**Objective:**

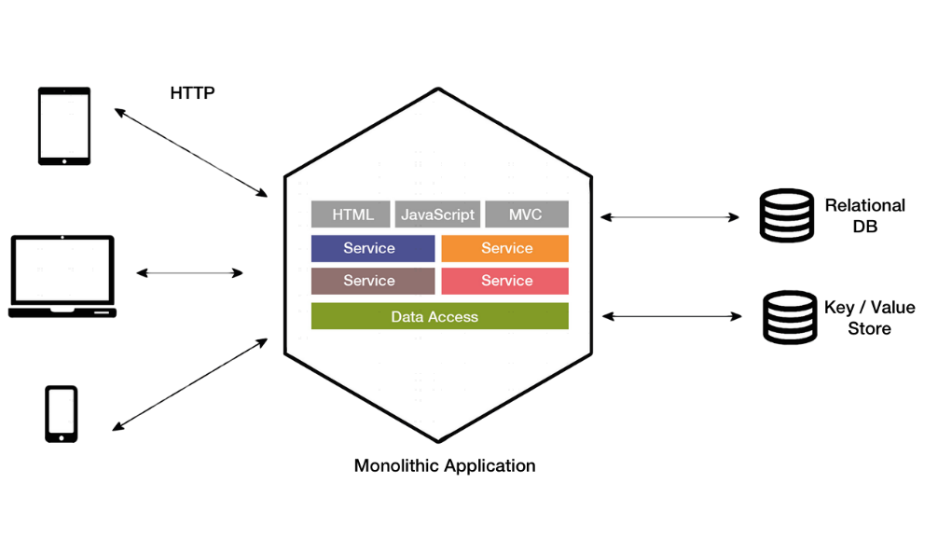
* Understanding Monolithic Architecture
  + What are Monolithic Applications
  + Deploying
  + Scaling Applications
  + Managing State and Data
  + Benefits and Drawbacks of Monolithic Architecture
* Microservices Architecture
  + What are Microservices
  + Monolithic vs Microservices Architecture
  + Characteristics of Microservices Architecture
  + Benefits of using Microservices Architecture
  + Microservices Design Principles.
* SOA vs. Microservice
* Communication between Microservices
  + Synchronous Communication across Microservices.
  + Asynchronous communication across Microservices.
* Handling Data in Microservices
  + SQL vs NoSQL
  + Eventually Consistency
* Microservices Patterns
  + Domain Driven Design
  + Command and Query Responsibility Separation (CQRS)
  + Circuit Breaker Pattern
  + API Gateway Pattern
  + Blue Green Deployment Pattern
* Creating Composite UI with Microservices
* Drawbacks of Microservices
* Overview of Orchestration (Kubernetes) – Docker
* Simple Demo with Kubernetes.

Monolithic Architecture

**What are Monolithic Applications?**

*In software engineering, a monolithic application describes a* ***single-tiered software application****in which the****user interface and data access code*** *are combined into a* ***single program*** *from a single platform.*

*A monolithic application is self-contained and independent from other computing applications.*



|  |  |
| --- | --- |
|  | When developing a server-side application you can start it with a layered architecture which consists of different types of components:   1. **Presentation / User Interface** — responsible for handling HTTP requests and responding with either HTML or JSON/XML (for web services APIs). 2. **Business** **logic** — the application’s business logic. 3. **Database** **access** — data access objects responsible for access the database. |

**Options for Deploying Monolithic Application:**

1. Published, package and deploy in an on-premise server.
2. In Microsoft Azure, they can be deployed
3. Using dedicated VMs for each instance.
4. Using Azure VM Scale Sets, you can easily scale the VMs.
5. Azure App Service can also run monolithic applications and easily scale instances without requiring you to manage the VMs.
6. Package as Single Docker Image and Push to Docker Registry.

**Benefits of Monolithic Architecture**

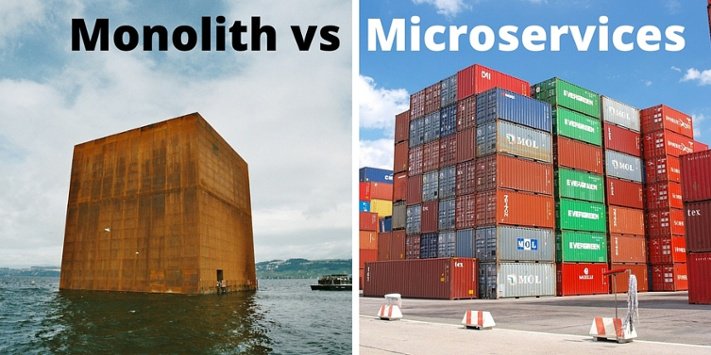
* They are comparatively **simple to develop**.
* They are **simple to test**. You can implement end-to-end testing by simply launching the application and testing the **UI with** **Selenium**.
* They are **simple to deploy**. You just have to copy the packaged application to a server or download the docker container image from docker registry.
* Very **simple to scale** (horizontally) as we just need to run multiple copies behind a load balancer.

**Drawbacks of Monolithic Architecture**

* Application with time becomes **too large and complex.** This makes it difficultto fully understand and make changes fast and correctly.
* The size of the application can **slow** down the **start-up time**.
* If any updates are done to any part of the application, you have to **redeploy** the entire application.
* Impact of a change is usually not very well understood which leads to do **extensive manual testing**.
* **Difficult to scale** when different modules have conflicting resource requirements.
* **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 have 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.
* Also, monolithic pattern conflicts with the container principle “***a container does one thing, and does it in one process***”.

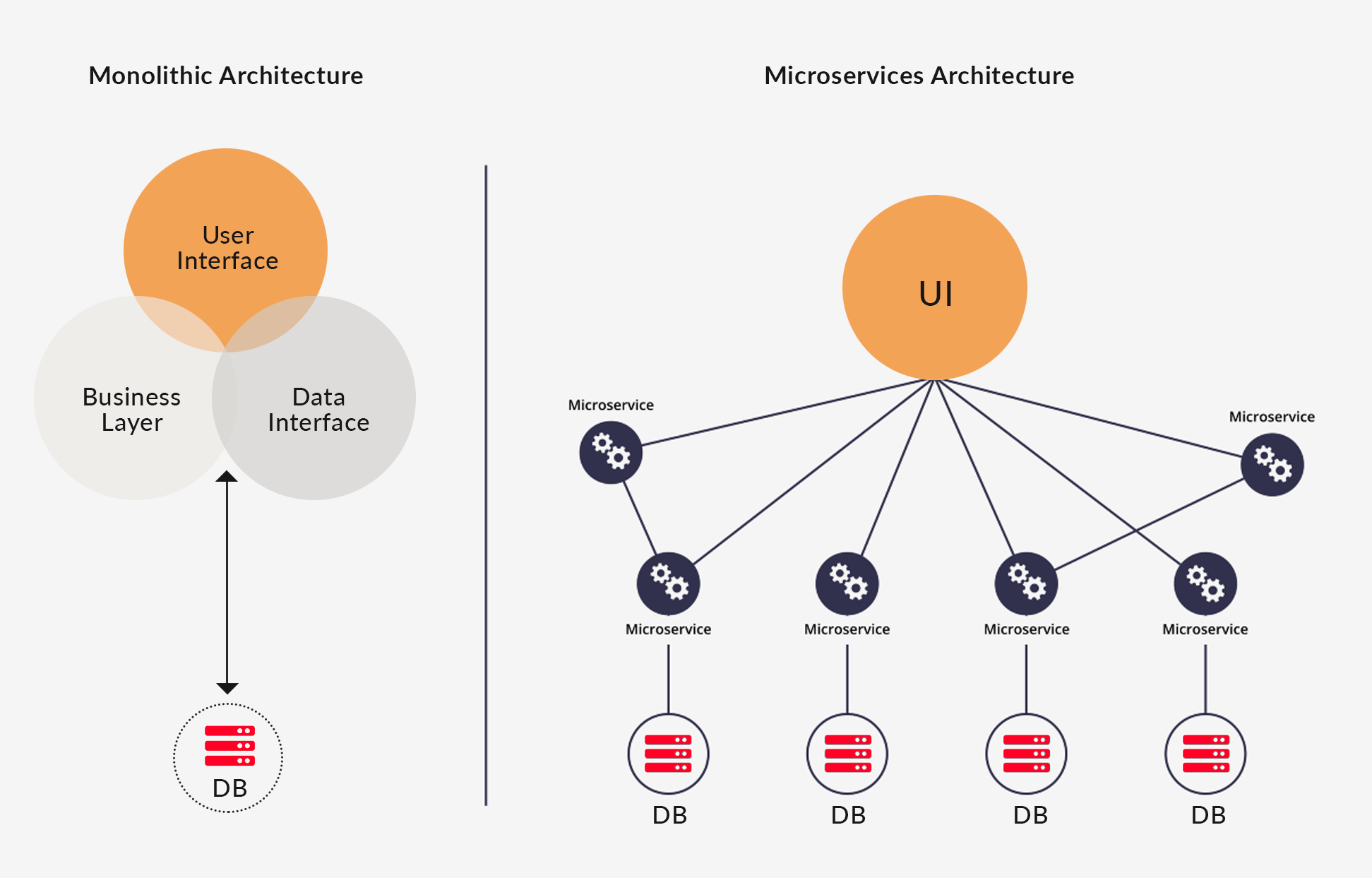
**What are Microservices**

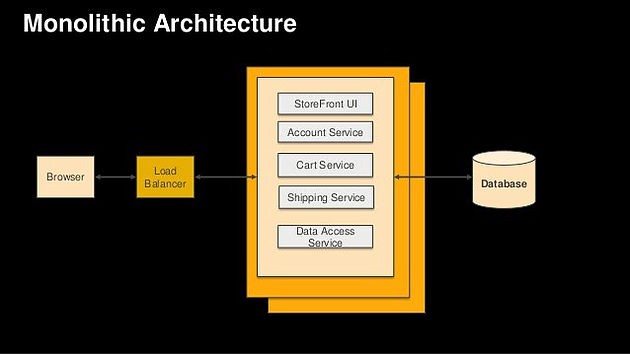
**Microservices – also known as Microservice Architecture - is an architectural style that structures an application as a collection of services where each service is composed of small, independently versioned, and scalable customer-focused service that communicate with other services over standard protocols with well-defined interfaces.**

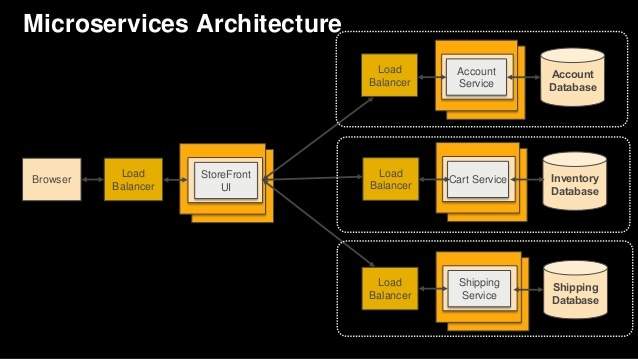


Each microservice is a small application that has its own hexagonal architecture consisting of business logic along with various adapters.

The idea is to split your application into a set of smaller, interconnected services.







Characteristics and Benefits of Microservices

* Each microservice is **relatively small** and implements **a specific end-to-end domain** or business functionality within a certain **context boundary**.
  + **Easier for developers** to understand, develop and maintain small services.
  + **It is quicker to evolve and release business features** because they are smaller and easier to analyze.
  + **Better testability** - services are smaller and faster to test.
  + Improved fault Isolation.
* Microservices are developed by a **small engineering team** using a particular technology stack.
  + Each service can be **developed independently** by a team that is focused on that service.
  + Can be developed autonomously and written in **any programming language** and use any framework.
  + Eliminates any **long-term commitment** to a technology stack.
* Microservice architecture enables each microservice to be **deployed independently**
  + Mostly deployed as containers which are very **simple to deploy**.
  + Can be **upgraded independently** of calling services.
  + There will be **no downtime** while upgrading them.
  + It makes **continuous deployment** possible for complex applications.
* Microservice architecture enables each service to be **scaled independently**.
  + Proper utilization of Server Resources like memory and processor.
* No single point of failure
  + In theory, the services should be **completely independent**, **and all the data and code that they need to execute an operation should be embedded in them,** so if some of the services fail, or their database is offline, other services are fully functional.
* Each service runs in its own process and **communicates** with other processes using **protocols** such as HTTP/HTTPS, WebSockets, or AMQP.
* Needs to have a **unique name** so that its current location is discoverable.
* A microservice needs to be **resilient to failures** and restart often on another machine for **availability** reasons.

Microservices Design Principles

1. **Domain Centric**: Should perform actions only for specific business domain.
2. **Single Focused:** Doing one thing and one thing only. Adhere to Single Responsibility Principle.
3. **Decoupled** from each other with asynchronous channels of communication.
4. **Autonomous**: Individually deployable and scalable.
5. **Resilient**: Ability to fail gracefully having mechanism in place to implement retry logic and pick something up from where left earlier.
6. **Observable:** Centralized monitoring and local mechanism.
7. **Automation:** Deploy using CI/CD pipeline, potentially coupled with DevOps culture.

Here is list of articles published by companies about their experiences using microservices:

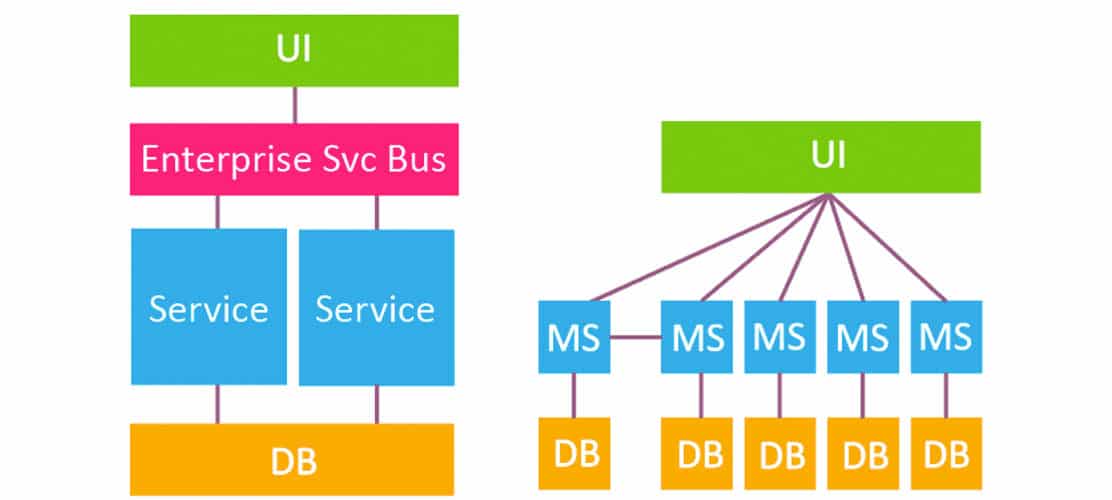
* [Uber](https://eng.uber.com/soa/)
* [Netflix](http://techblog.netflix.com/)
* [Amazon](http://highscalability.com/amazon-architecture)
* [Ebay](http://www.addsimplicity.com/downloads/eBaySDForum2006-11-29.pdf)
* [Sound Cloud](https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-2-breaking-the-monolith)
* [Karma](https://blog.yourkarma.com/building-microservices-at-karma)
* [Groupon](https://engineering.groupon.com/2013/misc/i-tier-dismantling-the-monoliths/)
* [Hailo](https://sudo.hailoapp.com/services/2015/03/09/journey-into-a-microservice-world-part-1/)
* [Gilt](https://qconnewyork.com/ny2015/presentation/microservices-and-art-taming-dependency-hell-monster)

Service Oriented Architecture vs Microservices

## A **Service-Oriented Architecture** is essentially a collection of services. These services communicate with each other. The communication can involve either simple data passing or it could involve two or more services coordinating some activity.

## Related image

Communication in SOA passes through an Enterprise Service Bus or ESB. An ESB promotes a monolithic structure, is characterized by slow communication speeds, and often ends up becoming a **single point of failure**.



|  |  |  |
| --- | --- | --- |
|  | **Traditional SOA** | **Microservices** |
| **Scope** | Enterprise wide reusability | Application wide reusability |
| **Messaging Type** | Synchronous / Wait for response | Asynchronous: Publish and Subscribe |
| **Programming Style** | Imperative model | Reactive programming (event / callback driven) |
| **Lines of Code per Service** | Hundreds / Thousands | Hundreds or fewer |
| **State** | Stateful | Stateless |
| **Database** | Large RDBMS Shared by multiple services. | RDBMS + NoSQL + …  Each microservice has its own database. |

**Summary**  
*Microservices are not the same as Service Orientated Architecture – Microservices is* an approach to developing a **single application** as a suite of small services, each running in its **own process** and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around **business capabilities and independently deployable** by fully automated deployment machinery.”

**“The microservice architecture is SOA done right.”**

Communication between Microservices

A microservices-based application is a distributed system running on multiple processes or services, usually even across multiple servers or hosts. Each service instance is typically a process. Therefore, services must interact using an inter-process communication protocol such as HTTP/REST, AMQP, or a binary protocol like TCP, depending on the nature of each service.

**The two commonly used protocols**

1. **Synchronous Protocol**: HTTP request/response.

Request/response communication is especially well suited for querying data for a real-time UI (a live user interface) from client apps.

Diagram

Description automatically generated

**Drawback of Request/response communication:**

If you make a call from one microservice to other microservices to be able to provide a response to a client application, you have an architecture that **won't be resilient** when some microservices fail.

Diagram

Description automatically generated

1. **Asynchronous Protocol** like AMQP when communicating updates across multiple microservices. It just sends the message as when sending a message to a any message broker like RabbitMQ or Azure Service Bus.

Diagram

Description automatically generated

**The second axis is defining if the communication has a single receiver or multiple receivers:**

* **Single receiver message-based communication**. Each request must be processed by exactly one receiver or service. An example of this communication is the **Command pattern**.

Diagram

Description automatically generated with medium confidence

* **Multiple-receivers message-based communication**. Each request can be processed by **zero to multiple receivers**. This type of communication must be **asynchronous**. An example is the **publish/subscribe** mechanism used in patterns like **Event-driven architecture**. This is based on an event-bus interface or message broker when propagating data updates between multiple microservices through events; it is usually implemented through a service bus or similar artifact like Azure Service Bus by using **topics and subscriptions**.

Diagram

Description automatically generated

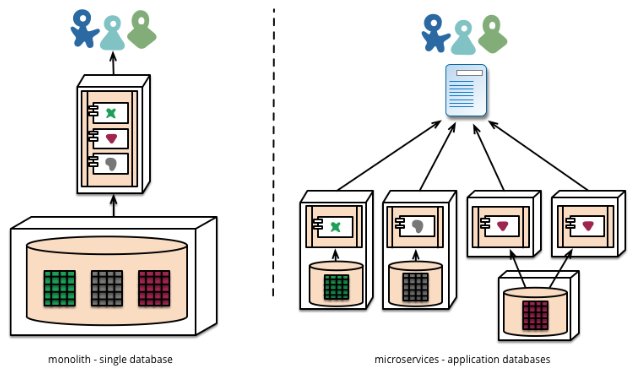
**Types of Message Types**

* **Commands**
  1. Do this Please
  2. Post a Command
  3. Eg: Send Email
* **Events**
  1. "This Happened"
  2. Subscribe to an event
  3. Eg: OrderPlaced

**Handling Data in Microservices**

The **traditional** (monolithic data) approach used in many applications is to have a **single centralized database**.

An important rule for microservices architecture is that each microservice **must own its domain data and logic**.



A monolithic application with typically a single relational database has important benefits: **ACID transactions** and the SQL language, both working across all the tables and data related to your application. This approach provides a way to easily write a query that combines data from multiple tables.

**Drawbacks of One Database for all Services:**

* **Single point of failure**. If database goes down all the service instances would stop.
* **Cannot scale** unless we manually take care of scaling the database.
* **No ownership control** – which service owns which part of the database, meaning that it's really hard to refactor, add or change functionality without potentially affecting other services.
* **Provider lock-in** – You are tied to one database provider, limited in your options of investing in alternative technologies.

**Database Autonomy in Microservices**

The data owned by each microservice is private to that microservice and can only be accessed via its microservice API. This ensures that the microservices are **loosely coupled** and can **evolve independently** of one another.

Going even further, different microservices often use **different kinds** of databases. It can be either NoSql Azure DocumentDB or MongoDB which offer better performance and scalability and, in some cases, a relational database is better approach.

**SQL vs NoSQL**

|  |  |
| --- | --- |
| **SQL** | **NoSQL** |
| RDBMS | Non-relational or distributed database system. |
| Fixed or predefined schema | Dynamic schema |
| Not suited for hierarchical data storage. | Best suited for hierarchical data storage. |
| Best suited for complex queries | Not so good for complex queries |
| Vertically Scalable | Horizontally scalable |
| Follows ACID property | Follows CAP (consistency, availability, partition tolerance) |

### **Designing atomicity and resiliency when publishing to the event bus**

How changes to database of one microservice can be reflected in database of another microservice.

**EventLog table: Status =** Ready to Publish to Event already Published

Diagram

Description automatically generated

**API Gateway Pattern**

In a microservices architecture, the client apps usually need to consume functionality from more than one microservice. If that consumption is performed directly, the client needs to handle **multiple calls to microservice endpoints**. What happens when the application evolves and new microservices are introduced or existing microservices are updated? If your application has many microservices, handling so many endpoints from the client apps can be a nightmare. Since the client app would be coupled to those internal endpoints, evolving the microservices in the future can cause high impact for the client apps.

Therefore, having an **intermediate level or tier of indirection (Gateway)** can be very convenient for microservice-based applications.

**What is API Gateway Pattern**

This is a service that provides a single-entry point for certain groups of microservices. The API Gateway pattern is also sometimes known as the “backend for frontend” (BFF) because you build it while thinking about the needs of the client app.

The most important and foundational features for any API Gateway are the following design patterns:

1. **Reverse proxy or gateway routing:** The gateway provides a single endpoint or URL for the client apps and then internally maps the requests to a group of internal microservices.
2. **Requests aggregation**. As part of the gateway pattern you can aggregate multiple client requests (usually HTTP requests) **targeting multiple internal microservices** into a single client request.

**Using an API Gateway implemented as a custom ASP.NET Core WebHost service running as a container:**

Diagram

Description automatically generated

**Functionality Offload:**

Depending on the features offered by each API Gateway product, you can offload functionality from individual microservices to the gateway, which simplifies the implementation of each microservice by consolidating cross-cutting concerns into one tier.:

1. Authentication and authorization
2. IP whitelisting
3. Headers, query strings, and claims transformation
4. Service discovery integration
5. Response caching
6. Retry policies and circuit breaker
7. Rate limiting and throttling
8. Load balancing
9. Logging, tracing, correlation

Domain Driven Design (DDD) - Identifying Domain-Model Boundaries for each Microservice

Domain modeling is at the heart of designing coherent and loosely coupled microservices.

The goal when identifying model boundaries and size for each microservice is **not to get to the most granular separation** possible, although you should tend toward small microservices if possible. Instead, your goal should be **to get to the most meaningful separation** guided by your domain knowledge. **The emphasis is not on the size, but instead on business capabilities.**

**What is Domain Driven Design (DDD)?**

**“DDD** **is an approach to building software with a focus on the domain.”**

But we can say that DDD is an approach for you to **build your domain model** in the right way.

In the context of building applications, DDD talks about problems as domains having **complex business logic**.

Implementing DDD is **not trivial** and should be analyzed if it really makes sense. If your domain is **not complex**, DDD may **not** be the **best choice.**

**Three Design Pillars in DDD**

1. **Strategic Design**
2. **Tactical Design**
3. **Architecture Design**
   1. **Strategic Design:**

* Also called strategic modeling, it is a pillar of the DDD whose main goal is to define **Bounded contexts and Ubiquitous Language** along with the entire project team.
* Where to draw the boundaries is the key task when designing and defining a microservice.

**Understanding Types of Subdomains:**

#### Core or Basic

#### Auxiliary or Support

#### Generic

Diagram

Description automatically generated

Usually, there is **only one Core**, that represents the main part of the Domain.

**Bounded Context:**

* Bounded Context is where the **Model** is implemented, that is, Bounded Context is the solution implementation in a technical way.
* For the domain model for each Bounded Context, you identify and define the **entities**, **value objects**, and **aggregates** that model your domain.
* The components within those boundaries end up being your **microservices**.

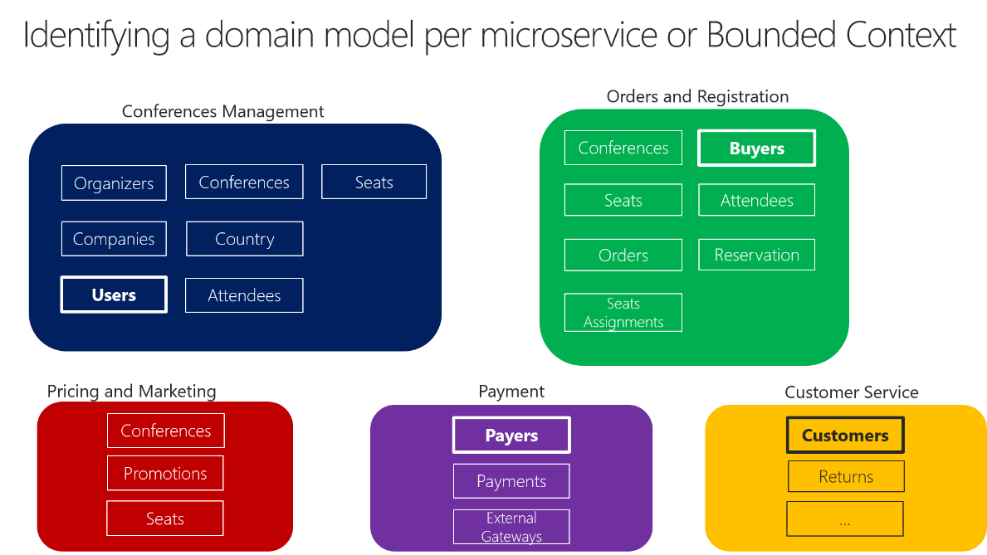
**Bounded Context vs SubDomain**

* Subdomains live in the **space of the problem**and the Bounded Contexts in the **solution space**.
* Subdomains are **logical** “separations” of the domain and Bounded Contexts are **technical solutions**.
* It is desirable that code belonging to a Bounded Context implements a single Subdomain. So, we segregate Domain Models by real business intent.

## What is domain logic? · Enterprise Craftsmanship

**Ubiquitous Language:**

* A language structured around the domain model and used by all team members including **developers and domain experts** to connect all the activities of the team with the software. Bridges the gap between developers and experts.
* Ubiquitous Language should never be a set of terms and technical jargons imposed by domain experts, on the contrary, Ubiquitous Language is developed in agreement of the whole team.
* The Ubiquitous Language must be spoken at all times between the team members and expressed in the software model.



## **Characteristics of the Ubiquitous Language**

* It must be expressed in the Domain Model.
* It unites the people of the project team.
* It eliminates inaccuracies and contradictions from domain experts.
* It is not a business language imposed by domain experts.
* It is not a language used in industries.
* It evolves over time, it is not defined entirely in a single meeting.
* Concepts that are not part of the Ubiquitous Language should be rejected.

**Tactical Design:**

The Tactical Design, is a set of technical resources used in the construction of your Domain Model, these resources must be applied to work in a single Bounded Context**.**

* Domain Entity
* Value Object
* Aggregate Root
* Services
* Respositories
* Factories
* Events

Diagram

Description automatically generated

**Layered Architecture in DDD:**

Most enterprise applications with significant **business and technical complexity** are defined by **multiple** layers. The layers are a **logical artifact**, and are not related to the deployment of the service. They exist to help developers manage the complexity in the code.

You want to design the system so that each layer communicates **only with certain other** layers. For instance, **the domain model layer** should **not** take a **dependency** on any other layer (the domain model classes should be Plain Old CLR Objects, or POCO, classes

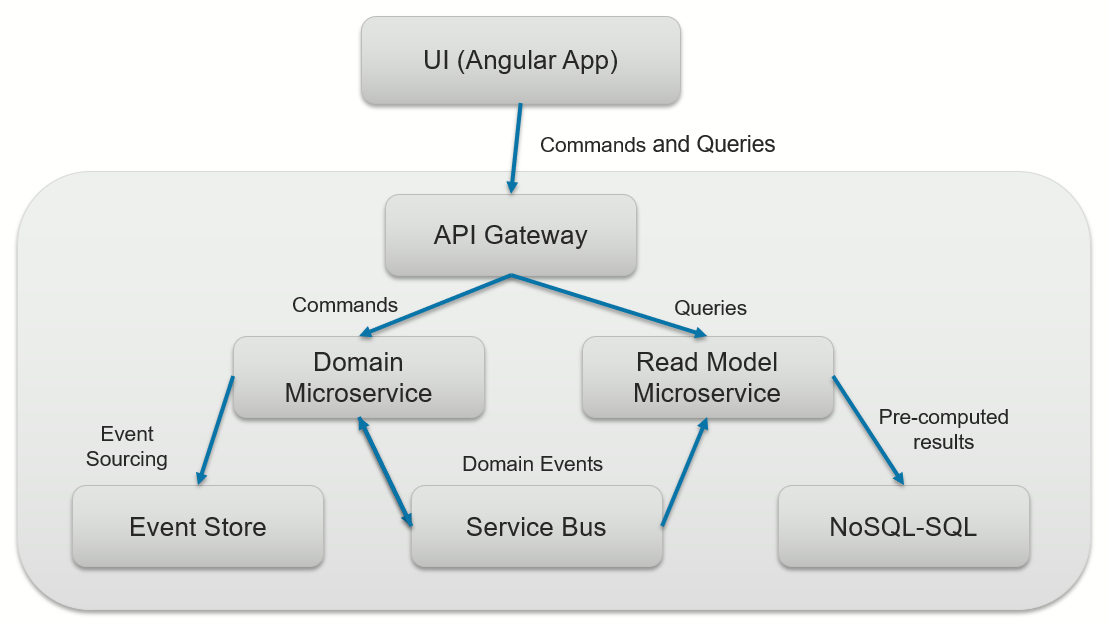
Diagram

Description automatically generated

**Command and Query Responsibility Segregation (CQRS) Pattern**

The CQRS pattern provides a guideline for identifying services and distributing various business aspects. The basic idea is to divide the operations that act on a domain object into two distinct categories:

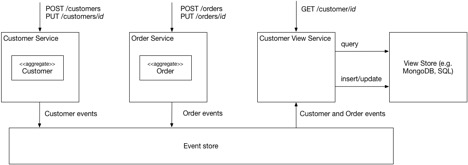
* **Queries**—methods that return a result and do not change the system state.
* **Commands**—methods that change the system state but do not return values.



So here is a simple walk through on how things work with this Architecture:

1. UI Application (for example an Angular application) makes a request to API Gateway over Http.
2. This request is either a “Command” or a “Query”.
3. Commands will be passed to a Domain Microservice, which only accepts commands. Domain Microservice does not accept queries at all.
4. Domain Microservice receives the Command, persists the event in memory and sends the event to be persisted to the Event Store database.
5. Domain Microservice will also publish those events to the Service Bus.
6. As Read Model Microservice is interested in this event, it is Subscribed to this event, so it will receive this Event through the Service Bus, and updates its version of denormalized data and having ready for queries.
7. If the request is a “Query”, it will be sent to the Read Model Microservice.

Figure below shows the CQRS pattern applied to the online store example. The Customer Service and the Order Service are command-side services. They provide APIs for creating and updating Customers and orders. The Customer View Service is a query-side service. It provides an API for querying customers.



The Customer View Service subscribes to the Customer and Order events published by the command-side services. It updates a view store that is implemented using MongoDB.

**When to use CQRS pattern**

* Task-based user interfaces where users are guided through a **complex process** as a series of steps or with complex domain models. The write model has a full command-processing stack with **business logic, input validation and business validation** to ensure that everything is always consistent in the write model. The **read model has no business logic** and just returns a DTO for use in a view model. The read model is **eventually consistent** with the write model.
* Scenarios where **performance** of data reads must be fine-tuned separately from performance of data writes, especially when the read/write ratio is very high and when horizontal scaling is required.
* Scenarios where one team of **developers can focus** on the complex domain model that is part of the write model and another team can focus on the read model and the user interfaces.

Circuit Breaker Pattern

There can also be situations where **faults** are due to unanticipated events that might take much **longer to fix**. These faults can range in severity from a partial loss of connectivity to the complete failure of a service. In these situations, it might be pointless for an application to continually retry an operation that's unlikely to succeed. Instead, the application should be coded to accept that the operation has failed and handle the failure accordingly.

1. We have four different services namely Service A, B, C & D.
2. Because of the Client request, Service A will connect with Service B and B to C and C to D.
3. But, Service D failed due to some reason.
4. So, Connections made to Service D by other Services will get to wait and unresponsive service D will remain in the same state.
5. Therefore, Service C will continuously retry to make the connection with Service D. As so, Service B and Service A.
6. This will cause Cascade failure and entire application, or the part of service will go unavailable

A picture containing text, device

Description automatically generated

The circuit breaker can be viewed as a **state machine** that starts in the **closed state/normal state** and allows the flow of request across it.

When a problem is detected the circuit breaker moves to the **open state**, blocking all requests for specified period.

After that period elapses the circuit breaker moves to a **half-open state** where the first request is treated as a test request.

If this test request succeeds the circuit **closes and normal** operation resumes, but if it **fails** the circuit moves **back to open** and remains there for a specified period before once again moving to half-open.

A picture containing text, clock

Description automatically generated

1. All the connections between services will go through Circuit Breaker.
2. When Service D fails and defined the maximum number of failures is reached Circuit breaker for Service C, Circuit Breaker will trip, and connection will immediately stop.
3. Then the Opened Connection will slowly start communicating with limited allowed number of connection to test the stability of unavailable service.
4. If the test connections are succeeded, then, communication across the Services will resume normally.
5. If the Connections are failed, then Circuit Breaker will create Monitor & Alert event or if we have any Self-healing implementation, will be triggered.

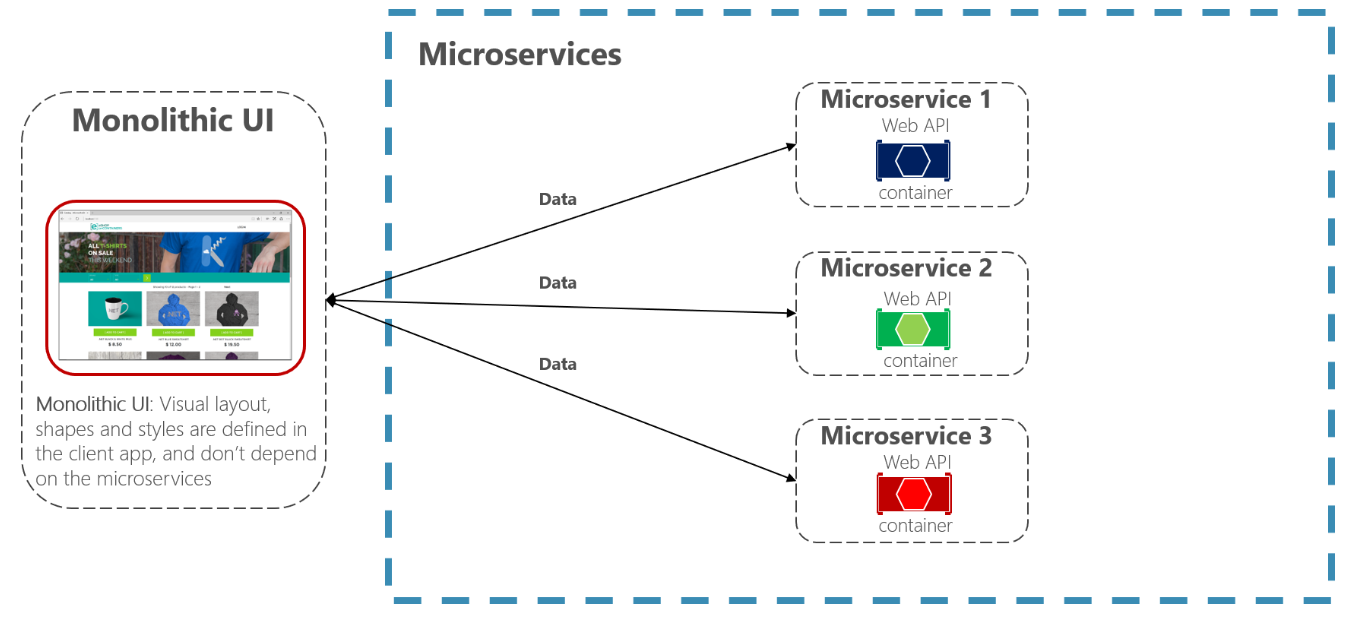
Diagram, schematic

Description automatically generated

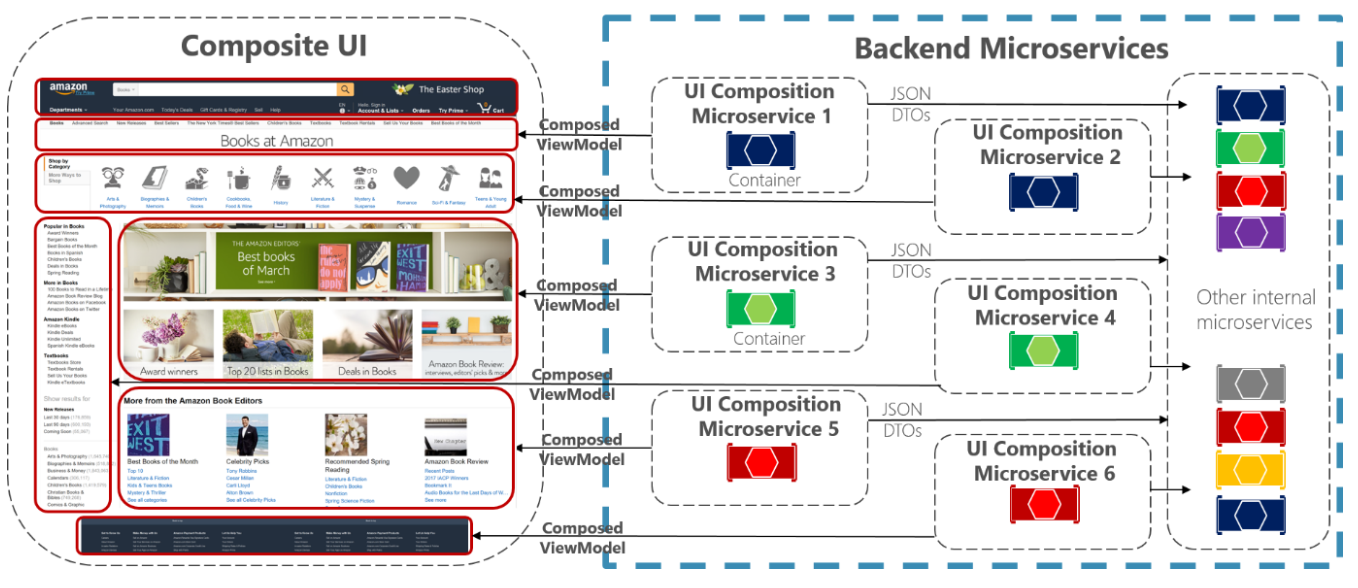
Creating Composite UI with Microservices

Microservices architecture often starts with the server-side handling data and logic. However, a more advanced approach is to design your application UI based on microservices as well. That means having a composite UI produced by the microservices, instead of having microservices on the server and just a monolithic client app consuming the microservices. With this approach, the microservices you build can be complete with both logic and visual representation.

**A monolithic UI application consuming back-end microservices**



**Example of a composite UI application shaped by back-end microservices**



Each of those UI composition microservices would be similar to a small API Gateway. But in this case each is responsible for a small UI area.

Drawback of Microservices

* **Distributed services adds more network communication**
  + Increased network hops.
  + Inter service communication using protocols like HTTP or AMPQ
  + Requires failure/recovery code
  + Need service discovery solution
* **Distributed application architecture adds complexity** for developers when they are designing and developing the services. For example, developers must implement inter service communication using protocols like HTTP or AMPQ, which adds complexity for testing and exception handling. It also adds **latency** to the system.
* **Deployment is complex**. An application that has **dozens** of microservices types and needs high **scalability** means a high degree of deployment complexity for IT operations and management.
* **Debugging and Testing is difficult**. For a similar test for a service you would need to launch that service and any services that it depends upon it. People used to just press F5 and run the whole thing and debug, but now when some call happens, one service is called, it sends a message to the “Service Bus”, gets picked up by another “Message Broker”, then does some data updates, and then some other service is notified and does some other stuff, and then posts another message to the “Service Bus” and then …
* **Multiple Repositories and Multiple Pull Requests**: With Monolith, there are very limited number of code repositories, so if you are building a feature, that touches 4 different areas of the system, you used to make all the changes, test it locally and make 1 pull request to merge your changes to the proper code branch. But now, these 4 different areas that you touched, might belong to different services, so you have to make 4 different pull requests for each service, and also make sure the order that they are merged and deployed and etc.
* **Monitoring/Logging is difficult**.
* **What’s the Current version of the system**: Now that we deploy each service independently, and each team can release whenever they want, and the version they want, what is the version of our system? What if we deploy 2 different versions of the same service at the same time? Do we need to version our Events and Messages that we publish to “Service Bus”?
* **Distributed databases make transactions hard**. Atomic transactions between multiple microservices usually are not possible. You need to forget about distributed transactions, and two phase commits, atomic database operations and etc and essentially, just implement workflows that you are able to execute them in reverse order.
* **Cluster and orchestration tools overhead**.
* **Advanced DevOps capability** will be required.

**Role of Orchestration Servers**

Containerization has brought a lot of flexibility for developers in terms of managing the deployment of the applications. However, the more granular the application is, the more components it consists of and hence requires some sort of management for those.

One still needs to take care of scheduling the deployment of a certain number of containers to a specific node, managing networking between the containers, following the resource allocation, moving them around as they grow and much more.

Nearly all applications nowadays need to have answers for things like

* Replication of components
* Auto-scaling
* Load balancing
* Rolling updates
* Logging across components
* Monitoring and health checking
* Service discovery
* Authentication

The process of organizing multiple **containers and managing them as needed** is known as **container orchestration**.

**Kubernetes**, a container orchestration technology used to orchestrate the deployment and management of hundreds and thousands of containers in clustered environment.

* Kubernetes is an open source project was started by Google in the year 2014.  **Redhat** is the second major contributor along with Microsoft, HP, VMWare etc…
* It is a **platform** designed to completely manage the life cycle of containerized applications and services using methods that provide **predictability, scalability, and high availability**.
* It orchestrates **computing**, **networking**, and **storage** infrastructure on behalf of user workloads.
* It is easier to **deploy**, **scale**, and **manage** applications.
* Kubernetes aims to support an extremely diverse variety of workloads, including **stateless**, **stateful**, and **data-processing** workloads. If an application can **run in a** **container**, it should run great on Kubernetes.

**Kubernetes is:**

* a container platform.
* a microservices platform
* a portable cloud platform.

**Following are some of the important features of Kubernetes.**

* Simplifies Application Deployment
* Continues development, integration and deployment
* Application-centric management and better hardware utilization.
* Modern tooling, have CLI and management REST API support.
* Highly Scalable infrastructure with Autoscaling support
* Kubernetes Monitors Health of each node and does self-healing of docker containers.
* Rolling updates

**Important features of Kubernetes.**

* **Simplifies Application Deployment**: Deploy a specified number of containers to a specified host and keep them running in a desired state.
* **Rolling updates:** A rollout is a change to a deployment. Kubernetes lets you initiate, pause, resume, or roll back rollouts.
* **Service discovery**: Kubernetes can automatically expose a container to the internet or to other containers using a DNS name or IP address.
* **Storage provisioning:** Set Kubernetes to mount persistent local or cloud storage for your containers as needed.
* **Load balancing and scaling:** When traffic to a container spikes, Kubernetes can employ load balancing and scaling to distribute it across the network to maintain stability.
* **Self-healing for high availability**: When a container fails, Kubernetes can restart or replace it automatically; it can also take down containers that don’t meet your health-check requirements.
* **DevOps Support:** Continues development, integration and deployment using DevOps tools and services.
* **Health Monitoring:** Kubernetes Monitors Health of each node and does self-healing of docker containers.