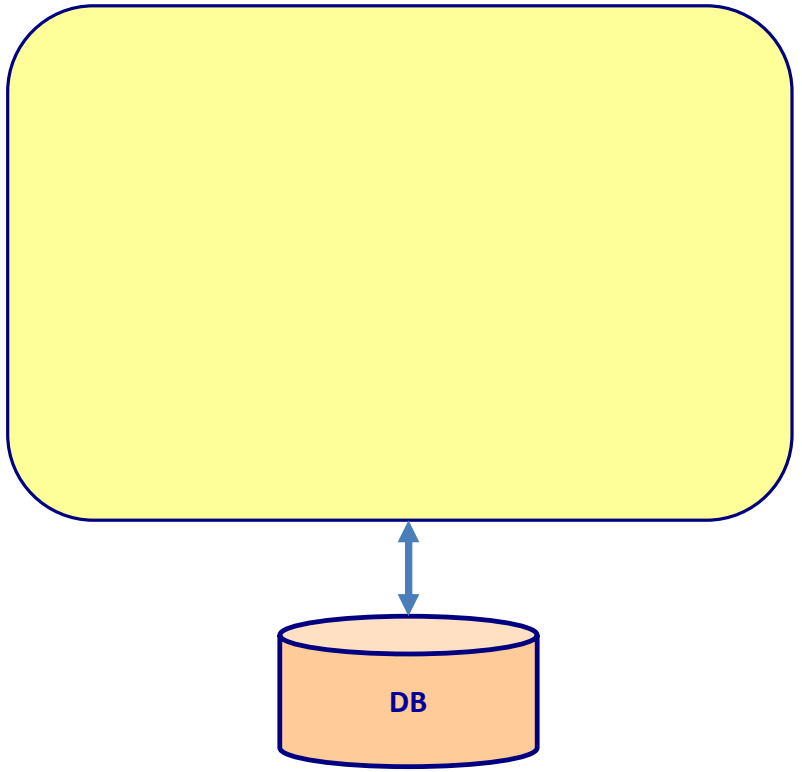
Lesson 7

MICROSERVICES

MONOLITH ARCHITECTURE

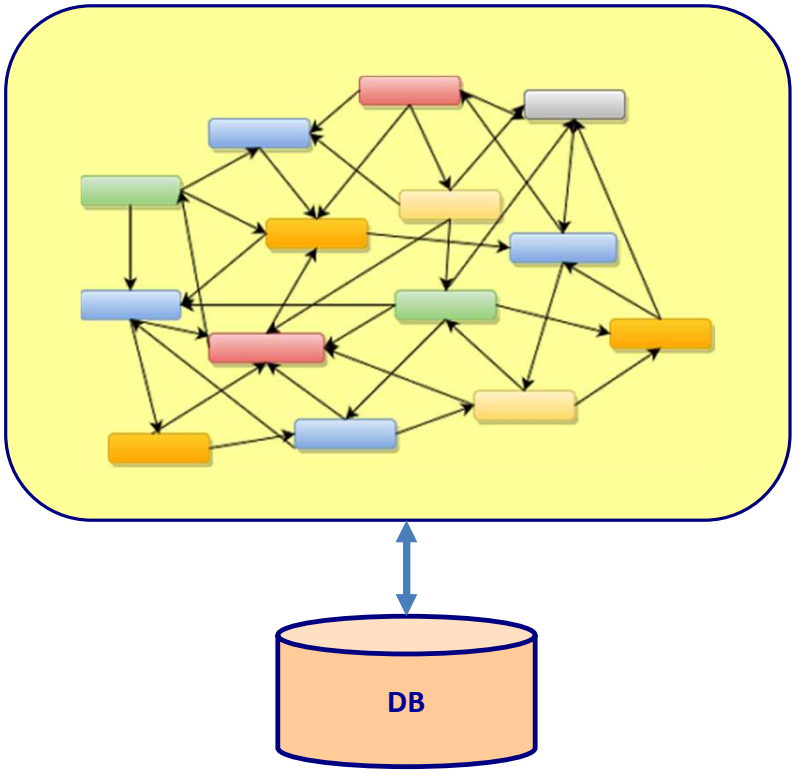
Monolith architecture

- Everything is implemented in one large system



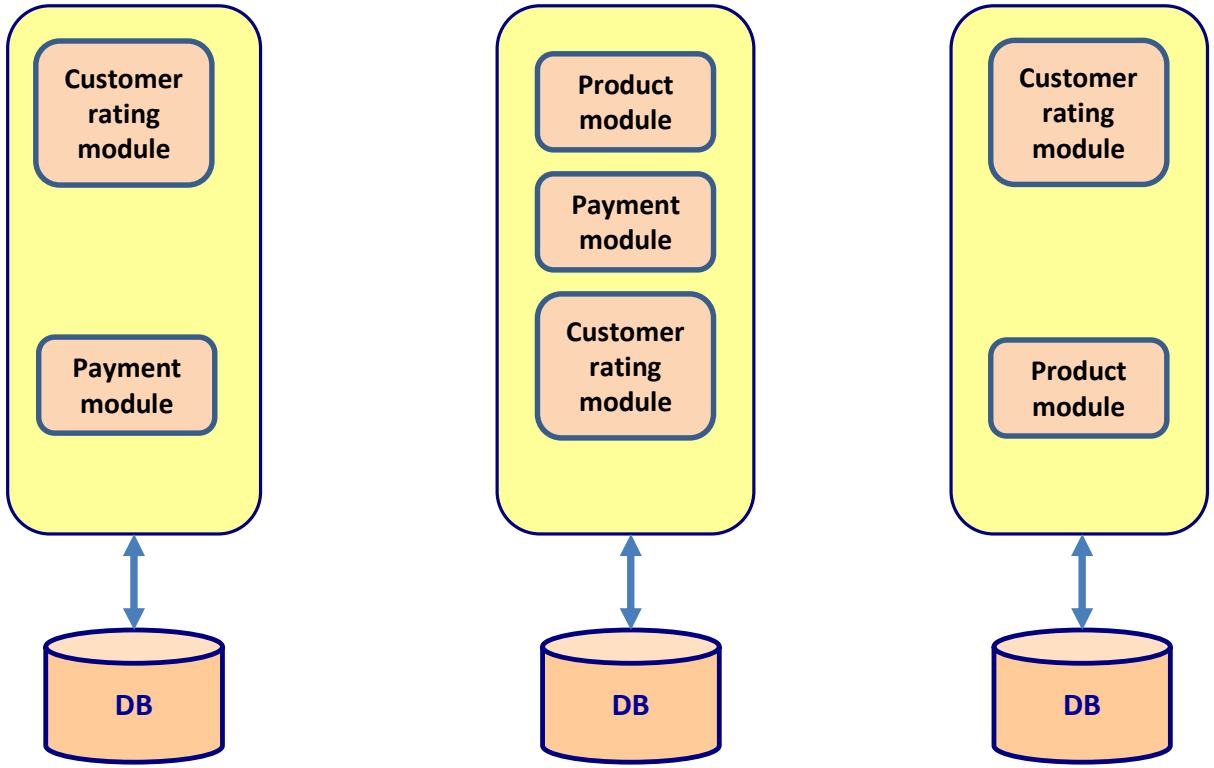
Monolith architecture

- Can evolve in a big ball of mud
  - Large complex system
    - Hard to understand
    - Hard to change



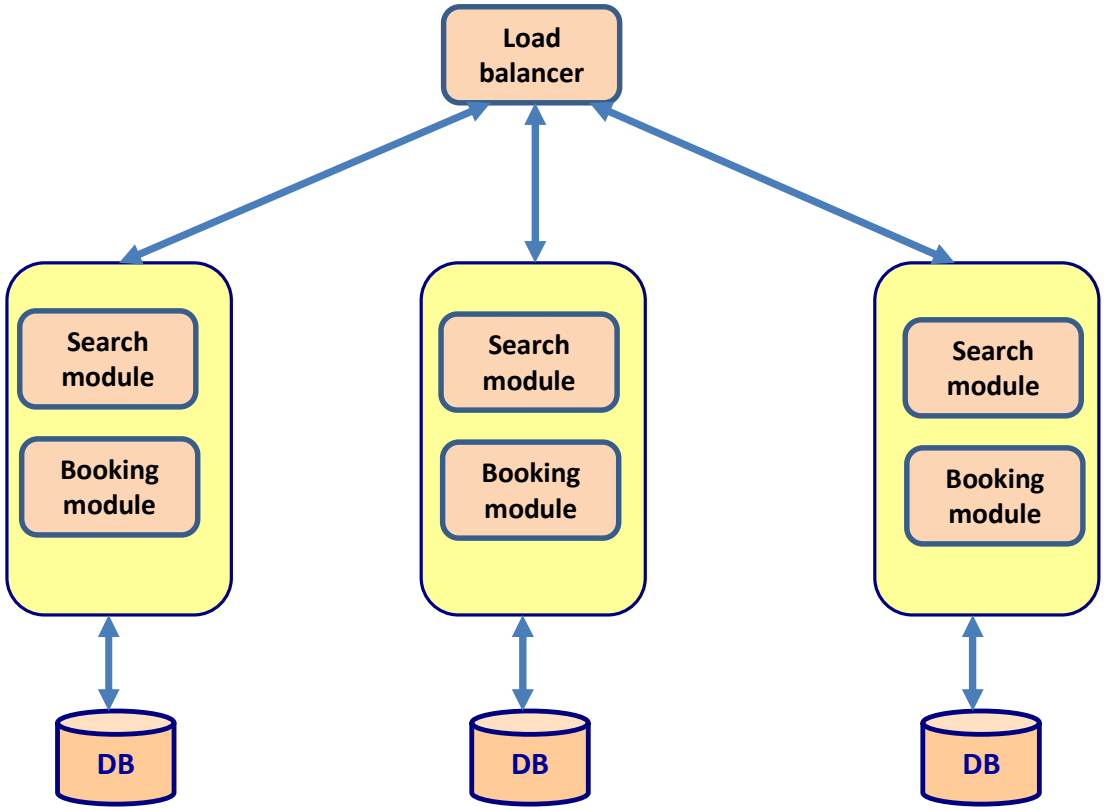
Monolith architecture

- Limited re-use is realized across monolithic applications



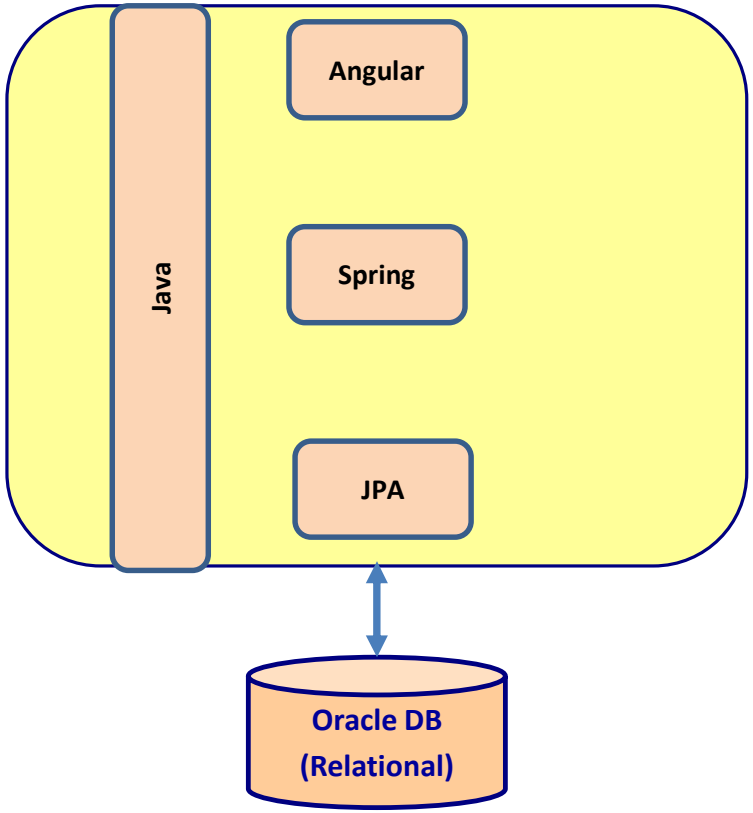
Monolith architecture

- All or nothing scaling
  - Difficult to scale separate parts



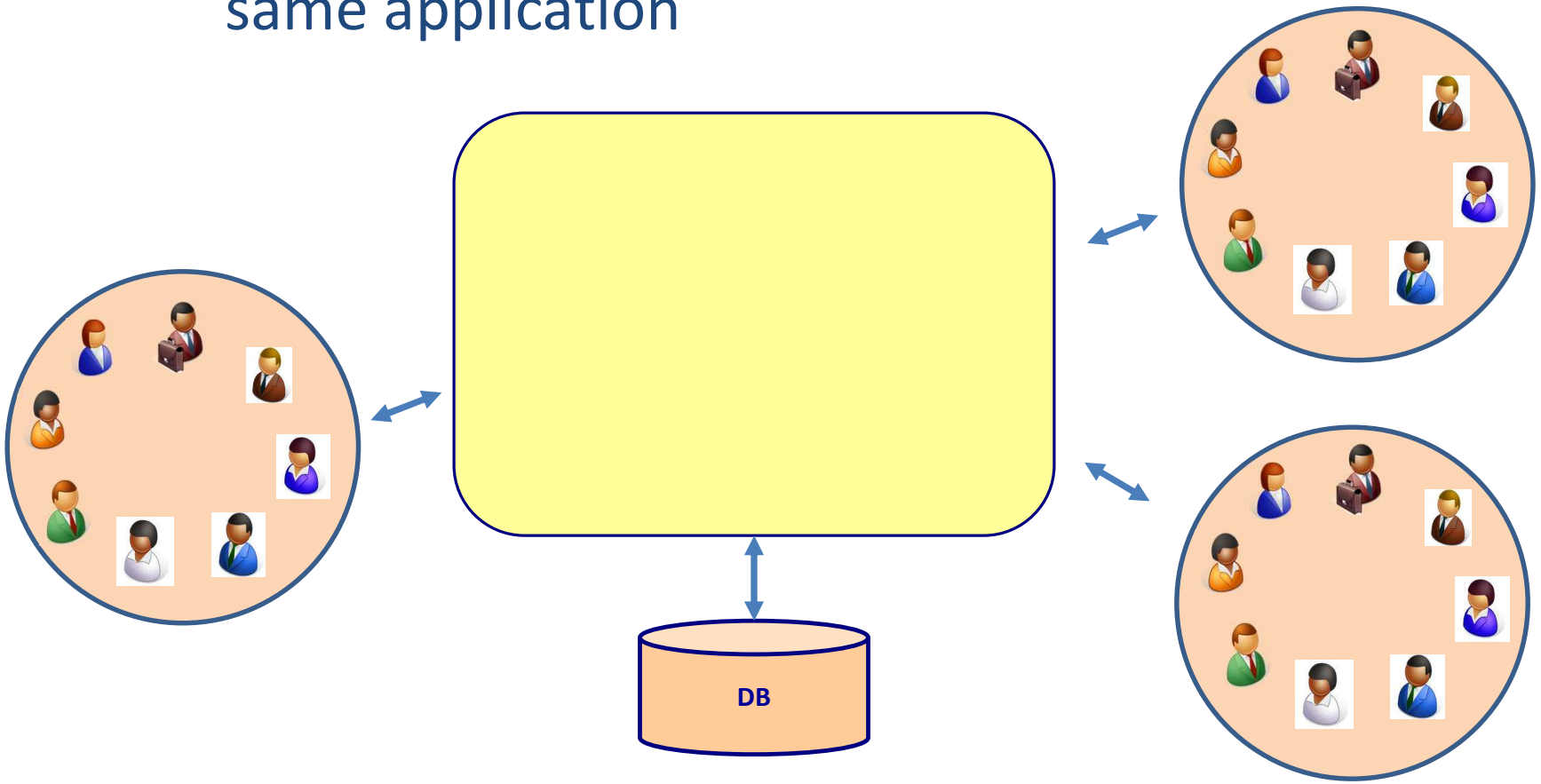
Monolith architecture

- Single development stack
  - Hard to use “the right tool for the job.”



Monolith architecture

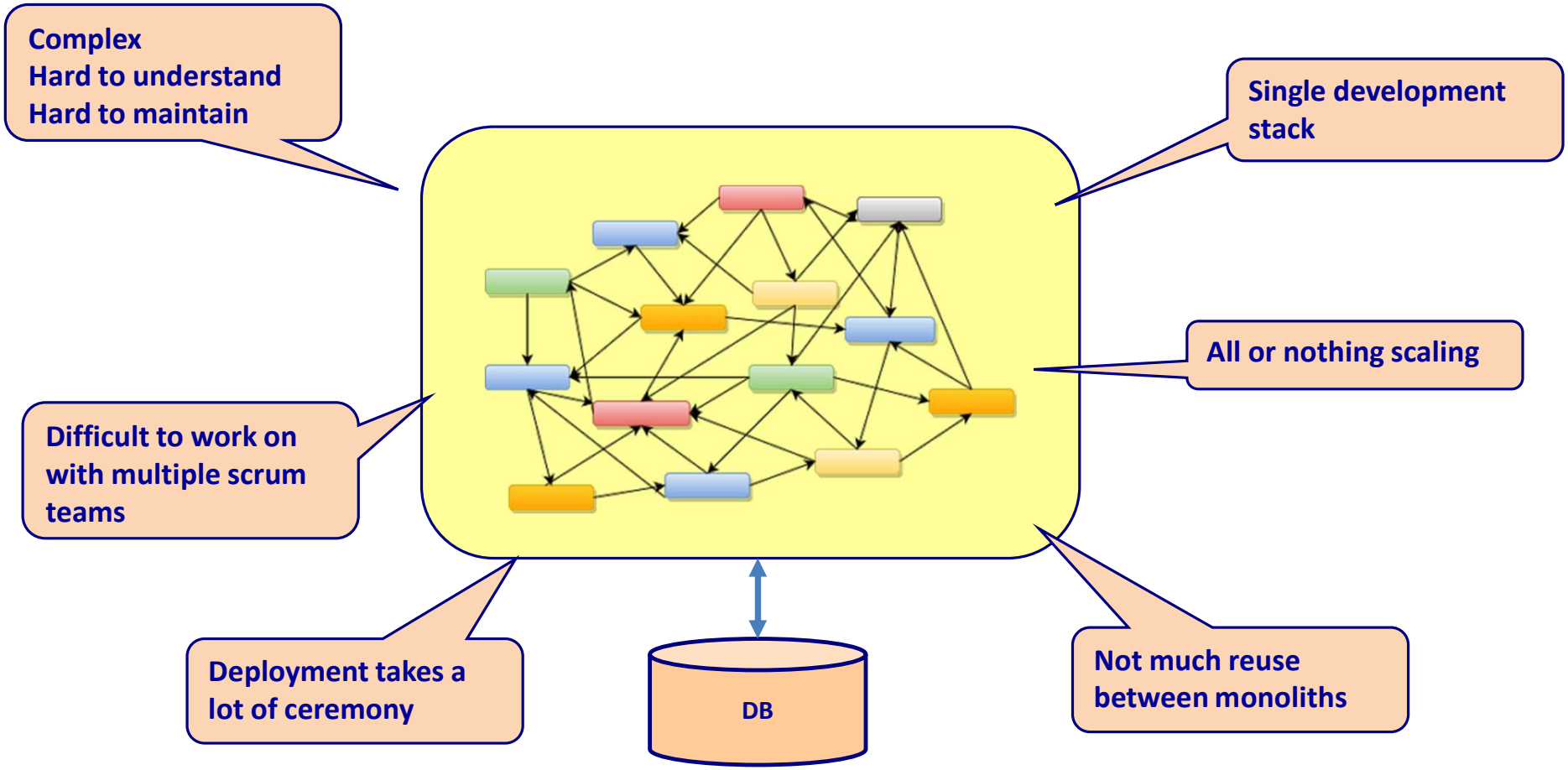
- Does no support small agile scrum teams
  - Hard to have different agile teams work on the same application



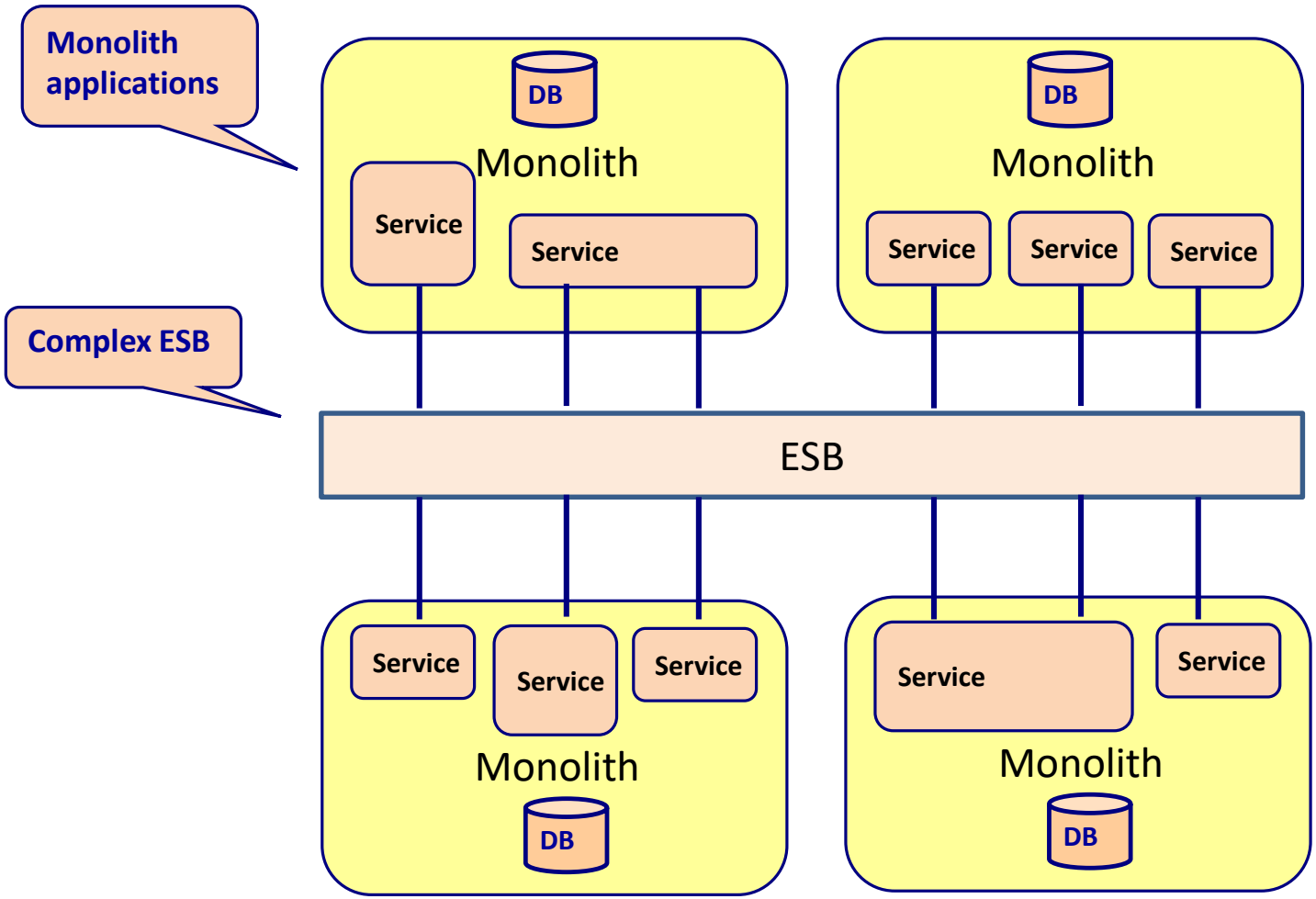
Monolith architecture

- Deploying a monolith takes a lot of ceremony
  - Every deployment is of high risk
  - I cannot deploy very frequently
  - Long build-test-release cycles

Problems with a monolith architecture



Problems with SOA



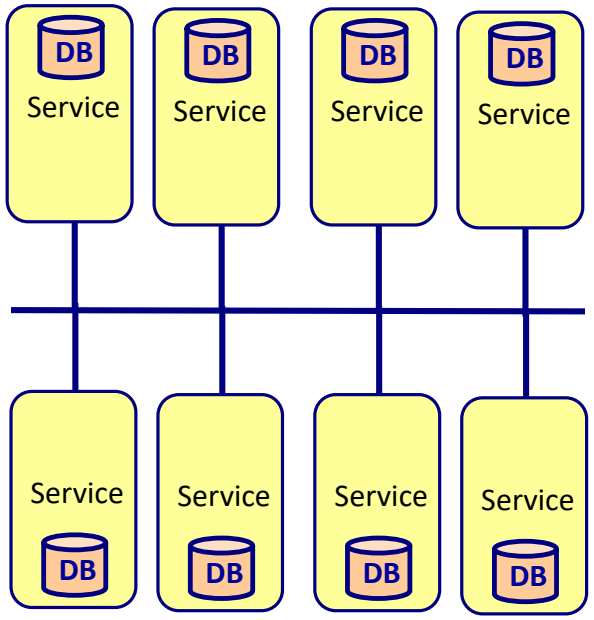
Microservice early adopters

- Netflix
- Uber
- Airbnb
- Orbiz
- eBay
- Amazon
- Twitter
- Nike

Common problem:
How to migrate from a monolith to more scalability, process automation, manageability,…

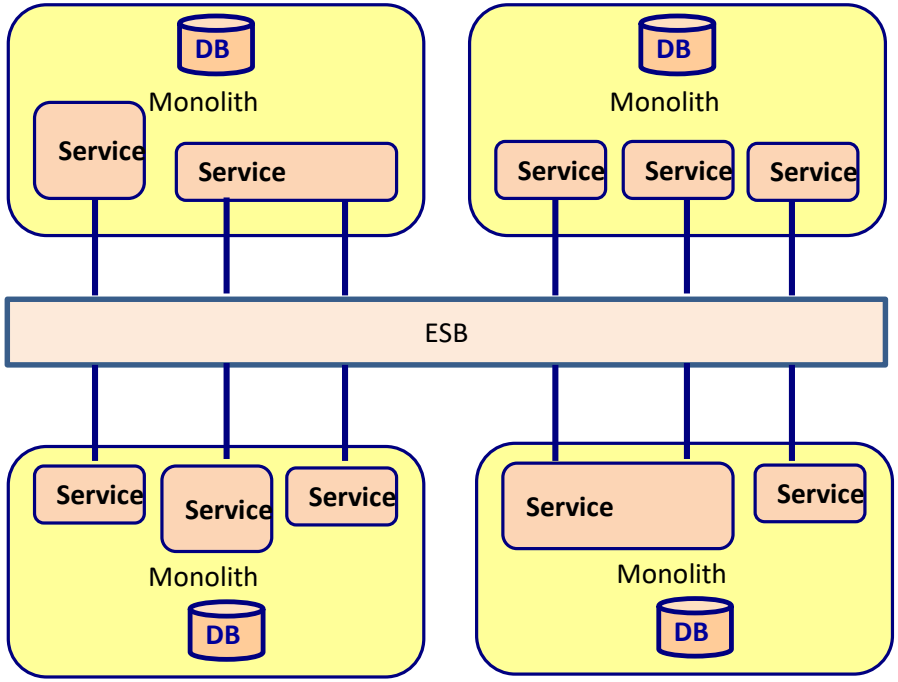
Microservices

- Small independent services
  - Simple and lightweight
  - Runs in an independent process
  - Language agnostic
  - Decoupled

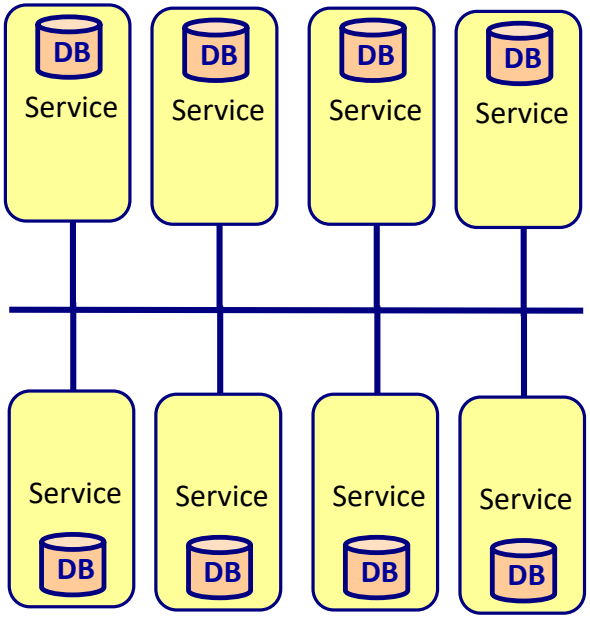


SOA vs Microservice

SOA



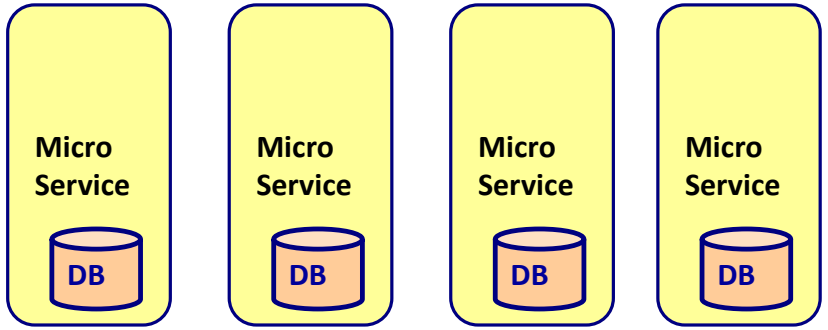
Microservice



CHARACTERISTICS OF A MICROSERVICE

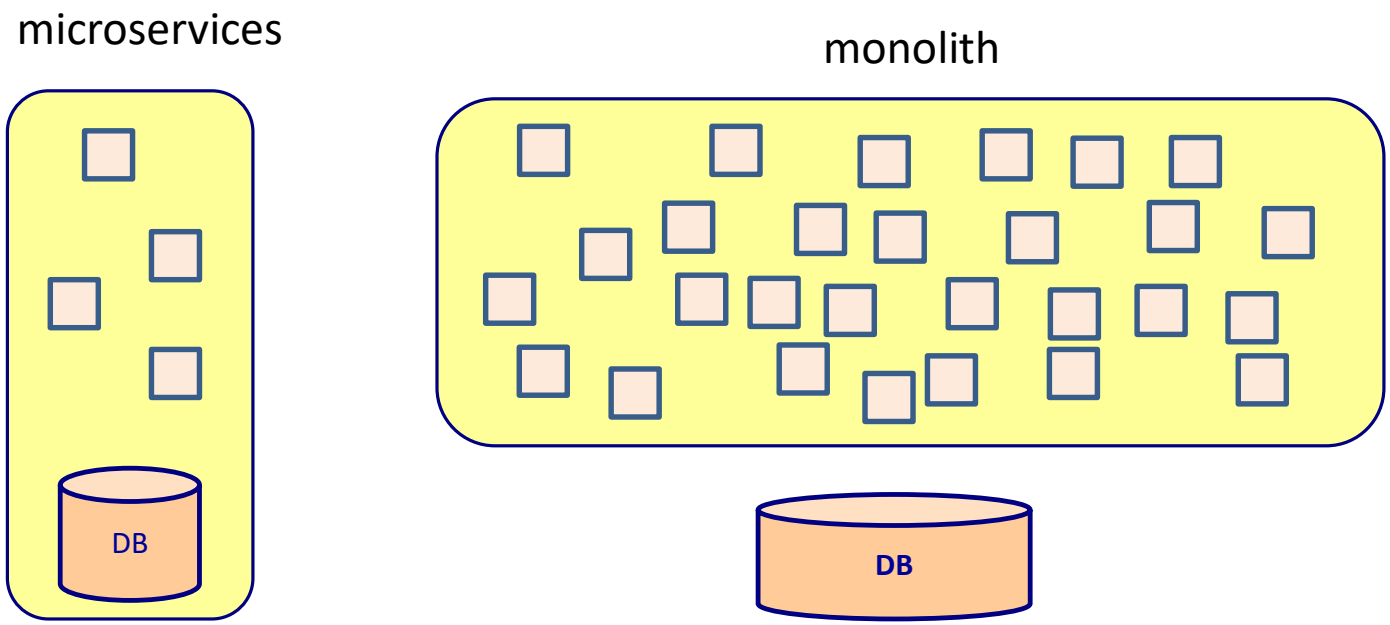
Microservices

- Small independent services
  - Simple and lightweight
  - Runs in an independent process
  - Technology agnostic
  - Decoupled

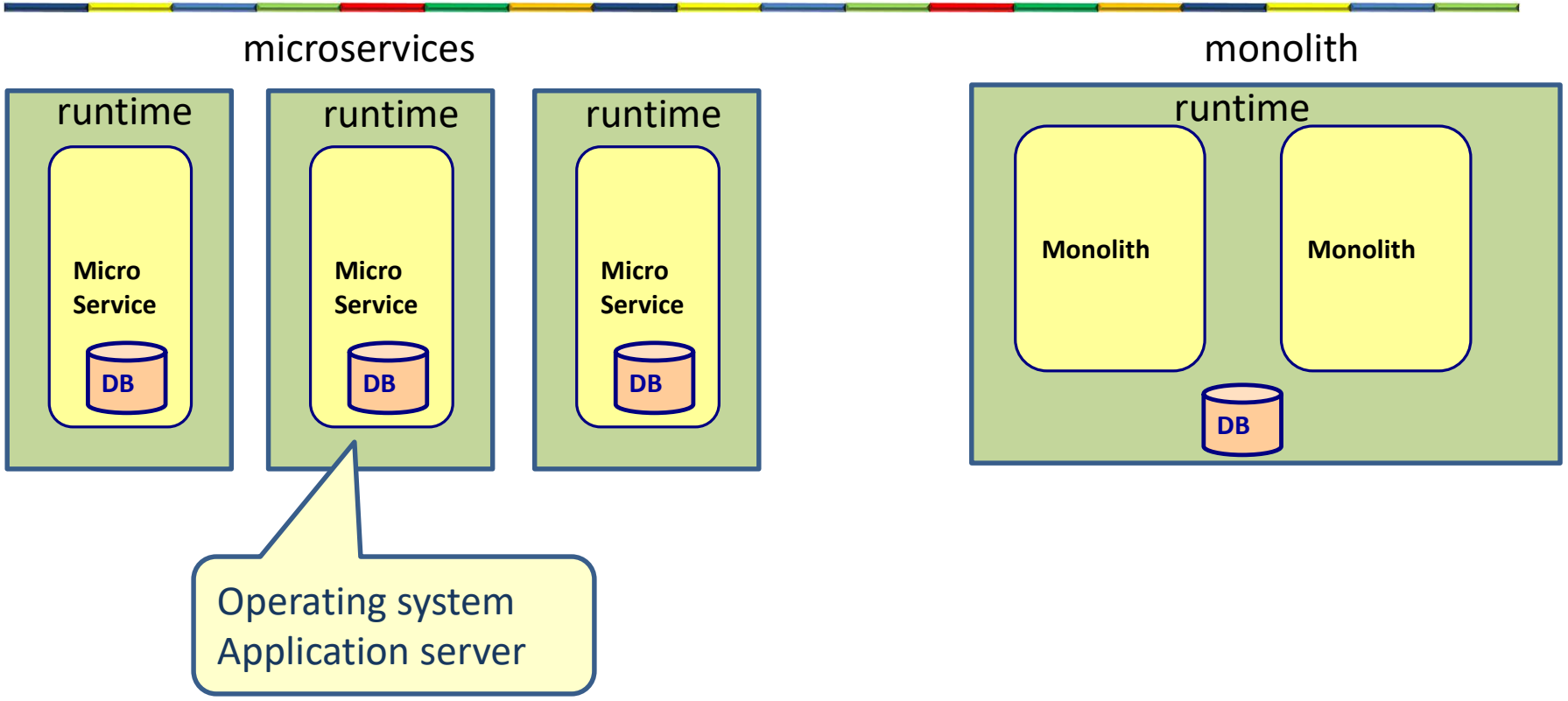


Simple and lightweight

- Small and simple
- Can be build and maintained by 1 agile team



Runs in an independent process



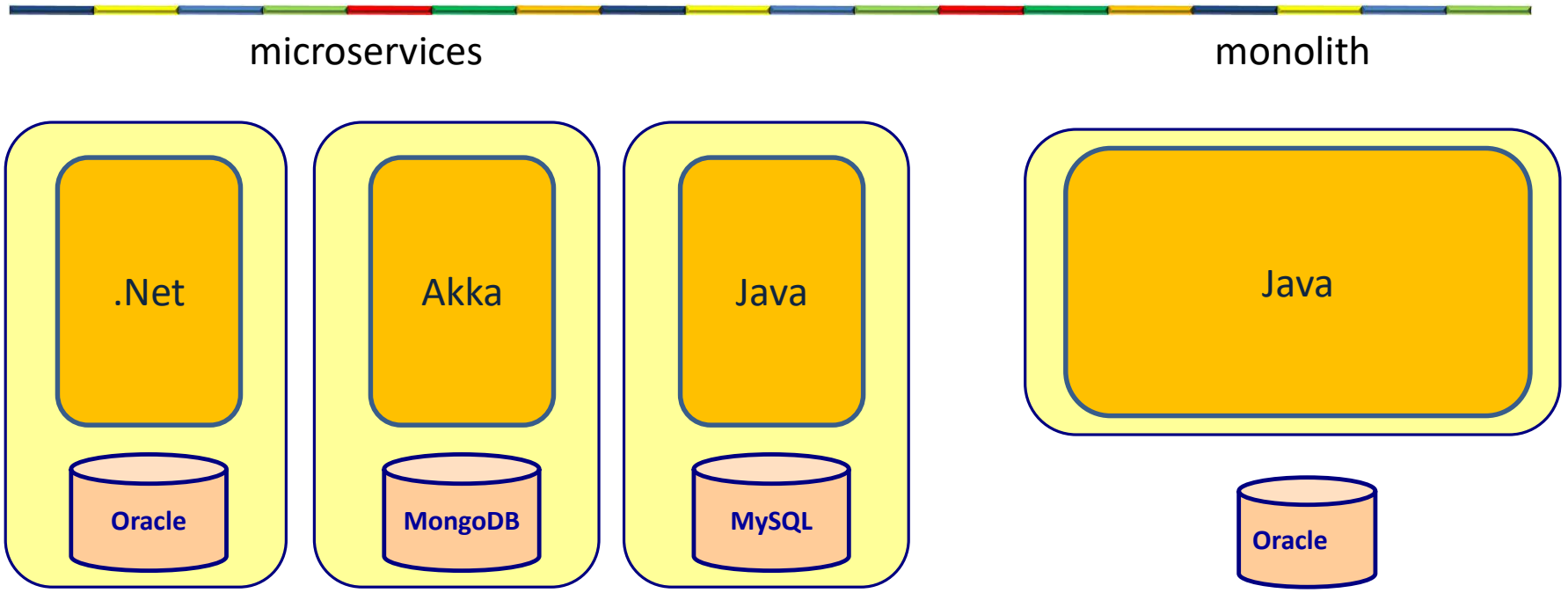
Advantages

- Runtime can be small
  - Only add what you need
- Runtime can be optimized
- Runtime can start and stop fast
- If runtime goes down, other services will still run

Disadvantages

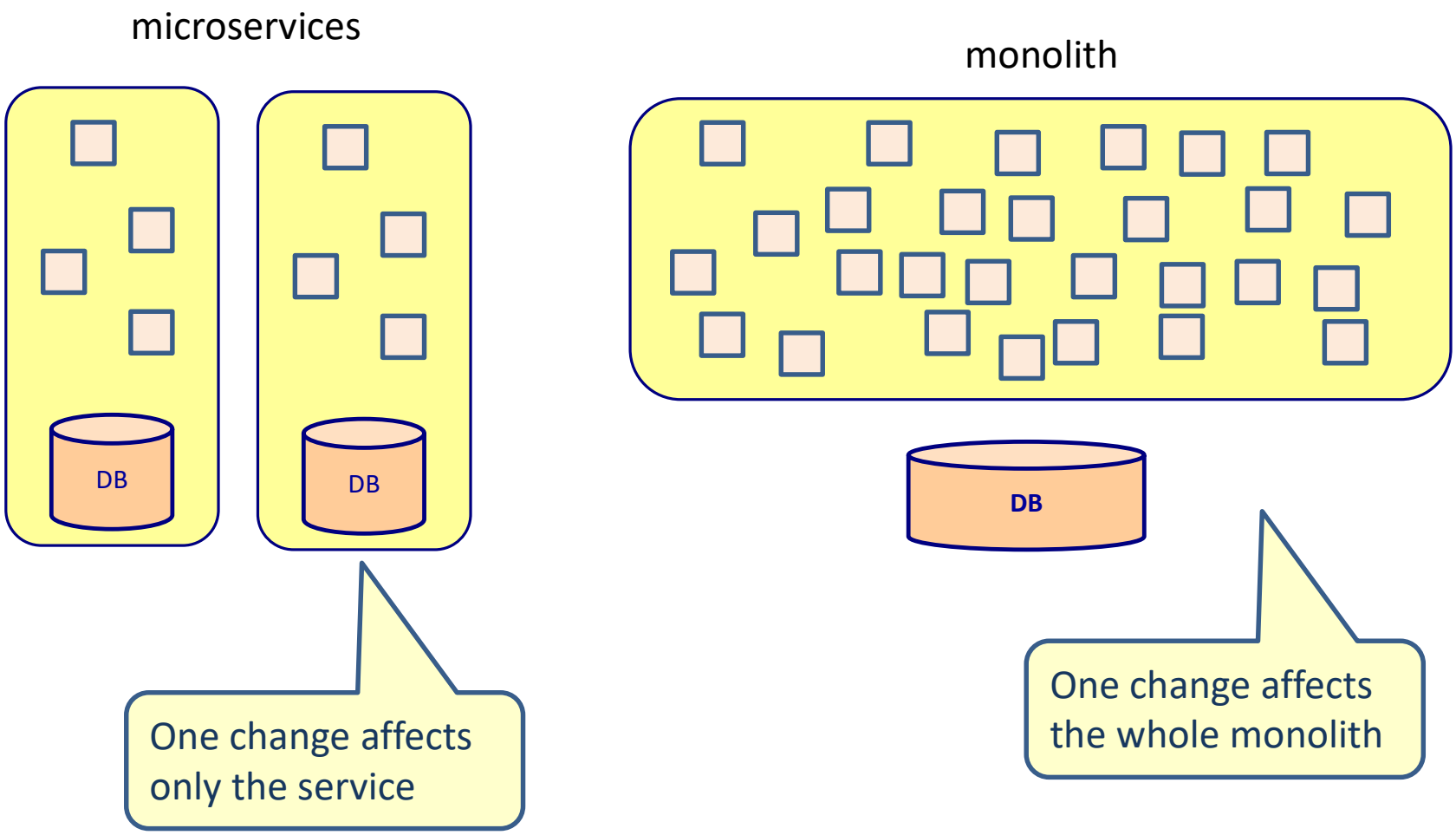
- We need to manage many runtimes

Technology agnostic

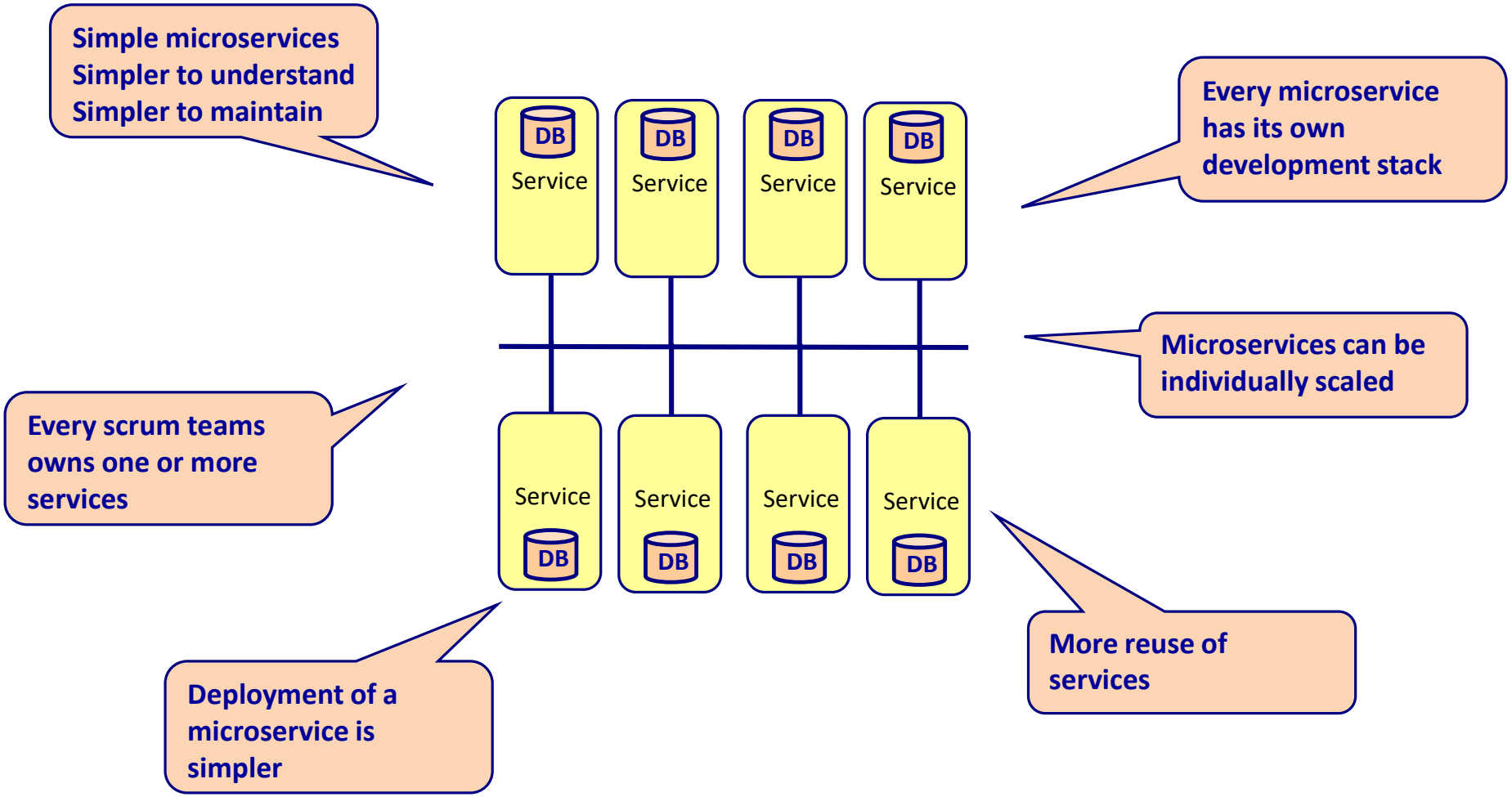


- Use the architecture and technologies that fits the best for this particular microservice

Decoupled

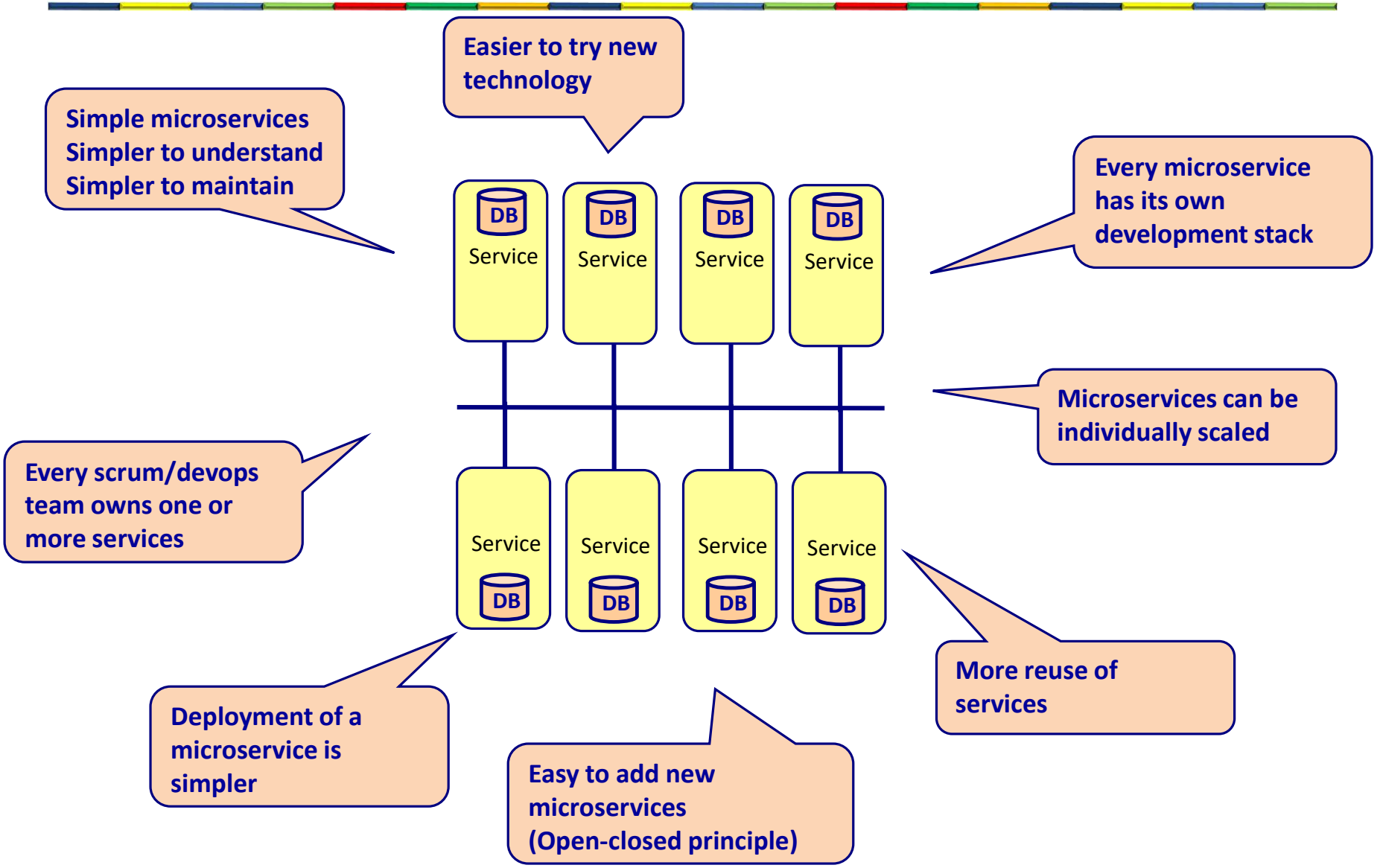


Microservice architecture

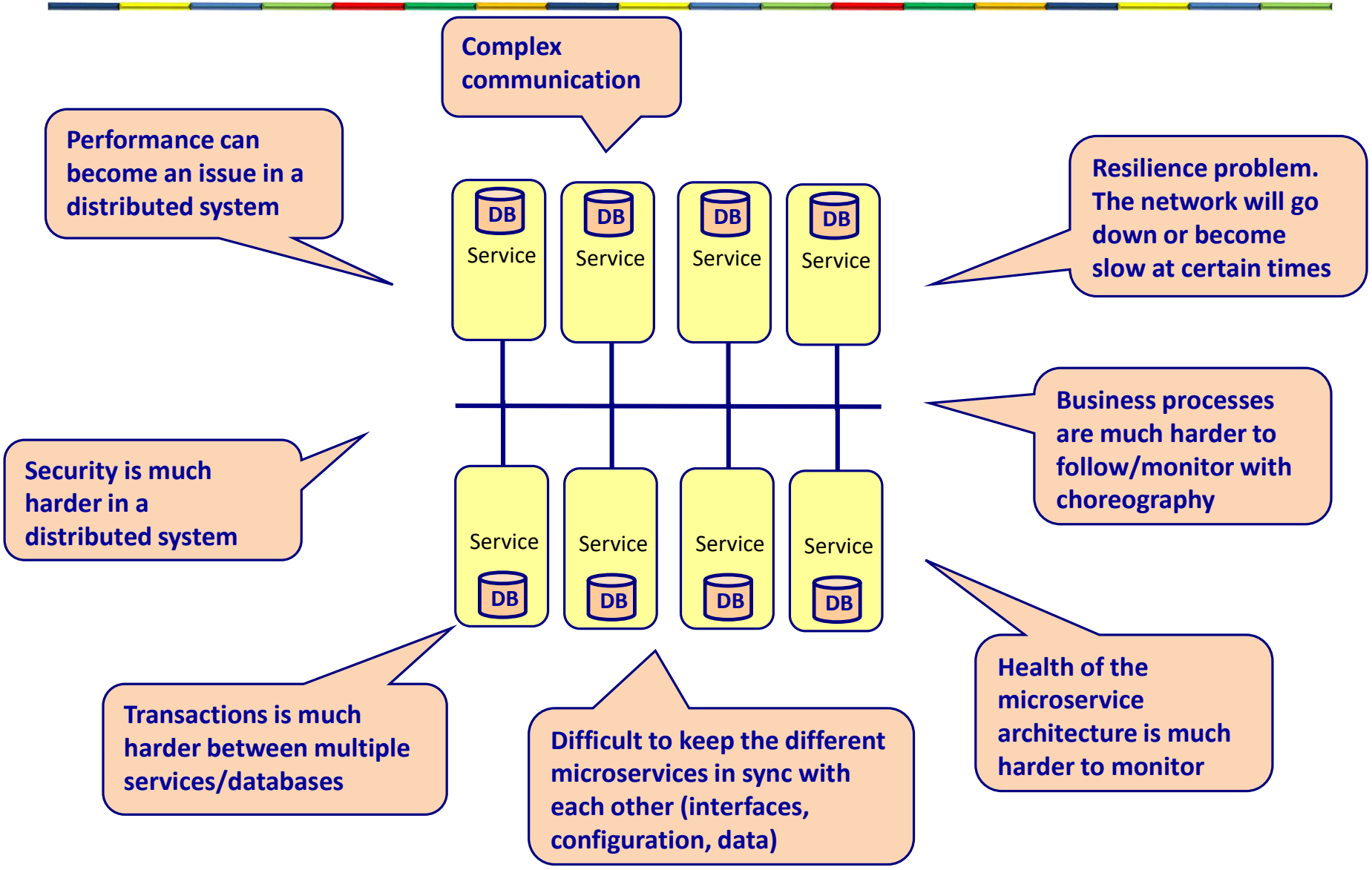


ADVANTAGES AND DISADVANTAGES OF A MICROSERVICE ARCHITECTURE

Advantages



Disadvantages

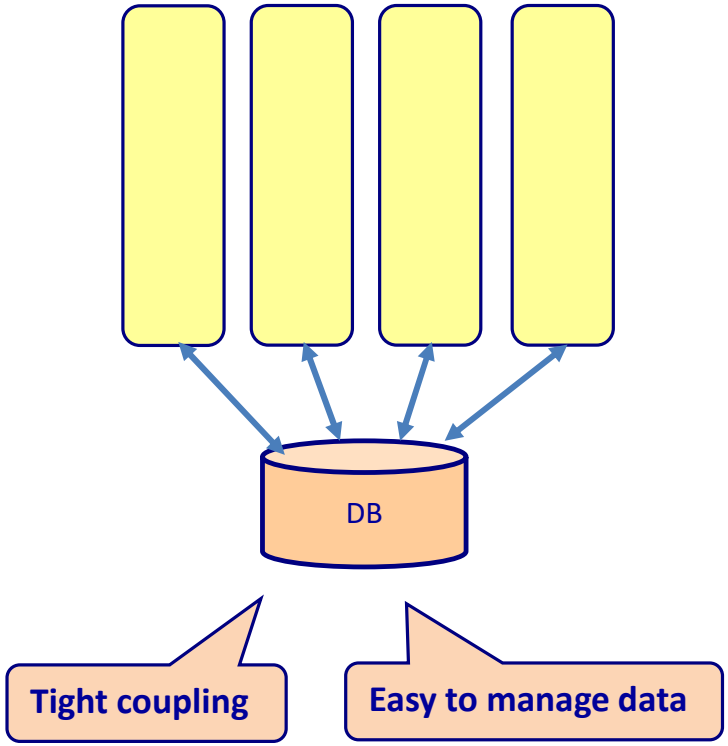


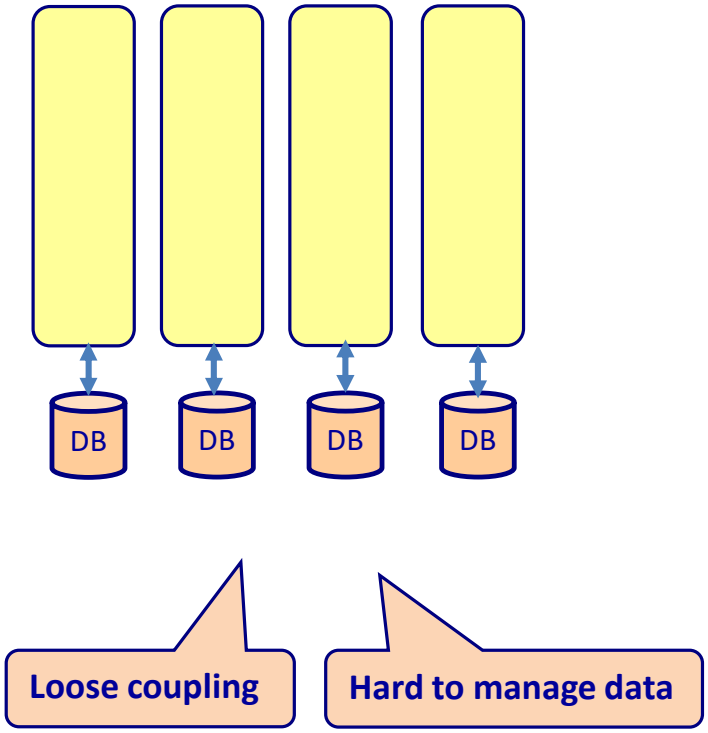
Challenges of a microservice architecture

Challenge	Solution
Complex communication	
Performance	
Resilience	
Security	
Transactions	
Following the process	
Keep data in sync	
Keep interfaces in sync	
Keep configuration in sync	
Monitor health of microservices	
Follow/monitor business processes	

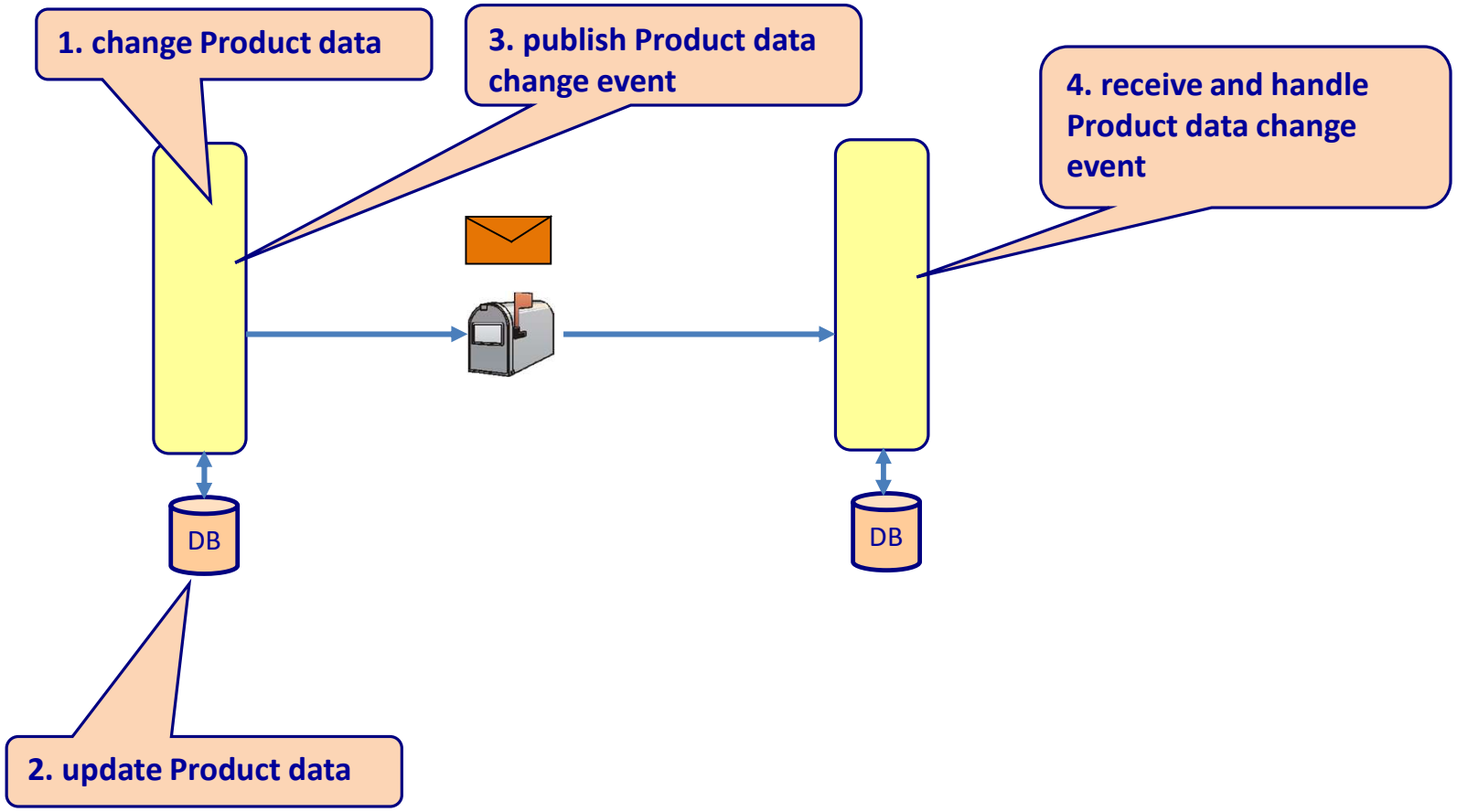
MICROSERVICE AND DATABASES

Every service manages its own data



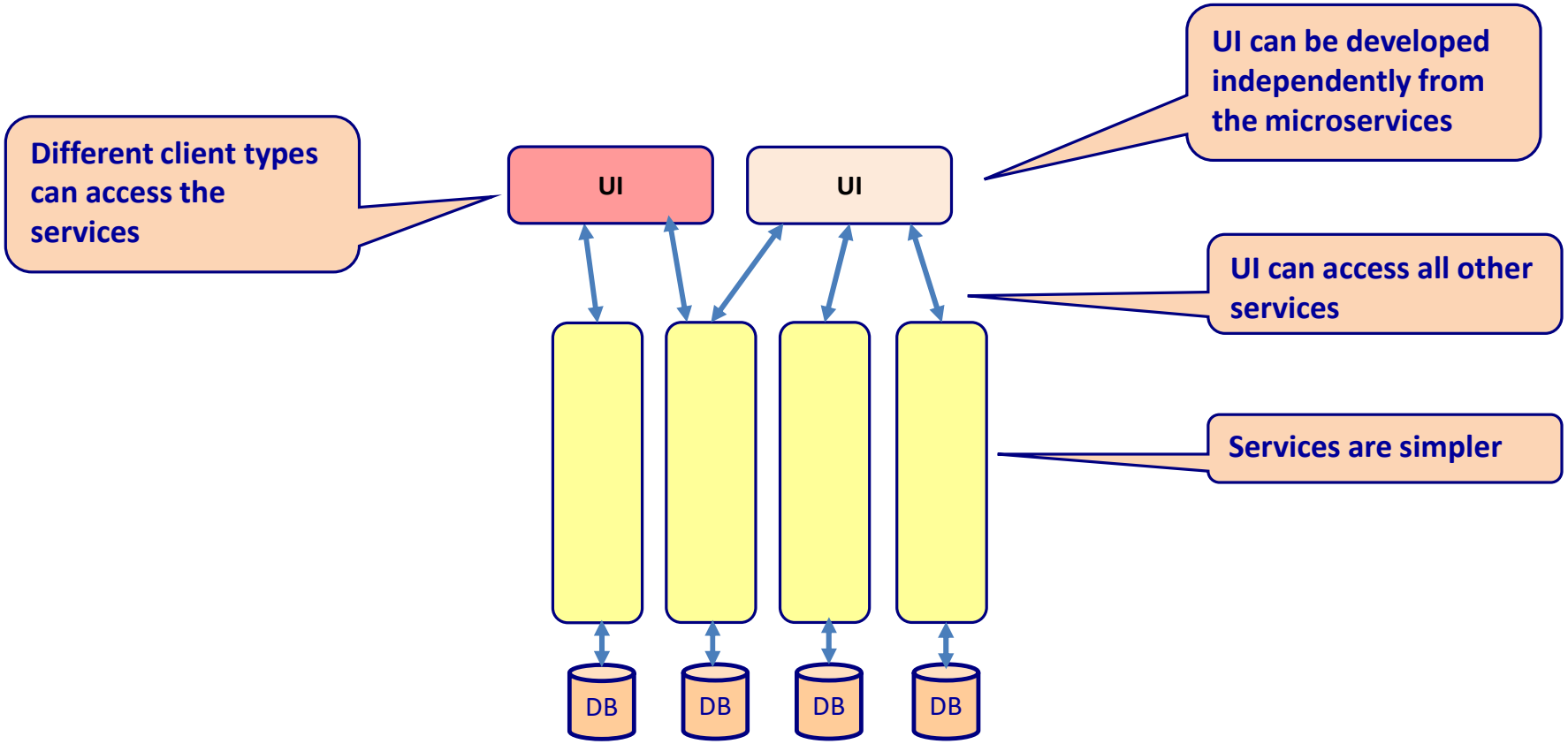


Data consistency



UI AND MICROSERVICE

Split front-end and back-end



MICROSERVICE BOUNDARIES

Appropriate boundaries

- DDD bounded context
  - Isolated domains that are closely aligned with business capabilities
- Autonomous functions
  - Accept input, perform its logic and return a result
    - Encryption engine
    - Notification engine
    - Delivery service that accept an order and informs a trucking service

Appropriate boundaries

- Size of deployable unit
  - Manageable size
- Most appropriate function or subdomain
  - What is the most useful component to detach from the monolith?
  - Hotel booking system: 60-70% are search request
    - Move out the search function
- Polyglot architecture
  - Functionality that needs different architecture
    - Booking service needs transactions
    - Search does not need transactions

Appropriate boundaries

- Selective scaling
  - Functionality that needs different scaling
    - Booking service needs low scaling capabilities
    - Search needs high scaling capabilities
- Small agile teams
  - Specialist teams that work on their expertise
- Single responsibility

Appropriate boundaries

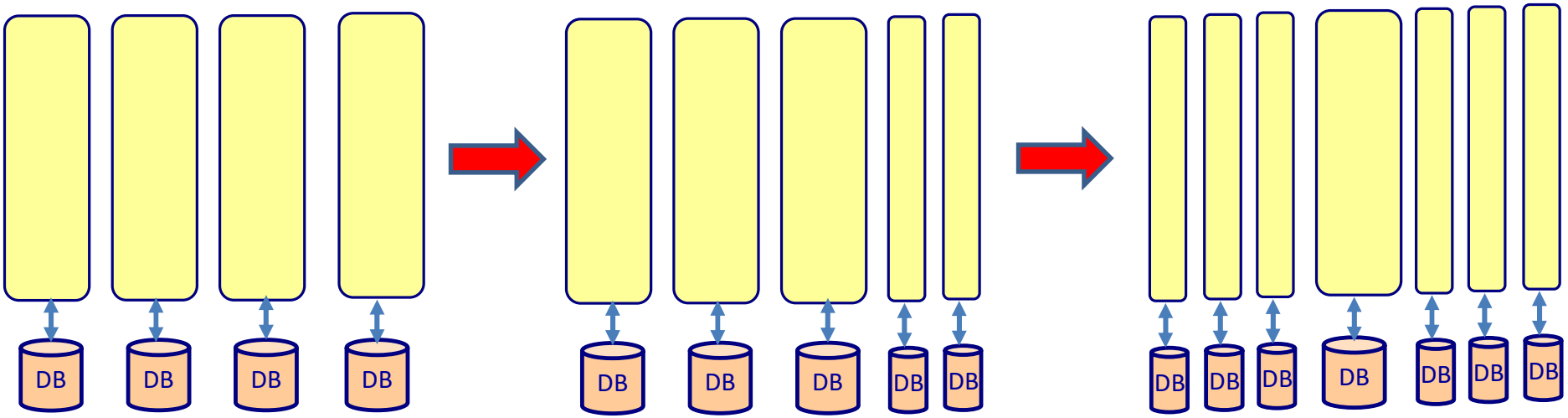
- Replicability or changeability
  - The microservice is easy detachable from the overall system
  - What functionality might evolve in the future?
- Coupling and cohesion
  - Avoid chatty services
  - Too many synchronous request
  - Transaction boundaries within one service

Appropriate boundaries

- DDD bounded context
- Autonomous functions
- Size of deployable unit
- Most appropriate function or subdomain
- Polyglot architecture
- Selective scaling
- Small agile teams
- Single responsibility
- Replicability or changeability
- Coupling and cohesion

Microservice boundaries

- Start with a few services and then evolve to more services



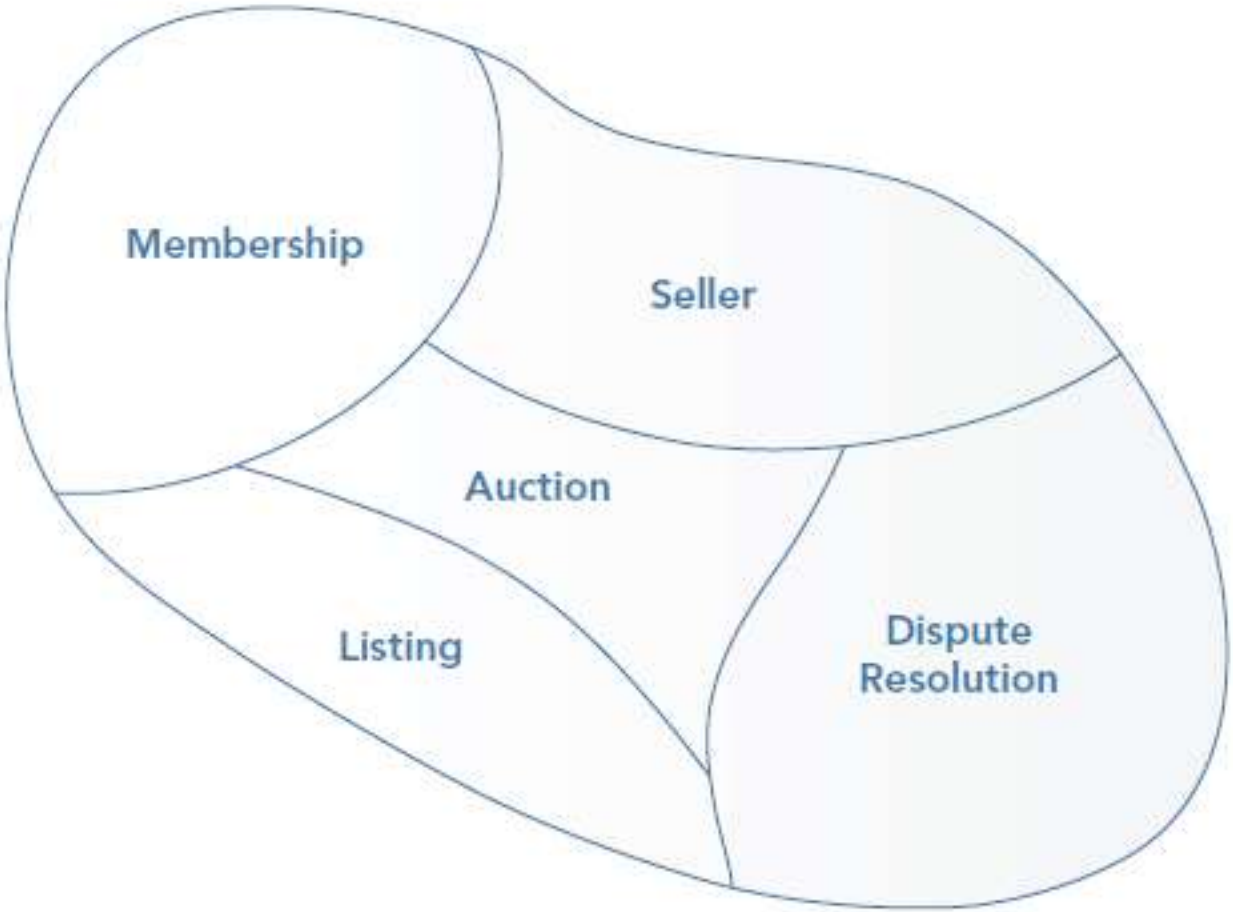
Domains

- Core subdomain
  - This is the reason you are writing the software.
- Supporting subdomain
  - Supports the core domain
- Generic subdomain
  - Very generic functionality
    - Email sending service
    - Creating reports service

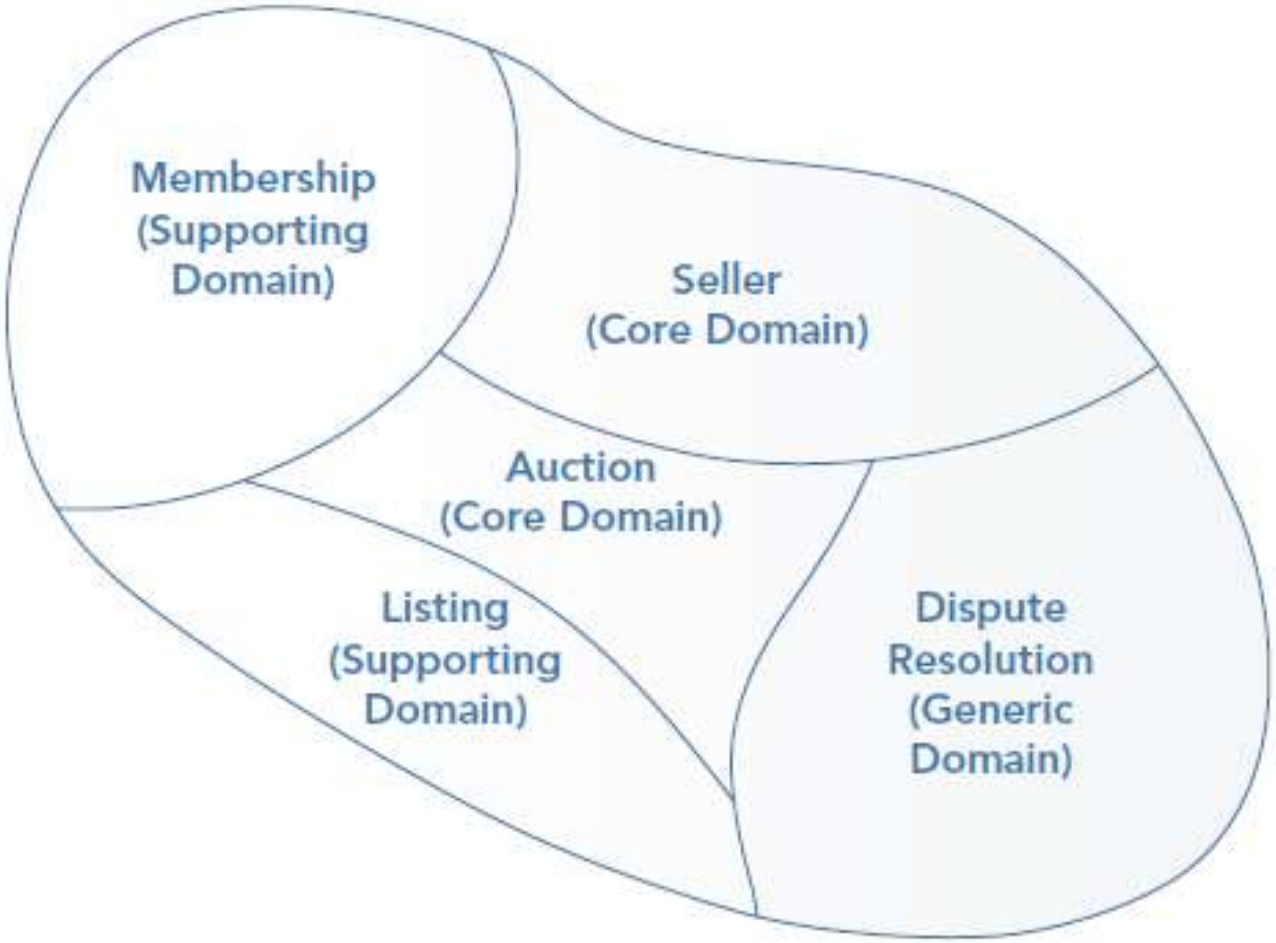
Distilling the domain

- The large domain of online auction

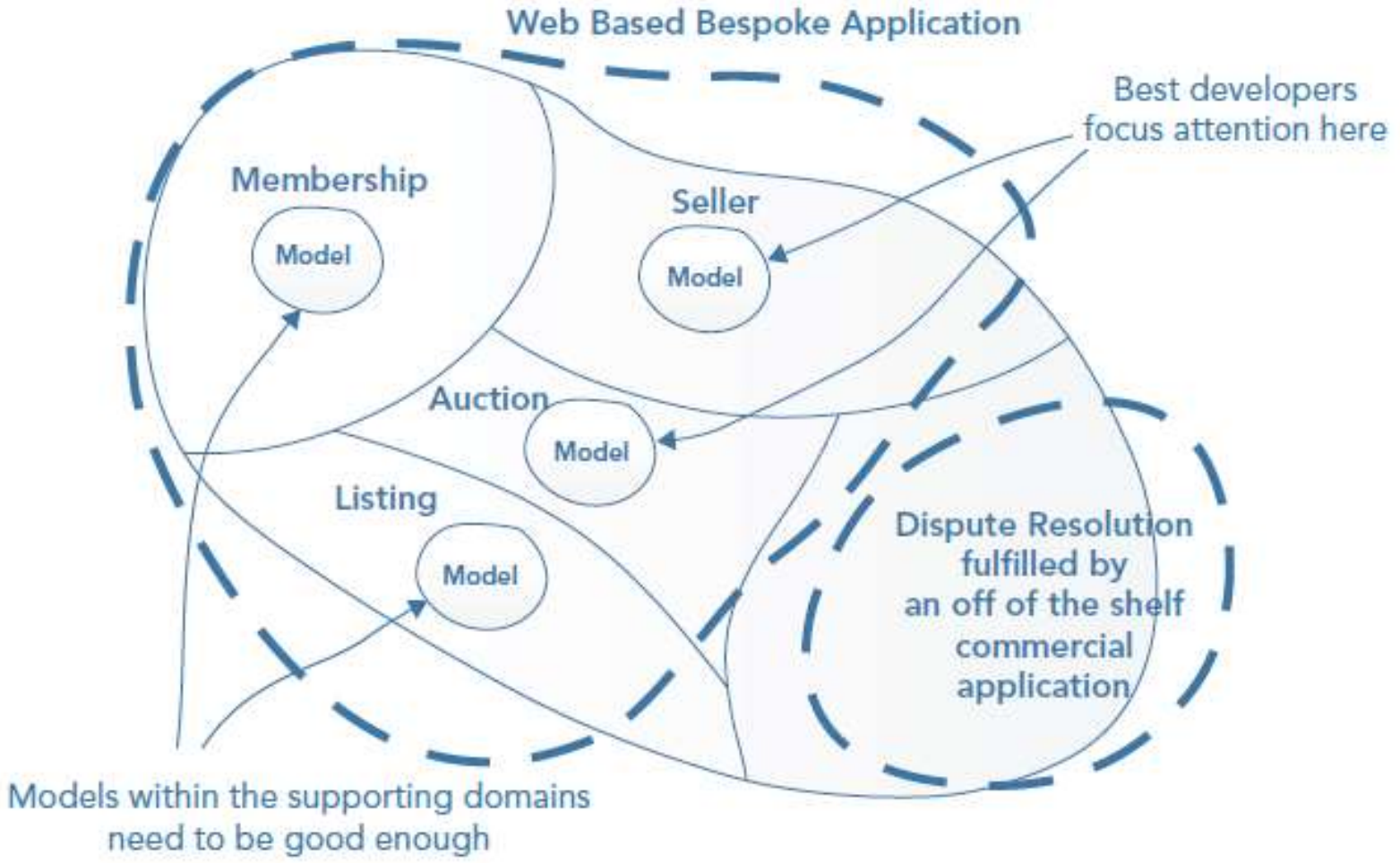
Find the subdomains



Identify the core domain

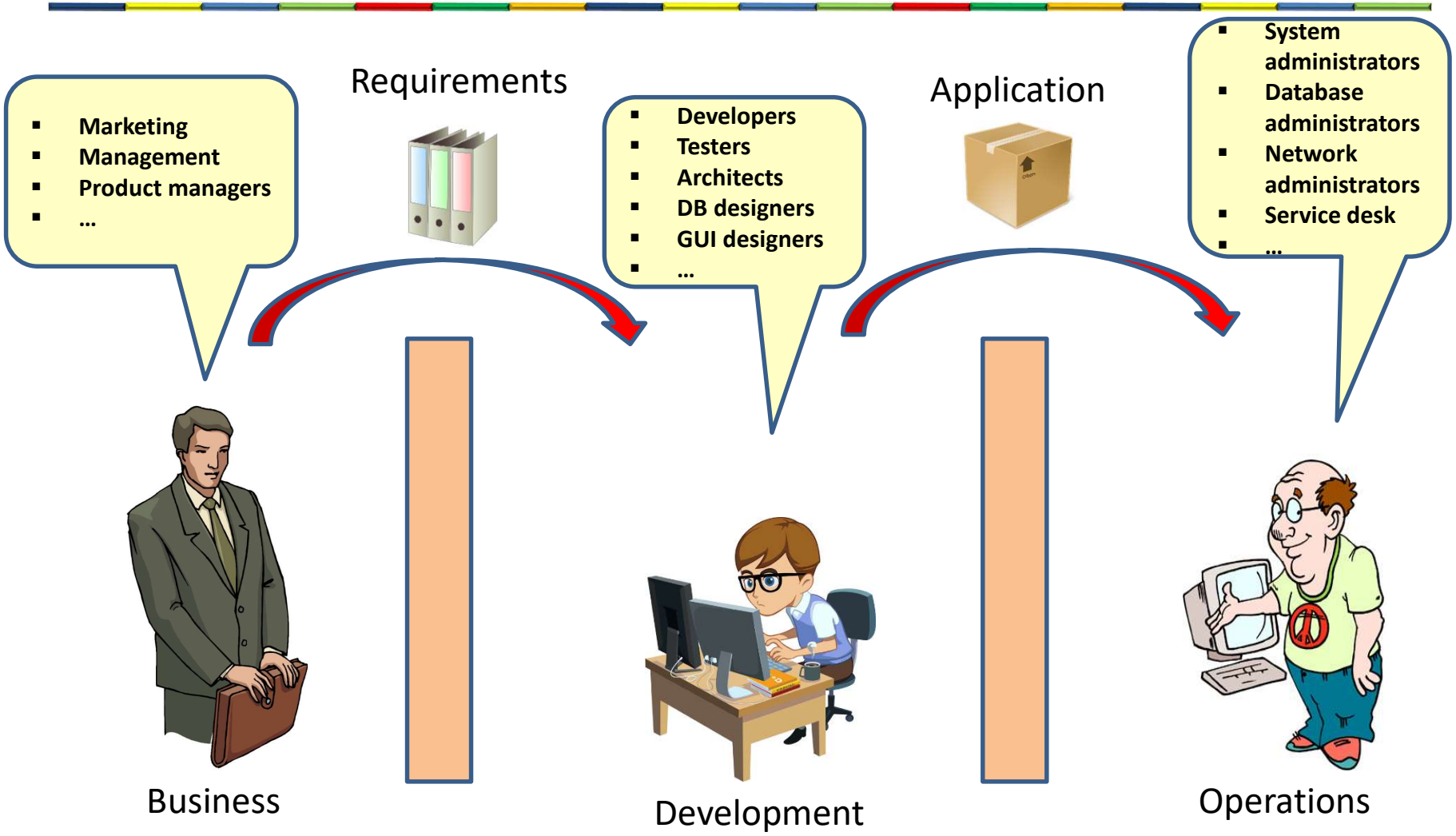


Subdomains shape the solution



MICROSERVICES IN THE ORGANIZATION

Traditional software development



Agile software development: Scrum

- **Close collaboration**
- **Better communication**
- **Short delivery cycles**
- **Short feedback loops**

Product owner (business)
and developers in one team

Application

Operations

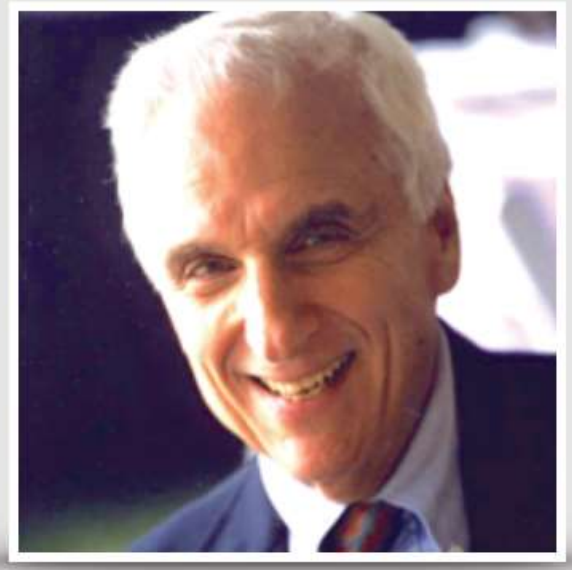
DevOps

- **Close collaboration between developers and operations**
- **Streamlines the delivery process of software from business requirements to production**
- **Better communication**
- **Identical development and production environment**
- **Shared tools**
  - **Automate everything**
  - **Monitor everything**

Product owner (business) and developers in one team

Operations

Conways law



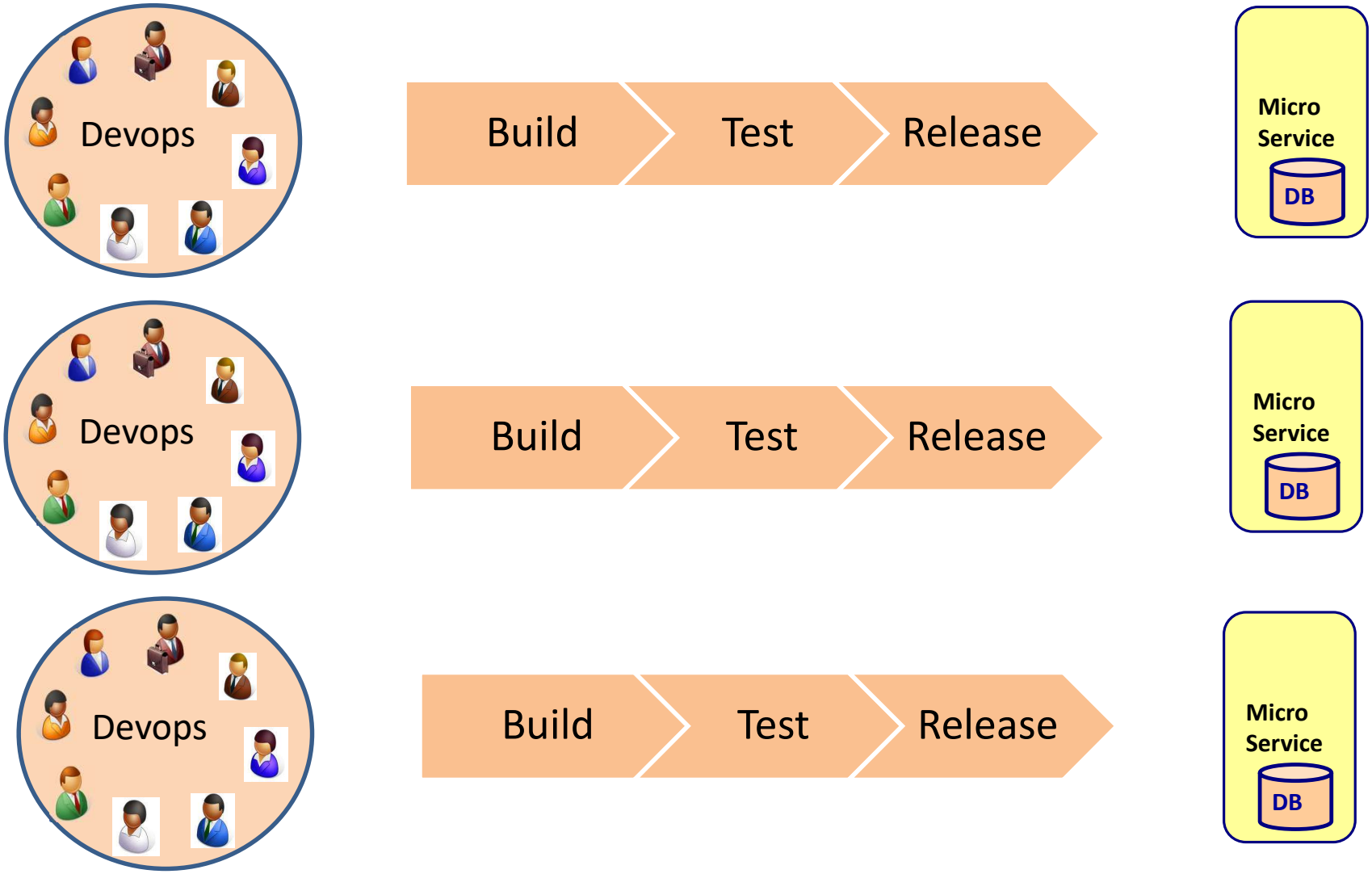
"If you have four groups working on a compiler, you'll get a 4-pass compiler"

—Eric S Raymond

"organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations "

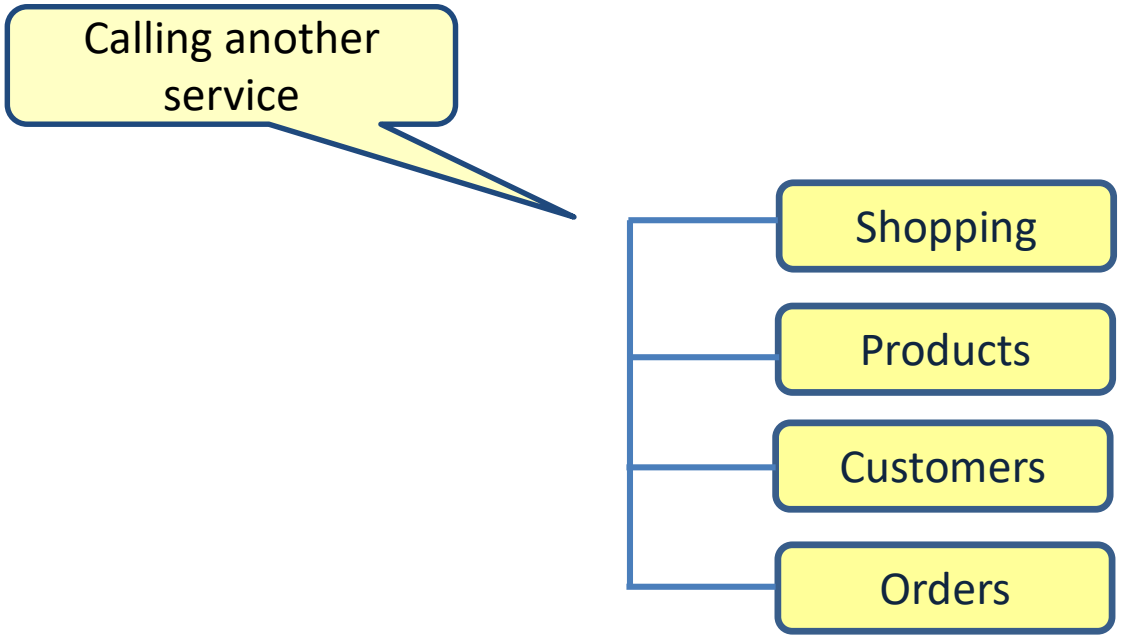
—Melvin Conway

Microservice organization

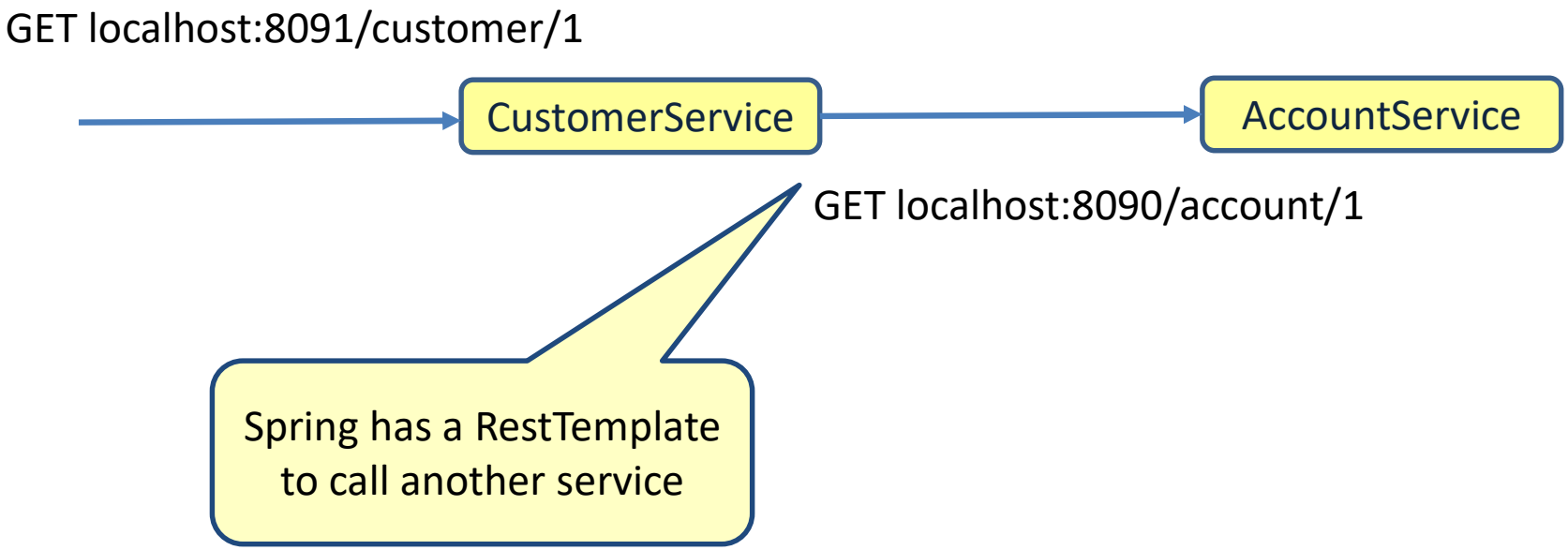


CALLING ANOTHER MICROSERVICE: FEIGN

Implementing microservices



Calling another service



RestTemplate

```
@Component
public class RestClient {
  @Autowired
  private RestOperations  restTemplate;

  public void callRestServer(){
        Greeting greeting =
           restTemplate.getForObject("http://localhost:8080/greeting", Greeting.class);
        System.out.println("Receiving message:"+greeting.getContent());
  }
}
```

RestTemplate has to be configured.
Developer has to know REST details

```
@Configuration
public class AppConfig {
   @Bean
   RestTemplate restTemplate(){
        return new RestTemplate();
   }
}
```

Feign

- Declarative HTTP client
  - Simplify the HTTP client
- You only need to declare and annotate the interface

AccountService

```java
@RestController
public class AccountController {
  @RequestMapping("/account/{customerid}")
  public Account getName(@PathVariable("customerid") String customerId) {
    return new Account("1234", "1000.00");
  }
}
```

```java
public class Account {
  private String accountNumber;
  private String balance;
  ...
}
```

application.yml

```yaml
server:
  port: 8090
```

```java
@SpringBootApplication
public class AccountServiceApplication {

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

Properties files and yml files

application.properties

```
Mapping single properties
myapp.mail.to=frank@hotmail.com
myapp.mail.host=mail.example.com
myapp.mail.port=250

#Mapping list or array
myapp.mail.cc=mike@gmail.com,david@gmail.com
myapp.mail.bcc=john@hotmail.com,admin@acme.com

#Mapping nested POJO class
myapp.mail.credential.user-name=john1234
myapp.mail.credential.password=xyz@1234
```

application.yml

```
myapp:
  mail:
    to: frank@hotmail.com
    host: mail.example.com
    port: 250
    cc:
      - mike@gmail.com
      - david@gmail.com
    bcc:
      - john@hotmail.com
      - admin@acme.com
    credential:
      user-name: john1234
      password: xyz@1234
```

CustomerService

```
@SpringBootApplication
@EnableFeignClients
public class CustomerServiceApplication {

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

Use Feign

application.yml

```
server:
  port: 8091
```

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

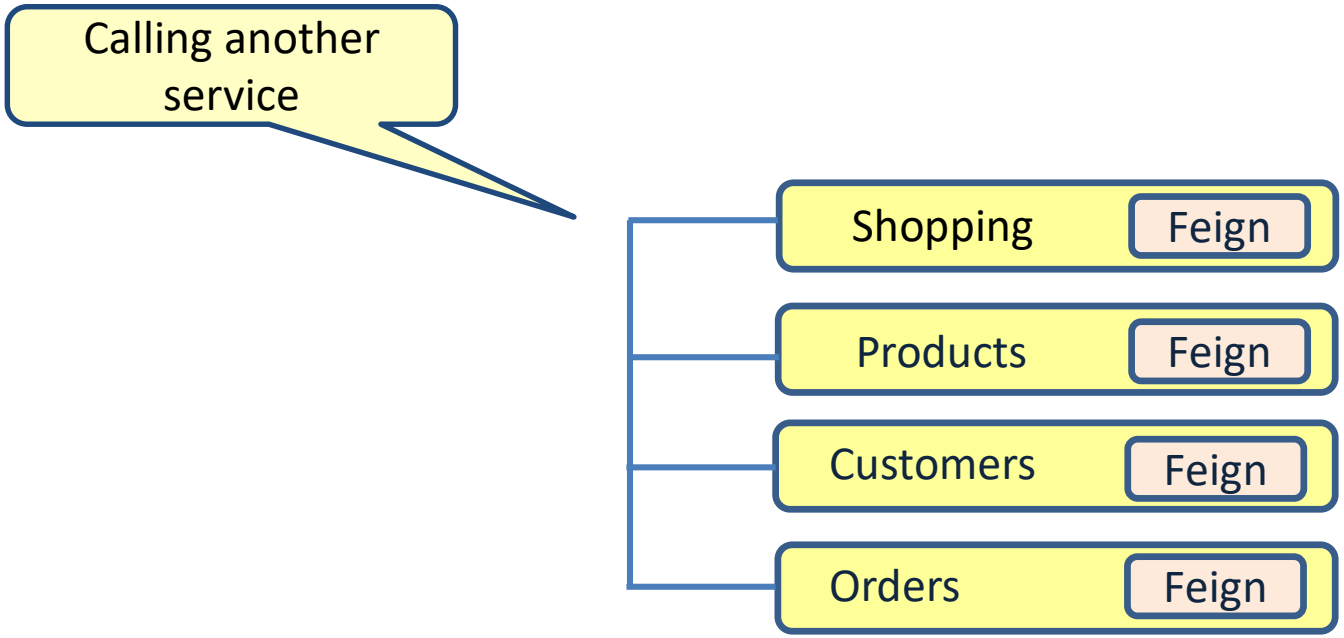
CustomerService: the controller

```
@RestController
public class CustomerController {
  @Autowired
  AccountFeignClient accountClient;

  @RequestMapping("/customer/{customerid}")
  public Account getName(@PathVariable("customerid") String customerId) {
        Account account = accountClient.getName(customerId);
        return account;
  }

  @FeignClient(name = "account-service", url = "http://localhost:8090")
  interface AccountFeignClient {
    @RequestMapping("/account/{customerid}")
    public Account getName(@PathVariable("customerid") String customerId);
  }
}
```

Autowire the client

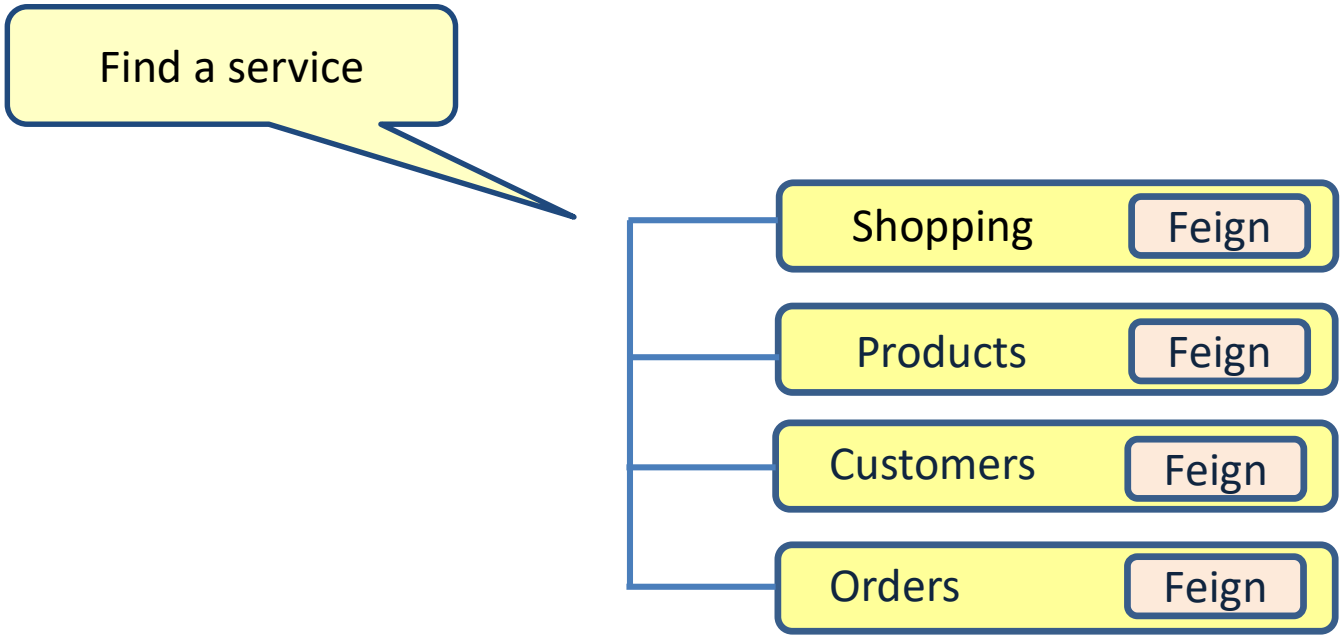
Declare the interface, Spring creates the implementation

Implementing microservices

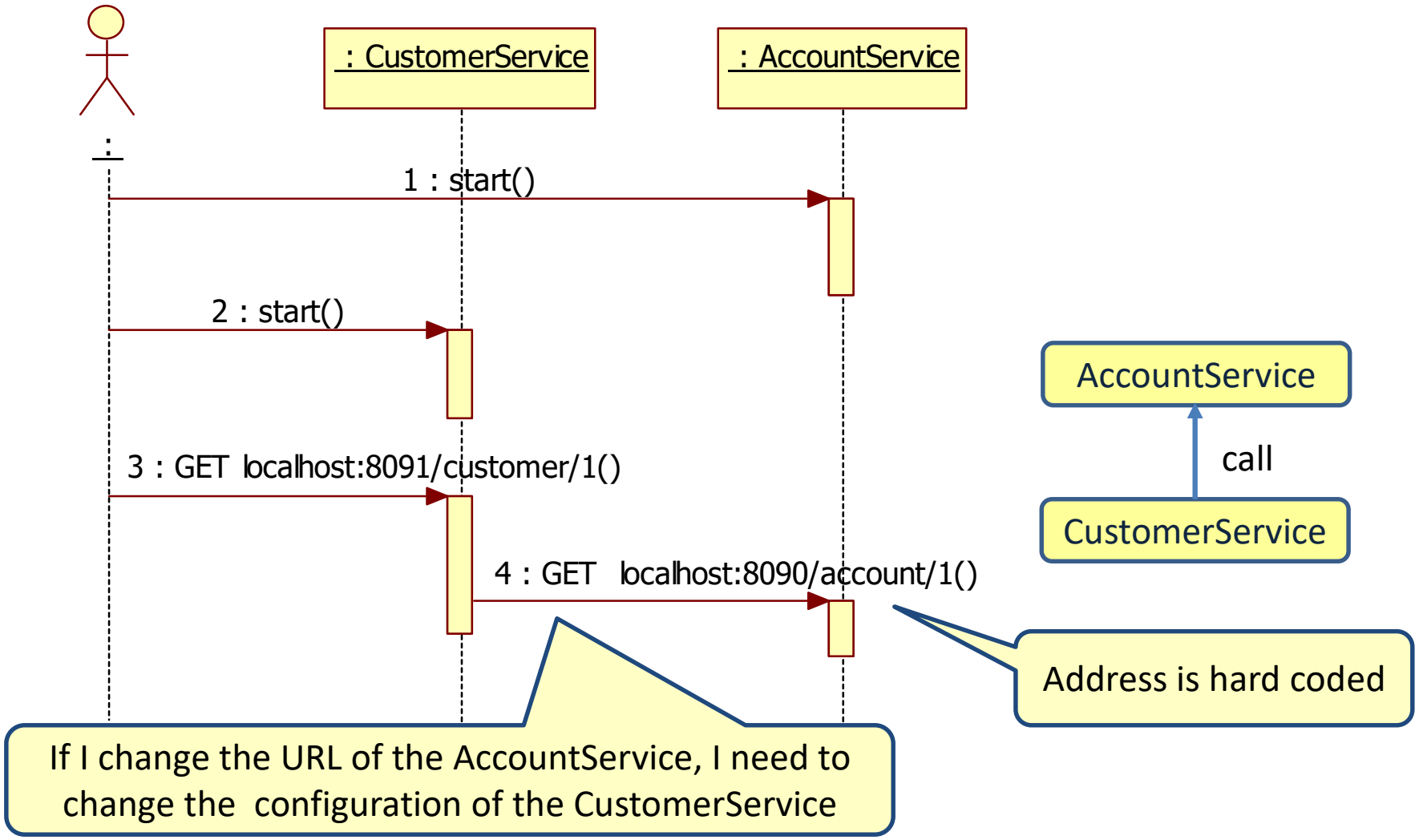


SERVICE REGISTRY: EUREKA

Implementing microservices



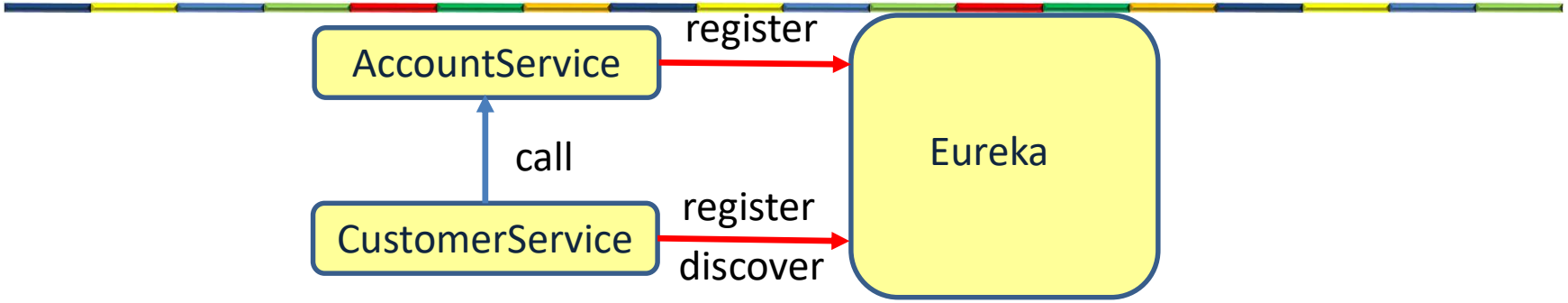
One service calling another service

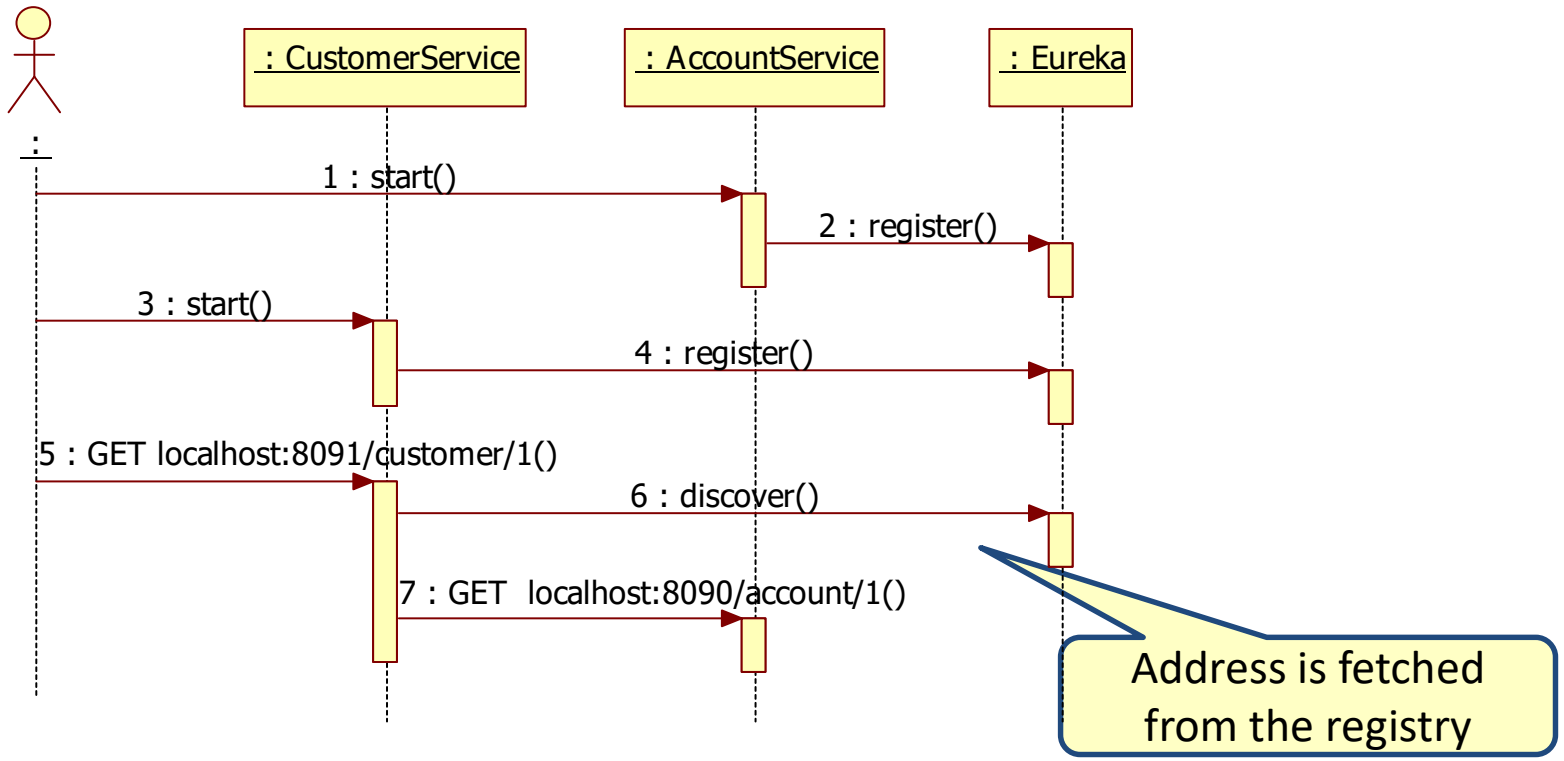


Service Registry

- Like the phone book for microservices
  - Services register themselves with their location and other meta-data
  - Clients can lookup other services
- Netflix Eureka

Using Eureka





Why service registry/discovery?

1. Loosely coupled services
   - Service consumers should not know the physical location of service instances.
     - We can easily scale up or scale down service instances
2. Increase application resilience
   - If a service instance becomes unhealthy or unavailable, the service discovery engine will remove that instance from the list of available services.

Eureka Server

```java
@SpringBootApplication
@EnableEurekaServer
public class EurekaServerApplication {

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

application.yml

```yaml
server:
  port: 8761

eureka:
  client:
    registerWithEureka: false   #telling the server not to register himself
    fetchRegistry: false
```

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

Running Eureka

AccountService

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

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

application.yml

```yaml
server:
  port: 8090

eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/

spring:
  application:
    name: AccountService
```

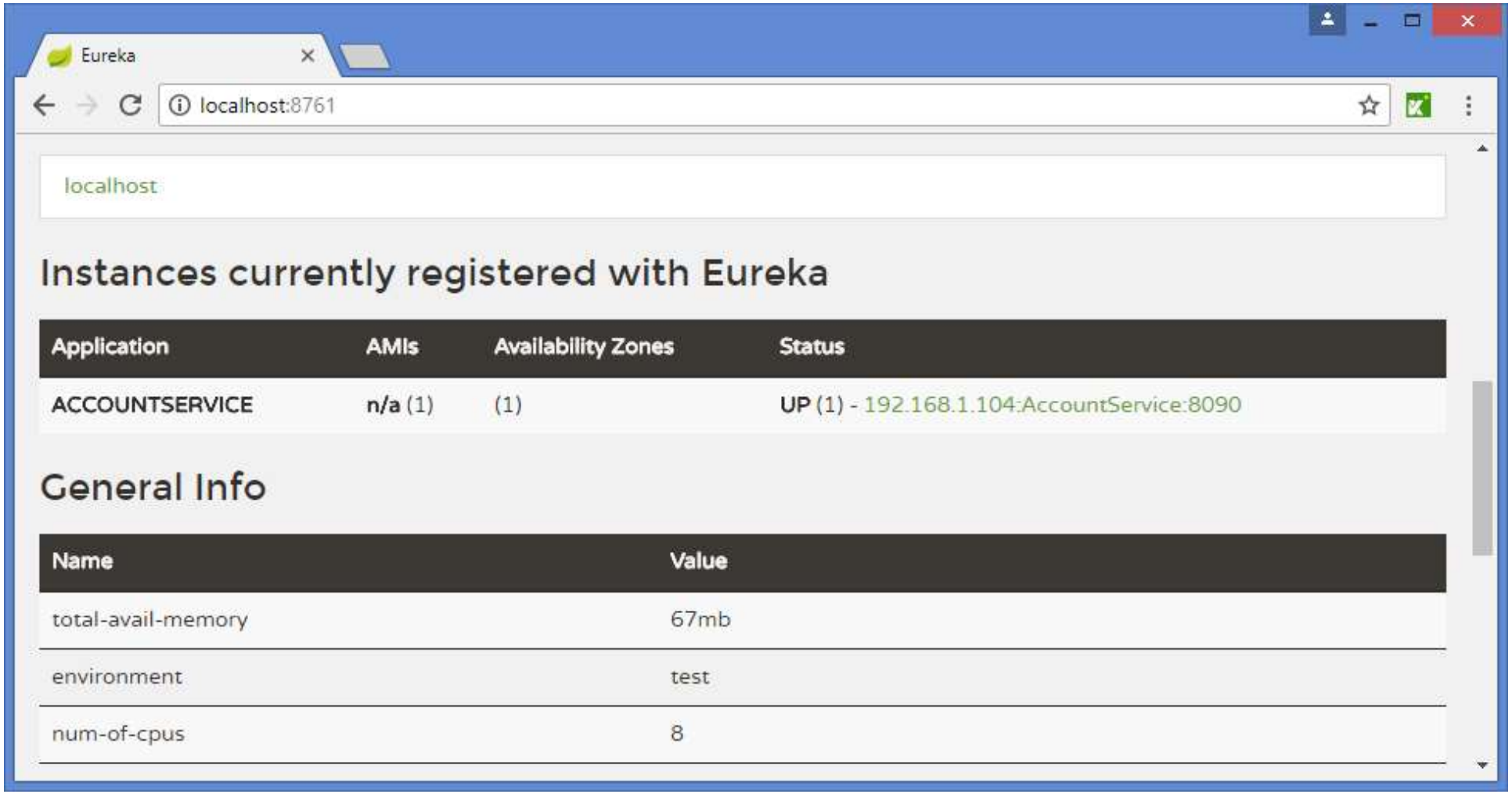
AccountService

```
@RestController
public class AccountController {
  @RequestMapping("/account/{customerid}")
  public Account getName(@PathVariable("customerid") String customerId) {
    return new Account("1234", "1000.00");
  }
}
```

```
public class Account {
  private String accountNumber;
  private String balance;
  ...
}
```

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

Running the AccountService



CustomerService

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients
public class CustomerServiceApplication {

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

Use Feign and Eureka

application.yml

```
server:
  port: 8091

eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/

spring:
  application:
    name: CustomerService
```

CustomerService: the controller

```
@RestController
public class CustomerController {
  @Autowired
  AccountFeignClient accountClient;

  @RequestMapping("/customer/{customerid}")
  public Account getName(@PathVariable("customerid") String customerId) {
    Account account = accountClient.getName(customerId);
    return account;
  }

  @FeignClient("AccountService")
  interface AccountFeignClient {
    @RequestMapping("/account/{customerid}")
    public Account getName(@PathVariable("customerid") String customerId);
  }
}
```

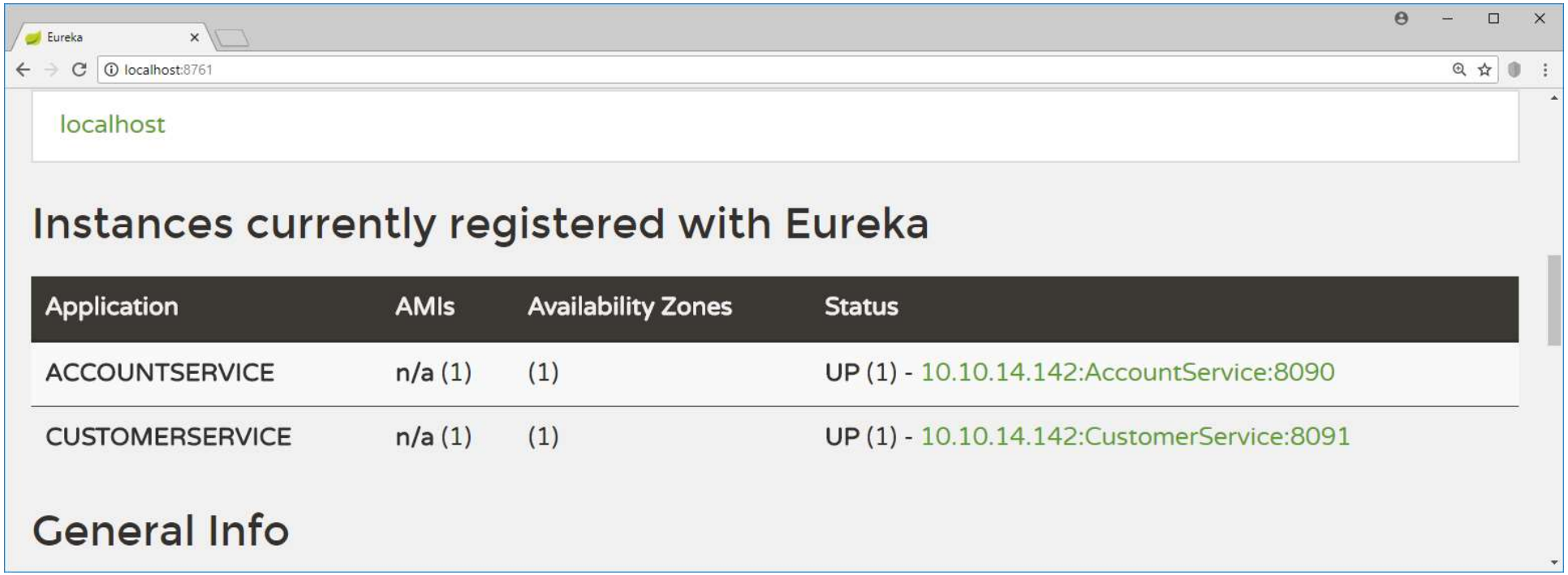
Name of the service instead of the URL

Feign works together with Eureka

application.yml

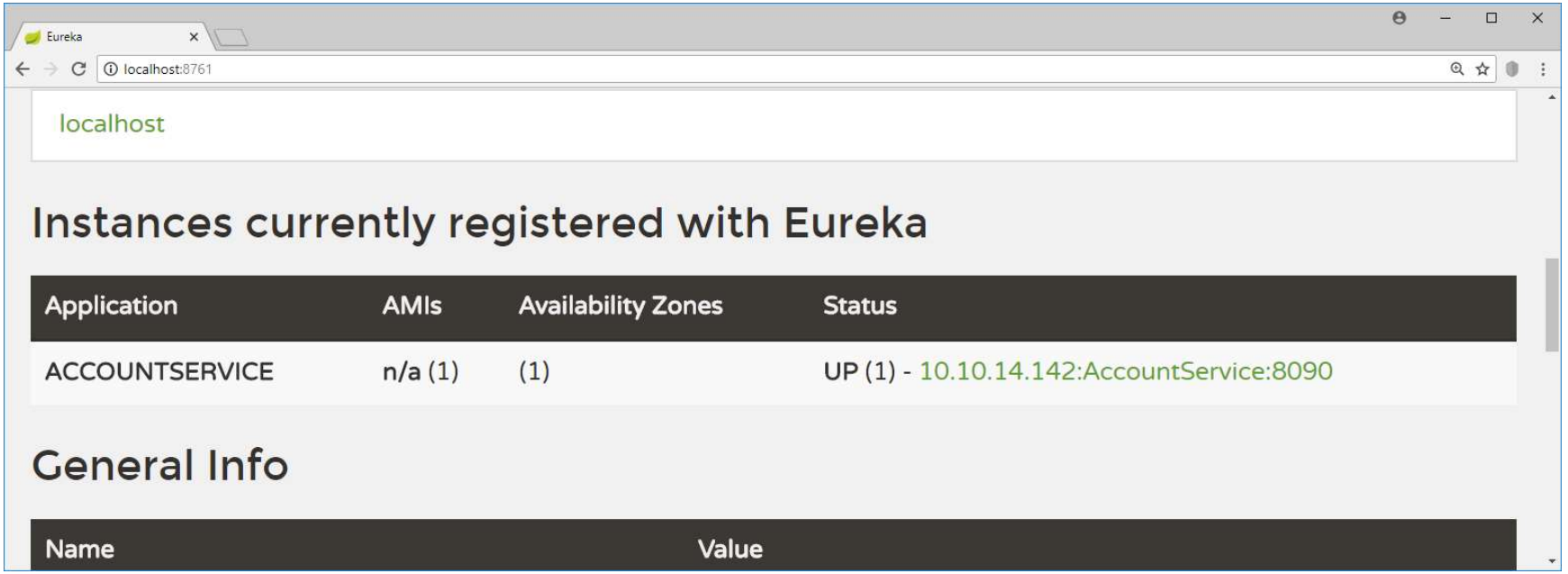
```
server:
  port: 8091
```

Running the CustomerService

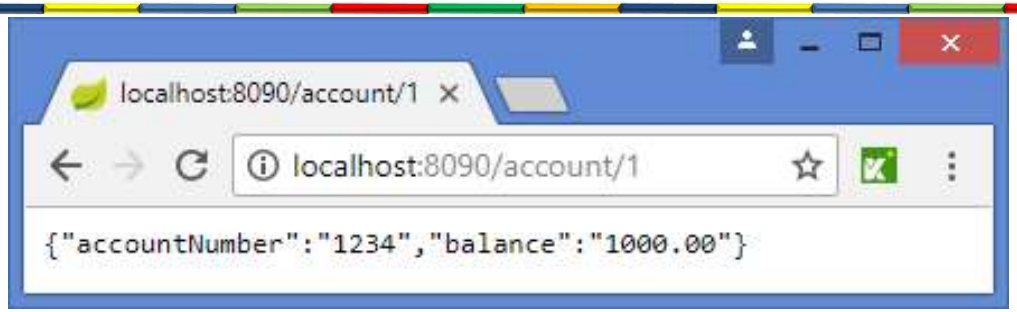


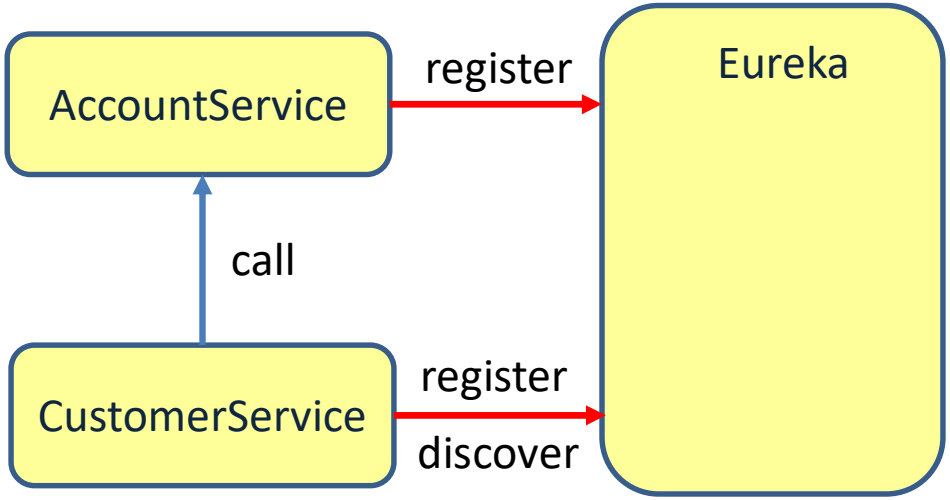
Stopping the CustomerService

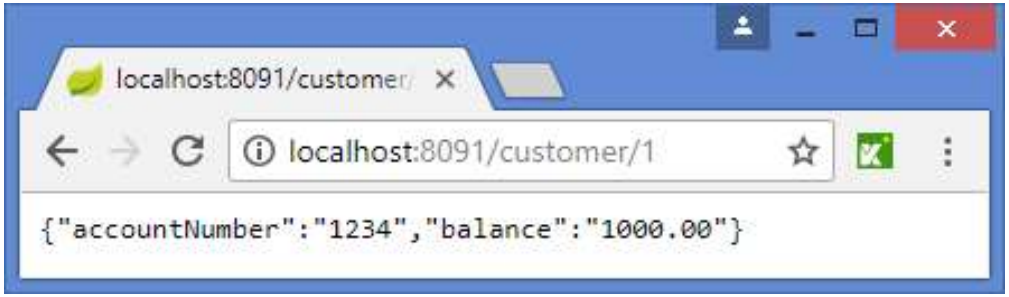
- Eureka monitors the health of registered services.
- If we stop the CustomerService, Eureka will notice that automatically



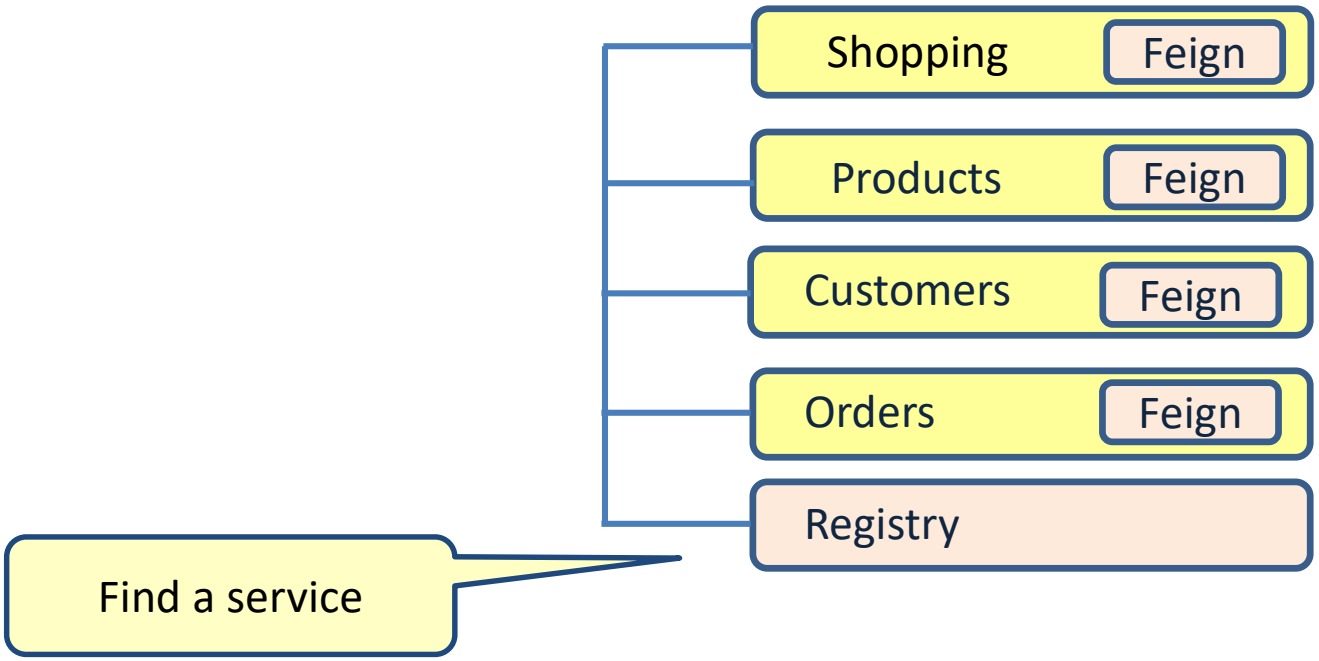
Using Eureka







Implementing microservices



Lesson 9

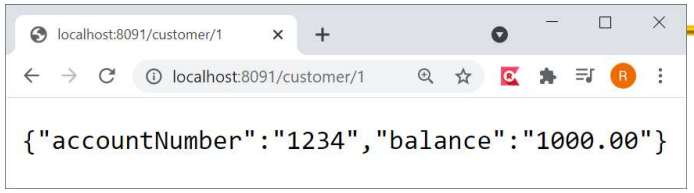
MICROSERVICES

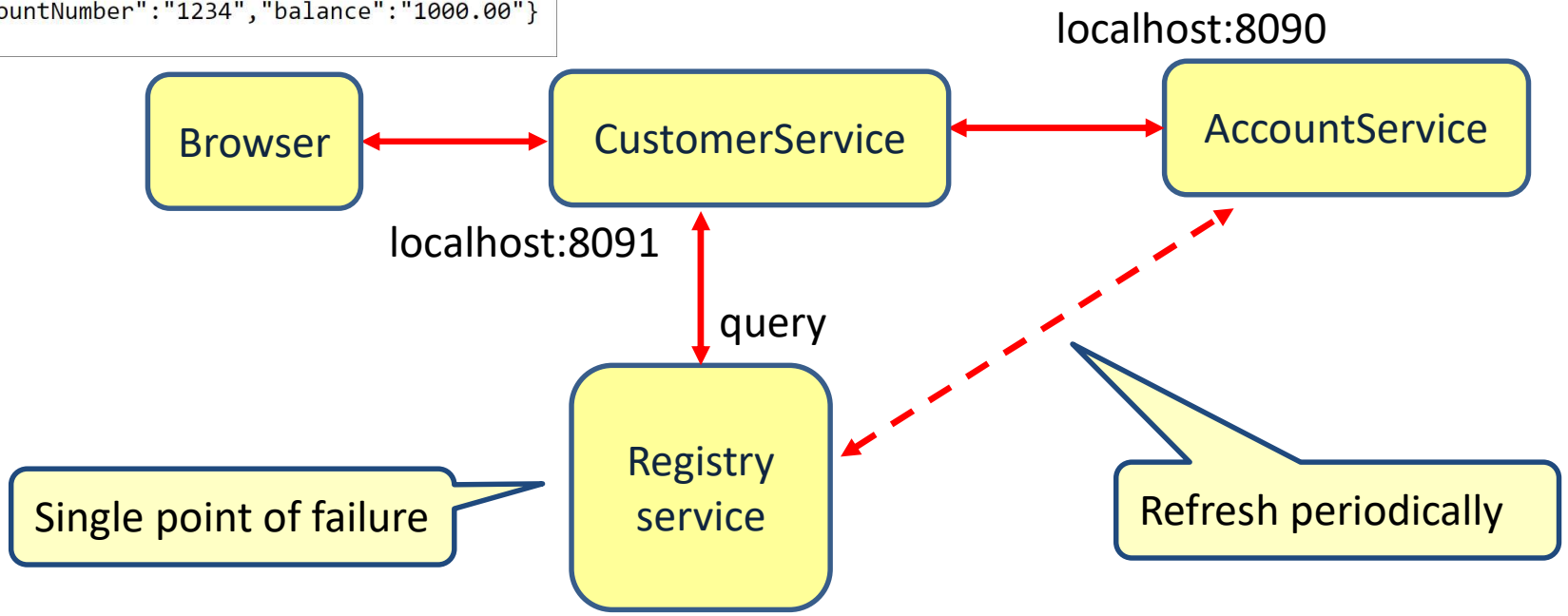
Challenges of a microservice architecture

Challenge	Solution
Complex communication	Feign Registry
Performance	
Resilience	
Security	
Transactions	
Following the process	
Keep data in sync	
Keep interfaces in sync	
Keep configuration in sync	
Monitor health of microservices	
Follow/monitor business processes	

EUREKA FAILOVER

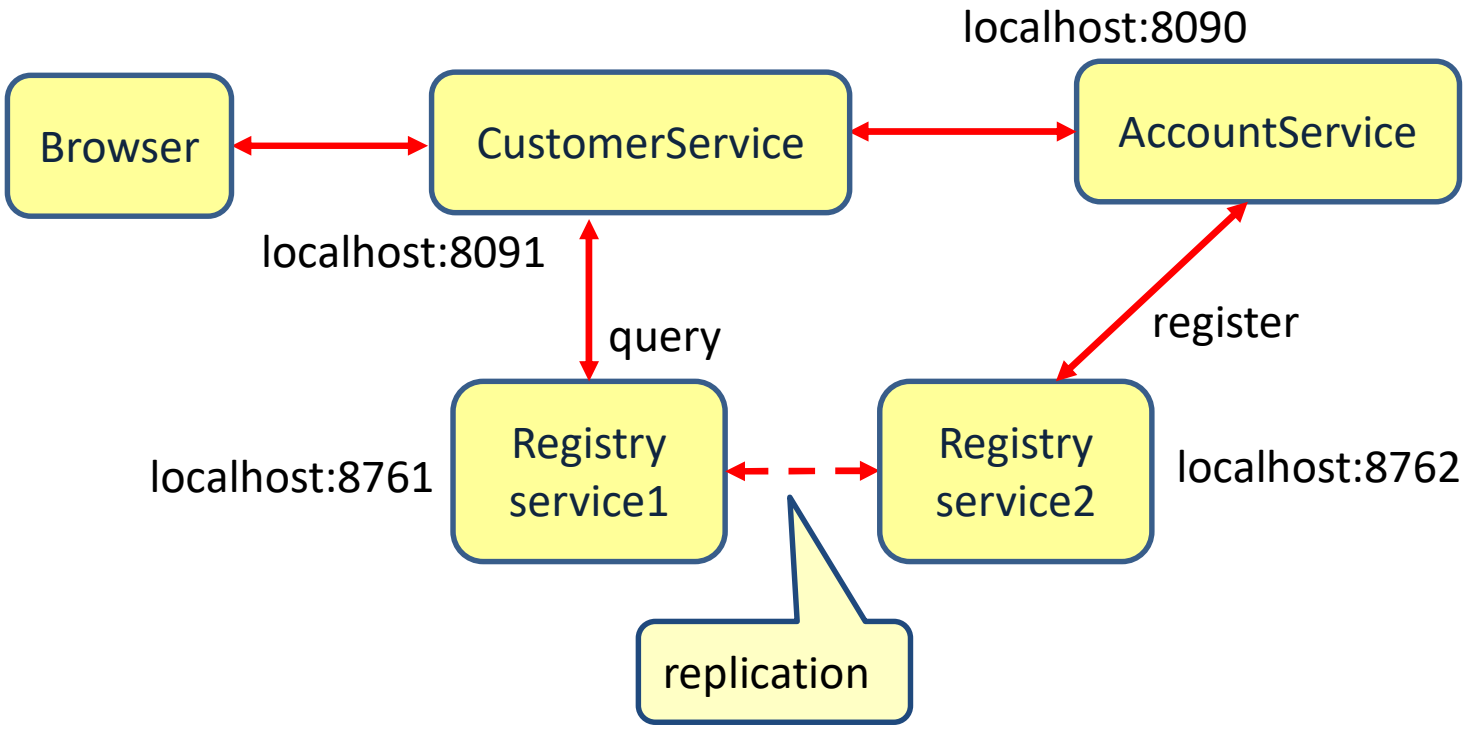
Eureka registry

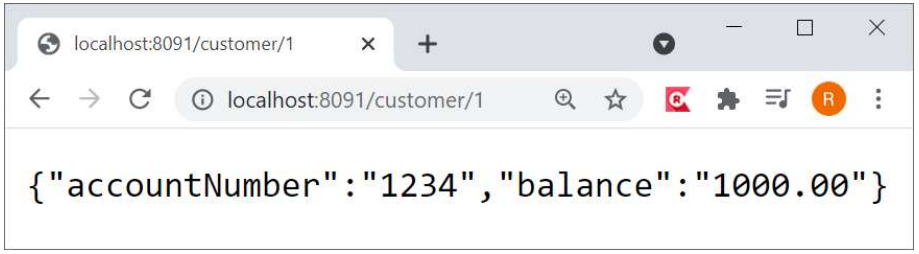




ACCOUNTSERVICE	n/a (1)	(1)	UP (1) - 10.10.14.142:AccountService:8090
CUSTOMERSERVICE	n/a (1)	(1)	UP (1) - 10.10.14.142:CustomerService:8091

Registry replication





Eureka replicas

eurekaserver1

```
server:
  port: 8761

eureka:
  instance:
    hostname: eurekaserver1
    appname: eureka-cluster
  environment: local
  client:
    serviceUrl:
      defaultZone:
            http://eurekaserver2:8762/eureka
    register-with-eureka: true
    fetch-registry: true
```

eurekaserver2

```
server:
  port: 8762

eureka:
  instance:
    hostname: eurekaserver2
    appname: eureka-cluster
  environment: local
  client:
    serviceUrl:
      defaultZone:
            http://eurekaserver1:8761/eureka
    register-with-eureka: true
    fetch-registry: true
```

Eureka replicas must have a different hostname (and port)

Same app name

Register eurekaserver1 in eurekaserver2

Register eurekaserver2 in eurekaserver1

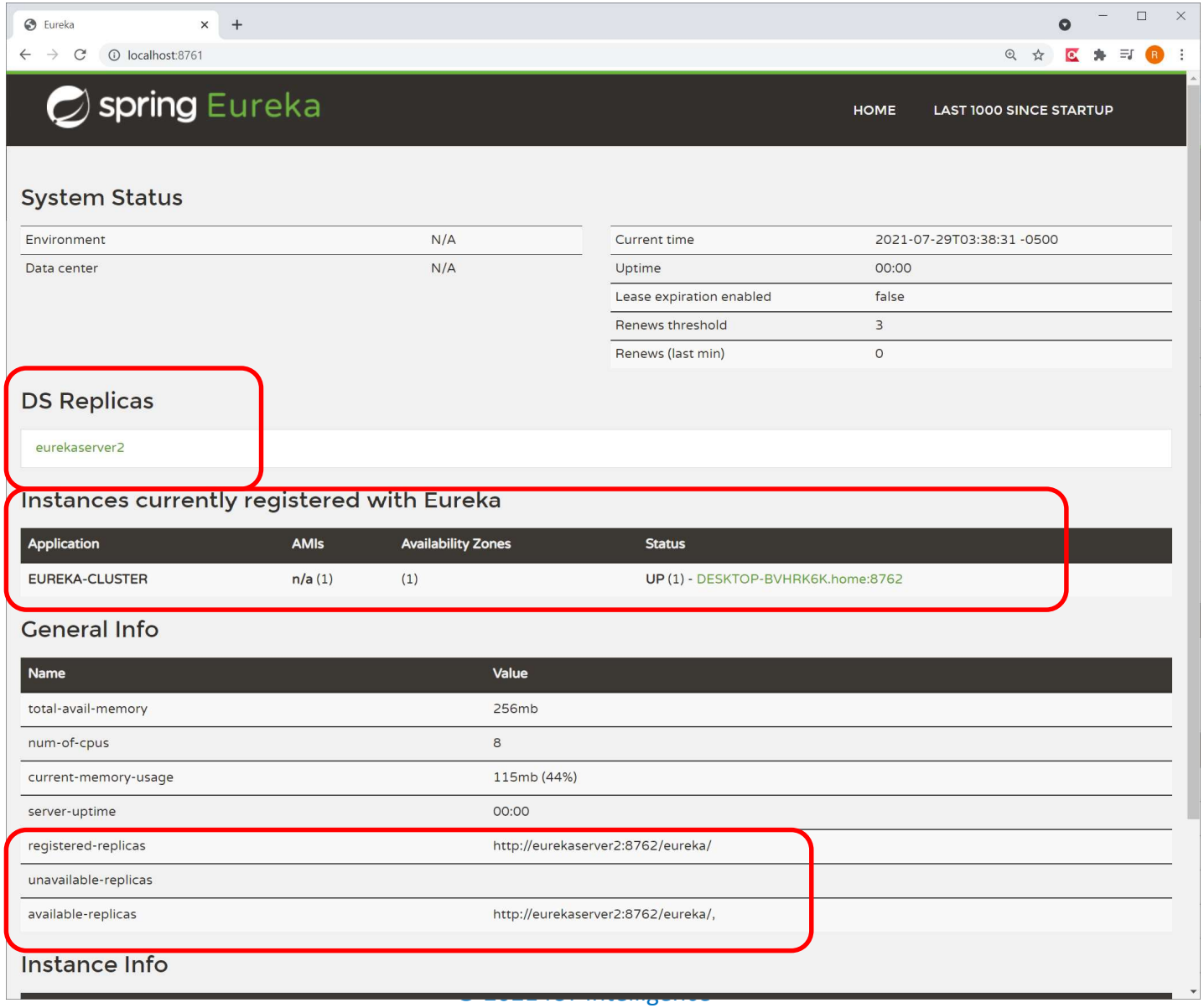
Hosts file

- Window: c:\Windows\System32\Drivers\etc\hosts
- Linux : /etc/hosts

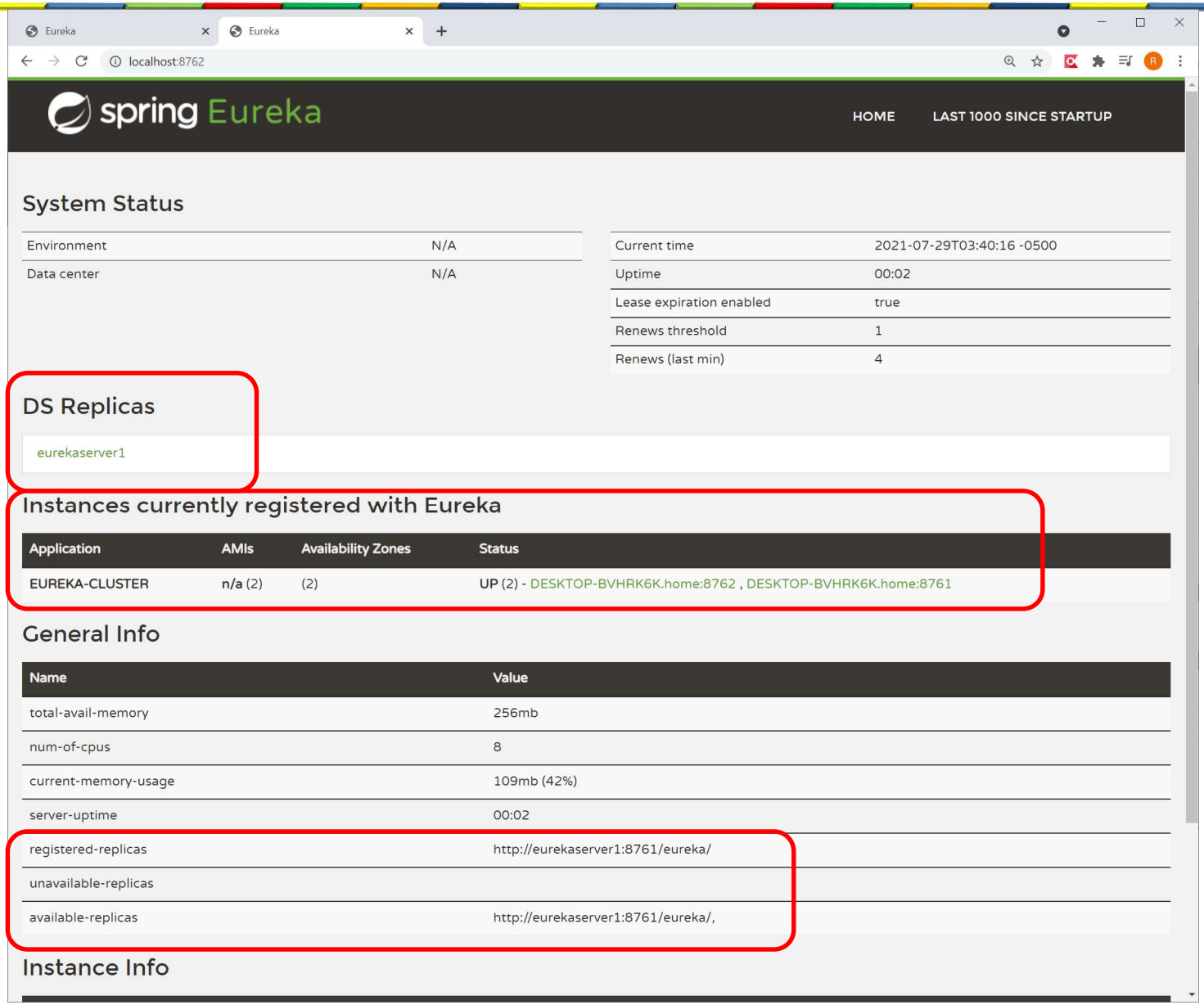
```
# localhost name resolution is handled within DNS itself.
#   127.0.0.1       localhost
#   ::1             localhost
127.0.0.1       eurekaserver1
127.0.0.1       eurekaserver2
```

Map host names to machine addresses

EurekaServer1



EurekaServer2

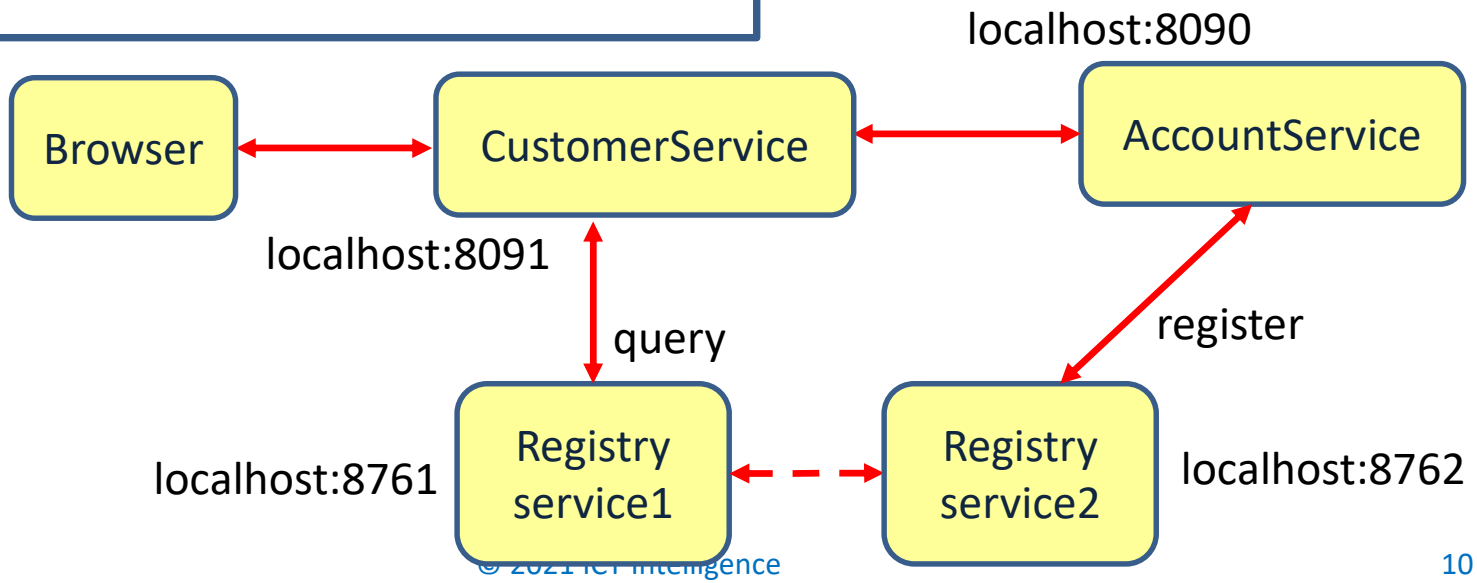


Accountservice

application.yml

```
server:
  port: 8090

eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8762/eureka/

spring:
  application:
    name: AccountService
```

Register in eurekaserver2

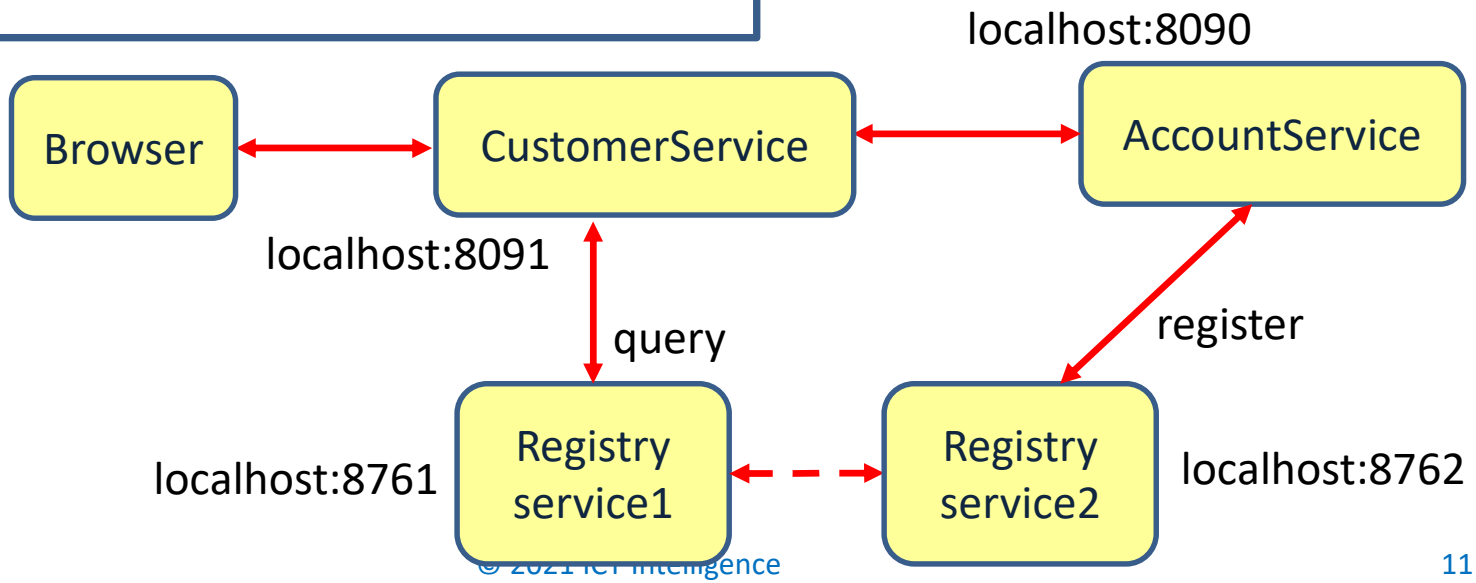


Customerservice

application.yml

```
server:
  port: 8091

eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/

spring:
  application:
    name: CustomerService
```

Use eurekaserver1



Eureka high availability

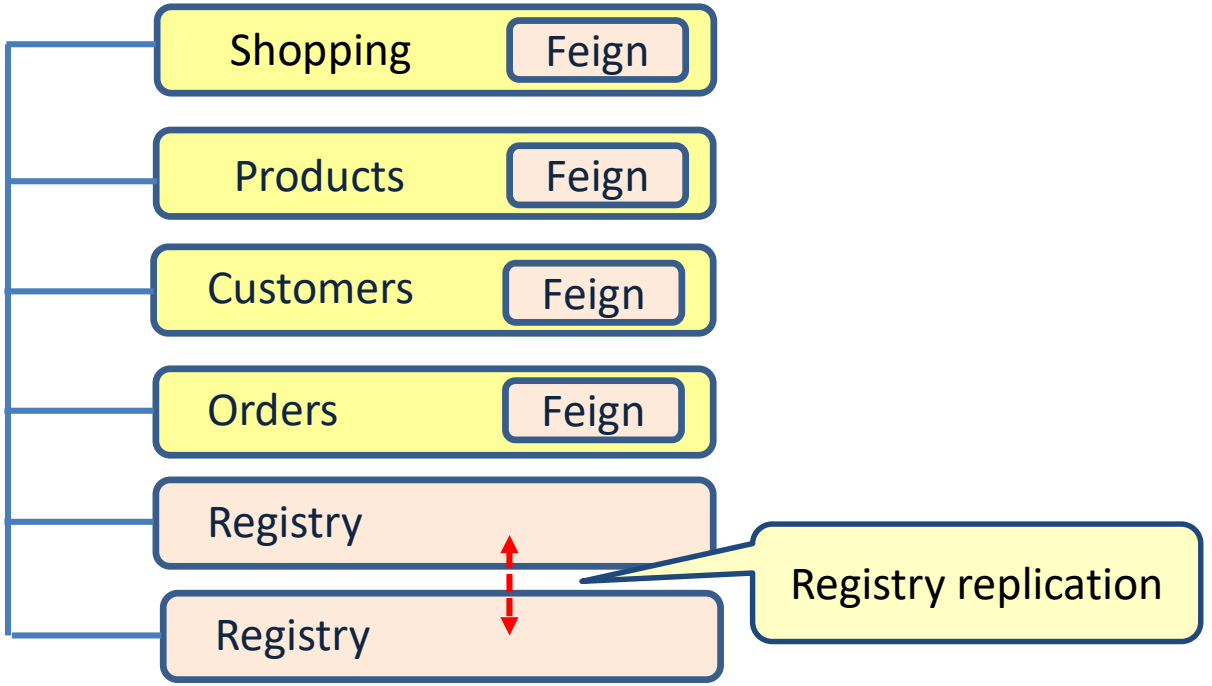
- In the client, multiple Eureka servers can be configured.

application.yml

```
server:
  port: 8091

eureka:
  client:
    serviceUrl:
      defaultZone: http://eurekaserver1:8761/eureka/,
                   http://eurekaserver2:8762/eureka/
```

This can be a comma separated list of Eureka instances.
If the first instance does not respond, we try the next instance

Implementing microservices

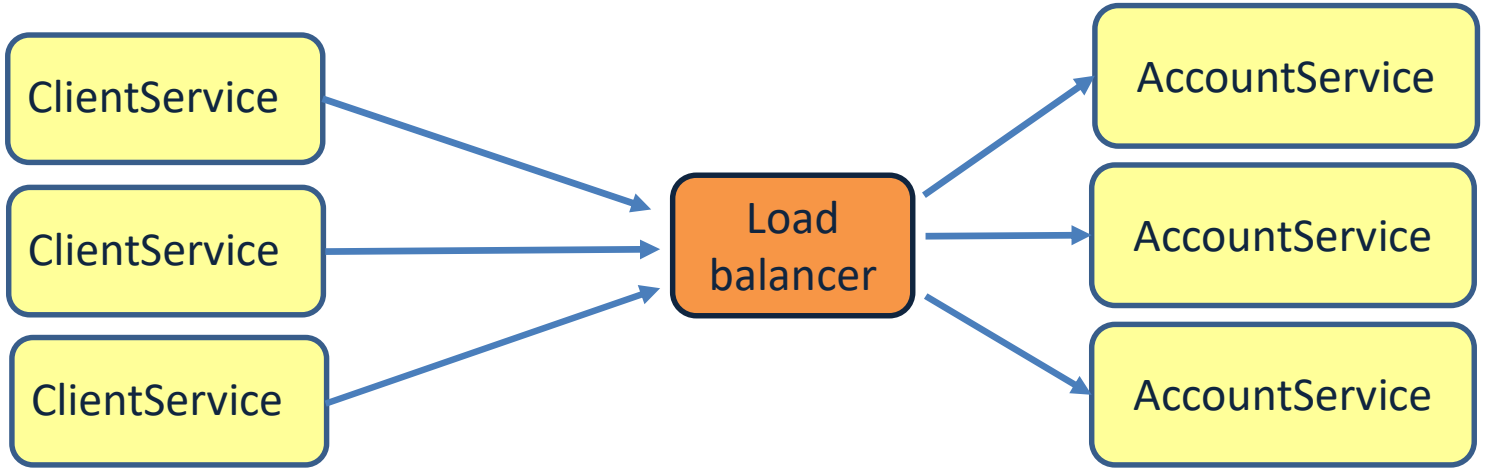


Challenges of a microservice architecture

Challenge	Solution
Complex communication	Feign Registry
Performance	
Resilience	Registry replicas
Security	
Transactions	
Following the process	
Keep data in sync	
Keep interfaces in sync	
Keep configuration in sync	
Monitor health of microservices	
Follow/monitor business processes	

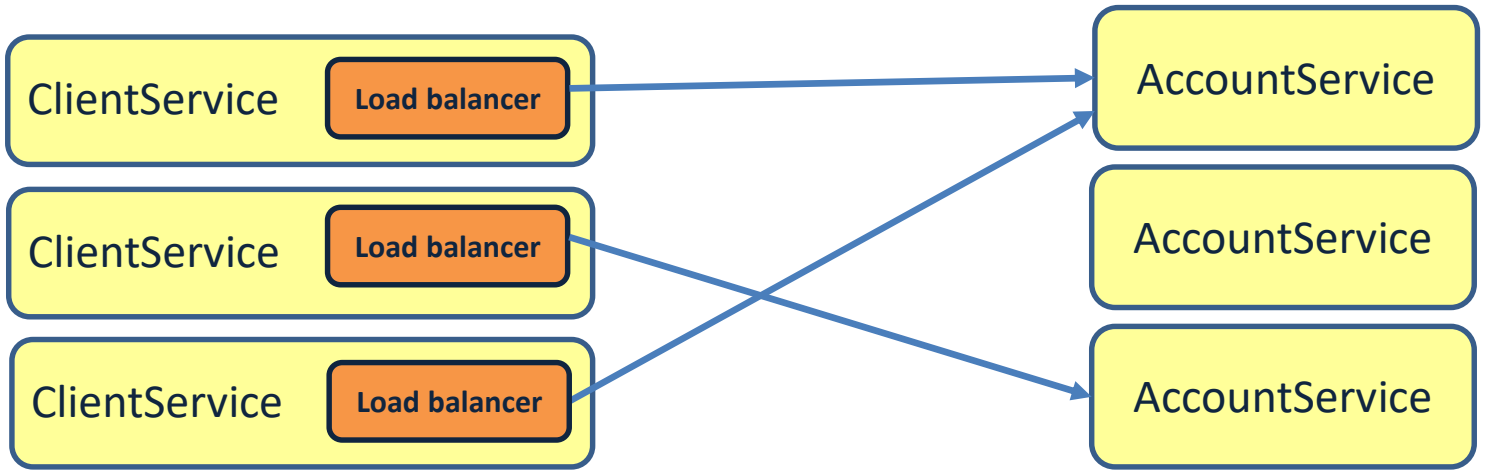
LOAD BALANCING: RIBBON

Server side load balancing



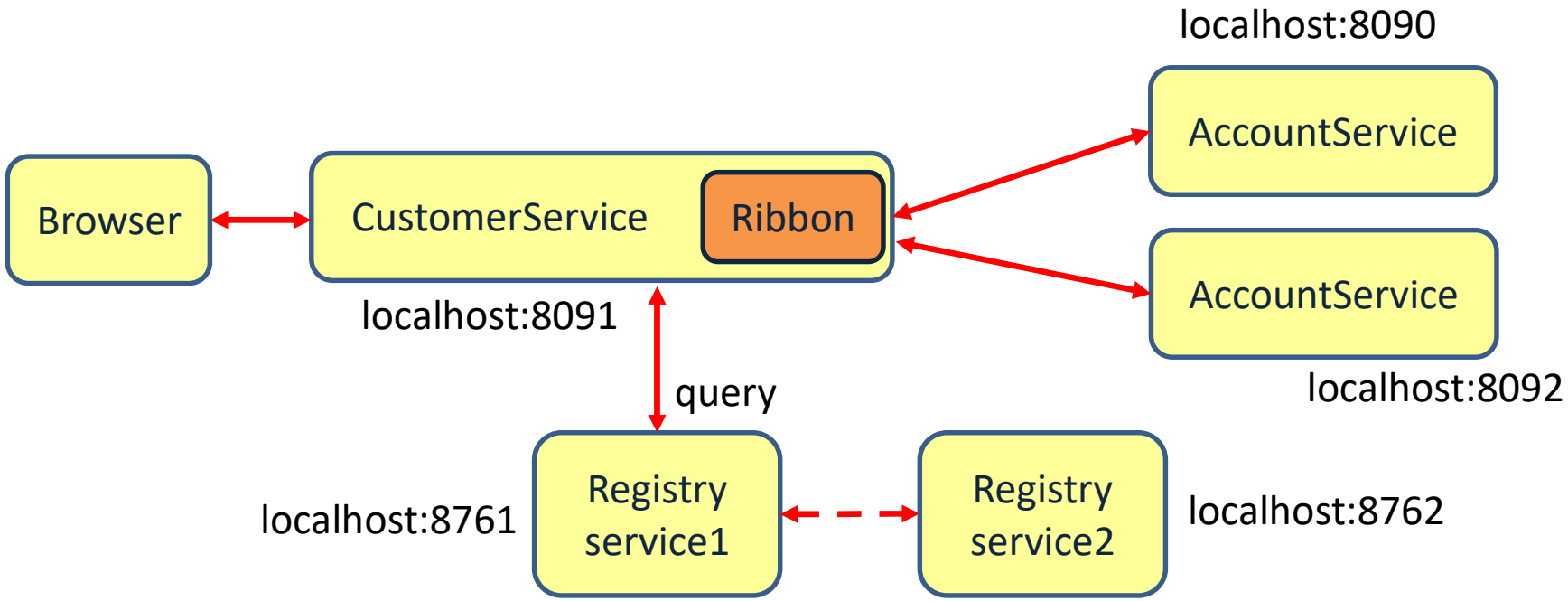
- Single point of failure
- If we add a new instance of AccountService, we need to reconfigure the load balancer
- Extra hop (performance)
- Every microservice needs its own load balancer
- Same load balance algorithm for every client
- Scaling limitation, load balance can handle only a certain number of requests

Client side load balancing



- No single point of failure
- Simplifies service management
- Only one hop (performance)
- Auto discovery with registry based lookup (flexibility)
- Every client can use its own load balancing algorithm
- Unlimited scalable

Load balancing using Ribbon



AccountService

```java
@RestController
public class AccountController {
  @GetMapping("/account/{customerid}")
  public Account getName(@PathVariable("customerid") String customerId) {
    return new Account("1234", "1000.00");
  }
}
```

application.yml

```yaml
server:
  port: 8090

eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/,
                   http://localhost:8762/eureka/

spring:
  application:
    name: AccountService
```

AccountService2

```java
@RestController
public class AccountController {
  @GetMapping("/account/{customerid}")
  public Account getName(@PathVariable("customerid") String customerId) {
    return new Account("1234", "2000.00");
  }
}
```

application.yml

```yaml
server:
 port: 8092

eureka:
 client:
  serviceUrl:
   defaultZone: http://localhost:8761/eureka/,
                http://localhost:8762/eureka/

spring:
 application:
  name: AccountService
```

CustomerService: the controller

```
@RestController
public class CustomerController {
  @Autowired
  AccountFeignClient accountClient;

  @RequestMapping("/customer/{customerid}")
  public Account getName(@PathVariable("customerid") String customerId) {
        Account account = accountClient.getName(customerId);
        return account;
  }

  @FeignClient(name = "account-service")
  interface AccountFeignClient {
    @RequestMapping("/account/{customerid}")
    public Account getName(@PathVariable("customerid") String customerId);
  }
}
```

Feign automatically uses the Ribbon load balancer

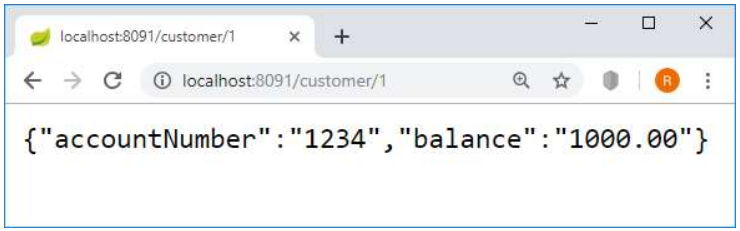
Eureka

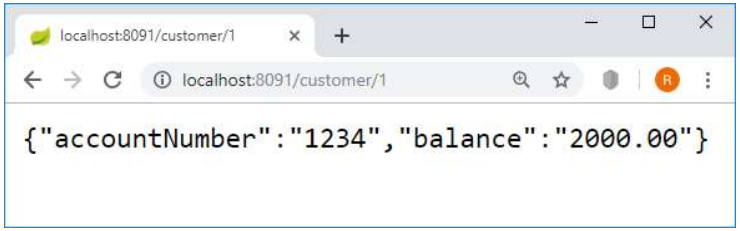
2 instances of accountservice

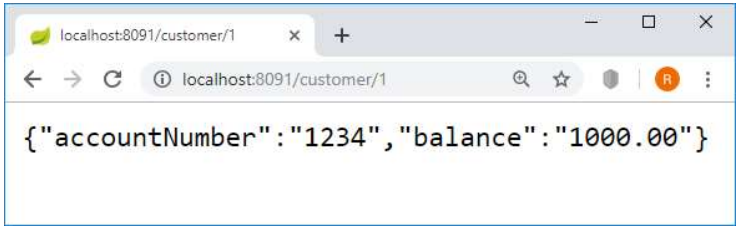
Instances currently registered with Eureka

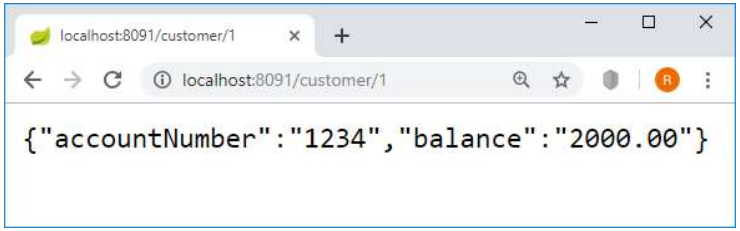
Application	AMIs	Availability Zones	Status
ACCOUNTSERVICE	n/a (2)	(2)	UP (2) - DESKTOP-BVHRK6K.home:AccountService:8092 , DESKTOP-BVHRK6K.home:AccountService:8090
CUSTOMERSERVICE	n/a (1)	(1)	UP (1) - DESKTOP-BVHRK6K.home:CustomerService:8091
EUREKA-CLUSTER	n/a (2)	(2)	UP (2) - DESKTOP-BVHRK6K.home:8762 , DESKTOP-BVHRK6K.home:8761

Round robin

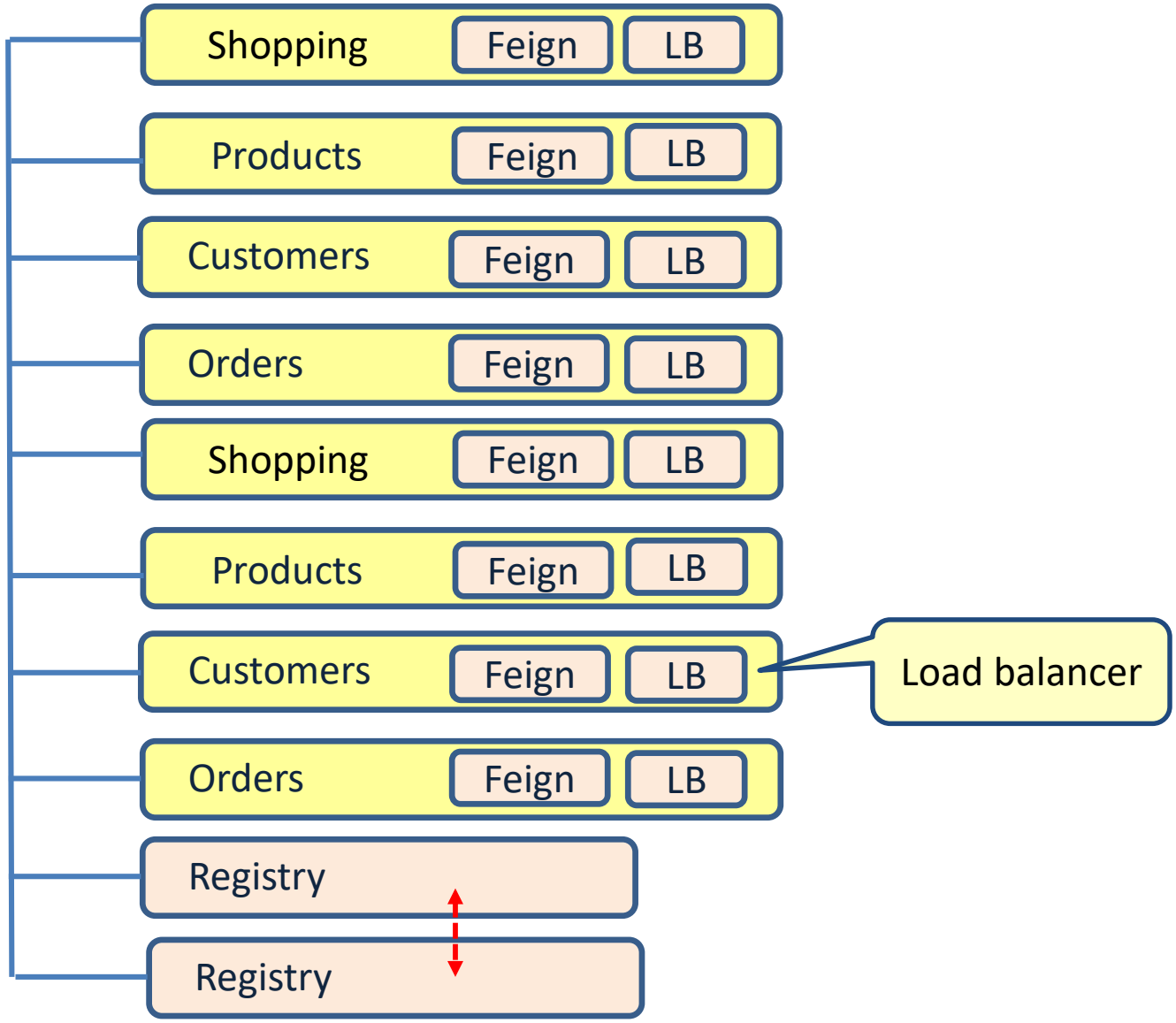








Implementing microservices

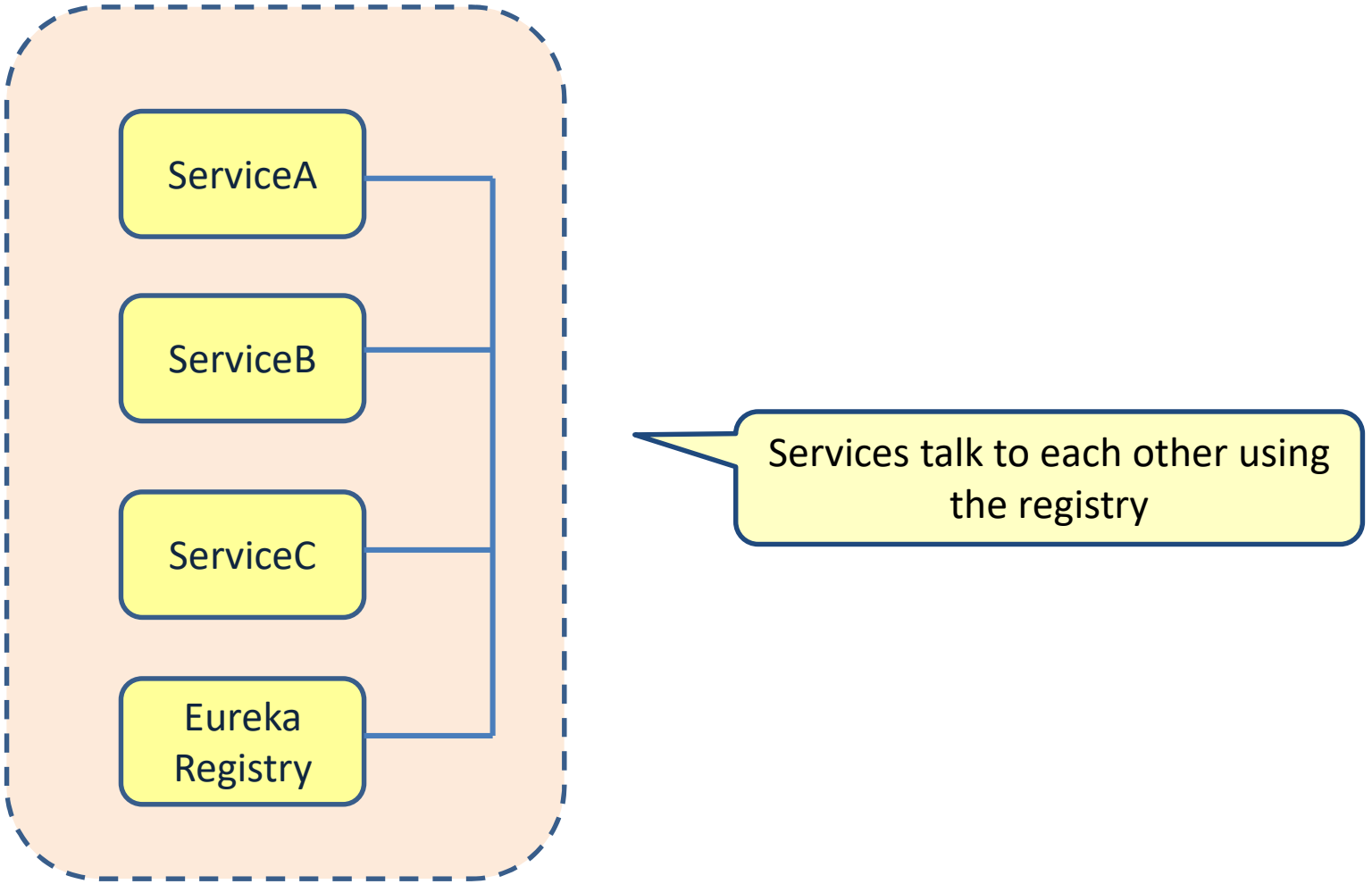


Challenges of a microservice architecture

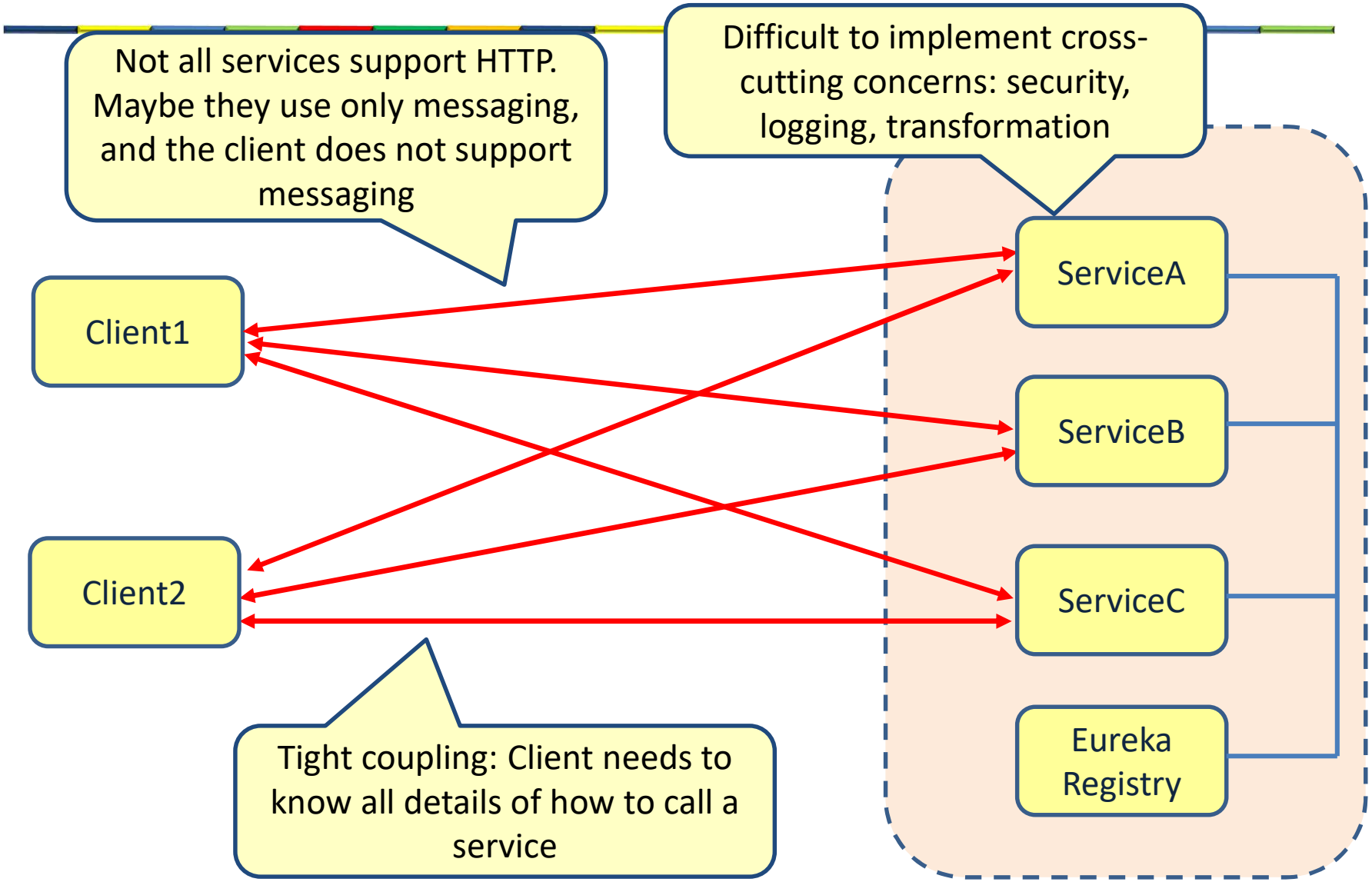
Challenge	Solution
Complex communication	Feign Registry
Performance	
Resilience	Registry replicas Load balancing between multiple service instances
Security	
Transactions	
Following the process	
Keep data in sync	
Keep interfaces in sync	
Keep configuration in sync	
Monitor health of microservices	
Follow/monitor business processes	

API GATEWAY

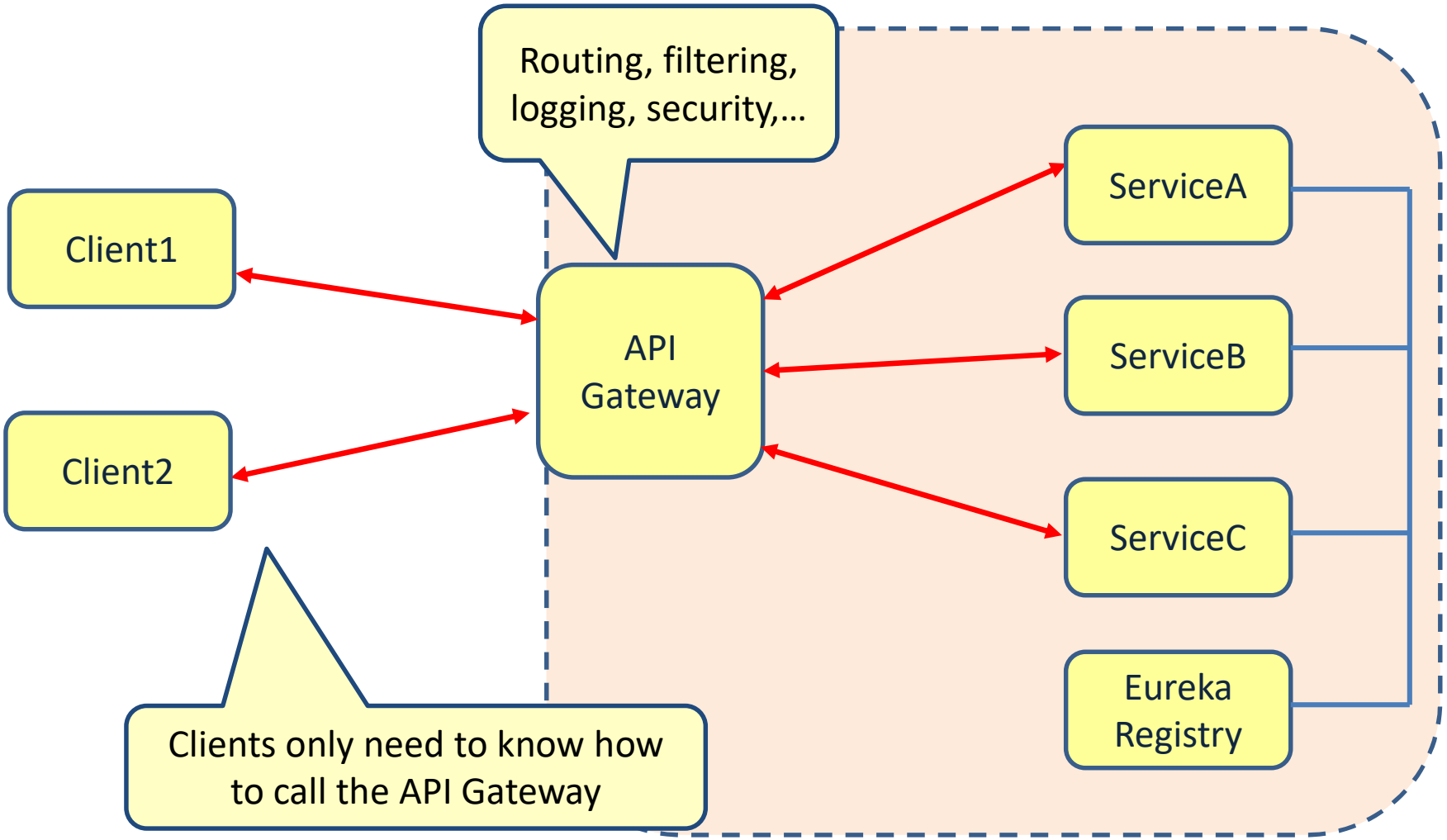
Microservice architecture



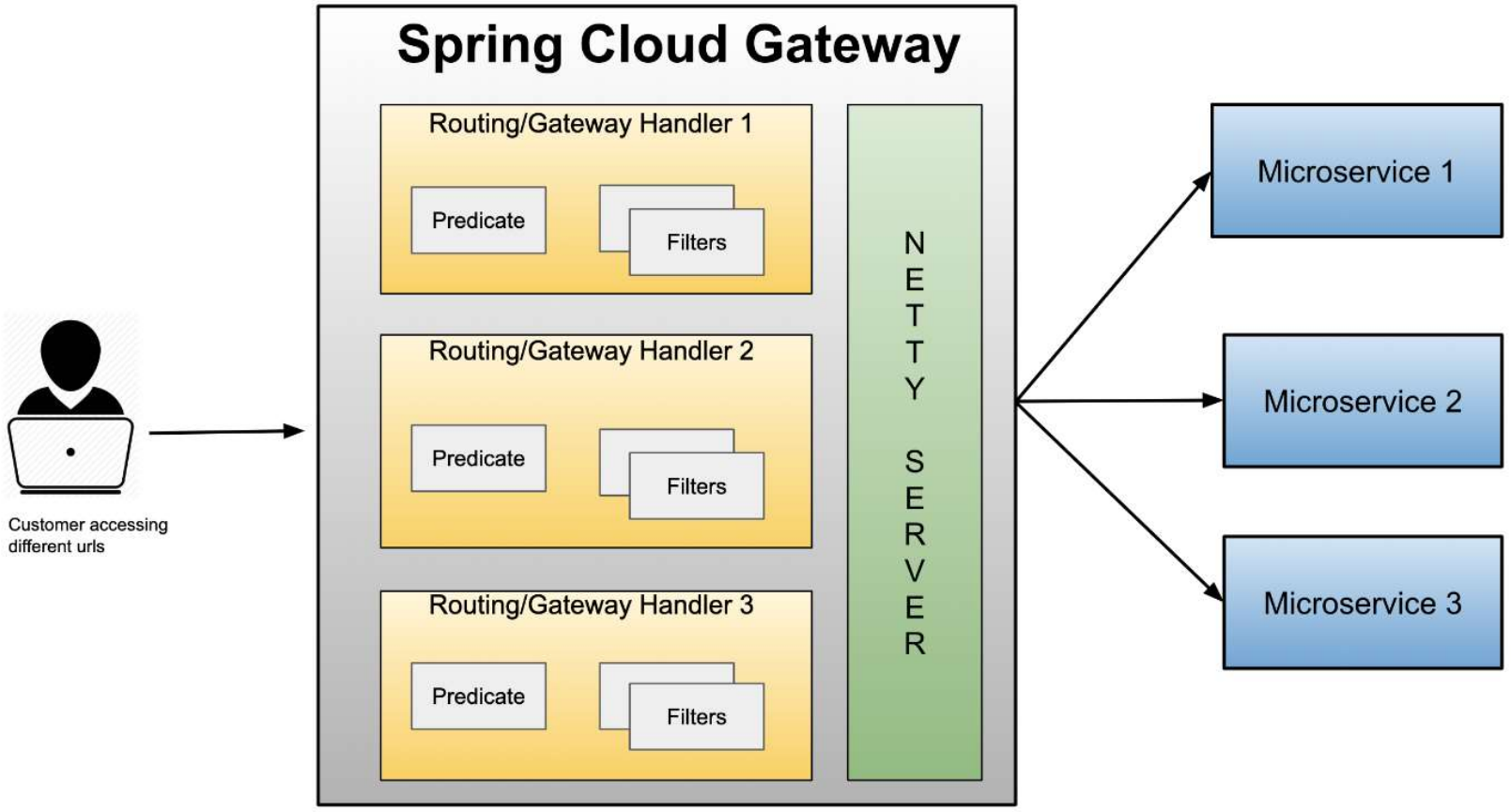
Adding clients



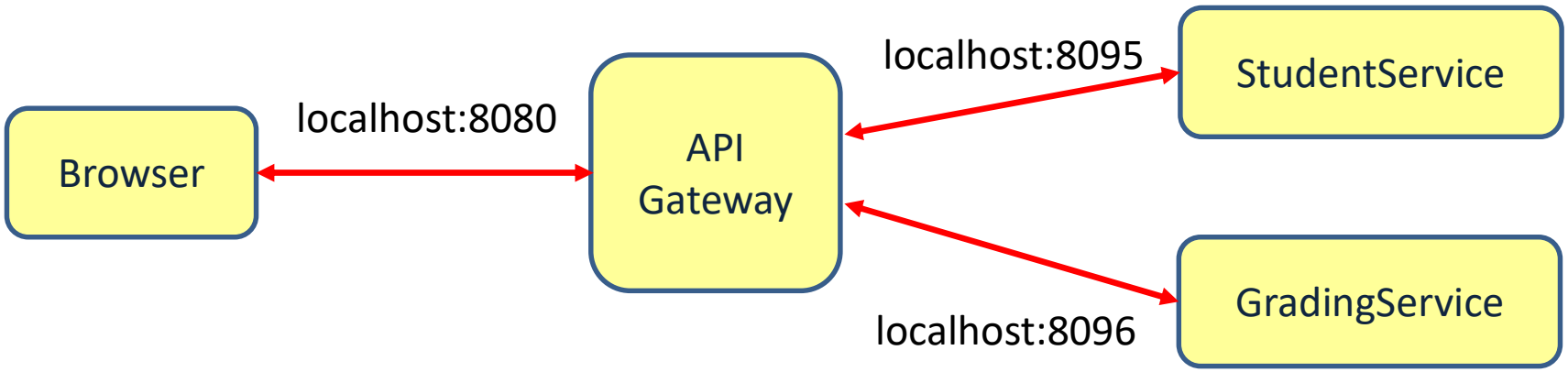
Api Gateway



Spring cloud gateway



Api Gateway example



StudentService

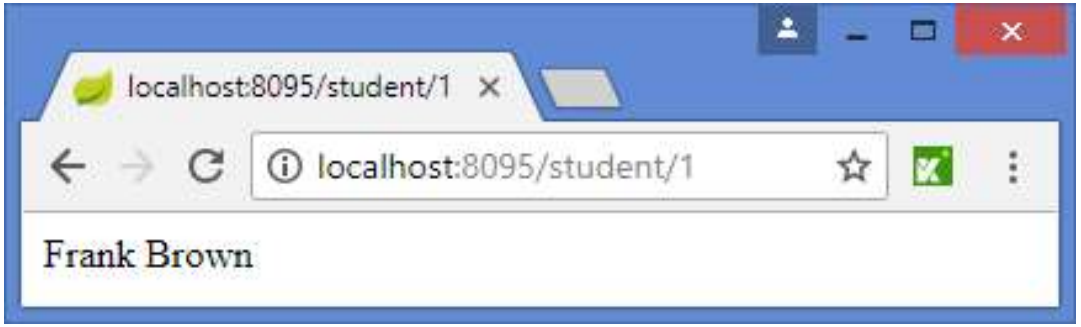
```java
@SpringBootApplication
@EnableDiscoveryClient
public class StudentServiceApplication {

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

application.yml

```yaml
server:
  port: 8095

spring:
  application:
    name: StudentService

eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

StudentService: the controller

```
@RestController
public class StudentController {
  @GetMapping("/student/{studentid}")
  public String getName(@PathVariable("studentid") String studentid) {
    return "Frank Brown";
  }
}
```



GradingService

```
@SpringBootApplication
@EnableDiscoveryClient
public class GradingServiceApplication {

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

application.yml

```
server:
  port: 8096

spring:
  application:
    name: GradingService

eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

GradingService: the controller

```
@RestController
public class GradingController {
  @GetMapping("/grade/{studentid}/{courseid}")
  public String getGrade(@PathVariable("studentid") String studentid,
                         @PathVariable("courseid") String courseid) {
    return "A+";
  }
}
```

Spring Cloud Gateway

```
@SpringBootApplication
public class CloudgatewayApplication {

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

POM.xml

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

Spring Cloud Gateway

application.yml

```yaml
spring:
  application:
    name: api-gateway
  cloud:
    gateway:
      routes:
      - id: studentModule
        uri: http://localhost:8095/
        predicates:
        - Path=/student/**
      - id: gradingModule
        uri: http://localhost:8096/
        predicates:
        - Path=/grade/**

server:
  port: 8080
```

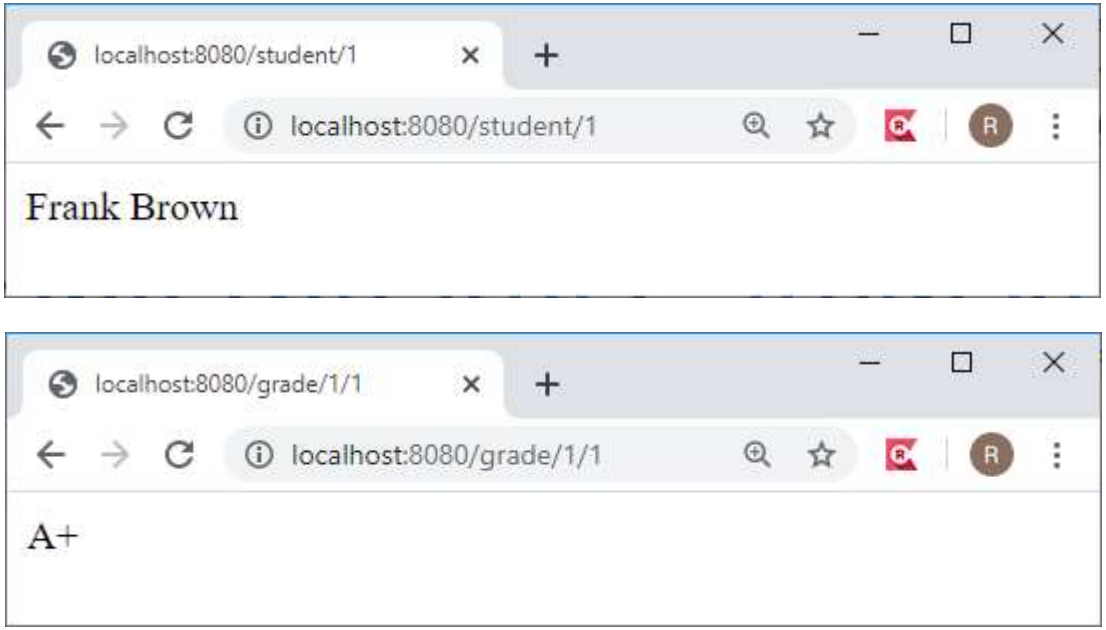
Id should be unique for every route

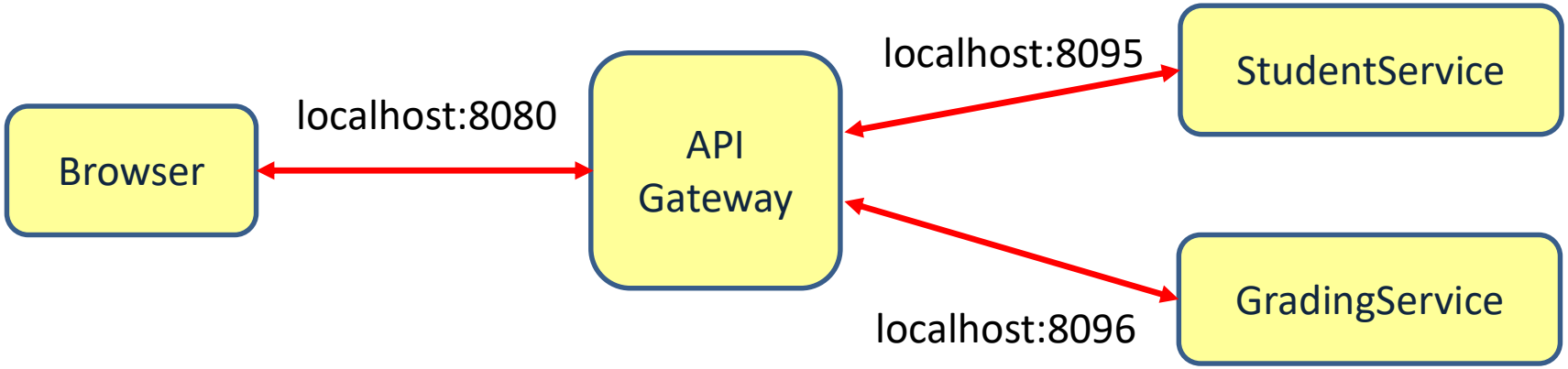
Route /student to localhost:8095

Route /grades to localhost:8096

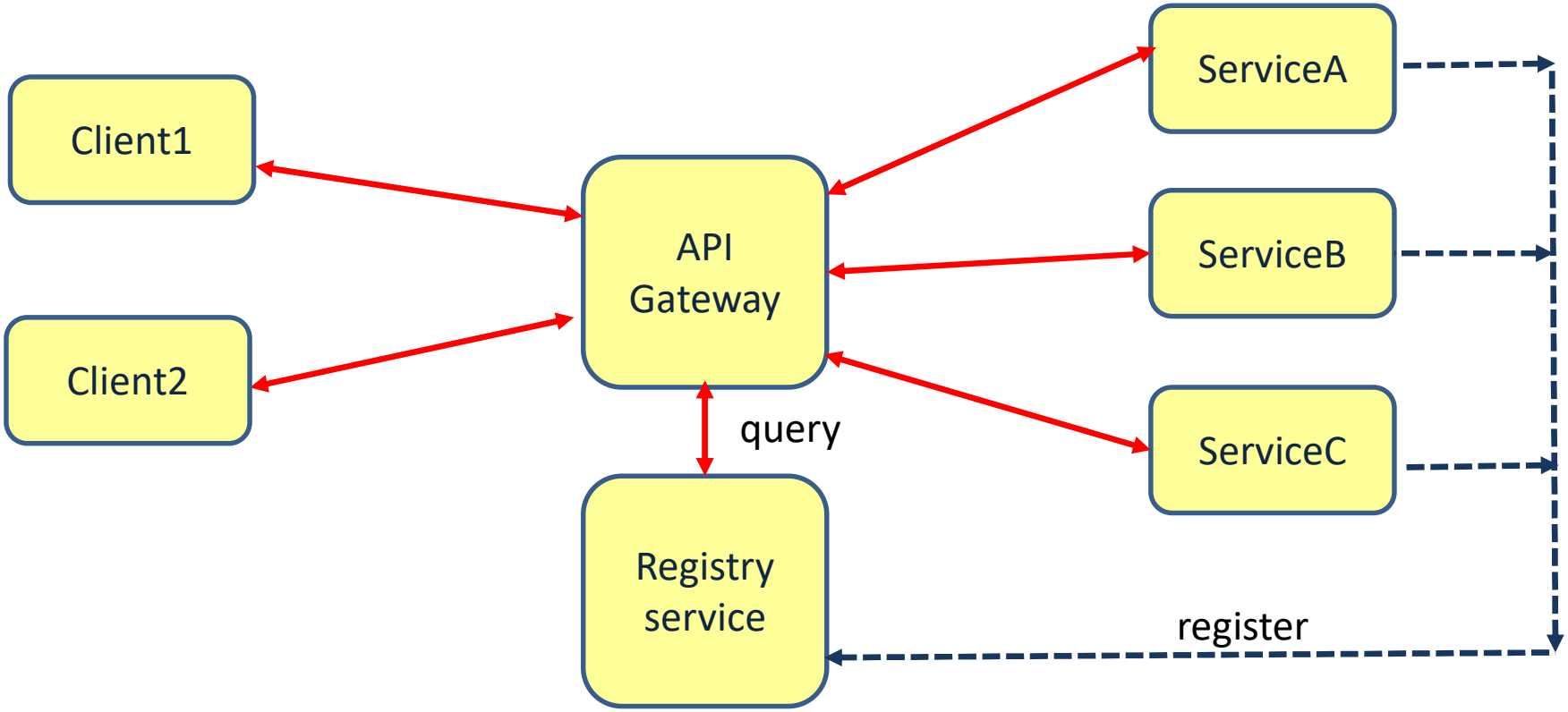
A route is matched if predicate is true

Using the API Gateway





Api Gateway and registry service



Spring cloud gateway with the registry

application.yml

```yaml
spring:
  application:
    name: api-gateway
  cloud:
    gateway:
      routes:
      - id: studentModule
        uri: lb://StudentService
        predicates:
        - Path=/student/**
      - id: gradingModule
        uri: lb://GradingService
        predicates:
        - Path=/grade/**

server:
  port: 8080

eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

Route /student to the service with the name StudentService using the load balancer

Route /grade to the service with the name GradingService using the load balancer

URL to Eureka

Java based config

```
@Configuration
public class BeanConfig {

  @Bean
  public RouteLocator gatewayRoutes(RouteLocatorBuilder builder) {
    return builder.routes()
                .route(r -> r.path("/student/**")
                        .uri("lb://StudentService")
                        .id("studentModule"))

                .route(r -> r.path("/grade/**")
                        .uri("lb://GradingService")
                        .id("gradesModule"))
                .build();
  }
}
```

Build-in predicates

Name	Description	Example
After Route	It takes a date-time parameter and matches requests that happen after it	`After=2017-11-20T...`
Before Route	It takes a date-time parameter and matches requests that happen before it	`Before=2017-11-20T...`
Between Route	It takes two date-time parameters and matches requests that happen between those dates	`Between=2017-11-20T..., 2017-11-21T...`
Cookie Route	It takes a cookie name and regular expression parameters, finds the cookie in the HTTP request's header, and matches its value with the provided expression	`Cookie=SessionID, abc.`
Header Route	It takes the header name and regular expression parameters, finds a specific header in the HTTP request's header, and matches its value with the provided expression	`Header=X-Request-Id, \d+`
Host Route	It takes a hostname ANT style pattern with the . separator as a parameter and matches it with the `Host` header	`Host=**.example.org`
Method Route	It takes an HTTP method to match as a parameter	`Method=GET`
Path Route	It takes a pattern of request context path as a parameter	`Path=/account/{id}`
Query Route	It takes two parameters—a required param and an optional regexp and matches them with query parameters	`Query=accountId, 1.`
RemoteAddr Route	It takes a list of IP addresses in CIDR notation, like `192.168.0.1/16`, and matches it with the remote address of a request	`RemoteAddr=192.168.0.1/16`

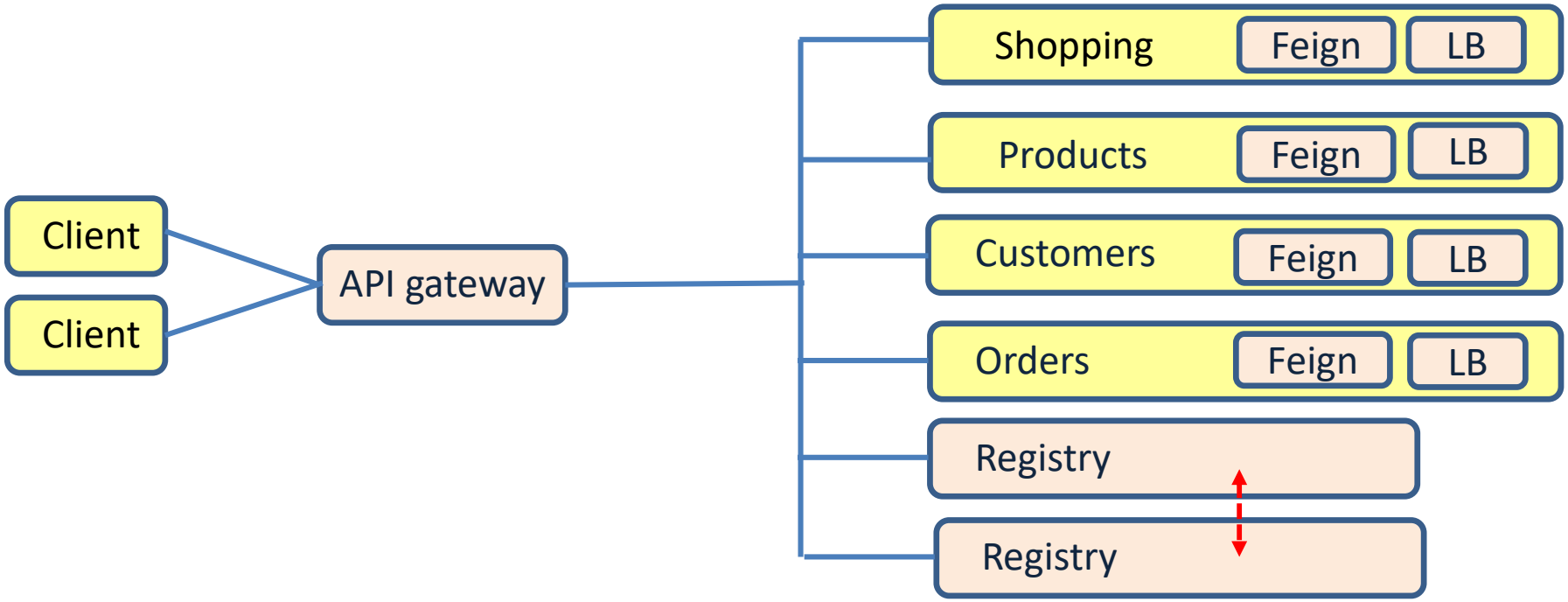
Filters

- Pre and post filters
  - Build-in filters
  - Custom filters
  - Global filters

```
cloud:
    gateway:
      routes:
      - id: studentModule
        uri: lb://StudentService
        filters:
        - AddRequestHeader=X-myHeader, Hello
        - AddRequestParameter=name, John
        - AddResponseHeader=X-someHeader, Hello World
        predicates:
        - Path=/student/**
```

Build-in filters

Implementing microservices



Challenges of a microservice architecture

Challenge	Solution
Complex communication	Feign Registry API gateway
Performance	
Resilience	Registry replicas Load balancing between multiple service instances
Security	
Transactions	
Following the process	
Keep data in sync	
Keep interfaces in sync	
Keep configuration in sync	
Monitor health of microservices	
Follow/monitor business processes	

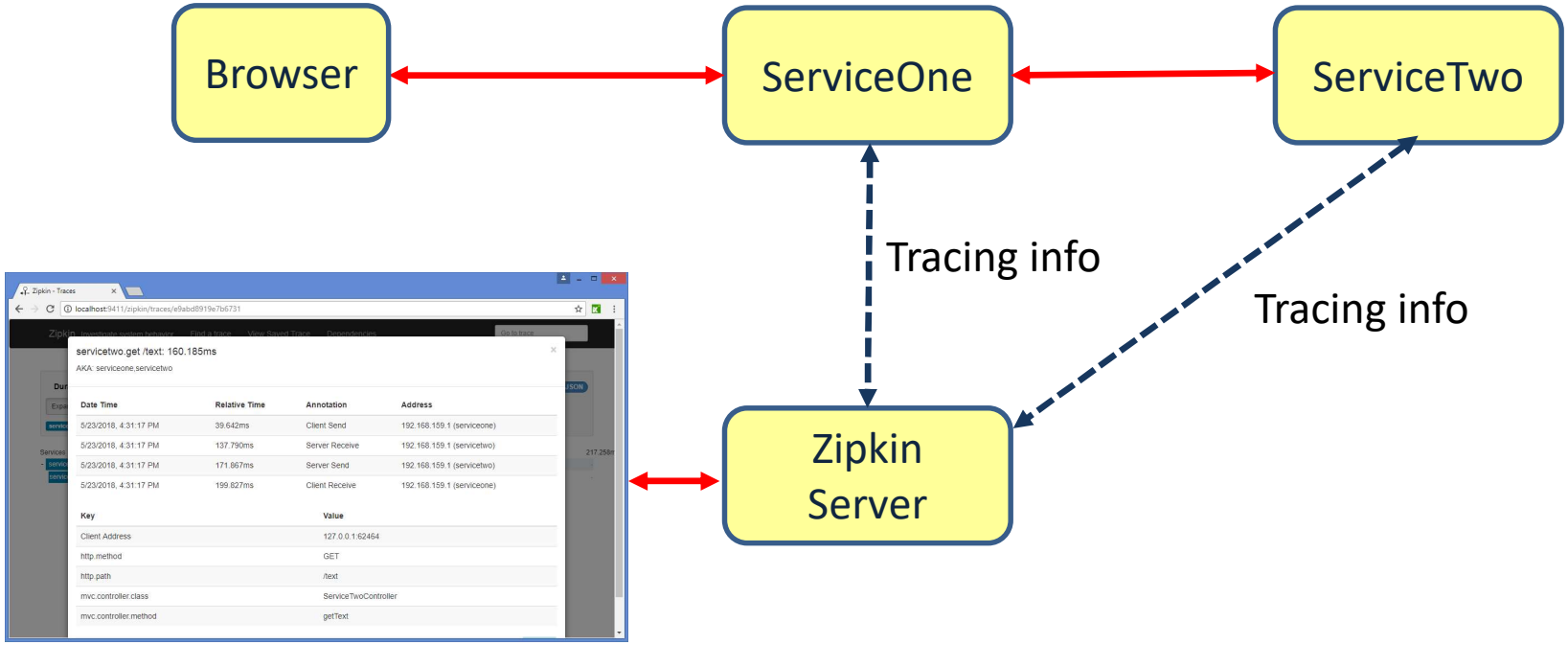
Lesson 9

MICROSERVICES

DISTRIBUTED TRACING: ZIPKIN

Distributed Tracing

- One central place where one can see the end-to-end tracing of all communication between services



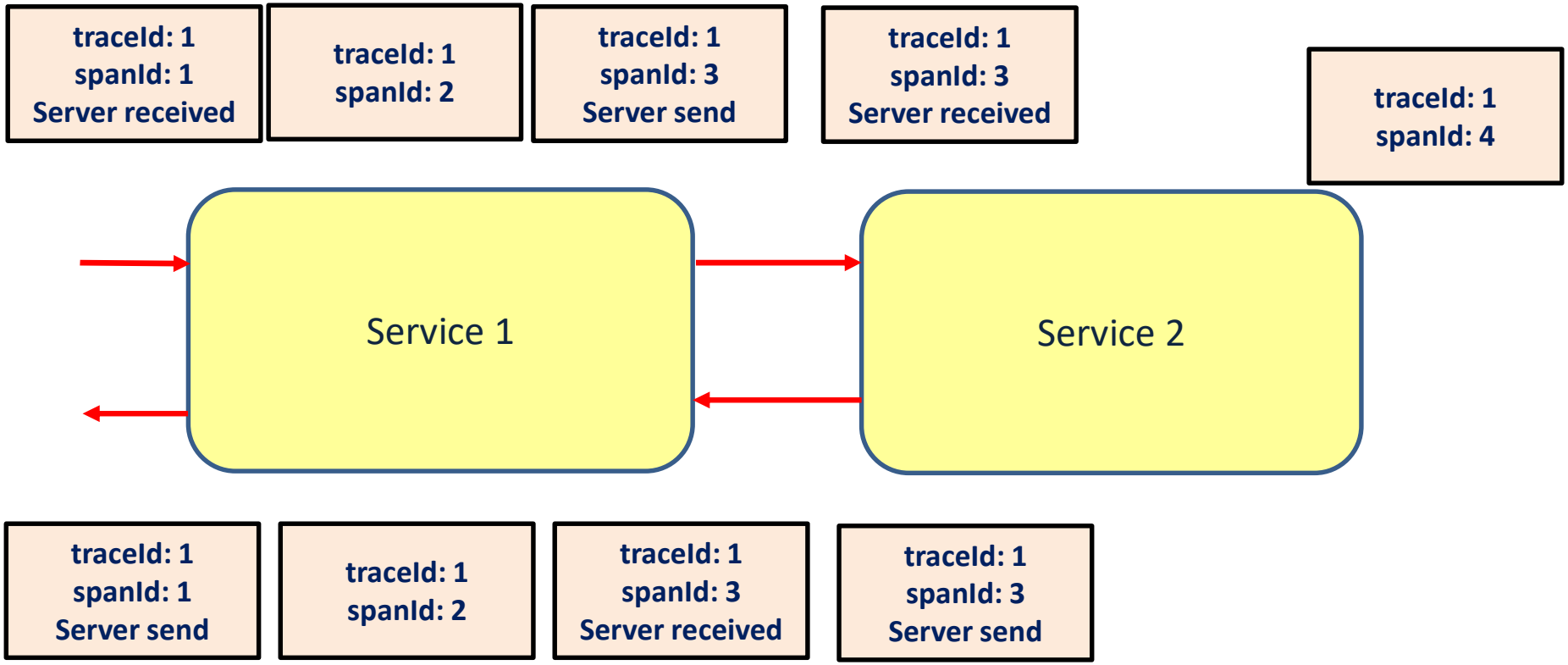
Spring cloud Sleuth

- Adds unique id's to a request so we can trace the request
  - Span id: id for an individual operation
  - Trace id: id for a set of spans

- Also embeds these unique id's to log messages

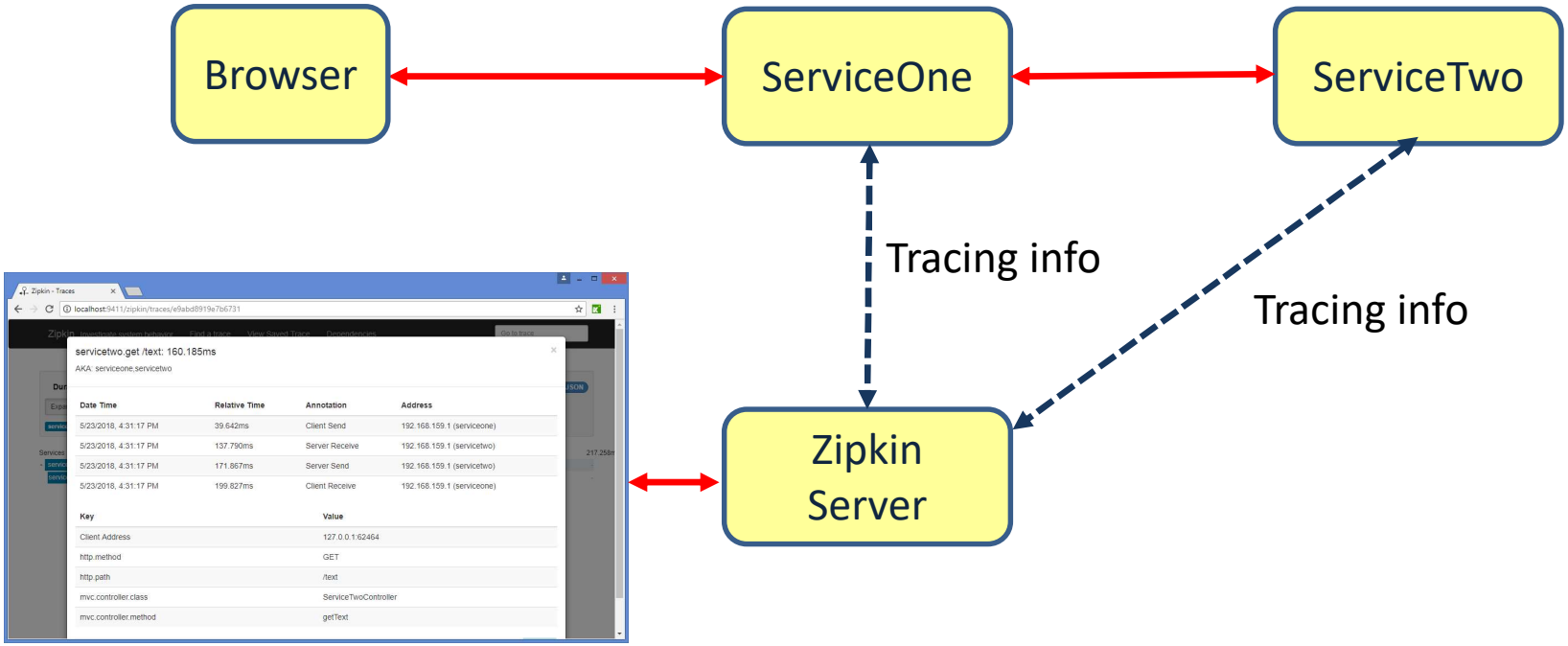
Spring cloud Sleuth

- Span: an individual operation
- Trace: a set of spans



Zipkin

- Centralized tracing server
  - Collects tracing information
- Zipkin console shows the data



Service1

```
@SpringBootApplication
public class Service1Application {
  public static void main(String[] args) {
    SpringApplication.run(Service1Application.class, args);
  }
}
```

```
@RestController
public class ServiceOneController {

  @Autowired
  RestTemplate restTemplate;

  @RequestMapping("/text")
  public String getText() {
    String service2Text = restTemplate.getForObject("http://localhost:9091/text",
                                                    String.class);
    return "Hello "+ service2Text;
  }

  @Bean
  RestTemplate getRestTemplate() {
    return new RestTemplate();
  }
}
```

Service1

application.yml

```yaml
server:
  port: 9090

spring:
  application:
    name: ServiceOne
  zipkin:
    base-url: http://localhost:9411/

  sleuth:
    sampler:
      probability: 1 #100% (default = 10%)
```

Service1

pom.xml

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-zipkin</artifactId>
  <version>2.2.3.RELEASE</version>
</dependency>
```

Service2

```java
@SpringBootApplication
public class Service2Application {
  public static void main(String[] args) {
    SpringApplication.run(Service2Application.class, args);
  }
}
```

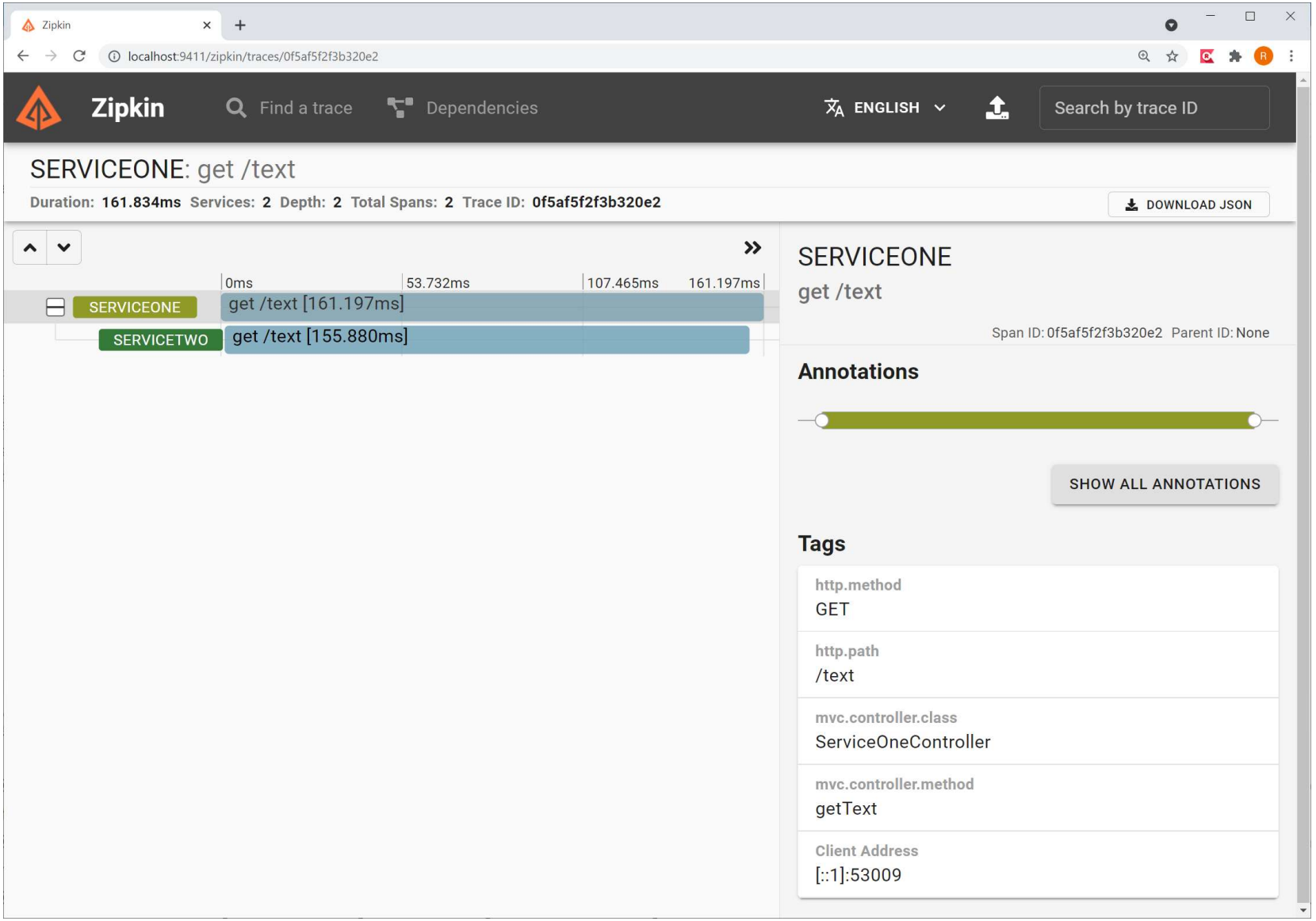
```java
@RestController
public class ServiceTwoController {

  @RequestMapping("/text")
  public String getText() {
    return "World";
  }
}
```

Service2

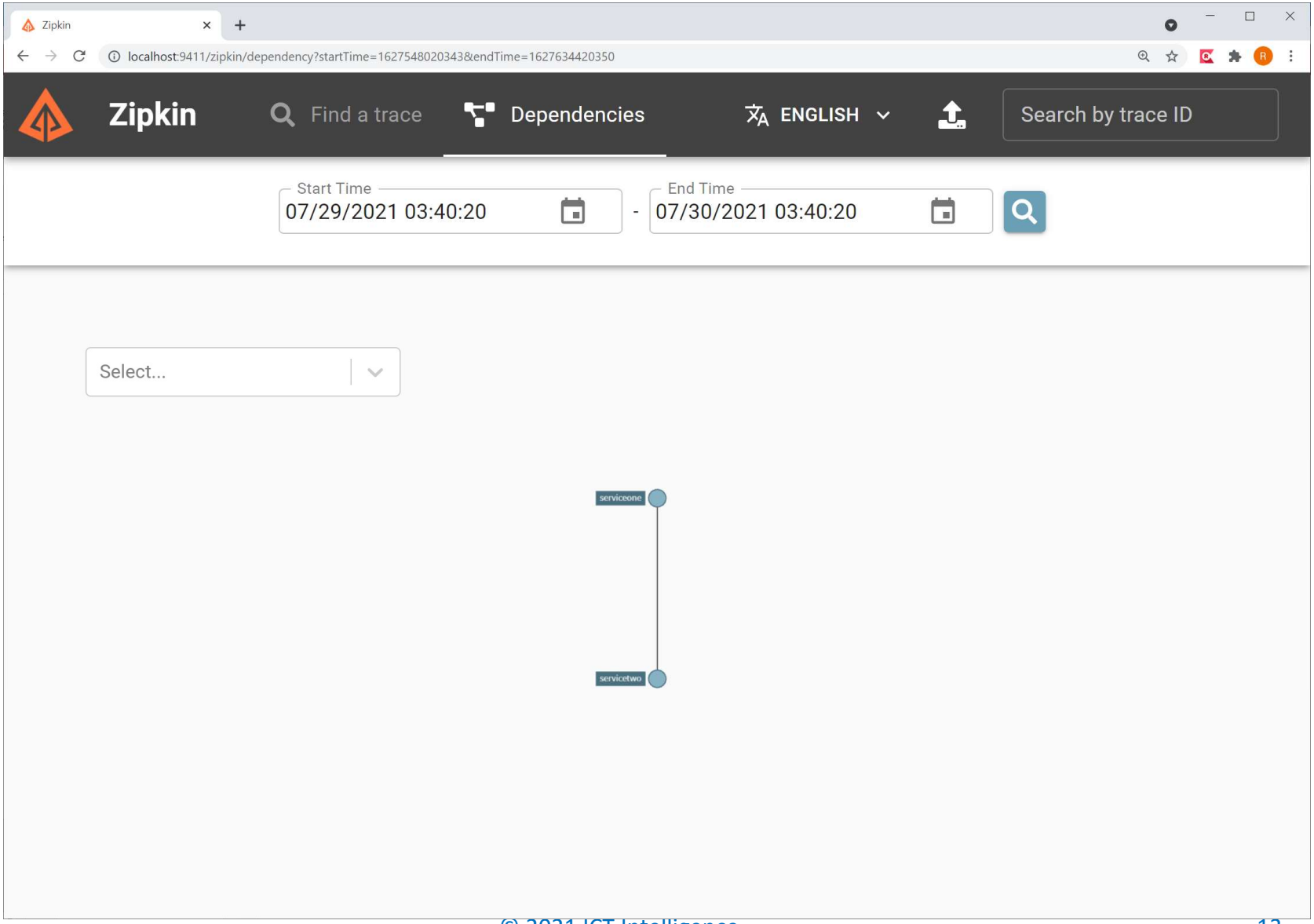
application.yml

```yaml
server:
  port: 9091

spring:
  application:
    name: ServiceTwo
  zipkin:
    base-url: http://localhost:9411/

sleuth:
  sampler:
    probability: 1 #100% (default = 10%)
```

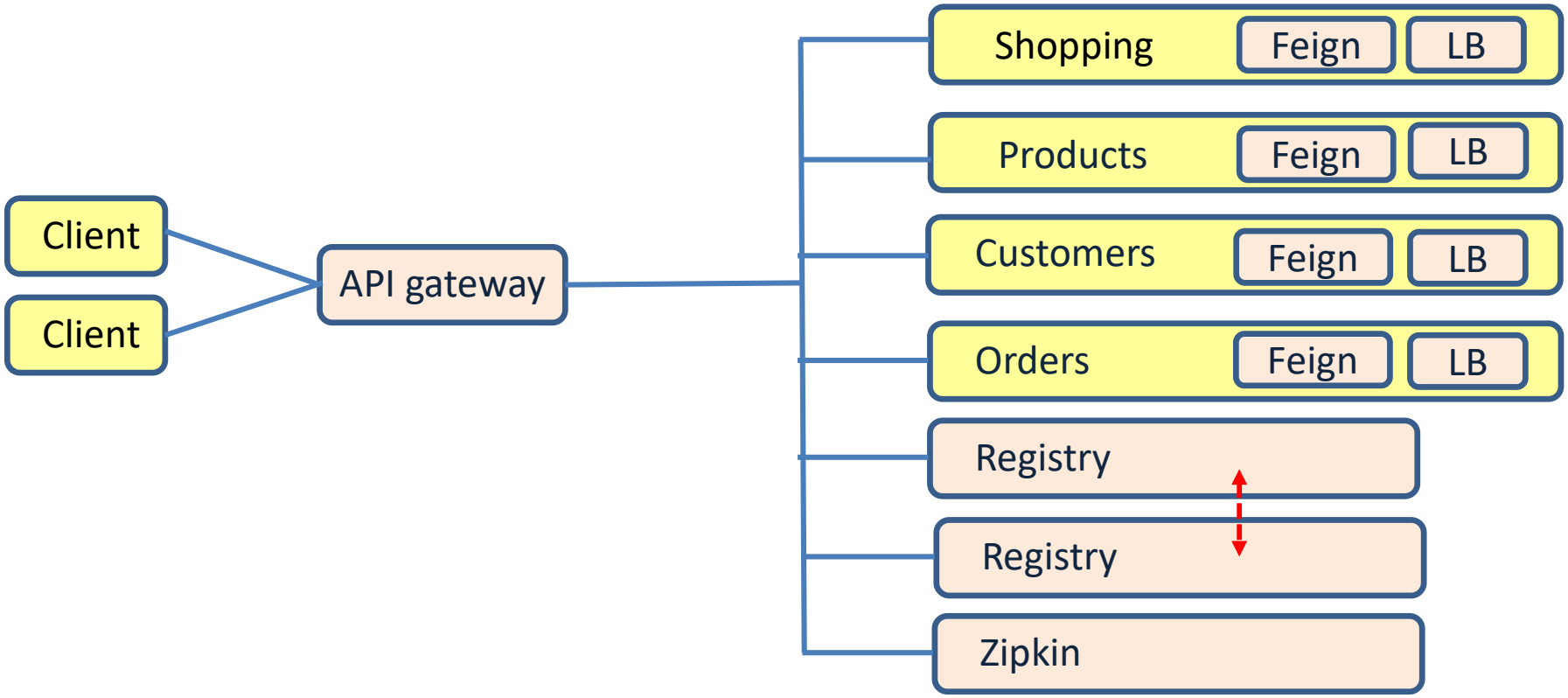
Zipkin console



Zipkin console



Implementing microservices



Challenges of a microservice architecture

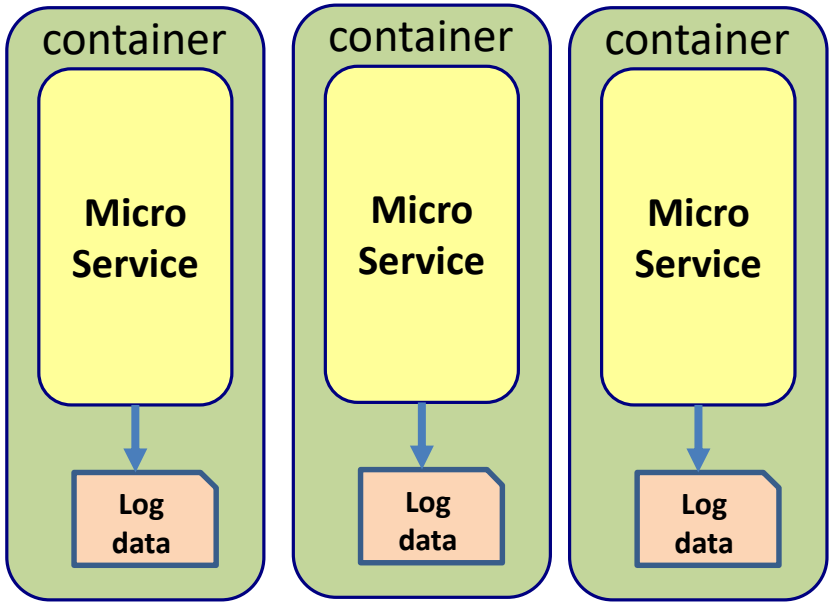
Challenge	Solution
Complex communication	Feign Registry API gateway
Performance	
Resilience	Registry replicas Load balancing between multiple service instances
Security	
Transactions	
Following the process	
Keep data in sync	
Keep interfaces in sync	
Keep configuration in sync	
Monitor health of microservices	
Follow/monitor business processes	Zipkin

Main point

- We need zipkin in order to monitor and debug service-to-service communication
- The Unified Field is the field of perfection

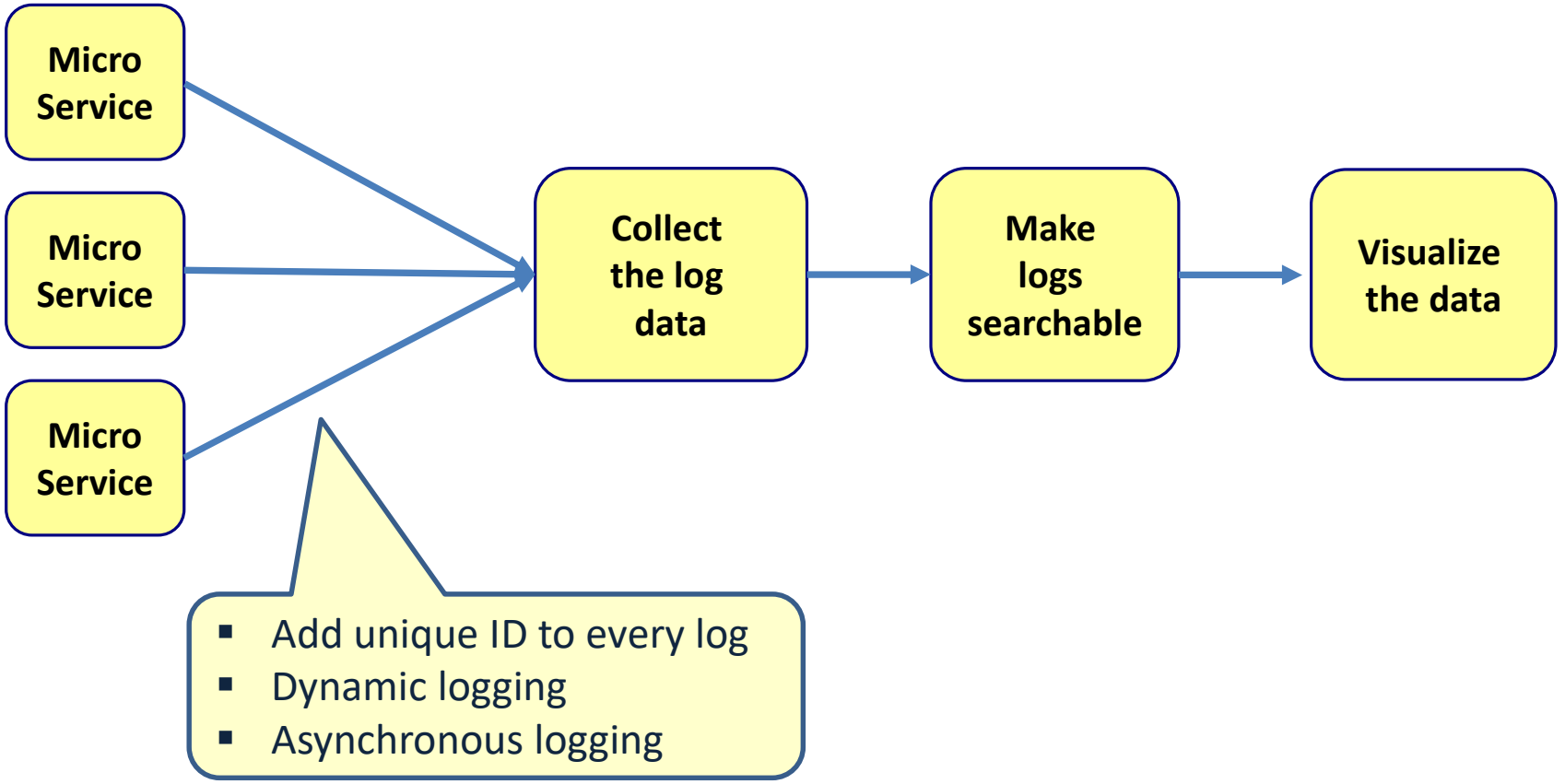
DISTRIBUTED LOGGING
ELK STACK

The need for centralized logging



- Local logging does not work
  - Containers come and go
  - Containers have no fixed address

Microservice logging architecture



ELK stack

Micro Service

Micro Service

Micro Service

Collect the log data

Make logs searchable

Visualize the data

logstash

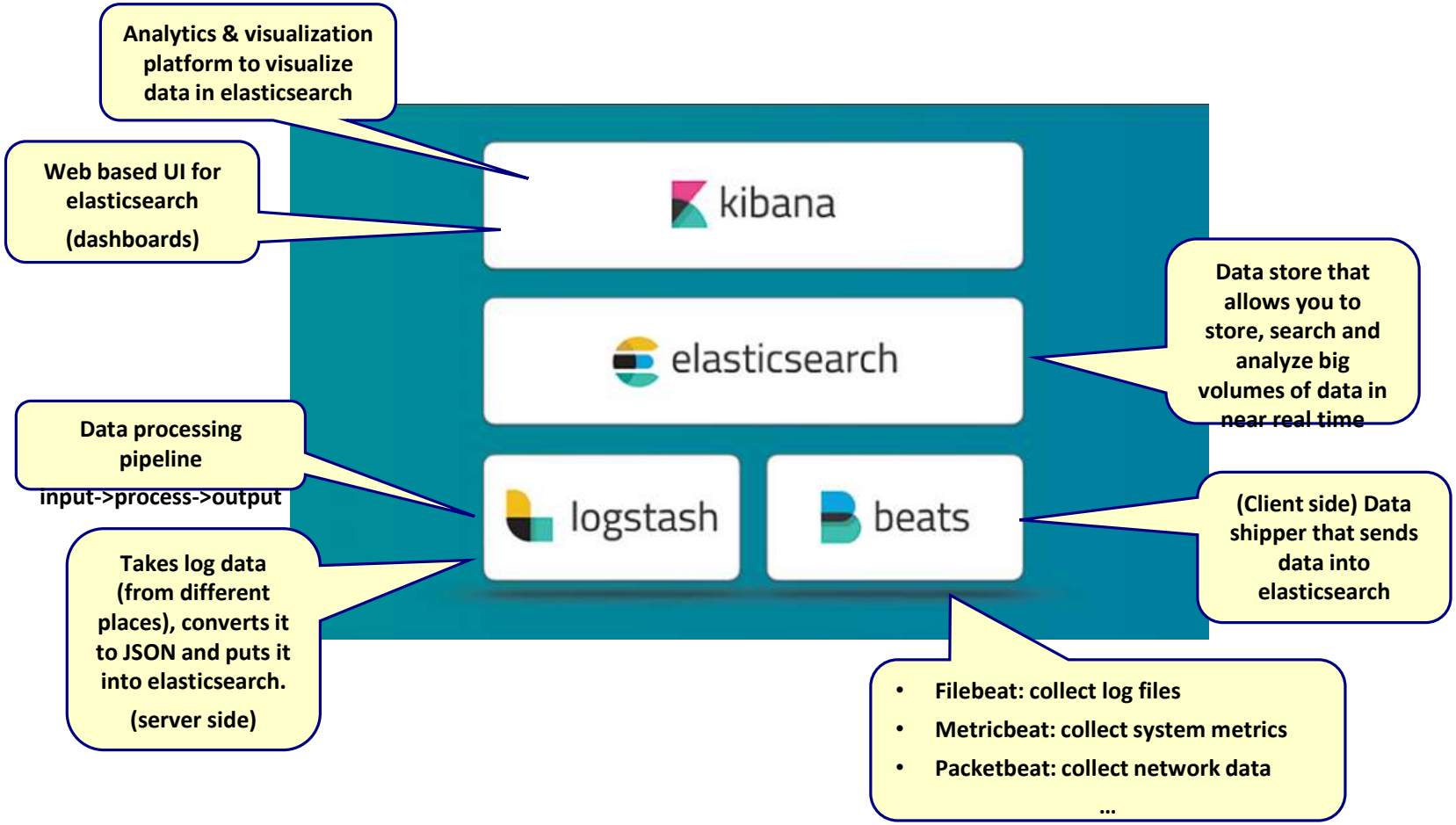
elasticsearch

kibana

Collect logs
Transform
Time normalization

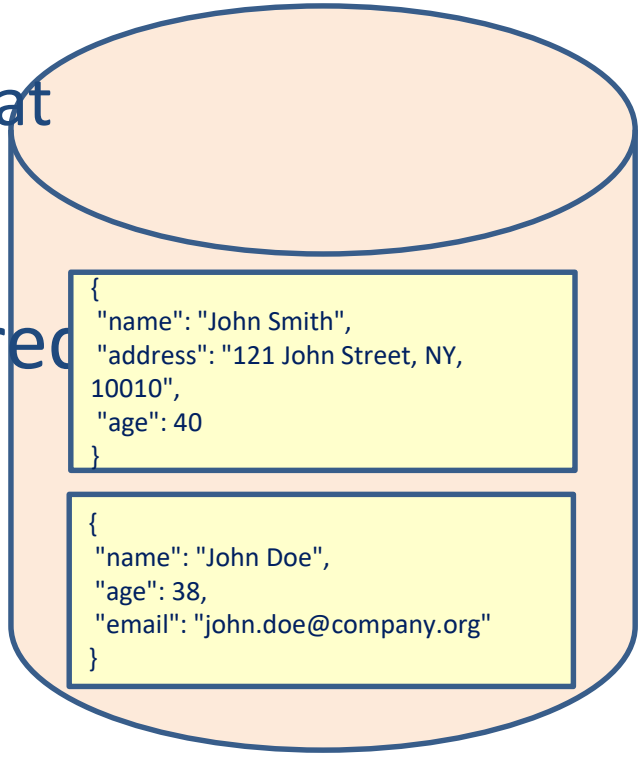
Schema less search DB
Highly scalable
Fast searching

Elastic stack components



What is Elasicsearch?

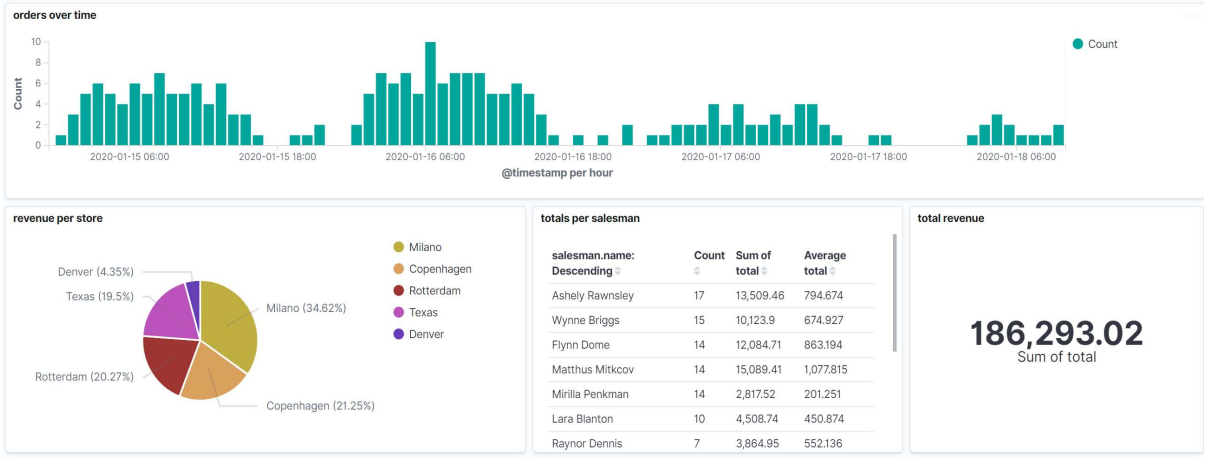
- Database
  - Data is stored as documents
  - Data is structured in JSON format
- Full text search engine
- Analytics platform for structured



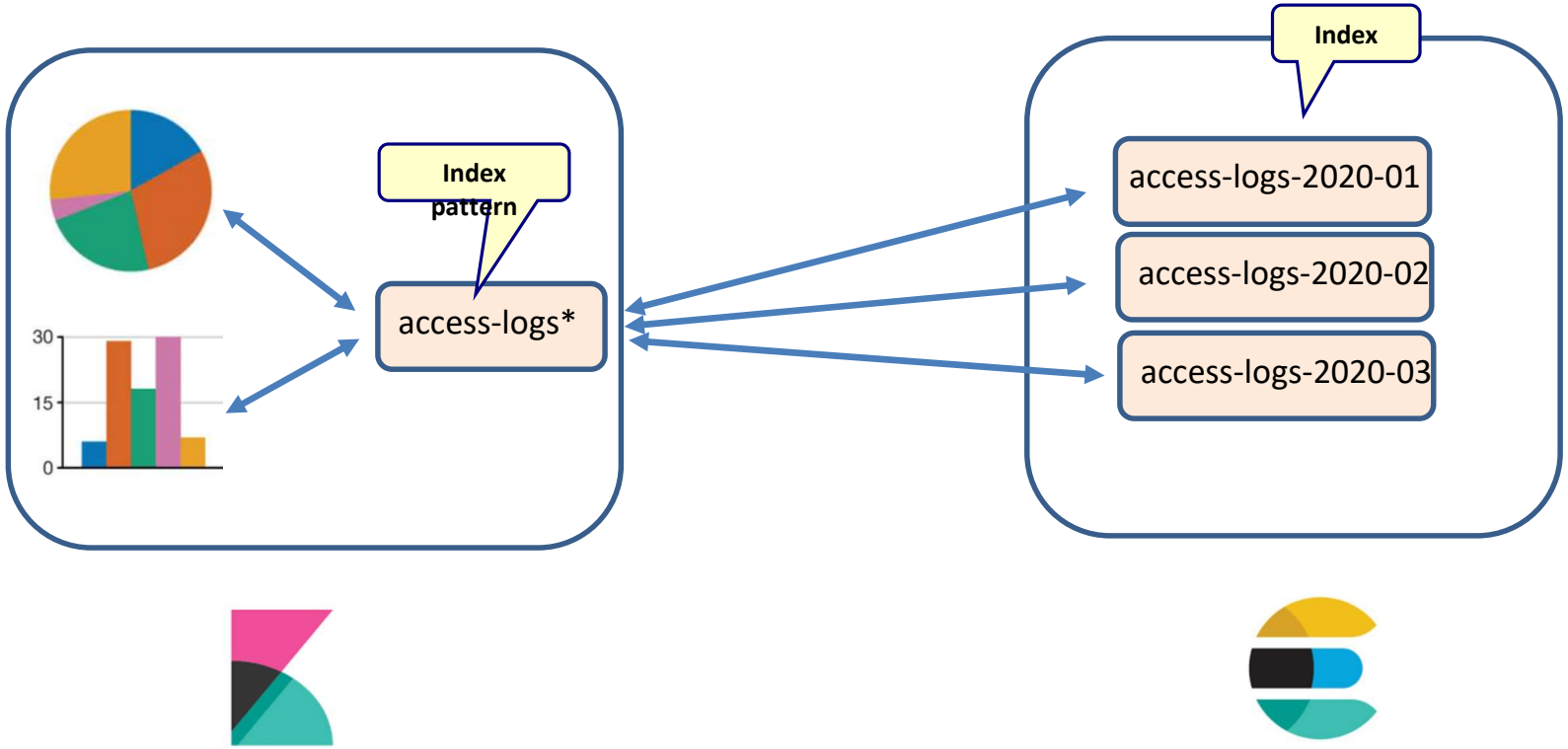
KIBANA

Kibana

- Web UI on top of elasticsearch
- Has its own Kibana query language (KQL)
- Objects (Queries, visualizations, dashboards, etc.) are saved in elasticsearch

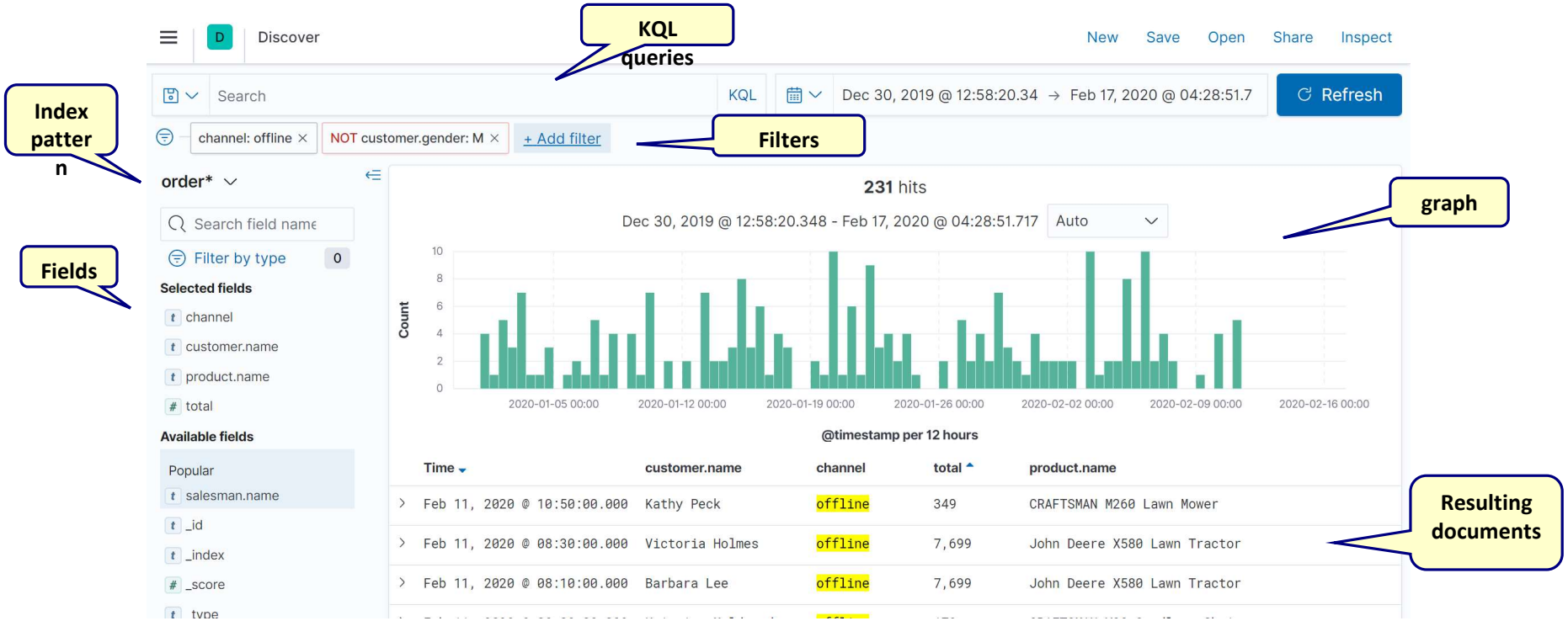


Index patterns

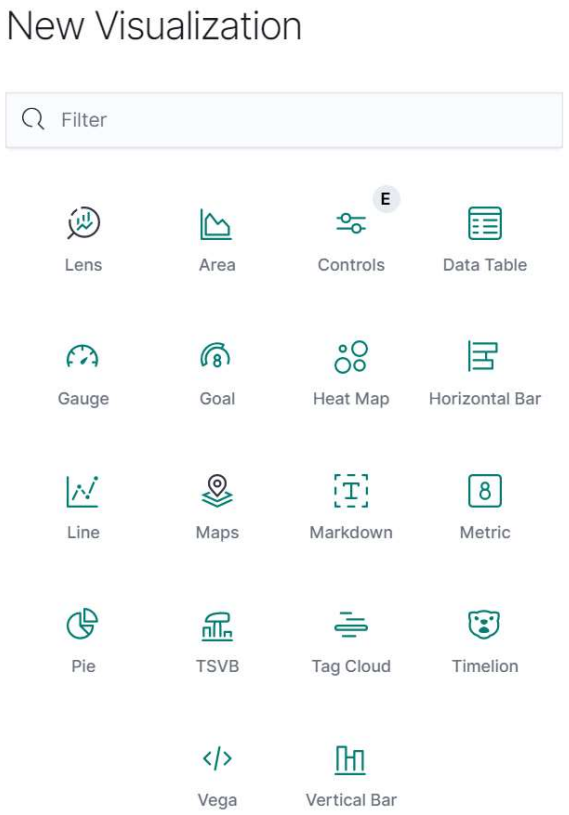


Discover app

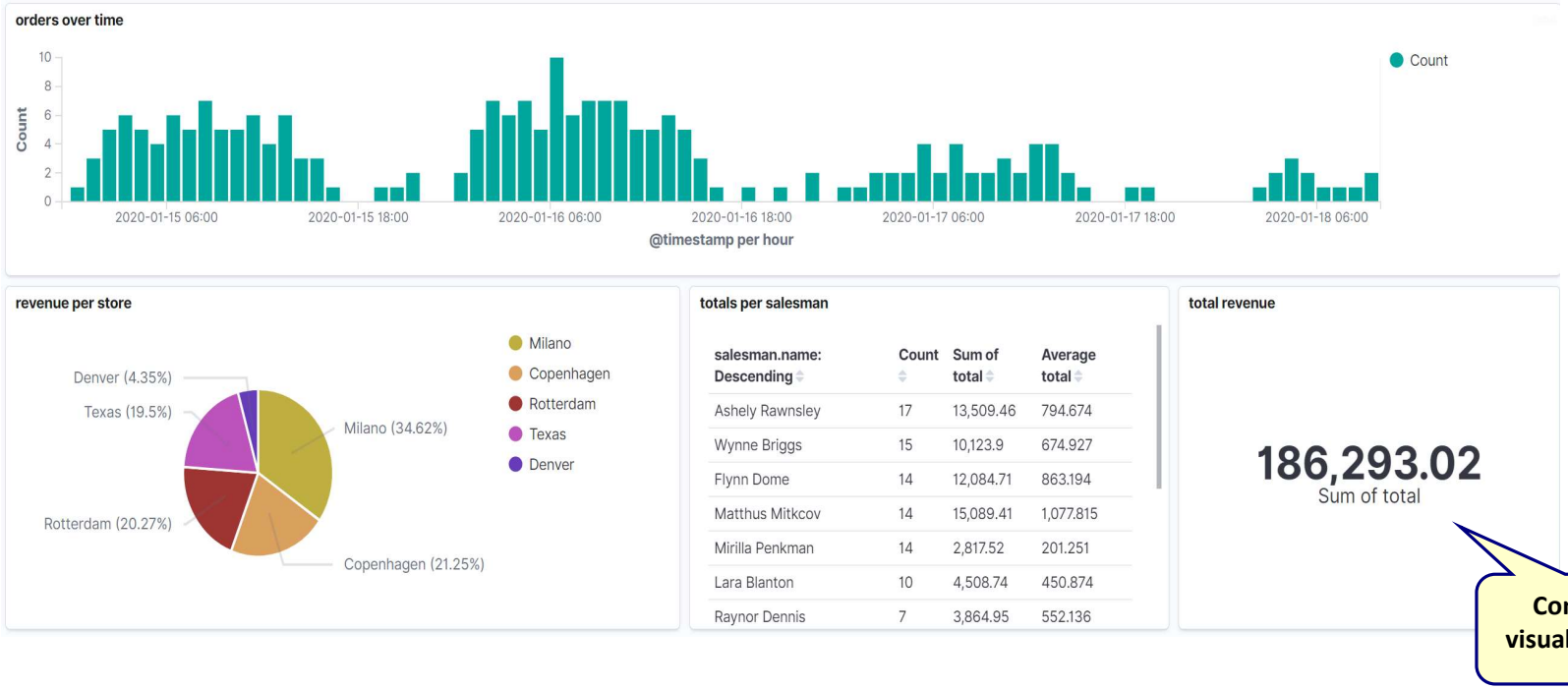
- Good for exploring and analyzing data



Visualizations



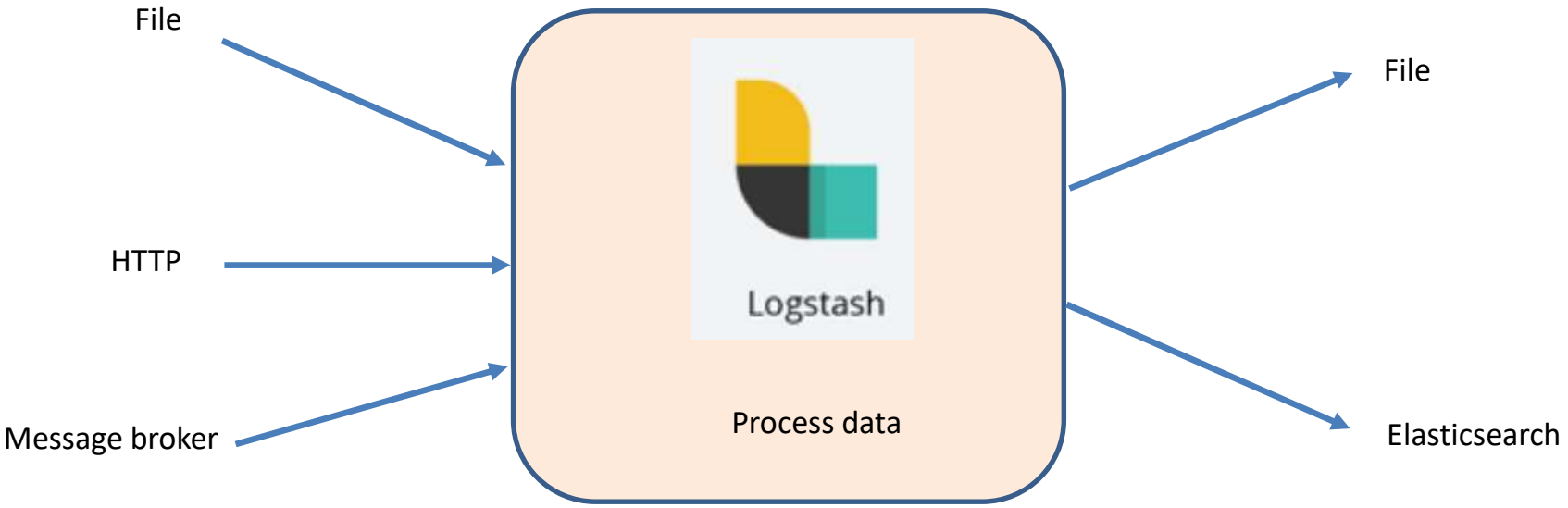
Dashboard



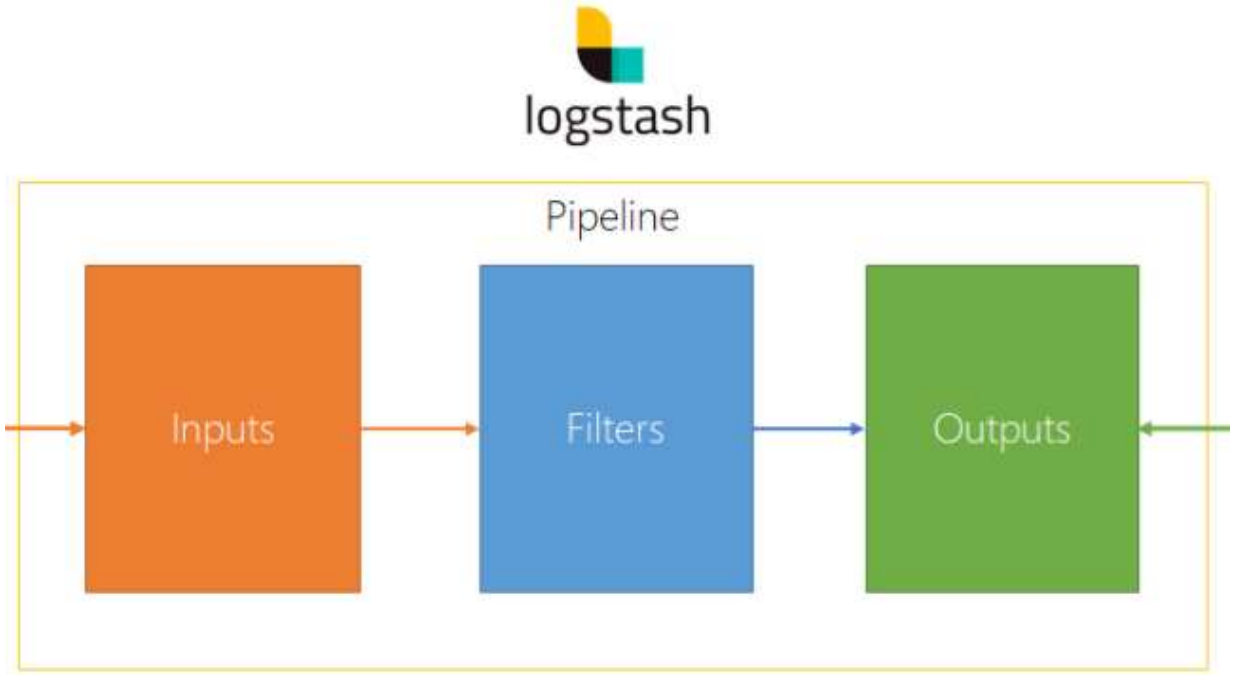
LOGSTASH

Logstash

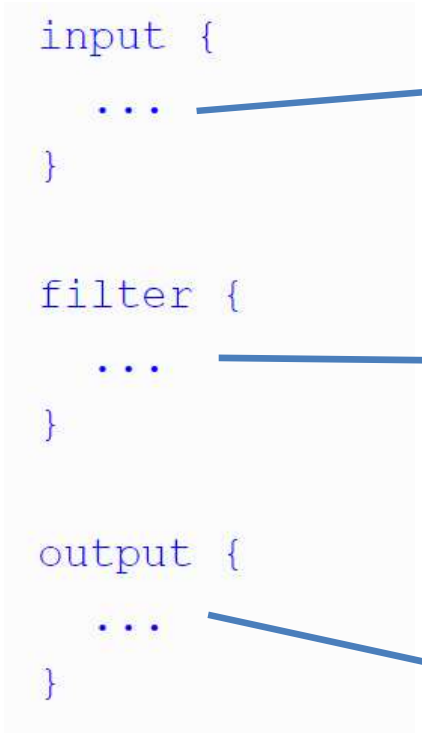
- Event processing engine



How does Logstash work?



Logstash configuration

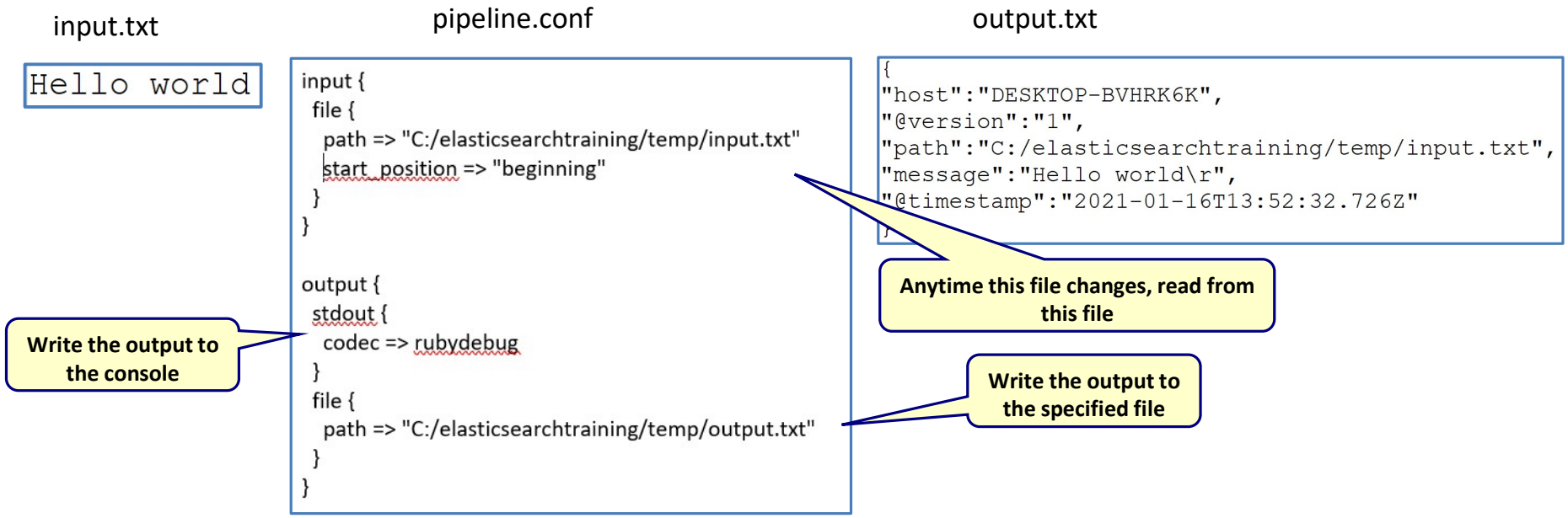


- **file**: This streams log events from a file
- **redis**: This streams events from a redis instance
- **stdin**: This streams events from standard input
- **syslog**: This streams syslog messages over the network
- **ganglia**: This streams ganglia packets over the network via udp
- **lumberjack**: This receives events using the lumberjack protocol
- **eventlog**: This receives events from Windows event log
- **s3**: This streams events from a file from an s3 bucket
- **elasticsearch**: This reads from the Elasticsearch cluster based on results of a search query

- **date**: This is used to parse date fields from incoming events, and use that as Logstash timestamp fields, which can be later used for analytics
- **drop**: This drops everything from incoming events that matches the filter condition
- **grok**: This is the most powerful filter to parse unstructured data from logs or events to a structured format
- **multiline**: This helps parse multiple lines from a single source as one Logstash event
- **dns**: This filter will resolve an IP address from any fields specified
- **mutate**: This helps rename, remove, modify, and replace fields in events
- **geoip**: This adds geographic information based on IP addresses that are retrieved from `Maxmind` database

- **file**: This writes events to a file on disk
- **e-mail**: This sends an e-mail based on some conditions whenever it receives an output
- **elasticsearch**: This stores output to the Elasticsearch cluster, the most common and recommended output for Logstash
- **stdout**: This writes events to standard output
- **redis**: This writes events to redis queue and is used as a broker for many ELK implementations
- **mongodb**: This writes output to mongodb
- **kafka**: This writes events to Kafka topic

Logstash configuration



Logstash configuration

input.txt

```
Hi there
```

pipeline.conf

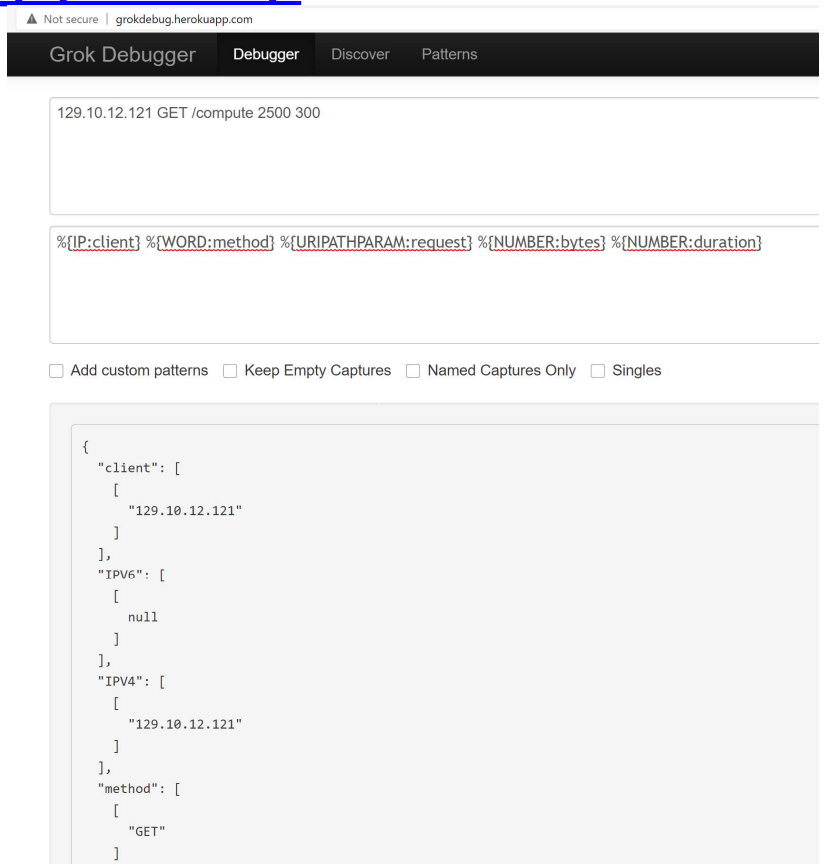
```
input {
 file {
   path => "C:/elasticsearchtraining/temp/input.txt"
          start_position => "beginning"
 }
}

filter {
   mutate {
      uppercase => ["message"]
   }
}

output {
 stdout {
   codec => rubydebug
 }
 file {
   path => "C:/elasticsearchtraining/temp/output.txt"
 }
}
```

output.txt

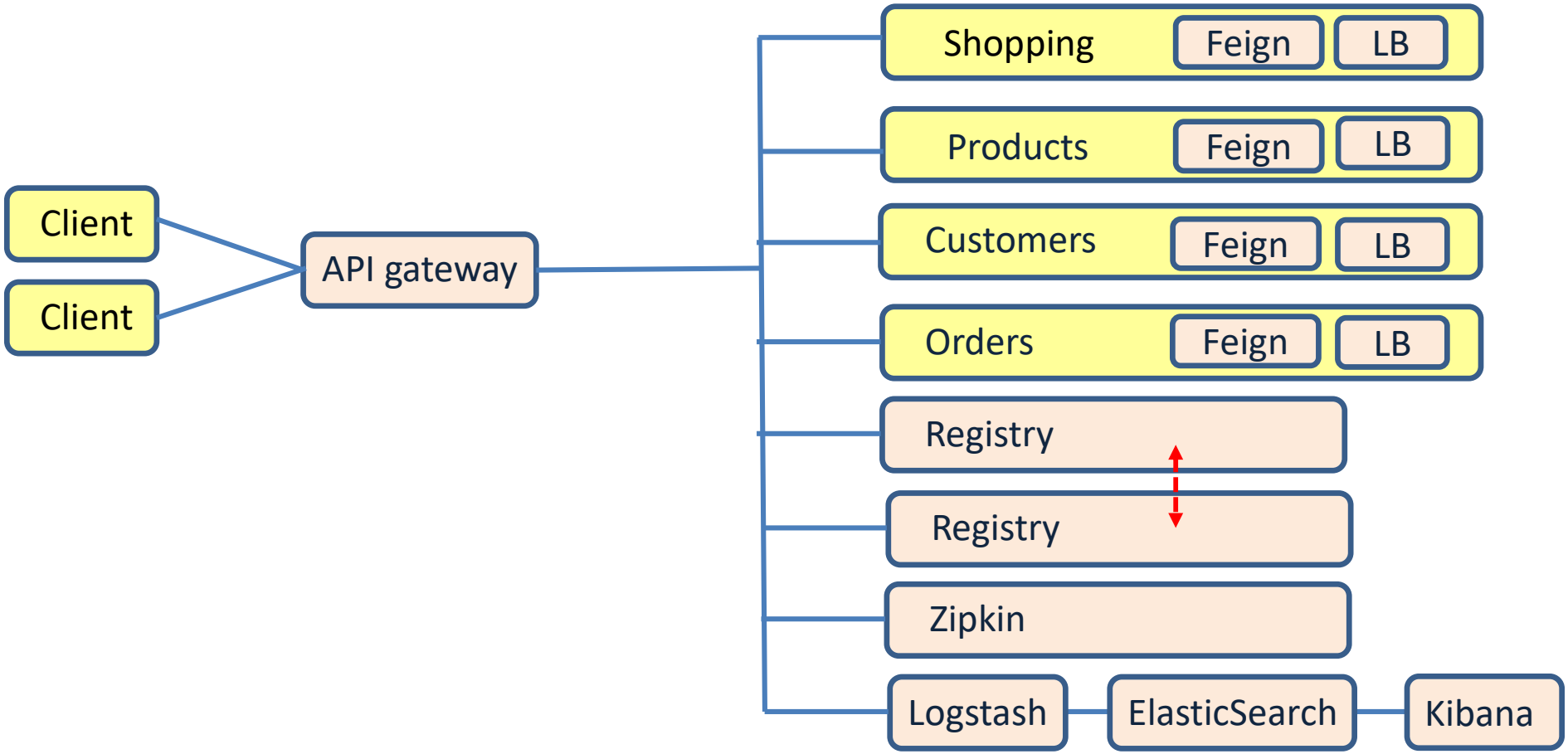
```
{
"path":"C:/elasticsearchtraining/temp/input.txt",
"message":"HI THERE\r",
"host":"DESKTOP-BVHRK6K",
"@version":"1",
"@timestamp":"2021-01-16T14:17:10.537Z"
}
```

Testing grok

- http://grokdebug.herokuapp.com/



Implementing microservices



Challenges of a microservice architecture

Challenge	Solution
Complex communication	Feign Registry API gateway
Performance	
Resilience	Registry replicas Load balancing between multiple service instances
Security	
Transactions	
Following the process	
Keep data in sync	
Keep interfaces in sync	
Keep configuration in sync	
Monitor health of microservices	
Follow/monitor business processes	Zipkin ELK

RESILIENCE

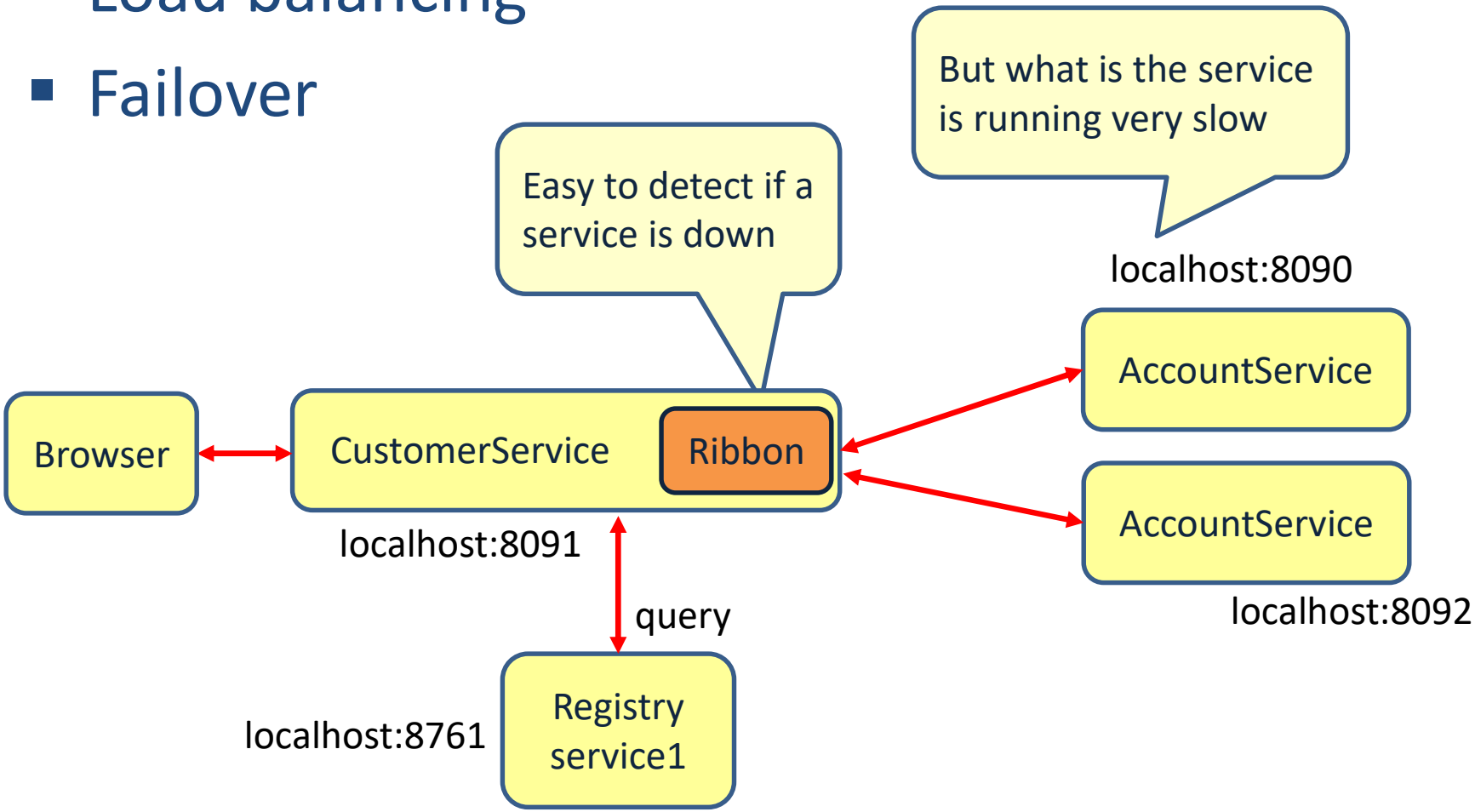
The ability to recover from failures

Fallacies of distributed computing

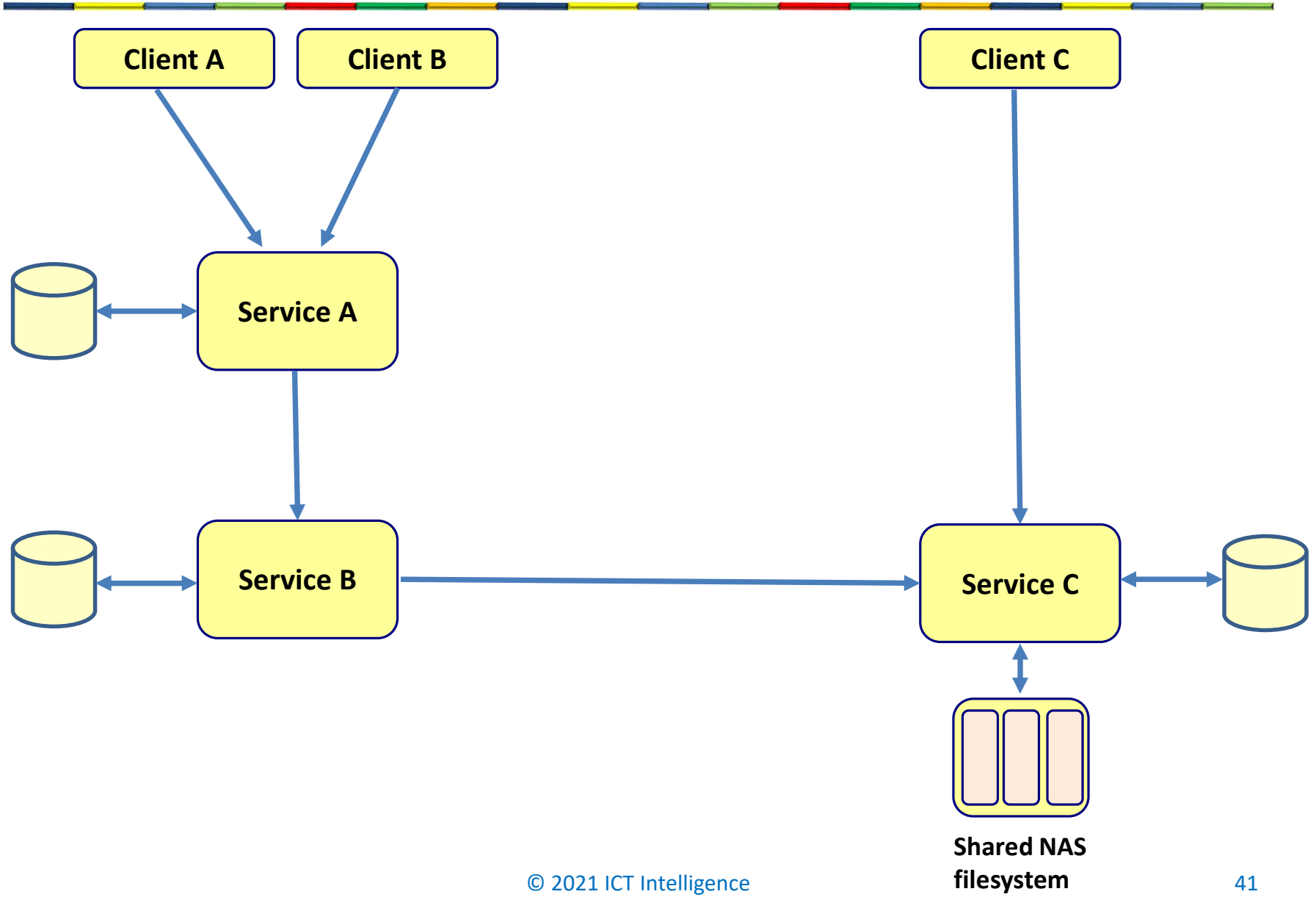
- The network is reliable
- Latency is zero
- Bandwidth is infinite
- The network is secure
- Topology doesn’t change
- There is one administrator
- Transport cost is zero
- The network is homogeneous

Clustering

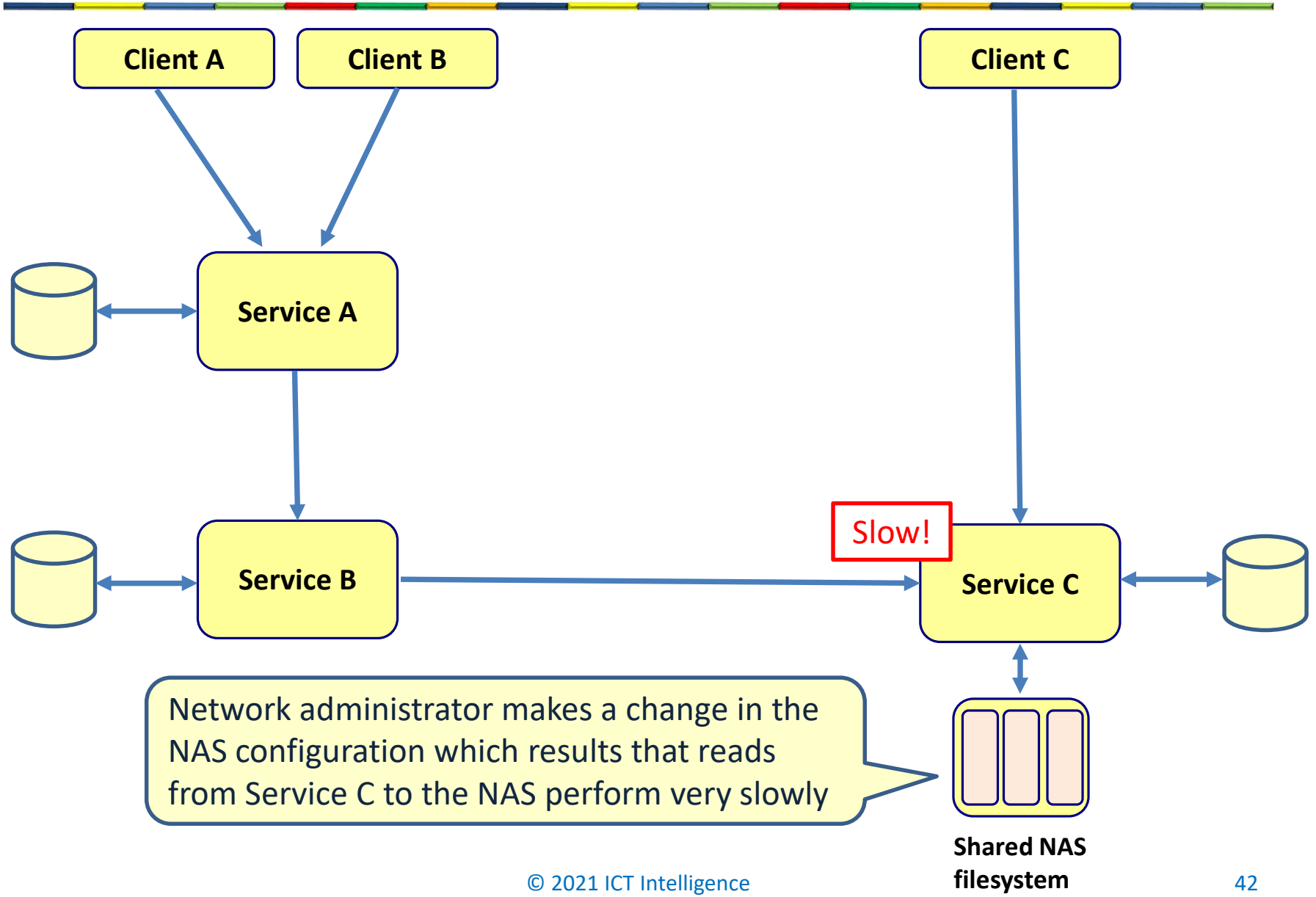
- Load balancing
- Failover



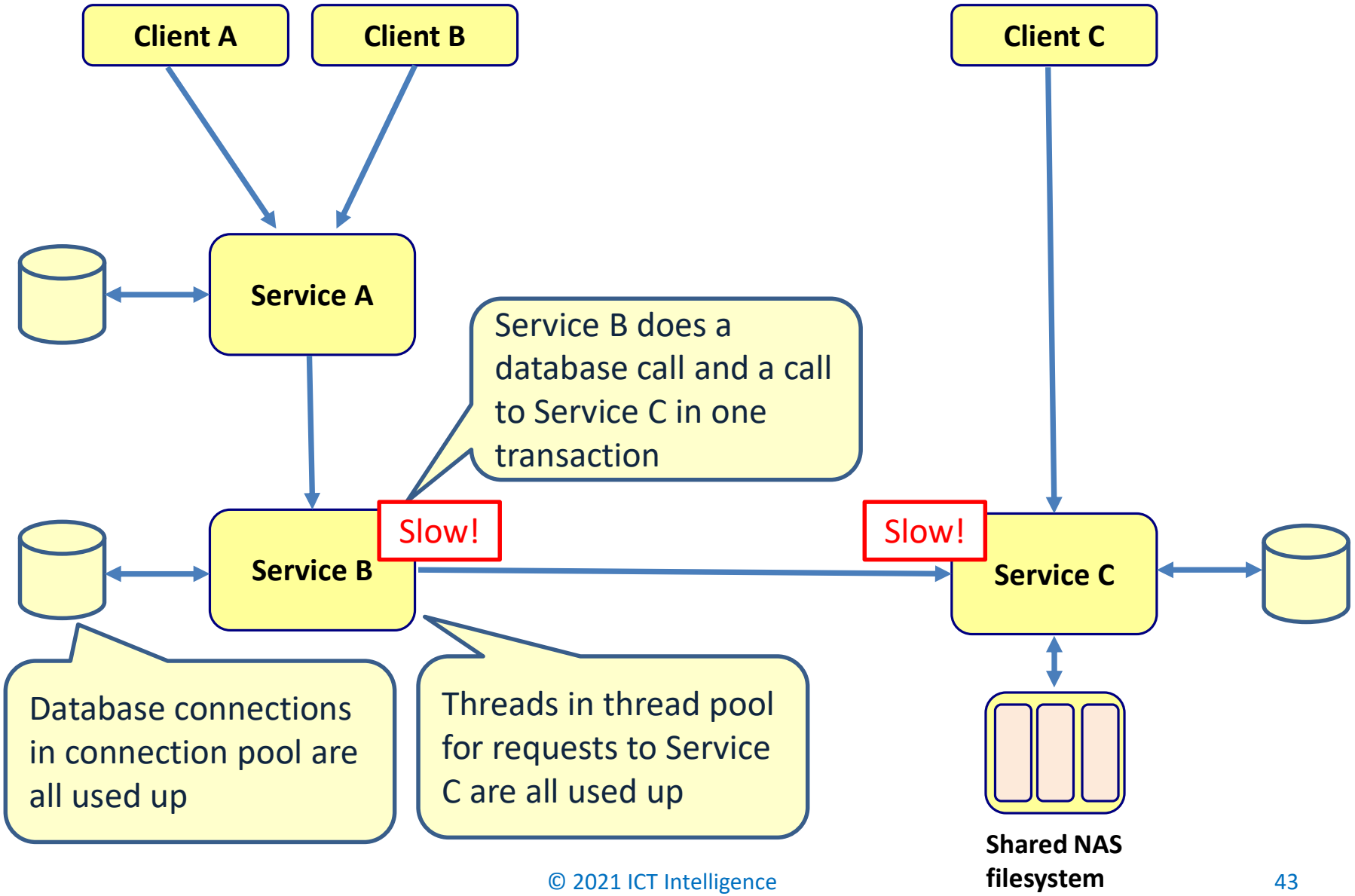
Example



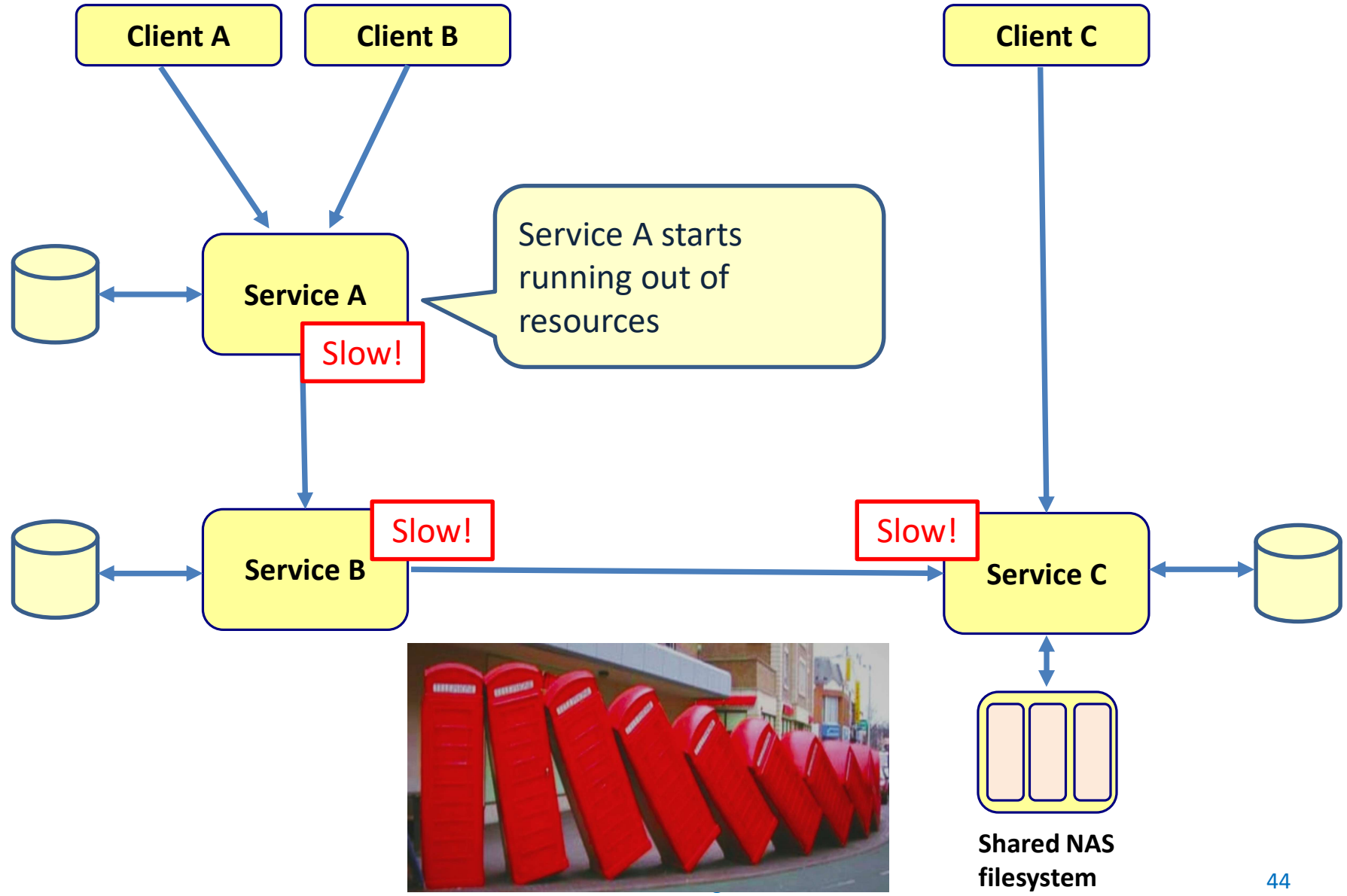
Example



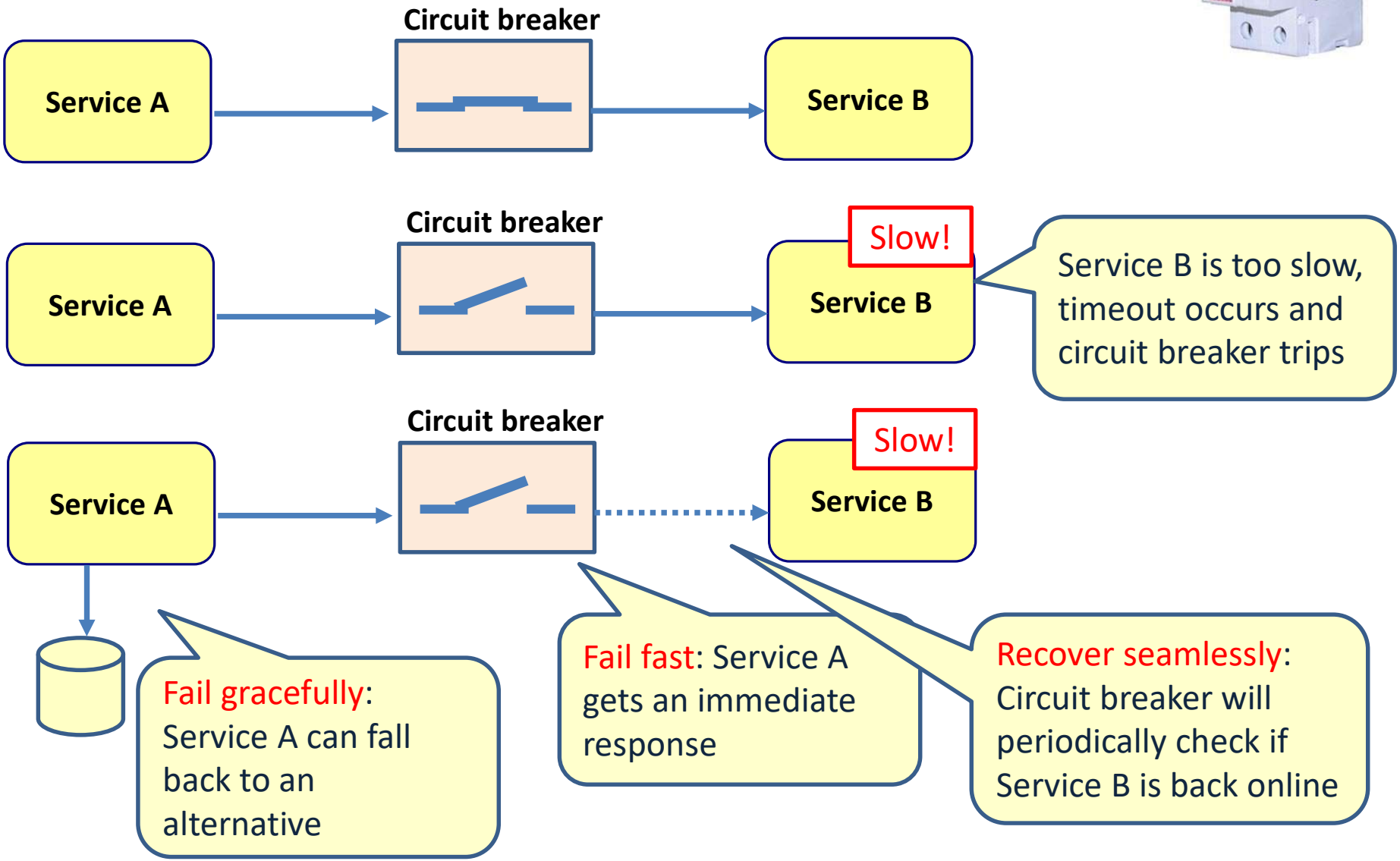
Example



Example


Client A
Client B
Client C
Service A starts running out of resources
Service A
Slow!
Service B
Slow!
Slow!
Service C
Shared NAS filesystem

Circuit breaker



Main point

- A circuit breaker takes care that not the whole microservice architecture gets slow when one service becomes slow.
- Every relative part of creation is connected at the level of pure consciousness.

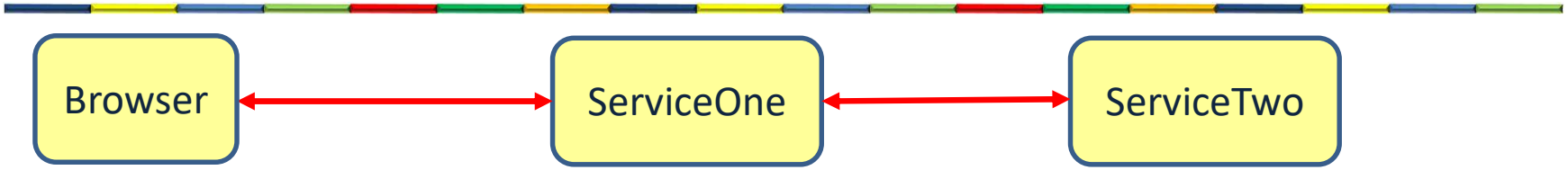
RESILIENCE: HYSTRIX

Hystrix dependency

pom.xml

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

Enable Hystrix




```
@SpringBootApplication
@EnableCircuitBreaker
public class ServiceOneApplication {

  public static void main(String[] args) {
    SpringApplication.run(Service2Application.class, args);
  }
}
```

Enable Hystrix

```
@SpringBootApplication
@EnableCircuitBreaker
public class ServiceTwoApplication {

  public static void main(String[] args) {
    SpringApplication.run(Service2Application.class, args);
  }
}
```

Enable Hystrix

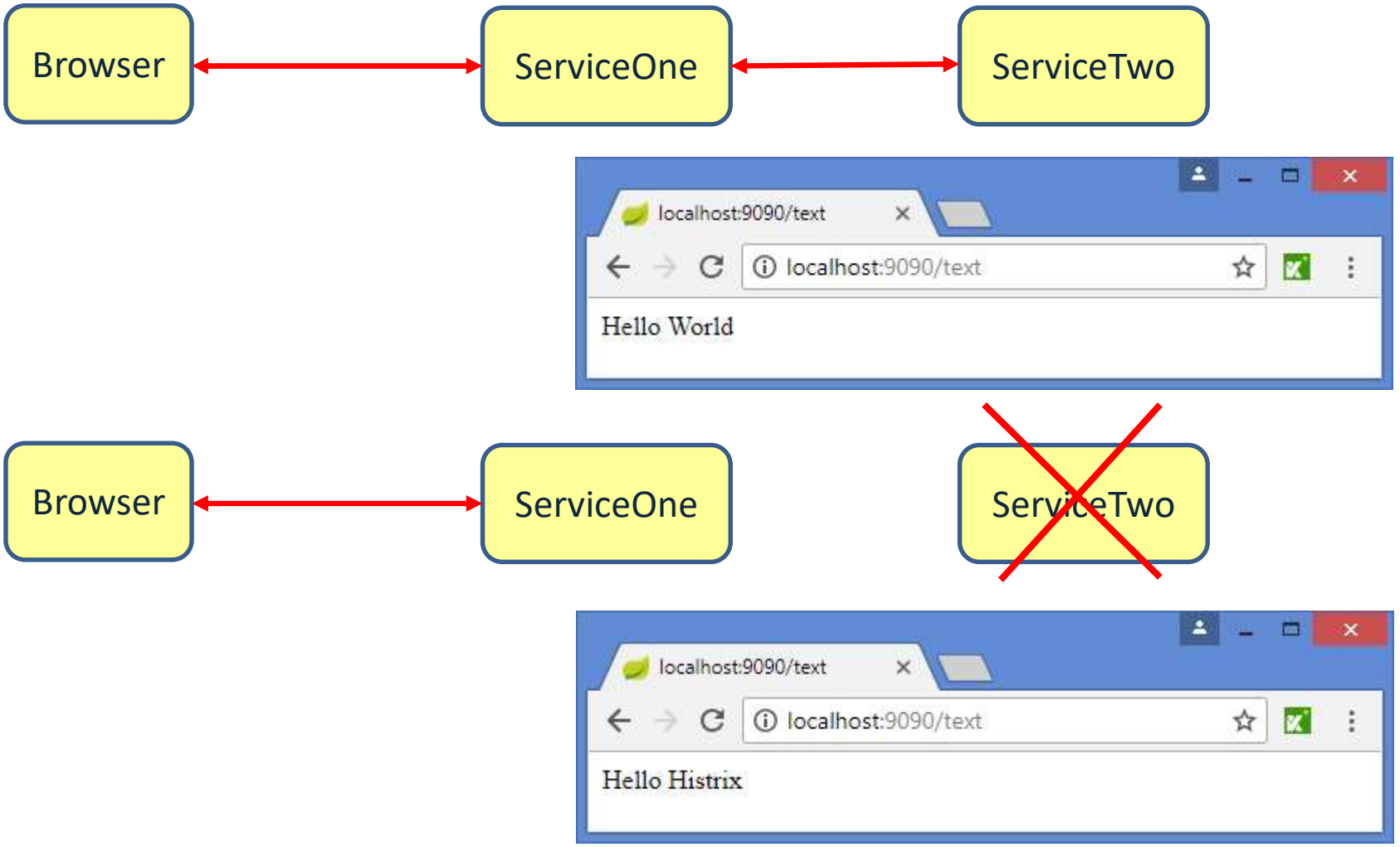
Using the circuit breaker

```
public class ServiceOneController {

  @Autowired
  RestTemplate restTemplate;

  @RequestMapping("/text")
  @HystrixCommand(fallbackMethod = "getTextFallback")
  public String getText() {
    String service2Text = restTemplate.getForObject("http://localhost:9091/text",
                                                    String.class);
    return "Hello "+ service2Text;
  }

  public String getTextFallback() {
   return "Hello Histrix";
  }

  @Bean
  RestTemplate getRestTemplate() {
    return new RestTemplate();
  }
}
```

If this method throws an exception or takes longer than 2 seconds, call the fallback method

Fallback method

Using Histrix



Setting the timeout

```
public class ServiceOneController {

  @Autowired
  RestTemplate restTemplate;

  @RequestMapping("/text")
  @HystrixCommand(fallbackMethod = "getTextFallback", commandProperties=
  {@HystrixProperty(name="execution.isolation.thread.timeoutInMilliseconds",
                    value="4000")})
  public String getText() {
    String service2Text = restTemplate.getForObject("http://localhost:9091/text",
                                                    String.class);
    return "Hello "+ service2Text;
  }

  public String getTextFallback() {
   return "Hello Histrix";
  }

  @Bean
  RestTemplate getRestTemplate() {
    return new RestTemplate();
  }
}
```

Set timeout to 4 seconds

Setting the timeout

```
@RestController
public class ServiceTwoController {

  @RequestMapping("/text")
  public String getText() throws InterruptedException {
    Thread.sleep(5000);
    return "World";
  }
}
```

Sleep of 5 seconds

RESILIENCE: RESILIENCE4J

dependency

pom.xml

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-circuitbreaker-resilience4j</artifactId>
  <version>1.0.2.RELEASE</version>
</dependency>
```

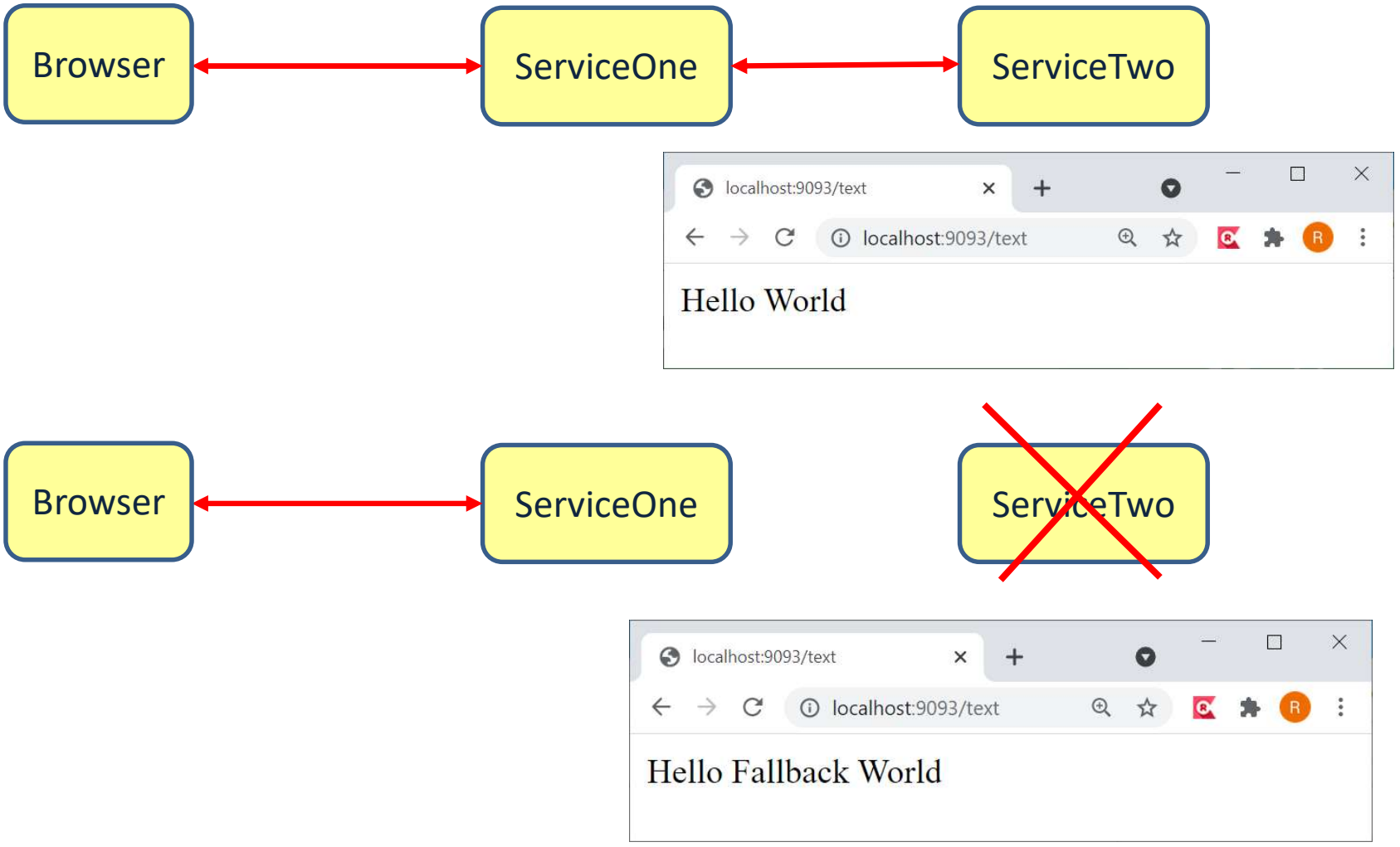
Using the circuit breaker

```
@RestController
public class ServiceOneController {
   @Autowired
   RestTemplate restTemplate;

   @Autowired
   private CircuitBreakerFactory circuitBreakerFactory;

   @RequestMapping("/text")
   public String getText() {
      CircuitBreaker circuitBreaker = circuitBreakerFactory.create("circuitbreaker");

      String service2Text = circuitBreaker.run(() -> restTemplate.getForObject("http://localhost:9091/text",
            String.class), throwable -> getFallbackName());
      return "Hello "+ service2Text;
   }

   private String getFallbackName() {
      return "Fallback World";
   }

   @Bean
   RestTemplate getRestTemplate() {
      return new RestTemplate();
   }
}
```

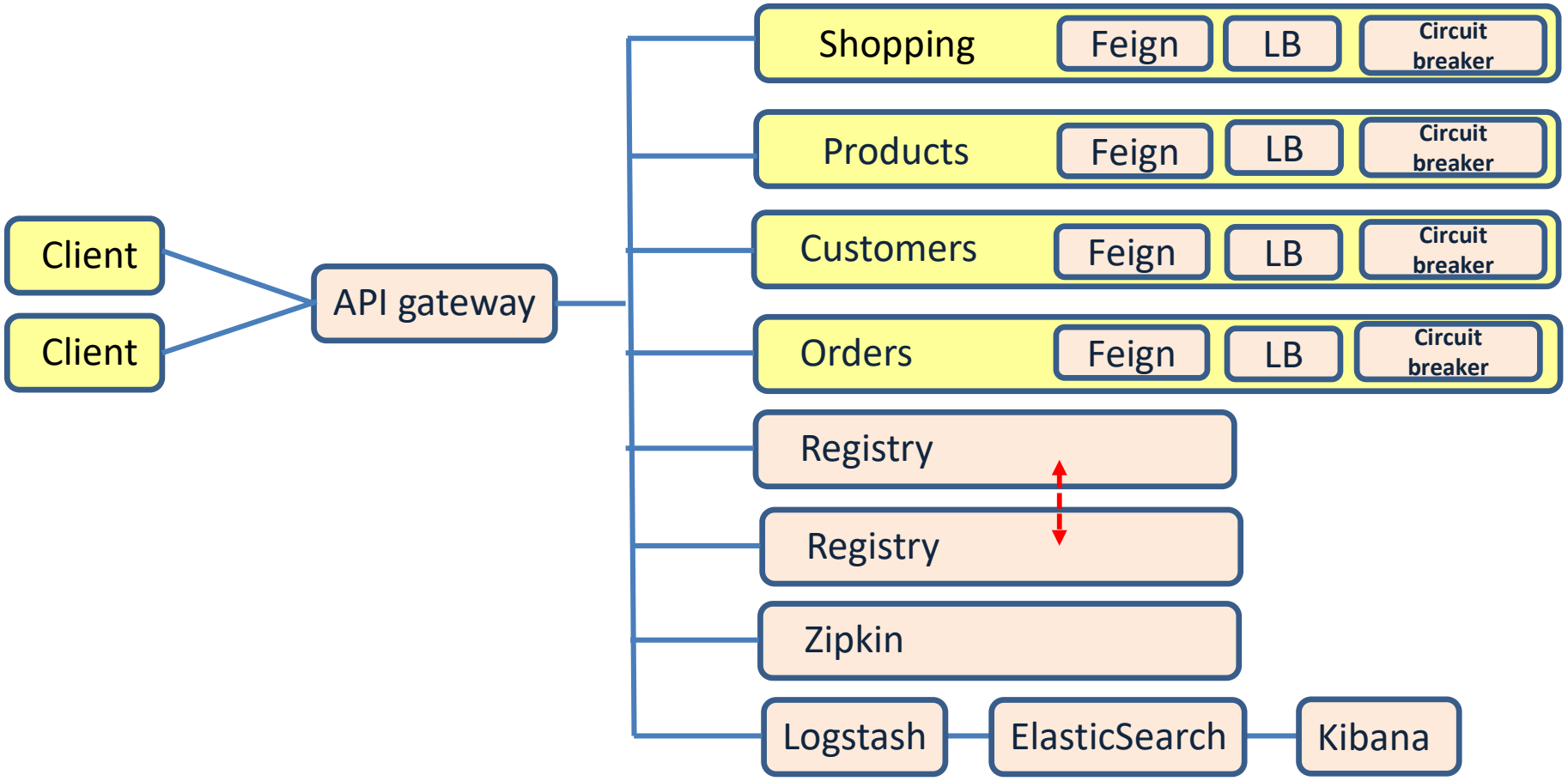
If this method throws an exception or takes longer than 2 seconds, call the fallback method

Fallback method

Using Resilience4J



Implementing microservices

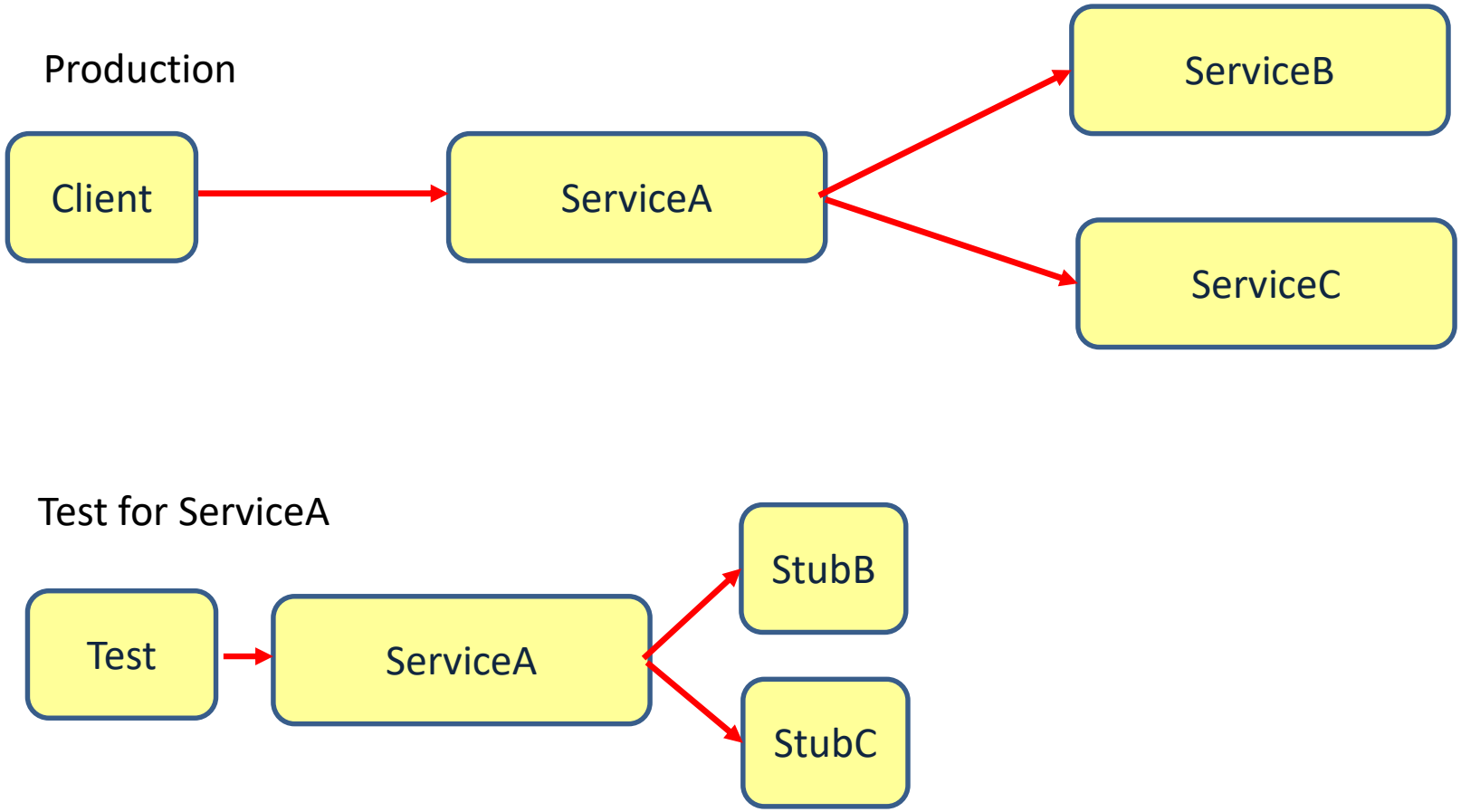


Challenges of a microservice architecture

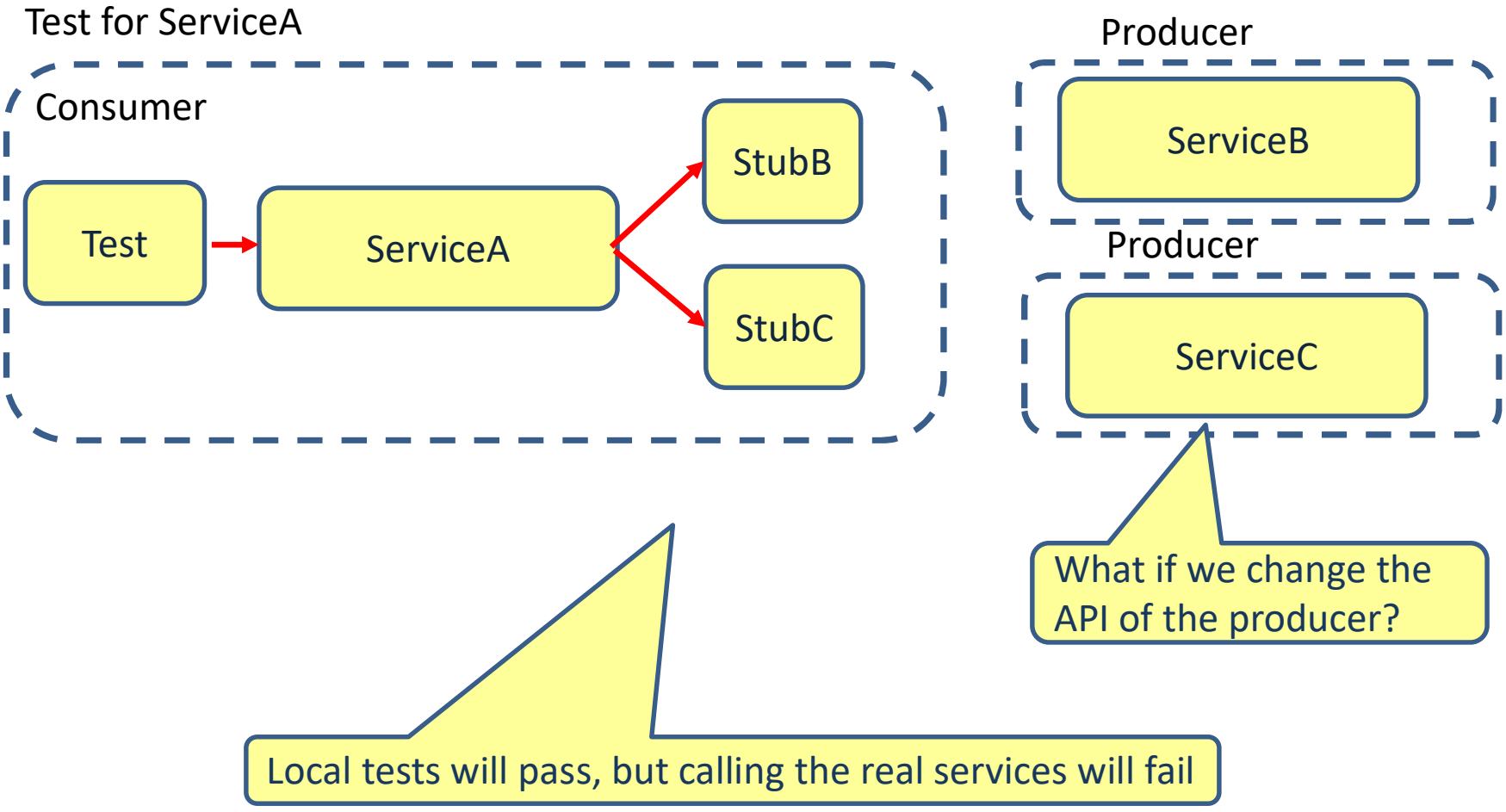
Challenge	Solution
Complex communication	Feign Registry API gateway
Performance	
Resilience	Registry replicas Load balancing between multiple service instances Circuit breaker
Security	
Transactions	
Following the process	
Keep data in sync	
Keep interfaces in sync	
Keep configuration in sync	
Monitor health of microservices	
Follow/monitor business processes	Zipkin ELK

SPRING CLOUD CONTRACT

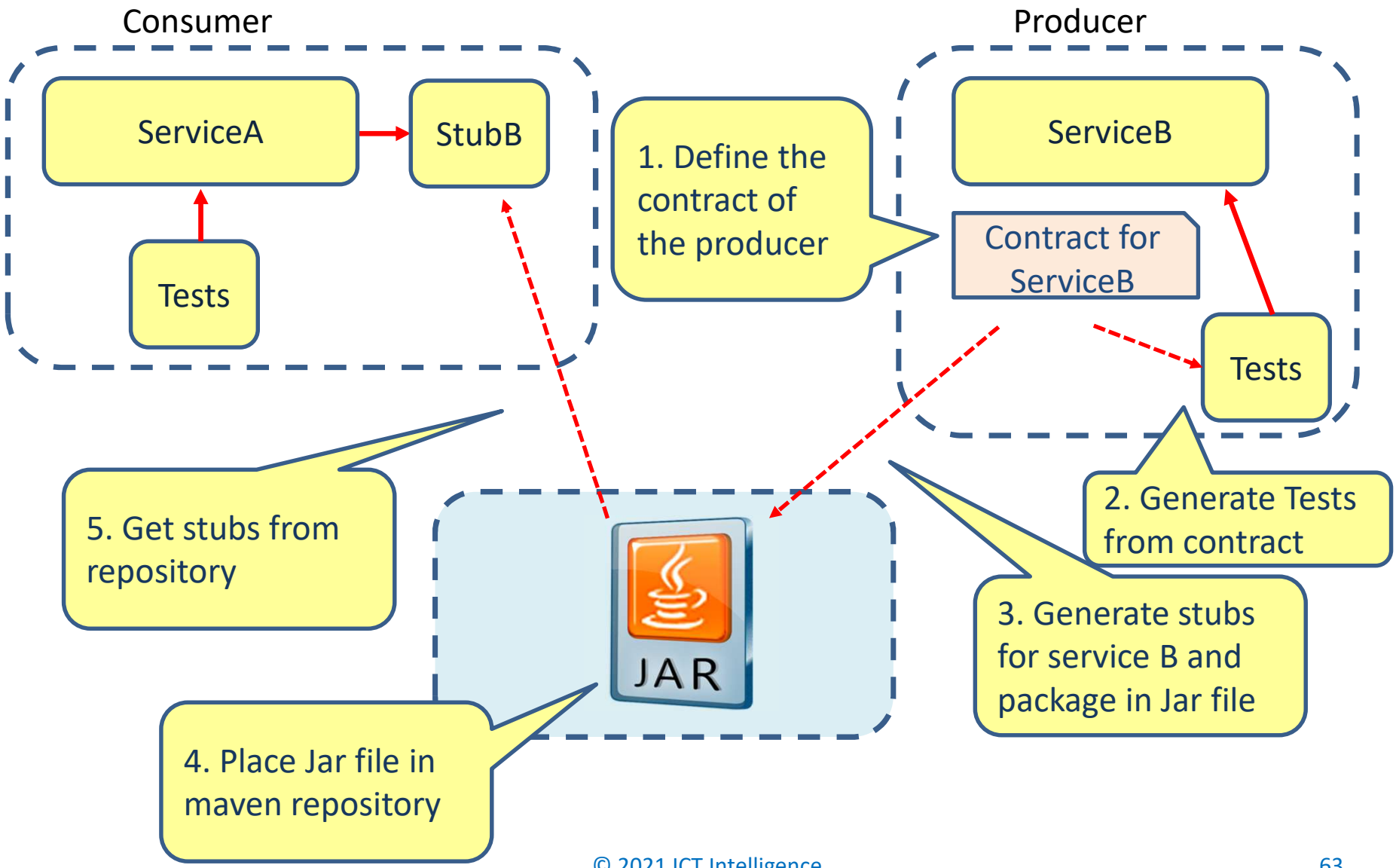
How to test microservices



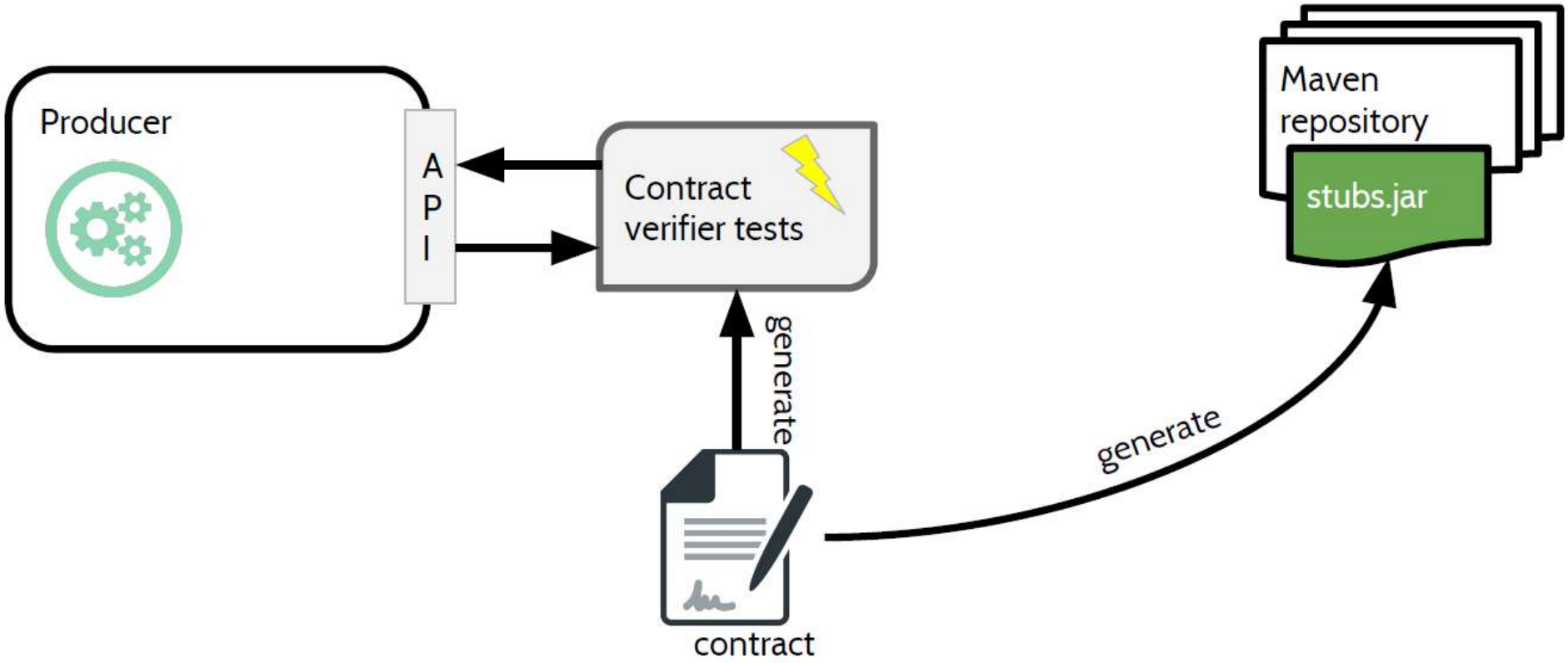
Stubs live at the consumer



Spring cloud contracts



Producer


Producer
A
P
I
Contract
verifier tests
Maven
repository
stubs.jar
generate
generate
contract

Producer maven configuration

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-contract-verifier</artifactId>
  <scope>test</scope>
</dependency>


<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>2.2.2.RELEASE</version>
  <extensions>true</extensions>
  <configuration>
    <baseClassForTests>service.BaseTestClass</baseClassForTests>
    <testFramework>JUNIT5</testFramework>
  </configuration>
</plugin>
```

Producer

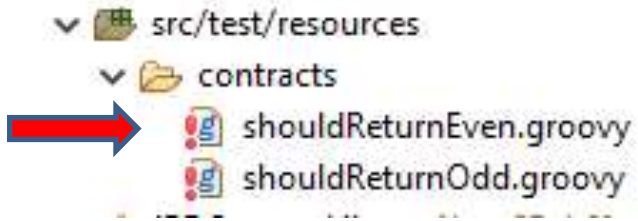
```
@RestController
public class EvenOddController {

    @GetMapping("/validate")
    public String evenOrOdd(@RequestParam("number") Integer number) {
        return number % 2 == 0 ? "Even" : "Odd";
    }
}
```

```
@SpringBootApplication
public class EvenoddServiceApplication {

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

Producer contract 1

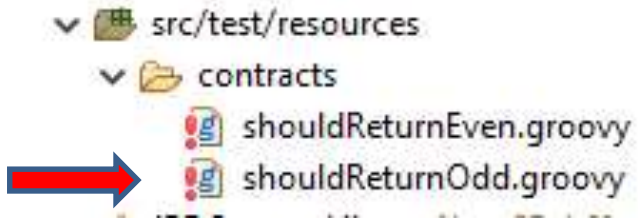
```
import org.springframework.cloud.contract.spec.Contract

Contract.make {
  description "should return even when number input is even"
  request{
    method GET()
    url("/validate") {
      queryParameters {
        parameter("number", "2")
      }
    }
  }
  response {
    body("Even")
    status 200
  }
}
```

Contract in groovy



Producer contract 2

```
import org.springframework.cloud.contract.spec.Contract

Contract.make {
  description "should return odd when number input is odd"
  request {
    method GET()
    url("/validate") {
      queryParameters {
        parameter("number", "1")
      }
    }
  }
  response {
    body("Odd")
    status 200
  }
}
```

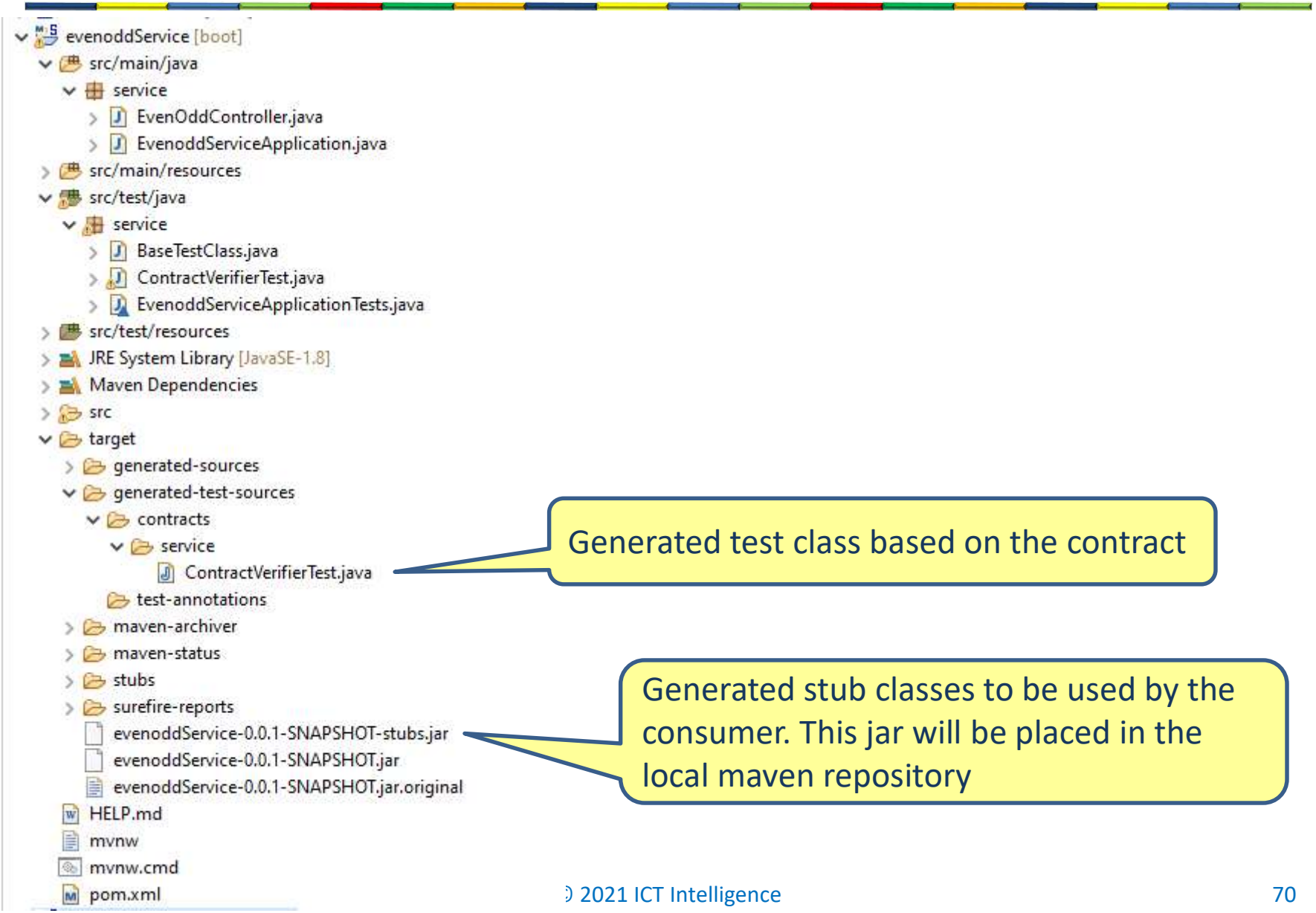


Producer: base test class

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.MOCK)
@DirtiesContext
@AutoConfigureMessageVerifier
public class BaseTestClass {

    @Autowired
    private EvenOddController evenOddController;

    @BeforeEach
    public void setup() {
        StandaloneMockMvcBuilder standaloneMockMvcBuilder
          = MockMvcBuilders.standaloneSetup(evenOddController);
        RestAssuredMockMvc.standaloneSetup(standaloneMockMvcBuilder);
    }
}
```

This is the base class for all to be generated test classes

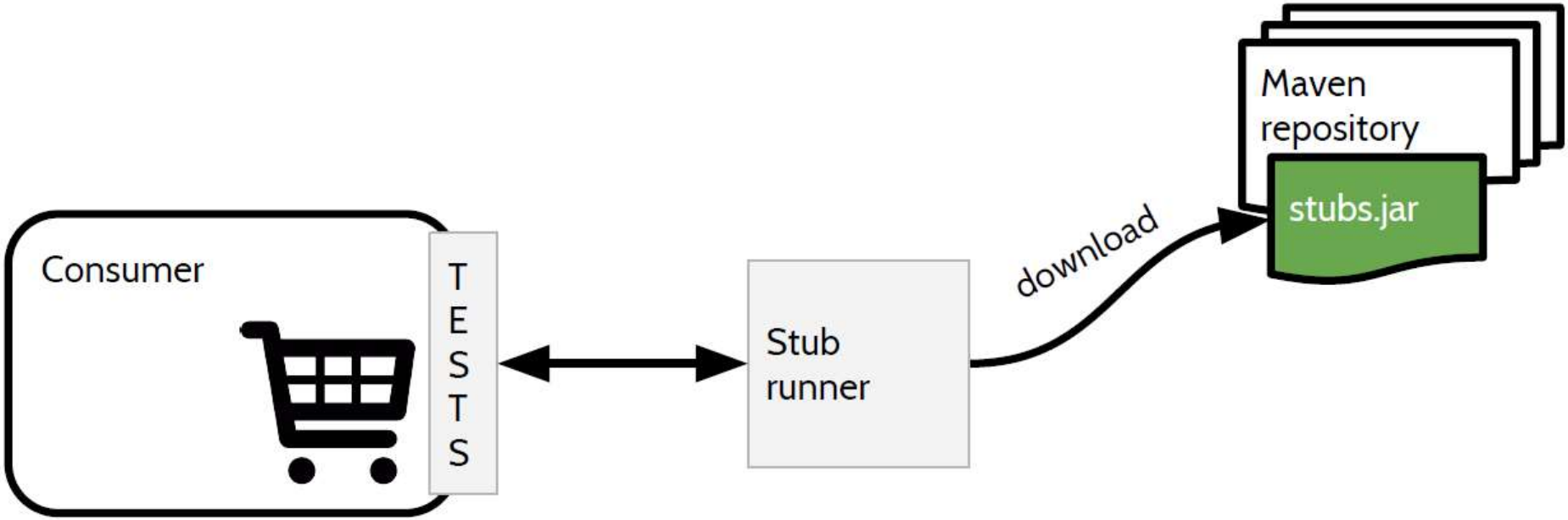
After running **maven install**



Spring cloud contract DSL

```
import org.springframework.cloud.contract.spec.Contract
Contract.make {
  description("GET employee with id=1")
  request {
    method 'GET'
    url '/employee/1'
  }
 response {
    status 200
    body("""
    {
    "id": "1",
    "fname": "Jane",
    "lname": "Doe",
    "salary": "123000.00",
    "gender": "M"
    }
    """)
    headers {
      contentType(applicationJson())
    }
  }
}
```

Consumer




```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-contract-stub-runner</artifactId>
  <version>2.2.2.RELEASE</version>
  <scope>test</scope>
</dependency>
```

Consumer

```
@RestController
public class MathController {

    @Autowired
    private RestTemplate restTemplate;

    @GetMapping("/calculate")
    public String checkOddAndEven(@RequestParam("number") Integer number) {
        HttpHeaders httpHeaders = new HttpHeaders();
        httpHeaders.add("Content-Type", "application/json");

        ResponseEntity<String> responseEntity = restTemplate.exchange(
          "http://localhost:8090/validate?number=" + number,
          HttpMethod.GET,
          new HttpEntity<>(httpHeaders),
          String.class);

        return responseEntity.getBody();
    }
}
```

Consumer

```
@SpringBootApplication
public class MathServiceApplication {

  public static void main(String[] args) {
    SpringApplication.run(MathServiceApplication.class, args);
  }

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

Consumer test

Get the stubs from the local repository

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.MOCK)
@AutoConfigureMockMvc
@AutoConfigureJsonTesters
@AutoConfigureStubRunner(stubsMode = StubRunnerProperties.StubsMode.LOCAL,
        ids = "com.acme:evenoddService:+:stubs:8090")
public class MathControllerIntegrationTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void given_WhenPassEvenNumberInQueryParam_ThenReturnEven() throws Exception {
        mockMvc.perform(MockMvcRequestBuilders.get("/calculate?number=2")
          .contentType(MediaType.APPLICATION_JSON))
          .andExpect(status().isOk())
          .andExpect(content().string("Even"));
    }

    @Test
    public void given_WhenPassOddNumberInQueryParam_ThenReturnOdd() throws Exception {
        mockMvc.perform(MockMvcRequestBuilders.get("/calculate?number=1")
          .contentType(MediaType.APPLICATION_JSON))
          .andExpect(status().isOk())
          .andExpect(content().string("Odd"));
    }
}
```

Consumer test

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.MOCK)
@AutoConfigureMockMvc
@AutoConfigureJsonTesters
@AutoConfigureStubRunner(stubsMode = StubRunnerProperties.StubsMode.LOCAL,
        ids = "com.acme:evenoddService:+:stubs:8090")
public class MathControllerIntegrationTest {
```

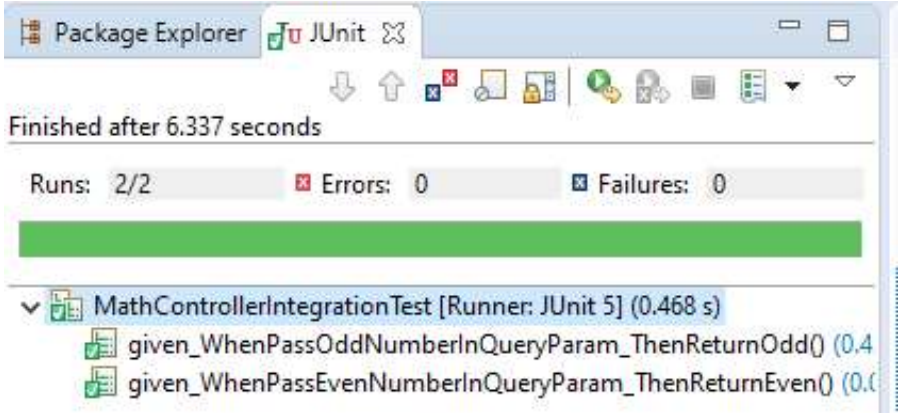
Group id

Artifact id

Version + means latest version

stubs

Port number to run the stubs on



Challenges of a microservice architecture

Challenge	Solution
Complex communication	Feign Registry API gateway
Performance	
Resilience	Registry replicas Load balancing between multiple service instances Circuit breaker
Security	
Transactions	
Following the process	
Keep data in sync	
Keep interfaces in sync	Spring cloud contract
Keep configuration in sync	
Monitor health of microservices	
Follow/monitor business processes	Zipkin ELK

Lesson 9

MICROSERVICES

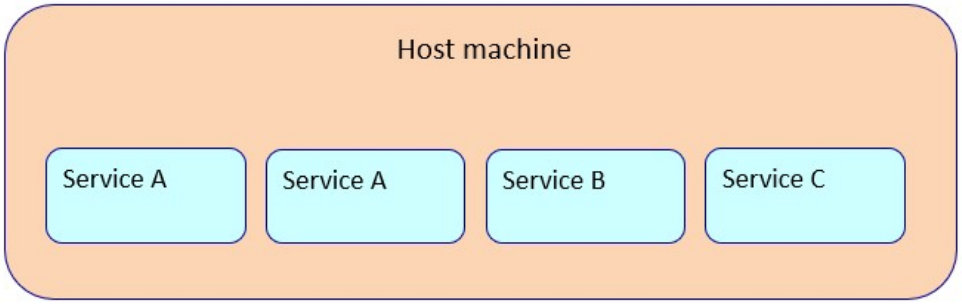
SERVICE DEPLOYMENT

Service deployment

- Service are written using different languages, frameworks, framework versions
- Run multiple service instances of a service for throughput and availability
- Building and deploying should be fast
- Instances need to be isolated
- Constrain the resources a service may consume (CPU, memory, etc.)
- Deployment should be reliable

Multiple service instances per host

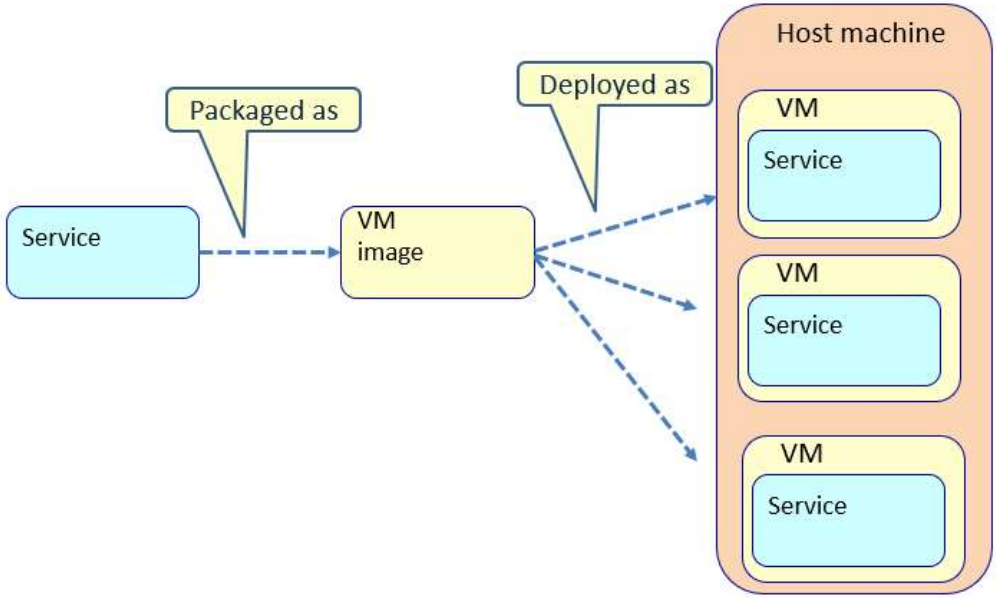
- Benefits
  - Efficient resource utilization
  - Fast deployment



- Drawbacks
  - Poor isolation
  - Poor visibility of resource utilization
  - Difficult to constrain resource utilization
  - Risk of dependency version conflicts
  - Poor encapsulation of implementation technology

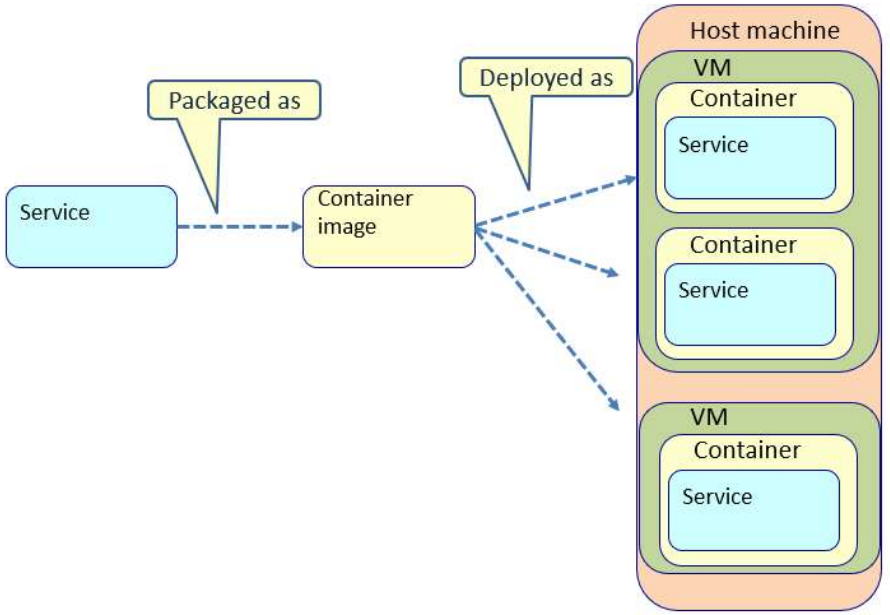
Service per VM

- Benefits
  - Great isolation
  - Great manageability
  - VM encapsulates implementation technology
  - Leverage cloud infrastructure for auto scaling/load balancing
- Drawbacks
  - Less efficient resource utilization
  - Slow deployment



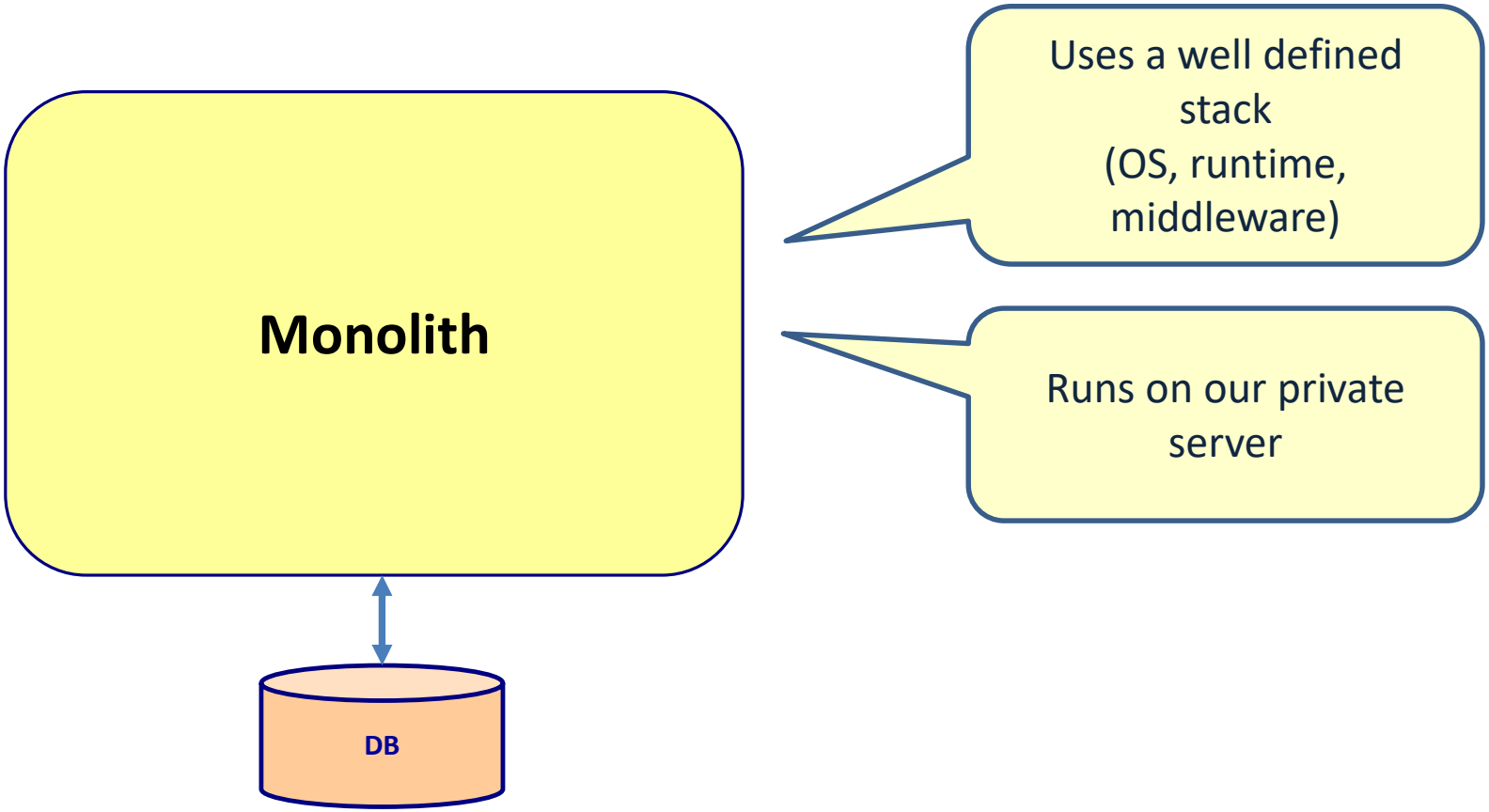
Service per container

- Benefits
  - Great isolation
  - Great manageability
  - Container encapsulates implementation technology
  - Efficient resource utilization
  - Fast deployment
- Drawbacks
  - Technology is not as mature as VM's
  - Containers are not as secure as VM's

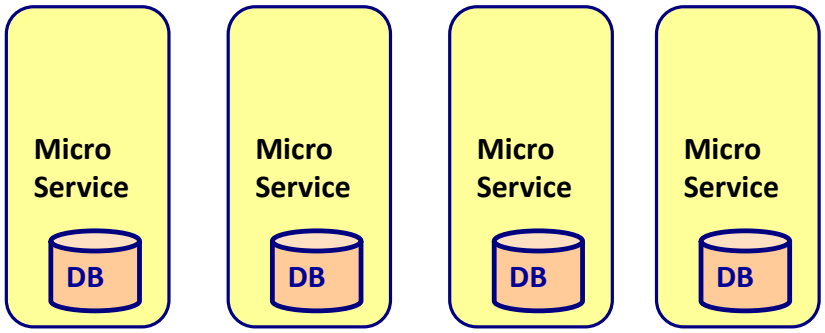


CONTAINERS

Monolith



Microservice



Microservices use different stacks. (OS, runtime, middleware)

Runs anywhere (private, cloud)

The challenge

User DB

postgresql + pgv8 + v8

Static website

nginx 1.5 + modsecurity + openssl + bootstrap 2

Queue

Redis + redis-sentinel

Analytics DB

hadoop + hive + thrift + OpenJDK

Background workers

Web frontend

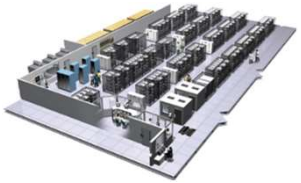
Ruby + Rails + sass + Unicorn

API endpoint

Development VM

QA server

Public Cloud



Production Cluster

Disaster recovery

Customer Data Center

Production Servers

Contributor's laptop

Matrix from hell

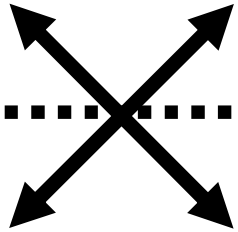
	Development VM	QA Server	Single Prod Server	Onsite Cluster	Public Cloud	Contributor's laptop	Customer Servers
Static website	?	?	?	?	?	?	?
Web frontend	?	?	?	?	?	?	?
Background workers	?	?	?	?	?	?	?
User DB	?	?	?	?	?	?	?
Analytics DB	?	?	?	?	?	?	?
Queue	?	?	?	?	?	?	?

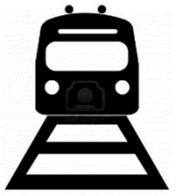
Cargo Transport Pre-1960

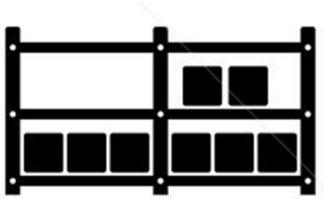






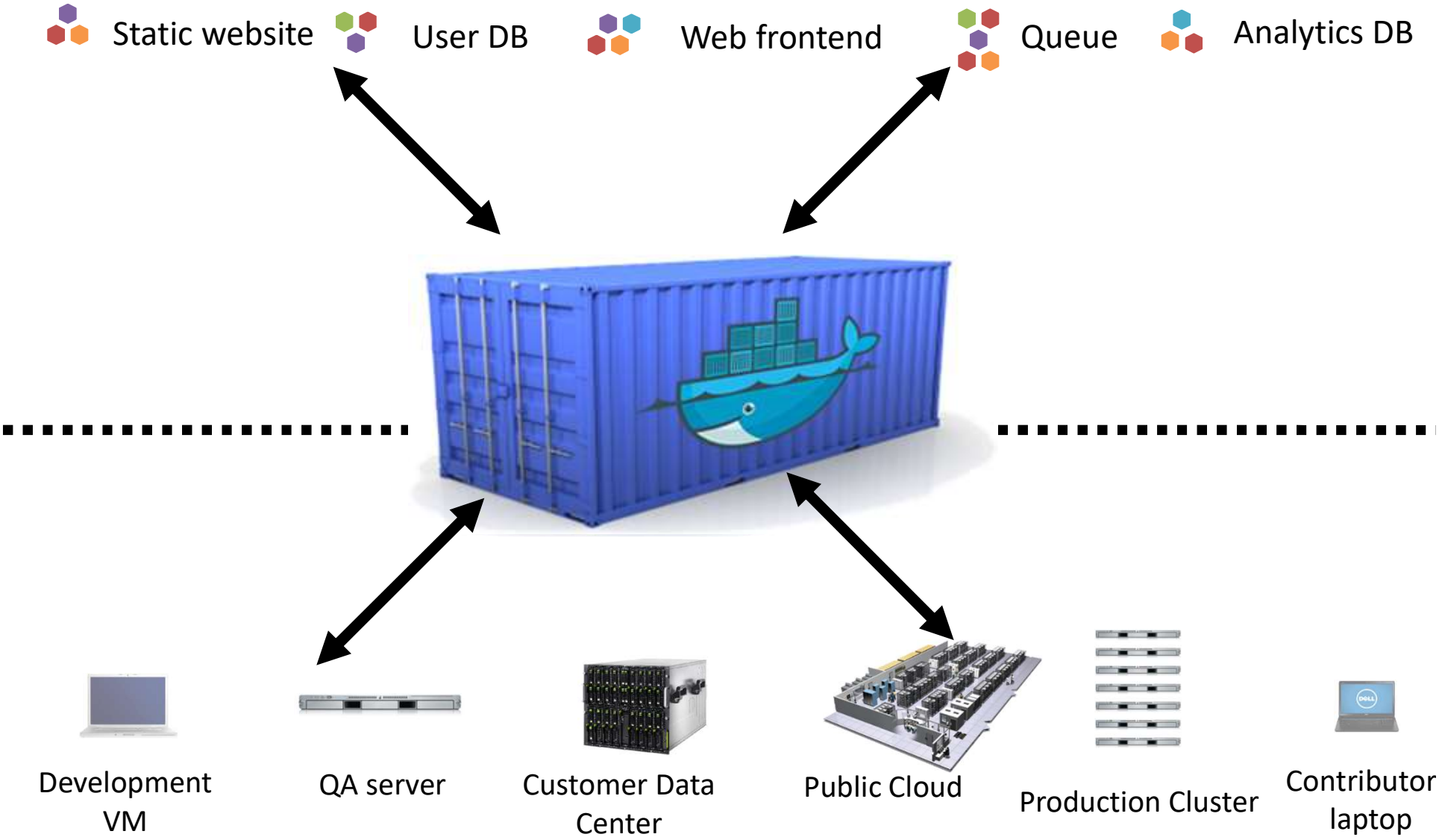



The solution

Docker is a shipping container for code

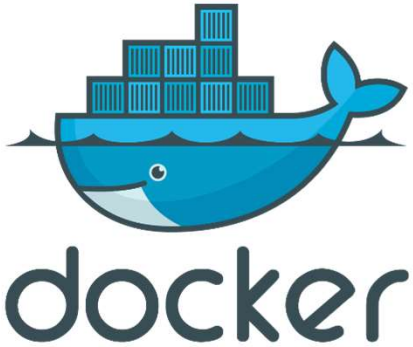


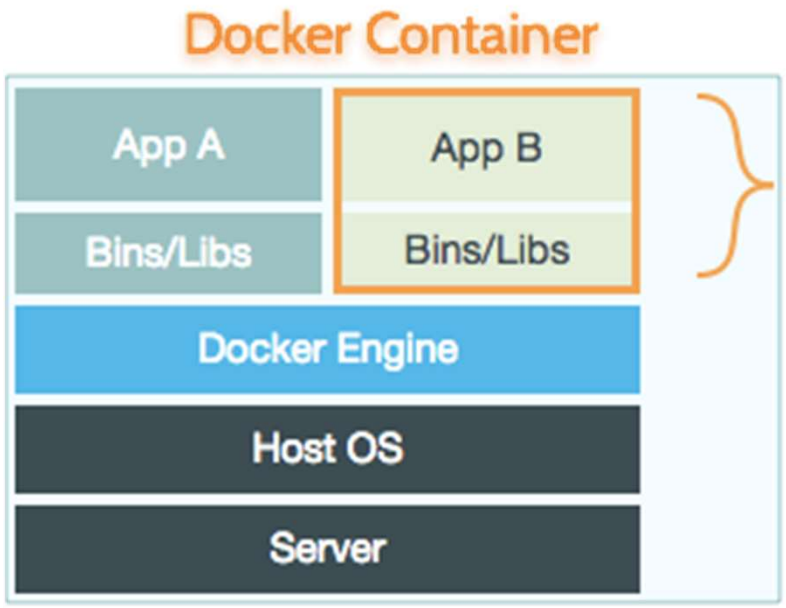
Separation of concern

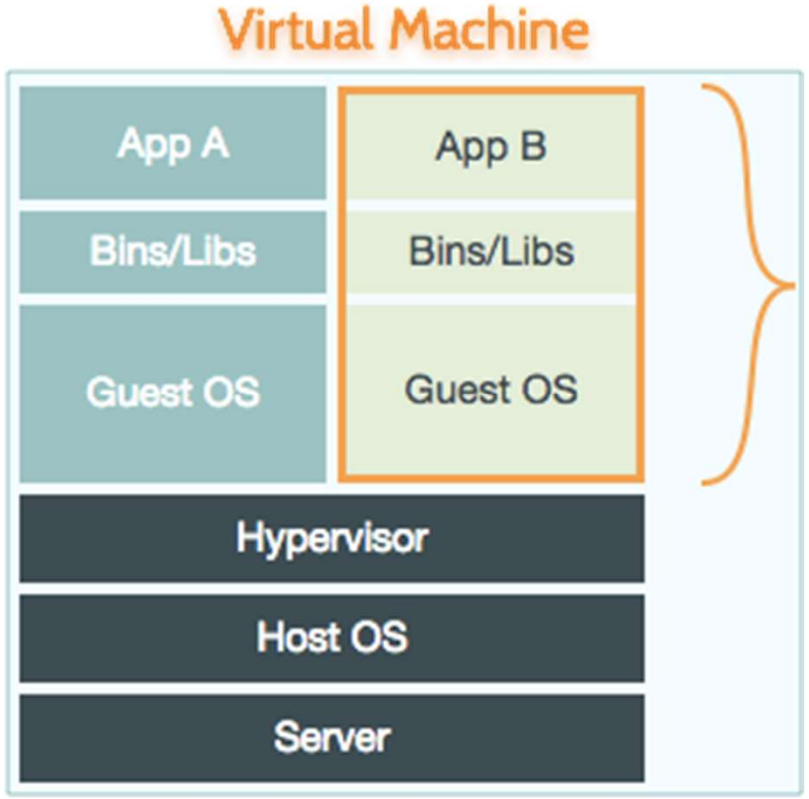
- Dan the Developer
  - Worries about what's "inside" the container
    - His code
    - His Libraries
    - His Package Manager
    - His Apps
    - His Data
- Oscar the Ops Guy
  - Worries about what's "outside" the container
    - Logging
    - Remote access
    - Monitoring
    - Network config
  - All containers start, stop, copy, attach, migrate, etc. the same way

Containers

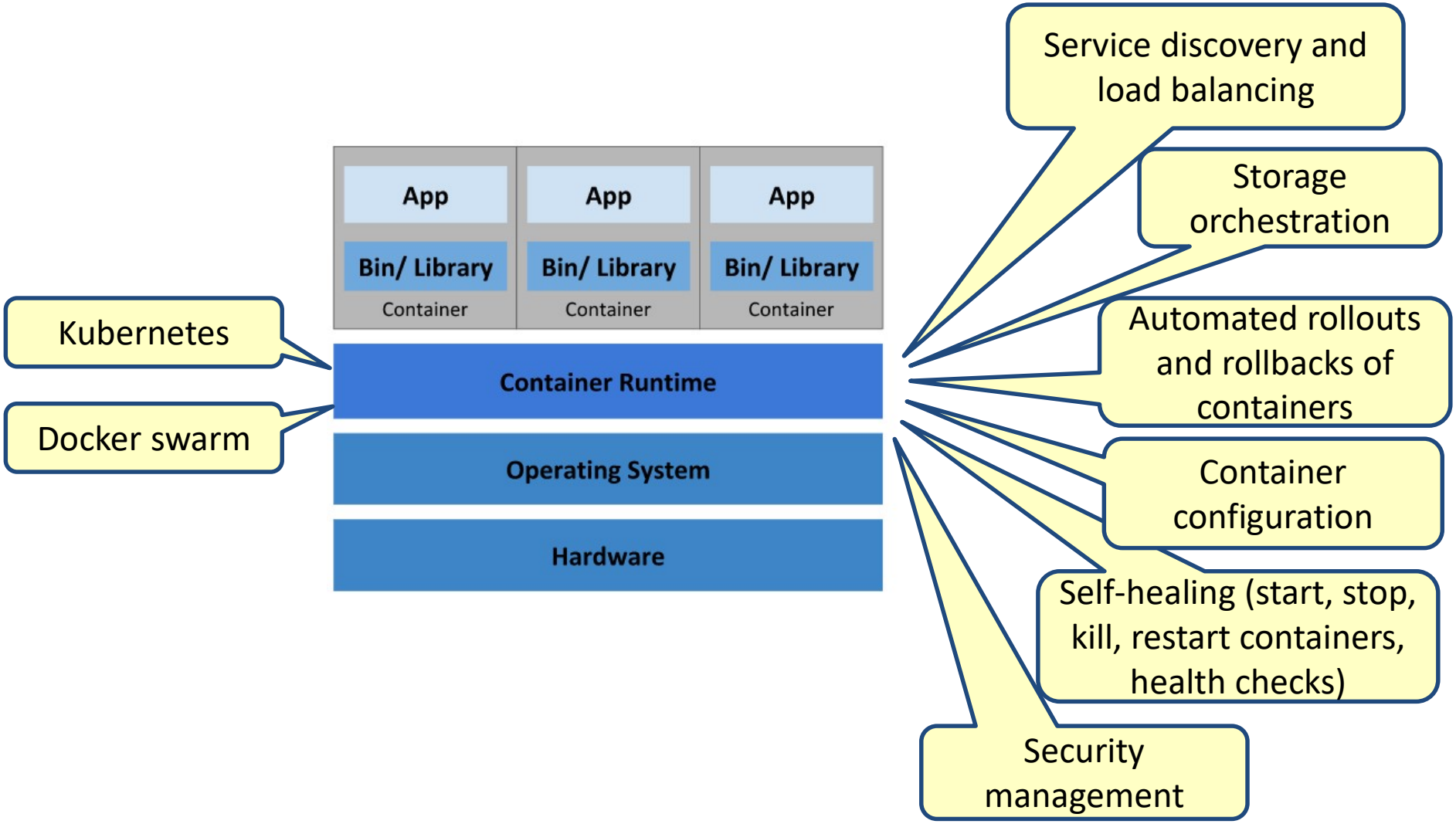
- Docker







Container management



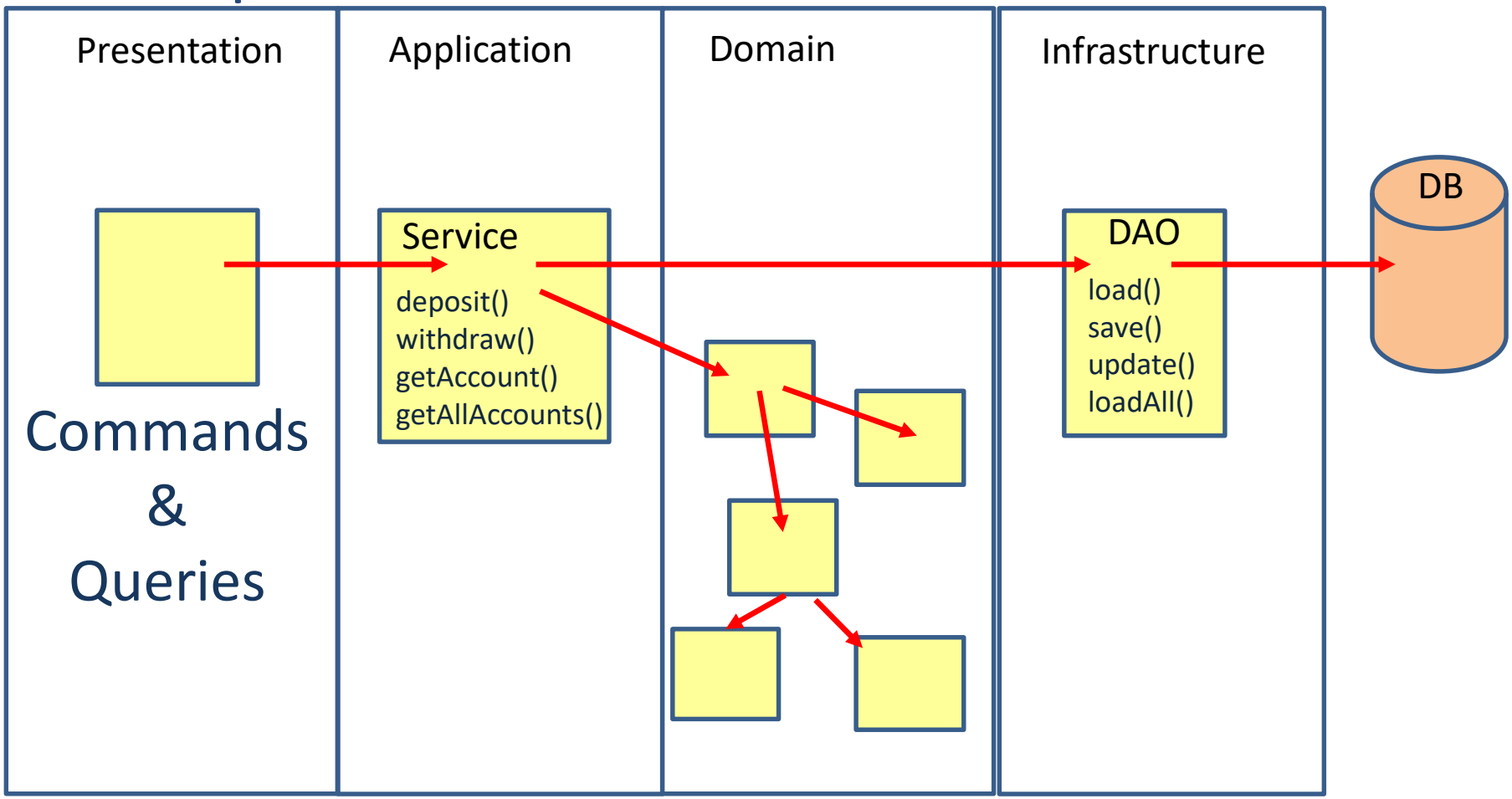
CQRS

Command Query Responsibility Segregation (CQRS)

- Separates the querying from command processing by providing two models instead of one.
  - One model is built to handle and process commands
  - One model is built for presentation needs (queries)

Typical architecture

- One domain model that is used for commands and queries

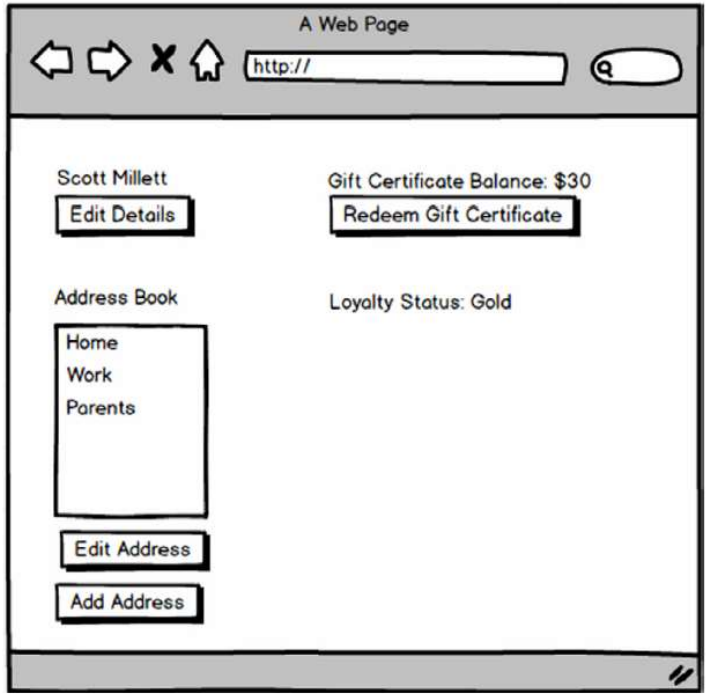


One model for both commands and queries

- To support complex views and reporting
  - Required domain model becomes complex
  - Internal state needs to be exposed
  - Aggregates are merged for view requirements
  - Repositories often contain many extra methods to support presentation needs such as paging, querying, and free text searching

- Result: single model that is full of compromises

Example of complex aggregates



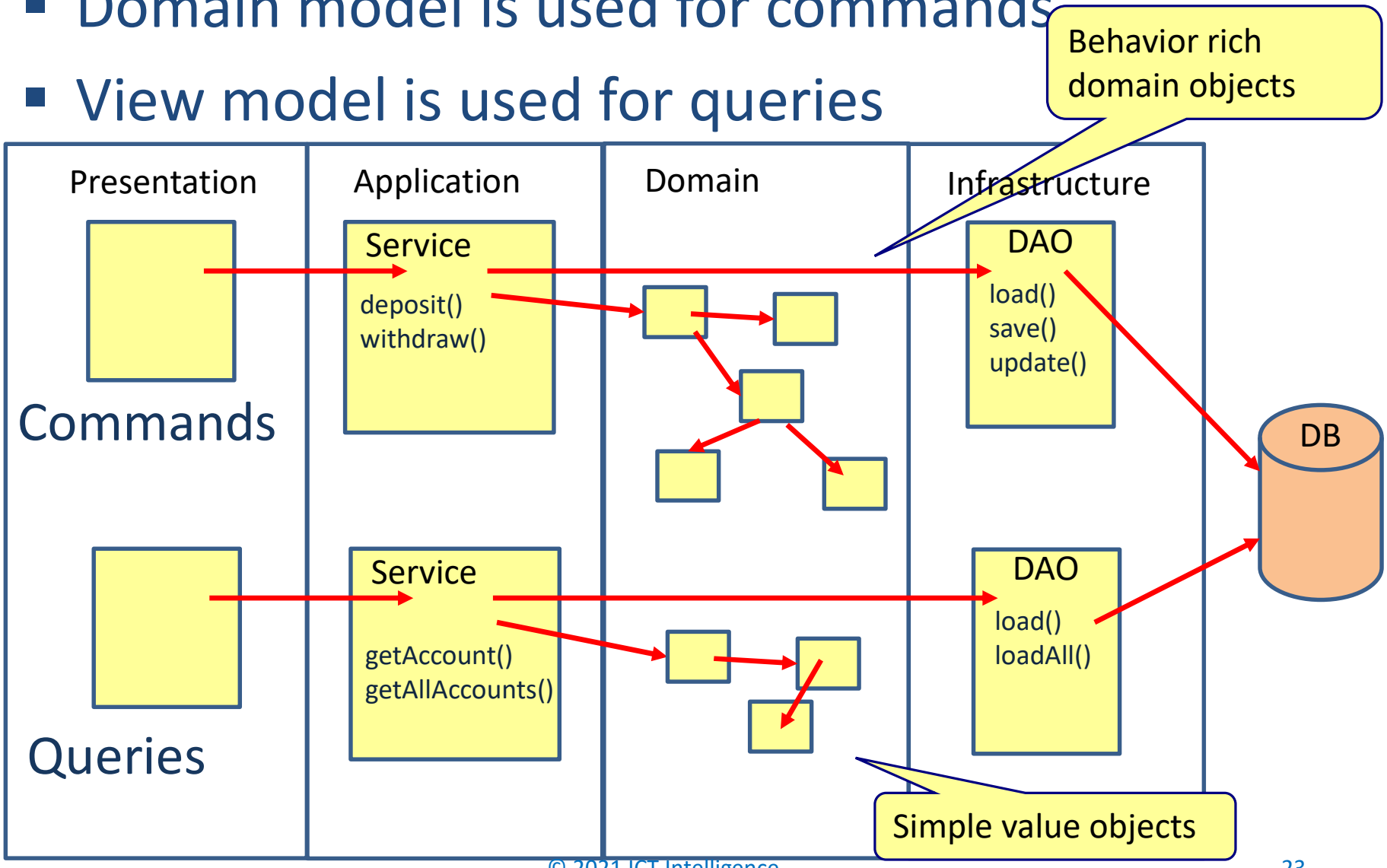
Complex aggregate because of UI needs

We don't need this complexity for commands

```
public class Customer
{
    // ...
    public ContactDetails ContactDetails { get; private set; }
    public LoyaltyStatus LoyaltyStatus { get; private set; }
    public Money GiftCertBalance { get; private set; }
    public IEnumerable<Address> AddressBook { get; private set; }
}
```

CQRS

- Domain model is used for commands
- View model is used for queries



2 services instead of one

Traditional service

CustomerService
void MakeCustomerPreferred(CustomerId)
Customer GetCustomer(CustomerId)
CustomerSet GetCustomersWithName(Name)
CustomerSet GetPreferredCustomers()
void ChangeCustomerLocale(CustomerId, NewLocale)
void CreateCustomer(Customer)
void EditCustomerDetails(CustomerDetails)

Service with CQRS

CustomerWriteService
void MakeCustomerPreferred(CustomerId)
void ChangeCustomerLocale(CustomerId, NewLocale)
void CreateCustomer(Customer)
void EditCustomerDetails(CustomerDetails)

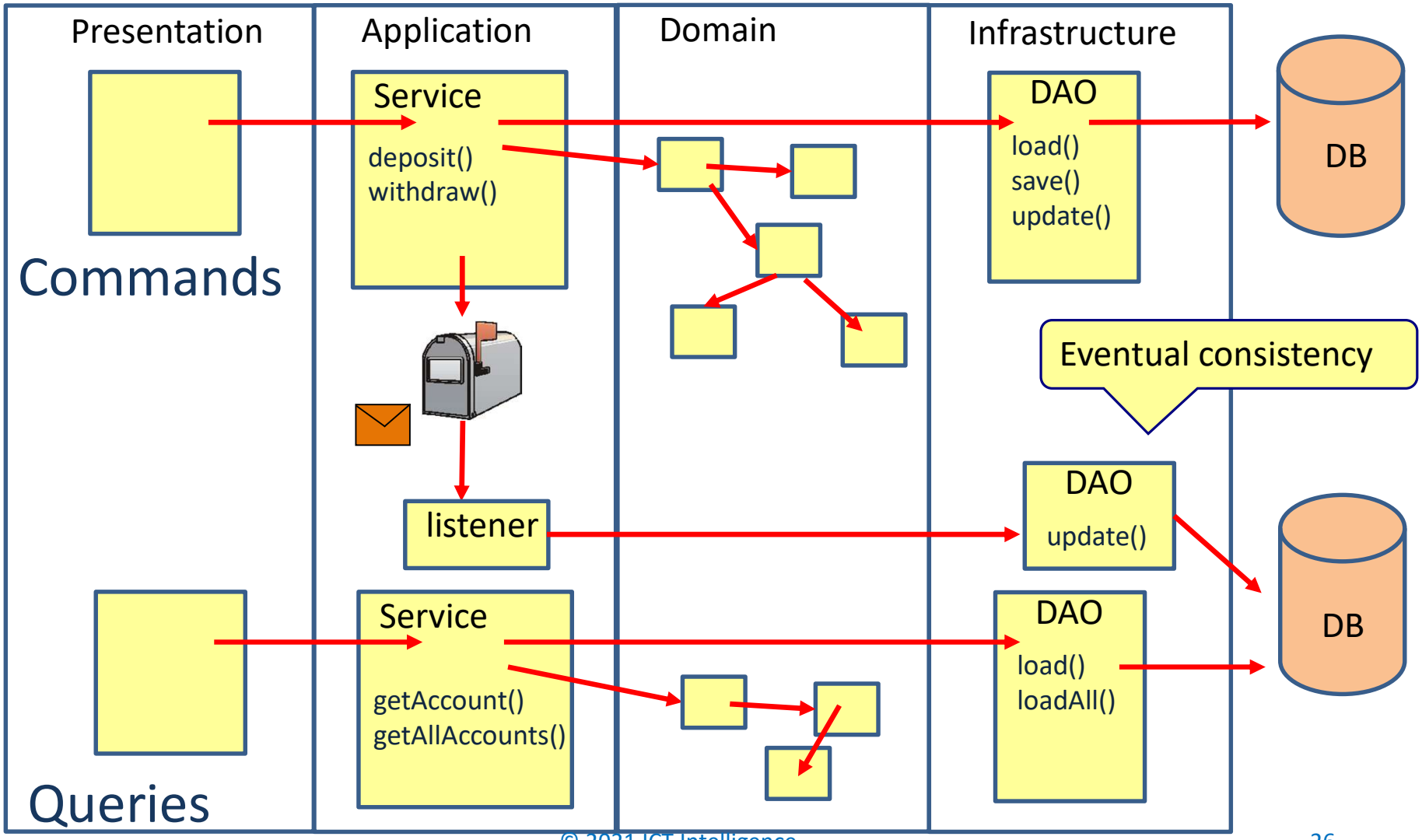
CustomerReadService
Customer GetCustomer(CustomerId)
CustomerSet GetCustomersWithName(Name)
CustomerSet GetPreferredCustomers()

Architectural properties

- Command and query side have different architectural properties
- Consistency
  - Command: needs consistency
  - Query: eventual consistency is mostly OK
- Data storage
  - Command: you want a normalized schema (3rd NF)
  - Query: denormalized (1st NF) is good for performance (no joins)
- Scalability
  - Command: commands happens not that often. Scalability is often not important.
  - Query: queries happen very often, scalability is important

Eventual consistency

- Views will become eventual consistent



CQRS advantages

- The query side is very simple
  - No transactions
- The query side can be optimized for performance
  - Fast noSQL database
  - Caching is easy
- Queries and commands can be scaled independently

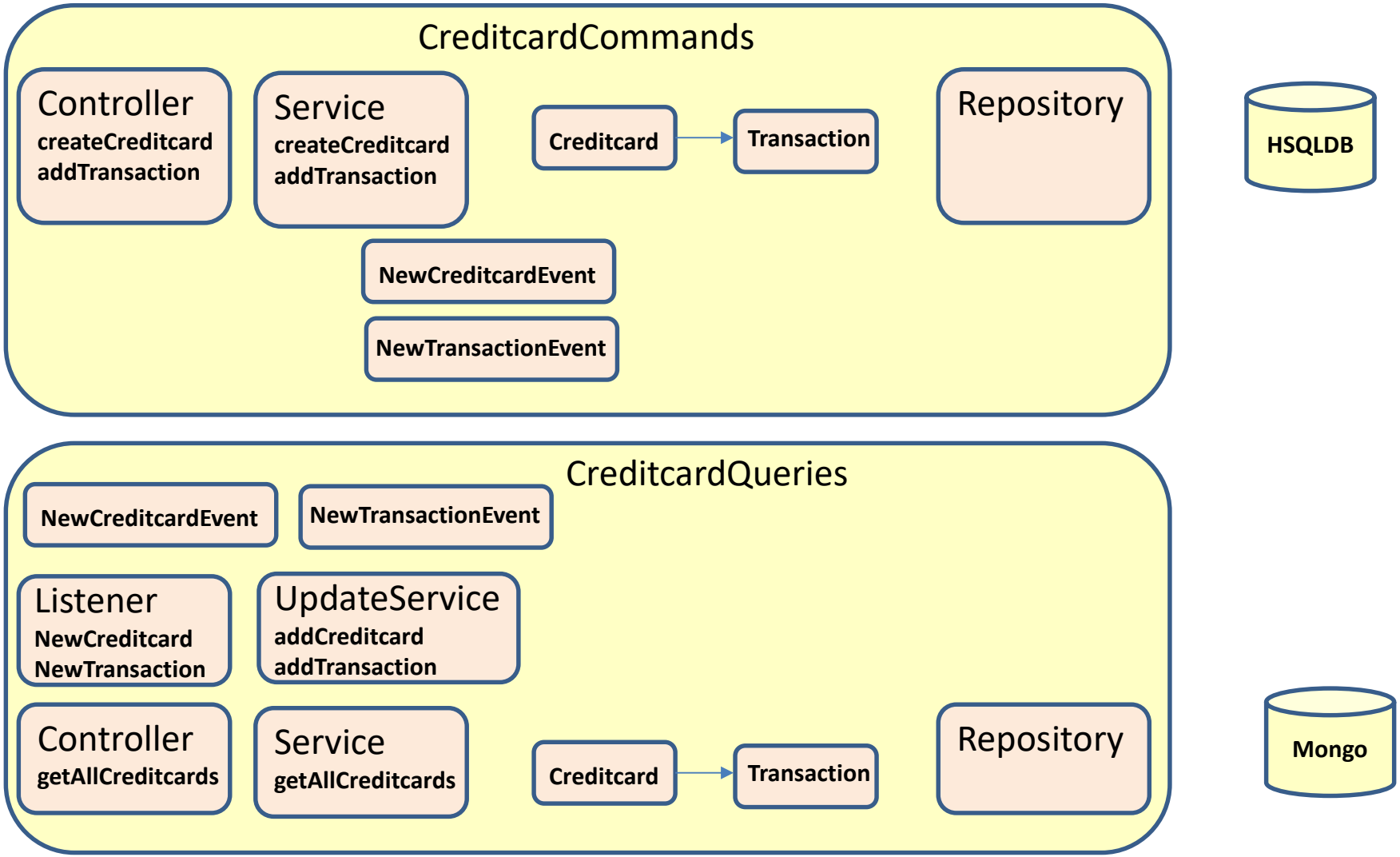
When to use CQRS?

- When queries and commands have different scaling requirements
- When read performance is critical
- When your screens start to look very different then your tables
- When you apply event sourcing

When not to use CQRS

- Systems with simple CRUD functionality
- When you need strict consistency (instead of eventual consistency)

CQRS example

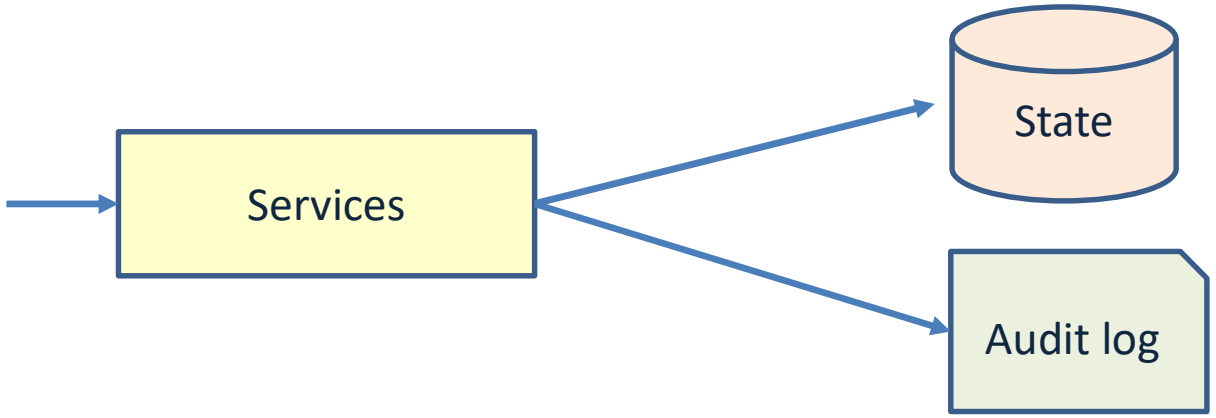


EVENT SOURCING

State based persistance

- You capture where you are, but not how you got there
- You don’t know what happened in the past
- You cannot fix bad state because of bugs in the code

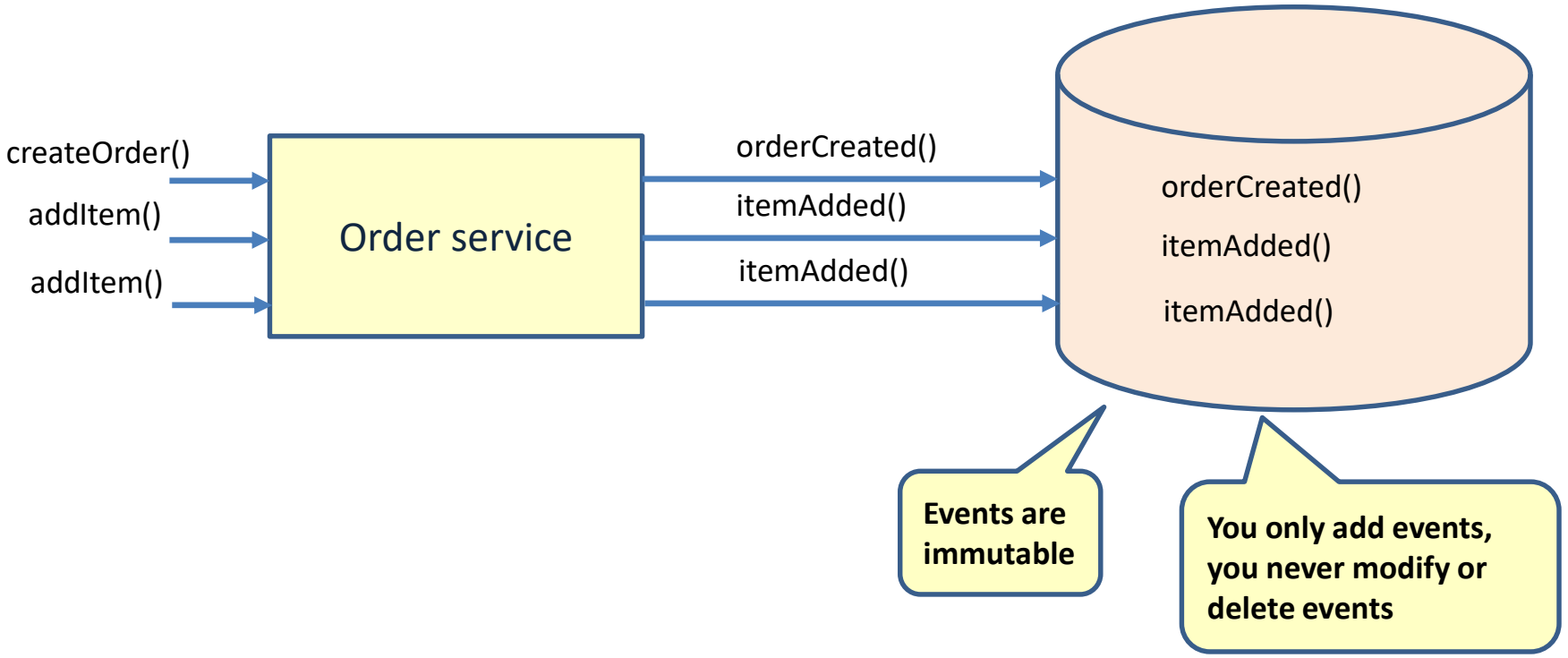
Audit log



- Now we know how we got there
- What if the audit log gets out of sync with the state due to a bug?
- Which is the source of truth? (Audit log)

Event sourcing

- Only capture how we got there



Event sourcing

- Bank account
  - Store all deposits and withdrawals
- Phone account
  - Store all phone calls
- Version control systems
  - Each commit or change is an event
- DBMS
  - Databases keep a transaction log of all inserts, deletes and updates

Storing state and storing events

- Store state

ID	status	data…
101	accepted	…

Store entity data

- Event sourcing

Entity ID	Entity type	Event ID	Event type	Event data…
101	Order	901	OrderCreated	…
101	Order	902	OrderApproved	…
101	Order	903	OrderShipped	…

Store state changing events

Store the state of a system

Structural representation

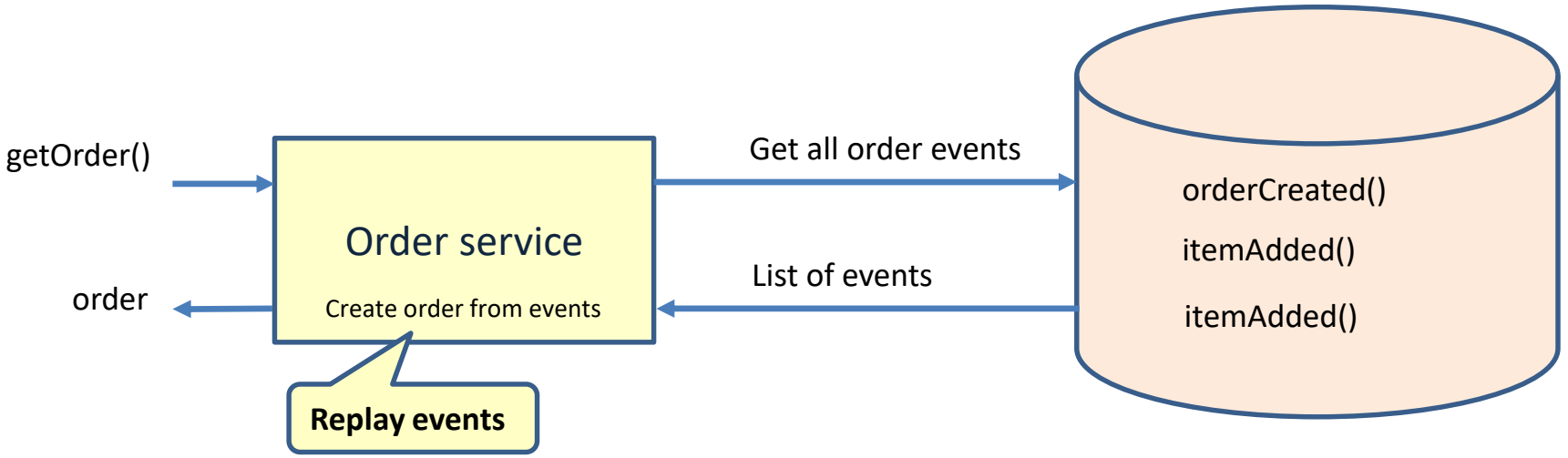
- List of ordered goods
- Payment information
- Shipping information

Store the events of a system

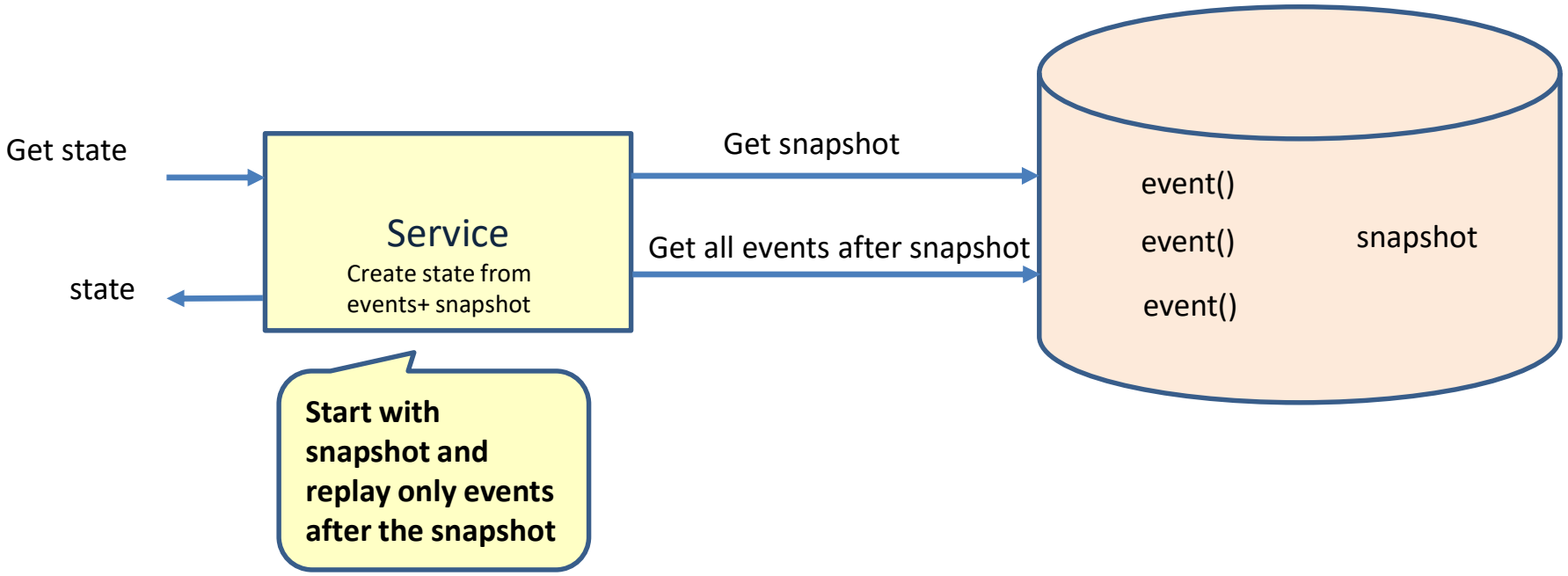
Event representation

- Add item #1
- Add item #2
- Add payment info
- Update item #2
- Remove item #1
- Add shipping info

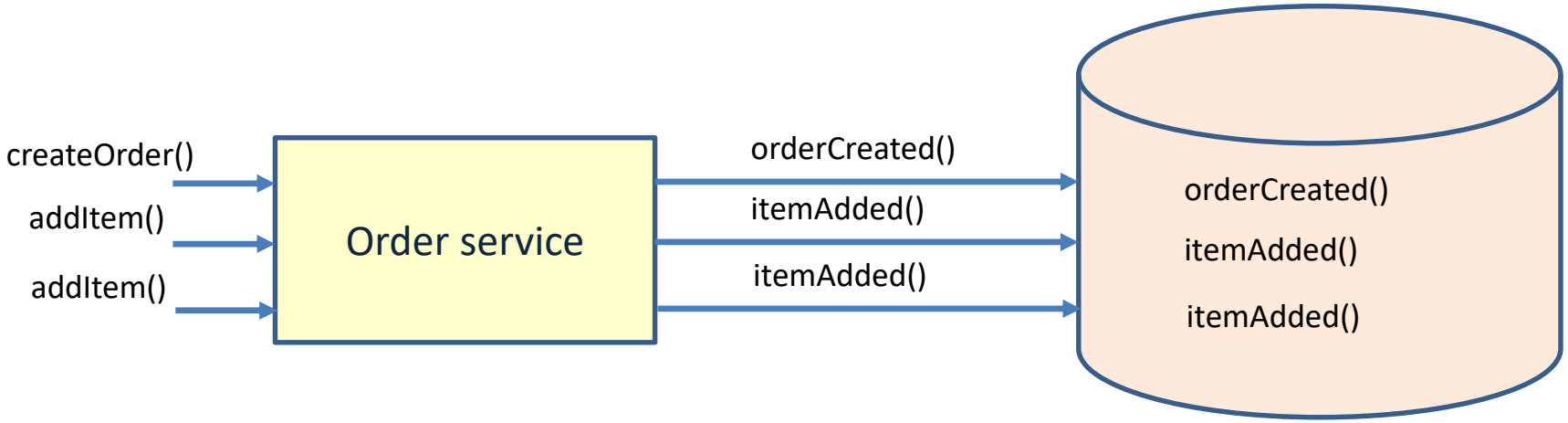
How do we get the state?



Snapshots

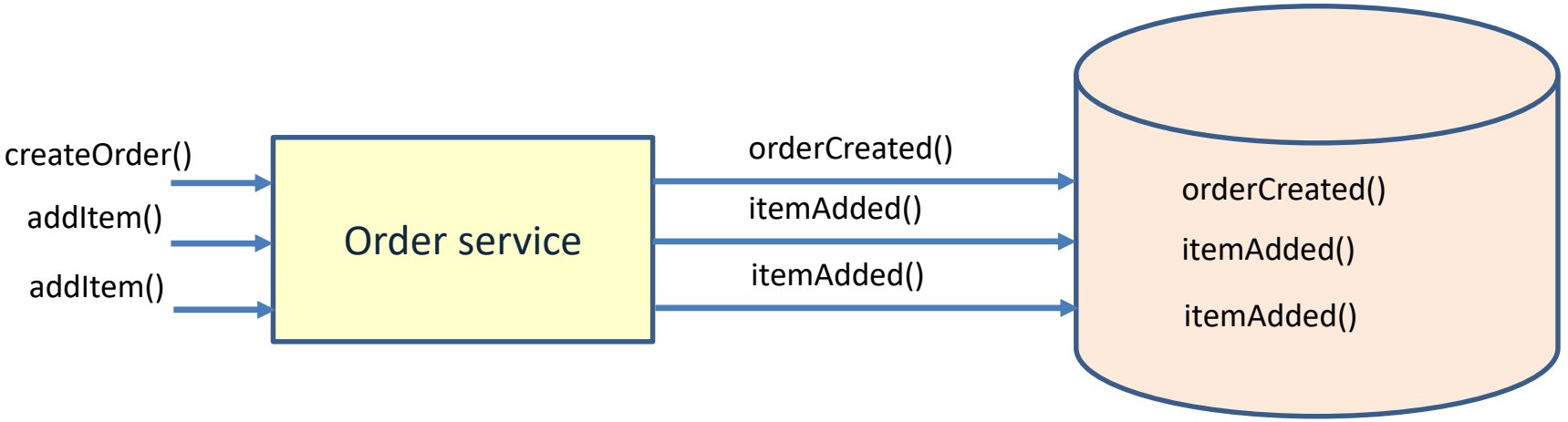


Event sourcing advantages



- You don’t miss a thing
  - Business can analyze history of events
  - Bugs can be solved easier
- Creates a built-in audit log
- Allows for rewinding or undo changes
- Append is usually more efficient in databases
- No transactions needed

Event sourcing disadvantages



- You cannot easily see what your state is
  - You always have to replay the events first (lower read performance)
  - Always use CQRS when you apply event sourcing
- You need more storage
- What if events have to change?
  - We need event versioning
  - The logic needs to support all versions

When to consider event sourcing?

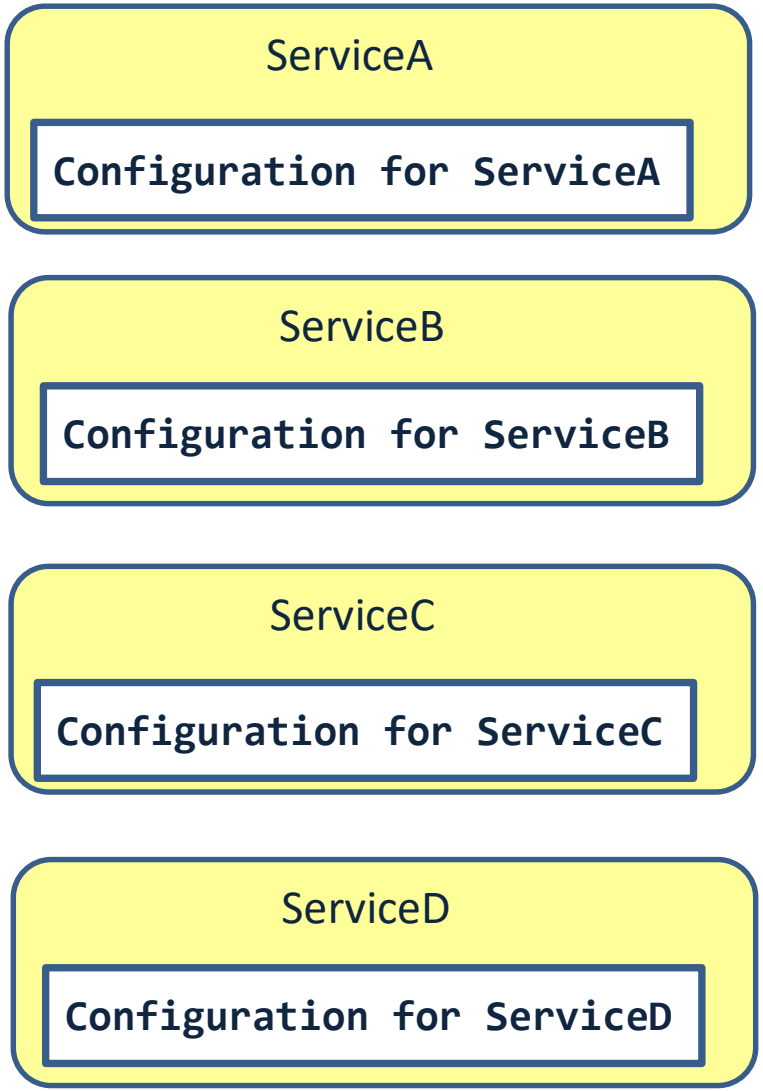
- When you need an audit log
- When you want to derive additional business value from the event history
- When you need fast performance on the command side

CENTRALIZED CONFIGURATION SERVICE

Configuration in microservices

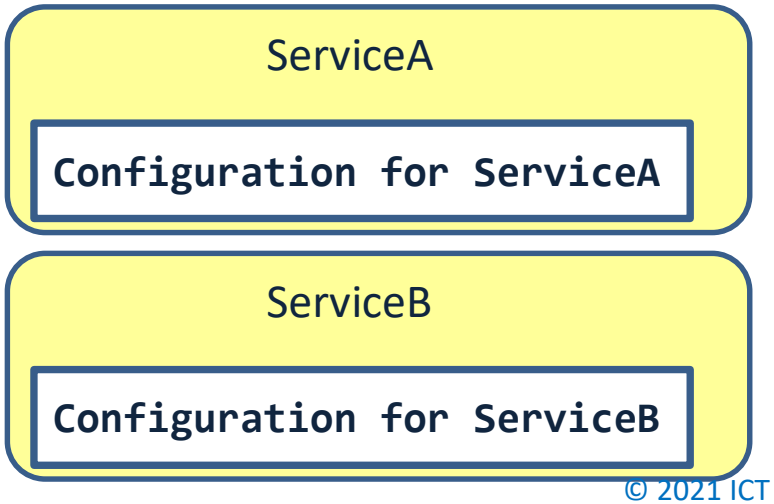
- Remove settings from code
- Change runtime behavior
- Enforce consistency across elastic services

Local configuration

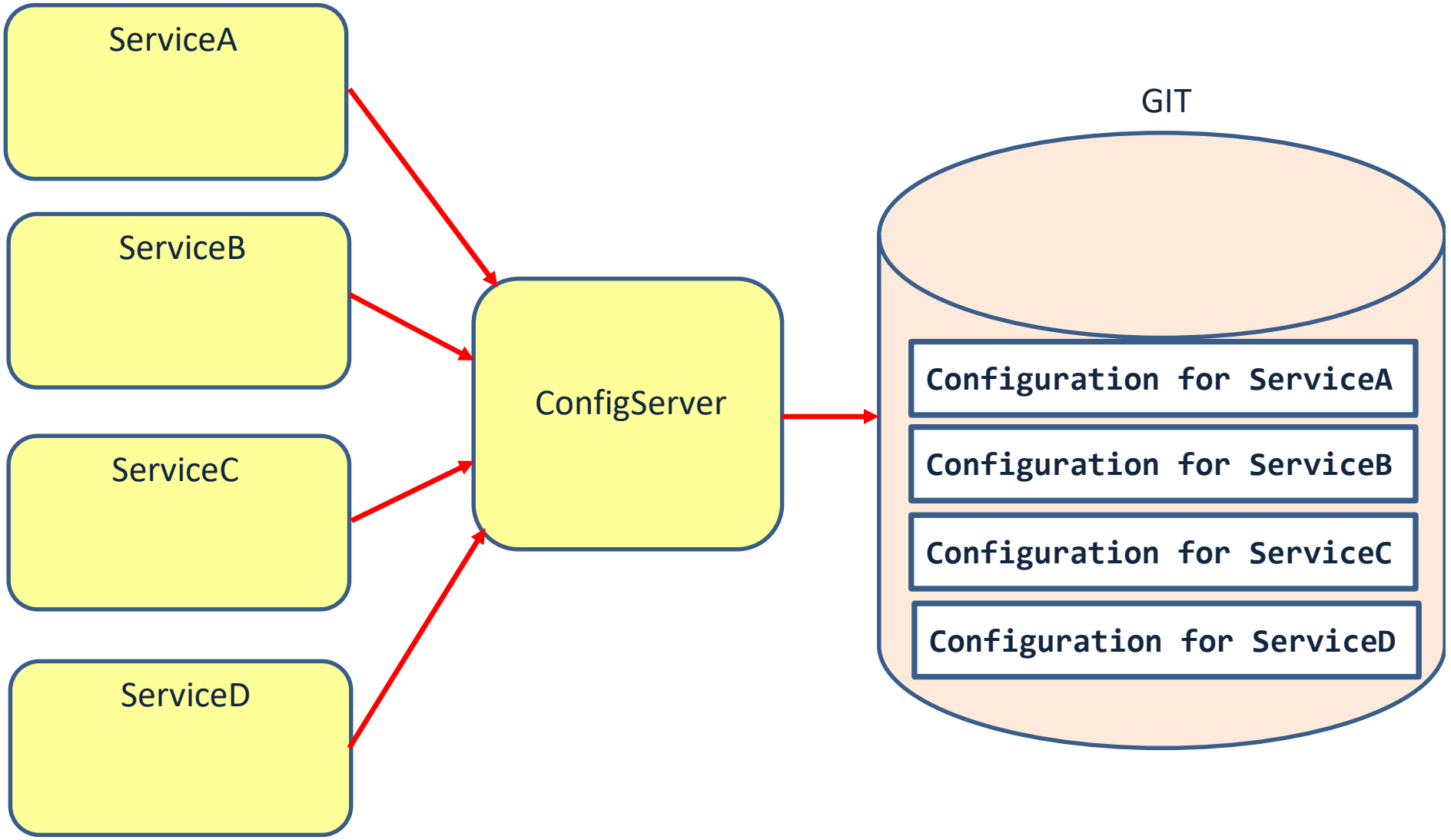


Local configuration challenges

- When we change the configuration we need to rebuild and redeploy the application
- Configuration may contain sensitive information
- Some of the properties are the same among services: lots of duplication

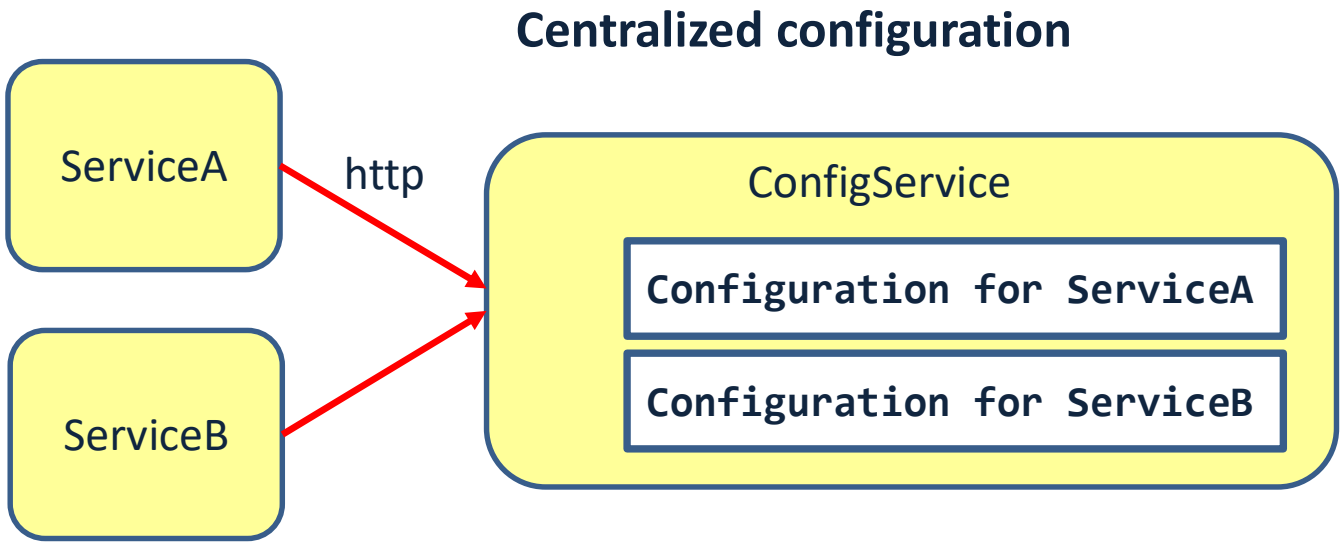


Spring cloud config server



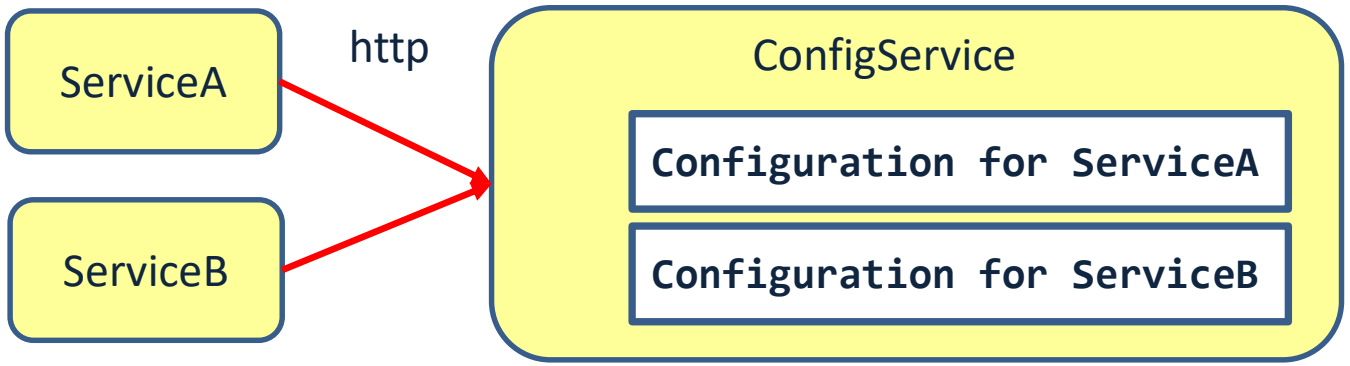
Spring cloud config

- HTTP access to centralized configuration

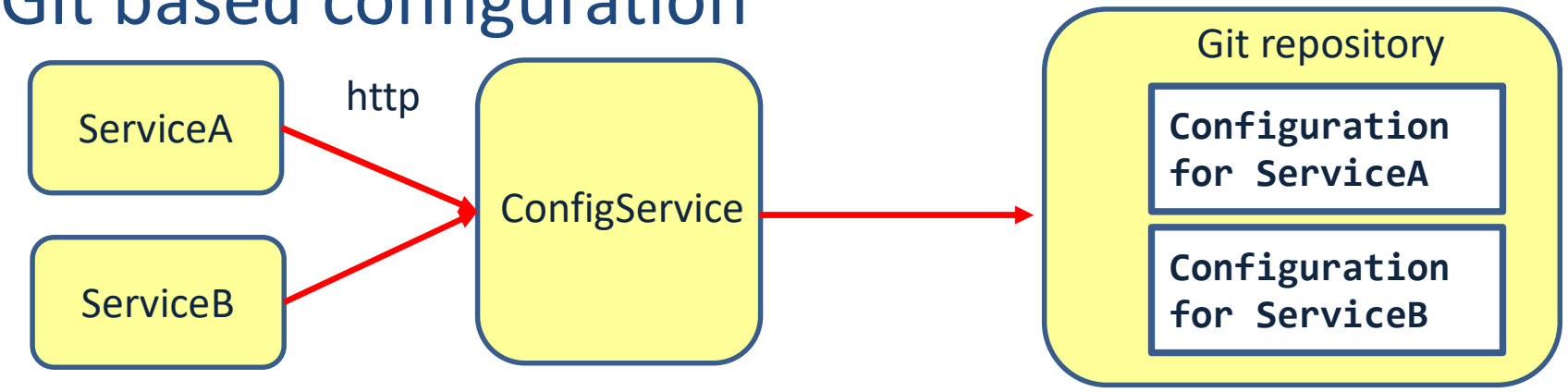


Spring cloud config

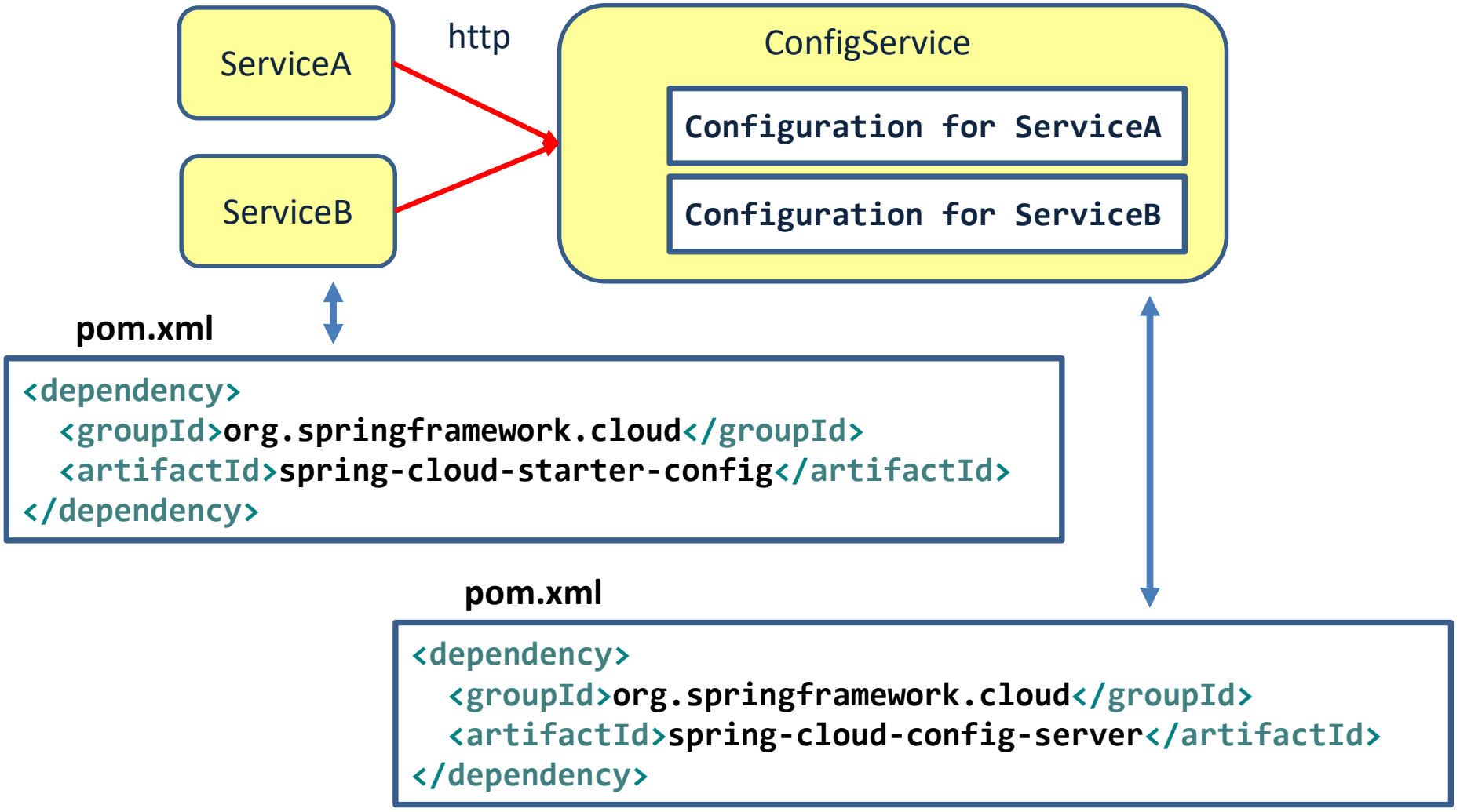
- File based configuration



- Git based configuration



Spring cloud config example



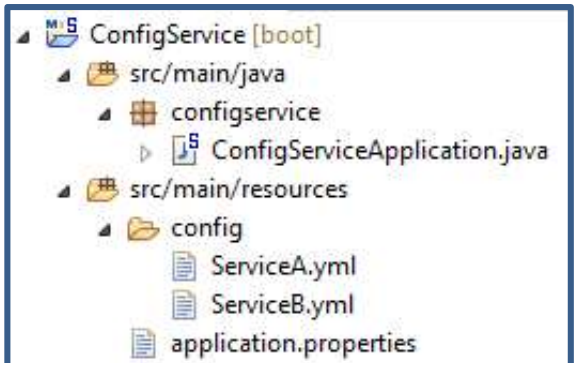
Configuration server

```java
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.config.server.EnableConfigServer;

@SpringBootApplication
@EnableConfigServer
public class ConfigServiceApplication {

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

application.properties

Do not use GIT, but local files

```
spring.profiles.active=native
server.port=8888
```



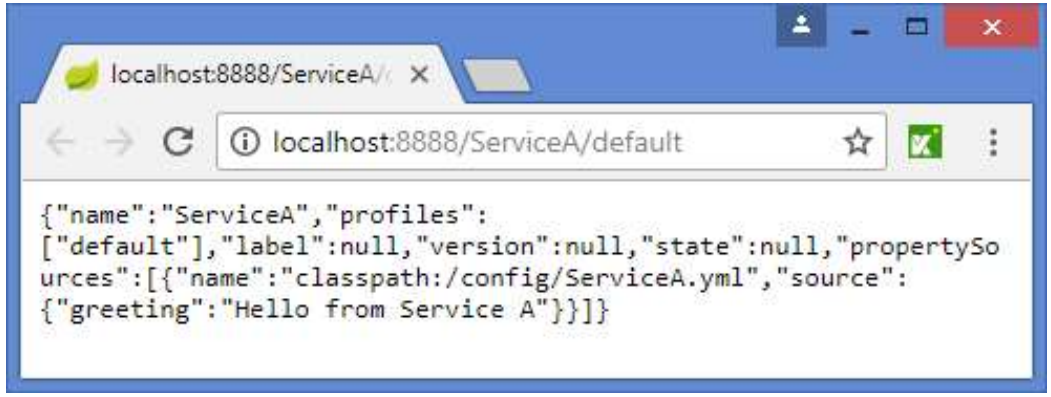
config/ServiceA.yml

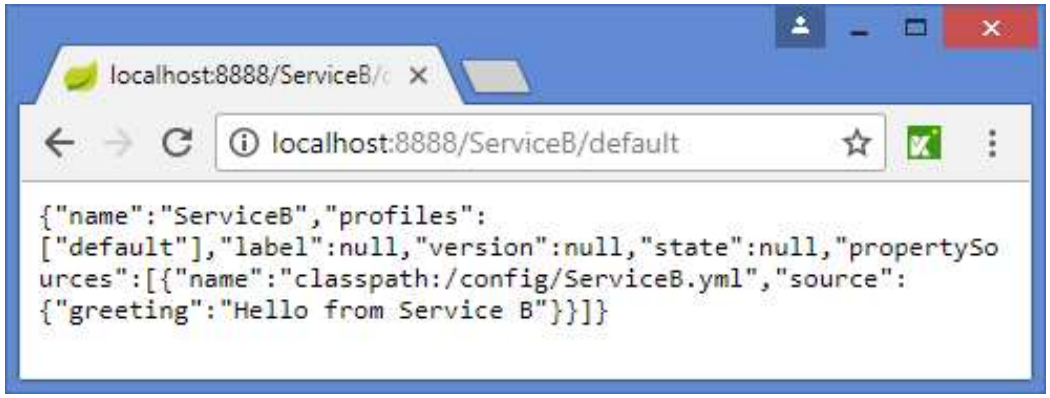
```
greeting: Hello from Service A
```

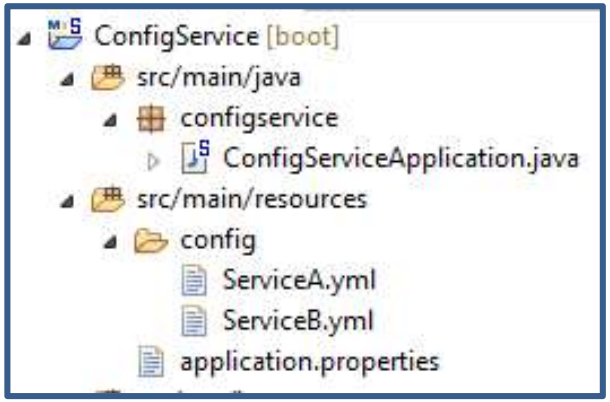
config/ServiceB.yml

```
greeting: Hello from Service B
```

Configuration server



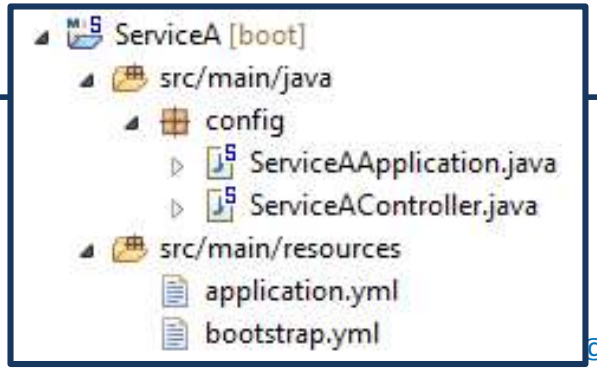




Configuration client: ServiceA

```java
@SpringBootApplication
public class ServiceAApplication {

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

```java
@RestController
public class ServiceAController {
  @Value("${greeting}")
  private String message;

  @RequestMapping("/")
  public String getName() {
    return message;
  }
}
```



application.yml

```yaml
server:
  port: 8090
```

bootstrap.yml

```yaml
spring:
  application:
    name: ServiceA
  cloud:
    config:
      url: http://localhost:8888
```

Spring cloud applications

- 2 configuration files
  - bootstrap.yml
    - Is loaded before applications.yml
    - Is needed when configuration is stored on a remote config server
    - Contains
      - The name of the application
      - Location of the configuration server
  - applications.yml
    - Contains standard application configuration

bootstrap.yml

```yaml
spring:
  application:
    name: ServiceA
  cloud:
    config:
      url: http://localhost:8888
```

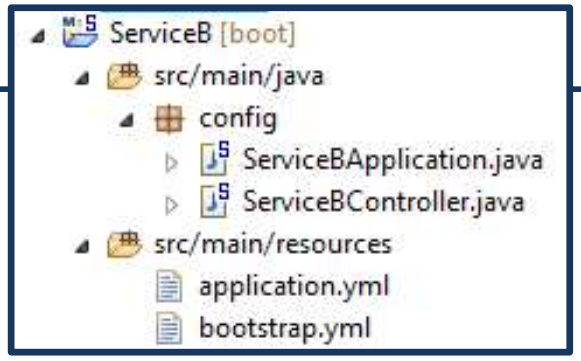
application.yml

```yaml
server:
  port: 8090
```

Configuration client: ServiceB

```java
@SpringBootApplication
public class ServiceBApplication {

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

```java
@RestController
public class ServiceBController {
  @Value("${greeting}")
  private String message;

  @RequestMapping("/")
  public String getName() {
    return message;
  }
}
```



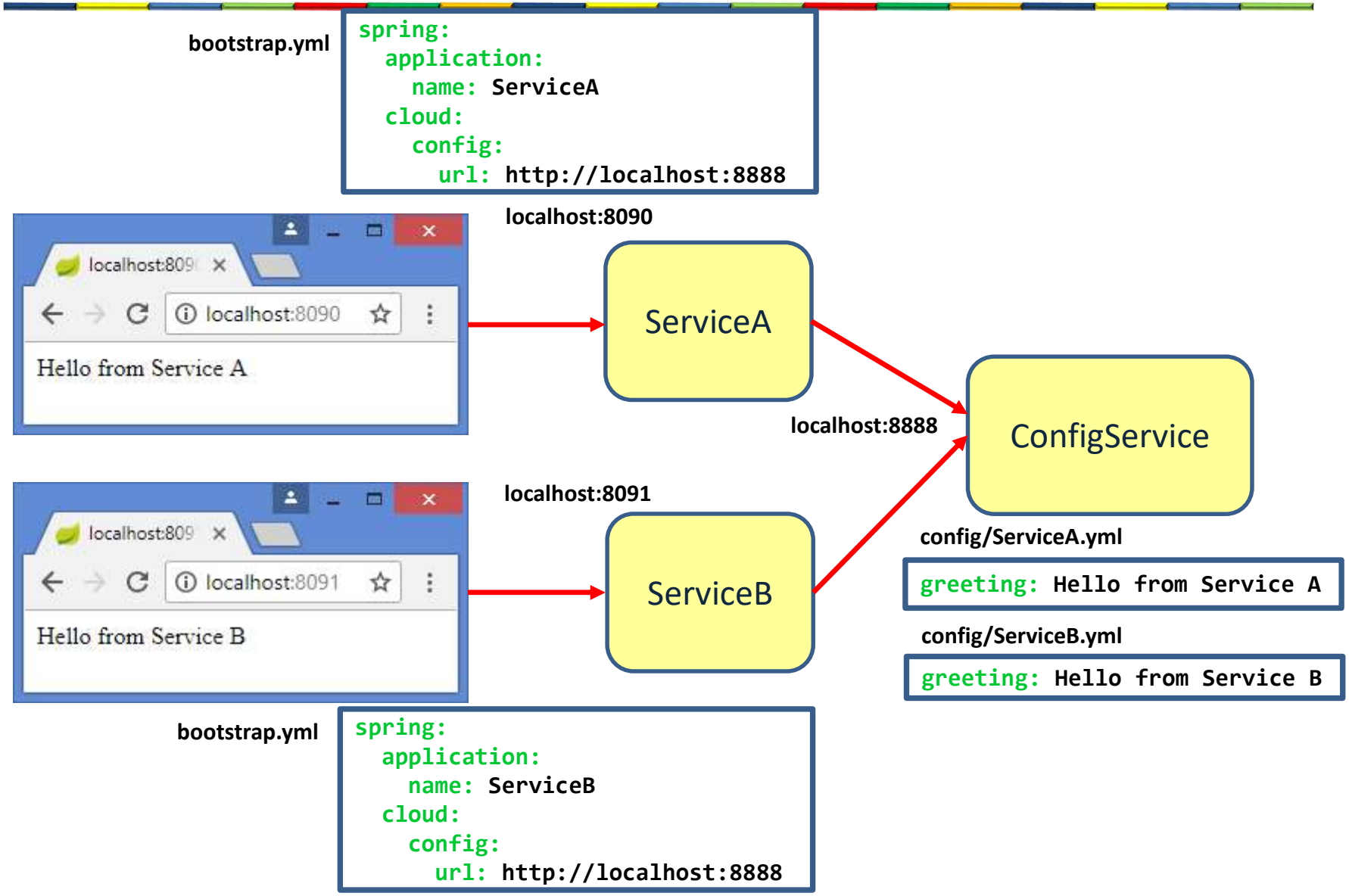
application.yml

```yaml
server:
  port: 8091
```

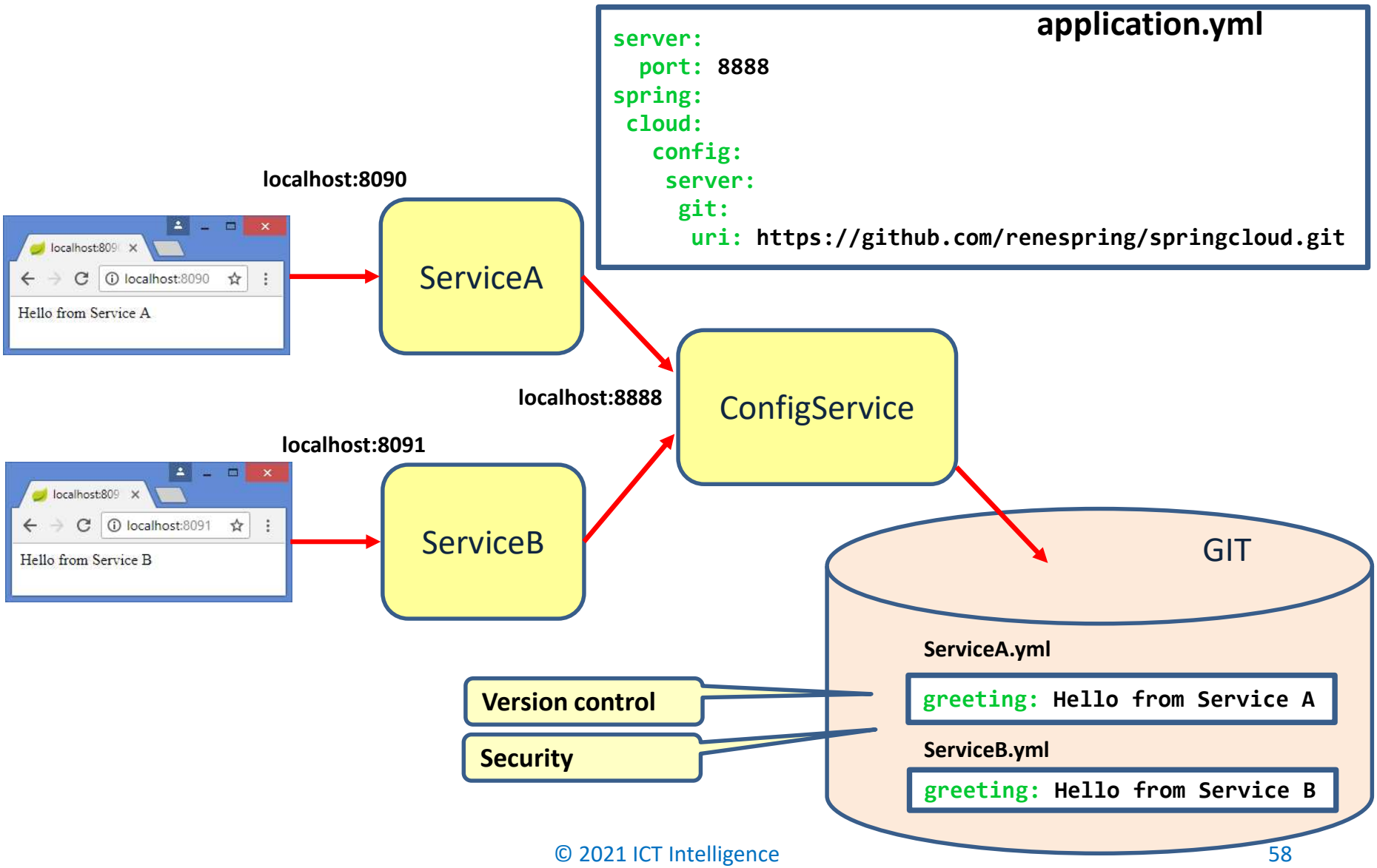
bootstrap.yml

```yaml
spring:
  application:
    name: ServiceB
  cloud:
    config:
      url: http://localhost:8888
```

Use of the Config Server



Using Git (or GitHub)



Refreshing configuration

- Option 1: use @RefreshScope and "/actuator/refresh" event

@RefreshScope

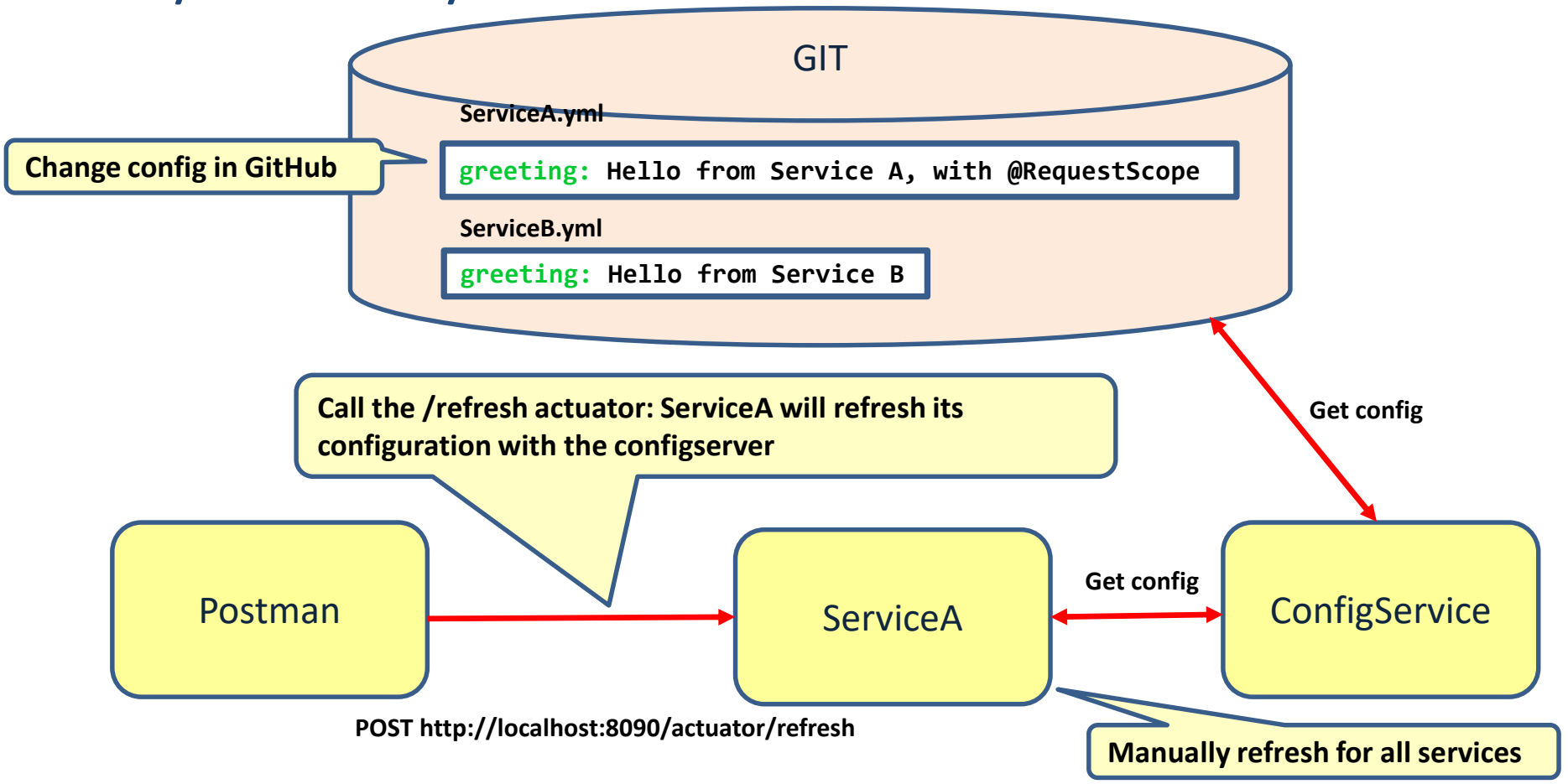
```
@RestController
@RefreshScope
public class ServiceAController {
  @Value("${greeting}")
  private String message;

  @RequestMapping("/")
  public String getName() {
    return message;
  }
}
```

```
server:
 port: 8888

spring:
 cloud:
  config:
   server:
    git:
     uri:
https://github.com/renespring/springcloud.git


management:
 endpoints:
  web:
   exposure:
    include: refresh
```

Expose the /refresh actuator

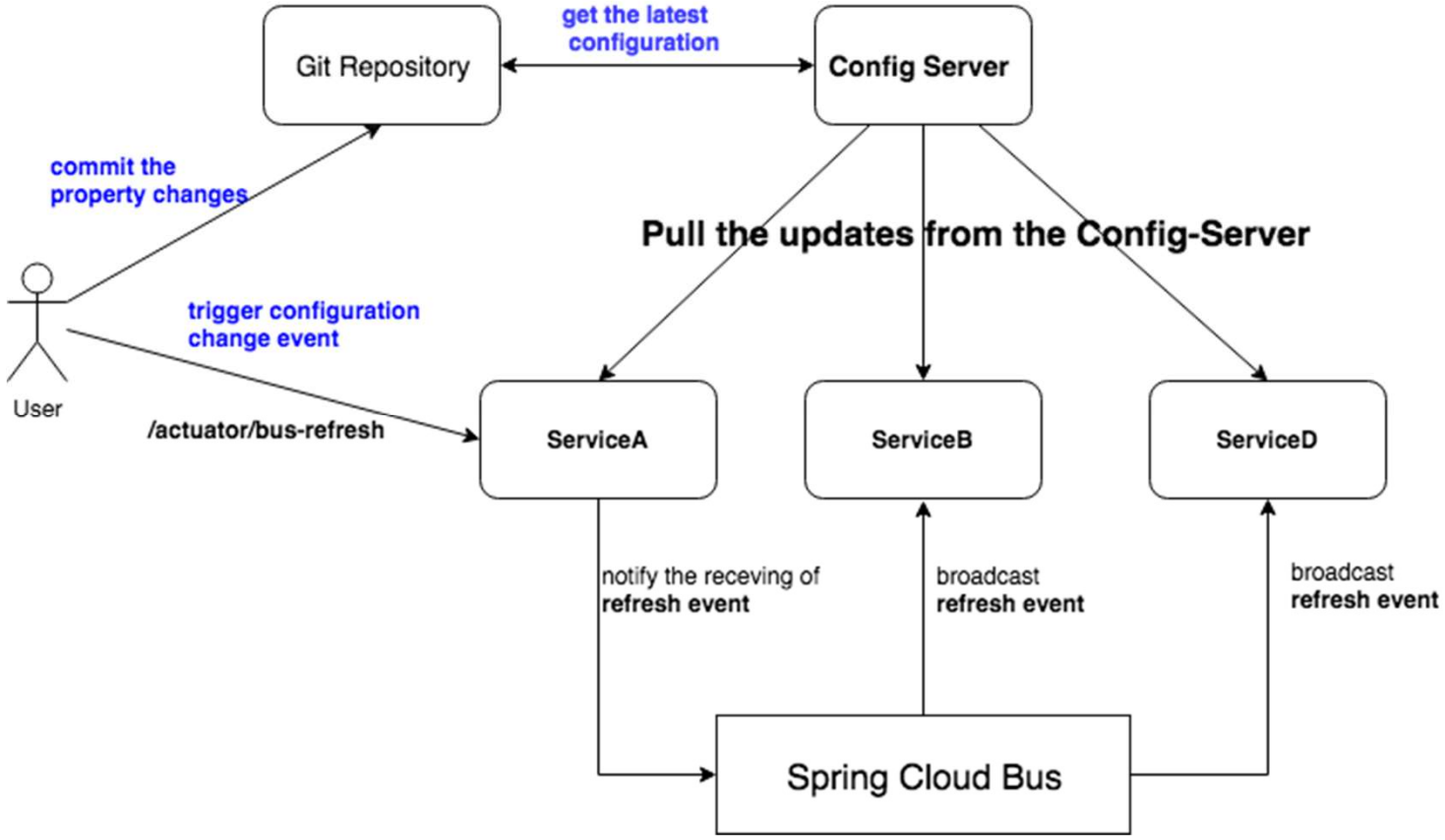
Refreshing configuration

- Option 1: use @RefreshScope and "/actuator/refresh" event

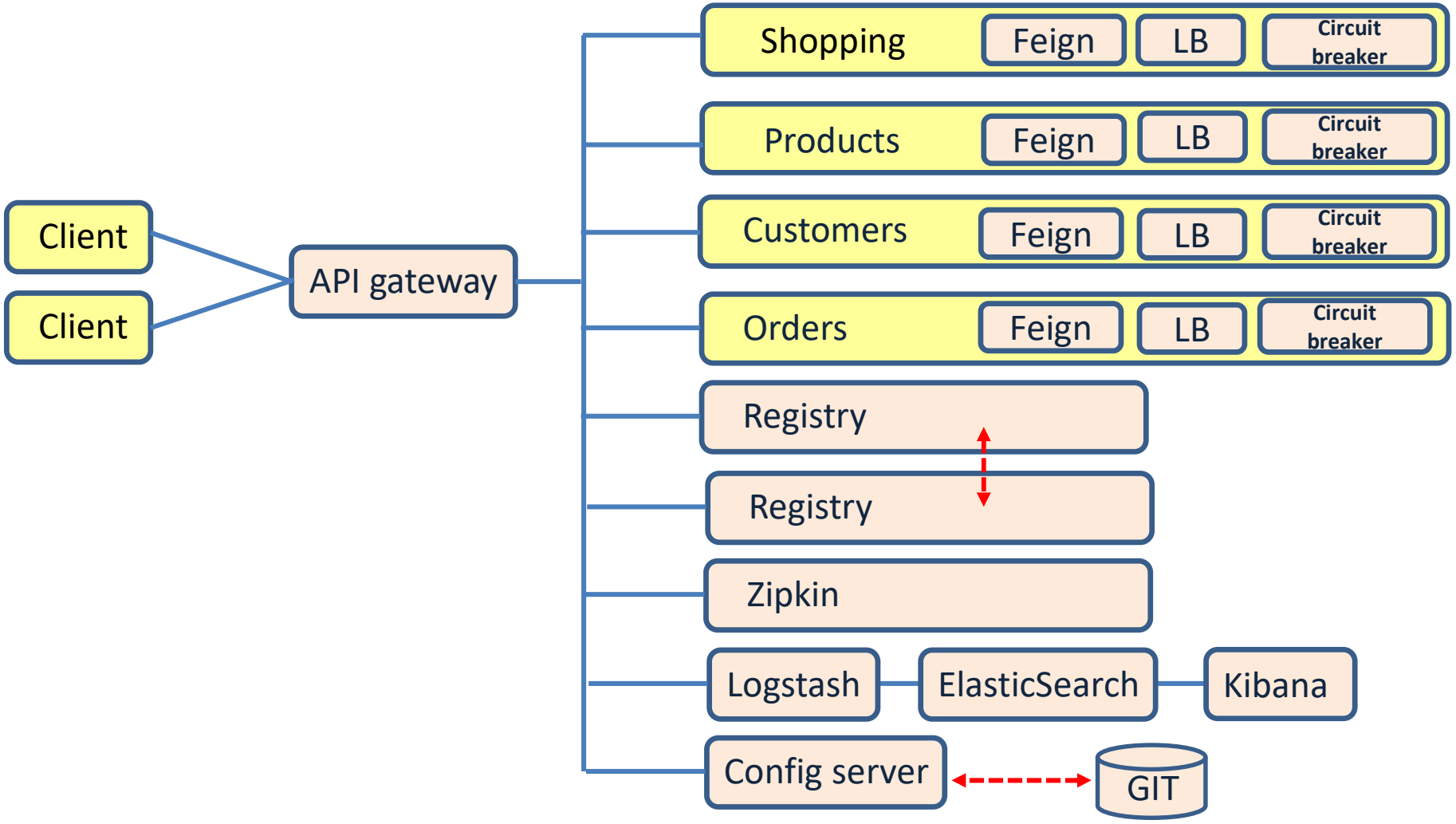


Refreshing configuration

- Option 2: use Spring cloud bus for broadcasting refresh events



Implementing microservices



Challenges of a microservice architecture

Challenge	Solution
Complex communication	Feign Registry API gateway
Performance	
Resilience	Registry replicas Load balancing between multiple service instances Circuit breaker
Security	
Transactions	
Keep data in sync	
Keep interfaces in sync	Spring cloud contract
Keep configuration in sync	Config server
Monitor health of microservices	ELK + beats
Follow/monitor business processes	Zipkin ELK