Introduction to Observability and Monitoring of Microservices

Inside this section, we are going to talk about the new challenge that we may face while building microservices. Right now, we have discussed and completed total seven challenges.

And inside this section we are going to talk about challenge number eight.

So, you might be very curious, what is this new challenge? What is this challenge about?

So let me reveal the details of this challenge.

So, the challenge that we are going to talk inside this section is, observability and monitoring of our,

microservices.

If you are hearing these observability and monitoring jargons very first time, do not worry, I am going

to spend a good amount of time explaining what is observability and what is monitoring and how we should implement them inside our microservices.

Before we try to understand the definition of observability and monitoring.

First, let me try to put some questions in front of you and post that we can see how these observability and monitoring are going to help us in resolving the problems or challenges that we are going to face.

The very first problem are the question that I have is, **how we are going to debug our microservices**.

If there is an issue identified inside your microservices, how are you going to debug them inside monolithic application you have only single application where you can debug the issue very easily, but inside microservices network, the request may travel across multiple microservices and containers.

So **how are you going to trace the transaction across multiple services and containers and how are you going to find where exactly the problem or the defect is**?

So, this is one of the major problems that we may have.

And apart from tracing the request, **we should also worry about combining all the logs from multiple services into a centralized location where the logs can be indexed, searched, filter and grouped to find the bugs that are contributing to a problem**.

Inside the monolithic application, the maintenance of the logs is going to be super, super easy. Because a single web application is generating the logs.

We can store them inside a folder location and whenever a developer wants to understand the issue,

he can easily download those logs and try to scan them to understand where the issue is.

But with microservices, it is not going to be the same because many containers and many services,

they are going to generate the logs inside their own containers.

And as a developer, we cannot go to all those multiple locations and try to analyse the logs.

Sometimes your request may travel more than 20 microservices to give a response, so in such scenarios it is going to be very tedious job to look for the logs inside the containers individually.

That is why to overcome this challenge we should look for the options on how we can combine all the logs from the multiple services into a centralized location.

I hope you are clear with this problem.

The next problem is **how are we going to monitor performance of our service calls**?

Like we discussed, a single request may travel multiple microservices inside the microservice network.

If there is a performance issue inside your microservice network, how we are going to track the path

of a specific chain service, call through the microservice network and see how long it took to complete at each microservice.

Until unless we know how much time my request is taking in a particular microservice, we cannot really identify which microservice is taking more time.

So, once we identify the microservice, which is taking more time, then only we can try to debug the

issue.

That is why identifying a microservice, which is taking a lot of time compared to other microservice

is very important to monitor and identify the performance issues inside your microservices.

Apart from performance, **we should also worry about how to monitor metrics and health of our microservices**.

We should look for the options using which we can easily monitor the metrics of containers and microservices like CPU usage, JVM metrics.

Since you will be having hundreds of containers and microservices running inside your microservice network, monitoring each of them with the help of actuator is going to be a super complex process.

That is why we should look for the options, which can help us in monitoring the status and health of

our microservice in a single centralized location and create alerts and notifications for any abnormal

behaviour of the services.

**Why do we need alerts and notifications?**

We cannot have our team members continuously looking at the dashboards or microservices to understand their health.

Sometimes it is not going to be a feasible option to monitor all the microservices 24 by seven.

That is why we should automatically trigger some alerts and notifications, if there is some abnormal behaviour inside a particular microservice.

So, these are the problems that we may face whenever we are trying to implement microservices inside any organization.

So how to overcome all these problems with the help of observability and monitoring by using these concepts, we can solve all these challenges very easily.

And that way we are going to avoid the outages that are going to happen inside our microservices.

Observability and monitoring is a very interesting topic.

Observability vs Monitoring

Inside this lecture, let us try to talk in detail about observability and monitoring.

**First, let us try to understand what is observability.**

Observability is the ability to understand the internal state of a system by observing or by understanding its output.

For example, in the context of microservices, observability can be achieved by collecting and analyzing data from a variety of sources such as metrics, logs and traces.

Using all these information, we can try to understand how our microservices that are running internally, how well a particular microservice is processing all the incoming requests or how many errors my microservice is throwing.

So, all such internal information we can try to understand with the concept of observability.

**There are three pillars for observability.**

The very first one is **metrics**.

Metrics are quantitative measurements of the health of a system.

They can be used to track things like CPU usage, memory usage and response times.

So, using these metrics, we can measure our microservices performance or health.

The next pillar that we have is **logs**.

So, what are logs?

Logs are a record of events that occur inside a system.

We can put log statements inside our method calls, so whenever that particular method is being executed, my logs will be generated indicating that a particular record of event is executed successfully.

So, using these logs we can track things like errors, exceptions, and other unexpected events.

We all know how important logs are while debugging an issue without logs, it is impossible to identify an issue inside the production environment.

After logs the last pillar of observability is **traces**.

So, what are traces?

Traces are a record of the path that a request takes through a system inside our microservices network.

They can be hundreds of microservices.

A particular request may travel through the multiple microservices.

So, to understand what is a path that my request travelled inside the microservice network, we can use those traces.

Using these traces information, we can track the performance of a request at each microservice or at each method.

And with that we should be able to easily identify any performance bottlenecks.

By collecting the data from these three pillars, we can analyze the data and gain a good understanding of the internal state of our microservices.

This understanding can help developers to identify and troubleshoot problems, and in sometimes we can also use this information to identify the bottlenecks and improve the performance.

And apart from performance and troubleshooting problems, with the help of this data, we can also

ensure the overall health of the system.

I hope you are clear what is observability.

**Now let us try to understand what is monitoring.**

Monitoring in microservice context involves checking the telemetry data available for the application and defining the alerts for known failure state.

We saw inside the observability we use the data like logs, metrics, and traces for troubleshooting any

defects or performance issues.

But with the help of monitoring, we can build some dashboards and build alerts and notifications based upon metrics, logs, and traces information.

Maybe we can try to build a dashboard which can be used by the operations team to monitor the overall health of our microservices.

If the CPU utilization of a particular container or a microservice crossed more than 80%, then I want

an alert to be triggered.

So, all such scenarios we can achieve with the help of monitoring.

Monitoring is very important inside microservices because it will allow us to identify and troubleshoot problems.

If you have hundreds of microservices running in different containers, in different virtual machines

monitoring all of them, 24 by seven is going to be super, super impossible task.

That is why by collecting and analyzing the data from the individual microservices, we can identify

problems before they cause any outages or other disruptions.

For example, if you take the same CPU usage example if a particular microservice CPU usage is more

than 80%, then I can try to add one more instance of the microservice so that the traffic will be divided between these two instances.

So here, using this monitoring information, I'm trying to avoid the outage by adding more number of

instances.

And apart from identifying and troubleshooting problems using monitoring, we can also track the health of our microservices.

Using dashboards, alerts, and notifications we can easily understand which microservice is underperforming or which microservice is having some network problems or any other kind

of problems that will affect the performance of my microservice.

And at last, using monitoring, we can try to optimize our microservices by onboarding more number

of instances or by killing the microservice, which is having problems.

And in the same place we can try to onboard a new container or a new microservice instance.

With all these approaches, we are going to improve the performance and reliability of the microservices.

**So here you may have a question like both observability and monitoring they are looking very similar and you may also have confusion like what is the difference between them?**

Let me try to answer the same.

Monitoring and observability can be considered as two sides of the same coin.

Both rely on the same types of telemetry data to enable insights about your microservices.

This telemetry data or data types, includes **metrics**, **traces**, and **logs** using these kind of information

only both observability and monitoring are going to work.

Let us try to understand in detail what is the difference between monitoring and observability.

For the same, if you try to take an example of iceberg, whatever you are able to see on the top of the iceberg, we can call that as monitoring because we will be easily able to see how our microservices health, CPU usage, how many threads are being used?

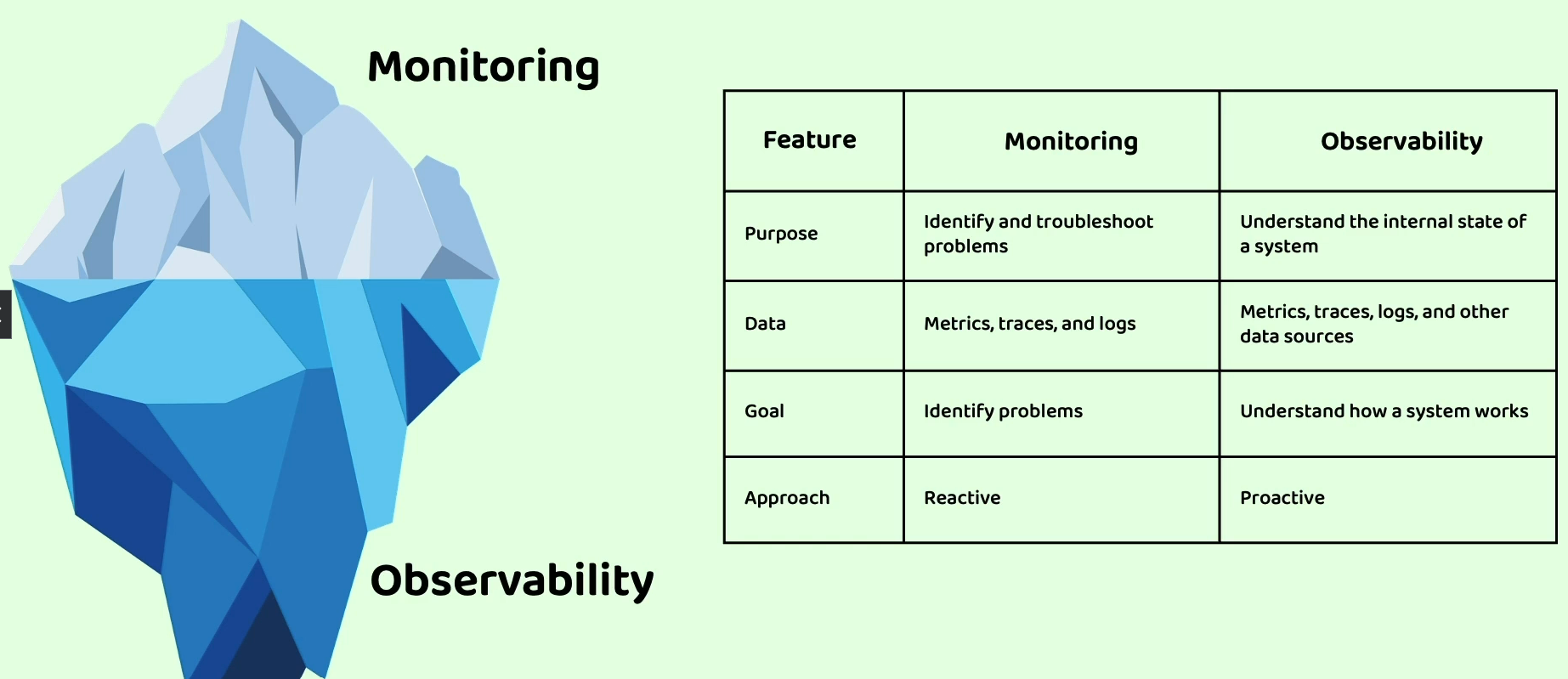
All such information we can easily see with the help of monitoring because inside the monitoring we

are going to build the dashboards, alerts, and notifications.

But whatever information which you cannot easily see, all such information, we call it as observability, there might be some NullPointerException inside my microservice network until unless I go and investigate the logs, I cannot really understand where the issue is.

So, such kind of information comes under the observability.

Let me give you a table which has differences between monitoring and observability.



**Purpose** of monitoring and observability, monitoring is going to help you to identify and troubleshoot problems. Whereas with the help of observability, we can try to understand the internal state of a system.

Coming to that **what kind of data that will be used** inside the monitoring is metrics, traces, and logs inside observability. Apart from these metrics, trace and logs, other kind of information also will be leveraged.

Moving on to the next information, **which is goal**.

The goal of the monitoring is to identify the problems, whereas the observability goal is to understand how a system works.

Now let us talk about the **approach**.   
Inside the monitoring the approach is going to be reactive.

So, what is **reactive approach**?

When I say reactive, we are going to react when a problem occurs.

For example, your operations team can react when a particular microservice is facing some network problems or some performance issues. Without seeing those events your operations team cannot react to them.

That is why inside the monitoring the approach will always be reactive, whereas inside the observability the approach is going to be proactive.

For example, your microservice might be throwing some NullPointerExceptions.

Maybe it is not throwing for each and every request for some random scenario or for some rare scenario.

A particular microservice is throwing a RuntimeException or NullPointerException.

So as a developer you will try to proactively identify the issues so that you can provide a fix in the

future release.

So, this is a **proactive approach**.

If you ask me to explain the difference of monitoring and observability in simple words:

I would say monitoring is about collecting data, and observability is about understanding data.

We can also say monitoring is reacting to the problems while observability is fixing them in real time.

Like any runtime exceptions or performance issues, we have to deep dive into the information to understand the internal state and fix them in real time.

Always, please note that observability will help you to understand the internal state of a system, whereas monitoring will help you to identify and troubleshoot problems with the help of alerts, dashboards, notifications.

Introduction to centralized logging or Log Aggregation in microservices

We discussed that there are three pillars for observability and monitoring, and these three pillars

are **logging,** **metrics,** and **traces**.

So, to implement these concepts inside our microservice network, we need to make sure we are generating all these data so that using them we can try to understand the internal state of our microservices and monitor them.

So first let us try to **implement log aggregation** inside our microservices.

**What are these logs?**

Logs are discrete records of events that happens inside any software application over time.

Usually logs contain a timestamp that indicates when the event happened, as well as the information

about the event and its context.

The information present inside the logs can be used to answer the questions like what happened at a

particular time, which thread was processing the event or which user or tenant was in the context.

We all know that logs are the essential tools for troubleshooting and debugging tasks.

They can be used to reconstruct the scenario.

What happened at a specific point of time in a single application instance.

Logs are typically categorized according to the types of severity such as trace, debug info, warn

and error.

This will allow us to log only the most severe events in the production.

We should not also do the over logging inside our application or microservices because that will have

some performance implications.

We should only log the severe events like there is an exception happened or there is an error happened inside production.

Off course, inside the testing environment and development environment, we can try to log as much as information with the help of severity like debug and info.

If you are not clear about how to implement logging inside any spring boot application with the help

of these severities, please check my course on spring Spring boot.

There is a separate section inside that course where I talked about how to perform logging inside the

spring boot applications, how to activate a specific severity based upon the environment.

So, all such best practices I have discussed inside the spring course.

So basically, what I am trying to say here is, we can also attach severity to a particular log statement the purpose of the severities is this will allow us to log only the most severe events in production and if needed, we can activate or deactivate a particular type based upon the environment where you are trying to run your application.

So, whenever we try to perform logging inside monolithic application, since all the code is in a single

codebase, all the logs that are generated from your code will be present inside a single location or

inside a single server.

This makes very easy for the developers to search the logs and to troubleshoot the problems because

you have all your logs in a single folder location.

Whereas with microservices logging is going to be complex. This is because each service has its own logs.

With that, the developer must check the logs from the multiple containers which are present in multiple locations to understand or to debug an issue.

So, to address this challenge inside Microservice Network, the best practice is to follow the centralized logging.

**So what is centralized logging?**

With the help of centralized logging, we can collect the logs from all the services and store them

in a single location.

This makes developer life easy to find and troubleshoot problems because the developer has to look in a one place only. Without log aggregation the developer will be forced to search the logs information in hundreds of microservices that you have deployed inside your organization.

Do you think it is a feasible option? Off course not.

That is why we need to perform log aggregation.

So here you may have a question like **how to perform this log aggregation**.

Do I need to write some logic inside my microservices as a developer, that is one of the most basic approach.

Like you can write some logic to save or to stream your container logs into a centralized location.

But this approach where the developer is responsible to handle the logic of log aggregation has many

disadvantages because log aggregation is not related to the business logic.

If you force your developers to worry about the log aggregation also, then you are wasting their time

on unnecessary things like log aggregation.

We as a developer or we as intelligent humans, we should always focus on the client problems or the

business logic.

That is why we are going to look for an option which is going to perform the log aggregation without

making any changes inside our microservices.

There are such beautiful products.

Introduction to managing logs with Grafana, Loki and Promtail

Grafana is a company which is building a lot of tools and plugins to implement observability monitoring inside any kind of applications.

It is not only about microservices, even for other type of applications like web applications, IoT

applications for any kind of applications, with the help of the tools provided by the Grafana, we

can implement observable and monitoring.

The reason why I choose the Grafana to implement observability and monitoring is, it provides open-source tools for various scenarios.

For example, if you want to implement log aggregating system, then go for the tool **Grafana Loki**.

Similarly, if you are looking around metrics, then you can integrate **Prometheus with Grafana** and

build some dashboards alerts very easily.

And for tracing information you can leverage **Grafana tempo**.

This way there are many tools available inside the Grafana for various kind of scenarios and it has

very good integration with other standards like **Open telemetry, Prometheus**.

We are going to discuss in detail about these tools.

Grafana itself is a very fascinating subject and many operations are platform engineers, it is a mandatory skill for them to learn everything about Grafana.

But inside this course I am going to teach you some basics and how to use Grafana, how to build observability and monitoring with the help of Grafana tools in the real projects.

It is not the developer responsibility to implement observability and monitoring.

The developer must work with the operations team or platform team and implement the same.

But I am trying to teach about this inside this course so that whenever you have challenges or problems related to observability and monitoring inside your projects and organizations, you should be able to give some directions to your operations team or platform team.

Or you can also build some demo applications with the help of all the concepts that we are going to

discuss.

And with that, you will become a super, super hero or a super developer inside your project.

When you climb yourself as a microservice developer, you need to aware about all these concepts, otherwise it is going to be super, super tough to clear the any interviews around microservices.

Now let us try to understand how to implement log aggregation or centralized logging with the help of Grafana.

When we decide to manage logs with the help of Grafana, we need to use other tools available inside Grafana ecosystem. Over all Grafana is a large ecosystem under which we have many tools based upon our scenario or problem.

We need to choose these tools and integrate with the Grafana.

So here to manage the logs, we need to use Grafana Loki and prompt.

Let's try to understand what are these like we discussed first, **Grafana is a open source analytical**

**and interactive visualization web application**.

This Grafana provides various features like we can build charts, graphs, alerts for our web applications or microservices by connecting with the other supporting tools like Loki and Promptail, we can easily install this Grafana and its tools with the help of Docker or Docker compose or Kubernetes .

Over all Grafana is a popular tool for visualizing metrics, logs and traces from variety

of sources.

It is being used by many organizations of all types right from startup to the enterprise organizations.

Majority of the projects they are using Grafana because it is open source.

So, this is a quick introduction about Grafana.

Now let us talk about Loki and Promptail.

**Grafana Loki is a horizontally scalable, highly available log aggregation system.**

It is designed to store any number of logs from your microservices and applications.

So, Loki is a centralized location where you can store all your microservices logs.

**But with Grafana, Loki alone, we cannot implement the log aggregation.**

**We need to have Promptail**, which is a lightweight log agent that will run inside your same network were your containers are running.

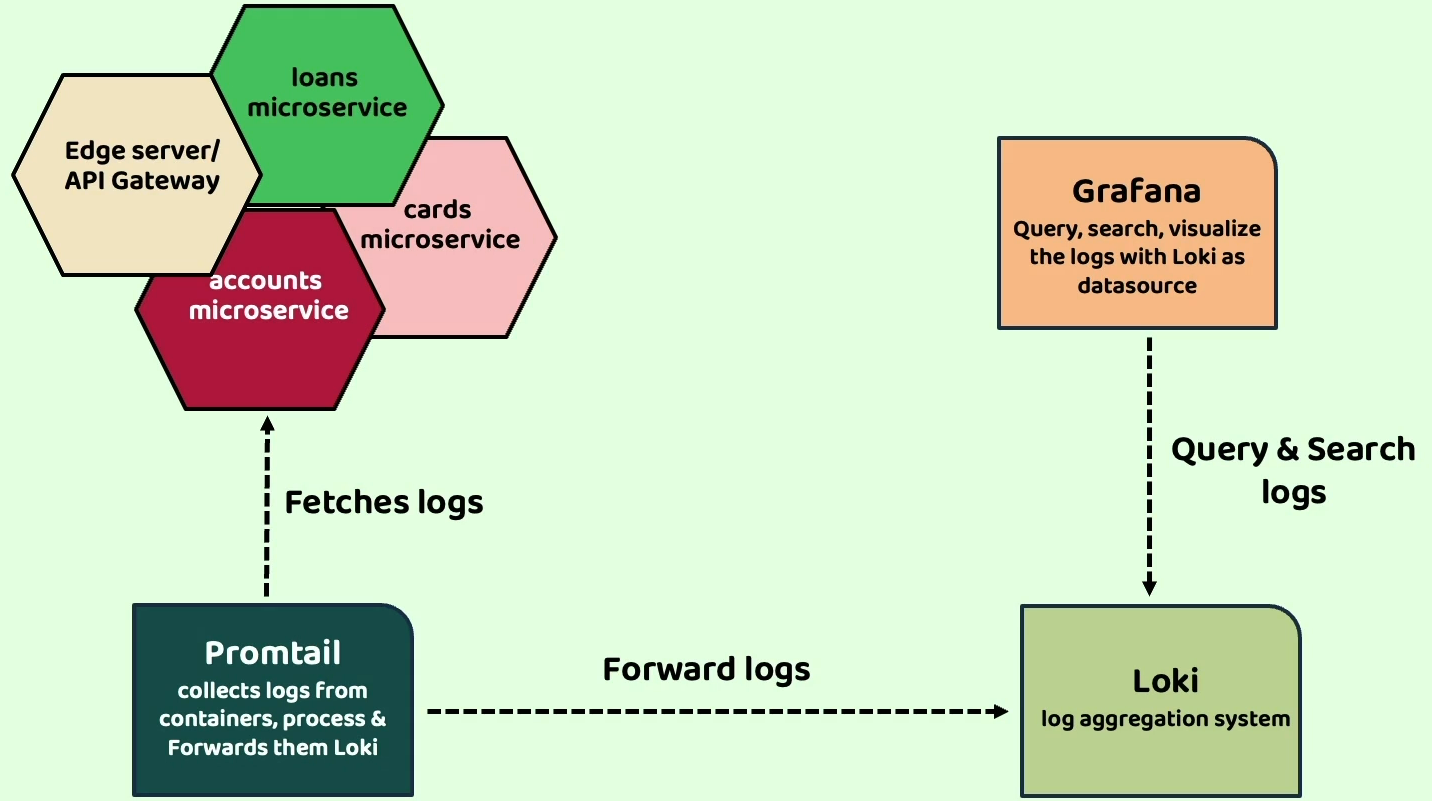
So, this **Promptail** it is going to read all the logs that are getting generated from your containers, the same it will collect and send to the Grafana Loki.

Grafana, Loki will store all these logs with the help of Grafana we can easily see the logs inside an application together Grafana, Loki and promptly provides a powerful logging solution that can help you to understand and troubleshoot your applications.

I know here you may have questions.

Let us try to understand more detail about these Grafana, Loki and Promptail.

If you see here, I have my microservices running inside their containers.



As a developer, I am not doing anything to send my logs to the Promptail or Loki or Grafana.

The Promptail, which is an agent running inside the same network of your microservice containers.

It is going to fetch and collect all the logs from the containers and forward them to the Loki.

Loki is a storage system where it can store all the logs, since it is a centralized location.

In other words, we can also call it as log aggregation system.

So, Loki is like a folder location where we can store all our logs from all the containers.

But do you think it is a good idea for the developers to go into these folder location and search for

the logs manually by opening all the logs?

Off course not.

That is why by integrating this Loki with the Grafana, we can visualize all the logs inside an application.

Apart from visualization, we can also query and search logs based upon our own criteria.

When you see all this in demo, you are going to super, super like the tools of Grafana.

But for now, I am assuming that you are clear about what is the role of Grafana, Loki and prompt how to achieve this.

IMPORTANT NOTE IN PROMTAIL

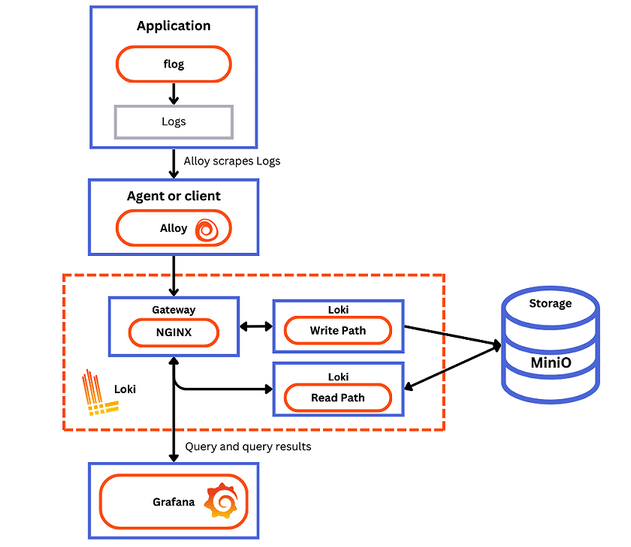
#### PROMTAIL IS REPLACED BY ALLOY

1. From **Grafana Loki version 3.0** onwards, **Promtail**, which is responsible for scraping log lines, has been replaced with a new product called **Alloy**. Even though I will discuss Promtail in the next few lectures, Alloy will function similarly. Since these are internal components of Grafana Loki, this change will not have a significant impact. We just need to use the config files related to Alloy in place of Promtail.
2. All the Docker Compose files have been updated with Alloy-related changes inside the GitHub repo. You can find the documentation for Alloy along with Loki in the following link:

[**https://grafana.com/docs/loki/latest/get-started/quick-start/**](https://grafana.com/docs/loki/latest/get-started/quick-start/)

1. If for any reason your real projects use older versions of Grafana Loki, then you will need to use Promtail. Promtail-related changes are available in the older branches of the course GitHub repo.

Sample demo of Logging using Grafana, Loki & Promtail – Theory



If you see here first, there is an application with the name Flog. This is just a random application provided by the Grafana team. This application is going to generate and emit logs continuously.

That is what you can see here. It is going to emit the logs and this flog is deployed into a container.

Now if you see in the same network where my log is deployed, the Alloy (Promptail) which is a log agent

will be running.

The responsibility of this prompt is, to read the new logs whenever generated by the log app and collect the same logs.

And once the Alloy (Promptail) collected those logs it must send to the Loki, but it will not send directly

to the Loki.

In between there is an edge server like a gateway.

With the help of this Gateway, it is going to send the logs to the Loki.

Inside Loki there are two components like Loki read component and Loki write component.

Why we need so many components.

Because like I said, Loki can store any number of logs. To make it scalable and handle any

amounts of log data.

Grafana team built many separate components, so whenever my prompt is trying to send the logs to the gateway, the gateway will see what is the URL that is invoked by the client. Here the client will

be prompted and in the case where the request is coming from, the Alloy (Promptail), it will redirect the

request to the Loki right component.

Once my Loki component receives the logs, it is going to store the logs in a component with the name Minio.

So, all these are internal components. We do not need to worry much about them behind the scenes.

Grafana is going to take care of all writing, reading with the help of this gateway.

We do not even have to develop all this gateway, Loki Read Component, Loki Write Component.

These are readily available tools which we can deploy inside our microservices.

Now you can see the sequence is the log is generated, the Alloy (Promptail) collected, the logs, send it to

Gateway and the Gateway forward the same to the Loki write component.

From the Loki write component the logs are written into a storage system called Minio.

Now think like the logs are stored inside the centralized location, which is Minio and Loki.

As a next step, one of the developers wants to understand the logs of a particular microservice.

So, what he will do, he will go to the Grafana and he will try to search the logs whenever developer

is trying to search the logs inside Grafana, the request will go to the Gateway.

The Gateway will forward the same request to the Loki read component because the developer only trying to read the logs and this log read component is going to read the logs from the Minio and eventually the same will be sent back to the Grafana where the developer can see the logs in an UI application.

So, this is the overall architecture that is recommended by the Grafana to implement log aggregation.

Please do not worry about that, it is going to be super, super easy because now we know Docker,

Docker compose with the scripts provided by the Grafana team.

We can easily implement this.

As of now, you can see inside this architecture we have a sample application with the name flog.

Instead of this flog, we are going to have all our microservices generating the logs and Alloy (Promptail) is

going to read them and send to the Loki from the Loki we are going to read them and see them inside the Grafana.

I hope this is clear.

If you can scroll down, there are steps that we can follow to implement the same inside any microservices network.

So, there are some Yaml configurations given by the Grafana team.

By following the same instructions, we can implement them inside our microservice as well.

I am going to mention this URL inside the GitHub repo.

Please read all the details inside this page.

Whatever I have explained, the same is present inside this page.

I also mentioned all those details inside this slide.

For your quick reference.

If you try to recall the 15-factor methodology that we discussed previously with this approach, we

Are trying to implement one of the recommendations mentioned inside the 15-factor methodology under the name logs.

So, what does 15 factor methodology is recommending to treat the logs as events stream to the standard output and not concerned with how they are processed or stored.

So, the individual microservices are the developers, they should not worry about streaming or storing the logs.

All the log aggregation should happen from the outside. That is what we are trying to achieve here.

I hope you are clear with this approach from the next lecture,

Implementing Logging using Grafana, Loki & Promtail – Part 1

Make sure Docker is running in your system.  
  
Add jib plugin to all projects below in pom.xml  
**configserver**, **eureka**, **accounts**, **loans**, **cards**

<build>

        <plugins>

….

            <plugin>

                <groupId>com.google.cloud.tools</groupId>

                <artifactId>jib-maven-plugin</artifactId>

                <version>3.4.1</version>

                <configuration>

                  <to>

                    <image>nileshzarkar/${project.artifactId}:s11</image>

                  </to>

                </configuration>

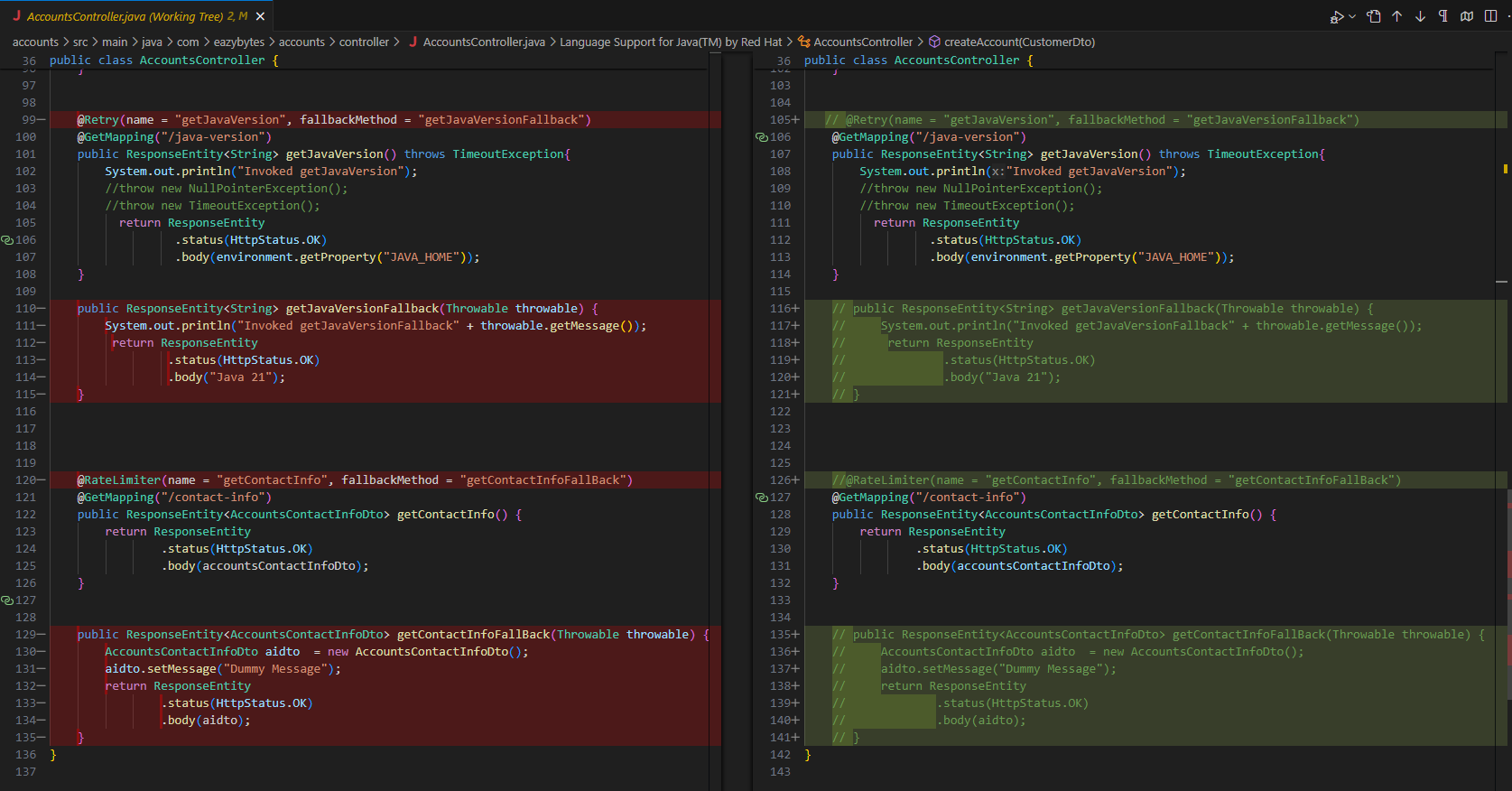
              </plugin>

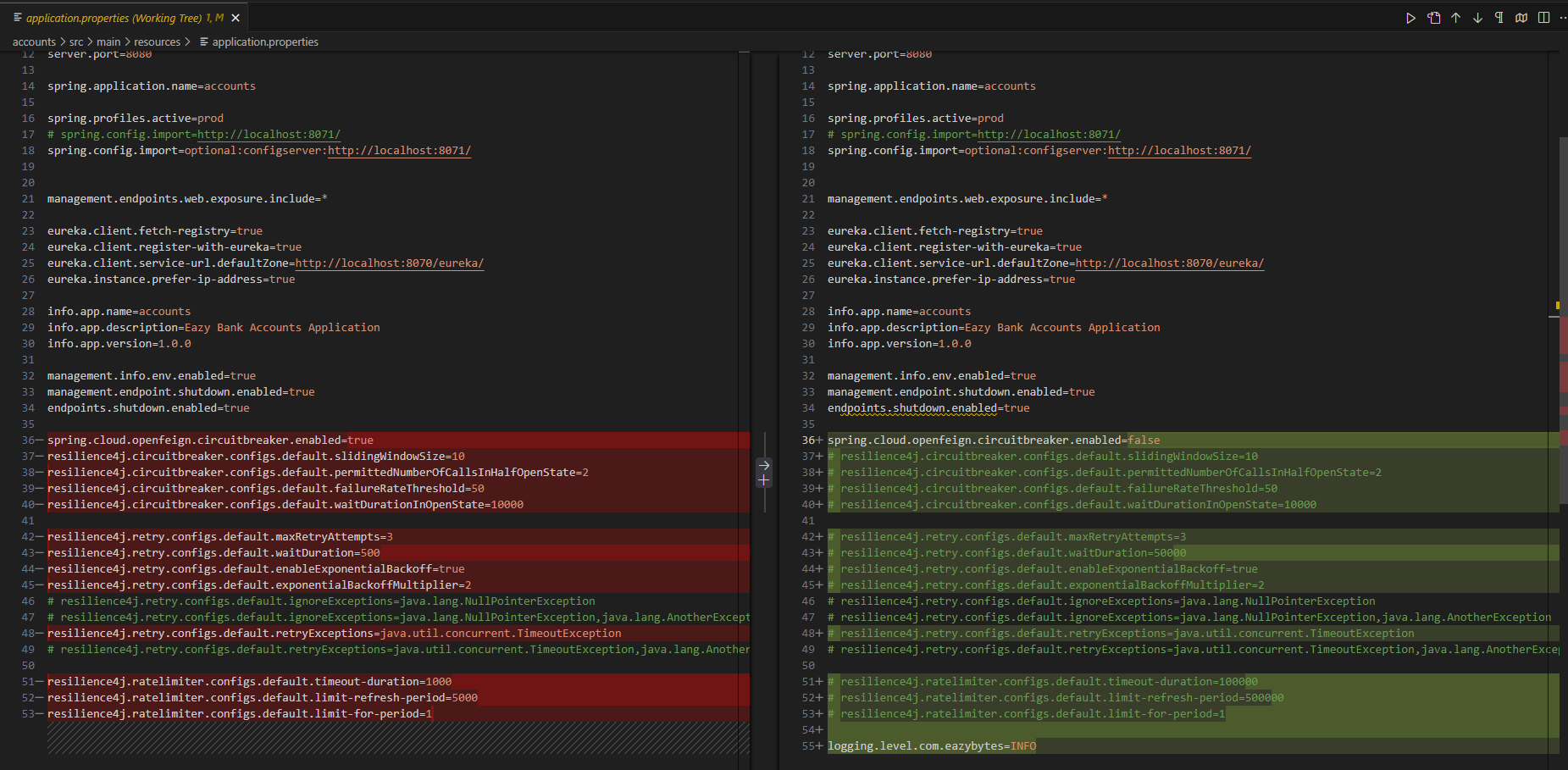
        </plugins>

</build>

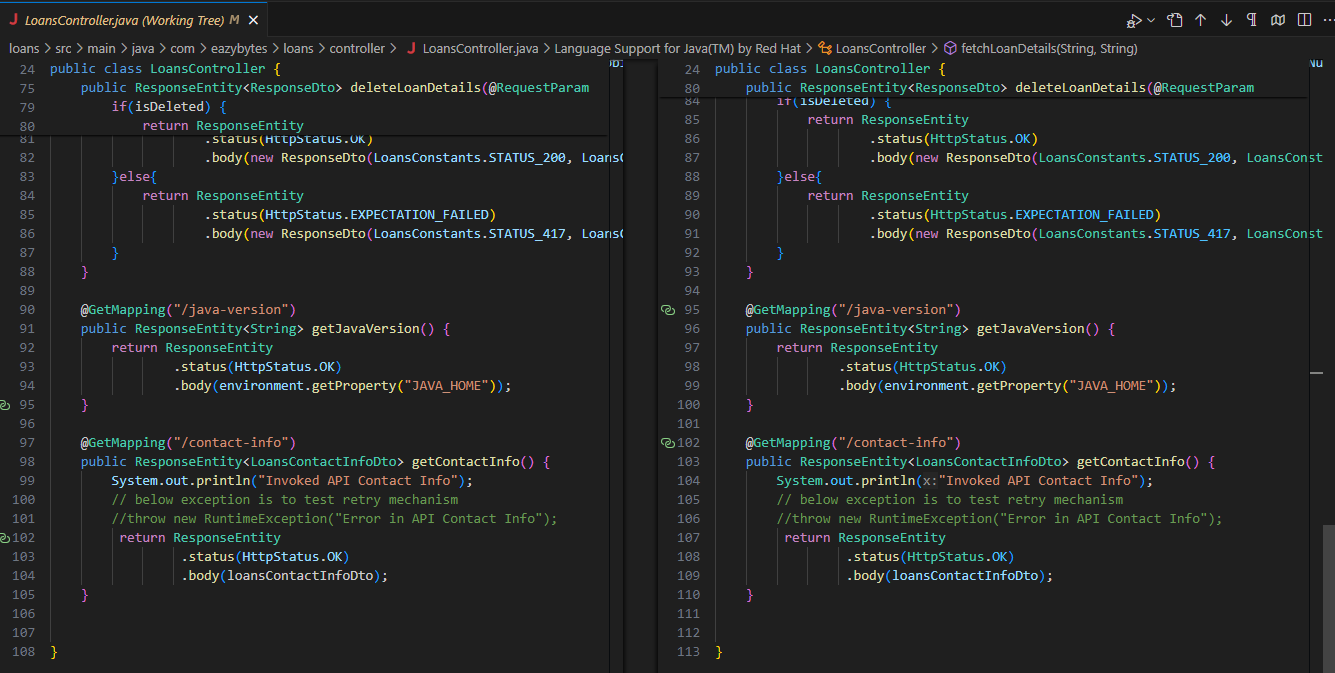
Update all tags to s11

Disable all circuit breaker, resilience, retry, rate limiter in all microservices.

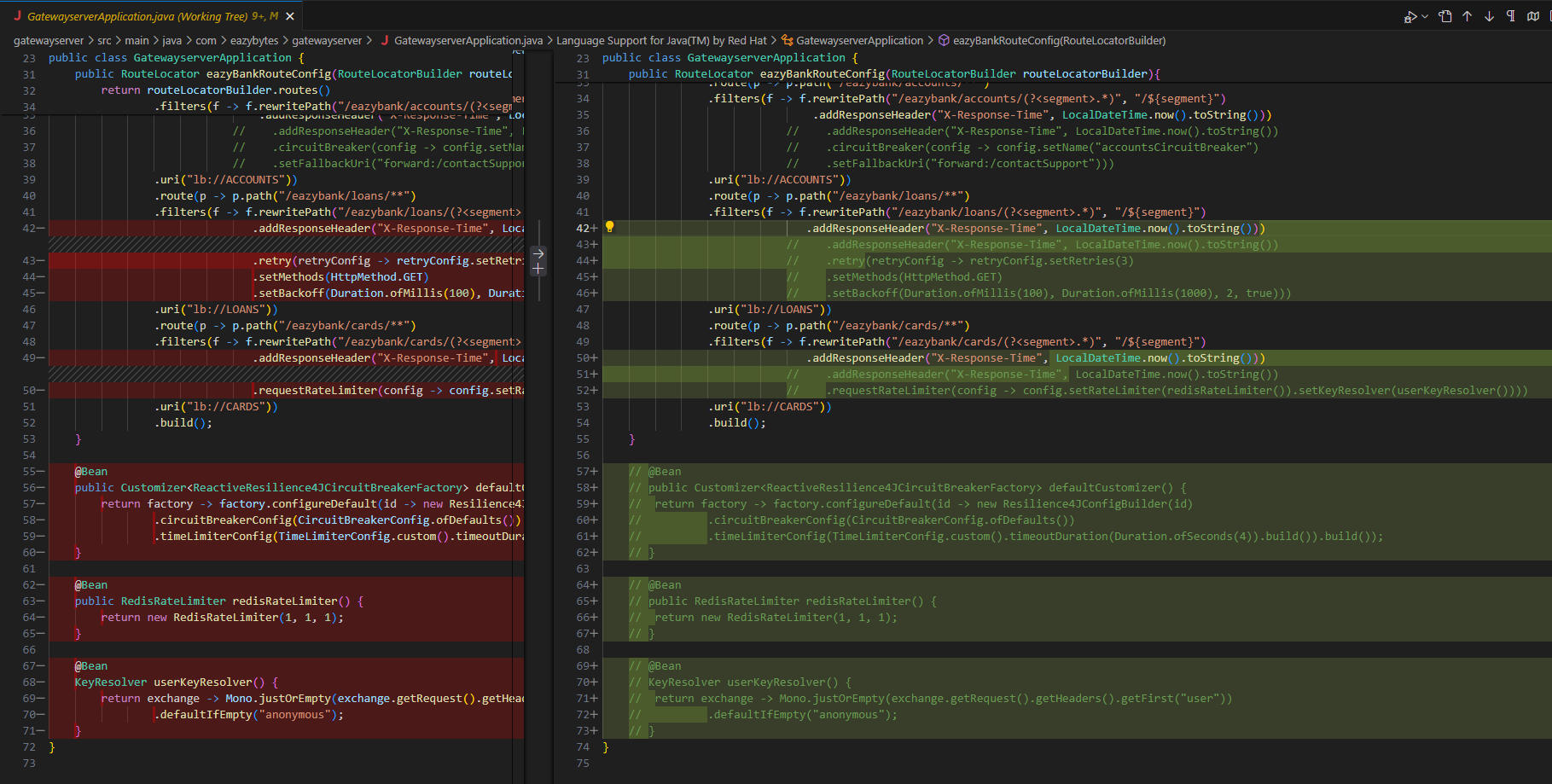
**Accounts microservice**

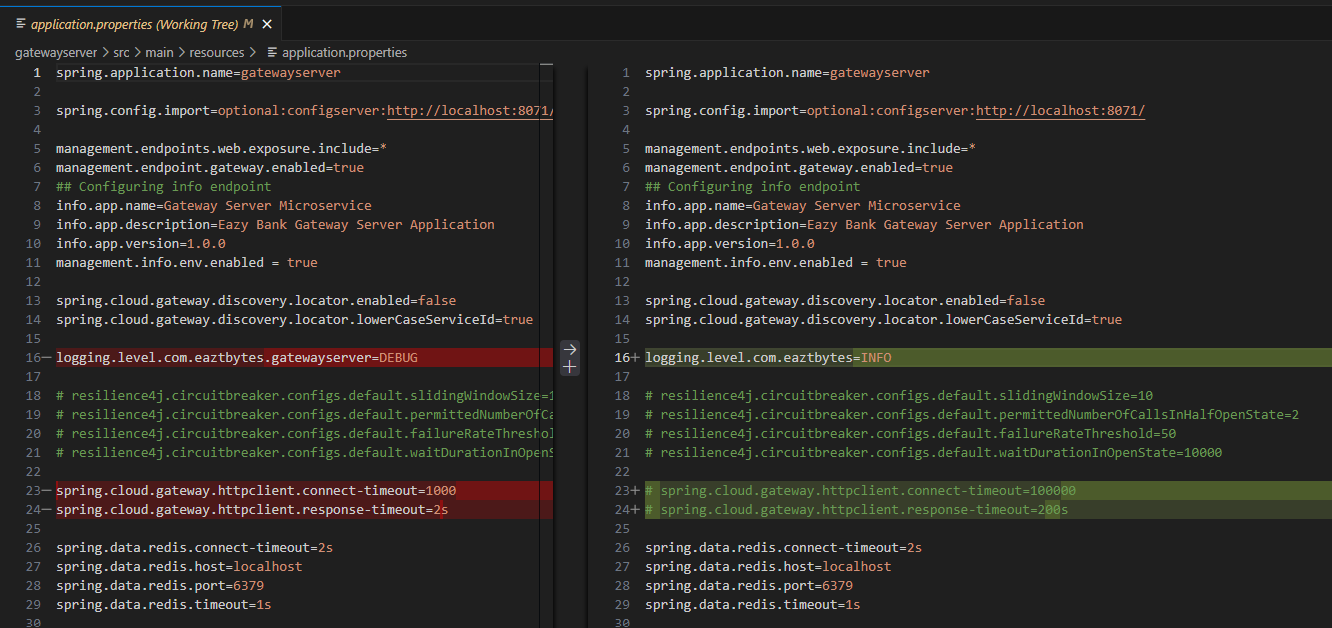


**Loans microservice**



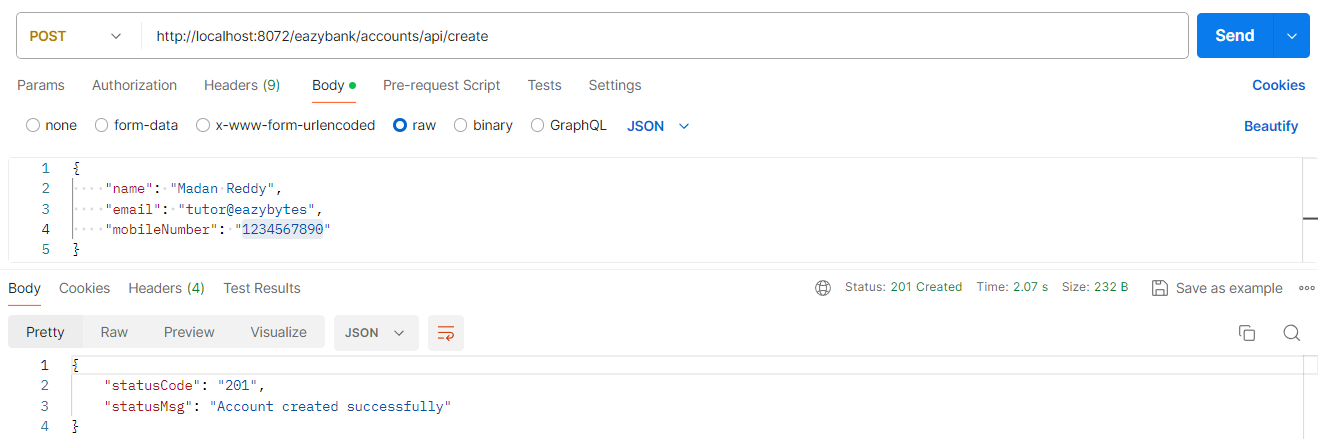
**Gateway microservice**

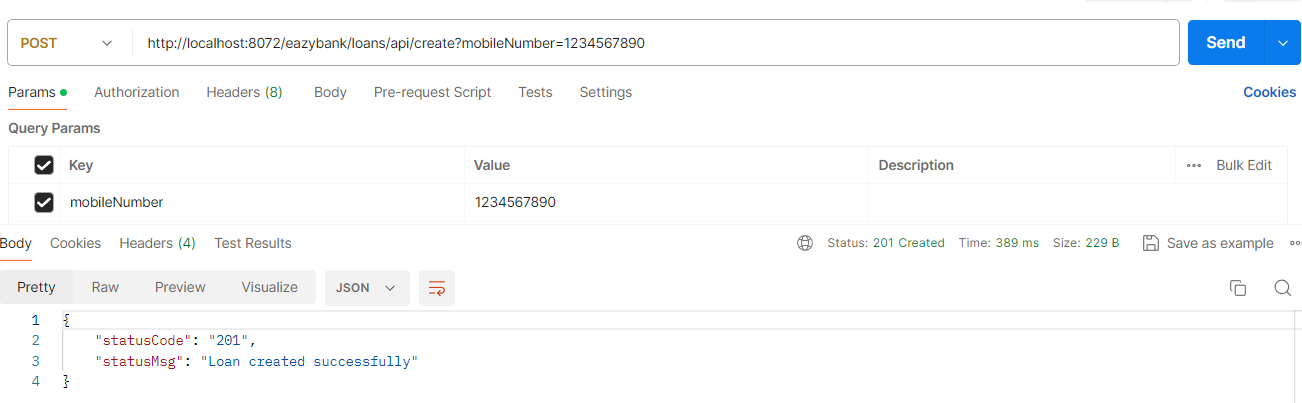


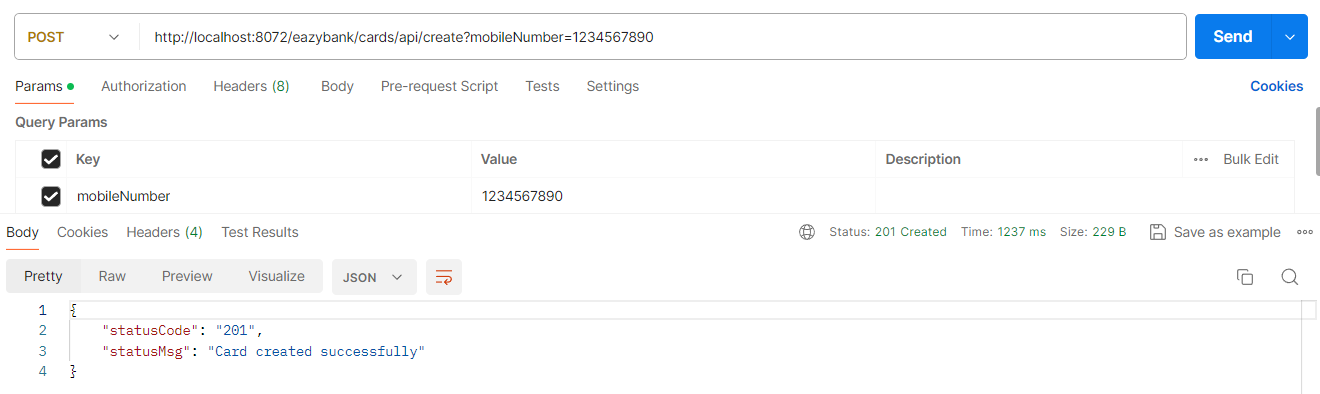


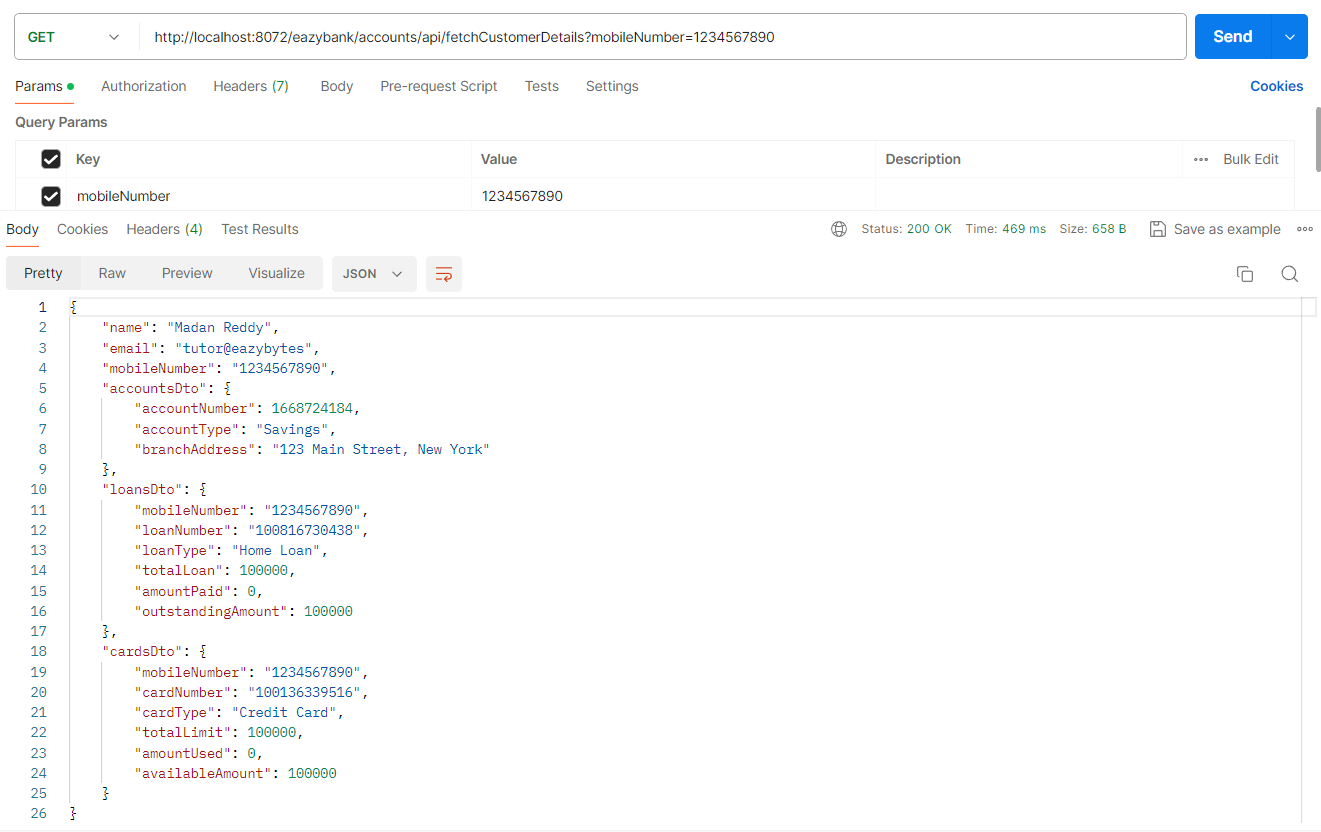
Run the microservices in below sequence

Configserver, eureka, accounts, loans, cards and gatewayserver









Watch the logs are getting printed in the console.

If we can scroll down first, they are going to tell you what are the pre requests.

So, the pre request are we should have it running Docker and Docker compose installed inside our local system.

<https://github.com/grafana/loki/tree/main/examples/getting-started>

<https://github.com/grafana/loki/blob/main/examples/getting-started/docker-compose.yaml>

<https://github.com/grafana/loki/blob/main/examples/getting-started/loki-config.yaml>

<https://github.com/grafana/loki/blob/main/examples/getting-started/alloy-local-config.yaml>

We will try to implement all these changes inside our Docker compose folder, present inside the workspace location.

Copy and paste the following commands into your command line to download

D:\Experiments\Microservices\sb-bank-application\docker-compose\default\alloy-local-config.yaml

D:\Experiments\Microservices\sb-bank-application\docker-compose\default\loki-config.yaml

D:\Experiments\Microservices\sb-bank-application\docker-compose\default\docker-compose\_grafana.yaml

Implementing Logging using Grafana, Loki & Promtail – Part 2

Generate all docker images with new tag

Make sure you have jib plugin

<plugin>

      <groupId>com.google.cloud.tools</groupId>

      <artifactId>jib-maven-plugin</artifactId>

      <version>3.4.1</version>

       <configuration>

           <to>

              <image>nileshzarkar/${project.artifactId}:s11</image>

            </to>

       </configuration>

  </plugin>

D:\Experiments\Microservices\sb-bank-application\accounts> mvn compile jib:dockerBuild

D:\Experiments\Microservices\sb-bank-application\loans> mvn compile jib:dockerBuild

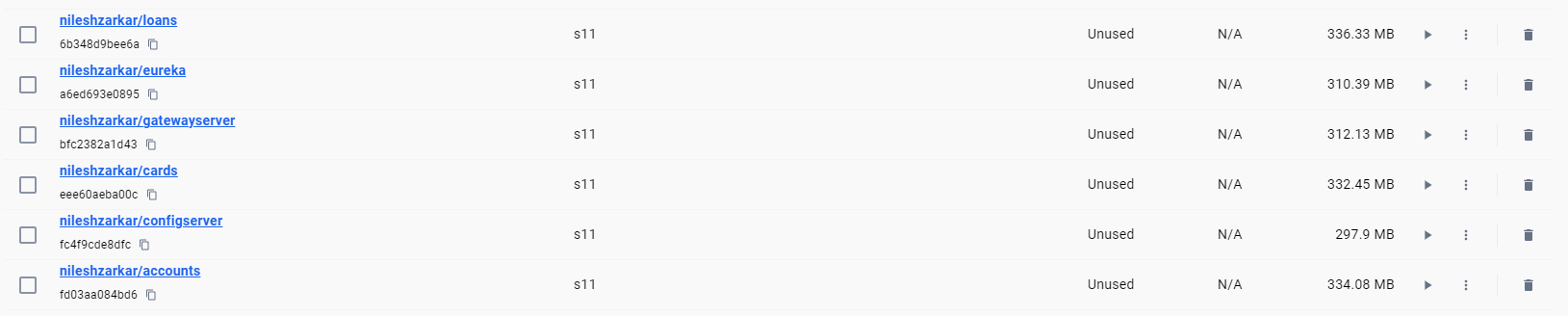
D:\Experiments\Microservices\sb-bank-application\cards> mvn compile jib:dockerBuild

D:\Experiments\Microservices\sb-bank-application\configserver> mvn compile jib:dockerBuild

D:\Experiments\Microservices\sb-bank-application\eureka> mvn compile jib:dockerBuild

D:\Experiments\Microservices\sb-bank-application\gatewayserver> mvn compile jib:dockerBuild

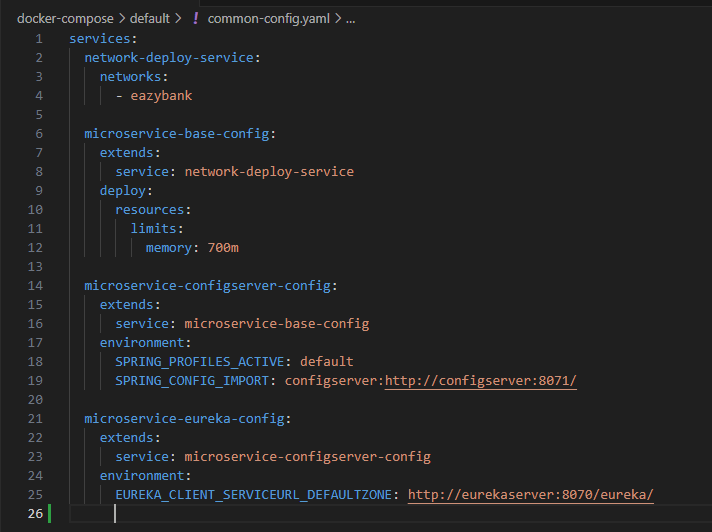
Docker push command

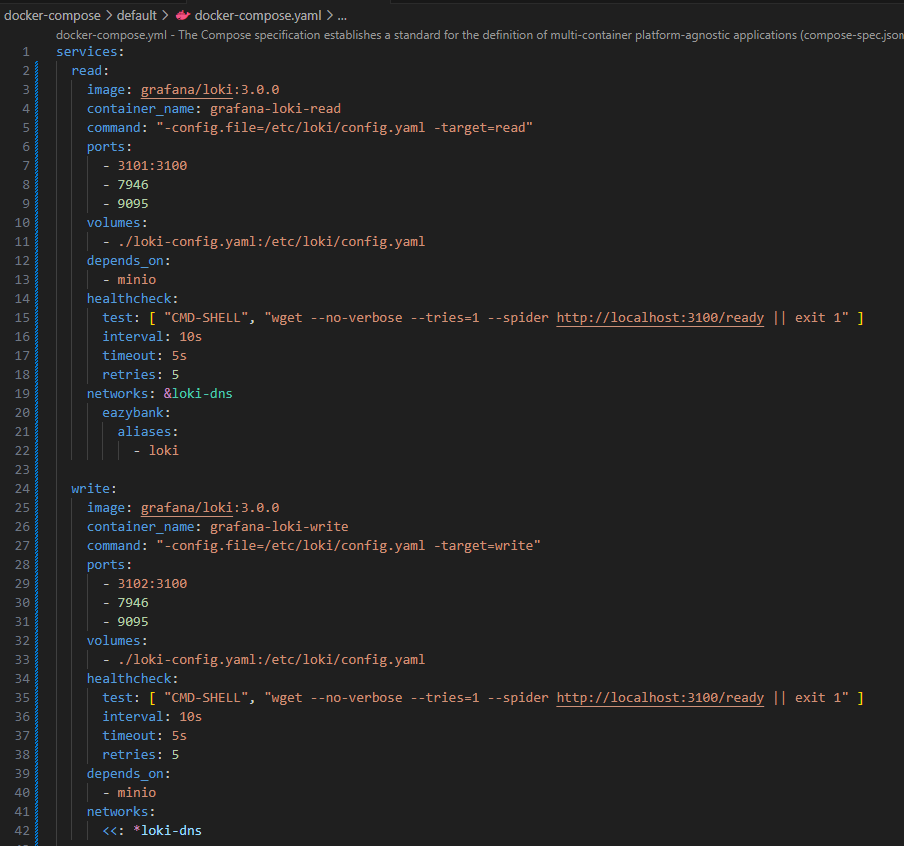


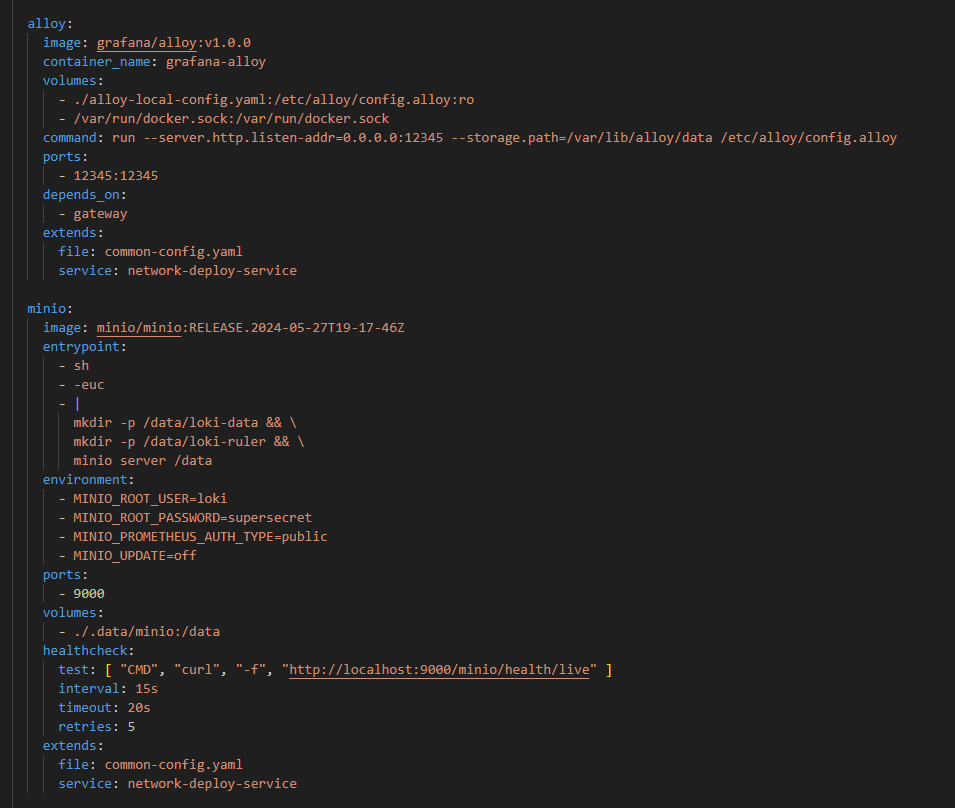
Update the original docker-compose.yaml as below :  
  
Basically integrate the changes of   
D:\Experiments\Microservices\sb-bank-application\docker-compose\default\docker-compose\_grafana.yaml

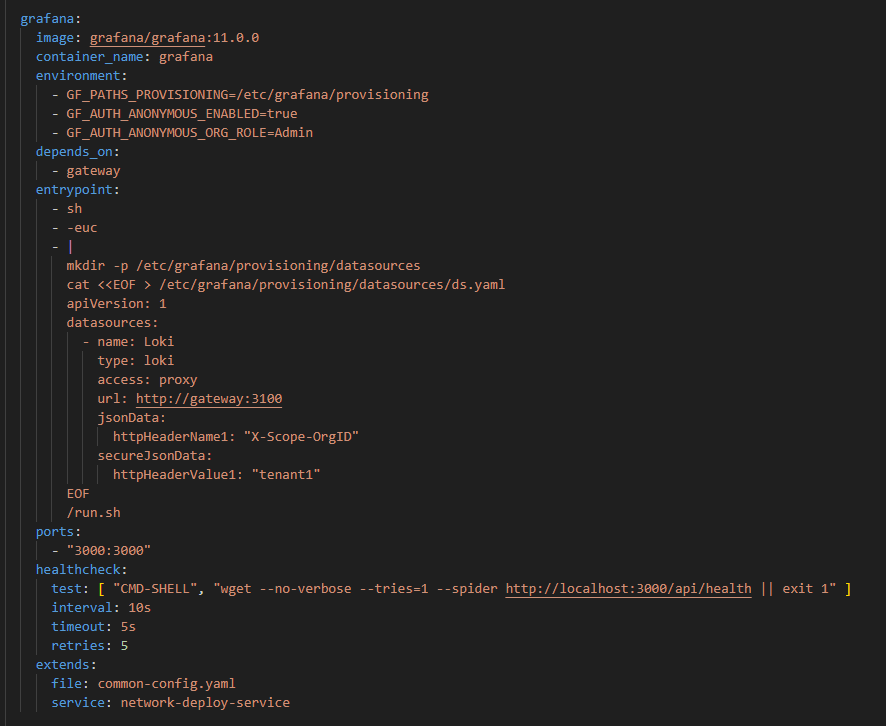
in   
  
D:\Experiments\Microservices\sb-bank-application\docker-compose\default\docker-compose.yaml

D:\Experiments\Microservices\sb-bank-application\docker-compose\default\common-config.yaml

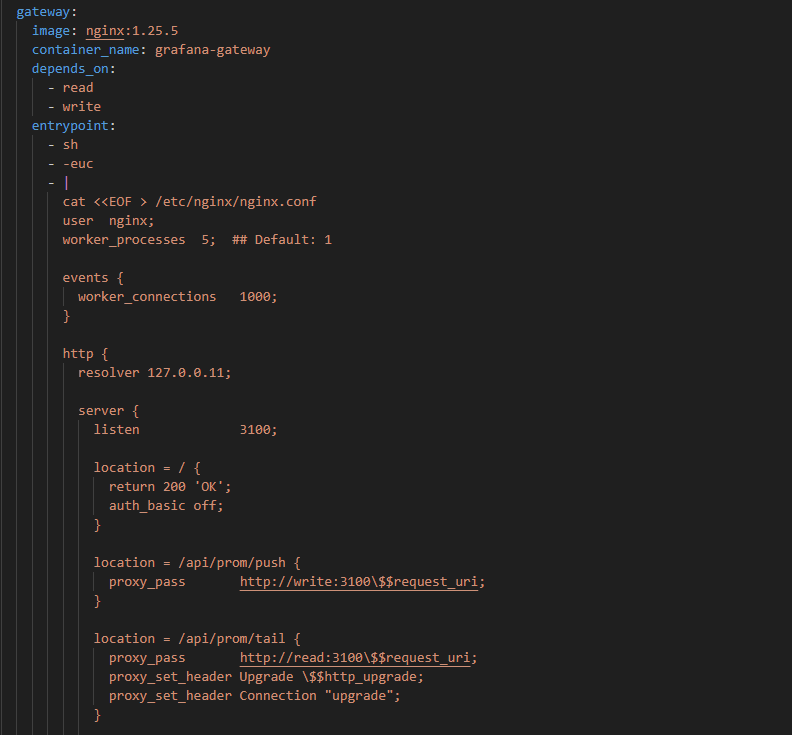


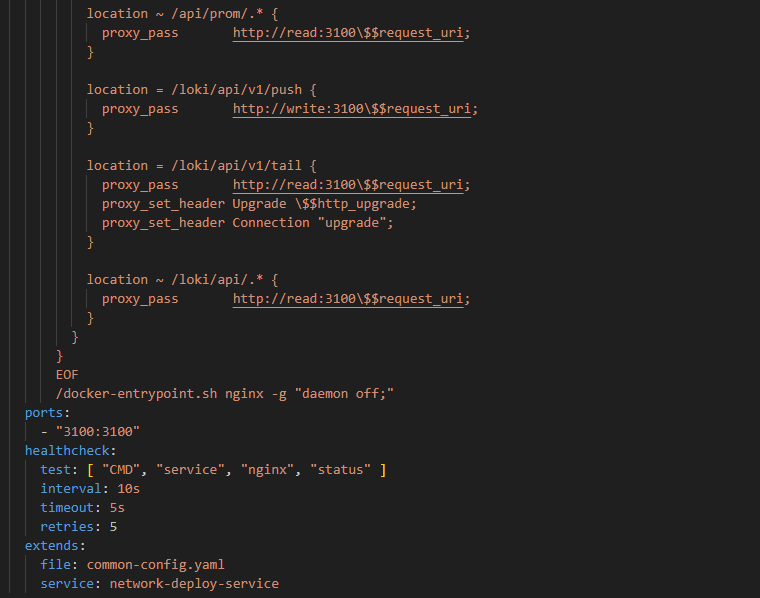
D:\Experiments\Microservices\sb-bank-application\docker-compose\default\docker-compose.yaml  
  


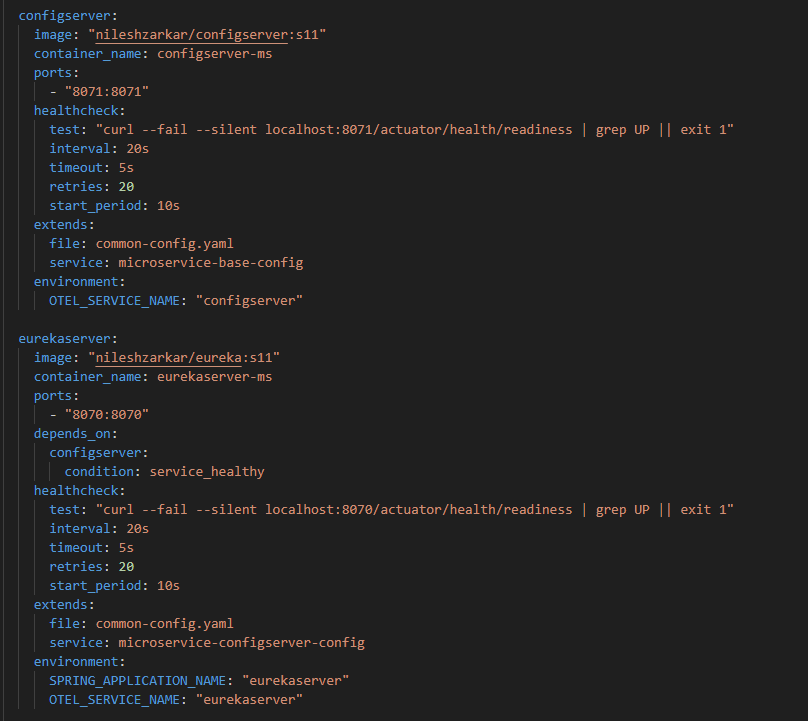


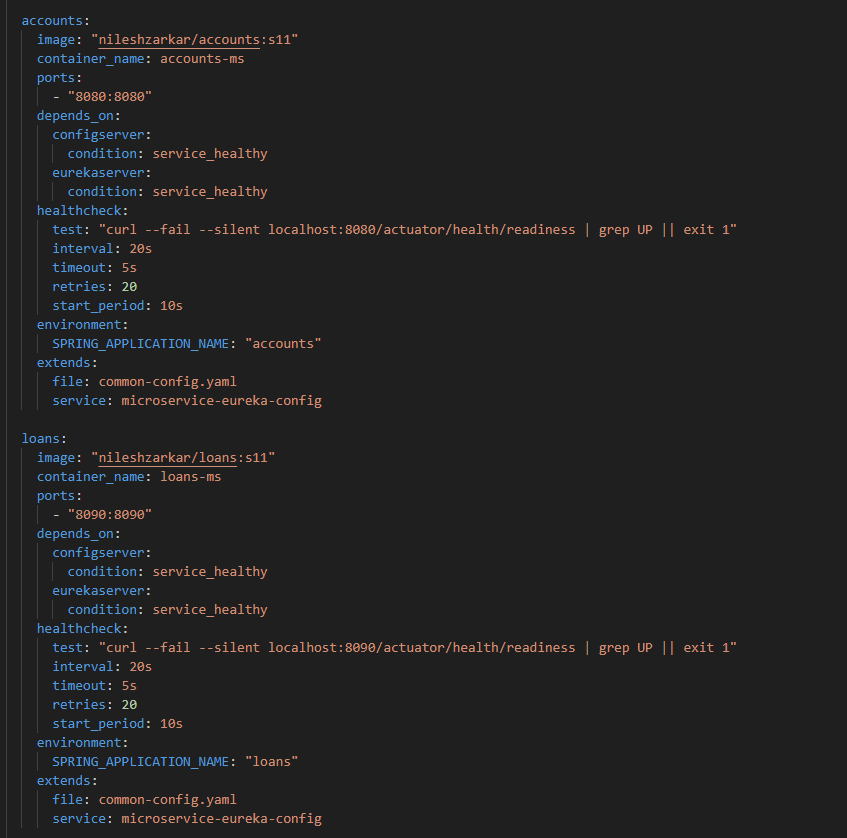


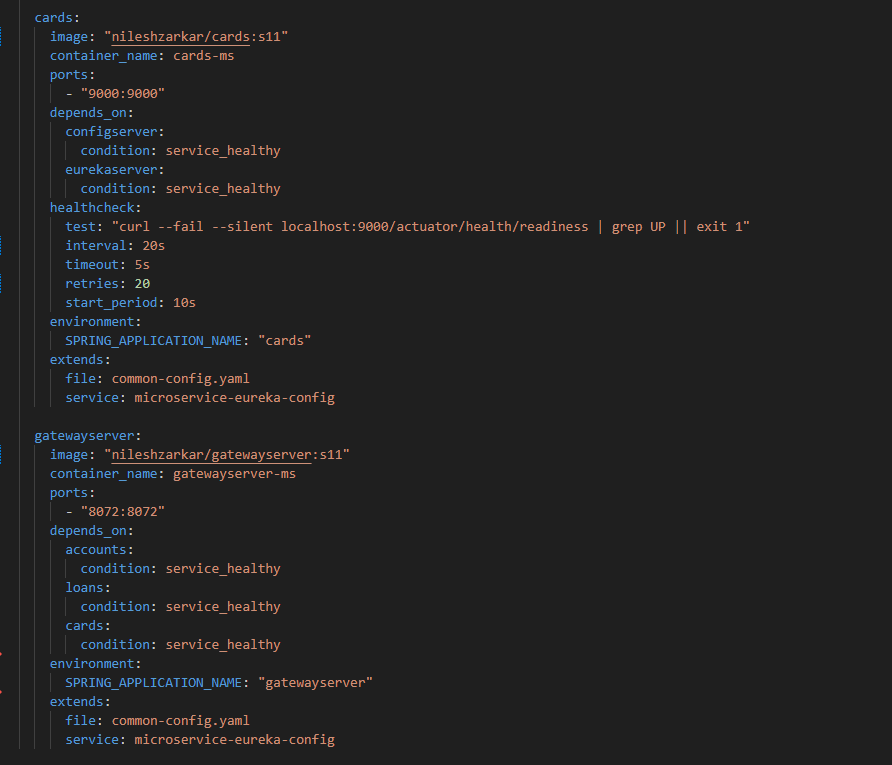


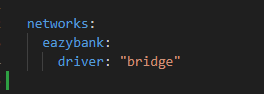






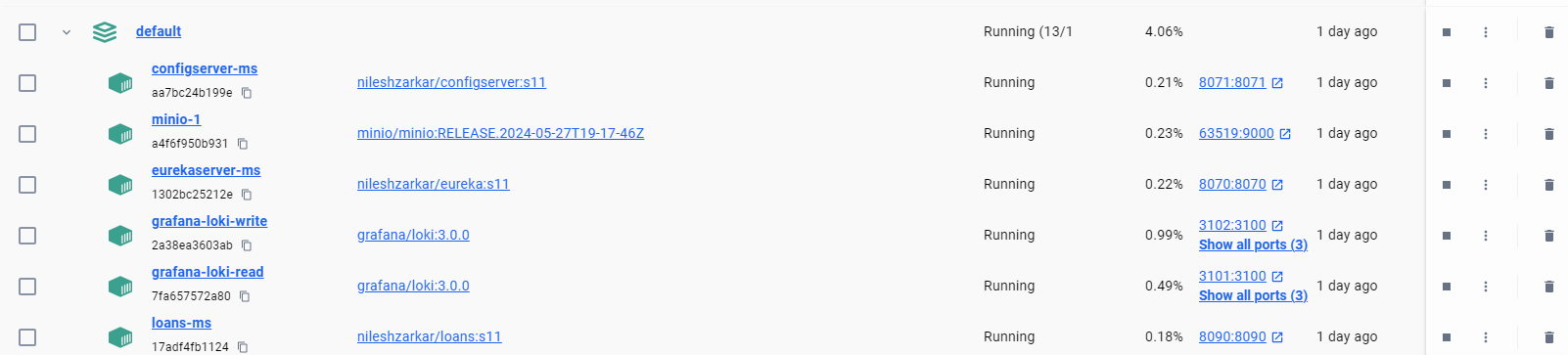






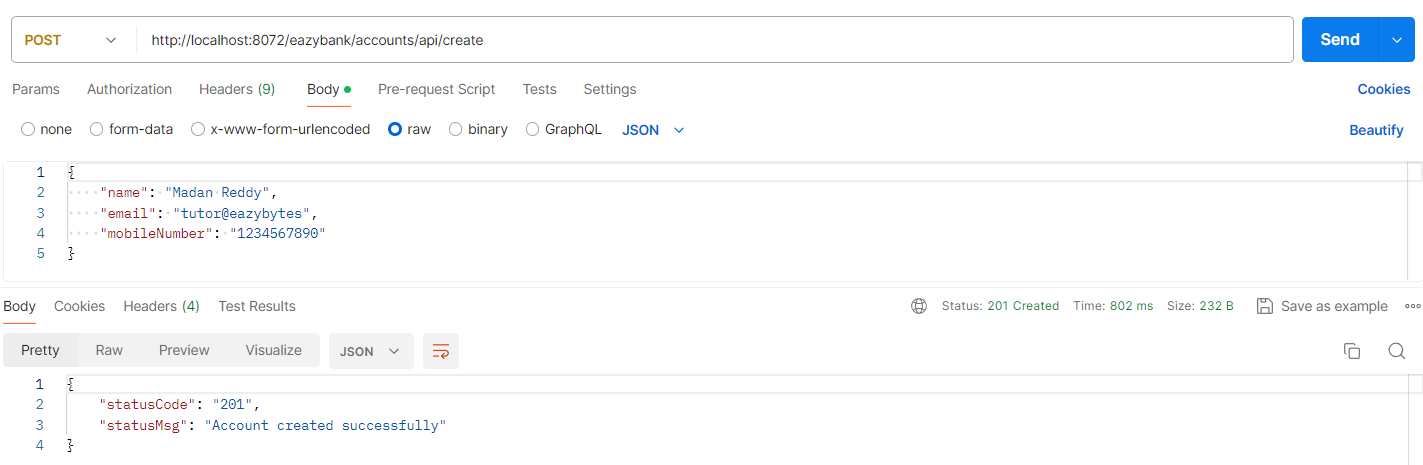
Implementing Logging using Grafana, Loki & Promtail – Part 3

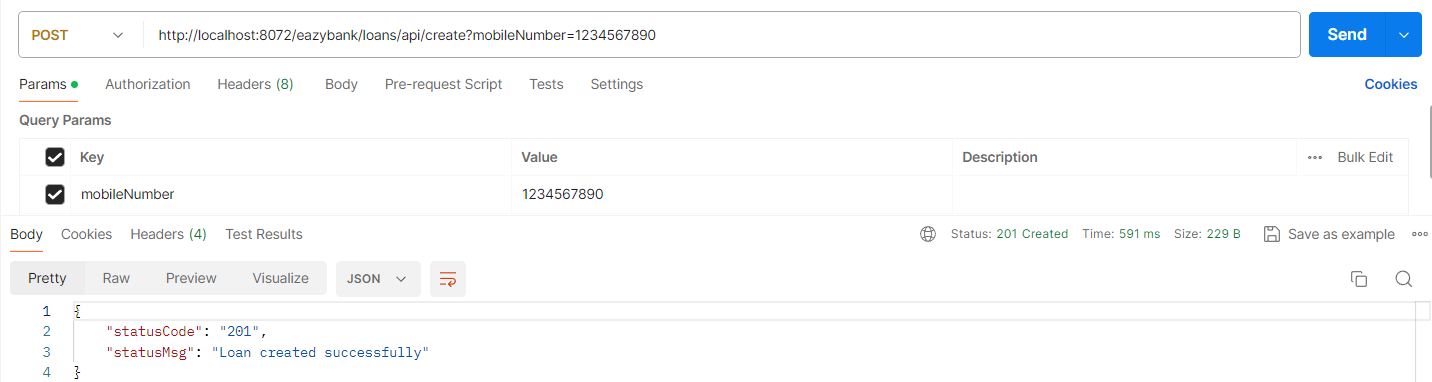
****

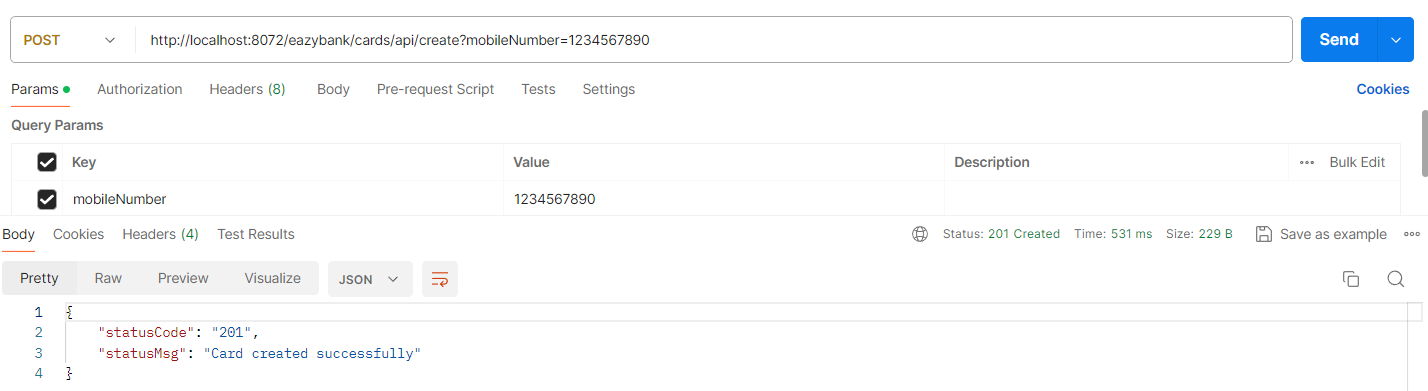
****

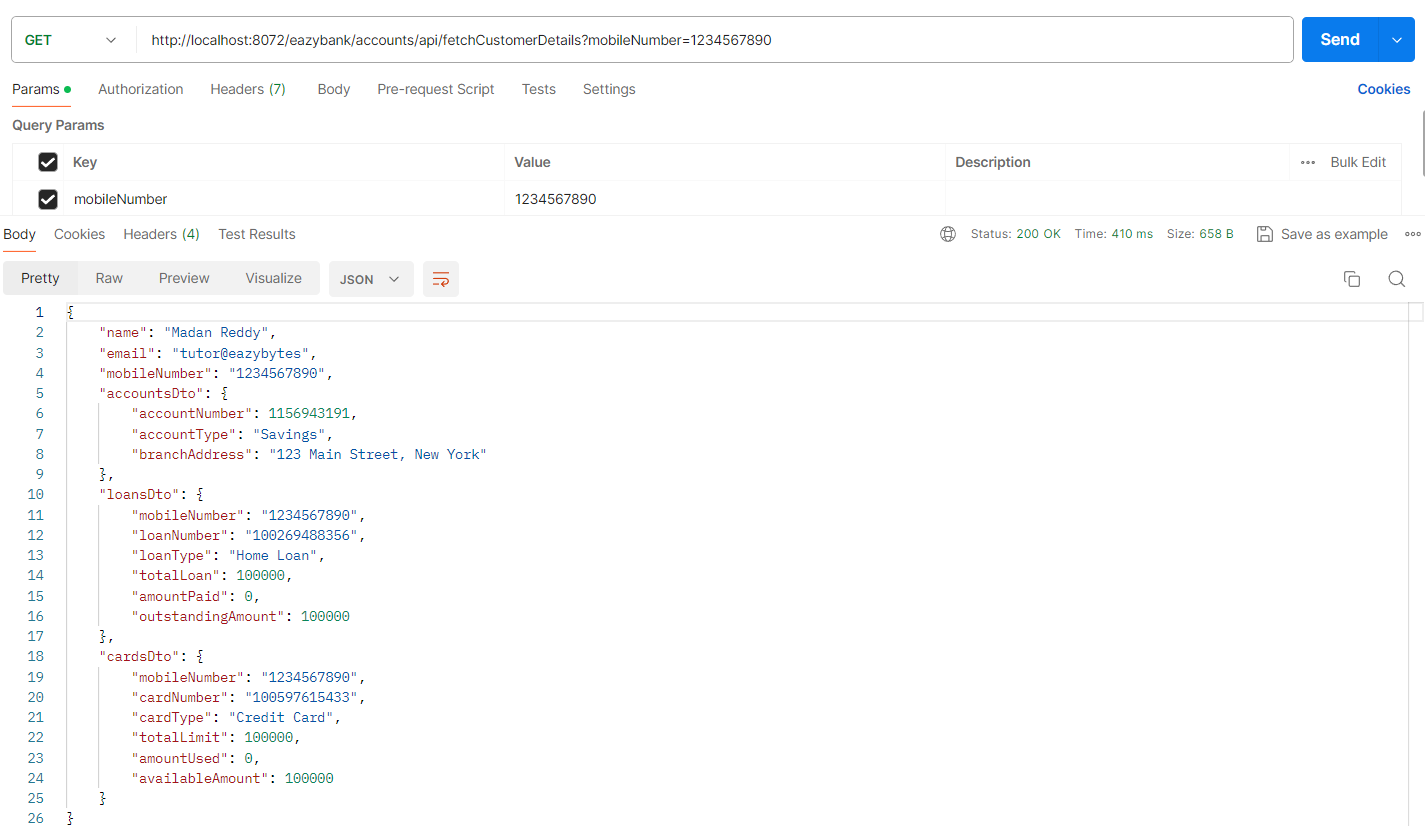
****

Check all containers are started and check the basic microservices functionality

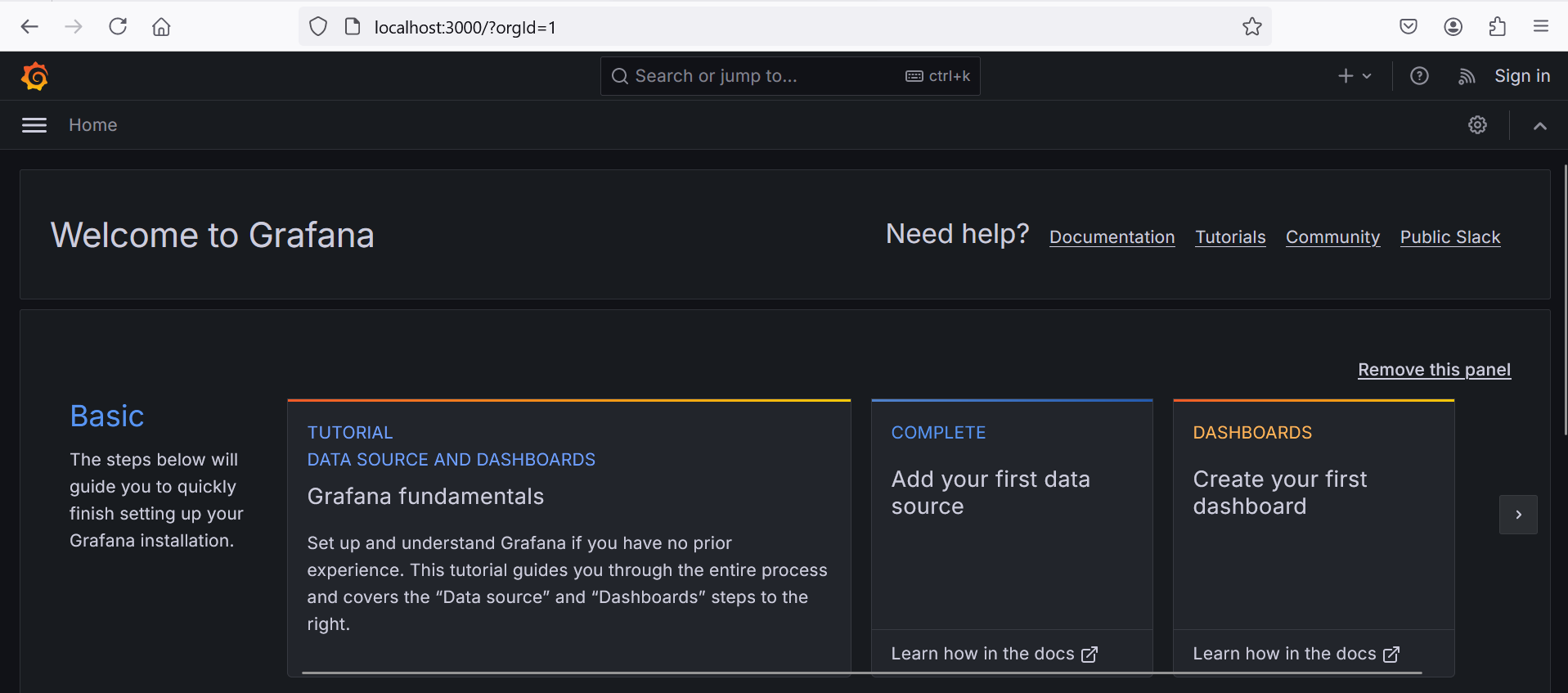






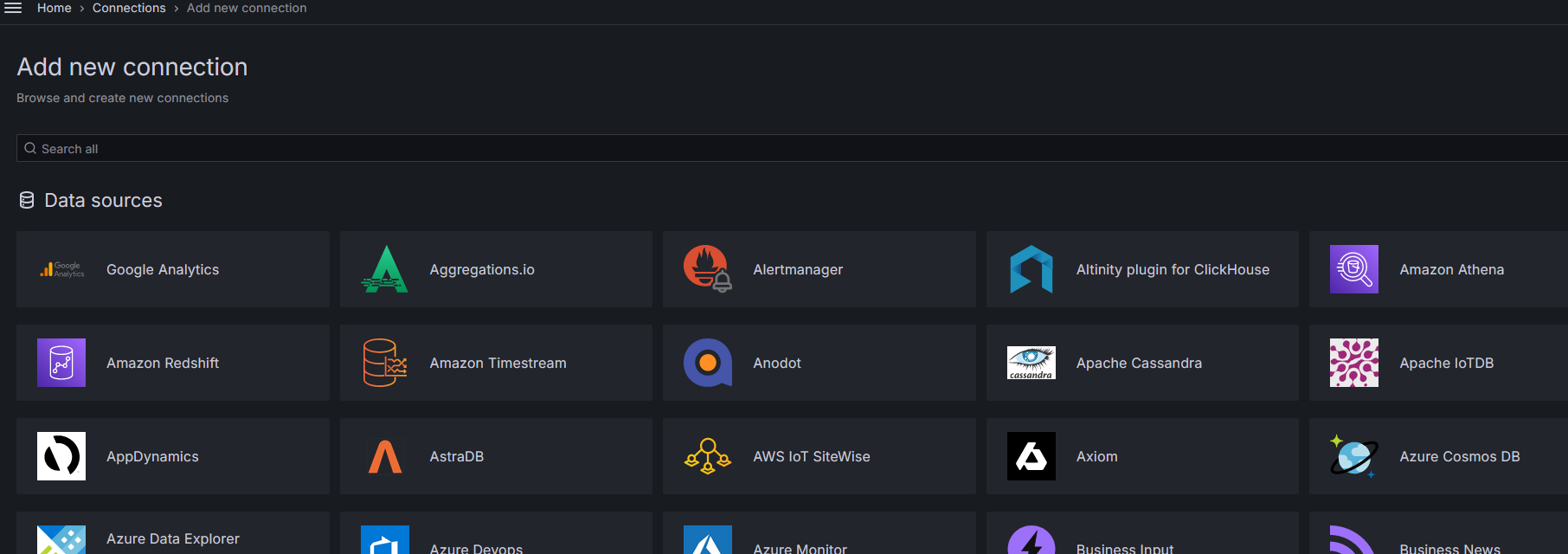


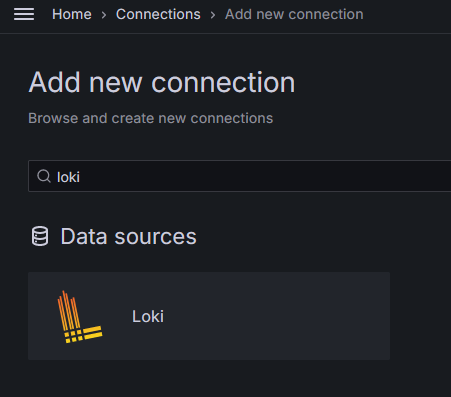
<http://localhost:3000/?orgId=1>



Like I said before, using Grafana, we can try to search the logs present inside the Loki system.

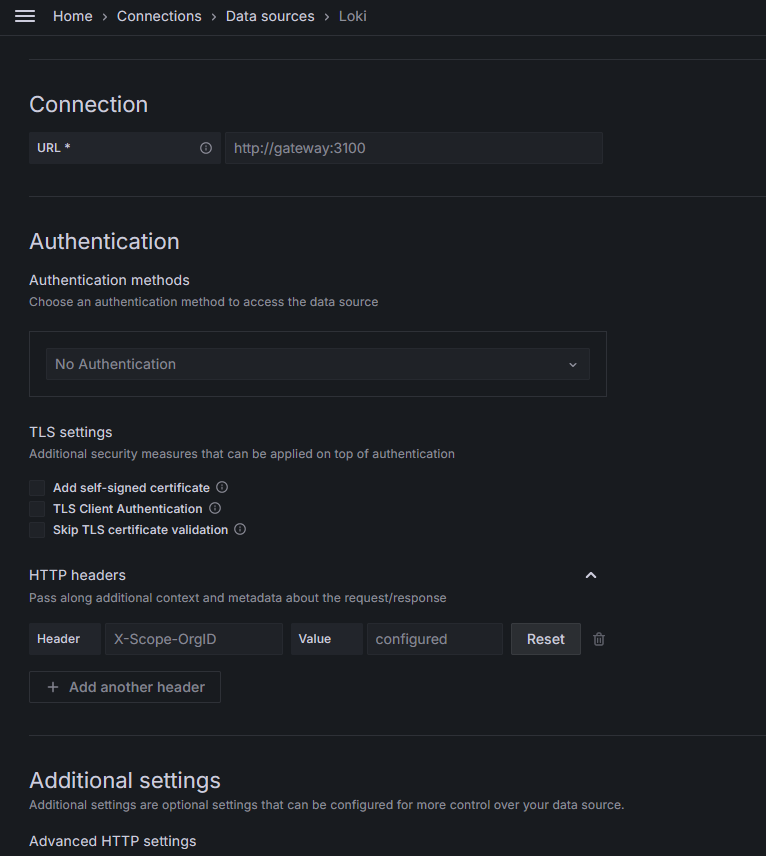
So how the link between the Grafana and Loki is established.





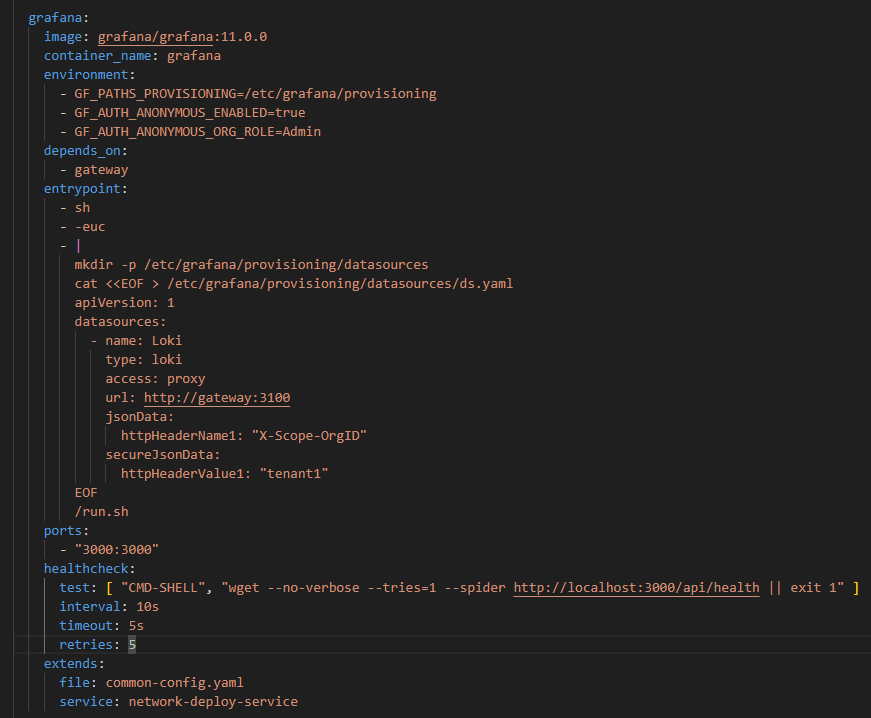
As of now you can see by default a Loki related data source is created if I can open this.

So, there is an URL configured using which my Grafana can connect with my Loki.



So, there is an URL configured using which my Grafana can connect with my Loki.

There are some headers also created, so all these connection details we have mentioned inside our Docker compose file.



So, inside the Docker compose file, if you can go to the service details related to Grafana here under

the Grafana service, if you can scroll to the entry point command, you can see we are trying to create

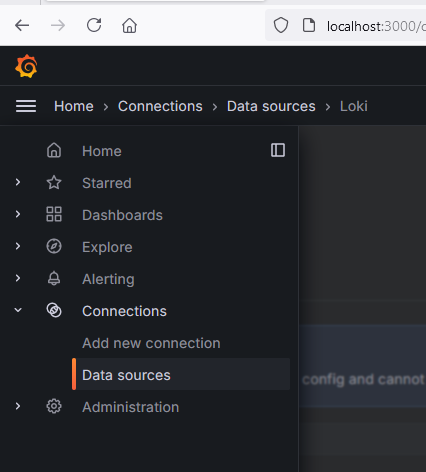
a data sources with the name Loki of Type Loki and we have mentioned the same URL and we have provided the same header and the value with the help of all these configurations.

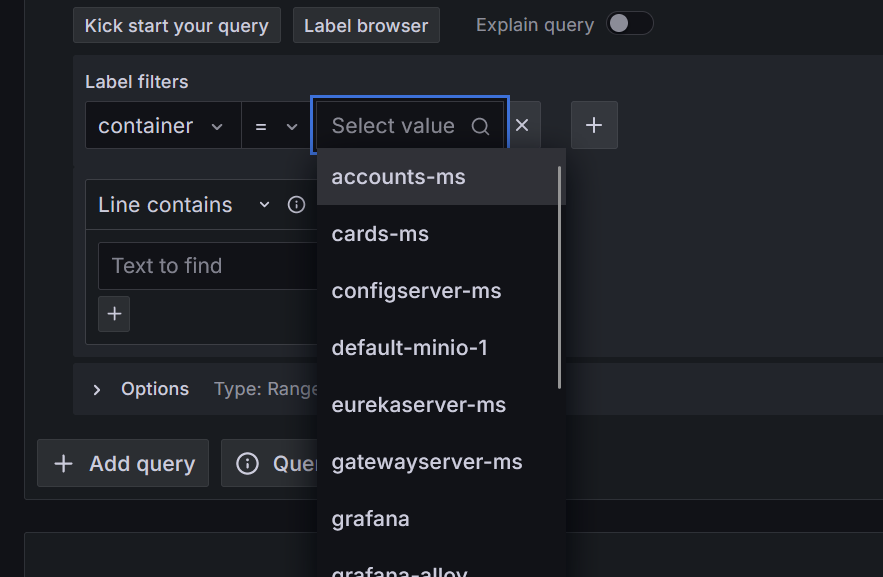
So, using the same data source configurations while my Grafana service is getting created, the data

source details are connection.

Details to the Loki is automatically created.

If you do not configure this, you need to manually create these connection details inside your Grafana by using these add new data resource.



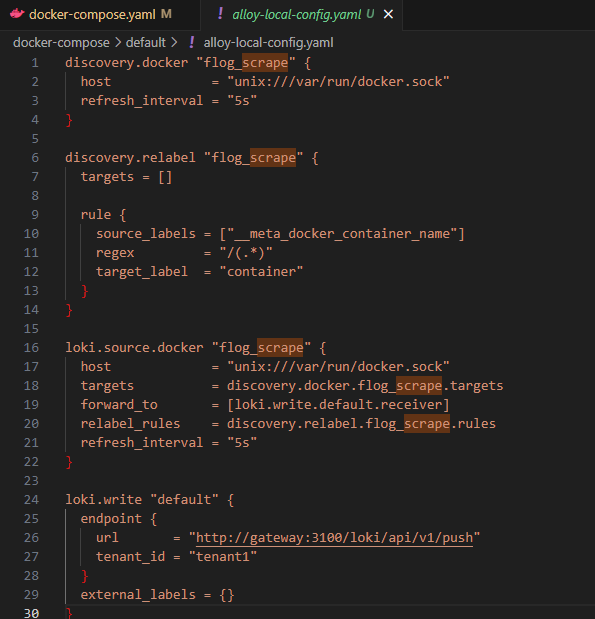


I can go to the explore button that we have here.

So here under the Select label I am going to select the label, which is container.

So, the same container label we have defined inside the alloy-local-config.yaml

So, you can see all the logs that are scraped by my alloy is going to be assigned to the target label container and the source label is going to be what is a Docker container name.



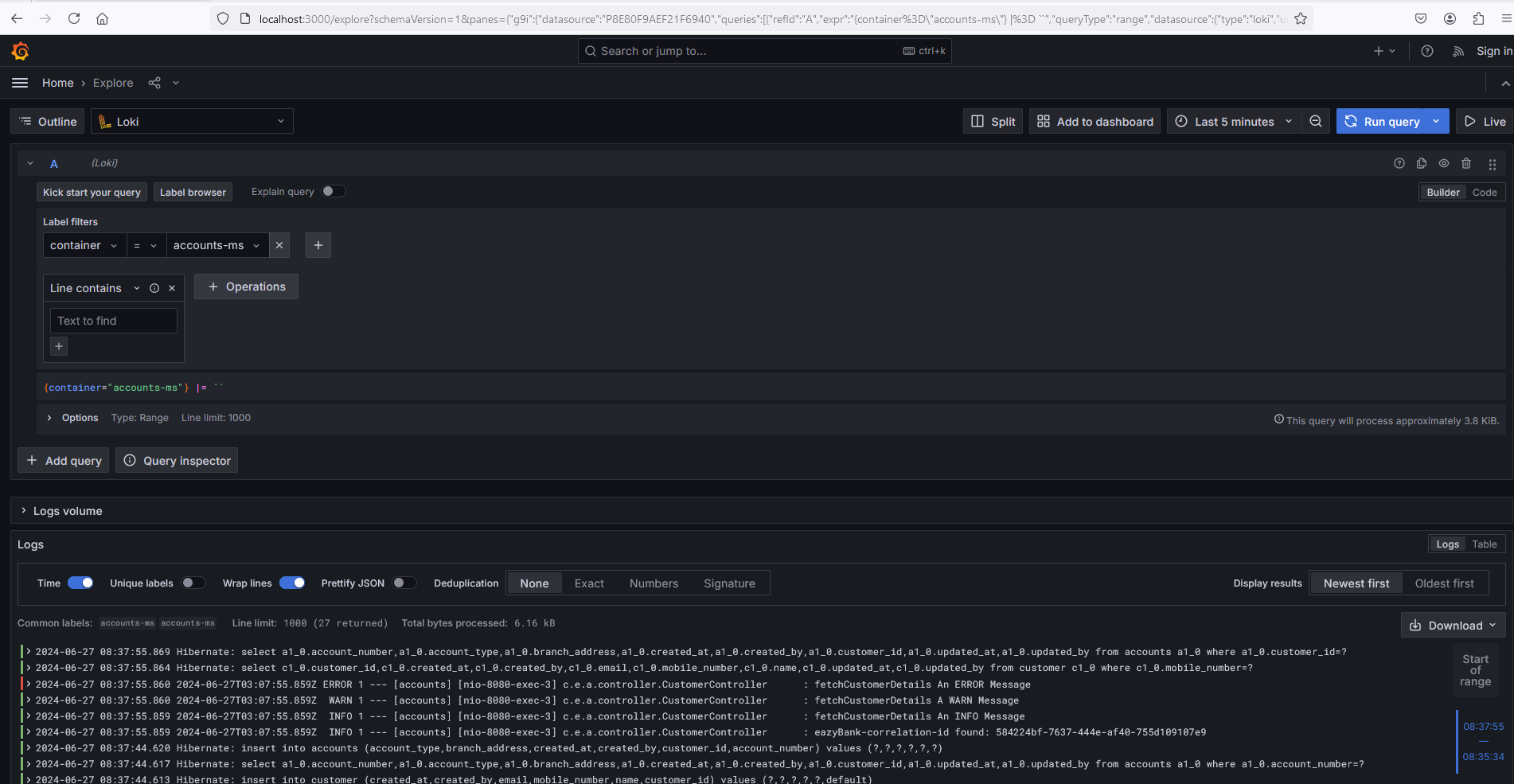
Once I select this label filter, you can see under the value I have all my docker container names.

For example, if I want to see the logs related to accounts micro service, I can select the same and

post that I can click on this run query.

With that, you should be able to see all the logs related to the accounts micro service inside your

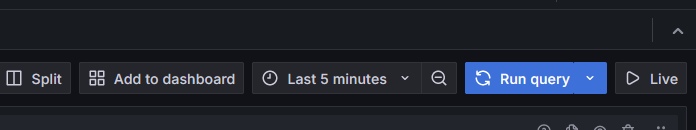
Grafana UI itself.



If needed, you can click on this live streaming with that any logs that are being generated behind

by your accounts micro service container, they will immediately come here after every five seconds.

So that is a power of Grafana here.



As a developer, I did not make even one line change inside my microservice.

All these log aggregations are centralized.

Logging is happening automatically with the help of Grafana Loki and prompt.

And here you may have a question like We are able to use these tools like Grafana Loki and alloy inside our containers.

So that is why always set up these inside your dev environment, qa environment, and production environment.

And of course, when you try to setup this in production environment, you may need to take the help from your platform team to configure some cloud storage.

As of now, all the logs that we generated is getting saved inside the Minio inside our local system

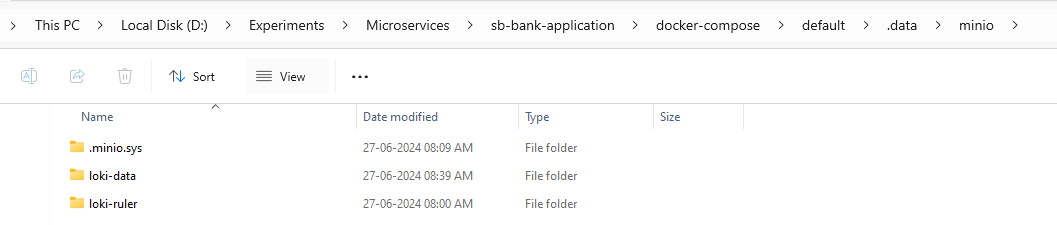
itself.

As of now, all the logs that we generated is getting saved inside the Minio inside our local system

itself.

I can also show you the same. So here you can see I came to my workspace location.

I am going to navigate to the section, underscore 11 and post that Docker compose.



So here you can see there is a folder created with the name .data and inside this and there are other folders related to Loki data and Loki ruler.

So, the same folder which is .data we mounted into the Docker container using volumes.



So, if you can search for Dot data, you can see under Minio service we have mounted the local folder

which is dot data Minio to the container by copying into your folder with the name data.

So whatever logs that are saved inside my local folder, they will get copied or mounted immediately

to this folder present inside my docker container.

But in real production applications we can leverage cloud storage systems like AWS, S3 or any other

cloud systems.

That way we can store any number of logs for any number of microservices.

Managing metrics & monitoring with Actuator, Micrometer, Prometheus & Grafana

Setup of Prometheus inside microservices

Demo of Prometheus

Demo of Prometheus & Grafana integration

Demo of Grafana inbuilt & custom Dashboards

Create Alerts & Send notifications using Grafana – Approach 1

Create Alerts & Send notifications using Grafana – Approach 2

Introduction to Distributed Tracing in microservices

Introduction to OpenTelemetry

Implement OpenTelemetry changes inside microservices

Implementing Tracing using Grafana, Tempo & OpenTelemetry – Part 1

Implementing Tracing using Grafana, Tempo & OpenTelemetry – Part 2

Implementing Tracing using Grafana, Tempo & OpenTelemetry – Part 3

Navigating to Tempo from Loki logs

Conclusion of Observability and Monitoring