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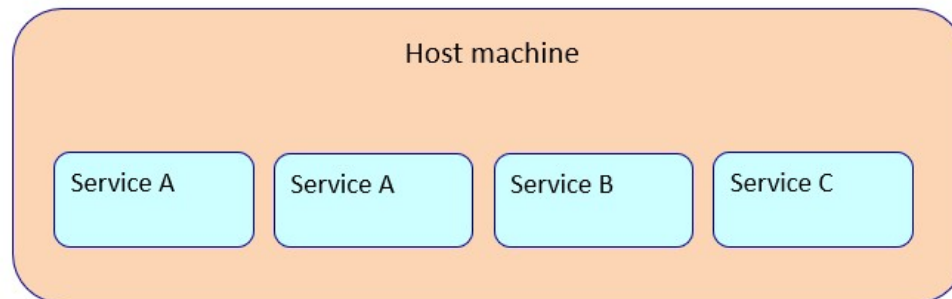
- **Drawbacks**
  - Poor isolation
  - Poor visibility of resource utilization
  - Difficult to constrain resource utilization
  - Risk of dependency version conflicts
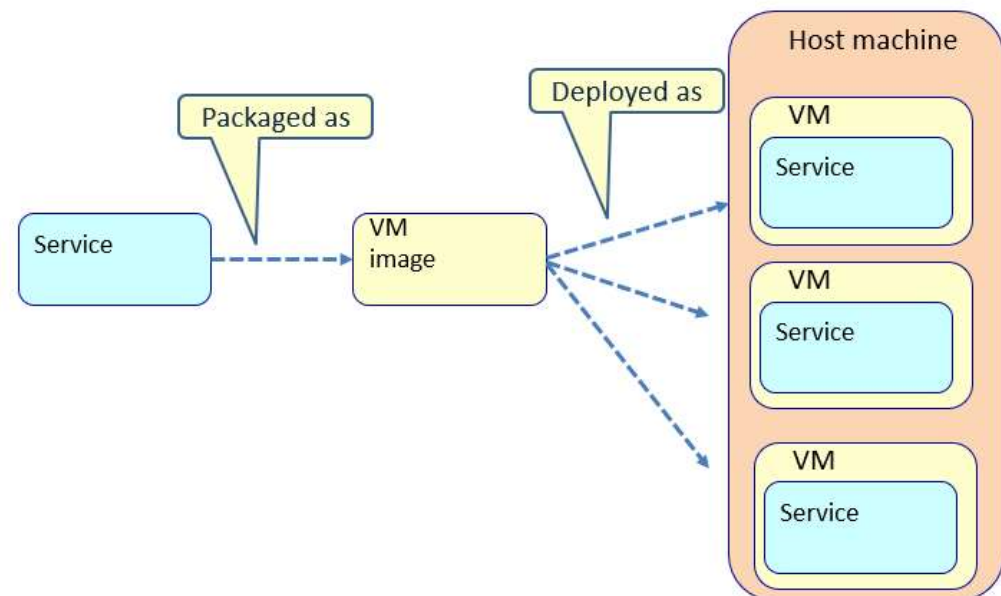 - Poor encapsulation of implementation technology

Host machine

| Service A | Service A | Service B | Service C |

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
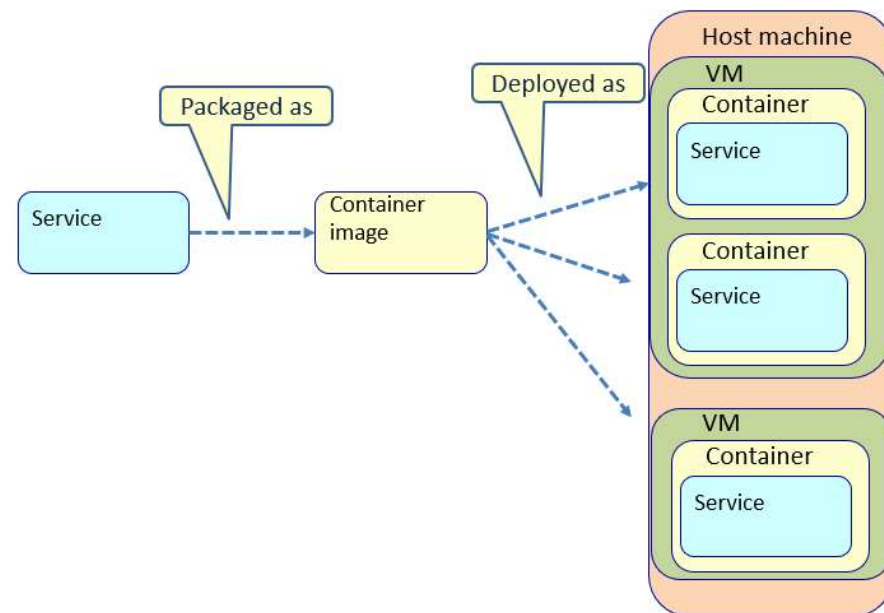 - Slow deployment

Service per container

Benefits

- Great isolation
- Great manageability
- Container encapsulates implementation technology
- Efficient resource utilization
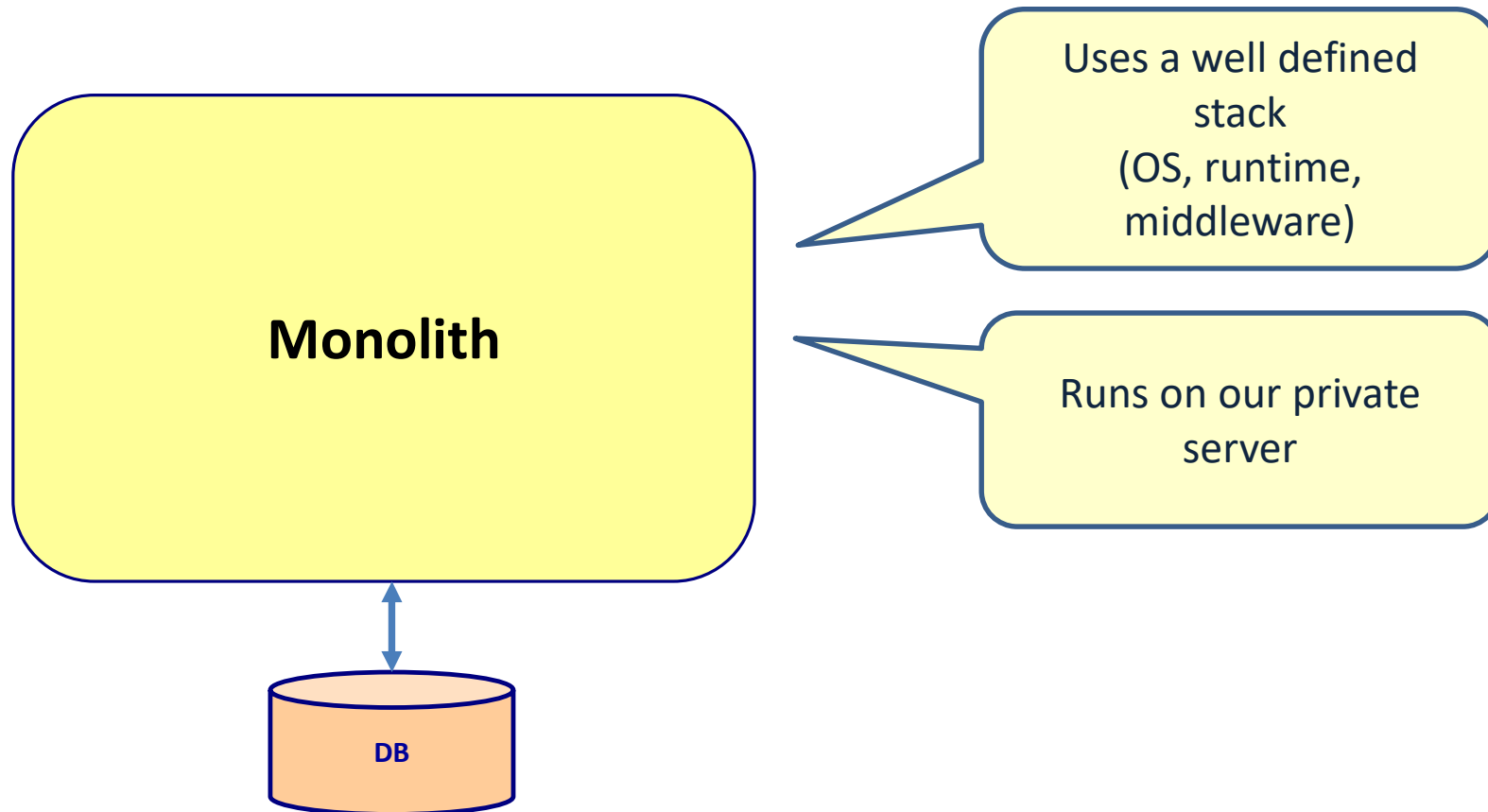- Fast deployment

Drawbacks

- Technology is not as mature as VM's
- Containers are not as secure as VM's

CONTAINERS

Monolith

Monolith

DB

Uses a well defined stack
(OS, runtime, middleware)

Runs on our private server

Microservice

Micro Service
DB

Micro Service
DB

Micro Service
DB

Micro Service
DB

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

Production Cluster

Public Cloud

QA server

Disaster recovery

Customer Data Center

Contributor's laptop

Production Servers

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
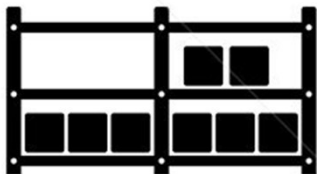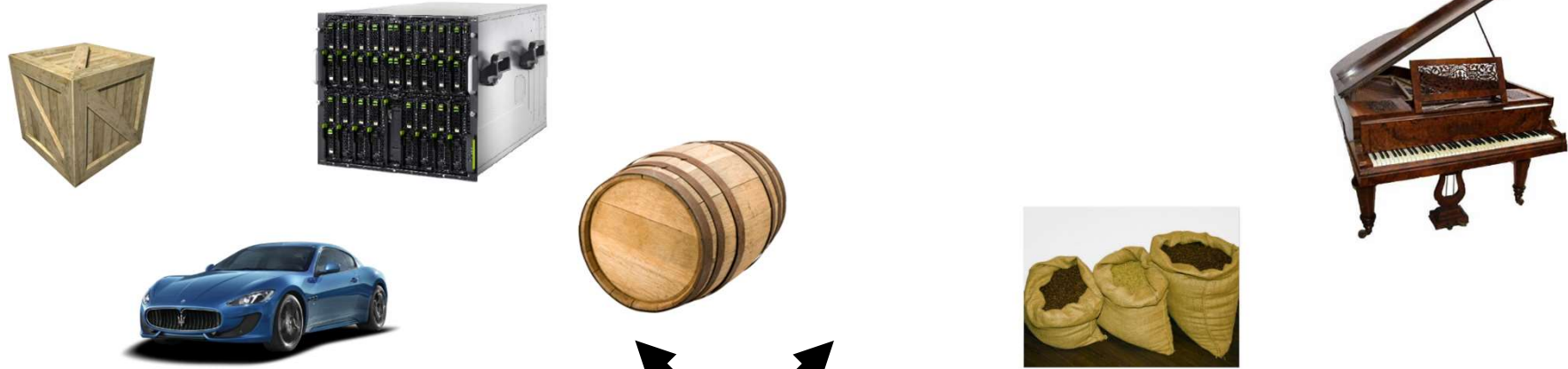
The solution
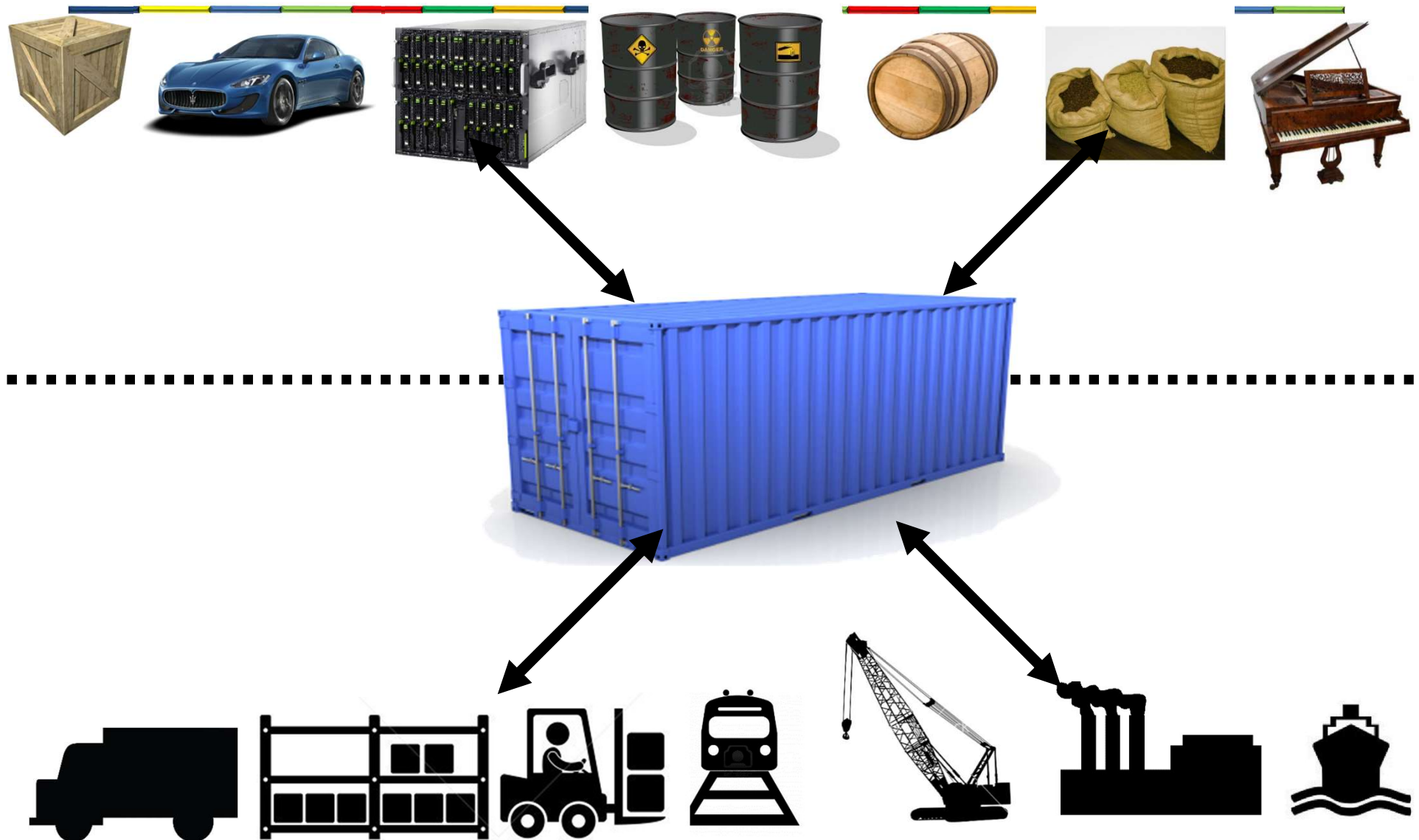
Docker is a shipping container for code

Static website   User DB   Web frontend   Queue   Analytics DB

Development VM   QA server   Customer Data Center   Public Cloud   Production Cluster   Contributor laptop
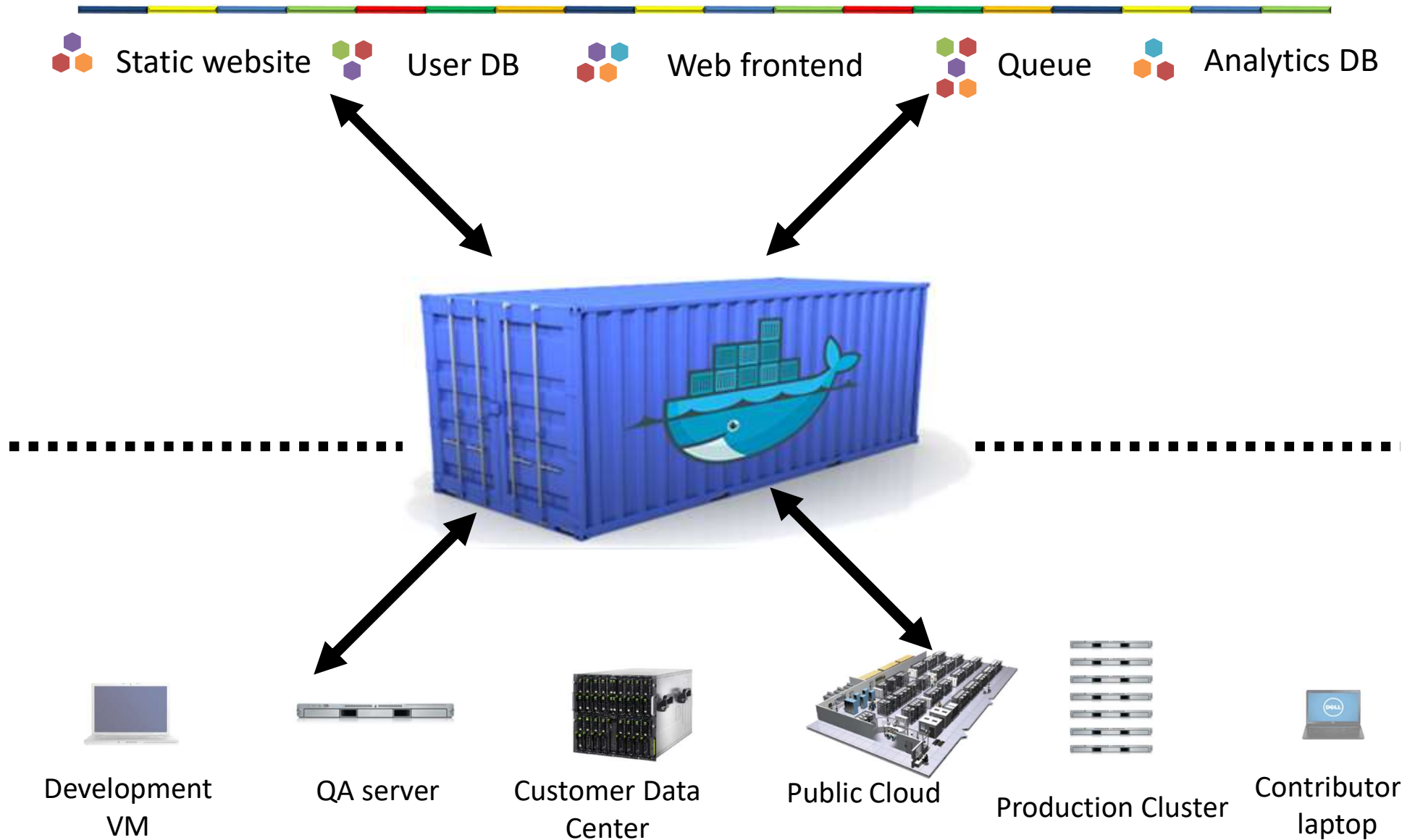
Separation of concern

- 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

Kubernetes

Docker swarm

App

Bin/ Library

Container

App

Bin/ Library

Container

App

Bin/ Library

Container

Container Runtime

Operating System

Hardware

Service discovery and load balancing

Storage orchestration

Automated rollouts and rollbacks of containers

Container configuration

Self-healing (start, stop, kill, restart containers, health checks)
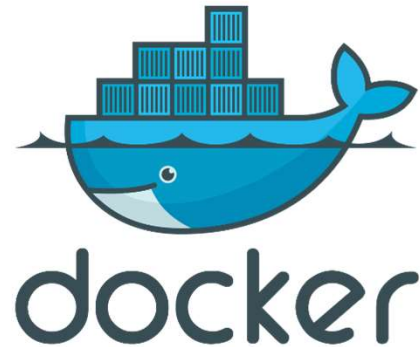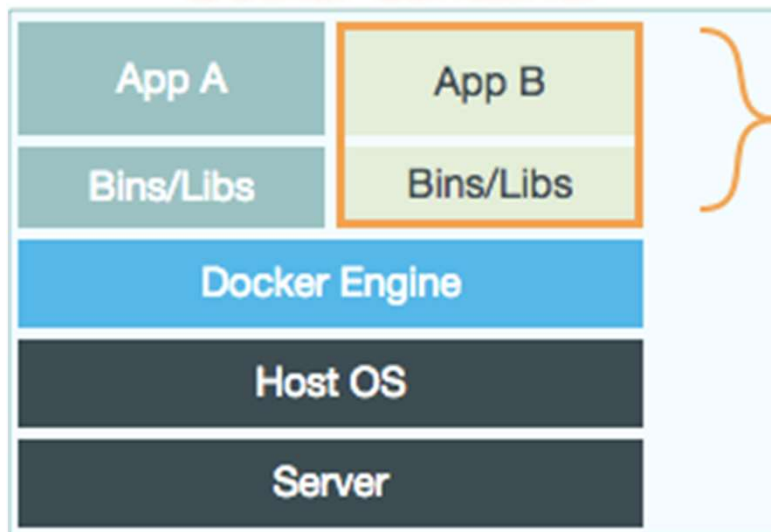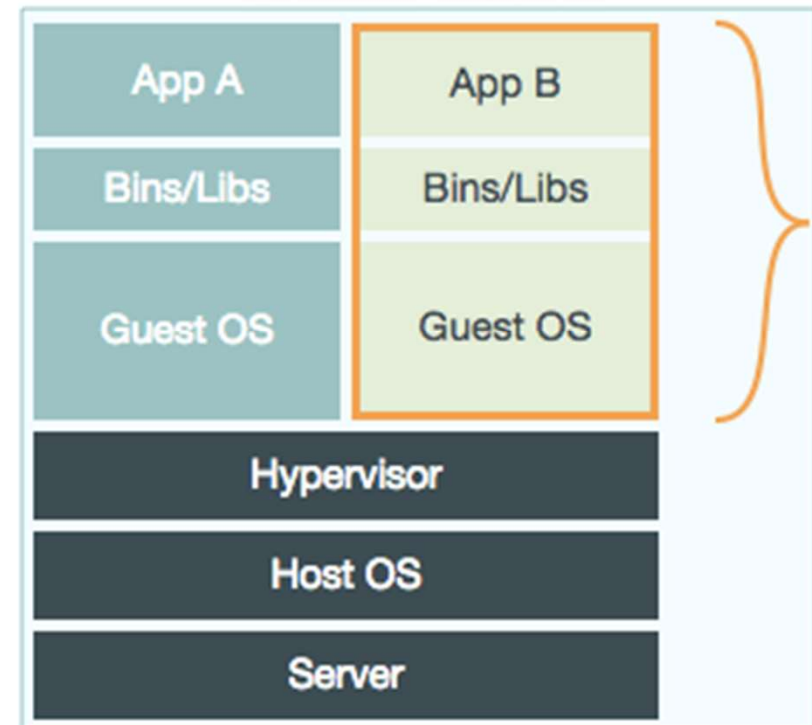
Security management
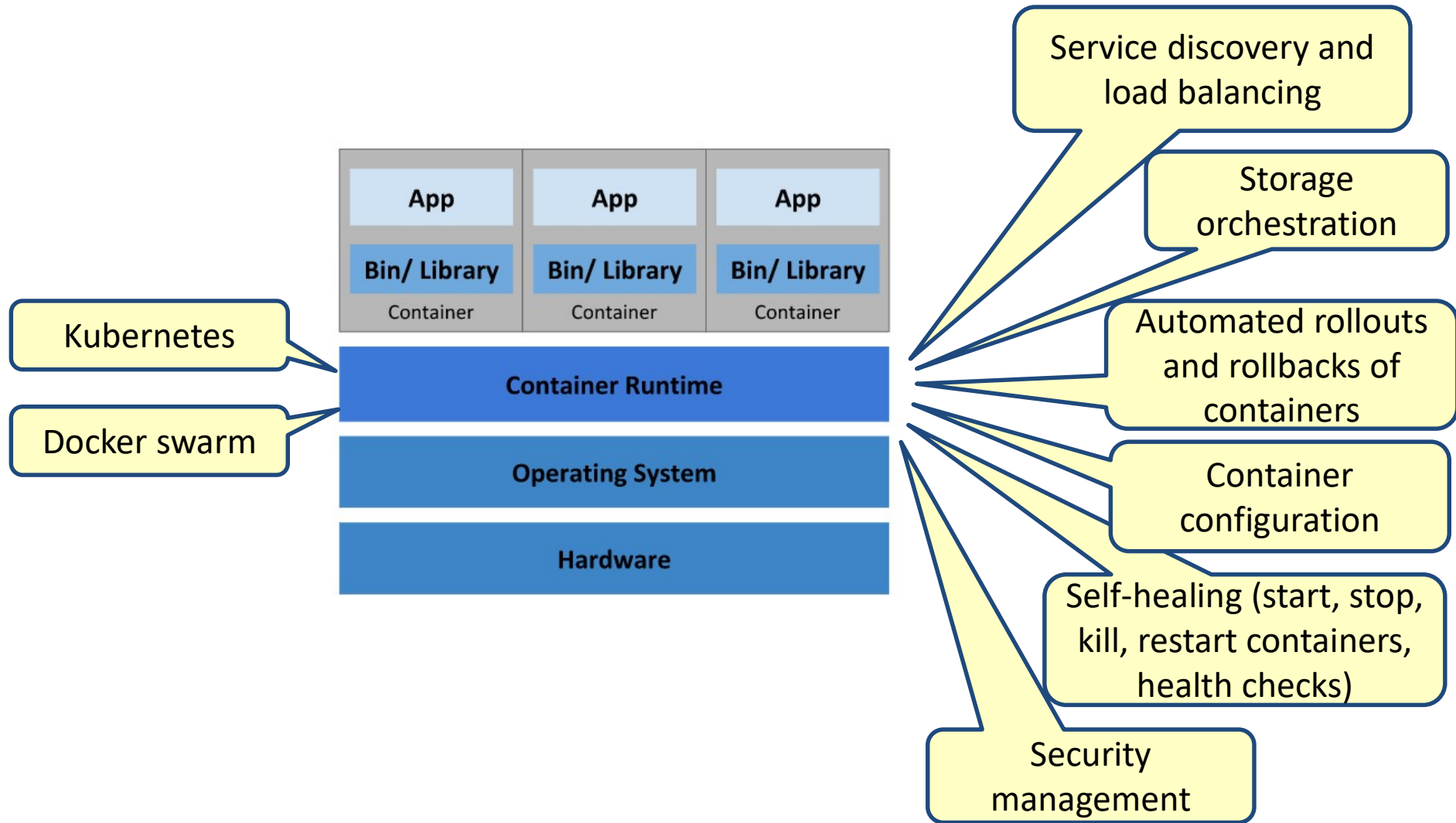
CQRS

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

Presentation	Application	Domain	Infrastructure

Service

deposit()
withdraw()
getAccount()
getAllAccounts()

DAO

load()
save()
update()
loadAll()

DB
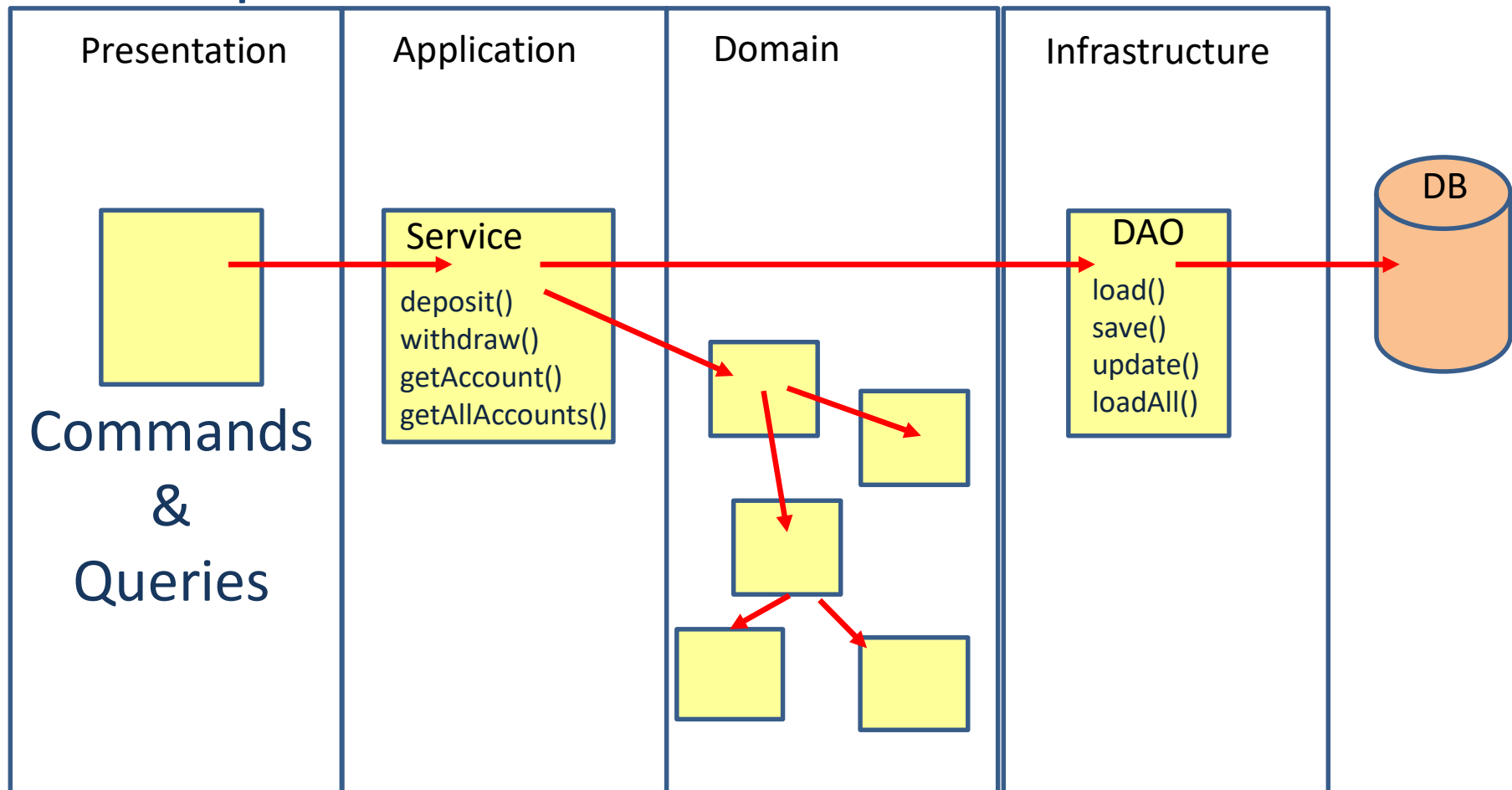
Commands
&
Queries

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

Presentation | Application | Domain | Infrastructure

Commands

Service
deposit()
withdraw()

DAO
load()
save()
update()

Behavior rich domain objects

DB

Queries

Service
getAccount()
getAllAccounts()

DAO
load()
loadAll()

Simple value objects

23
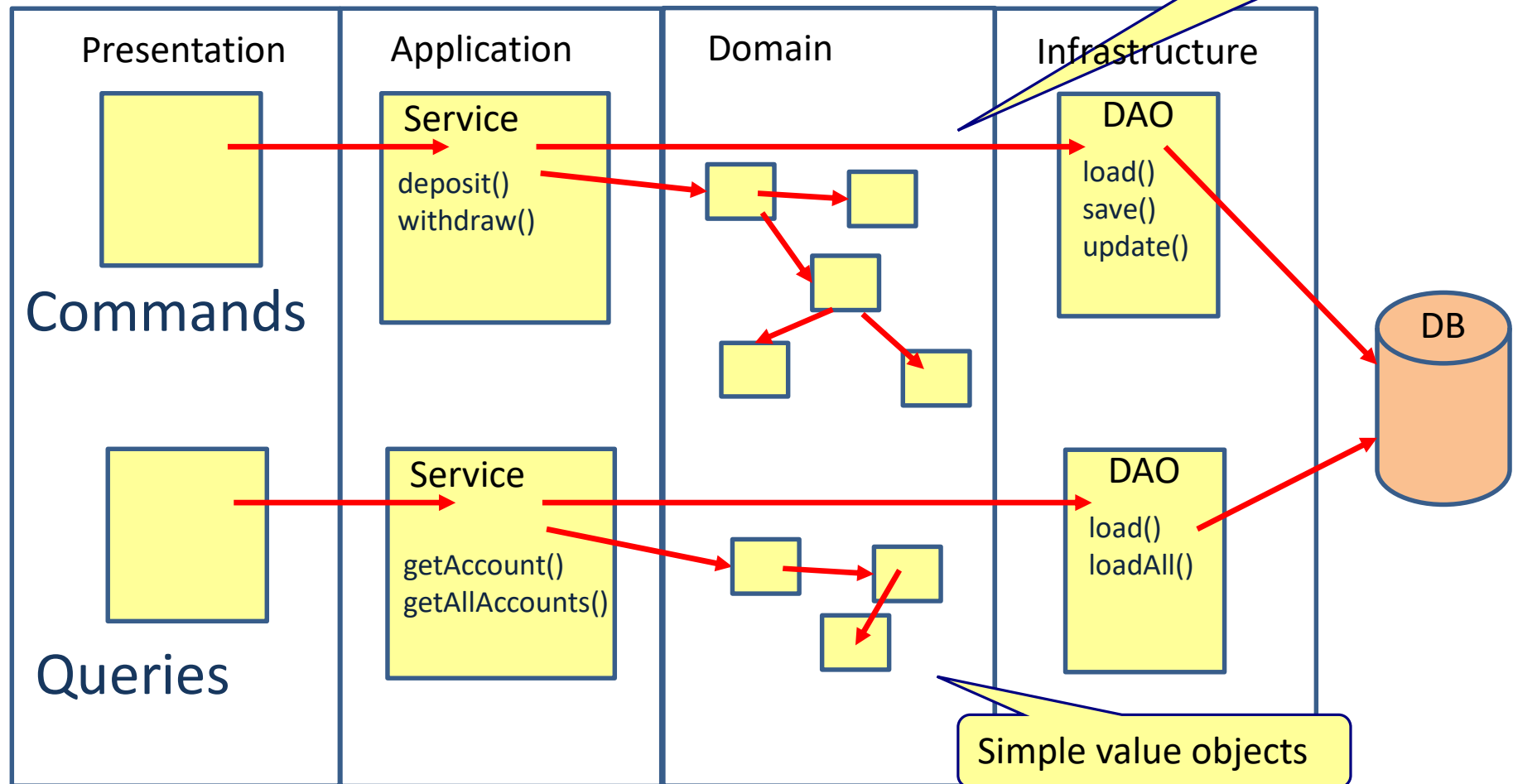
2 services instead of one

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

Presentation	Application	Domain	Infrastructure

Commands

Service
deposit()
withdraw()

DAO
load()
save()
update()

DB

Eventual consistency

listener

DAO
update()

DB

Queries

Service
getAccount()
getAllAccounts()

DAO
load()
loadAll()

DB
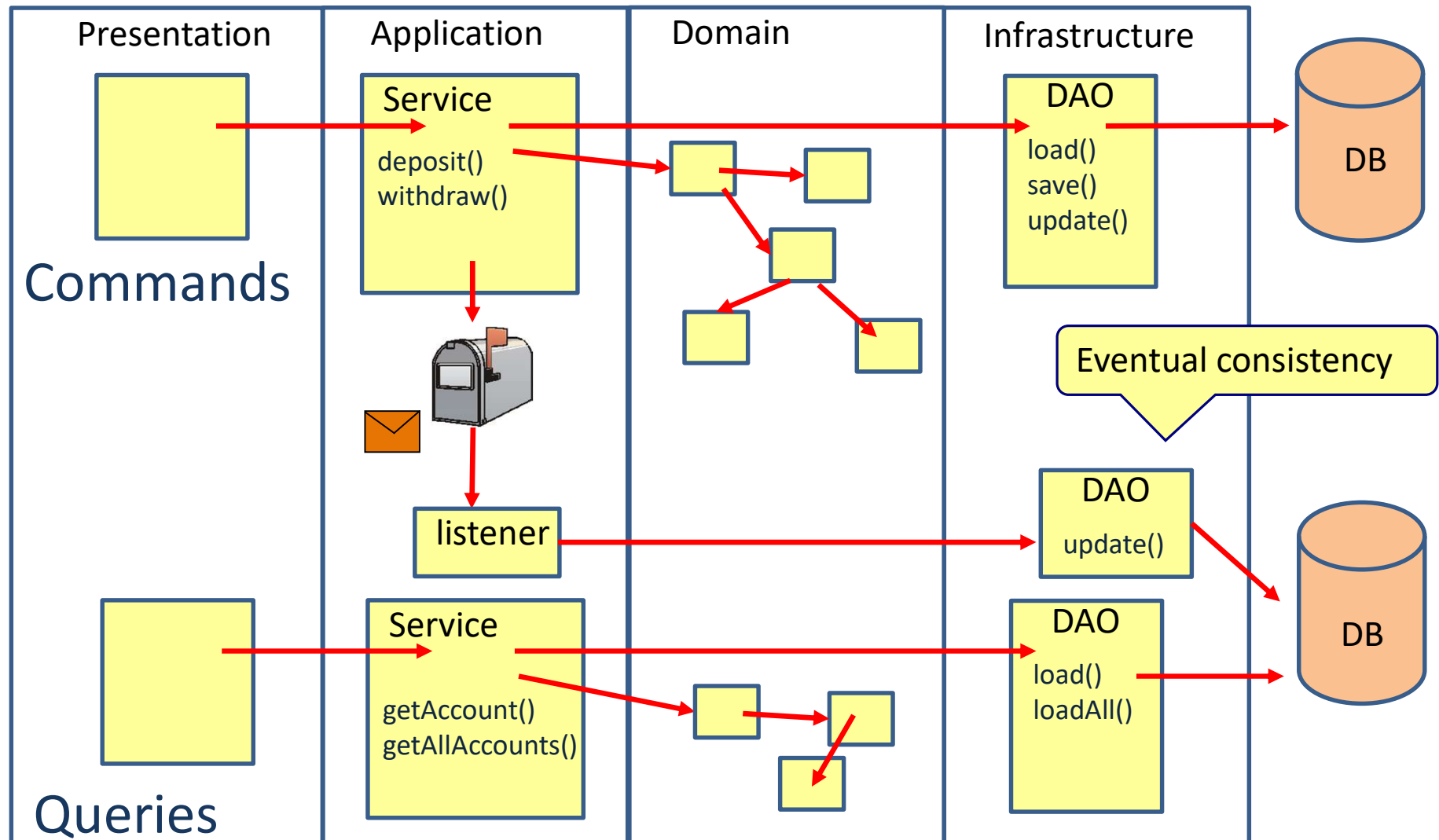
CQRS advantages

- 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

CreditcardCommands

Controller
createCreditcard
addTransaction

Service
createCreditcard
addTransaction

Creditcard → Transaction

Repository

NewCreditcardEvent

NewTransactionEvent

HSQLDB

CreditcardQueries

NewCreditcardEvent

NewTransactionEvent

Listener
NewCreditcard
NewTransaction

UpdateService
addCreditcard
addTransaction

Controller
getAllCreditcards

Service
getAllCreditcards

Creditcard → Transaction

Repository

Mongo

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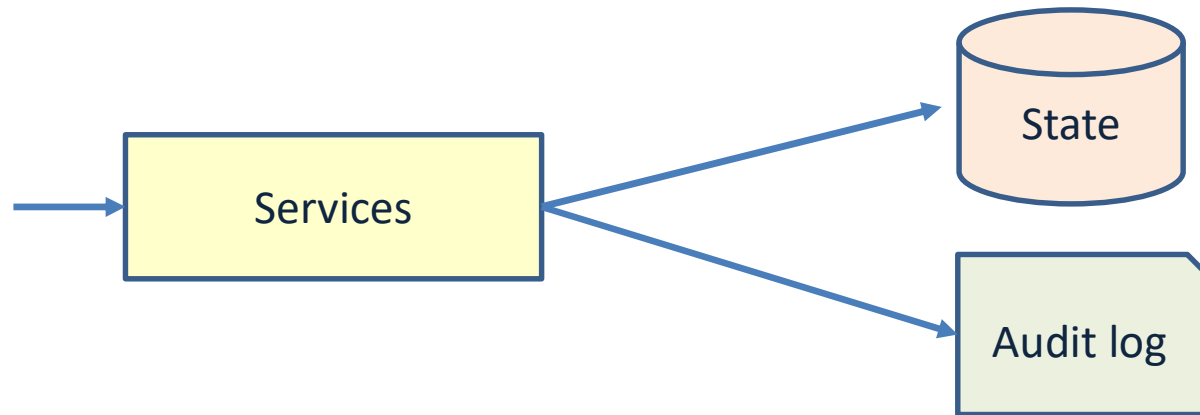
- Now we know how we got there
- What if the audit log gets out of sync with the state due to a bug?
- Which is the source of truth? (Audit log)

Event sourcing

- Only capture how we got there
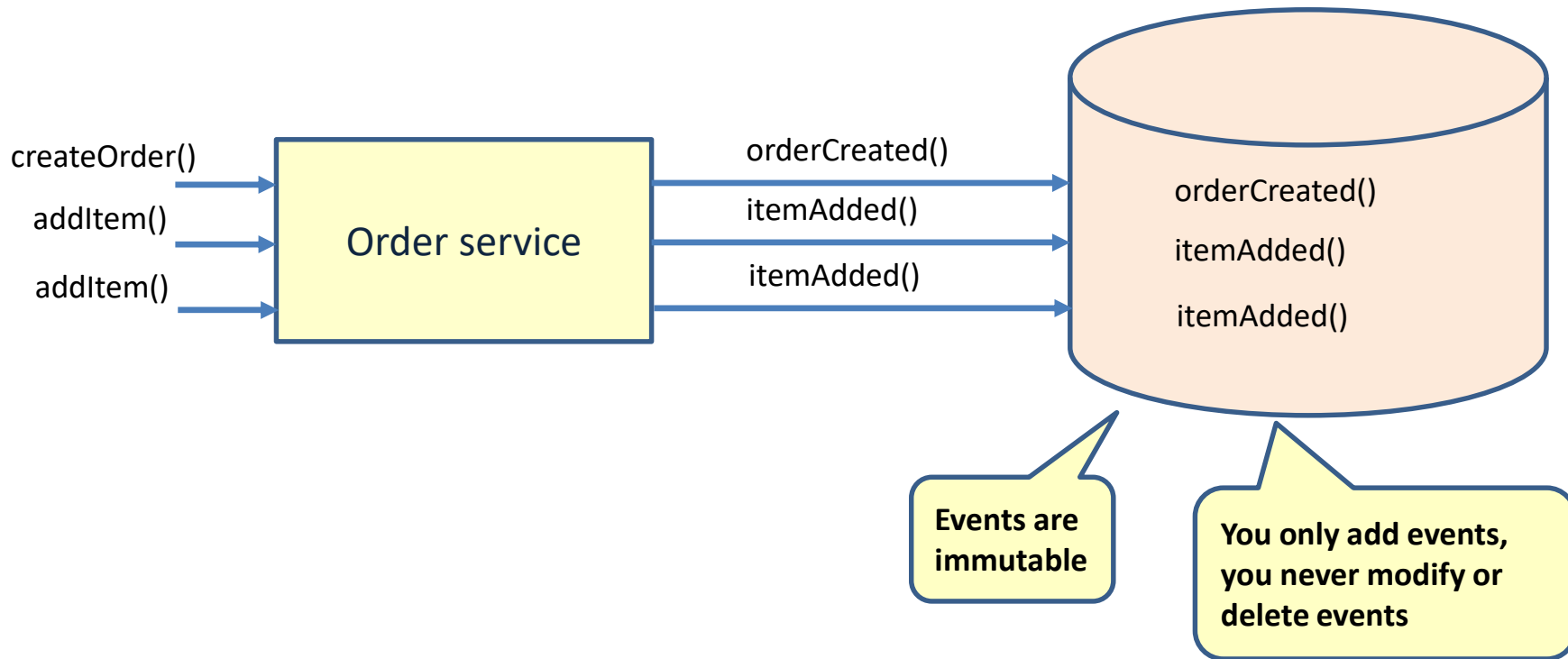
Event sourcing

- **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**
- **Event sourcing**

ID	status	data...
101	accepted	...

Store entity data

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?

getOrder() → **Order service**

Create order from events

Get all order events →

← List of events

order ←

Replay events

orderCreated()

itemAdded()

itemAdded()
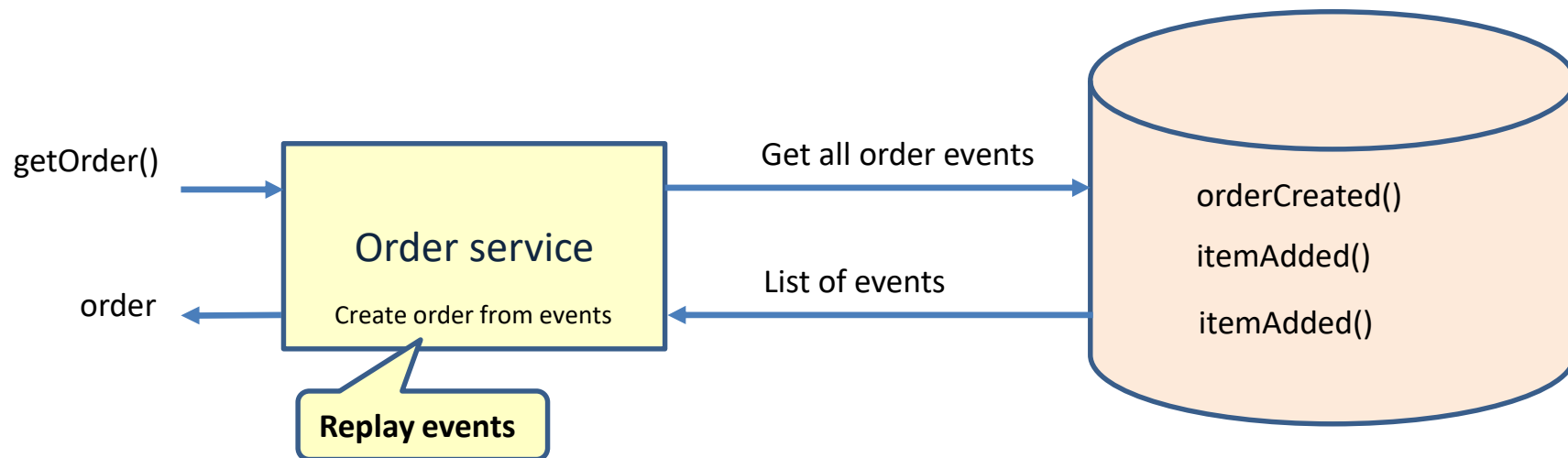
Snapshots
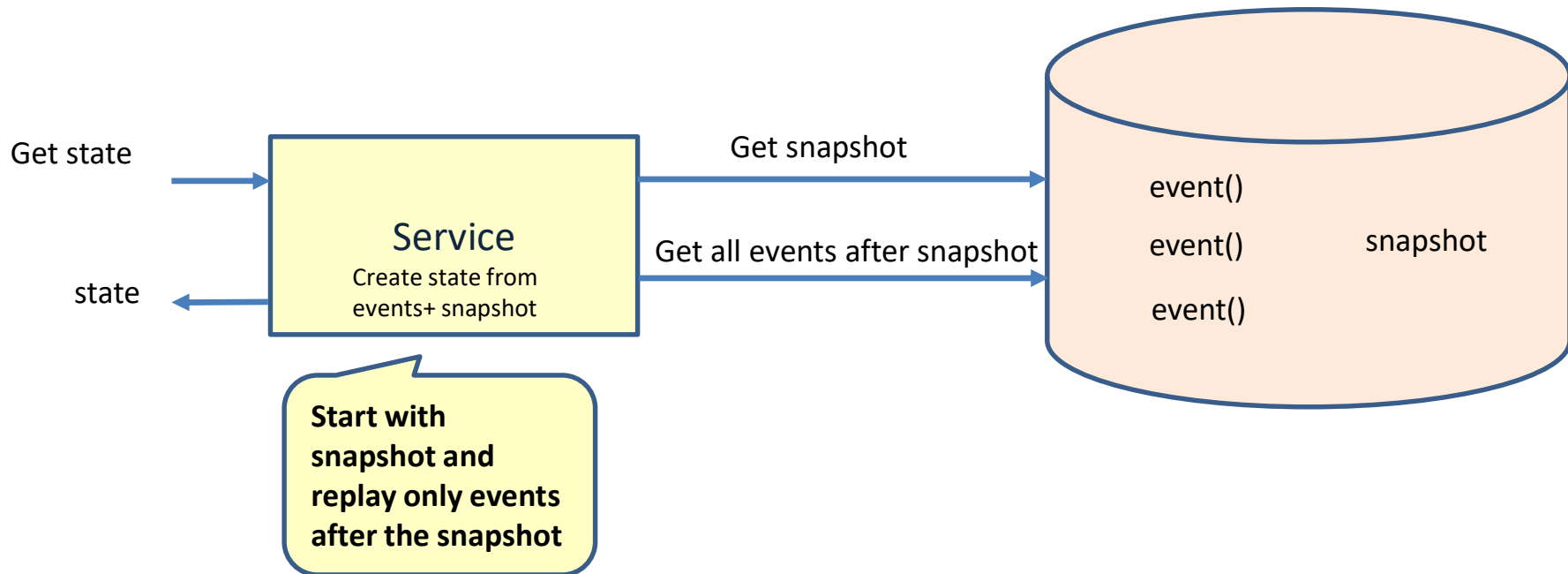
Event sourcing advantages

createOrder() → | Order service | → orderCreated()
addItem() → | | → itemAdded()
addItem() → | | → itemAdded()

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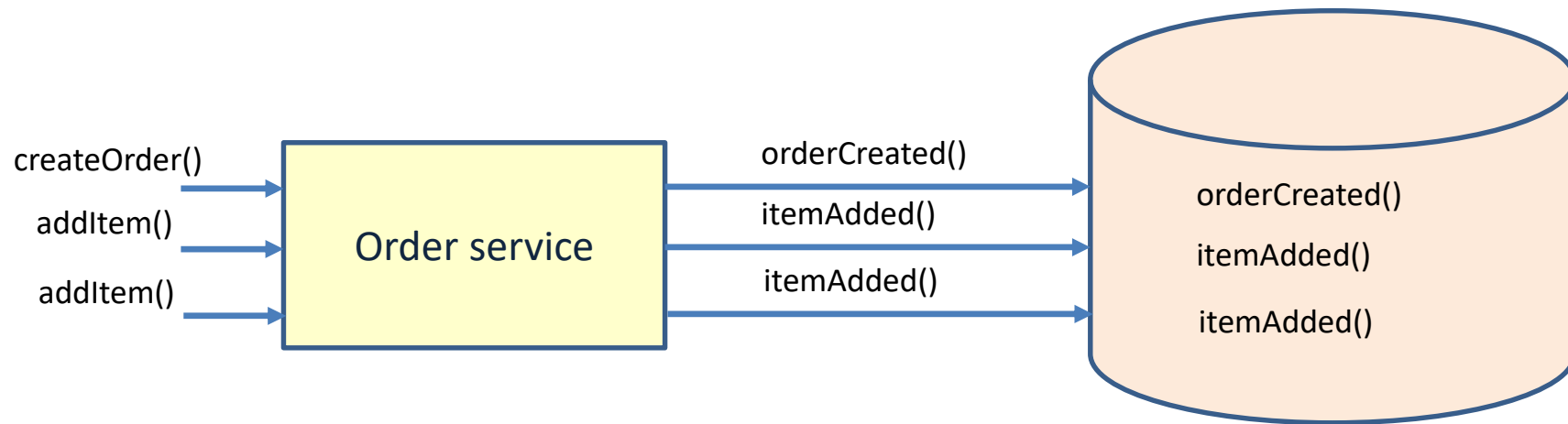
- You don't miss a thing
  - Business can analyze history of events
  - Bugs can be solved easier
- Creates a built-in audit log
- Allows for rewinding or undo changes
- Append is usually more efficient in databases
- No transactions needed

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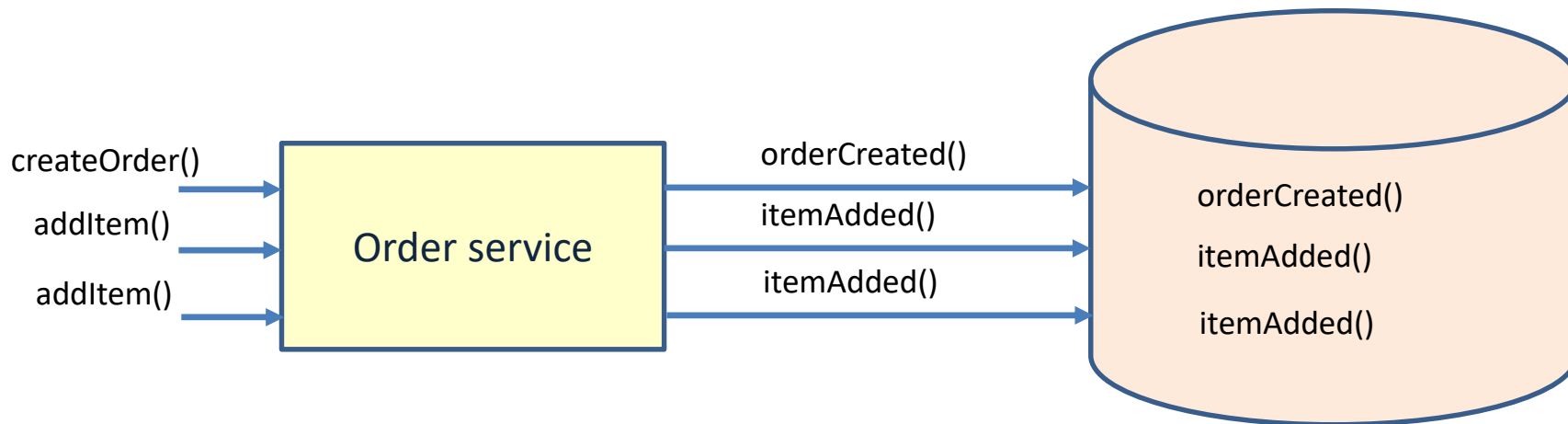
- You cannot easily see what your state is
  - You always have to replay the events first (lower read performance)
  - Always use CQRS when you apply event sourcing
- You need more storage
- What if events have to change?
  - We need event versioning
  - The logic needs to support all versions

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

ServiceA

Configuration for ServiceA

ServiceB

Configuration for ServiceB

ServiceC

Configuration for ServiceC

ServiceD

Configuration for ServiceD
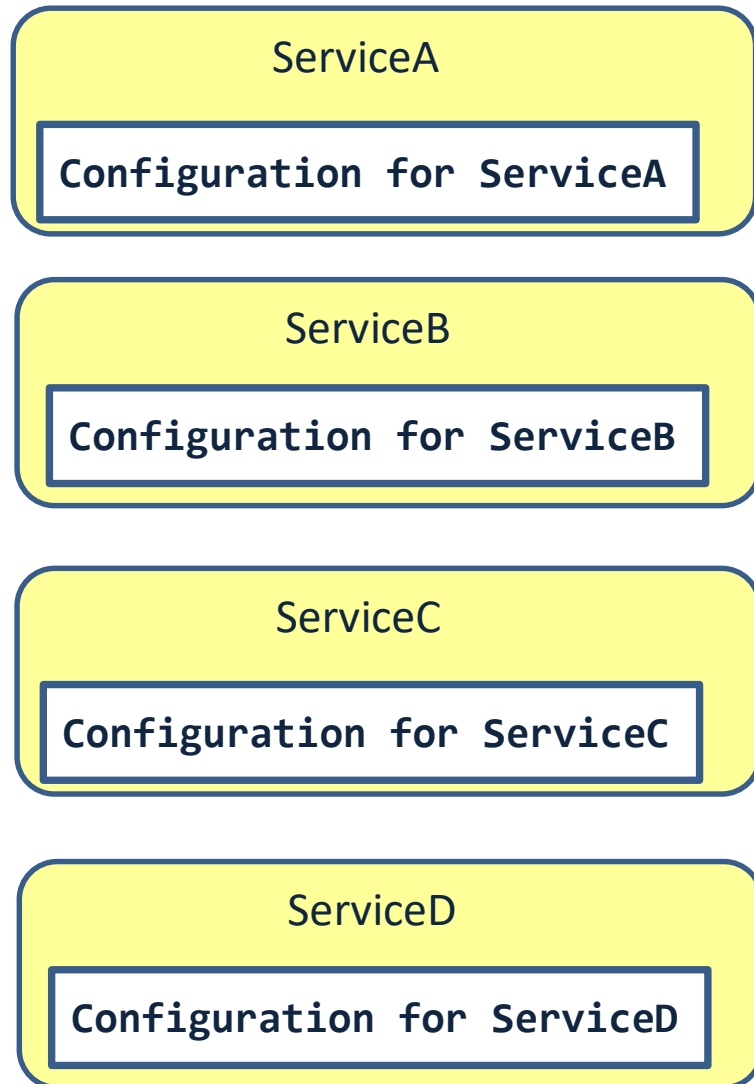
Local configuration challenges

- 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

ServiceA

`Configuration for ServiceA`

ServiceB

`Configuration for ServiceB`
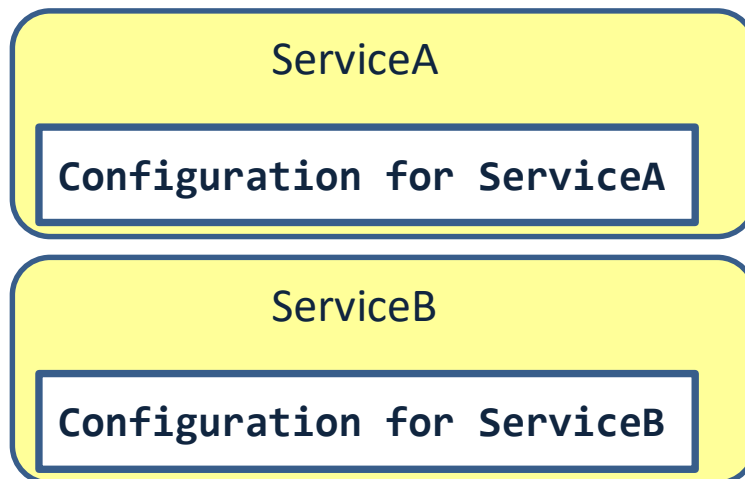
Spring cloud config server
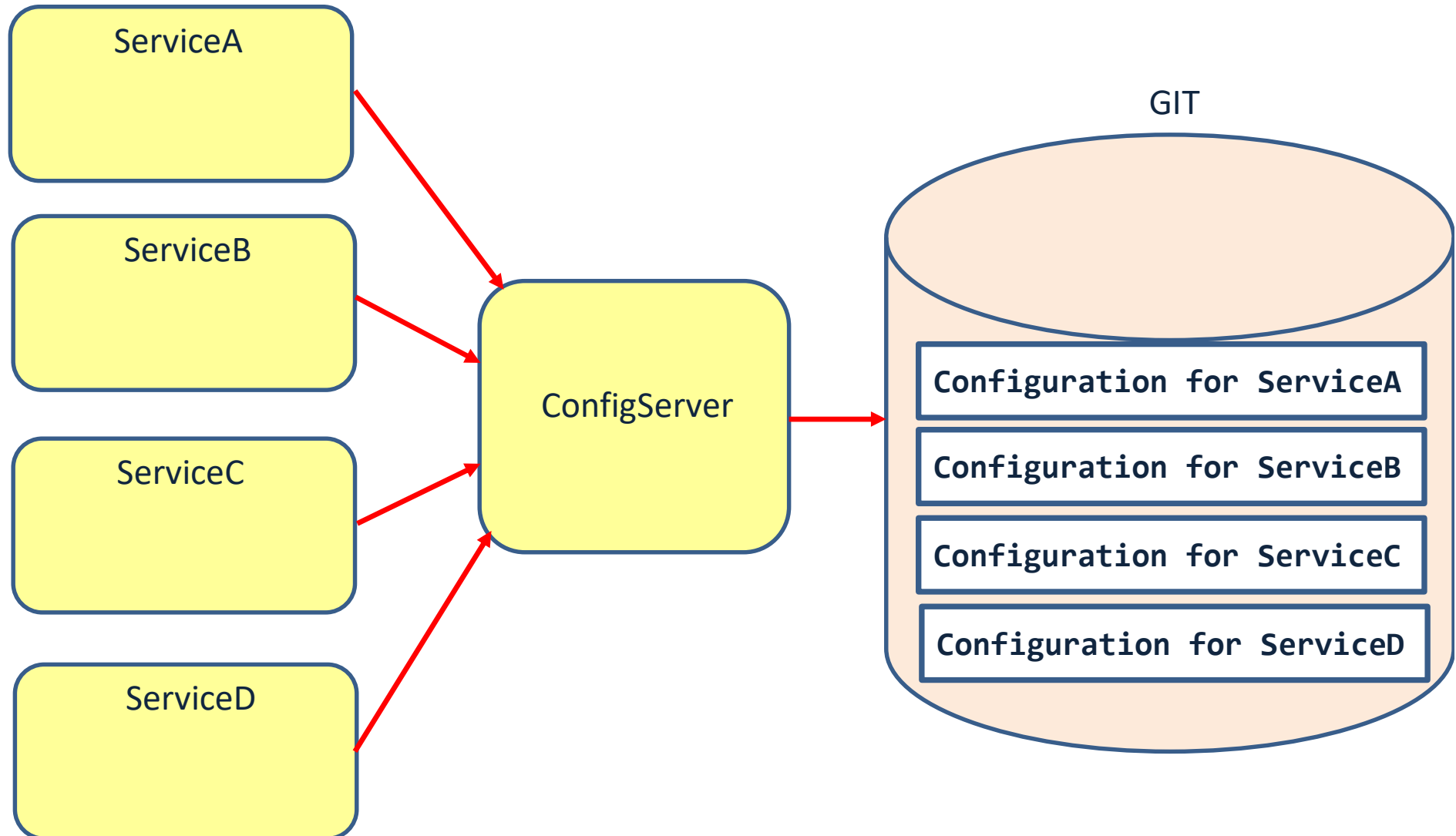
Spring cloud config

- HTTP access to centralized configuration

Centralized configuration

ServiceA → http → ConfigService

| Configuration for ServiceA |
| Configuration for ServiceB |

ServiceB →
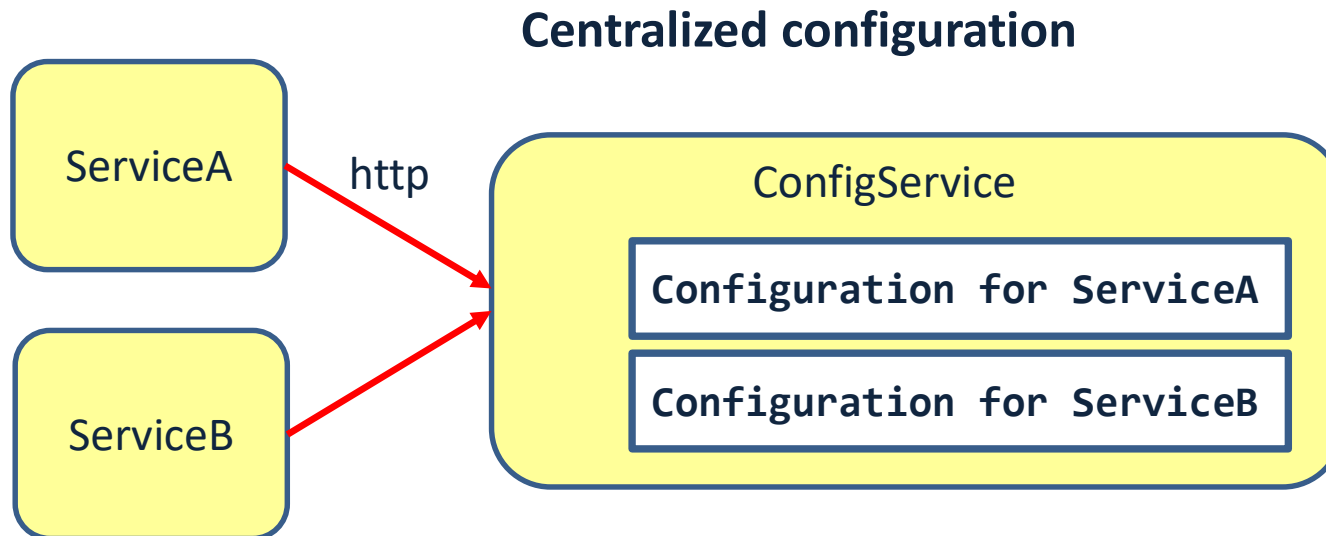
Spring cloud config

- ## File based configuration

```
ServiceA ──http──┐
                 ├──→  ConfigService
ServiceB ────────┘     ┌─────────────────────────────┐
                       │ Configuration for ServiceA  │
                       ├─────────────────────────────┤
                       │ Configuration for ServiceB  │
                       └─────────────────────────────┘
```

- ## Git based configuration

```
ServiceA ──http──┐
                 ├──→  ConfigService ──→  Git repository
ServiceB ────────┘                        ┌──────────────────┐
                                          │ Configuration    │
                                          │ for ServiceA     │
                                          ├──────────────────┤
                                          │ Configuration    │
                                          │ for ServiceB     │
                                          └──────────────────┘
```
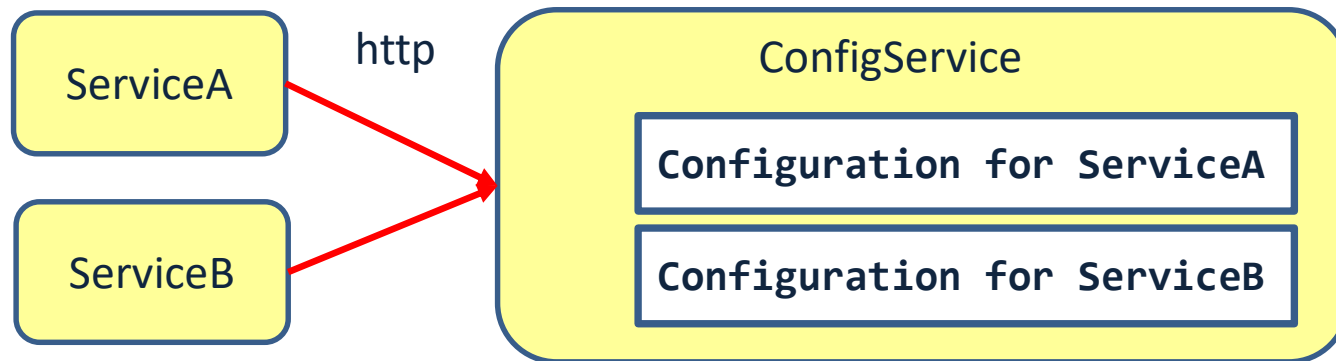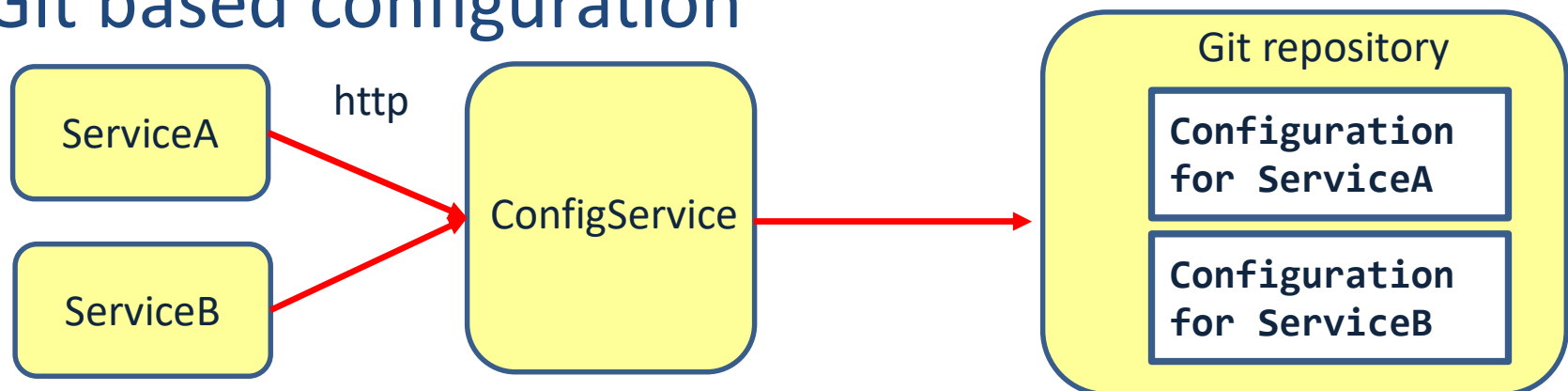
Spring cloud config example

```
ServiceA ──http──┐
                 ├──▶ ┌─────────────────────────────────────┐
ServiceB ────────┘    │  ConfigService                      │
                      │  ┌─────────────────────────────────┐│
                      │  │ Configuration for ServiceA      ││
                      │  └─────────────────────────────────┘│
                      │  ┌─────────────────────────────────┐│
                      │  │ Configuration for ServiceB      ││
                      │  └─────────────────────────────────┘│
                      └─────────────────────────────────────┘
```

pom.xml

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

pom.xml

```xml
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```
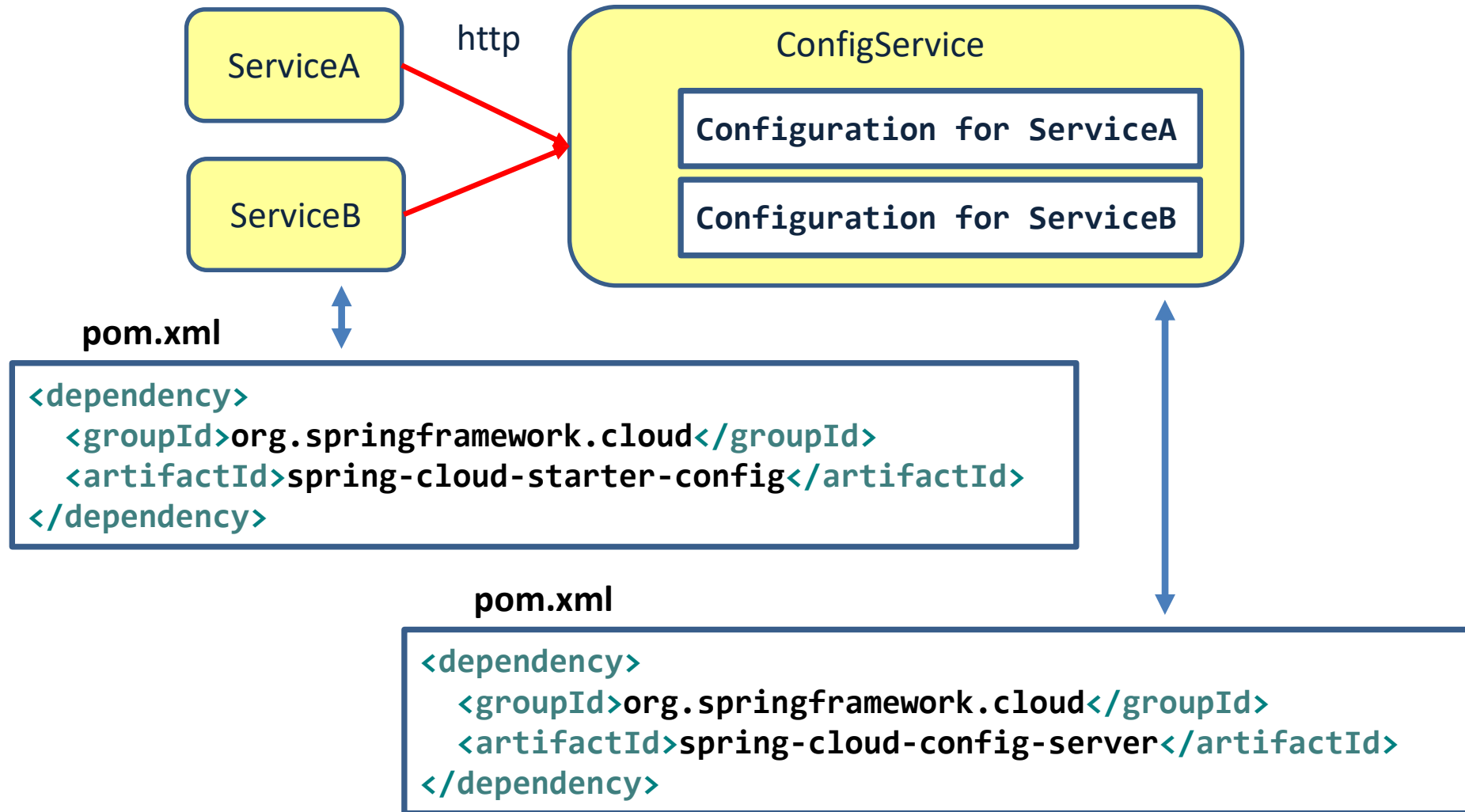
Configuration server

```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

```
spring.profiles.active=native
server.port=8888
```
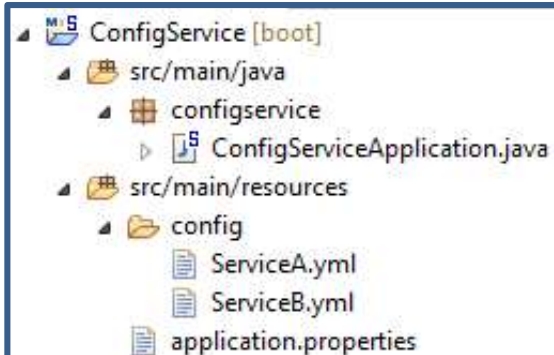
Do not use GIT, but local files

config/ServiceA.yml

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

config/ServiceB.yml

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

ConfigService [boot]
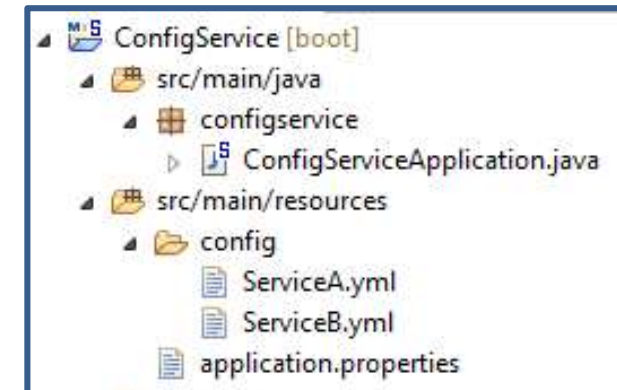- src/main/java
  - configservice
    - ConfigServiceApplication.java
- src/main/resources
  - config
    - ServiceA.yml
    - ServiceB.yml
  - application.properties

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
```

ServiceA [boot]
- src/main/java
  - config
    - ServiceAApplication.java
    - ServiceAController.java
- src/main/resources
  - application.yml
  - bootstrap.yml

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

```
application.yml      server:
                       port: 8090
```
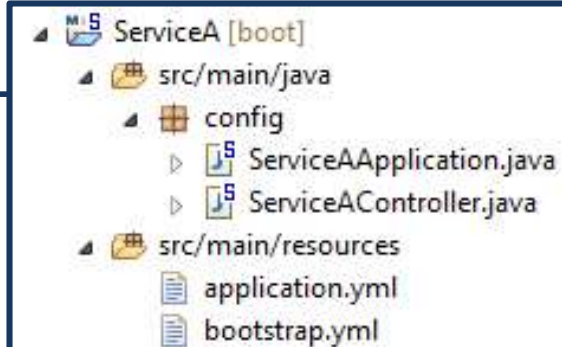
Configuration client: ServiceB

```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
```

ServiceB [boot]
  src/main/java
    config
      ServiceBApplication.java
      ServiceBController.java
  src/main/resources
    application.yml
    bootstrap.yml

Use of the Config Server

bootstrap.yml

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

localhost:8090

ServiceA

Hello from Service A

localhost:8888

ConfigService

localhost:8091

ServiceB

Hello from Service B

config/ServiceA.yml

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

config/ServiceB.yml

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

bootstrap.yml

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

Using Git (or GitHub)

application.yml

```yaml
server:
  port: 8888
spring:
 cloud:
    config:
     server:
      git:
       uri: https://github.com/renespring/springcloud.git
```

localhost:8090

localhost:809 ×
← → C ⓘ localhost:8090 ☆ ⋮
Hello from Service A

ServiceA

localhost:8091

localhost:809 ×
← → C ⓘ localhost:8091 ☆ ⋮
Hello from Service B

ServiceB

localhost:8888

ConfigService

GIT

Version control

Security

ServiceA.yml

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

ServiceB.yml

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

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;
  }
}
```
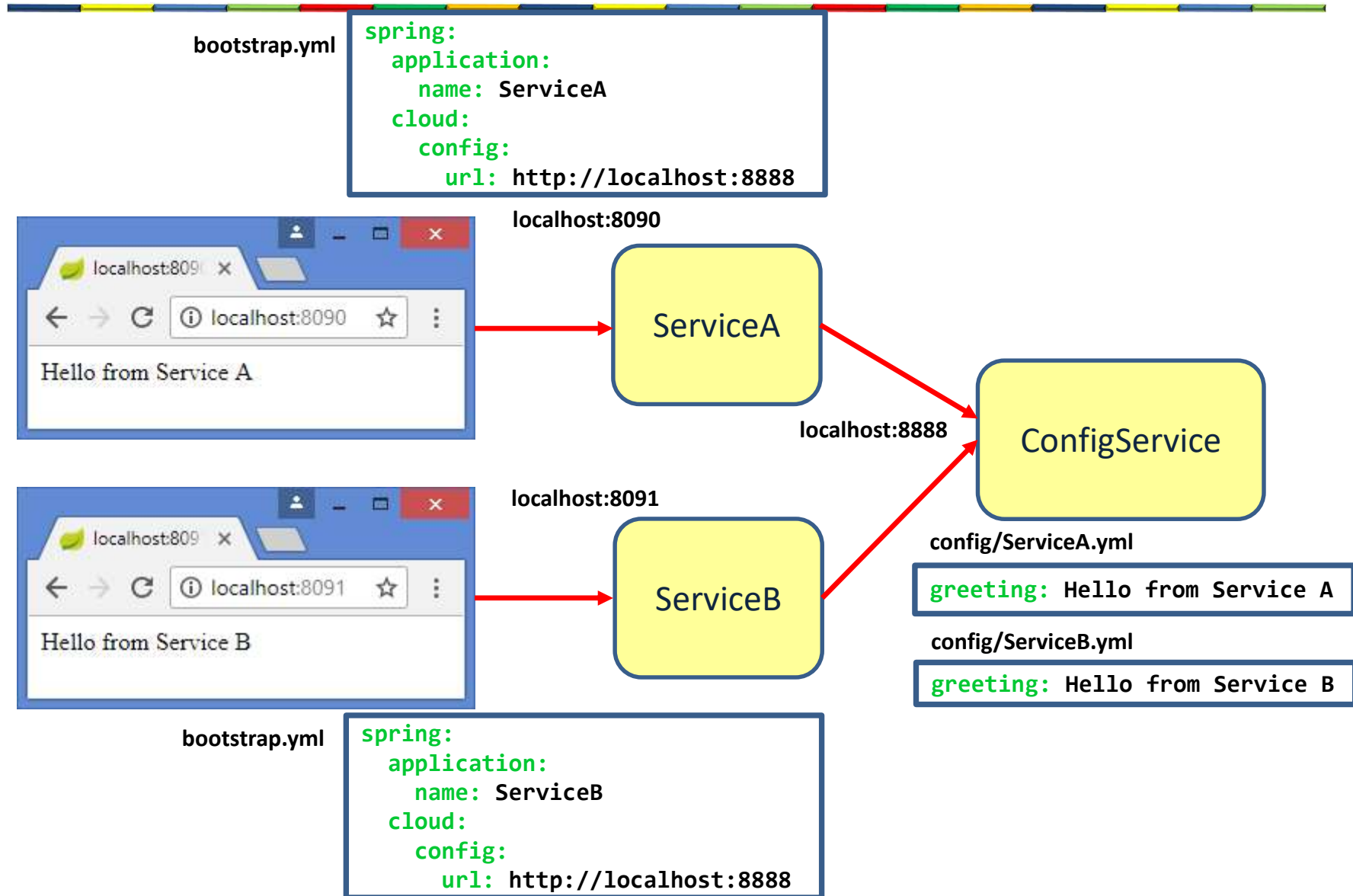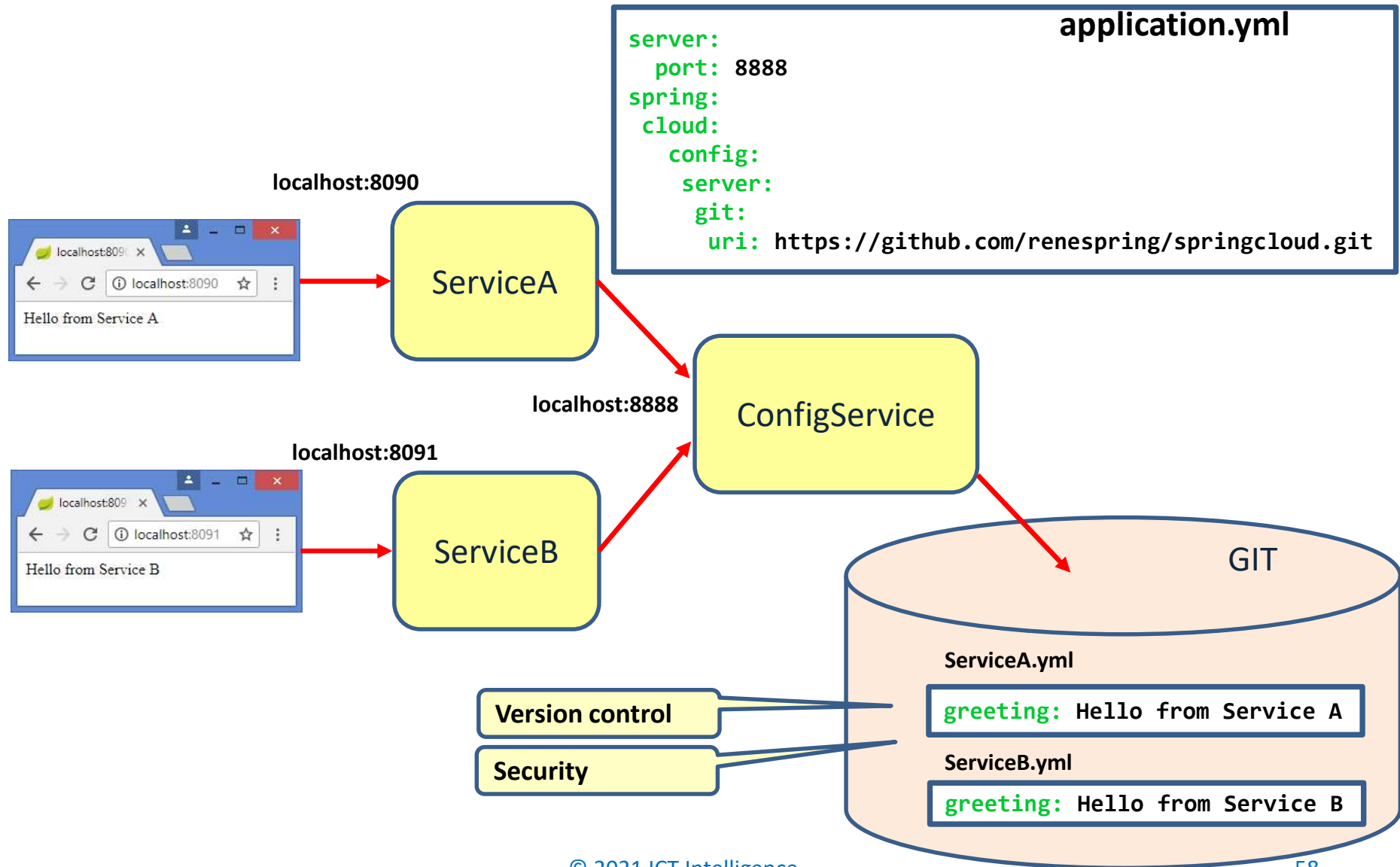
```
server:
 port: 8888

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




management:
 endpoints:
  web:
   exposure:
    include: refresh
```
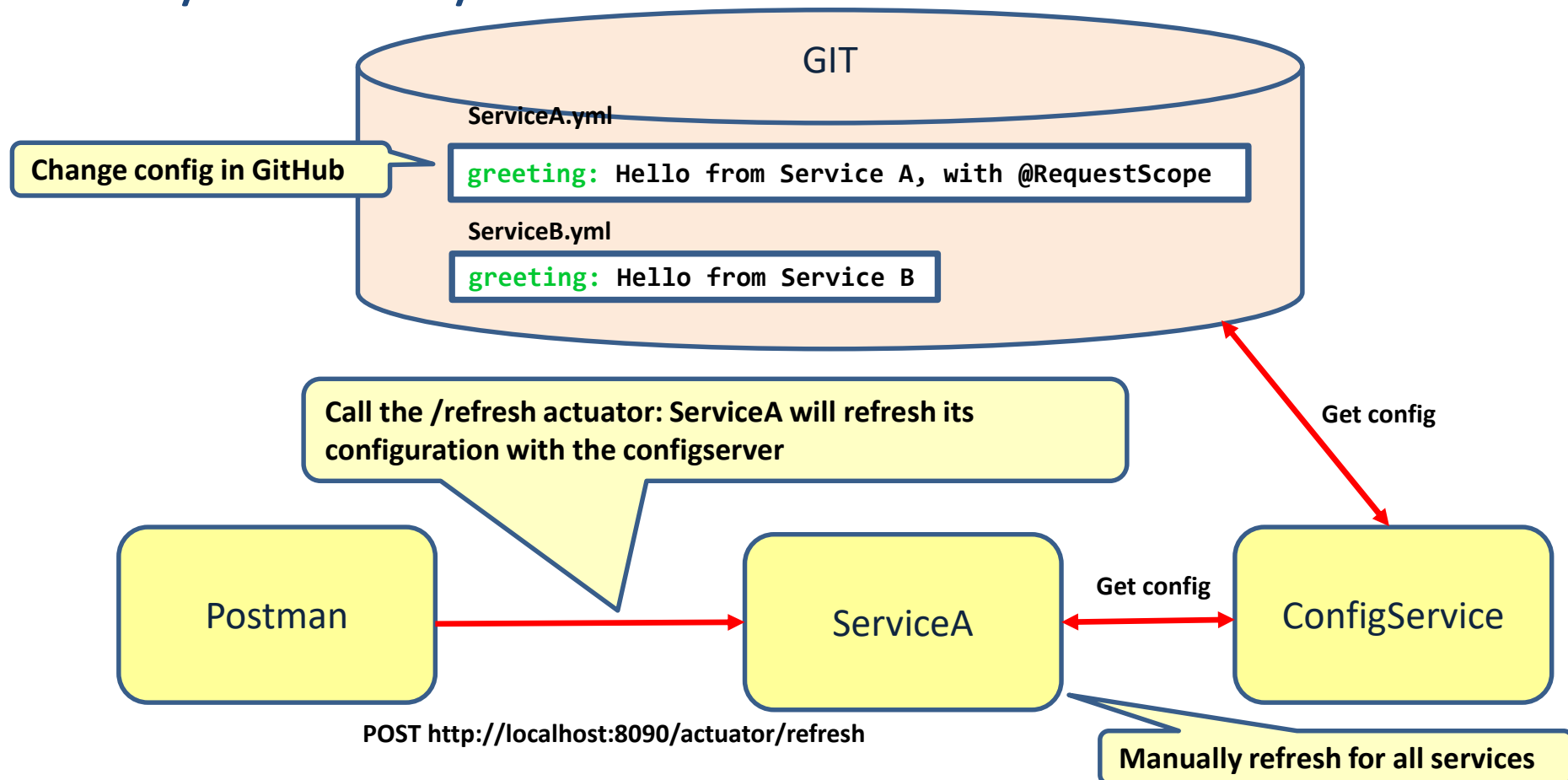
Expose the /refresh actuator

Refreshing configuration

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

GIT

Change config in GitHub

ServiceA.yml

```
greeting: Hello from Service A, with @RequestScope
```

ServiceB.yml

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

Call the /refresh actuator: ServiceA will refresh its configuration with the configserver

Get config

Get config

Postman → ServiceA ↔ ConfigService

POST http://localhost:8090/actuator/refresh

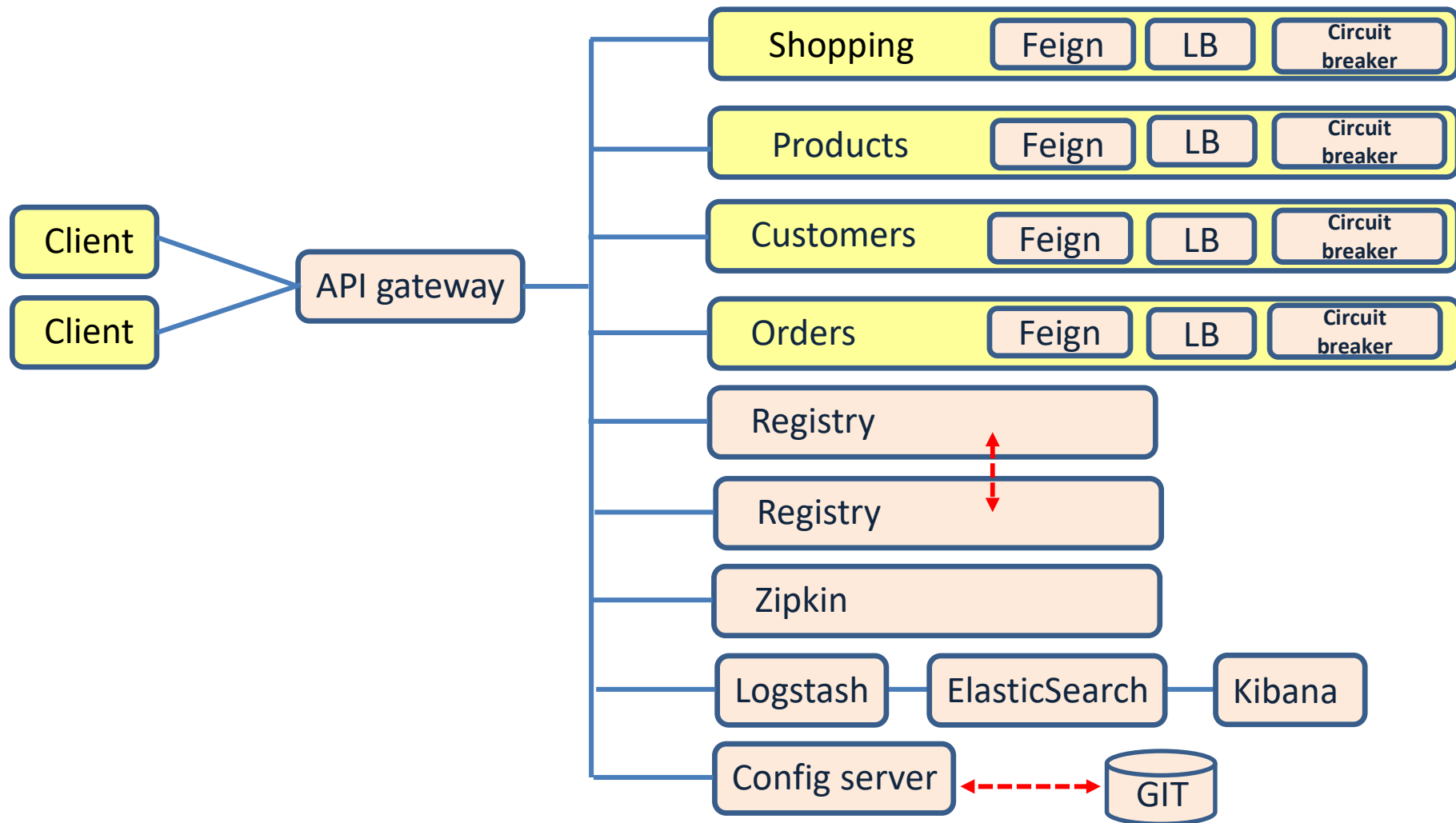**Manually refresh for all services**

Refreshing configuration

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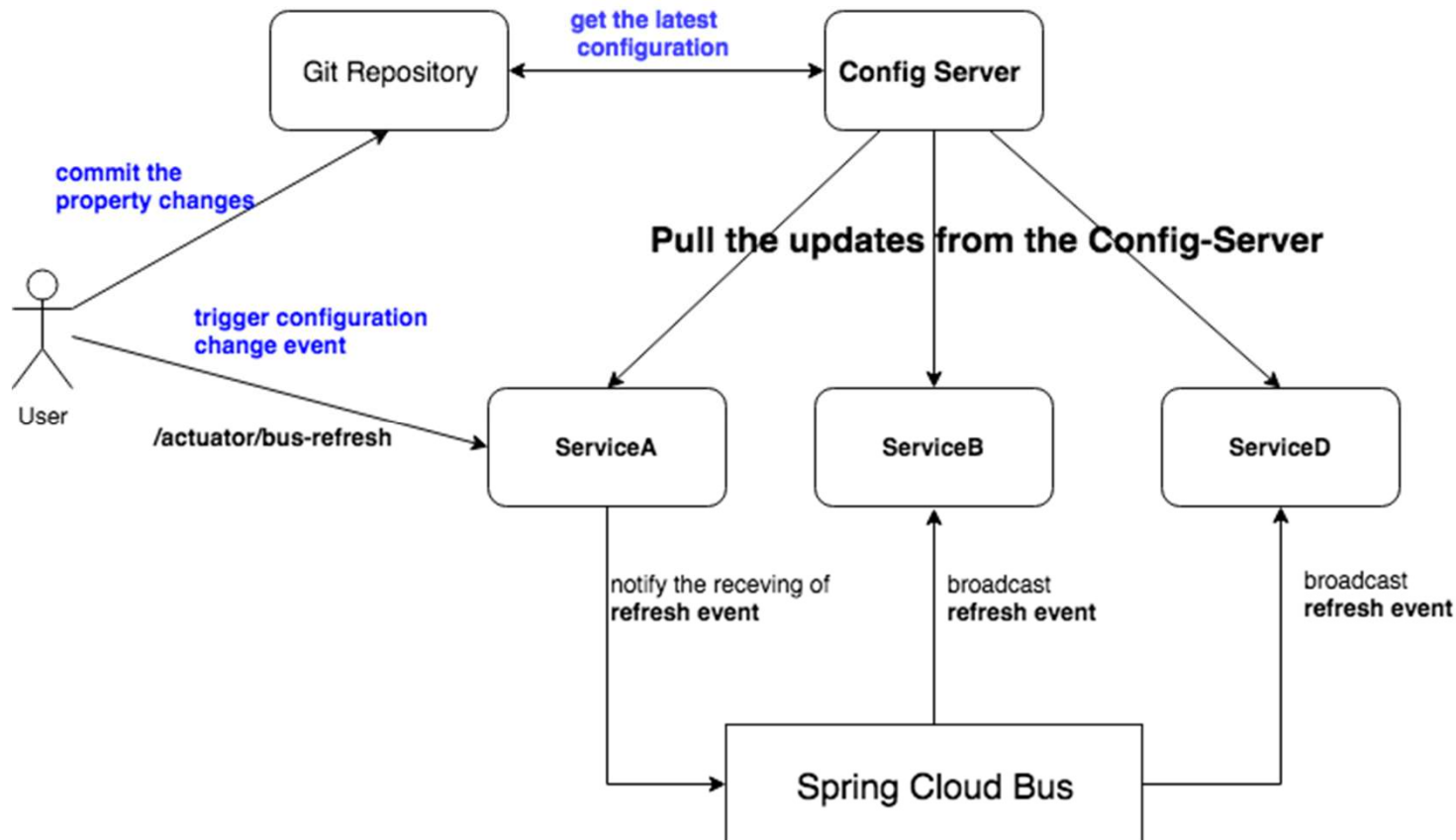
Implementing microservices

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