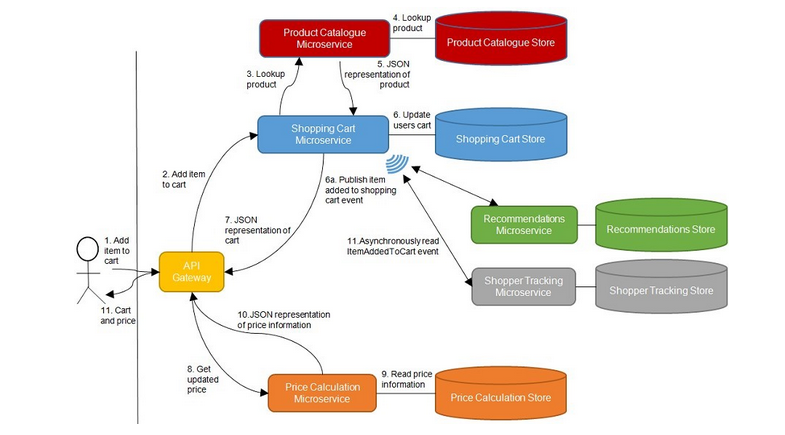
**MICROSERVICES FAILING CONDITION:**

When working with any non-trivial software systems, we must expect failures to occur. Hardware can fail. The software itself might fail due, for instance, to unforeseen usage or corrupt data. A distinguishing factor of a microservice system is that there is a lot of communication between the microservices.

Figure 1 shows the communication resulting from a user adding an item to his/her shopping cart. From figure 1 we see that just one user action results in a good deal of communication. Considering that a system will likely have concurrent users all performing many actions, we can see that there really is a lot of communication going on inside a microservice system.

We must expect that communication to fail from time to time. The communication between only two microservices may not fail very often, but in regard to a microservice system as a whole, communication failures are likely to occur often simply because of the amount of communication going on.



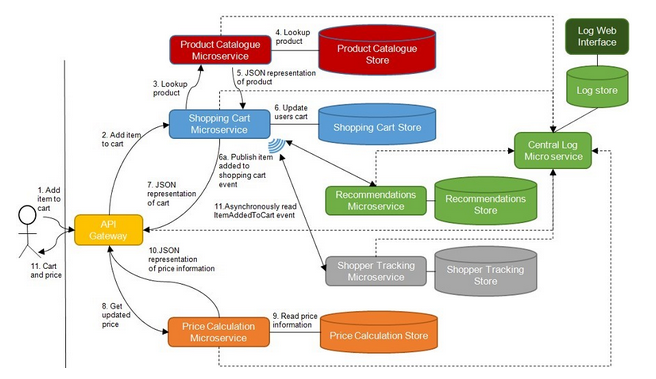
Since we have to expect that some of the communication in our microservice system will fail, we should design our microservices to be able to cope with those failures.

We can divide the collaborations between microservices into three categories: Query, command and event based collaborations. When a communication fails, the impact depends on the type of collaboration and way the microservices cope with it:

* *Query*  based collaboration: When a query fails, the caller does not get the information it needs. If the caller copes well with that, the impact is that the system keeps on working, but with some degraded functionality. If the caller does not cope well, the result could be an error.
* *Command*  based collaboration: When sending a command fails, the sender won’t know if the receiver got the command or not. Again, depending on how the sender copes, this could result in an error, or it could result in degraded functionality.
* *Event*  based collaboration: When a subscriber polls an event feed, but the call fails, the impact is limited. The subscriber will poll the event feed later and, assuming the event feed is up again, receive the events at that time. In other words, the subscriber will still get all events, only some of them will be delayed. This should not be a problem for an event-based collaboration, since it is asynchronous anyway.

**Have Good Logs**

Once we accept that failures are bound to happen and that some of them may result, not just in a degraded end-user experience, but in errors, we must make sure that we are able to understand what went wrong when an error occurs. That means that we need good logs that allow us to trace what happened in the system leading up to an error situation. "What happened" will often span several microservices, which is why you should consider introducing a central *Log Microservice*, as shown in figure 2, that all the other microservices send log messages to, and which allows you to inspect and search the logs when you need to.



The *Log Microservice* is a central component that all other microservices use. We need to make certain that a failure in the *Log Microservice* does not bring down the whole system when all other microservices fail because they are not able to log messages. Therefore, sending log messages to the *Log Microservice* must be *fire and forget* - that is, the messages are sent and then forgotten about. The microservice sending the message should not wait for a response.

|  |  |
| --- | --- |
|  |  |

**Correlation Tokens**

In order to be able to find all log messages related to a particular action in the system, we can use *correlation tokens*. A correlation token is an identifier attached to a request from an end user when it comes into the system. The correlation token is passed along from microservice to microservice in any communication that stems from that end-user request. Any time one of the microservices sends a log message to the *Log Microservice,* the message should include the correlation token. The *Log Microservice* should allow searching for log messages by correlation token. Referring to figure 2, the *API Gateway* would create and assign a correlation token to each incoming request. The correlation is then passed along with every microservice-to-microservice communication.

**Roll Forward vs Roll back**

When errors happen in production, we are faced with the question of how to fix them. In many traditional systems, if errors start occurring shortly after a deployment, the default would be to roll back to the previous version of the system. In a microservice system, the default can be different. Microservices lend themselves to continuous delivery. With continuous delivery, microservices will be deployed very often and each deployment should be both fast and easy to perform. Furthermore, microservices are sufficiently small and simple so many bug fixes are also simple. This opens the possibility of *rolling forward* rather than rolling backward.

Why would we want to default to rolling forward instead of rolling backward? In some situations, rolling backward is complicated, particularly when database changes are involved. When a new version that changes the database is deployed, the microservice will start producing data that fits in the updated database. Once that data is in the database, it has to stay there, which may not be compatible with rolling back to an earlier version. In such a case, rolling forward might be easier.

**Do Not Propagate Failures**

Sometimes things happen around a microservice that may disturb the normal operation of the microservice. We say that the microservice is under stress in such situations. There are many sources of stress. To name a few, a microservice may be under stress because:

* One of the machines in the cluster its data store runs on has crashed
* It has lots network connectivity to one of its collaborators
* It is receiving unusually high amounts of traffic
* One of its collaborators is down

In all of these situations, the microservice under stress cannot continue to operate the way it normally does. That doesn’t mean that it’s down, only that it must cope with the situation.

When one microservice fails, its collaborators are put under stress and are also at risk of failing. While the microservice is failing, its collaborators will not be able to query, send commands or poll events from the failing microservice. As illustrated in figure 3, if this makes the collaborators fail, even more, microservices are at risk of failing. At this point, the failure has started propagating through the system of microservices. Such a situation can quickly escalate from one microservice failing to lot of microservices failing.

**Figure 3 If the microservice marked FAILED is failing, so is the communication with it. That means that the microservices at the other end of those communications are under stress. If the stressed microservices fail due to the stress, the microservices communicating with them are put under stress. In that situation, the failure in the failed microservice has propagated to several other microservices.**

* When one microservice tries to send a command to another microservice, which happens to be failing at the time, that request will fail. If the sender simply fails as well, we get the situation illustrated in figure 3 where the failures propagate back through the system. To stop the propagation, the sender might act as if the command succeeded, but actually store the command into a list of failed commands. The sending microservice can periodically go through the list of failed commands and try to send them again. This is not possible in all situations because the command may need to be handled here and now, but when this approach is possible it stops the failure in one microservice from propagating.
* When one microservice queries another one that’s failing, the caller could use a cached response. In case the caller has a stale response in the cache but a query for a fresh response fails, it might decide to use the stale response anyway. Again, this is not something that will be possible in all situations, but when it is, the failure will not propagate.
* An *API Gateway* that is stressed because of high amounts of traffic from a certain client can throttle that client by not responding to more than a certain number of requests per second from that client. Notice that the client may be sending an unusually high amount of requests because it is somehow failing internally. When throttled, the client will get a degraded experience, but will still get some responses. Without the throttling, the *API Gateway* might become slow for all clients or it might fail completely. Moreover, since the *API Gateway*  collaborates with other microservices, handling all the incoming requests would push the stress of those requests onto other microservices too. Again, the throttling stops the failure in the client from propagating further into the system to other microservices.