**Agenda: Introduction to Microservices**

* Monolithic Architecture
* Microservices Architecture
  + Monolithic vs Microservices Architecture
  + Characteristics of Microservices Architecture
  + Benefits of using Microservices Architecture
* SOA vs. Microservice
* Communication between Microservices
  + Synchronous Communication across Microservices.
  + Asynchronous communication across Microservices.
* Creating Composite UI with Microservices
* Drawbacks of Microservices
* Role of Orchestration Servers and Kubernetes
* Understanding JWT Token.

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.*

A picture containing text, diagram, font, screenshot

Description automatically generated

|  |  |
| --- | --- |
|  | 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. |

**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**

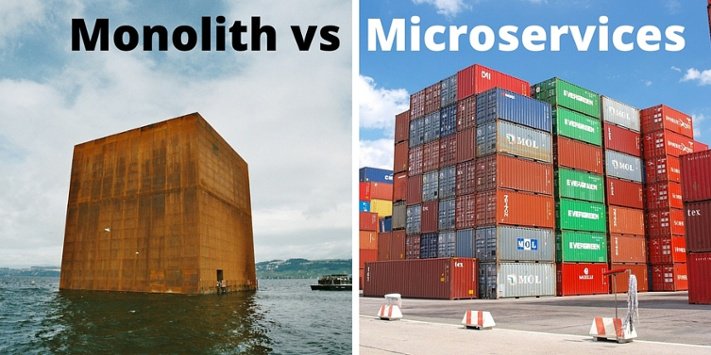
* Application with time becomes **too large and complex.** This makes it difficultto fully understand and make changes fast and correctly.
* The size of the application can **slow** down the **start-up time**.
* If any updates are done to any part of the application, you have to **redeploy** the entire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

**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.**

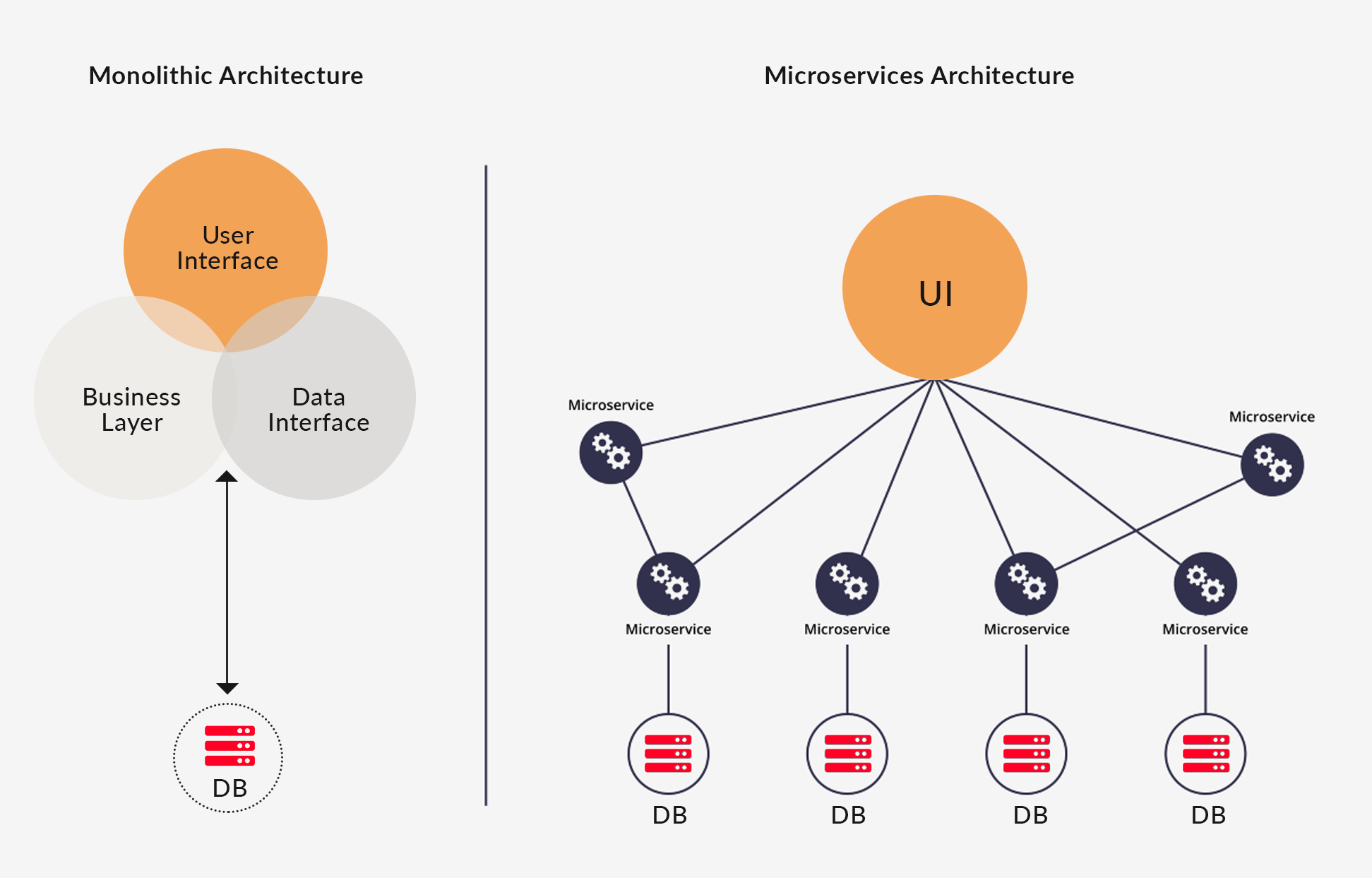
Each microservice is a small application that has its own hexagonal architecture consisting of business logic along with various adapters.

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



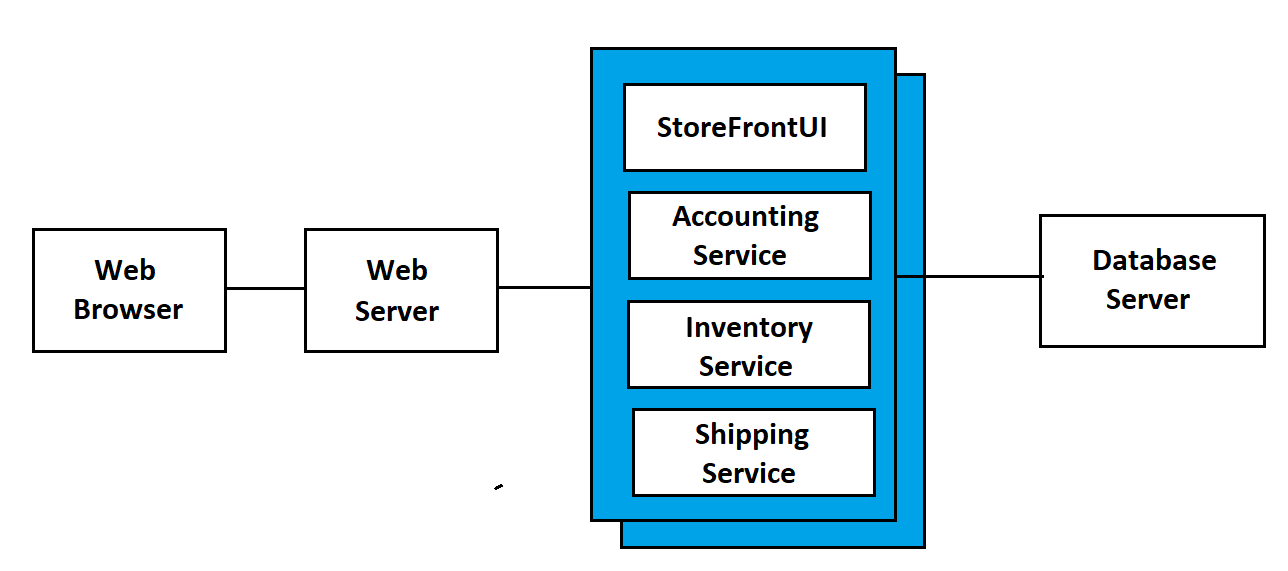
Also, monolithic pattern conflicts with the container principle "***a container does one thing, and does it in one process***".

The microservices approach allows agile changes and rapid iteration of each microservice, because you can change specific, small areas of complex, large, and scalable applications.

A diagram of a software company

Description automatically generated with medium confidence

**StoreFront Monolithic Architecture:**



**StoreFront Microservices Architecture:**



A picture containing text, diagram, software, computer icon

Description automatically generated

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.
  + Interoperability of code
* 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.
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.

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)

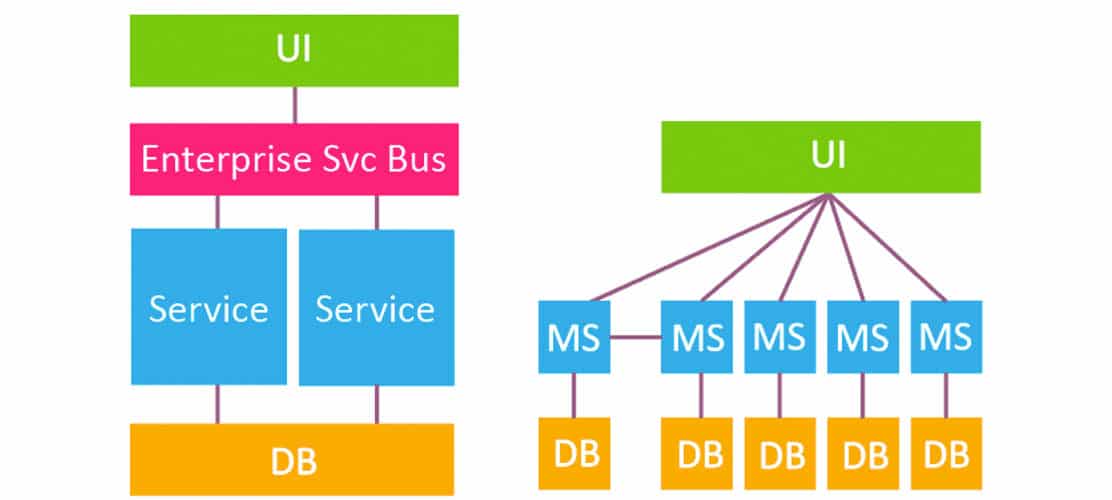
Service Oriented Architecture vs Microservices

## 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

|  |
| --- |
| A diagram of a service bus  Description automatically generatedEnterprise service bus (ESB) is a standardized integration platform that provides features such as messaging, data transformation, intelligent routing, and mediation. It acts as a ‘middleware layer’ to ensure that data is exchanged seamlessly and securely. It binds the organization’s IT infrastructure together. This helps manage the company’s data from one place rather than spreading it across different systems. |

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. Decisions about services are made by a central architecture or governance team. | Application wide reusability. Each microservice team can operate independently with minimal coordination with other teams. |
| **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 / fewer |
| **State** | Stateful | Stateless |
| **Database** | Large RDBMS Shared by multiple services. | RDBMS + NoSQL + …  Each microservice has its own database. |
| **Protocols** | SOAP – Verbose because of XML | RESTfull = HTTP + JSON |

**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**

* **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 generatedDiagram

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

* **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

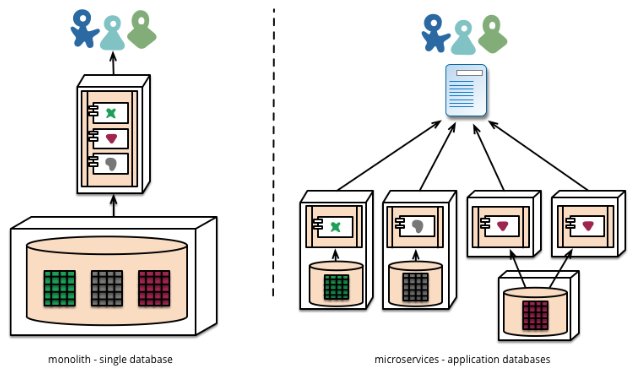
**Types of Message Types**

* **Commands**
  1. Do this please…
  2. Post a Command for just one receiver
  3. Fire and Forget the message.
  4. Eg: Send Email, Create Order, Delete User
  5. Queue
* **Events**
  1. "This Happened" (in past)
  2. Multiple Subscriber to an event
  3. Fire and Forget the message.
  4. Eg: OrderPlaced, UserLimitChanged
  5. Topic and Subscription

**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.

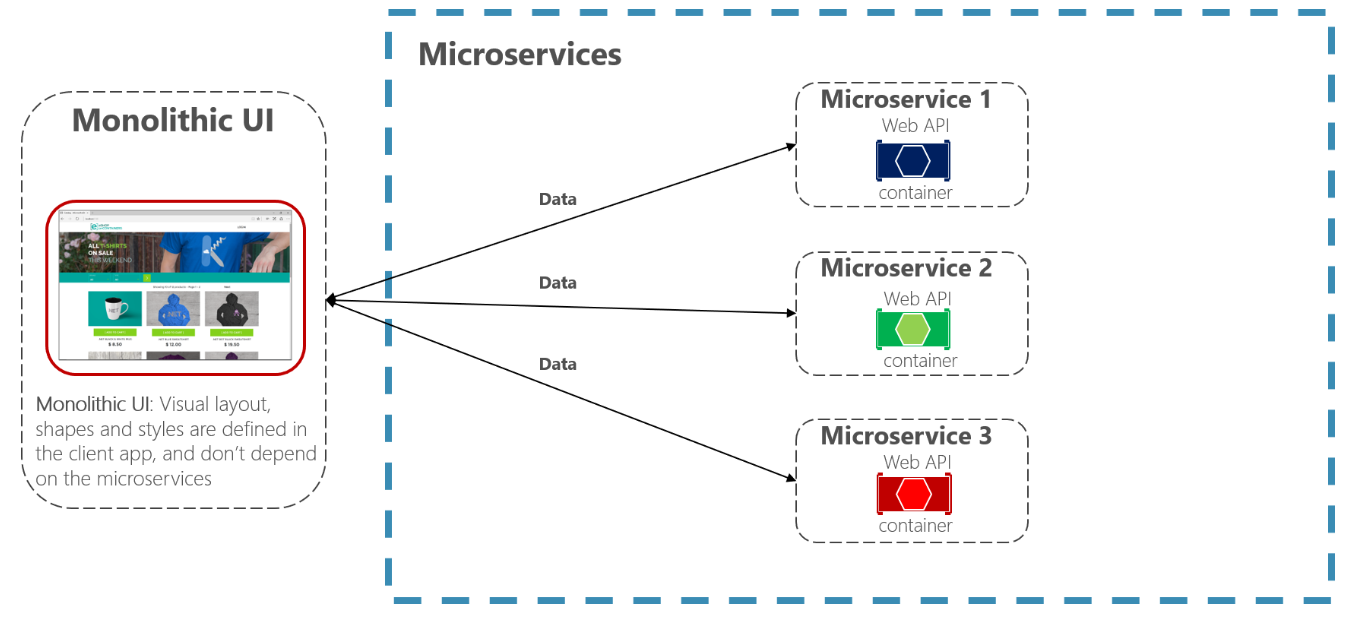
**Drawback:**

Distributed data structures mean that you **cannot make a single ACID transaction** across microservices. Also, it is much harder to implement than simple SQL joins; similarly, many other relational database features are not available across multiple microservices. Also, it often results in **duplication of some data**.

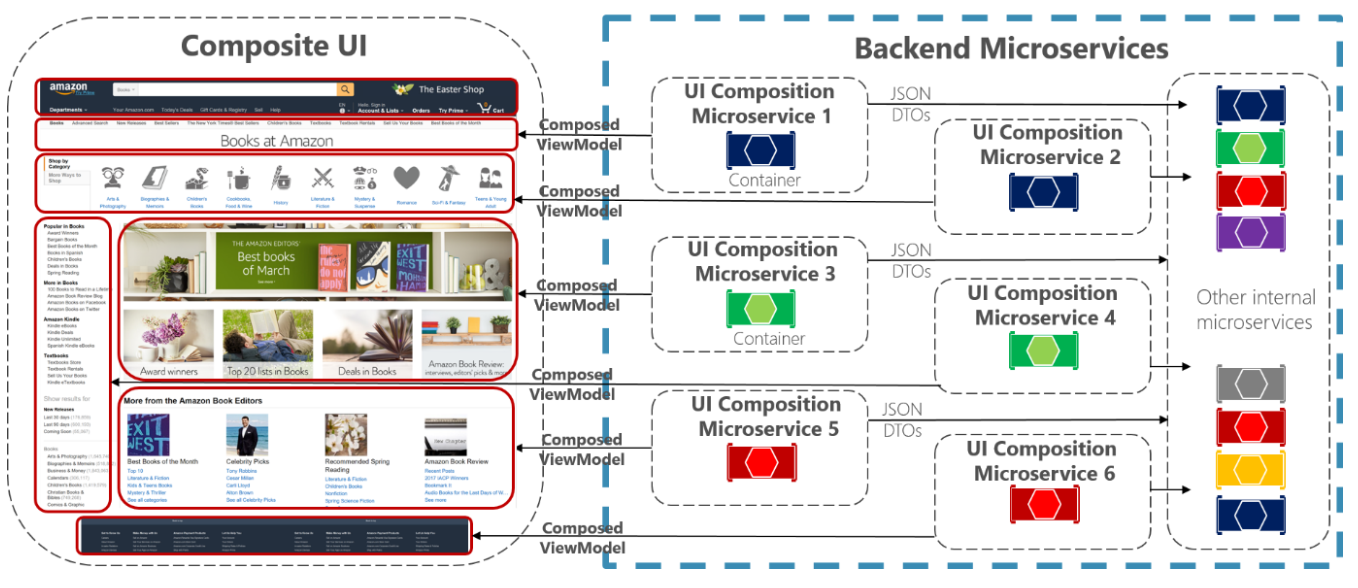
Creating Composite UI with Microservices

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
* 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.

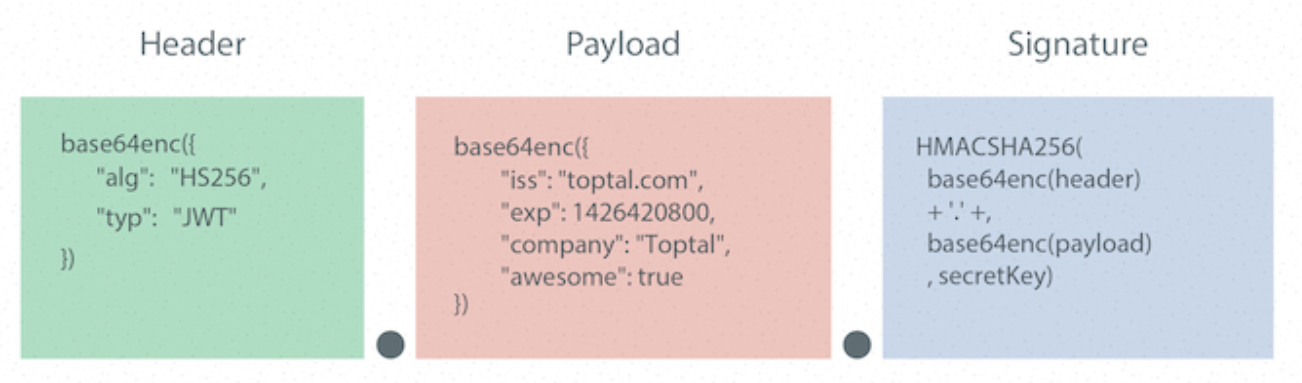
**About JWT Token**

JWT are an important piece in ensuring trust and security in your application. JWT allow claims, such as user data, to be represented in a secure manner.

A JSON Web Token (JWT) is a JSON object that is defined in RFC 7519 as a safe way to represent a set of information between two parties.

The token is composed of a header, a payload, and a signature.

**Format of JWT Token = header.payload.signature**



**JSON Web Token example:**

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.

eyJpc3MiOiJ0b3B0YWwuY29tIiwiZXhwIjoxNDI2NDIwODVlLCJjb21wYW55IjoiVG9wdGFsIiwiYXdlc29tZSI6dHJ1ZX0.

yRQYnWzskCZUxPwaQupWkiUzKELZ49eM7oWxAQK\_ZXw

**Steps to Create a JWT Token and Authenticate User:**

**Step 1. Create the HEADER**

The header component of the JWT contains information about how the JWT signature should be computed.

JWT Header

{

"alg": "HS256",

"typ": "JWT"

}

This JWT Header declares that the encoded object is a JSON Web Token, and that it is signed using the HMAC SHA-256 algorithm.

Once this is base64 encoded, we have the first part of our JWT.

**Step 2. Create the PAYLOAD**

In the context of JWT, a claim can be defined as a statement about an entity (typically, the user), as well as additional meta data about the token itself. The claim contains the information we want to transmit, and that the server can use to properly handle authentication. There are multiple claims we can provide;

These claims are **not** intended to be **mandatory** but rather to provide a starting point for a set of useful, interoperable claims.

* **exp**: Token expiration time defined in Unix time
* **iss**: The issuer of the token
* **sub**: The subject of the token
* **aud**: The audience of the token
* **nbf**: "Not before" time that identifies the time before which the JWT must not be accepted for processing
* **iat**: "Issued at" time, in Unix time, at which the token was issued
* **jti**: JWT ID claim provides a unique identifier for the JWT

**Example Payload**

{

"iss": "toptal.com",

"exp": 1426420800,

"company": "deccansoft",

"awesome": true

}

**Step 3. Create the SIGNATURE**

The signature is computed using the following pseudo code:

data = base64urlEncode(header) + "." + base64urlEncode(payload)

hashedData = HMACSHA256(data, secretkey)

**Step 4: Create JWT Token**

**JWT Token** = base64urlEncode( header ) + "." + base64urlEncode( payload ) + "." + base64urlEncode( hashedData )

**How an application uses JWT to verify the authenticity of a user.**

