Kai Spichale

February 21, 2018 4 minute read

What is wrong with API-first Microservices?

Microservices architecture is a popular pattern for enterprise applications. A very common variation of this pattern is *API-first Microservices*, which is in very simple terms; API-first, implementation-second, and UI optional. Nothing is wrong with API-first Microservices per se. Depending on the concrete customer requirements this architectural pattern could be advantageous but software architecture is always a tradeoff and you should consider important alternatives, too.

Let me first clarify what is meant by API-first development and Microservices architecture, so that we all know that we are talking about the same things. Then I try to explain the limitations of API-first Microservices.

Let's start with API-first development, which is, on the one hand, a design technique, but on the other hand a business strategy.

API-first development as a design technique

When you design an API, you have to understand your clients' perspective. A good API is easy to use, hard to misuse, and sufficiently powerful to satisfy your clients' requirements. A good API has many more

characteristics, but let's stop here. If you start with the API design, you effectively make sure that the API does not depend on its implementation but rather focuses on a great developer experience.

There are other secondary advantages: If the new API is for example implemented by a backend developer and used by another frontend developer, then both could work in parallel as soon as both agreed on the API as their integration contract. An SAP UI5 based application could, for instance, be built on top of an OData API.

API-first development as a business strategy

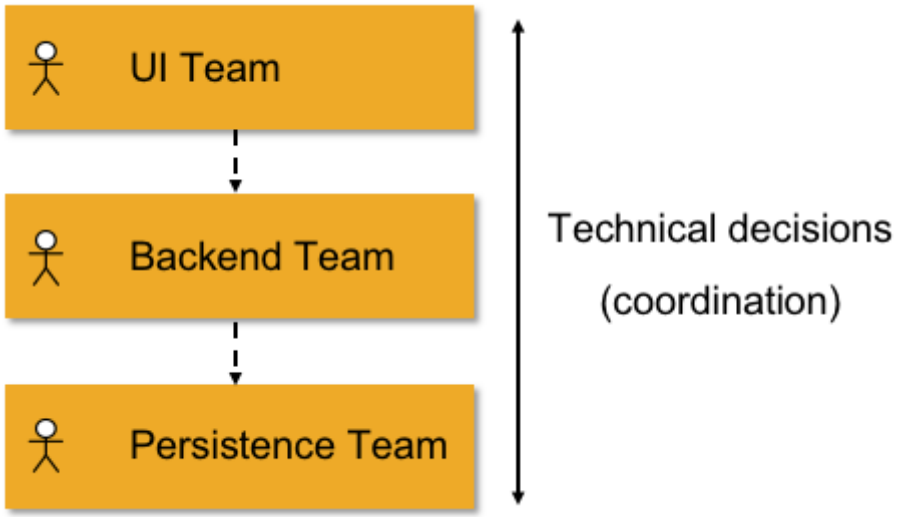
A lot of companies or teams build mobile or web applications and the API is added later for integration purposes. The idea behind the API-first strategy is to build the API first and then optionally the mobile or the web applications on top of that API. These companies see the API as the main product. If the API is well-designed and easy to use, then developers from partners or customers will come and build great applications on top of it. Even SDKs could be provided to support the adoption of the APIs.

Microservices architecture

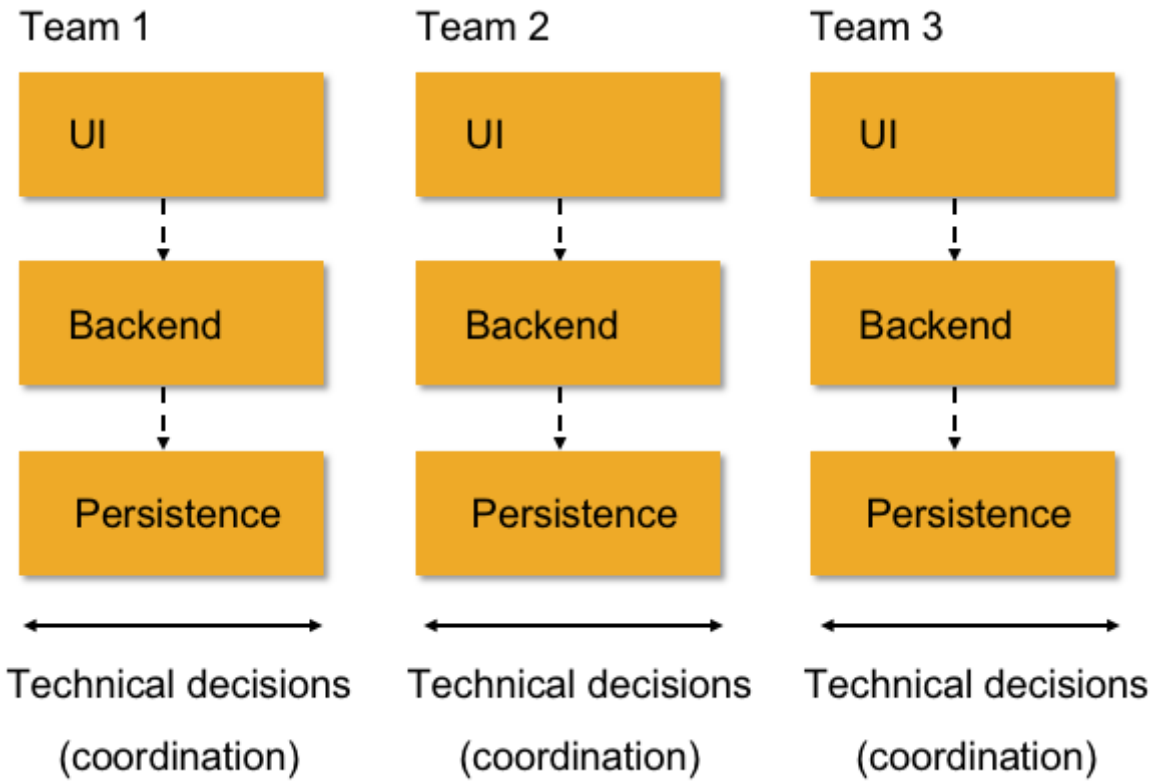
After clarifying API-first development, let's briefly reflect on the basic idea behind Microservices.

Microservices represent a special modularization concept that allows you to scale the development of big applications with multiple development teams. Instead of building a big monolith, the software is assembled from smaller applications that can be deployed independently.

Conway's law tells us that the structures of organizations have a strong impact on the systems they create. If teams are for instance organized by technical skills, then dependencies and the need for communication can slow down the implementation of features that require changes on several technical layers.



Alternatively, we can use Conway's law to our advantage and organize teams with diverse technical skills. Instead of letting the architecture be the result of the team structure, the architecture drives the team structure. Remember: First and foremost, Microservices try to solve an organizational problem!



Self-Contained Systems

Self-Contained Systems (SCS) is a special type of Microservices. Each SCS is an autonomous web application with its own web interface, business logic, and persistence. It is owned by one team. The features of an SCS are usable to end-users via the SCS' own UI. The integration to other systems is asynchronous wherever possible and service APIs are optional.

From my personal experience, I know that not every Microservice can be an SCS. But every SCS can help minimize the number of dependencies between the Microservices. Otherwise, when a system gets bigger, the number of dependencies between the Microservices grows rapidly until a distributed *big ball of mud* emerges.

SCS have their own user interfaces to minimize dependencies. Development teams can use SCS to independently develop and deploy new features.

Make sure you get rid of monolithic frontends. With a monolithic frontend, you never get the flexibility to scale across teams as promised by Microservices.

When would I recommend using API-first Microservices?

A good example is a published API, a.k.a. open API. These APIs are used outside the organizations that created them, and sometimes the clients are even unknown. Usually, API gateways are used to create, publish, maintain, monitor, and secure the published APIs. The gateway can also help monetizing the published APIs. Throttling is another important feature to prevent the APIs from being overwhelmed by too many requests.

In contrast, closed APIs are used inside the organization by the Microservices of the system or by other connected systems. A gateway would probably be overkill here but API-first development can, of course, help to improve the design of the used APIs.

Conclusion

A Microservices architecture is a modularization approach that primarily tries to solve an organizational problem. It has many advantages and disadvantages and depending on your concrete requirements it might or might not be the optimal approach for you. API-first development is a powerful technique or strategy to design and build APIs with a great developer experience. API-first development is an independent concept. You can use API-first development for Microservices and other architectures.

API-first Microservices are a good fit for published APIs but do not forget Self-Contained Systems that help to avoid monolithic UIs and minimize dependencies among the Microservices.

Assigned tags

SAP Cloud Platform | SAP API Management | microservices | self-contained systems |

Related Blog Posts

We are sorry but we are currently unable to retrieve related content.

Related Questions

We are sorry but we are currently unable to retrieve related content.