MICROSERVICES ARCHITECTURE

Event-Driven Microservice Architecture

→ In an event-driven microservice architecture, services communicate each-other via message.

→ pub-sub model :-

↳ When business event occur, producers publish them with message.

↳ At the same time, other services consume them through event listeners.

→ The main benefit of event-driven systems are asynchronous behaviour and loosely coupled structure.

* Event-Driven Architecture as a Solution

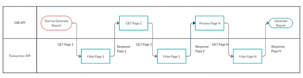
Often than building your system with event-driven structures, you can also use it as a solution to an already built tightly coupled environment.

↳ A Basic REST Driven Approach

Real-World Scenario: Trendyol (GEB API + Transaction API)

Let's consider a reporting application. It transmits all sale reports to the govt. So, this app has to fetch all the sale data from another API.

In the beginning, the transaction volume was very low. Thus, the API was quickly built with REST approach.



↳ Event-Driven Approach

Let's consider the volume of data increased suddenly. So, the team faced the problem of performance issues. The solution is simple: converting to event messaging.

1) Solution 1: Converting to Event Messaging

As soon as the team realized that the reports are not being generated efficiently, they applied the event-driven solution.

The plan was as follows :-

i) Publish an event when a transaction item is created.

ii) Fetch the related data when the event is received.

iii) Convert to a piece of report string

iv) Persist in the RDBMS

v) Query the data when generating the report.

vi) Concat the string data and persist as a file to the disk

As a result of this, the needed transaction item are persisted in the Reporting API. As soon as report creation starts, it gathers and concatenates the report data from the RDBMS.

Another problem

While the team converted the sync process into an async architecture, the transaction API faced another performance issue. Because the reporting (GEB) API requested the detail every time a transaction item created, the transaction API went under a heavy load.

The reason is, the transaction records are created for every item sold in the marketplace (Trendyol). So, the huge number of transaction item detail requests choked the API.

2) Solution 2: Fat Event

A fat event means that the message contains the detail with the entity identifier.

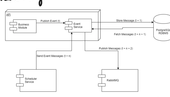
After converting the message into a fat event, the team didn't need any additional REST calls. As a result of this, the architecture became a complete async event-driven system.

3) Solution 3: Outbox Pattern

Because it is a financial business and we are talking about the sale transactions, the data is quite important. Thus, the calculation must be correct 100%.

To be able to access this accuracy, we must be sure that our system is not losing any event messages. Thus, we can apply the outbox pattern.

When our API publishes event messages, it doesn't directly send them in the outbox pattern. Instead, the messages are persisted in a DB table. A job sends cumulative messages in predefined time intervals.



i) Business API publishes an event.

ii) Event service persists the message in RDBMS

iii) Scheduler service triggers the job "Send Event Messages"

iv) Event service queries the cumulative event messages.

v) Event service publishes the messages via RabbitMQ.

Pros

a) Persistence of event messages

b) When an event is lost, the message can be checked from the DB.

c) A lost event can be recovered from the RDBMS.

Cons

a) Delay in publishing events

b) Increase in complexity

c) To publish a basic event, at least two technologies are needed: Storage system and Message Queuing Protocol.

↳ Benefits of the Event-Driven Microservice Architecture

1) Loosely coupled structure

2) Completely isolation of the microservices

3) No synchronous REST calls

4) Asynchronous event-driven functionality

5) Performance gain

Among all of the above, the most important benefit is the first one. Because we want to separate the components by microservice architecture, all of the units must be separated enough (loosely-coupled). Otherwise, microservice architecture won't work and your system will turn into a distributed-monolith.

↳ Downsides

1) Single point of failure

If your RabbitMQ/Kafka fails any issues during the production process, your whole system will also fail.

To overcome failures:

i) Construct your RabbitMQ/Kafka as a cluster

ii) Create your queues durable.

iii) Publish your messages persisted

As a result of the above measures, you can quickly recover any failures. If any error happens, your other instances in the cluster will take over the work and recreate the durable queues. Also, your persisted messages will be recovered from the disk.

2) Duplicated event messages

An event publisher API can face trouble and resend the same message. To resolve any duplication in the system, any consumer endpoint has to be idempotent: always consider to check first if our API acquired the event before.