

Jaypee Institute of Information Technology, Noida

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING AND
INFORMATION TECHNOLOGY



Project Title: ChatSphere

Group - 59

Enrollment No.	Name of Student
9921103002	Rishabh Ralli
9921103003	Astha Mishra
9921103043	Divya Darshana

Course Name: Minor Project-1

Course Code: 15B19CI591

Program: B. Tech. CS&E

3rd Year 5th Sem

2023- 2024

ACKNOWLEDGEMENT

I would like to place on record my deep sense of gratitude to Ms. Akanksha Mehndiratta, Assistant Professor, Jaypee Institute of Information Technology, India for her generous guidance, help and useful suggestions.

I also wish to extend my thanks to friends and other classmates for their insightful comments and constructive suggestions to improve the quality of this project work.

Digital Signature(s) of Students

Rishabh Ralli(9921103002)

Astha Mishra(9921103003)

Divya Darshana(9921103043)

DECLARATION

We hereby declare that this submission is our own work and that, to the best of our knowledge and beliefs, it contains no material previously published or written by another person nor material which has been accepted for the award of any other degree or diploma from a university or other institute of higher learning, except where due acknowledgment has been made in the text.

Place: Noida, Uttar Pradesh, India -
201304

Date: December 2, 2023

Name: Rishabh Ralli

Enrolment No.: 9921103002

Name: Astha Mishra

Enrolment No.: 9921103003

Name: Divya Darshana

Enrolment No.: 9921103043

CERTIFICATE

This is to certify that the work titled “ChatSphere” submitted by Rishabh Ralli, Astha Mishra and Divya Darshana of B. Tech of Jaypee Institute of Information Technology, Noida has been carried out under my supervision. This work has not been submitted partially or wholly to any other University or Institute for the award of any other degree or diploma.

Signature of Supervisor

Ms. Akanksha Mehndiratta

Assistant Professor

December 2,2023

ABSTRACT

This project delves into the intricacies of microservices architecture, utilizing ChatSphere—a social chatting application—as a practical showcase. In the ever-evolving digital landscape, the shift from traditional monolithic architectures to microservices becomes increasingly crucial. Microservices offer a dynamic framework, enhancing scalability and adaptability to the demands of modern applications. By dissecting ChatSphere, this project aims to spotlight the key principles and advantages intrinsic to microservices, showcasing its potential as a contemporary architectural paradigm.

In the traditional realm of monolithic architecture, applications function as tightly-integrated units, presenting challenges in scalability and agility. Microservices, on the other hand, decompose applications into independently deployable services, fostering flexibility and resilience. This report contrasts these paradigms, emphasizing the benefits of microservices, such as scalability, fault isolation, technology diversity, continuous deployment, and enhanced developer productivity. Through this analysis, we underscore the compelling nature of microservices in addressing the evolving needs of software development.

List of Figures

Figure	Title	Page
5.1	Folder Structure for Microservice Backend Architecture.....	17
5.2	Testing Api using Postman.....	17
5.3	Client Architecture.....	18
5.4	Backend Architecture.....	18
5.5	Docker File.....	19
5.6	Images built and stored in google cloud container registry.....	19
5.7	Yaml file for setting up each deployment and services.....	20
5.8	Stateful set for mongodb deployment.....	20
5.9	Load Balancer (Ingress Nginx)	21
5.10.1	Deployments and Services in Google cloud Console.....	21
5.10.2	Deployments and Services in Google cloud Console.....	22
6.1	Kubernetes Cluster.....	23
6.2	Deployments and load balancer.....	23
6.3	Faul Tolerance.....	23

Table of Contents

	Pg. No.
<i>Acknowledgement</i>	2
<i>Declaration</i>	3
<i>Abstract</i>	5
Chapter 1: Introduction	
1.1 General Introduction	8
1.2 Problem Statement	9
Chapter 2: Background Study	10-11
Chapter 3: Requirement Analysis	
3.1 Technologies Required	12
3.2 Other Requirements	12
3.3 Hardware Requirements	12
Chapter 4: Detailed Design	13-16
Chapter 5: Implementation	17-23
Chapter 6: Results	24
Chapter 7: Conclusion and Future Scope	25
References	26

Chapter 1: Introduction

1.1 General Introduction

Monolithic architecture, also known as monolithic application architecture or single-tiered architecture, is a traditional approach to software development where the entire application is built as a single, self-contained unit. All of the application's code, data, and infrastructure are tightly coupled and managed together. This means that the application is packaged and deployed as a single unit, and any changes to the application require a rebuild and redeployment of the entire application.

Advantages of Monolithic Architecture

1. **Simplicity:** Monolithic architecture is relatively simple to understand and develop, especially for smaller applications. The single codebase and centralized management make it easier to manage the application as a whole.
2. **Ease of Testing:** Monolithic applications are easier to test than microservices-based applications, as all components are tightly coupled and can be tested together.
3. **Reduced Complexity:** Monolithic applications have a simpler architecture compared to microservices-based applications, which can reduce complexity and make it easier for developers to understand and maintain the application.

Microservice architecture, also known as microservices or SOA (Service-Oriented Architecture) is an architectural style for building software applications as a suite of small, independent services. Each service is a self-contained unit that performs a specific business function and can be deployed, scaled, and updated independently. Microservices communicate with each other through well-defined interfaces, typically using HTTP or other lightweight protocols.

Advantages of Microservices Architecture

1. **Scalability:** Microservices enable independent scaling of specific services, allowing for optimal resource allocation and efficient handling of variable workloads.
2. **Fault Isolation:** Microservices enhance fault isolation, containing failures within individual services and preventing widespread disruptions in the overall system.
3. **Continuous Deployment:** The independent deploy ability of microservices facilitates continuous deployment practices, enabling faster time-to-market and streamlined release cycles.

1.2 Problem Statement

In today's world of mass communication, there is an issue of ever-increasing users worldwide. This raises the issue of scaling the application to handle multiple concurrent users and handle their data. Scaling in traditional methods is very costly as one might need to invest in additional hardware and infrastructure, leading to resource inefficiency and increased maintenance costs.

Using traditional methods, when we develop code in a specific computing environment it often results in errors and bugs when you transfer it to a new location. For instance, a number of issues arise when we transfer code from your desktop computer to a VM or from a Windows to Linux operating system.

Monolithic architectures have tightly connected processes, so scaling or adding features can be challenging due to their interdependence. As the code base grows, complexity increases, hindering experimentation and innovation. Moreover, if one process fails, the whole application's availability is at risk due to their tight coupling.

Chapter 2: Background study

The core idea behind choosing this topic is to explain what is monolithic and microservices architecture, delineate the difference between them and to showcase through an application why microservices are a necessity for applications being built in modern times.

[1]Paper name: A Systematic Literature Review on Microservices

Authors: Hulya Vural, Murat Koyuncu, Sinem Guney

Conference: International Conference on Computational Science and Its Applications

Summary of the paper:

This paper conducts a thorough systematic mapping study focusing on the evolving landscape of microservices within the realm of cloud architecture. As the cloud continues to shape different approaches and standards, the study explores current trends, the driving motivations behind microservices research, emerging standards, and potential research gaps. Microservices, acknowledged as a key architectural style for building scalable and rapidly evolving cloud applications, takes center stage in the analysis.

Conclusion:

In conclusion, this study emphasizes the pivotal role of microservices in the dynamic realm of cloud architecture. Explored trends, research motivations, emerging standards, and identified gaps underscore microservices as a crucial architectural style for scalable and agile cloud applications. Microservices' modular and decentralized nature aligns with Service-Oriented Architecture, offering practical solutions for enhanced scalability and rapid application evolution.

[2]Paper name: A Comparative Review of Microservices and Monolithic Architectures

Authors: Omar Al-Debagy, Peter Martinek

Summary of the paper:

The research paper investigates the performance disparities between Microservices Architecture and Monolithic Architecture in contemporary software development. It commences by illustrating the prevalent industry shift, exemplified by companies like Netflix and Amazon, towards cloud migration for scalable computing resources. Microservices Architecture is introduced as a method wherein small, independent services communicate through lightweight mechanisms, while Monolithic Architecture encompasses a single code base with tightly coupled services. The paper underscores the purported advantages of Microservices, such as technological flexibility, resilience to isolated failures, scalable services, independent deployment, organizational alignment, composability, and replaceability optimization.

Conversely, Monolithic Architecture is depicted as posing challenges due to its tightly integrated services, prompting the migration of many companies towards Microservices for improved coordination among development teams.

Conclusion:

The research indicates that under normal loads, microservices and monolithic applications perform similarly, with a slight advantage for monolithic systems in small-scale scenarios. The second test scenario reveals that monolithic applications exhibit higher throughput, making them preferable for developers prioritizing faster request handling. Furthermore, a comparison of microservices applications employing different service discovery technologies demonstrates that Consul outperforms Eureka, showing a 4% improvement in throughput. Thus, microservices applications with Consul as the service discovery technology are recommended for optimal performance in terms of the number of handled requests per second.

Chapter 3: Requirement Analysis

3.1 Technologies Required

- Language – JavaScript
- Framework- React.js
- OS – Linux
- Kubernetes
- Docker
- Google Cloud

3.2 Other Requirements (tools)

- Redux
- Nodemon
- Mongoose
- Postman
- Express.js
- Node.js

3.3 Hardware Requirements

- Processor - Intel i5 (11th generation)
- RAM - 8.00 GB
- Disk Space - 512 GB

Chapter 4: Detailed Design

4.1. This Project's aims to achieve the following three objectives through its architecture and development practice:

1. Containerization of the application, that facilitates making the application accessible across all system requirements.
2. Provide flexibility in scalability of storage as well as requests on any one of the features of the application, while keeping the others intact by implementing microservices backend architecture.
3. Managing Features and communication between pods using container orchestration tool Kubernetes.

4.2. To demonstrate the following we have built a containerized Social Media App - ChatSphere upon microservices backend architecture. The features of ChatSphere are:

- User-auth microservice - User Authentication (Login/Register).
- User-auth microservice - Manage and Update personal information. Define your personality and connect with other people that resonate with you.
- Chats microservice - Follow/Unfollow other users to add them as a friend and start conversation.
- Posts microservice - "Spill the tea!" Feed provides a fun place for the user to share their thoughts and events with the people they follow.
- Trend-tags microservice - Our Application shows the trending personality traits that the users of our application have. Users can search on the basis of traits and connect with them.
- Client Microservice - Client Side is deployed as a separate microservice that using load balancer forwards its request to the desired Api.
- Socket Microservices - Realtime Chat Updates and updating current users.
- Databases are deployed separately that communicate with these services. Other services can still communicate with the database, even if one service has an error or goes under maintenance.

4.3. The Concept of Containerization:

Containers are lightweight, portable, and consistent environments that package an application and its dependencies, ensuring that it runs reliably across different computing environments. Containers are a form of operating system virtualization. A single container might be used to run anything from a small microservice or software process to a larger application. Inside a container are all the necessary executables, binary code, libraries, and configuration files.

Advantages of Containers:

- Less overhead
- Increased portability
- More consistent operation
- Greater efficiency
- Better application development

Use cases

- “Lift and shift” existing applications into modern cloud architectures
- Refactor existing applications for containers Develop new container-native applications
- Provide better support for microservices architectures
- Provide DevOps support for continuous integration and deployment (CI/CD)
- Provide easier deployment of repetitive jobs and tasks.

Container vs Virtual Machine

Virtual machines run in a hypervisor environment where each virtual machine must include its own guest operating system inside it, along with its related binaries, libraries, and application files. This consumes a large amount of system resources and overhead, especially when multiple VMs are running on the same physical server, each with its own guest OS.

In contrast, each container shares the same host OS or system kernel and is much lighter in size, often only megabytes. This often means a container might take just seconds to start (versus the gigabytes and minutes required for a typical VM).

Containers Provide better support for microservices architectures. Distributed applications and microservices can be more easily isolated, deployed, and scaled using individual container building blocks.

4.4. Kubernetes - A Container Orchestration Tool

Kubernetes is an open-source platform designed to automate the deployment, scaling, and management of containerized applications. Containers are lightweight, portable, and consistent environments that package an application and its dependencies, ensuring that it runs reliably across different computing environments.

Kubernetes provides a robust framework for automating the deployment and management of containerized applications at scale. Some key features of Kubernetes include:

1. Container Orchestration: Kubernetes automates the deployment, scaling, and operation of application containers. It can manage the deployment of containers across a cluster of machines, ensuring that the desired state of the application is maintained.

2. Service Discovery and Load Balancing: Kubernetes provides built-in mechanisms for service discovery and load balancing. It enables containers to communicate with each other and ensures that traffic is distributed efficiently across multiple instances of a service.

3. Scaling and Self-healing: Kubernetes allows you to scale your application up or down based on demand. It can automatically scale the number of running containers to handle increased load. Additionally, it supports self-healing by automatically replacing failed containers or instances.

4. Declarative Configuration: Kubernetes uses declarative configuration files (YAML or JSON) to define the desired state of the application. Users specify how the application should run, and Kubernetes works to make the current state match the desired state.

5. Resource Management: Kubernetes manages the allocation of resources (CPU, memory, storage) to containers, ensuring that each container gets its fair share and that the overall system operates efficiently.

6. Rolling Updates and Rollbacks: Kubernetes supports rolling updates, allowing you to update your application without downtime. If something goes wrong, you can easily roll back to a previous version.

7. Secrets and Configuration Management: Kubernetes provides a secure way to manage sensitive information such as API keys and passwords. It allows you to store and manage secrets and configuration information separately from application code.

Overall, Kubernetes has become a widely adopted tool in the world of containerized applications, providing a standardized and scalable platform for deploying and managing container workloads. It abstracts away the underlying infrastructure, making it easier for developers and operators to focus on building and running applications.

4.5. Microservices & Kubernetes

Microservices require a number of non-functional requirements.

- Service Discovery
- Auto Scaling
- Load Balancing
- Centralized Configuration
- Lot of Flexibility with our release management.

This is what is actually provided by container orchestration solution Kubernetes. Microservices and Kubernetes go hand in hand.

4.6. Continuous Development using Skaffold

Skaffold is a tool designed to streamline the development workflow for Kubernetes applications. It facilitates the continuous development and deployment of applications by automating the build, push, and deploy process. Skaffold uses a configuration file called ``skaffold.yaml`` to define the settings and instructions for the development workflow.

A typical Skaffold configuration (``skaffold.yaml``) includes information such as:

- 1. Build Configuration:** Defines how the application should be built. This can include specifying the build context, the Dockerfile location, and any custom build scripts or commands.
- 2. Image Configuration:** Specifies the details of the container image, such as the image name, tag, and registry. Skaffold can automatically tag images with unique identifiers during development to avoid conflicts.
- 3. Deployment Configuration:** Describes how the application should be deployed to the Kubernetes cluster. This may include details about the Kubernetes manifests, namespaces, and any other deployment-specific settings.
- 4. Sync Configuration:** Defines how source code changes should trigger updates in the running containers or pods. Skaffold supports various sync methods, such as using file system events or file polling.
- 5. Dev Configuration:** Specifies additional settings for the development environment, like port forwarding or other runtime configurations.

Developers can customize the ``skaffold.yaml`` file based on the requirements of their specific projects and development environments. Skaffold then uses this configuration to automate tasks like building container images, deploying to a Kubernetes cluster, and watching for changes during the development process.

Chapter 5: Implementation

Creating microservice backend/frontend for the social media app

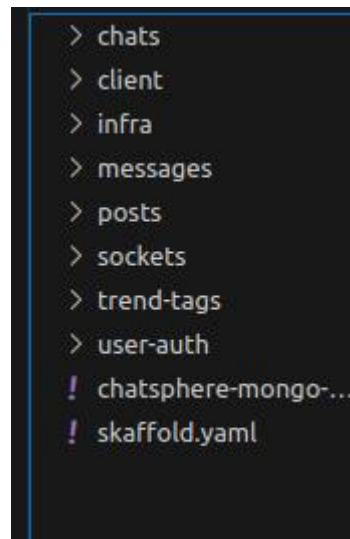


Fig 5.1: Folder Structure for Microservice Backend Architecture

This is the file structure for the front-end and back-end components of our chatting application ChatSphere.

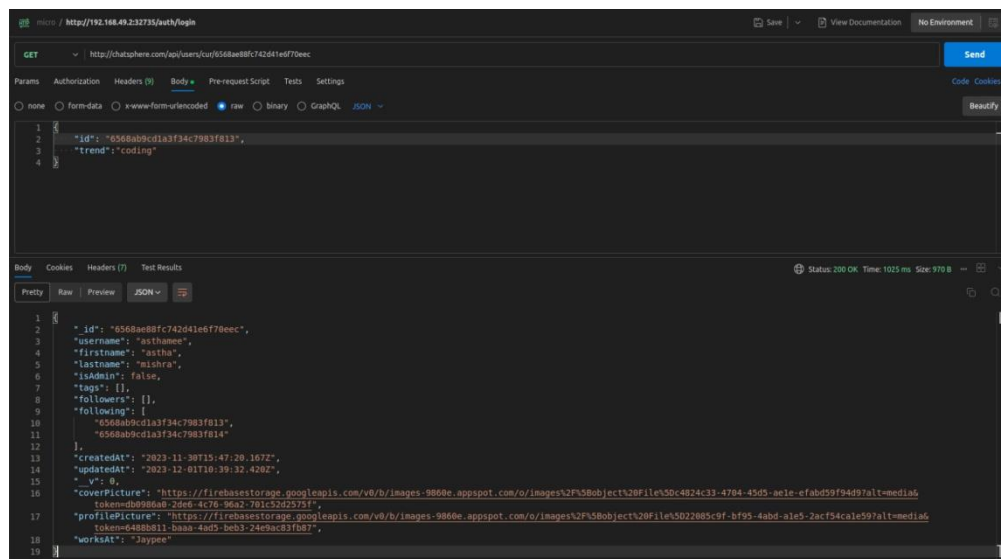


Fig 5.2: Testing Api using Postman

Here we have tested the API response using Postman

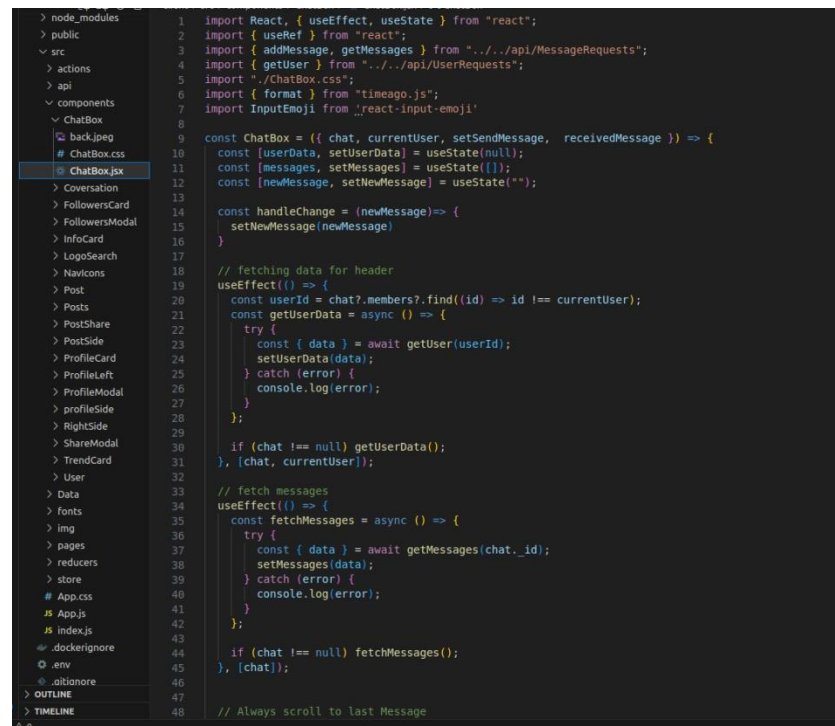


Fig 5.3: Client Architecture
This is the implementation of the chatting interface

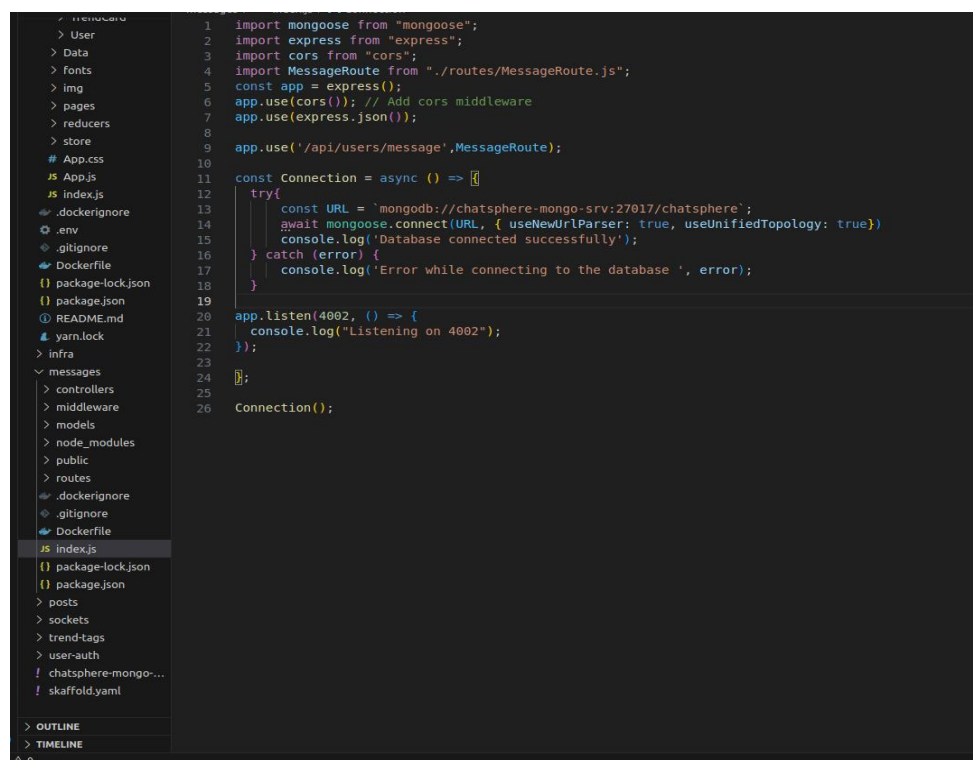
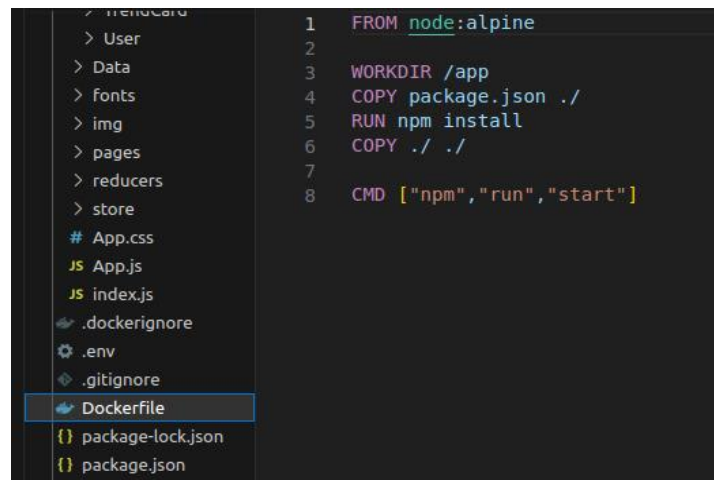


Fig 5.4: Backend Architecture
Connection to database is established here

Creating containers using docker images fetched from google cloud



```
1 FROM node:alpine
2
3 WORKDIR /app
4 COPY package.json ./
5 RUN npm install
6 COPY ./ ./
7
8 CMD ["npm", "run", "start"]
```

Fig 5.5: Docker File

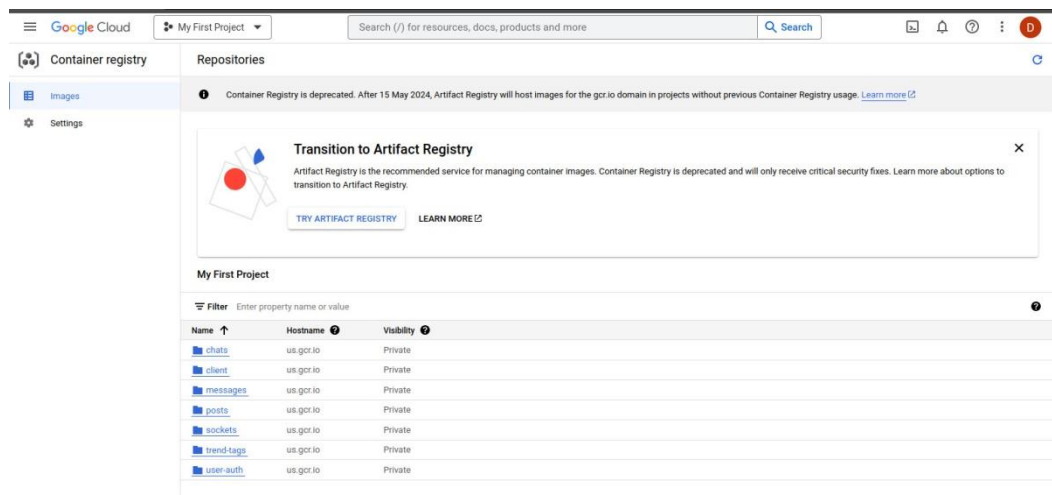


Fig 5.6: Images built and stored in google cloud container registry

We create Deployments for the different microservices. These deployments can have varied number of pods in them. We set up their corresponding services as well. These pods then communicate outside the cluster or with each other through these services.

Below is the yaml configuration file for deployment and service of each api and the client side. We make a statefulset for persistent storage in MongoDB.

```

infra > k8s > ! client-depl.yaml
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: client-depl
5  spec:
6    replicas: 1
7    selector:
8      matchLabels:
9        app: client
10   template:
11     metadata:
12       labels:
13         app: client
14   spec:
15     containers:
16     - name: client
17       image: us.gcr.io/airy-discovery-349015/client
18   ---
19   apiVersion: v1
20   kind: Service
21   metadata:
22     name: client-srv
23   spec:
24     selector:
25       app: client
26     ports:
27     - name: client
28       protocol: TCP
29       port: 3000
30       targetPort: 3000
31

```

Fig 5.7: yaml file for setting up each deployment and services

```

! chatsphere-mongo-depl.yaml
1  apiVersion: apps/v1
2  kind: StatefulSet
3  metadata:
4    name: chatsphere-mongo-depl
5  spec:
6    serviceName: chatsphere-mongo-srv
7    replicas: 1
8    selector:
9      matchLabels:
10       app: chatsphere-mongo
11   template:
12     metadata:
13       labels:
14         app: chatsphere-mongo
15   spec:
16     containers:
17     - name: chatsphere-mongo
18       image: mongo
19       ports:
20       - containerPort: 27017
21       volumeMounts:
22       - name: pvc
23         mountPath: /data/db
24   volumeClaimTemplates:
25   - metadata:
26       name: pvc
27     spec:
28       accessModes:
29       - ReadWriteOnce
30       resources:
31         requests:
32           storage: 1Gi
33   ---
34   apiVersion: v1
35   kind: Service
36   metadata:
37     name: chatsphere-mongo-srv
38   spec:
39     selector:
40       app: chatsphere-mongo
41     ports:
42     - name: db
43       protocol: TCP
44       port: 27017
45       targetPort: 27017
46

```

Fig 5.8: stateful set for mongodb deployment

Creating load balancer using ingress nginx to communicate between the pods Here we specify the ingress load balancer routes to redirect the request from the client deployment to the different services built inside the cluster. It exposes the pods inside the cluster to the outside world.

```

1  apiVersion: networking.k8s.io/v1
2  kind: Ingress
3  metadata:
4    name: ingress-srv
5    annotations:
6      kubernetes.io/ingress.class: nginx
7      nginx.ingress.kubernetes.io/use-regex: 'true'
8  spec:
9    rules:
10     - host: chatsphere.com
11       http:
12         paths:
13           - path: /api/users/(.*)
14             pathType: Prefix
15             backend:
16               service:
17                 name: user-auth-clusterip-srv
18                 port:
19                   number: 4000
20           - path: /api/users/chats/(.*)
21             pathType: Prefix
22             backend:
23               service:
24                 name: chats-srv
25                 port:
26                   number: 4001
27           - path: /api/users/message/(.*)
28             pathType: Prefix
29             backend:
30               service:
31                 name: messages-srv
32                 port:
33                   number: 4002
34           - path: /api/users/posts/(.*)
35             pathType: Prefix
36             backend:
37               service:
38                 name: posts-srv
39                 port:
40                   number: 4003
41           - path: /api/users/tags/(.*)
42             pathType: Prefix
43             backend:
44               service:
45                 name: trend-tags-srv
46                 port:
47                   number: 4004

```

Fig 5.9: Load Balancer (Ingress Nginx)

Deployments corresponding to each api are built that have stateless property. However, MongoDB being a database pod requires persistent storage. Hence, we built a stateful state for it with persistent volume claim. Google Cloud console can be used to monitor them.

Filter	Name	Status	Type	Pods	Namespace	Cluster
Is system object: False	chats-depl	OK	Deployment	1/1	default	chatsphere-cluster
	chatsphere-mongo-depl	OK	Deployment	1/1	default	chatsphere-cluster
	client-depl	OK	Deployment	1/1	default	chatsphere-cluster
	ingress-nginx-admission-create	OK	Job	0/1	ingress-nginx	chatsphere-cluster
	ingress-nginx-admission-patch	OK	Job	0/1	ingress-nginx	chatsphere-cluster
	ingress-nginx-controller	OK	Deployment	1/1	ingress-nginx	chatsphere-cluster
	messages-depl	OK	Deployment	1/1	default	chatsphere-cluster
	posts-depl	OK	Deployment	1/1	default	chatsphere-cluster
	user-auth-depl	OK	Deployment	1/1	default	chatsphere-cluster

Fig 5.10.1: Deployments and Services in Google cloud Console

Filter	Status	Name	Location	Number of nodes	Total vCPUs	Total memory	Notifications	Labels
Enter property name or value	OK	chatsphere-cluster	us-central1-c	3	3	5.1 GB		

Fig 5.10.1: Deployments and Services in Google cloud Console

Persistent volume claim for the MongoDB pod Stateful State

Implementing Continuous Integration using Skaffold:

Yaml File listing files and directories that Skaffold has to monitor to notice changes. The build process is launched for that pod, accordingly.

```
! skaffold.yaml x | trend-tags-deploy.yaml | JS userModel.js posts/... | Dockerfile posts | JS PostsRequests.js | JS UploadRequest.js | .env | () package.json | JS firebase.js | Profile

1 apiVersion: skaffold/v2alpha3
2 kind: Config
3 deploy:
4   kubectl:
5     manifests:
6       - ./infra/k8s/*
7 build:
8   #local:
9   # push: false
10  googleCloudBuild:
11    projectId: airy-discovery-349015
12  artifacts:
13    - image: us.gcr.io/airy-discovery-349015/user-auth
14      context: user-auth
15      docker:
16        dockerfile: Dockerfile
17      sync:
18        manual:
19          - src: 'src/**/*.js'
20            dest: .
21    - image: us.gcr.io/airy-discovery-349015/posts
22      context: posts
23      docker:
24        dockerfile: Dockerfile
25      sync:
26        manual:
27          - src: '**/*.js'
28            dest: .
29    - image: us.gcr.io/airy-discovery-349015/client
30      context: client
31      docker:
32        dockerfile: Dockerfile
33      sync:
34        manual:
35          - src: '**/*.js'
36            dest: .
37    - image: us.gcr.io/airy-discovery-349015/chats
38      context: chats
39      docker:
40        dockerfile: Dockerfile
41      sync:
42        manual:
43          - src: '**/*.js'
44            dest: .
45    - image: us.gcr.io/airy-discovery-349015/trend-tags
46      context: trend-tags
47      docker:
48        dockerfile: Dockerfile
```

Fig 5.10.2: Deployments and Services in Google Cloud Console

Chapter 6: Results

Kubernetes Cluster setup using Google Cloud:

```
● dora-the-explorer@NotAlone:~/Desktop/microservices demonstration/blog$ kubectl config current-context gke_aary-discovery-349015_us-central1-c_chatsphere-cluster
○ dora-the-explorer@NotAlone:~/Desktop/microservices demonstration/blog$
```

Fig 6.1 Kubernetes Cluster

Deployments and Services and Ingress Load Balancer corresponding to different microservices:

```
● dora-the-explorer@NotAlone:~/Desktop/microservices demonstration/blog$ kubectl get deployments
NAME          READY   UP-TO-DATE   AVAILABLE   AGE
chats-depl    1/1     1             1           7m22s
client-depl   1/1     1             1           7m20s
messages-depl 1/1     1             1           7m15s
posts-depl    1/1     1             1           7m14s
trend-tags-depl 1/1     1             1           7m12s
user-auth-depl 1/1     1             1           7m10s
● dora-the-explorer@NotAlone:~/Desktop/microservices demonstration/blog$ kubectl get services
NAME          TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
chat-srv      ClusterIP   10.56.14.89   <none>         4001/TCP   2d19h
chataphere-mongo-srv ClusterIP   10.56.15.136 <none>         4001/TCP   2d19h
chats-srv     ClusterIP   10.56.13.15   <none>         4001/TCP   7m29s
chatsphere-mongo-srv ClusterIP   10.56.3.168   <none>         27017/TCP  2d18h
client-srv    ClusterIP   10.56.11.86   <none>         3000/TCP   7m27s
kubernetes    ClusterIP   10.56.0.1     <none>         443/TCP    6d16h
messages-srv  ClusterIP   10.56.3.147   <none>         4002/TCP   7m22s
mongo         ClusterIP   None          <none>         27017/TCP  2d21h
posts-srv     ClusterIP   10.56.11.228   <none>         4003/TCP   7m21s
trend-tags-srv ClusterIP   10.56.12.116   <none>         4004/TCP   7m19s
user-auth-clusterip-srv ClusterIP   10.56.1.194   <none>         4000/TCP   7m18s
○ dora-the-explorer@NotAlone:~/Desktop/microservices demonstration/blog$
```

Fig 6.2 Deployments and load balancer

Microservice Automatic Redeployment (FaultTolerance):

```
● dora-the-explorer@NotAlone:~/Desktop/microservices demonstration/blog$ kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
chats-depl-7cb4d9d578-fmc2m 1/1     Running   0           3m43s
chatsphere-mongo-depl-0      1/1     Running   0           21h
client-depl-797495d45b-2mqc2 1/1     Running   0           3m41s
messages-depl-58b9f47b97-chqv2 1/1     Running   0           3m36s
posts-depl-679ccf9bb5-rgdmg 1/1     Running   0           3m35s
trend-tags-depl-7474b64d65-2ts7j 1/1     Running   0           3m33s
user-auth-depl-855bfc7588-52cbl 1/1     Running   0           3m31s
● dora-the-explorer@NotAlone:~/Desktop/microservices demonstration/blog$ kubectl delete pod chats-depl-7cb4d9d578-fmc2m
pod "chats-depl-7cb4d9d578-fmc2m" deleted
● dora-the-explorer@NotAlone:~/Desktop/microservices demonstration/blog$ kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
chats-depl-7cb4d9d578-f8pwv 1/1     Running   0           32s
chats-depl-7cb4d9d578-fmc2m 1/1     Terminating 0           5m1s
chatsphere-mongo-depl-0      1/1     Running   0           21h
client-depl-797495d45b-2mqc2 1/1     Running   0           4m59s
messages-depl-58b9f47b97-chqv2 1/1     Running   0           4m54s
posts-depl-679ccf9bb5-rgdmg 1/1     Running   0           4m53s
trend-tags-depl-7474b64d65-2ts7j 1/1     Running   0           4m51s
user-auth-depl-855bfc7588-52cbl 1/1     Running   0           4m49s
● dora-the-explorer@NotAlone:~/Desktop/microservices demonstration/blog$ kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
chats-depl-7cb4d9d578-f8pwv 1/1     Running   0           57s
chatsphere-mongo-depl-0      1/1     Running   0           21h
client-depl-797495d45b-2mqc2 1/1     Running   0           5m24s
messages-depl-58b9f47b97-chqv2 1/1     Running   0           5m19s
posts-depl-679ccf9bb5-rgdmg 1/1     Running   0           5m18s
trend-tags-depl-7474b64d65-2ts7j 1/1     Running   0           5m16s
user-auth-depl-855bfc7588-52cbl 1/1     Running   0           5m14s
○ dora-the-explorer@NotAlone:~/Desktop/microservices demonstration/blog$
```

Fig 6.3: Fault Tolerance

Here even when a service fails, it is automatically redeploying.

Chapter 7: Conclusion and Future Scope Conclusion:

Conclusion:

We have successfully demonstrated and implemented the microservices architecture through our ChatSphere application. The application has many features such as user registration, authentication, daily trends, posts, followers and real time chat. All of these features have been implemented using microservices which makes this application robust and improves the availability. Since the application uses a microservice architecture, even in the case where one of the services fails, then, the entire application will not fail since each service is isolated. ChatSphere is a perfect example that showcases that as we progress into the future, it is a must for applications to follow the microservice architecture.

Future Scope:

- We can further deploy the clusters to the cloud environment. This will enable us to leverage cloud services to monitor the logs for scalability requests and improve the availability and reliability, simplify application delivery and end-user access and enhance business continuity.
- Improve the overall security features of the application by providing better authentication methods like two step verification etc.
- We can improve the overall user experience by improving on the ux/ui design.

References:

Journal articles

- [1] Vural, Hulya, Murat Koyuncu, and Sinem Guney. "A systematic literature review on microservices." Computational Science and Its Applications–ICCSA 2017: 17th International Conference, Trieste, Italy, July 3-6, 2017, Proceedings, Part VI 17. Springer International Publishing, 2017.
- [2] O. Al-Debagy and P. Martinek, "A Comparative Review of Microservices and Monolithic Architectures," 2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI), Budapest, Hungary, 2018, pp. 000149-000154, doi: 10.1109/CINTI.2018.8928192.

Online

- [3] Ansari, S. (2023, October 9). How to write a YAML configuration file? | A Step-by-Step Guide for Beginners. DEV Community.
- [4] Bennett, A. (2021, December 29). An introduction to microservices - Microservice Geeks - Medium. Medium