Spring Cloud Stream-Config Server

MOSTAFA RASTGAR

JUNE 2019

Agenda

- Spring Cloud Stream
- Cross-Cutting Concern Patterns
- Spring Cloud Config

Spring Cloud Stream

- Spring Cloud Stream is a framework built on top of Spring Boot and Spring Integration that helps in creating event-driven or message-driven microservices
- Main Concepts
  - EnableBinding — configures the application to bind the channels INPUT and OUTPUT defined within the interface Processor
  - Bindings — a collection of interfaces that identify the input and output channels declaratively
  - Binder — messaging-middleware implementation such as Kafka or RabbitMQ
  - MessageChannel — Defines methods for sending messages
  - ServiceActivator — Indicates that a method is capable of handling a message or message payload

Spring Cloud Stream

- Producer Configs:

```yaml
spring:
  cloud:
    stream:
      bindings:
        output:
          producer:
            partitionCount: 2
          destination: balance-check
          contentType: application/json;charset=UTF-8
      kafka:
        binder:
          brokers: 192.168.161.99:9092,192.168.161.99:9093,192.168.161.99:9094
          autoCreateTopics: true
```

- Consumer Configs:

```yaml
spring:
  cloud:
    stream:
      bindings:
        input:
          consumer:
            concurrency: 2
            partitioned: true
          group: balancechecker
          destination: balance-check
          contentType: application/json;charset=UTF-8
      kafka:
        binder:
          brokers: 192.168.161.99:9092,192.168.161.99:9093,192.168.161.99:9094
          autoCreateTopics: true
```

Cross-Cutting Concern Patterns

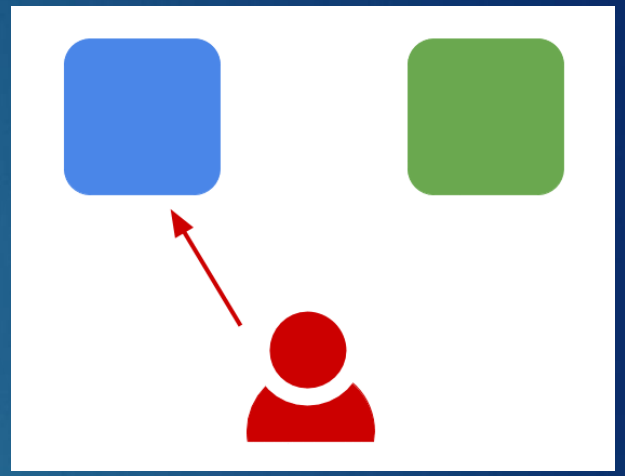
- Service Discovery Pattern
  - Problem:
    - Each service URL has to be remembered by the consumer and become tightly coupled
    - So how does the consumer or router know all the available service instances and locations
  - Solution:
    - A service registry needs to be created which will keep the metadata of each producer service
    - The consumer or router should query the registry and find out the location of the service
    - There are two types of service discovery: client-side and server-side
      - An example of client-side discovery is Netflix Eureka
      - an example of server-side discovery is AWS ALB

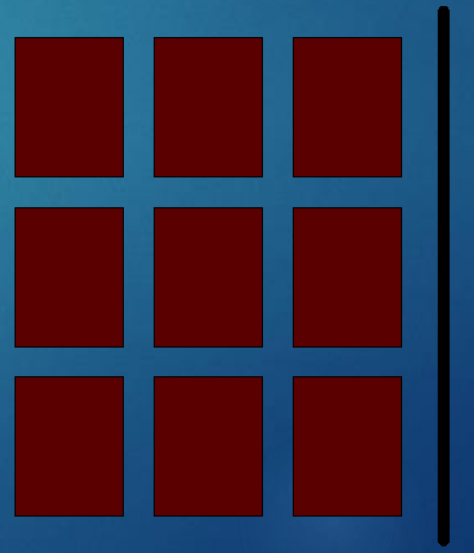
Cross-Cutting Concern Patterns

- Circuit Breaker Pattern
  - Problem:
    - What will happen if downstream service is down?
      - First, the request will keep going to the down service, exhausting network resources and slowing performance
      - Second, How do we avoid cascading service failures and handle failures gracefully
  - Solution:
    - The consumer should invoke a remote service via a proxy that behaves in a similar fashion to an electrical circuit breaker
    - When the number of consecutive failures crosses a threshold, the circuit breaker trips
    - After the timeout expires the circuit breaker allows a limited number of test requests to pass through
    - If those requests succeed, the circuit breaker resumes normal operation
- Netflix Hystrix is a good implementation of the circuit breaker pattern. It also helps you to define a fallback mechanism which can be used when the circuit breaker trips

Cross-Cutting Concern Patterns

- Deployment Pattern
  - Problem
    - With microservice architecture, one application can have many microservices
    - If we stop all the services then deploy an enhanced version, the downtime will be huge and can impact the business
    - Also, the rollback will be a nightmare
  - Solution:
    - Blue-Green
    - Canary





Cross-Cutting Concern Patterns

- External Configuration
  - Problem:
    - A service typically calls other services and databases as well
    - For each environment like dev, QA, prod, the endpoint URL or some configuration properties might be different
    - A change in any of those properties might require a re-build and re-deploy of the service
    - How do we avoid code modification for configuration changes?
  - Solution:
    - Externalize all the configuration, including endpoint URLs and credentials
    - The application should load them either at startup or on the fly
    - Spring Cloud config server provides the option to externalize the properties to GitHub and load them as environment properties
    - These can be accessed by the application on startup or can be refreshed without a server restart

Spring Cloud Config

- The applications settings can be moved to an external place so that applications will be easily configurable and can even change their settings
- To do this, a configuration server should be created
  - Just add @EnableConfigServer on the class level
- Clients should read the configuration of that server
  - Just add spring.cloud.config.uri=[the server url] in the bootstrap file

Spring Cloud Config

- Profiles and Auto of the Box Implementations
  - **File System Backend:** There's a **native** profile available where the "Config Server" searches for the properties/YAML files from the local classpath or file system
    - You can point to any location using spring.cloud.config.server.native.searchLocations
  - **Git Backend:** There's also a git profile where you can point to an external git repository that contains all the configurations files for your microservices
    - You can point to your git location using spring.cloud.config.server.git.uri
  - **Vault Backend:** There's also a vault profile that enables integration with Vault to securely store the application properties.
    - You should set spring.cloud.config.server.vault.host, spring.cloud.config.server.vault.port

Spring Cloud Config

- You can also secure properties without vault using encryption
- We can encrypt the values in property file using /encrypt & /decrypt URL available in config server
  - curl localhost:8888/encrypt -d mysecret
  - curl localhost:8888/decrypt -d 682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda
  - The methods of the above two actions should be post
- Now the encrypted value can be used in property files and should start with {cipher} such as: password: '{cipher}FKSAJDFGYOS8F7GLHAKERGFHLSAJ'
- they are decrypted before sending to clients over HTTP
- If a value cannot be decrypted, it is removed from the property source and an additional property is added with the same key but prefixed with invalid and a value that means "not applicable" (usually <n/a>)
- Key Management
  - symmetric key:
    - To configure a symmetric key, you need to set encrypt.key to a secret String
    - or use the ENCRYPT_KEY environment variable to keep it out of plain-text configuration files
  - asymmetric key (RSA key pair):
    - use a keystore
    - encrypt.keyStore.location: Contains a Resource location
    - encrypt.keyStore.password: encrypt.keyStore.password
    - encrypt.keyStore.alias: encrypt.keyStore.alias: Identifies which key in the store to use
    - encrypt.keyStore.type: The type of KeyStore to create. Defaults to jks