Dependencies and Data Sharing

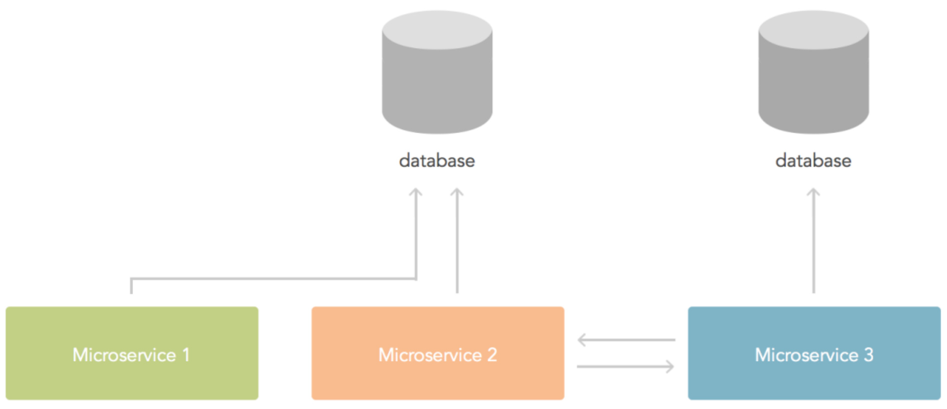
The problem of dependencies

In a microservice-based architecture, services are modeled as isolated units that manage a reduced set of problems. However, fully functional systems rely on the cooperation and integration of its parts, and microservice architectures are not an exception.

In a traditional monolithic application, dependencies usually appear as method calls. It is usually a matter of *importing* the right parts of the project to access their functionality. In esence, doing so creates a dependency between the different parts of the application. With microservices, each microservice is meant to operate on its own. However, sometimes one may find that to provide certain functionality, access to some other part of the system is necessary. In concrete, some part of the system needs access to data managed by other part of the system.

This is what is commonly known as data sharing: two separate parts of a system *share* the same data. If you are familiar with multithreaded programming you have a taste of how hard data sharing can get. However, in contrast to multithreaded applications, data sharing in a distributed architecture has its own sets of problems:

- Can we share a single database? Does this scale as we add services?
- Can we handle big volumes of data?
- Can we provide consistency guarantees while reducing data-access contention using simple locks?
- What happens when a service developed by a team requires a change of schema in a database shared by other services?



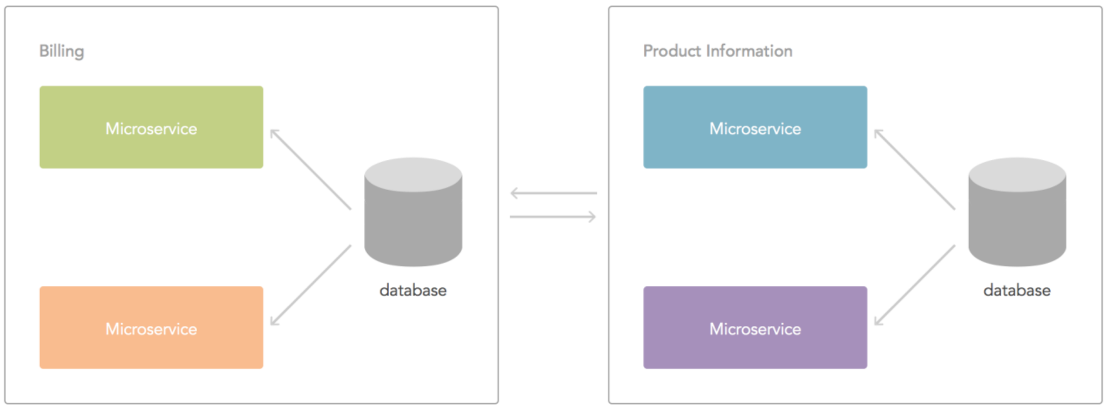
We will now study how some of these questions are answered in practice.

Separating concerns

Before going back to our problem of shared data and calls between services we need to take a step back and ask ourselves the obvious question: if we have these problems, could it be that we made a mistake while modeling our data or APIs? Certainly. This is why we need to talk about what we mean by separate concerns in greater detail. This can be described with the help of two concepts:

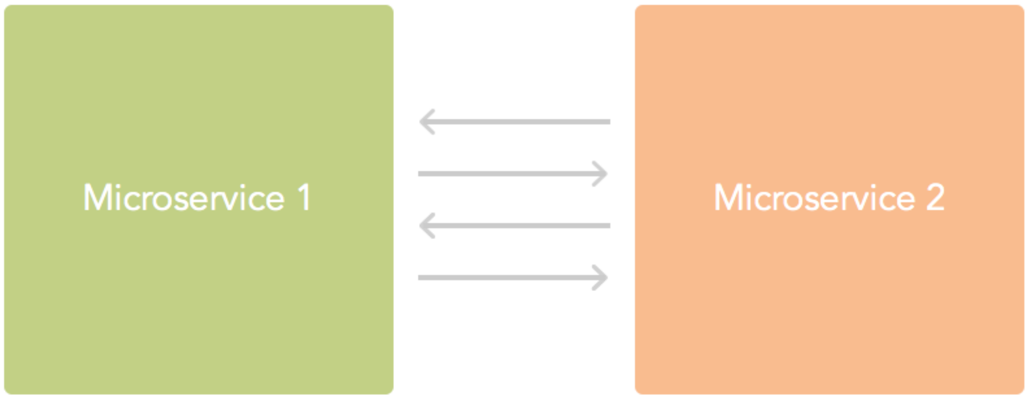
- Loose coupling: which means microservices should be able to be modified *without* requiring changes in other microservices.
- Problem locality: which means related problems should be grouped together (in other words, if a change requires an update in another part of the system, those parts should be close to each other).

In concrete, loose coupling means microservices should provide clear interfaces that model the data and access patterns related to the data sitting behind them, and they should stick to those interfaces (when changes are necessary, versioning, which we will discuss later, comes into play). Problem locality means concerns and microservices should be grouped according to their problem domain. If an important change is required in a microservice related to billing, it is much more likely other microservices in the same problem domain (billing) will require changes, rather than microservices related to, say, product information. In a sense, developing microservices means drawing clear boundaries between different problem domains, then splitting those problem domains into independent units of work that can be easily managed. It makes much more sense to share data inside a domain boundary if required than share data between unrelated domains.



The case for merging services into one

One important question you should ask yourself when working with separate microservices inside a problem domain is: are these services talking too much with eachother? If so, consider the impact of making them a single service. Microservices should be small, but no smaller than necessary to be convenient. Bam! Data sharing and dependency problems are gone. Of course, the opposite applies: if you find your services getting bigger and bigger to reduce chattiness, then perhaps you should rethink how your data is modeled, or how your problem domains are split. Trying to keep balance is the key.



Two microservices sending lots of messages back and forth candidates for turning into a single service

By the way, remember improving your solutions through iteration is part of the benefits of the microservices approach. Do your best effort to get things right from the beginning, but know you can make changes if things don't work out.

Data sharing

Now that we understand that data sharing may not be the only answer, we will focus on what to do when it is. We will split shared data in two groups: static data and mutable data.

Static data

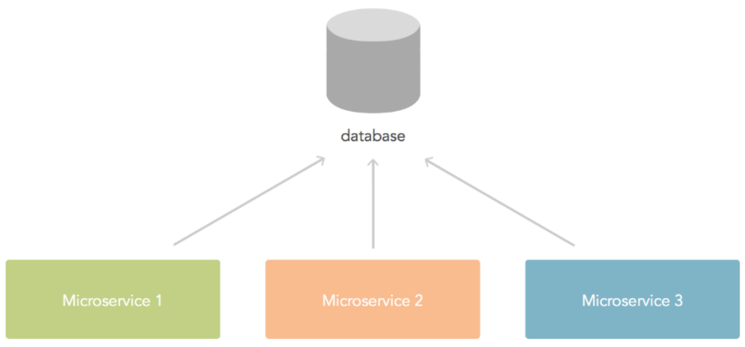
Static data is data that is usually read but rarely (if ever) modified. Sometimes it is necessary to study access patterns after the system goes live before finding out what data can be considered *static*. The nice thing about static data is that even if it is shared, no locks or consistency algorithms are necessary. All services can read the data concurrently and not care about any other readers. However you can choose different approaches to store this data according to your needs:

- Keep it in a database: this may or may not be a good approach according to the database you have picked. Two things to keep in mind: 1) Is it sensible to have each service query this database through the wire each time this data is required? 2) Can transactional updates to the data be performed whenever this data needs to be updated (even if it is once in a while)?
- Embed it in the code or share it as a file: if the data is small enough, it may make sense to embed it in the code or distribute it as part of a file to be deployed with each service. Things to keep in mind: 1) How easy is it to atomically restart all services sharing this data? 2) Is it small enough to not cause performance issues when loading each service?
- Make it into a service: similar to the database approach with the added benefit of being able to make arbitrarily complex decisions about how the data is sent over the wire and who can access it.

Mutable data

As we have mentioned before, the biggest problem with shared data is what to do when it changes. Suppose our microservice architecture is the heart of an online game distribution service. One of our microservices handles the game list of a customer. Other microservice handles the purchase of a game. When a customer purchases a game, that game is added to the list of games he or she owns. Our purchase microservice needs to tell our game-list microservice of the games that are added to a customer's list. How can we approach this problem? What follows is a list of common approaches to the problem. We will describe each and their advantages and disadvantages (note this is not an exhaustive list).

- Shared database
- Another microservice
- Event/subscription model
- Data pump model



Shared database

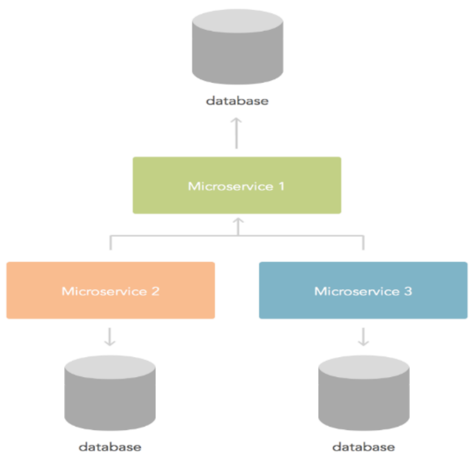
We have noted some of the problems with the shared database approach before, so we will now focus on what we can do to avoid them. When dealing with shared data across databases (or tables within a database) there are essentially two approaches: transactions and eventual consistency.

Transactions are mechanisms that allow database clients to make sure a series of changes either happen or not. In other words, transactions allow us to guarantee consistency. In the world of *distributed systems*, there are distributed transactions. There are different ways of implementing distributed transactions, but in general, there is a transaction manager that must be notified when a client wants to start a transaction. Only if the transaction manager (that usually communicates this intention to other clients) allows us to move forward the transaction can be performed. The downside to this approach is that scaling is usually harder. Transactions are useful in the context of small or quick changes.

Eventual consistency deals with the problem of distributed data by allowing inconsistencies for a time. In other words, systems that rely on eventual consistency assume the data will be in an incosistent state at some point and handle the situation by postponing the operation, using the data as-is, or ignoring certain pieces of data. Eventual consistency systems are easier to reason about but not all data models or operations fit its semantics. Eventual consistency is useful in the context of big volumes of data.

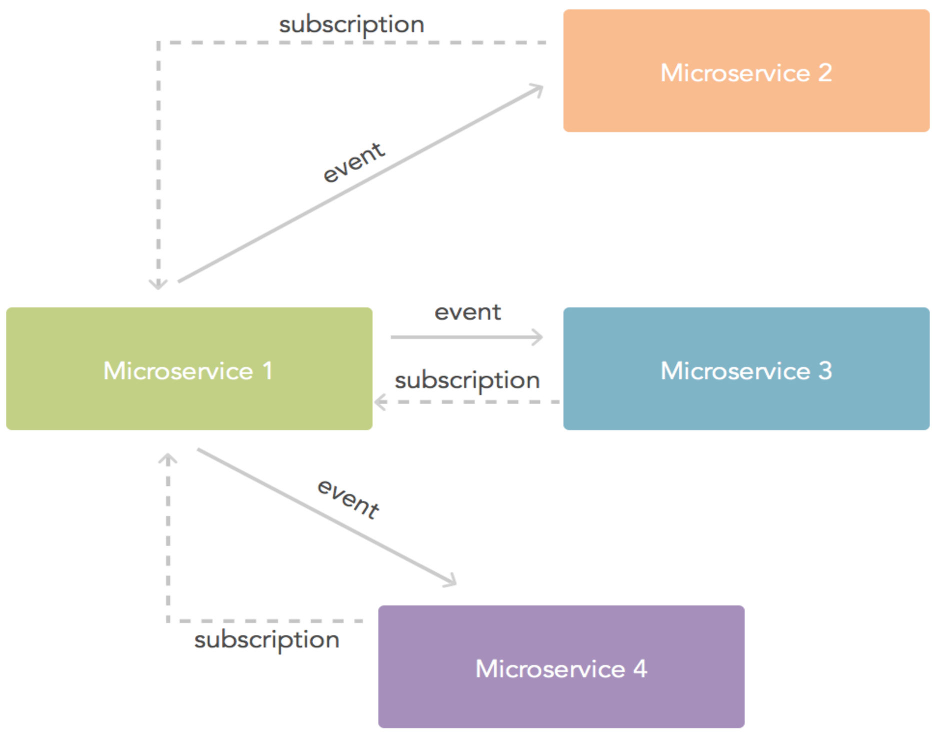
When facing the problem of a shared database, try very hard to keep the data in a single place (i.e. not to split it). If there is no other option but to split the data, study the options above in detail before committing to any.

Another microservice



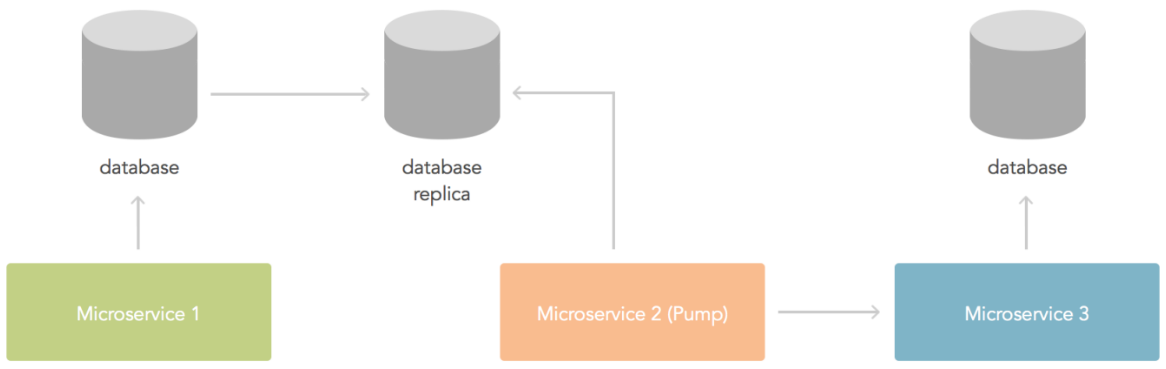
In this approach rather than allowing microservices to access the database directly, a new microservice is developed. This microservice manages all access to the shared data by the two services. By having a common entry point it is easier to reason about changes in various places. For small volumes of data, this can be a good option as long as the new microservice is the only one managing the data. Consider if this is something you can do and whether the microservice can scale to your future requirements.

Event/subscription model



In this approach, rather than relying on each service fetching the data, services that make changes to data or that generate data allow other services to subscribe to events. When these events take place, subscribed services receive the notification and make use of the information contained in the event. This means that at no point any microservice is reading data that can be modified by other microservices. The simplicity of this approach makes it a powerful solution to many use cases, however there are downsides: a good set of events must be integrated into the generating microservice and lost events are a possibility. You should also consider the case of big volumes of data: the data gets sent to as many subscribers as registered.

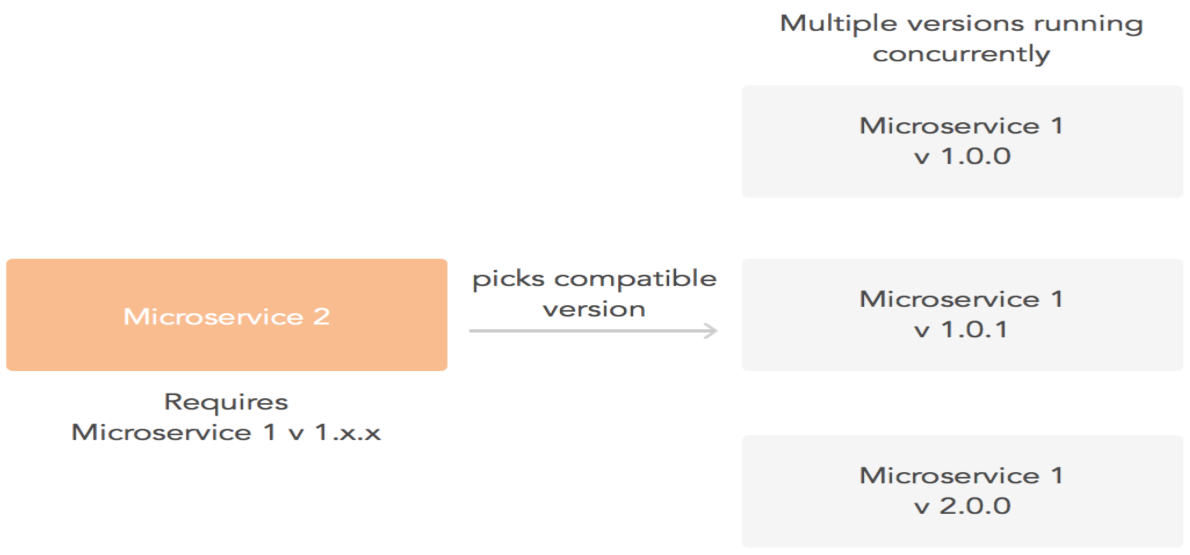
Data pump model



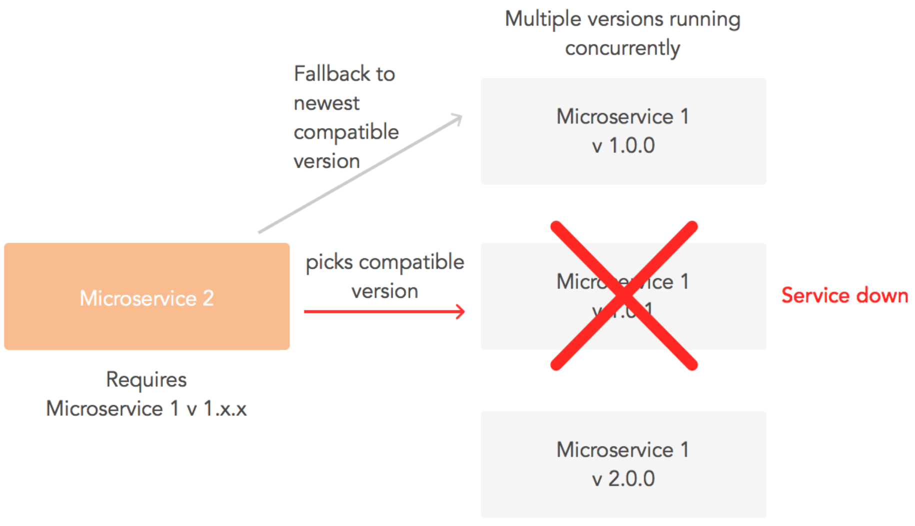
This is related to the eventual consistency case and the additional microservice case: a microservice handles changes in one part of the system (either by reading from a database, handling events or polling a service) and updates another part of the system with those changes atomically. In esence, data is pumped from one part of the system to the other. A thing to keep in mind: consider the implications of duplicating data across microservices. Remember that duplicated data means changes in one copy of the data create inconsistencies unless updates are performed to each copy. This is useful for cases where big volumes of data need to be analyzed by slow processes (consider the case of data analytics, you need recent copies of the data, but not necessarily the latests changes). For long running pumps, remember that consistency requirements are still important. One way to do this is to read the data from a read-only copy of the database (such as a backup).

Versioning and failures

An important part of managing dependencies has to do with what happens when a service is updated to fit new requirements or solve a design issue. Other microservices may depend on the semantics of the old version or worse: depend on the way data is modeled in the database. As microservices are developed in isolation, this means a team usually cannot wait for another team to make the necessary changes to a dependent service before going live. The way to solve this is through versioning. All microservices should make it clear what version of a different microservice they require and what version they are. A good way of versioning is through semantic versioning, that is, keeping versions as a set of numbers that make it clear when a breaking change happens (for instance, one number can mean that the API has been modified).



The problem of dependency and changes (versions) rises an interesting question: what if things break when a dependency is modified (in spite of our efforts to use versioning)? Failure. We have discussed this briefly in previous posts in this series and now is good time to remember it: graceful failure is *key* in a distributed architecture. Things will fail. Services should do whatever is possible to run even when dependencies fail. It is perfectly acceptable to have a fallback service, a local cache or even to return less data than requested. Crashes should be avoided, and all dependencies should be treated as things prone to failure.


Multiple versions running concurrently
Fallback to newest compatible version
Microservice 1 v 1.0.0
Microservice 2
picks compatible version
Microservice 1 v 1.0.1
Service down
Requires Microservice 1 v 1.x.x
Microservice 1 v 2.0.0