SOA4

Microservice Inter-process communication

Reference to book

- **The following notes closely follow and use examples from Chapter 4 of “Building Microservices” by Sam Newman.**

- **https://www.oreilly.com/library/view/building-microservices-2nd/9781492034018/ch04.html**

Microservices inter-process vs in-process

- Moving from a **monolith** to a **microservice** results in **inter-process** calls as opposed to **in-process** calls
- There are some **performance** issues here as the calls are over a network
  - Round-trip time (negligible for in-process).
    - Need to consider number of calls between processes and try to keep them to a minimum.
  - Data transfer (negligible for in-process; object reference of an object passed into method), but this can be a problem when transferring large amounts of data across a network.
    - Need to think about the payload being transferred and try to reduce the size of the data being transferred.

Error Handling

- **Errors Handling** is another issue when dealing with inter-process communication

- Network errors are outside your control e.g. timeouts, other microservices going offline, network disconnection etc. Some common types of failure with distributed systems are:
  - Crash failure e.g. server crash
  - Omission failure: e.g. no response to request
  - Timing failure: e.g. response didn’t arrive on time

Error Handling

- To deal with distributed errors, HTTP provides error codes – REST takes advantage of these when used properly:
  - 404 Not Found (maybe a problem with request or maybe service not running)
  - 501: Not Implemented (so no point trying again)
  - 503: Service Unavailable (this could be temporary; try again later)

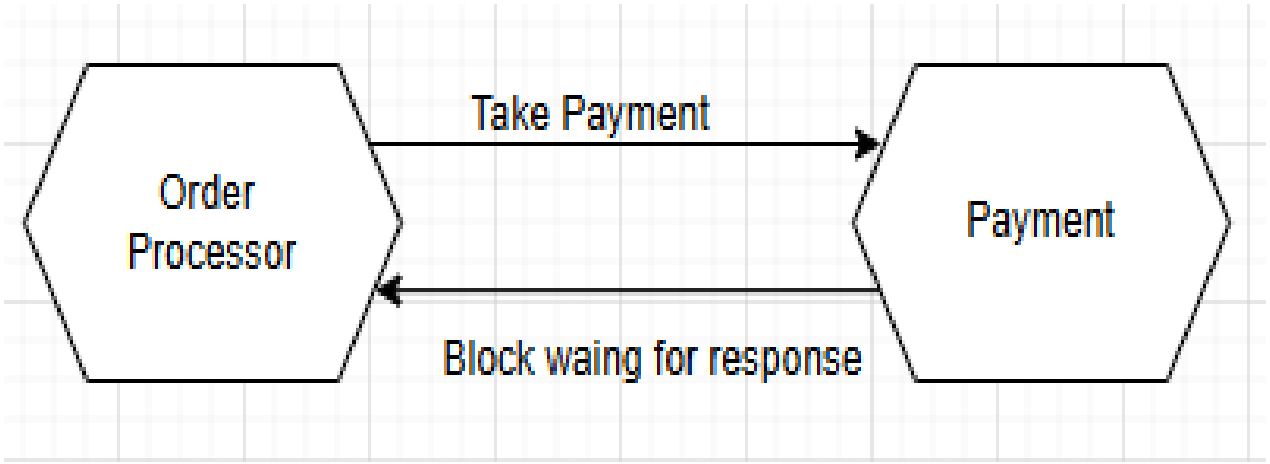
Styles of Microservice Communication

- There are three main forms of microservice communication:
  - Request-Response (Synchronous Blocking)
  - Request-Response (Asynchronous Nonblocking)
  - Event-Driven Communication

The following slides will outline these styles.

Request-response (Synchronous Blocking)

- In this case, a microservice sends a request to another microservice and waits (blocks) until the response is received. E.g. an **Order** microservice sends a request to a **Payment** microservice and waits until the payment goes through.



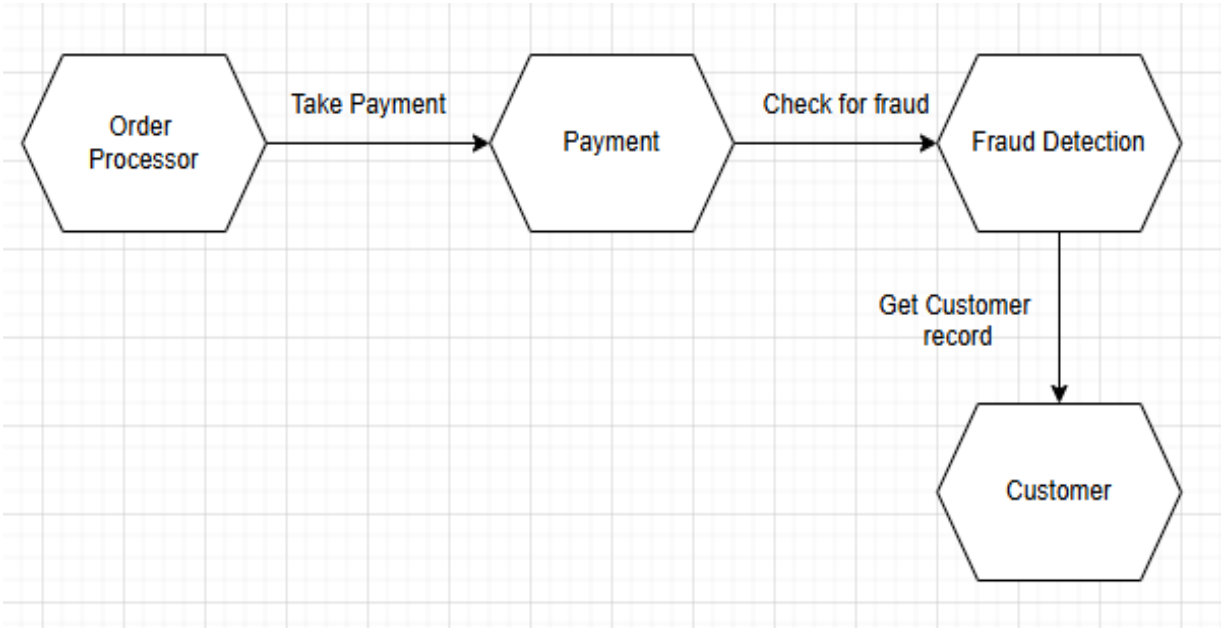
Request-response (Synchronous Blocking)

- Advantages:
  - Simple and familiar for programmers – sequential style of programming

- Disadvantages:
  - If the payment microservice is unavailable the call will fail, so the Order service needs to be more complex to figure out what to do – retry, wait, give up?
  - If the Payment service is slow or the network latency is an issue, then the Order service will be blocked for a long time.

Request-response (Synchronous Blocking)

- Disadvantages contd..
  - If there are chains of calls, this leads to any of the services causing the whole operation to fail. There's also an issue with resource usage here as all the services have network connections open waiting for the responses.



This type of communication is necessary if the origin microservice needs an answer to request before it can proceed e.g. checking the Stock for an item.

Request-response (Asynchronous Nonblocking)

- In this case, a microservice sends a request to another microservice and does not block. When the request completes, successfully or not, the origin microservice receives the response.

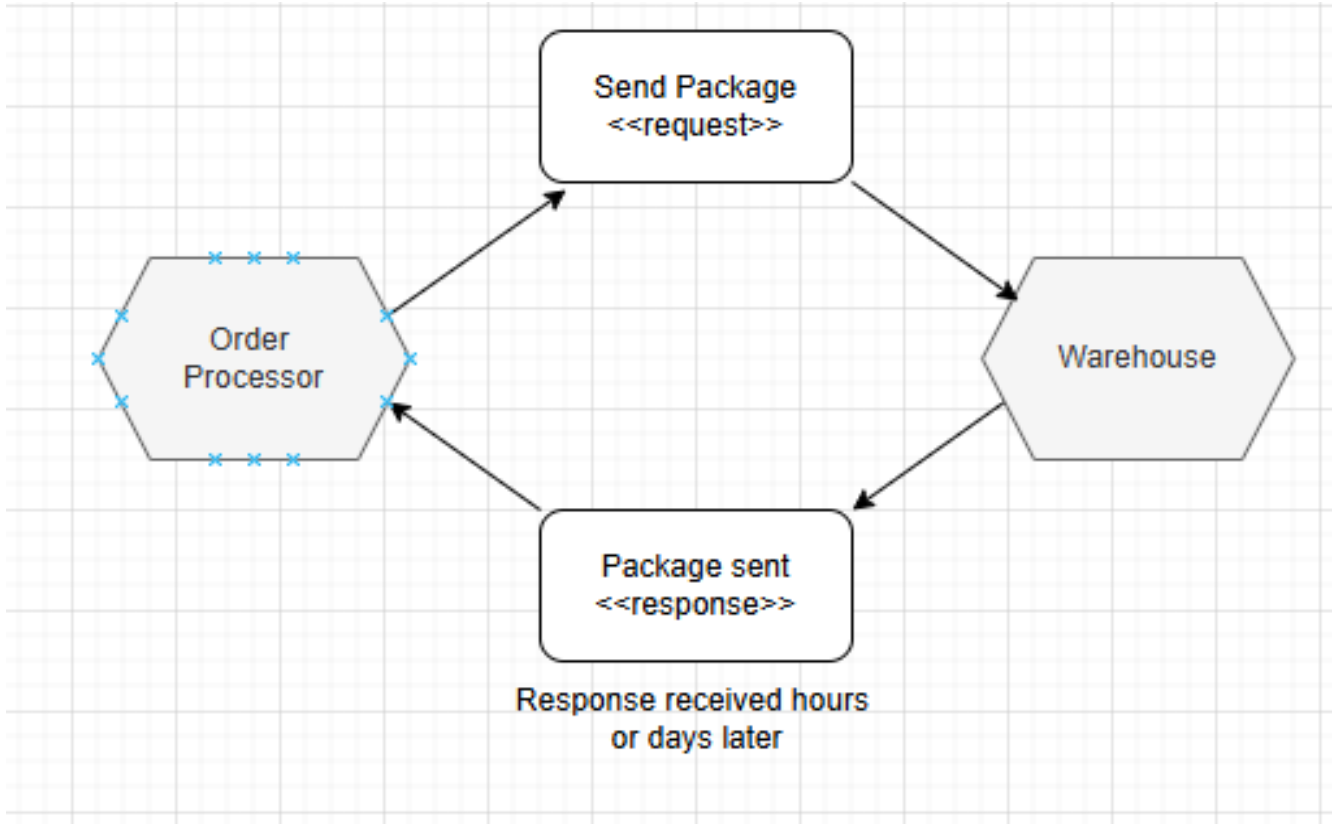
- E.g. we've seen these types of calls when using AJAX to fetch data from APIs – need to define call-back functions.

Request-response (Asynchronous Nonblocking)

- Advantages:
  - Microservices don’t need to be reachable at the time of the call (need to use queues for this to work)
  - Microservices don’t block
  - Useful in situations where response could take a long time to complete e.g. the Order Service sends a request to the Warehouse to dispatch a parcel; the dispatch process could take hours or days.

Request-response (Asynchronous Nonblocking)

- e.g. the **Order** Service sends a request to the **Warehouse** to dispatch a parcel; the dispatch process could take hours or days.



Request-response (Asynchronous Nonblocking)

- Disadvantages:
  - Increases complexity of programming (call-back functions)
  - Microservice instance that made the request may not be alive anymore, so ***any*** instance must be able to deal with the response (need to store state of original Order in a database for example and let the new instance update)

Event-Driven Communication

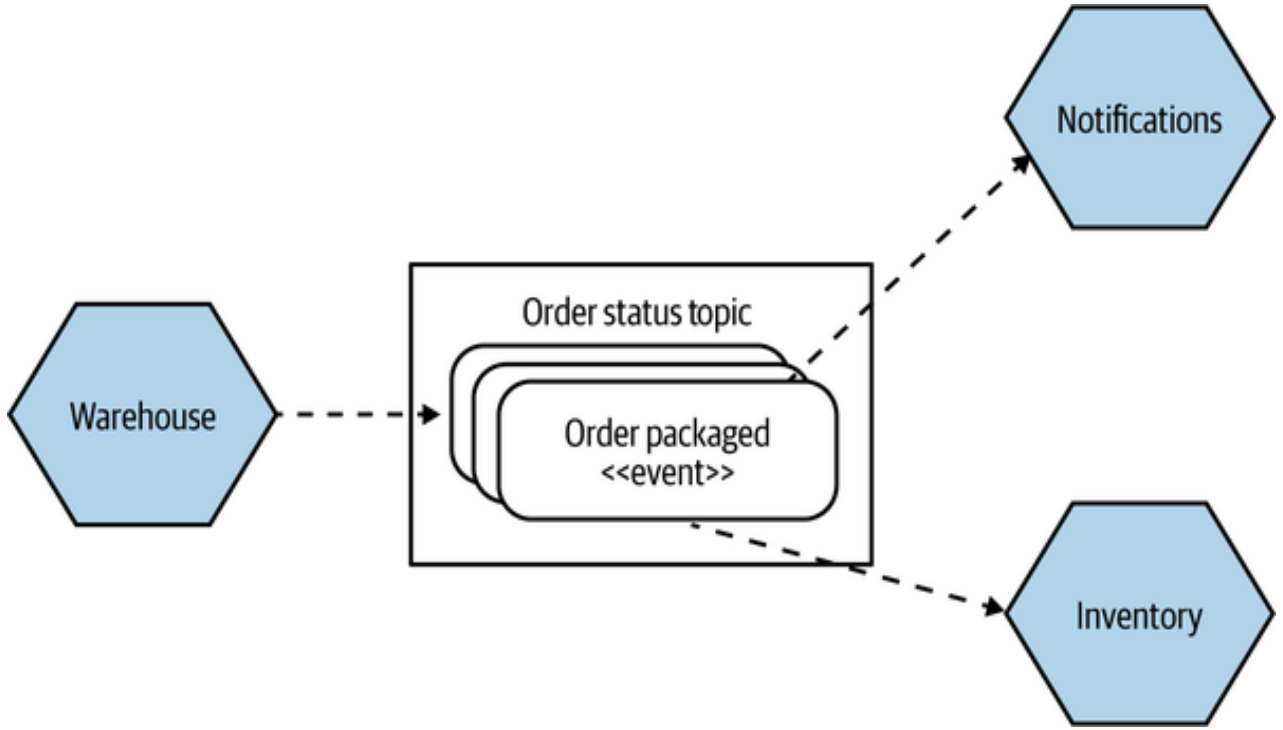
- With event-driven communication, a microservice transmits events that are received by other microservices that have subscribed to those events.

- The event contains information about something that has occurred inside the microservice. The event emitter is not necessarily aware which other microservices are receivers of the event; it just sends the event and that’s the end of its responsibilities.

- Its up to the receiver of the event to act accordingly i.e. the responsibility is moved to the receiver of the event.

Event-Driven Communication

- E.g. a ***Warehouse*** microservice may emit an event to represent an order having been packaged. A ***Notifications*** service may receive the event and send an email to the order recipient (customer). In addition, the ***Inventory*** service may also receive the event and update the stock levels accordingly.



Event-Driven Communication

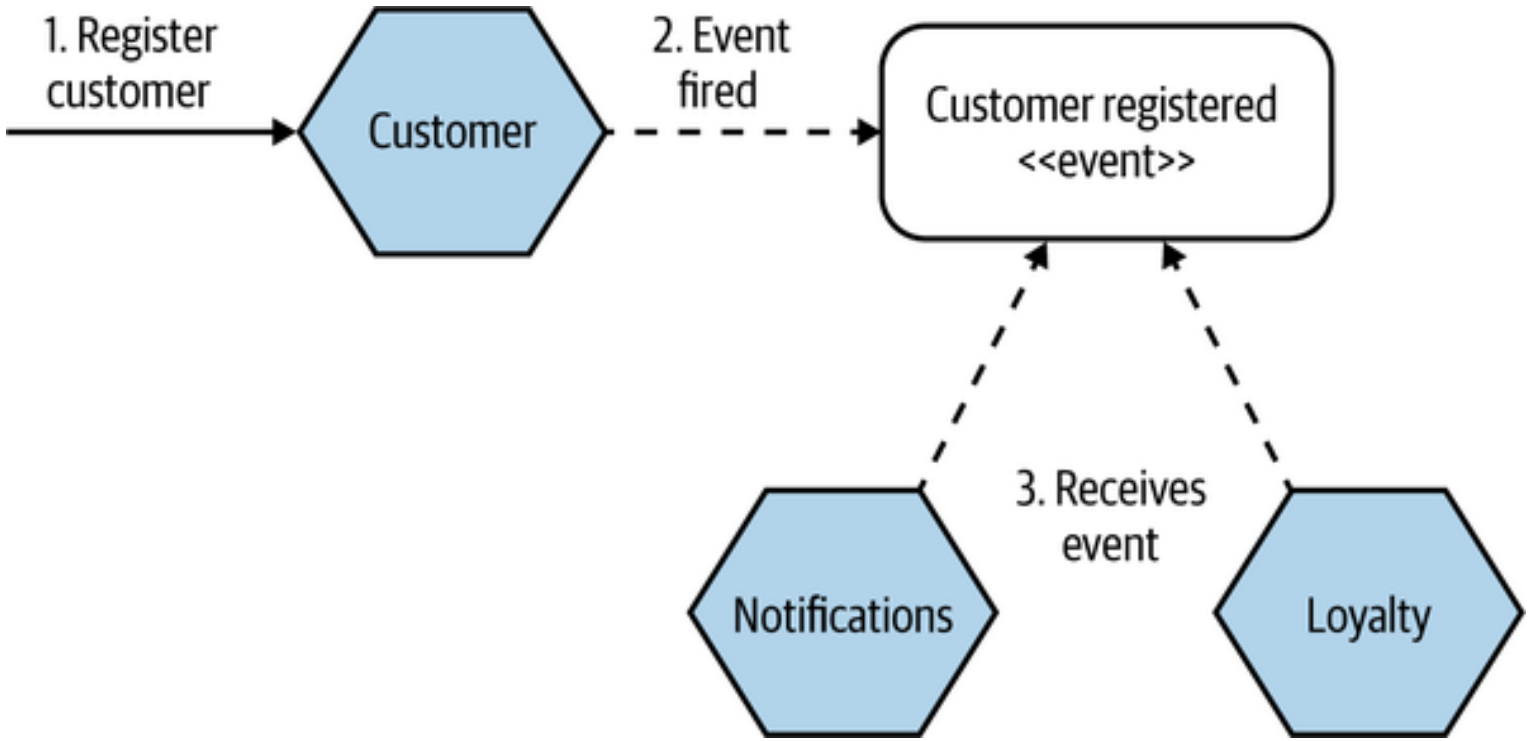
- **Advantages:**
  - This type of communication greatly reduces coupling i.e. services are loosely coupled.
  - It helps to reduce complexity in some microservices (e.g. ***Warehouse***) by pushing the responsibility onto others (e.g. ***Notifications*** and ***Inventory***). This has the effect of balancing the complexity among the services (and the teams responsible for the services).

Implementation of Event-Driven Communication:

- Producers publish events to a broker
- Broker handles subscriptions from consumers
- When the message is received by the broker it attempts to send the message to all subscribing consumers. There is a queue available to avoid overwhelming the consumer.
- RabbitMQ is one implementation.

What information is in an Event?

- Consider the following scenario where the Customer service transmits an event when a new customer registers on the site:



What information is in an Event?

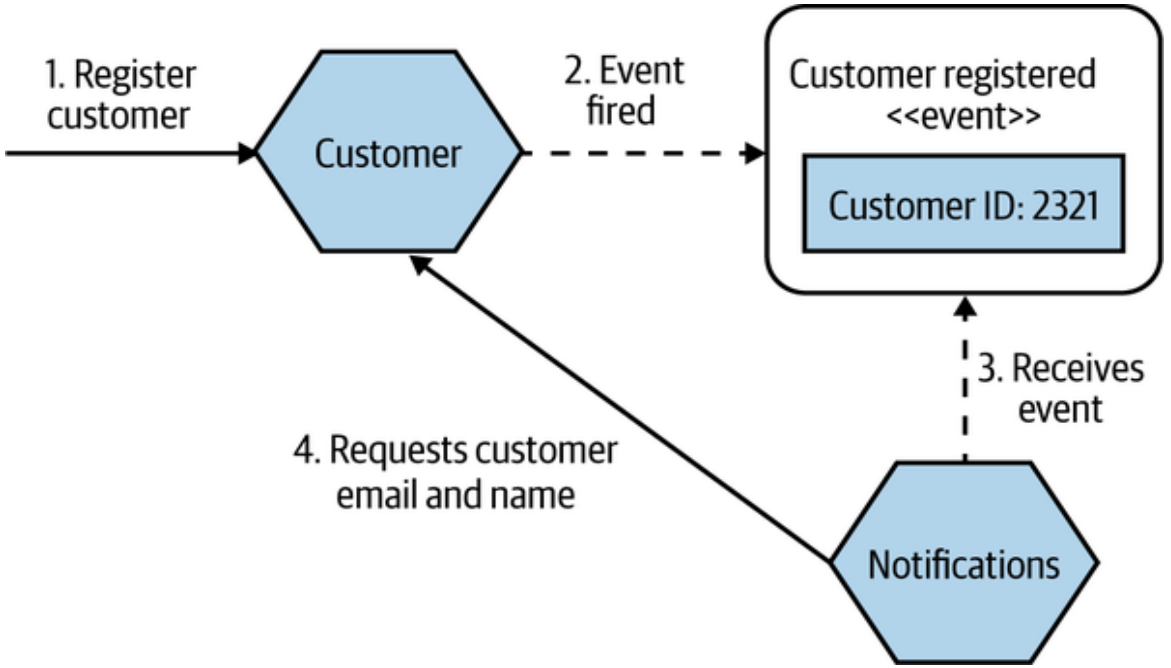
- The ***Notifications*** service sends an email to the customer to welcome them to the site, while the ***Loyalty*** service creates a loyalty account for the new customer.

- What information do these services need to carry out their tasks?
  - ***Notifications*** service: needs **email address** and **name**
  - ***Loyalty*** service: just needs the **cus_id** (add a line to a database table with **cus_id** and **numPoints**)

So what information should we send in these events?

Sending just the ID in the event:

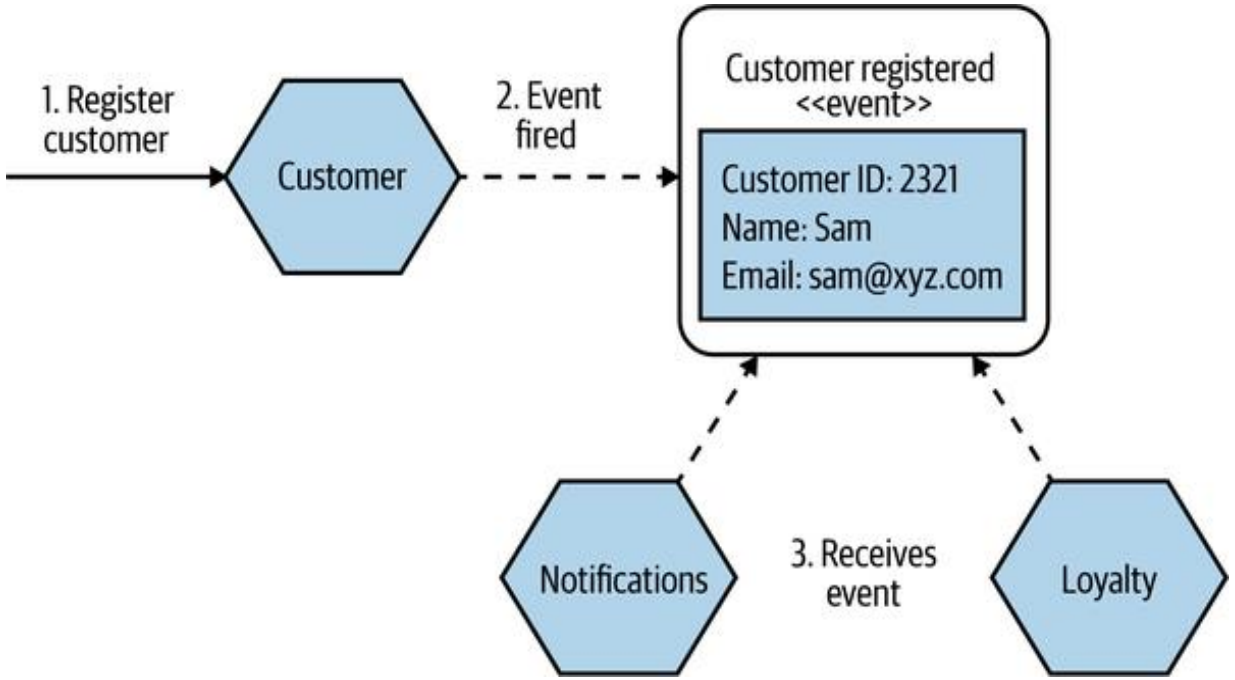
- In the interest of information hiding, you could just emit the id and let the Notifications service query the Customer service for additional information:



- This would ensure that only the consumers that need the information get it. But it could cause a lot of connections to the ***Customer*** service if many recipient services needed extra information after the event is sent.
- This also would result in more coupling between the ***Notifications*** and ***Customer*** services.

Sending fully detailed events:

- The alternative is to send more detailed information in the event:



- **Advantages:**
  - It results in looser coupling.
  - These events can be used as an historical record or an auditing system

Sending fully detailed events:

- **Disadvantages:**
  - Very large messages could be problematic for communications
  - Some consumers get information they don’t need. You could send two different types of messages e.g. one with only the id and the other with id, name, and email.