Building microservices in 2019 and beyond

Microservice architecture is always evolving. Learn modern best practices to do it right.

BY STEN PITTET

"Microservices" is a popular, modern, software engineering organizational practice. The guiding principle of microservices is to build an application by splitting its business components in small services that can be deployed and operated independently from each other. The separation of concerns between services is defined as "service boundaries".

Service boundaries are closely tied to business demands and organizational hierarchy boundaries. Individual services may be tied to seperate teams, budgets, and roadmaps. Some example service boundaries might be "payment processing" and "user authentication" services. Microservices differ from legacy software development practices in which all components were bundled together.

This document will reference an imaginary startup called "Pizzup" to illustrate the application of microservices to a modern software business.

How to build microservices

Step 1: Start with a monolith

The first best practice of microservices is that you probably don't need them. If you don't have any users for your application chances are that the business requirements are going to rapidly change while you're building your MVP. This is simply due to the nature of software development and the feedback cycle that needs to happen while you're identifying the key business capabilities that your system needs to provide. Microservices do add exponential overhead and complexity to manage. For this reason it is much less overhead for new projects to keep all the code and logic within a single codebase as it's makes it easier to move the boundaries of the different modules of your application.

For instance with Pizzup we might have started with a very basic idea of the problem we want to solve for our customers: we want people to be able to order pizza online.

As we start thinking of the pizza ordering issue we will begin to identify the different capabilities required in our application in order to fulfil that need. We'll need to be able to manage a list of the different pizzas we can make, we'll need to let customers pick one or many pizzas, handle the payment, schedule the delivery and so on. We may as well decide that letting our customers create an account will facilitate re-ordering the next time they use Pizzup, and after talking to our first users we might figure out that live-tracking of the delivery and mobile support would definitely give us an advantage on the competition.

RELATED TUTORIAL

Tips for scripting tasks with Bitbucket Pipelines

Try this tutorial →

SUBSCRIBE

Sign up for more articles

Email

email@example.com

Subscribe

What was a simple need at the beginning quickly turns in a list of capabilities that you need to provide.

Microservices work well when you have a good grasp of the roles of the different services required by your system. They're much more difficult to handle if the core requirements of an application are still being worked out. It's indeed quite costly to redefine service interactions, APIs and data structures in microservices as you may have many more moving parts that need to be coordinated. This is why our advice is to keep things simple until you have collected enough user feedback to give you confidence that the basic needs of your customers are understood and planned for.

A word of caution though as building a monolith can quickly lead to complicated code that will be hard to break down in smaller pieces. Try as much as you can to have clear modules identified so that you can extract them later out of the monolith. You can also start by separating the logic from your web UI and make sure that it interacts with your backend via a RESTful API over HTTP. This will make the transition to microservices easier in the future when you start moving some of the API resources to different services.

Step 2: Organise your teams the right way

Up until now it would have seemed that building microservices is mostly a technical affair. You'll need to split a codebase in multiple services, implement the right patterns to fail gracefully and recover from network issues, deal with data consistency, monitor service load, etc. There will be a bunch of new concepts to grasp but one thing that must not be ignored is that you'll need to restructure the way your teams are organized.

> Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.
>
> - Conway's Law

Conway's Law is a real thing that can be observed in all types of teams, and if a software team is organized with a backend team, a frontend team and an ops team separated they will end up delivering separate frontend and backend monoliths that get thrown away to the ops team so that they can manage it in production.

This type of structure is not a good fit for microservices as each service can be seen as its own product that needs to be shipped independently of the others. Instead you should create smaller teams that have all the competencies required to develop and maintain the services they're in charge of. Werner Vogels, CTO of Amazon, described this situation with the phrase "you build it, you run it". There are great benefits to arranging your teams this way. First of all your developers will get a better understanding of the impact that of their code in production – this will help produce better release and reduce the risk of seeing issues released to your customers. Secondly your deployments will become a second nature for each team as they will be able to work together on improvements to the code as well as the automation of the deployment pipeline.

Step 3: Split the monolith to build a microservices architecture

When you've identified the boundaries of your services and when you've figured out how you can change your teams to be more vertical in terms of competencies you can start splitting your monolith to build microservices. Here are the

key points to think about at that time.

Keep communication between services simple with a RESTful API

If you're not already using a RESTful API now would be a good time to adopt it in your system. As Martin Fowler explains it you want to have "smart endpoints and dumb pipes". This means that the communication protocol between your services should be as simple as possible, only in charge of transmitting data without transforming it. All the magic will happen in the endpoints themselves – they receive a request, process it, and emit a response in return.

This is also where microservices can be distinguished from SOA by avoiding the complexity of the Enterprise Service Bus. Microservice architectures strive to keep things as straightforward as possible to avoid tight coupling of the components. In some cases you might find yourself using an event-driven architecture with asynchronous message-based communications. But once again you should look into basic message queue services like RabbitMQ and avoid adding complexity to the messages transmitted over the network.

Divide your data structure

It is quite common to have a single database for all the different capabilities in a monolith. When a user accesses its order you'll look directly in the user table to display the customer information, and the same table might be used to populate the invoice managed by the billing system. This seems logical and simple but with microservices you will want the services to be decoupled–so that invoices can still be accessed even if the ordering system is down–and because it allows you to optimize or evolve the invoice table independent of others. This means that each service might end up having its own datastore to persist the data that it needs.

It obviously introduces new problems as you will end up having some data duplicated in different databases. In this case you should aim for eventual consistency and you can adopt an event-driven architecture to help syncing data across multiple services. For instance your billing and delivery tracking services might be listening for events emitted by the account service when a customer updates their personal information. Upon reception of the event those services will update their datastore accordingly. This event-driven architecture allows the account service logic to be kept simple as it doesn't need to know all the other dependent services. It simply tells the system what it did and other services listen and act accordingly.

You can also choose to keep all the customer information in the account service and only keep a foreign key reference in your billing and delivery service. They would then interact with the account service to get the relevant customer data when needed instead of duplicating existing records. There isn't a universal solutions for these problems and you'll have to look into each specific case to determine what the best approach is.

Build your microservices architecture for failure

We've seen how microservices can provide you with great benefits over a monolithic architecture. They're smaller in size and specialized which makes them easy to understand. They're decoupled which means that you can refactor a service without having to fear breaking the other components of the system, or slowing down the development of the other teams. They also give more flexibility to your developers as they can pick different technologies if required without being constrained by the needs of other services.

In short, having a microservice architecture makes developing and maintaining each business capability easier. But things become more complicated when you look at all the services together and how they need to interact to complete actions. Your system is now distributed with multiple points of failure and you need to cater for that. You need to take in account not only cases where a service is not responding, but also be able to deal with slower network responses. Recovering from a failure can also be tricky at times as you need to make sure that services that get back up and running do not get flooded by pending messages

messages.

As you start extracting capabilities out of your monolithic systems make sure that your designs are built for failure right from the beginning.

Emphasize monitoring to ease microservices testing

Testing is another drawback of microservices compared to a monolithic system. An application that is built as a single codebase doesn't need much to have a test environment up and running. In most cases you'll have to start a backend server coupled with a database to be able to run your test suite.

In the world of microservices things are not as easy. When it comes to unit tests it will still be quite similar as the monolith and you shouldn't feel more pain at that level. However when it comes to integration and system testing things will become much more difficult. You might have to start several services together, have different datastores up and running, and your setup might need to include message queues that you did not need with your monolith. In this situation it becomes much more costly to run functional tests and the increasing number of moving parts makes it very difficult to predict the different types of failures that can happen.

This is why you'll need to put a great emphasis on monitoring to be able to identify issues early and be able to react accordingly. You'll need to understand the baselines of your different services and be able to react not only when they go down, but also when they're behaving unexpectedly. One advantage of adopting a microservice architecture is that your system should be resilient to partial failure, so if you start to see anomalies in the delivery tracking service of our Pizzup application it won't be as bad as if it were a monolithic system. Our application should be designed so that all the other services respond properly and let our customers order pizzas while we restore the live-tracking.

Embrace continuous delivery to reduce deployment friction

Releasing a monolithic system to production manually is a tedious and risky effort but it can be done. Of course we do not recommend this approach and encourage every software team to embrace continuous delivery for all types of development, but at the beginning of a project you might do your first deployments yourself via the command line.

This approach is not sustainable when you have an increasing number of services that need to be deployed multiple times a day. So, as part of your transition to microservices it is critical that you embrace continuous delivery to reduce the risks of release failure, as well as making sure that your team is focused on building and running the application, rather than being stuck deploying it. Practicing continuous delivery will also mean that your service have passed acceptance tests before going to production – of course bugs will occur but over time you will build a robust test suite that should increase your the confidence of your team in the quality of the releases.

Running microservices is not a sprint

Microservices are quickly becoming a popular and widely adopted industry best practice. For complex projects they offer a greater flexibility in the way you can build and deploy software. They also help identify and formalize the business components of your system, which comes in handy when you have several teams working on the same application. But there are also some clear drawbacks to managing distributed systems, and splitting a monolithic architecture should only be done when there's a clear understanding of the service boundaries.

Building microservices should be seen as a journey rather than the immediate goal for a team. Start small to understand the technical requirements of a distributed system, how to fail gracefully and scale individual components. Then you can gradually extract more and more services as you gain experience and knowledge.

The migration to a Microservices architecture does not need to be accomplished in one holistic effort. An iterative strategy to sequentially migrate smaller components to

microservices is a safer bet. Identify the most well defined service boundaries within an established monolith application and iteratively work to decouple them into their own microservice.

Summary

To recap, Microservices is a strategy that is beneficial to both raw technical code development process, and overall business organization strategy. Microservices help organize teams into units that focus on developing and owning specific business functions. This granular focus improves the overall business communication and efficiency. There are tradeoffs for the benefits of Microservices. It is important that service boundaries are clearly defined before migrating to a microservice architecture. The microservice architecture is still fairly young but it's a promising way of developing applications and it's definitely worth looking into. Just remember that it might not (yet) be a good fit for your team.

SHARE THIS ARTICLE

STEN PITTET

I've been in the software business for 10 years now in various roles from development to product management. After spending the last 5 years in Atlassian working on Developer Tools I now write about building software. Outside of work I'm sharpening my fathering skills with a wonderful toddler.

TUTORIAL

Tips for scripting tasks with Bitbucket Pipelines

Learn our five tips for automating and scripting manual tasks with Bitbucket Pipelines.

Try this tutorial →

ARTICLE

Continuous Delivery | Get started with CI/CD | Atlassian

Continuous Delivery (CD) is the practice of using automation to release software in short iterations. Learn how to get started with CD with our free guide.

Read this article →

CI/CD Topics

Continuous Delivery
Continuous Integration
Continuous Deployment
Pipelines
Software Testing
Microservices
Tutorials

Sign up for more CI/CD articles and tutorials.

Email

email@example.com

Subscribe