Microservices infrastructure

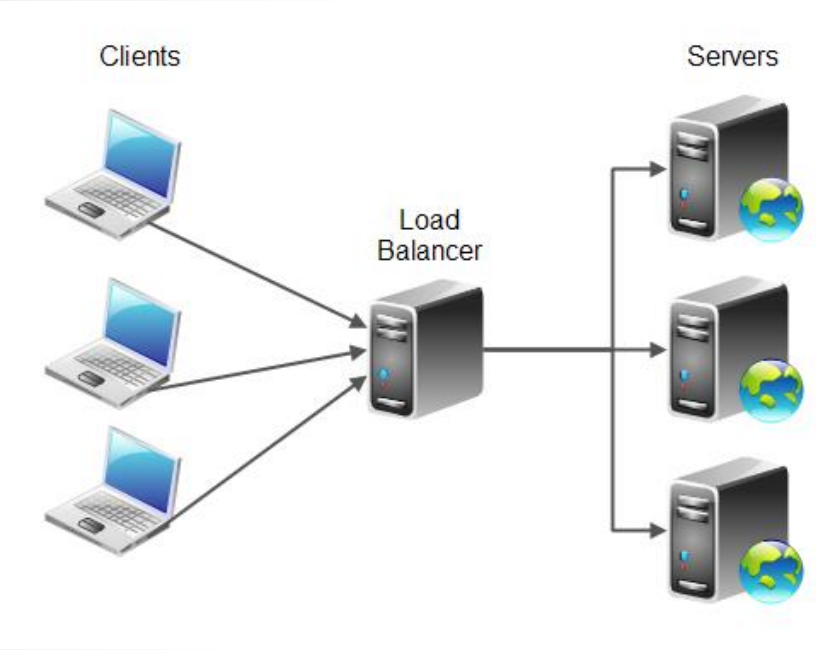
August, 3-4nd 2019

Sergey Morenets, 2019

Load balancing

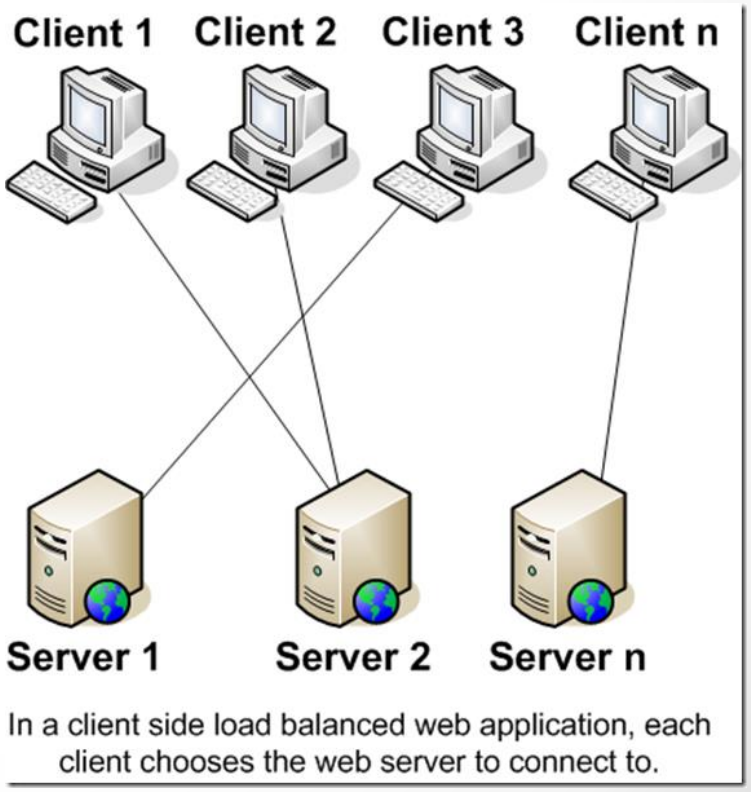
- ✓ Servers may go down and become unavailable
- ✓ Servers may be heavy loaded
- ✓ Servers may reside in the different zones

Load balancing



Server-side

Client-side



Load balancing. Pro & cons

Central solution
Single point of failure
Statistics aggregation
Easy to change

Server-side

Load balancer per service
Distributed processing
Add client-side library
Easy to customize

Client-side

Spring Cloud Netflix Ribbon

- ✓ Client-side load balancer which is grouped into application set by specific name
- ✓ Integrates with Spring Cloud Eureka
- ✓ Caching/batching
- ✓ Built-in failure resilience (skipping non-live servers)
- ✓ Implements Circuit Breaker pattern
- ✓ Supports TCP/UDP/HTTP
- ✓ Analog is Elastic Load Balancer

Ribbon API

- **IRule (ZoneAvoidanceRule)**
  Load balancing rule in the application
- **IPing (NoOpPing)**
  Algorithm to determine server availability at run-time
- **ServerList (ConfigurableBasedServerList)**
  Can be dynamic or static
- **ILoadBalancer (ZoneAwareLoadBalancer)**
  Defines operations (get up/down servers, mark server down) for load balancer
- **ServerListFilter (HealthServiceServerListFilter)**
  Filtering the configured or dynamically obtained list of candidate servers with desirable characteristics

Ribbon load-balancing strategies

- **Simple Round Robin**

  use “round robin” algorithm
- **Weighted response time**

  use the average/percentile response times to assign dynamic "weights" per server
- **Random**

  randomly distributes traffic amongst existing servers

Ribbon configuration


```
spring.application.name=hello
```

Server application.application.properties

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-ribbon</artifactId>
</dependency>
```

Client application

RestTemplate

High-level abstraction over Apache Http components library

HTTP method	RestTemplate method
DELETE	delete
GET	getForObject, getForEntity
POST	postForObject
PUT	put

RestTemplate


```
public class RestClient {

    private final RestTemplate restTemplate =
            new RestTemplate();

    public List<Book> findBooks() {
        return (List<Book>) restTemplate.getForObject(
                "http://localhost:8080/book", List.class);
    }
```

Ribbon(client)


```
@SpringBootApplication
@RibbonClient(name="hello")
public class ClientApplication {
    public static void main(String[] args) {
        SpringApplication.run(ClientApplication.class,
                args);
    }

    @LoadBalanced
    @Bean
    RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

Remote service identifier

Ribbon(client)


```
@Autowired
RestTemplate restTemplate;

public String hi(String name) {
    return restTemplate.getForObject("http://hello/greeting/"
            + name, String.class);
}
```

Ribbon. Multiple servers


```
@SpringBootApplication
@RibbonClients({ @RibbonClient(name = "products"),
    @RibbonClient(name = "customers") })
public class ClientApplication {
    public static void main(String[] args) {
        SpringApplication.run(ClientApplication.class, args);
    }

    @Bean
    public IRule ribbonRule() {
        return new RandomRule();
    }
```

Ribbon configuration


```
hello:
  ribbon:
    eureka:
      enabled: false
    listOfServers: localhost:8080,localhost:7000
    ServerListRefreshInterval: 3000
```

Disable Eureka

```
hello:
  ribbon:
    MaxAutoRetries: 2
    MaxAutoRetriesNextServer: 2
    ServerListRefreshInterval: 2000
    ConnectTimeout: 5000
    ReadTimeout: 90000
```

Connection settings

Task 9. Ribbon client

1. Adds new method to **HitRepository** that returns number of hits for specific book. Adds new **REST service** that calls this method and returns hit number. Verify service behavior.
2. Add Ribbon dependency to the book application
3. Adds **@RibbonClient** annotation and **RestTemplate bean** for the bootstrap class of the book application.
4. Add **hitCount** field to the book class. Update **GET REST service** in the

Task 10. Ribbon client without Eureka

1. Remove Eureka dependency from book application
2. Start multiple instances of the logging application
3. Update **application.properties** (or .yml) of book application and include list of available logging servers
4. Start book application and invoke new REST service multiple times. Which logging instances are used now?

Spring Cloud Feign

- ✓ Based on OpenFeign 10.x
- ✓ Declarative REST client
- ✓ Allows to make service calls without single line of implementation
- ✓ Similar to RestTemplate
- ✓ Avoids duplication/boilerplate code
- ✓ API documentation
- ✓ Automatically integrates with Hystrix/Ribbon/Eureka

OpenFeign

- ✓ Java to HTTP client binder
- ✓ Based on Netflix Feign library which was deprecated and not maintained since 2016
- ✓ Inspired by Retrofit, JAX-RS and WebSocket
- ✓ Java 8 - based
- ✓ Integrates with Gson, Jackson, SAX and JAXB (for serialization)
- ✓ Uses OkHTTP, Apache HTTP Components or Java 11 HTTP Client

Spring Cloud Feign


```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

```
@SpringBootApplication
@EnableFeignClients
public class ClientApplication {
    public static void main(String[] args) {
        SpringApplication.run(ClientApplication.class,
                args);
    }
```

Spring Cloud Feign


```java
@FeignClient("product")
public interface ProductClient {

    @GetMapping("/")
    List<Product> getProducts();

    @GetMapping("/{id}")
    Product getProduct(@PathVariable("id") int id);

    @PostMapping("/")
    void saveProduct(@RequestBody Product product);
```

Service identifier

```java
@Autowired
private ProductClient productClient;
```

Feign Builder API


```
@Configuration
@Import(FeignClientsConfiguration.class)
public class FeignConfiguration {

    @Bean
    public ProductClient productClient(Decoder decoder,
            Encoder encoder, Client client) {
        return Feign.builder().client(client)
                .encoder(encoder).decoder(decoder)
                .requestInterceptor(
                        new BasicAuthRequestInterceptor("admin", "admin")
                .contract(new SpringMvcContract())
                .target(ProductClient.class, "http://product");
    }
```

Feign Builder API. Authentication


```
public class TokenRequestInterceptor implements
                        RequestInterceptor{
    private final String token;

    public TokenRequestInterceptor(String token) {
        this.token = token;
    }

    @Override
    public void apply(RequestTemplate template) {
        template.header("Authorization", token);
    }

}
```

Task 11. Feign client

1. Add **Feign** dependency to the book application:
2. Update bootstrap class of book application and enable Feign clients using **@EnableFeignClients** annotation.
3. Create new Feign client **HitClient** interface that will provide communication with logging application.
4. Update your REST service that sends requests to the logging application and switch from **RestTemplate** to **HitClient**.

Spring Cloud Loadbalancer

- ✓ Ribbon replacement
- ✓ Still incubator project
- ✓ Supports reactive programming paradigm
- ✓ Contains only round-robin algorithm

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-loadbalancer</artifactId>
</dependency>
```

Spring Cloud Loadbalancer


```
@Configuration
@LoadBalancerClients({
    @LoadBalancerClient(name = "order",
            configuration = LBServiceConfig.class),
    @LoadBalancerClient(name = "payment",
            configuration = LBServiceConfig.class) })
public class LoadBalancerConfiguration {
}

class LBServiceConfig {
    @Bean
    public RoundRobinLoadBalancer loadBalancer(
            LoadBalancerClientFactory clientFactory, Environment env) {
        String serviceId = clientFactory.getName(env);
        return new RoundRobinLoadBalancer(serviceId, clientFactory
```

```
loadbalancer:
  client:
    name: library
```

application.yml

Fault-tolerant systems

- Fault is incorrect internal state
- Error visible incorrect behavior
- Failure is inaccessibility/unavailability of the application feature
- Fault-tolerant systems still operate if a fault or faults happen
- If you execute operation against external system (service) and observes an error you can: retry, fallback or fail fast.

Fault-tolerant systems

- ✓ Fault was uncontrolled power boost
- ✓ Error was destruction of thermal elements
- ✓ Failure was explosion

Cascading failure

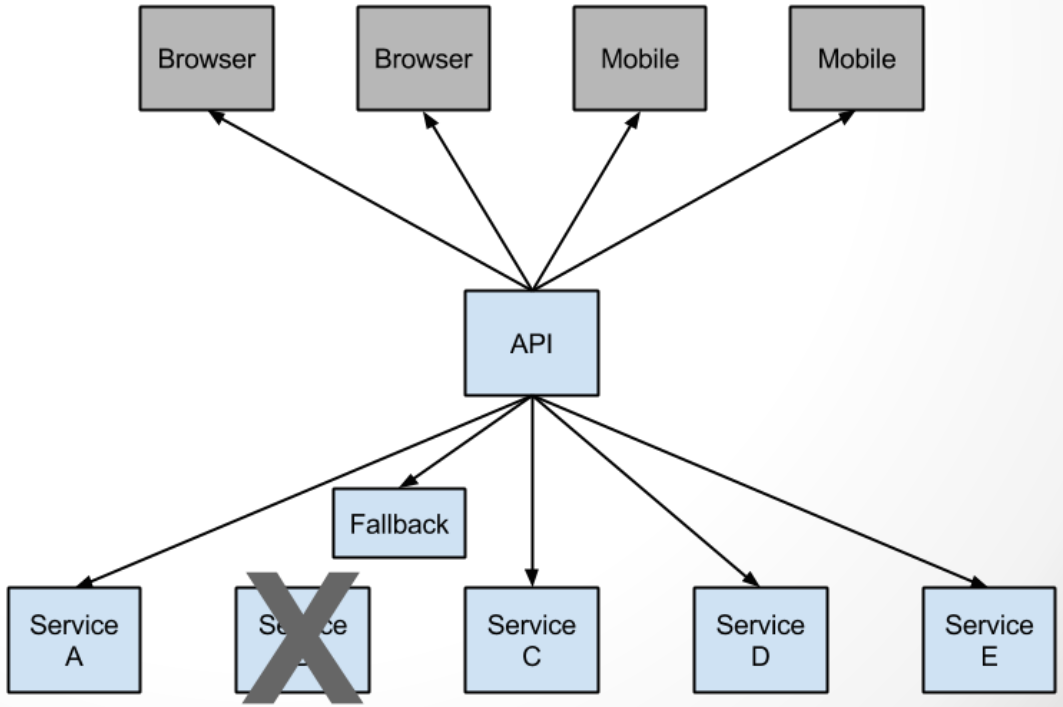
Sergey Morenets, 2019

Resilience attributes

- ✓ Circuit breaker
- ✓ Retries/ timeouts
- ✓ Fault injection/handling
- ✓ Failover

Circuit breaker pattern

- ✓ Prevents cascading failures
- ✓ When it discovers a failure it disconnects(opens) circuit
- ✓ When a problem is gone you can switch on(close) circuit



Spring Cloud Hystrix

- Sometimes called **bulk head**
- Integrates with **Turbine**
- By default detects 20 failures in 5 seconds
- Provides fallback in case of service dependency failure
- Automatically closes circuit breaker after 5 seconds

Hystrix status

- ✓ Hystrix is no longer in active development, and is currently in maintenance mode.
- ✓ Hystrix (at version 1.5.18) is stable enough
- ✓ It’s recommended to research new projects like resilience4j for real-time applications/adaptive performance
- ✓ Final version is 1.5.18

Spring Cloud Hystrix


```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
</dependency>
```

```
@SpringBootApplication
@EnableHystrix
public class ClientApplication {
    public static void main(String[] args) {
        SpringApplication.run(ClientApplication.class,
                args);
    }
```

Spring Cloud Hystrix. Fallback


```
@GetMapping("/handshake/{name}")
@HystrixCommand(fallbackMethod = "fallback")
public String hi(@PathVariable String name) {
    return helloClient.handshake(name);
}

public String fallback(String name) {
    return "Service temporarily unavailable";
}
```

```
@HystrixCommand(fallbackMethod = "fallback")
public String hi(String name) {
    return helloClient.handshake(name);
}

public String fallback(String name, Throwable t) {
    LOGGER.error(t);
    return "Service temporarily unavailable";
}
```

Hystrix. Configuration settings


```
@HystrixCommand(fallbackMethod = "fallback", commandProperties = {
        @HystrixProperty(name =
                "circuitBreaker.errorThresholdPercentage", value = "1")
public String hi(String name) {
    return helloClient.handshake(name);
}
```

Default config →

Command name →

```
hystrix:
  command:
    default:
      circuitBreaker:
        enabled: true
    hi:
      circuitBreaker:
        sleepWindowInMilliseconds: 60000
```

https://github.com/Netflix/Hystrix/wiki/Configuration

Hystrix. No fallback


```
@HystrixCommand
public String hi(@PathVariable String name) {
    return helloClient.handshake(name);
}
```

```
java.util.concurrent.TimeoutException: null
        at com.netflix.hystrix.AbstractCommand.handleTimeoutViaFallback(Abstra
        at com.netflix.hystrix.AbstractCommand.access$500(AbstractCommand.java
        at com.netflix.hystrix.AbstractCommand$12.call(AbstractCommand.java:60
        at com.netflix.hystrix.AbstractCommand$12.call(AbstractCommand.java:60
        at rx.internal.operators.OperatorOnErrorResumeNextViaFunction$4.onErro
        at rx.internal.operators.OnSubscribeDoOnEach$DoOnEachSubscriber.onErro
        at rx.internal.operators.OnSubscribeDoOnEach$DoOnEachSubscriber.onErro
        at com.netflix.hystrix.AbstractCommand$HystrixObservableTimeoutOperato
```

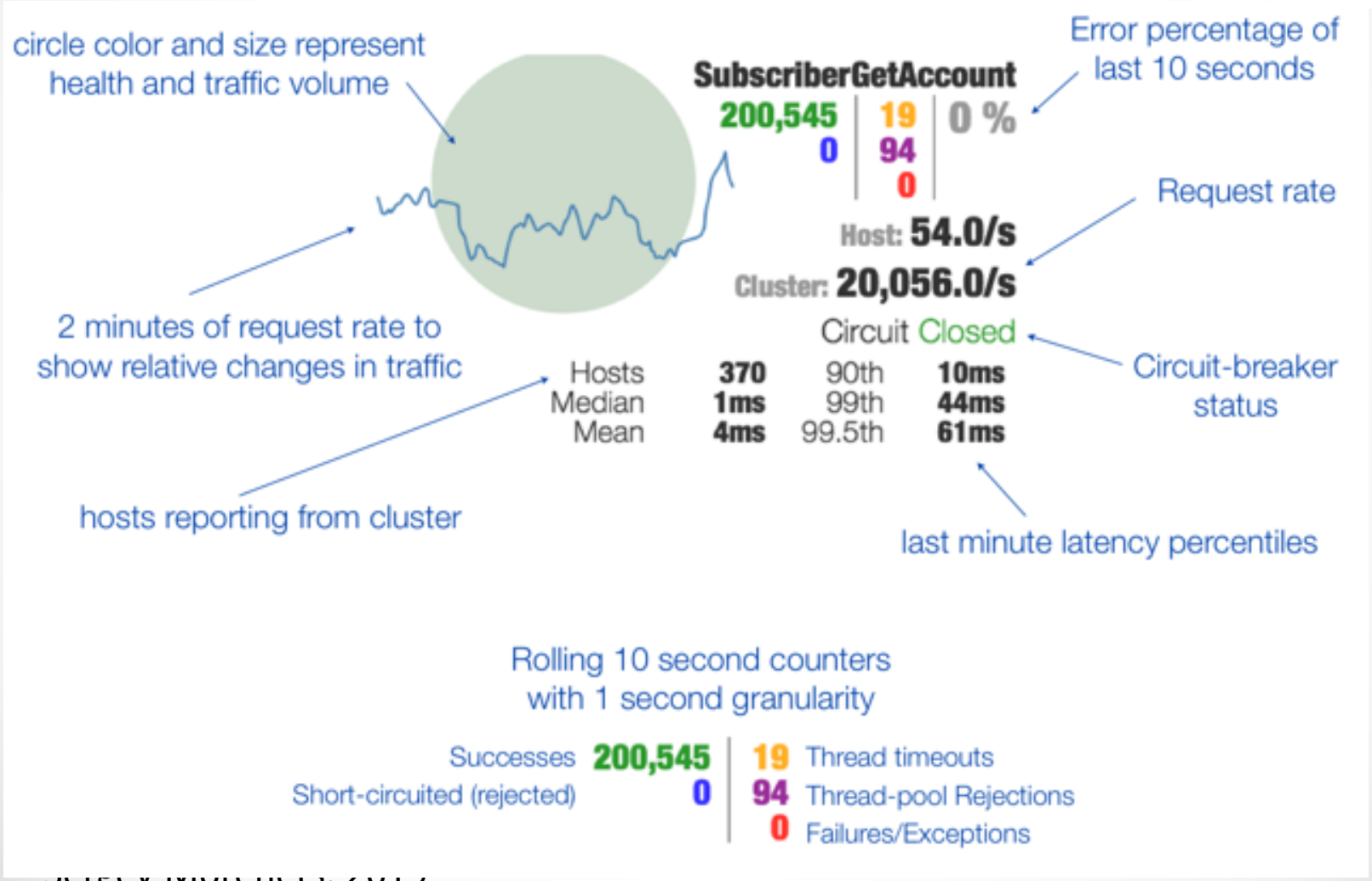
Netflix Chaos Monkey

- Requires Spinnaker (continuous delivery) and relational database(MySQL)
- Chaos Monkey runs as a cron job to schedule random terminations
- Integrates with etcd/Consul

Hystrix. Best practices

- If your client returns some response it would be better to return cached version in case of failure
- If you client executes a command it would be better to signal an error to the caller for re-try rather than return error message

Hystrix dashboard



Hystrix dashboard. Configuration


```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>
</dependency>
```

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

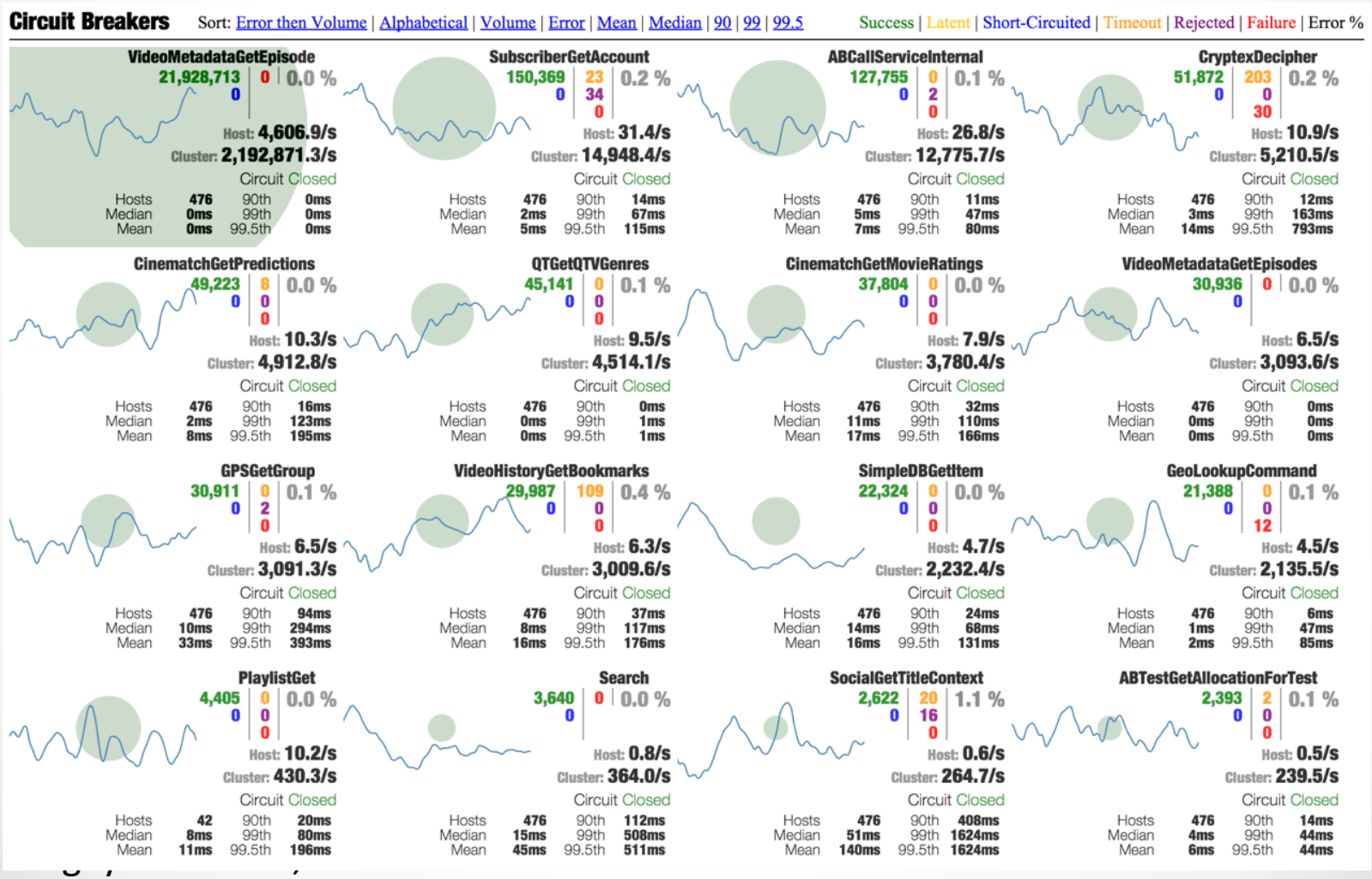
```
@SpringBootApplication
@EnableHystrix
@EnableHystrixDashboard
public class ClientApplication {
    public static void main(String[] args) {
        SpringApplication.run(ClientApplication.class,
                args);
    }
```

Hystrix Dashboard

- Allows to monitor Hystrix status on a dashboard
- Hystrix application send circuit breaker metrics as SSE events
- Turbine helps to aggregate streams from different Hystrix applications

```
management:
  endpoints:
    web:
      exposure:
        include: hystrix.stream,info,health
```

Turbine dashboard



Task 12. Hystrix

1. Add **Spring Cloud Hystrix** dependency:
2. Update ClientApplication and enable Hystrix support.
3. Update REST service that sends requests to the server and add support for **Hystrix fallback**.
4. Start **client/server** applications and make sure client correctly obtains responses from the server.

Spring Cloud Circuit Breaker

- ✓ Abstraction API to add different circuit breaker implementations to the application
- ✓ The following implementations are supported: Hystrix, Resilience4j, Sentinel, Spring Retry
- ✓ Still incubator project
- ✓ Supports reactive programming

Spring Cloud Circuit Breaker. Hystrix

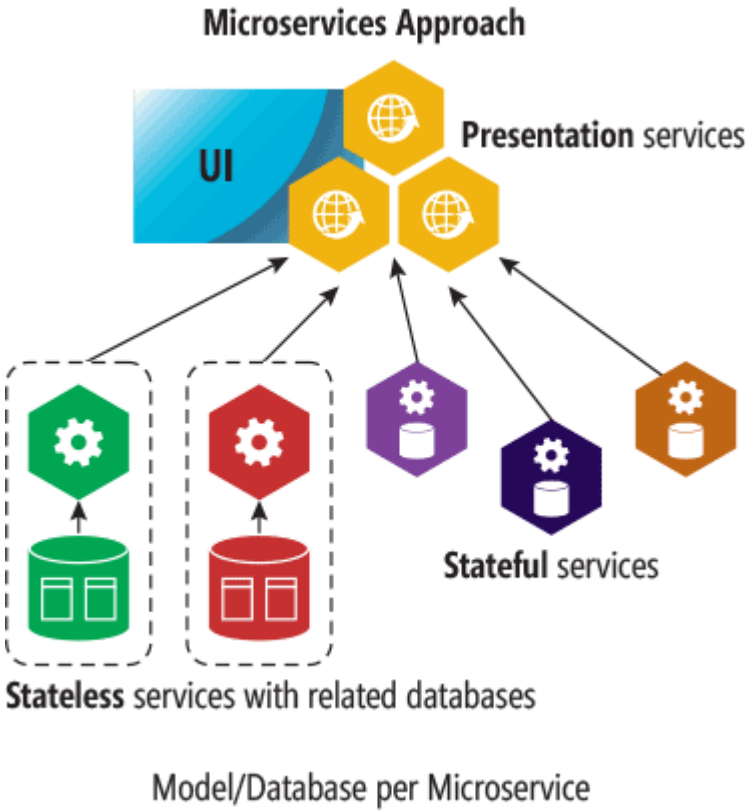

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-circuitbreaker-hystrix</artifactId>
    <version>0.0.1.BUILD-SNAPSHOT</version>
</dependency>
```

```
@Service
@RequiredArgsConstructor
public class OrderService {
    private final RestTemplate restTemplate;
    private final CircuitBreakerFactory factory;

    public Customer getCustomer(int id) {
        return factory.create("customer").run(() ->
            restTemplate.getForObject("/customer/" + id,
                    Customer.class),
            throwable -> getFromCache(id));
    }
```

Hystrix. Default configuration


```
@Configuration
public class CircuitBreakerConfiguration {

    @Bean
    public Customizer<HystrixCircuitBreakerFactory> defaultConfig() {
        return factory -> factory.configureDefault(id ->
                HystrixCommand.Setter
                .withGroupKey(HystrixCommandGroupKey.Factory.asKey(id))
                .andCommandPropertiesDefaults(
                        HystrixCommandProperties.Setter()
                .withExecutionTimeoutInMilliseconds(4000)
                .withCircuitBreakerErrorThresholdPercentage(30)
                .withFallbackEnabled(true)));
    }
}
```

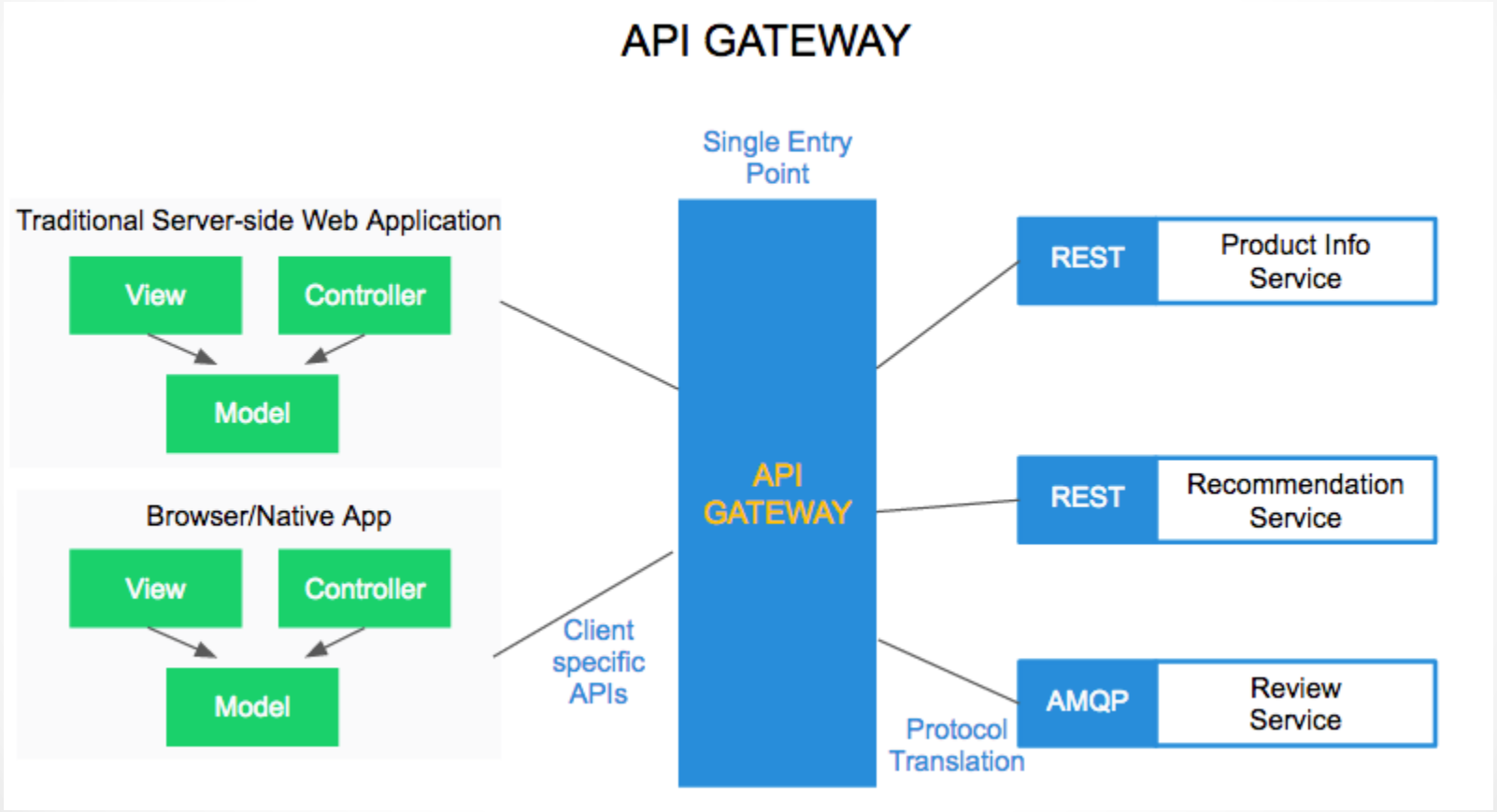
API gateway


Microservices Approach
UI
Presentation services
Stateful services
Stateless services with related databases
Model/Database per Microservice

Concerns

- ✓ **Internal API** exposing
- ✓ Security(CORS)
- ✓ Multiple calls

API gateway



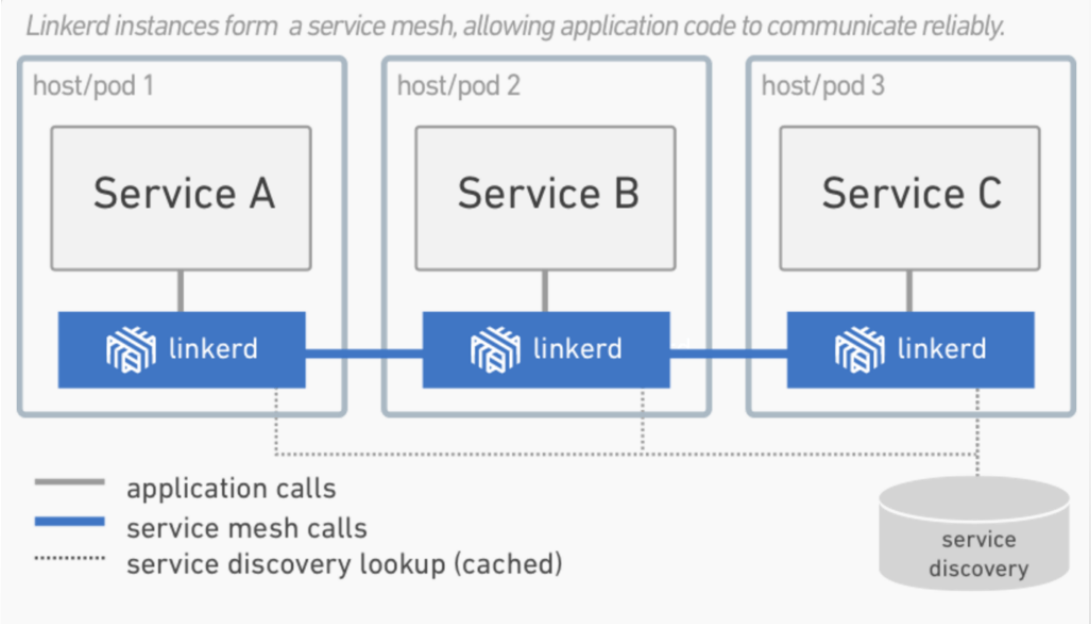
Sergey Morenets, 2019

API gateway

- ✓ **Façade pattern** providing concrete client needs
- ✓ Decreases remote calls
- ✓ Allows to easily add new clients
- ✓ Security
- ✓ Caching
- ✓ Protocol translation

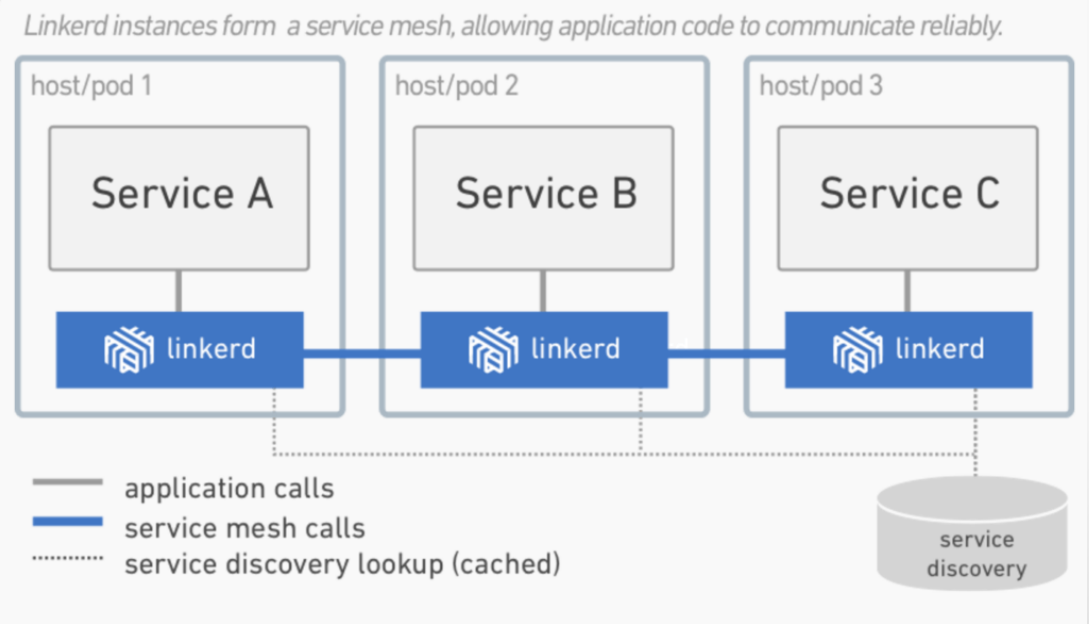
Service mesh pattern

- ✓ Inter-service communication infrastructure
- ✓ Contains proxy instance (sidecar) for each service instance so microservices don’t communicate directly but using proxy
- ✓ Provides container orchestration, resilience, service discovery, load balancing, encryption and authentication



Service mesh

- ✓ Language-agnostic
- ✓ Can use HTTP 1.x/2, gRPC or other protocols
- ✓ All proxies are managed by Control Plane
- ✓ Implementation: Istio (Kubernetes + etcd), Linkerd



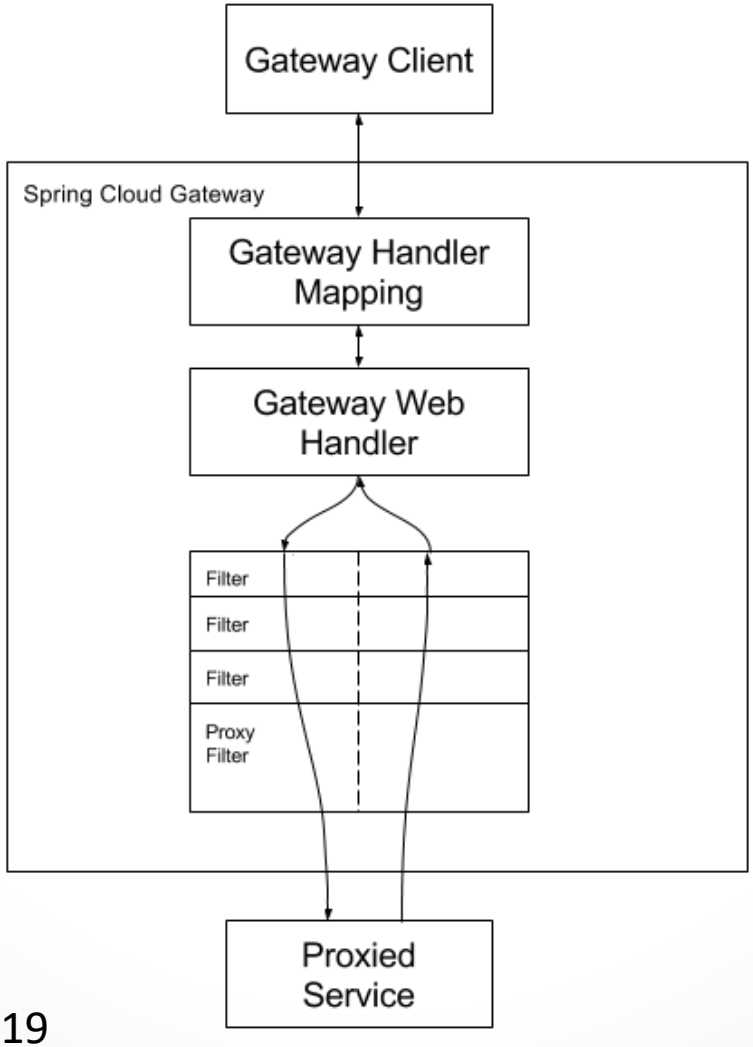
Spring Cloud Zuul

- **Router and server-side load balancer**
- Includes Ribbon/Hystrix
- Eureka service ID is URI
- Analogs are Kong, ApiGee, MuleSoft, Linkerd and Envoy
- Based on Zuul 1.3.x which is blocking/servlet 2.x-oriented
- Zuul 2.x is asynchronous/non-blocking/Netty-based
- **Spring Cloud Gateway** is Zuul competitor and based on Spring 5/Reactor/Spring Boot 2 and completely asynchronous

Spring Cloud Gateway

- ✓ Based on Reactor and Spring Framework 5
- ✓ Integration with Hystrix and Spring Cloud Discovery Client
- ✓ Path rewriting
- ✓ Contains routes, predicates and filters
- ✓ Predicates allow to match input requests
- ✓ Routes contains identifier, target URI and collection of predicates/filters

Spring Cloud Gateway



Spring Cloud Gateway. Configuration


```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
```

```
@SpringBootApplication
public class GatewayApplication {
    public static void main(String[] args) {
        SpringApplication.run(GatewayApplication.class, args);
    }
}
```

Service discovery support


```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

```
spring:
  cloud:
    gateway:
      discovery:
        locator:
          enabled: true
```

application.yml

Monitoring


```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

```
management.endpoint.gateway.enabled=true
management.endpoints.web.exposure.include=gateway
```

application.properties

Gateway routes


```
1:
  route_id:                "CompositeDiscoveryClient_LIBRARY-LOGGING"
  route_definition:
    id:                    "CompositeDiscoveryClient_LIBRARY-LOGGING"
    predicates:
      0:
        name:              "Path"
        args:
          pattern:         "/LIBRARY-LOGGING/**"
    filters:
      0:
        name:              "RewritePath"
        args:
          regexp:          "/LIBRARY-LOGGING/(?<remaining>.*)"
          replacement:     "/${remaining}"
    uri:                   "lb://LIBRARY-LOGGING"
    order:                 0
  order:                   0
```


Upper-case id

/actuator/gateway/routes

Gateway routes


```
spring.cloud.gateway.discovery.locator.lowerCaseServiceId=true
```


```
0:
   route_id:                "CompositeDiscoveryClient_LIBRARY-LOGGING"
  route_definition:
     id:                    "CompositeDiscoveryClient_LIBRARY-LOGGING"
    predicates:
      0:
         name:              "Path"
        args:
           pattern:         "/library-logging/**"
    filters:
      0:
         name:              "RewritePath"
        args:
           regexp:          "/library-logging/(?<remaining>.*)"
           replacement:     "/${remaining}"
```

Routes configuration

application.yml

```
spring:
  cloud:
    gateway:
      routes:
      - id: simulator
        uri: http://it-simulator.com
        predicates:
        - Path=/simulator
        - Method=GET
```

Handles requests where path starts with /simulator

Handles all GET requests

http://localhost:8080/simulator → http://it-simulator/simulator

Routes. Java API


```
@Bean
public RouteLocator routeLocator(RouteLocatorBuilder builder) {
    return builder.routes()
      .route("simulator", r ->
                r.uri("http://it-simulator.com:80"))
    .build();
}
```

java.lang.IllegalArgumentException: predicate can not be null

```
@Bean
public RouteLocator routeLocator(RouteLocatorBuilder builder) {
    return builder.routes().route("simulator",
            r -> r.method(HttpMethod.GET)
            .uri("http://it-simulator.com:80"))
        .build();
}
```

Routes. Filters

Check if HTTP request header exists

```
@Bean
public RouteLocator routeLocator(RouteLocatorBuilder builder) {
    return builder.routes().route("simulator",
            r -> r.method(HttpMethod.GET)
            .and().header("X-Req-Simulator", "true")
            .filters(f -> f.addResponseHeader(
                    "X-Resp-Simulator", "true"))
            .uri("http://it-simulator.com:80"))
        .build();
}
```

Add HTTP response header

Task 13. Spring Cloud Gateway

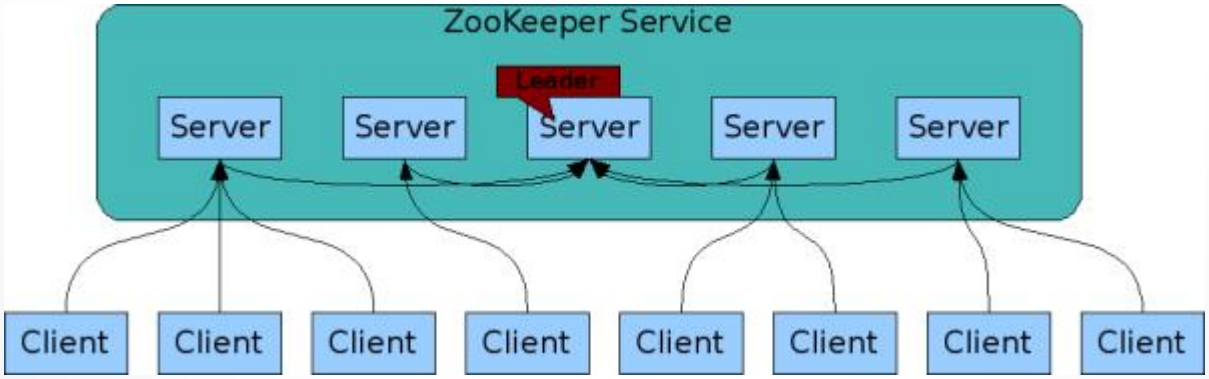
1. Review **gateway** project.
2. Update **application.yml** :
3. Add port settings
4. Add Eureka support
5. Start Gateway application. Which endpoints are available there?
6. Start other applications and try to access it using gateway routes.

Zookeeper

- Originally developed at Yahoo
- Centralized service for distributed synchronization
- Provides a distributed configuration service, synchronization service, and naming registry for large distribution systems
- Offers hierarchical key-value store
- Usually run in cluster
- Used by Kafka, Hadoop, Hive, Solr, Mesos, Neo4j and HBase

Zookeeper

- Stores data in-memory
- Transaction logs and snapshots are persisted
- Namespace contains data registers (znodes)
- Provides atomicity, sequential consistency, reliability and single system image



Spring Cloud Zookeeper

- ✓ Acts as a service discovery
- ✓ Integrates with Ribbon/Feign
- ✓ Includes Spring Cloud Zookeeper Config

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zookeeper-discovery</artifactId>
</dependency>
```

Spring Cloud Zookeeper


```java
@SpringBootApplication
@EnableDiscoveryClient
public class ZookeeperClientApplication {
```

```yaml
spring:
  cloud:
    zookeeper:
      discovery:
        enabled: true
        register: true
      connect-string: localhost:2181
      max-retries: 5
```

application.yml

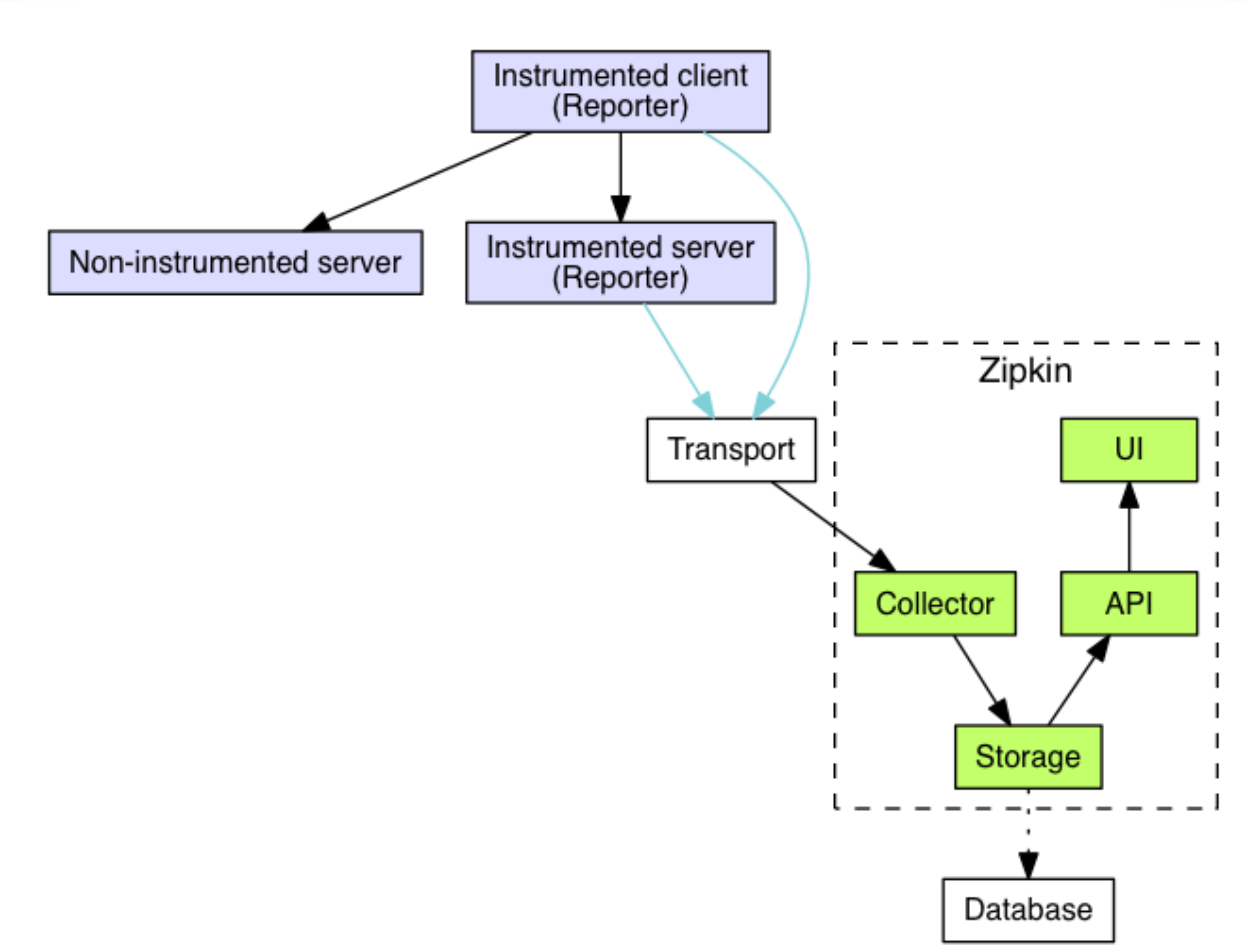
Task 14. Spring Cloud Zookeeper

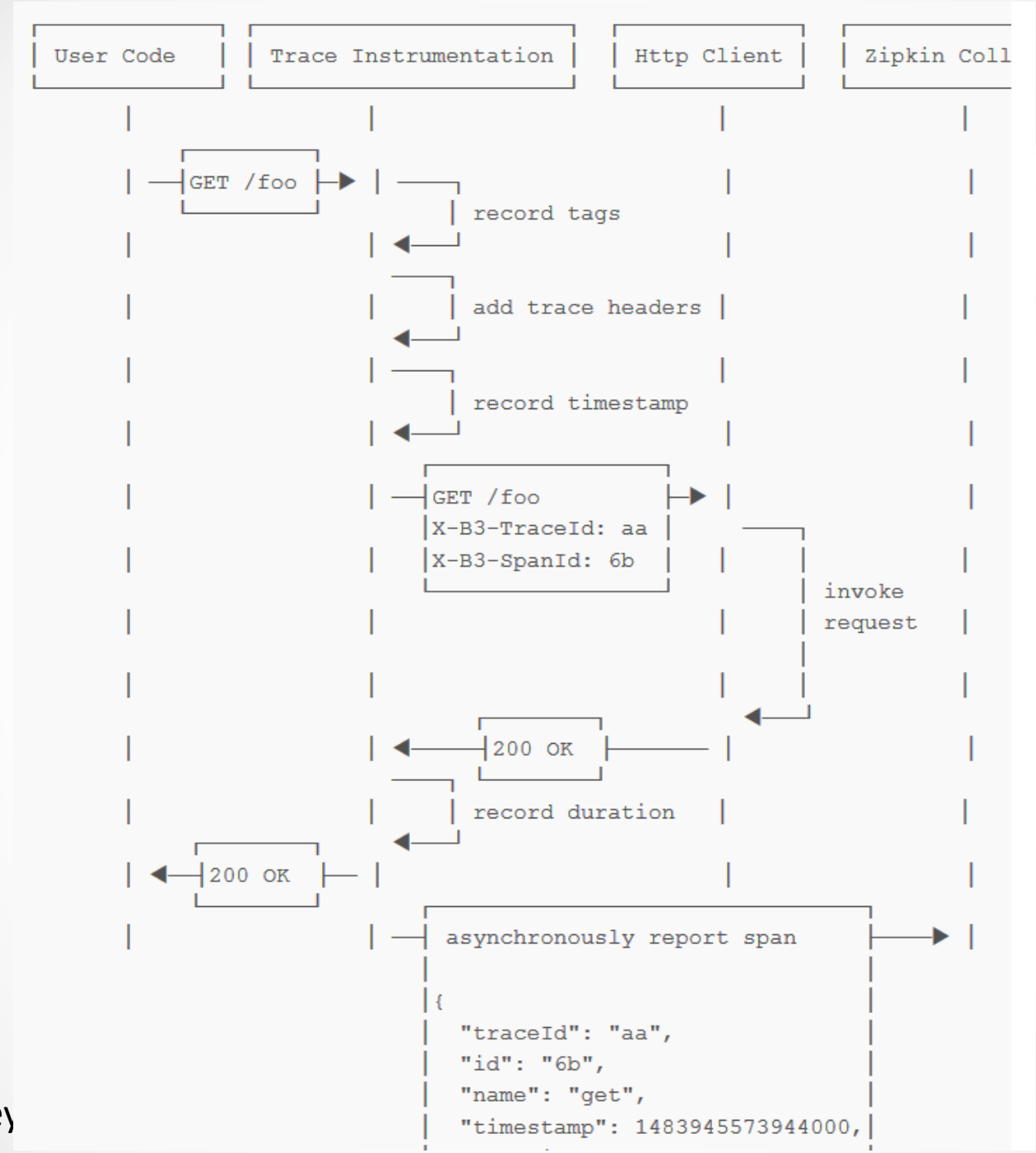
1. Start Zookeeper server
2. Run Zookeeper CLI in another terminal
3. Type “list” command to see all available commands
4. Type “ls /” command to view all the Zookeeper nodes
5. Replace Eureka service discovery with Zookeeper discovery, starting with dependencies:
6. Verify that service discovery works properly.

Zipkin

- Observability: logging, metrics and tracing
- Distributed tracing system which allows to track a request that goes through multiple microservices
- Distributed tracing allows to identify issues related to latency, losses and throughput
- Zipkin allows to visualize timing data as dependency graph
- Uses Http, Kafka, Scribe as transport
- Contains 4 components: collector, storage, search and Web UI
- Supports in-memory storage, MySQL, Cassandra, ElasticSearch

Zipkin architecture

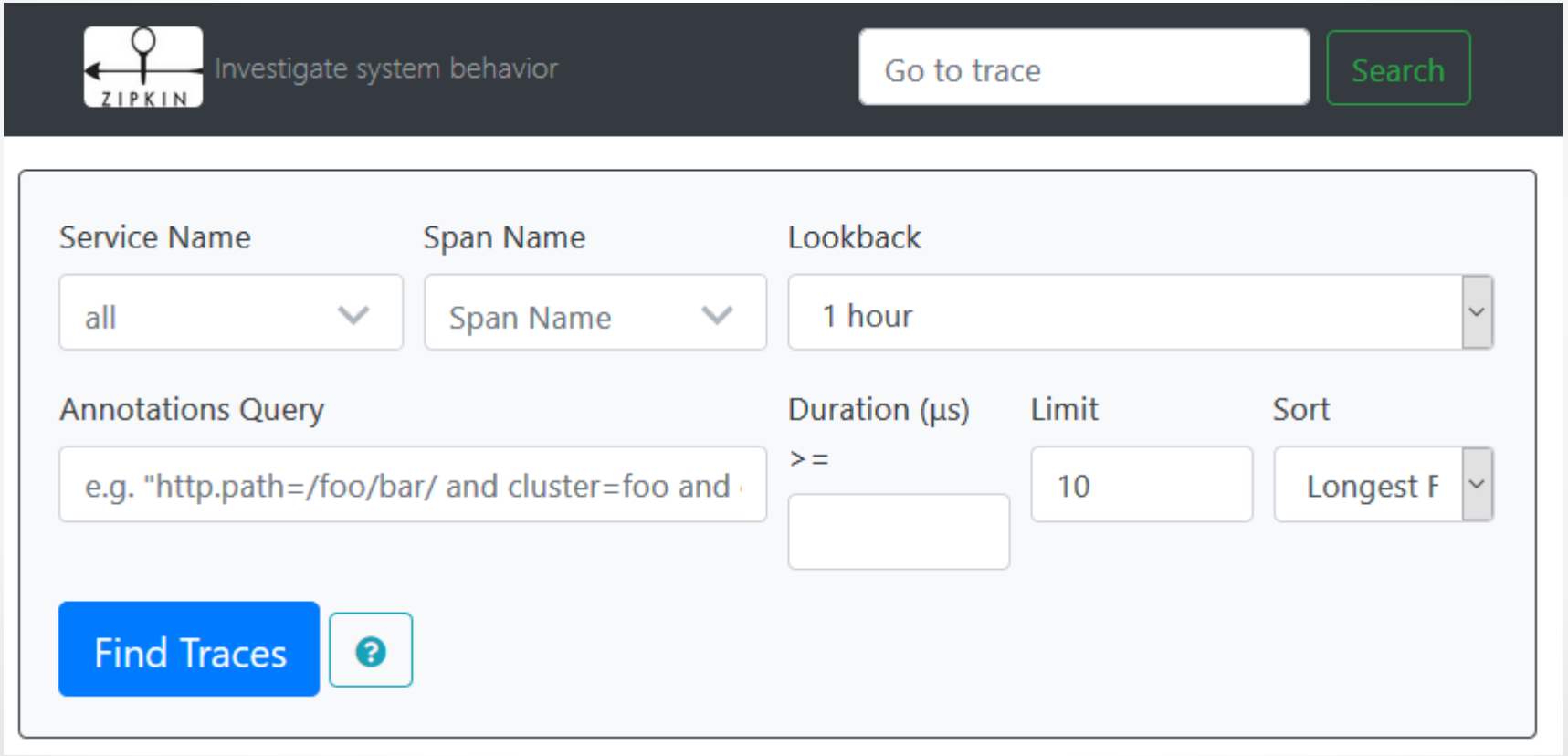



User Code
Trace Instrumentation
Http Client
Zipkin Coll
GET /foo
record tags
add trace headers
record timestamp
GET /foo
X-B3-TraceId: aa
X-B3-SpanId: 6b
invoke
request
200 OK
record duration
200 OK
asynchronously report span
{
"traceId": "aa",
"id": "6b",
"name": "get",
"timestamp": 1483945573944000,


IT
DISCOVERY

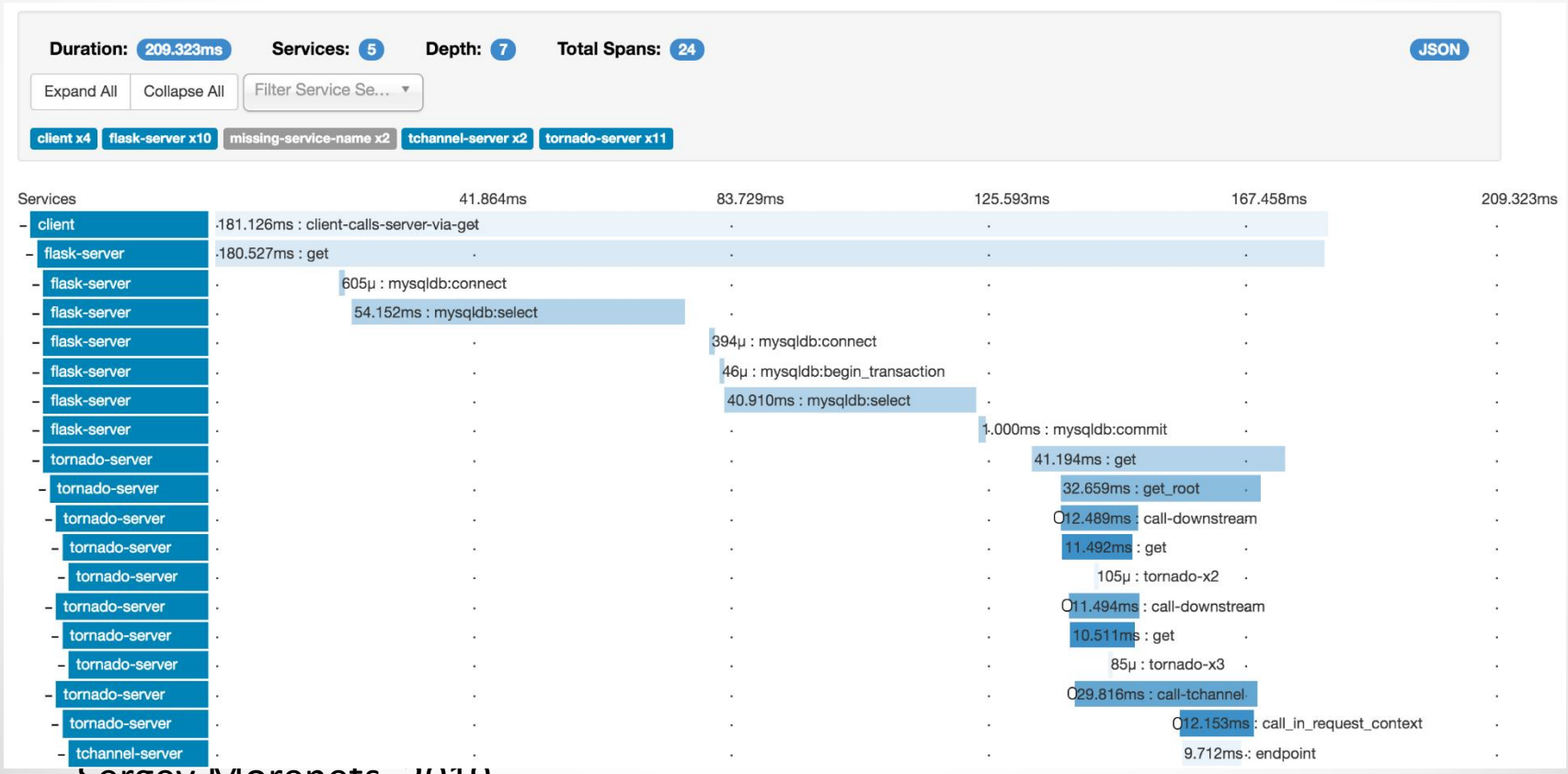
- Sergey

Zipkin UI



- Sergey Morenets, 2019

Zipkin UI


Duration: 209.323ms
Services: 5
Depth: 7
Total Spans: 24
JSON
Expand All
Collapse All
Filter Service Se...
client x4
flask-server x10
missing-service-name x2
tchannel-server x2
tornado-server x11
Services
41.864ms
83.729ms
125.593ms
167.458ms
209.323ms
client
181.126ms : client-calls-server-via-get
flask-server
180.527ms : get
flask-server
605µ : mysqldb:connect
flask-server
54.152ms : mysqldb:select
flask-server
394µ : mysqldb:connect
flask-server
46µ : mysqldb:begin_transaction
flask-server
40.910ms : mysqldb:select
flask-server
1.000ms : mysqldb:commit
tornado-server
41.194ms : get
tornado-server
32.659ms : get_root
tornado-server
12.489ms : call-downstream
tornado-server
11.492ms : get
tornado-server
105µ : tornado-x2
tornado-server
11.494ms : call-downstream
tornado-server
10.511ms : get
tornado-server
85µ : tornado-x3
tornado-server
29.816ms : call-tchannel
tornado-server
12.153ms : call_in_request_context
tchannel-server
9.712ms : endpoint

Task 15. Zipkin server

1. Start Zipkin server using Docker command: *docker run -d -p 9411:9411 openzipkin/zipkin*
2. Open URL http://localhost:9411 and review ZipkinUI

Spring Cloud Sleuth

- Development started in 2016
- Htrace/**Zipkin**/Dapper (Google project) based abstraction layer of the distributed tracing data model
- Can propagate tracing context between processes
- Each request is called **span**
- Span has 64-bit identifier and some information fields
- **Trace** also has 64-bit identifier and tree of spans
- Integrates with RestTemplate, Feight client, Zuul, Hystrix
- Provides HTTP integration with Zipkin (default) or with RabbitMQ/Kafka
- Based on top of **Brave** library starting since 2.0

Brave library

- Developed since 2013
- Distributed tracing library
- Reports data to Zipkin as spans
- Dependency-free project
- Supports tracing for JDBC/Servlets/Spring
- Java 6 (and Android) – compatible
- Alternative is OpenTracing

Spring Cloud Sleuth

Reshmi Krishna

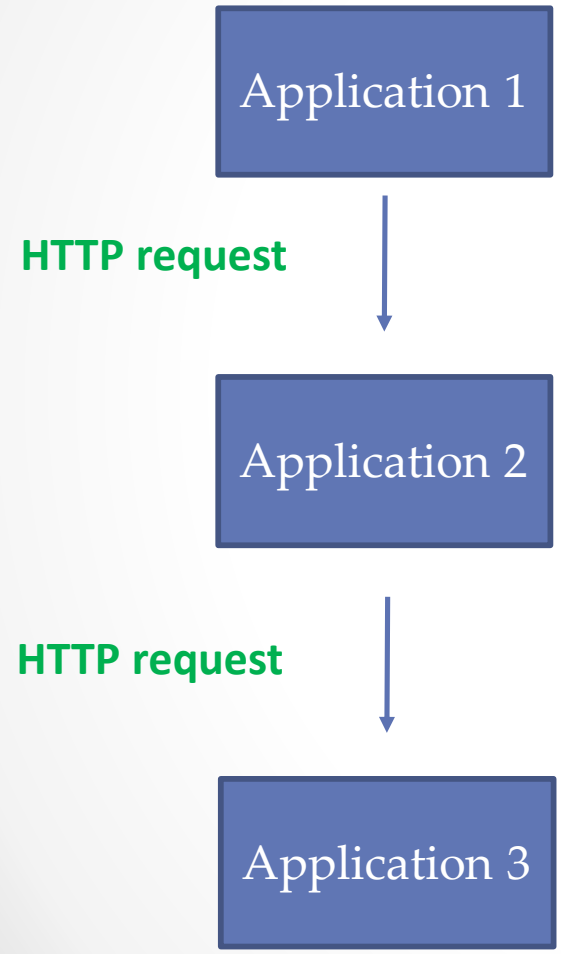
Adrian Cole

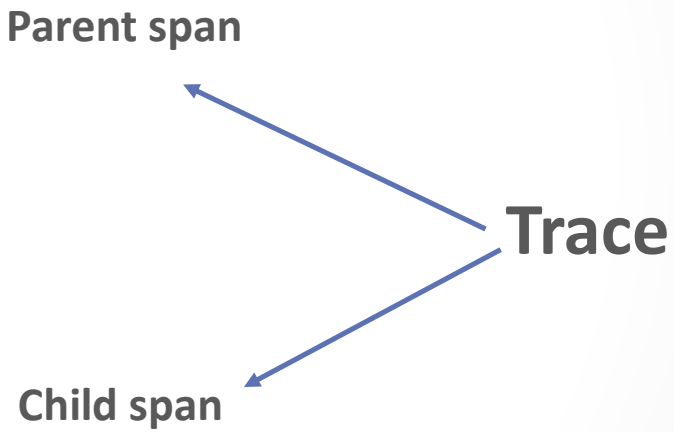
Marcin Grzejszczak

Spring Cloud Sleuth. Dependencies


```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-sleuth-zipkin</artifactId>
</dependency>
```

Spring Cloud Sleuth. Data flow





Spring Cloud Sleuth

library-logging.http:/product/value: 5.000ms

AKA: library-client,library-logging

Date Time	Relative Time	Annotation	Address
15.11.2017, 12:33:53	8.000ms	Client Send	192.168.0.103:8003 (library-client)
15.11.2017, 12:33:53	10.000ms	Server Receive	192.168.0.103:8080 (library-logging)
15.11.2017, 12:33:53	12.000ms	Server Send	192.168.0.103:8080 (library-logging)
15.11.2017, 12:33:53	13.000ms	Client Receive	192.168.0.103:8003 (library-client)

Key	Value
http.host	DESKTOP-0F6GKS0
http.method	GET
http.path	/product/value
http.url	http://DESKTOP-0F6GKS0:8080/product/value
mvc.controller.class	ProductController
mvc.controller.method	getValue
spring.instance_id	DESKTOP-0F6GKS0:library-logging
spring.instance_id	DESKTOP-0F6GKS0:library-client:8003

More Info

Spring Cloud Sleuth. Spans


```
@JsonAutoDetect(fieldVisibility = JsonAutoDetect.Visibility.ANY)
@JsonInclude(JsonInclude.Include.NON_DEFAULT)
public class Span implements SpanContext {

    public static final String SAMPLED_NAME = "X-B3-Sampled";
    public static final String PROCESS_ID_NAME = "X-Process-Id";
    public static final String PARENT_ID_NAME = "X-B3-ParentSpanId";
    public static final String TRACE_ID_NAME = "X-B3-TraceId";
    public static final String SPAN_NAME_NAME = "X-Span-Name";
    public static final String SPAN_ID_NAME = "X-B3-SpanId";
    public static final String SPAN_EXPORT_NAME = "X-Span-Export";
    public static final String SPAN_FLAGS = "X-B3-Flags";
    public static final String SPAN_BAGGAGE_HEADER_PREFIX = "baggage";
    public static final Set<String> SPAN_HEADERS = new HashSet<>(
            Arrays.asList(SAMPLED_NAME, PROCESS_ID_NAME, PARENT_ID_NAME
                    SPAN_ID_NAME, SPAN_NAME_NAME, SPAN_EXPORT_NAME));
```

Spring Cloud Sleuth. Local tracing


```
@Autowired
private Tracer tracer;

@PostMapping("/sample")
public void trace() throws Exception {
    ScopedSpan span = tracer.startScopedSpan("local");
    try {
        span.tag("name", "test");
        span.annotate("Started");

        Thread.sleep(1000);
        span.annotate("Finished");
    } catch (Throwable ex) {
        span.error(ex);
        throw new RuntimeException(ex);
    } finally {
        span.finish();
    }
}
```

Brave API

Log event and link it with current timestamp

Report an error to complete span

Spring Cloud Sleuth. Custom tracing

library-logging.sample: 6.022s

AKA: library-logging

Date Time	Relative Time	Annotation	Address
15.11.2017, 16:24:32	3.614s	Started	192.168.0.103:8080 (library-logging)
15.11.2017, 16:24:35	5.814s	Finished	192.168.0.103:8080 (library-logging)

Key	Value
Local Component	unknown
name	test
Local Address	192.168.0.103:8080 (library-logging)

More Info

- Sergey Morenets, 2019

Spring Cloud Sleuth. Nested spans


```
public void nestedSpan() throws Exception {
    Span parent = tracer.newTrace();
    try {
        parent.name("Main").start();
        parent.tag("type", "parent");

        Span child = tracer.nextSpan();
        child.tag("type", "child");
        Thread.sleep(500);
        child.finish();

        Thread.sleep(1000);
    } catch (Throwable ex) {
        parent.error(ex);
        throw new RuntimeException(ex);
    } finally {
        parent.finish();
    }
}
```

Create parent span

Create child span

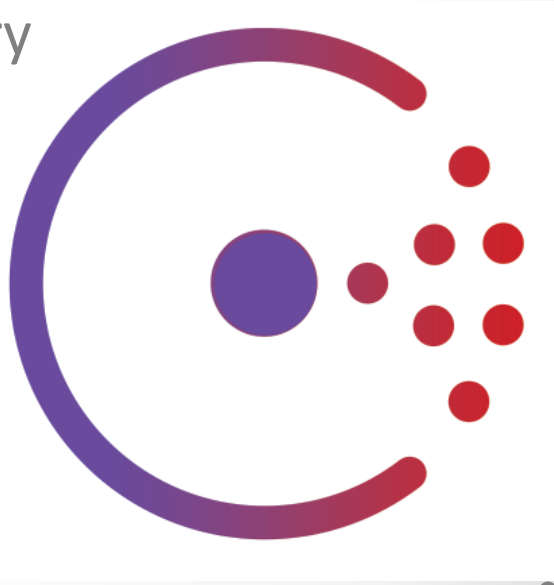
Task 16. Spring Cloud Sleuth

1. Add Spring Cloud Sleuth dependencies to the applications where you want to gather traces
2. Run scenario where microservices call each other using REST endpoints. Open Zipkin UI and review collected spans/metrics.
3. Try to implement custom tracing using Tracer object.

Consul

- Distributed highly-available tool for service discovery, configuration and orchestration
- Each Consul Agent can be run in server or client mode inside Consul cluster
- Applications communicate with Consul via DNS or HTTP API
- Doesn't require app support for discovery
- Developed by HashiCorp in 2012
- Based on Serf and uses Gossip

and Raft (consensus protocol)



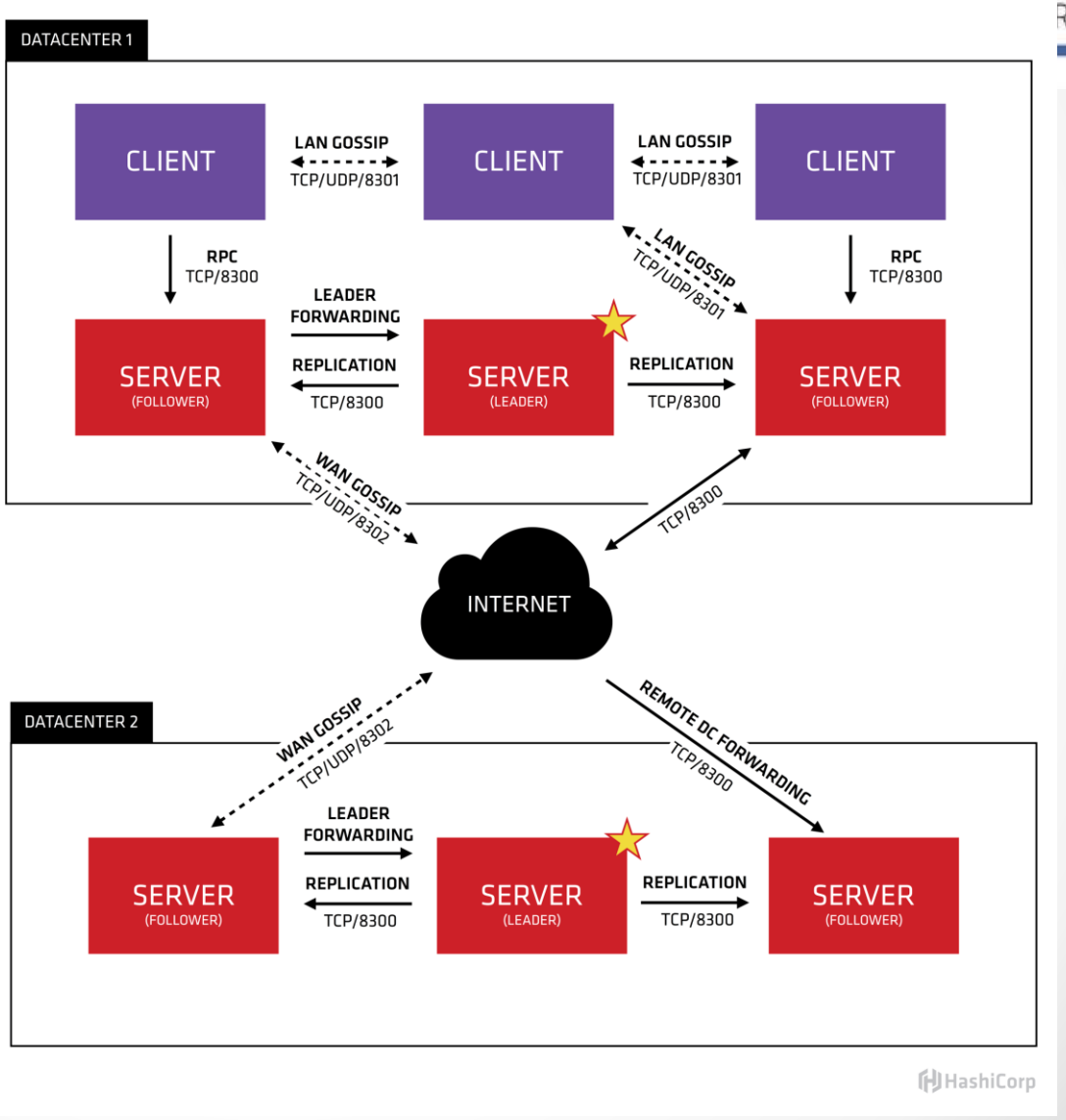
Consul ports

- HTTP API(default 8500)
- DNS queries (8600)
- Server address (8300)
- Server LAN port (8301)
- Server WAN port (8302)

Consul. Cluster membership

- Agents (nodes) are responsible for registering services, maintains system state, perform health-checks and handle queries
- Server agents stores cluster state, handle queries
- Client agents are lightweight/stateless and forward RPC request to the server agents
- Servers agents replicate information within each other
- 5 server agents per datacenter are recommended

Consul. Cluster membership



- Sergey Morenets, 2019

Spring Cloud Consul

- Spring Cloud uses HTTP API for service registration/discovery
- By default Consul hits /actuator/health service endpoint each 10 seconds (using TCP, HTTP or Nagios scripts)
- Zuul, Ribbon and Control Bus are integrated
- Consul doesn't require applications to be aware of itself
- Provides strong consistency whereas Eureka provides better availability
- Spring Cloud Consul allows to register service and discover other services

Consul vs Eureka

- One Eureka server per availability zone
- Eureka servers replicate service connections
- Eureka requires heart-beats from clients
- Eureka is part of NetFlix platform
- Consul offers strong consistency by using Raft protocol
- Consul offers HTTP/TCP for health checking
- Client nodes in Consul distribute health checking by using Gossip protocol
- Consul expects 3-5 servers per DataCenter and can support DataCenter-per-DataCenter communication

Consul. Build management


```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-consul-dependencies</artifactId>
            <version>${spring.cloud.consul.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

```
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-consul-all</artifactId>
    </dependency>
</dependencies>
```

Spring Cloud Consul. Bootstrap


```yaml
spring:
  cloud:
    consul:
      host: localhost
      port: 8500
```

Consul server info

application.yml

```java
@SpringBootApplication
@EnableDiscoveryClient
public class ConsulApplication {
    public static void main(String[] args) {
        SpringApplication.run(ConsulApplication.class, args);
    }
}
```

Spring Cloud Consul. Customization


```yaml
spring:
  cloud:
    consul:
      host: localhost
      port: 8500
      discovery:
        health-check-url: /app/health
        health-check-critical-timeout: 5m
        health-check-interval: 30s
        enabled: true
        hostname: ${HOST}
        health-check-timeout: 1m
        port: 9000
        prefer-ip-address: true
```

Overrides host/port from Instance id

application.yml

Task 17. Spring Cloud Consul

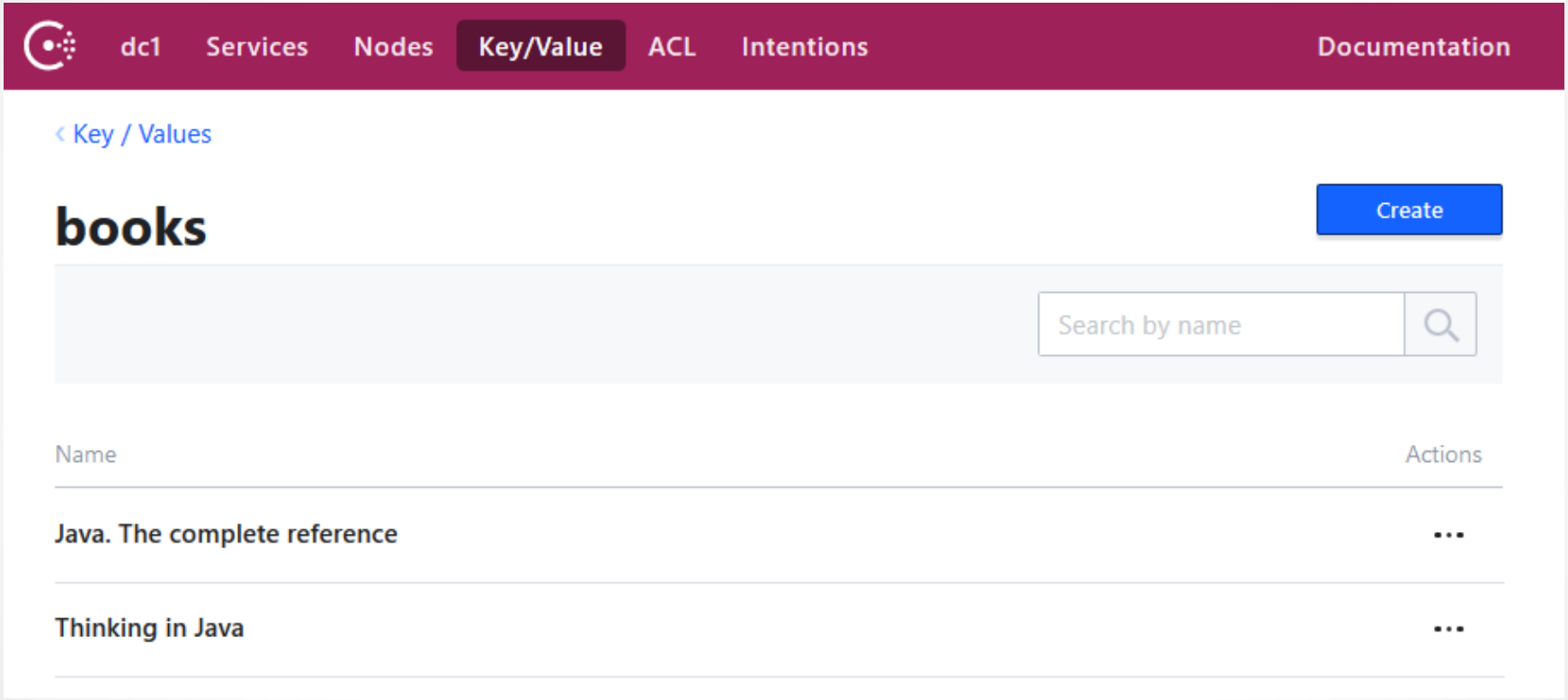
1. Run Consul in the development mode
2. Confirm that Consul server started properly
3. Review Spring Consul application and run it. Review application log and verify that application connected to Consul. Review Consul log
4. Write REST-service that returns services/instances using **DiscoveryClient** and confirm that your application is discovered by this API.

Distributed configuration with Consul

- Consul provides hierarchical **key/value** store
- Spring Cloud Consul Config is similar to Spring Cloud Config
- Configuration is stored in /config folder by default

config/may-app,dev/
config/my-app/ ← **Available to my-app service only**
config/application,dev/
config/application/ ← **Available to all services**

Consul Web UI



Consul Cloud Consul Config

bootstrap.yml

```yaml
spring:
  cloud:
    consul:
      config:
        enabled: true
        prefix: configuration
        defaultContext: my-app
        profileSeparator: '::'
```

```xml
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-consul-config</artifactId>
</dependency>
```

Task 18. Consul key-value store

1. Run Consul agent in UI mode: *consul agent -server -bootstrap-expect 1 -data-dir c:\Consul\Data -ui*
2. Open URL http://localhost:8500/ui and review contents
3. Add new property **config/application/library** and set its value to **Sample**.
4. Check via Consul command line that property was saved: *consul kv get config/application/library*

Vault

- Externalized configuration management
- Supports encryption and db integration
- Provides HTTP API
- Developed by HashiCorp

Vault. Secret backends

- ✓ AWS
- ✓ Cassandra
- ✓ Consul
- ✓ MSSQL/Postgres/MySQL
- ✓ RabbitMQ

Vault. Authentication methods

- ✓ Username/password
- ✓ LDAP
- ✓ Token
- ✓ OAuth2
- ✓ Certificates

Task 19. Spring Cloud Vault

1. Download and unzip vault: https://www.vaultproject.io/downloads.html
2. Run **vault status** command to make sure it works properly
3. Run Vault in the development mode: **vault server -dev**
4. Export environment variable **VAULT_ADDR** as mentioned in the log.
5. Create new property: *vault write secret/vault-app library=local*

Centralized logging

- ✓ Elastic log
- ✓ Logstash
- ✓ Splunk
- ✓ Kibana
- ✓ Graphite
- ✓ Fluentd
- ✓ Filebeat

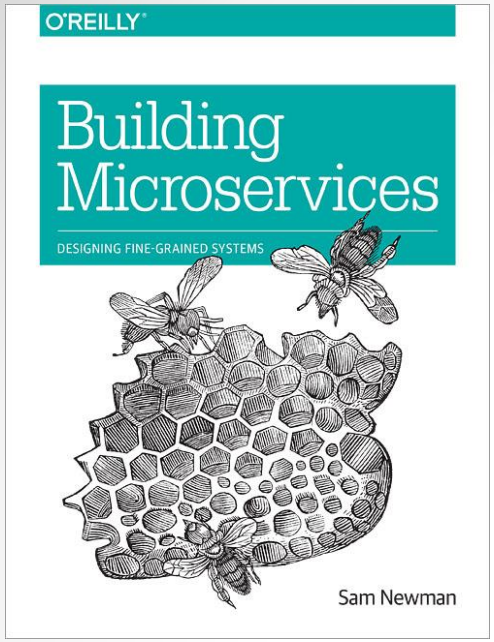
Monitoring & logging (SAAS)

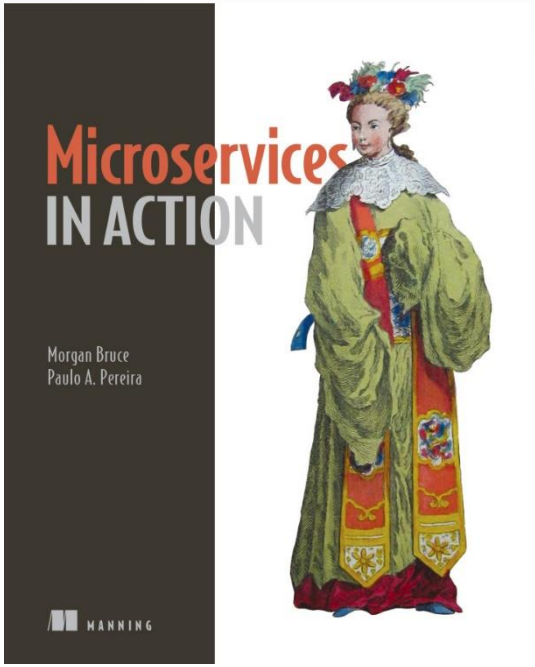
Hoverfly. Distributed testing

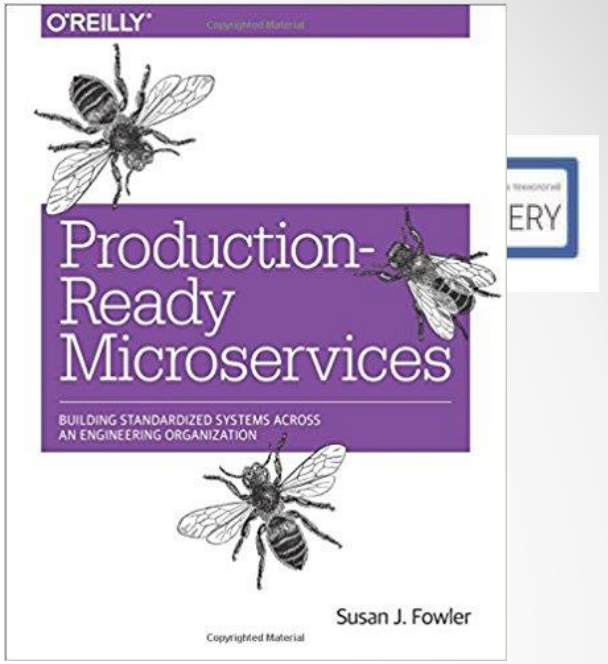
- ✓ Realistic API simulation (network failure, latency)
- ✓ Flexible customization with any programming language
- ✓ Export/share/edit API simulations
- ✓ Lightweight/ high-performance

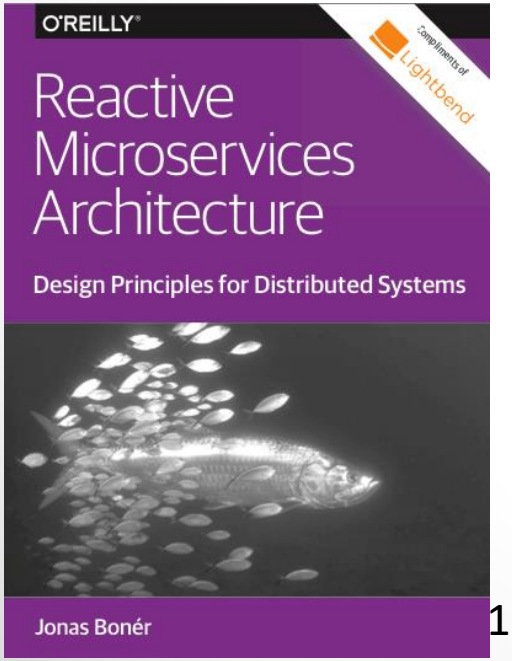
Hoverfly

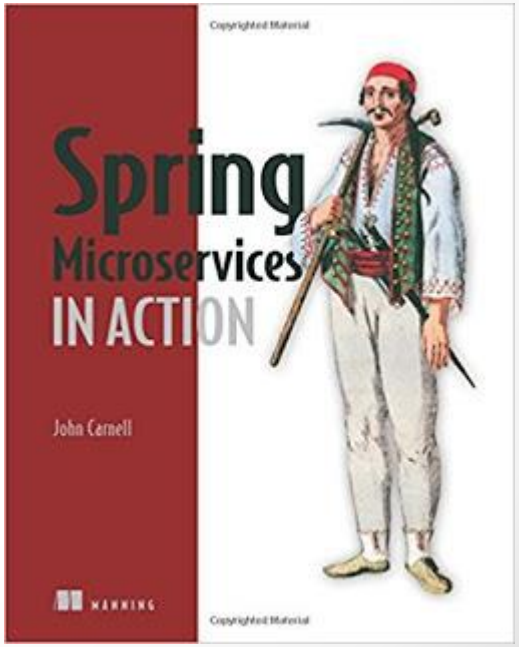
Sergey Morenets, 2019


O'REILLY
Building Microservices
DESIGNING FINE-GRAINED SYSTEMS
Sam Newman


Microservices IN ACTION
Morgan Bruce
Paulo A. Pereira
MANNING


O'REILLY
Production-Ready Microservices
BUILDING STANDARDIZED SYSTEMS ACROSS AN ENGINEERING ORGANIZATION
Susan J. Fowler
Copyrighted Material


O'REILLY
Compliments of Lightbend
Reactive Microservices Architecture
Design Principles for Distributed Systems
Jonas Bonér


Copyrighted Material
Spring Microservices IN ACTION
John Carnell
MANNING

✓ Sergey Morenets, sergey.morenets@gmail.com