Lesson 9

MICROSERVICES

© 2021 ICT Intelligence

SERVICE DEPLOYMENT

© 2021 ICT Intelligence

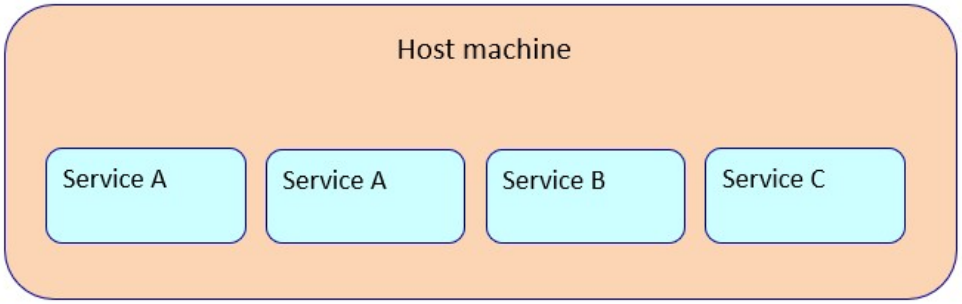
Service deployment

- 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

© 2021 ICT Intelligence

Multiple service instances per host

- Benefits
  - Efficient resource utilization
  - Fast deployment

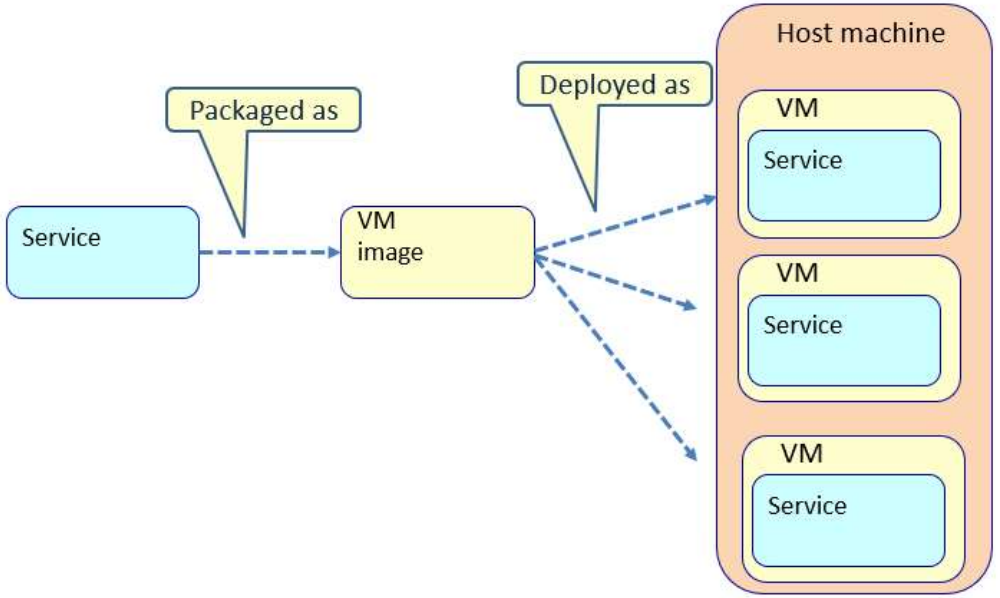


- Drawbacks
  - Poor isolation
  - Poor visibility of resource utilization
  - Difficult to constrain resource utilization
  - Risk of dependency version conflicts
  - Poor encapsulation of implementation technology

© 2021 ICT Intelligence

Service per VM

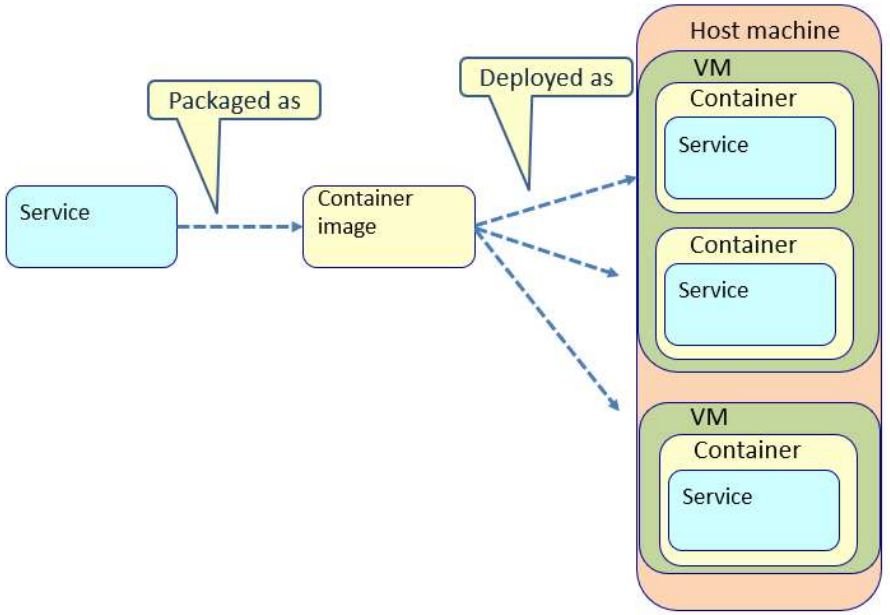
- Benefits
  - Great isolation
  - Great manageability
  - VM encapsulates implementation technology
  - Leverage cloud infrastructure for auto scaling/load balancing
- Drawbacks
  - Less efficient resource utilization
  - Slow deployment



© 2021 ICT Intelligence

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

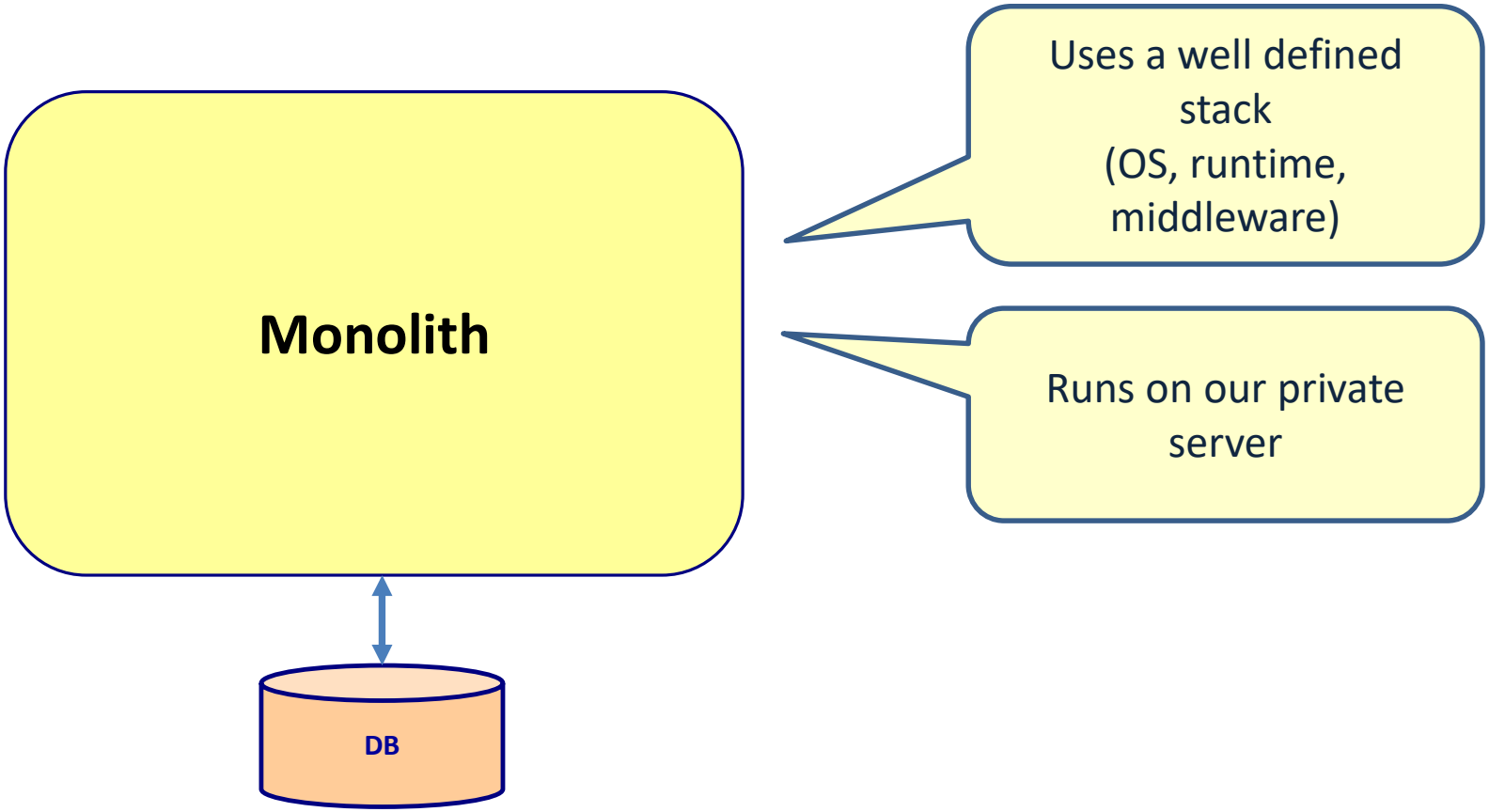


© 2021 ICT Intelligence

CONTAINERS

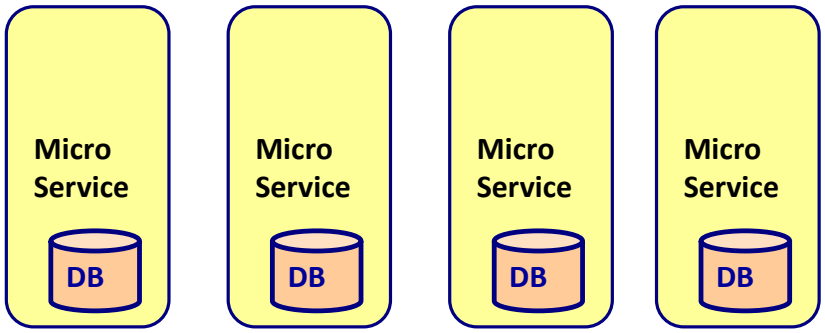
© 2021 ICT Intelligence

Monolith



© 2021 ICT Intelligence

Microservice



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

Runs anywhere (private, cloud)

© 2021 ICT Intelligence

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

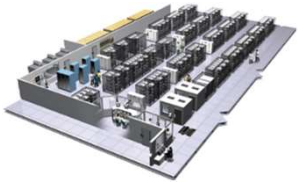
Ruby + Rails + sass + Unicorn

API endpoint

Development VM

QA server

Public Cloud



Production Cluster

Disaster recovery

Customer Data Center

Production Servers

Contributor's laptop

© 2021 ICT Intelligence

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	?	?	?	?	?	?	?

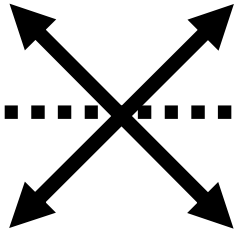
© 2021 ICT Intelligence

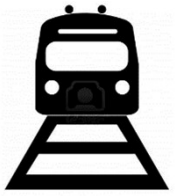
Cargo Transport Pre-1960

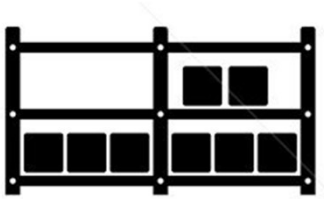








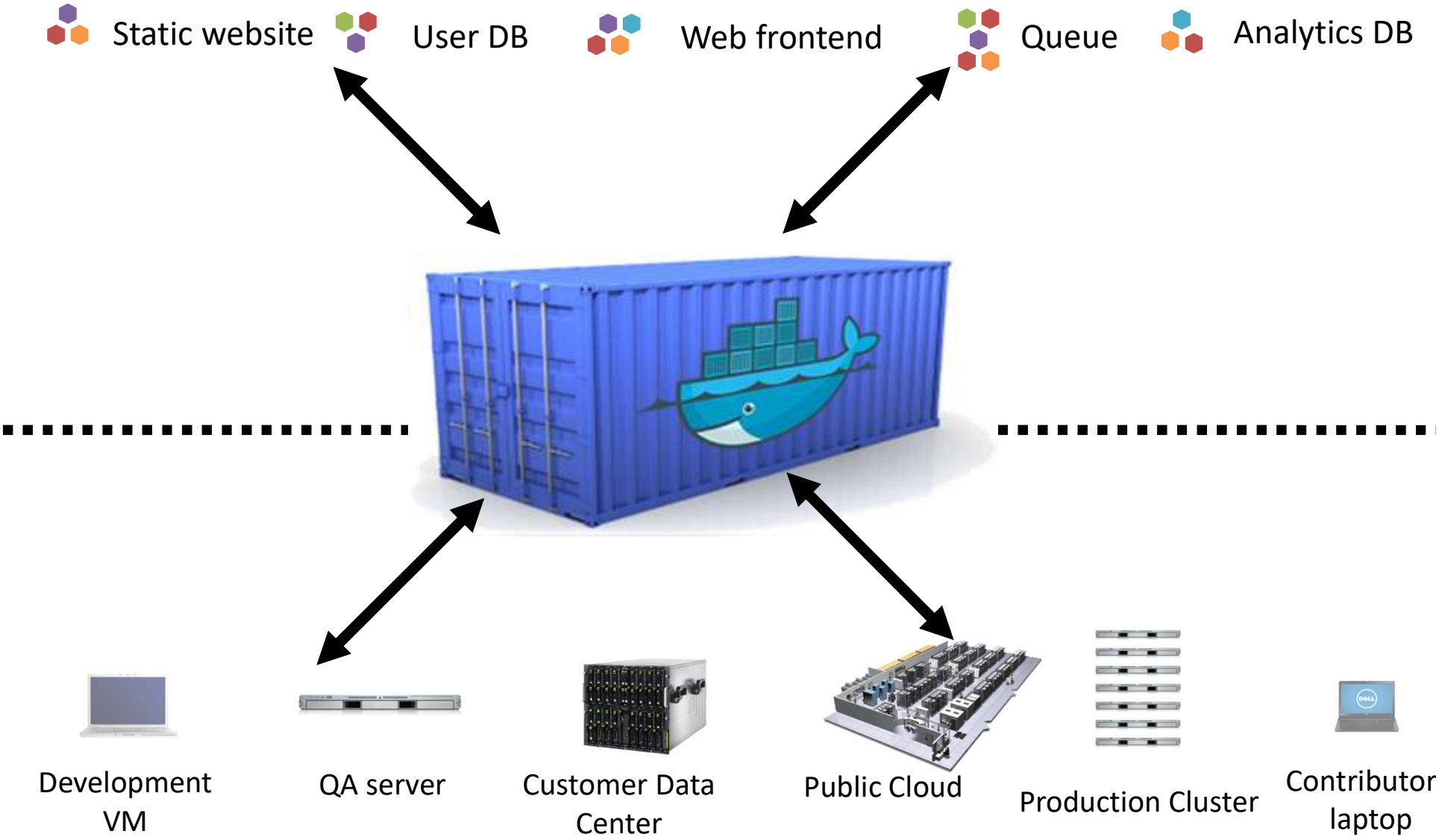




© 2021 ICT Intelligence

The solution

© 2021 ICT Intelligence

Docker is a shipping container for code



© 2021 ICT Intelligence

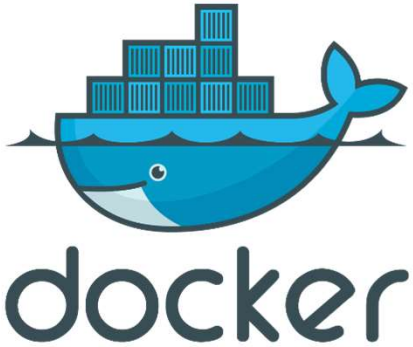
Separation of concern

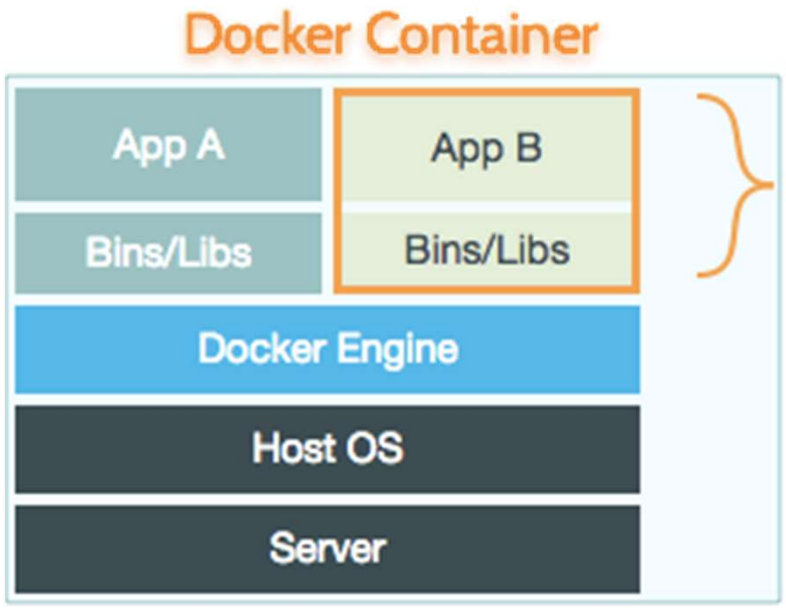
- Dan the Developer
  - Worries about what's "inside" the container
    - His code
    - His Libraries
    - His Package Manager
    - His Apps
    - His Data
- Oscar the Ops Guy
  - Worries about what's "outside" the container
    - Logging
    - Remote access
    - Monitoring
    - Network config
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

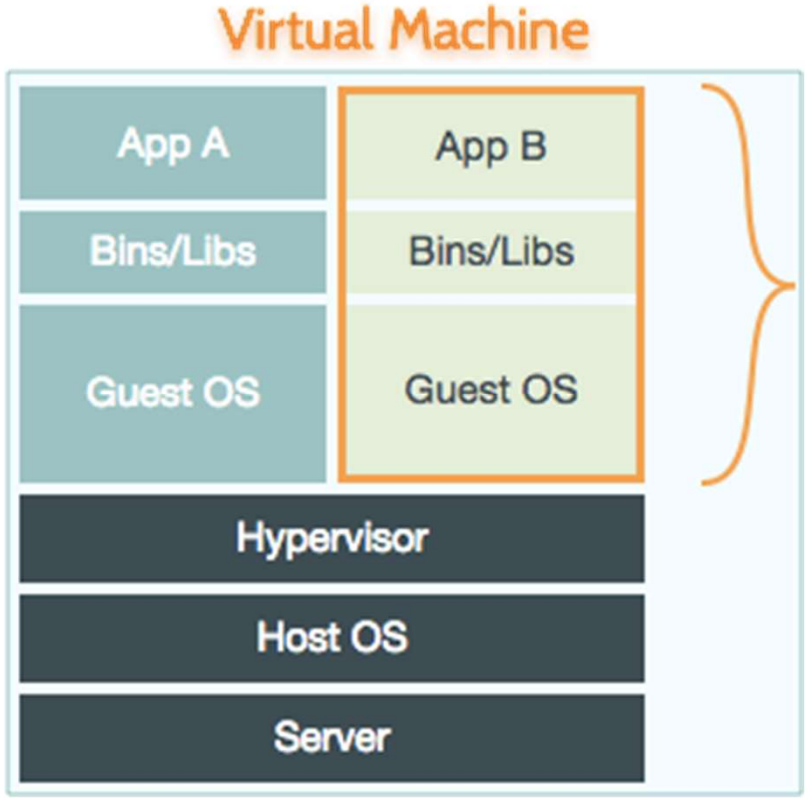
© 2021 ICT Intelligence

Containers

- Docker

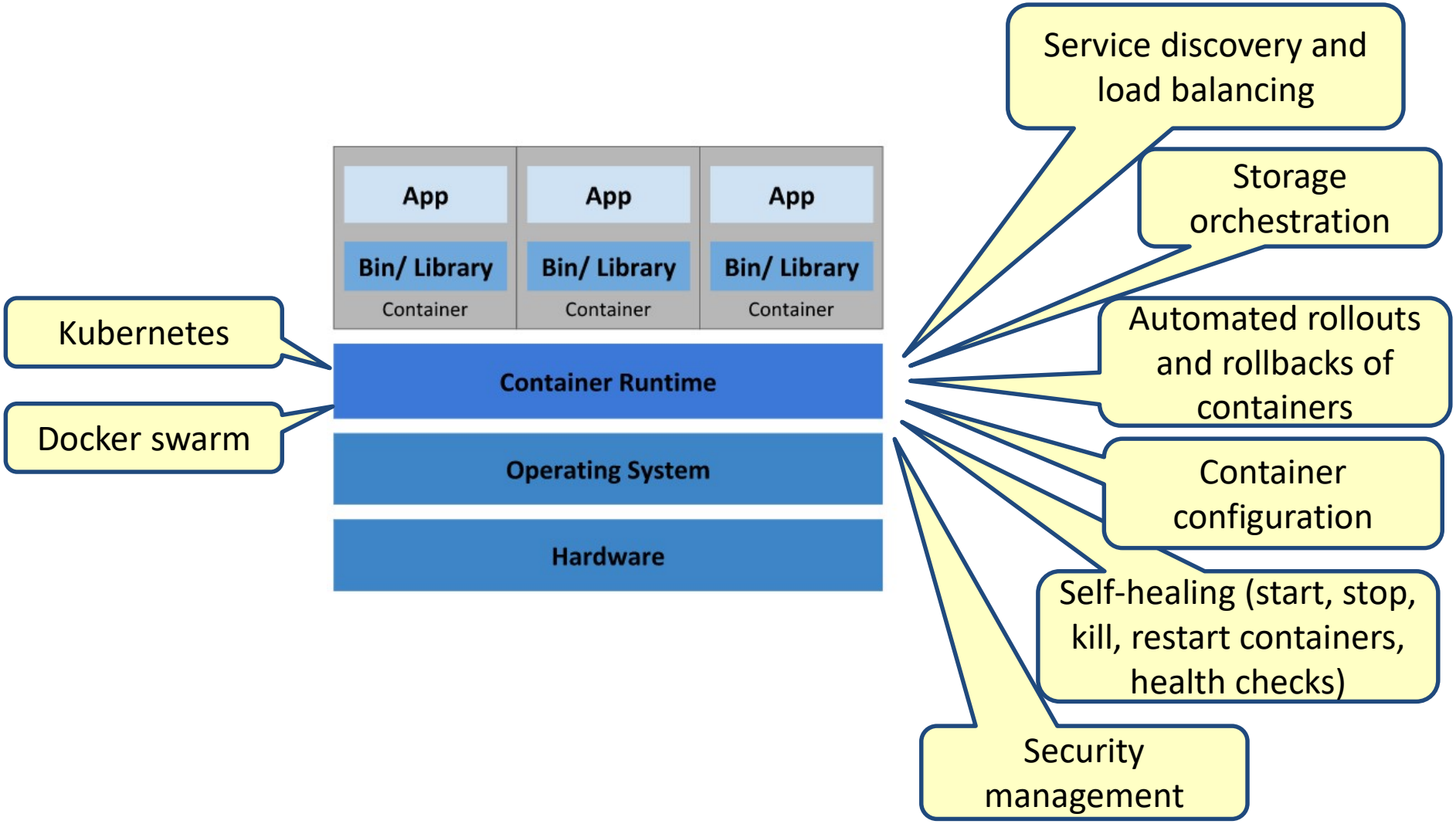






© 2021 ICT Intelligence

Container management



© 2021 ICT Intelligence

CQRS

© 2021 ICT Intelligence

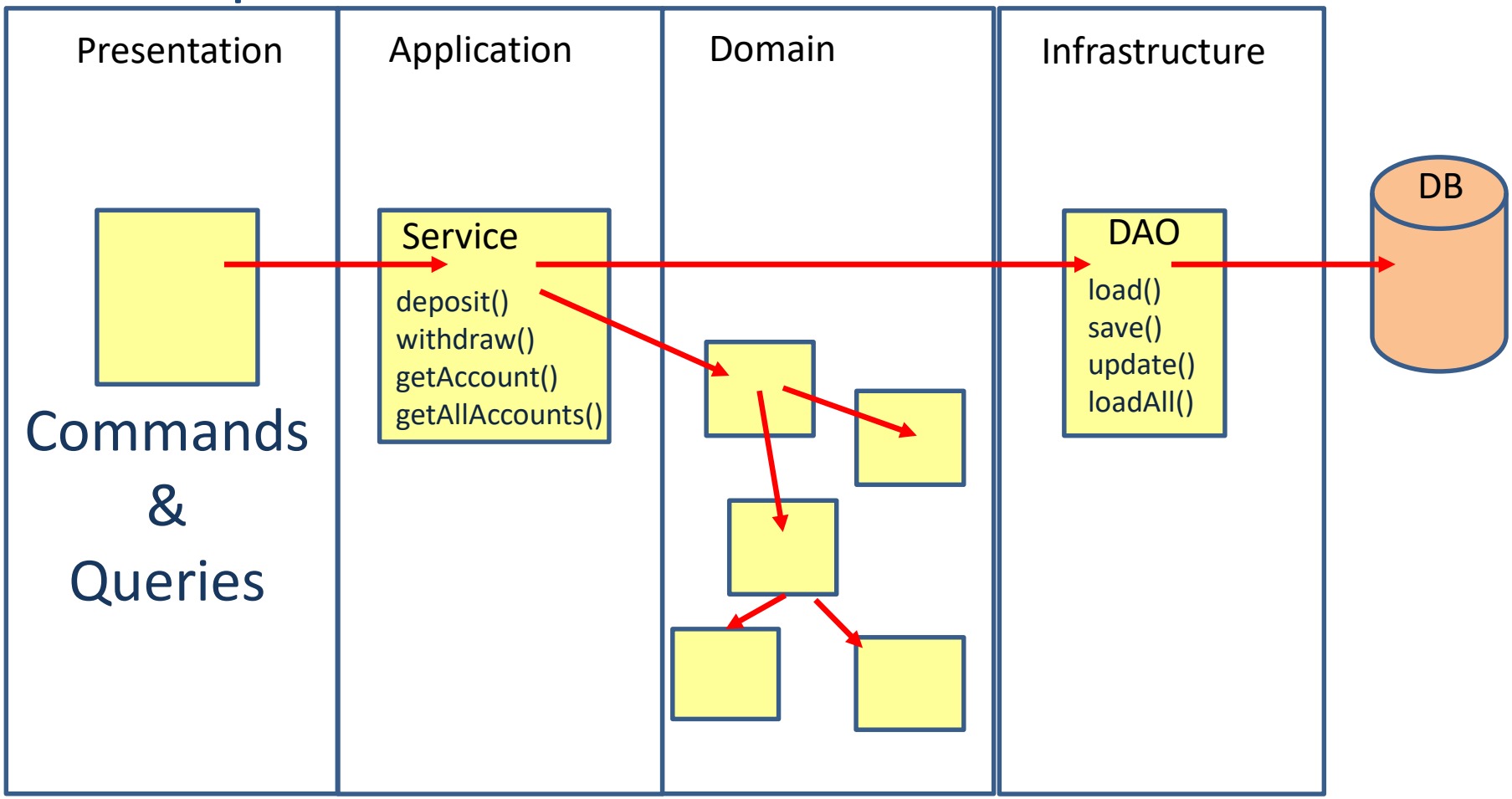
Command Query Responsibility Segregation (CQRS)

- 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)

© 2021 ICT Intelligence

Typical architecture

- One domain model that is used for commands and queries



© 2021 ICT Intelligence

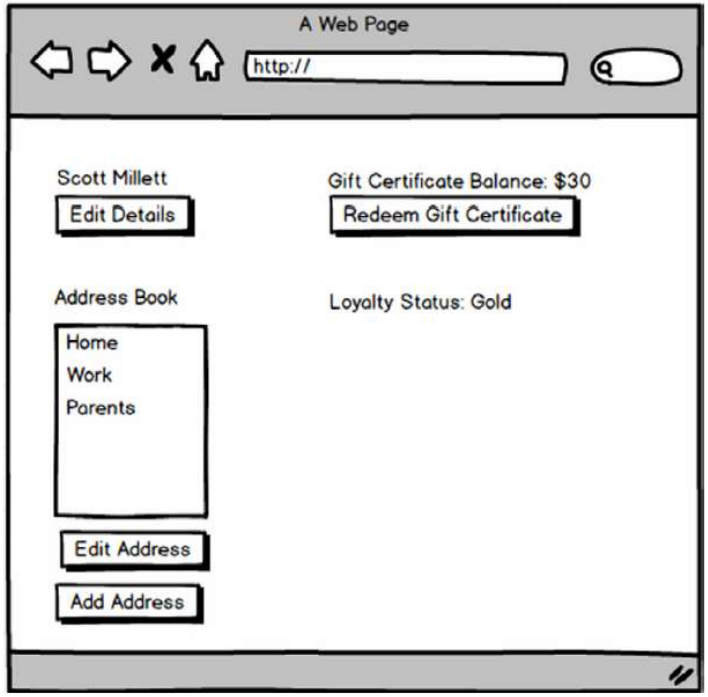
One model for both commands and queries

- 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

© 2021 ICT Intelligence

Example of complex aggregates



Complex aggregate because of UI needs

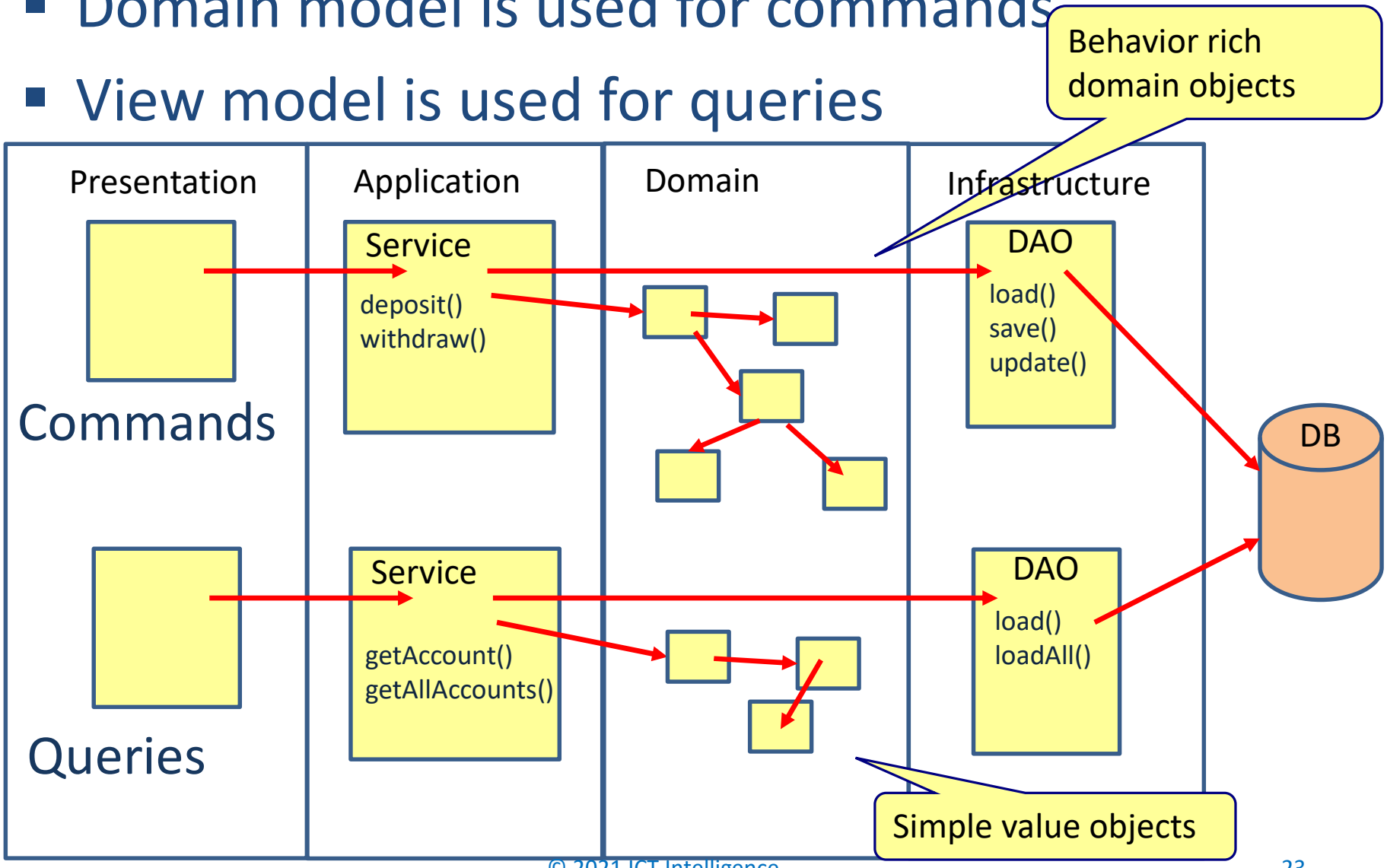
We don't need this complexity for commands

```
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; }
}
```

© 2021 ICT Intelligence

CQRS

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



© 2021 ICT Intelligence

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()

© 2021 ICT Intelligence

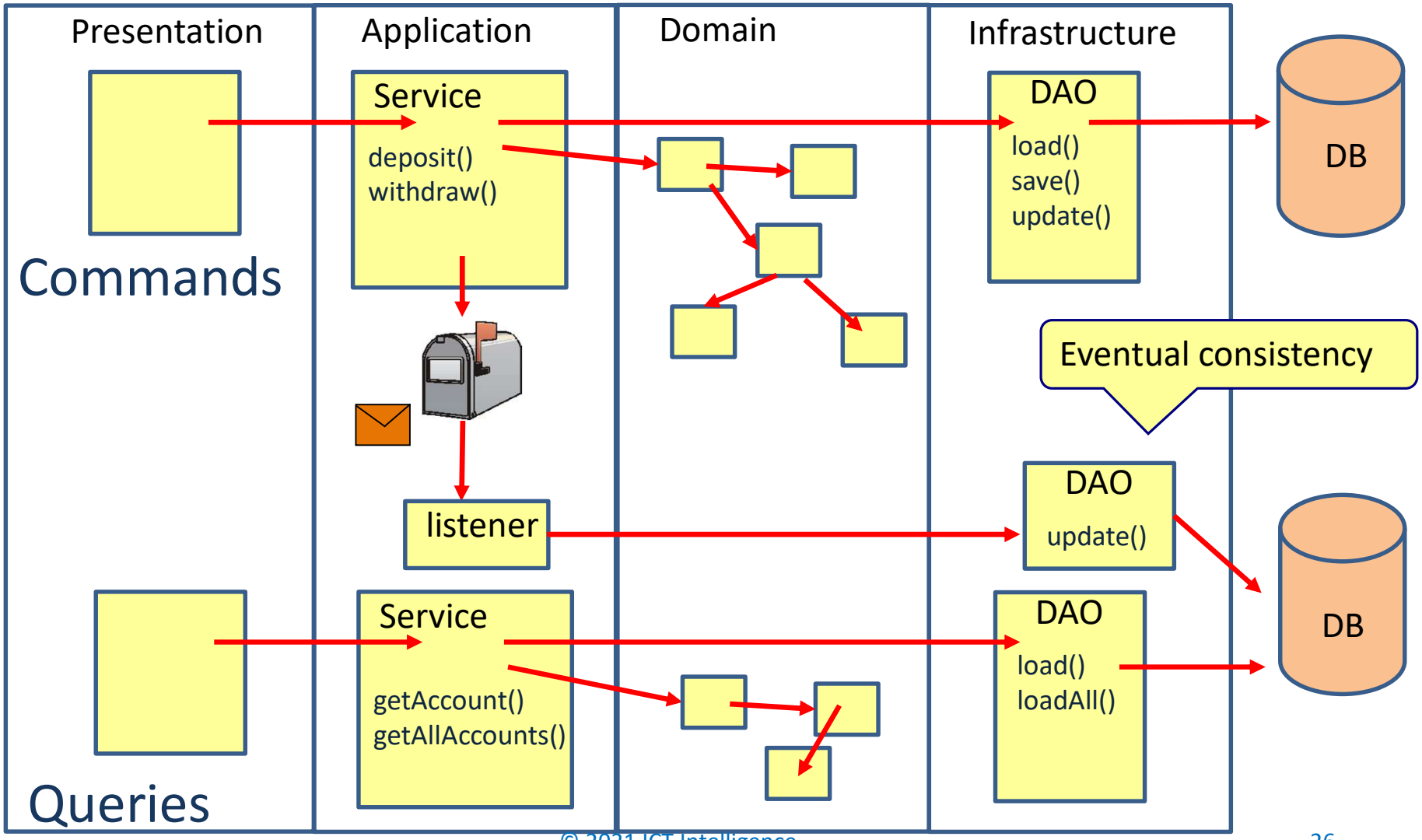
Architectural properties

- 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

© 2021 ICT Intelligence

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

© 2021 ICT Intelligence

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

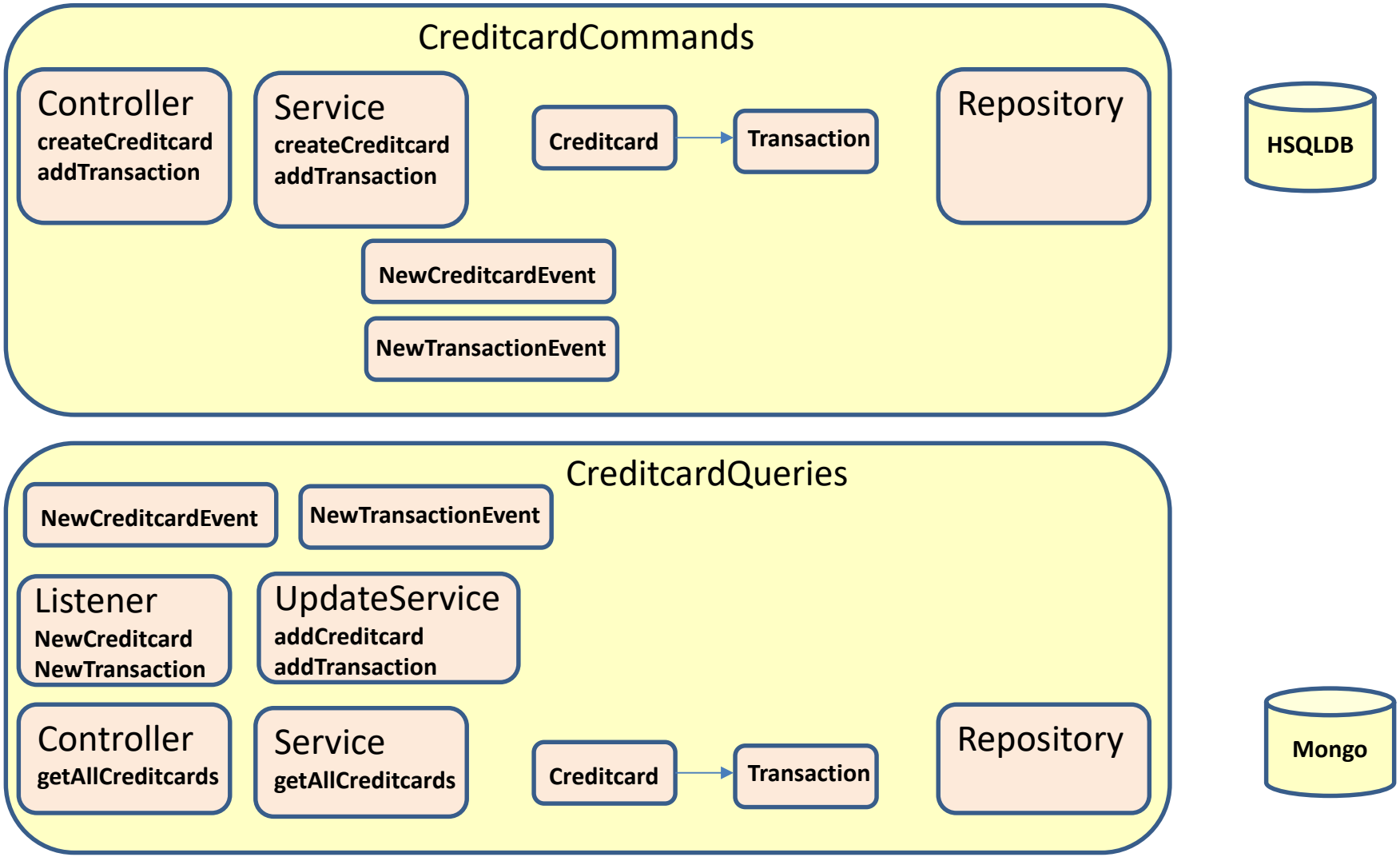
© 2021 ICT Intelligence

When not to use CQRS

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

© 2021 ICT Intelligence

CQRS example



© 2021 ICT Intelligence

EVENT SOURCING

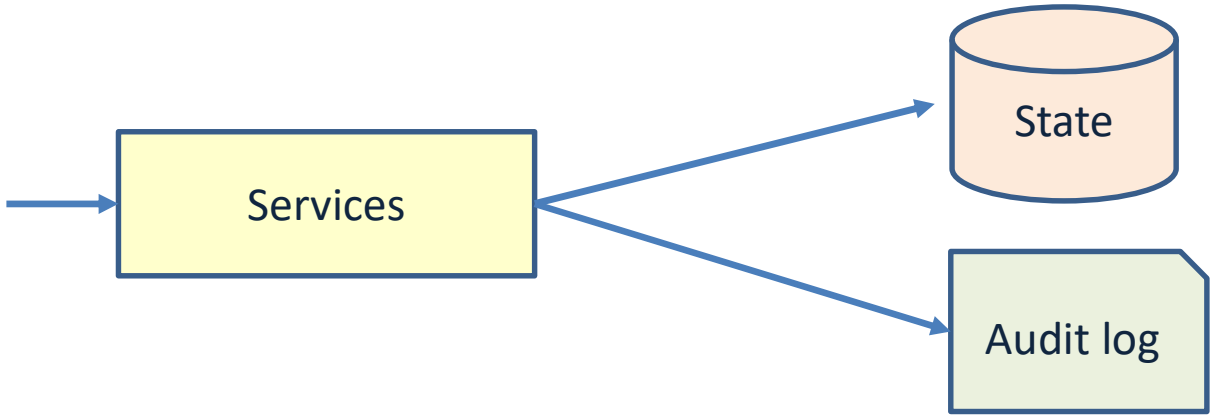
© 2021 ICT Intelligence

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

© 2021 ICT Intelligence

Audit log

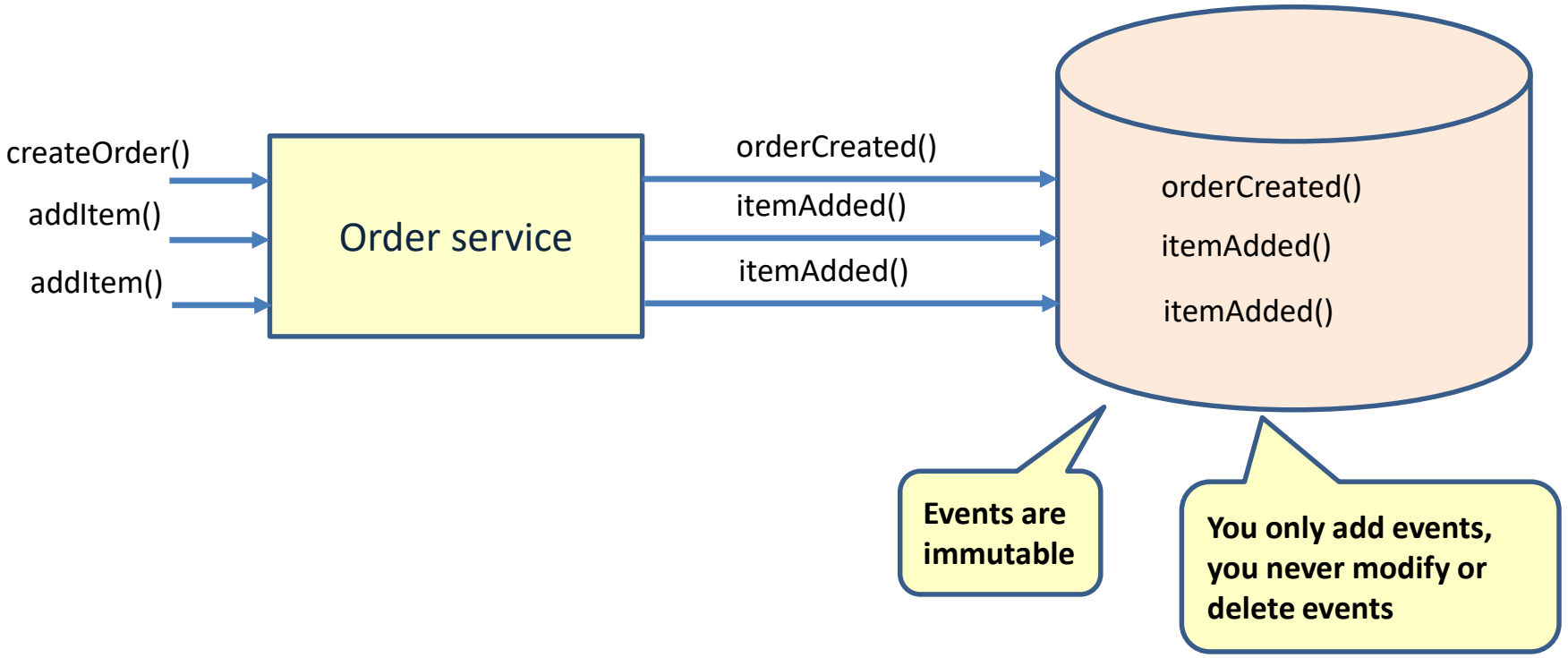


- Now we know how we got there
- What if the audit log gets out of sync with the state due to a bug?
- Which is the source of truth? (Audit log)

© 2021 ICT Intelligence

Event sourcing

- Only capture how we got there



© 2021 ICT Intelligence

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

© 2021 ICT Intelligence

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

© 2021 ICT Intelligence

Store the state of a system

Structural representation

- List of ordered goods
- Payment information
- Shipping information

© 2021 ICT Intelligence

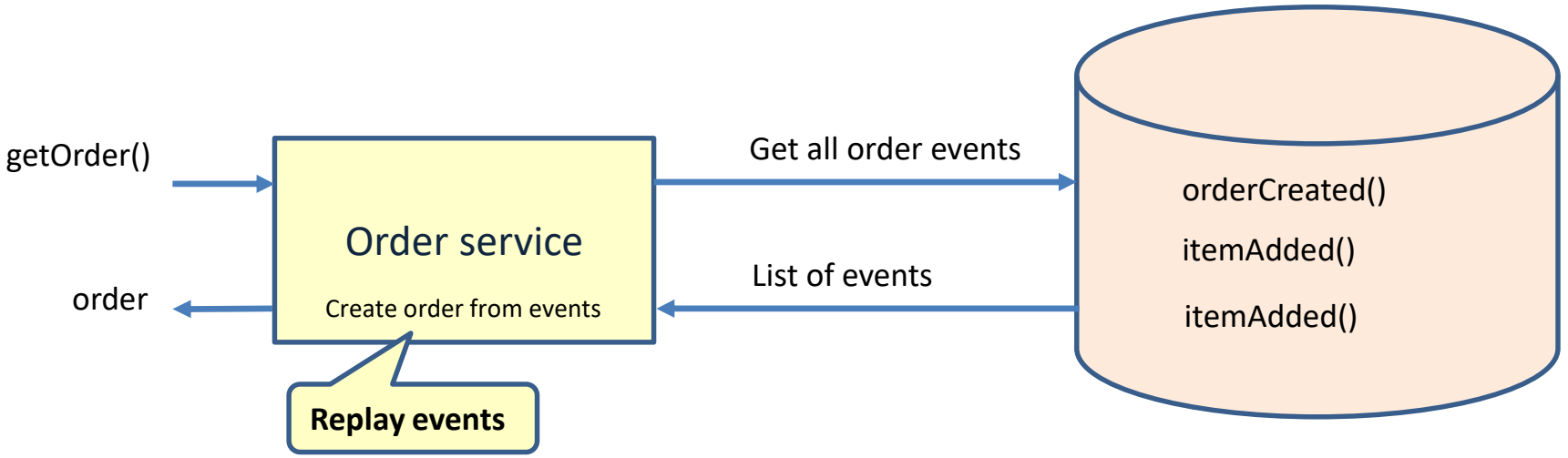
Store the events of a system

Event representation

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

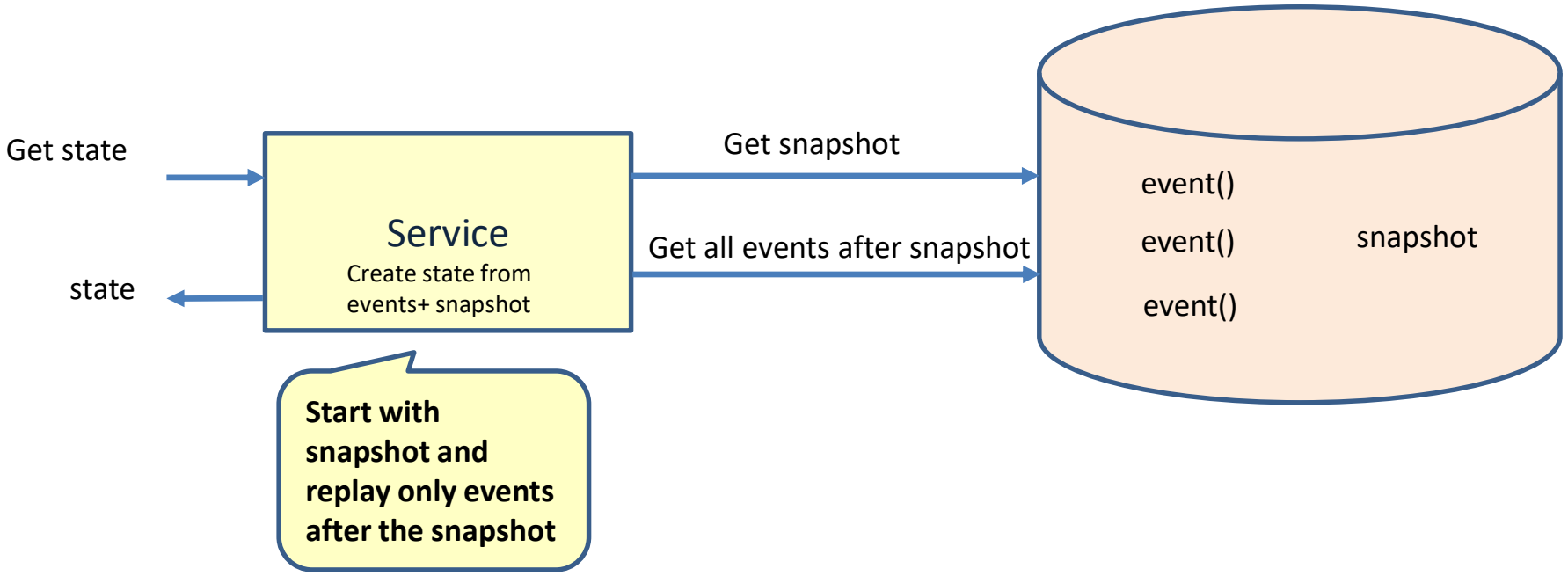
© 2021 ICT Intelligence

How do we get the state?



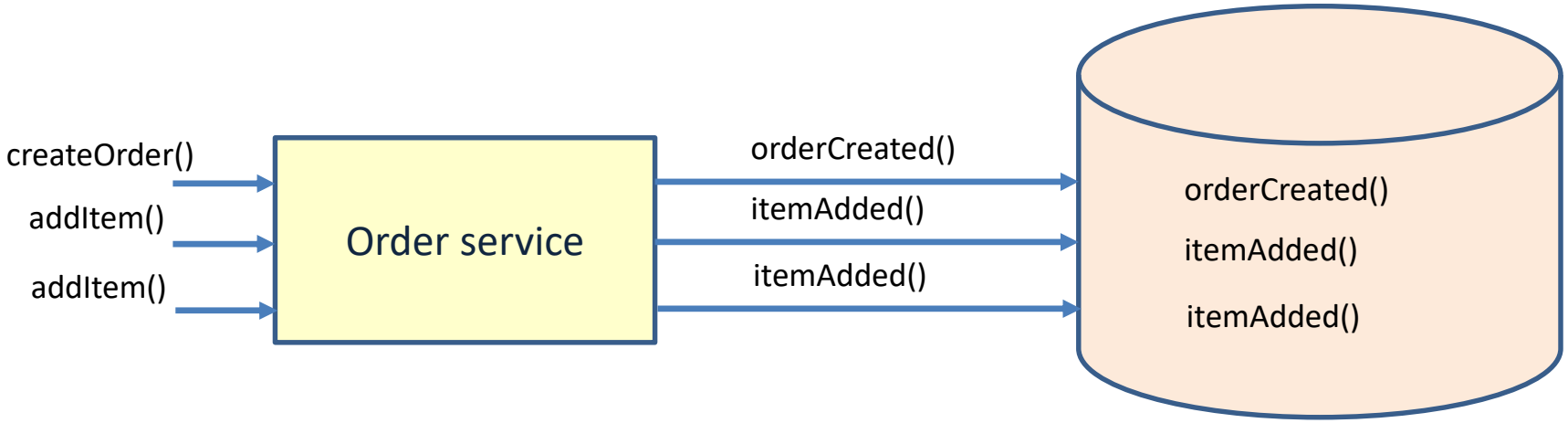
© 2021 ICT Intelligence

Snapshots



© 2021 ICT Intelligence

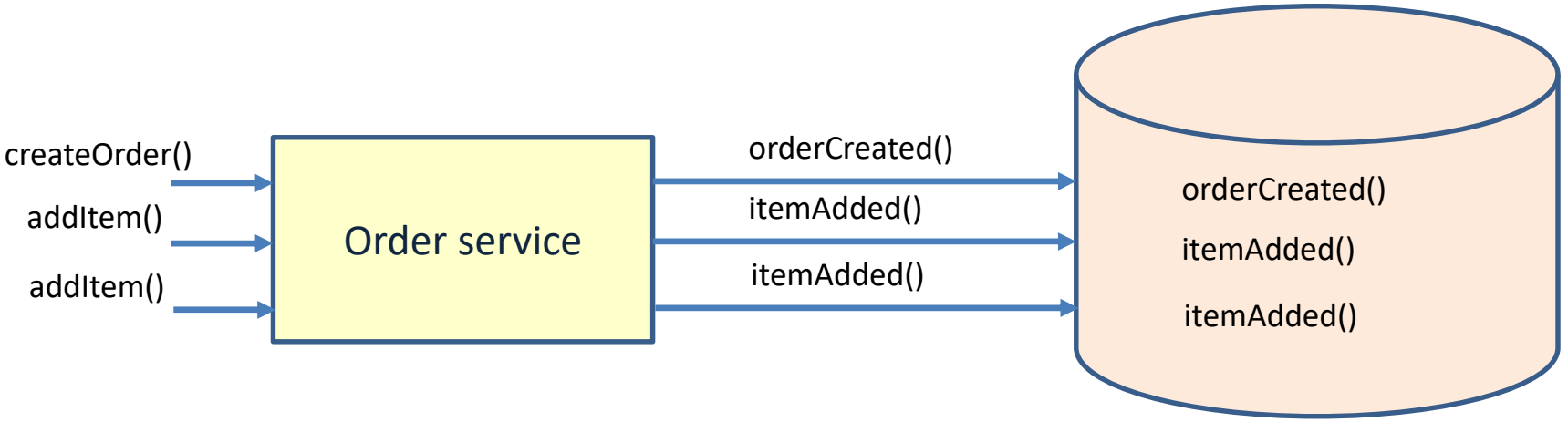
Event sourcing advantages



- 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

© 2021 ICT Intelligence

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

© 2021 ICT Intelligence

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

© 2021 ICT Intelligence

CENTRALIZED CONFIGURATION SERVICE

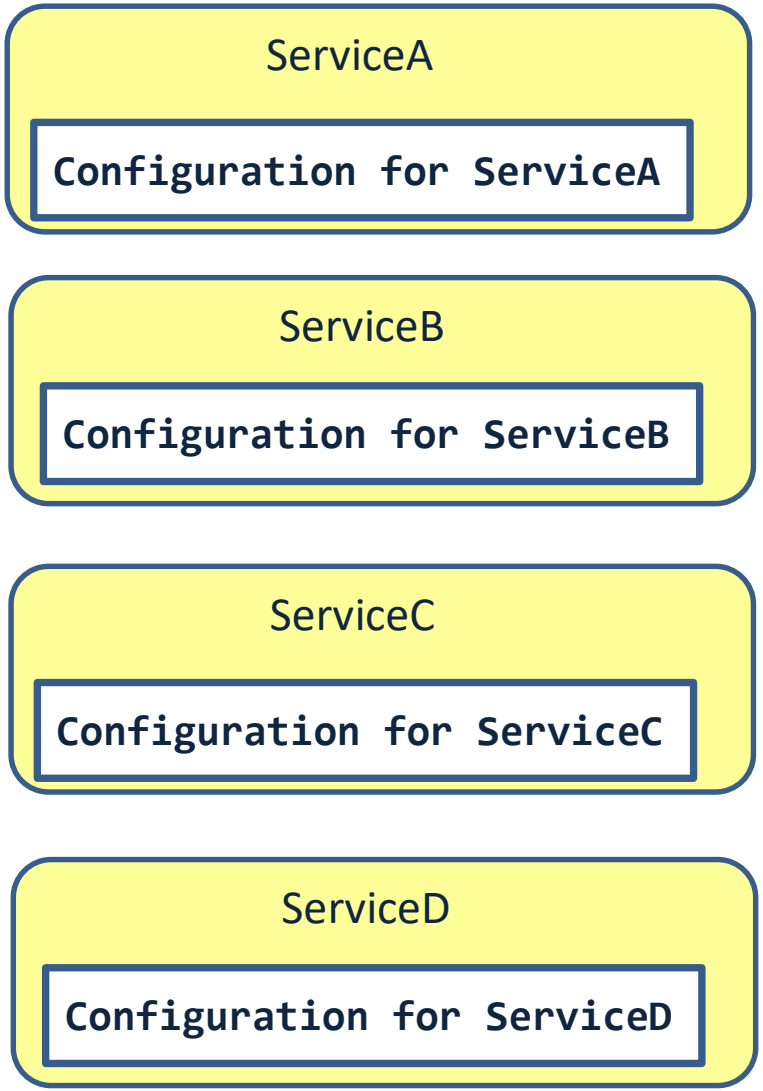
© 2021 ICT Intelligence

Configuration in microservices

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

© 2021 ICT Intelligence

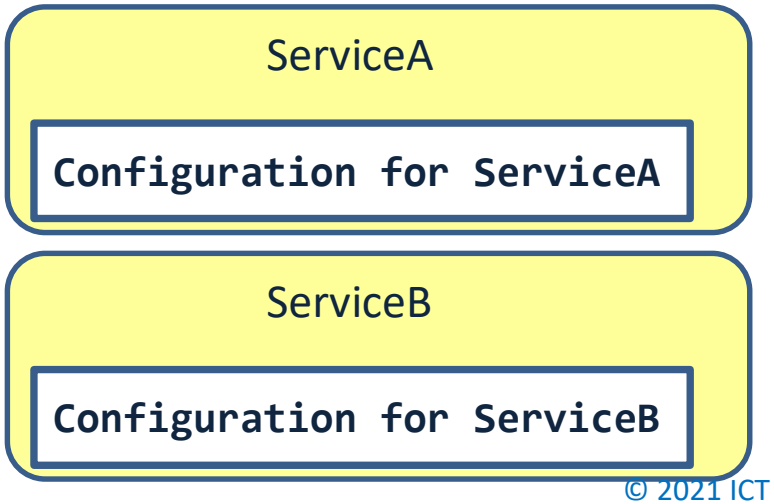
Local configuration



© 2021 ICT Intelligence

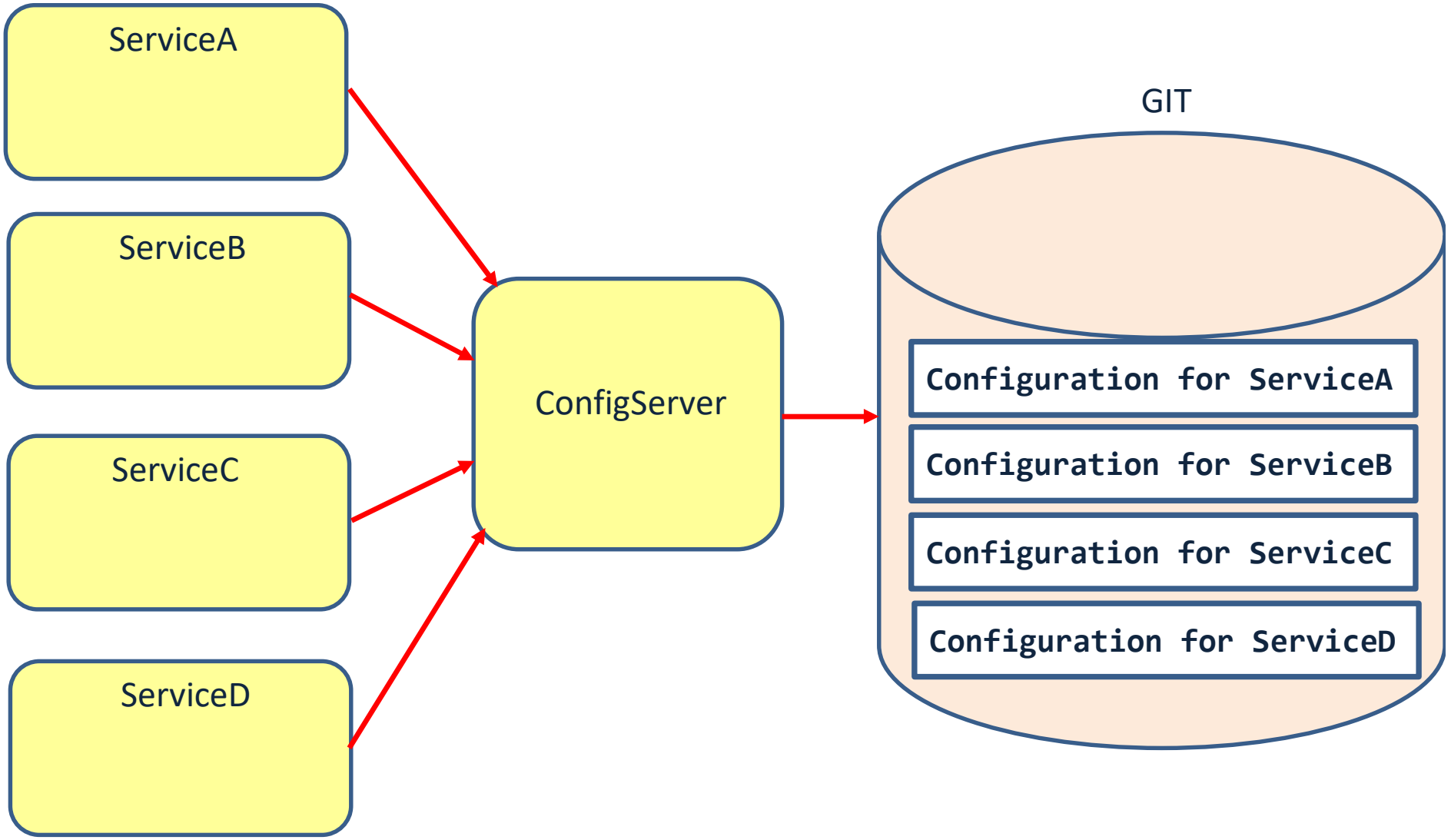
Local configuration challenges

- 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



© 2021 ICT Intelligence

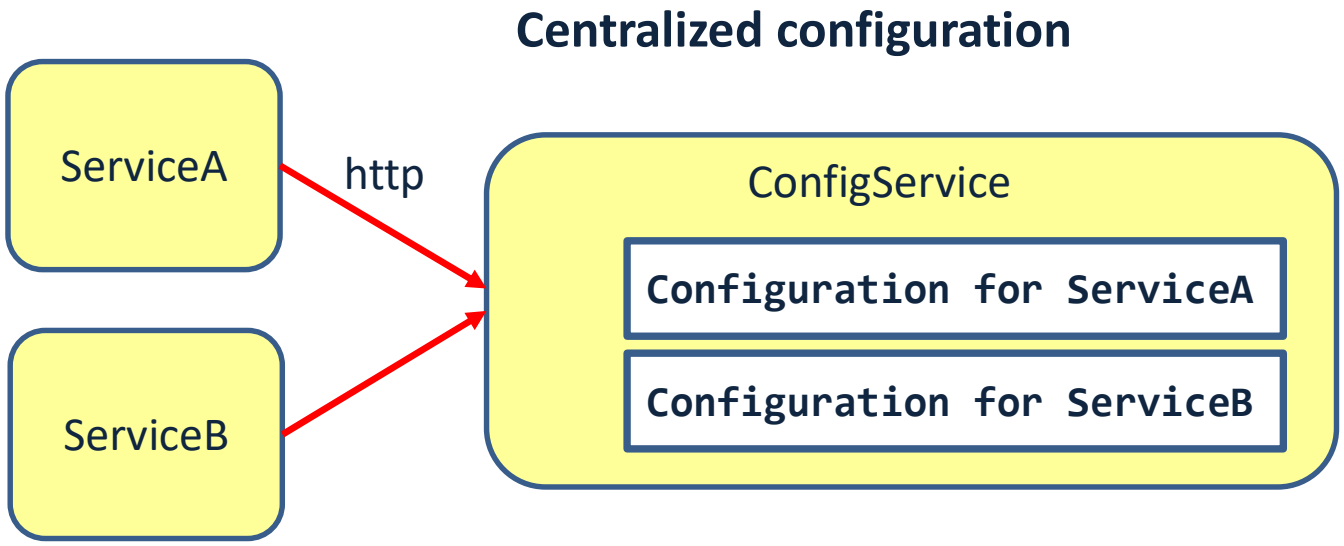
Spring cloud config server



© 2021 ICT Intelligence

Spring cloud config

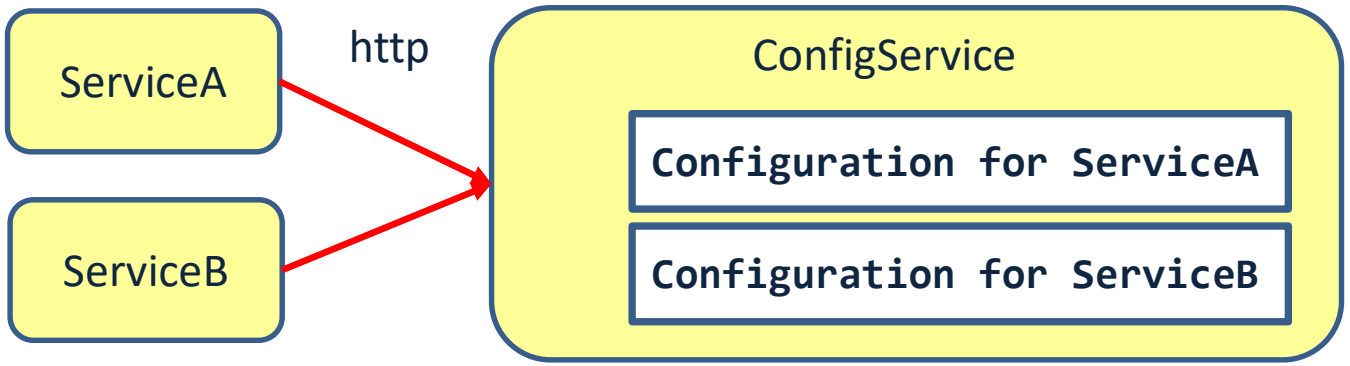
- HTTP access to centralized configuration



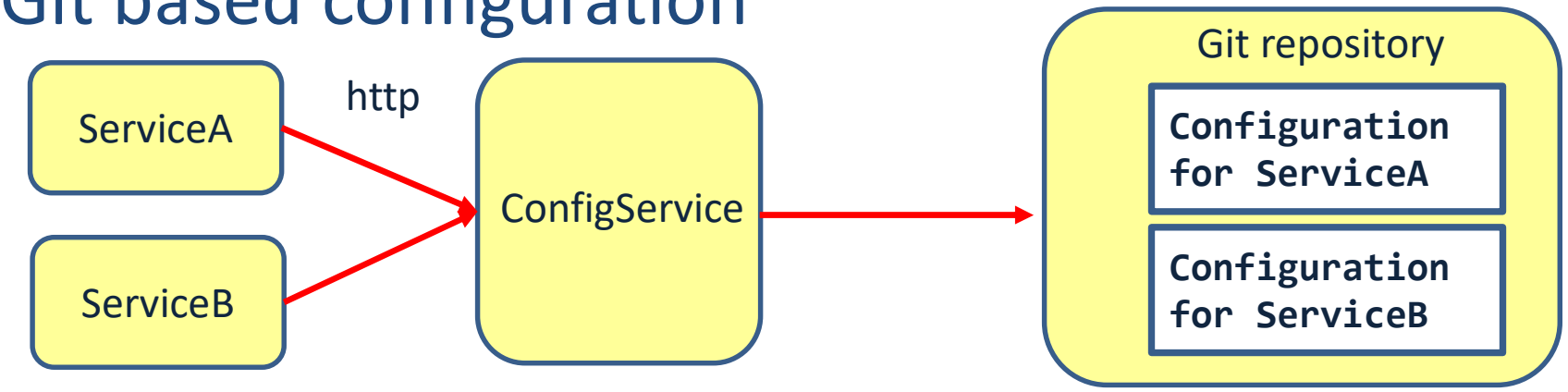
© 2021 ICT Intelligence

Spring cloud config

- File based configuration

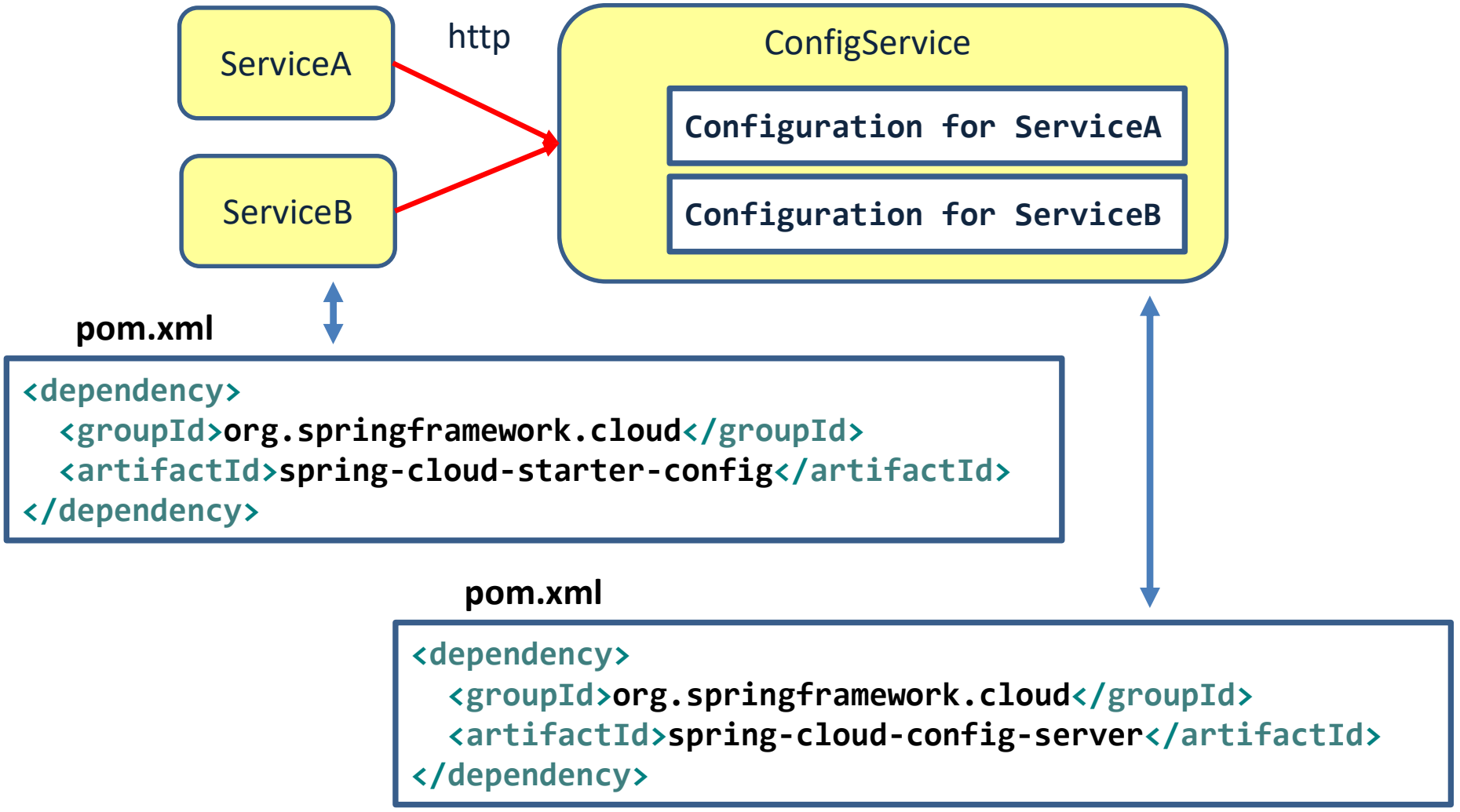


- Git based configuration



© 2021 ICT Intelligence

Spring cloud config example



© 2021 ICT Intelligence

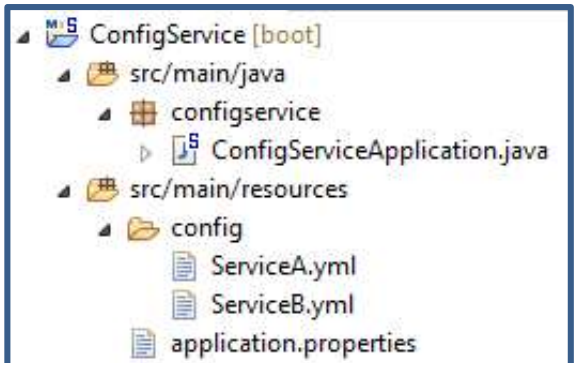
Configuration server

```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

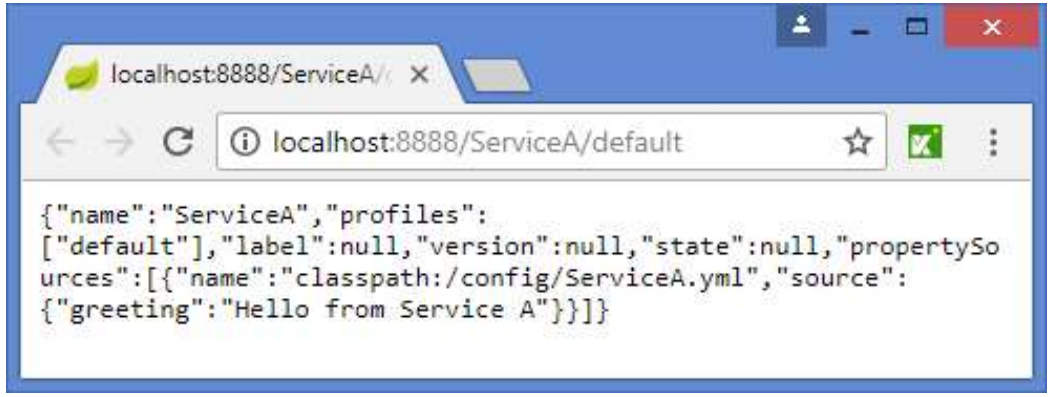
```
greeting: Hello from Service A
```

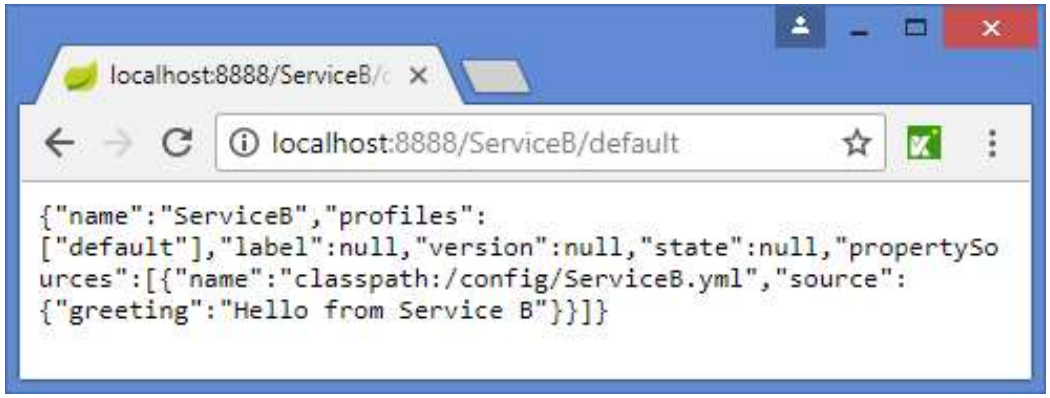
config/ServiceB.yml

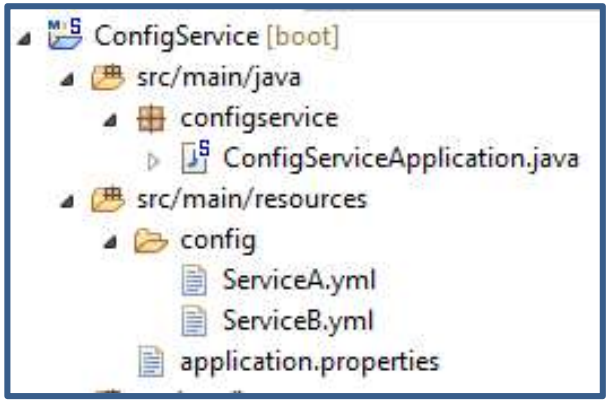
```
greeting: Hello from Service B
```

© 2021 ICT Intelligence

Configuration server





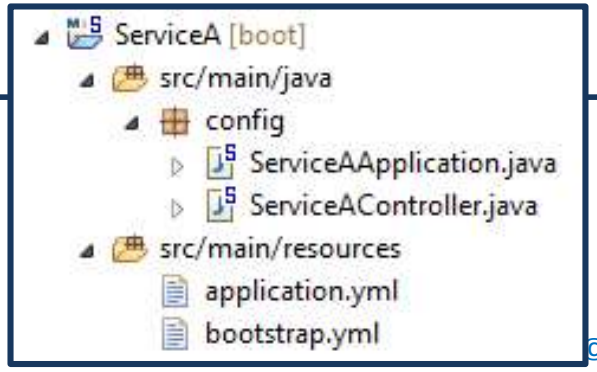


© 2021 ICT Intelligence

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
```

© 2021 ICT Intelligence

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

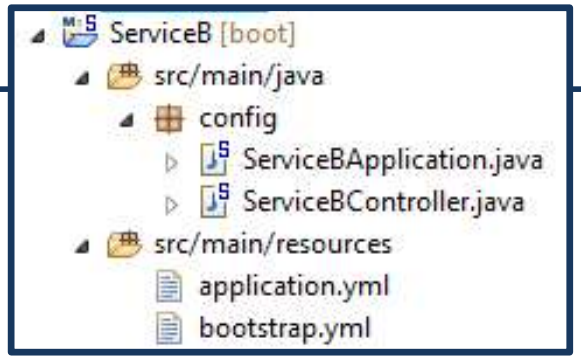
```yaml
server:
  port: 8090
```

© 2021 ICT Intelligence

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

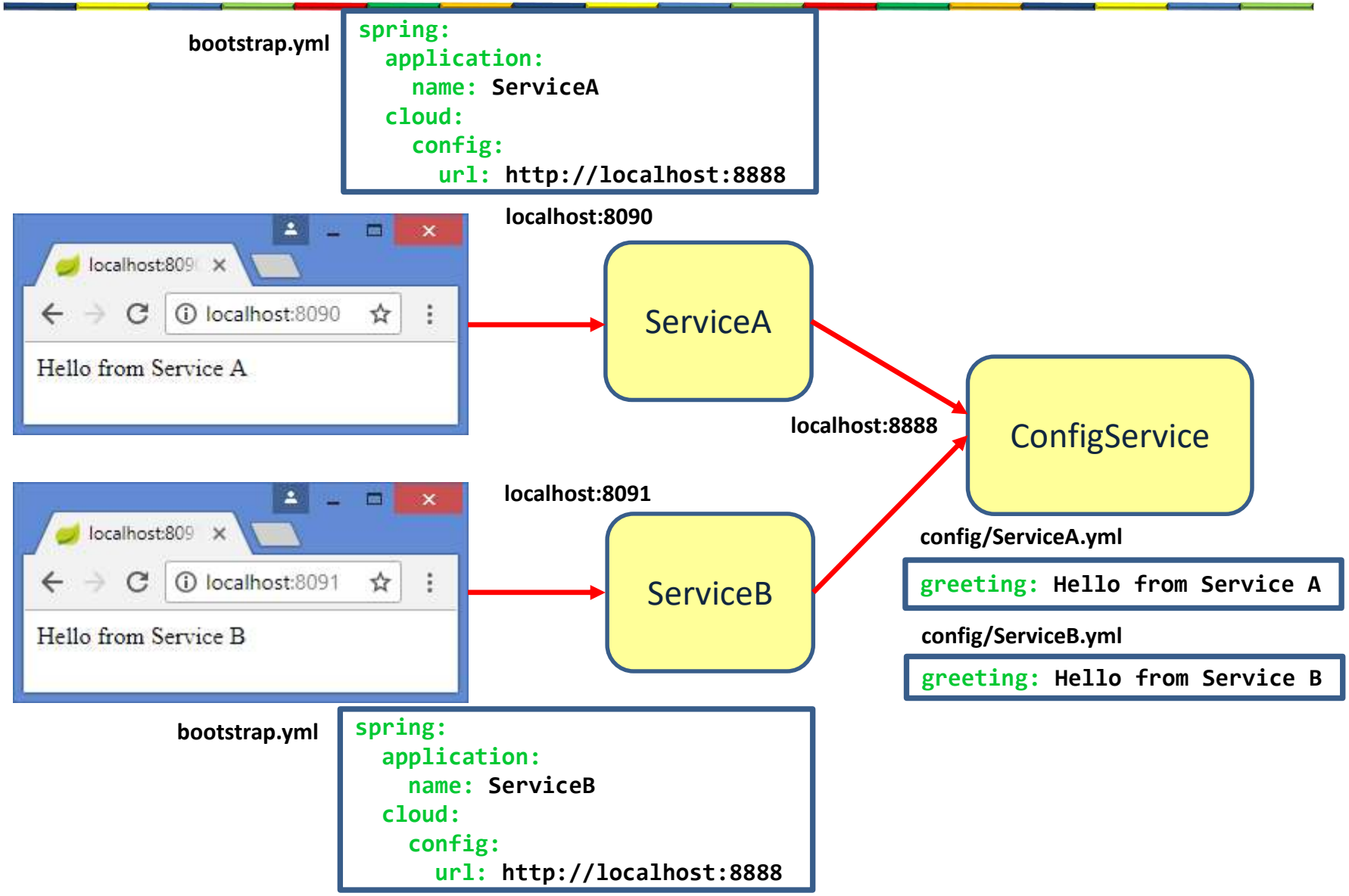
```yaml
server:
  port: 8091
```

bootstrap.yml

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

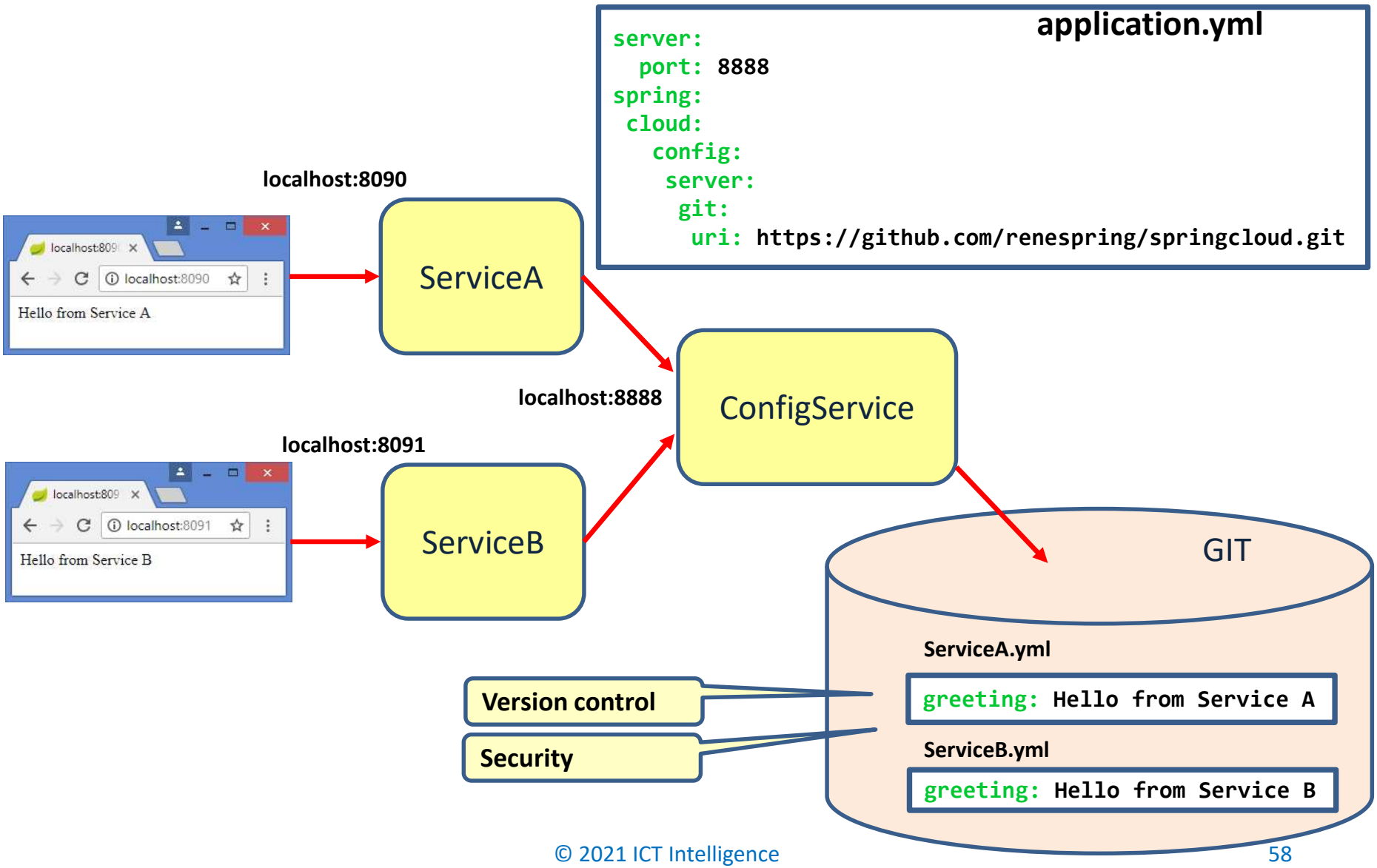
© 2021 ICT Intelligence

Use of the Config Server



© 2021 ICT Intelligence

Using Git (or GitHub)



© 2021 ICT Intelligence

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;
  }
}
```

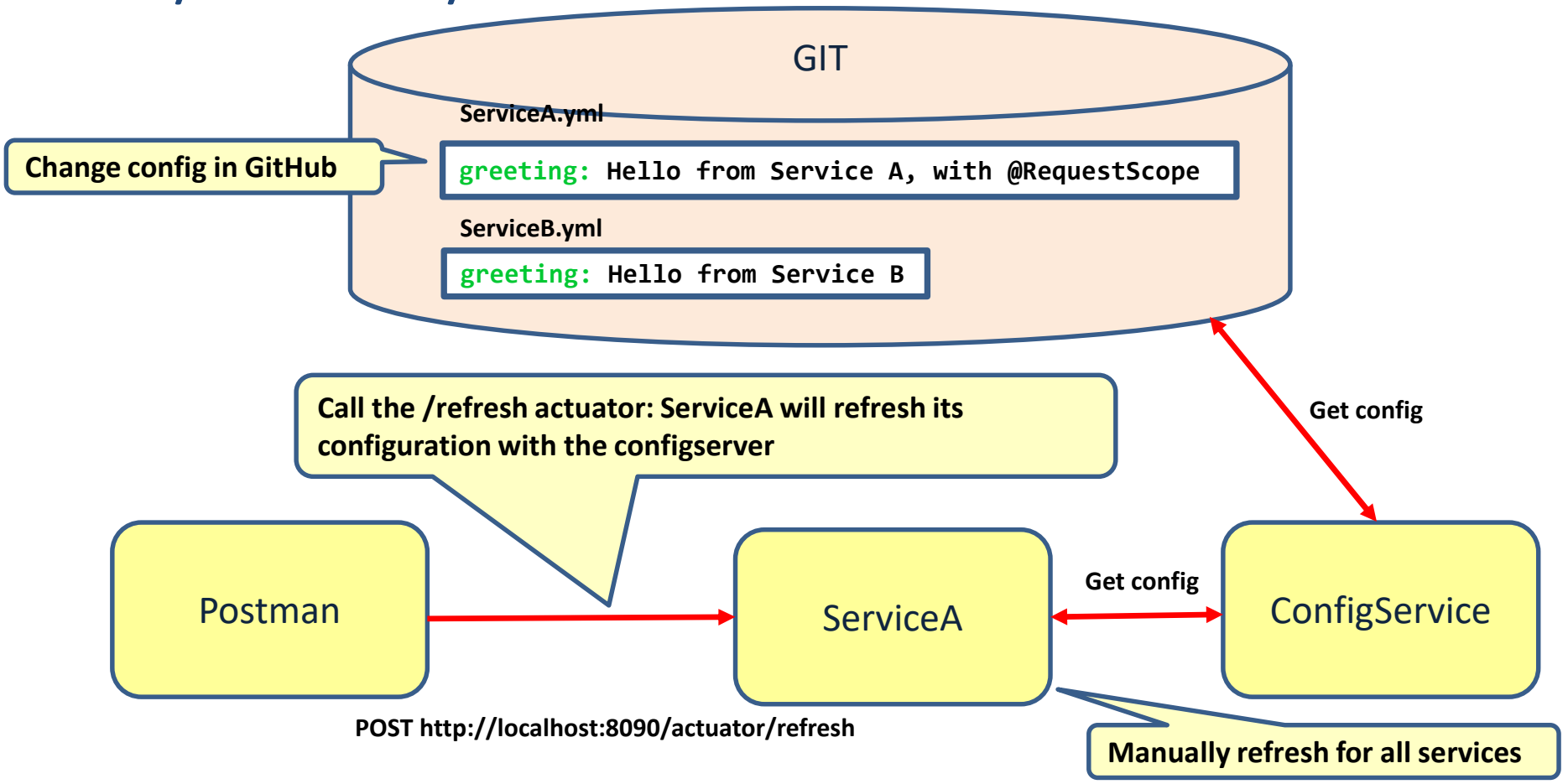
```
server:
 port: 8888

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


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

Expose the /refresh actuator

© 2021 ICT Intelligence

Refreshing configuration

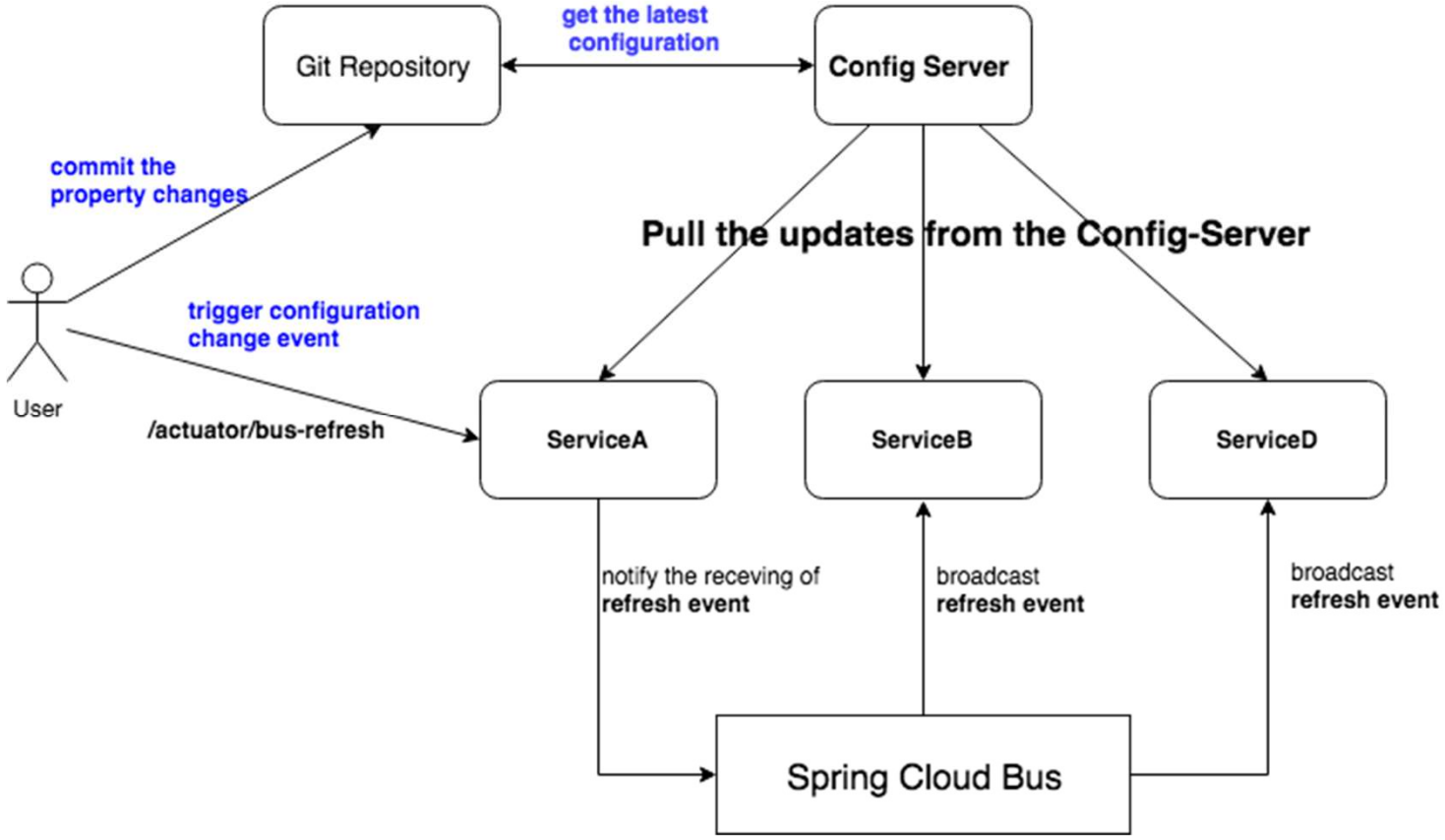
- Option 1: use @RefreshScope and "/actuator/refresh" event



© 2021 ICT Intelligence

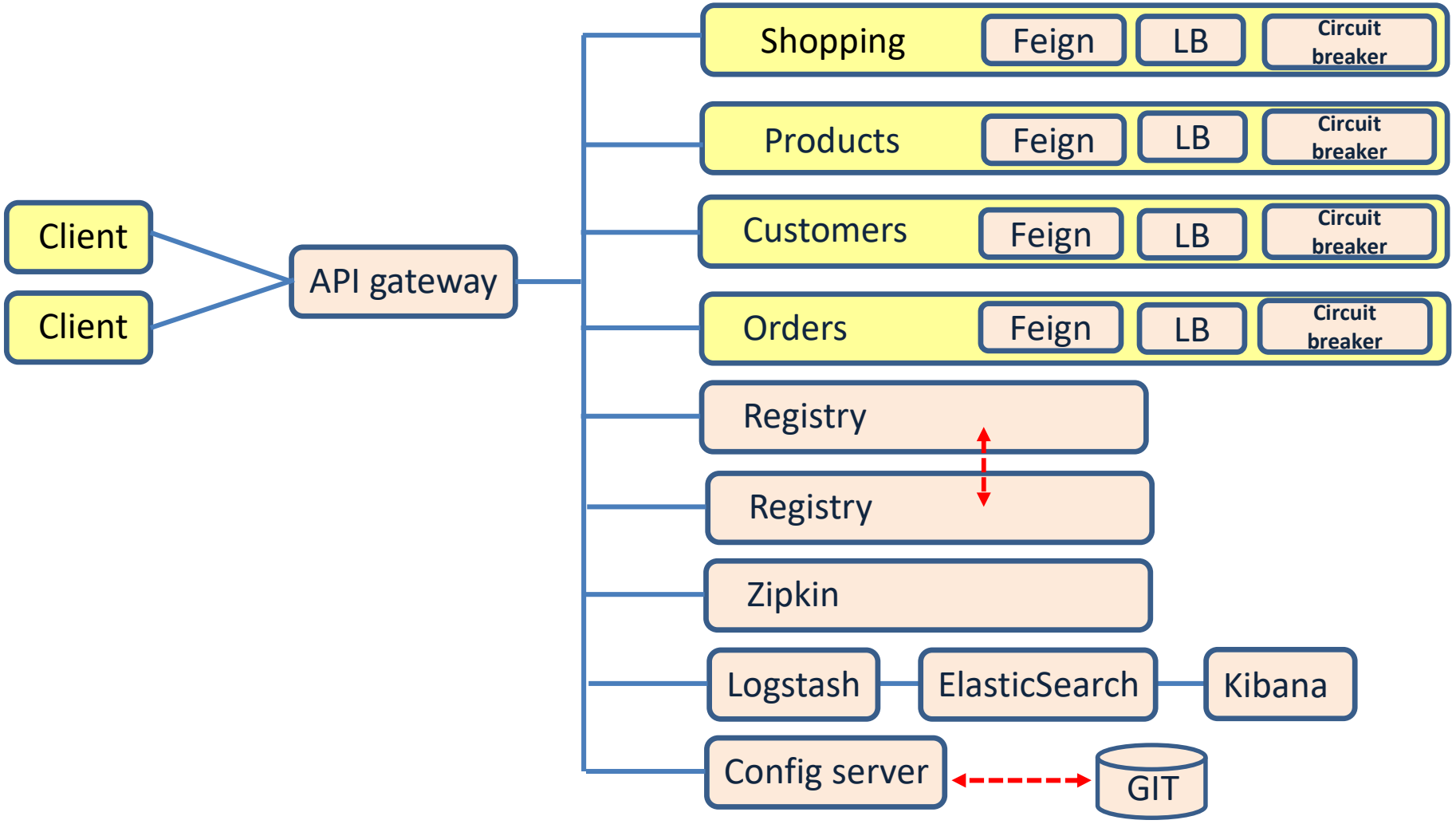
Refreshing configuration

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



© 2021 ICT Intelligence

Implementing microservices



© 2021 ICT Intelligence

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

© 2021 ICT Intelligence