Microservices: Brownfield: Migration: Database

acarlstein.com/

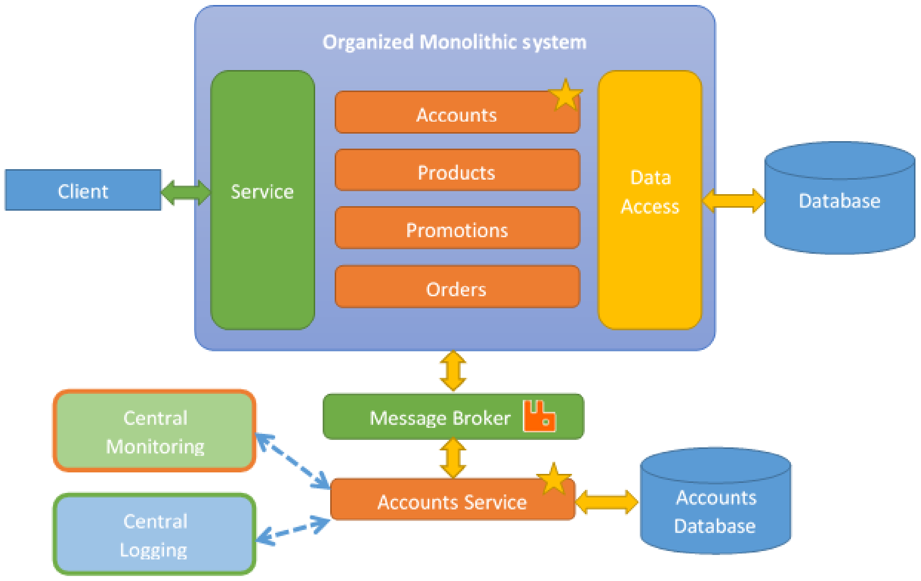
Posted by Alejandro G. Carlstein Ramos Mejia on January 12, 2017 January 17, 2017 About Programming / Architecture / Microservices

Microservices: Brownfield: Migration | Microservices: Brownfield: Transactions

In this section, we are going to go over splitting the monolithic database into databases that will be used on each micro-service. In this way, each micro-service will have its own database which makes it easier to maintain and part of the whole micro-services concept.

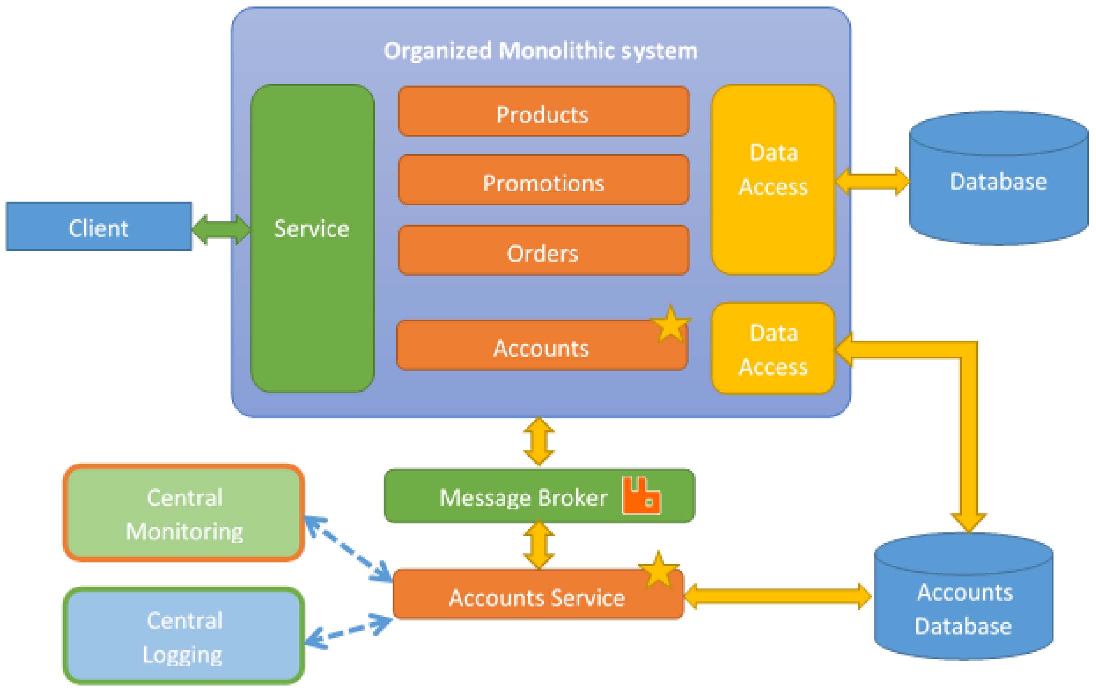
As establish on the previous articles related with micro-services, we want to avoid shared databases. We want our micro-services to be as independent as possible. In this way, they can be independently changeable and deployable. A shared database limits us and makes our micro-services dependent.

The approach to split our monolithic database into micro-services databases is similar to split the code into bounded contexts as explained on the previous article, Microservices: Brownfield: Migration.



We split seams in the database which are related to seams in the code. In other words, we can take the tables that are related with our code and move them (or recreate them) into the new database. In our case, all the tables needed for the account functionality, will be taken from the shared database into the single database exclusive used for the account micro-service.

Note that in the process of moving from monolithic to micro-services, we may have to modify our the data layer of our monolithic system to access multiple databases.



A question may cross your mind which is, what do we do when we have a table which is linked across seams? For example, you may have a promotion which is linked to an order. So you have two services, the promotion service and the order services working together. Then, we must provide API calls which allow us to fetch the data for that relationship. In our example case between the promotion and the order, we will have the Order service fetching specific data from the Promotion service.

Remember that we are refactoring our database into multiple database. We must worry about data referential integrity. This means that if we delete an account of a customer, for example, we might have to take care of orders related with that customer. Those orders exists in the Orders service. We would do this by calling the method in the Orders micro-service which would instruct in our example case to delete or disable specific orders related with the specific account ID that was deleted in the Account service. We must ensure that our micro-services talk to each other in order to keep the data referential integrity.

In the case where we have static tables that are required by all micro-services, The best action is

- Make that data into a configuration file available to all micro-services.
- Or, have a specific micro-service just for these static tables.

The same principles apply when you have valid shared data that is read and written by multiple services. You move the data to a configuration file or you create a micro-service that can be used by the other micro-services.