SPRING IO

#Springio19

Building resilient scheduling in distributed systems with Spring

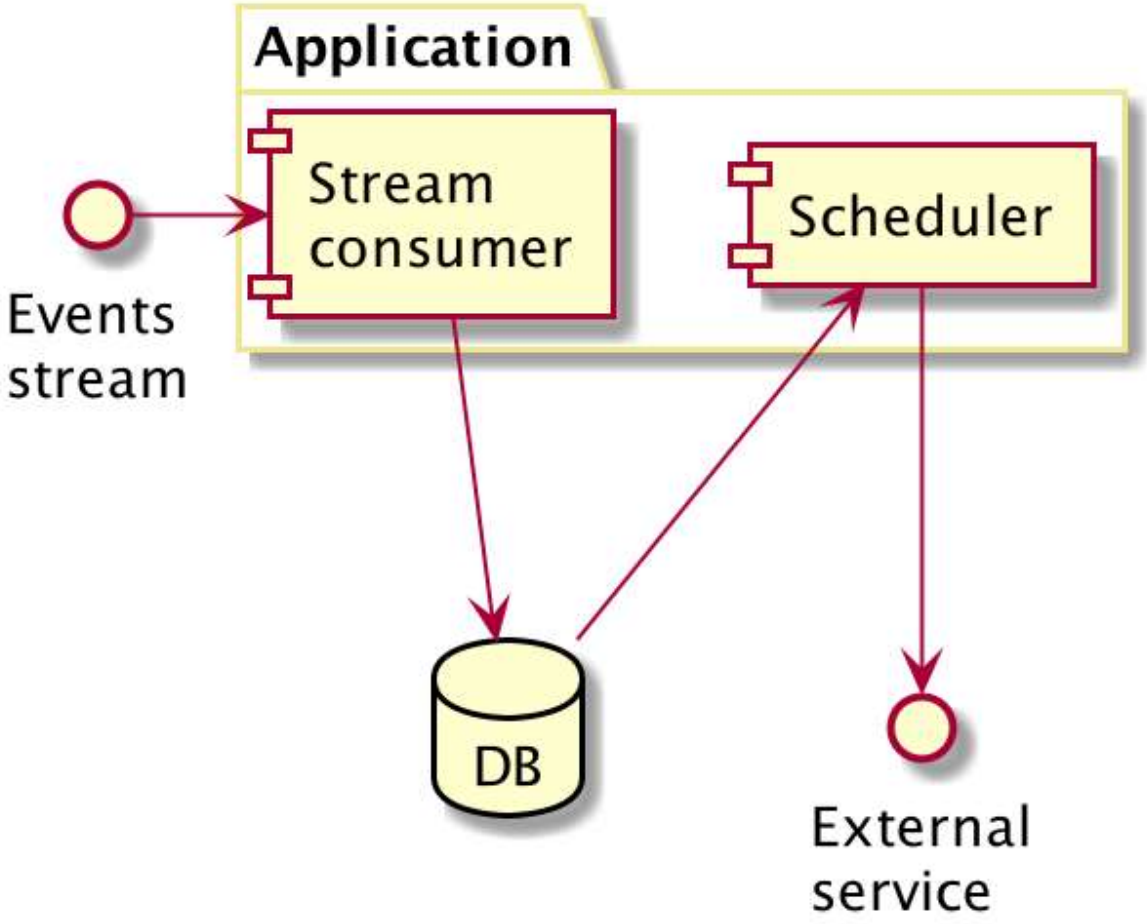
Marek Jeszka
@logic_marc

Agenda

- Background
- Approach
- Results
- Conclusion

Use-case

- asynchronous communication
- reliable processing
- better visibility


Application
Stream consumer
Scheduler
Events stream
DB
External service

```
@Component
public class SimpleService {

    @Scheduled(cron = "0 * * * * *")
    public void runQuiteOften() {
                // process events
    }
}
```

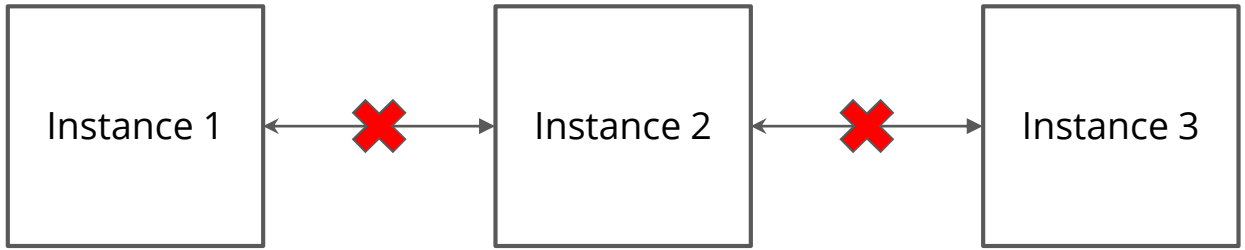
Distributed Systems

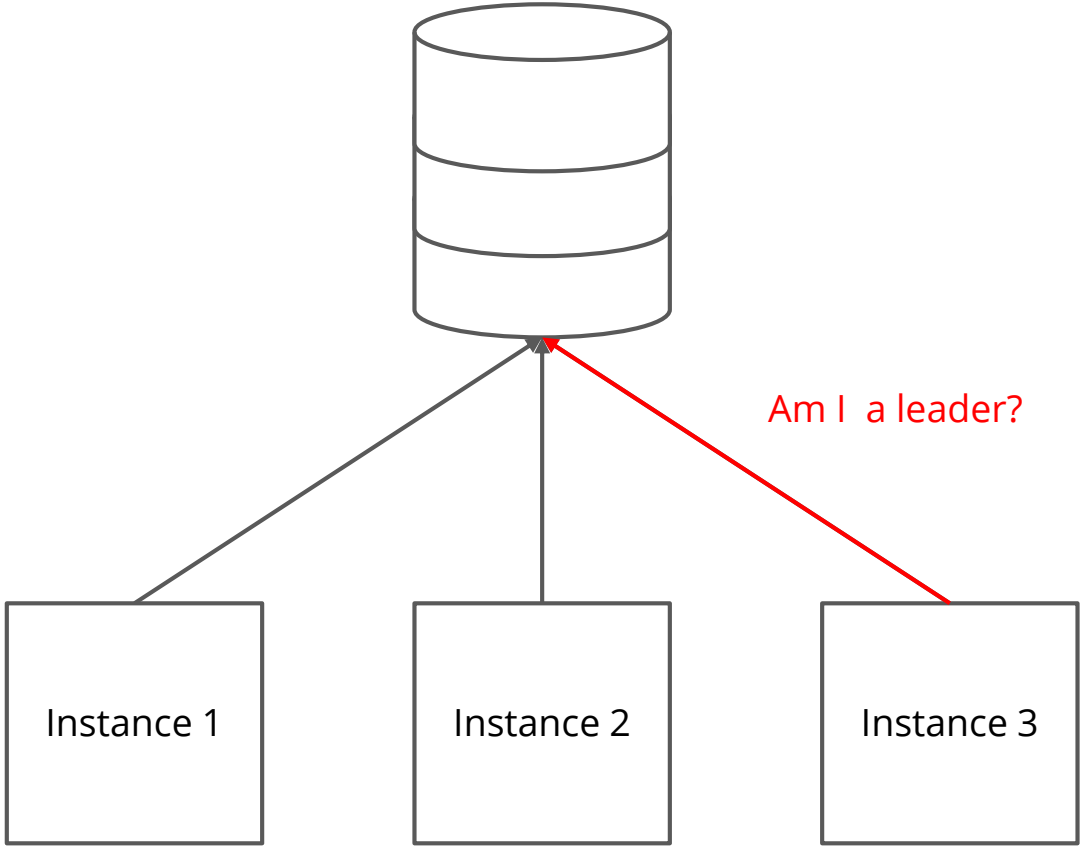
- vertical scaling
  - actually it doesn’t scale…
- horizontal scaling
  - cost-efficiency
  - higher reliability
  - easier to expand

N26

Running on a single node

- How to select the node?
- Where to keep information about the selected node?


Instance 1
Instance 2
Instance 3


Am I a leader?
Instance 1
Instance 2
Instance 3

Leader election in Spring-based application

```
@Component
public class SimpleService {

    @RunIfLeader
    @Scheduled(cron = "0 * * * * *")
    public void runQuiteOften() {
        // process events
    }
```

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.METHOD })
public @interface RunIfLeader {
}
```

Aspect Oriented Programming with Spring

- Aspect - *crosscutting* concern
  - Logging
  - Transaction management
- Enabled with dependency:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

Types of advices

- ~~After~~
- Around
- ~~Before~~

```
@Aspect
public class RunIfLeaderAspect {

    @Around("@annotation(com.n26.RunIfLeader) && execution(void *(..))")
    public void annotatedMethod(ProceedingJoinPoint joinPoint)
      throws Throwable {

        if (isLeader()) {
            joinPoint.proceed();
        }

        // do not execute
    }
```

Why we didn’t like it?

- No clear separation between business and scheduling logic
- Hard to test
- Scheduled jobs spread across the application

Issue with the @SqsListener

```
@SqsListener(value = "eventsQueue",
             deletionPolicy = ON_SUCCESS)
@RunIfLeader
void onEvent(String eventAsJson) {
  // process event
}
```

N26

Selecting a leader in programmatic approach

Programmatic approach

— SchedulingConfigurer from org.springframework.scheduling.annotation

```
public interface SchedulingConfigurer {

    void configureTasks(
        ScheduledTaskRegistrar taskRegistrar);
}
```

```
@Configuration
@EnableScheduling
public class SchedulingConfig implements SchedulingConfigurer {

    @Override
    public void configureTasks(ScheduledTaskRegistrar taskRegistrar) {

        taskRegistrar.addCronTask(
                new CronTask(() -> {
                    // process events
                }, "0 * * * * *"));
    }
```

```
    @Autowired
    private Runnable processEventsTask;

    @Override
    public void configureTasks(ScheduledTaskRegistrar taskRegistrar) {

        taskRegistrar.addCronTask(
            new CronTask(processEventsTask, "0 * * * * *"));
    }


@Component
public class ProcessEventsTask implements Runnable {

    @Override
    public void run() {
        // process events
    }
}
```

What are the benefits of programmatic approach?

- Tasks are scheduled in one place
- Custom executor service

```
@Configuration
@EnableScheduling
public class SchedulingConfig implements SchedulingConfigurer {

    @Override
    public void configureTasks(ScheduledTaskRegistrar taskRegistrar) {
        taskRegistrar.setScheduler(taskScheduler());
    }

    @Bean(destroyMethod = "shutdown")
    public ExecutorService taskScheduler() {
        return Executors.newScheduledThreadPool(
                4, // pool size
                new ThreadFactoryBuilder()
                  .setNameFormat("scheduler-thread-%d").build());
    }
}
```

What are the benefits of programmatic approach?

- Tasks are scheduled in one place
- Custom executor service
- Convenient testing

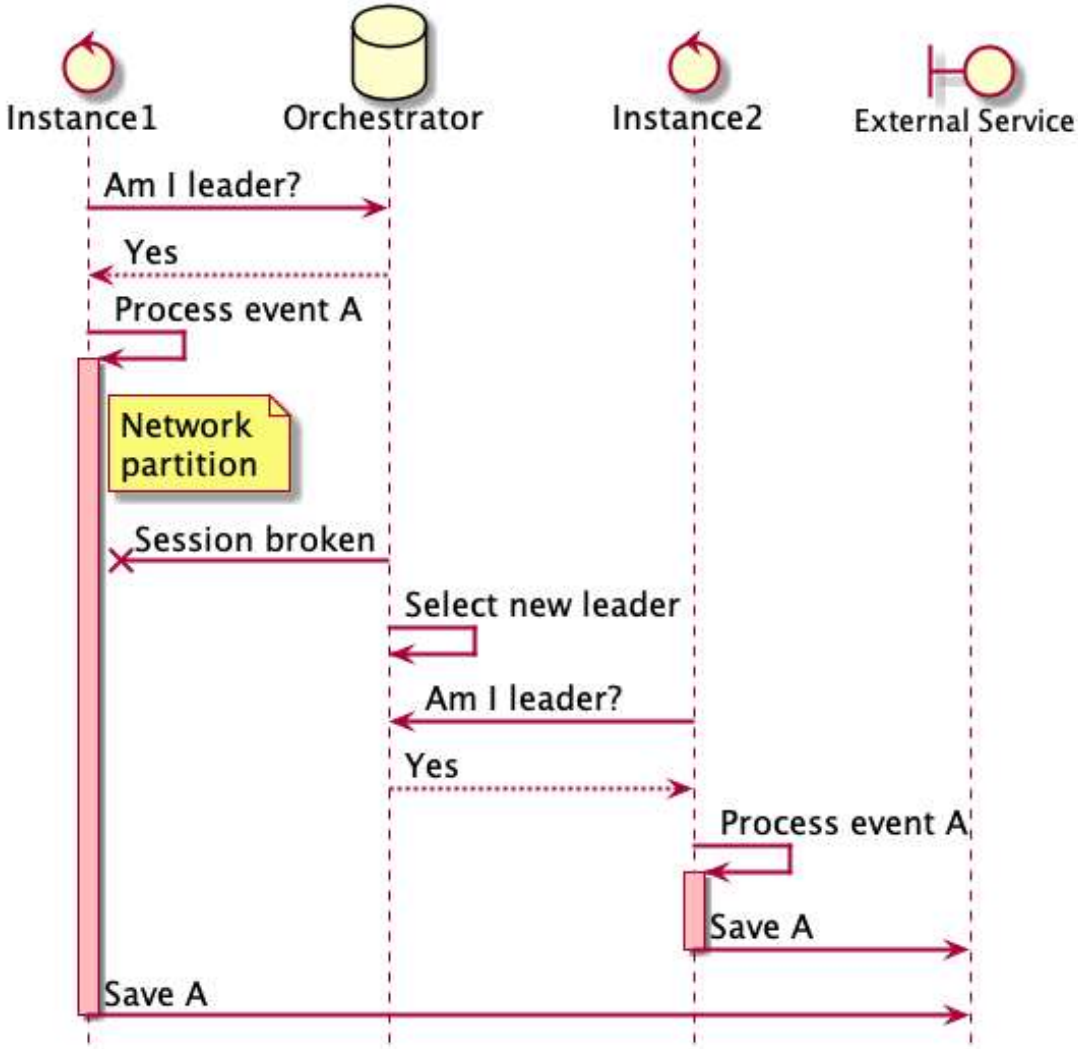
```
@RunWith(MockitoJUnitRunner.class)
public class SchedulingConfigTest {

    @InjectMocks
    private SchedulingConfig underTest;

    @Mock
    private ScheduledTaskRegistrar taskRegistrarMock;

    @Mock
    private ProcessEventsTask processEventsTaskMock;

    @Test
    public void schedulesCronTask() {
        underTest.configureTasks(taskRegistrarMock);

        verify(taskRegistrarMock)
                .addCronTask(processEventsTaskMock, "0 * * * * *");
    }
```

```java
@RunWith(MockitoJUnitRunner.class)
public class SchedulingConfigTest {

    @InjectMocks
    private SchedulingConfig underTest;

    @Mock
    private ScheduledTaskRegistrar taskRegistrarMock;

    @Test
    public void usesScheduledThreadPoolExecutor() {
        ArgumentCaptor<ScheduledThreadPoolExecutor> captor =
            forClass(ScheduledThreadPoolExecutor.class);

        underTest.configureTasks(taskRegistrarMock);

        verify(taskRegistrarMock).setScheduler(captor.capture());
        assertThat(captor.getValue().getCorePoolSize()).isEqualTo(4);
    }
```

```java
@Configuration
@EnableScheduling
public class SchedulingConfig implements SchedulingConfigurer {

    @Override
    public void configureTasks(ScheduledTaskRegistrar taskRegistrar) {

        taskRegistrar.addCronTask(
            new CronTask(() -> {
                if (isLeader()) {
                    // process events
                }
            }, "0 * * * * *"));

    }
```

```java
@Configuration
@EnableScheduling
public class SchedulingConfig implements SchedulingConfigurer {

    @Autowired
    private Runnable processEventsTask;

    @Override
    public void configureTasks(ScheduledTaskRegistrar taskRegistrar) {

        Runnable leaderAwareTask =
                new LeaderAwareTaskDecorator(processEventsTask);

        taskRegistrar.addCronTask(
                new CronTask(leaderAwareTask, "0 * * * * *"));
    }
```

```
public final class LeaderAwareTaskDecorator implements Runnable {

    private Runnable delegate;

    public LeaderAwareTaskDecorator(Runnable delegate) {
        this.delegate = delegate;
    }

    @Override
    public void run() {
        if (isLeader()) {
            delegate.run();
        }
    }
```


Instance1
Orchestrator
Instance2
External Service
Am I leader?
Yes
Process event A
Network partition
Session broken
Select new leader
Am I leader?
Yes
Process event A
Save A
Save A

Resiliency

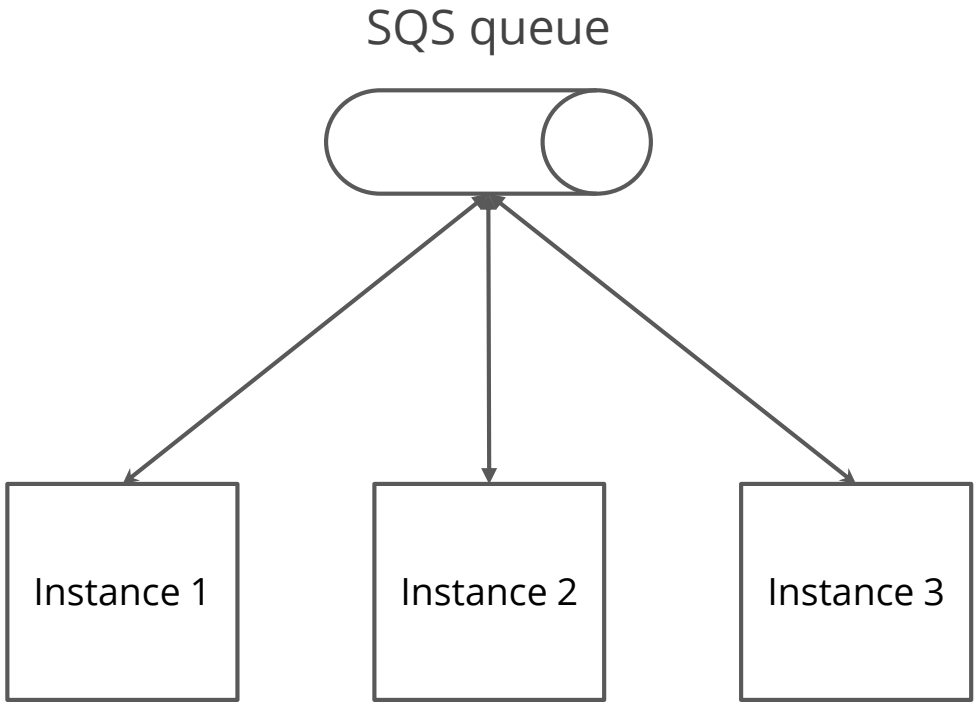
- What if the response didn’t come?
- Can we safely repeat?
  - Duplicate entries created
- Is the action idempotent?
  - One or multiple identical requests give the same result

Improvements

— Distribute the jobs

```
SELECT * FROM events FOR UPDATE SKIP LOCKED;
```

Further improvements



What have we learned?

- Annotation-driven development is hard
- Keep (code) consistency
- Increase resilience & predictability
- Think about observability

Thank you

Questions?

References

- AOP: https://docs.spring.io/spring/docs/2.5.x/reference/aop.html
- SchedulingConfigurer: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/scheduling/annotation/SchedulingConfigurer.html
- Postgresql select: https://www.postgresql.org/docs/9.5/sql-select.html