Spring Cloud Configuration Server and Client

Spring Cloud Config provides a central place to manage externalized configuration for applications in a distributed system. The **Config Server** is a Spring Boot application that provides configuration properties to clients, while a **Config Client** is any Spring Boot application that retrieves its configuration from the server. This setup allows you to manage configurations for all environments (dev, test, prod) in one place, typically a Git repository, and change them without rebuilding and redeploying your applications.

Concepts

- **Config Server**: A Spring Boot app with the `@EnableConfigServer` annotation that serves configuration from a Git repository.
- **Config Client**: A Spring Boot app that imports `spring-cloud-starter-config` and specifies the Config Server's URI in its `bootstrap.properties` or `bootstrap.yml`.
- **Configuration Repository**: A Git repository that stores the configuration files. The server pulls configs from here. Files are typically named `{application-name}-{profile}.properties` or `.yml`.

Code Implementation and Steps

1. Setup the Config Server

- Go to **Spring Initializr** (start.spring.io).
- Add dependencies: **Config Server**, **Spring Web**.
- In the main application class, add the `@EnableConfigServer` annotation.
- Create an `application.yml` file and configure it to point to your Git repository:

YAML

```
server:
  port: 8888
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/your-username/config-repo.git
```

- Create the Git repository specified in `uri` and add a configuration file, for example, `config-client.yml`:

YAML

```
message: "Hello from the Config Server!"
```

2. Setup the Config Client

- Go to **Spring Initializr**.
- Add dependencies: **Config Client**, **Spring Web**, **Actuator**.
- Create a `bootstrap.yml` file and configure it to connect to the Config Server:

YAML

```
spring:
  application:
    name: config-client
  cloud:
    config:
      uri: http://localhost:8888
```

- Create a simple REST controller to expose the configuration property:

Java

```
@RestController
@RefreshScope
public class MessageController {
    @Value("${message}")
    private String message;

    @GetMapping("/message")
    public String getMessage() {
        return this.message;
    }
}
```

The `@RefreshScope` annotation allows the bean to be reloaded at runtime when the configuration changes.

3. Running and Checking the Output

- Run the **Config Server** application. It will start on port 8888.
- Run the **Config Client** application. It will start on port 8080 (by default) and connect to the server to fetch its configuration.
- Access the client's endpoint in a web browser or using `curl`: `http://localhost:8080/message`.
- You should see the output: `Hello from the Config Server!`.

- To test dynamic updates, change the `message` in `config-client.yml` in your Git repo and push the changes. Then, on the client, trigger a refresh by sending a `POST` request to the actuator endpoint: `curl -X POST http://localhost:8080/actuator/refresh`. The next time you access `http://localhost:8080/message`, you'll see the new message.

API Gateway - Spring Cloud Gateway

In a microservices architecture, an **API Gateway** acts as a single entry point for all client requests. Spring Cloud Gateway is a reactive, non-blocking gateway built on Spring 5, Spring Boot 2, and Project Reactor. It provides features like routing, filtering, rate limiting, and circuit breaking. It's a modern, performant alternative to the older Netflix Zuul.

Concepts

- **Route**: Defines the path to a specific microservice. It consists of an ID, a destination URI, predicates, and filters.
- **Predicate**: A condition that determines if a request should be routed. Examples include `Path`, `Method`, and `Host`.
- **Filter**: Modifies the request or response. Filters can be applied globally or per route. They can be used for things like adding headers, rate limiting, or authenticating.

We use an **API Gateway** to provide a single, unified entry point for all client requests in a microservices architecture.Instead of clients having to know the addresses of multiple individual services, they just interact with one gateway.

Real-Time Usage

The real-time use cases for an API Gateway are critical for building scalable and manageable microservices systems.

- **Request Routing**: In a company like Netflix, when you access their app, the API Gateway routes your request to the correct microservice. For example, a request for a movie recommendation might go to the `recommendation-service`, while a request to view your account details goes to the `user-service`.
- **Load Balancing**: The gateway can distribute incoming requests across multiple instances of the same service, preventing any single instance from becoming overwhelmed. This is essential for high-traffic applications.
- **Authentication and Authorization**: The gateway can handle security at the edge. A request for a user's data can be authenticated and authorized by the gateway before it's ever forwarded to the `user-service`, which simplifies security logic within each microservice.
- **Rate Limiting**: To protect services from abuse or to manage usage tiers, the gateway can enforce rate limits. For example, it might allow a user to make only 100 requests per minute

to a specific service.

Centralized Logging and Monitoring: The gateway provides a central point to log all requests, which is vital for monitoring the health and performance of your entire microservices system.

Code Implementation and Steps

1. Setup the Gateway

- Go to **Spring Initializr**.
- Add dependencies: **Spring Cloud Gateway**, **Spring Webflux**.
- Create an `application.yml` file to configure the routes:

YAML

```
server:
  port: 8080
spring:
  cloud:
    gateway:
      routes:
        - id: example_route
          uri: http://httpbin.org:80
          predicates:
            - Path=/get
```

This configuration sets up a route with the ID `example_route` that forwards requests to `http://httpbin.org/get`when the request path is `/get`.

2. Running and Checking the Output

- Run the Spring Cloud Gateway application.
- Access the configured route in your browser: `http://localhost:8080/get`.
- You should see a JSON response from the `httpbin.org` service, indicating that the gateway successfully routed your request.

Distributed Log Tracing with Spring Cloud Sleuth and Zipkin

In a microservices environment, a single user request can pass through multiple services. Distributed tracing helps you track the entire request flow, troubleshoot performance issues, and perform root cause analysis. **Spring Cloud Sleuth**and **Zipkin** work together to achieve this.

Concepts

- **Spring Cloud Sleuth**: A library that adds unique IDs (**Trace ID** and **Span ID**) to log messages, automatically propagating them across service boundaries.
    - **Trace ID**: A unique ID for an entire request flow.
    - **Span ID**: A unique ID for a single unit of work within a trace (e.g., an API call to a specific service).
- **Zipkin**: A distributed tracing system that collects and visualizes the trace data generated by Sleuth. It shows you how a request traveled through your services and how long each step took.

Code Implementation and Steps

1. Setup a Zipkin Server

- The easiest way to run Zipkin is with a Docker container.
- Open a terminal and run the following command:
  Bash

```
docker run -d -p 9411:9411 openzipkin/zipkin
```

- The Zipkin server will be available at `http://localhost:9411`.

2. Setup a Spring Boot application with Sleuth

- Go to **Spring Initializr**.
- Add dependencies: **Spring Web**, **Zipkin Client**.
- No code changes are needed! Spring Boot and Spring Cloud Sleuth automatically configure everything. The `Zipkin Client` dependency is sufficient to enable log tracing and send data to the Zipkin server.
- Add a simple REST controller to your application:

Java

```
@RestController
public class HelloController {
    @GetMapping("/hello")
    public String hello() {
        return "Hello from the service!";
    }
}
```

3. Running and Checking the Output

- Ensure the Zipkin Docker container is running.

- Run your Spring Boot application.
- Access the endpoint: `http://localhost:8080/hello`.
- Go to the Zipkin UI at `http://localhost:9411`. You will see a new trace with the logs from your application, showing the `Trace ID` and `Span ID` that were generated and the duration of the request.

Introduction to Spring Security and Basic Auth

Spring Security is a powerful and customizable authentication and access control framework for Java applications. **Basic Authentication** is a simple, non-session-based method of securing HTTP endpoints. It sends the username and password, Base64 encoded, in an HTTP header with every request.

Concepts

- **Authentication**: Verifying a user's identity (e.g., checking if a username and password are correct).
- **Authorization**: Determining what an authenticated user is allowed to do (e.g., granting access to specific resources based on their role).
- **Basic Auth**: A simple authentication scheme where the client sends a header like `Authorization: Basic <base64_encoded_username_and_password>`.

Code Implementation and Steps

1. Setup a Spring Boot Application

- Go to **Spring Initializr**.
- Add dependencies: **Spring Security**, **Spring Web**.

2. Code Implementation

- By simply adding the `spring-boot-starter-security` dependency, Spring Security will automatically protect all endpoints with Basic Auth. It generates a default username (`user`) and a random password that is printed in the console on startup.
- To provide your own in-memory users, create a security configuration class:

Java

```
@Configuration
public class SecurityConfig {
    @Bean
    public InMemoryUserDetailsManager userDetailsService() {
        UserDetails user = User.withDefaultPasswordEncoder()
            .username("user")
```

```
            .password("password")
            .roles("USER")
            .build();
        return new InMemoryUserDetailsManager(user);
    }
}
```

This configuration sets up a single user named "user" with the password "password" and a "USER" role.

3. Running and Checking the Output

- Run the Spring Boot application.
- Access any endpoint in your browser, e.g., `http://localhost:8080`.
- A pop-up window will appear asking for a username and password. Enter "user" and "password".
- If the credentials are correct, you will be granted access. If you are using `curl`, you can provide the credentials in the command:
  Bash

  ```
  curl --user user:password http://localhost:8080
  ```