Microservices development

July, 20-21 2019

- Sergey Morenets, 2019

DEVELOPER 14 YEARS

TRAINER 6 YEARS

WRITER 4 BOOKS

- Sergey Morenets, 2019

FOUNDER

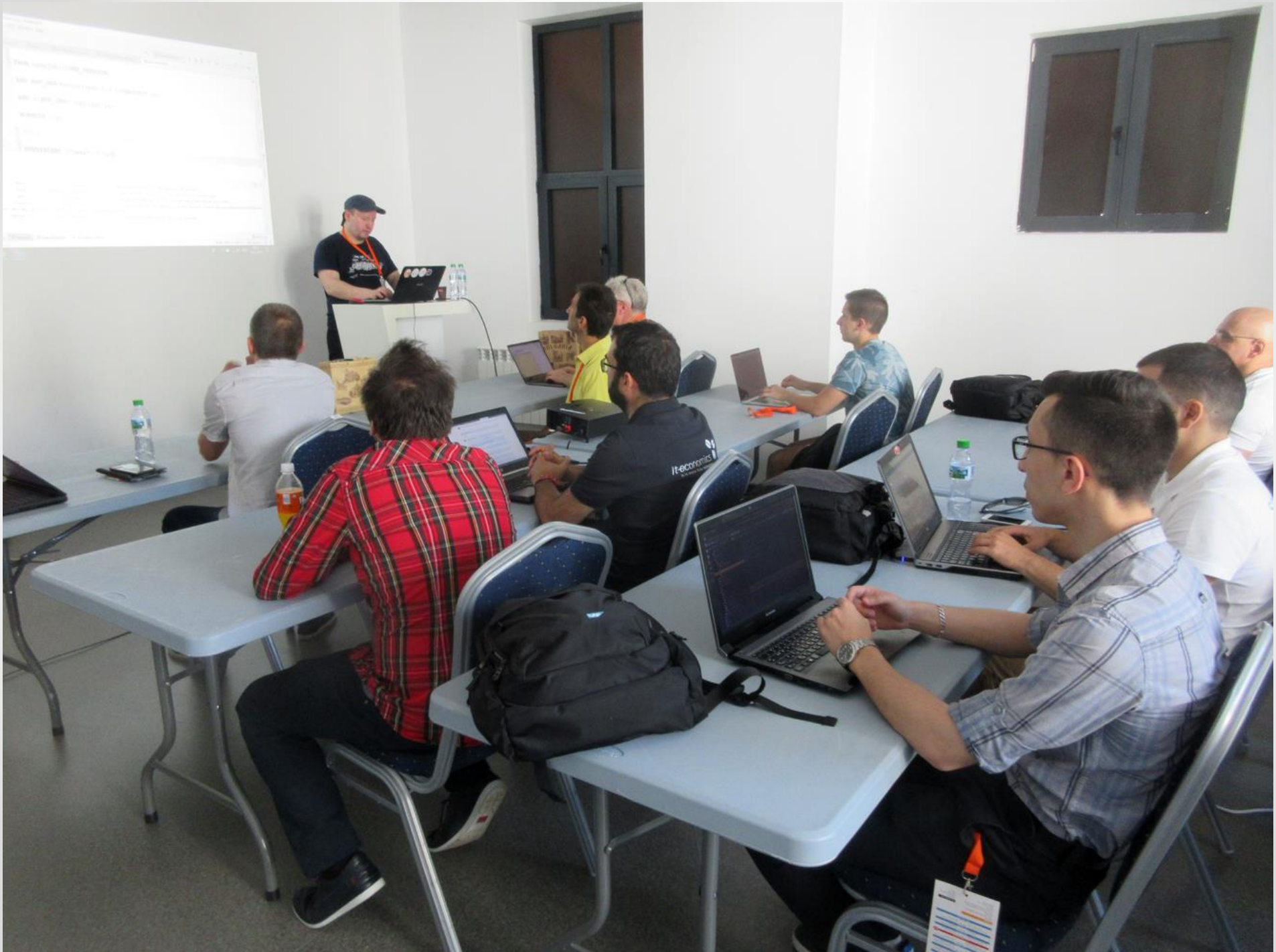
ITSimulator

SPEAKER

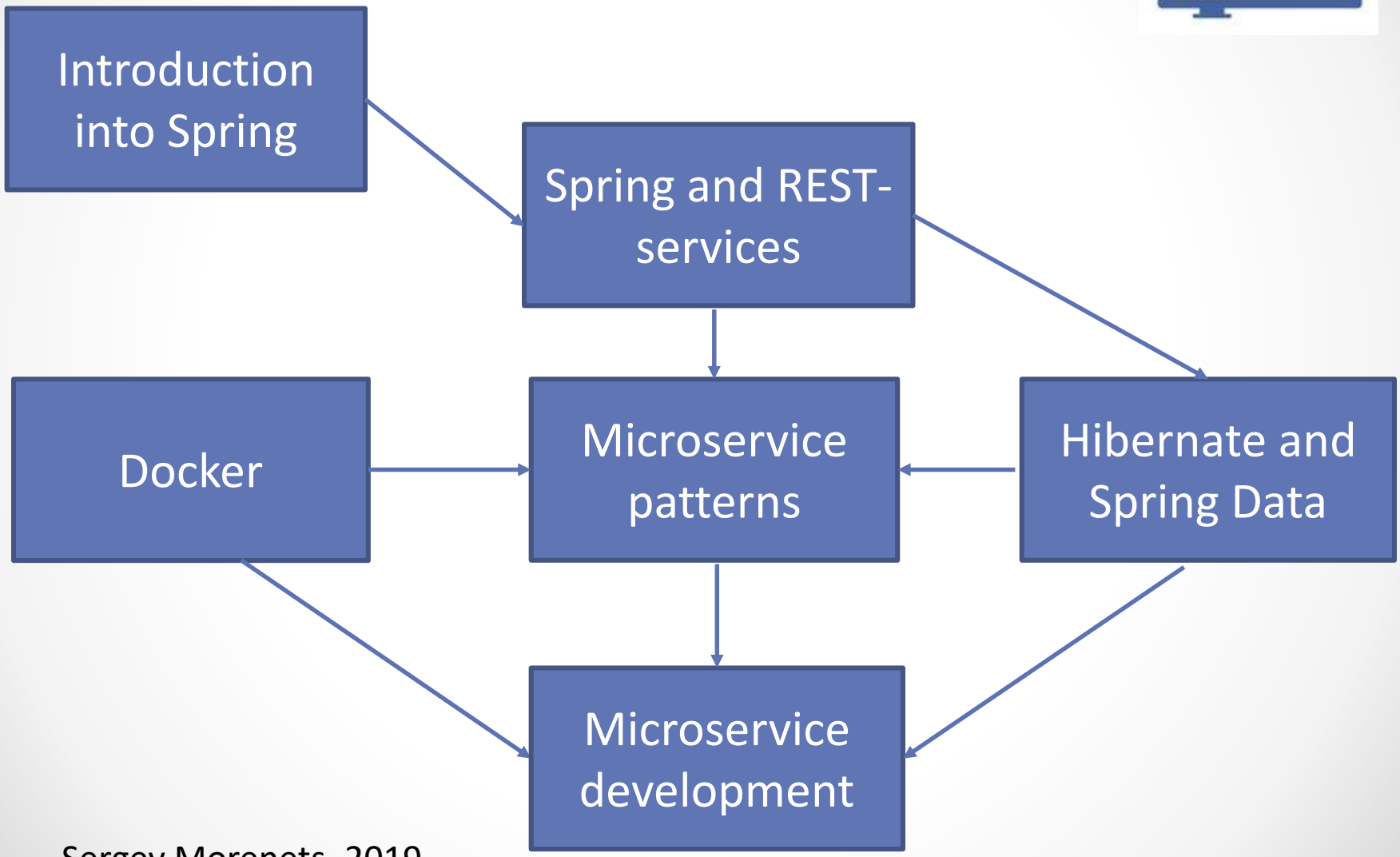
Java
frameworks
day

- Sergey Morenets, 2


@sergey-morenets
DEVOXX
POLAND


it-economics

Traning roadmap

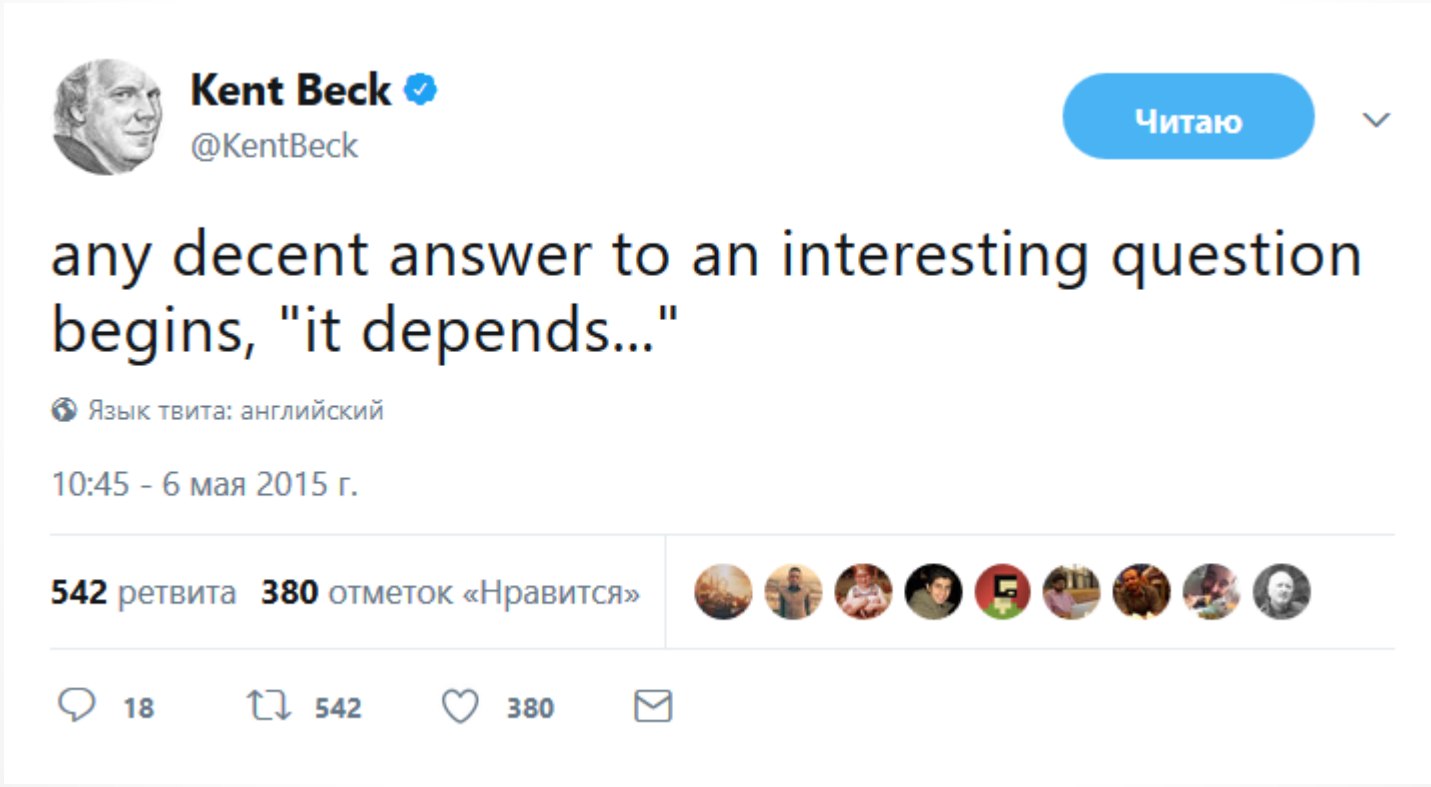


- Sergey Morenets, 2019

Agenda

- Sergey Morenets, 2019

Agenda



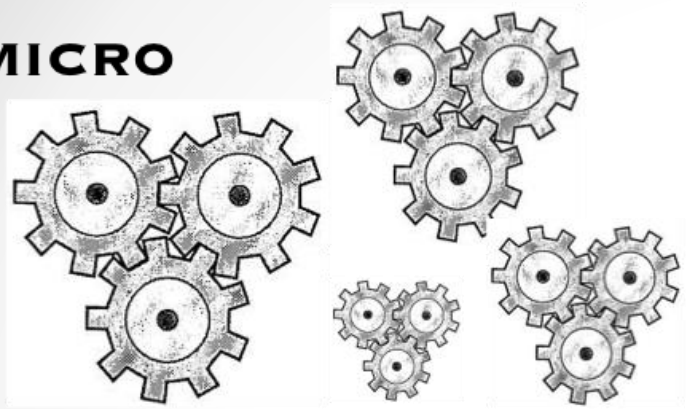
- Sergey Morenets, 2019

Agenda

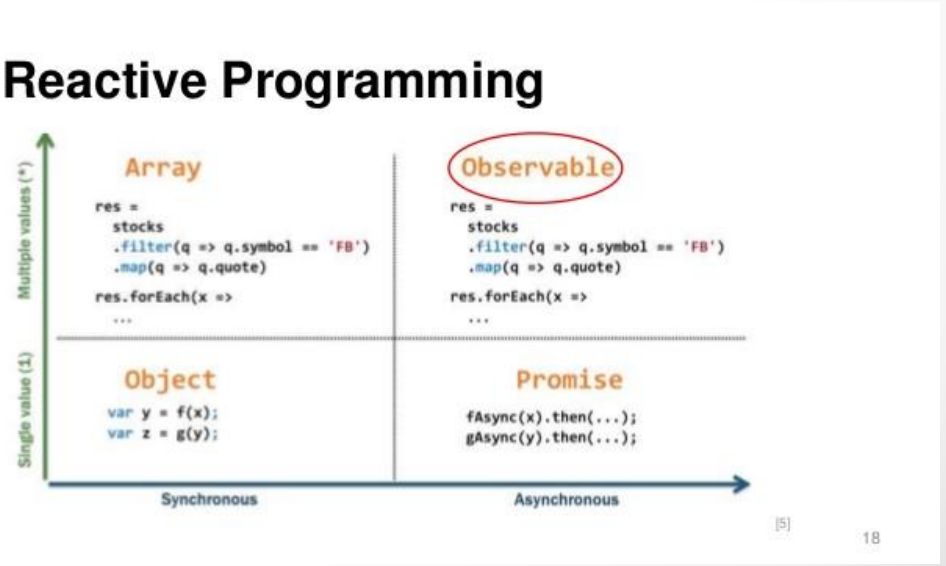
- ✓ Spring Framework/Spring Boot infrastructure
- ✓ Complexity of monolith applications
- ✓ Micro-service architecture. Pro and cons
- ✓ Event-Driven architecture & patterns
- ✓ Event sourcing
- ✓ CQRS
- ✓ Redis
- ✓ Apache Kafka
- ✓ Spring Data/MongoDB
- ✓ Testing

- Sergey Morenets, 2019

MICRO

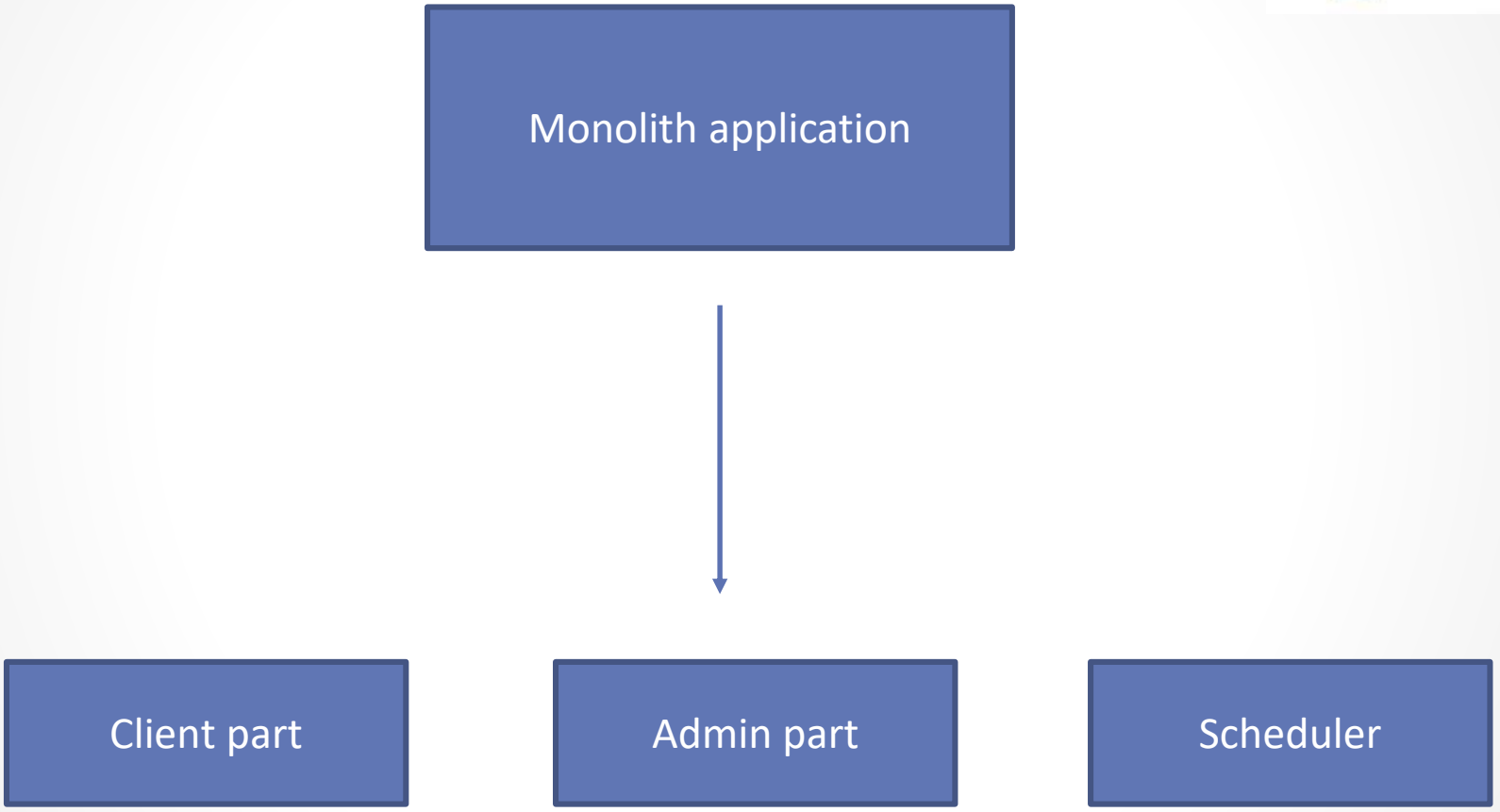


SERVICES



- Sergey Morenets, 2019

Story with happy-end

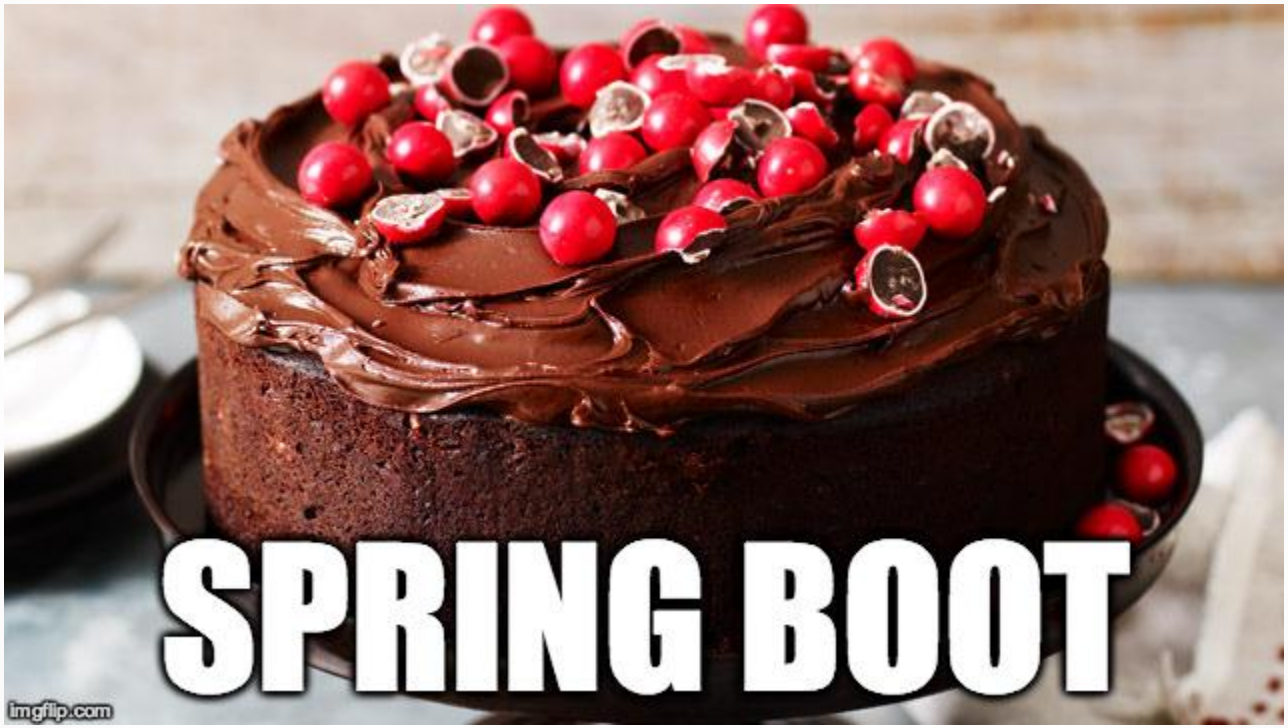


- Sergey Morenets, 2019

Spring Framework

- Analog to **EJB**
- Developed by **SpringSource**(now Pivotal)
- First milestone release in 2003
- **1.0** released in 2004 with Hibernate support
- **2.0** released in 2006
- **2.5** introduced annotations in 2007
- **3.0** released in 2009
- **4.0** supports Java 8, Groovy 2 and Java EE7
- **5.0** includes **Kotlin** support, Reactive Streams **(**Spring **WebFlux) and Junit 5/Java 9** support

- Sergey Morenets, 2019



- Sergey Morenets, 2019

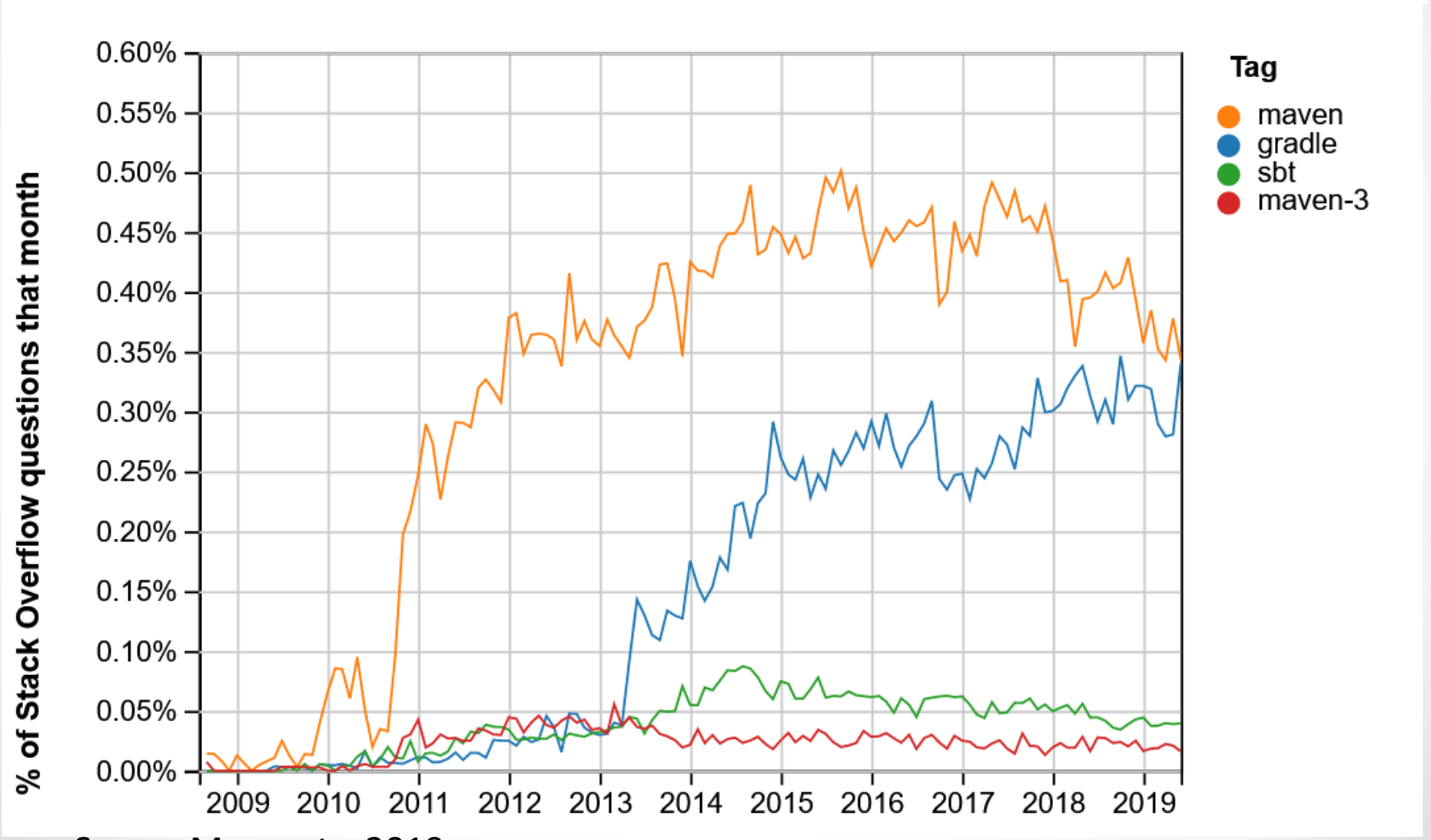
Spring Boot

- Stand-alone Spring applications
- Embed Tomcat, Jetty, Undertow or Netty
- Automatically Spring configuration
- Convention-over-configuration
- Zero XML configuration
- POM starters
- Actuator
- Dev tools

Build management

- Sergey Morenets, 2019

Maven vs Gradle vs Sbt



- Sergey Morenets, 2019

Dependency management. Maven

- ✓ Default compiler level/source encoding
- ✓ Common dependencies management
- ✓ Resource filtering and plugin configurations

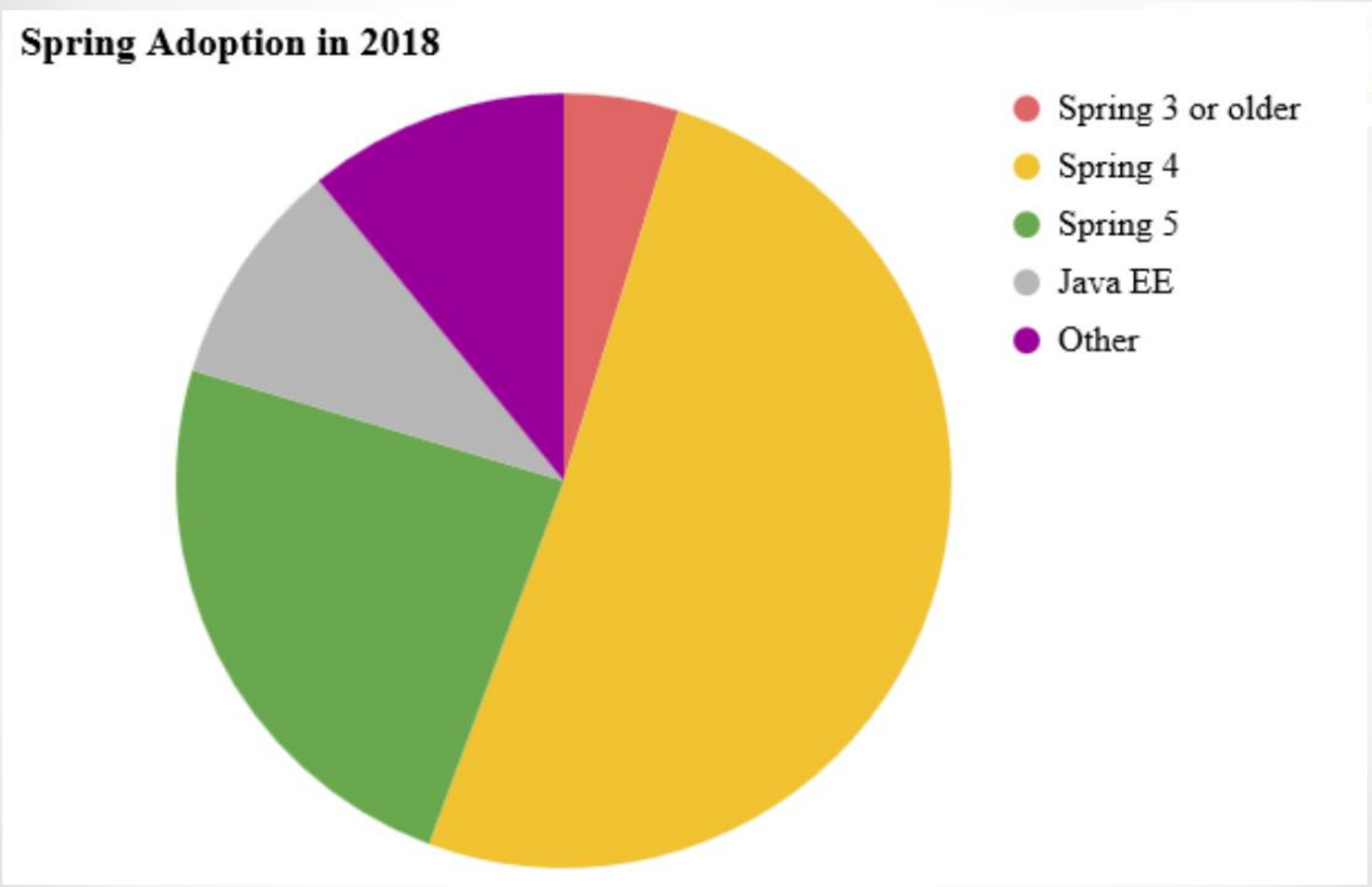
```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.6.RELEASE</version>
</parent>
```

REST service


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

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

IDE

- Sergey Morenets, 2019



- Sergey Morenets, 2019

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 '2.0.0.RELEASE'
}
```

Built management tasks

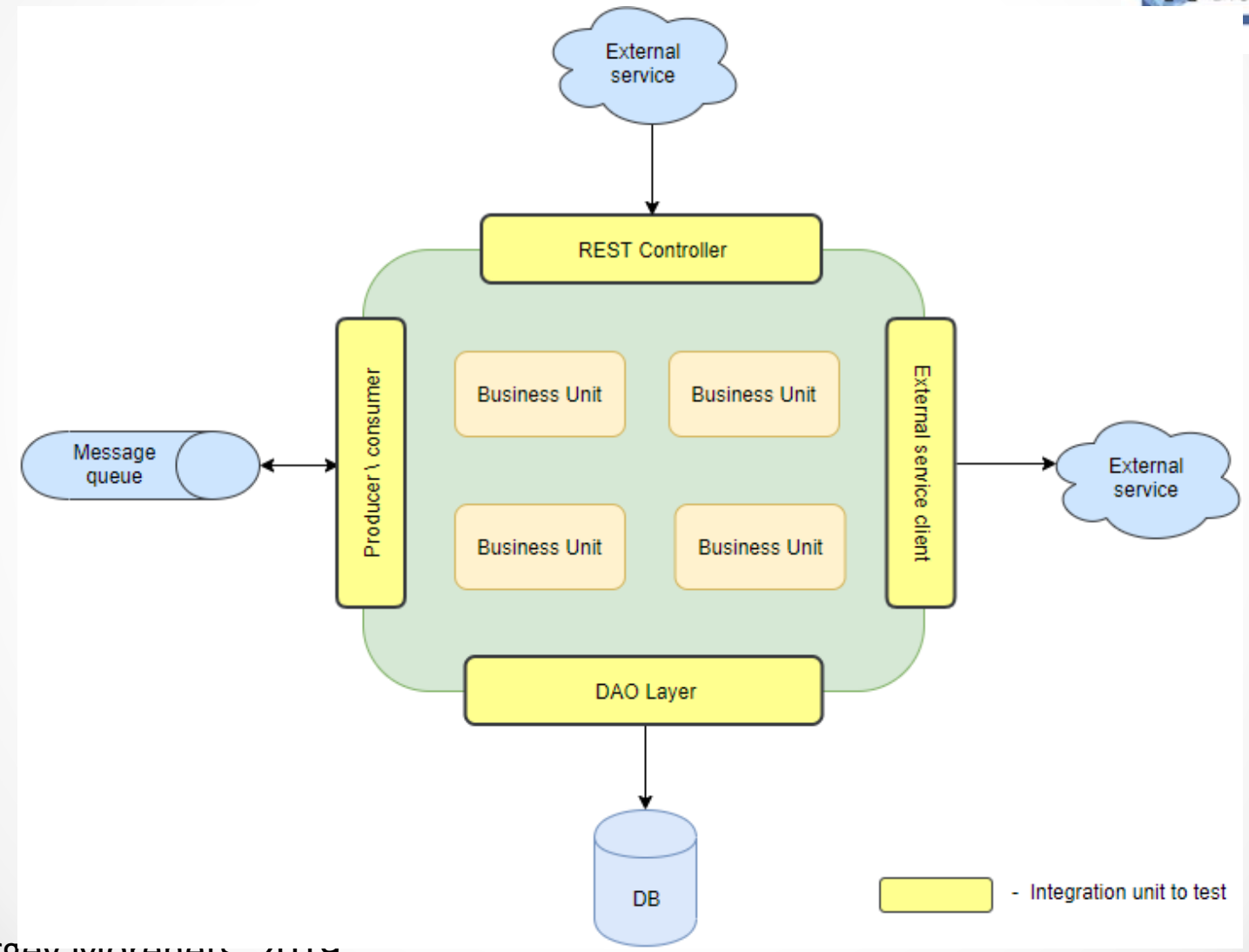
Maven

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

Gradle

- bootRun
- bootRepackage

Integration testing



- Sergey Morenets, 2019

JUnit 5

- ✓ JUnit 5 = Platform + Jupiter + Vintage
- ✓ Separation of concerns
- ✓ API improvements, test extension model
- ✓ Supports repeated, parametrized and dynamic tests
- ✓ New assumptions concept
- ✓ Java 8 - based

JUnit 5

A first test case

```
import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;

class FirstJUnit5Tests {

    @Test
    void myFirstTest() {
        assertEquals(2, 1 + 1);
    }

}
```

Spring MVC Test. JUnit 5


```
@SpringJUnitWebConfig(BookApplication.class)
@AutoConfigureMockMvc
public class BookControllerTest {
    @Autowired
    private MockMvc mockMvc;
```

Spring Boot startup class

Spring MVC bridge

Spring MVC Test example

import static
org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;

```
@Test
public void getBooks_RepositoryEmpty_NoBooksReturned()
        throws Exception  {
    //Given
    //When
    ResultActions actions = mockMvc.perform(get("/book"));
    //Then
    actions.andExpect(status().isOk())
            .andExpect(content().contentType(
                    MediaType.APPLICATION_JSON_UTF8_VALUE))
            .andExpect(jsonPath("$", Matchers.hasSize(0)));
}
```

1.Find controller
2.Find method

import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

What is monolith application?

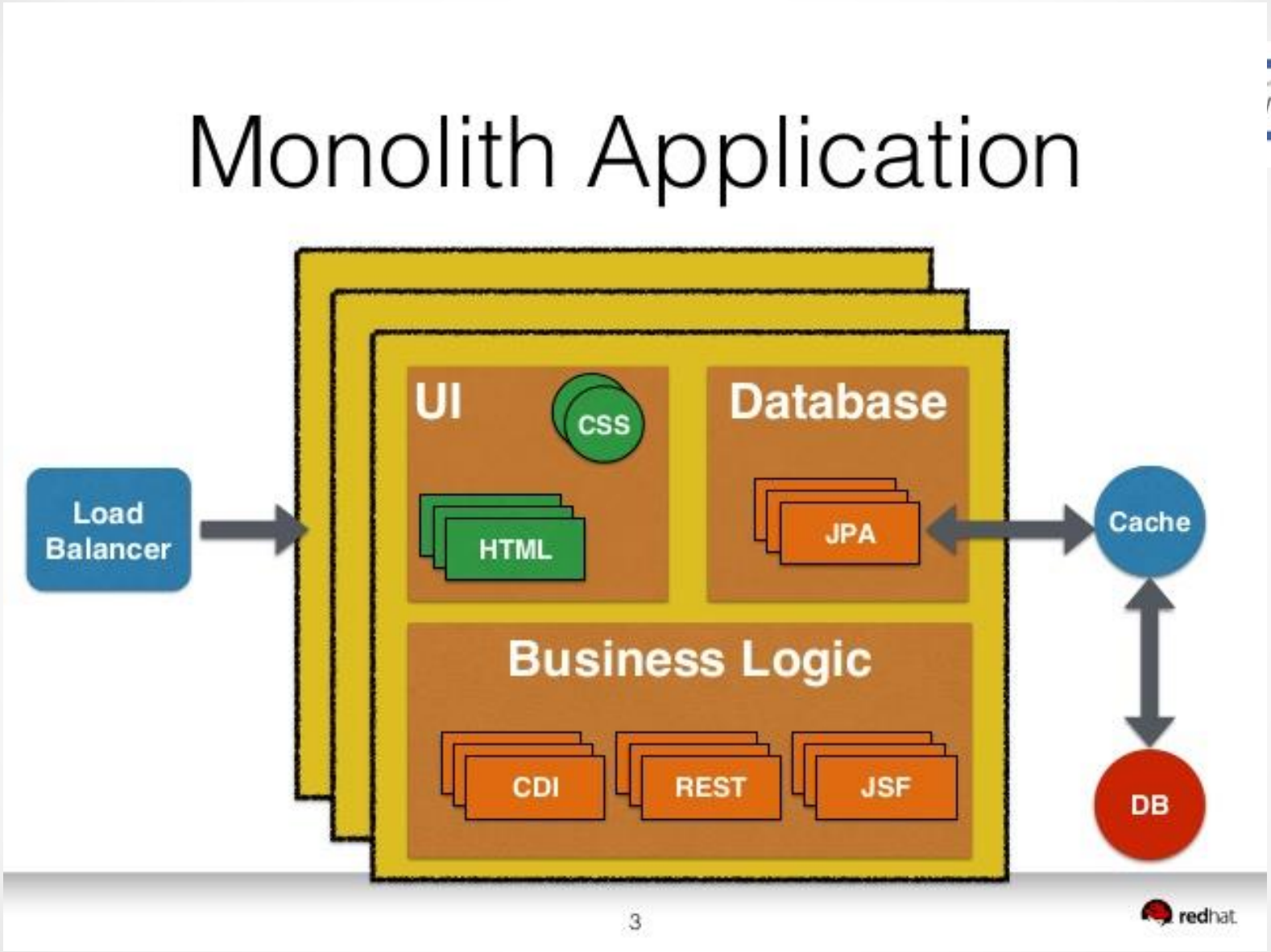
- Sergey Morenets, 2019

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
- ✓ Enterprise Service Bus (ESB)
- ✓ One or few **teams**

Monolith application

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



- Sergey Morenets, 2019

Issues

- Sergey Morenets, 2019

UI

- Sergey Morenets, 2019

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 1. 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, 2019

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, 2019

Micro-services. Pros

- ✓ **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. Pros

- ✓ **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 design patterns

- ✓ Circuit breaker
- ✓ Event sourcing
- ✓ Health Endpoint (expose endpoints for external monitoring)
- ✓ Leader election (in distributed application)
- ✓ Publisher/subscriber
- ✓ Retry
- ✓ Sharding
- ✓ Sidecar
- ✓ Anti-corruption layer (translation between different sub-systems)

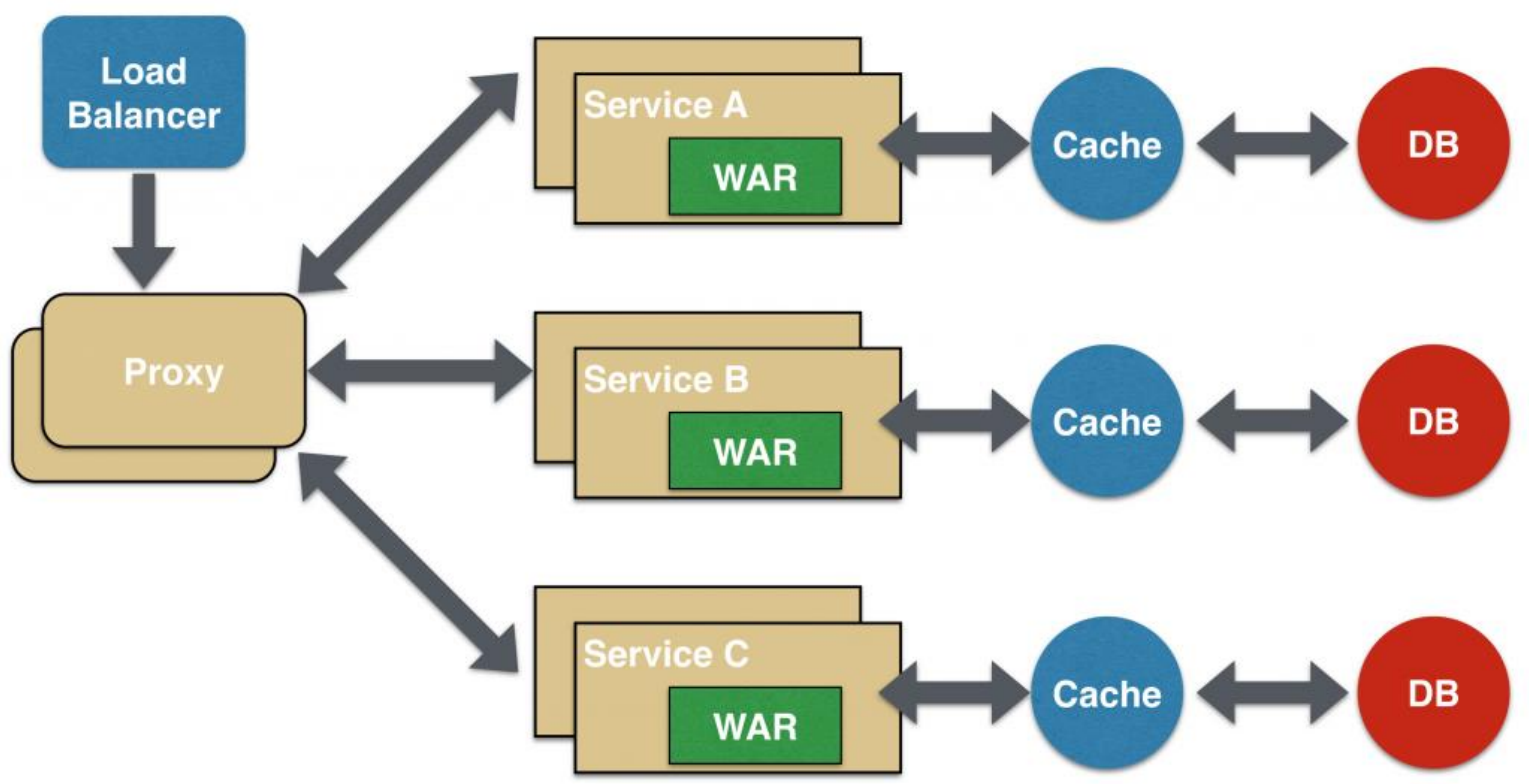
Micro-services

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

Microservices. Drawbacks

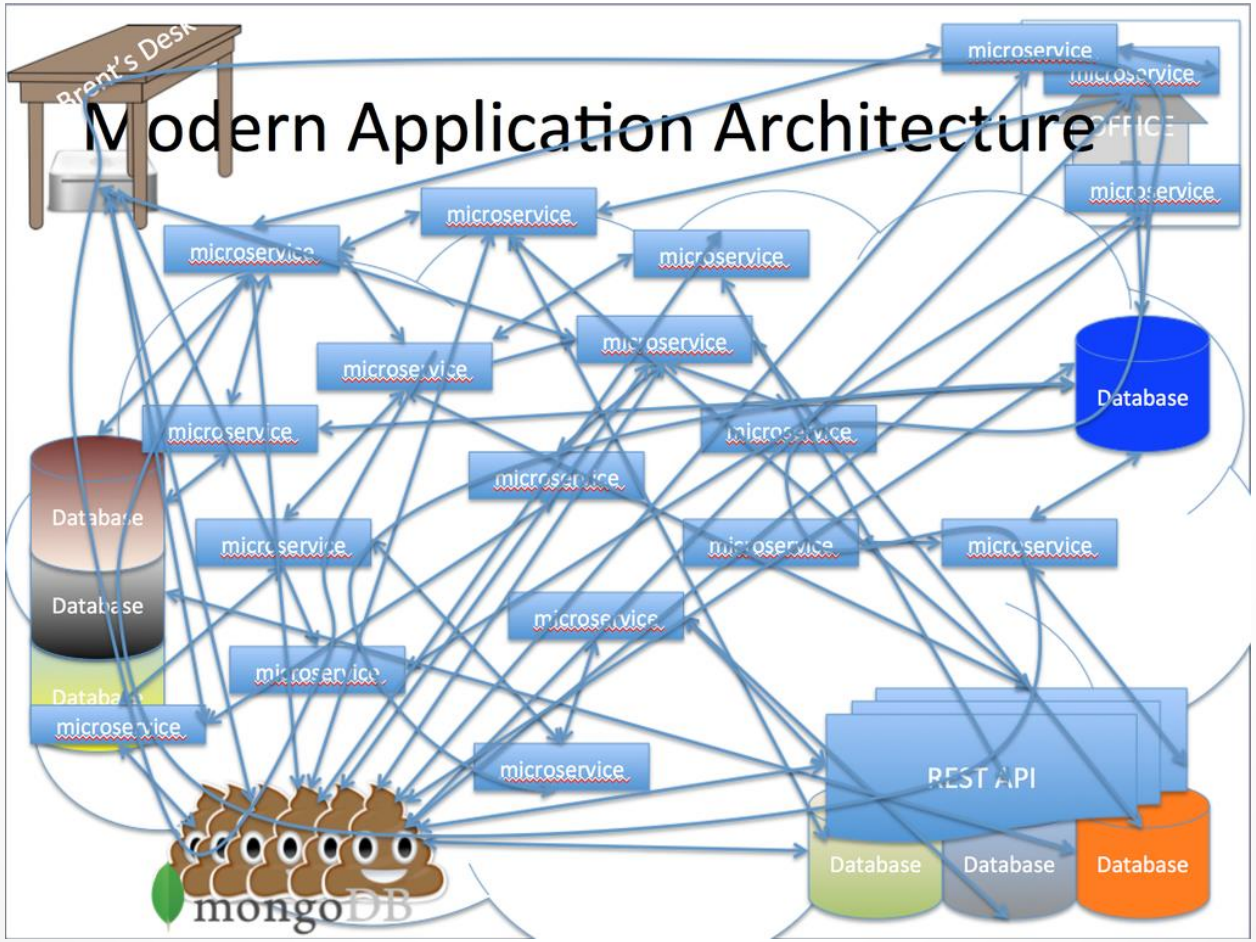
- ✓ Higher level of complexity
- ✓ Transaction management
- ✓ Testing of distributed application
- ✓ Deployment and management
- ✓ Cost of remote calls

Drawbacks



- Sergey Morenets, 2019

Drawbacks

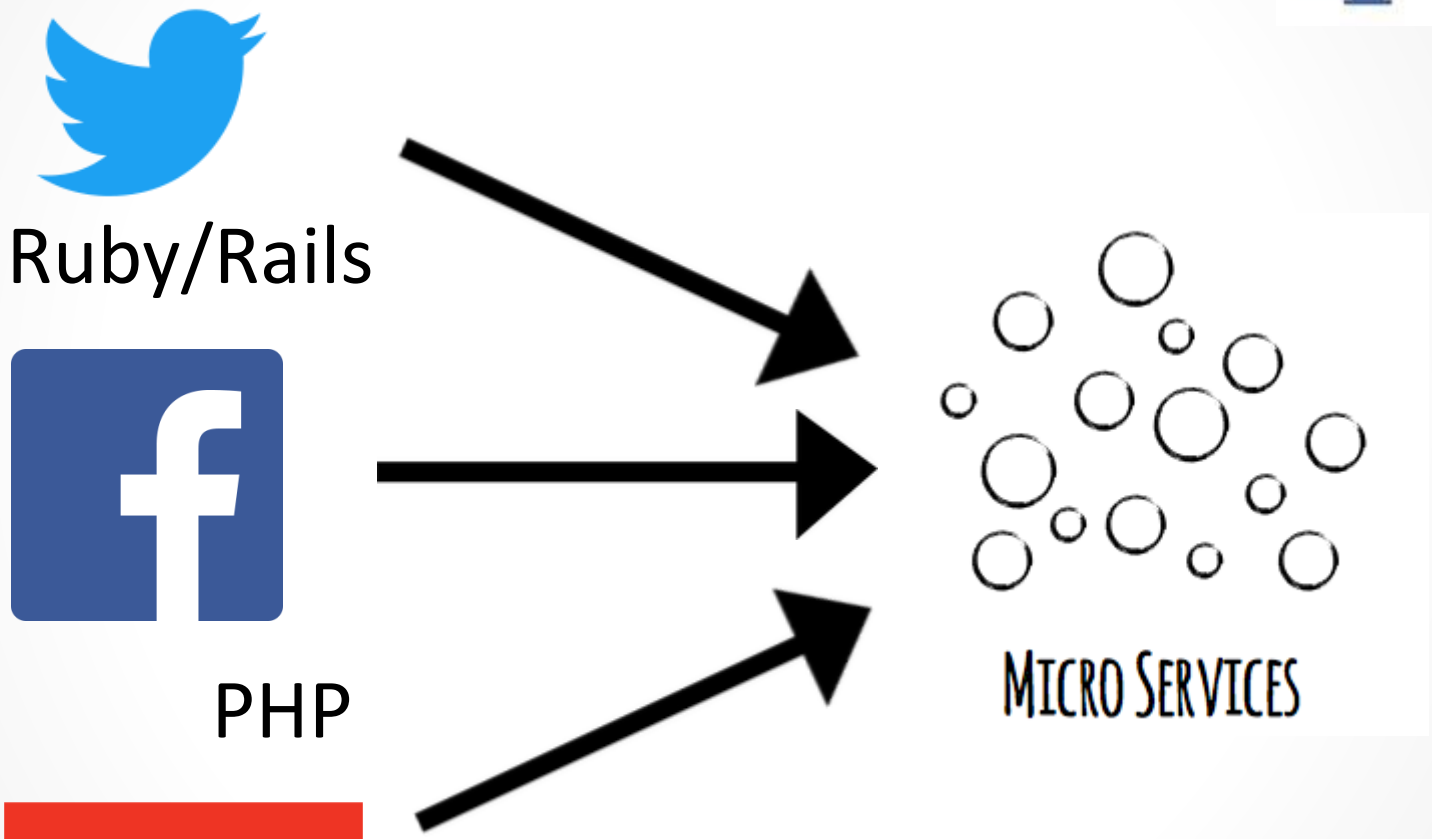


- Sergey Morenets, 2019

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)

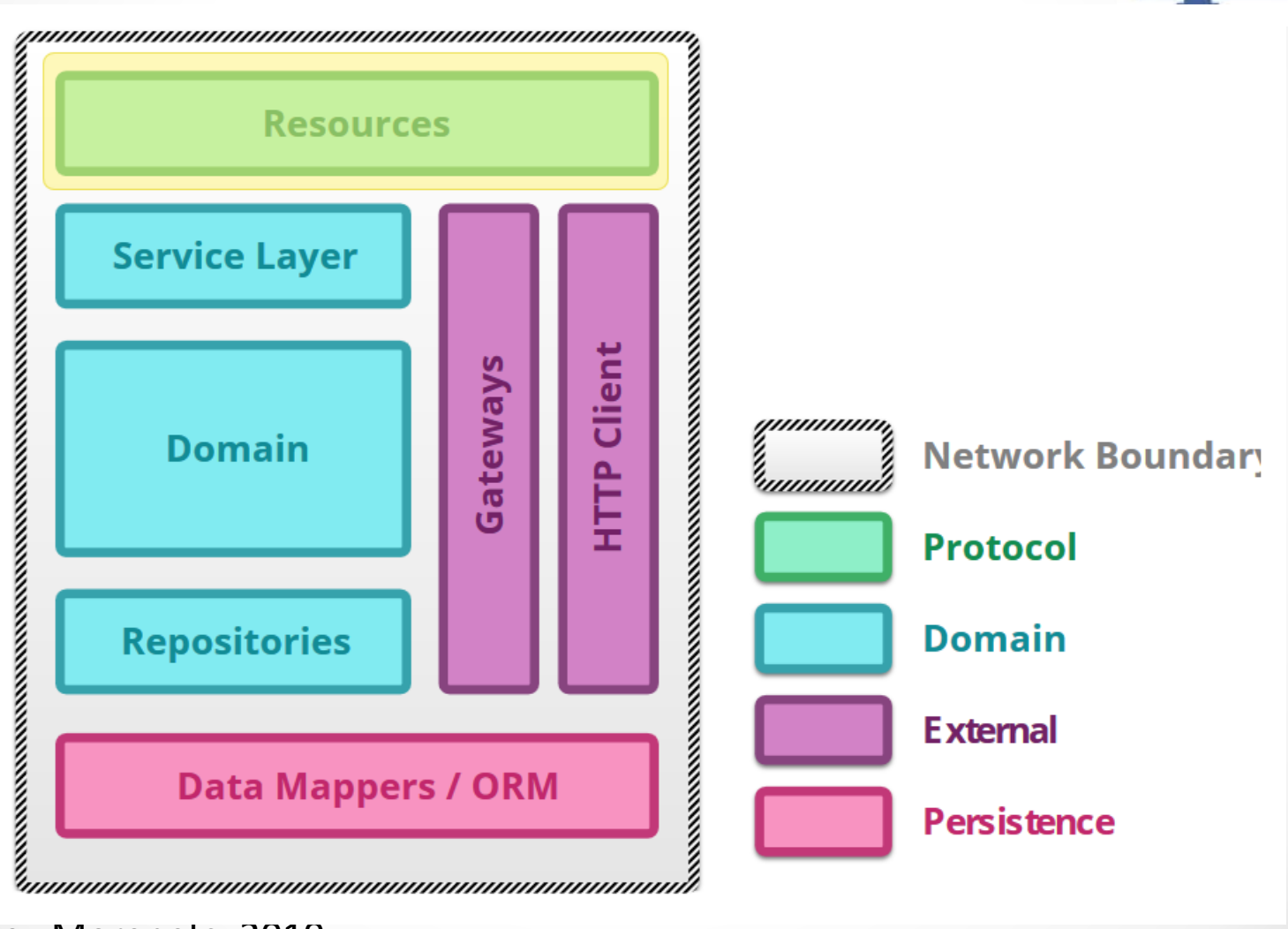
Transition



Transition

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

Microservice structure



- Sergey Morenets, 2019

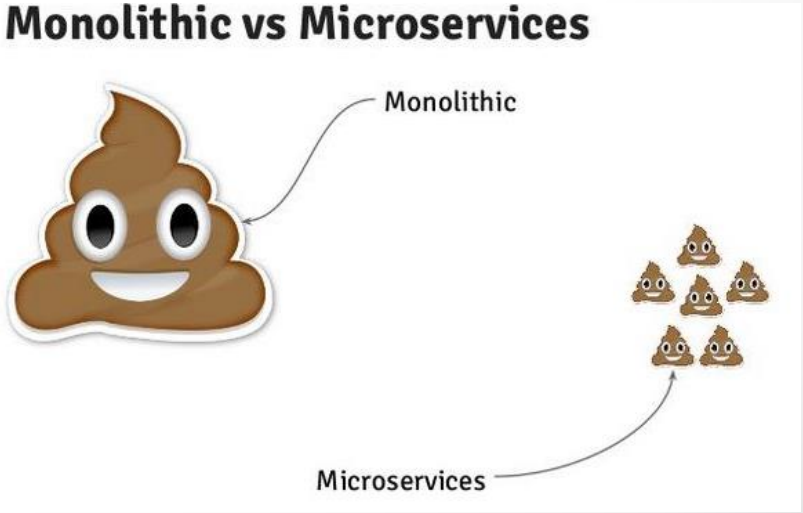
Polyglot persistence

- Sergey Morenets, 2019

- Sergey Morenets, 2019

Anti-patterns

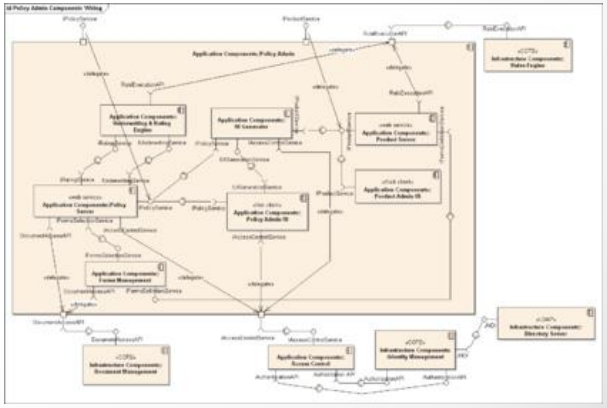
- ✓ Distributed monolith
- ✓ Single feature goes into all the micro-services



- Sergey Morenets, 2019

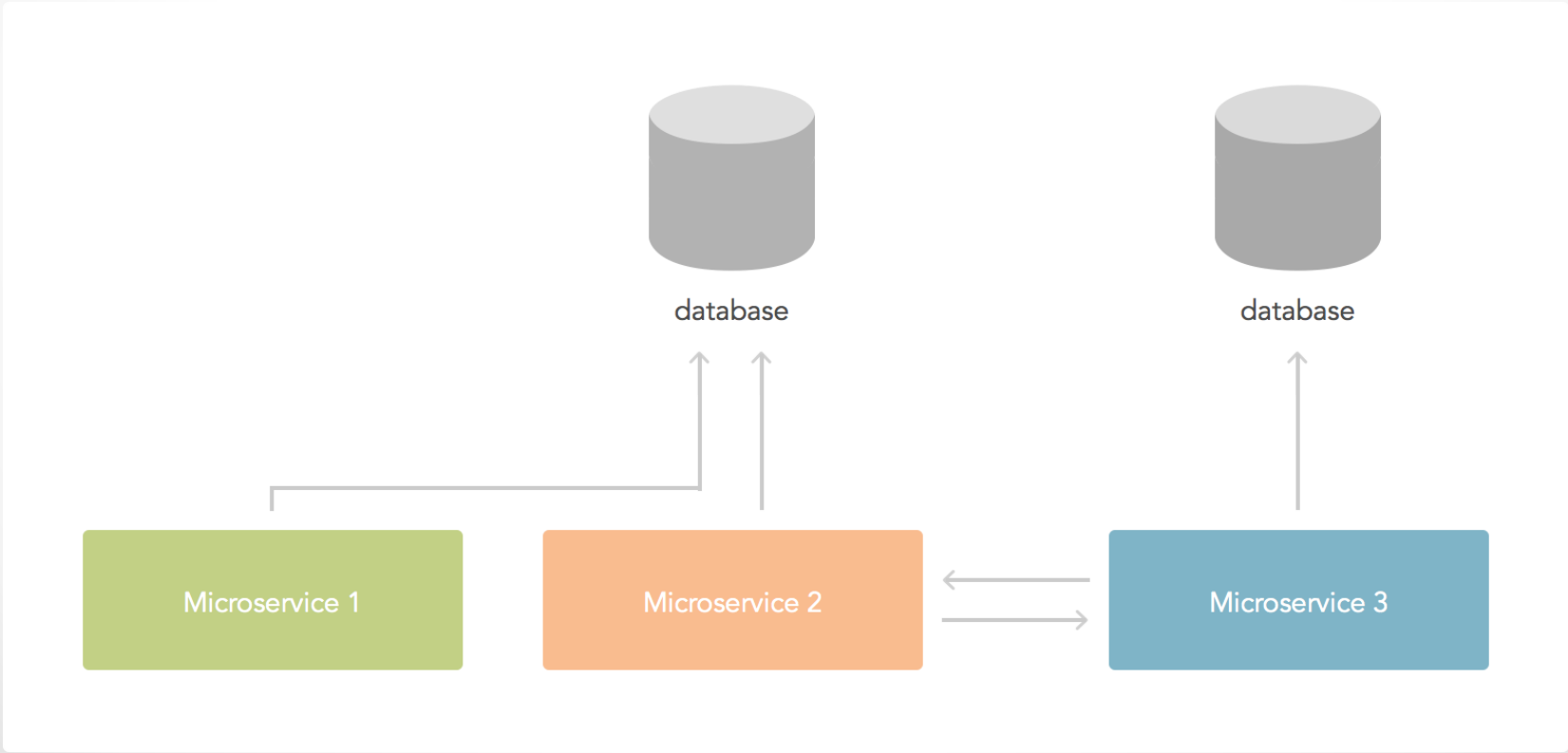
Anti-patterns

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



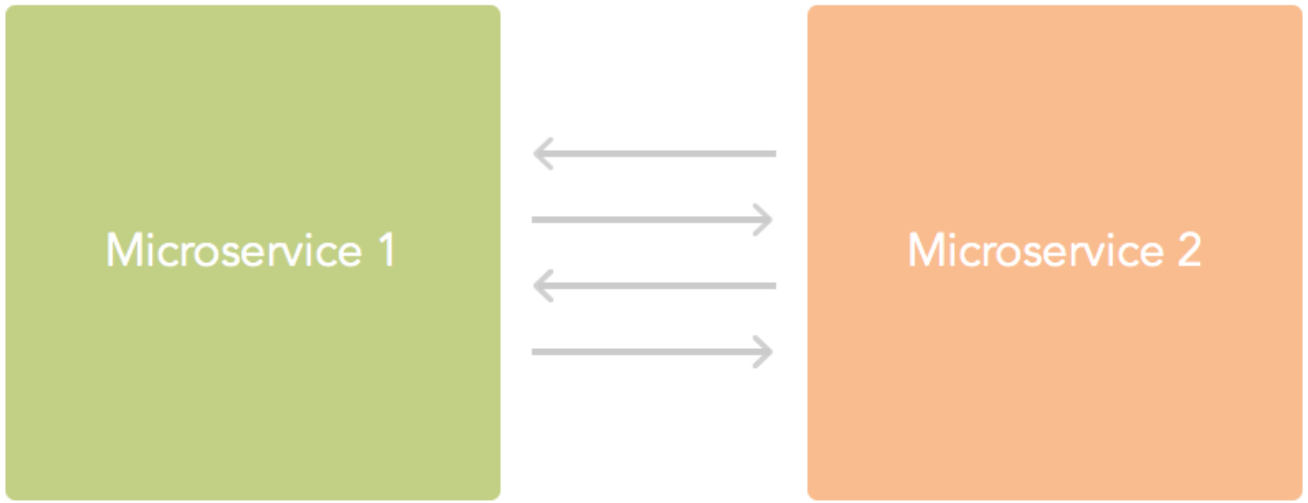
- Sergey Morenets, 2019

Anti-patterns



- Sergey Morenets, 2019

Anti-patterns

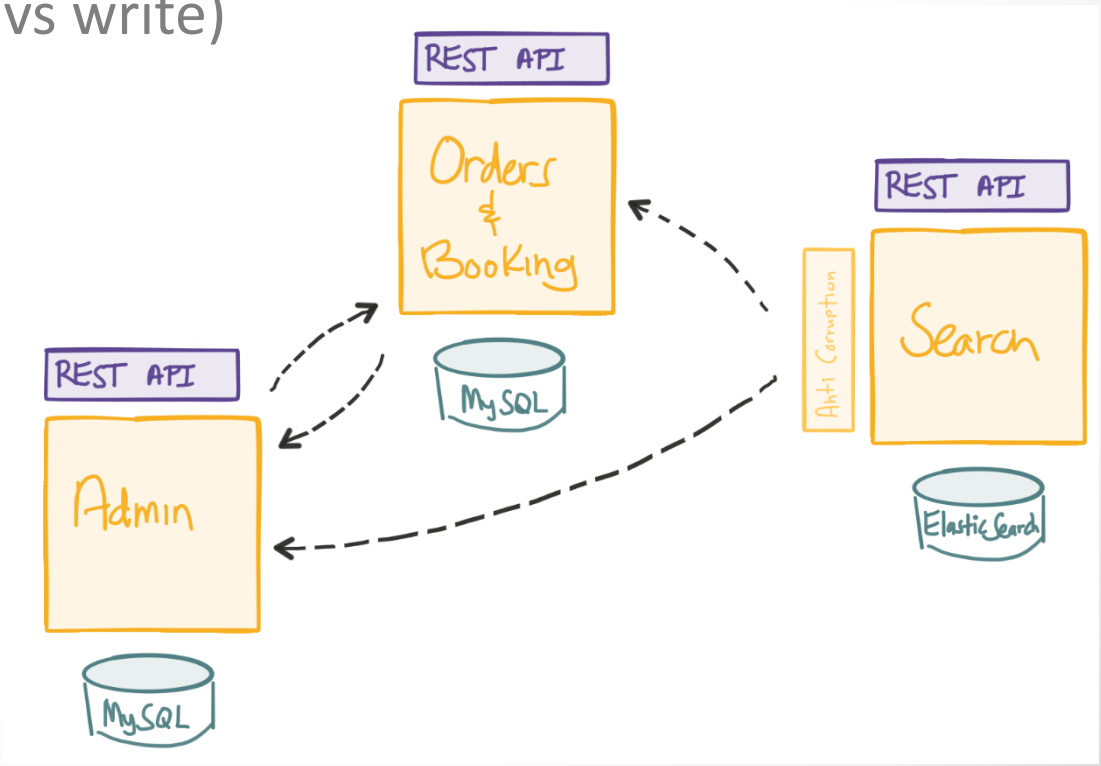


Two microservices sending lots of messages back and forth
candidates for turning into a single service

- Sergey Morenets, 2019

Partitioning strategies

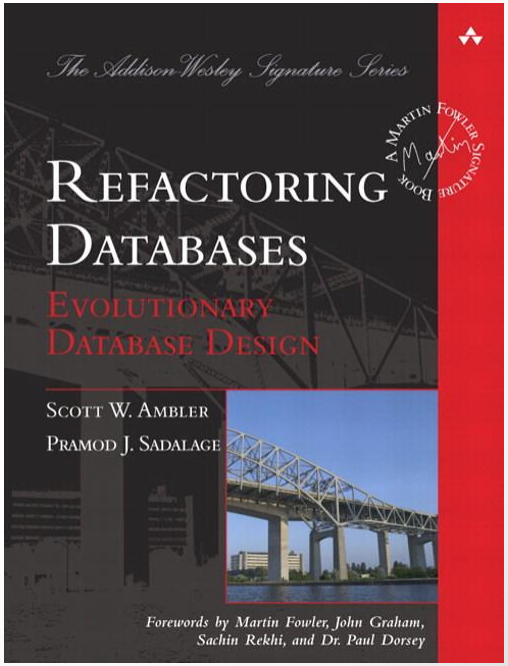
- ✓ By entity(Customer – Product – Order)
- ✓ By use case(Book – Buy – Search)
- ✓ Single Responsibility Principle
- ✓ By data access (read vs write)
- ✓ By data source
- ✓ Bounded context



- Sergey Morenets, 2019

Transition

- ✓ Split domain model/**business logic**
- ✓ Split data model/persistence layer
- ✓ Split **database** (including DB migration)
- ✓ Split/introduce **technologies**
- ✓ Split **API** (optional)
- ✓ Split into **deployable modules**



- Sergey Morenets, 2019

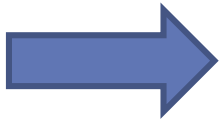
Enterprise model. Client

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

Splitting data model

Client. **Purchase service**

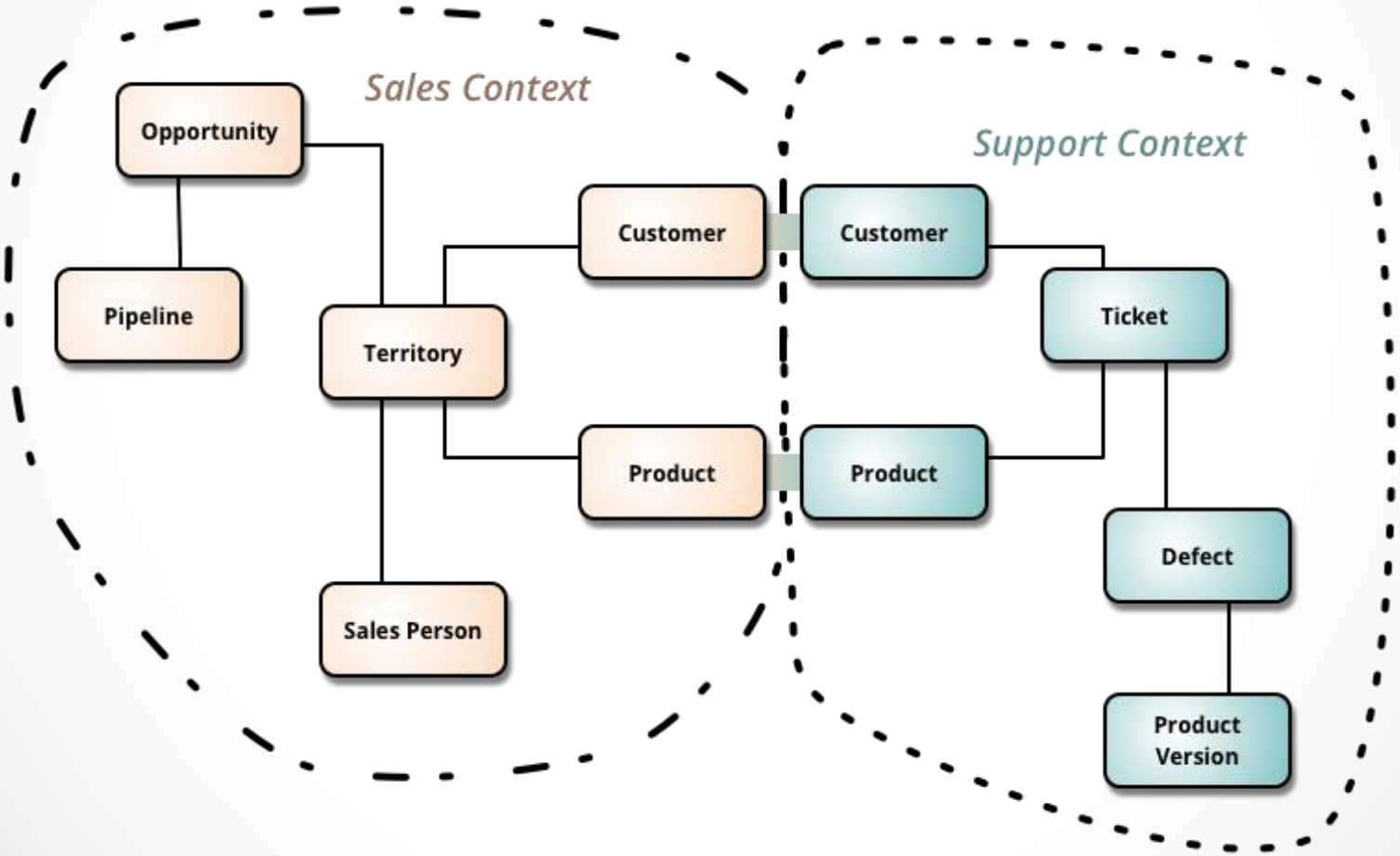
- ✓ Identifier
- ✓ Credit card number
- ✓ Expiration date
- ✓ Discount
- ✓ Purchases



Client. **Delivery service**

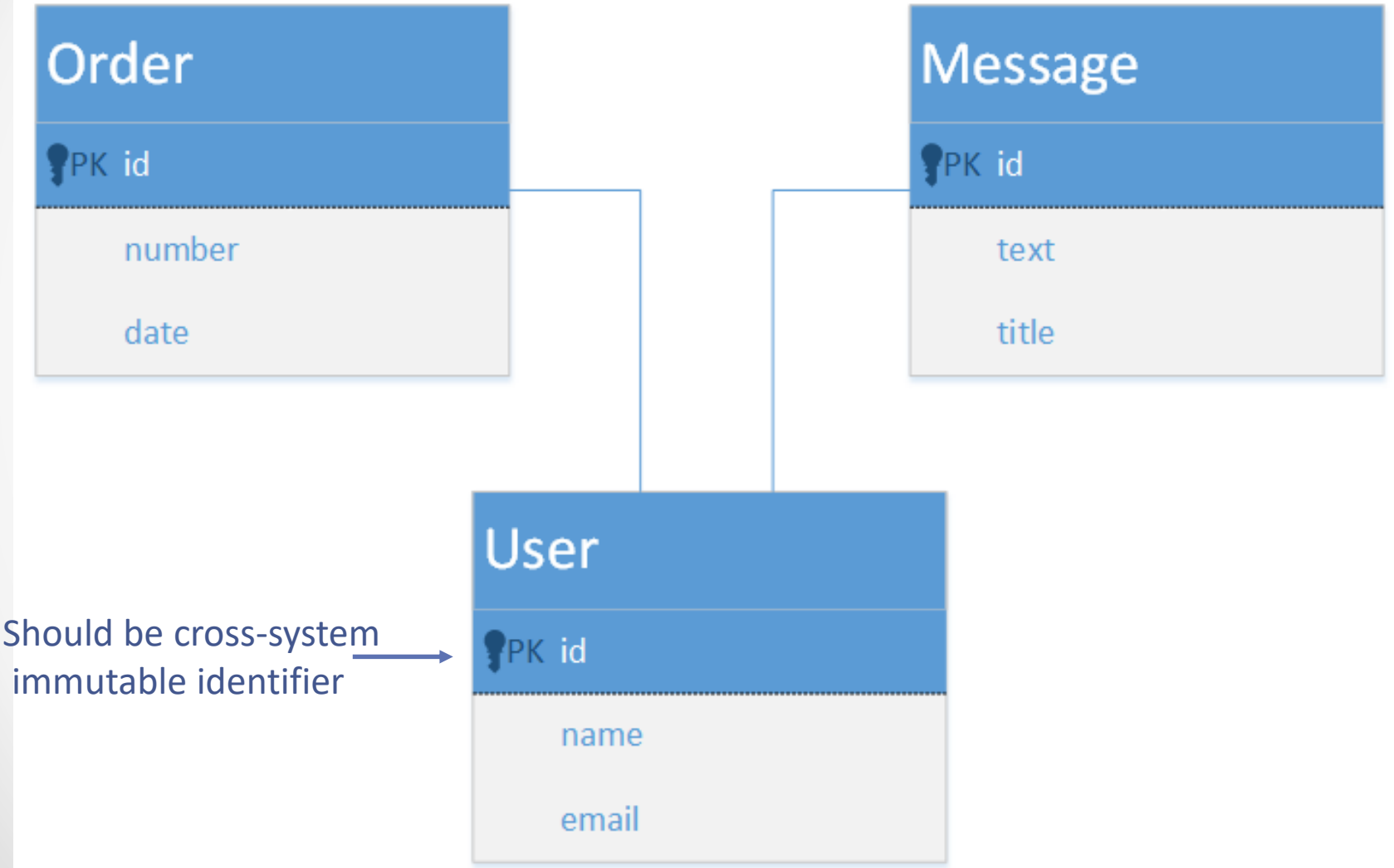
- ✓ Identifier
- ✓ Name
- ✓ Address
- ✓ Email/Phone

Bounded context



- Sergey Morenets, 2019

Splitting data model



Task 2. 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**

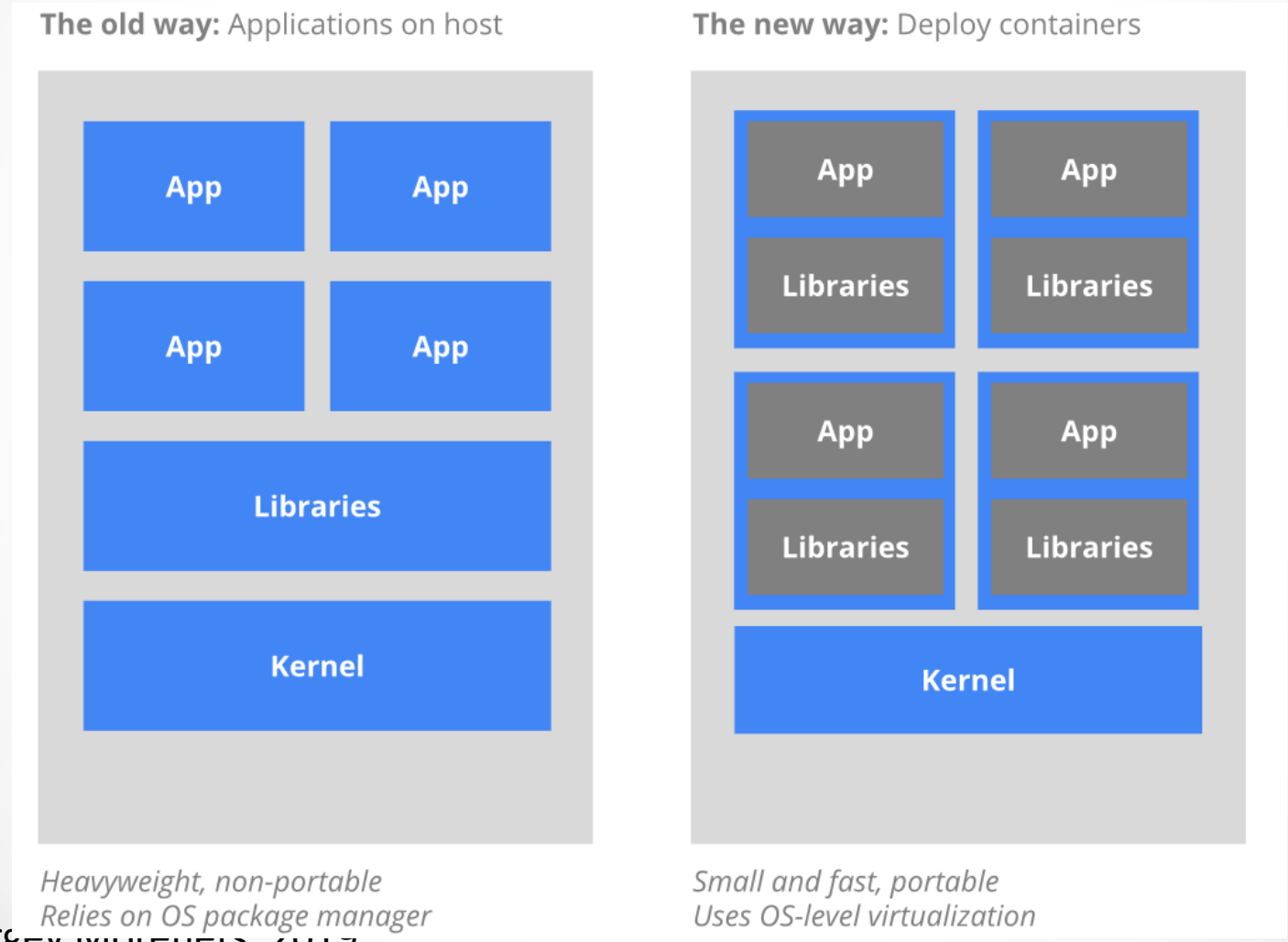
Questions

- 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

- Sergey Morenets, 2019

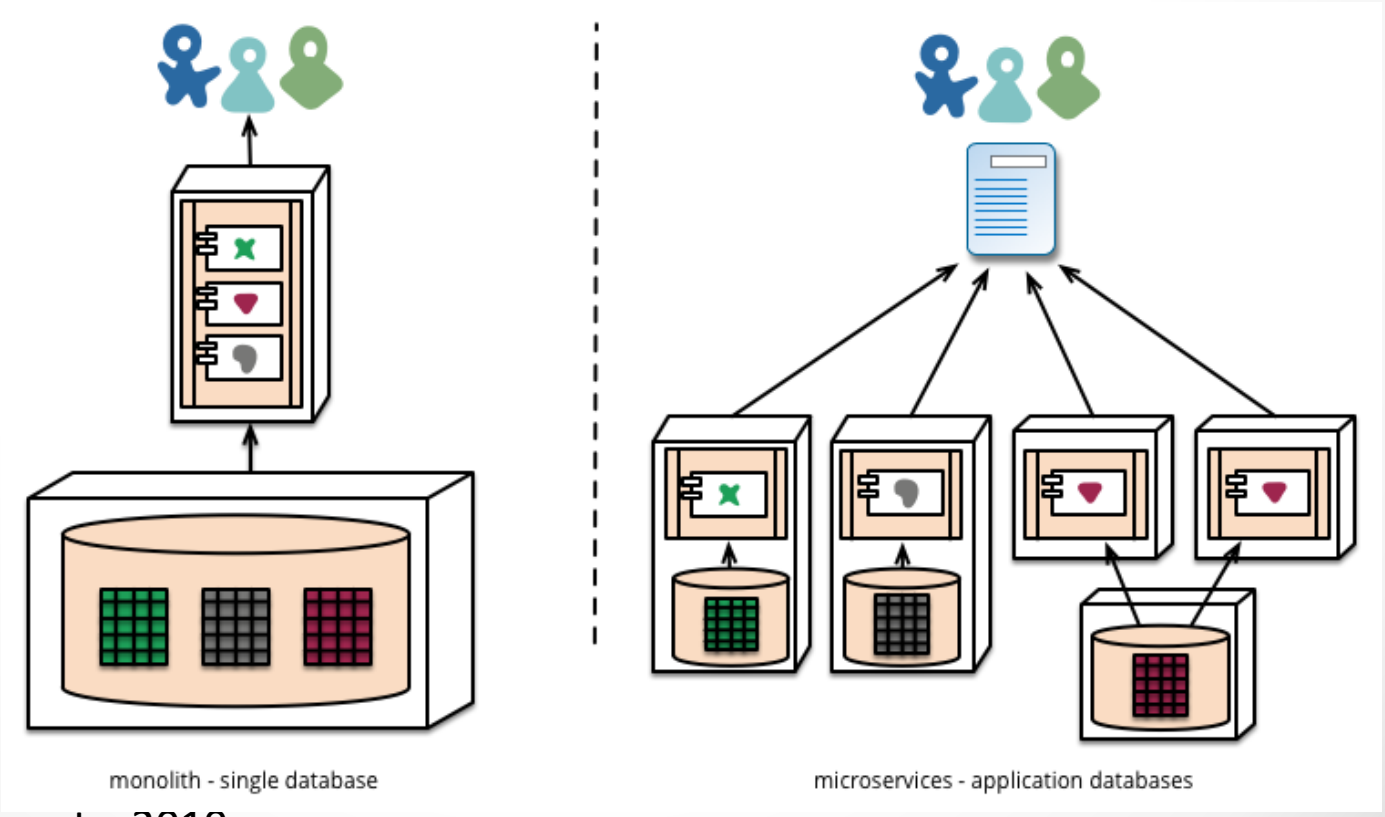
Infrastructure



- Sergey Morenets, 2019

Data storage

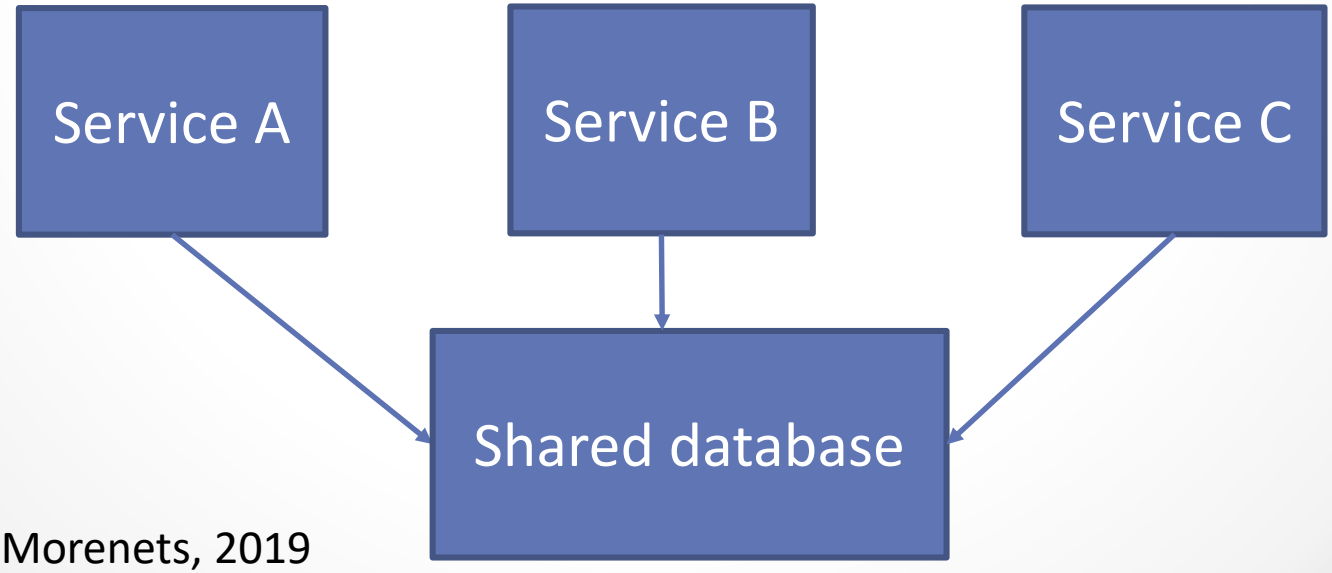
- ✓ Shared database
- ✓ Database per service



- Sergey Morenets, 2019

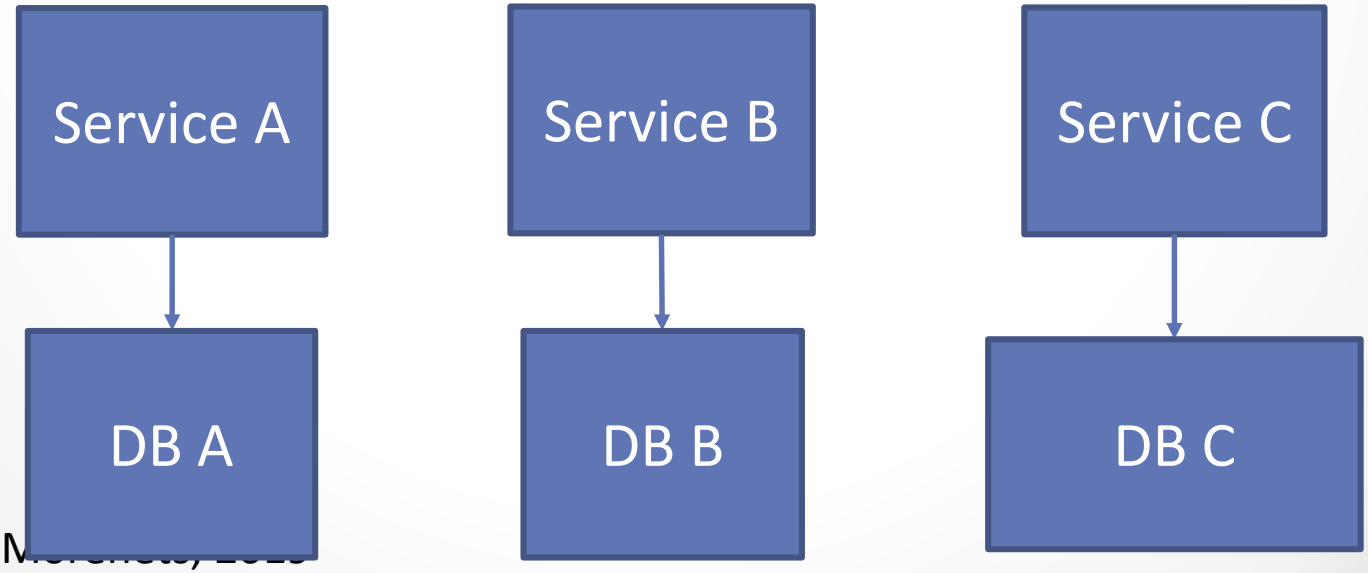
Shared database

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

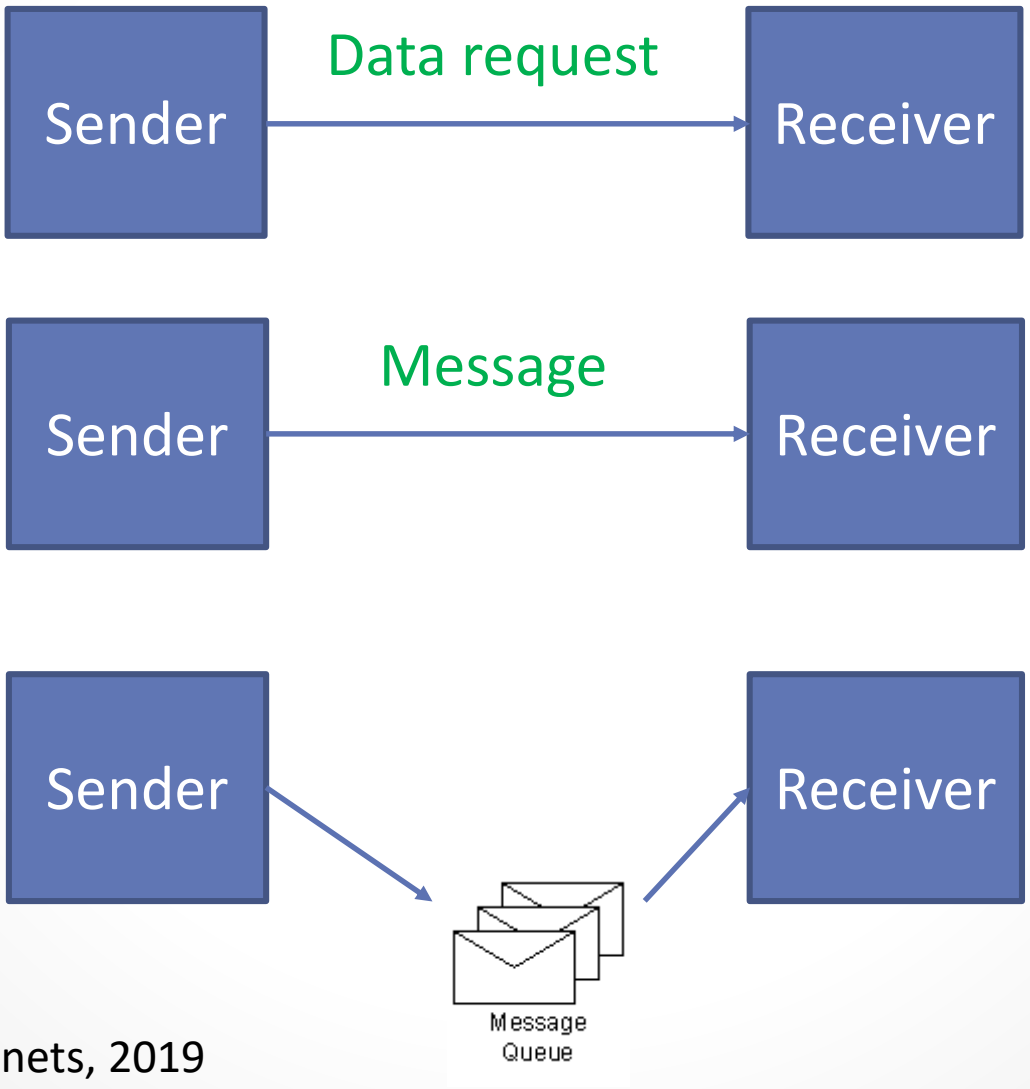


Database per service

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



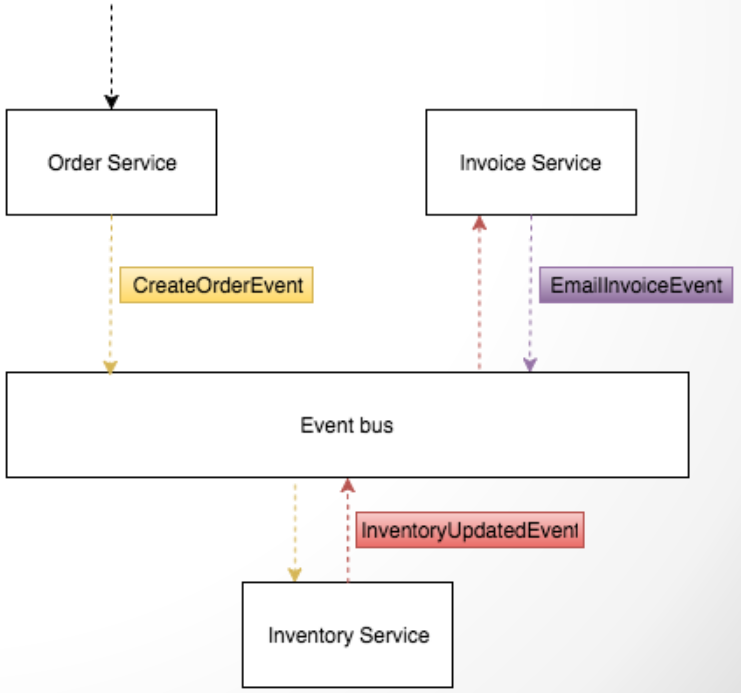
Service communication. Strategies



- Sergey Morenets, 2019

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



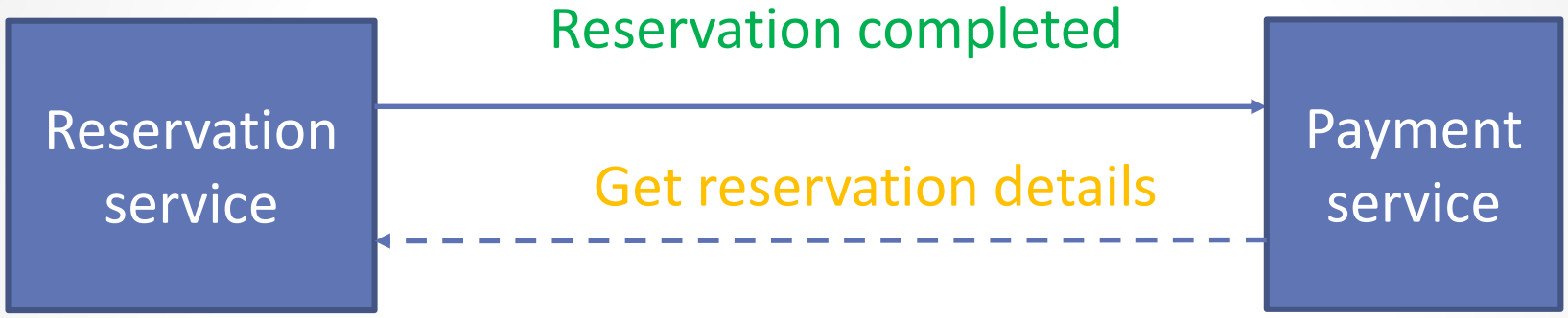
- Sergey Morenets, 2019

Event-driven patterns

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

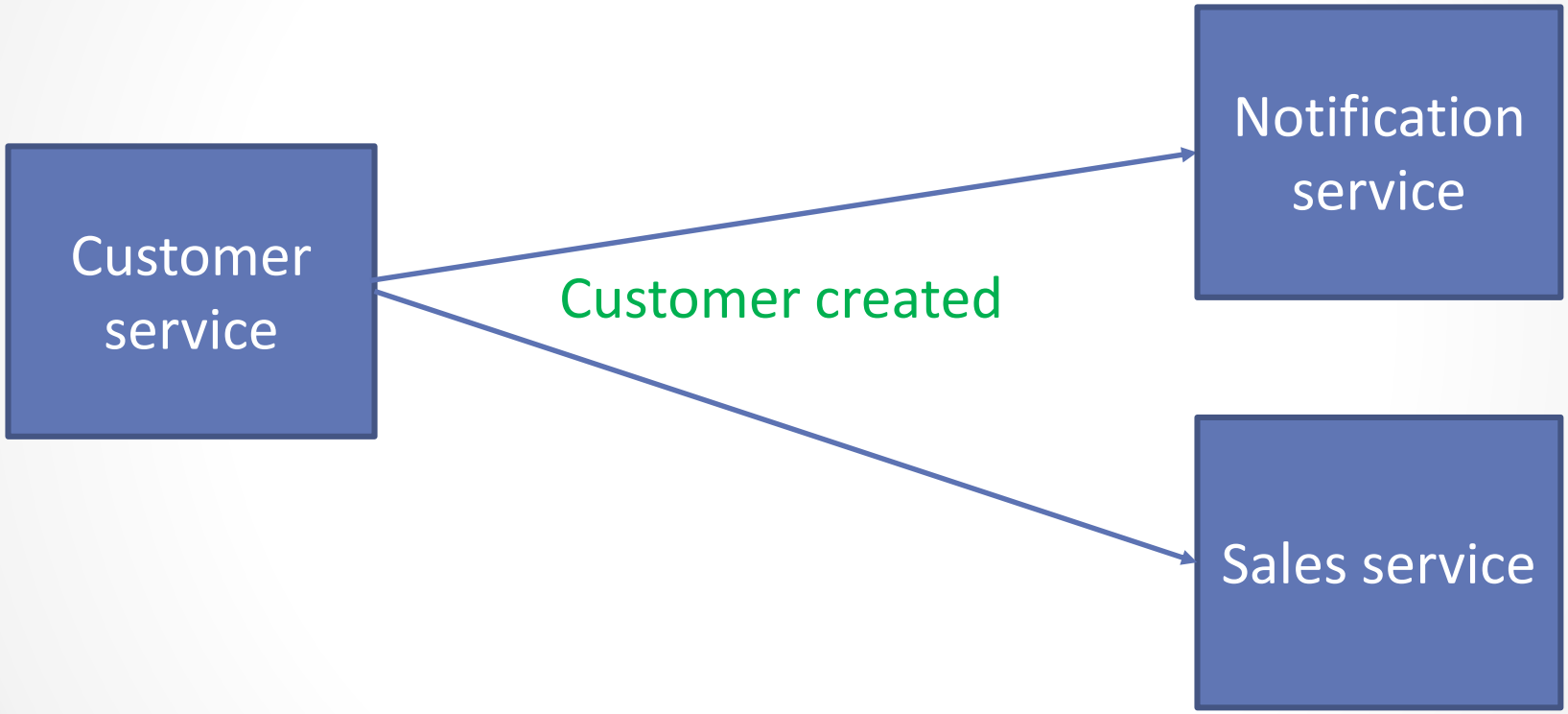
- Sergey Morenets, 2019

Event notification




```
@Value
public class ReservationCompleted {
    private String reservationId;

}
```

Event-carried state transfer



- Sergey Morenets, 2019

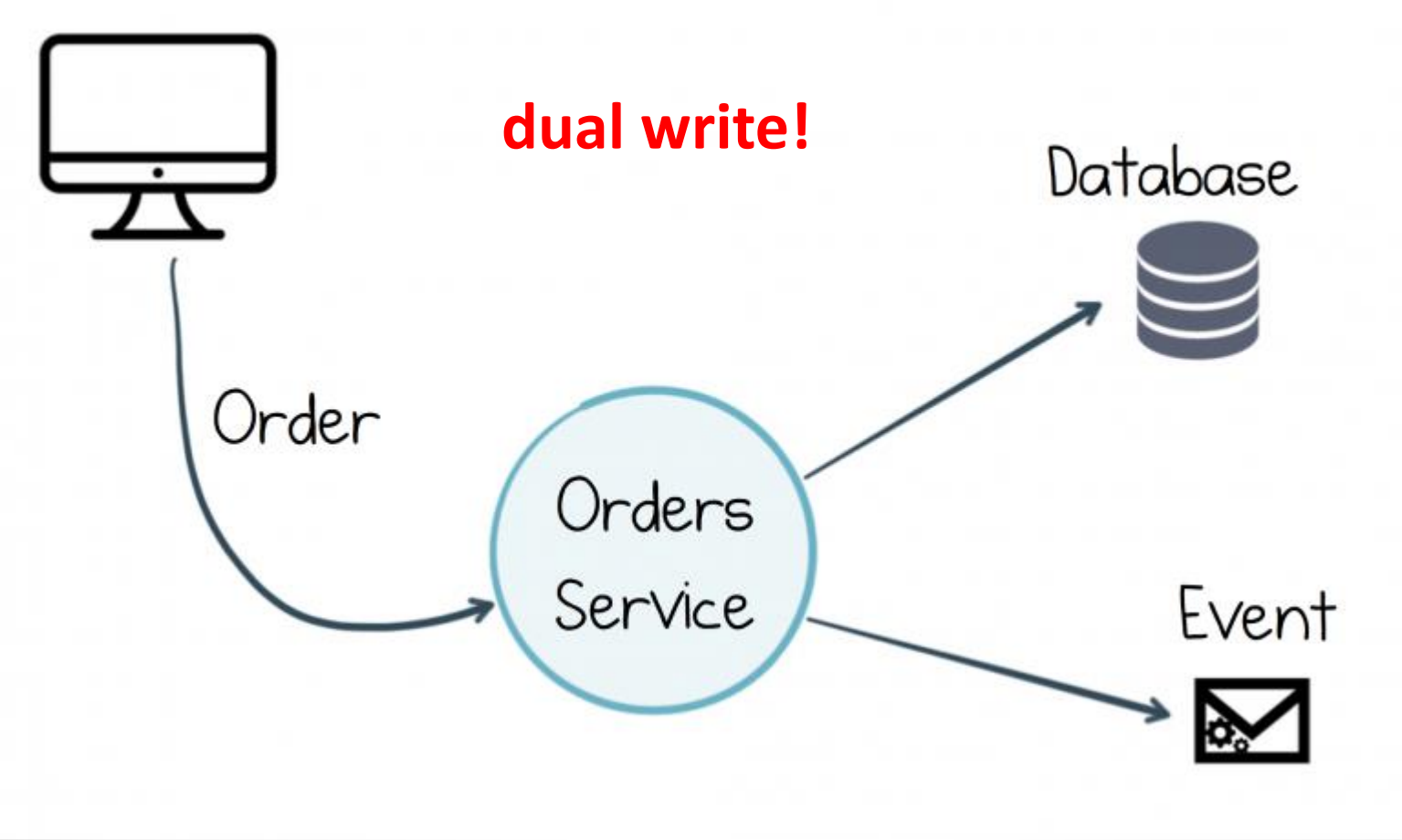
Event-carried state transfer


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

    private String name;

    private String email;
}
```

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

- Sergey Morenets, 2019

Event-driven architecture



- Sergey Morenets, 2019

Task 3. 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 4. 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

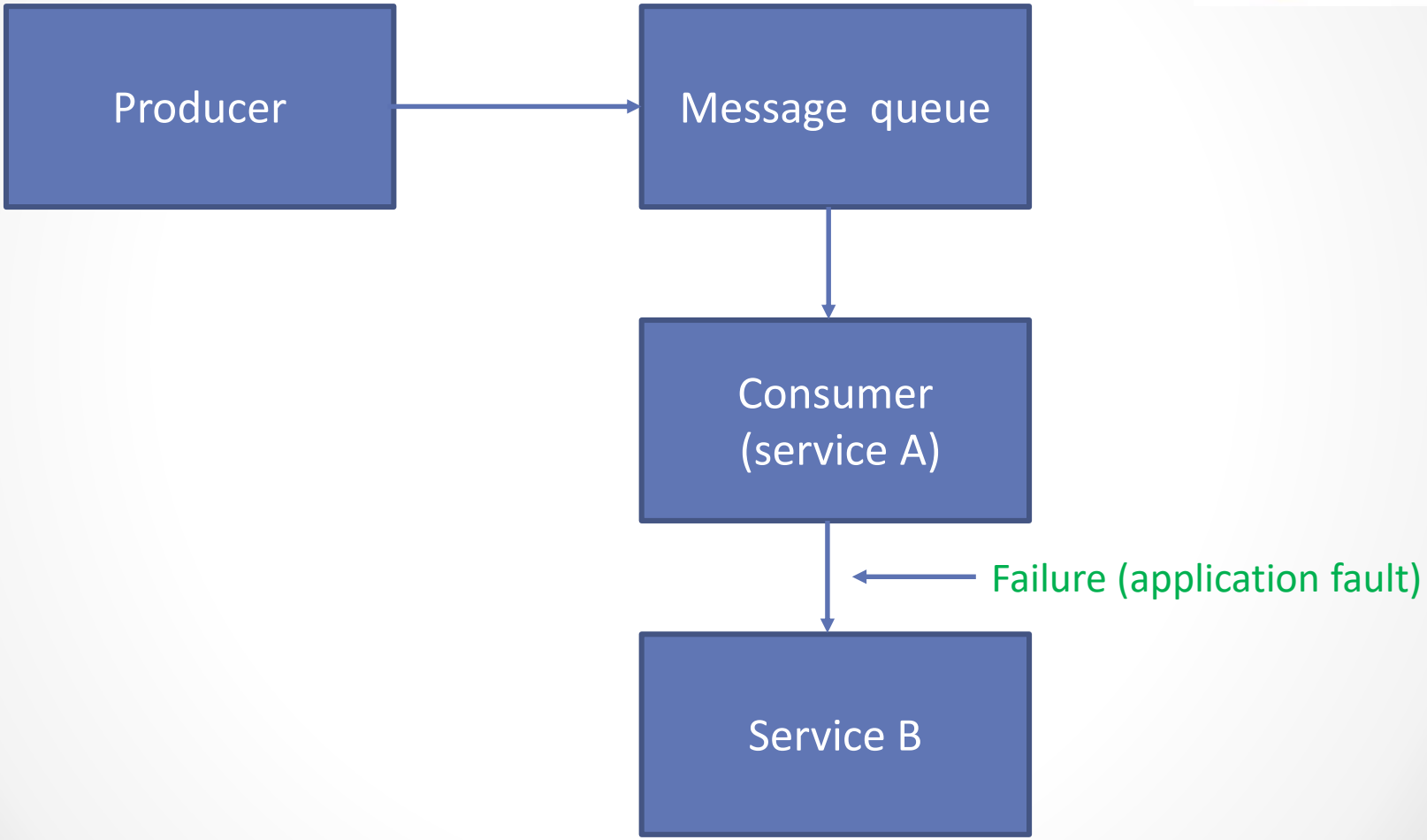
Task 5. Microservice structure

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

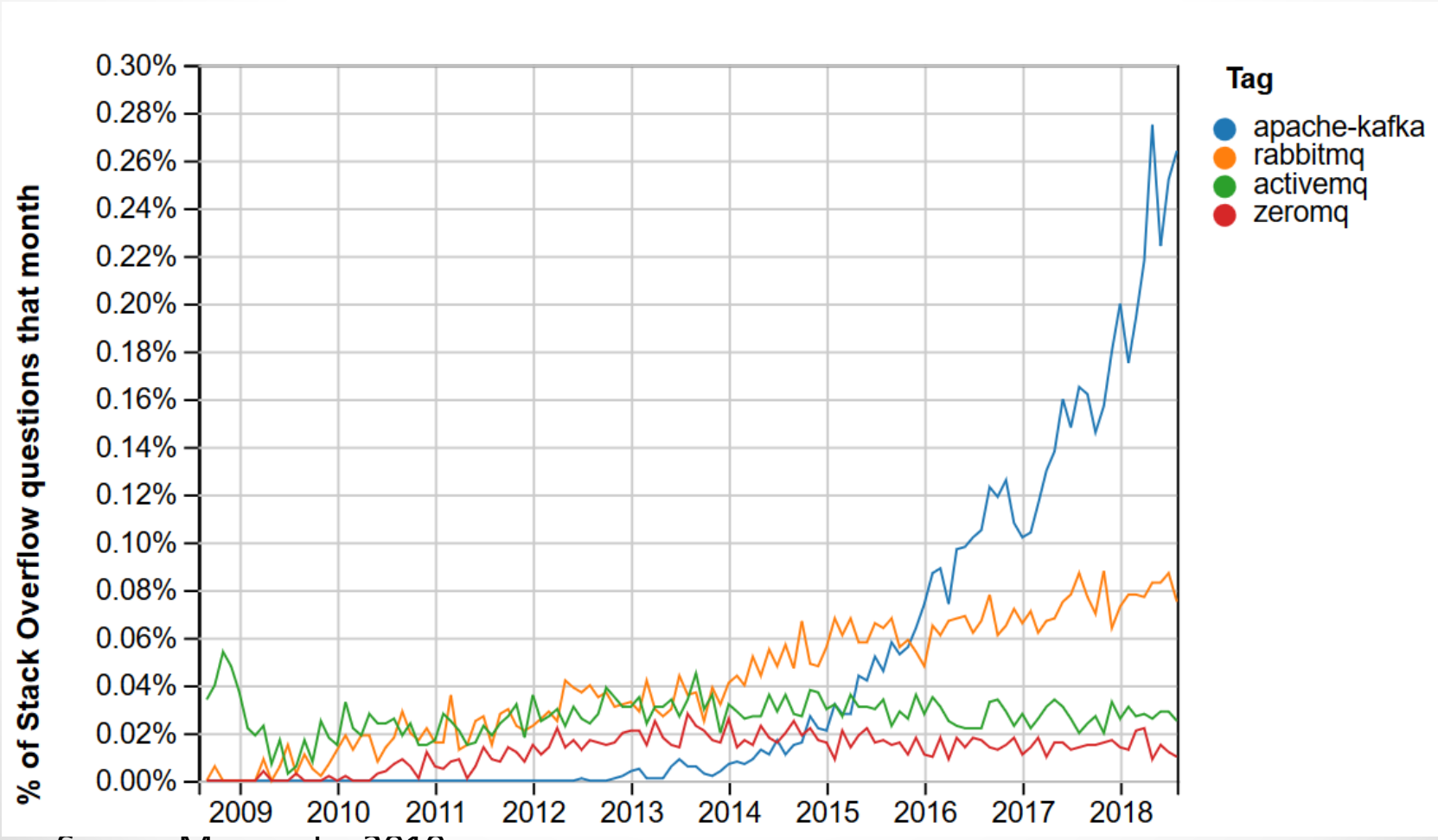
- ✓ Replaces legacy integration layers (enterprise service bus, SOA)
- ✓ Breaks tight coupling
- ✓ Introduce multiple independent consumers
- ✓ Message broker doesn’t block producers
- ✓ Avoid availability issues

Messaging issues



- Sergey Morenets, 2019

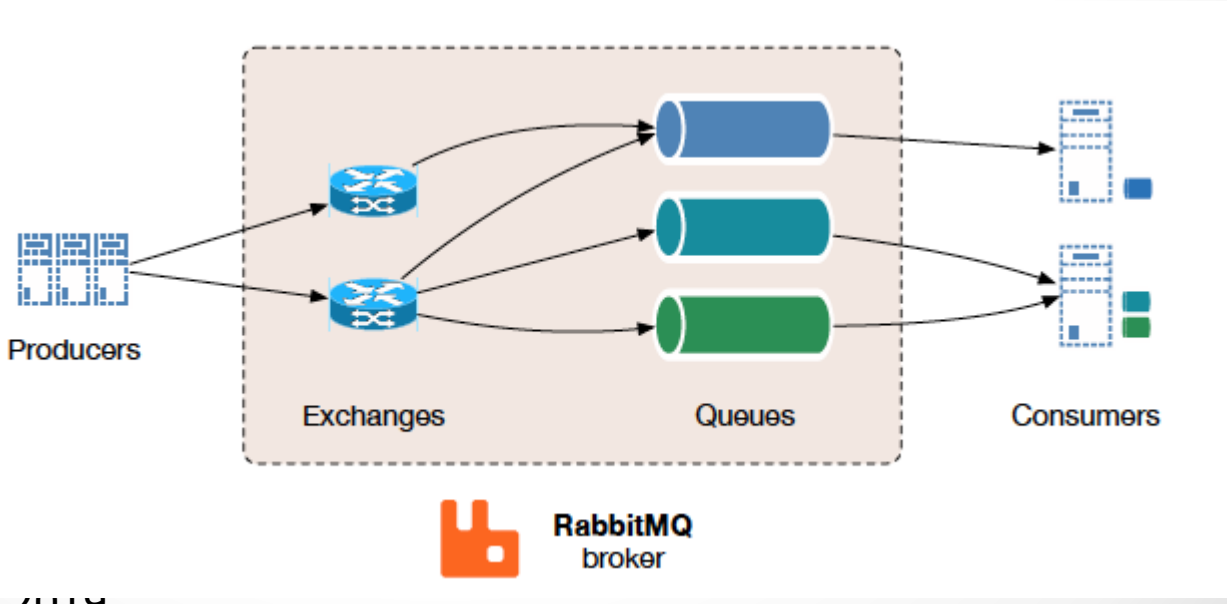
Choose messaging system



- Sergey Morenets, 2019

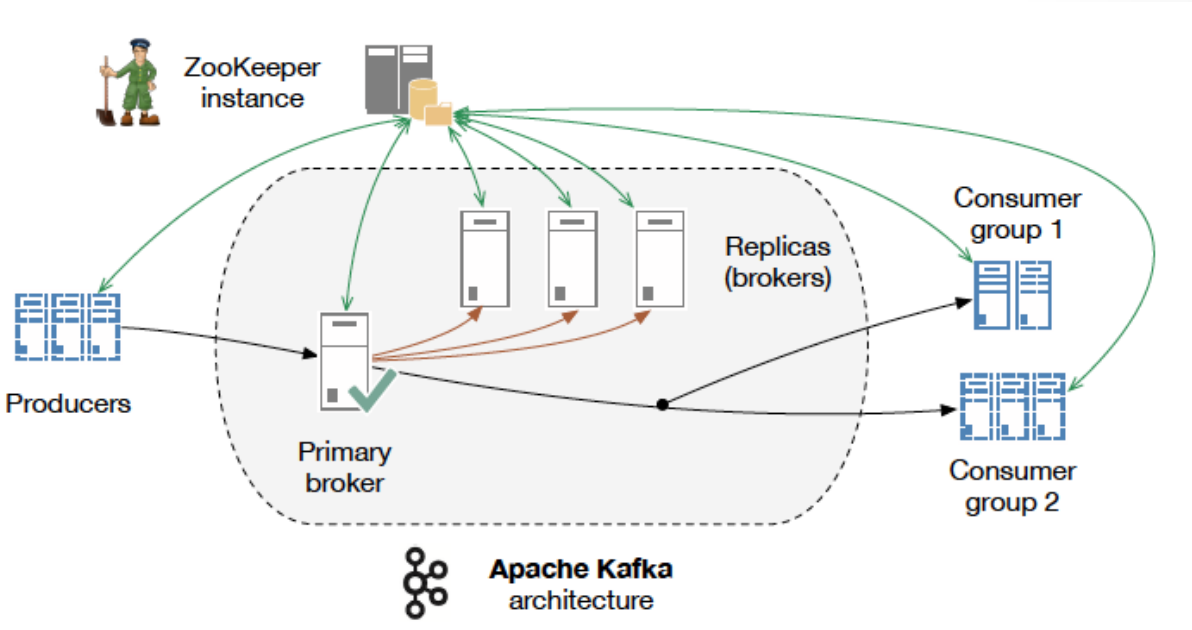
Messaging systems. RabbitMQ

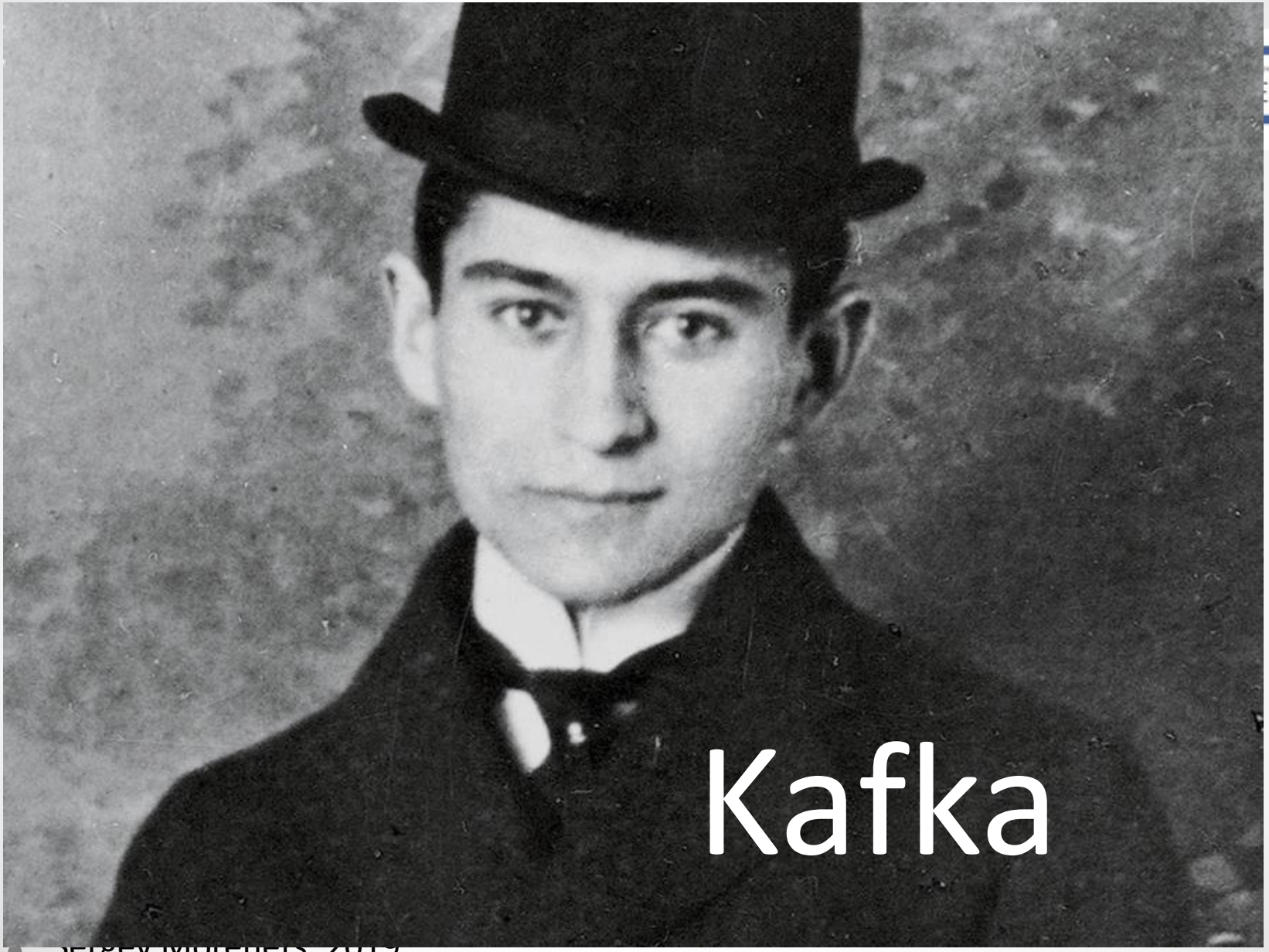
- **RabbitMQ** is one of the first message brokers that supports AMQP/STOMP protocol, and also point-to-point, request-reply and publish-subscriber style patterns
- Support sync/async communication and clustering



Messaging systems. Kafka

- ✓ **Apache Kafka** was design for high volume data, real-time processing and durability
- ✓ Uses Zookeeper for broker management
- ✓ Supports streaming with Kafka streams

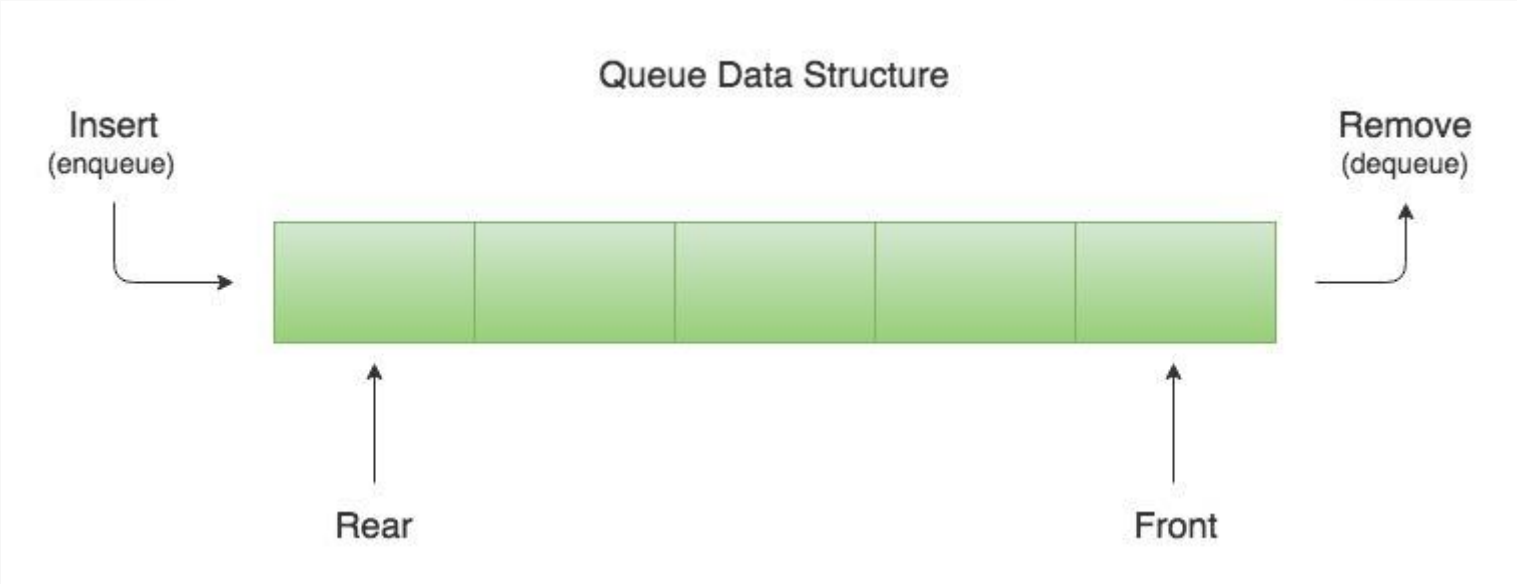



Kafka

Apache Kafka

- ✓ Distributed multi-client messaging system
- ✓ High availability and throughput, resilient to node failures, automatic recovery
- ✓ Development starts in 2009 and open-sourced in 2011
- ✓ Developed by LinkedIn in Java/Scala and now supported by Confluent
- ✓ Author is Jay Kreps (currently CEO at Confluent, also author of Voldemort)

Standard queue



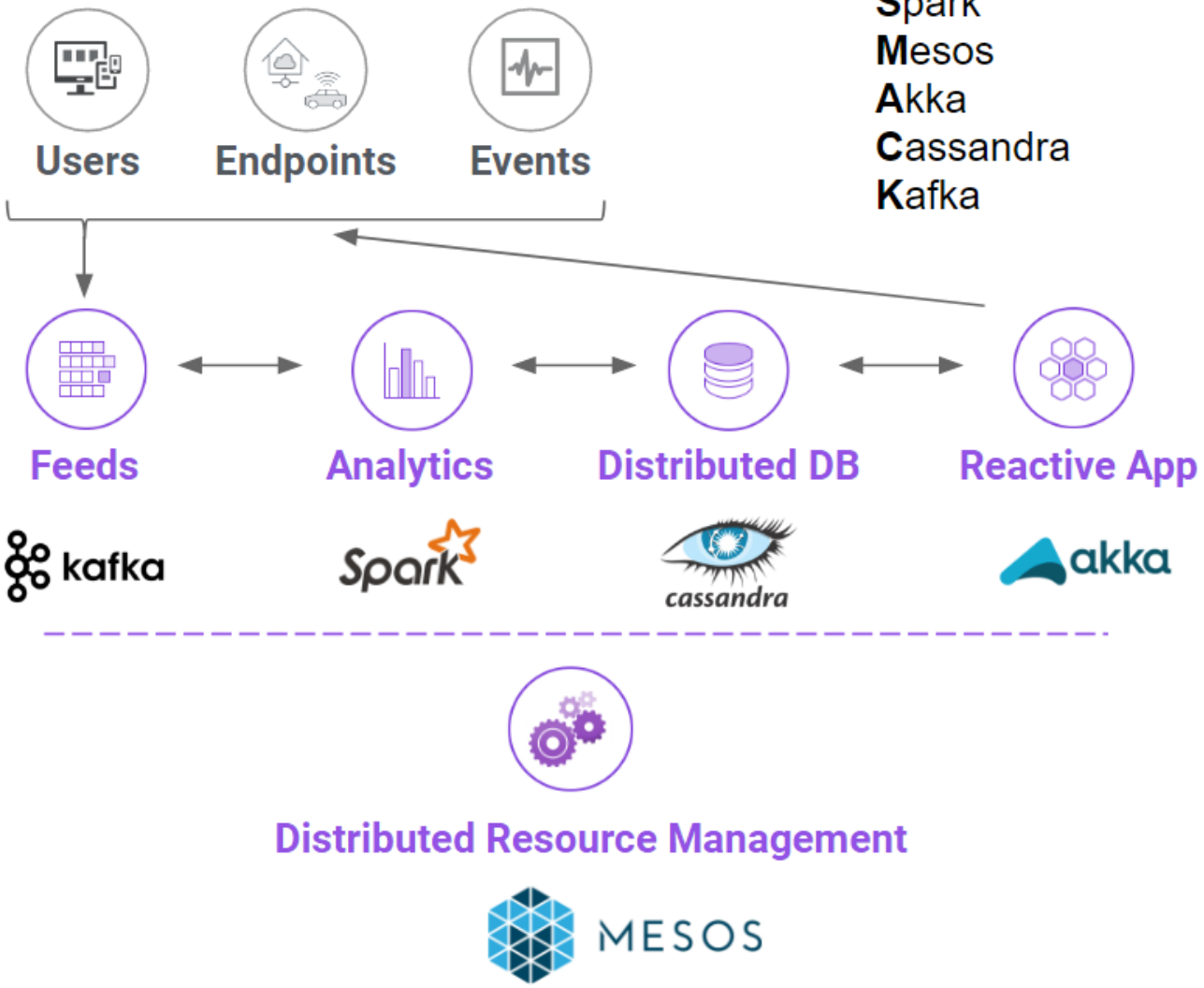
- Sergey Morenets, 2019

Apache Kafka

- ✓ Uses **ZooKeeper** for heath status/cluster membership/routing/configuration management
- ✓ Competes with RabbitMQ/ActiveMQ/ZeroMQ
- ✓ O(1) performance cost
- ✓ Also supports streaming engine (stream processing)
- ✓ Famous customers are LinkedIn, Uber, PayPal, Oracle, Twitter (5 billion sessions per day), Yahoo, Spotify and Netflix
- ✓ Common use-cases are real-time monitoring, real-time analytics and log aggregation

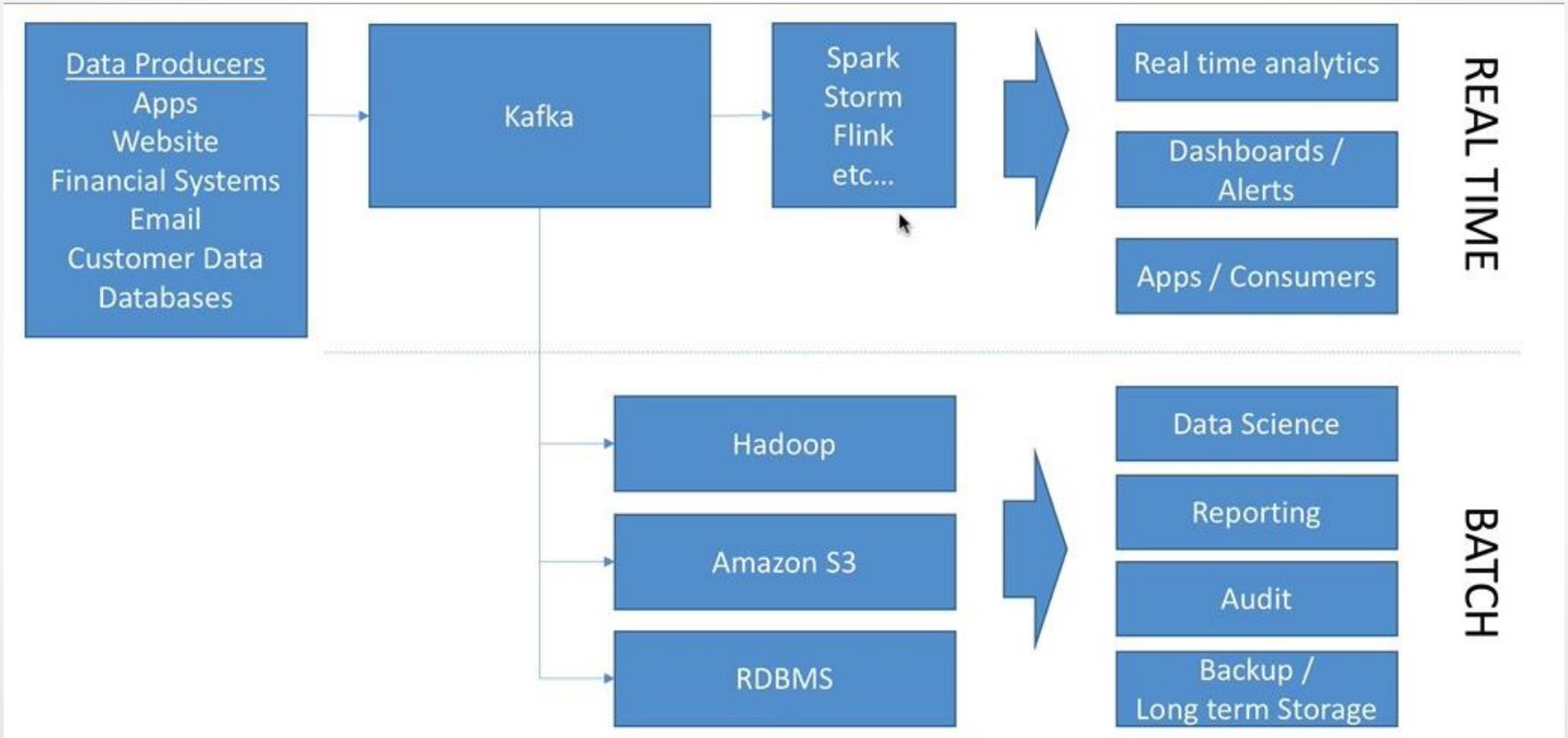
SMACK Stack

Powering scalable real-time & data-driven applications



- Serge

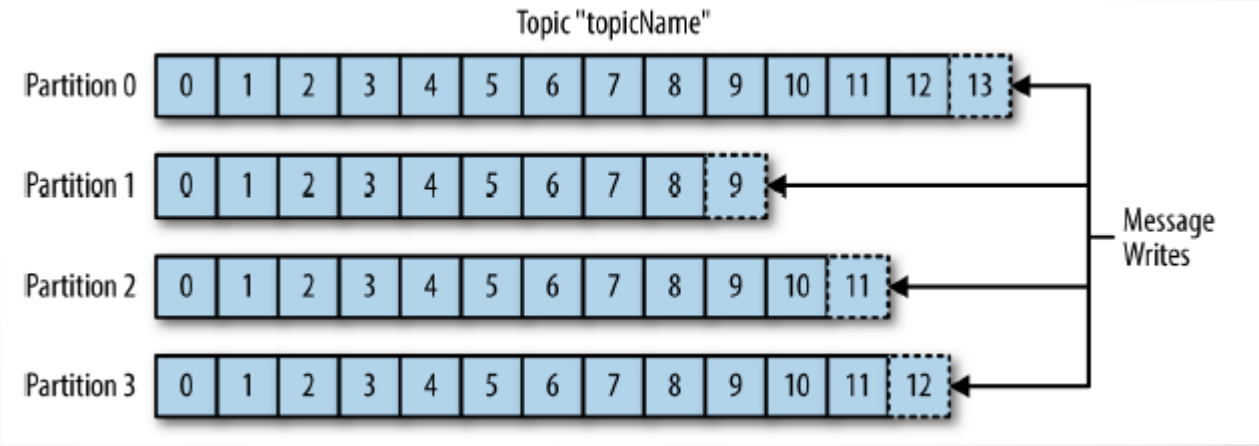
Kafka. Common usage



- Sergey Morenets, 2019

Apache Kafka. Topics

- All Kafka messages are grouped into topics
- Producers publish messages to the topic
- Subscribers subscribe to one or more topic
- Each message has unique sequential identifier (offset) which identify message inside partition



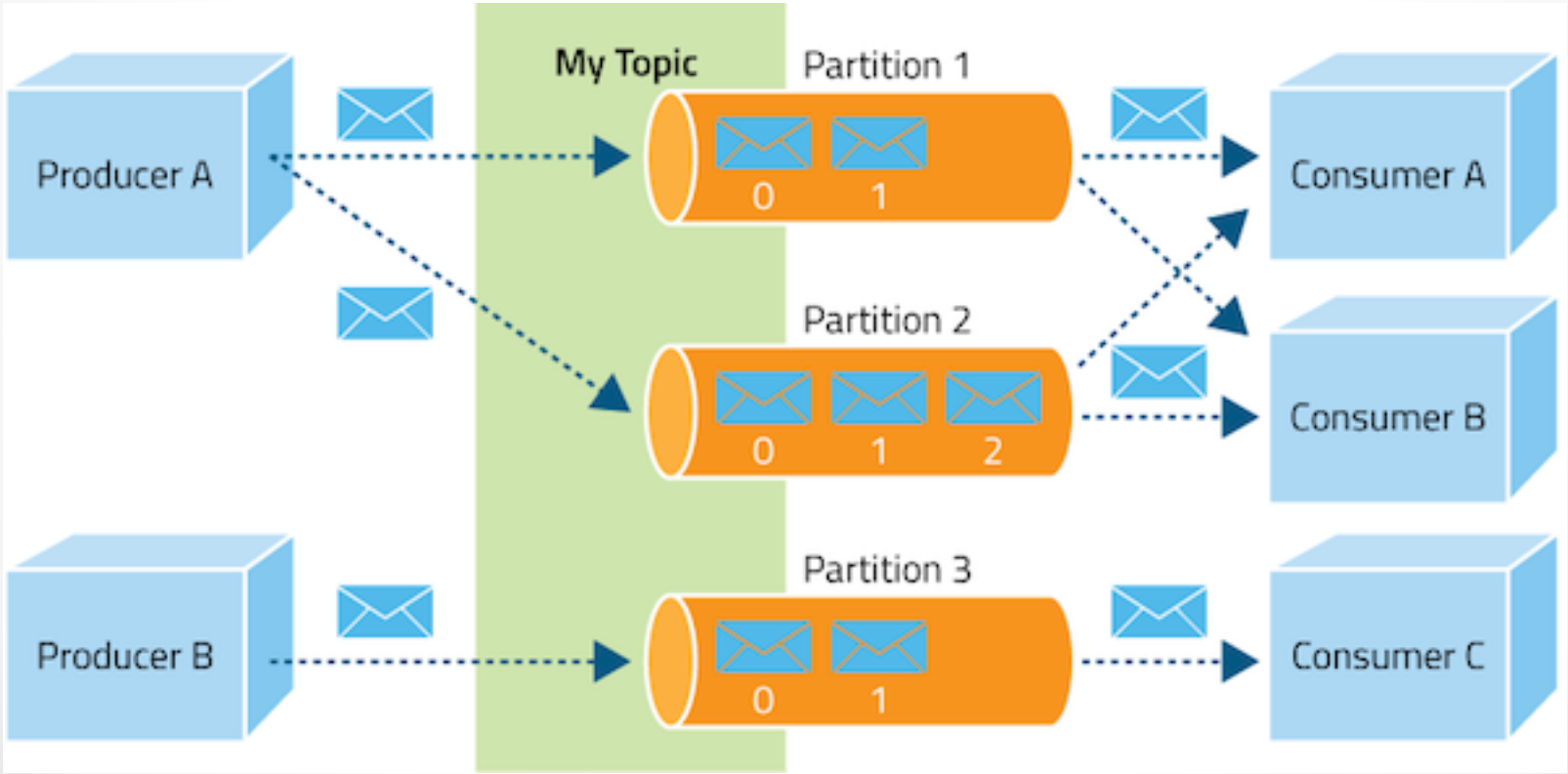
Apache Kafka. Partitions

- Topics are divided into partitions to parallelize topic access for multiple consumers
- Partition allocation is round-robin algorithm or custom function
- Internally partition is a log file that contains group of segment files of the same size
- When message is published broker appends it to the last segment
- Partitions are replicated between the servers for fault tolerance
- You can’t remove partitions for existing topics

Kafka. Messaging & consumers

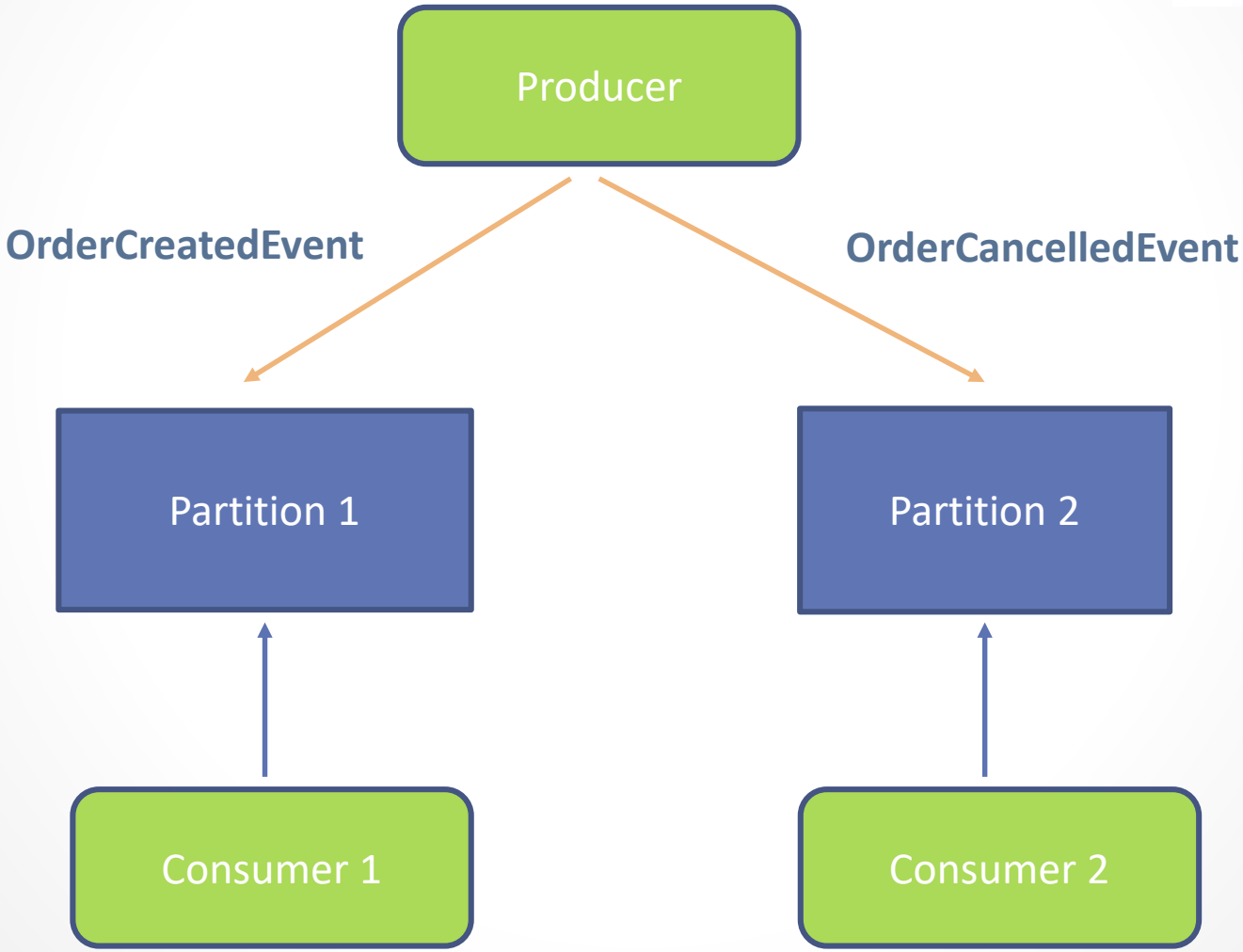
- Each message is key/value pair. Key is used to determine partition
- If key remains the same then message will go to the same partition
- Consumer can read only recent messages or from any offset
- Consumer can read from any partition
- Consumer should specify topic name and broker to subscribe to the topic
- After consumption message is available for re-consumption
- Consumer offsets are stored in a topic __consumer_offets

Kafka. Partitions



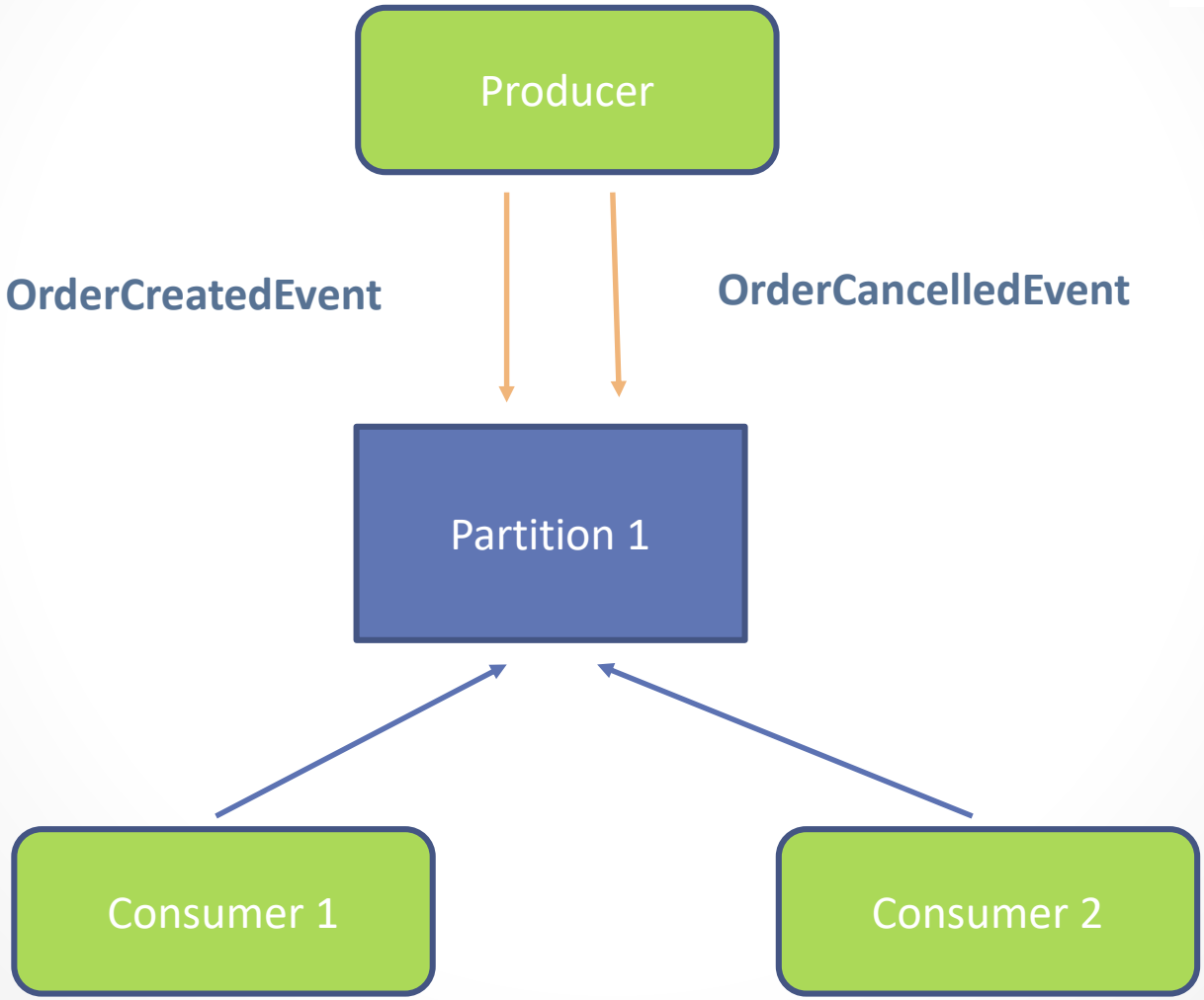
- Sergey Morenets, 2019

Kafka. Partitions



- Sergey Morenets, 2019

Apache Kafka. Partitions



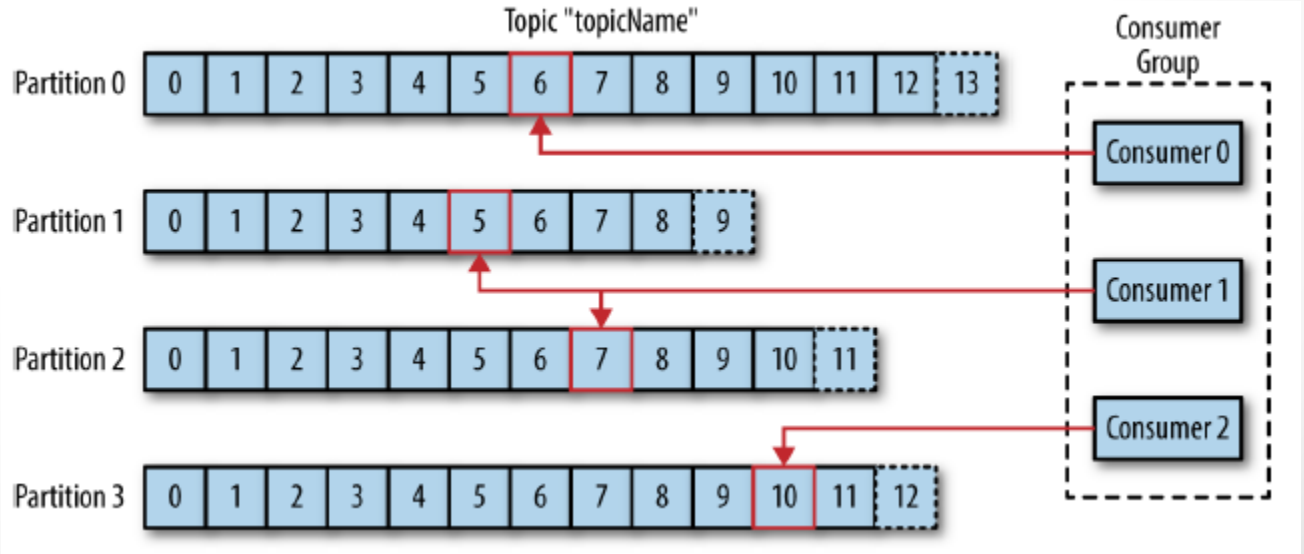
- Sergey Morenets, 2019

Partition trade-offs

- ✓ Each partition provides 10MB/sec throughput
- ✓ Large numbers of partitions can impact Zookeper overhead
- ✓ Complex message ordering (if some global order is required)
- ✓ Long leader fail-over time
- ✓ More partitions lead to greater number of open files and greater latency for replication
- ✓ Usually topic requires no more than 10 partitions

Apache Kafka. Consumer groups

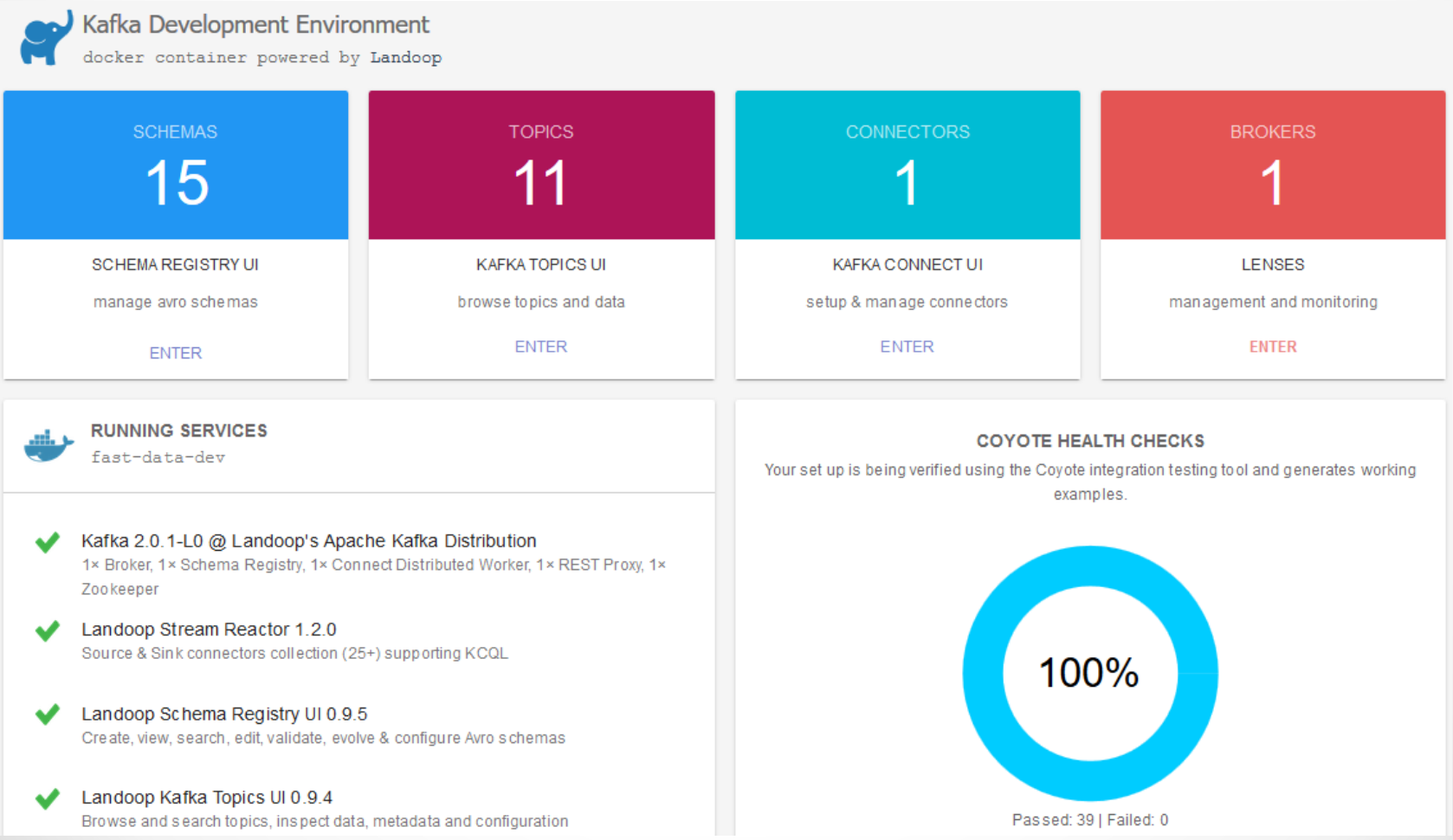
- ✓ Each partition is consumed by one group member
- ✓ Consumer can scale to handle more messages
- ✓ If a consumer fails remaining consumer rebalance partitions
- ✓ Numbers of partitions >= number of consumers



Kafka. Retention & compression

- ✓ Message are stored in the log for specific amount of time (7 days) or till the topic reaches critical size (1GB)
- ✓ After that messages are expired and deleted
- ✓ Individual topics can have their own retention settings
- ✓ You can setup topic compression by specifying compression.type setting (gzip, lz4, **producer**, uncompressed)
- ✓ Consumer will uncompress the data
- ✓ Compression is relevant for text-based messages (JSON) and if you use message batching

Practicing. Landoop



- Sergey Morenets, 2019

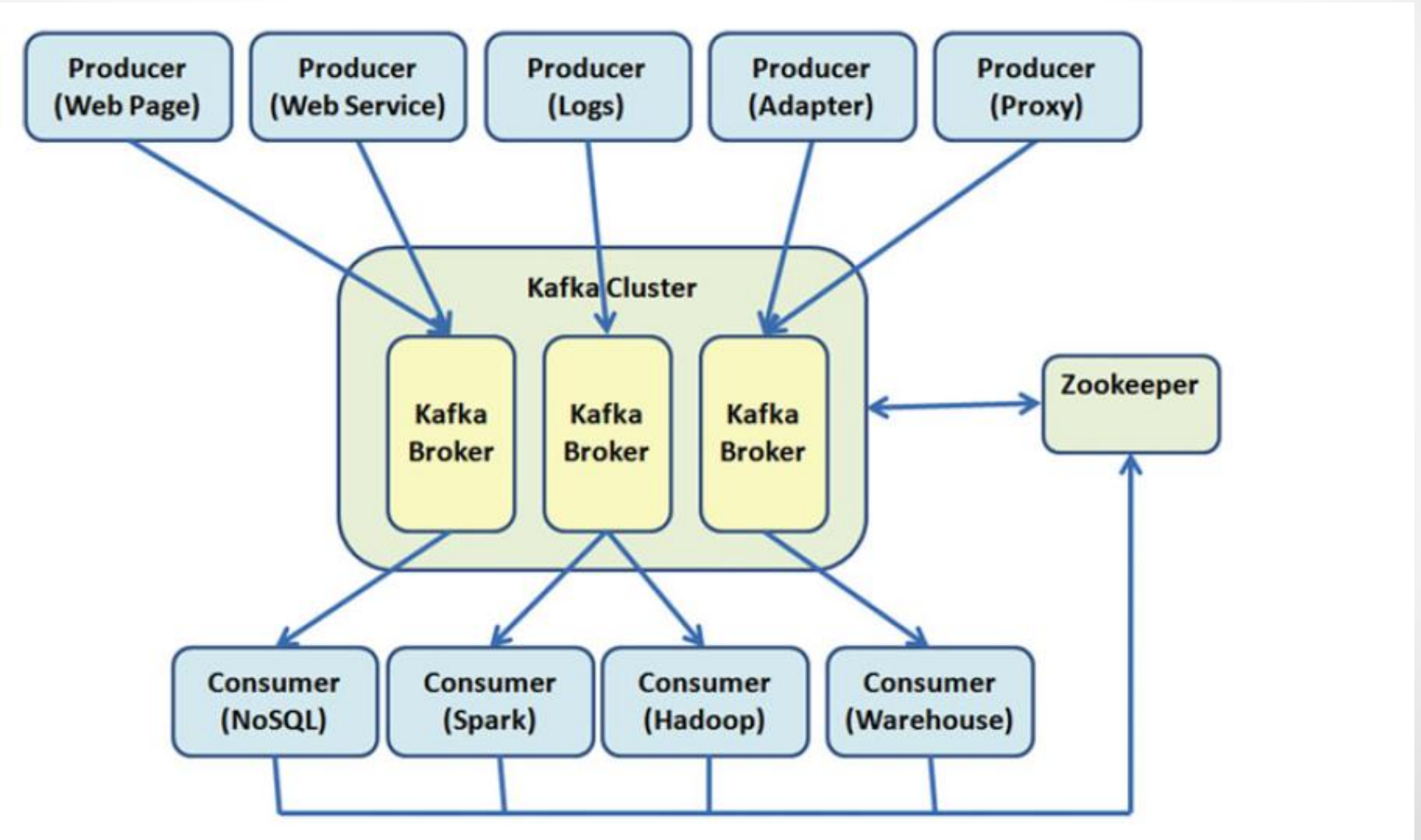
Task 6. Apache Kafka

1. Run Kafka-dev container using Docker. Review Docker log and verify that container starts properly.
2. Open Kafka Dashboard UI. Wait until health-check completes. How many brokers/topics do you have?
3. Enter Kafka Topics UI. Review system and non-system topics. Which serialization format is used for the messages?
4. Click _schemas topic and review messages and their keys/values

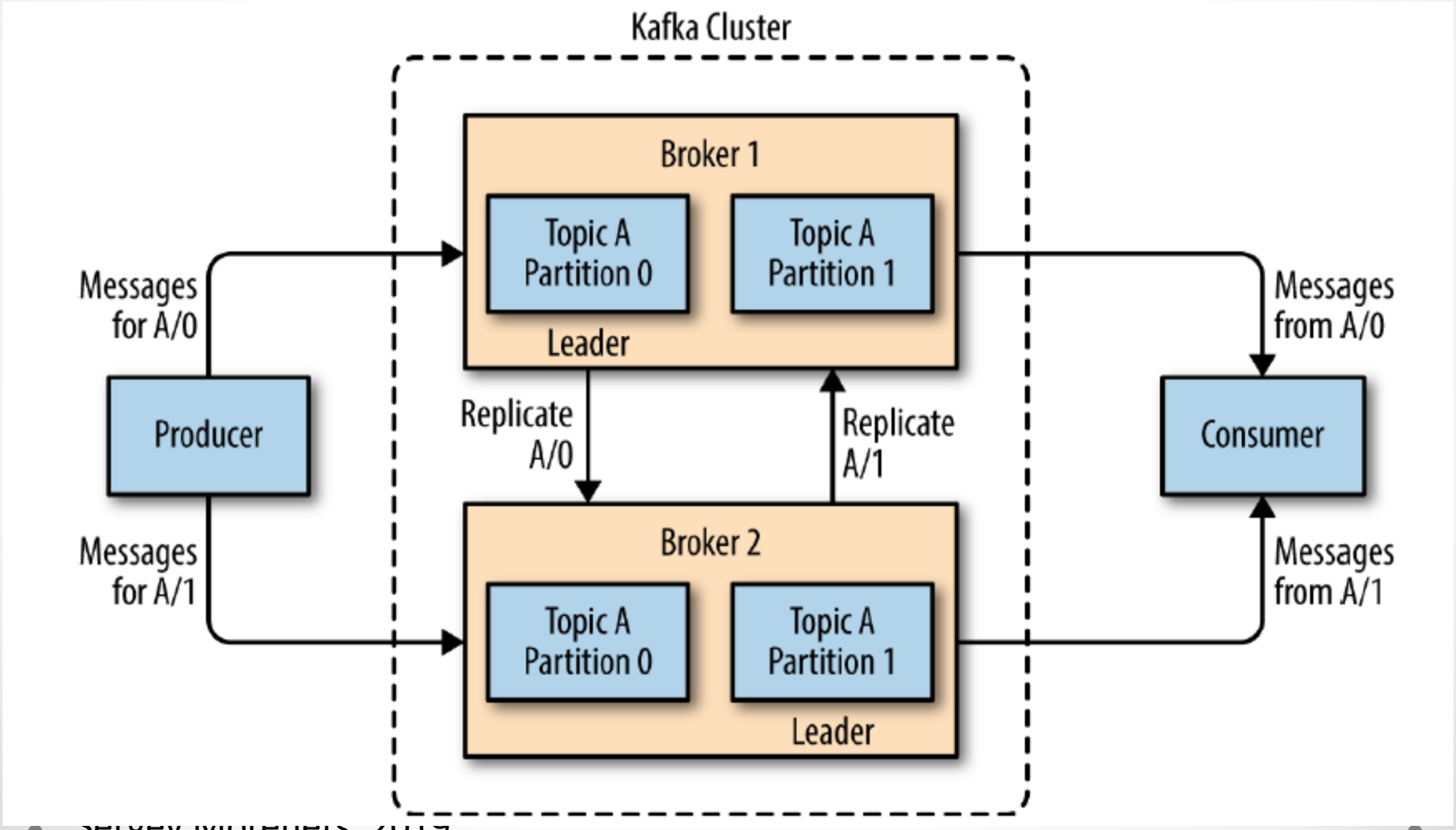
Kafka. Brokers

- Usually Kafka works inside clusters which contains servers (brokers)
- Each broker has numeric identifier
- Recommended 3 brokers per cluster for start
- Clients connects to a broker and other brokers get available for it
- Brokers are useful if topics have replication factor > 1 (recommended 2 or 3)

Kafka clustering



Apache Kafka. Replication



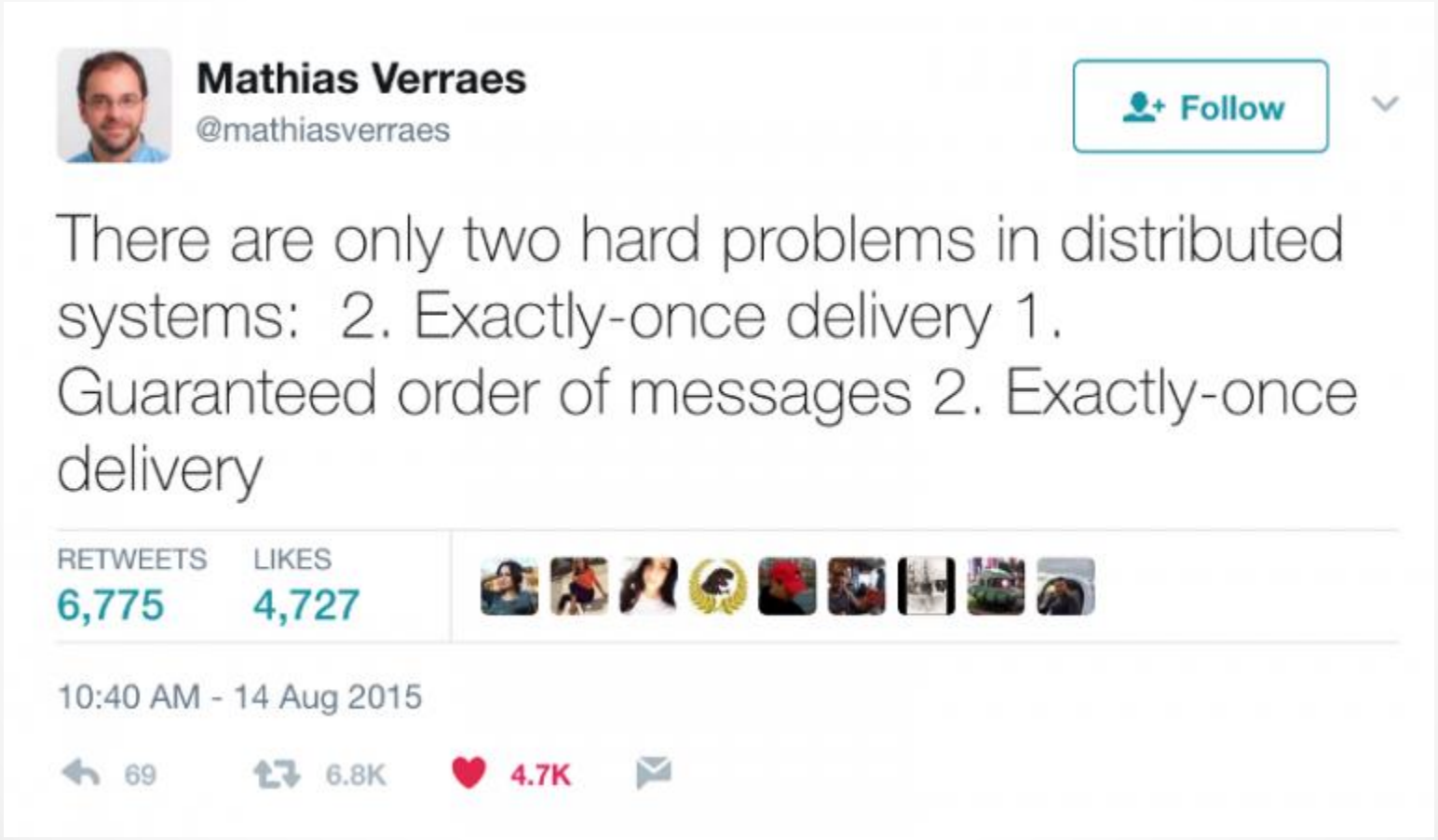
Kafka. Consistency and availability

- 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
- If producer doesn't wait for ack(the fastest approach) then it can lead to data loss. If producer waits for ack from leader then we random data loss can happen. If producer waits for ack from leader/replices then no data loss can happen
- Message cannot be lost if at least one replica is alive
- For efficiency, messages are written and consumed in batches and also compressed

Kafka. Message delivery

- Kafka has three modes to confirm delivery
- **At most once** mode persists consumer offset once message is received. If an error occurs during message processing then message will be lost
- **At least once** mode persists consumer offset once message has been processed. If an error occurs during message processing then message will be read again and in that case you will have to provide custom logic to avoid duplication
- **Exactly once** mode is hard achieve

Kafka. Message delivery



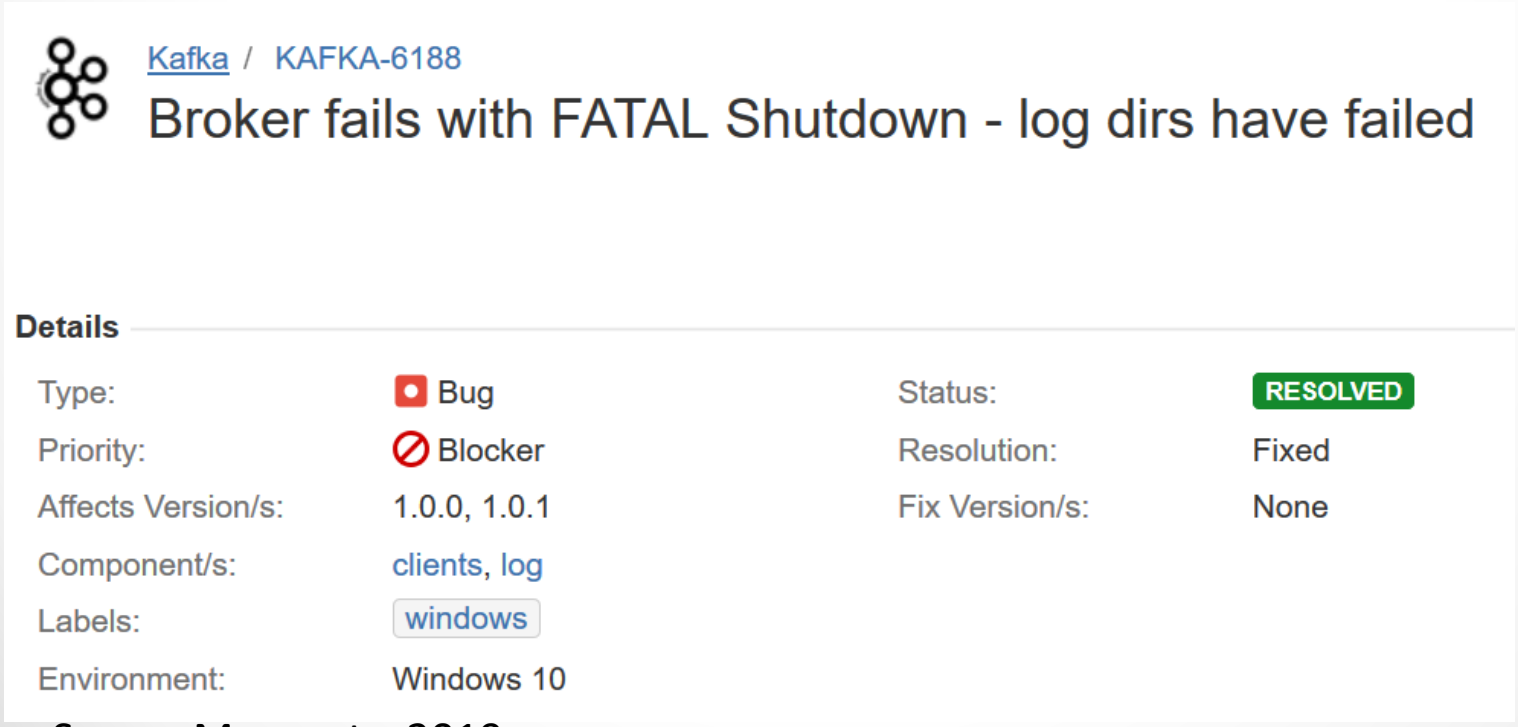
- Sergey Morenets, 2019

Kafka & Zookeeper

- Zookeeper stands for broker registry
- Broker management
- Assists in leader election for partitions
- Sends notification to Kafka (new topic, topic deletion, broker terminates)

Kafka stability

- ✓ Windows is not supported platform
- ✓ Broker may contain up to 4000 partitions and cluster up to 200000 partitions



- Sergey Morenets, 2019

Kafka security

- Security is optional
- You can enable authentication between brokers and clients or between Zookeeper and brokers (SSL or SASL)
- Data encryption is supported
- Authorization for read/write operations

Kafka in production

- ✓ Used by New Relic, Uber, Square, LinkedIn, Netflix, AirBnB
- ✓ Cluster at New Relic processes 15 million of messages per second with aggregate rate 1 TBps
- ✓ Latency less than 10ms
- ✓ Can scale up to 100 brokers
- ✓ Currently processes 1.1 trillion messages per day by 1400 brokers

Kafka. Best practices

- receive.buffer.bytes stores socket buffer size (64 kb). For data-intensive operations it should be increased
- Use wait for acknowledgment mode for producers not to lose messages
- Adjust number of producer **retries** (3 by default) for data-intensive operations
- *buffer.memory* and *batch.size* should be increased for data-intensive operations
- Configure Kafka logging to filter only important events
- Cleanup unused topics
- Use random partitioning unless if you have other project requirements

Task 7. Kafka. Clustering and fault-tolerance

1. Run container with Kafka/Zookeeper
2. Open new terminal and run this command: *docker exec -it spotify_kafka bash*
3. Navigate to /opt/ kafka_2.11-0.10.1.0 folder
4. Copy config/server.properties into server2.properties. Change **broker.id** to 1, also update log.dirs property. Add new line

port=9093

Spring Kafka

- Wrapper over Apache Kafka Client
- Introduced template abstraction (**KafkaTemplate**) for messaging operations
- Supports message-driven approach with @KafkaListener operation
- Transactions support since client 0.11
- Synchronous and asynchronous message delivery
- Requires Spring Framework 5, Spring Integration 3.0 and Kafka client 0.11

Spring Kafka. Dependencies


```
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-clients</artifactId>
  <version>2.0.1</version>
  <scope>compile</scope>
</dependency>
```

Pure Java libary

Spring integration

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

Spring Kafka. Listeners


```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());
    }
```

Requires to listen to the messages

application.yml

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

Which offset to use if initial offset doesn't exist

Also exception, latest

Spring Kafka. Listeners & partitions


```
@KafkaListener(id = "myGroupId",
        topicPartitions = { @TopicPartition(
                topic = "data1", partitions = { "0", "2" }),
        @TopicPartition(topic = "data2",
        partitions = "1",
        partitionOffsets = @PartitionOffset(
                partition = "1", initialOffset = "100")) })
public void listen(ConsumerRecord<?, ?> record) {
```

- Sergey Morenets, 2019

Spring Kafka. Metadata


```
@KafkaListener(topics = "orders")
public void listen(@Payload String item,
        @Header(KafkaHeaders.RECEIVED_MESSAGE_KEY) Integer key,
        @Header(KafkaHeaders.RECEIVED_PARTITION_ID) int partition,
        @Header(KafkaHeaders.RECEIVED_TOPIC) String topic,
        @Header(KafkaHeaders.RECEIVED_TIMESTAMP) long ts,
        Acknowledgment ack) {
    try {

    } finally {
        ack.acknowledge();
    }
}
```

Manual message acknowledgment, all the previous messages in the partition assumed to be processed

Spring Kafka. Batch processing


```
@Bean
public KafkaListenerContainerFactory<?> batchFactory() {
    ConcurrentKafkaListenerContainerFactory<Integer, String>
            factory = new ConcurrentKafkaListenerContainerFactory<>();
    factory.setConsumerFactory(consumerFactory());
    factory.setBatchListener(true);
    return factory;
}
```

```
@KafkaListener(topics = "orders",
        containerFactory = "batchFactory")
public void listen(List<Message<?>> messages,
        Acknowledgment ack) {
}
```

- Sergey Morenets, 2019

KafkaTemplate. Producers


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

@PostMapping
public void send() {
    template.send("orders", "message");
    template.send("orders", "key", "message_with_key");
    template.send("orders2", 0, "key2", "message2_with_key");

    template.flush();
}
```

Topic name

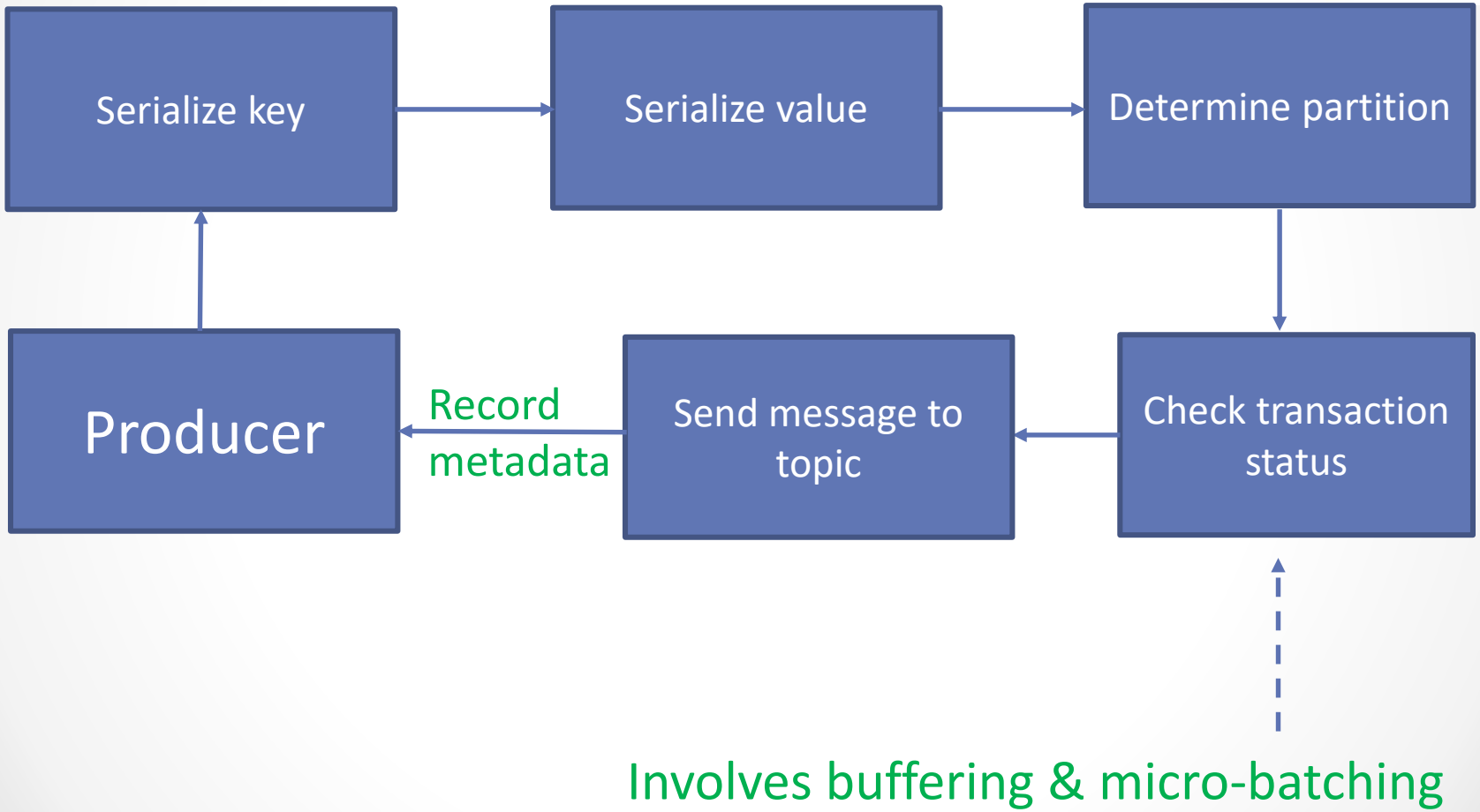
Async sending

Partition

```
var future = template.send("orders", 1,
        "key3", "message3_with_key");
future.addCallback(result -> System.out.println(result),
        ex -> Log.error(ex.getMessage(), ex));
```

Success callback

Sending message flow



- Sergey Morenets, 2019

Partitioning


```
public class RandomPartitioner implements Partitioner{

    @Override
    public int partition(String topic, Object key,
            byte[] keyBytes, Object value, byte[] valueBytes,
            Cluster cluster) {
        if(key == null) {
            return 0;
        } else {
            int numPartitions = cluster.
                    partitionCountForTopic(topic);
            return key.hashCode() % numPartitions;
        }
    }
```

Kafka. Producer configuration


```
@Configuration
public class KafkaConfiguration {
    @Bean
    public ProducerFactory<String, String> producerFactory() {
        Map<String, Object> configProps = new HashMap<>();
        configProps.put(
          ProducerConfig.PARTITIONER_CLASS_CONFIG,
          RandomPartitioner.class);
        return new DefaultKafkaProducerFactory<>(configProps);
    }

    @Bean
    public KafkaTemplate<String, String> kafkaTemplate() {
        return new KafkaTemplate<>(producerFactory());
    }
}
```

org.apache.kafka.common.config.ConfigException: Missing required configuration "key.serializer" which has no default value.

Kafka. Producer configuration


```
Map<String, Object> configProps = new HashMap<>();
configProps.put(
  ProducerConfig.PARTITIONER_CLASS_CONFIG,
  RandomPartitioner.class);
configProps.put(
        ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
        StringSerializer.class);
configProps.put(
        ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
        StringSerializer.class);
configProps.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
        "localhost:9092");
return new DefaultKafkaProducerFactory<>(configProps);
```

Kafka. Producer configuration


```
@Bean
public ProducerFactory<String, String> producerFactory() {
    Map<String, Object> configProps = new HashMap<>();
    configProps.put(
      ProducerConfig.PARTITIONER_CLASS_CONFIG,
      RandomPartitioner.class);
    configProps.put(
            ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
            StringSerializer.class);
    configProps.put(
            ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
            StringSerializer.class);
    return new DefaultKafkaProducerFactory<>(configProps);
}
```

org.apache.kafka.common.config.ConfigException: No resolvable bootstrap urls given in bootstrap.servers

Kafka. Producer & custom value

```
var future = template.send("orders", 0,
        "key3", new Book());
```

java.lang.ClassCastException: it.discovery.microservice.book.Book cannot be cast to java.base/java.lang.String

```
Map<String, Object> configProps = new HashMap<>();
configProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
        StringSerializer.class);
configProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
        JsonSerializer.class);
```

Uses Jackson serializer

Message delivery

application.yml

```
spring:
  kafka:
    producer:
      retries: 3
      acks: 1
```

Controls acknowledgment of the message:

0 – no confirmation from leader
1 – confirmation from leader
all – confirmation after successful replication

Testing Kafka applications

- Start embedded Kafka/Zookeeper servers on random port before each tests

```
<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka-test</artifactId>
    <scope>test</scope>
</dependency>
```

Testing Kafka applications


```
@ExtendWith(SpringExtension.class)
@EmbeddedKafka(topics = { "topic1", "topic2" },
               partitions = 1)
public class KafkaTest {
    @Autowired
    private EmbeddedKafkaBroker embeddedKafka;
```

Starts embedded Kafka/Zookeeper

Stream processing

- Stream processor is software component that receives and sends stream of data
- Apache Storm, Apache Flink or Kafka Streams

Task 8. 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.

Event sourcing

- ✓ Captures all changes to an application state as a sequence of events

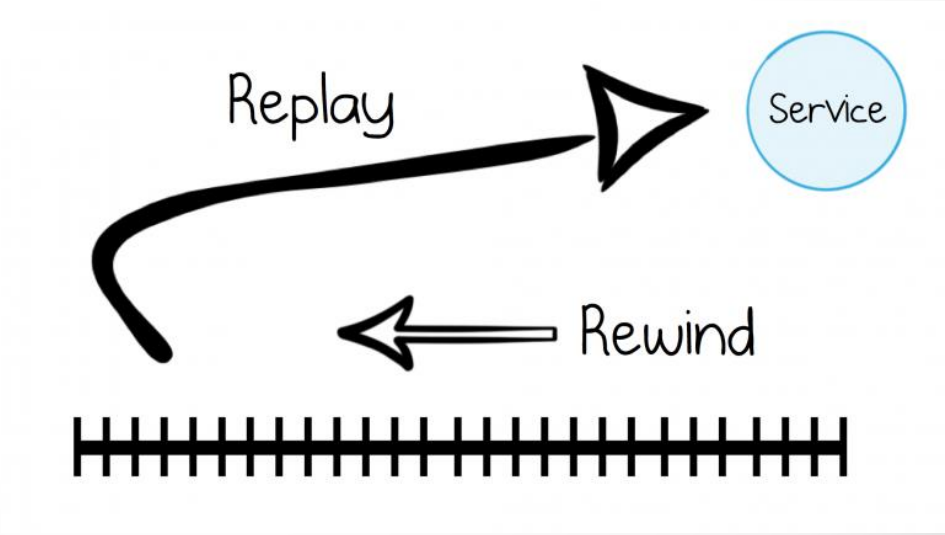
- Sergey Morenets, 2019

Event sourcing

- ✓ Enterprise design pattern
- ✓ Allows to see state changes (at some period of time)
- ✓ Every state change is stored in an event object
- ✓ Each name has a meaningful name
- ✓ All changes to the domain object are initiated by the event objects
- ✓ Past events are immutable

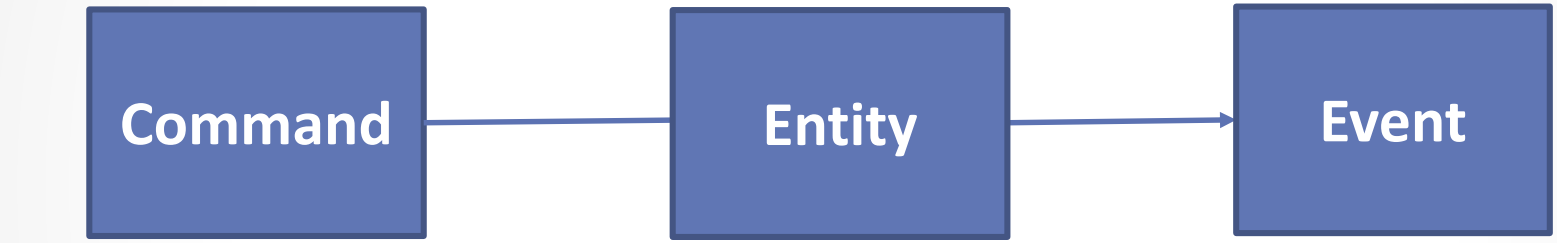
Event sourcing

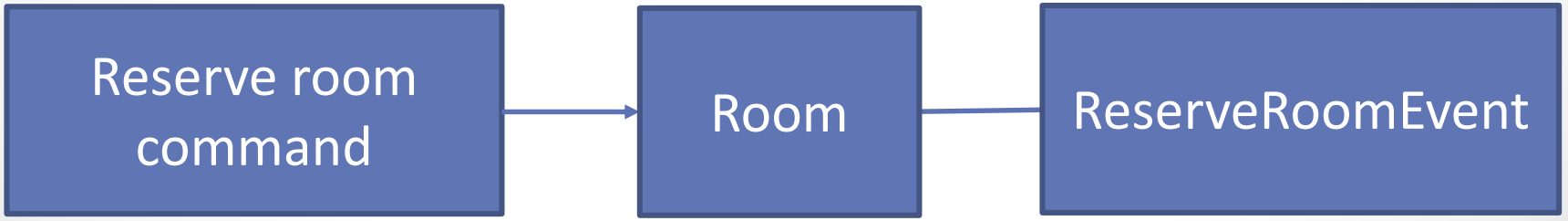
- ✓ You can cache temporal application state in memory
- ✓ UI requires projects of all the event log
- ✓ Simplifies data consistency
- ✓ You cannot remove event types
- ✓ You can replay events log from production and reproduce a bug



- Sergey Morenets, 2019

Event sourcing





- Sergey Morenets, 2019

Commands


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

    private String name;

    private String email;
}

@Value
public class WithdrawMoneyCommand {
    private int id;

    private double amount;
}
```

Domain entity. Commands


```java
@Data
public class Customer {
    private int id;

    private String name;

    private String email;

    private double balance;

    public List<BaseEvent> process(CreateCustomerCommand cmd) {
        return Arrays.asList(new CustomerCreatedEvent(cmd.getId(),
                cmd.getName(), cmd.getEmail()));
    }
}
```

Domain entity. Commands

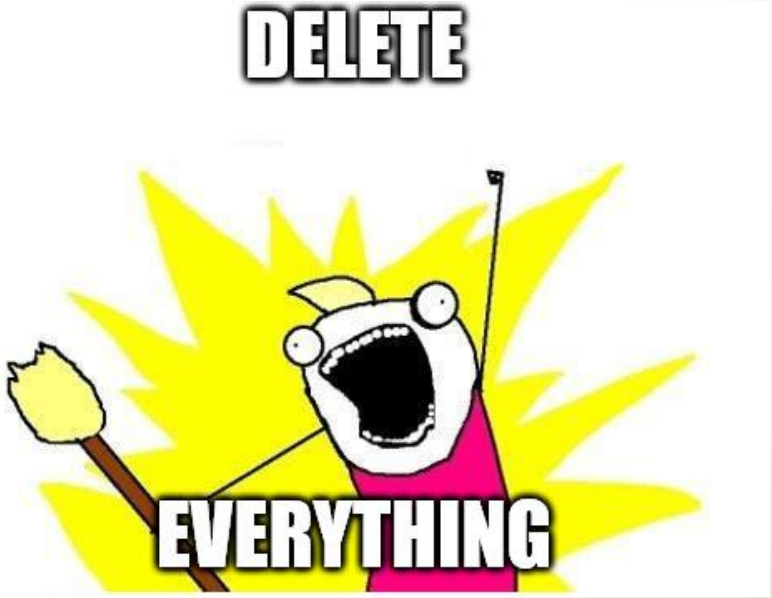

```
public List<BaseEvent> process(WithdrawMoneyCommand cmd) {
    if (cmd.getAmount() > balance) {
        return Arrays.asList(
                new WithdrawMoneyEvent(cmd.getId(),
                        cmd.getAmount()));
    } else {
        return Arrays.asList(
                new WithdrawMoneyFailureEvent(
                        cmd.getId(), "Insufficient funds"));
    }
}
```

Domain entity. Events


```
public void apply(CustomerCreatedEvent event) {
    this.id = event.getId();
    this.name = event.getName();
    this.email = event.getEmail();
}

public void apply(WithdrawMoneyEvent event) {
    this.balance -= event.getAmount();
}
```

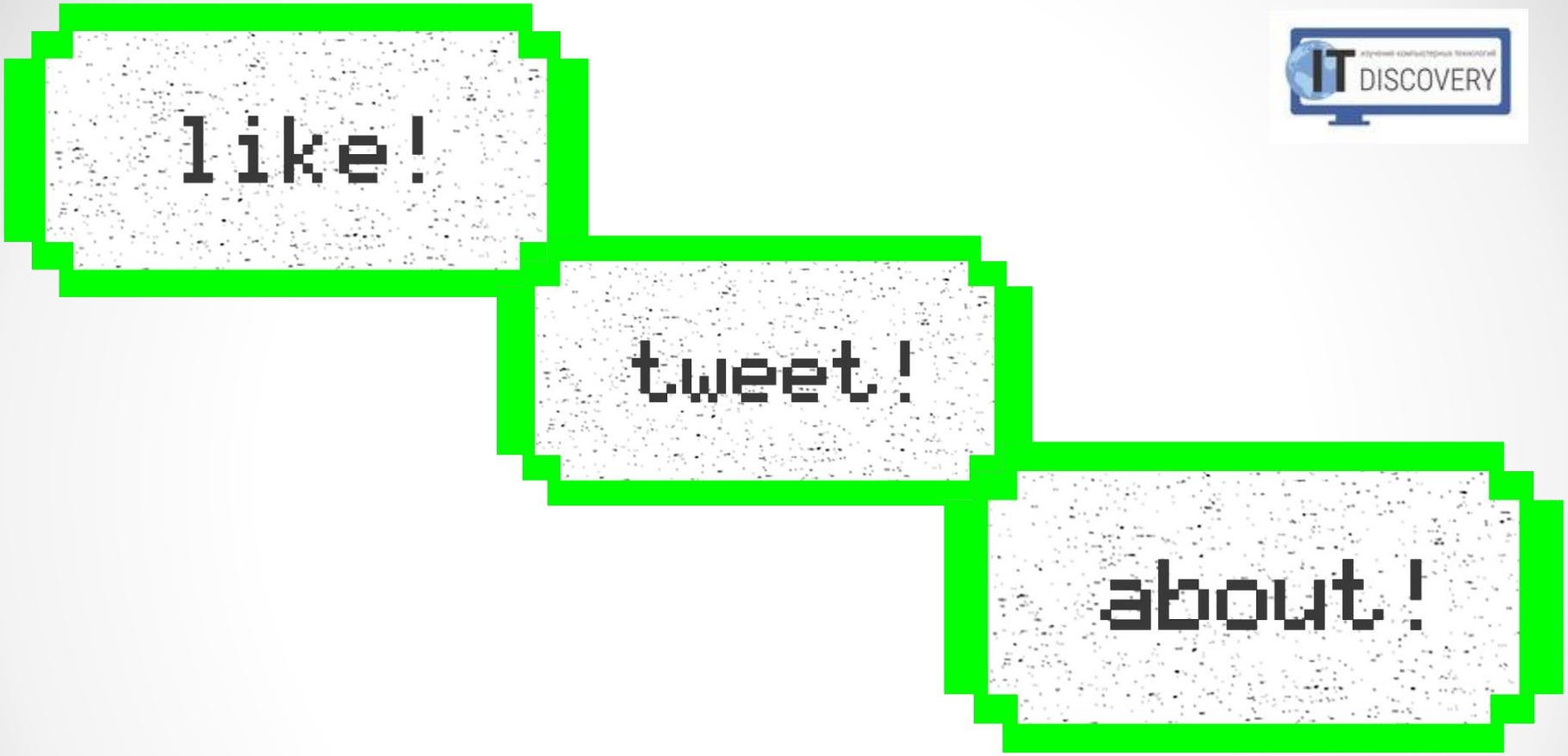
Event purge/compaction

- ✓ Storage is not free and not unlimited
- ✓ Security purposes
- ✓ GDPR rules
- ✓ Compaction means keeping most recent data



Task 9. Event sourcing

1. Create command classes that will start any change in the application event.
2. Update controllers/services to use these commands
3. Update **Order** entity to react to the specified commands, update state and generate events
4. Create entity **EventLog** that will store all the generated events (event type, event payload, creation date)



✓ Sergey Morenets, sergey.morenets@gmail.com