What are microservices?

In this lesson we will understand what microservices are, how to decompose our previous monolith into microservices, the pros and cons of using microservices and when is the right time to start thinking on using them.

What are microservices?

There are multiple definitions of microservices out there but here's one I have synthetized from a few sources and that I think covers the fundamental characteristics of microservices:

Microservices refers to **An architectural style that structures an application as a collection of independently deployable services that are modeled around a business domain and are usually owned by a small team.**

So, instead of having a single application deployed as a single monolithic unit we break it down into smaller services where each of them is closely aligned to some part of our business domain. Each of these services can be deployed independently, which by the way includes granting each of them their own database, which is one of the main challenges of moving to microservices. Also, there's usually only one small and fairly autonomous team behind each microservice.

Let's see how this looks like for the game backend example we talked about in the previous lesson.

Decomposing the monolith

We start again with our client apps which will still interact with some backend pieces to enable the required game experience. And, just like before, we come up with most of the modules we talked about before to address the multiple scenarios.

However, this time each of these modules live in their own individual server process (or likely a web server process), fully isolated from the other ones and with their own lifecycle. Not only that, but also each of these services have their own associated database that has no relationship with the ones from the other services. Also notice that we decided to group the Users and Authz modules in a single process that we are now calling Identity service.

But, even when the services are now independent, they still have ways to collaborate with each other so that, for instance, when a purchase starts in Trading, this service will communicate somehow with the Identity and Inventory services in order to achieve the exchange of gil for catalog items. This here is what we call the microservices architecture pattern.

Next, let's learn about the benefits and drawbacks involved in a microservices architecture.

Microservices pros and cons

These are some of the pros and cons of using microservices. Let's start with the Pros.

PROS

- **They have a small, easier to understand code base.** Since the microservices are now independent, the source code for each of them is usually stored in its own git repository. This implicitly results in a much smaller repository which is much easier to understand, especially for devs new to the code base
- **They are quicker to build.** The smaller, independent code base per microservice allows for building each service much faster and in parallel, both on dev boxes and in CI pipelines.
- **They allow for independent, faster deployments and rollbacks.** Each service can now be deployed individually as soon as the relevant source code has been built and testing for the domain covered by the service can start right away and is much smaller in scope. Also, if an issue is detected with the deployment it can be rolled back to the previous version without having to roll back any other service.
- **They are independently scalable.** Different services can be scaled according to their needs. So if there is one service that needs to scale given the amount of ram or CPU it uses, it can be scaled independently of the needs of any of the other services.
- **There is much better isolation from failures.** Microservices are isolated from each other so a crash in one service has no impact in the availability of other services.
- **They are designed for continuous delivery.** Since microservices are small, usually start fast and are independent from each other, they are great candidates for continuous delivery. They can be deployed as soon as the code has been built and this can be done safely, in an automated way, multiple times a day.
- **It is easier to adopt new, varied tech.** Different services don't need to use the same versions of their multiple dependencies, including the .NET runtime. Even better, if one of them needs to be moved to a completely different language or web framework, this is much more straightforward to do with a single small microservice than with a big monolith.
- **Microservices grant autonomy to teams and lets them work in parallel.** The natural independence of deployment and isolation of microservices and the fact that they are closely aligned to a specific part of the business domain, results in the owning teams to have great autonomy around the entire lifecycle of the service. Some teams might decide to deploy their microservice every night, while others will do it every hour or whenever an urgent fix needs to get shipped. Different teams will do what's best for them.

Now, the cons:

CONS

- **It is not easy to find the right set of services.** How do you know you have split the app into the right set of services? Are 3 microservices enough, or 10 perhaps? How much granularity? These questions are not easy to answer but techniques like domain driven design and aligning the services to specific pieces of the domain and the teams behind them, helps to find the right balance.

- **Using microservices adds the complexity of distributed systems.** The fact that each service lives in its separate silo introduces a lot of complex issues that are nonexistent in monoliths. How to have a service communicate with another one if a module can no longer make a simple method call to the other? Issues like this are the reason for this course.
- **The shared code moves to separate libraries.** Any piece of code that needs to be shared across two or more microservices now needs to be either duplicated in each service (which is not ideal) or moved to a separate library and likely shared via nuget packages. This is technically good since it forces the owners of the shared code to have a good way to test the shared code independently. But on the flip side things are no longer as easy as adding a new shared class or method in any of the services code base. Also, you may end up with dozens of shared libraries, that may need to collaborate and have the right set of dependencies.
- **There is no good tooling for distributed apps.** IDEs are usually focused on a single app, not on running several services together. This means that devs will usually work on one service at a time and will need to figure out ways for their code to interact with other services in their dev environment where needed.
- **Releasing features across services is hard.** Simple things like adding a new attribute to, say, a user record and have it been used across the entire system can involve touching multiple services in a very coordinated fashion. Even worse, updating or removing attributes from data classes is extremely painful and usually avoided as much as possible.
- **It is hard to troubleshoot issues across services.** When a customer reports an issue with the system or application, where should we start looking? With features potentially distributed across dozens of microservices you are forced to have a very good story around tracing issues across services to be able to quickly find out the root cause and address it.
- **You can't use transactions across services.** Since each service has its own database, atomic transactions that impact multiple tables are usually out of the question. How to ensure a purchase operation will both debit gil from the user and grant it the relevant items when one of the involved services could unexpectedly fail in the middle of the purchase process? A few new patterns and techniques need to be introduced to properly handle these concerns.
- **Finally, it raises the required skillset for the team.** With each team fully owning their service, and since not all services will be built similarly, each team now needs to learn not just how to code features for the service but also how to build it, deploy it, test it in isolation, diagnose it, roll it back, etc. Many times this also involves building a ton of automation and shared infrastructure to ensure teams can remain agile building features and not have to worry too much about all the other aspects. This requires a lot of investment.

Let's close this lesson by understanding when you should start considering microservices.

When to use microservices?

You should keep in mind that it is perfectly fine to start with a monolith and only later move to a microservices architecture. Your business domain and the set of requirements for your system might not be fully flushed out by the time you need to start designing and building the system. Therefore, jumping into building multiple microservices from the start could not only significantly slow down the small starter team at a time when quick proofs of concept are desired, but also you may end up building the wrong set of microservices with the additional cost of having to restructure them later. The heavy investments on shared infrastructure and automation will only make sense as the boundaries are clearly defined, and the team has grown in size.

You should start looking at microservices when:

- The code base size is more than what a small team can maintain
- The different teams can't move fast anymore due to the amount of shared code
- Builds have become too slow due to the large code base
- The time to market is compromised due to infrequent deployments and long verification times

In the end, microservices are all about team autonomy. Any time you notice multiple teams no longer being able to deliver features in an agile way due to the drawbacks of the monolithic architecture, you know that your teams have lost their autonomy and they could potentially greatly benefit from using microservices instead.

In the next module we will get our hands dirty by building our first microservice.