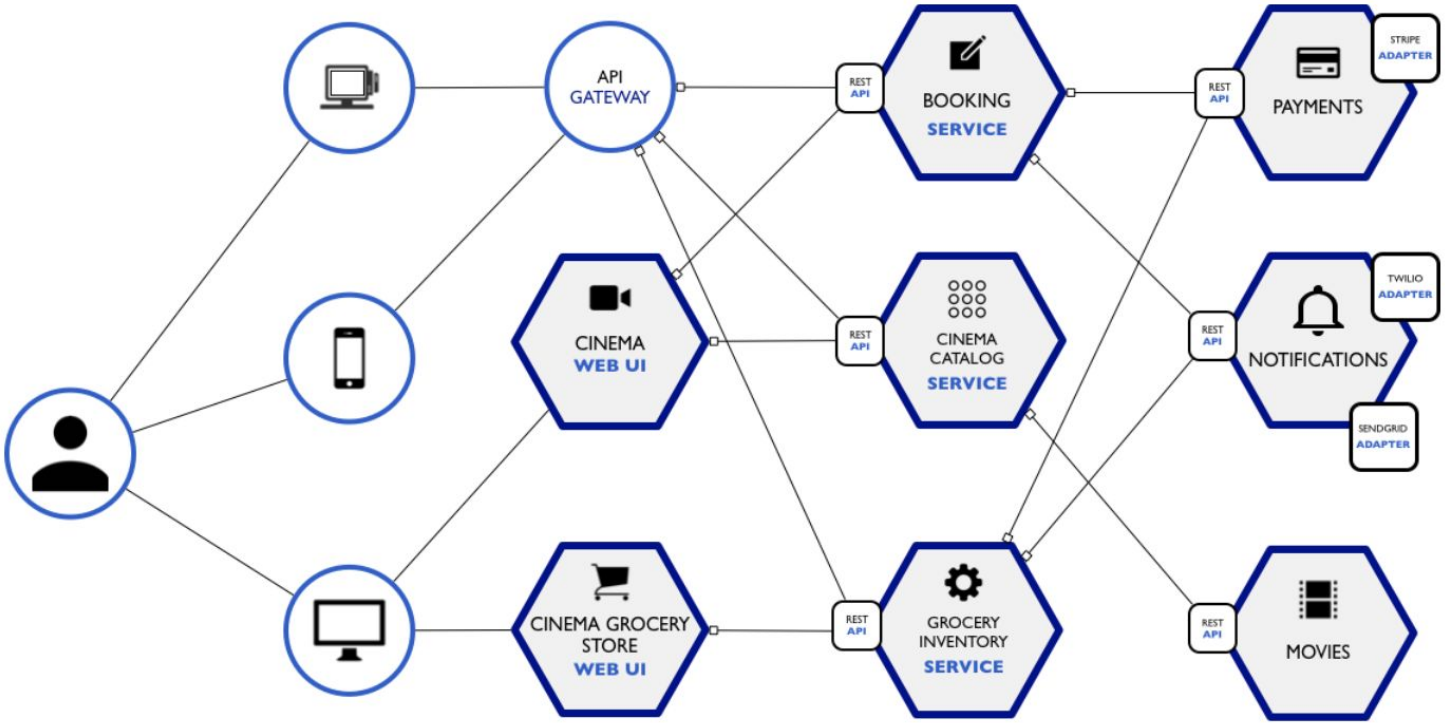
Microservices

Introduction

Microservices



Cinema Microservice Example

A microservice is a single self-contained unit which, together with many others, makes up a large application. By splitting your app into small units every part of it is independently deployable and scalable, can be written by different teams and in different programming languages and can be tested individually.

— Max Stoiber

A microservice architecture means that your app is made up of lots of smaller, independent applications capable of running in their own memory space and scaling independently from each other across potentially many separate machines.

— Eric Elliot

What is a Microservices Architecture?

Adrian Cockroft, an engineer who oversaw Netflix transition from a monolithic to a microservices architecture defines it as a service-oriented architecture composed of loosely coupled elements that have bounded contexts.

Loosely coupled means that you can update the services independently. Updating one service doesn't require changing any other services. If you have a bunch of small, specialized services but still have to update them together, they're not microservices because they're not loosely coupled. One kind of coupling that people tend to overlook as they transition to a microservices architecture is database coupling, where all services talk to the same database and updating a service means changing the schema. You need to split the database up and denormalize it.

Discussion:

What are the Benefits of Microservices?

10 min

- The application starts faster
- Deployments are faster
- Each service can be deployed independently of other services — easier to deploy new versions of services frequently
- Easier to scale development and can also have performance advantages.
- Eliminates any long-term commitment to a technology stack. When developing a new service you can pick a new technology stack.
- Microservices are typically better organized, since each microservice has a very specific job, and is not concerned with the jobs of other components.
- Decoupled services are also easier to recompose and reconfigure to serve the purposes of different apps (for example, serving both the web clients and public API).

Discussion:

What are the Drawbacks of Microservices?

10 min

- Developers must deal with the additional complexity of creating a distributed system.
- Deployment complexity. In production, there is also the operational complexity of deploying and managing a system comprised of many different service types.
- As you're building a new microservice architecture, you're likely to discover lots of cross-cutting concerns that you did not anticipate at design time.

Microservices

Best Practices for Designing

Create a Separate Datastore for Each Microservice

Do not use the same backend data store across microservices. You want the team for each microservice to choose the database that best suits the service. Moreover, with a single data store it's too easy for microservices written by different teams to share database structures, perhaps in the name of reducing duplication of work. You end up with the situation where if one team updates a database structure, other services that also use that structure have to be changed too.

Create a Separate Datastore for Each Microservice

Breaking apart the data can make data management more complicated, because the separate storage systems can more easily get out sync or become inconsistent, and foreign keys can change unexpectedly. You need to add a tool that performs master data management (MDM) by operating in the background to find and fix inconsistencies. For example, it might examine every database that stores subscriber IDs, to verify that the same IDs exist in all of them (there aren't missing or extra IDs in any one database). You can write your own tool or buy one. Many commercial relational database management systems (RDBMSs) do these kinds of checks, but they usually impose too many requirements for coupling, and so don't scale.

Keep Code at a Similar Level of Maturity

Keep all code in a microservice at a similar level of maturity and stability. In other words, if you need to add or rewrite some of the code in a deployed microservice that's working well, the best approach is usually to create a new microservice for the new or changed code, leaving the existing microservice in place. This is sometimes referred to as the immutable infrastructure principle.

This way you can iteratively deploy and test the new code until it is bug free and maximally efficient, without risking failure or performance degradation in the existing microservice. Once the new microservice is as stable as the original, you can merge them back together if they really perform a single function together, or if there are other efficiencies from combining them.

Do a Separate Build for Each Microservice

Do a separate build for each microservice, so that it can pull in component files from the repository at the revision levels appropriate to it. This sometimes leads to the situation where various microservices pull in a similar set of files, but at different revision levels. That can make it more difficult to clean up your codebase by decommissioning old file versions (because you have to verify more carefully that a revision is no longer being used), but that's an acceptable trade-off for how easy it is to add new files as you build new microservices. The asymmetry is intentional: you want introducing a new microservice, file, or function to be easy, not dangerous.

Deploy in Containers

Deploying microservices in containers is important because it means you just need just one tool to deploy everything. As long as the microservice is in a container, the tool knows how to deploy it. It doesn't matter what the container is. That said, Docker seems very quickly to have become the de facto standard for containers.

Treat Servers as Stateless

Treat servers, particularly those that run customer-facing code, as interchangeable members of a group. They all perform the same functions, so you don’t need to be concerned about them individually. Your only concern is that there are enough of them to produce the amount of work you need, and you can use autoscaling to adjust the numbers up and down. If one stops working, it’s automatically replaced by another one. Avoid “snowflake” systems in which you depend on individual servers to perform specialized functions.

Features of Microservices

- A microservices architecture exists as a cluster of loosely-connected, pluggable, and autonomous components.
- Each microservice carries out a well-defined function – or satisfies a specific business need – for the larger application or IT infrastructure.
- Each one runs autonomously without dependency or knowledge of the other microservices or the larger application.
- Because they run independently and are only loosely connected to the architecture, developers can add, remove, or upgrade them more quickly and easily without affecting the larger ecosystem.

Features of Microservices

- The microservices that comprise an application can be written in different programming languages and run on different platforms. Microservices could be running in the cloud, on-premises, or a mix of both.
- Using an advanced iPaaS tool like DreamFactory, enterprises can add them to any IT infrastructure.
- Cloud-based SaaS microservices can update automatically after rollout so businesses immediately benefit from the latest versions and security updates.

Discussion:

Microservices: what do you think?

15 min

In groups, discuss and answer the following questions:

- Are they “aware” of the other microservices?
- How do they connect and communicate?
- How do APIs work with these?
- Are REST API connections stable for microservices?
- Where do you run them?
- How do you manage and control resources for different ones?

Discussion: Microservices

Q - Are they “aware” of the other microservices?

Microservices don’t generally know about or acknowledge the existence of the other microservices that make up the application.

Q - How do they connect and communicate?

They integrate with other microservices and applications through language and platform-agnostic APIs (Application Programming Interfaces).

Q - How do APIs work with these?

You can expose APIs for microservices as REST endpoints, WebHooks, or invoke them via lightweight messaging protocols like RabbitMQ. The most common approach involves an HTTP/REST API with JSON.

Discussion: Microservices

Q - Are REST API connections stable for microservices?

The relative decoupling of microservices data connections reduces the chances of a single service causing a failure cascade that interferes with the operation of the entire system. Loose connections help avoid synchronous and blocking-calls from disrupting system performance.

Q - Where do you run them?

Most developers agree that containerized environments are ideal for running microservices. While it's possible to run microservices on a virtual machine, containers serve as a lighter-weight, faster, and more cost-effective way to virtualize a runtime environment for microservices.

Q - How do you manage and control resources for different ones?

When running a container-based microservices architecture, developers use container orchestration tools to manage and control the distribution of server resources to different containers based on the needs of each of them.

Wrap-ups