**Understanding the Gateway: A Single-Entry Point to Microservices**

In this section, we will explore how to establish a single-entry poin**t** into our microservices network. The solution lies in using a gateway or edgeserver, which acts as the central access point for all client requests. Additionally, we will delve into how these gateways/edge servers route requests to the appropriate microservices.

**Why Do We Need an Edge Server?**

When multiple clients directly interact with various microservices, several challenges arise:

* Clients must manage and remember the URLs of all the services.
* Common requirements like security, auditing, logging, and routing need to be enforced repeatedly across all services.

These challenges make the system complex, inconsistent, and harder to maintain. To address this, we introduce an edge server as the single gateway to the microservices network. This central gateway simplifies client interactions and ensures consistent implementation of cross-cutting concerns across the system.

**What Is an Edge Server?**

An edge server is an application positioned at the edge of the system, responsible for implementing key functionalities such as API gateway capabilities and handling cross-cutting concerns. Key benefits of using an edge server include:

1. **Routing Requests**: Routes incoming traffic to the respective microservices.
2. **Failure Prevention**: Prevents cascading failures in the system by enabling retries and timeouts for downstream service calls.
3. **Traffic Control**: Manages ingress traffic and enforces quota policies to regulate resource usage.
4. **Security Enforcement**: Implements authentication and authorization mechanisms, allowing secure token passing to downstream services for controlled access.

Edge servers play a crucial role in maintaining the reliability, security, and scalability of a microservices architecture.

In this section, we’ll use **Spring Cloud Gateway** to create an **edge server**, which acts as the **gatekeeper** for all inbound traffic to our microservices. With the service gateway in place, service clients no longer directly call the URLs of individual microservices; instead, they send all requests to the gateway, which routes them appropriately.

**Why Use Spring Cloud Gateway?**

Spring Cloud Gateway is designed to sit between a **requester** and the **requested resource**, where it intercepts, analyzes, and modifies every request. This provides the flexibility to route requests based on their context, enabling advanced functionalities such as:

* **Context-Aware Routing**: For example, routing based on API version headers.
* **Session Management**: Supporting sticky sessions by tracking each user's session.

**Creating an Edge Server Using Spring Cloud Gateway**

In this section, we’ll use Spring Cloud Gateway to create an edge server, which acts as the gatekeeper for all inbound traffic to our microservices. With the service gateway in place, service clients no longer directly call the URLs of individual microservices; instead, they send all requests to the gateway, which routes them appropriately.

**Why Use Spring Cloud Gateway?**

Spring Cloud Gateway is designed to sit between a requester and the requested resource, where it intercepts, analyzes, and modifies every request. This provides the flexibility to route requests based on their context, enabling advanced functionalities such as:

* Context-Aware Routing: For example, routing based on API version headers.
* Session Management: Supporting sticky sessions by tracking each user's session.

Spring Cloud Gateway is the preferred API gateway over Zuul because of its:

1. Reactive Programming Model: Built on Spring Reactor and Spring WebFlux, making it non-blocking in nature.
2. Performance: Superior performance compared to Zuul due to its reactive design.
3. Integration Features: Provides built-in support for circuit breakers, service discovery with Eureka, and more.

These features make Spring Cloud Gateway an excellent choice for modern, scalable microservices architectures.

**Adding Required Dependencies to the Gateway Server Project**

To set up a gateway/edge server using Spring Cloud Gateway, include the following dependencies in your project:

1. **Gateway:** Provides core API gateway functionality.
2. **Eureka Discovery Client:** Enables service discovery and registration.
3. **Config Client:** Manages centralized configuration for consistent environment settings.
4. **Spring Boot Actuator:** Facilitates monitoring and management of the gateway application.
5. **Spring Boot DevTools:** Enhances development productivity with features like live reload.

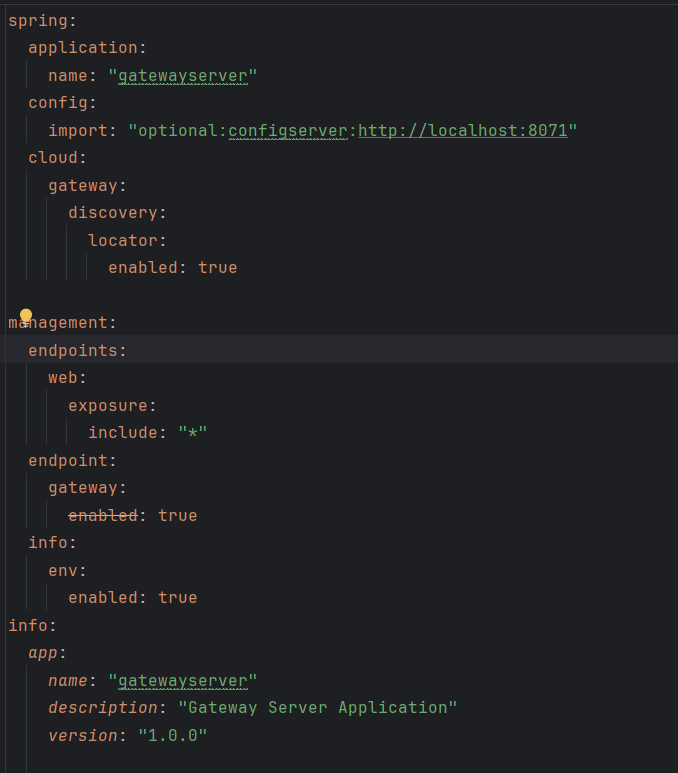
By leveraging Spring Cloud Gateway, you ensure a robust, high-performing, and scalable entry point to your microservices network.

This structure improves clarity and provides a smooth progression from problem identification to solution implementation.

After adding all the necessary dependencies, we need to update the application.yml file.

Most of the configuration will remain unchanged, but here’s a detailed explanation of the key properties:

* **spring.application.name**: Specifies the name of the application. This is used for identification in a microservices architecture.
* **spring.config.import**: Indicates that the application will use an external configuration server (such as a Spring Cloud Config Server) to load common configurations shared across multiple services.
* **spring.cloud.gateway.discovery.locator.enabled**: Enables the gateway server to connect to the Eureka discovery server, allowing it to dynamically locate and route requests to the appropriate microservices.
* Other management-related properties: These are used for monitoring and managing the application through Actuator endpoints.

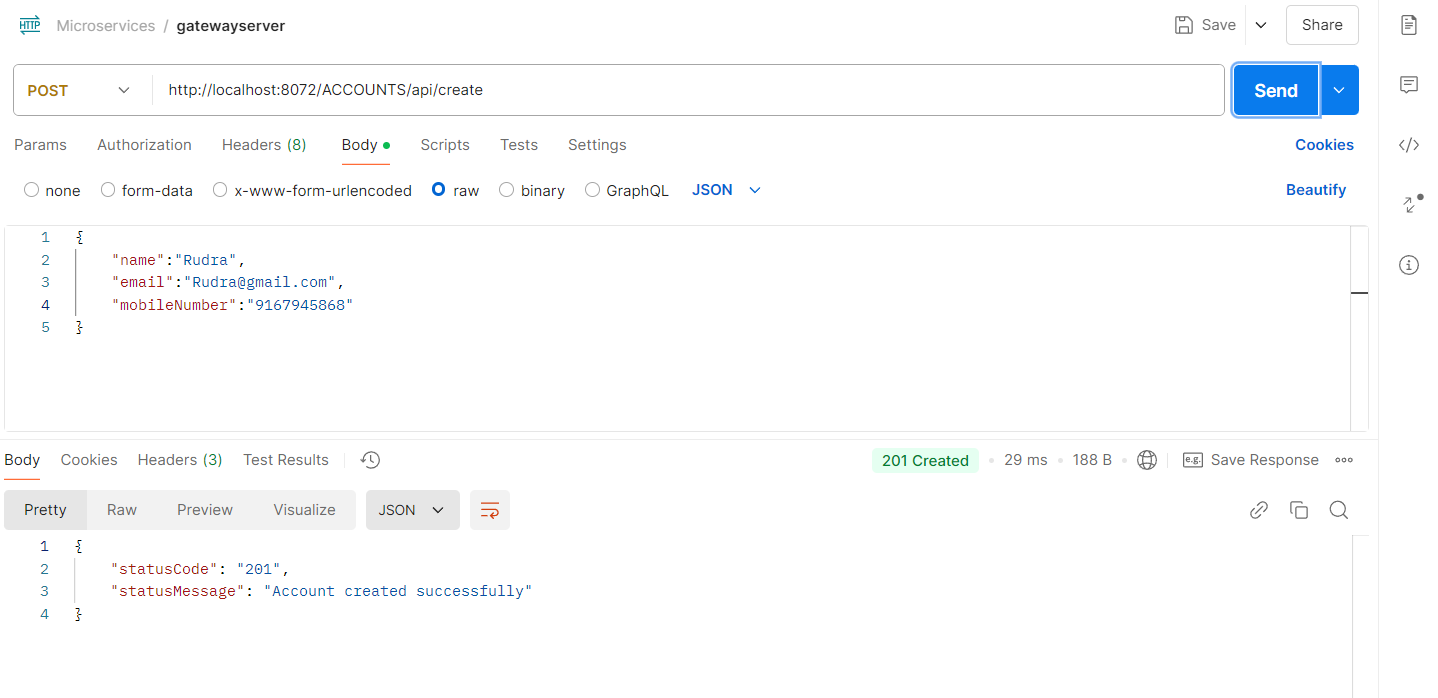


This configuration ensures proper integration with the configuration and discovery services, streamlining the communication and management of microservices.

This configuration enables seamless integration with configuration and discovery services, simplifying communication and management within the microservices ecosystem.

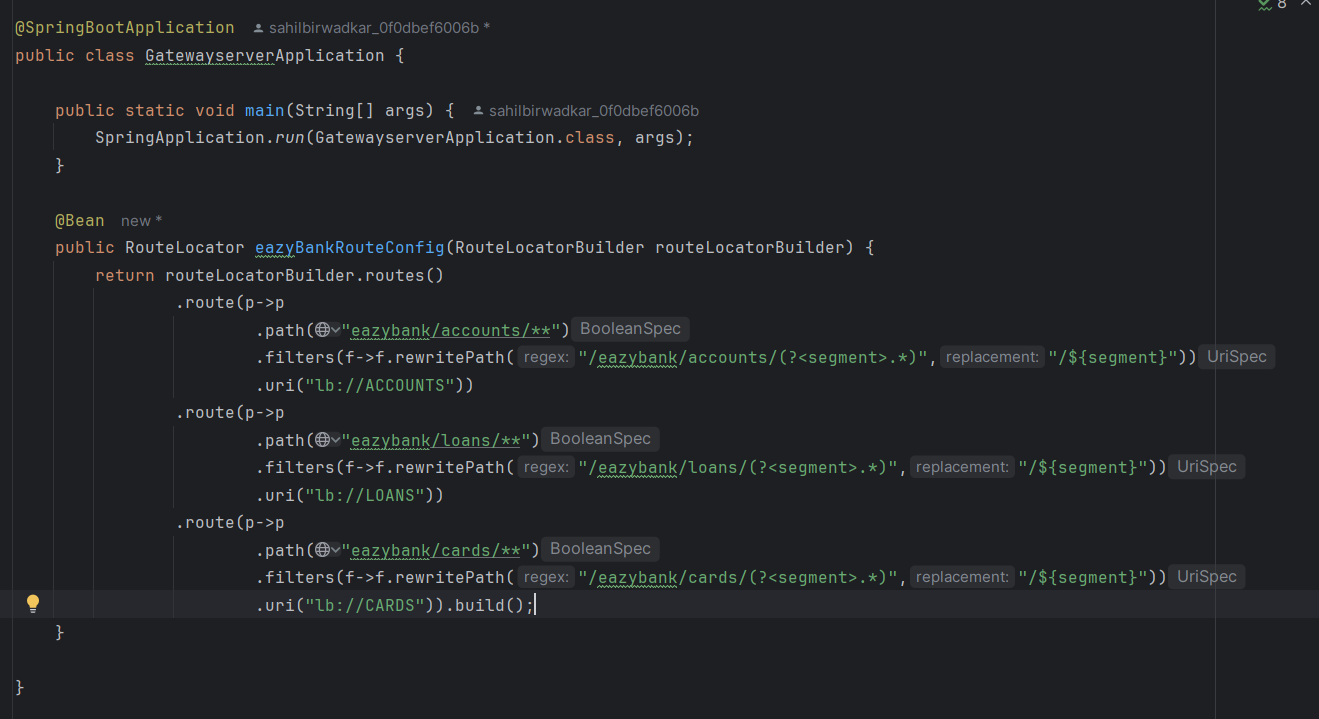
When making API calls, use the gateway server’s port number, followed by the specific microservice's application name (as registered in Eureka) and the desired API endpoint. For example:

“http://localhost:8072/ACCOUNTS/api/create"



Here, we specify the application name in uppercase because that’s how it’s stored in Eureka. If you want to use lowercase service IDs in API calls, set spring.cloud.gateway.discovery.locator.lower-case-service-id=true in the configuration.

If you prefer a URL structure that includes your organization name, such as http://localhost:8072/eazybank/ACCOUNTS/api/create, you can define custom routes in the gateway. To do this, create a method of type RouteLocator to configure these custom routes.



In this method, pass a RouteLocatorBuilder parameter and use routeLocatorBuilder.routes() to define a series of routes. Each .route(p -> p ... .build() block configures an individual route, specifying a path pattern, optional filters (such as path rewrites), and a destination URI.

In the route configuration, we use the following properties:

* **Path Matching**: This route matches any requests starting with eazybank/accounts/\*\*, allowing all paths under this prefix to be routed.
* **Path Rewrite**: The rewritePath filter applies a regular expression to capture everything after /eazybank/accounts/ and rewrites it as /${segment}. For instance, a request to eazybank/accounts/xyz will be transformed to /xyz before forwarding.
* **Destination URI**: The uri("lb://ACCOUNTS") specifies that the request should be routed to the ACCOUNTS microservice. The lb:// prefix indicates load balancing, typically through a service discovery platform like Eureka, to route to available instances of the ACCOUNTS service.

This configuration allows you to organize routes in a way that aligns with your organizational structure and naming conventions, while maintaining flexible routing and load balancing through Spring Cloud Gateway and Eureka.



In the setup shown above, both URL patterns are currently available: /eazybank/<application> and /<application>. If you want to restrict access to only the custom URL structure (e.g., /eazybank/<application>), you can set spring.cloud.gateway.discovery.locator.enabled=false in the configuration.

Setting this property to false disables the automatic route creation for service discovery, so only the routes explicitly defined in your custom configuration (like /eazybank/<application>) will be accessible. This ensures that requests must follow the specified custom path structure for routing through the gateway.