Open in app ↗

Search Write

Get unlimited access to the best of Medium for less than $1/week. Become a member

Sharing Data Between Microservices

Denat Hoxha · Follow

7 min read · Oct 24, 2022

1.8K 30

Two robots sharing a sunset moment in NYC while enjoying a cocktail (DALL-E)

When I started working with microservices, I took the common rule of “two services must not share a data source” a bit too literally.

I saw it stapled everywhere on the internet: “thou shalt not share a DB between two services”, and it definitely made sense. A service must own its data and retain the freedom to change its schema as it pleases, without changing its external-facing API.

But there’s an important subtlety here that I didn’t understand until much later. To apply this rule properly, we have to distinguish between **sharing a data source** and **sharing data**.

Why sharing a data source is bad

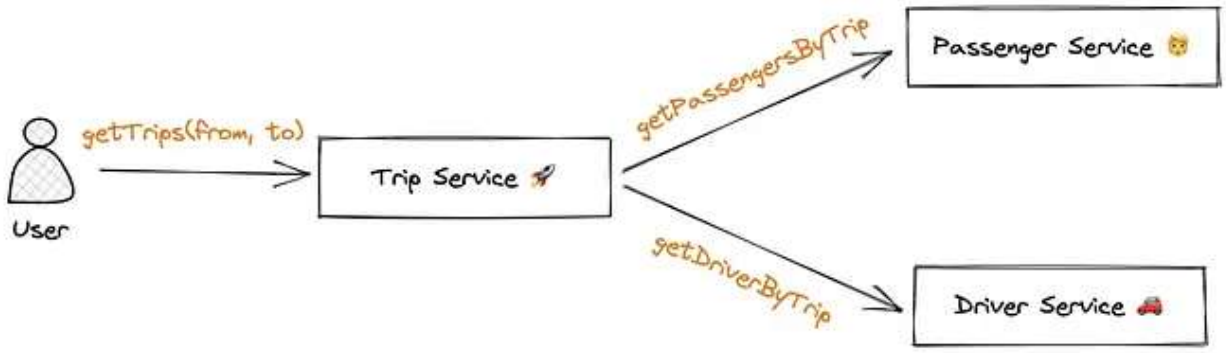
An example: the Products service should own the `products` table, and all the records in it. They expose this data to other teams via an API, a `products` GraphQL query, and the creation of these records via a `createProduct` mutation.

The Products service has ownership over the products' source of truth, and no other team should reach into this directly, ever. If they want data out of it, they should ask the Products service for it via the contract (API) they adhere to. Under no circumstance should you allow direct access to the database, or you’ll lose the freedom to make changes to your schema. I learned this the hard way.

Sharing data is OK

The fact is, services need data that belong to other services.

For example, a Trip service will need access to passengers (from Passenger service) and drivers (from Driver service) to serve trip overviews.



A simple architecture with three services

Trip service asks each respective service for its data, synchronously, to fulfill the original request (`getTrips`). We can rest assured that the data is fresh, and the requesting client will get a *strongly consistent* view of the data (some of you might see where I'm heading at this point ;).

This synchronous request/response model to transmit data between microservices is a very natural mental model for teams that start off in microservices, at least in my experience. You need some data, you know where to get it, you ask the owning service for it, and it gives the data to you, *on demand*.

On top of that, serving fresh, *strongly consistent* data was a no-brainer for the teams I was in. Strongly consistent data means up-to-date data, the absolute "freshest" piece of data, straight from the source (of truth). To me, back then,

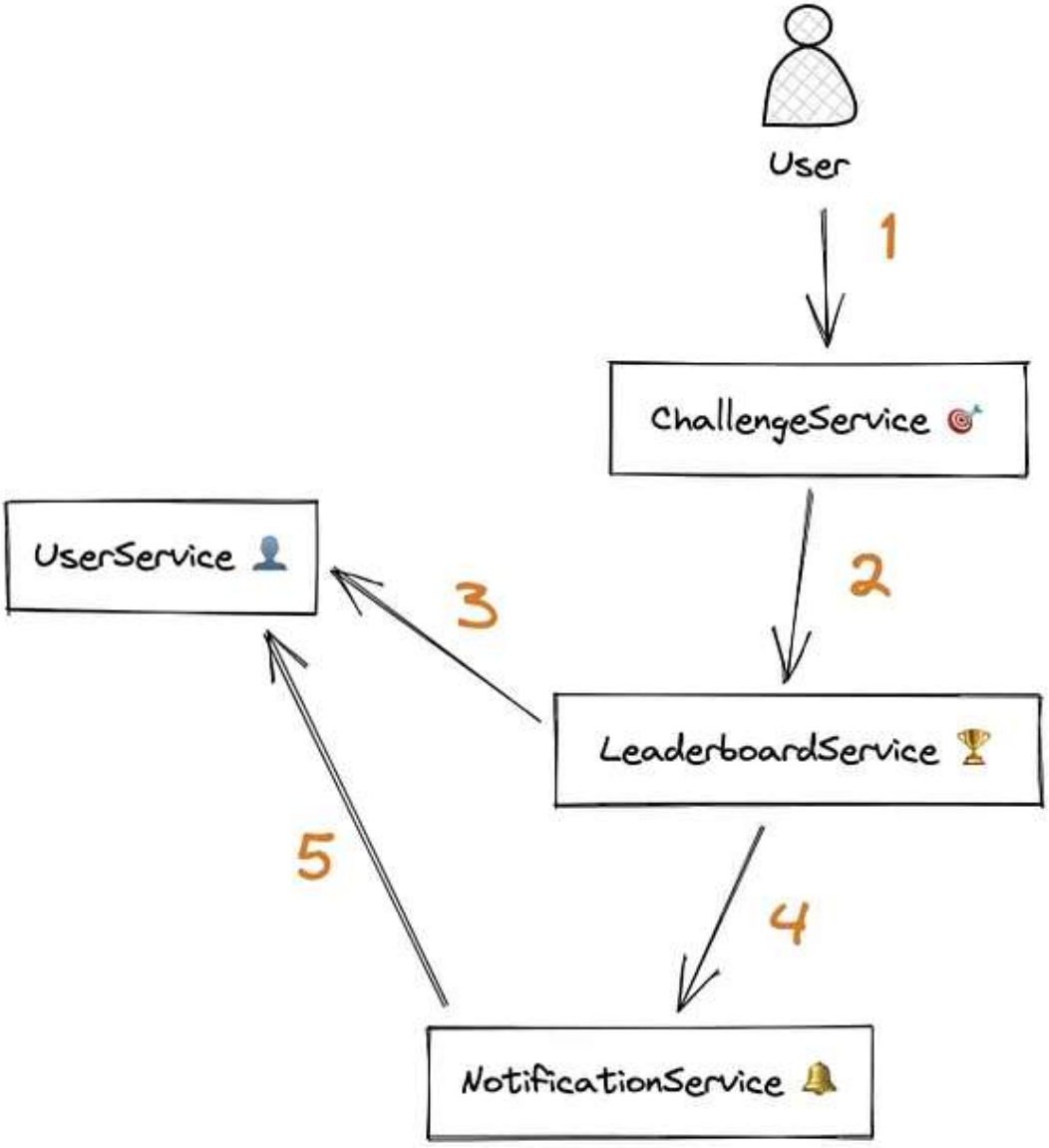
serving anything other than consistent data was unacceptable. How could you serve anything other than up-to-date data? Anything else would be a lie!

We applied these patterns as dogma because we saw no other way, and above all, it felt natural.

Synchronicity and strong consistency don't scale

Architectures that rely heavily on synchronous requests and strong consistency do not scale well. Sometimes it's just not feasible, or strictly necessary, to always go straight to the source for your data needs.

The Trips service example above looks neat at first, but it's rarely the case that systems remain that simple. New services are born, and they'll require data from existing services. Sticking to the synchronous request pattern will, in time, have you end up with a tangled web of requests between services. Here's a scenario:



Example flow with synchronous requests

1. A user completed a challenge, runs a `completeChallenge` mutation to Challenge service
2. After storing the completion, the Challenge service lets the Leaderboard service know, so it can update the leaderboard
3. The Leaderboard service asks User service for user display names and avatars to build the new leaderboard state

4. The Leaderboard service sees that there's a new leader in the new leaderboard state, and lets the Notification service know so it can notify participants that there's a new leader!
5. Notification service asks User service for the up-to-date email addresses of the users in that particular leaderboard, so it can send emails out

The User service is clearly a point of contention here: everyone is in one way or another depending on it. Imagine this service being down: it'll render most of the other services down as well. Not only that, but you're gonna have to make sure to keep this server buffed up at all times with more replicas and a high-performant DB to keep up with the demand.

On top of that, each hop in this chain of requests adds latency to the entire request. Each hop has the potential to add an exponential amount of latency because every service in the dependency chain could fire off more than one request to its own dependencies. Before you know it, you've hit unbearable levels of latency.

Finally, every additional dependency in the request chain increases the likelihood of the entire request chain failing. In a request chain involving five services with an SLA of 99.9% (~9h of yearly downtime), the composite SLA becomes 99.5%. That's almost 2 days of downtime per year!

We can avoid all of these downsides by asking one question: do services really need up-to-date data?

The Notification service (step 5) arguably does. If a user changed their address and the Notification service doesn't know about it, it'll risk sending

an email to the wrong address and not getting the notification to the intended user.

The Leaderboard service, on the other hand, probably doesn't need the up-to-date display names and avatars to build the leaderboard — it's not that big of a deal if users see stale avatars or display names.

As you can see, services have different data consistency needs. There are trade-offs that we can use as leverage to apply different data-sharing methods and build a more robust distributed system.

Enter Eventual Consistency

It's at this point in my career that I discovered that services can maintain a copy of other services' data, locally in their own DB tables. It comes with the responsibility to retain that data via events or polling.

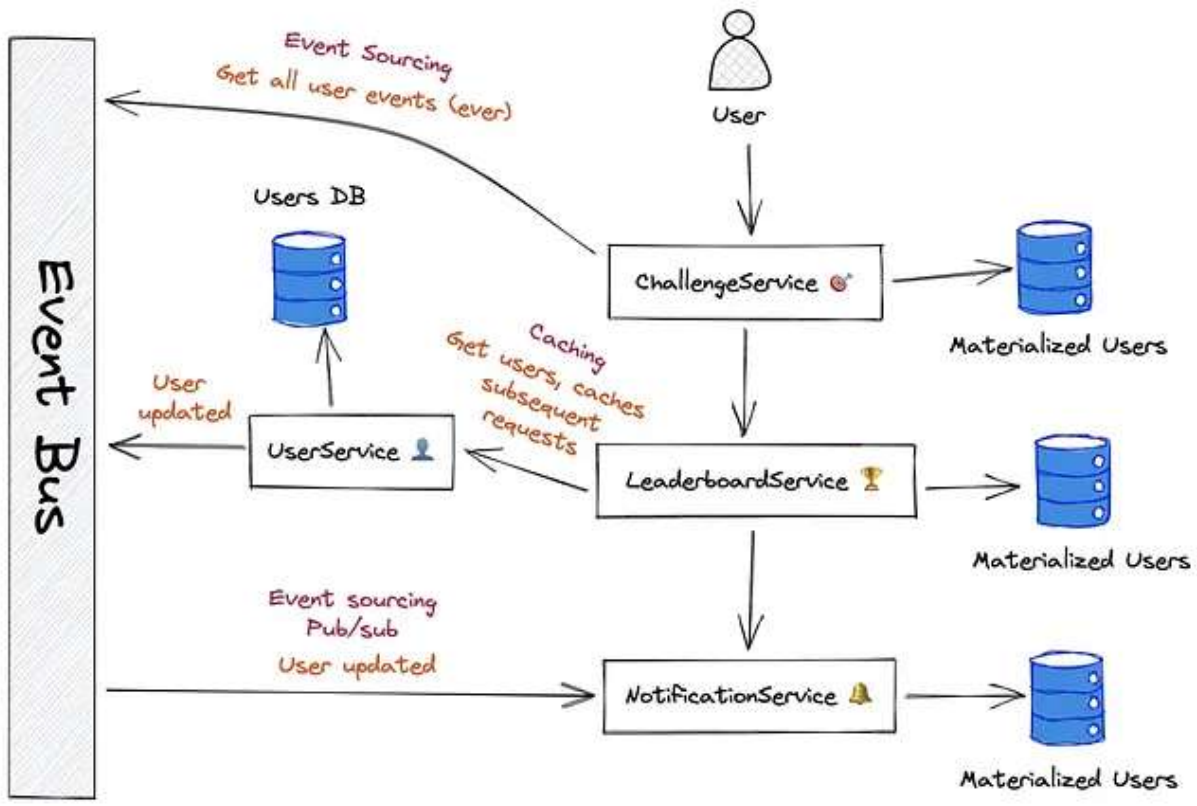
Included in this package is the fact that the data may be stale for some time, but that it'll eventually be updated, meaning the data is *eventually consistent*. We can't guarantee that the data isn't stale, but we can guarantee that we'll eventually catch up.

The moment it "clicked" for me was when I thought about it from the perspective of a backend service depending on a public weather API for weather data. Instead of retrieving weather data for Prishtina or Berlin every time a user from those respective cities needs weather data, I cache it (maybe multiple times a day) by materializing it in a local table and serving cached data to those users. I've made the trade-off in favor of eventual consistency because it's not crucial for my users to see *the freshest data*, it's OK if it's a few hours stale.

Going back to the Challenge example: we can cut off many synchronous dependencies to the User service by just maintaining a local copy of users in the services:

- Leaderboard service can maintain a local copy of users and avoid having to make requests to User service. No one really minds if the data is a bit old, it's not a showstopper if someone sees a slightly old avatar.
- Challenge service can do the same; say if it exposed a `getChallengeDetails` query and needed user display names and avatars to show the current challenge participants — it can also serve this eventually consistent data from its own materialized users table.
- Notification service, although a bit more sensitive, can also utilize data sharing to remove its dependency on the Users service. It can materialize users locally and maintain a best-effort updated state by listening to User updated events to ensure it has the most up-to-date emails.

Although we didn't talk much about *how* services share this data (a topic for another time), a final example architecture would make use of a combination of event sourcing and caching. Here's a sneak peek into what this kind of architecture would look like:



Example architecture with two main methods of sharing data between services: event sourcing and caching

If you want more examples, take a look at How to Share Data Between Microservices on High Scale by Shiran Metsuyanim, an engineer at Fiverr. It's a great post that shows how to maintain robustness when adding a new service. It starts out by laying down the constraints, and then discussing the trade-offs between synchronous, asynchronous, and hybrid solutions.

Conclusion

I wanted to get this point across to developers that, like me a few years ago, are stuck in the literal sense of "do not share data" but must realize that this only applies to not sharing the source of truth. Maintaining a copy of a service's data in another service's domain is perfectly fine, and embraces the spirit of eventual consistency.

Thanks for reading. What other microservices topics would you like to read about? What isn't discussed enough? Feel free to reply!

Eventual Consistency | Microservices | Distributed Systems | Software Architecture | Programming

Written by Denat Hoxha

Follow

634 Followers

Software Engineer, helping the world become data fluent @DataCamp. Personal tech blog: denhox.com

More from Denat Hoxha