


By Gero Vermaas (<http://blog.xebia.com/author/gero>)

sonatype-

Microservices are the latest architectural style promising to resolve all the issues we had with previous architectural styles. Like other styles, it has its own challenges. The issue discussed in this post is how to realise coupling between microservices while keeping the services as autonomous as possible. Here, four options will be described and a clear winner will be selected in the conclusion.

To me, microservices are autonomous services that take full responsibility for one business capability. Full responsibility includes presentation, API, data storage and business logic. Autonomous is the keyword for me, by making the services autonomous the services can be changed with no or minimal impact on others. If services are autonomous, then operational issues in one service should have no impact on the functionality of other services.

This all sounds like a good idea, but services will never be fully isolated islands. A service is virtually always dependent on data provided by another service. For example, imagine a shopping cart microservice as part of a web shop, some other service must put items in the shopping cart and the shopping cart contents must be provided to yet other services to complete the order and get it shipped. The question now is how to realise these couplings while keeping maximum autonomy. The goal of this blog post is to explain which pattern should be followed to couple microservices while retaining maximum autonomy.

I'm going to structure the patterns in two dimensions; the  rr-ps interaction pattern, and the information exchanged using this pattern.

Interaction pattern: Request-Reply vs. Publish-Subscribe.

- Request-Reply means that one service does a *specific* request for information (or to take some action). It then expects a response. The requesting service therefore needs to know what to ask and where to ask it. This could still be implemented asynchronously and of course you could put some abstraction in place such that the request service does not have to know the physical address of the other service, the point still remains that one service is explicitly asking for *specific* information (or action to be taken) and functionally waiting for a response.
- Publish-Subscribe: with this pattern a service registers itself as being interested in certain information, or being able to handle certain requests. The relevant information or requests will then be delivered to it and it can decide what to do with it. In this post we'll assume that there is some kind of middleware in place to take care of delivery of the published messages to the subscribed services.

Information exchanged: Events vs. Queries/Commands

- Events are facts that cannot be argued about. For example, an order with number 123 is created. Events only state what has happened. They do not describe what should happen as a consequence of such an event.
- Queries/Commands: Both convey what should happen. Queries are a specific request for information, commands are a specific request to the receiving service to take some action.

Putting these two dimensions in a matrix results into 4 options to realise couplings between microservices. So what are the advantages and disadvantages for each option? And which one is the best for reaching maximum autonomy? ^

In the description below we'll use 2 services to illustrate each pattern. The Order service which is responsible for managing orders, and the Shipping service, which is responsible for shipping stuff, for example the items included in an order. Services like these could be part of a webshop, which could then also contain services like a shopping cart, a product (search) service, etc.

## 1. REQUEST-REPLY WITH EVENTS: (http://blog.xebia.com/wp-content/uploads/2015/03/rre1.png)

In this pattern, one service asks a *specific* other service for events that took place (since the last time it asked). This implies strong dependency between these two services. The Shipping service must know which service to connect to for events related to orders. There is also a runtime dependency since the shipping service will only be able to ship new orders if the Order service is available.

Since the Shipping service only receives events, it has to decide by itself when an order may be shipped based on information in these events. The Order service does not have to know anything about shipping, it simply provides events stating what happened to orders and leaves the responsibility to act on these events fully to the services requesting the events.

## 2. REQUEST-REPLY WITH COMMANDS/QUERIES:

(http://blog.xebia.com/wp-content/uploads/2015/03/rrc.png) In this pattern the shipping Order service is going to request the Shipping service to ship an order. This implies strong coupling since the Order service is explicitly requesting a specific service to take care of the shipping. Now the Order service must determine when an order is ready to be shipped. It is aware of the existence of a Shipping service and it even knows how to interact with it. If other factors not related to the order itself should be taken into account before shipping the order (e.g. credit status of the customer), then the order services should take this into account before requesting the shipping service to ship the order. Now the business process is baked into the architecture and therefore the architecture cannot be changed easily.

Again there is a runtime dependency since the Order service must ensure that the shipping request is successfully delivered to the Shipping service.

## 3. PUBLISH-SUBSCRIBE WITH EVENTS

(http://blog.xebia.com/wp-content/uploads/2015/03/pse1.png) In Publish-Subscribe with Events, the Shipping service registers itself as being interested in events related to Orders. After registering itself, it will receive all events related to Orders without being aware what the source of the order events is. It is loosely coupled to the source of the Order events. The shipping service will need to retain a copy of the data received in the events such that it can conclude when an order is ready to be shipped.

The Order service needs to have no knowledge about shipping. If multiple services provide order related events containing relevant data for the Shipping service then this is not recognisable by the Shipping service. If (one of) the service(s) providing order events is down, the Shipping service will not be aware, it just receives less events. The Shipping service will not be blocked by this.

## 4. PUBLISH-SUBSCRIBE WITH COMMANDS/QUERIES

^

([http://blog.xebia.com/wp-](http://blog.xebia.com/wp-content/uploads/2015/03/psc.png)



content/uploads/2015/03/psc.png) In Publish-Subscribe with Command/Queries, the Shipping service registers itself as a service being able to ship stuff. It then receives all commands that want to get something shipped. The Shipping service does not have to be aware of the source of the Shipping commands and on the flip side the Order service is not aware of which service will take care of shipping. In that sense, they are loosely coupled. However, the Order service is aware of the fact that orders must get shipped since it is sending out a ship command, this does make the coupling stronger.

## CONCLUSION

Now that we have described the four options we go back to the original question, which pattern of the above 4 provides maximum autonomy?

Both Request-Reply patterns imply a runtime coupling between two services and that implies strong coupling. Both Command/Queries patterns imply that one service is aware of what another service should do (in the examples above the order service is aware that another service takes care of shipping) and that also implies strong coupling, but this time on functional level. That leaves one option: 3. Publish-Subscribe with Events. In this case, both services are not aware of each others existence from both runtime and functional perspective. To me this is the clear winner for achieving maximum autonomy between services.

The next question pops up immediately – should you always couple services using Publish-Subscribe with events? If your only concern is maximum autonomy of services, the answer would be yes, but, there are more factors that should be taken into the account. Always coupling using this pattern comes at a price. For example, data is replicated, measures must be taken to deal with lost events, events driven architectures do add extra requirements on infrastructure, and there might be extra latency.

In a future post, I'll dive into these trade-offs and put things into perspective. For now, remember that Publish-Subscribe with Events is a good basis for achieving autonomy of services.

amazon (<https://www.voxxed.com/tag/amazon/>), api exchange (<https://www.voxxed.com/tag/api-exchange/>), api management (<https://www.voxxed.com/tag/api-management/>), big data (<https://www.voxxed.com/tag/big-data/>), businessevents (<https://www.voxxed.com/tag/businessevents/>), businessworks (<https://www.voxxed.com/tag/businessworks/>), CEP (<https://www.voxxed.com/tag/cep/>), Chef (<https://www.voxxed.com/tag/chef/>), cloud (<https://www.voxxed.com/tag/cloud-2/>), complex event processing (<https://www.voxxed.com/tag/complex-event-processing/>), Continuous Delivery (<https://www.voxxed.com/tag/continuous-delivery/>), Continuous Integration (<https://www.voxxed.com/tag/continuous-integration/>), devops (<https://www.voxxed.com/tag/devops/>), Docker (<https://www.voxxed.com/tag/docker/>), EAI (<https://www.voxxed.com/tag/eai/>), Enterprise Application Integration (<https://www.voxxed.com/tag/enterprise-application-integration/>), Enterprise Service Bus (<https://www.voxxed.com/tag/enterprise-service-bus/>), ESB (<https://www.voxxed.com/tag/esb/>), IBM (<https://www.voxxed.com/tag/ibm/>), in-memory (<https://www.voxxed.com/tag/in-memory/>), integration (<https://www.voxxed.com/tag/integration/>), internet of things (<https://www.voxxed.com/tag/internet-of-things/>), JBoss (<https://www.voxxed.com/tag/jboss/>), micro service (<https://www.voxxed.com/tag/micro-service/>), microservice (<https://www.voxxed.com/tag/microservice/>), microservices (<https://www.voxxed.com/tag/microservices/>), microsoft (<https://www.voxxed.com/tag/microsoft/>), mulesoft (<https://www.voxxed.com/tag/mulesoft/>), open api (<https://www.voxxed.com/tag/open-api/>), openstack (<https://www.voxxed.com/tag/openstack/>), oracle (<https://www.voxxed.com/tag/oracle-2/>), puppet (<https://www.voxxed.com/tag/puppet/>), Red Hat (<https://www.voxxed.com/tag/red-hat/>), REST (<https://www.voxxed.com/tag/rest/>), sap (<https://www.voxxed.com/tag/sap/>) ^