When developing a server-side application you can start it with a modular hexagonal or layered architecture which consists of different types of components:

- Presentation — responsible for handling HTTP requests and responding with either HTML or JSON/XML (for web services APIs).
- Business logic — the application's business logic.
- Database access — data access objects responsible for access the database.
- Application integration — integration with other services (e.g. via messaging or REST API).

Despite having a logically modular architecture, the application is packaged and deployed as a monolith

WHAT IS A MONOLITH?

A monolithic application is built as a single, unified unit. Often a monolith consists of three parts: a database , a client-side user interface (consisting of HTML pages and/or JavaScript running in a browser), and a server-side application. The server-side application will handle HTTP requests, execute domain-specific logic, retrieve and update data from the database, and populate the HTML views to be sent to the browser.

Another characteristic of a monolith is that it's often one massive code base. Server side application logic, front end client side logic, background jobs, etc, are all defined in the same code base. This means if developers want to make any changes or updates, they need to build and deploy the entire stack all at once.

Contrary to what you might think, a monolith isn't a dated architecture that we need to leave in the past. In certain circumstances, a monolith is ideal. I spoke to Steven Czerwinski, Head of Engineering at Scaylr and former Google employee, to better understand this.

"Even though we had had these positive experiences of using microservices at Google, we [at Scaylr] went [for a monolith] route because having one monolithic server means less work for us as two engineers," he explained. This was back in the early beginnings of Scalyr.

In other words, because his team was small, a unified application was more manageable in comparison to splitting everything up into microservices.

Monolith Pros:

- **Fewer Cross-cutting Concerns:** A major advantage associated with monolithic architecture is that you only need to worry about cross-cutting concerns, such as logging or caching, for one application.
- **Less Operational Overhead:** Focusing your finances on one application means that there's only one application that you need to set up logging, monitoring and testing for. A Monolith is also generally less complex to deploy since you aren't organizing multiple deployments.
- **Easier Testing:** With a monolith, automated tests are easier to setup and run because, once again, everything is under the same roof. With microservices, tests will need to accommodate for different applications on different runtime environments — which can get complex.
- **Performance:** A monolith can also boast performance advantages in comparison to microservices. That's often down to a monolith using local calls instead of an API call across a network.

- Simple to develop.
- Simple to test. For example you can implement end-to-end testing by simply launching the application and testing the UI with Selenium.
- Simple to deploy. You just have to copy the packaged application to a server.
- Simple to scale horizontally by running multiple copies behind a load balancer.
-

Monolith Cons:

- **Overly-tight Coupling:** While monoliths can help you avoid entanglement as previously mentioned, a monolith becomes more vulnerable to entanglement the larger it grows. Because everything is so tightly coupled, isolation

of services within the monolith becomes arduous, making life difficult when it comes to independent scaling or code maintenance.

- **Harder To Understand:** It's common to find that monoliths are more difficult beasts to understand in comparison to microservices, a problem which rears its head when on-boarding new team members. This is sometimes a direct result of the tight coupling, as well as the fact that there may be dependencies and side-effects which are not obvious when you're looking at a particular service or controller.

- This simple approach has a limitation in size and complexity.
- Application is too large and complex to fully understand and made changes fast and correctly.
- The size of the application can slow down the start-up time.
- You must redeploy the entire application on each update.
- Impact of a change is usually not very well understood which leads to do extensive manual testing.
- Continuous deployment is difficult.
- Monolithic applications can also be difficult to scale when different modules have conflicting resource requirements.
- Another problem with monolithic applications is reliability. Bug in any module (e.g. memory leak) can potentially bring down the entire process. Moreover, since all instances of the application are identical, that bug will impact the availability of the entire application.
- Monolithic applications has a barrier to adopting new technologies. Since changes in frameworks or languages will affect an entire application it is extremely expensive in both time and cost.
-

WHAT ARE MICROSERVICES?

The problem is that that there is nothing inherently "micro" about microservices per se. While they tend to be smaller than the average monolith, they do not have to be tiny. Some are, but size is relative and there's no standard of unit of measure across organizations.

At this point, it's worth mentioning that — as you might have gathered from the slightly varying definitions given above — there is no industry consensus of what exactly microservices are. Nevertheless, here's my take on the definition of microservices:

Microservice architecture refers to the concept of developing a single application as a suite of small services, in contrast to developing them as one, large 'monolith'. Each microservice is a small application that has its own hexagonal architecture consisting of business logic along with various adapters. Some microservices would expose a REST, RPC or message-based API and most services consume APIs provided by other services. Other microservices might implement a web UI.

Each of those broken-up, individualized services run on their own process, communicating with lightweight mechanisms, often an HTTP resource API. Fully-fledged microservices are independently deployable, and yet can work in tandem when necessary.

Microservices Pros :

- **Better Organization:** Microservice architectures are typically better organized, since each microservice has a very specific job, and is not concerned with the jobs of other components. It tackles the problem of complexity by decomposing application into a set of manageable services which are much faster to develop, and much easier to understand and maintain.

- **Decoupled:** 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). They also allow for fast, independent delivery of individual parts within a larger, integrated system.

- **Performance:** Under the right circumstances, microservices can also have performance advantages depending on how they're organized because it's possible to isolate hot services and scale them independent of the rest of the app. It enables each service to be developed independently by a team that is focused on that service.
- **Fewer Mistakes:** Microservices enable parallel development by establishing a hard-to-cross boundary between different parts of your system. By doing this, you make it hard – or at least harder – to do the wrong thing: Namely, connecting parts that shouldn't be connected, and coupling too tightly those that need to be connected.

- It reduces barrier of adopting new technologies since the developers are free to choose whatever technologies make sense for their service and not bounded to the choices made at the start of the project.

- Microservice architecture enables each microservice to be deployed independently. As a result, it makes continuous deployment possible for complex applications.

- Microservice architecture enables each service to be scaled independently.

Microservices Cons -

- **Cross-cutting Concerns Across Each Service:** 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. You'll either need to incur the overhead of separate modules for each cross-cutting concern (i.e. testing), or encapsulate cross-cutting concerns in another service layer that all traffic gets routed through. Eventually, even monolithic architectures tend to route traffic through an outer service layer for cross-cutting concerns, but with a monolithic architecture, it's possible to delay the cost of that work until the project is much more mature.
- **Higher Operational Overhead:** Microservices are frequently deployed on their own virtual machines or containers, causing a proliferation of VM wrangling work. These tasks are frequently automated with container fleet management tools.

- Microservices architecture adding a complexity to the project just by the fact that a microservices application is a distributed system. You need to choose and implement an inter-process communication mechanism based on either messaging or RPC and write code to handle partial failure and take into account other fallacies of distributed computing.

- Microservices has the partitioned database architecture. Business transactions that update multiple business entities in a microservices-based application need to update multiple databases owned by different services. Using distributed transactions is usually not an option and you end up having to use an eventual consistency based approach, which is more challenging for developers.

- Testing a microservices application is also much more complex then in case of monolithic web application. For a similar test for a service you would need to launch that service and any services that it depends upon (or at least configure stubs for those services).

- It is more difficult to implement changes that span multiple services. In a monolithic application you could simply change the corresponding modules, integrate the changes, and deploy them in one go. In a Microservice architecture you need to carefully plan and coordinate the rollout of changes to each of the services.

- Deploying a microservices-based application is also more complex. A monolithic application is simply deployed on a set of identical servers behind a load balancer. In contrast, a microservice application typically consists of a large number of services. Each service will have multiple runtime instances. And each instance need to be configured, deployed, scaled, and monitored. In addition, you will also need to implement a service discovery mechanism. Manual approaches to operations cannot scale to this level of complexity and successful deployment a microservices application requires a high level of automation.

When To Start With A Monolith -

Here are some scenarios that indicate that you should start your next project using monolithic architecture.

- **Your Team Is At Founding Stage:** Your team is small, between 2-5 members, and is thus unable to tackle a broader and high-overhead microservices architecture.

- **You're Building An Unproven Product or Proof of Concept:** Are you building an unproven product in the market? If it's a new idea, it's likely going to pivot and evolve over time, so a monolith is ideal to allow for rapid product iteration. Same applies to a proof of concept where your goal is just to learn as much as possible as quickly as possible, even if you end up throwing it away.
- **You Have No Microservices Experience:** If your team has no prior experience with microservices, unless you can justify taking the risk of learning "on the fly" at such an early stage, it's likely another sign you should stick to a monolith to start.

When To Start With Microservices -

Here are some scenarios that indicate that you should start your next project using microservices:

- **You Require Quick, Independent Service Delivery:** If it's snappy, isolated service delivery that you need, microservices are your best bet. However, depending on the size of your team, it can take some time before you see any service delivery gains versus starting with monolith.
- **A Piece of Your Platform Needs to Be Extremely Efficient:** If your business is doing intensive processing of petabytes of log volume, you'll likely want to build that service out in a very efficient language (i.e. C++) while your user dashboard may be built in Ruby on Rails.
- **You Plan To Scale Your Team:** Starting with microservices gets your team used to developing in separate small teams from the beginning, and having teams separated by service boundaries makes scaling your development organization easier.