UDC 004-04

S.O. Diakov, T. E. Zubrei

MANAGING STATE IN A MICROSERVICE

Abstract: this report takes a close look at the problem of state in modern applications, how it affects the architecture. Popularity of microservices raised the issue of information acquisition. Analyzing solutions to this problem is the main topic of the article. Multiple approaches to communication were examined with a focus on defining its advantages and disadvantages. Coupling, availability and consistency were analysed, suitable use cases were defined and ways to negate drawbacks were suggested. Microservice architecture implies multiple sources of data so accessing required information is a task in and of itself. This article will review many approaches to dealing with state within a service: stateless microservice, command query responsibility segregation principle, on demand direct communication with and without caching on top and event sourcing.

Keywords: microservice architecture, state management, inter-service communication patterns.

What is state

Everything in the world has its state. It’s a set of characteristics describing an object. State is a natural thing as well as its ability to change over time. It’s the last part that is so quirky. It’s not enough to be aware of something you also need to keep track of its mutations. Imagine if your favourite café had a menu without prices. It would be definitely cheaper for a manager to account for price fluctuations since they wouldn’t have to update the menu if coffee beans became more expensive. It may look like a sound idea, but why haven’t you seen its rise in popularity? That’s because the responsibility of providing customers with a price is now laid on customers themselves. Each time you want to order something you would as your waiter for the latest price. Doesn’t sound very convenient, but it sure made menu printing cheaper.

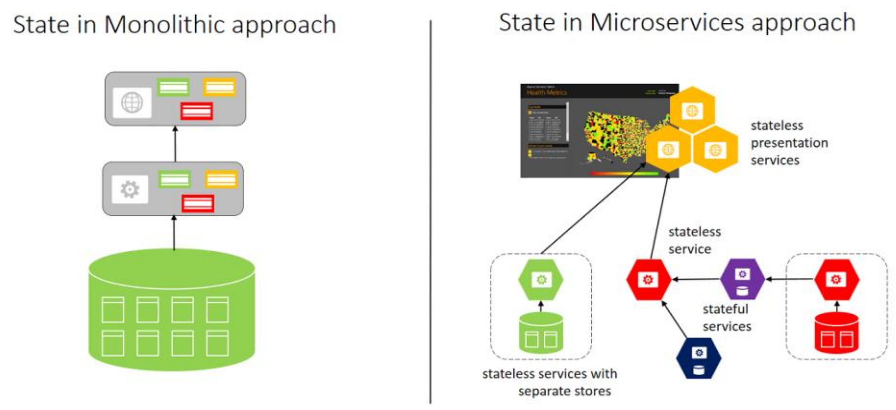
State in Monolithic approach
State in Microservices approach
stateless presentation services
stateless service
stateful services
stateless services with separate stores

Figure 1. State in microservices comparing with traditional monolith

© **S.O. Diakov, T. E. Zubrei**

We face such dilemmas every day. We need information to make an educated decision, but how we acquire this information may vary from case to case. It’s always a game of balancing pros and cons. Dealing with this information is managing its state. It’s a necessary precondition for any business logic. While you may be tempted to always focus on the process that actually makes you money you need to acknowledge the importance of data which allows you to do the math so to say.

Microservice architecture implies multiple sources of data so accessing required information is a task in and of itself. This article will review many approaches to dealing with state.

Stateless service

If you can make it stateless you should – it’s a simple rule of thumb. Managing state is no ordinary task and can lead to numerous complications. Therefore, by avoiding it altogether you can focus on what’s important – business logic.

There are cases when the input stream has all the information required to decide relieving you from needing to have additional data for reference. A lot of ETL [1] (extract transform load) operations are stateless. You may want to aggregate data for a period of time or save internally for a later processing.

Having service fully devoid of state may seem unrealistic and it is a rare sight. Most practical use cases involve 2 or more data sources being combined to make a proper decision. With great power comes great responsibility and having stateless service shouldn’t be a sole aim of architectural design as it may lead you to wrong context cuts and even more coupling.

For instance, let’s imagine we have an email service which whole idea is to send text messages via 3rd party providers. Then we have other services which may require this functionality, like registration completion or notification about business process evaluation results. You may have a desire to make email service stateless in a way that it receives requests, transforms it into a text and sends it to some mail provider for further dispatching. Sounds simple enough but there’s a caveat – who should coordinate when to send email and provide all the necessary data. Email service is stateless, it can’t do all of these things otherwise we will break this design. Then other services that make requests have to take care of all the requirements. Imagine there are 5 of them. You would need to have all of them be aware of the existence of email service and fulfil the contract. It’s an additional responsibility those services may not necessarily were designed to ensure.

On the contrary API gateway [2] would be a great example of a stateless service which applies some predefined rules to route requests. The most common use case is to unify API between multiple providers of similar data. With no internal state there’s nothing to worry about rather than the actual routing – main business logic. While this pattern has its merits, simplicity is definitely one of them, you still need to pay attention to scalability and throughput since gateway may be a potential bottleneck of the system.

While it’s hard to come by a great opportunity to make microservice fully stateless it’s one you shouldn’t miss out on. When making this decision consider whether you are taking this responsibility from a new service and putting it on another (or even multiple ones).

Bounded contexts and concept of external data

The first thing in designing microservices – defining boundaries. Each service is supposed to cover some piece of business logic. Slice too much and you will come up with another monolith. Slice too thin and your inter-communication will take the main focus rather than desired business logic. Slice in the wrong direction and tight coupling will slow everything down, from development to deployment.

It’s a fine line and should be treated with great respect. This is when the concept of bounded contexts enters the scene. It’s not a new thing but it has raised in popularity in recent years becoming the main pillar of DDD [3] (domain driven design) in pair with ubiquitous language. Basically, it’s a way to split the system on the higher-level using domains as an abstraction over business processes. This focus on business problems and operations allows us to better model real life use cases.

However, with a concept of boundaries [4] the natural split is inevitable: internal and external data. Each context has boundaries with another one because they all work together to perform required tasks. This is the place of communication and it’s the key for having architecture suit the use case.

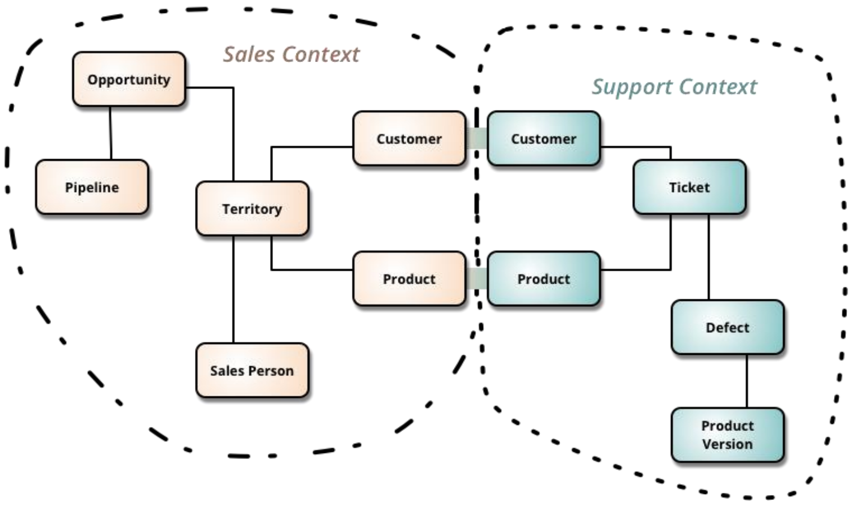

Figure 2. Boundaries between 2 contexts

Sharing is only natural for people. The same is for microservices. Each one of them consumes some data and provides another. However, service may also require data from another bounded context, in other words external data. The main difference is that it doesn’t own this information, it puts responsibility to maintain consistency on the provider of the data. Moreover, this service can’t be queried for such data since it’s not their goal. But still this information is vital for internal decision making and hence it should be accounted for how to acquire it.

Dual read/write service model

The simplest way to share data is to just share data storage. Then both services are working with the same dataset. However, this solution contradicts one of the main ideas of microservice architecture: one service – one database. This is true, sharing a storage leads to tight coupling which is a very bad idea in an everchanging world.

Not all hope is lost for this approach though. There are benefits for having the same storage and intelligent use may even lower all the drawbacks. CQRS [5] (command query responsibility segregation) describes 2 separate models: one for reading and another for writing. It allows us to have each part take care of its concerns. It’s a powerful tool when the way you write data substantially differs from the way you read it. Then you can have 2 services: reader and writer which both share the same storage but focus on different things.

What can we gain from such separation? Well, better scalability for one. There are cases when one part is much more loaded then the other. You may want to have 10 writers and only one reader. In the standard all-in-one implementation, you would have all of them doing both jobs which would slow down the read operations as all the resources would be taken by writing.

Clear separation of concerns is another benefit. Writing stage may require additional data enrichment and transformations to prepare them for read optimised view.

Having said that drawbacks are heavy as well to maintain a world order and balance. It is a right coupling any way you look at it. Any schema change is a double pain because now you have to deal with both services. Moreover, they work in tandem, meaning you need to deploy and test them exclusively together since the next version of the reader may be incompatible with the previous writer. You can tackle this difficulty by applying such architecture only when the schema is relatively stable. If you can develop both sides independently most of the time then this coupling won’t make a huge difference.

However, there’s a coupling in the core of this idea and there’s nothing to do with it. Making the writer prepare data optimized for read operations requires the former to know a lot about the latter. You need to know how you query to store data appropriately. You should consider it when deciding the application of this pattern.

On demand information

If you don’t know something – just ask for it. Any teacher would tell you that. And they will be right. There’s no shame in asking. Problem is to know who to come to. If I want to know more about the latest trends in football tactics I need to ask a sports teacher. But this would require that I already know about them which I may not. Same is applicable in a world of microservices. Service A calls service B to provide necessary information and return it to service C. That’s microservices in a nutshell some may say.

Let’s illustrate this using our previous example. We have template service which can return a template by id and mail service which needs a template for dispatching purposes.

How can we tight them together? Just make a call. We can define a client on mail service which will query a template. Then we use it on each dispatch. Problem solved and we can spend our time somewhere else. Or is it?

Take a step back and look at what we’ve done. We had two separate services with its concerns and responsibilities. Now they know about each other, now they need each other. What would happen if template service goes down? Well, you can’t send emails anymore. Even though you can actually dispatch, mail service is in perfect condition to do its work, it can’t because it depends on faulty service.

Let’s look from another angle. What is happening more often – template modification of email sending? I would say latter in most cases. This means that we make a lot of calls which result in the same response over and over again. Each time it slows the dispatching process to reaffirm that template is the same as it was a second ago.

Finally, you need to know that template service actually exists and where it is located, what format does it accept. Adding network issues and possible errors on the provider side we end up with a lot of cases needed to be covered on the client side. What to do if the service didn’t respond? What to do if service didn’t find such a template? All of these questions need to be answered otherwise you will have a nice time debugging production when someone asks you why an email wasn’t sent.

With all these drawbacks, it’s still a convenient way to set up communication. It leads to clear separation of concerns: template service manages templates and mail service only queries the latest version of it. Furthermore, you don’t duplicate the data having it exactly in one place with a service that manages it. If something goes wrong there’s only one place you need to modify. You can tackle network issues with scalability, finding service location with service registries and external routing (e.g. kubernetes [6] takes it in its own hands to connect two services via HTTP just by knowing their names).

Caching as intermediate solution

Adding cache on top of anything is always a complexity. Maintaining it consistent is a whole another topic worth of a few hundred-page book. However, it does solve a few problems: performance and availability in particular.

Caching [7] allows you to avoid making repeated calls for the same information – the main source of problems in the previous solution. Sounds like it makes a good solution perfect. Not quite. It’s true that having information locally removes the tight coupling to another service, but there’s a caveat – who is responsible for maintaining this data in a consistent state? Service, who doesn’t own this information. How can you guarantee consistency if you don’t have full control over the data? That’s the main question to ask yourself when considering caching.

There are multiple ways to solve this problem or at least lessen the blow. For instance, you can set up clever time to leave and retrieve data from the source from time to time. This always comes at a cost since you cannot always know whether an object was changed until the next cycle comes and you make a call. But you don’t always need to know. There’s no such concept as now in programming, we always work with delays and stale data. We can embrace it and simplify the architecture.

Alternatively, you can update cache asynchronously by consuming updates from source service and applying them to internal cache. However, there’s still a catch. Cache is a temporary storage, meaning you need to populate it each time you start a service. This logic is very different from updates but is essential as well. That’s a lot of work just to serve the cache not business needs.

Consuming open data

We can iterate from caching to permanent storage. All we need is for each service to provide a way to serve its data asynchronously. Event sourcing [8] covers this idea in depth, but in a nutshell, service has a log of events which they guarantee to keep in check. Then anyone can subscribe to that log and always have eventually consistent data. This way you completely decouple both sides. Subscriber has no idea who is writing the log. Publisher doesn’t care who will read this data.

The core difference of this approach is an idea that service needs to keep consistent other storage – the log. Achieving it is not a hard task, there are even automated solutions that can record all changes to the storage and publish them to some queue. Moreover, most databases are using log internally as its source of truth, having tables and collections as its snapshots.

In the example above, mail service can listen to updates for templates and store only its required fields internally. Then it will have everything it needs to work fully relying on itself.

Everything has its drawbacks and event sourcing is not an exclusion. First of all, it’s data duplication whichever way you look at it. You can’t just manually change field value in one database because it may be stored in dozens of others. Ironically it leads to better automation and transparency which is a good thing.

Secondly, log has its limitations. It’s not infinite and you may come to a problem where you need to populate the storage with historic data. However unlike with caching you would need to do this only once and this code may not even be a part of the service. Furthermore, log is not just a concept. It’s a concrete technology and every technology is faulty so you should keep in mind a case when you can’t either publish or subscribe because this component went down.

Finally, eventual consistency may scare you away though it is a very natural concept. As was said previously we also have some delays and rather than fight them we can use it to our advantage. In the end everything will be consistent but it’s us who control the speed we are heading at. Still it’s something to consider and not all cases can neglect this aspect.

Conclusion

State is a natural part of every process and managing it is as important as the process itself. There is no right way to do it, no silver bullet. Finding the right balance between benefits and drawbacks is the main task when deciding on any architecture.

Table 1

Comparison of architectures in state management

Approach	Advantages	Disadvantages
Stateless	No need to manage state	This responsibility may be put on other services
CQRS	Scalability and flexibility	Tight coupling between 2 services
Direct communication	Data deduplication and separation of concerns	Tight coupling between multiple services, availability concerns
Caching	Speed and circumstantial availability	Tight coupling, additional complexity
Event sourcing	Loose coupling, availability, flexibility	Data duplication

In the end your main focus should be on business logic. If you think like you paying too much attention to support communication take a look at other options, maybe one of them will suit you better.

REFERENCES

1. Beal V. ETL - Extract, Transform, Load // https://www.webopedia.com. Available at: https://www.webopedia.com/TERM/E/ETL.html
2. Richardson C. Pattern: API Gateway / Backends for Frontends // https://microservices.io. Available at: https://microservices.io/patterns/apigateway.html
3. Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software 1st Edition / E. Evans. — Addison-Wesley Professional, 2003. — 560c.
4. Fowler M. BoundedContext // https://martinfowler.com. Available at: https://martinfowler.com/bliki/BoundedContext.html.
5. Fowler M. CQRS // https://martinfowler.com. Available at: https://martinfowler.com/bliki/CQRS.html.
6. Connecting Applications with Services // https://kubernetes.io. Available at: https://kubernetes.io/docs/concepts/services-networking/connect-applications-service.
7. Handy J. Cache Memory Book, The (The Morgan Kaufmann Series in Computer Architecture and Design) 2nd Edition / J. Handy. – Burlington: Morgan Kaufmann, 1998. – 229 c
8. Kleppmann M. Making Sense of Stream Processing / M. Kleppmann. — Sebastopol: O'Reilly Media, Inc., 2016. — 172 c