WELCOME
&
THANK YOU

Reactive & Event Driven Programming

Level 1

MEET YOUR BITTY BYTE TEAM

TANGY F.

CEO

EDDIE K.

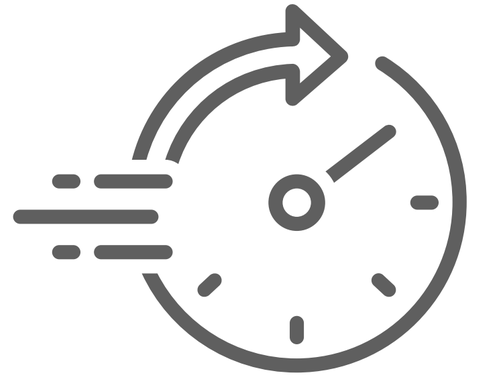
DEVELOPER

WILLIAM D.

DEVELOPER

Rapid Review

TRIVIA

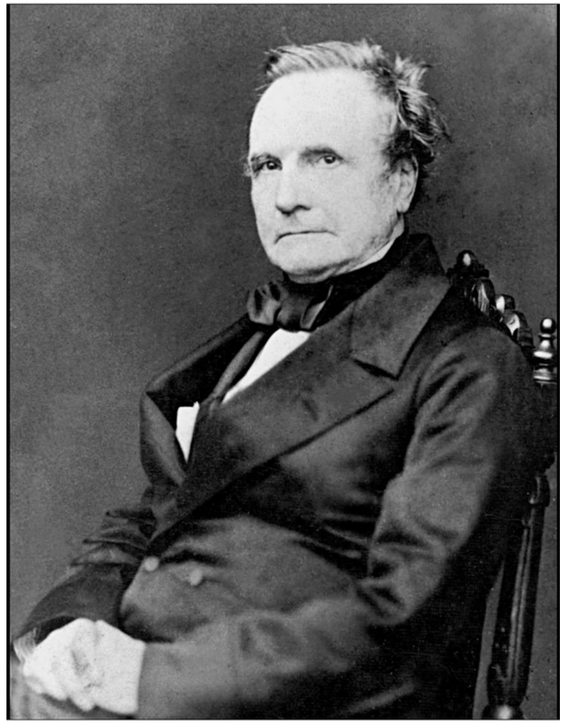


Rapid Trivia

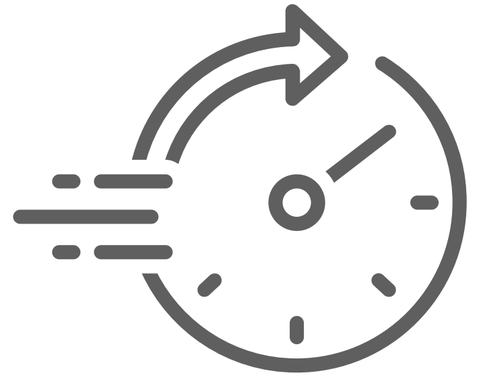
Who was Charles Babbage?

Rapid Review

TRIVIA

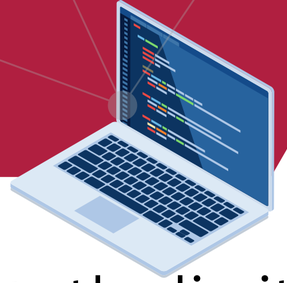


Charles Babbage 1791 – 1871



Babbage originated the concept of digital programable computer. Considered to be the father of computers. Created a mechanical computer called the Difference Engine

CODING EXERCISE



On the reactive Flux that sends integers from 1 to 50, change the limit from 50 to 10, add a filter to include items greater than 5 as follow;

Make the filter to produce this output

```
Sending 1
Receiving 1
Sending 2
Receiving 2
Sending 3
Receiving 3
Sending 4
Receiving 4
Sending 5
Receiving 5
5
Sending 6
Receiving 6
6
Sending 7
Receiving 7
7
Sending 8
Receiving 8
8
Sending 9
Receiving 9
9
Sending 10
Receiving 10
10
```

Change the filter to make this output

```
Sending 1
Sending 2
Sending 3
Sending 4
Sending 5
Receiving 5
5
Sending 6
Receiving 6
6
Sending 7
Receiving 7
7
Sending 8
Receiving 8
8
Sending 9
Receiving 9
9
Sending 10
Receiving 10
10
```

Change the filter again to make this output

```
Sending 5
Receiving 5
5
Sending 6
Receiving 6
6
Sending 7
Receiving 7
7
Sending 8
Receiving 8
8
Sending 9
Receiving 9
9
Sending 10
Receiving 10
10
```

Let's see the code.

TODAY'S AGENDA

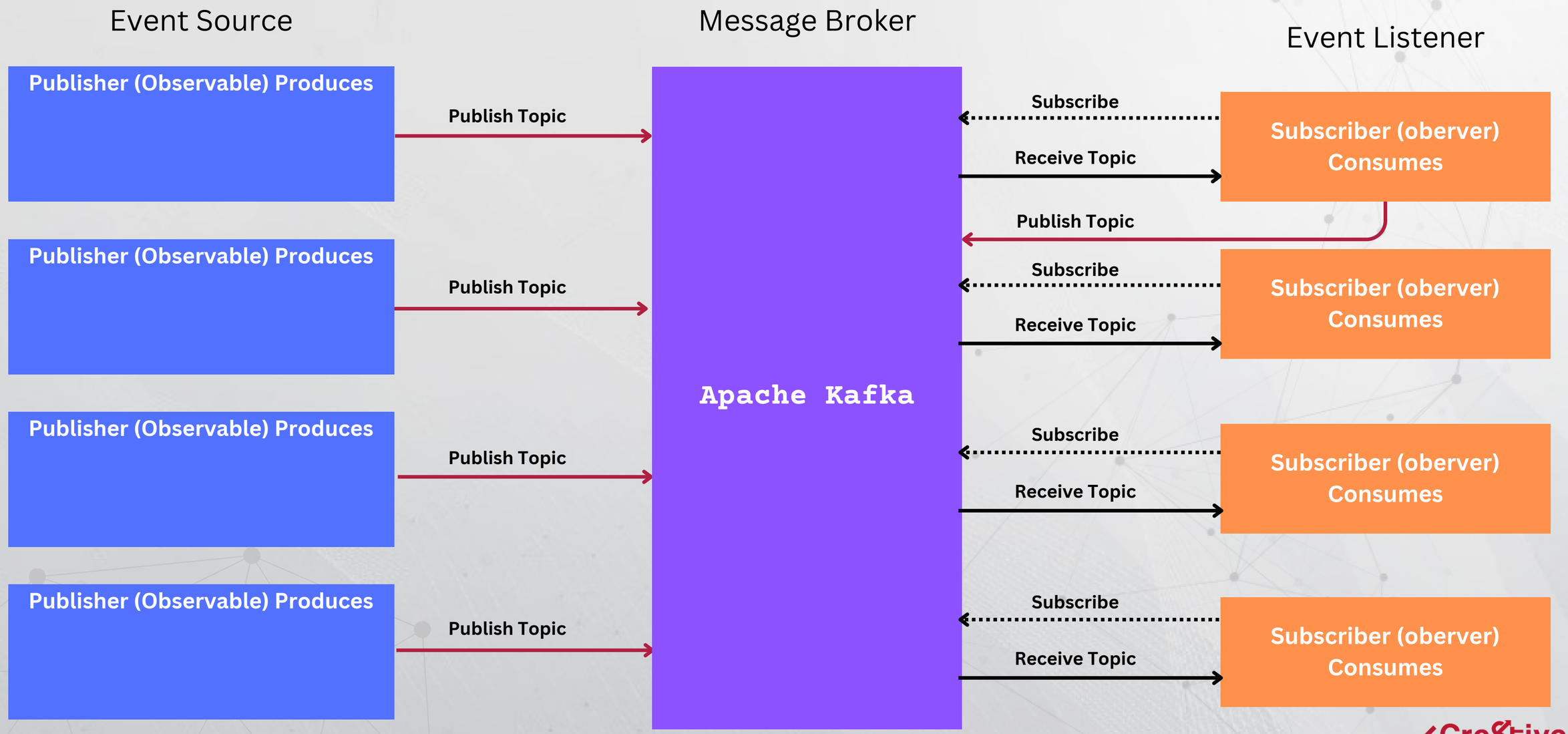
DAY 4

1. **Subscribing and consuming event with Kafka**
   - Within Main Class
   - Within Spring Boot Rest API
2. **Backpressure Handling**

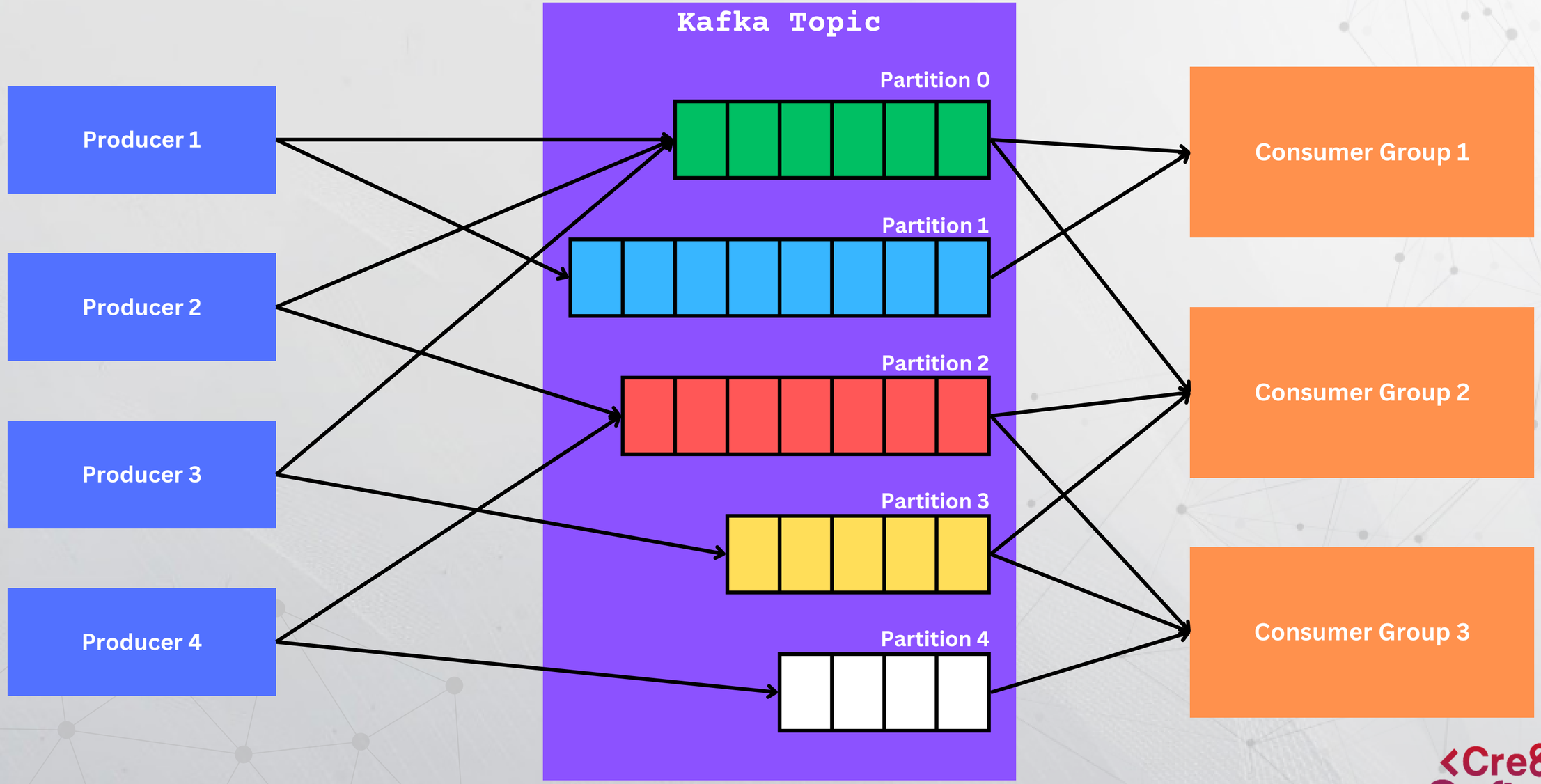
DAY 4
LESSON 1

Subscribing and consuming event with Kafka

MESSAGE DRIVEN WITH KAFTKA



KAFKA TOPICS AND PARTITONS



KAFKA WITHIN A MAIN CLASS

Using **KafkaProducer** to produce

Setup Properties

```
Properties producerProps = new Properties();
// bootstrap.servers=localhost:9092
producerProps.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
// key.serializer=org.apache.kafka.common.serialization.StringSerializer
producerProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
// value.serializer=org.apache.kafka.common.serialization.StringSerializer
producerProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
```

Instantiate KafkaProducer and pass properties

```
KafkaProducer<String, String> producer = new KafkaProducer<>(producerProps);
```

Instantiate ProduceRecord, pass topic and data, then send using ProduceRecord.send().get()

```
ProducerRecord<String, String> record = new ProducerRecord<>(TOPIC, value:"Hello, Kafka2!");
RecordMetadata metadata = producer.send(record).get();
```

KAFKA WITHIN A MAIN CLASS

Using **KafkaConsumer** to receive

Setup Properties

```
Properties consumerProps = new Properties();
// bootstrap.servers=localhost:9092
consumerProps.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
// group.id=group_id
consumerProps.put(ConsumerConfig.GROUP_ID_CONFIG, value:"group_id");
// key.serializer=org.apache.kafka.common.serialization.StringSerializer
consumerProps.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());
// value.serializer=org.apache.kafka.common.serialization.StringSerializer
consumerProps.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());
```

Instantiate a KafkaConsumer and subscribe to the topic

```
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(consumerProps);
consumer.subscribe(Collections.singletonList(TOPIC));
```

Use KafkaCOnsumer.poll() to recieve

```
ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(millis:100));
for (ConsumerRecord<String, String> record : records) {
    System.out.println("Received message: " + record.value());
}
```

Let's see the code.

KAFKA WITHIN SPRING BOOT

Using **KafkaTemplate** to produce

Instantiate a KafkaTemplate object.

```
@Autowired
private KafkaTemplate<String, Greeting> greetingKafkaTemplate;
```

Execute KafkaTemplate.send(). Pass topic name and payload as parameter.

```
greetingKafkaTemplate.send(greetingTopicName, greetingMessage);
```

For consuming use **@KafkaListener** annotation in a Spring **@Component** class

Provide topic, groupId and contianer factory to @KafkaListener

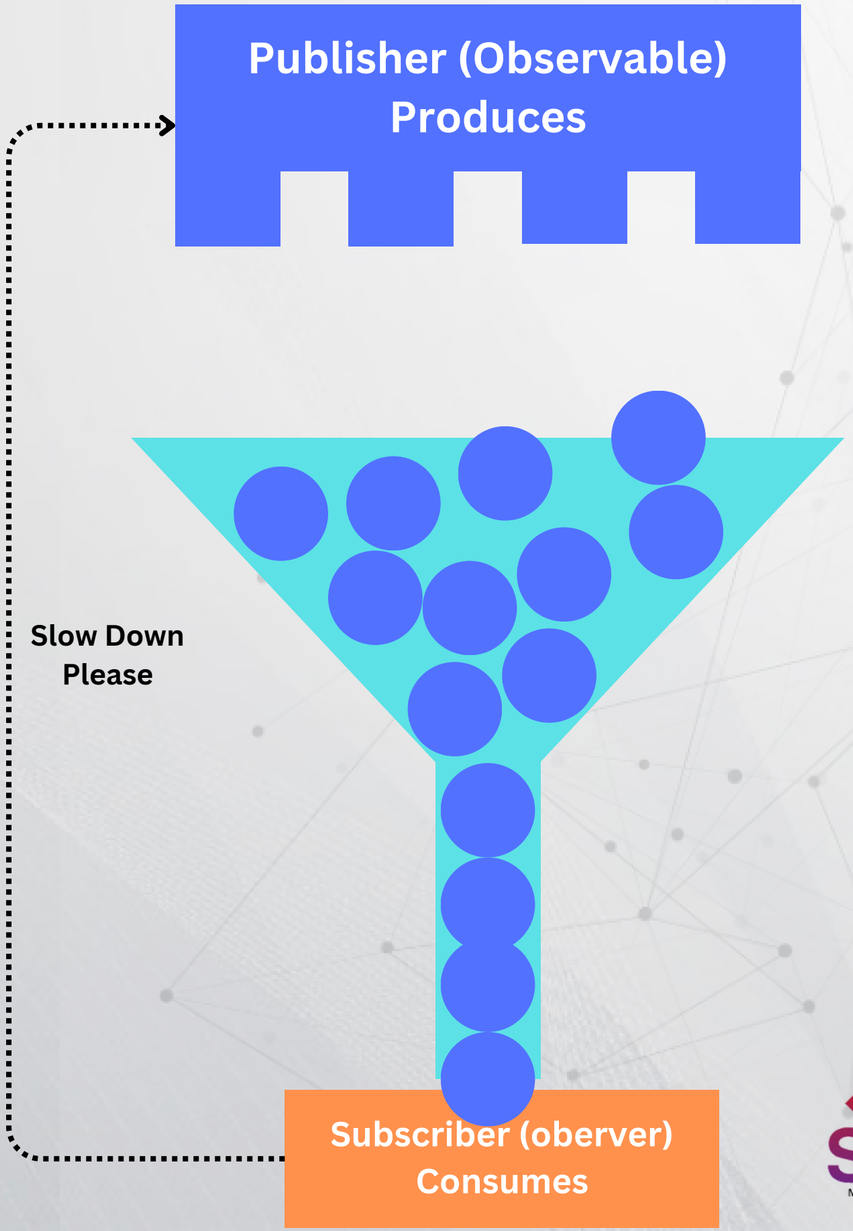
```
@KafkaListener(topics = "${greeting.topic.name}",groupId = "${greeting.group.id}",
                    containerFactory = "greetingKafkaListenerContainerFactory")
public void receive(Greeting greeting) {
    LOGGER.info(format:"received payload='{}'", greeting.toString());
    payload = greeting.toString();
    latch.countDown();
}
```

Let's see the code.

DAY 4
LESSON 2

Backpressure

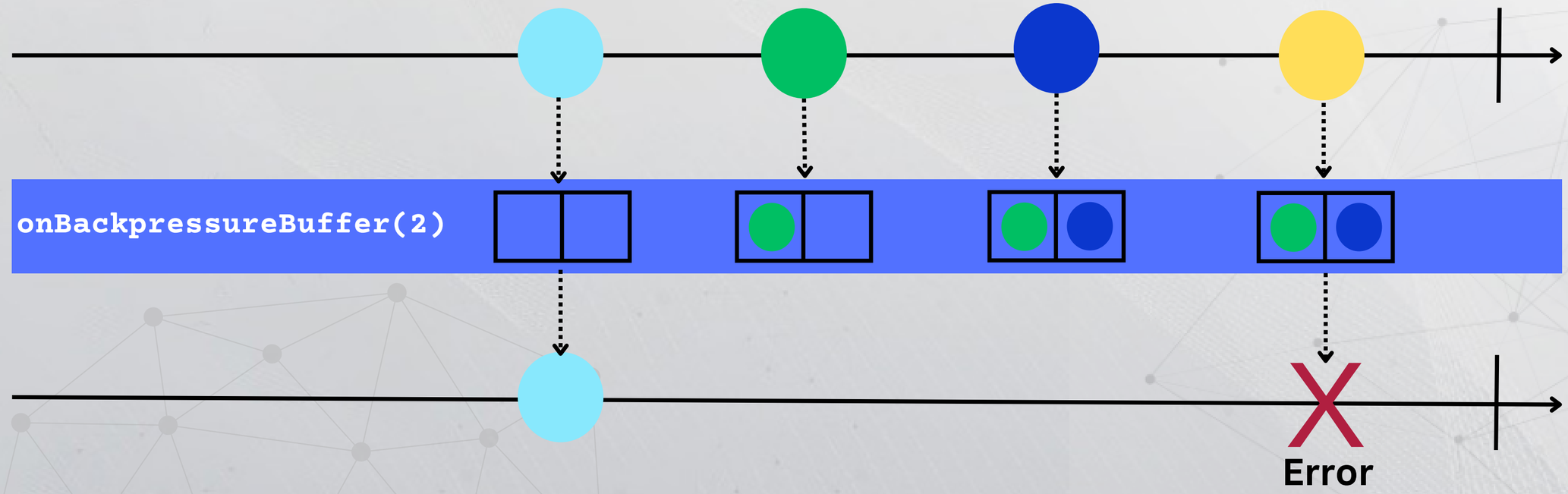
BACKPRESSURE



BACKPRESSURE

.onBackpressureBuffer(100)

```
Flux.range(start:0, count:50)
    .onBackpressureBuffer(maxSize:100);
```

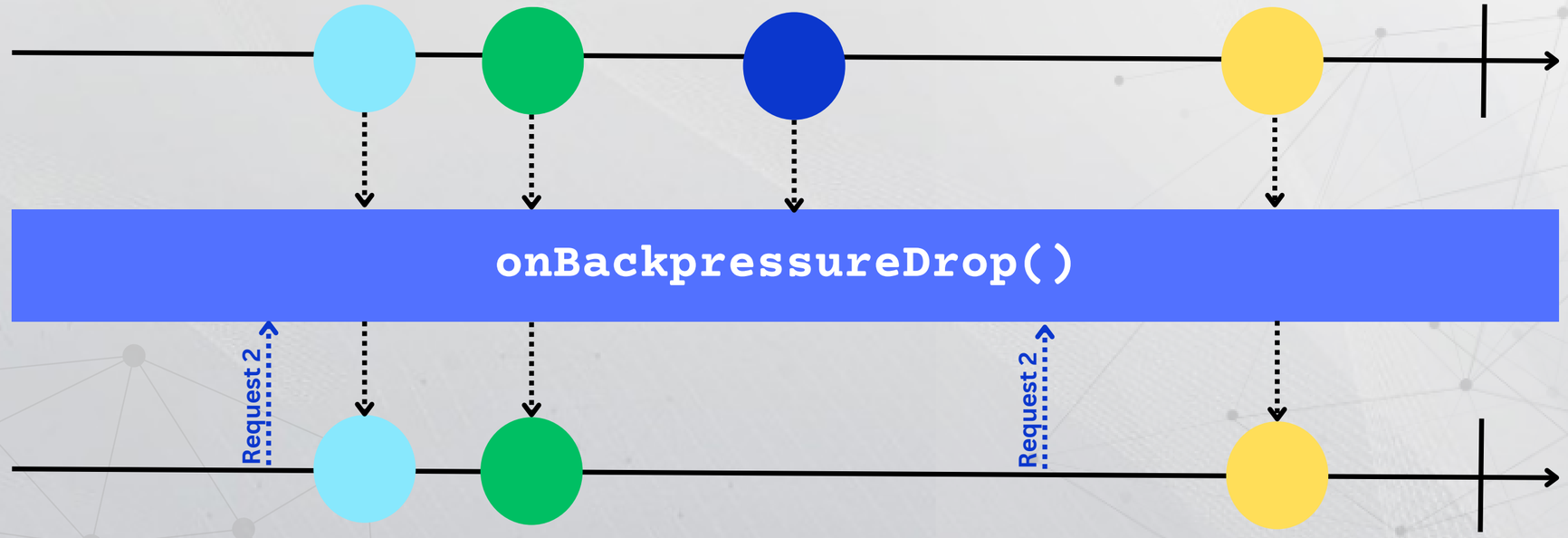


Let's see the code.

BACKPRESSURE

.drop()

```
Flux.range(start:0, count:50)
    .onBackpressureDrop();
```

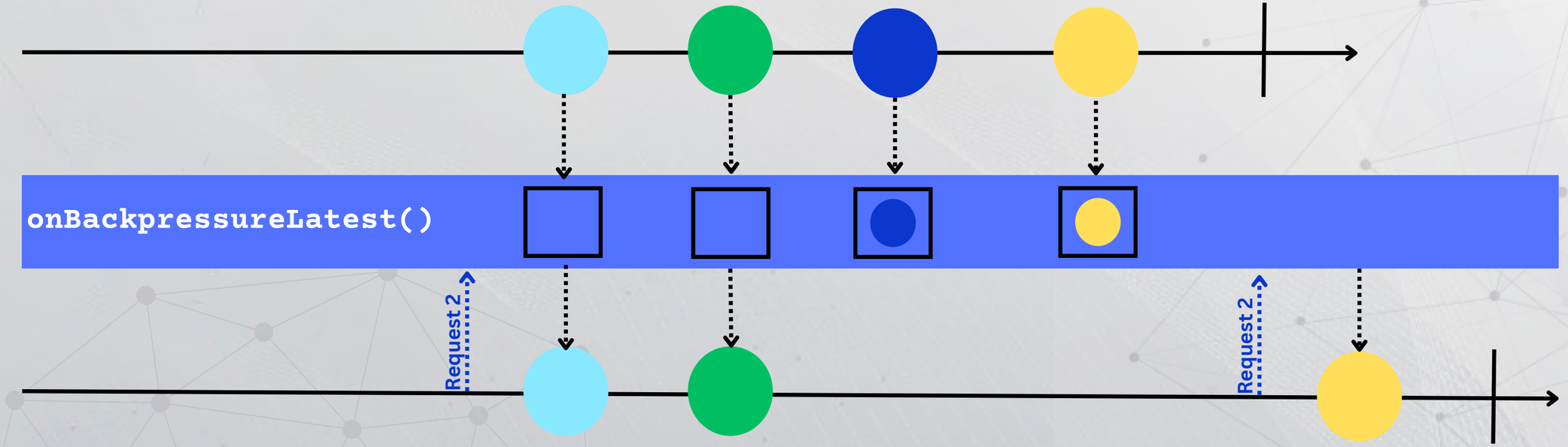


Let's see the code.

BACKPRESSURE

.latest()

```
Flux.range(start:0, count:50)
    .onBackpressureLatest();
```



Let's see the code.

LESSON QUESTION

In the code shown below, in which statement does the producer will start sending the collection items to its consumer? Provide line number or statement.

```
2   Iterable<Integer> collection = List.of(1, 2, 3, 4, 5);
3
4   Flux<Integer> flux = Flux.fromIterable(collection);
5
6   flux.subscribe(
7           item -> System.out.println("Received: " + item),
8           error -> System.err.println("Error: " + error),
9           () -> System.out.println("Processing completed")
10  );
```

LESSON ANSWER

Line 6 on **flux.subscribe()**

```
2   Iterable<Integer> collection = List.of(1, 2, 3, 4, 5);
3
4   Flux<Integer> flux = Flux.fromIterable(collection);
5
6   flux.subscribe(
7           item -> System.out.println("Received: " + item),
8           error -> System.err.println("Error: " + error),
9           () -> System.out.println("Processing completed")
10  );
```

CODING EXERCISE

Time to try it yourself. Write the following class and run it. It will produce an error. To correct, add the appropriate backpressure strategy that will produce numbers indefinitely from 0 to n. You will need to stop the execution manually.

```java
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;
import java.time.Duration;

public class Practice4a {
    Run | Debug
    public static void main(String[] args) throws InterruptedException {

        Flux.interval(Duration.ofMillis(millis:1))
            .concatMap(a -> Mono.delay(Duration.ofMillis(millis:100)).thenReturn(a))
            .doOnNext(item -> System.out.println("Sending " + item))
            .blockLast();
    }
}
```

Let's see the code.

THANK YOU

Bitty Byte

We will see you soon