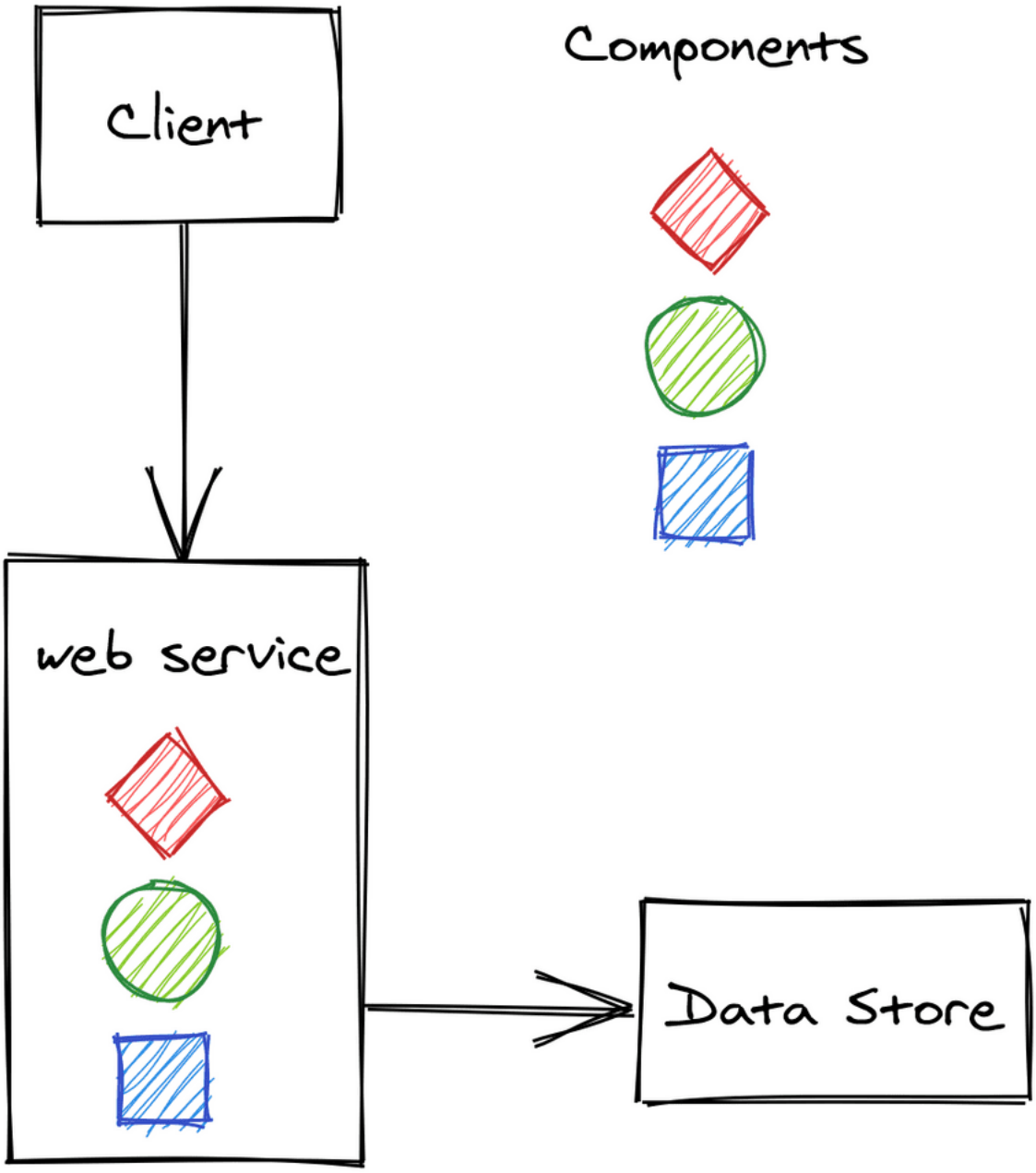
The costs of microservices

November 22, 2020

An application typically starts its life as a monolith. Take a modern backend of a single-page Javascript application, for example - it starts out as a single stateless web service that exposes a RESTful HTTP API and uses a relational database as a backing store. The service is composed of a number of components, or libraries, that implement different business capabilities:

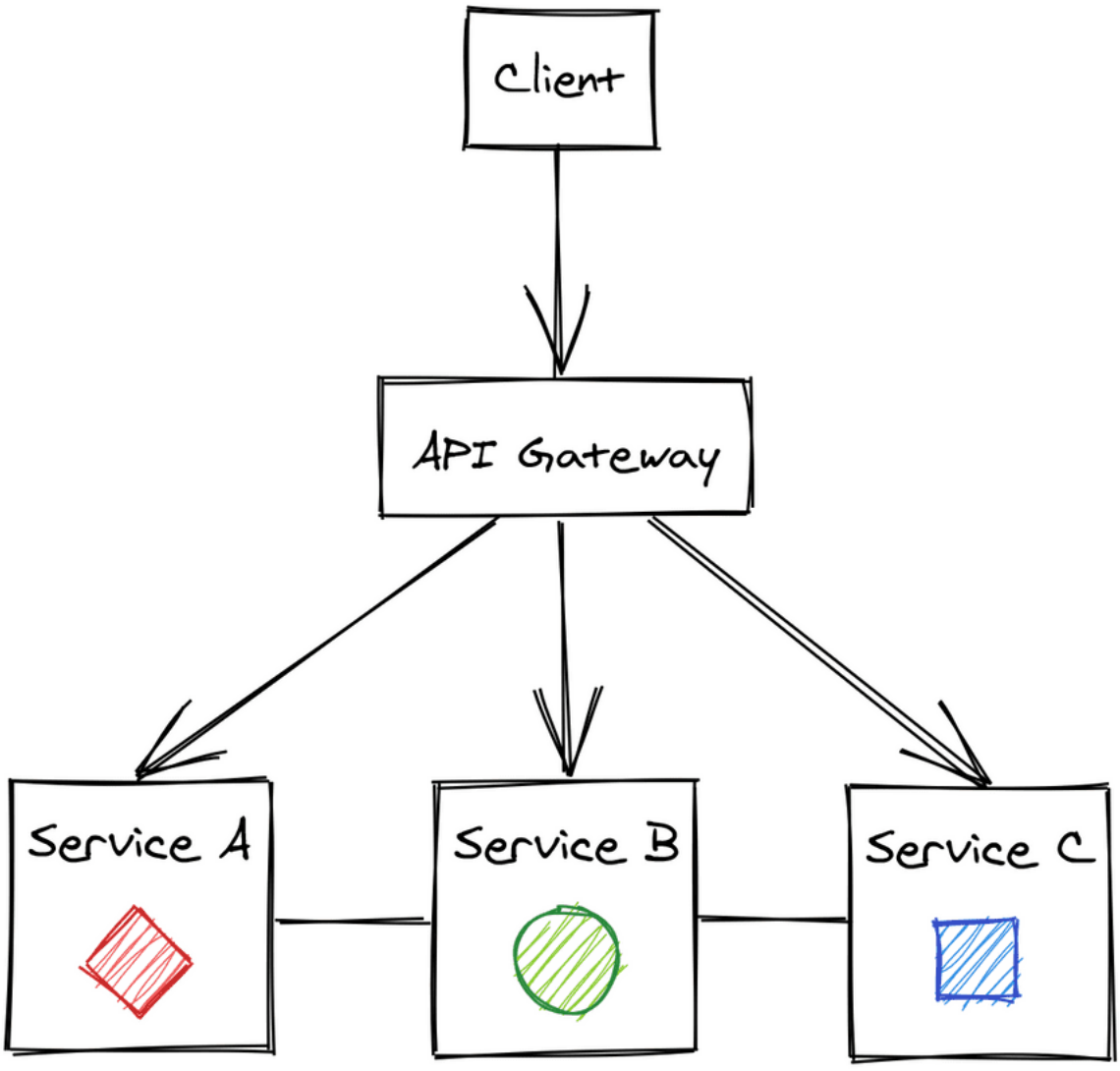


As the number of feature teams contributing to the same codebase increases, its components become increasingly coupled over time. This leads the teams to step on each other's toes more and more frequently, decreasing their productivity.

The codebase becomes complex enough that nobody fully understands every part of it, and implementing new features or fixing bugs becomes time-consuming. Even if the backend is componentized into different libraries owned by different teams, a change to a library requires the service to be redeployed. And if a change introduces a bug - like a memory leak - the entire service can potentially be affected by it.

Additionally, rolling back a faulty build affects the velocity of all teams, not just the one that introduced the bug.

One way to mitigate the growing pains of a *monolithic* backend is to split it into a set of independently deployable services that communicate via APIs. The APIs decouple the services from each other by creating boundaries that are hard to violate, unlike the ones between components running in the same process:



This architectural style is also referred to as the microservice architecture. The term *micro* can be misleading, though - there doesn't have to be anything micro about the services. In fact, I would argue that if a service doesn't do much, it just creates more operational toll than benefits. A more appropriate name for this architecture is service-oriented architecture, but unfortunately, that name comes with some old

baggage as well. Perhaps in 10 years, we will call the same concept with yet another name, but for now we will have to stick to microservices.

Breaking down the backend by business capabilities into a set of services with well-defined boundaries allows each service to be developed and operated by a single small team. The reduced team size increases the application's development speed for a variety of reasons:

- Smaller teams are more effective as the communication overhead grows quadratically with the team's size.
- As each team dictates its own release schedule and has complete control over its codebase, less cross-team communication is required, and therefore decisions can be taken in less time.
- The codebase of a service is smaller and easier to digest by its developers, reducing the time it takes to ramp up new hires. Also, a smaller codebase doesn't slow down IDEs, which makes the developers more productive.
- The boundaries between services are much stronger than the boundaries between components in the same process. Because of that, when a developer needs to change a part of the backend, they only need to understand a small part of the whole.
- Each service can be scaled independently and adopt a different technology stack based on its own needs. The consumers of the APIs don't care how the functionality is implemented after all. This makes it easy to experiment and evaluate new technologies without affecting other parts of the system.
- Each microservice can have its own independent data model and data store(s) that best fit its use-cases, allowing developers to change its schema without affecting other services.

Costs

The microservices architecture adds more moving parts to the overall system, and this doesn't come for free. The cost of fully embracing microservices is only worth paying if it can be amortized across dozens of development teams.

Development Experience

Nothing forbids the use of different languages, libraries, and datastores for each microservice - but doing so transforms the application into an unmaintainable mess. For example, it makes it more challenging for a developer to move from one team to another if the software stack is completely different. And think of the sheer number of libraries - one for each language adopted - that need to be supported to provide common functionality that all services need, like logging.

It's only reasonable then that a certain degree of standardization is needed. One way to do that - while still allowing some degree of freedom - is to loosely encourage specific technologies by providing a great development experience for the teams that stick with the recommended portfolio of languages and technologies.

Resource Provisioning

To support a large number of independent services, it should be simple to spin up new servers, data stores, and other commodity resources - you don't want every team to come up with their own way of doing it. And once these resources have been provisioned, they have to be configured. To be able to pull this off, you will need a fair amount of automation.

Communication

Remote calls are expensive and introduce new and fun ways your systems can crumble. You will need defense mechanisms to protect against failures, like timeouts, retries and circuit breakers. You will also have to leverage asynchrony and batching to mitigate the performance hit of communicating across the network. All of which increases the system's complexity. A lot of what I describe in my book about distributed systems is about dealing with this complexity.

That being said, even a monolith doesn't live in isolation as it's being accessed by remote clients, and it's likely to use third-party APIs as well. So eventually, these issues need to be solved there as well, albeit on a smaller scale.

Continuous Integration, Delivery, and Deployment

Continuous integration ensures that code changes are merged into the main branch after an automated build and test processes have run. Once a code change has been merged, it should be automatically published and deployed to a production-like environment, where a battery of integration and end-to-end tests run to ensure that the microservice doesn't break any service that depends on it.

While testing individual microservices is not more challenging than testing a monolith, testing the integration of all the microservices is an order of magnitude harder. Very subtle and unexpected behavior can emerge when individual services interact with each other.

Operations

Unlike with a monolith, it's much more expensive to staff each team responsible for a service with its own operations team. As a result, the team that develops a service is typically also on-call for it. This creates friction between development work and operational toll as the team needs to decide what to prioritize during each sprint.

Debugging systems failures becomes more challenging as well - you can't just load the whole application on your local machine and step through it with a debugger. The system has more ways to fail, as there are more moving parts. This is why good logging and monitoring becomes crucial at all levels.

Eventual Consistency

A side effect of splitting an application into separate services is that the data model no longer resides in a single data store. Atomically updating records stored in different data stores, and guaranteeing strong consistency, is slow, expensive, and hard to get

right. Hence, this type of architecture usually requires embracing eventual consistency.

Practical Considerations

Splitting an application into services adds a lot of complexity to the overall system. Because of that, it's generally best to start with a monolith and split it up only when there is a good reason to do so.

Getting the boundaries right between the services is challenging - it's much easier to move them around within a monolith until you find a sweet spot. Once the monolith is well matured and growing pains start to rise, then you can start to peel off one microservice at a time from it.

You should only start with a microservice first approach if you already have experience with it, and you either have built out a platform for it or have accounted for the time it will take you to build one.

Written by Roberto Vitillo