Team Distributed-project report

Zhilin Yang

<20204619>

Yuan Yang

<20210849>

Yan Liu

<22204504>

▼ **Synopsis:**

Describe the system you intend to create:

What is the application domain?

HR management

What will the application do?

HR management, salary management, and user authentication.

▼ **Technology Stack**

List of the main distribution technologies you will use

- *Tech 1* ***Spring Cloud Gateway*** *(instead of Zuul): API gateway.*
- *Tech 2* ***Spring Cloud Netflix Eureka****: Service discovery.*

· *Tech 3* ***Resilience4j****: Circuit breaker. Hystrix which was mentioned in the class is also a great circuit breaker but Netflix announced that Hystrix is entering maintenance mode in 2018****. [1]][2][4]***

· *Tech 4* ***Spring Cloud Sleuth and Zipkin****: Distributed Tracing(Zipkin to visualise trace information through UI)*

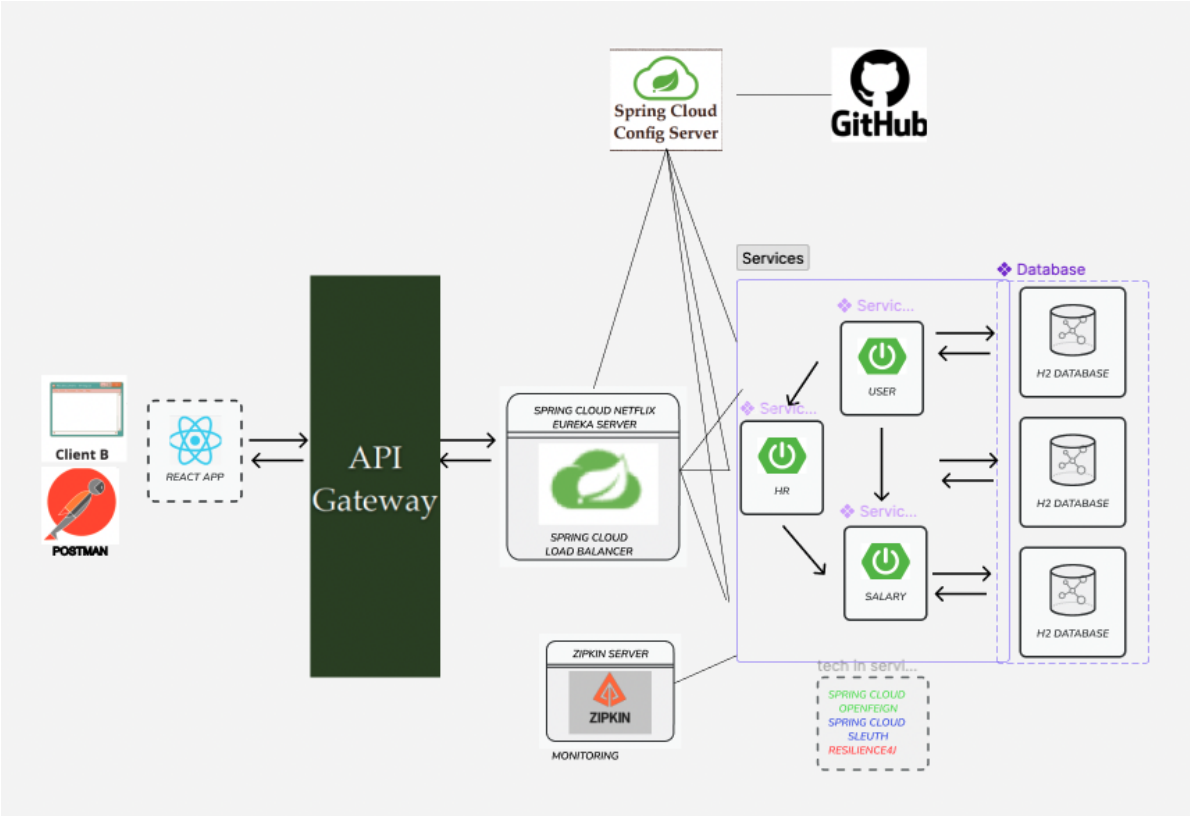
· *Tech 5* ***Spring Cloud Config Server****: For being able to reload the configuration changes without requiring to restart the application.*

· *Tech 6* ***Eureka, Open Feign, and Spring Cloud LoadBalancer****: Load Balancing. (Spring Cloud Netflix Ribbon has been deprecated and is not included in the 2020.0.0 release train. Spring Cloud LoadBalancer is the suggested alternative.)* ***[3][5]***

▼ 1 System Overview

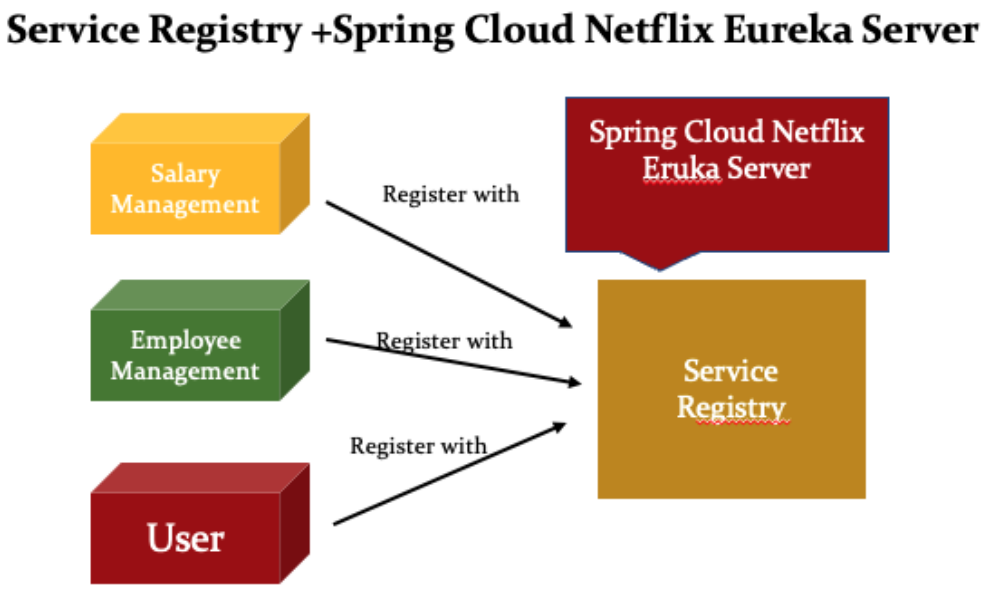
Describe the main components of your system.

Include your system architecture diagram in this section.



Explain how your system works based on the diagram.

1. Service Registry and Discovery with Spring Cloud Netflix Eureka Server:



In the microservices projects, Service Registry and Discovery play an important role because we most likely run multiple instances of services and we need a mechanism to call other services without hardcoding their hostnames or port numbers.

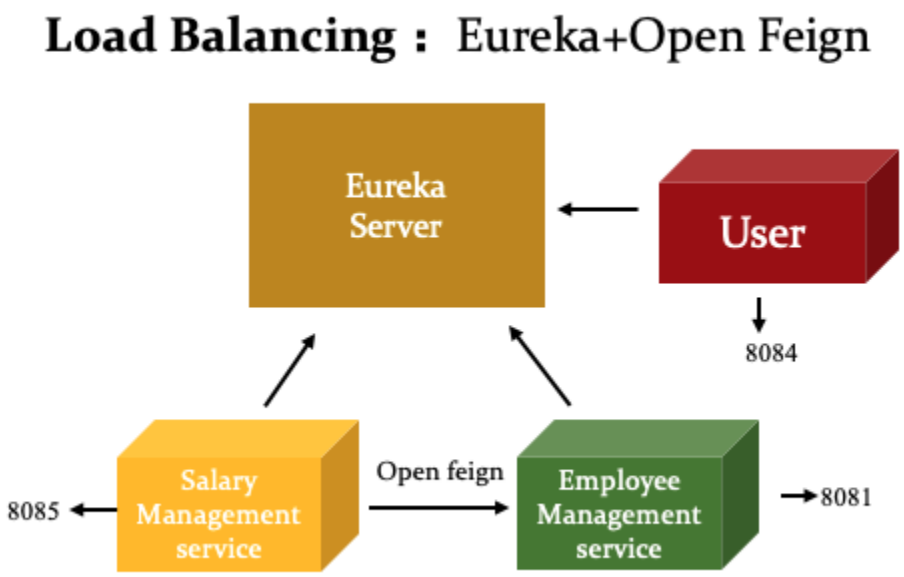
In Cloud environments service instances may come up and go down anytime. Eureka Server provide automatic service registration and discovery mechanism.

1. Each microservice that wants to be part of the service registry and discovery process runs a Eureka client. The client is responsible for registering the service with the Eureka server and sending heartbeat messages to the server to keep the service's registration up to date.
2. The Eureka server maintains a list of all the registered services and their current status. It also provides an API that allows other services to query the registry to discover the service's location and status.
3. When a microservice needs to invoke another service, it uses the Eureka client to discover the location of the service it needs to call. The client

queries the Eureka server to get a list of available instances of the service and their current status.

4. The Eureka client then chooses an available instance of the service and invokes it. If the service instance is not available or the connection fails, the Eureka client can try a different instance or retry the request.
5. The Eureka server periodically sends heartbeat messages to the registered services to check their status. If a service does not respond to the heartbeat message, the Eureka server removes the service from the registry. This helps ensure that the registry only contains available and healthy services.

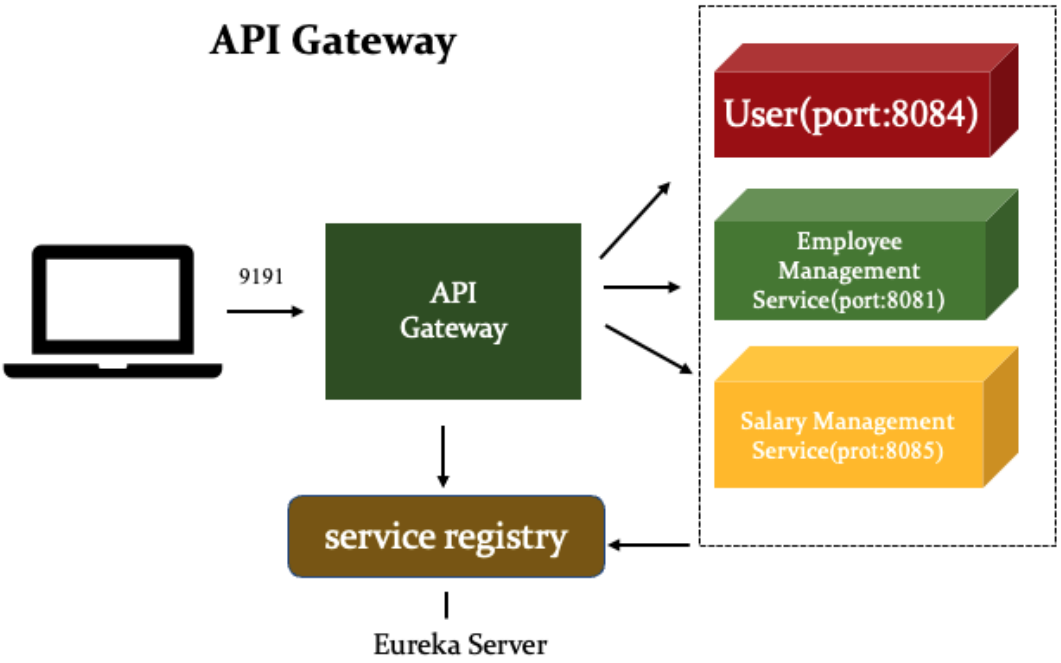
2. Load Balancing with Eureka, Open Feign and Spring Cloud LoadBalancer:



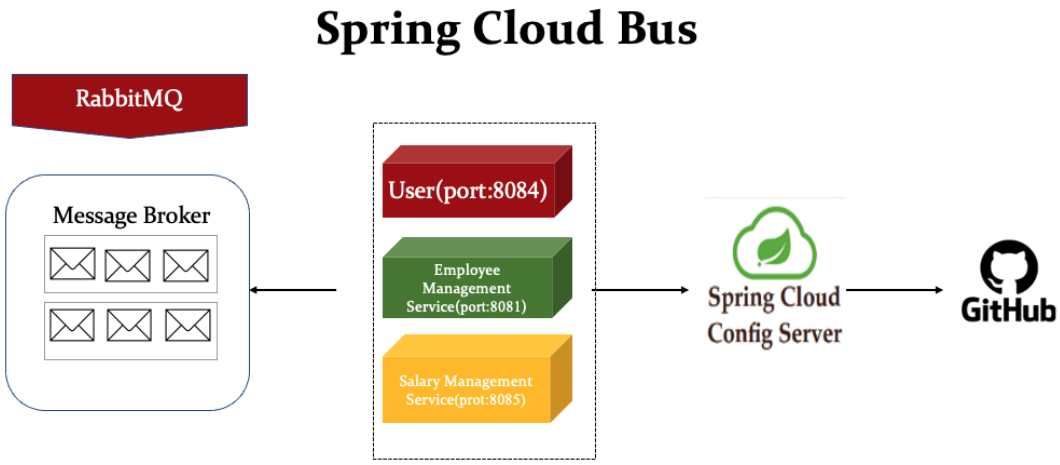
1. Each microservice that wants to be load balanced runs a Eureka client. The client is responsible for registering the service with the Eureka server and sending heartbeat messages to the server to keep the service's registration up to date.
2. The microservice that wants to invoke another service uses Open Feign to send HTTP requests to the service. Open Feign is a declarative HTTP client that simplifies the process of making HTTP requests to other services.

3. The microservice that wants to invoke another service also includes the Spring Cloud LoadBalancer dependency in its project. This enables the use of the `@LoadBalanced` annotation on the Feign client interface to enable client-side load balancing.
4. When a request is made to the Feign client, the Spring Cloud LoadBalancer client communicates with the Eureka server to discover available instances of the service.
5. The Spring Cloud LoadBalancer client then delegates the actual load balancing to a load balancer strategy, such as the Round Robin strategy, which distributes requests evenly across the available instances of the service.
6. The Feign client sends the request to the selected instance of the service and receives a response. If the connection fails or the service instance is not available, the Spring Cloud LoadBalancer client can try a different instance or retry the request.

3. API Gateway with Spring Cloud Gateway: The API gateway listens for incoming HTTP requests and routing them according to the configured routes(it uses service registry (Eureka Server)to discover correct hostname and port of these microservices.).



4. Auto refresh config changes with Spring Cloud Config Server, RabbitMQ and Spring Cloud Bus:

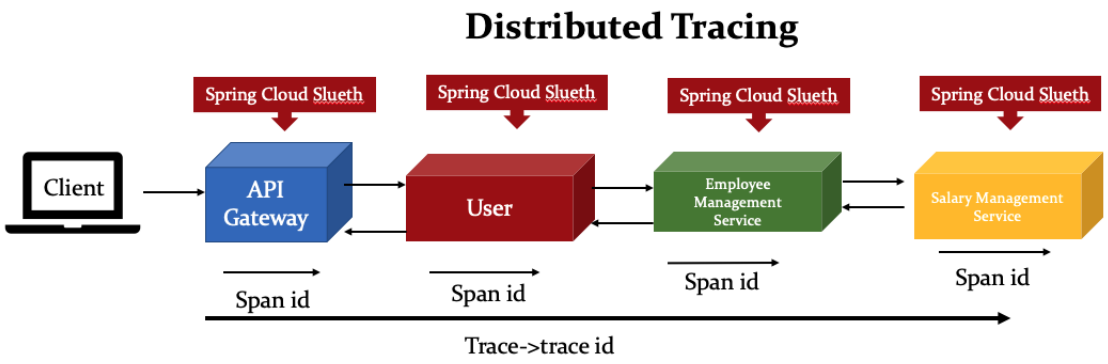


Spring Cloud Config is a centralized configuration management tool that allows us to store your application's configuration in a Git repository and access it from multiple applications. Spring Cloud Bus is a message-based event bus that

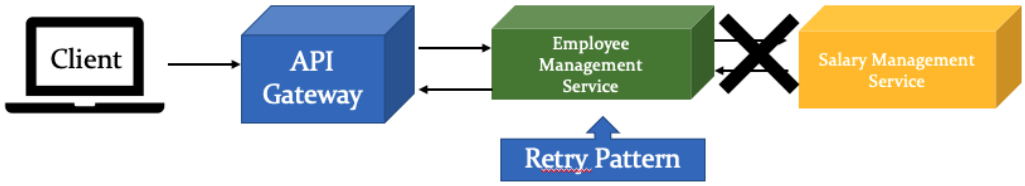
allows you to broadcast configuration updates to all connected applications. RabbitMQ is a message broker that can be used as the message bus for Spring Cloud Bus.

1. The Spring Cloud Config Server receives the refresh message and retrieves the updated configuration from the Git repository. It then broadcasts the updated configuration to all connected Spring Cloud Config Clients using the Spring Cloud Bus and RabbitMQ.
2. The Spring Cloud Config Clients receive the updated configuration and refresh their internal configuration cache. Any beans that are annotated with `@RefreshScope` will be re-initialized with the new configuration values.

5. Distributed Tracing with Spring Cloud Sleuth and Zipkin: We use Spring Cloud Sleuth for distributed tracing and Zipkin to visualize trace information through UI. We can now trace the flow of requests through a distributed system and see the details of each span, such as the operation name, start and end timestamps, and any tags or logs associated with the span.



6. Retry Pattern Implementation with Resilience4j: The retry pattern allows a failed request to be retried after a certain amount of time has passed. In this case is 2s and it will retry 5 times max.



▼ 2 ***Explain how your system is designed to support scalability and fault tolerance.***

1. scalability

*We choose Microservices Architecture using Spring boot and Spring Cloud. Compared to classical monolithic Architecture, this approach offers many benefits for development, extensibility, scalability and integration.**[6]***

See 5. technology benefit of

A. docker and docker compose

C. Spring Cloud Sleuth and Zipkin

D Spring Cloud Netflix Eureka Server ***and LoadBalancer and Open Feign***

E ***Spring Cloud Config Server***

F ***Spring Cloud Gateway***

k8s to be continued.

2.fault tolerance

See technology benefit of G.Resilience4j

Reflections

▼ 3 ***What were the key challenges you have faced in completing the project? How did you overcome them?***

1. Docker compose all the micro services：I read all the documentation and bought a course on Udemy. I finish dockerize all the micro services .Still haven't figure out the right docker compose file.

2.spring security:jwt

▼ 4 ***What would you have done differently if you could start again?***

1. banking system or E-commerce instead of HR management.
2. *Try more trending tech stack other than REST. Such as gRPC: graphQL or message queue like RabbitMQ or Kafka.*

▼ 5 ***What have you learnt about the technologies you have used? Limitations? Benefits?***

A. docker and docker compose:

Benefits:

We use spring-boot-maven-plugin to help us to generate a docker image based on the pom file of each microservices using Buildpacks. So we don't have to worry about creating dockerfiles.The command I used:mvn spring-boot:build-image -DskipTests

Scalability: Docker makes it easy to scale our applications up or down, by adding or removing containers as needed. This can be especially useful when using a container orchestration tool such as Kubernetes.

Limitations:

1. Resource constraints: Docker containers run in a shared kernel and are limited in the resources they can use. This can be a problem if we need to run resource-intensive applications in containers.
2. Compatibility issues: Docker containers are isolated from the host system, which can cause compatibility issues with certain applications. Some applications may not work correctly in a containerized environment.
3. Lack of flexibility: Docker Compose is a tool for defining and running multi-container Docker applications, but it does not provide much flexibility for customizing the configuration of the containers.

4. Limited networking: Docker containers use a virtual network to communicate with each other and the host system, which can limit the options for networking and integration with other systems.
5. Security risks: Docker containers can pose security risks if they are not configured and managed properly. For example, containers can access host resources and potentially compromise the host system.
6. Lack of support for certain platforms: Docker is not supported on all platforms, including certain versions of Windows and macOS.

B. Spring boot project follows DTO patterns:

Benefits :

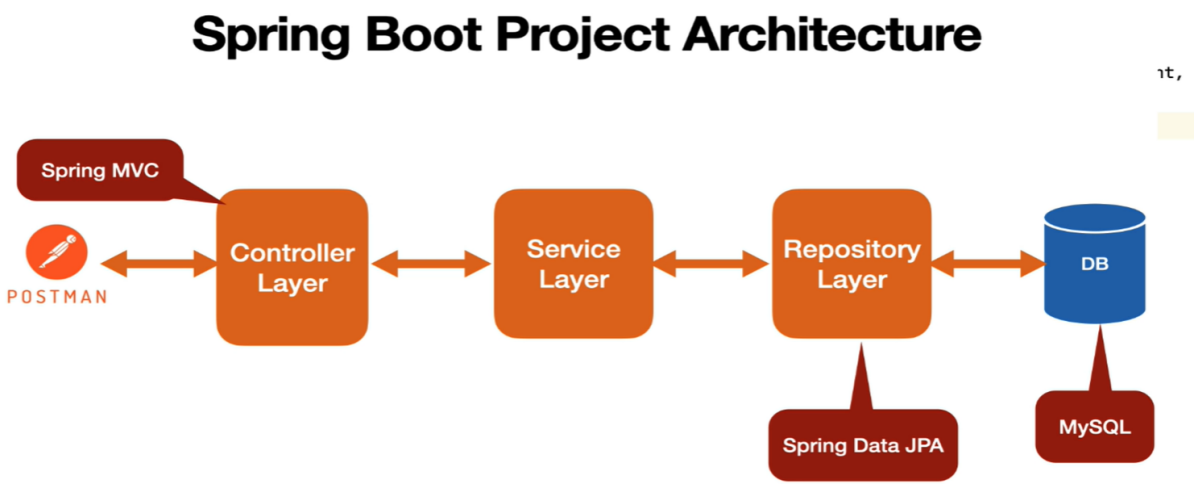
We create DTO to gather all the information and send it back to the client.

So there are 3 advantages:

1.*The client will reduce the number of remote calls .*

2.The server doesn't have to send the whole entity or domain object to the client, and will only send the required information to the client.

3. Converting entities to DTO helps hiding implementation details of domain objects (aka. entities).Exposing entities through endpoints can become a security issue. Using DTO we reduced the security risk of exposing more information which might contain sensitive information that we don't want to expose.



Limitations :

1. Limited flexibility: DTOs are designed to transfer data between layers of an application, but they do not provide much flexibility for customizing the data that is transferred. For example, may not be able to include certain types of data or custom objects in a DTO.
2. Dependency on the underlying data model: DTOs are usually based on the data model of the application, which means that changes to the data model may require changes to the DTOs as well. This can make it difficult to maintain the DTOs over time.
3. Limited validation capabilities: DTOs do not have built-in validation capabilities, which means that we need to implement our own validation logic to ensure that the data in the DTOs is valid.
4. Potential for data inconsistencies: DTOs do not have any built-in mechanisms to prevent data inconsistencies, which means that it is possible for the data in a DTO to become inconsistent with the data in the underlying data model.
5. Limited reuse: DTOs are usually specific to a particular application or use case, which means that they may not be reusable in other contexts.
6. Performance overhead: DTOs may introduce some performance overhead, as they require additional serialization and deserialization steps to transfer data between layers of the application. This can affect the overall performance of the application, especially if you are transferring large amounts of data.

C. Spring Cloud Sleuth and Zipkin:

Benefits :

1. Visibility: Distributed tracing allows you to see the flow of requests through a distributed system and understand how different microservices interact with each other. This can help you identify performance bottlenecks and debug errors.
2. Performance optimization: By identifying performance bottlenecks, you can optimize the performance of your system and improve the user experience.
3. Debugging: Distributed tracing can help you understand the root cause of errors and debug them more efficiently.

4. Observability: Distributed tracing provides a comprehensive view of your system and allows you to monitor its health and performance in real-time.

Limitations ：

1. Performance overhead: Adding Sleuth and Zipkin to your application can introduce some performance overhead, as it generates additional log messages and sends them to the Zipkin server for processing. This can affect the overall performance of your application, especially if you are running a high-traffic application.
2. Limited tracing capabilities: Sleuth and Zipkin are designed for distributed tracing in microservice-based applications. They may not provide the same level of tracing capabilities as more advanced tools such as AppDynamics or New Relic.
3. Dependency on Zipkin server: Sleuth relies on the Zipkin server to store and process trace data. If the Zipkin server is unavailable or experiencing issues, it can affect the ability of Sleuth to generate trace data.
4. Data privacy concerns: Zipkin stores trace data in a database, which can raise concerns about data privacy and security. You should carefully consider the data that is being collected and transmitted, and ensure that it is compliant with relevant laws and regulations.
5. Complex setup: Setting up and configuring Sleuth and Zipkin can be complex, especially if you are using a distributed architecture with multiple microservices. It requires some knowledge of distributed systems and trace data processing to get started.
6. Limited support for certain platforms: Sleuth and Zipkin may not be supported on all platforms and environments, and may require additional configuration to work correctly.

D. Spring Cloud Netflix Eureka Server *and LoadBalancer and Open Feign*

Benefits :

1. Service discovery: Eureka Server helps microservices discover and communicate with each other by maintaining a registry of all available services.

This allows microservices to find and communicate with each other without having to hardcode the locations of the other services.

2. Load balancing: Eureka Server can provide load balancing support to help distribute incoming requests evenly across multiple instances of a service. This can improve the performance and reliability of the overall system.
3. Fault tolerance: Eureka Server can help mitigate the effects of service failures by detecting when a service instance goes offline and removing it from the registry. This can help improve the fault tolerance of the overall system.
4. Scalability: Eureka Server can scale horizontally by adding more instances of the service registry to handle increasing workloads. This can help ensure that the service registry can keep up with the demands of a growing system.

Limitations ：

1. Dependency on a central server: Eureka Server relies on a central server to store and manage the registry of available services. This can be a single point of failure, as the entire system can become unavailable if the server goes down.
2. Performance overhead: Eureka Server generates additional traffic to maintain the registry of available services, which can introduce some performance overhead. This can affect the overall performance of the system, especially if we are running a high-traffic application.
3. Limited customizability: Eureka Server provides a fixed set of features and functionality, which may not be sufficient for certain use cases. It may not be possible to customize or extend the functionality of Eureka Server to meet specific requirements.
4. Limited support for certain platforms: Eureka Server may not be supported on all platforms and environments, and may require additional configuration to work correctly.
5. Security risks: Eureka Server stores sensitive information about the services in the system, which can raise concerns about data privacy and security.

E.*Spring Cloud Config Server*

Benefits :

1. Centralized configuration: Spring Cloud Config Server provides a central location for storing and managing configuration data, which makes it easier to manage the configuration of a large, distributed system.
2. Improved flexibility: Spring Cloud Config Server allows us to store configuration data in a variety of formats, including plain text, YAML, and JSON. This allows us to choose the format that is most suitable for our needs.
3. Support for multiple environments: Spring Cloud Config Server can store different versions of the configuration data for different environments, such as development, staging, and production. This allows us to easily switch between environments and use different configurations for each environment.
4. Improved security: Spring Cloud Config Server stores configuration data in a secure central location, which can help protect sensitive information from unauthorized access. We can also use encryption and other security measures to further secure the data.
5. Improved **scalability**: Spring Cloud Config Server can scale horizontally by adding more instances of the service to handle increasing workloads. This can help ensure that the configuration server can keep up with the demands of a growing system.
6. Improved reliability: Spring Cloud Config Server can help improve the reliability of the system by providing a central location for storing and managing configuration data. This can reduce the risk of data loss or corruption, and ensure that the configuration data is always available when needed.

Limitations：

1. Dependency on a central server: Eureka Server relies on a central server to store and manage the registry of available services. This can be a single point of failure, as the entire system can become unavailable if the server goes down.
2. Performance overhead: Eureka Server generates additional traffic to maintain the registry of available services, which can introduce some performance overhead. This can affect the overall performance of the system, especially if you are running a high-traffic application.
3. Limited customizability: Eureka Server provides a fixed set of features and functionality, which may not be sufficient for certain use cases. It may not be

possible to customize or extend the functionality of Eureka Server to meet specific requirements.

4. Complex setup: Setting up and configuring Eureka Server can be complex, especially if weare using a distributed architecture with multiple microservices. It requires some knowledge of distributed systems and service registry concepts to get started.

F.*Spring Cloud Gateway*

Benefits :

1. Centralized routing: Spring Cloud Gateway provides a central location for managing the routing of requests to microservices, which makes it easier to manage the communication between microservices and external clients.
2. Fault tolerance: Spring Cloud Gateway can help mitigate the effects of service failures by detecting when a service instance goes offline and routing requests to other available instances. This can help improve the fault tolerance of the overall system.
3. Security: Spring Cloud Gateway can provide security features such as authentication, authorization, and encryption to help protect against unauthorized access to microservices and sensitive data.
4. Scalability: Spring Cloud Gateway can scale horizontally by adding more instances of the service to handle increasing workloads. This can help ensure that the gateway can keep up with the demands of a growing system.
5. Customization: Spring Cloud Gateway provides a flexible architecture that allows us to customize and extend the functionality of the gateway to meet specific requirements.
6. Integration: Spring Cloud Gateway integrates seamlessly with other Spring Cloud components, such as Spring Cloud Netflix Eureka Server and Spring Cloud Config Server, which makes it easy to use as part of a larger microservices system.

Limitations ：

1. Dependency on a central server: Spring Cloud Gateway relies on a central server to manage the routing of requests to microservices. This can be a single point of failure, as the entire system can become unavailable if the server goes down.
2. Performance overhead: Spring Cloud Gateway generates additional traffic to route requests to microservices, which can introduce some performance overhead. This can affect the overall performance of the system, especially if we are running a high-traffic application.
3. Limited support for certain platforms: Spring Cloud Gateway may not be supported on all platforms and environments, and may require additional configuration to work correctly.

G.Resilience4j

Benefits :

1. Lightweight and easy to use: Resilience4j is a small library that is easy to integrate into your application. It provides a simple, functional API that is easy to understand and use.
2. Fault tolerance: Resilience4j provides a range of fault tolerance features, including circuit breaking, retry, and bulkhead, which can help improve the reliability and resilience of our application.
3. Asynchronous execution: Resilience4j supports asynchronous execution of code, which can help improve the performance and scalability of your application.
4. Functional programming: Resilience4j is designed for functional programming, which makes it easy to use in a functional style and integrate with functional libraries such as Java 8 streams.
5. Customization: Resilience4j provides a range of customization options, including the ability to define custom failure handlers and fallback functions, which allows you to fine-tune the behavior of the library to meet your specific requirements.
6. Integration: Resilience4j integrates seamlessly with other libraries, such as Spring, which makes it easy to use as part of a larger application.

Limitations ：

1. Limited documentation: Resilience4j has limited documentation, which can make it difficult for new users to get started with the library.
2. Lack of community support: Resilience4j is a relatively new library with a small community of contributors. This may make it more difficult to get help or find resources when working with the library.

▼ 6 Contributions

Provide a sub section for each team member that describes their contribution to the project. Descriptions should be short and to the point.

1.Zhilin Yang: config server, video, docker, report.

2.Yuan Yang: employeeMan service code, user service code

3.Yan Liu: salary, video,gateway,registry,unit testing.

Reference:

1.Spring Cloud Greenwich.RC1 available now

2.spring boot - Resilience4j vs Hystrix. What would be the best for fault tolerance? - Stack Overflow

3.Cloud Native Applications (spring.io)

4.Cloud Native Applications (spring.io)

5.Versions spring boot, spring cloud, ribbon not working - Stack Overflow

6.Microservices Architecture: Scalability, DevOps, Agile development (fiorano.com)