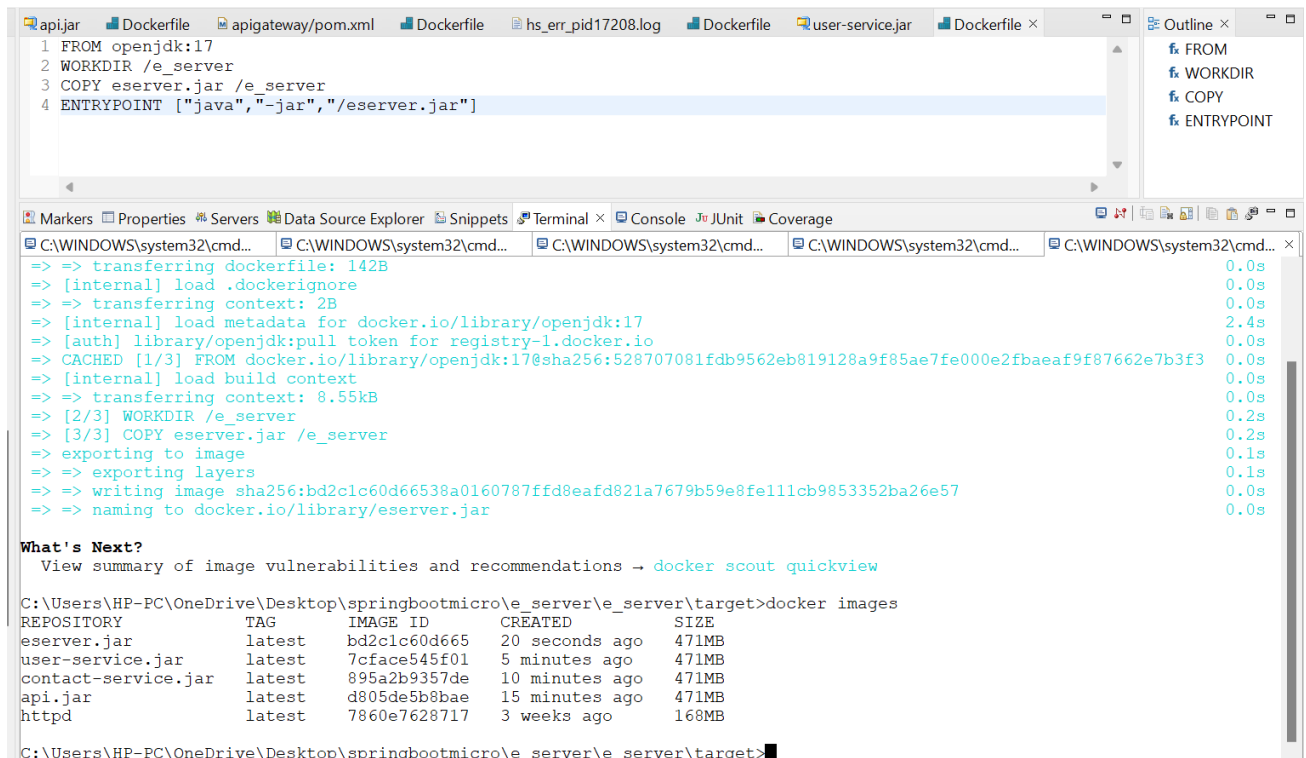
What's Next?
View summary of image vulnerabilities and recommendations → [docker scout quickview](#)

C:\Users\HP-PC\OneDrive\Desktop\springbootmicro\user_service\user_service\target>docker images

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
user-service.jar	latest	7cfac545f01	22 seconds ago	471MB
contact-service.jar	latest	895a2b9357de	5 minutes ago	471MB
api.jar	latest	d805de5b8bae	10 minutes ago	471MB
httpd	latest	7860e7628717	3 weeks ago	168MB

C:\Users\HP-PC\OneDrive\Desktop\springbootmicro\user_service\user_service\target>

Dockerfile with Docker image build for Eureka Server:



The screenshot shows an IDE with a Dockerfile editor and a console window. The Dockerfile contains the following instructions:

```
1 FROM openjdk:17
2 WORKDIR /e_server
3 COPY eserver.jar /e_server
4 ENTRYPOINT ["java", "-jar", "/eserver.jar"]
```

The console output shows the build process:

```
=> => transferring dockerfile: 142B 0.0s
=> [internal] load .dockerignore 0.0s
=> => transferring context: 2B 0.0s
=> [internal] load metadata for docker.io/library/openjdk:17 2.4s
=> [auth] library/openjdk:pull token for registry-1.docker.io 0.0s
=> CACHED [1/3] FROM docker.io/library/openjdk:17@sha256:528707081fdb9562eb819128a9f85ae7fe000e2fbaeaf9f87662e7b3f3 0.0s
=> [internal] load build context 0.0s
=> => transferring context: 8.55kB 0.0s
=> [2/3] WORKDIR /e_server 0.2s
=> [3/3] COPY eserver.jar /e_server 0.1s
=> exporting to image 0.1s
=> => exporting layers 0.0s
=> => writing image sha256:bd2c1c60d66538a0160787ffd8eafd821a7679b59e8fe11cb9853352ba26e57 0.0s
=> => naming to docker.io/library/eserver.jar 0.0s
```

Below the console output, there is a section titled "What's Next?" with a link to [docker scout quickview](#). At the bottom, the command `docker images` is executed, showing a table of images:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
eserver.jar	latest	bd2c1c60d665	20 seconds ago	471MB
user-service.jar	latest	7cface545f01	5 minutes ago	471MB
contact-service.jar	latest	895a2b9357de	10 minutes ago	471MB
api.jar	latest	d805de5b8bae	15 minutes ago	471MB
httpd	latest	7860e7628717	3 weeks ago	168MB

Explain how containers facilitate scalability and isolation of microservices.

1. Scalability: Containers allow individual microservices to be scaled independently. Each microservice runs in its own container, making it straightforward to replicate and deploy more instances of a particular service when needed. This ensures that resources are allocated efficiently based on the specific demands of each microservice, enhancing the project's overall scalability.

2. Isolation: Containers provide a high degree of isolation between microservices. Each microservice runs in its own container with its own runtime environment, including its dependencies and libraries. This isolation prevents conflicts and interference between microservices, ensuring that failures or issues in one container do not impact others. It also simplifies updates and maintenance since changes to one microservice's container won't affect the others.

Overall, containers streamline the management of microservices in the Contact Manager project by offering both scalability and isolation, allowing for efficient resource utilization and reliable service operation.

Integration and Interactions

Explain how microservices will interact with each other through APIs or other means.

In the Contact Manager project with microservices (User, Contact, Eureka, API Gateway):

API Gateway: Serves as the entry point for client requests. It routes requests to the appropriate microservices based on the request path or URL.

Eureka (Service Registry): Microservices register themselves with Eureka upon startup. It maintains a registry of all available services and their network locations.

User Service: Manages user-related data. The API Gateway forwards user-related requests to this service.

Contact Service: Handles contact-related data. The API Gateway routes contact-related requests to this service.

Interactions:

- Microservices communicate via HTTP-based RESTful APIs.
- API Gateway routes client requests to the correct microservice.
- Eureka helps services discover each other dynamically.
- User and Contact Services may communicate directly if required (e.g., fetching user details for contact operations).

This architecture ensures efficient, organized, and modular communication between microservices in the Contact Manager project.

GITHUB LINK: https://github.com/sanaya-bhardwaj/ContactHub-Contact_management_system.git