INSTITUTO FEDERAL
Paraíba
Campus Cajazeiras

PROGRAMAÇÃO P/ WEB 2

3

10. DESENVOLVENDO SERVIÇOS PRONTOS PARA PRODUÇÃO

PROF. DIEGO PESSOA

DIEGO.PESSOA@IFPB.EDU.BR

@DIEGOEP

CST em Análise e Desenvolvimento de Sistemas

ROTEIRO

- Desenvolvendo serviços seguros
- Aplicando o padrão de externalização da configuração
- Aplicando os padrões de observação:
  - Health check API
  - Log aggregation
  - Distributed tracing
  - Exception tracking
  - Application metrics
  - Auditing Logging
- Simplificando o desenvolvimento de serviços aplicando o padrão chassis

INTRODUÇÃO

- Para um serviço ser implantado em produção, é preciso garantir que ele satisfaz três importantes atributos de qualidade:
- Segurança
- Configuração
- Observação

INTRODUÇÃO

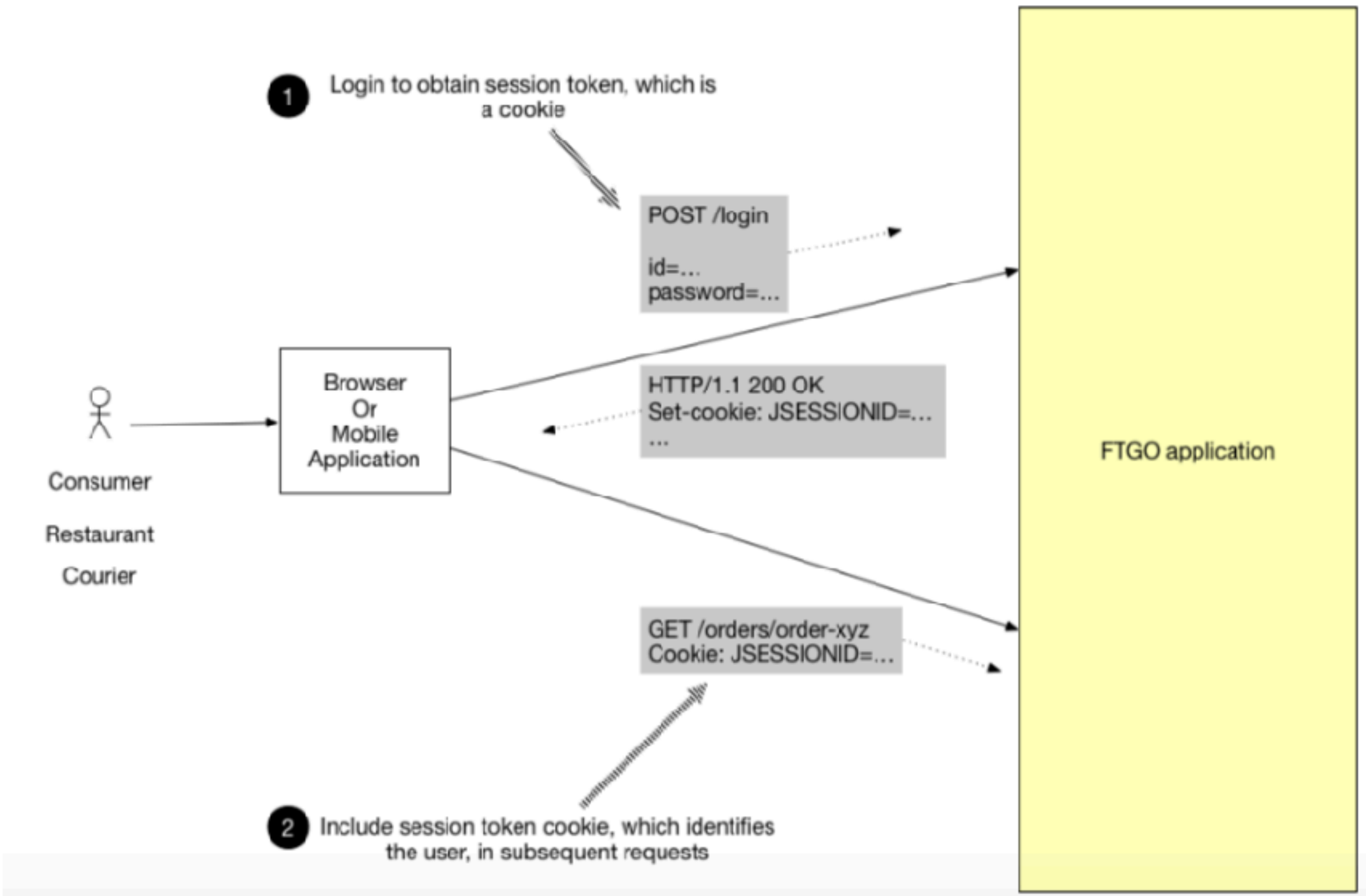
- É essencial desenvolver uma aplicação segura. A arquitetura de microsserviços força a implementação de segurança no nível de aplicação (identificar o usuário de um serviço para outro)
- Um serviço tipicamente usa um ou mais serviços externos (MB, DB). As credenciais dependem do ambiente em que roda, mas mantê-las num arquivo junto ao artefato não é viável
- Demanda pela implementação dos padrões de observação para entender o comportamento da aplicação e resolver problemas

1. DESENVOLVENDO SERVIÇOS SEGUROS

ASPECTOS DA SEGURANÇA EM APLICAÇÕES WEB

- ***Autenticação -*** verificar identidade de uma aplicação ou humano (aka. *principal*) que tenta acessar a aplicação;
- ***Autorização -*** verificar que o *principal* possui permissão de realizar a operação solicitada.
- ***Auditoria -*** rastrear as operações que um principal realiza
- ***Comunicação inter-processo segura -*** idealmente, toda entrada/saída deve ocorrer em TLS (Transport Layer Security). Comunicação inter-serviços também deve ser autenticada.

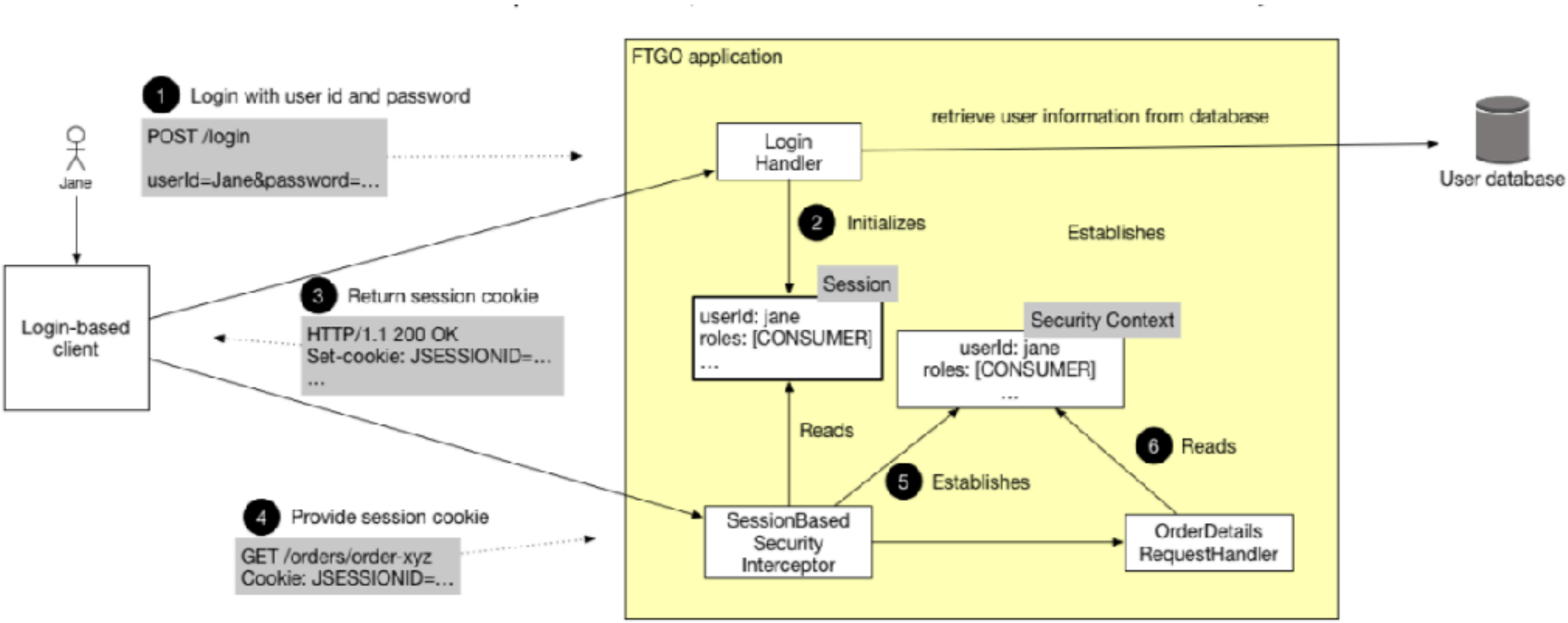
SEGURANÇA EM APLICAÇÕES MONOLÍTICAS



USO DE UM FRAMEWORK DE SEGURANÇA

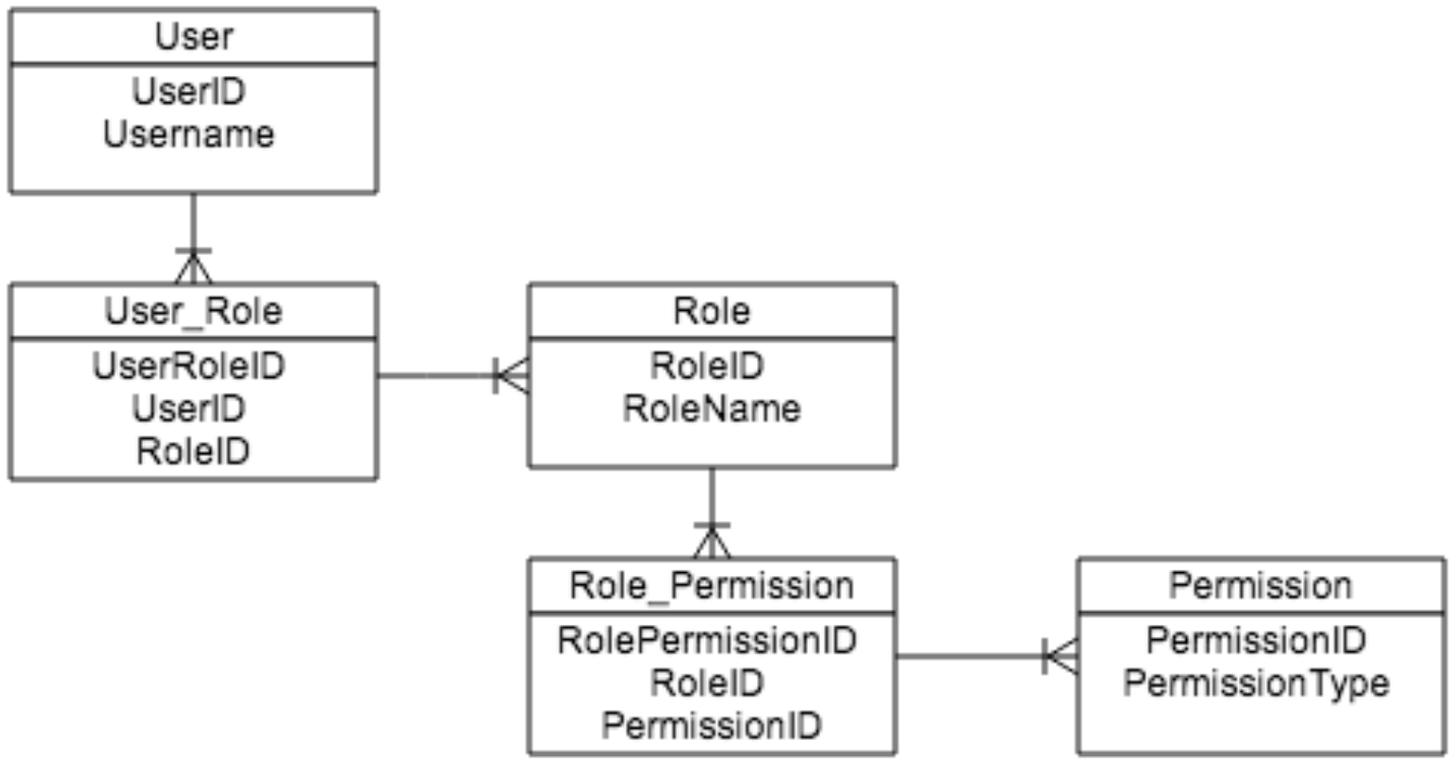
- Implementar autenticação e autorização não é trivial. Para facilitar, pode-se usar um framework de segurança.
- A escolha do framework depende das demais tecnologias utilizadas no projeto.
- Alguns frameworks populares:
  - Spring Security
  - Apache Shiro
  - Passport - framework de segurança para aplicações NodeJS

ARQUITETURA DE SEGURANÇA COM SESSÕES



MODELO DE AUTENTICAÇÃO RBAC

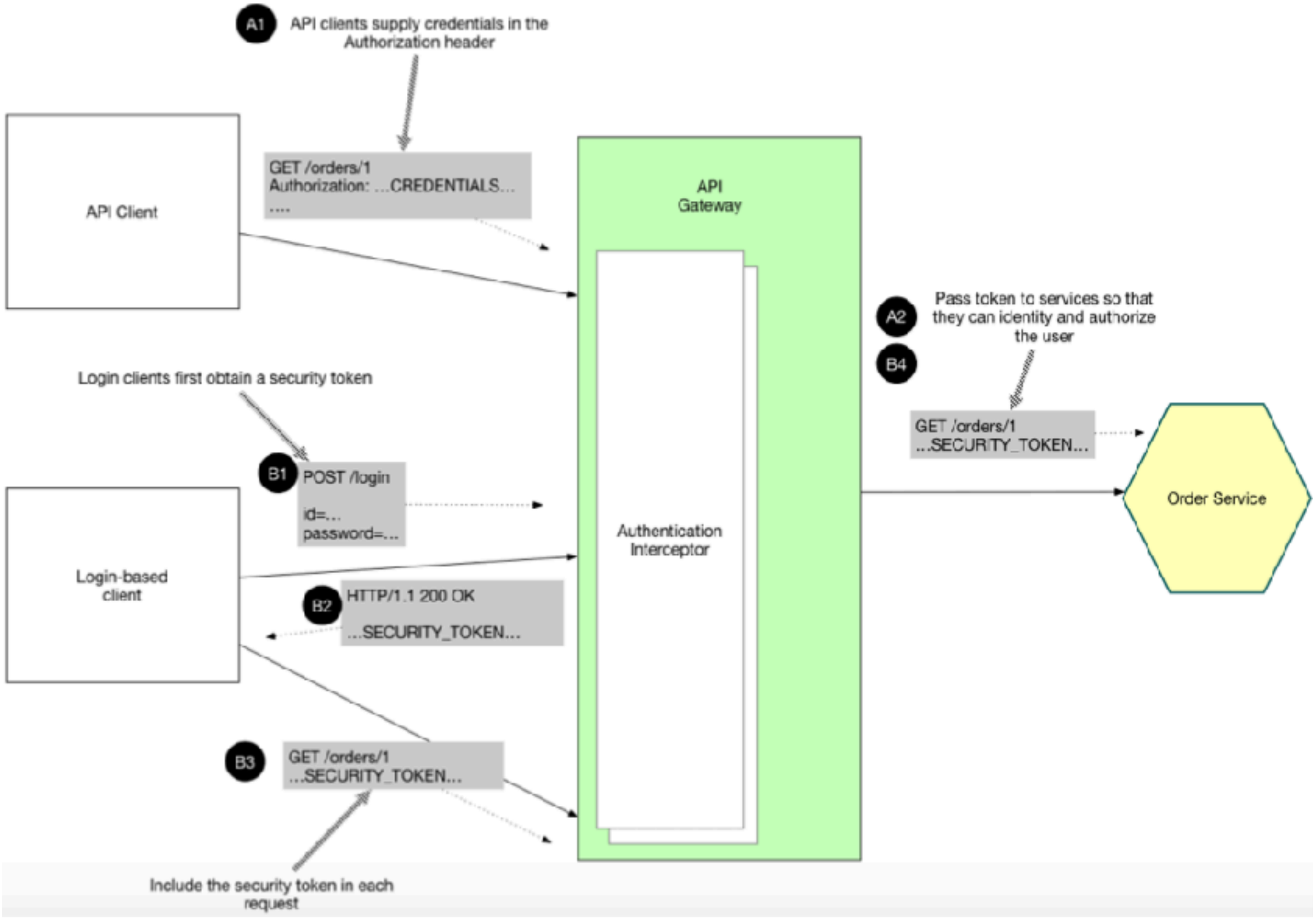
- Role-based access control



TRATANDO AUTENTICAÇÃO NO API GATEWAY

- ***Padrão: Access Token***
- O API gateway passa um token contendo informações sobre o usuário, tais como identidade, papéis e serviços que ele invoca.
- Exemplo: JSON Web Token (https://jwt.io)

IMPLEMENTANDO SEGURANÇA COM TOKENS NUMA ARQUITETURA DE MICROSSERVIÇOS



TRATANDO AUTORIZAÇÃO EM MICROSSERVIÇOS

- Pode-se restringir o acesso a rotas (ex.: GET /order/{orderId} de acordo com os papéis do usuário
- Essa restrição pode ser feita no API Gateway ou nos serviços
- Mas como isso é possível se os serviços não possuem acesso à base de usuários?

USANDO TOKENS JWT

- Através do Token JWT, o API Gateway pode passar informações do usuário para os serviços
- Um JWT tem um payload em JSON que contém informações sobre o usuário, tais como identificação dos papéis e outros metadados como data da expiração
- Ele é assinado com um segredo que só é conhecido pelo criador do JWT, como o API gateway e os receptores (serviços)
- Uma desvantagem do JWT é que os tokens são irrevogáveis. Logo, a recomendação é criar tokens com curto tempo de duração. No entanto, isso faz com que a aplicação fique sempre gerando novos tokens.

USANDO OAUTH 2.0 EM ARQUITETURAS DE MICROSSERVIÇOS

- Imagine que você pense em implementar um User Service responsável por gerenciar os dados do usuário (como credenciais e papéis).

- Isso não é necessário, basta usar algum framework que implemente o protocolo OAuth 2.0

OAUTH 2.0

- OAuth é um protocolo que foi originalmente criado para acessar informações de serviços públicos na nuvem, como Github ou Google, sem ter de compartilhar a senha de acesso.

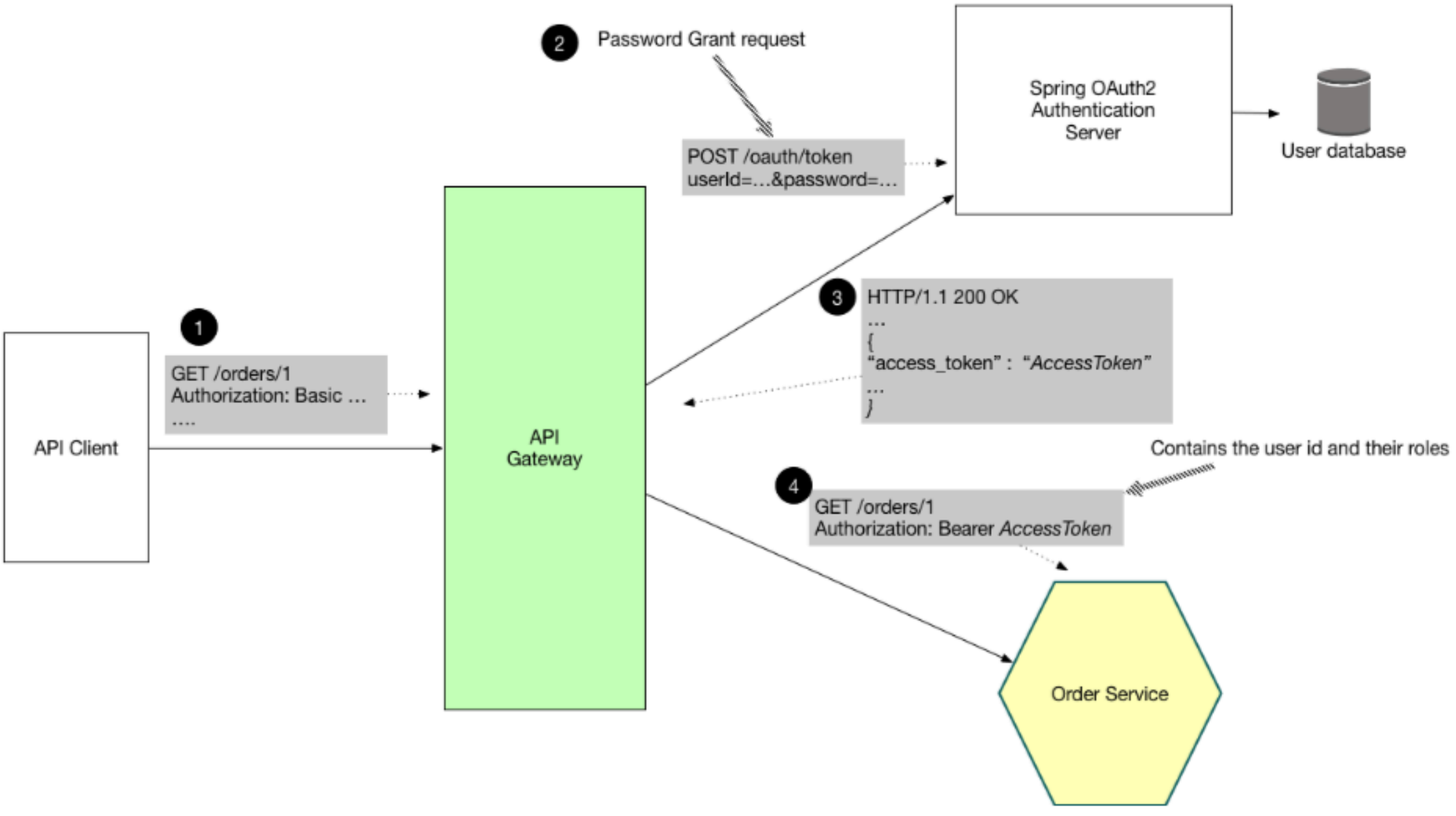
OAuth é um protocolo que foi originalmente criado para acessar informações de serviços públicos na nuvem, como Github ou Google, sem ter de compartilhar a senha de acesso.

- Para mais informações sobre OAuth 2.0, veja o livro “OAuth 2 In Action” e o capítulo 7 do livro “Spring Microservices in Action”.

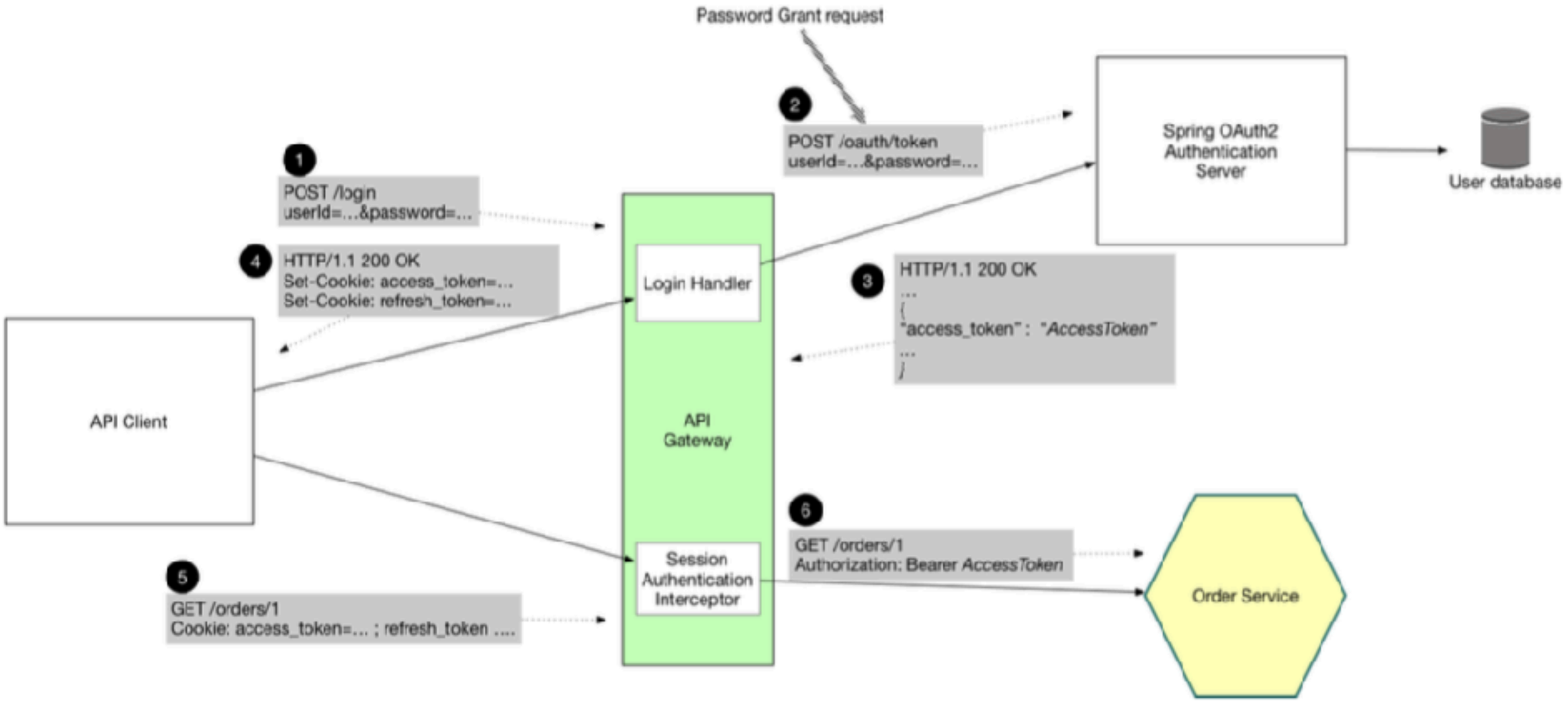
OAUTH 2.0 – CONCEITOS-CHAVE

- **Authorization Server -** API para autenticar usuários e obter um token de acesso. Spring OAuth é um bom exemplo de framework para construir um servidor de autorização OAuth 2.0.
- **Access Token -** um token que garante acesso ao Resource Server. O formato do token é independente. No entanto, Spring OAuth usa JWT.
- **Refresh Token -** um token de longa vida, revogável, que um *Client* usa para obter um novo *Access Token*
- **Resource Server -** um serviço que usa um Access Token para autorizar acesso. Numa arquitetura de microsserviços, os serviços são resource servers.
- **Client -** um cliente que deseja acessar um Resource Server. Numa arquitetura de microsserviço, o API Gateway é o cliente OAuth 2.0.

API GATEWAY COM OAUTH

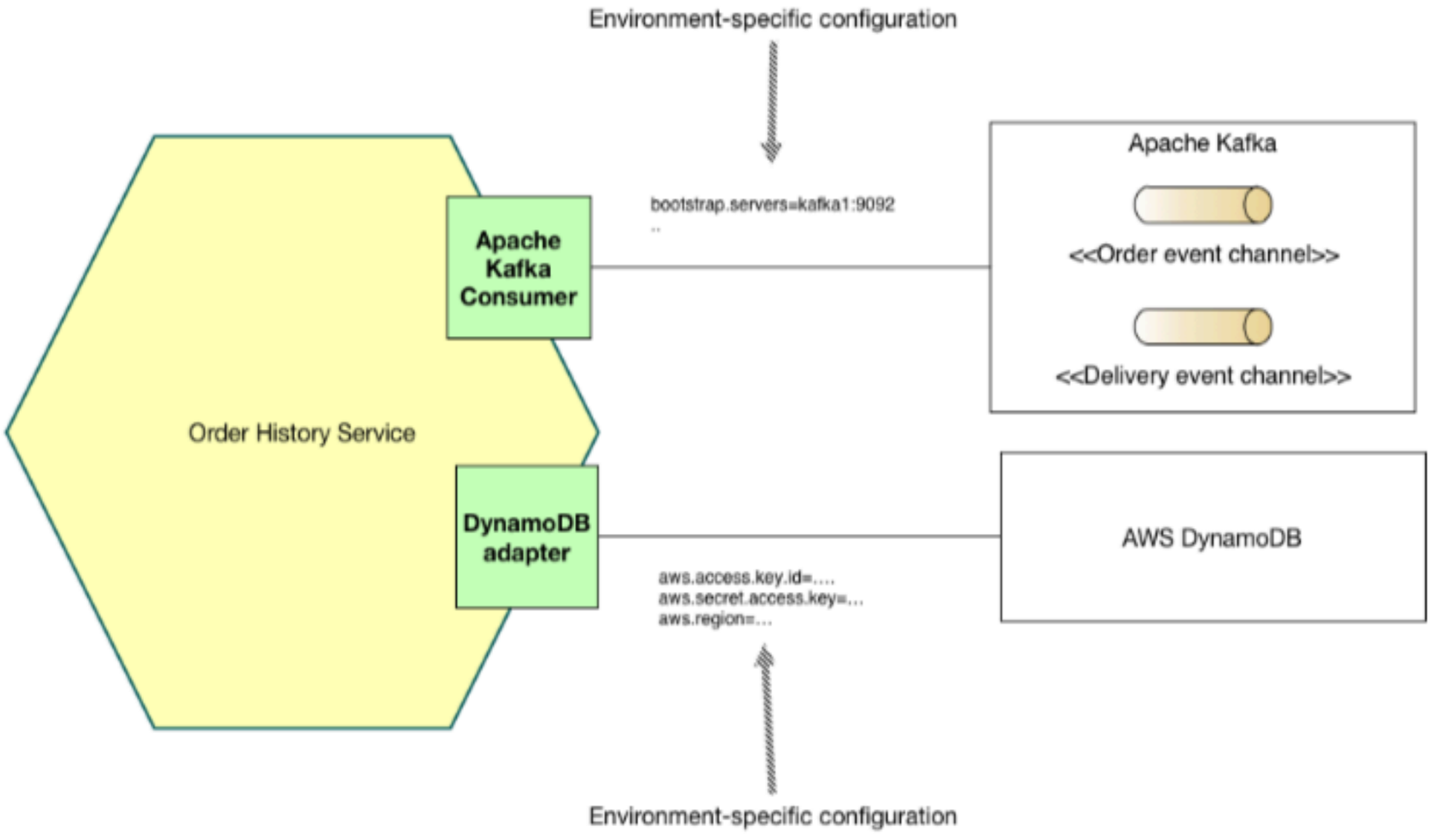


API GATEWAY COM OAUTH



2. DEFININDO SERVIÇOS CONFIGURÁVEIS

NECESSIDADE DE CONFIGURAÇÃO ESPECÍFICA DE AMBIENTE



NECESSIDADE DE CONFIGURAÇÃO ESPECÍFICA DE AMBIENTE

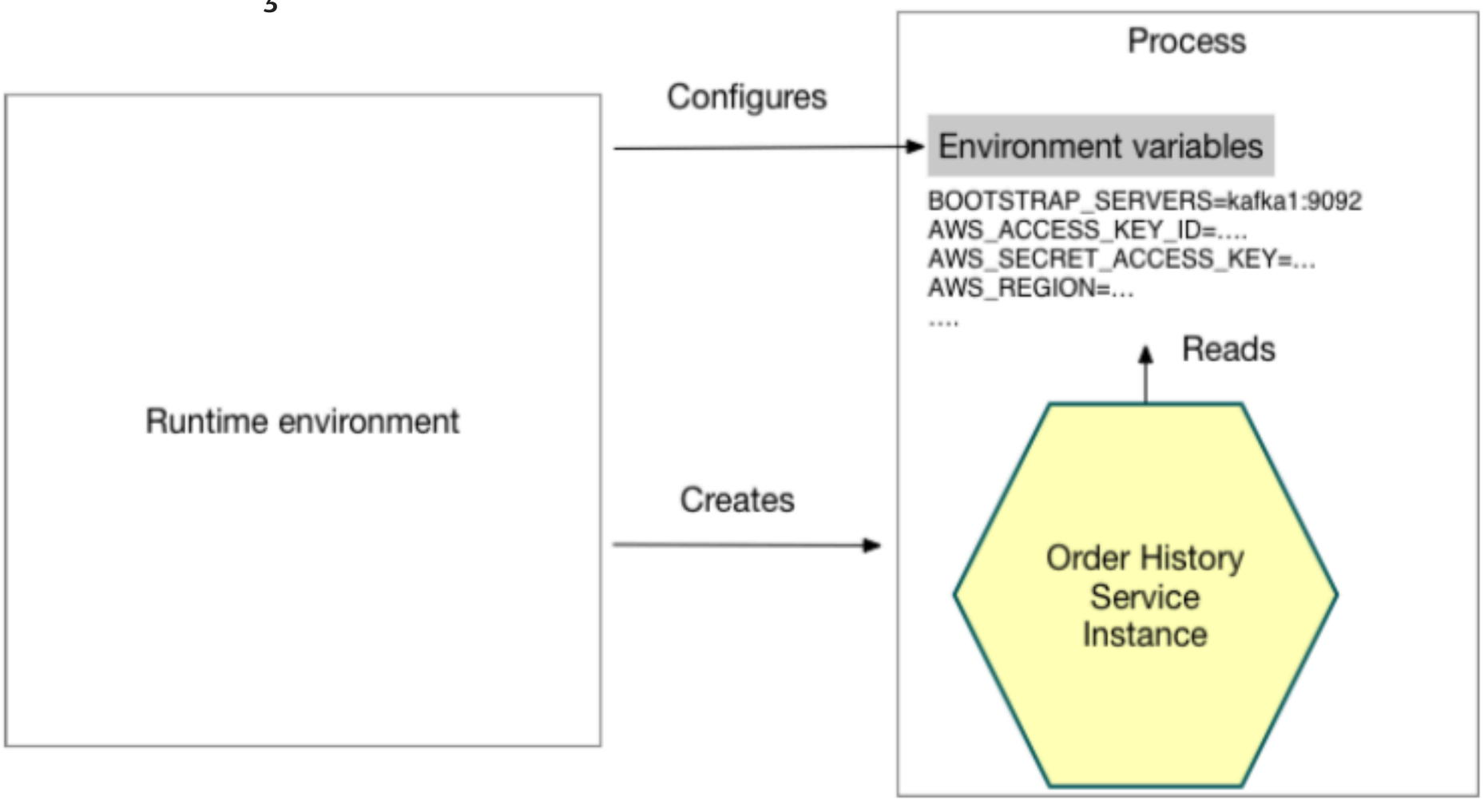
- Os valores dessas propriedades de configuração dependem do ambiente em que o serviço está rodando
- Não faria sentido configurar diversos arquivos de propriedades e usar para configurar o serviço para os possíveis ambientes em que ele possa ser implantado.
- Solução: prover as propriedades de configuração adequadas ao serviço em tempo de execução usando o padrão "Externalized Configuration"

PADRÃO EXTERNALIZED CONFIGURATION

- Provê valores de propriedades de configuração, tais como as credenciais de banco ou localização de rede, para um serviço, em tempo de execução. (detalhes)
- Há duas abordagens principais:
- ***Push model -*** a infraestrutura em tempo de execução passa as propriedades de configuração para as instâncias do serviço que está usando.
- ***Pull model -*** a instância do serviço lê as propriedades de configuração de um servidor de configuração

USANDO A CONFIGURAÇÃO EXTERNA PUSH-BASED

- Baseia-se na colaboração do ambiente de implantação e o serviço. O ambiente provê as propriedades de configuração que serão usadas quando a instância do serviço for criada



USANDO A CONFIGURAÇÃO EXTERNA PUSH-BASED

- O framework Spring Boot lê propriedades de várias fontes:
  - Argumentos de linha de comando (ex.: -Dpropriedade=valor)
  - SPRING_APPLICATION_JSON, que é uma variável de ambiente ou propriedade de sistema da JVM contendo um JSON
  - Propriedades da JVM
  - Variáveis de ambiente do sistema operacional
  - Arquivo de configuração no diretório local

USANDO A CONFIGURAÇÃO EXTERNA PUSH-BASED

- Spring Boot disponibiliza essas propriedades no ApplicationContext. Um serviço pode, por exemplo, obter o valor de uma propriedade usando a anotação @Value:

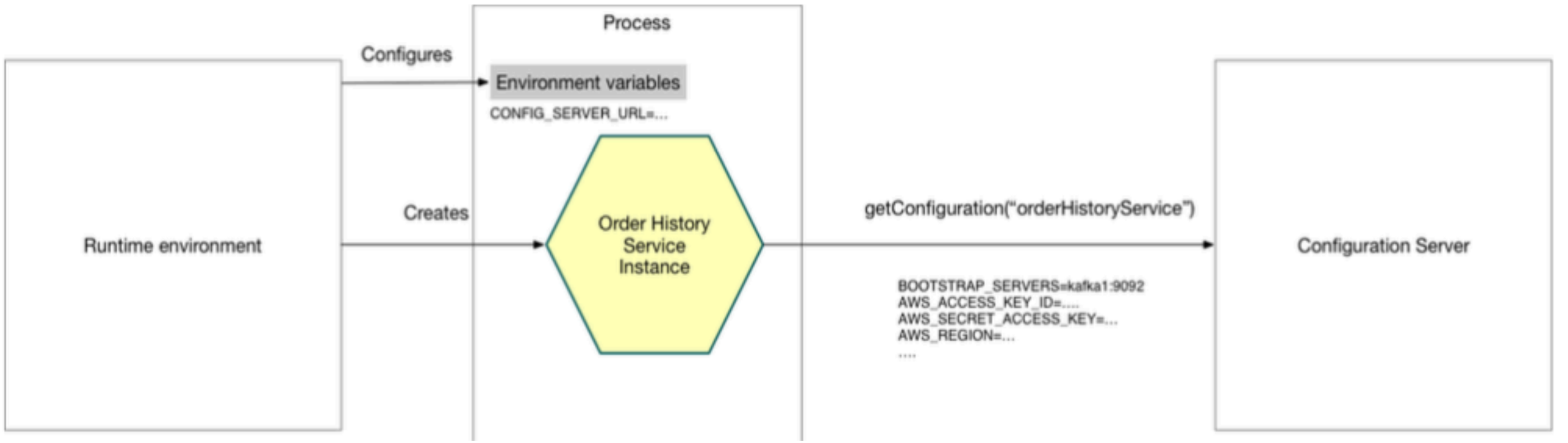
```
public class OrderHistoryDynamoDBConfiguration {

    @Value("${aws.region}")
    private String awsRegion;
```

USANDO A CONFIGURAÇÃO EXTERNA PUSH-BASED

- O push model é um mecanismo eficiente para configuração de serviços
- Uma limitação é que reconfigurar um serviço que já está rodando pode ser complicado
- A infraestutura de implantação pode não permitir mudar a configuração externa sem reiniciar o serviço
- Neste caso, não seria possível mudar variáveis de ambiente de um processo rodando
- Outra limitação é que há o risco dos valores da propriedade de configuração serem disseminadas para a definição de outros serviços.
- Por conta disso, pode-se considerar usar o modelo pull-based

USANDO A CONFIGURAÇÃO EXTERNA PULL-BASED

- A instância de serviço lê as configurações de um servidor



USANDO A CONFIGURAÇÃO EXTERNA PULL-BASED

- Formas de implementar:
  - Controle de versão, como o Git
  - Bancos de dados SQL ou NoSQL
  - Servidores especializados em configuração, como o Spring Cloud Config, Hashicorp Vault (que armazena dados sensíveis como credenciais) e o AWS Parameter Store.

BENEFÍCIOS DE USAR UM SERVIDOR DE CONFIGURAÇÃO

- ***Configuração centralizada* -** todas as propriedades são guardadas em um único lugar, o que facilita a manutenção. Além disso, elimina a duplicação de propriedades de configuração, podendo-se definir uma base global e a sobrescrita por cada serviço.
- ***Decodificação transparente de dados sensíveis* -** o serviço poderá automaticamente decodificar propriedades antes de retorná-las ao serviço
- ***Configuração dinâmica* -** um serviço poderá detectar atualizações nas propriedades através, por exemplo, de polling, e se reconfigurar automaticamente.

BENEFÍCIOS DE USAR UM SERVIDOR DE CONFIGURAÇÃO

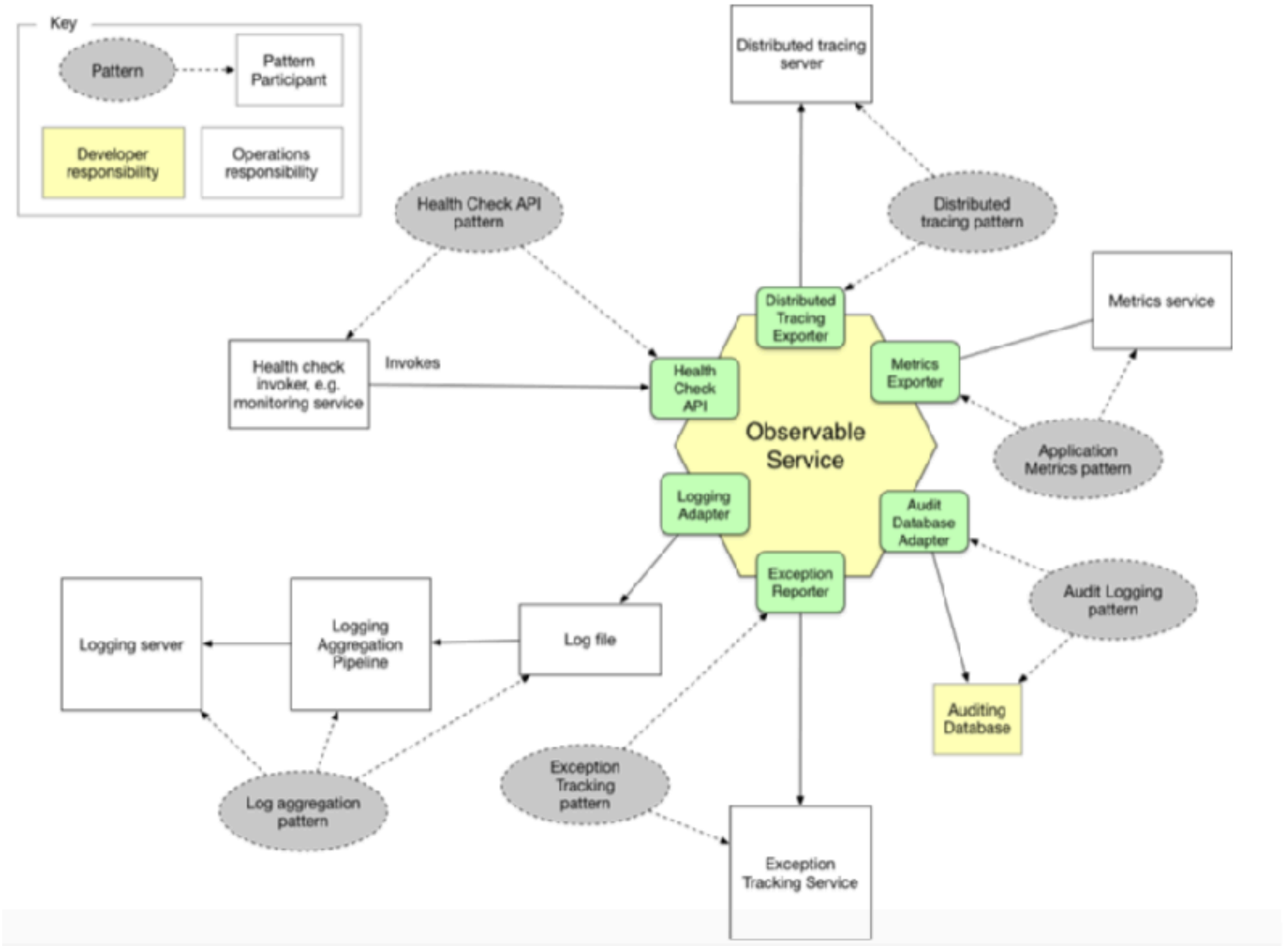
- Desvantagem: servidor de configuração é um componente adicional que deve ser configurado e mantido
- No entanto, o Spring Cloud Config facilita a configuração e execução de um serviço de configuração
- Uma possibilidade é rodar o servidor de configuração junto com o serviço de Service Descovery (ex.: Eureka)

3. DEFININDO SERVIÇOS "OBSERVÁVEIS"

MONITORAMENTO DE SERVIÇOS

- Há muitos aspectos que vão além do escopo do desenvolvedor, como monitorar a disponibilidade do hardware e sua utilização, essas são responsabilidades do Ops (operações)
- Há no entanto alguns padrões que um desenvolvedor de serviço deve implementar para torná-lo mais fácil de gerenciar e resolver problemas.
- Esses padrões expõem o comportamento e a saúde (health) de uma instância de serviço
- Isso permite que um sistema de monitoramento possa rastrear e visualizar o estado de um serviço e gerar alertas quando há um problema

MONITORAMENTO DE SERVIÇOS



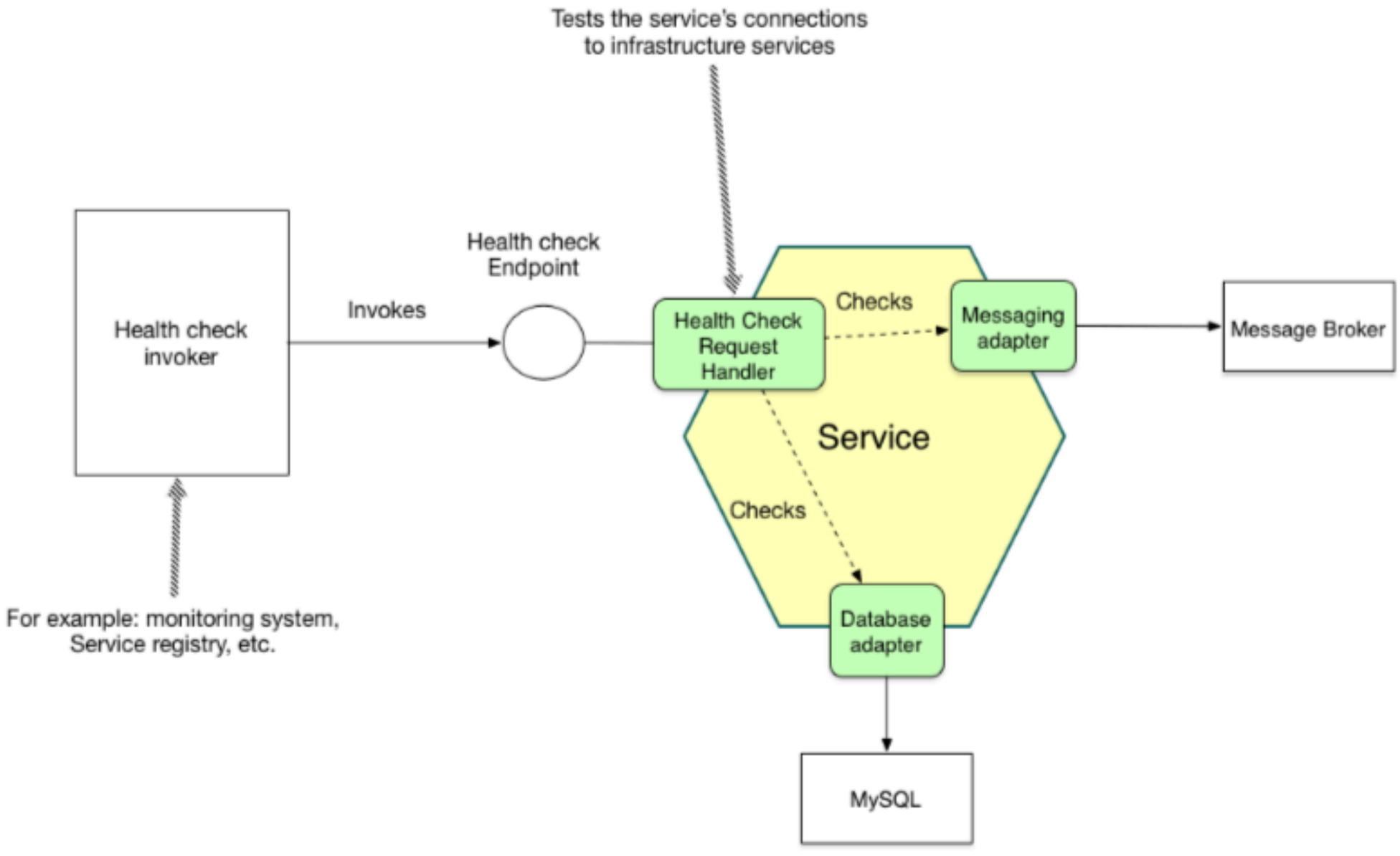
MONITORAMENTO DE SERVIÇOS

- É possível usar os seguintes padrões para monitoramento de serviços:
  - ***Health check API -*** expor um endpoint que retorna a saúde do serviço
  - ***Log aggregation -*** serviço de escrita e leitura de logs em um servidor central, que provê busca e notificações
  - ***Distributed tracing -*** atribui a requisições externas um ID único e rastreia requisições como também o seu fluxo entre serviços
  - ***Exception tracking -*** reporta exceções para um serviço de rastreamento de exceções, que interpreta as exceções, alerta desenvolvedores e acompanha a resolução de cada exceção
  - ***Application metrics -*** serviço que mantém métricas, como contadores e indicadores, expondo-as a um servidor de métricas
  - ***Audit logging -*** guarda log de ações do usuário

HEALTH CHECK API

- Pode acontecer de um serviço que está rodando, não responda requisições adequadamente. (ex.: ainda está subindo)
- Outra situação é o sistema falhar sem a aplicação parar de rodar (ex.: falha na conexão ao banco)
- ***Padrão Health Check API:*** *Um serviço expõe um endpoint que informa dados sobre "saúde" do serviço, como por exemplo /health*
- A biblioteca Spring Boot actuator, por exemplo, implementa um endpoint GET /health, que retorna 200 se o serviço está "saudável" e 503 caso não.

IMPLEMENTAÇÃO DE HEALTH CHECK ENDPOINT



IMPLEMENTAÇÃO DE HEALTH CHECK ENDPOINT

- O Spring Boot Actuator implemente um conjunto de checagens baseado na infraestrutura de serviços utilizados
- Se, por exemplo, o serviço usa um datasource JDBC, logo o Spring Boot Actuator configura um health check que executa uma consulta de teste
- Da mesma forma, se usa o Message Broker Kafka, ele automaticamente verifica se o serviço do kafka está on-line
- Também é possível personalizar esse comportamento implementando health checks adicionais para o serviço
- É possível definir uma classe que implementa a interface HealthInidicator, que define um método health() que será chamada pela implementação do endpoint /health.

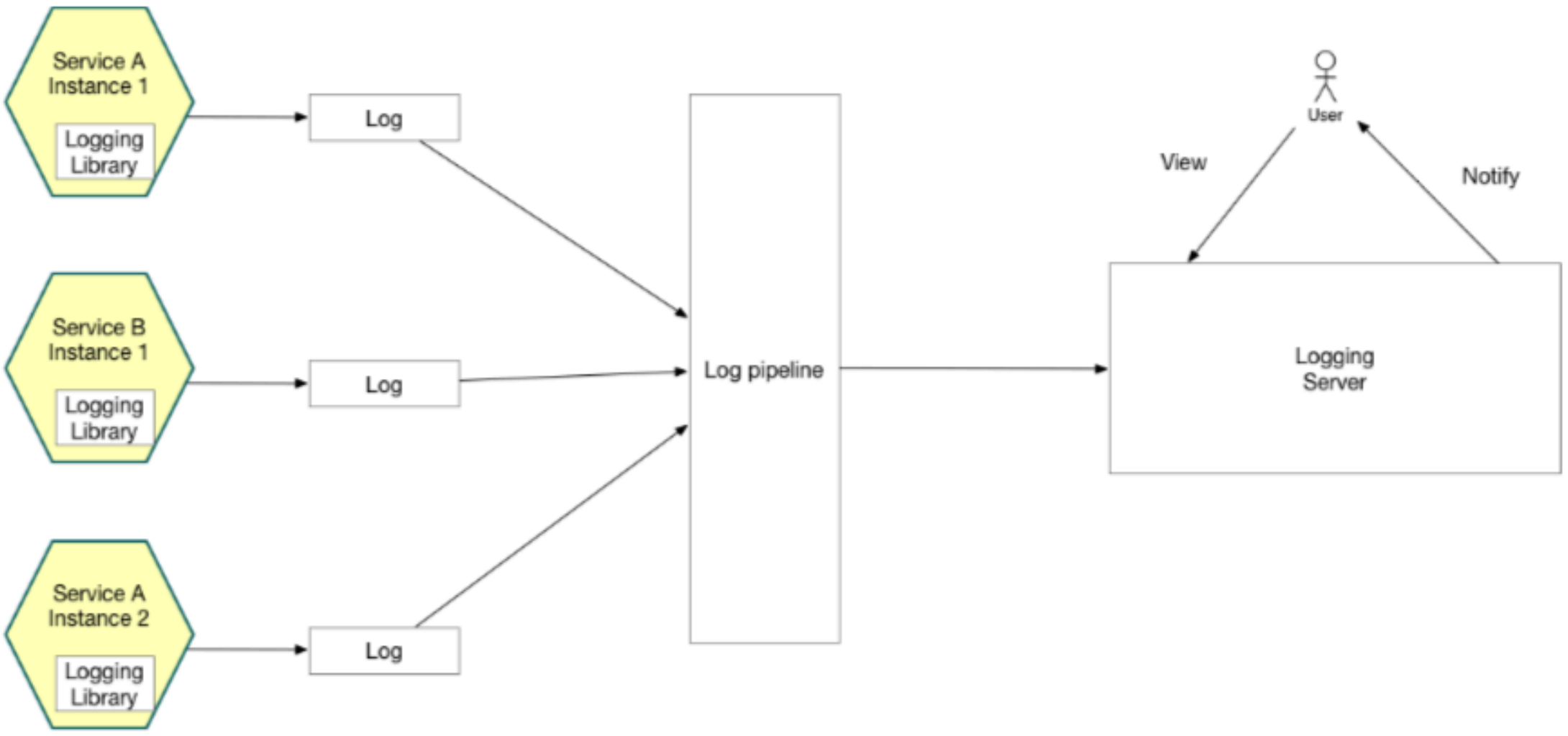
INVOCANDO UM HEALTH CHECK ENDPOINT

- De nada adianta ter um health check endpoint se ninguém o chamar
- Quando a aplicação é implantada, a infraestrutura deve ser configurada para chamar o endpoint
- Isso depende dos detalhes da infraestrutura. Por exemplo, é possível configurar alguns serviços, como o Eureka, para invocar o health check visando determinar se o tráfego deve ser roteado para aquela instância de serviço
- Outras opções são configurar o Docker e Kubernetes para invocar um endpoint de health check.

APLICANDO O PADRÃO LOG AGGREGATION

- Problema: Logs são uma ferramenta valiosa para resolução de problemas. No entanto, seu uso é desafiador numa arquiteutra de microsserviços, visto que é preciso consultar múltiplos logs referentes a cada serviço.
- ***Padrão: Log aggregation*** *- Agrega os logs de todos os serviços em uma base de dados central que suporta busca e notificações.*
  - Exemplos de serviços: CloudWatch, Loggly

APLICANDO O PADRÃO LOG AGGREGATION



COMO GERAR O LOG

- Duas decisões a tomar:
  - 1. Qual biblioteca de logging utilizar
    - Bibliotecas Java populares para logging libraries: Logback, log4j, JUL (java.util.logging) e SLF4J (API de fachada para os vários frameworks de logging).
  - 2. Onde escrever as entradas de log
    - Tradicionalmente, pode-se configurar o framework de logging para escrever um arquivo de log em alguma localização do sistema de arquivos
    - No entanto, com as tecnologias de implantação mais modernas, geralmente essa não é a melhor abordagem.
    - Em alguns ambientes, como AWS Lambda, não há um filesystem permanente para escrever logs. Neste caso, o serviço deve jogar os logs para a saída padrão (stdout) e a infraestrutura de implantação que irá decidir o que fazer com a saída do serviço.

A INFRAESTRUTURA DE AGREGAÇÃO DE LOGS

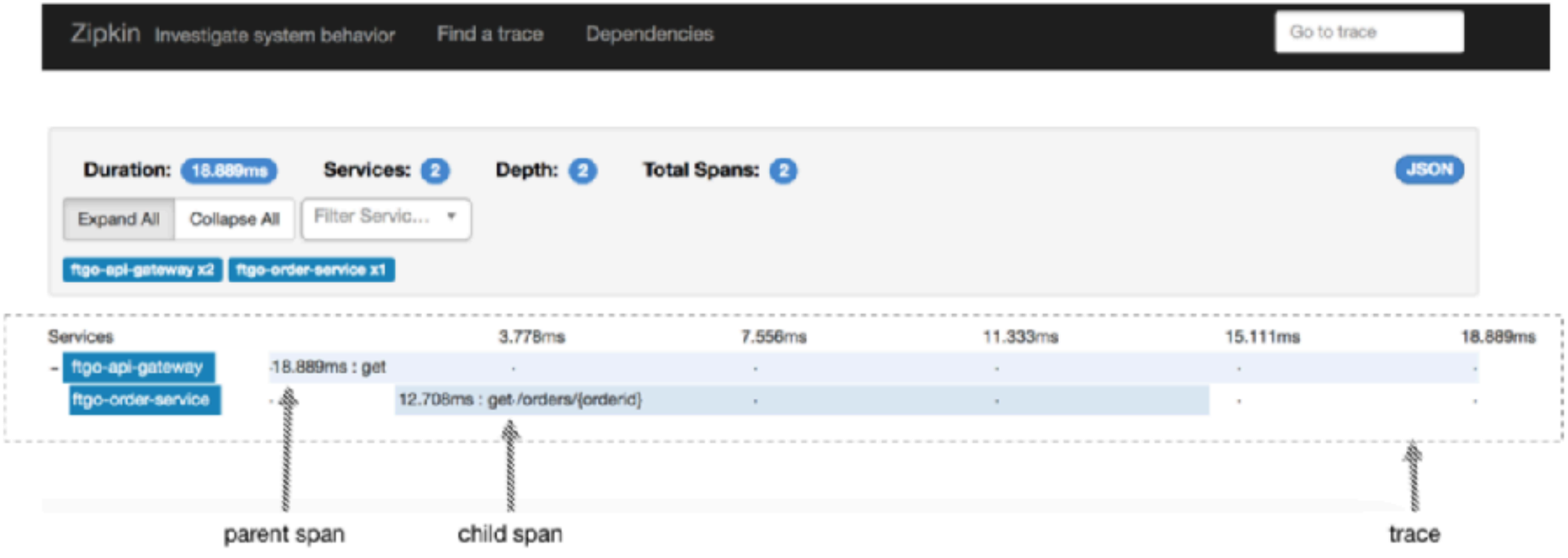
- Infraestrutura responsável por agregar, guardar e permitir que o usuário busque nos logs
- Uma infraestrutura popular é a ELK stack. ELK consiste em três produtos open source:
  - Elasticsearch - um banco de dados NoSQL orientado a texto que é usado como servidor de logging
  - Logstash - um pipeline de log que agrega os logs de serviço e escreve eles no Elasticsearch
  - Kibana - uma ferramenta de visualização para Elasticsearch
- Outro exemplos de infraestrutura de log incluem Fluentd e Apache Flume

USANDO O PADRÃO DISTRIBUTED TRACING

- Suponha que um desenvolvedor queira investigar porque um determinado método está respondendo lentamente.
- Uma opção seria olhar para o tempo de resposta total, porém a requisição pode envolver chamadas a vários serviços, o que dificulta a identificação.
- Padrão: Distributed tracing - Atribui para cada requisição um ID único e grava como o fluxo ocorreu dentro do sistema de um serviço para o próximo em um servidor central que provê visualização e análise.
- Distributed tracing grava informações (ex.: tempo inicial, tempo final) sobre a árvore de chamadas ao serviços que são feitas durante o tratamento de uma requisição.
- É possível ver quais serviços interagem, incluindo o detalhamento de quanto tempo é gasto em cada chamada.

USANDO O PADRÃO DISTRIBUTED TRACING

- Um trace representa uma requisição externa e consiste de uma ou mais spans. Uma span representa uma operação e seus atributos, como nome, início e fim.



USANDO O PADRÃO DISTRIBUTED TRACING

- Um efeito colateral valioso do distributed tracing é que ele assinala um id único para cada requisição, de maneira que um serviço pode incluir esse id nos registro de log.
- Quando combinado com log aggregation, o id da requisição permite encontrar facilmente todos os registros de log para uma requisição particular.

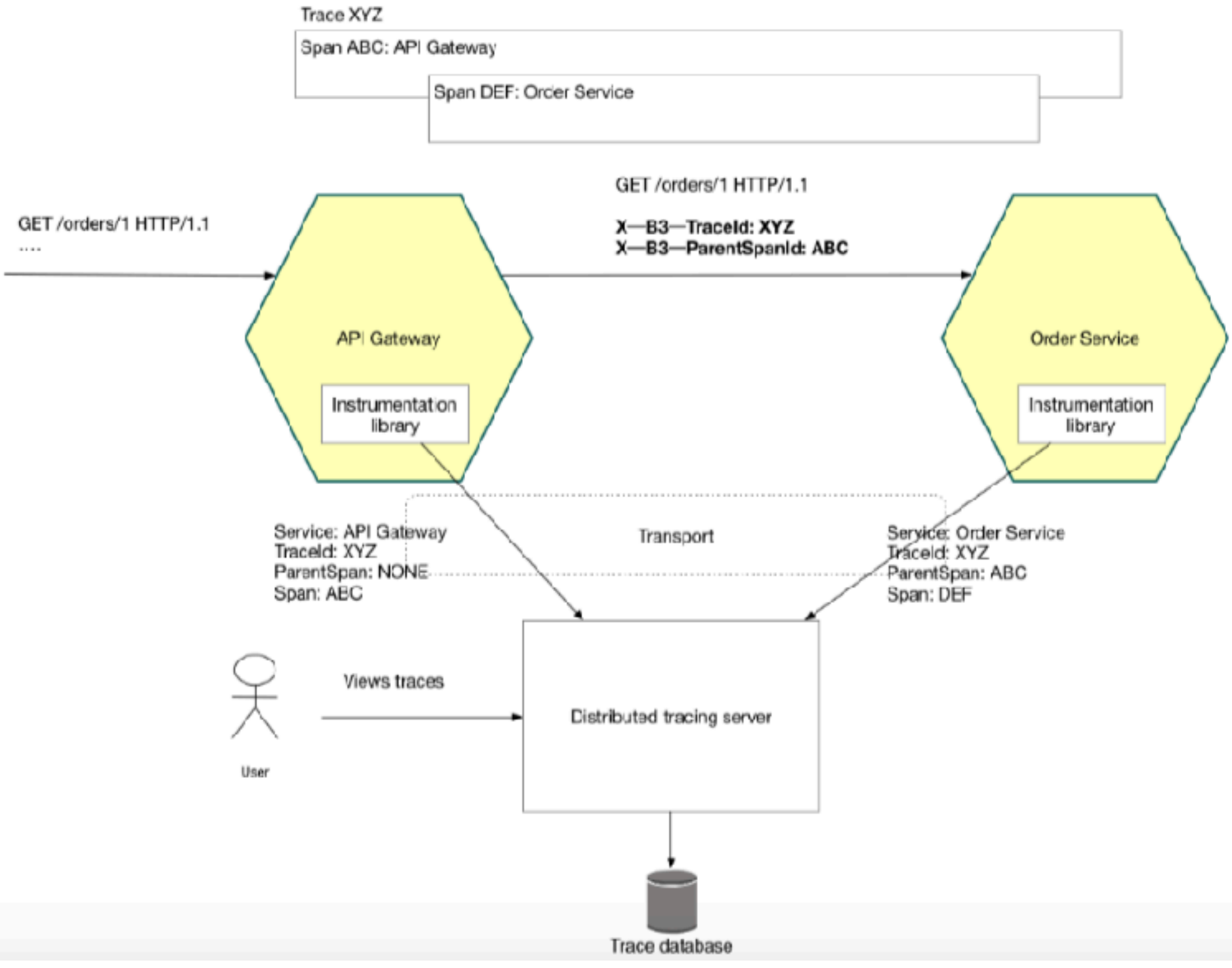
USANDO O PADRÃO DISTRIBUTED TRACING

- Exemplo:

```
2018-03-04 17:38:12.032 DEBUG [ftgo-order- service,
8d8fdc37be104cc6,8d8fdc37be104cc6,false] 7 --- [nio-8080-exec-6]
org.hibernate.SQL : select order0_.id as id1_3_0_, order0_.consumer_id as
consumer2_3_0_, order0_.city as city3_3_0_, order0_.delivery_state as
delivery4_3_0_, order0_.street1 as street5_3_0_, order0_.street2 as street6_3_0_,
order0_.zip as zip7_3_0_, order0_.delivery_time as delivery8_3_0_, order0_.a
```

- ftgo-order-service - nome da aplicação
  - 8d8fdc37be104cc6 - id do trace
  - 8d8fdc37be104cc6- span do trace
  - false - indica que este span não foi expotado para o servidor de distributed tracing
- Ao buscar os nos logs por 8d8fdc37be104cc6, é possível encontrar todos os registros de log para aquela requisição

DISTRIBUTED TRACING – COMO FUNCIONA



USANDO UMA BIBLIOTECA DE INSTRUMENTAÇÃO

- Cria a árvore de spans e as manda para o servidor de distributed tracing.
- O serviço pode chamar a biblioteca de instrumentação diretamente, porém isso poluiria a lógica de negócio
- Outra abordagem é usar interceptadores ou Aspect-Oriented Programming (AOP)
- Um bom exemplo de framework AOP é o Spring Cloud Sleuth. Ele usa o mecanismo de AOP do Spring para "automagicamente" integrar o distributed tracing no serviço.
- Como resultado, você apenas precisa adicionar a dependência para o Spring Cloud Sleuth.

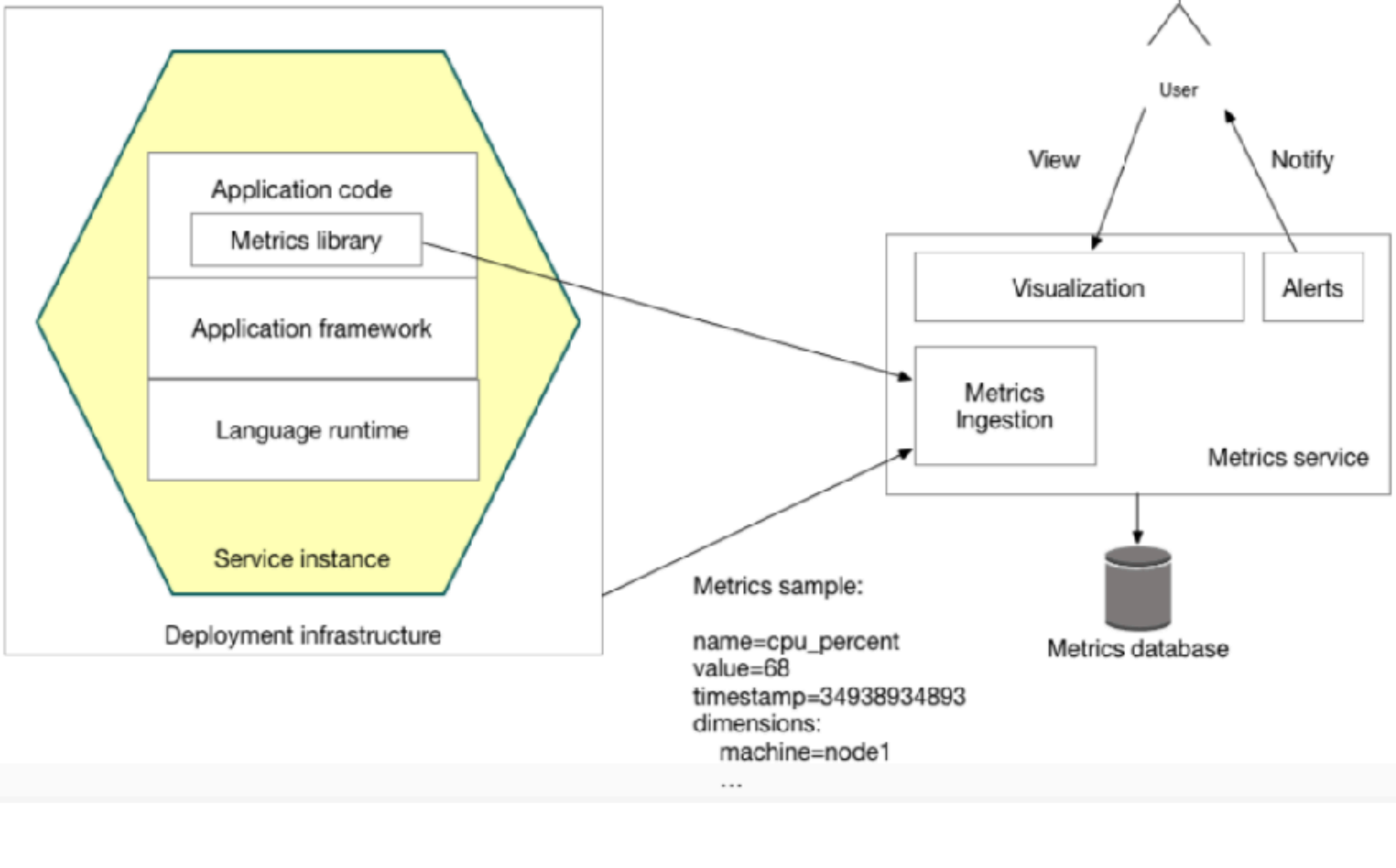
SOBRE O SERVIDOR DE DISTRIBUTED TRACING

- A biblioteca de instrumentação envia os spans para o servidor de distributed tracing
- O servidor junta os spans para formar traces completas e as guarda em um banco de dados
- Um servidor popular de distributed tracing é o Open Zipkin.
- Zipkin foi originalmente desenvolvido pelo Twitter.
- Serviços podem entregar spans para o Zipkin usando tanto HTTP como um Message Broker
- Zipkin guarda os traces em um backend que pode usar um banco SQL ou NoSQL.
- Zipkin também possui uma interface gráfica que exibe as traces.
- Outro exemplo de servidor para distributed tracing é o AWS X-ray

APLICANDO O PADRÃO METRICS

- Uma parte essencial de um ambiente de produção é monitoramento e notificações.
- O sistema de monitoramento coleta métricas, que proveem informações críticas sobre o estado da aplicação e seus componentes
- Métricas abordam desde o nível de infraestrutura, como uso de CPU, memória e disco até métricas no nível da aplicação, como latência e número de requisições executadas.
- ***Padrão: Application metrics*** *- Serviços reportam métricas para um servidor central que provê agregação, visualização e notificação.*

EXEMPLO DE COLETA DE MÉTRICAS



COLETANDO MÉTRICAS

- Métricas são coletadas periodicamente
- Uma métrica possui as seguintes propriedades: nome (ex.: jvm_memory_max_bytes, pedidos_realizados), valor (um valor numérico) e timestamp (hora da coleta)
- Muitos aspectos do monitoramento de métricas é responsabilidade de "Ops". No entanto, o desenvolvedor é responsável por dois aspectos:
  - Instrumentalizar os serviços para coletar métricas sobre seu comportamento
  - Expor essas métricas juntamente com métricas da JVM e do framework da aplicação a um servidor de métricas

COLETANDO MÉTRICAS NO NÍVEL DE SERVIÇO

- A quantidade de métricas a ser coletada depende da tecnologia utilizada pela aplicação
- Serviços Spring Boot podem coletar, por exemplo, métricas básicas da JVM simplesmente incluindo a biblioteca Micrometer Metrics como dependência e incluindo algumas linhas de configuração.

COLETANDO MÉTRICAS NO NÍVEL DE SERVIÇO

- Exemplo:

```
public class OrderService {

  @Autowired
  private MeterRegistry meterRegistry;                                    (1)

  public Order createOrder(...) {
    ...
    meterRegistry.counter("placed_orders").increment();                   (2)
    return order;
  }

  public void approveOrder(long orderId) {
    ...
    meterRegistry.counter("approved_orders").increment();                 (3)
  }

  public void rejectOrder(long orderId) {
    ...
    meterRegistry.counter("rejected_orders").increment();                 (4)
  }
```

ENTREGANDO MÉTRICAS AO SERVIÇO DE MÉTRICAS

- Um serviço compartilha métricas de duas formas: push ou pull.
- No modelo push, a métrica é enviada via API (ex.: métricas do AWS Cloudwatch)
- No modelo pull, as métricas solicitam (via API) ao servidor que puxe as métricas da instância do serviço. (ex.: Prometeus)

ENTREGANDO MÉTRICAS AO SERVIÇO DE MÉTRICAS

- O servidor Prometheus periodicamente faz polling neste endpoint para recuperar métricas
- Uma vez que as métricas estão no Prometheus, é possível visualizá-las usando Grafana.
- Grafana é uma ferramenta de visualização que pode fornecer alertas sobre essas métricas (ex.: o uso de memória excede o limite).
- As métricas da aplicação proveem insights valiosos sobre o coportamento da aplicação
- O disparo de alertas a partir de métricas permitem que um problema em produção seja tratada de maneira rápida, talvez até mesmo antes de impactar nos usuários

USANDO O PADRÃO EXCEPTION TRACKING

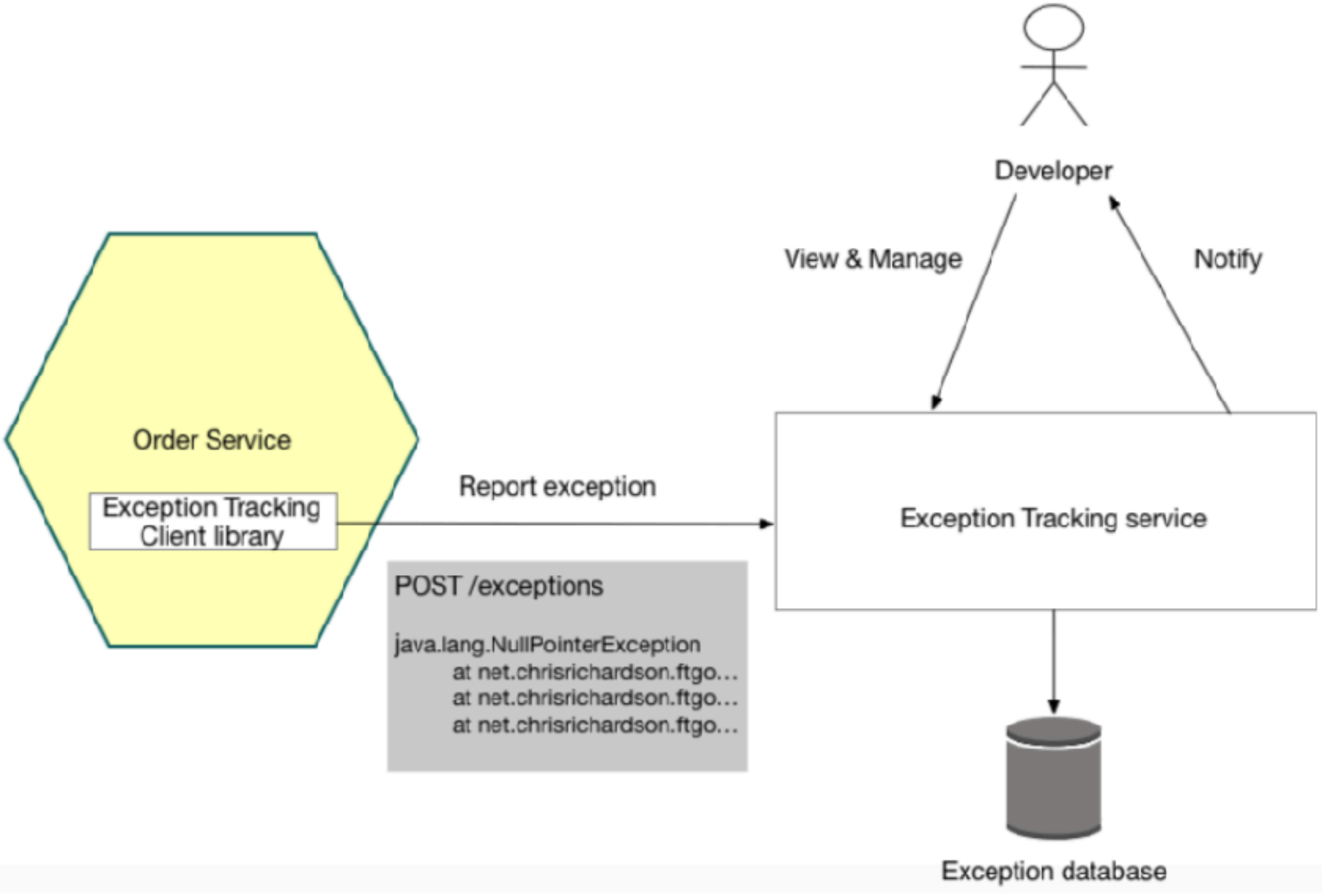
- Um serviço raramente deve logar uma exception e quando faz, é importante identificar a causa raiz. A exceção deve ser um sintoma de falha ou um bug de programação. A forma tradicional de visualizar exceções é olhar nos logs
- O servidor de logging deve ser configurado para alertar o desenvolvedor se uma exceção aparecer no log
- Há porém, alguns problemas com essa abordagem:
  - Logs são arquivos orientados a linhas únicas, enquanto que exceções consistem de múltiplas linhas
  - Não há mecanismo para rastrear a resolução de exceções que ocorrem nos arquivos de log
  - Geralmente há exceções duplicadas, mas não há um mecanismo automático para tratá-las como uma só

USANDO O PADRÃO EXCEPTION TRACKING

- Pattern: Exception tracking - *Serviços reportam exceções para um serviço central que unifica exceções, gera alertas e gerencia a resolução de exceções*

- A melhor abordagem para isso é utilizar um serviço de rastreamento de exceções

USANDO O PADRÃO EXCEPTION TRACKING



SERVIÇOS DE RASTREAMENTO DE EXCEÇÕES

- HoneyBadger.io (totalmente baseado na nuvem)
- Sentry.io (possui versão open-source que pode ser implantada em infra-estrutura própria)

- Esses serviços recebem exceções da aplicação e geram alertas
- Eles proveem um console para visualização de exceções e gerenciamento de suas resoluções
- Eles proveem bibliotecas de clientes em várias linguagens

INTEGRANDO RASTREAMENTO DE EXCEÇÕES NA APLICAÇÃO

- Uma forma de se integrar com um serviço de rastreamento e exceções é chamar a API serviço diretamente a partir do lançamento da exceção
- Uma melhor abordagem é usar a biblioteca fornecida pelo serviço.
- Por exemplo, a do HoneyBadger provê um mecanismo de integração fácil incluindo um Servlet Filter que recupera e reporta exceções.

APLICANDO O PADRÃO AUDIT LOGGING

- O propósito de audit loggin é gravar cada ação de usuários
- Um audit log tipicamente é usado para suporte aos clientes, garantir a conformidade do sistema e detectar comportamento suspeito.
- Cada log de auditoria identifica o usuário, a ação realizada e o objeto de negócio
- Uma aplicação geralmente grava o log de auditoria numa tabela de banco de dados
- Pattern: Audit logging - *Registra ações de usuários em um banco de dados visando ajudar no suporte de clientes, garantir conformidade e detectar comportamento suspeito.*

IMPLEMENTANDO O PADRÃO AUDIT LOGGING

- **Adicionar o código de auditoria à lógica de negócios**
  - Cada método cria um registro de audit log e salva no banco. Desvantagens: polui negócio e depende do desenvolvedor.
- **Usa programação orientada a aspectos (AOP)**
  - Intercepta automaticamente cada ação e persiste como audit log. Desvantagens: pode ser difícil mapear automaticamente um método pelo nome para uma operação do sistema e quais objetos de negócio estão relacionados
- **Usar event sourcing**
  - Identificando o usuário de cada evento, é possível rastrear o histórico completo para cada entidade. Desvantagens: não armazena consultas e exige manutenção da uma infraestrutura mais complexa.

4. DESENVOLVENDO MICROSSERVIÇOS USANDO O PADRÃO CHASSIS

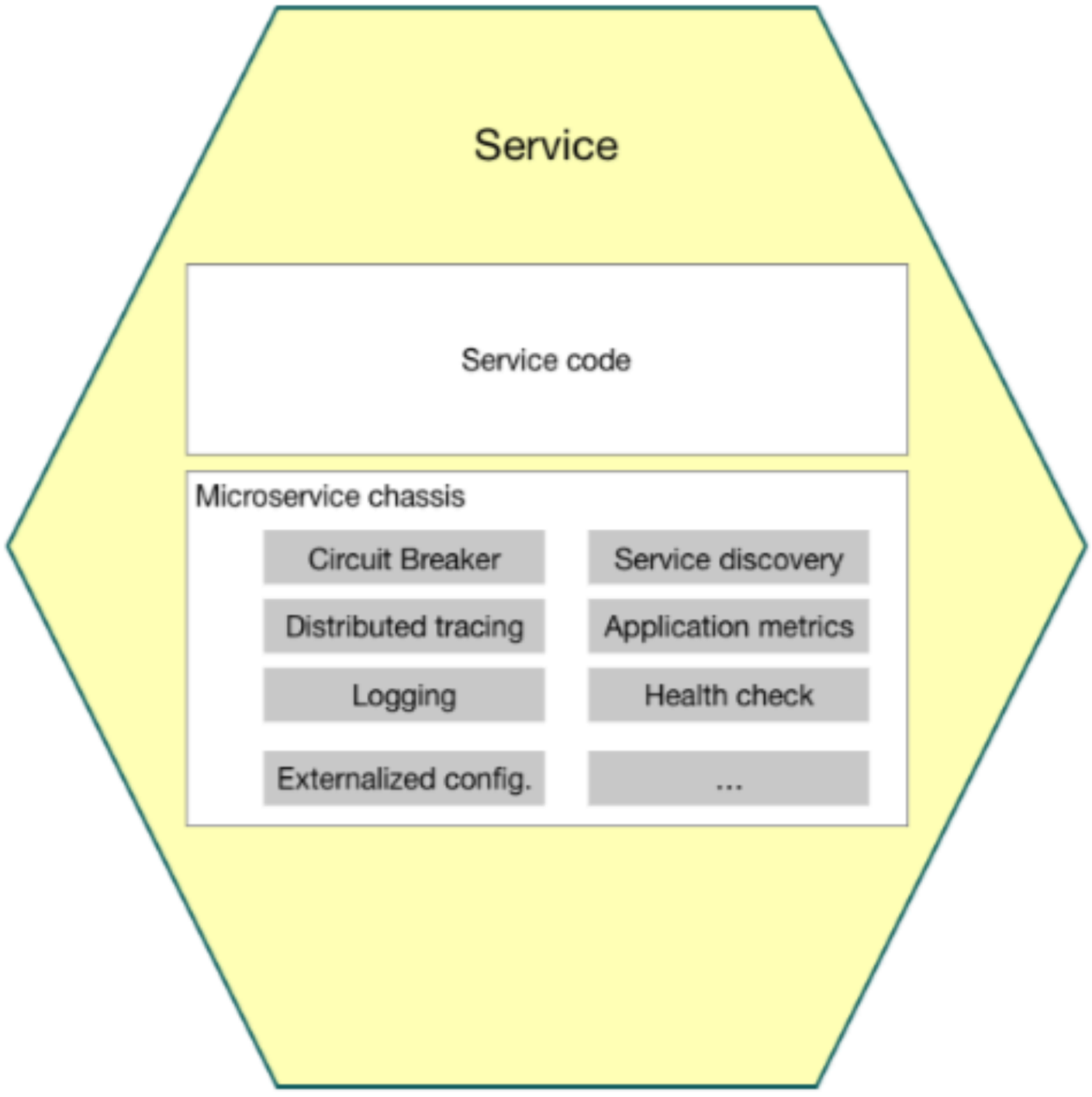
USANDO O PADRÃO MICROSERVICE CHASSIS

- Até então foram descritos vários cuidados que um serviço precisa implementar
- Além disso, um serviço também precisa implementar Service Discovery e Circuit Breakers.
- Isto não é algo que você precise configurar do zero toda vez que for implementar um novo serviço, visto que isso poderia levar dias ou até semanas…

USANDO O PADRÃO MICROSERVICE CHASSIS

- Uma forma mais fácil de desenvolver os serviços é fazer o desenvolvimento em cima de microservices chassis.
- Pattern: Microservice chassis - Construir serviços sob um framework ou coleção de frameworks que tratam as principais demandas de microsserviços, tais como exception tracking, logging, health checks, externalized configuration e distributed tracing.

USANDO O PADRÃO MICROSERVICE CHASSIS



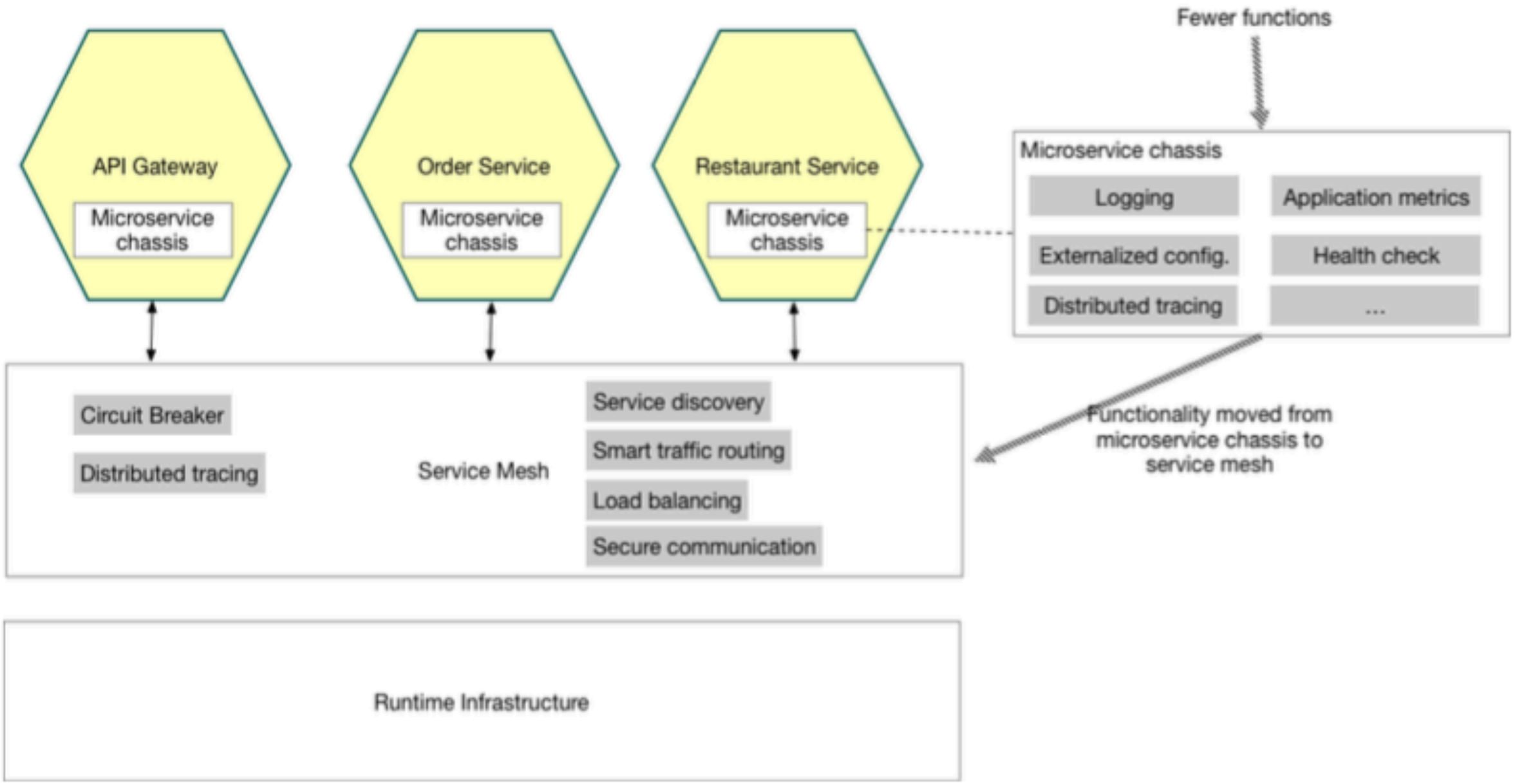
USANDO O PADRÃO MICROSERVICE CHASSIS

- Spring Boot e Spring Cloud podem ser usados como microservice chassis
- Spring Boot provê funções como externalized configuration.
- Spring cloud provê funções como circuit breaker e service discovery.
- Spring Boot e Spring Cloud não são os únicos frameworks de microservices chassis. Se os serviços forem escrito em GoLang há o Go Kit ou o Micro.

MICROSERVICES MESH

- Uma desvantagem de microservice chassis é que precisa ser escolhido um para cada linguagem de programação utilizada.
- Spring Bott e Spring cloud são úteis para o desenvolvimento em Java, mas não se deseja-se escrever um serviço baseado em NodeJS.
- Pattern: Service Mesh - Roteia todo o tráfego de saída e entrada através de uma camada acessível pela rede que implementa vários cuidados como circuite breaker, distributed tracing, service discovery, load balancing e roteamento de tráfego baseado em regras

MICROSERVICES MESH



ESTADO ATUAL DAS IMPLEMENTAÇÕES DE SERVICE MESH

- Implementações:
  - Istio
  - Linkerd2
- O conceito de service mesh é uma ideia extremamente promissora.
- Isso deixa o desenvolvedor livre de lidar com vários cuidados
- A habilidade do service mesh também permite a separação do ambiente de implantação da release
- Isso dá a habilidade para implantar uma nova versão de um serviço em produção somente para alguns usuários, como para testes internos
- Esse conceito é bastante aplicado com o uso de Kubernetes