Microservices development

November, 11-12 2017

Sergey Morenets, 2017

DEVELOPER 12 YEARS

TRAINER 4 YEARS

WRITER 3 BOOKS

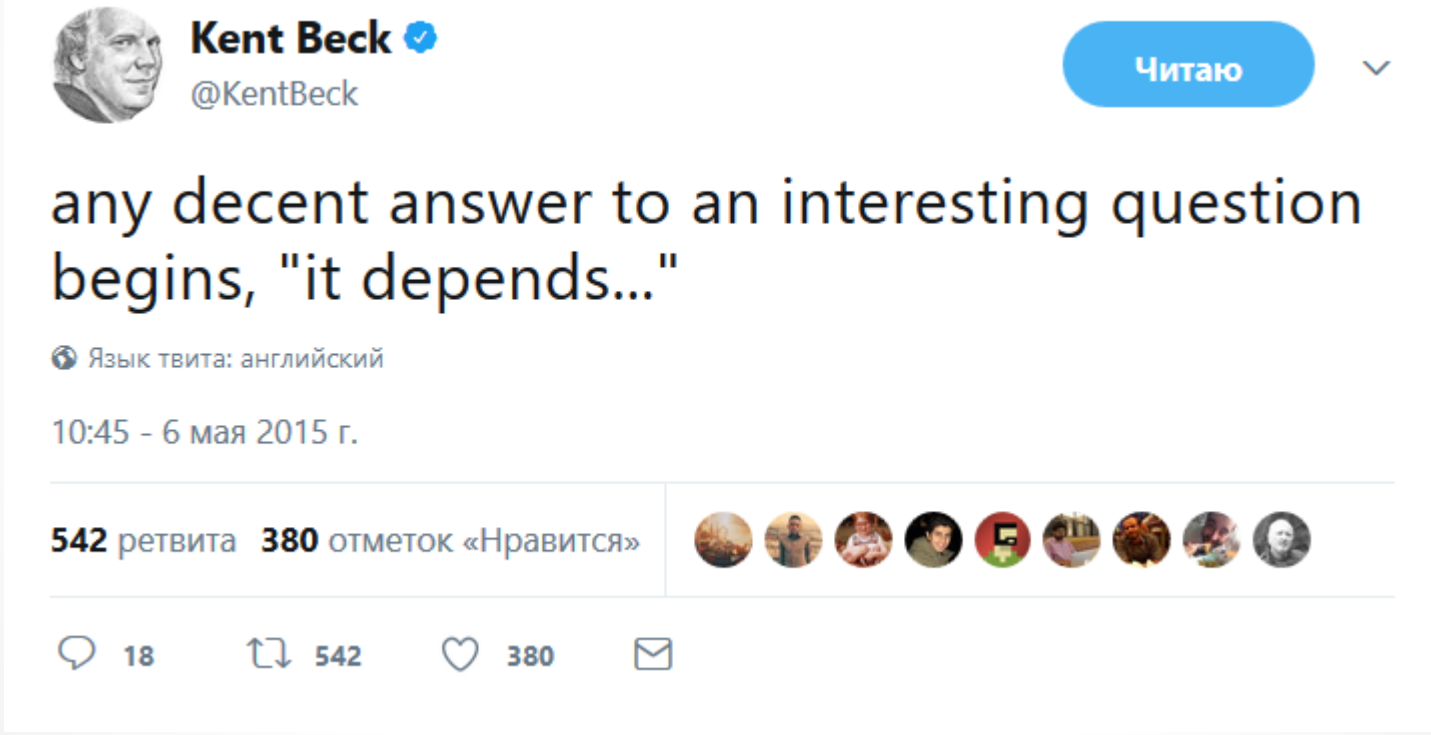
FOUNDER

SPEAKER

Sergey Morenets, 20

Agenda

Agenda



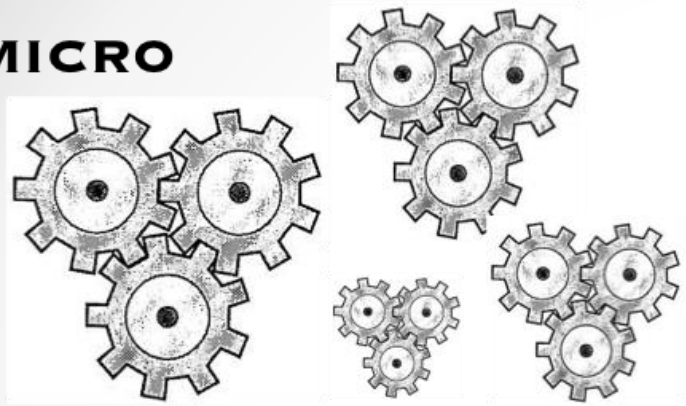
- Sergey Morenets, 2017

Agenda

- ✓ Spring Framework infrastructure
- ✓ Complexity of monolith applications
- ✓ Micro-service architecture. Pro and cons
- ✓ Event-Driven architecture & patterns
- ✓ Event sourcing
- ✓ CQRS
- ✓ Redis
- ✓ Apache Kafka
- ✓ Testing

Sergey Morenets, 2017

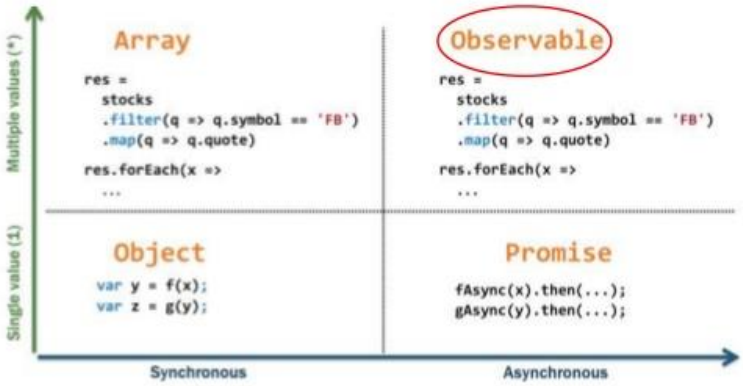
MICRO



SERVICES

IT DISCOVERY

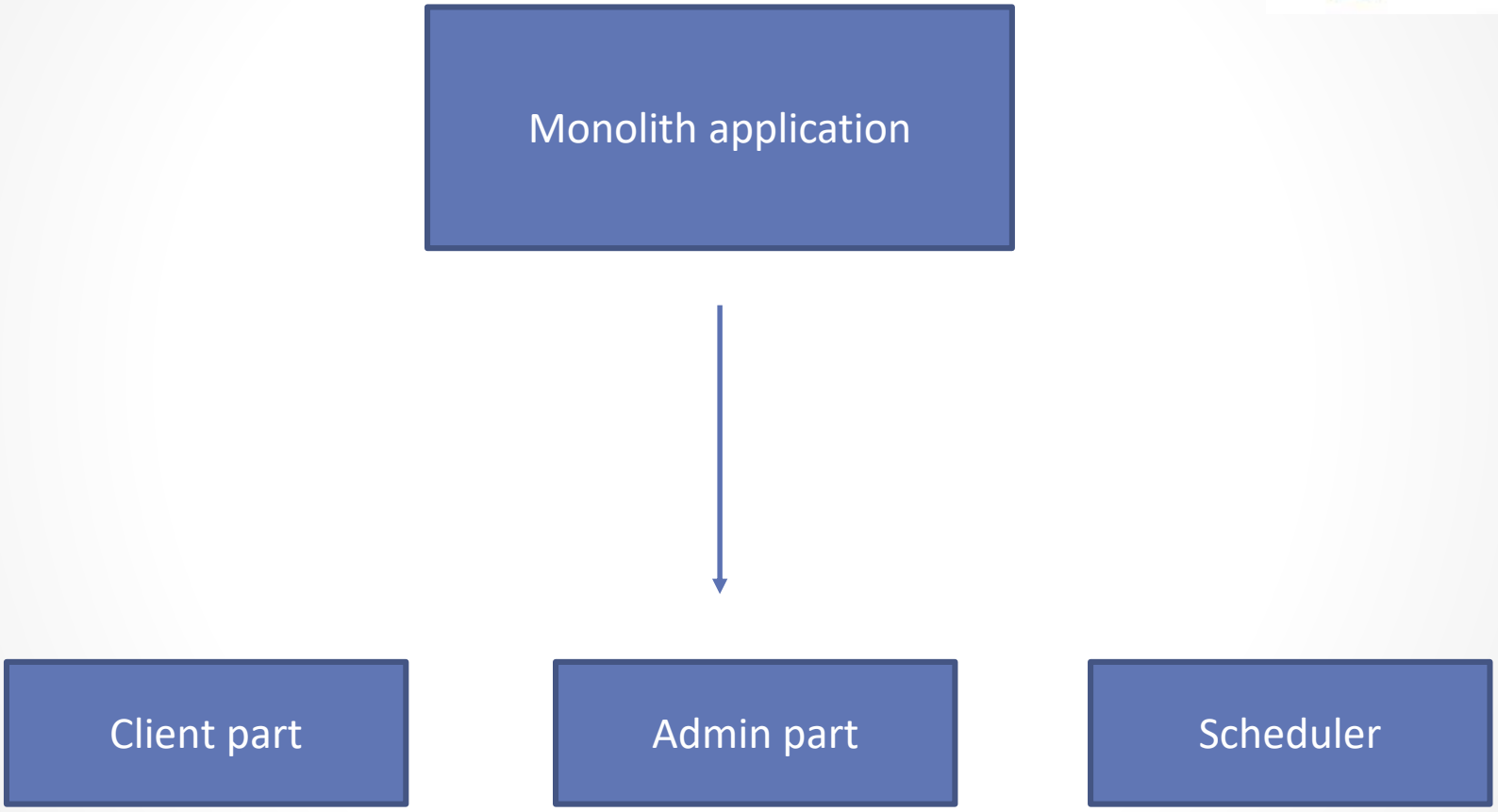
Reactive Programming



[5]

- Sergey Morenets, 2017

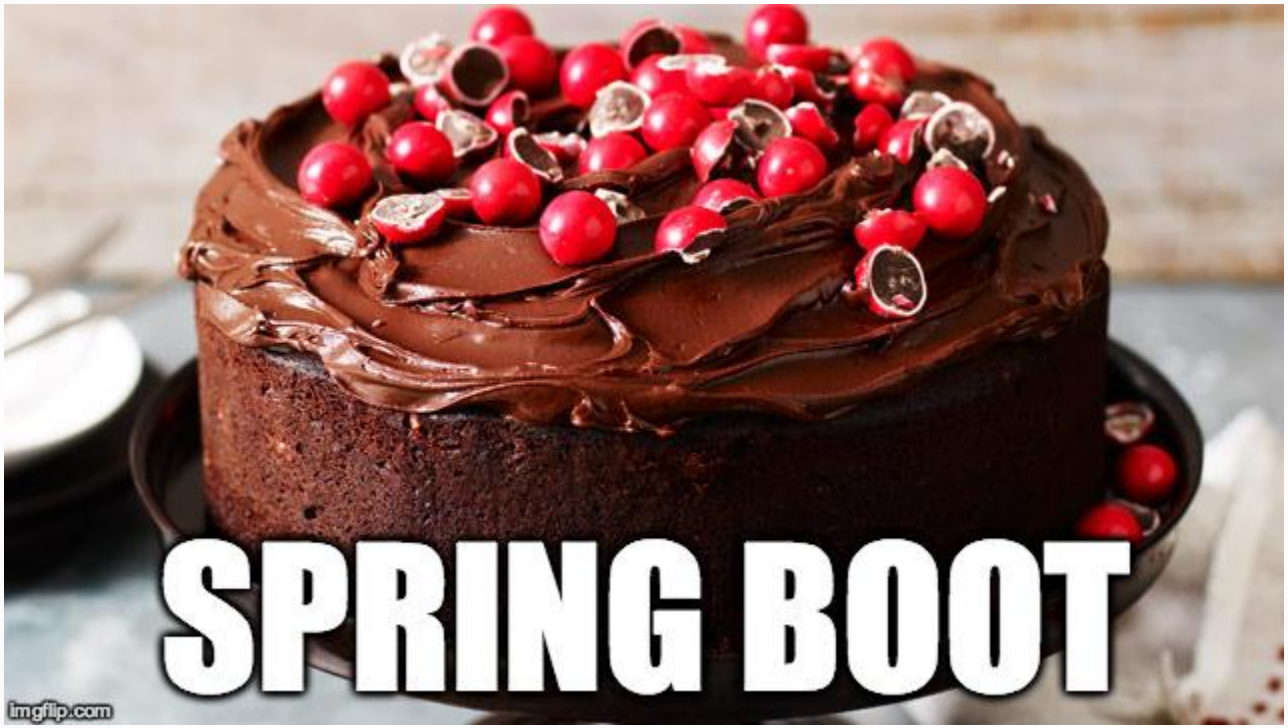
Story with happy-end



Spring Framework

- Based on **DI**(IoC) conversion
- Integrated with most popular frameworks
- Contains over **30** sub-projects

Sergey Morenets, 2017



Sergey Morenets, 2017

Spring Boot

- Stand-alone Spring applications
- Embed Tomcat, Jetty or Undertow directly
- Automatically Spring configuration
- Convention-over-configuration
- Absolutely no code generation and no requirement for XML configuration
- Focus on business features and less on infrastructure

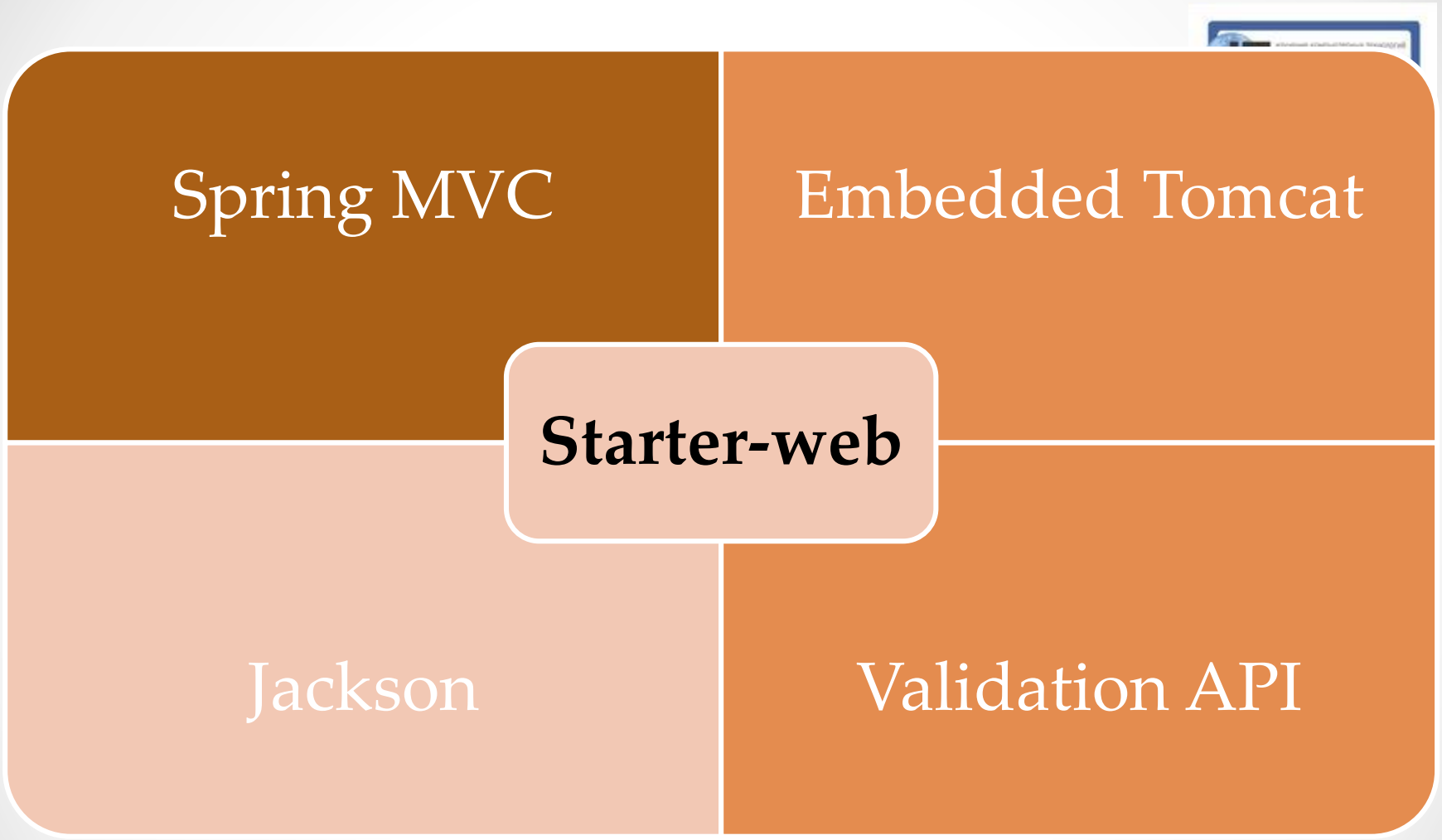
Build management

Maven. Web starter


```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>${spring.boot.version}</version>
</dependency>
```

Startup code


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


Spring MVC
Embedded Tomcat
Starter-web
Jackson
Validation API

REST service

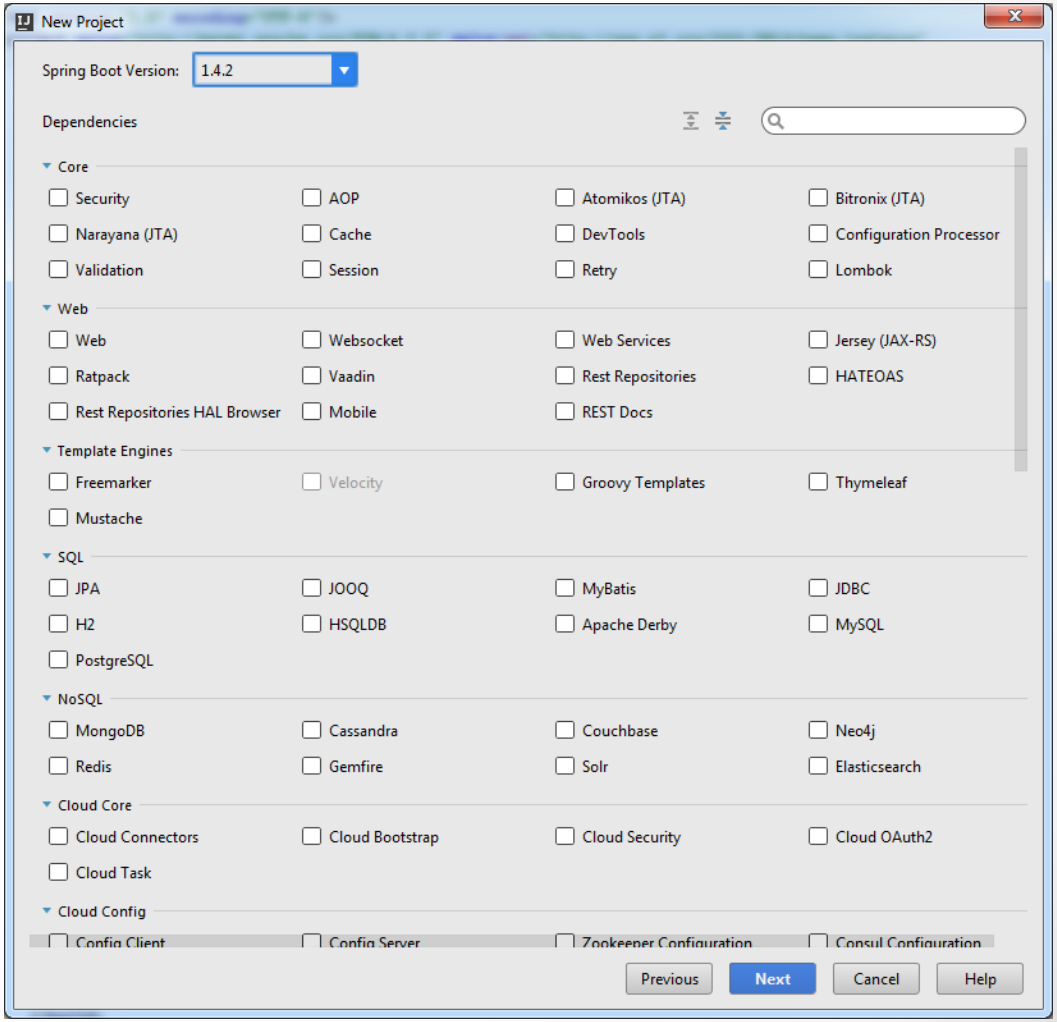

```
@RestController
@RequestMapping("book")
public class BookController {

    @GetMapping("/{id}")
    public Book getBook(@PathVariable int id) {
        return bookRepository.findBookById(id);
    }
```

IDE

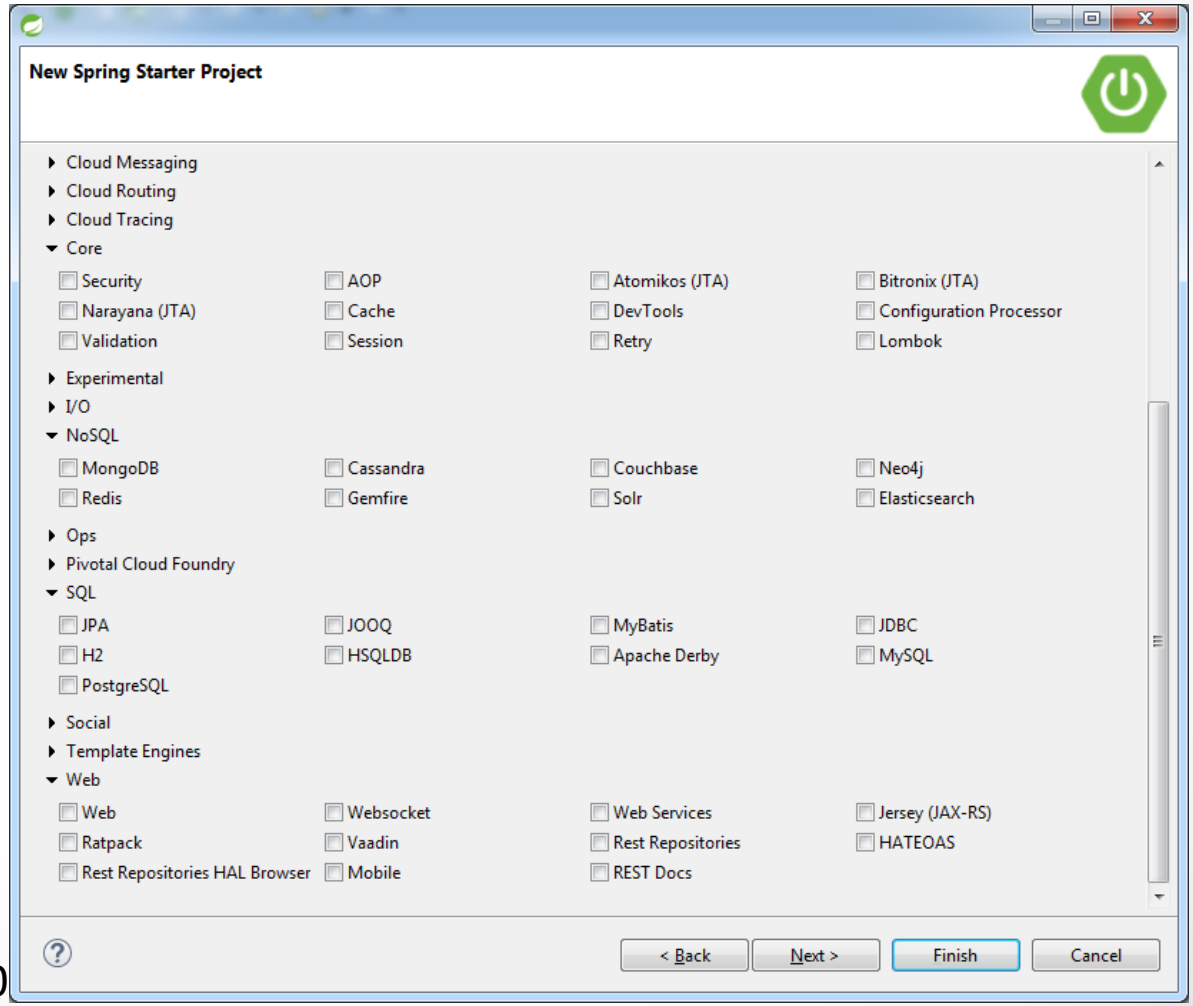
- Sergey [illegible]

Spring Initializr



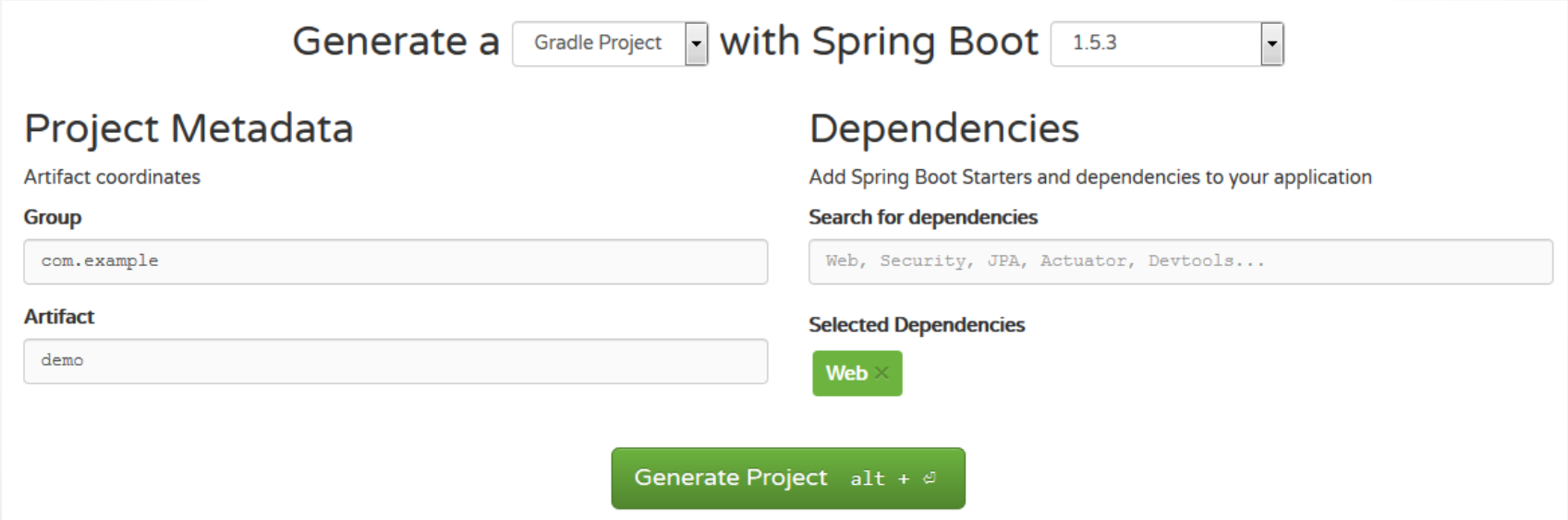
- Sergey Morenets, 2017

Spring Boot Starter



- Sergey Morenets, 20

start.spring.io



- Sergey Morenets, 2017

Spring Boot plugins


```
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <version>${spring.boot.version}</version>
    <executions>
        <execution>
            <goals>
                <goal>repackage</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

```
plugins { id 'org.springframework.boot' version '1.5.7.RELEASE' }
```

```
apply plugin: 'org.springframework.boot'
```

Built management tasks

Maven

- spring-boot:run
- spring-boot:repackage

Gradle

- bootRun
- bootRepackage

Spring Boot. Dev tools

- ✓ Automatic restart when file(s) on a classpath changes
- ✓ **LiveReload** server support
- ✓ Remote application support
- ✓ Dev customization by default

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <version>${spring.boot.version}</version>
    <optional>true</optional>
</dependency>
```

Task 1. Spring Boot and REST services

1. Create Spring Boot project using **your IDE**
2. Create Spring Boot project using http://start.spring.io and open it in IDE
3. **Review** project configuration/contents
4. Write simplest REST service(GET and POST methods)

What is monolith application?

Sergey Morenets, 2017

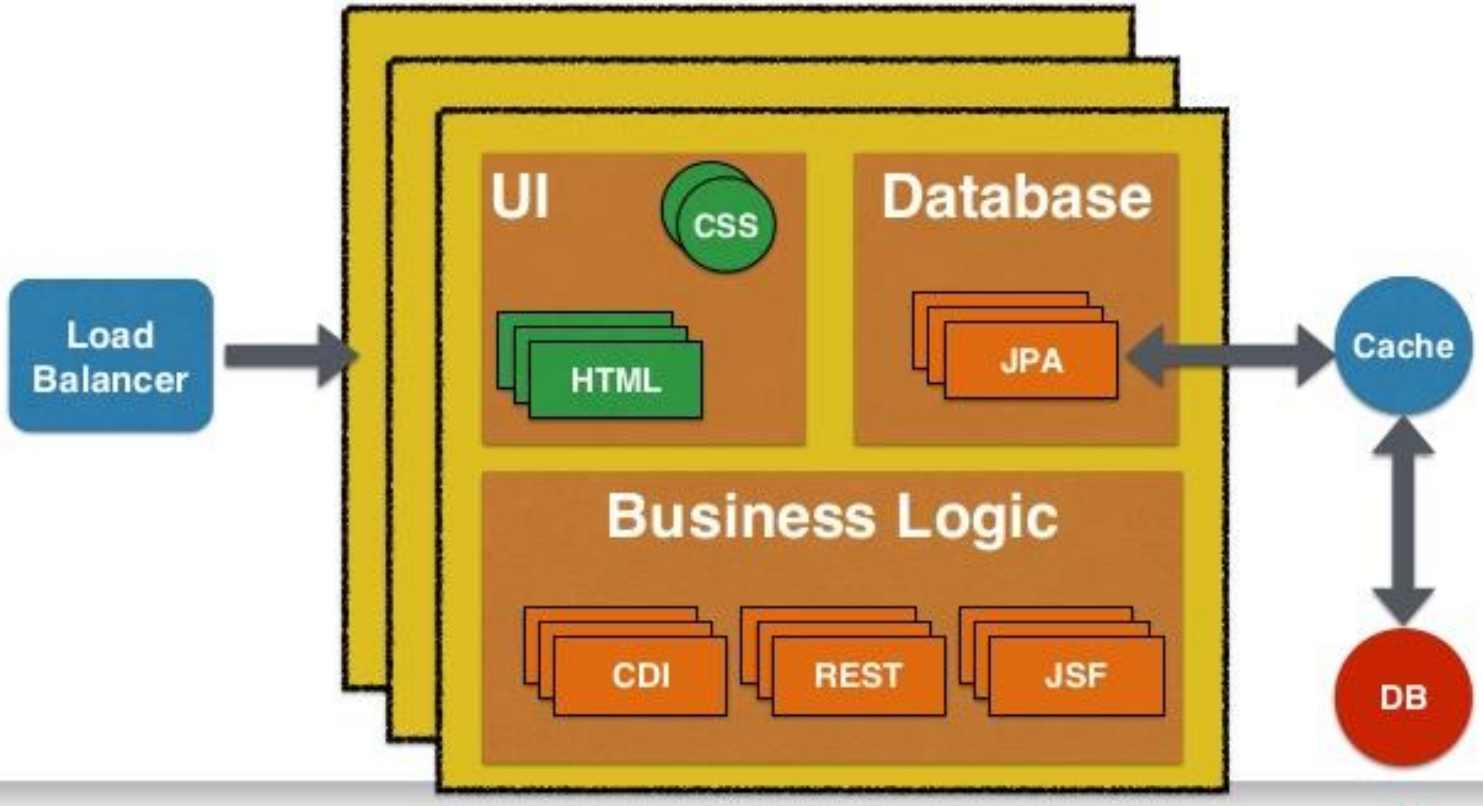
Monolith application

- ✓ Single **deployment** unit(WAR, EAR)
- ✓ Single codebase
- ✓ No restrictions on the **project size**
- ✓ Single **database** (RDBMS)
- ✓ Single **language**
- ✓ Long development **iterations**
- ✓ Fixed technology **stack**(JEE, Spring, Hibernate)
- ✓ **ACID** principle
- ✓ One or few **teams**

Monolith application

- ✓ **Tight coupling** between modules
- ✓ **Failures** could affect the whole application
- ✓ **Good** for small/average applications

Monolith Application



redhat.

Sergey Morenets, 2017

Issues

UI

Sergey Morenets, 2017

Issues

- ✓ Hard to maintain
- ✓ Hard to add new features (**fast**)
- ✓ Hard to scale (specific components)
- ✓ Hard to deploy
- ✓ Slower to start
- ✓ Slower to work in IDE
- ✓ Cannot deploy single module
- ✓ Cannot learn the whole project

Task 2. Monolith application

1. Import **monolith** project into your IDE
2. Review project functionality.
3. Update **BookController** class and add necessary **Spring** annotations.
4. Run application as **Spring Boot** project and observe its behavior.
5. Identify issues related by the monolith architecture and possible solutions for them.

What is micro-service?

Sergey Morenets, 2017

What is micro-service?

- ✓ **100** lines of code
- ✓ **1 week** of coding
- ✓ **1 day** of documenting
- ✓ **Single** package
- ✓ Application packaged into **container**
- ✓ Single **framework/language**
- ✓ Work for **one man/team**
- ✓ **Single** functionality

Micro-service

- ✓ Loosely coupled service oriented architecture with bounded contexts

- ✓ Small autonomous service

Sergey Morenets, 2017

Micro-services

- ✓ **Separately** written, deployed, scaled and maintained
- ✓ **Independently** upgraded
- ✓ **Easy** to understand/document
- ✓ Provides **business** features
- ✓ **Fast** deployment
- ✓ Use **cutting-edge** deployment
- ✓ Resolve **resource conflicts**(CPU, memory)
- ✓ Communication via **lightweight** protocols/formats

Micro-services

- ✓ **Smaller** and simpler applications
- ✓ Fewer **dependencies** between components
- ✓ Scale and develop independently
- ✓ Easy to introduce new **technologies**
- ✓ Cost, size and risk of changes reduced
- ✓ Easy to **test** single functionality
- ✓ Easy to introduce **versioning**
- ✓ **Cross-functional** distributed teams
- ✓ Improved security due to multiple data-sources
- ✓ Increased uptime

Micro-services design principles

- ✓ High cohesion (**SOLID**)
- ✓ Autonomous (from other services)
- ✓ Business-domain centric (business function)
- ✓ Resilience (respond to **failures**)
- ✓ Observable (system **health**, monitoring, logging)
- ✓ Automated (**reduce** time to setup environment & test)

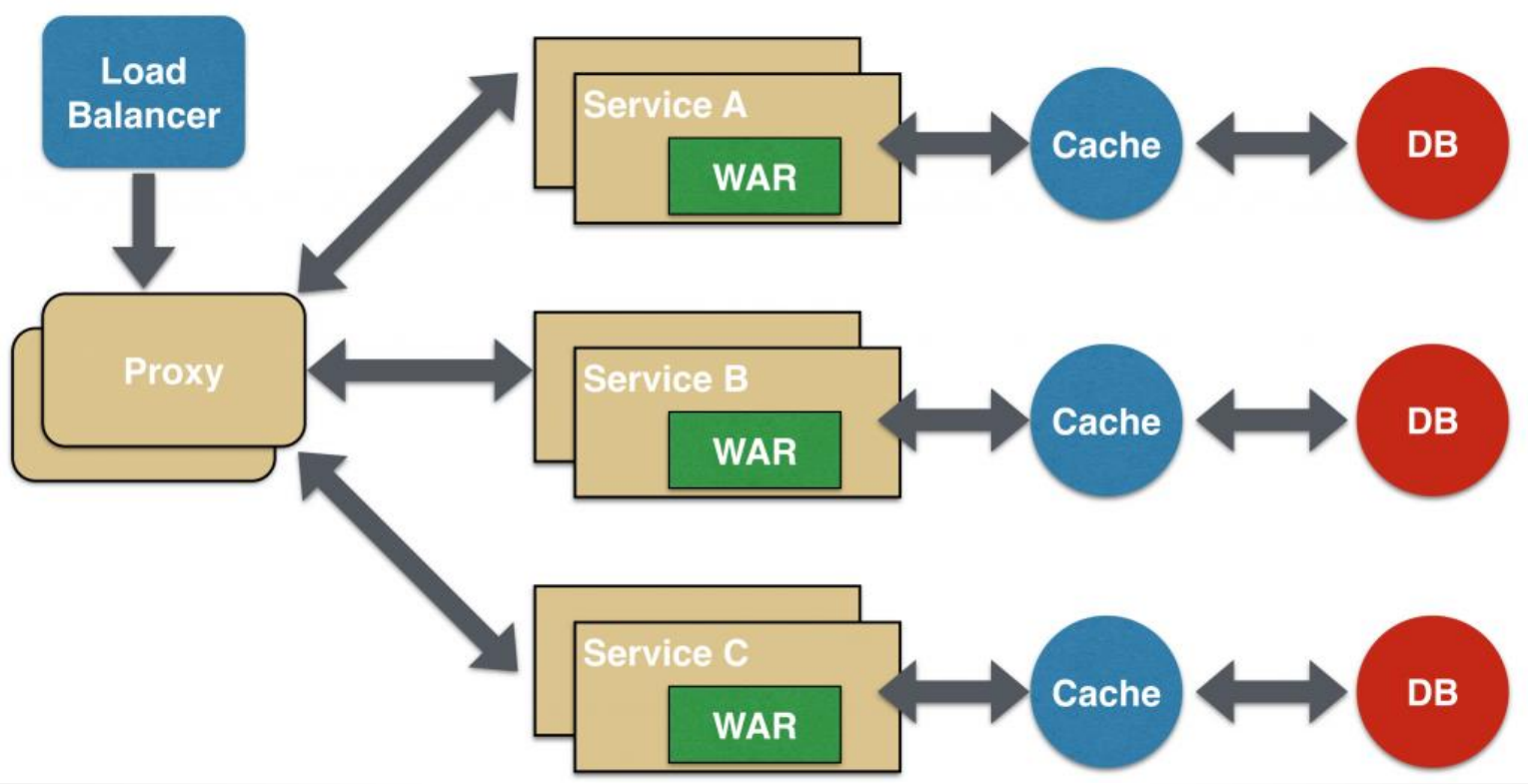
Micro-services

- ✓ No enterprise data model
- ✓ No transactions
- ✓ Micro-services don’t resist changes in other services

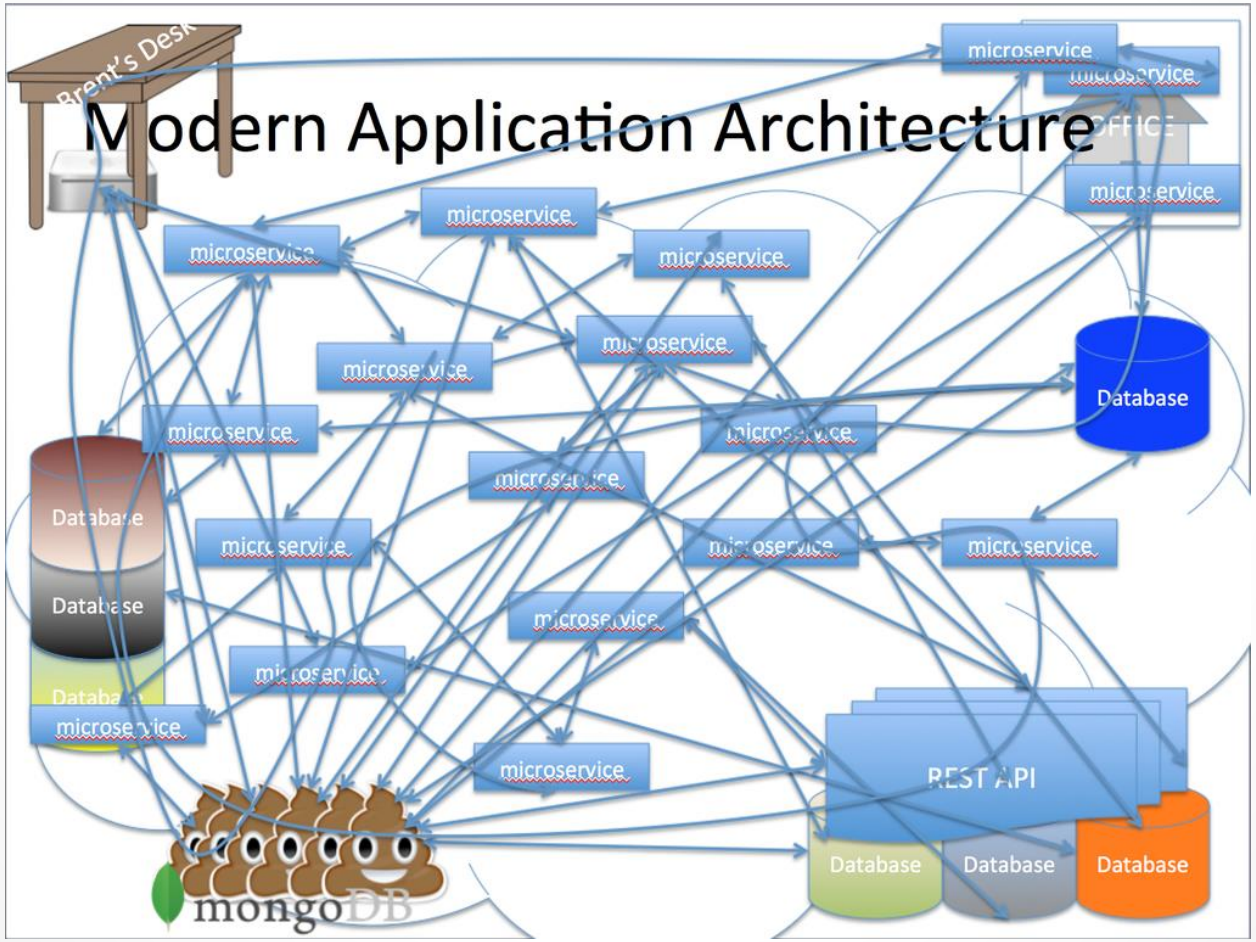
Drawbacks

- ✓ Higher level of complexity
- ✓ Transaction management
- ✓ Testing of distributed application
- ✓ Deployment and management
- ✓ Cost of remote calls

Drawbacks


Load Balancer
Proxy
Service A
WAR
Service B
WAR
Service C
WAR
Cache
Cache
Cache
DB
DB
DB

Drawbacks

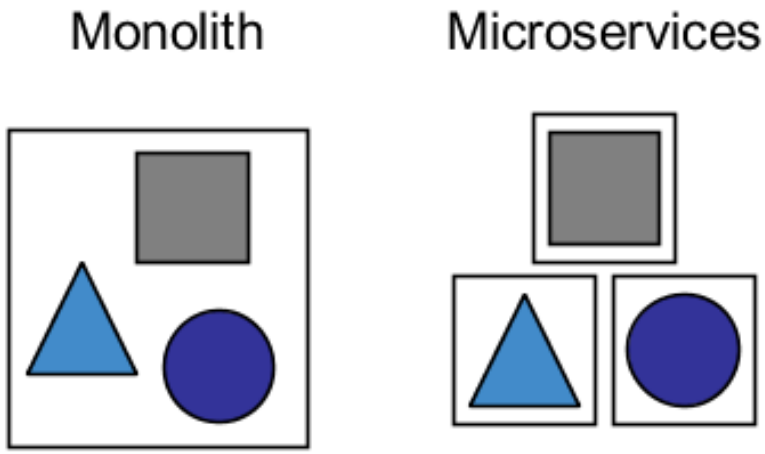


Sergey Morenets, 2017

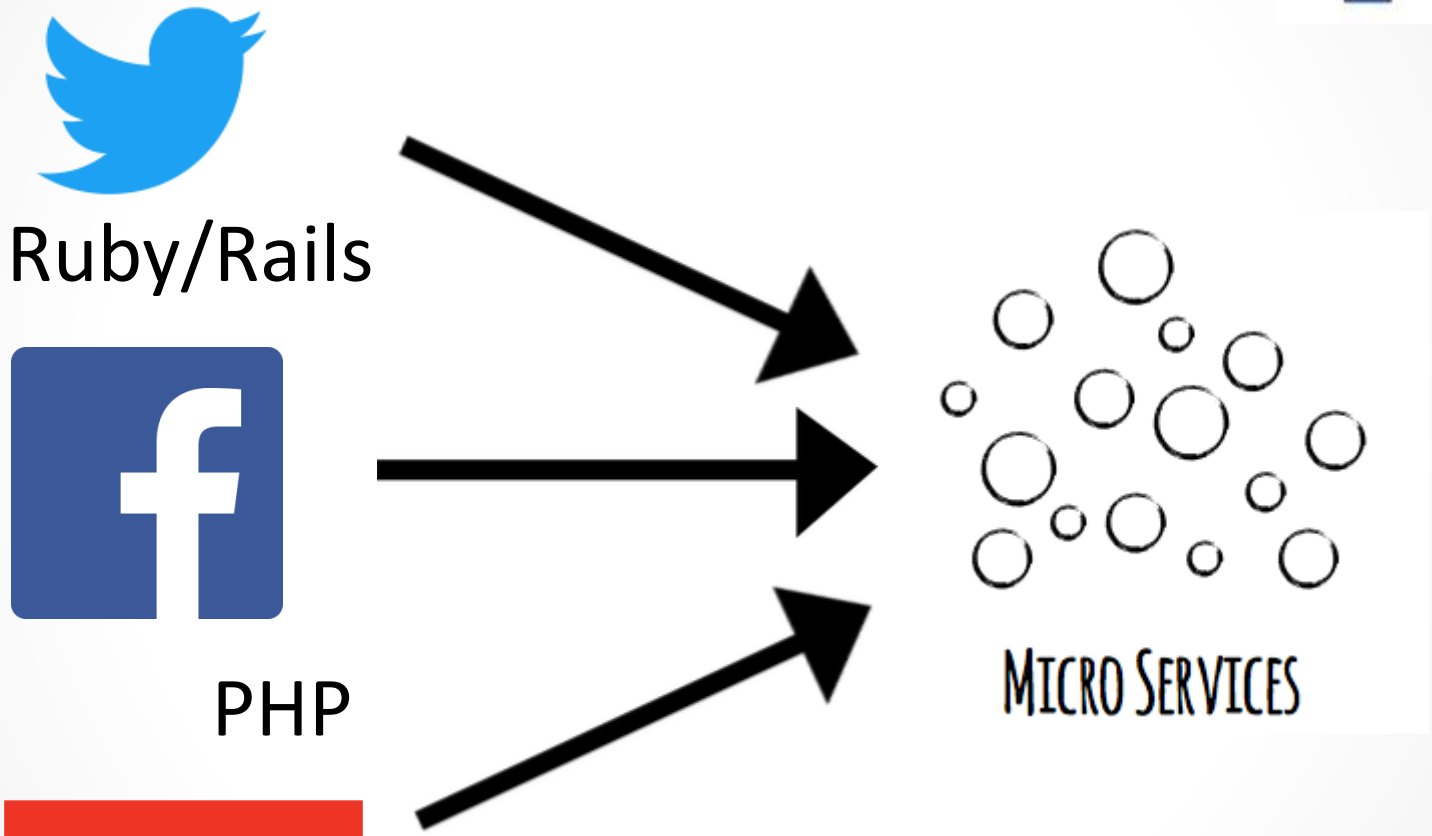
Challenges

- ✓ Services unavailability
- ✓ Advanced monitoring
- ✓ Cost of remote calls
- ✓ Eventual consistency (instead of ACID)
- ✓ Single feature is moved into few services
- ✓ Version management
- ✓ Dependency management
- ✓ Multiple data sources(databases)

Transition



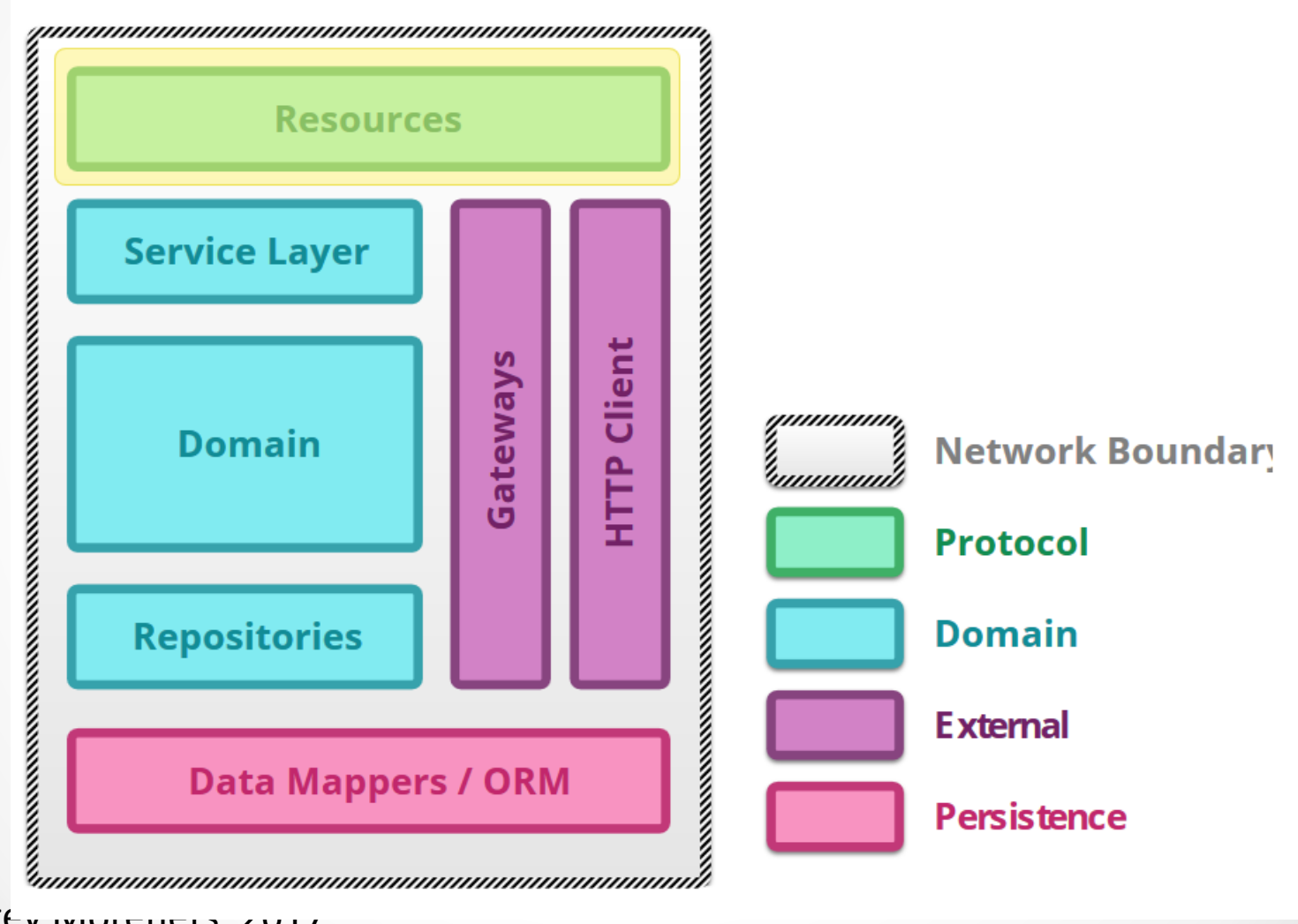
Transition



Transition

- ✓ Decomposition of an application
- ✓ Single responsibility principle. Single micro-service = single business feature
- ✓ Based on business functionality
- ✓ Bounded context

Microservice structure

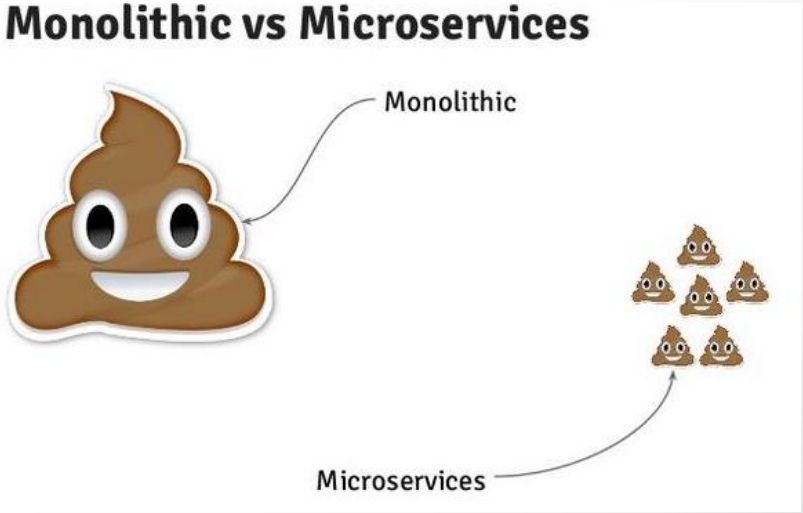


Polyglot persistence

Sergey Morenets, 2017

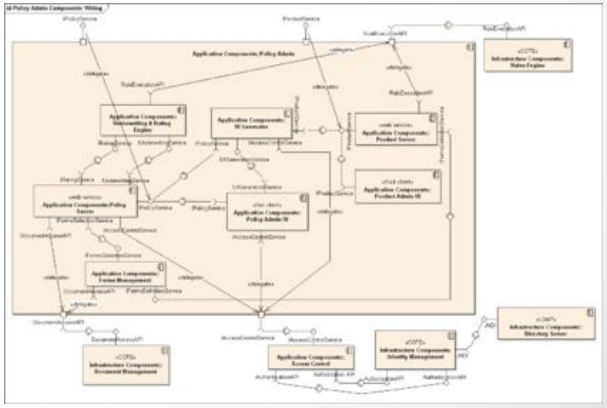
Anti-patterns

- ✓ Distributed monolith
- ✓ Single feature goes into all the micro-services

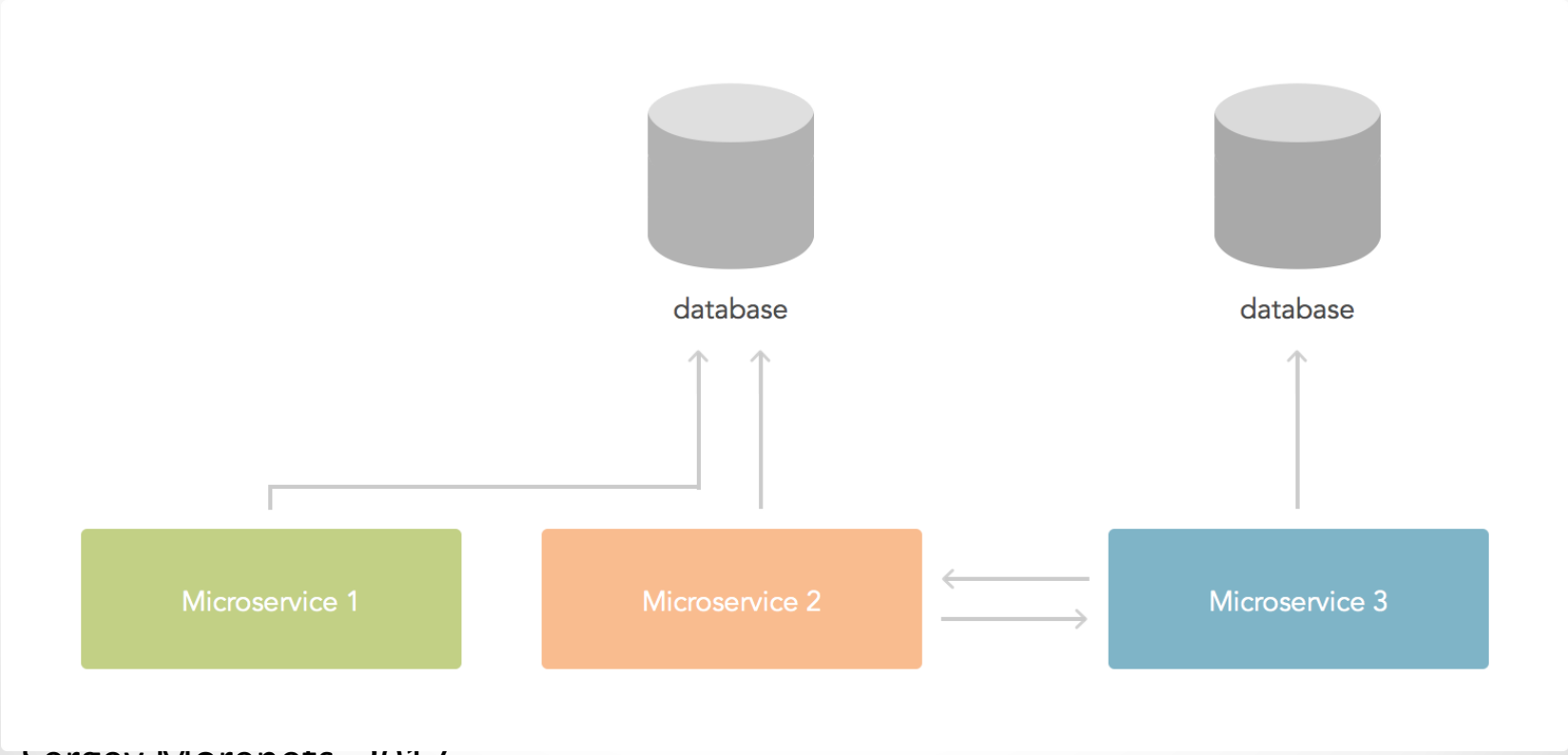


Anti-patterns

- ✓ Nano-service
- ✓ Huge performance/complexity/maintenance overhead

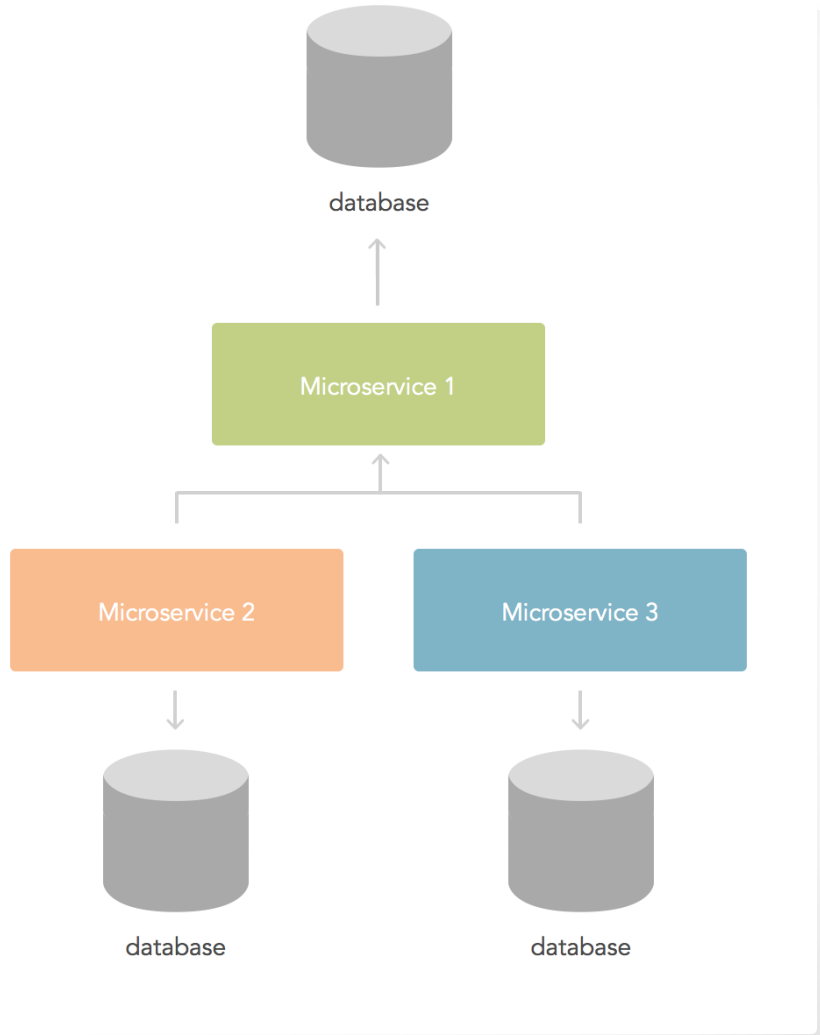


Anti-patterns



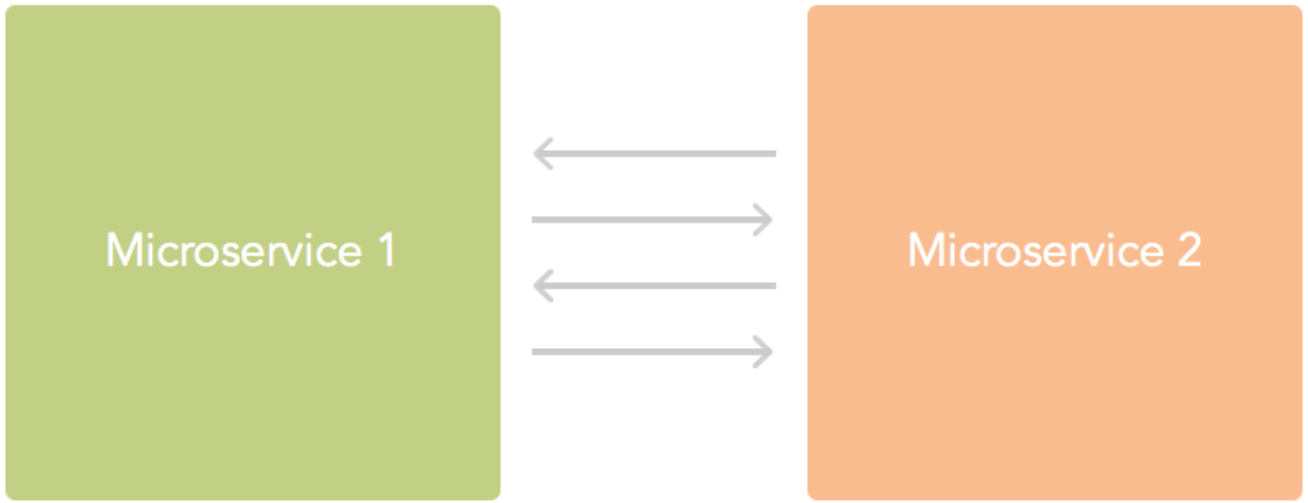
Sergey Morenets, 2017

Anti-patterns



- Sergey Morenets, 2017

Anti-patterns



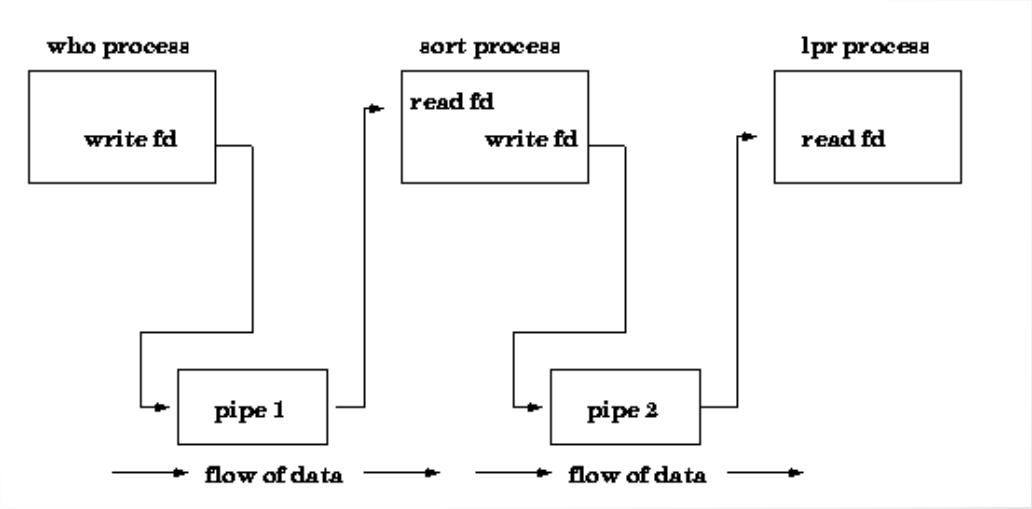
Two microservices sending lots of messages back and forth
candidates for turning into a single service

Transition

- ✓ Split domain model/**business logic**
- ✓ Split data model/persistence layer/**DB**
- ✓ Split/introduce **technologies**

Partitioning strategies

- By entity(Customer – Product – Order)
- By use case(Book – Buy – Search)
- Single Responsibility Principle
- Bounded context
- **Unix utilities**



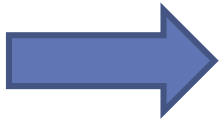
Enterprise model. Client

- ✓ Identifier
- ✓ Name
- ✓ Address
- ✓ Email/Phone
- ✓ Credit card number
- ✓ Expiration date
- ✓ Discount
- ✓ Purchases

Splitting data model

Client. **Purchase service**

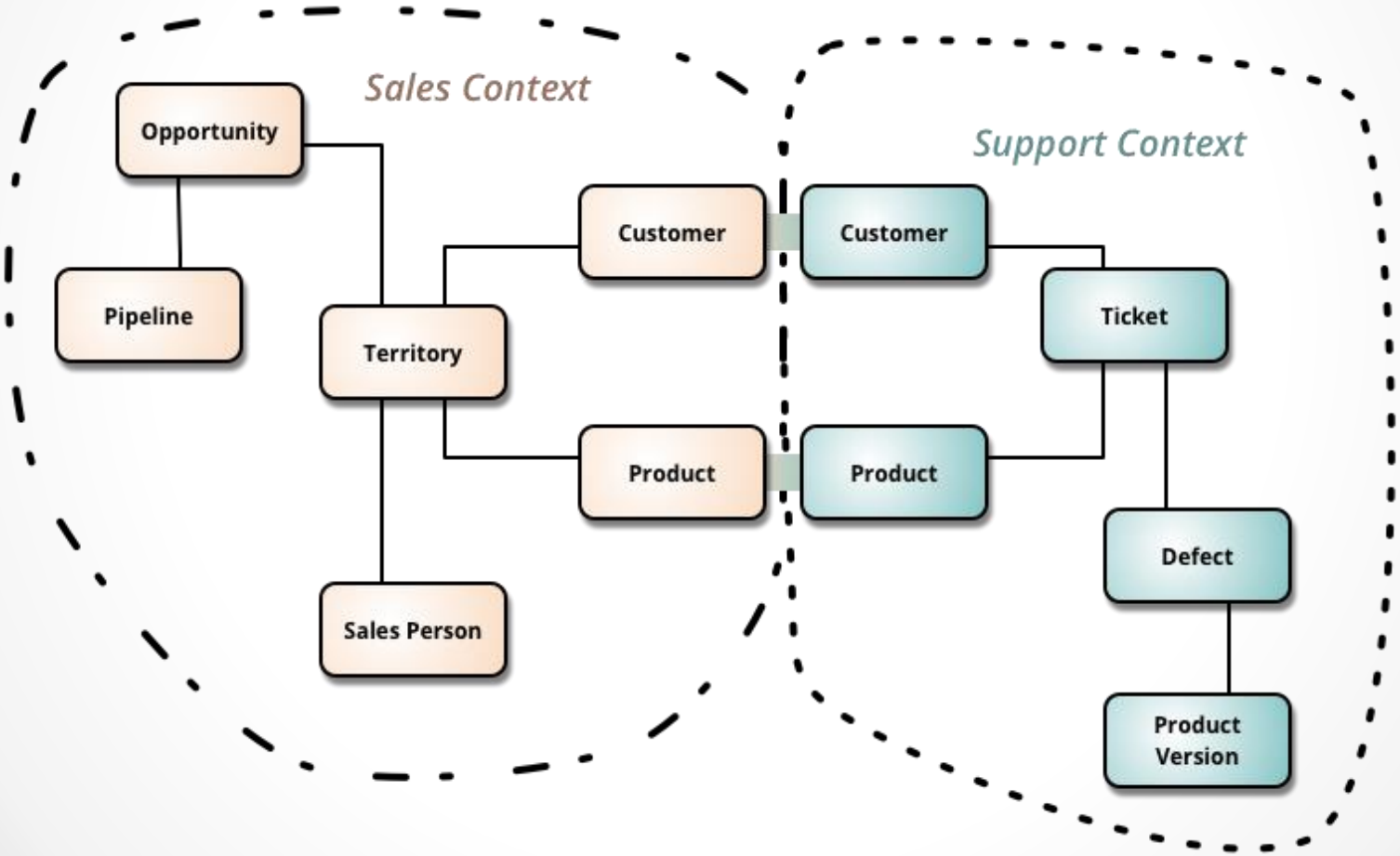
- ✓ Identifier
- ✓ Credit card number
- ✓ Expiration date
- ✓ Discount
- ✓ Purchases



Client. **Delivery service**

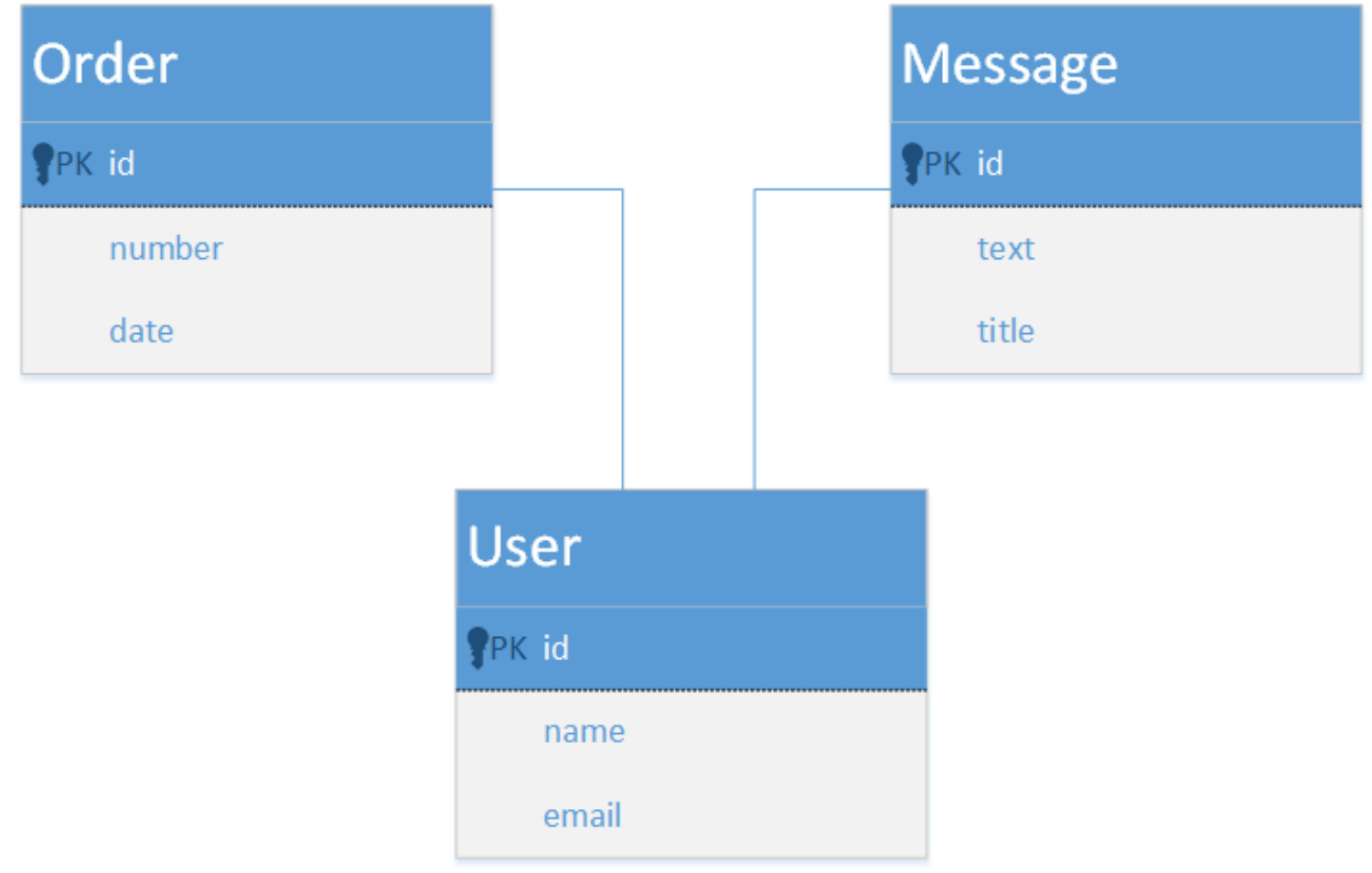
- ✓ Identifier
- ✓ Name
- ✓ Address
- ✓ Email/Phone

Bounded context



Sergey Morenets, 2017

Splitting data model



- Se

Task 3. Splitting monolith

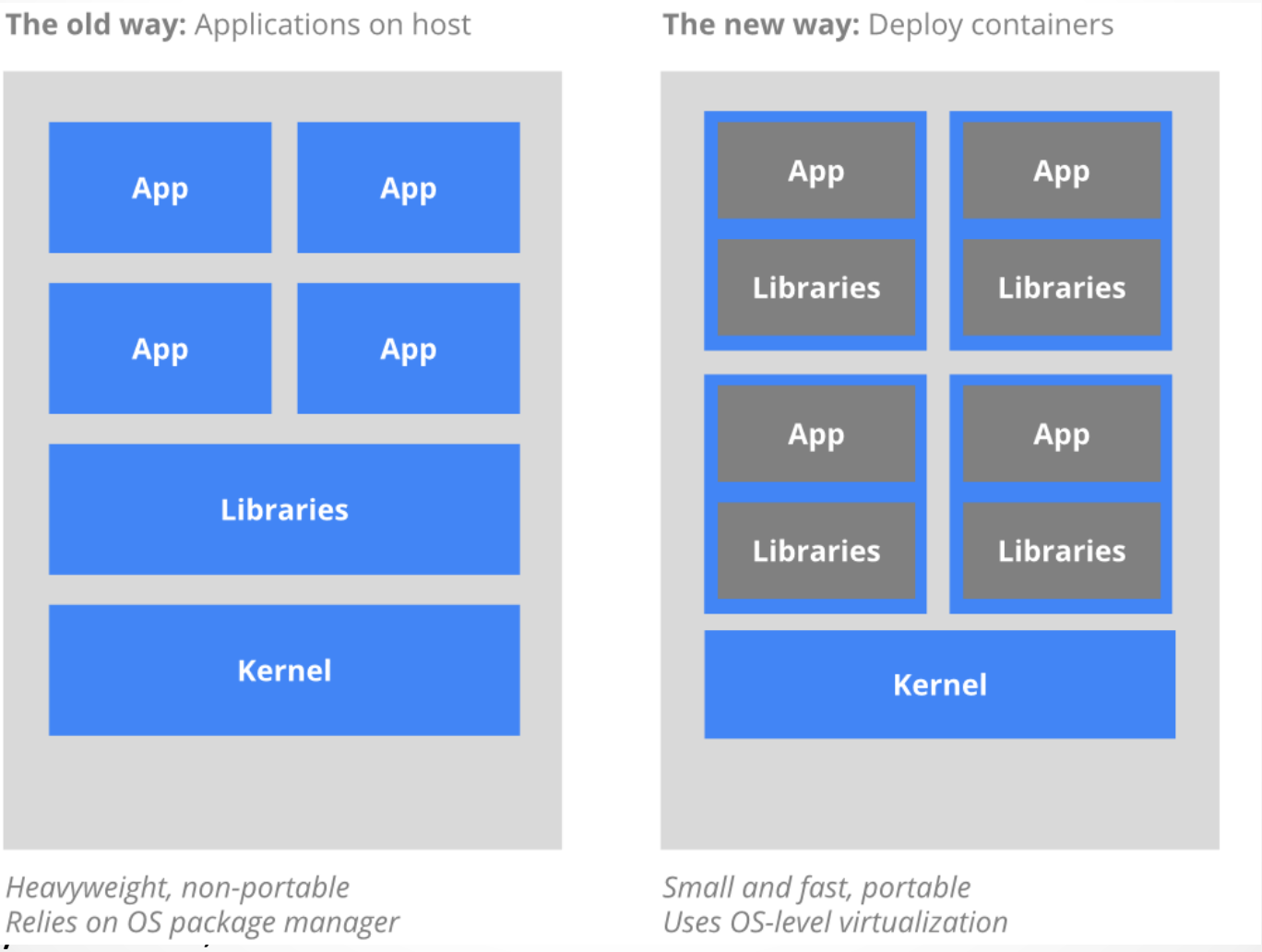
1. Review monolith application again. Try to convert it to micro-service architecture gradually.
2. Extract **functionality** that belong to different services and put into logical components of the projects (for example, different packages).
3. Split **domain model**
4. Split **services**
5. Split **DAO layer(repository)**
6. Split REST **controllers**

Questions

- ✓ How to deploy
- ✓ How service communicate with each other
- ✓ How client and service communicate
- ✓ How to split monolith into services
- ✓ How to handle failures

Infrastructure

Infrastructure

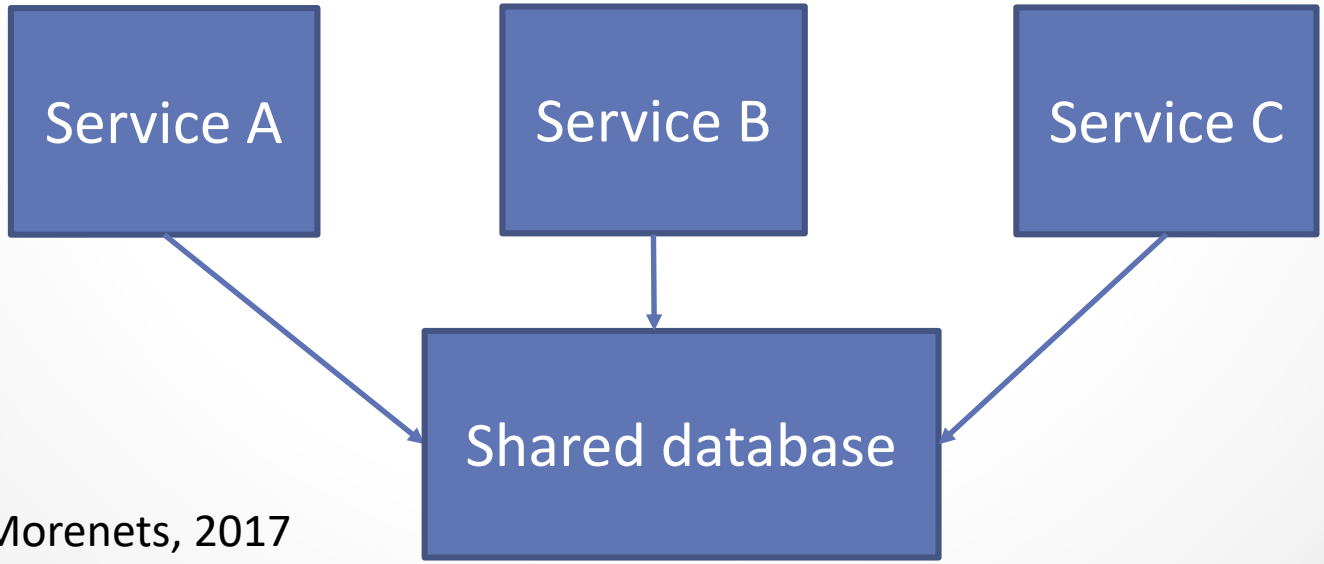


Data storage

- ✓ Shared database
- ✓ Database per service

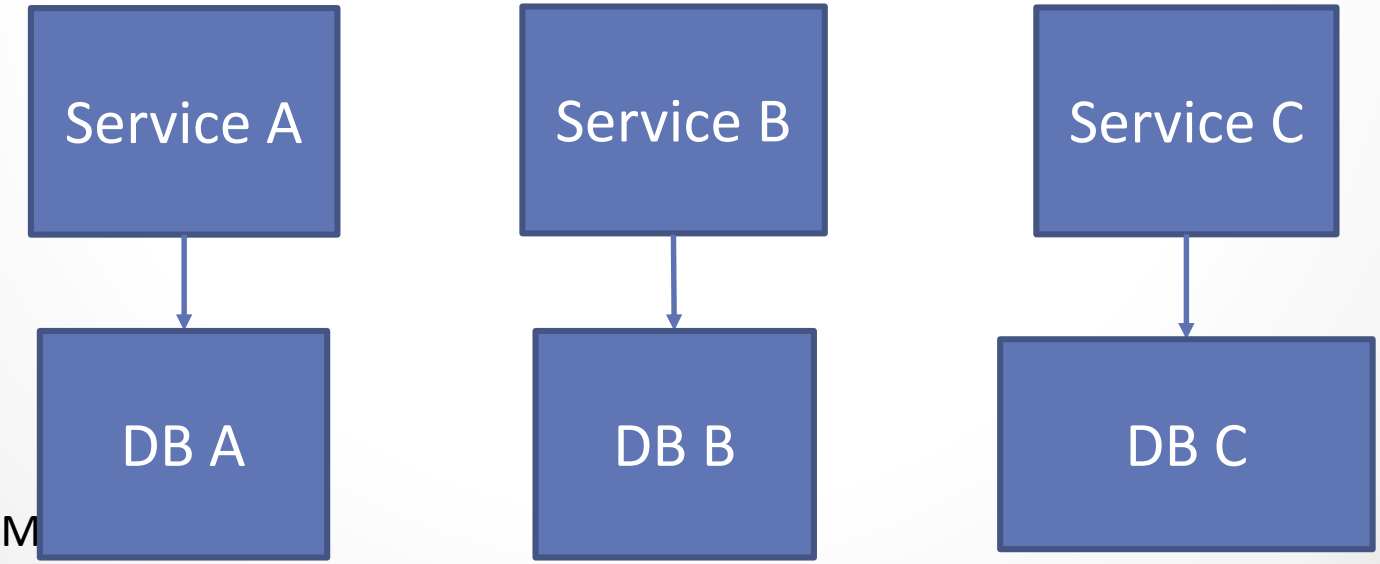
Shared database

- ✓ Easy to manage
- ✓ ACID (transactions)
- ✓ Tight coupling
- ✓ Database requirements depend on services(relational, NoSQL)
- ✓ Harder to change

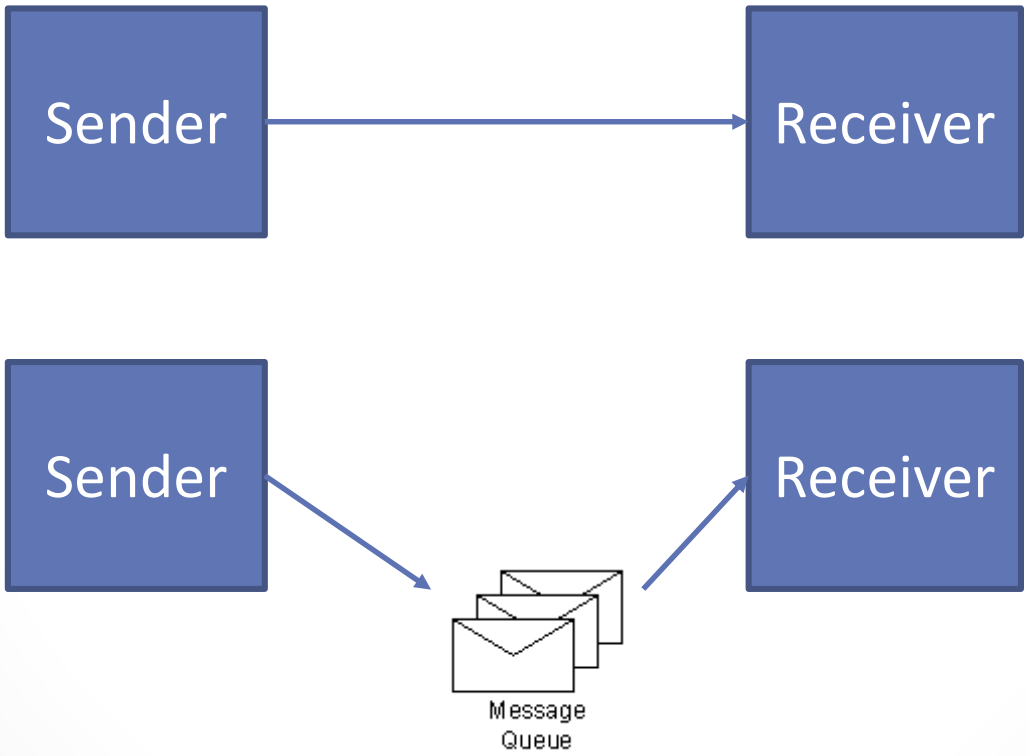


Database per service

- Private tables/schema/server
- Any kind of server: relational, NoSQL, text search, blob storage
- No transactions and join queries



Service communication. Strategies



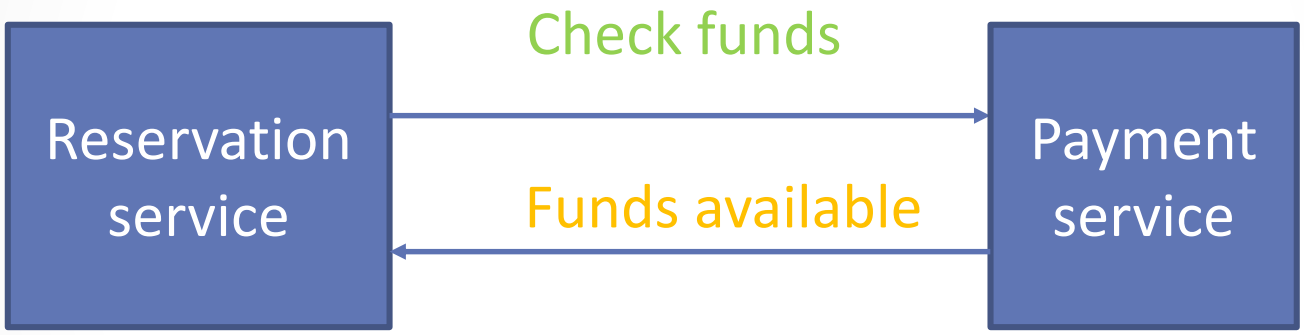
Event-driven architecture

- ✓ Service can publish an event when application state changes
- ✓ Service can subscribe to relevant events and respond to them
- ✓ Leads to eventual consistency
- ✓ Widely used in UI

Event-driven patterns

- ✓ Event notification
- ✓ Event-carried state transfer
- ✓ Event-sourcing
- ✓ Command-query responsibility separation(CQRS)

Event notification



Event notification

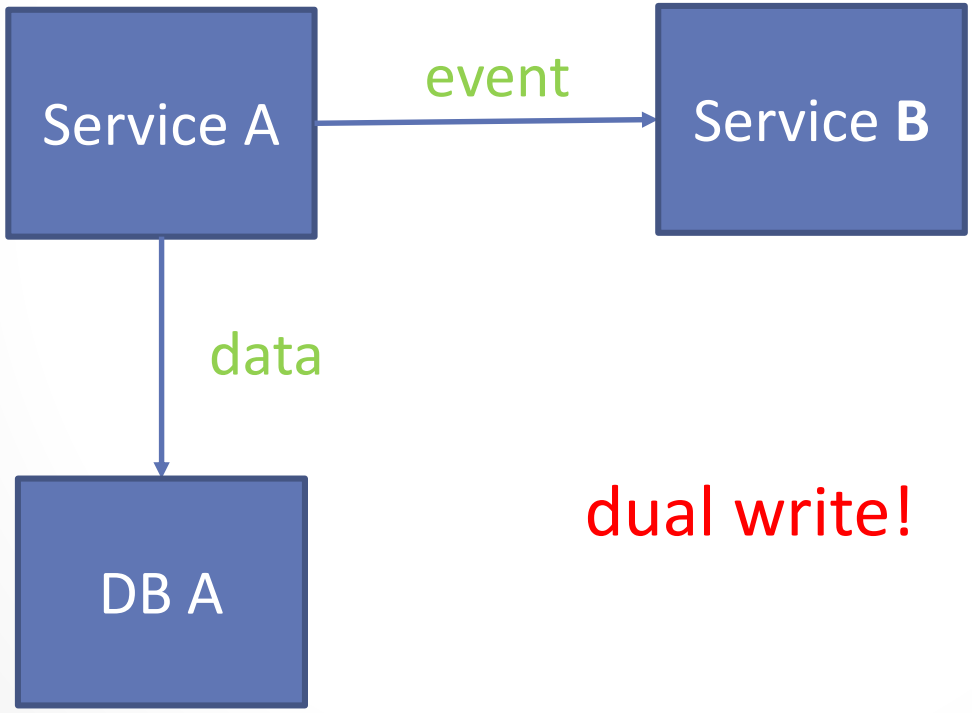

```
@Value
public class CustomerCreatedEvent {
    private int id;

    private String name;

    private String email;
}
```

```
@Value
public class CustomerCreatedEvent {
    private Customer customer;
}
```

Event-driven architecture



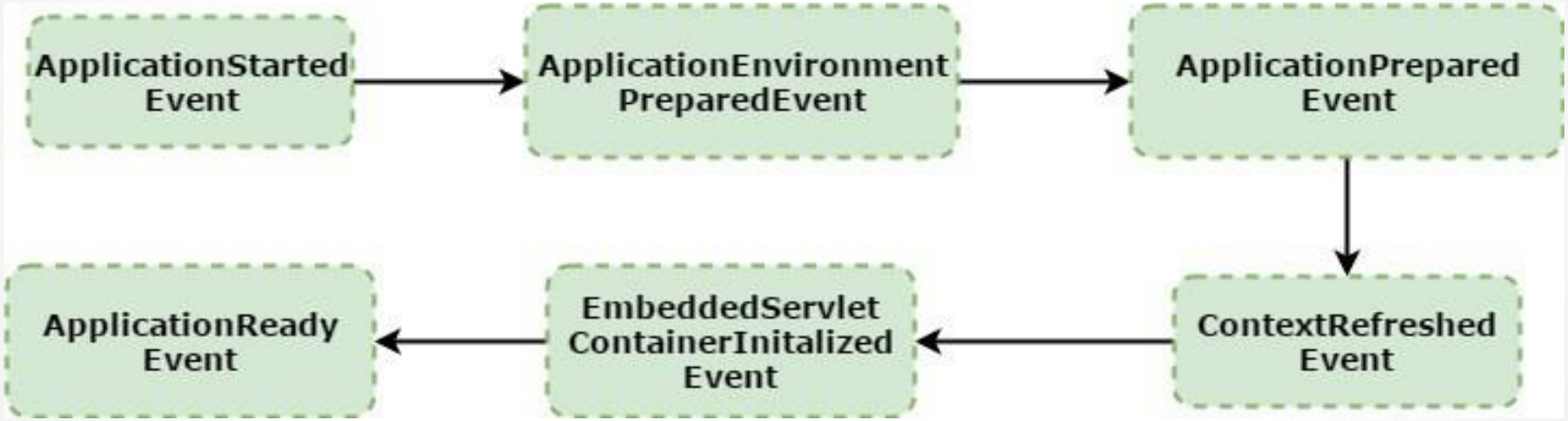
Task 4. Events

1. Try to identify all the events that occur in the whole application. These events will be transferred between micro-services in your project.
2. Create event classes. What will be their payload?

Task 5. Event bus

1. Create **event bus** component (you can use singleton pattern here) that will allow sending events and subscribing to new events. Try to create two implementations: synchronous and asynchronous. What is advantage of both implementations?
2. Try to use event bus to provide communication between order service and other services.
3. Update automation tests to test both synchronous and asynchronous implementation

Events and Spring Framework



Event listener


```
@Component
public class AppListener {

    @EventListener
    public void handleContextRefresh(
            ContextRefreshedEvent event) {

    }

    @EventListener
    public void globalHandler(ApplicationEvent event) {

    }
```

Event publisher


```
@Component
public class AppPublisher {

    private final ApplicationEventPublisher publisher;

    public AppPublisher(ApplicationEventPublisher publisher){
        this.publisher = publisher;
    }

    public void createEvent() {
        this.publisher.publishEvent(new AppEvent(this));
    }

    public static class AppEvent extends ApplicationEvent {

        public AppEvent(Object source) {
            super(source);
        }}
```

Task 6. Event bus and Spring

1. Create new **event bus** implementation based on Spring events.
2. Let your event classes extend **ApplicationEvent** class.
3. Use **ApplicationEventPublisher** class to send events and **@EventListener** annotation to subscribe to events.

Task 7. Microservice structure

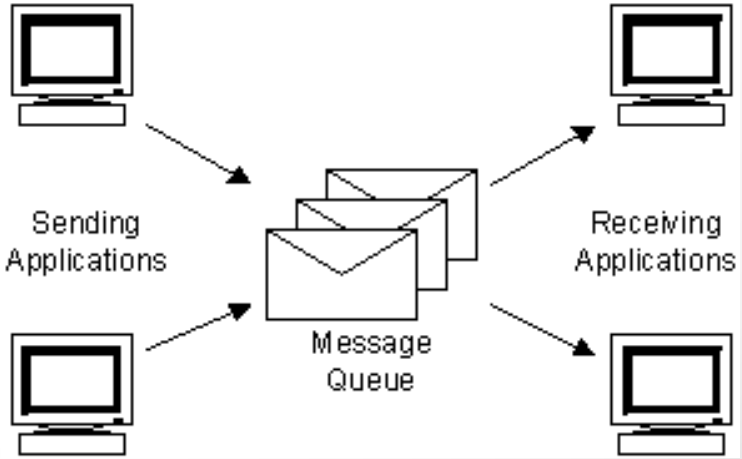
1. Try to create several sub-projects (modules) of main project and put each microservice into separate sub-project.
2. How will you handle shared classes that belongs to different microservices (for example, **events**)?
3. Update automation tests

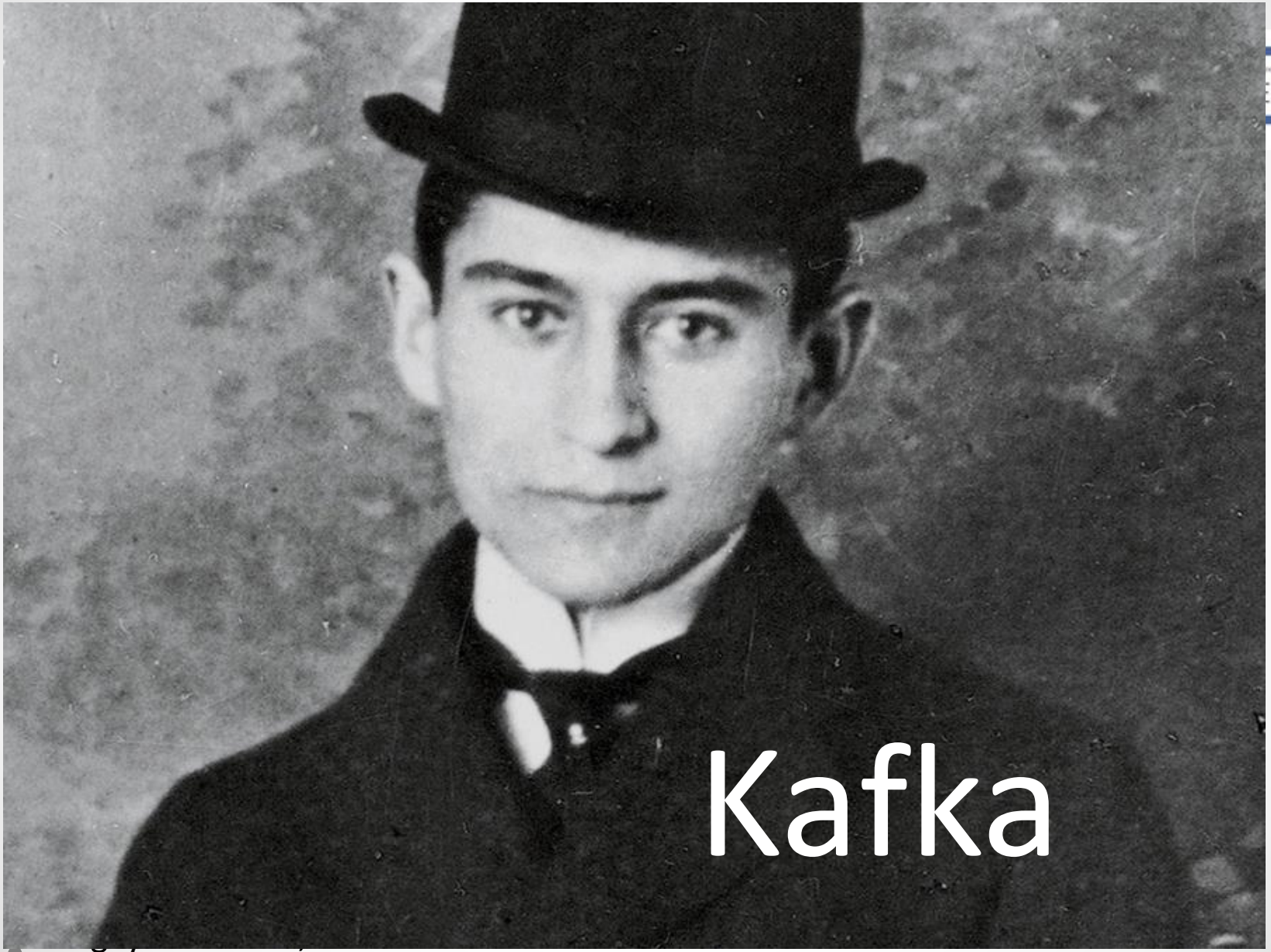
Message queue advantage

- ✓ Breaks tight coupling
- ✓ Introduce multiple independent consumers
- ✓ Avoid availability issues

Message queue attributes

- ✓ Delivery
- ✓ Transactions (group messages together)
- ✓ Durability
- ✓ Sync/Async messaging
- ✓ High availability
- ✓ Load balancing




Kafka

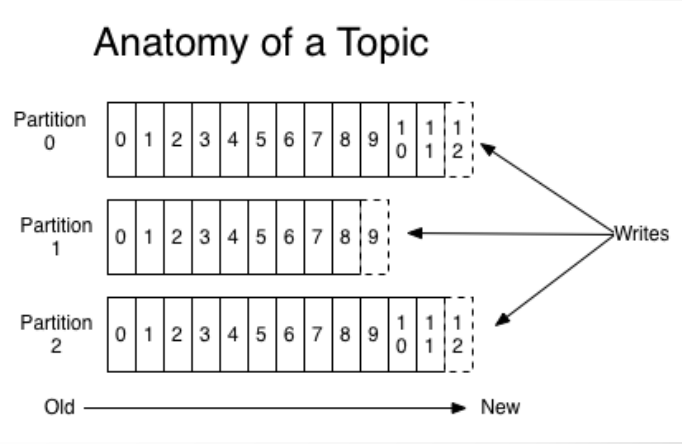
Apache Kafka

- ✓ Distributed messaging system
- ✓ High availability, resilient to node failures, automatic recovery
- ✓ Developed by LinkedIn in Java/Scala
- ✓ Open-sourced in 2011
- ✓ Publisher/subscriber message queue
- ✓ Uses **ZooKeeper** for cluster membership/ routing
- ✓ Competes with RabbitMQ/ActiveMQ

ets

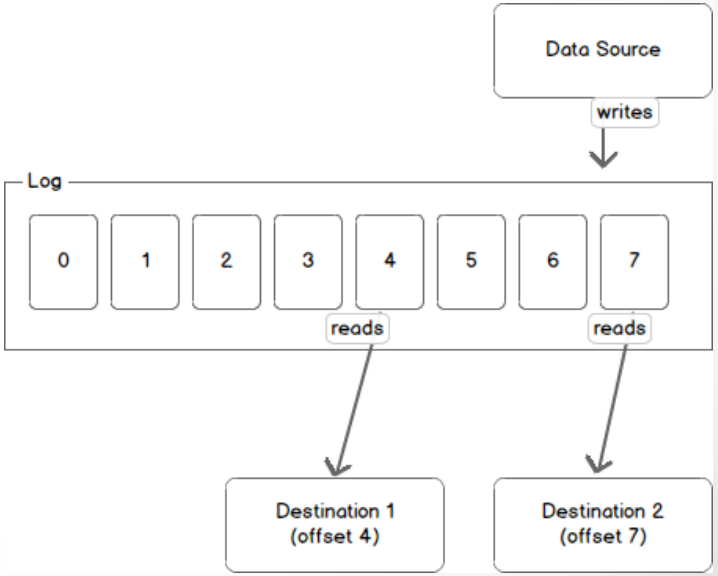
Apache Kafka. Messaging

- All Kafka messages are grouped into topics
- Topics are divided into partitions to parallelize topic access for multiple consumers
- Producers publish messages to the topic
- Subscribers subscribe to one or more topic
- Each message has unique identifier (offset)

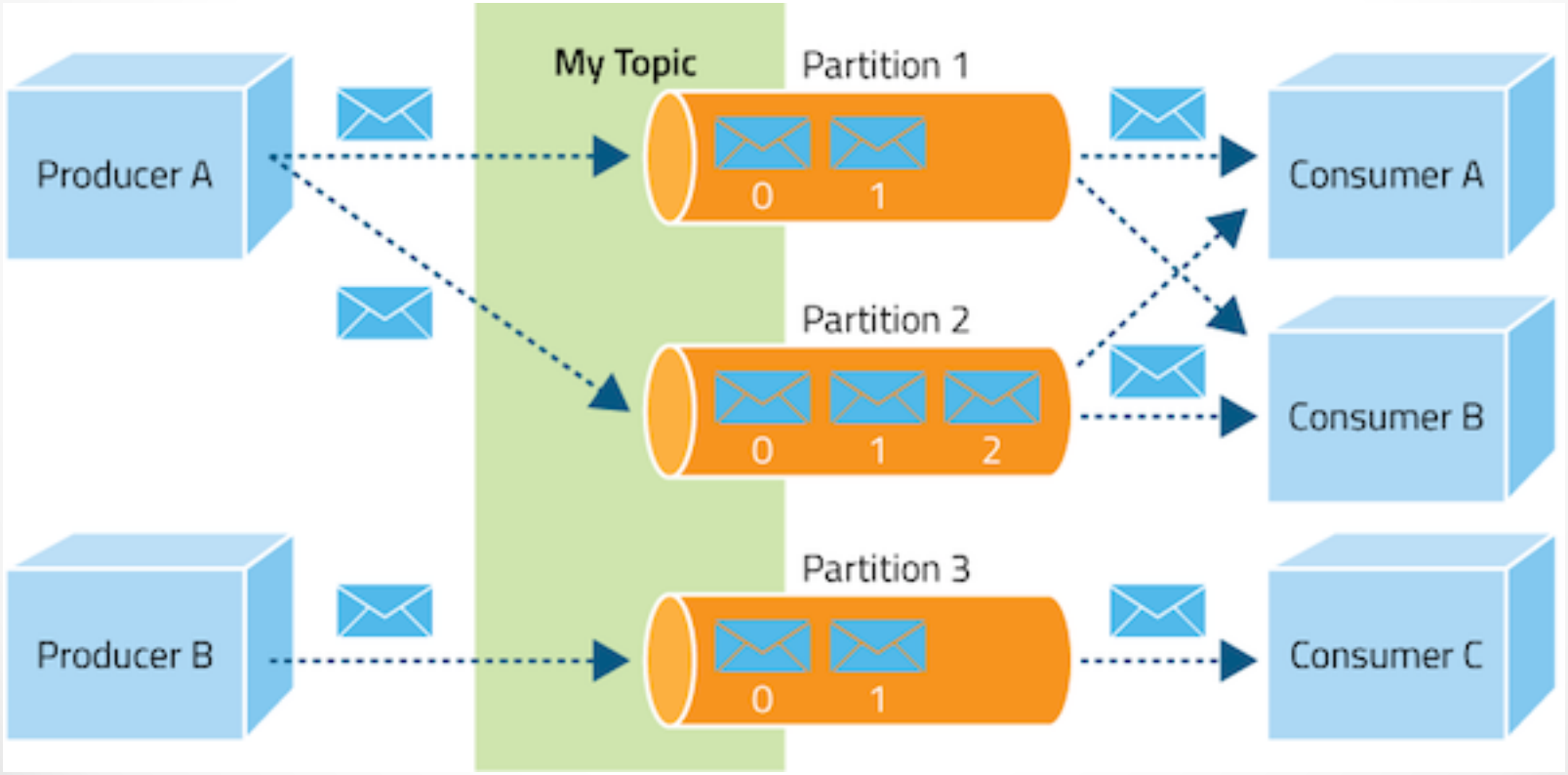


Apache Kafka. Messaging

- Each message is key/value pair. Key is used to determine partition
- Consumer can read only recent messages or from any offset
- Consumer can read from any partition
- Partition is like a log

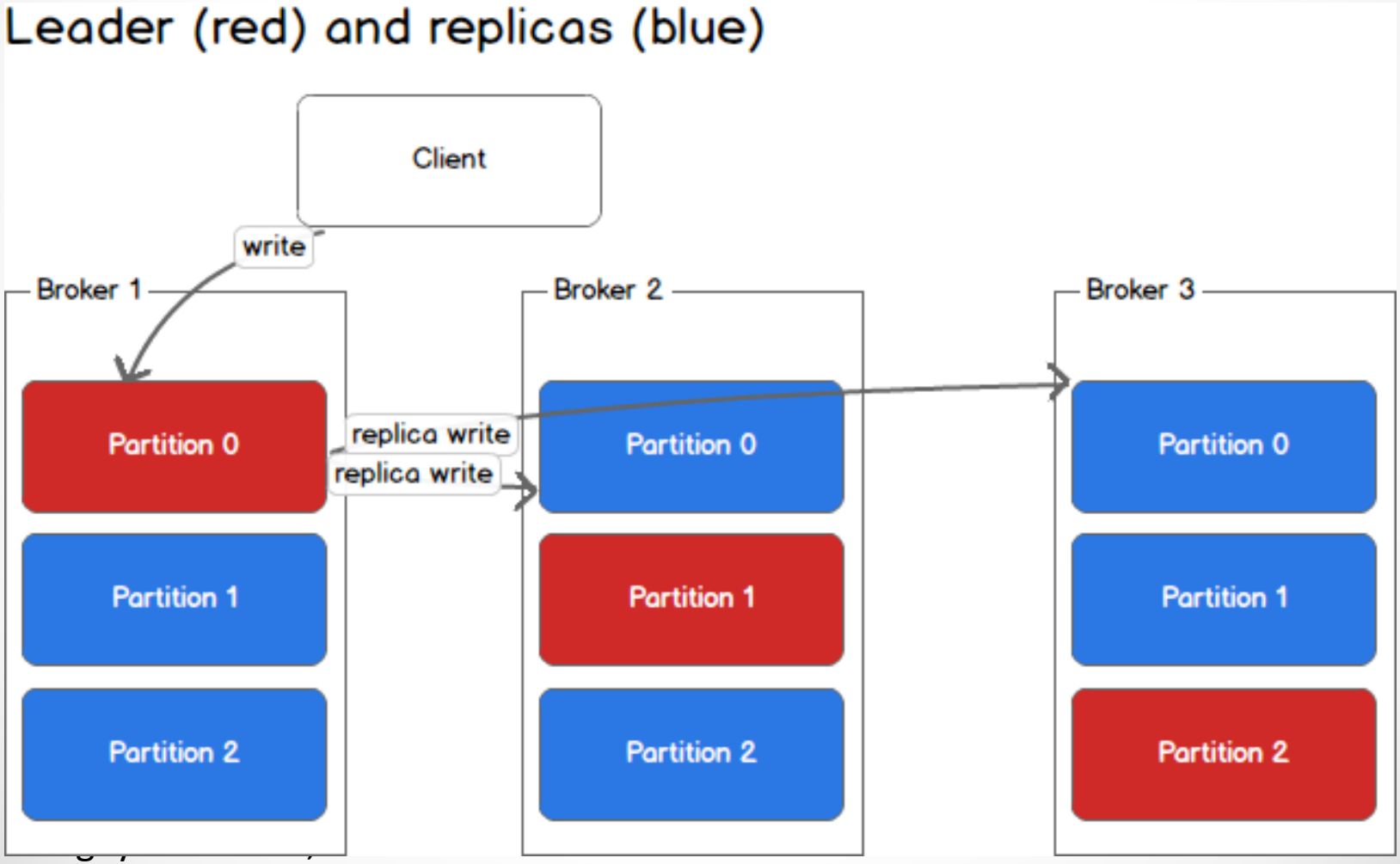


Apache Kafka. Messaging

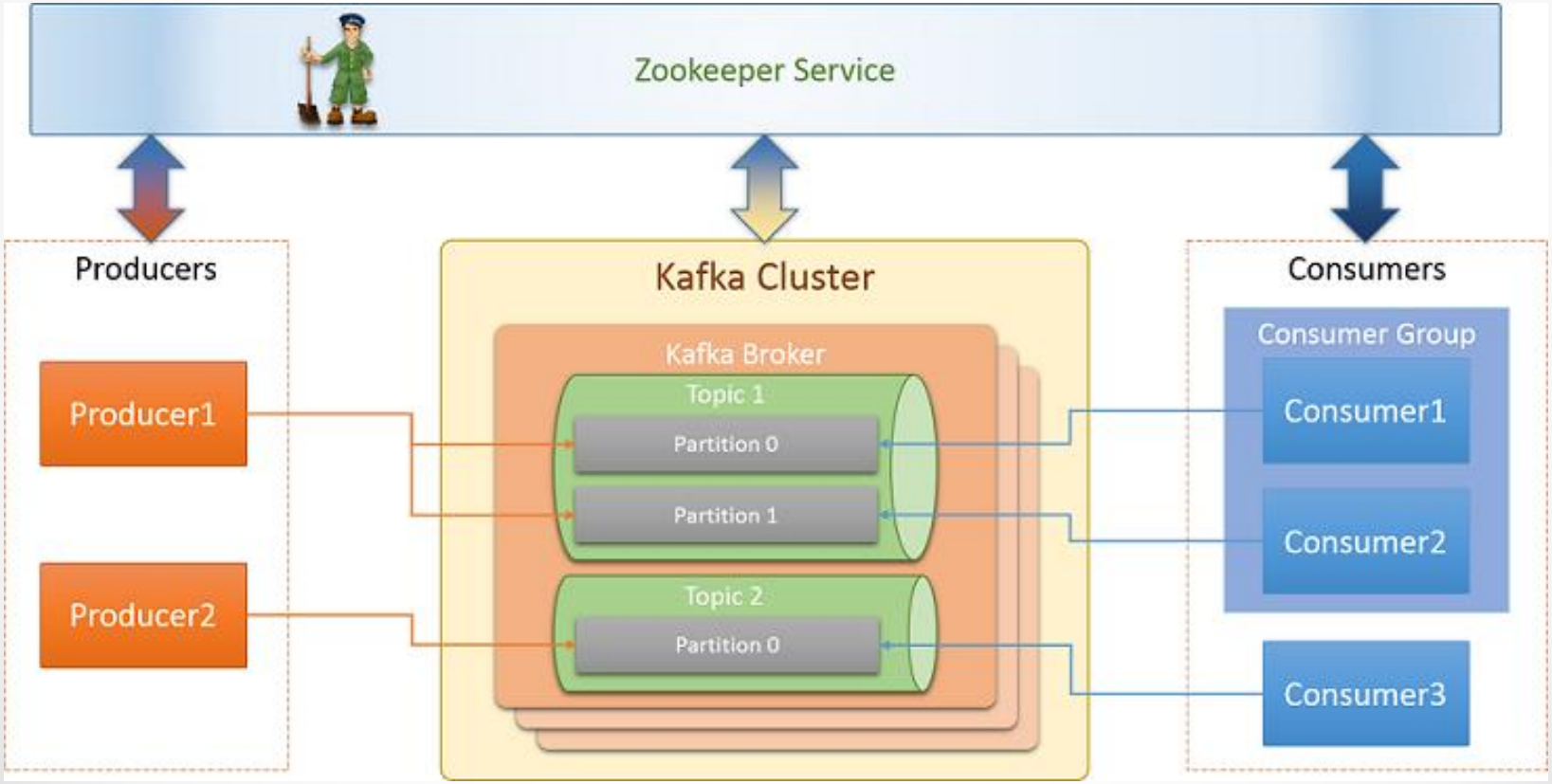


Sergey Morenets, 2017

Apache Kafka. Brokers



Apache Kafka. Clustering



Kafka. Consistency and availability

- ✓ Messages will be appended to topic partition in the order they were sent
- ✓ Single consumer will read the messages in the order they are put in the log
- ✓ Message is committed when all replicas have got copy of this message
- ✓ Message cannot be lost if at least one replica is alive

Task 8. Apache Kafka

1. Download and install **Apache Kafka**
2. Review **server.properties** and **zookeeper.properties** in c:/Kafka/config folder
3. Run Apache **Zookeeper.** Review console logs.
4. Run Apache **Kafka**
5. Run command *“kafka-topics.bat --list --zookeeper localhost:2181”* to view all existing topics.

Spring Kafka

- Introduced template abstraction for messaging operations
- Supports message-driven approach with @KafkaListener operation
- Requires Spring Framework 5, Spring Integration 3.0 and Kafka client 0.11

Spring Kafka. Dependencies


```
<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka</artifactId>
    <version>2.0.0.RELEASE</version>
</dependency>
```

Spring Kafka. Configuration


```java
@Configuration
@EnableKafka
public class KafkaConfiguration {

    @KafkaListener(topics = "orders")
    public void listen(ConsumerRecord<?, ?> record) {
        System.out.println(record.offset());
        System.out.println(record.partition());
        System.out.println(record.topic());
        System.out.println(record.toString());
    }
```

src/main/resources/application.yml

```yaml
spring:
  kafka:
    consumer:
      group-id: app
      auto-offset-reset: earliest
```

KafkaTemplate

```
@Autowired
private KafkaTemplate<String, String> template;

@PostMapping
public void send() {
    template.send("orders", "message");
    template.send("orders", "key", "message_with_key");
    template.send("orders", 0, "key2", "message2_with_key");

    template.sendDefault("key", "message");
}
```

Topic name

Message key

Message body

Task 9. Spring Kafka

1. Add new dependency to your project:
2. Add Kafka configuration class; put **@EnableKafka** annotation on it.
3. Update (or create) *src/main/resources/application.yml* file and put group-id and auto-offset-reset properties.
4. Create class **KafkaListener** that will listens for messages in **library** topic and print it to the console. Put **@KafkaListener** annotation on it.

✓ Sergey Morenets, sergey.morenets@gmail.com