# **Monolithic vs. microservices architectures**Monolith vs microservice architecture

Monolith vs microservice architecture

# **⚡ Introduction**

* A monolithic application is a software application compiled into a single executable, usually stored in a single source control repository.
* Microservice architectures include a series of independently developed and deployable services, stored in separate source control repositories or a mono repo.
* Even though they may be seen as “uncool”, monolithic architectures are the best place to start when developing a new system or in a small organization.

# **🚀 Let’s kick-off**

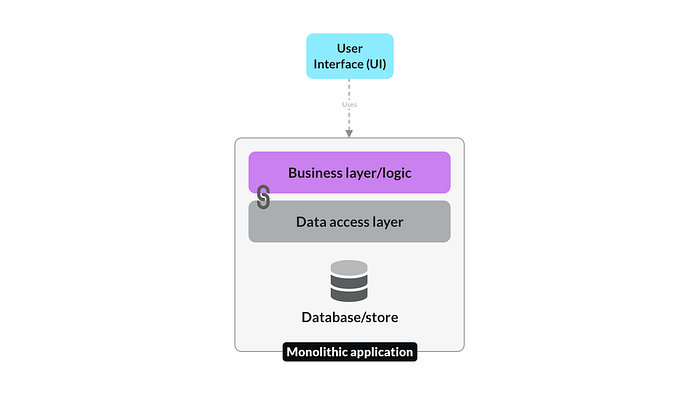
Since microservices architectures entered the scene and become popular in the early 2010s, there’s been an ongoing debate about whether they are better than traditional monolithic architecture patterns. Big tech companies like Netflix, Amazon, Spotify, eBay and Uber have switched to microservices, but does that mean you should too? Let’s explore some of the advantages and disadvantages of monolithic vs. microservice architectures.

# **📦 Monoliths**

A monolithic application is a software application compiled into a single executable, usually stored in a single source control repository. Traditionally, most applications were developed as monoliths, but today it’s seen as a way to quickly get up and running with a simple codebase in the early days of launching a new product. Even though they may be seen as “uncool”, monolithic architectures are the best place to start when developing a new system or in an organization that’s still relatively small.

Monoliths consist of a [3 tier architecture](https://www.ibm.com/topics/three-tier-architecture), which includes the following:

* A client-side user interface consisting of HTML/JS/CSS pages.
* A server-side application that handles HTTP requests from the client and executes queries in the database.
* A database consisting of tables or documents for storing and retrieving data.



Monolithic architecture

# **👍 Benefits of monolithic architectures**

* **A simple codebase that’s easy to work on.** Modular codebases inside monolithic applications are quick to set up and easy for developers to reason about. The whole app is compiled into a single executable, making it easy to access data and functions without worrying about [inter-process communication](https://en.wikipedia.org/wiki/Inter-process_communication#:~:text=In%20computer%20science%2C%20inter%2Dprocess,processes%20to%20manage%20shared%20data).
* **Quick to deploy with fewer security concerns.** Setting up your [continuous integration and delivery](https://en.wikipedia.org/wiki/CI/CD) pipeline is much simpler for a monolithic application, as there is only one executable and one service to deploy. It’s far easier to make security considerations as there are fewer services that could be compromised, with less sensitive data flowing through the network.
* **Lower network latency for requests.** All requests are handled by one process, which reduces latency as fewer calls are happening in the backend. There may only be one request made to the database rather than waiting for other services to complete tasks.

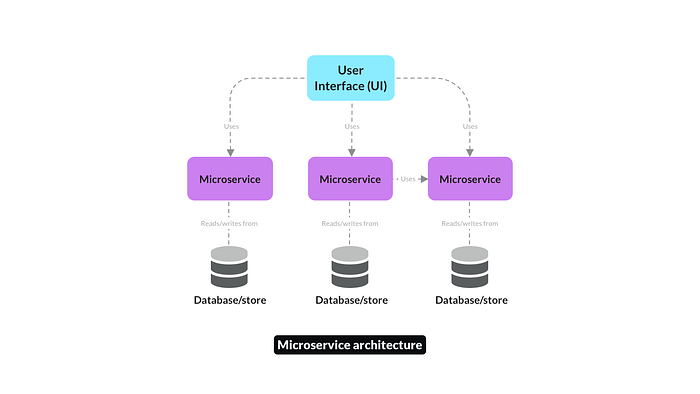
# **👎 Disadvantages of monolithic architectures**

* **They become slow and difficult to manage as the codebase increases.** As the codebase increases, the benefits provided by monolithic architectures reduce. Application boot time increases, and managing the large codebase becomes problematic, requiring refactoring and modularization to keep it organized and maintainable.
* **Scaling horizontally becomes a challenge as the load increases.** Monolithic applications are typically [scaled vertically](https://www.section.io/blog/scaling-horizontally-vs-vertically) by adding CPU and memory to serve more concurrent requests. Scaling monolithic applications horizontally (adding identical instances) is inefficient, making it difficult to scale them beyond a certain threshold.
* **Difficult to adopt and test new technologies.** Switching programming languages or frameworks requires rewriting the entire monolithic application, which may be impossible due to time or funding limitations. This makes it difficult to test new technologies without spending a lot of effort on transitioning to them.

# **🧩 Microservices**

Microservice architectures include a series of independently developed and deployable services. They are [loosely coupled](https://en.wikipedia.org/wiki/Loose_coupling) with separate responsibilities and concerns. Each microservice can have a dedicated source control repository or be part of a larger [mono repo](https://monorepo.tools/). This type of architecture is more appropriate for mature products running at a larger scale or in organizations with many teams and developers.

The biggest consideration for these architectures is how the services interact with each other to fulfill an end-to-end user request. An event bus such as [RabbitMQ](https://www.rabbitmq.com/) or [Kafka](https://kafka.apache.org/) is often used as a communication layer between lots of services. Each microservice usually defines an API specification that describes how it can be interacted with by other services.



Microservice architecture

# **👍 Benefits of microservice architectures**

* **Teams can own services and move at different velocities.** The clean separation of concerns with microservices allows teams to own and work on services independently and asynchronously. This works great in larger organizations where teams may want a faster or slower release cadence depending on the project.
* **Multiple languages and frameworks can be used.** [Polyglot](https://www.dictionary.com/browse/polyglot) architectures allow teams to choose their own programming languages and technologies. This means they can select the best tool for the job rather than being constrained by languages that have been used in the past.
* **Services can handle a large number of requests with horizontal scaling.** Microservices can autoscale horizontally once they have been made [stateless](https://sparkequation.com/2020/11/12/stateless-vs-stateful-microservices-addressing-the-benefits-and-quandaries), allowing them to handle much higher request volume at a lower cost. This also makes them more resilient to errors, as new instances can easily be launched.

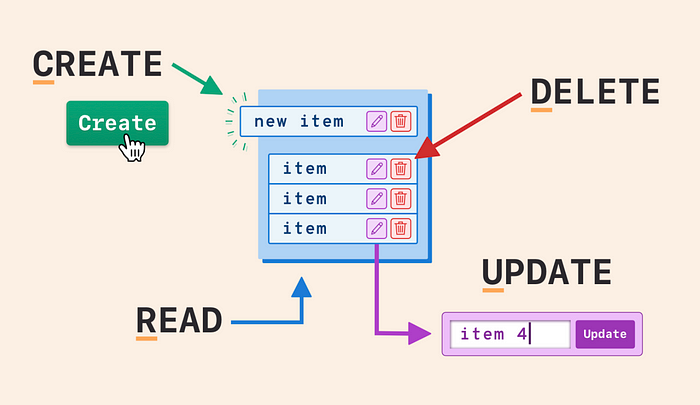
# **👎 Disadvantages of microservice architectures**

* **Making changes across multiple services is complex.** When making a change you must consider the impact it will have on all the upstream and downstream services involved. This makes it much harder to reason about the knock-on effects your change may have on related services.
* **Database transactions across many services are difficult.** Database transactions are commonly used in systems to ensure data integrity when editing multiple pieces of data at once. This becomes challenging with distributed systems, and the solution often involves switching to an [eventually consistent](https://www.scylladb.com/glossary/eventual-consistency) model, which is more complex to implement.
* **Debugging and testing services can be challenging.** Debugging a request from an end-user becomes increasingly complex the more touchpoints it has with various backend services. [Distributed tracing](https://www.datadoghq.com/knowledge-center/distributed-tracing) is required to understand the flow of messages through your microservice architecture.

# **🏁 To wrap up**

As with most things in software, there is no right or wrong answer here. Make sure to weigh the pros, cons and compromises for either approach to make the best decision. Consider the company stage and product stage you’re at and have a plan if you hope to switch from a monolithic to a microservice architecture.

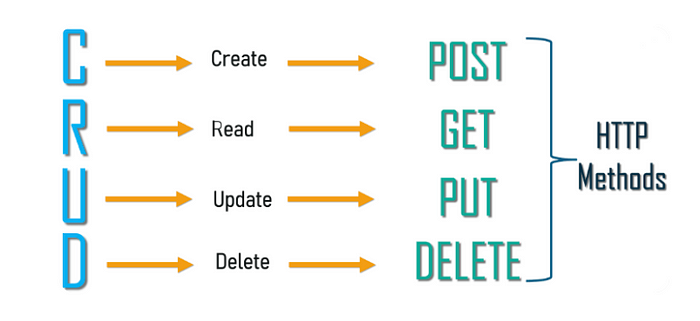
CRUD operations



What is CRUD?

If you are new to programming, seeing “CRUD” in the title may raise concerns considering that the [Merriam-Webster](https://www.merriam-webster.com/dictionary/crud) dictionary defines crud as either “a deposit or incrustation of grease or a slang for a contemptible person.” As fun as it may be to expand on that definition, this is not what we will be expanding on for this article. In the computer programming world, CRUD is an acronym worth knowing. We will review what CRUD operations (aka CRUD functionalities) are and why it is an important fundamental to learn early on.

CRUD is an acronym that stands for Create, Read, Update and Delete.Think of CRUD as a simple concept that represents the four basic functions that models should be able to do and are considered necessary to implement a persistent storage application. In simpler terms, it represents the four basic operations you can do on any data. You can create something new, read or view the newly created data, edit or update the data and finally the option to delete it.



**CRUD Functions**

These four major functions are used to interact with database applications and is a reminder of what data manipulation functions are needed for an application to feel complete. When working with web services, CRUD corresponds to the to HTTP methods, which communications to a web server how you want to interact with a website.

In this breakdown, I will use my [New Beginnings](https://www.youtube.com/watch?v=cmxgzGF6azU) project as an example for the CRUD functions and the HTTP requests associated with them. This was my first (Ruby/HTML) application that was built for a local animal shelter where users can create an account, log in, browse all animals for adoption and schedule playdates or adopt them directly from the site.

Meet Togo! He volunteered to help with our CRUD operations today.

**Create**

After logging into the New Beginnings site, browsing all the animals available for adoption, we’d click on Togo’s name. Once we’ve been directed to his show page, we can create a playdate with him at the shelter. After we complete the appointment form, those inputs are then correlated to the model table in the database. When we submit the data, a POST requested is sent to our API and our playdate with Togo will be stored in the database.

The route for this POST request — /appointments/new

**Read**

Nice! We’ve scheduled our playdate with Togo, but now we want to see that confirmation on our page right? After all, read is the main functionality for us to use the other operations. Now, our API should allow us to see the playdate confirmation on our page. To take a look at all of our appointments, we would use a GET request that allows us to view the scheduled appointment without making any changes to the data stored on our API. This HTTP method is used to only retrieve data and should have no other effects.

The route for this GET request — /appointments.

**Update**

Togo is really adorable, maybe we should see him sooner! For us to reschedule the appointment for an earlier time we can use the corresponding HTTP method for updating your playdate with PUT. This replaces all current data of the target resource (Togo) with the uploaded content (new appointment time/date). The ‘id’ in the route is how the resource is targeted (Togo) to ensure we only update the specified appointment, while leaving any others we may have scheduled untouched.

The route for this PUT request — /appointments/:id.

**Delete**

You know what? Togo has been so helpful with these CRUD functionalities, I think we should just bring him home. Since we can adopt Togo directly from the app, let’s go ahead and cancel the playdate we scheduled. To do this, we can use the HTTP method, DELETE, to remove the targeted appointment from our page. To reiterate, each playdate has a unique id and the id in the request below identifies the specific appointment you are removing from the database.

The route for the DELETE request — /appointments/:id

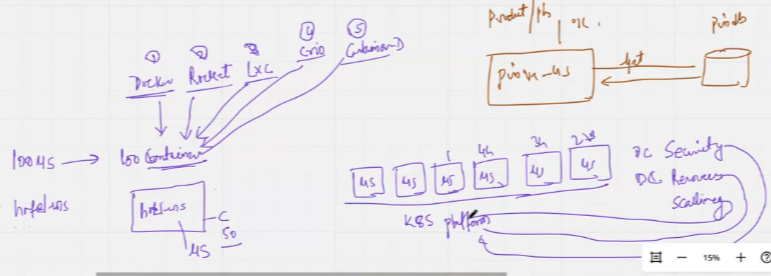
**Conclusion**

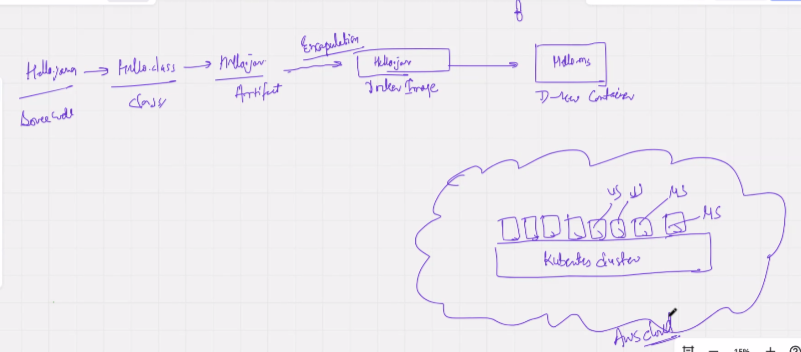
However simple these four actions may seem, they can be found everywhere because they add important functionality that are vital to modern day web development. As a developer, I encourage you to build your applications around this basic functionality. Remember to use CRUD as a guideline in the initial stages of development to help think through what a user should or should not be able to do within your app. So, get creative and show the world what you can build!

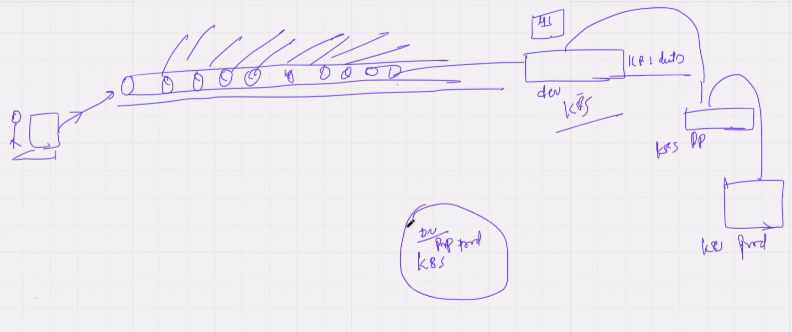
Remember, an application without CRUD is a dud!

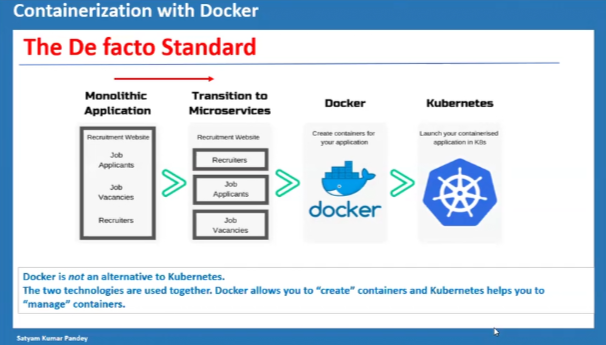
Docker-Kubernetes(Container-Cluster process)

Docker is a containerization platform that allows developers to package applications and their dependencies into a single, portable container. - Kubernetes is an orchestration platform which will manage the entire container lifecycle.







Importance of docker

>> IntelliJ idea runs maven using global path by setting maven environment variable & path.

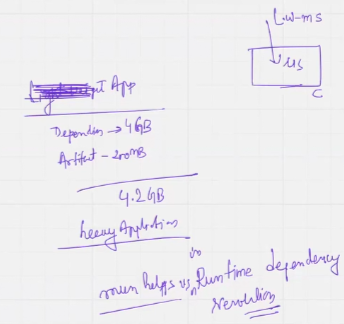
>> Apache maven is a shining example of Apache Software Foundation - <https://en.wikipedia.org/wiki/List_of_Apache_Software_Foundation_projects>

>> POM.XML - Project Object Model. Maven projects hosted intelliJ as a build automation tool refers to pom.xml for dependencies and it is the main file of the microservices. POM.XML can be used with only a block of <project>. Blocks in POM.XML:

1. <parent> → provides properties, inheritance. Pom inherits these attributes from parent pom
2. <properties> → Version details
3. <dependencies> → microservices needs various dependencies which are added here
4. <plugin> → needs addition of spring-boot details

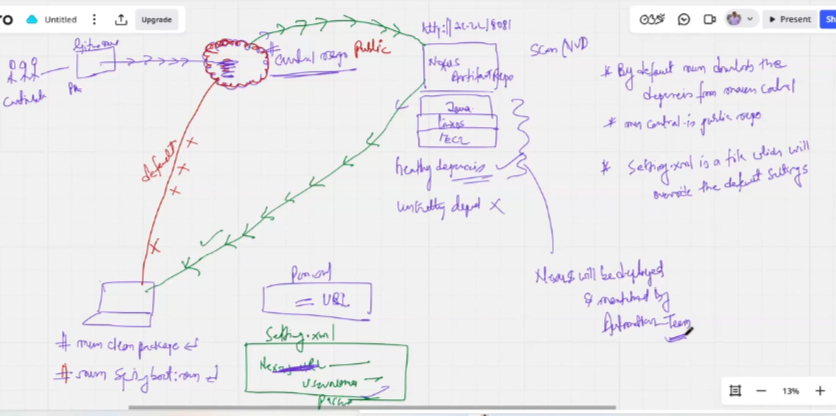
>> .M2 directory stores the repositories on local C: . Maven has a dependency central repository which it uses for downloading to local as per POM requirement - <https://repo.maven.apache.org/maven2/>

>> Light weight applications are the primary requirement of docker image.

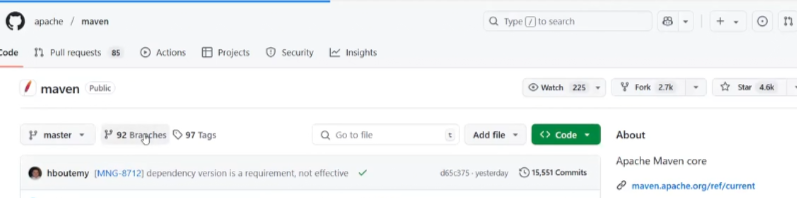


>> <https://www.sonatype.com/products/sonatype-nexus-repository> used at enterprise level by adding nexus settings in settings.xml file which will override the default settings. ~/.m2/settings.xml in linux and c:\users\<you-name>\.m2\settings.xml. Update the server block in the settings.xml and also update the pom.xml

>> National vulnerability database maintained by U.S. to report infected code databases, nexus scan the database and provides only the healthy database by discarding the unhealthy dependencies



>>Contributors of central repository:



>> Maven lifecycle

Install - will install the artefacts to local repository .M2

Deploy - will deploy the code to the nexus repository, so it will need nexus by default.

Running a package command will automatically also run the previous lives.

