### Architecting Micro-Service

You don’t have to start with micro-services. It’s possible to evolve an existing application towards micro-services. Micro-services are autonomous, mean it owns it’s own data. Micro-Services should be independently deployable, which means it need to have a clearly defined and backwards-compatible public interface.

**What if I already have an existing (monolithic) application or May be a distributed Monolith?**

It is possible to perfectly evolve towards a micro-services architecture. You can Augment a monolith with micro-services, where every time you add a new capability, you create a new micro-service and you can decompose a monolith into micro-services where you identify existing capabilities that should be extracted out into their own independent micro-services.

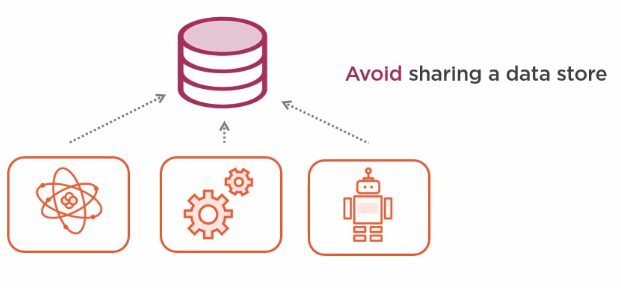
In fact some people argue that “*you should avoid starting a brand new project with a micro-services architecture. The benefits of micro-services are not typically seen in small, new projects. And it can be very hard to get your service boundaries right. So, it might be better to allow a system to grow a bit until it becomes apparent what an appropriate segregation of responsibilities into micro-services would be.”*

Now, of course, when you are using at some point you are going to need to identify the responsibilities of a micro-service and define what its public interface should be.

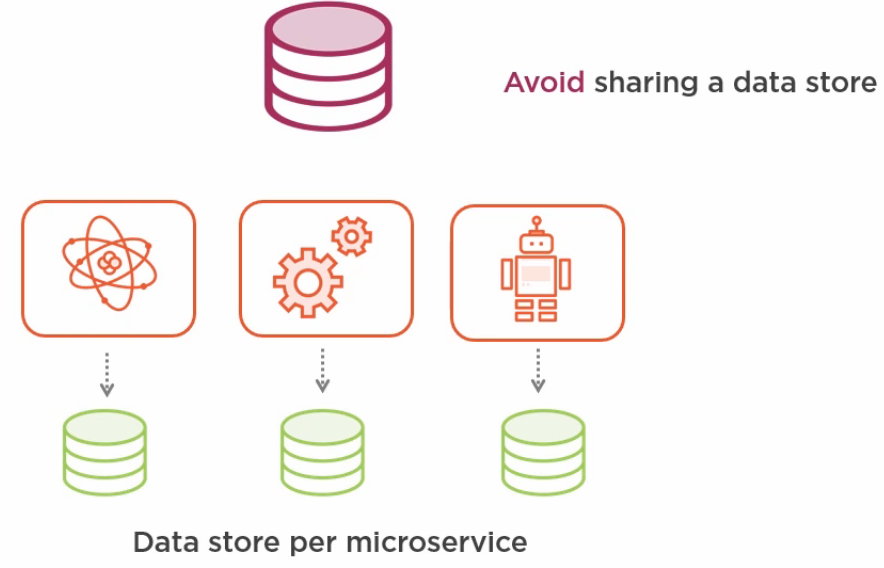


**Micro-Services owns its own Data**

We are not allowed to have the following situation:



Here multiple micro-services both read and write from the same data store. Instead, each micro-services should have its own data store, and any other micro-service wishing to access needs to do so via the public API of the micro-services that owns it.

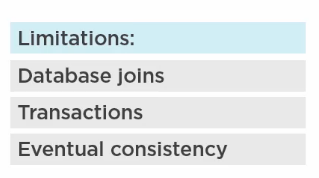


Now, there are some obvious trade-offs here:

\* By splitting our data out, it means that we can no longer perform database joins across the data that’s own by two different micro-services. Instead, we are going to make separate calls to each database.

\* We can no longer update two tables owned by different micro-services within a single database transaction. Instead,

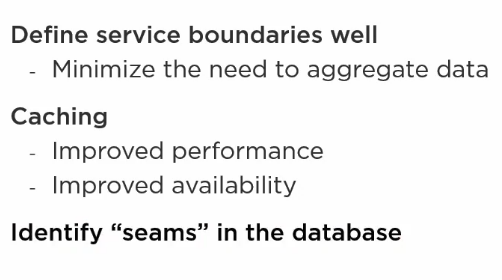
\* We’d either have to use distributed transactions, which are very complex to implement, or, more commonly design our system to work in what’s called an eventually consistent manner where we may have to wait a while for the overall state of the data to be fully consistent. What this means in practice is that when a single business operation requires updates to more than one data store, there’s going to be a sort window of time when the change has been made in one data store but not the others. And so you need to develop your application in a way that is able to cope with this temporary inconsistency.



**Mitigating Data Ownership Limitations:**

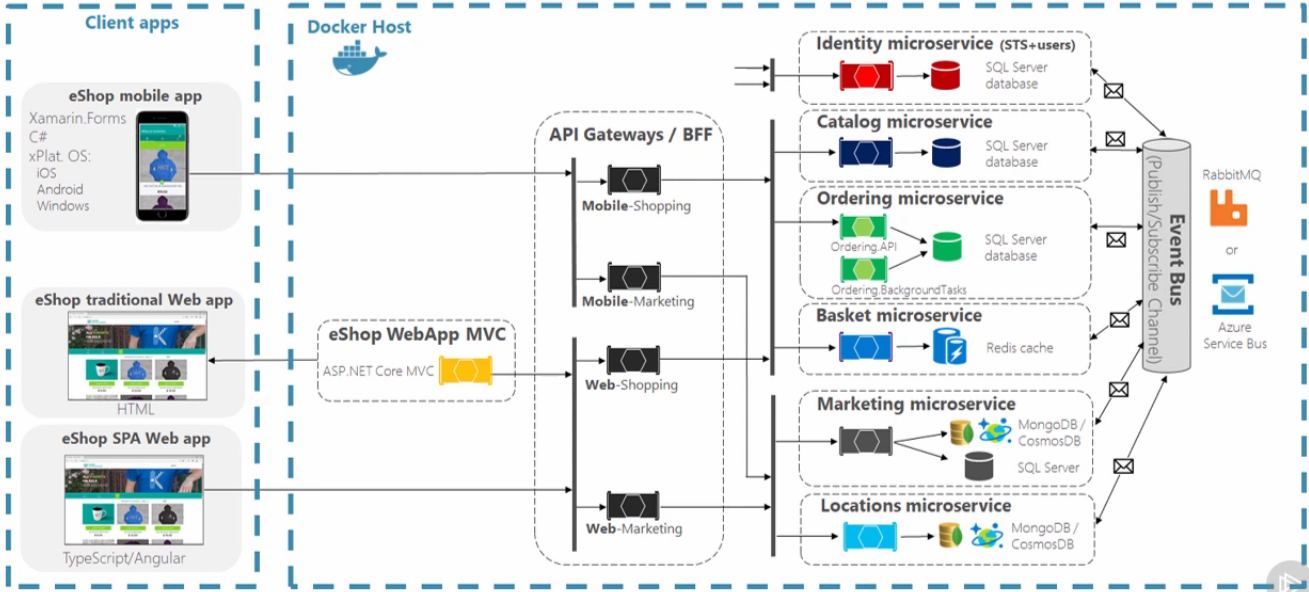
If we define our service boundaries well, we can minimize the need to aggregate the data from multiple micro-services to perform a single operation.

Another approach is that one micro-service might hold its own cache of a subset of a data that’s owned by another micro-service. An example of this might be that rather than having to constantly talk to a user profile service to get the name or email address of a user, we could maintain our own cached copies of this data. And this approach greatly improves performance because it removes the need for lots of chatty networks communication and improves availability since now our micro-service is able to do its job even when the user profile service is unavailable. It’s important that when you’re considering how to break your application into micro-services, that you identify natural seams in the database schema, and this will help you avoid the performance penalty of having to involve multiple micro-services in a single business operation.



**EshopOnContainers Architecture**

To illustrate how micro-services should own their own data, let’s take a look at the eShopOnContainers sample application that we already introduced.

We can see that the catalog service uses a SQL Server Database. The ordering service also uses SQL Server, but it’s a different database. Neither of these services can see or access each other’s data by going directly to the database.

The Basket Service uses a Redis cache, and this is a more appropriate choice for short-lived data.

We can also see that document databases such as Cosmos DB or Mongo have been chosen by some of the Micro-Services in this application.

And so, this Architecture illustrate two key Micro-Services characteristics. First, **each micro-service in the eShopOnContainers owns its own data.** And, second, each micro-service is free to use the most appropriate database technology for the type of data it needs to store.

This break down of Micro-Services also highlights a way to approach one of the difficulties that we just discussed.

The Ordering Micro-Service has got a concept of an “*OrderItem”*, which represents a single item in your order. The “*OrderItem”* entity contains a *“ProductId”,* a *“ProductName”,* and a *“UnitPrice”.* But, shouldn’t the Catalog Micro-service be responsible for owning product names and prices? And yes, indeed, the Catalog micro-service has got an entity called *“CatalogItem”*, which has an *“Id”,* a  *“Name”* and a *“Price”.* So, when you order something, the Ordering micro-service records not only the “*Id”* of the Product you ordered, but also takes a copy of its name and price. And so we’ve actually got duplicate information, which is sometimes referred to as de-normalization in databases.

Now this means that if we were to update a product’s name or price in the Catalog Micro-Service, then the information in the “*OrderItem*” would become stale. So, wouldn’t it be better then if the Ordering Micro-service didn’t store the product name and price and instead asked the Catalog Micro-service whenever it needed it. Well, actually may be not. For one thing that wouldn’t perform quite so well as now we’ve introduced a network call. But, another consideration is that it might actually a good thing for your order to include the name and price of the product as it was at the time when you placed the order. In other words, the Ordering Micro-Service doesn’t really care about what the current price and name of the product are. What’s it cares about is what they were at the time you placed the order. So, in fact, this isn’t really, duplicate information. These pieces of data have different meaning within the context of each micro-service. And so, duplication of data between micro-services and the ability to do a direct database join between the data owned by two micro-services is not necessarily the problem that it might first appear to be.