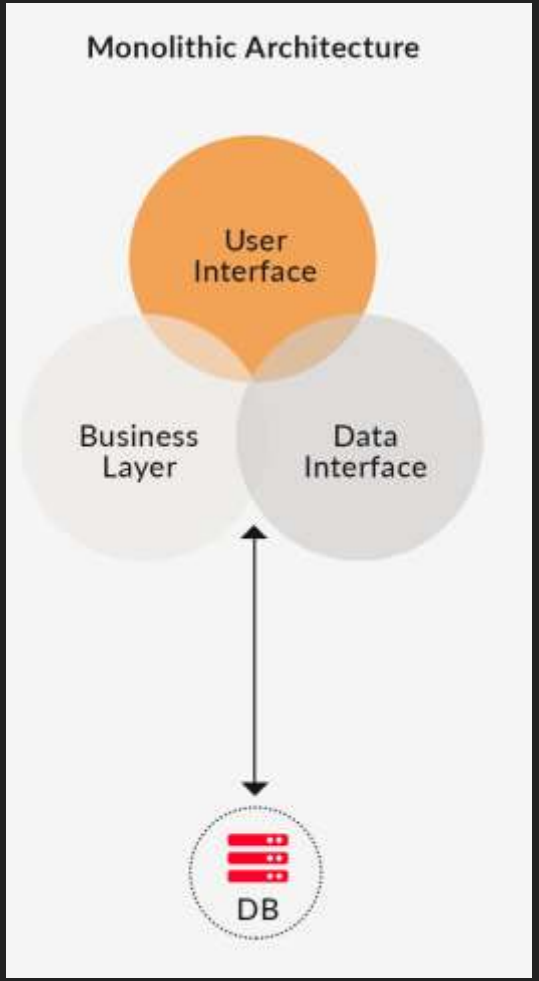
Microservices Architecture

Background

In many systems design questions, you will have to design a good amount of functionality for some hypothetical use case. Typically, it makes sense to split each piece of function into separate microservices, which act as self contained units of servers that can interact with other microservices as if they were external services.

Original Approach: Monoliths

All code contained in the same repository, same set of nodes in a cluster responsible for handling any possible application request.



Benefits of a Monolith Approach

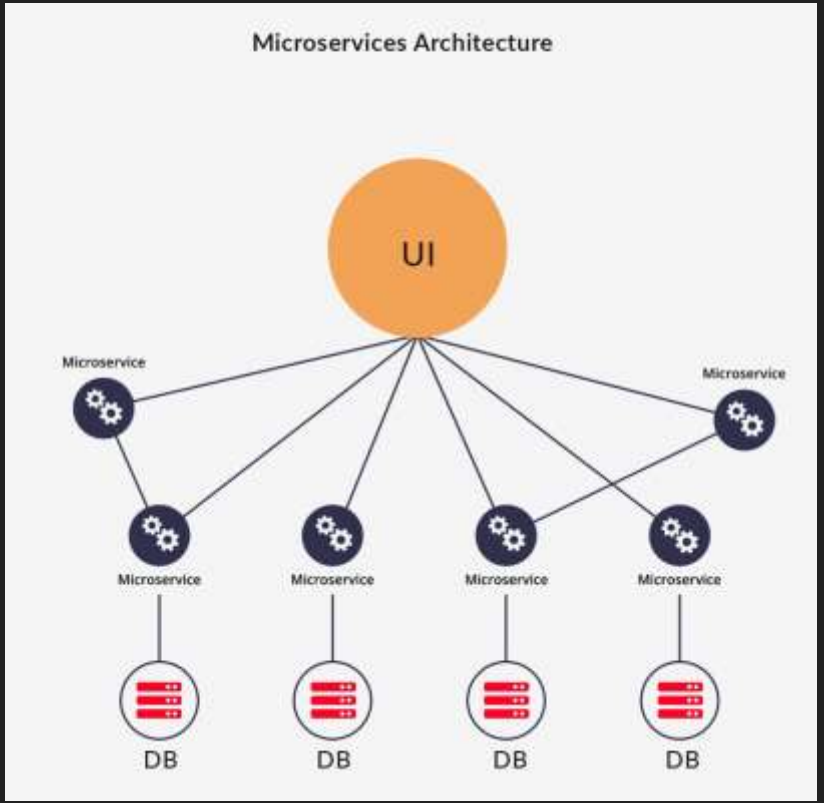
- Easy to implement
  - Do not need to think about how to split up functionality
  - All code goes in the same repository
- Easy to test
  - As opposed to using blackbox testing with other microservices, can transparently make sure that everything works properly via unit tests in one repository
- Easy to deploy and scale
  - Only one repository to deploy
  - If we need more compute we simply redeploy the same repository on other nodes

Cons of a Monolith Approach

- Large application codebase makes it harder to onboard new developers
- Deployments are huge and likely take a while
- Harder for multiple teams to work on at the same time
  - Too many code changes to merge in
  - One team may introduce bugs that block another team working on unrelated things, or worse yet bring down the entire production service
  - Every small change requires a complete redeploy (which are slow)
- Pieces of the service that differ in popularity cannot be scaled independently
- Wanting to use a new technology or framework requires rewriting the entire monolith, as opposed to just a small piece of it

Microservice Architectures

Teams each manage a few self-contained services that can be deployed and scaled independently, using a tech stack of their choice. This is far more scalable and solves all of the major issues mentioned regarding monoliths in the previous slide, at the cost of adding some complexity to your application layout.



Quick Aside: Docker

Very relevant for microservice architectures:

- Applications are packaged into “containers”
  - Each container holds all of the dependencies of an application, as well as their versions, so that the environment that the application will run on is the same whether the code is running locally on a laptop, in a testing environment, or in a production environment on the cloud
- When deploying to a cloud, you typically preconfigure the capacity of the virtual machine that you want
  - If we wanted a microservice running on each virtual machine, we would have to figure out the capacity that each service needed in advanced, likely would lead to much wasted compute
  - Docker allows running multiple containers on just one virtual machine (acts like a lightweight operating system), giving each container the ability to take the resources that they need in a given moment, leads to far less waste of compute power

Conclusion

While it generally makes sense for most new applications with a smaller team to start out under a monolith design to develop quickly and reduce added complexity, using a microservice architecture at scale is virtually a necessity. It allows teams to develop quickly on their own without a need for a unified codebase, and provides them with the ability to deploy and scale their service as they see fit. Finally, a bug introduced in the code by one team will not affect the services of another, and failures will be siloed.