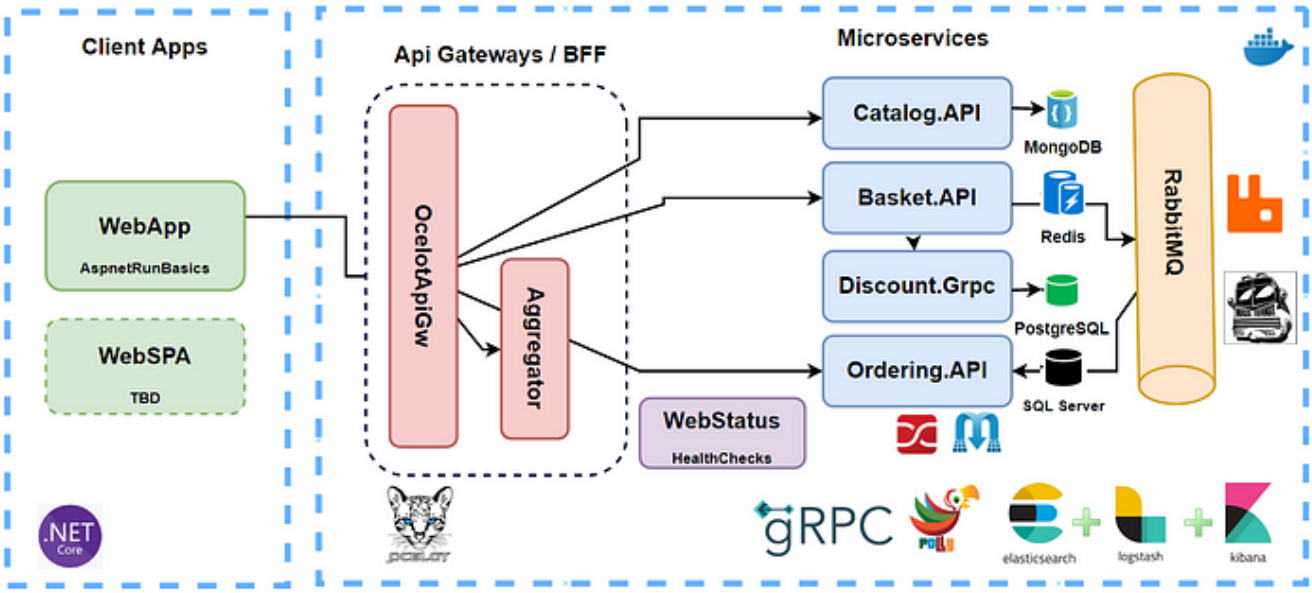
Microservices Architecture on .NET with applying CQRS, Clean Architecture and Event-Driven Communication

Building Microservices on .Net platforms which used Asp.Net Web API, Docker, RabbitMQ, gRPC, Ocelot API Gateway, MongoDB, Redis, PostgreSQL, SqlServer, Entity Framework Core, Dapper, CQRS and Clean Architecture implementation.

Introduction

In this article we will show **how to build microservices on .NET environments** with using **ASP.NET Core Web API** applications, **Docker** for containerize and orchestrator, Microservices **communications** with **gRPC** and **RabbitMQ** and using API Gateways with **Ocelot API Gateway**, and using different databases platforms **NoSQL(MongoDB, Redis)** and **Relational databases(PostgreSQL, SqlServer)** and using **Dapper, Entity Framework** Core for ORM Tools, and using best practices **CQRS** with **Clean Architecture** implementation.

Look at the big picture of final architecture of the system.



There is a couple of microservices which implemented **e-commerce modules** over **Catalog, Basket, Discount** and **Ordering** microservices with **NoSQL (MongoDB, Redis)** and **Relational databases (PostgreSQL, Sql Server)** with communicating over **gRPC** and **RabbitMQ Event Driven Communication** and using **Ocelot API Gateway.**

Get Udemy Course with discounted — Microservices Architecture and Implementation on .NET.

Along with this implemented below features over the **run-aspnetcore-microservices repository**;

Catalog microservice which includes;

- ASP.NET Core Web API application
- REST API principles, CRUD operations
- **MongoDB NoSQL database** connection on docker containerization
- N-Layer implementation with Repository Pattern
- Swagger Open API implementation
- Dockerfile and docker-compose implementation

Basket microservice which includes;

- ASP.NET Core Web API application
- REST API principles, CRUD operations
- **Redis database** connection on docker containerization
- **Consume Discount Grpc Service** for inter-service sync communication to calculate product final price
- **Publish BasketCheckout Queue** with using **MassTransit** and **RabbitMQ**
- Swagger Open API implementation
- Dockerfile and docker-compose implementation

Discount microservice which includes;

- ASP.NET **Grpc Server** application
- Build a Highly Performant **inter-service gRPC Communication** with Basket Microservice
- **Exposing Grpc Services** with creating **Protobuf messages**
- Using **Dapper** for **micro-orm** implementation to simplify data access and ensure high performance
- **PostgreSQL database** connection and containerization
- Dockerfile and docker-compose implementation

Microservices Communication

- Sync inter-service **gRPC Communication**
- **Async** Microservices Communication with **RabbitMQ Message-Broker Service**
- Using **RabbitMQ Publish/Subscribe Topic Exchange Model**
- Using **MassTransit** for abstraction over **RabbitMQ Message-Broker system**
- **Publishing BasketCheckout event queue** from **Basket** microservices and Subscribing this event from **Ordering** microservices
- Create **RabbitMQ EventBus.Messages Common Class Library** and add references Microservices

Ordering microservice which includes;

- ASP.NET Core Web API application
- Implementing **DDD, CQRS and Clean Architecture** with using **Best Practices**
- Developing **CQRS** with using **MediatR, FluentValidation** and **AutoMapper** nuget packages

- **Consuming RabbitMQ BasketCheckout** event queue with using **MassTransit-RabbitMQ Configuration**
- **SqlServer** database connection and containerization
- Using **Entity Framework Core** ORM and auto migrate to **SqlServer** when application Startup
- Swagger Open API implementation
- Dockerfile and docker-compose implementation

API Gateway Ocelot microservice which includes;

- Implement API Gateways with **Ocelot**
- Sample microservices/containers to **reroute through the API Gateways**
- Run multiple different **API Gateway/BFF** container types
- **The Gateway aggregation pattern** in Shopping.Aggregator
- Dockerfile and docker-compose implementation

WebUI ShoppingApp microservice which includes;

- ASP.NET Core Web Application with Bootstrap 4 and Razor template
- Call Ocelot APIs with HttpClientFactory and Polly
- Aspnet core razor tools — View Components, partial Views, Tag Helpers, Model Bindings and Validations, Razor Sections etc..
- Dockerfile and docker-compose implementation

Microservices Cross-Cutting Implementations

- Implementing **Centralized Distributed Logging with Elastic Stack (ELK)**; **Elasticsearch**, **Logstash**, **Kibana** and **SeriLog** for Microservices
- Use the **HealthChecks** feature in back-end ASP.NET microservices
- Using **Watchdog** in separate service that can watch health and load across services, and report health about the microservices by querying with the **HealthChecks**

Microservices Resilience Implementations

- Making Microservices more **resilient** Use **IHttpClientFactory** to implement resilient **HTTP requests**
- Implement **Retry** and **Circuit Breaker patterns** with exponential backoff with **IHttpClientFactory** and **Polly policies**

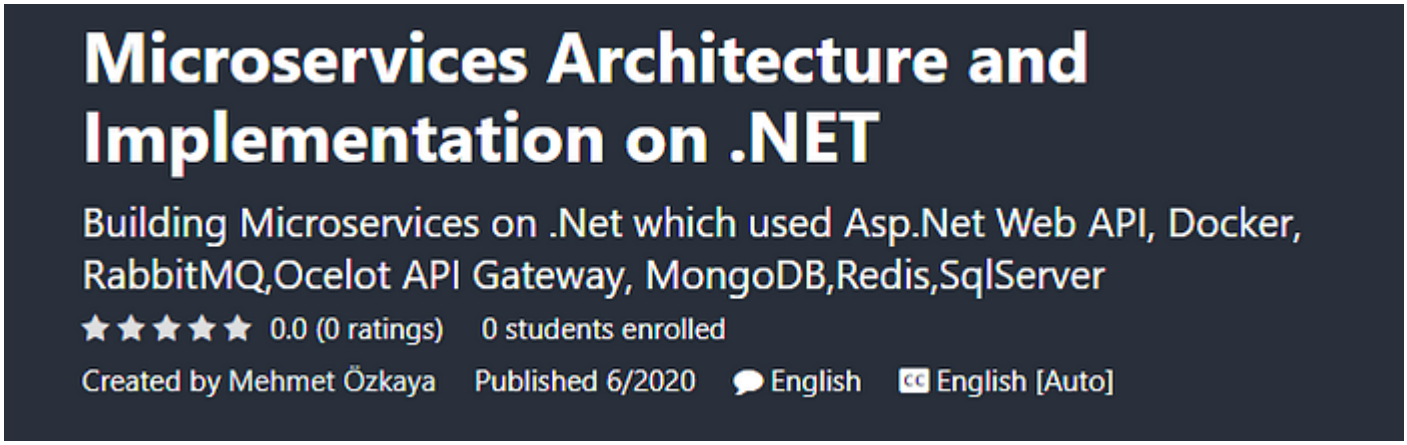
Ancillary Containers

- Use **Portainer** for Container lightweight management UI which allows you to easily manage your different Docker environments
- **pgAdmin PostgreSQL Tools** feature rich Open Source administration and development platform for **PostgreSQL**

Docker Compose establishment with all microservices on docker;

- Containerization of microservices
- Containerization of databases
- Override Environment variables

Udemy Course — Microservices Architecture and Step by Step Implementation on .NET



Get Udemy Course with discounted — Microservices Architecture and Implementation on .NET.

As of now, you'll finally be able to enroll in the course — where you'll learn to using .net core in microservices world, understand how can used **Asp.Net Web API, Docker, RabbitMQ, gRPC, Ocelot API Gateway, MongoDB, Redis, PostgreSQL, SqlServer, Entity Framework Core, CQRS and Clean Architecture implementation**, and develop **real-world applications** that make a difference!

Source Code

Get the Source Code from AspnetRun Microservices Github — Clone or fork this repository, if you like don't forget the star. If you find or ask anything you can directly open issue on repository.

Prerequisites

- Install the .NET 5 or above SDK
- Install Visual Studio 2019 v16.x or above
- Docker Desktop — **Memory: 4 GB**

Run Application

Follow these steps to get your development environment set up: (Before Run Start the Docker Desktop)

1. Clone the repository
2. At the root directory which include **docker-compose.yml** files, run below command:

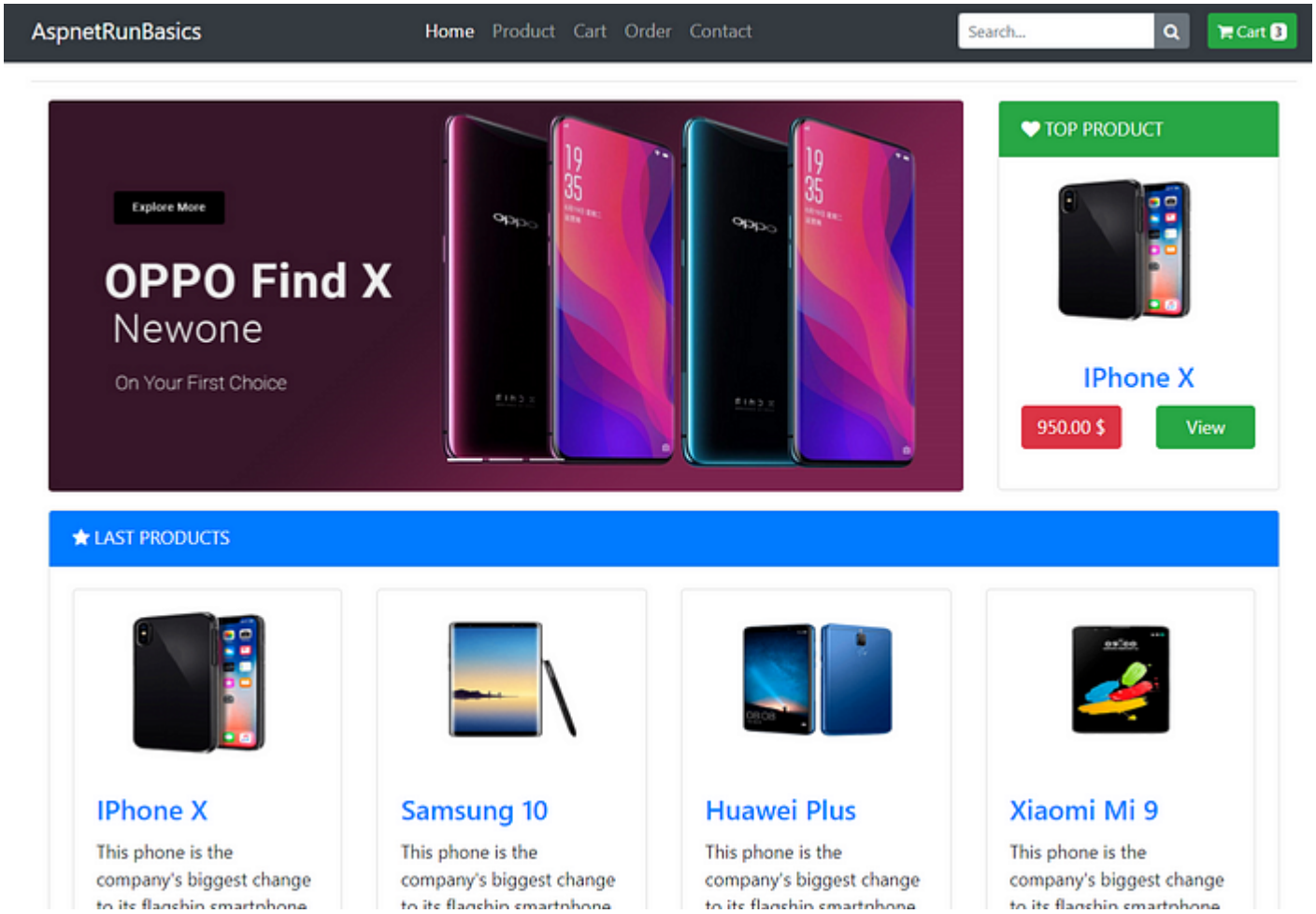
```
docker-compose -f docker-compose.yml -f docker-compose.override.yml up -d
```

3. Wait for docker compose all microservices. When you run this command for the first time, it can take more than 10 minutes to prepare all docker containers.

4. You can **launch microservices** as below urls:

Launch **http://host.docker.internal:8007** in your browser to view the **Web Status**. Make sure that every microservices are **healthy**.

Launch **http://host.docker.internal:8006** in your browser to view the **Web UI.** You can use Web project in order to call microservices over **API Gateway**. When you checkout the basket you can follow queue record on **RabbitMQ dashboard**.



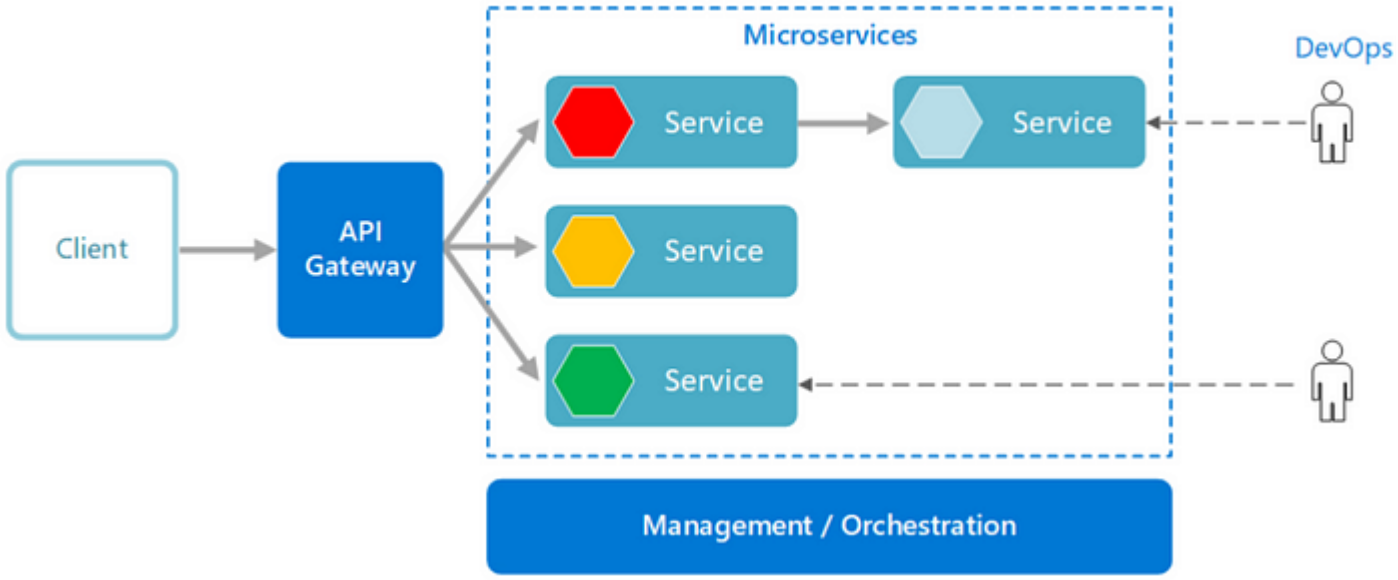
According to this design, we have a web application which basically implement **e-commerce domain**. This application has below functionalities;

- **Retrieving Products** and **Categories** and listing, filtering them
- **Add products to Basket** with applying quantity, color and calculating total basket price
- **Check out Basket** and **Create Order** with submitting the basket

So you can perform E2E test with following above use cases.

What are Microservices ?

Microservice are small business services that can work together and can be deployed autonomously / independently. These services communicate with each other by talking over the network and bring many advantages with them.

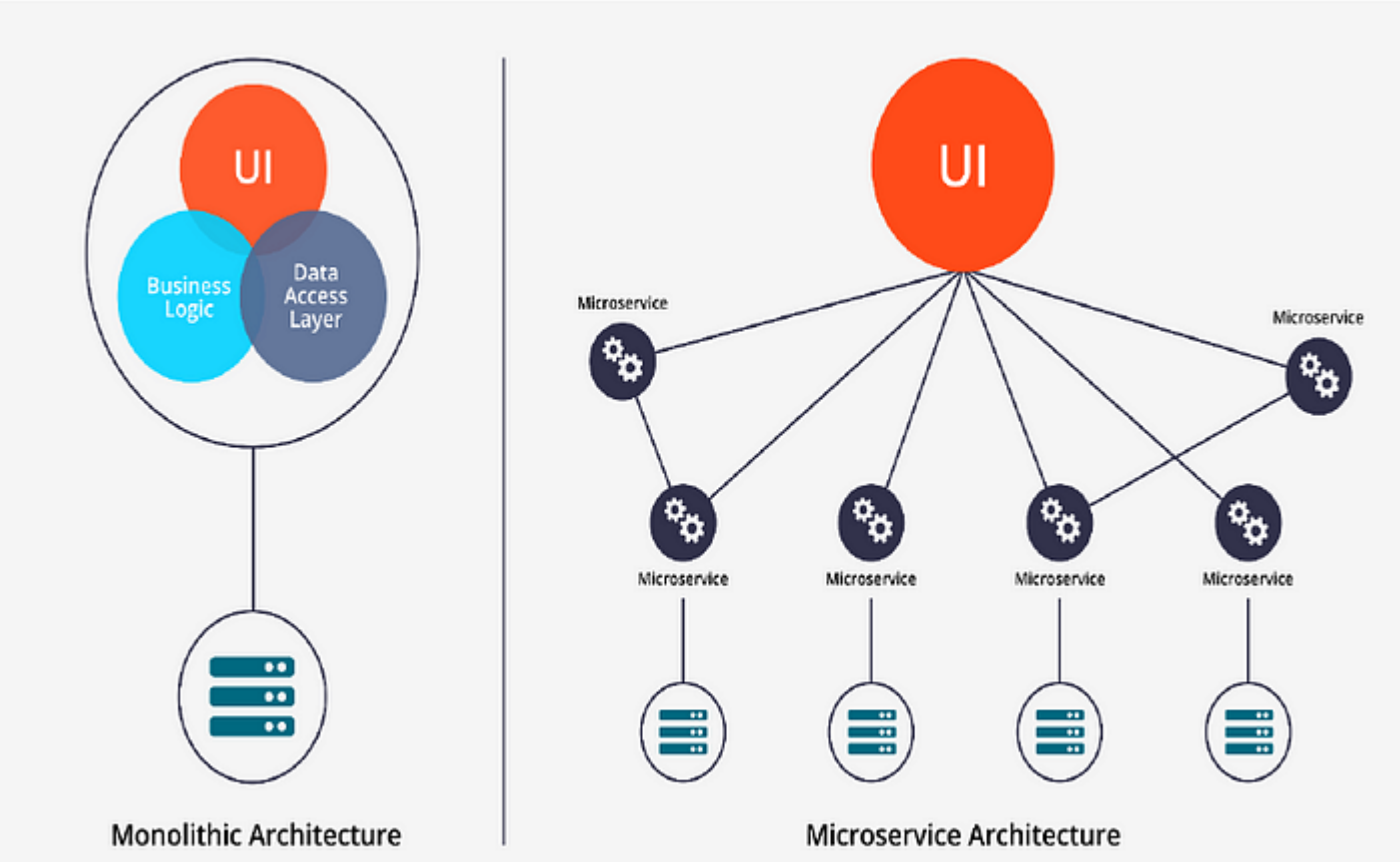


One of the biggest advantages is that they can be deployed independently.
However, it offers the opportunity to work with many different technologies (technology agnostic).

Microservices Characteristics

Microservices are small, independent, and loosely coupled. A single small team of developers can write and maintain a service. Each service is a separate codebase, which can be managed by a small development team.

Services can be deployed independently. A team can update an existing service without rebuilding and redeploying the entire application.

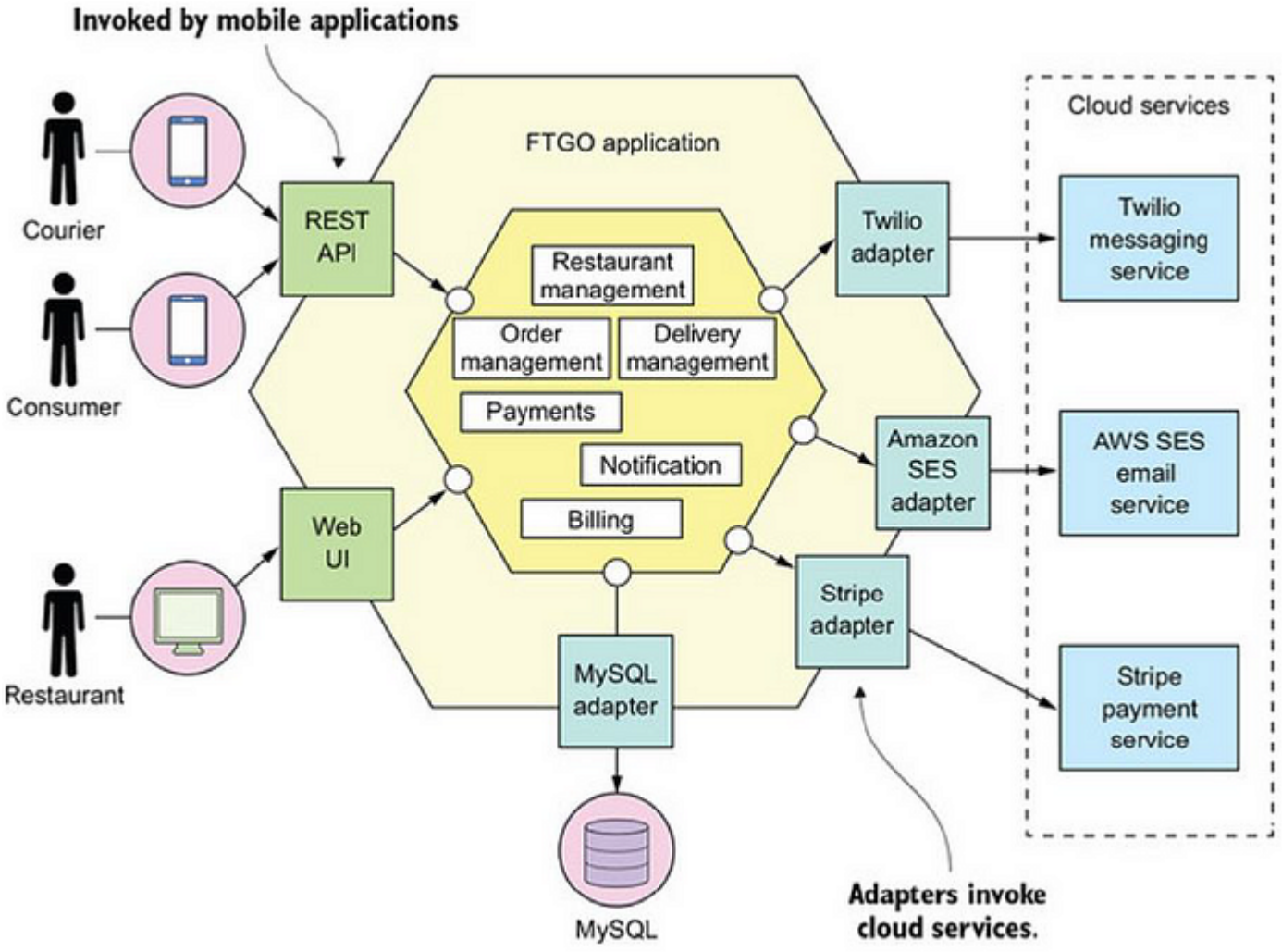


Services are responsible for persisting their own data or external state. This differs from the traditional model, where a separate data layer handles data persistence.

Services communicate with each other by using well-defined APIs. Internal implementation details of each service are hidden from other services. Services don't need to share the same technology stack, libraries, or frameworks.

Monolithic Architecture Pros-Cons

A monolithic application has a single codebase that contains multiple modules. Modules are divided according to their functional or technical characteristics. It has a single build system that builds the entire application. It also has a single executable or deployable file.



Strengths of Monolithic Architecture

Easier debugging and end-to-end testing: Unlike the Microservice architecture, monolithic applications are much easier to debug and test.
It is easy to develop and maintain for small projects. Application can be developed quickly. Easy to perform test operations, you can perform end-to-end tests much faster.
Easy deployment: An advantage associated with monolithic applications being a single piece is easy deployment. It is much easier to deploy a single part than to deploy dozens of services.
Also Transaction management is easy.

Weaknesses of Monolithic Architecture

Complexity: When a monolithic application grows, it becomes too complex to understand. As the application grows, it becomes difficult to develop new features and maintain existing code.

With the increase in the number of teams and employees working on the project, development and maintenance becomes more difficult. Because of their dependencies on each other, changes made in one functionality can affect other places.

The challenge of making changes: Making changes is very cumbersome in such a large and complex application. Any code change affects the entire system, so all parts must be thoroughly checked. This

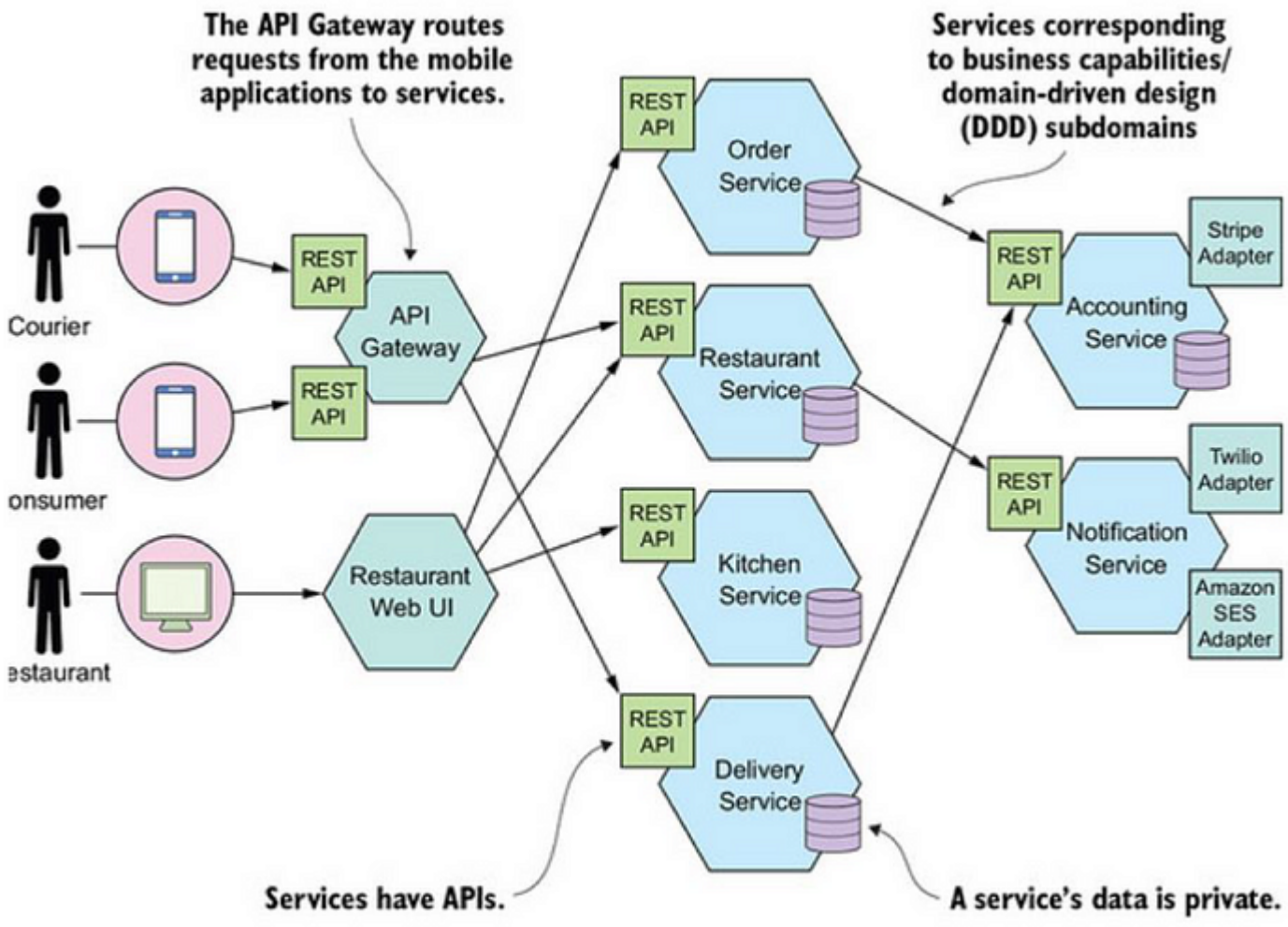
can make the overall development process much longer. Even with a small change in the application, the entire application must be deploy.

Inability to apply new technology: Implementing a new technology in a monolithic application is extremely problematic, The same programming language and same frameworks must be used. integrating the application into a new technology means redeveloping the whole application.

Low Saleable: You cannot scale components independently. You have to scale the entire application.

Microservices Architecture Pros-Cons

Microservice are small business services that can work together and can be deployed autonomously / independently.



Strengths of Microservice Architecture

Independent Services: Each service can be deployed and updated independently, providing more flexibility. An error in a microservic, only has an effect on a specific service and does not affect the entire application.
Also, adding new features to a microservice application is easier than a monolithic one.

Whether the application is very large or very small, adding new features and maintaining existing code is easy. Because it is sufficient to make changes only within the relevant service.

Better scalability: Each service can be scaled independently.
Therefore, the entire application does not need to be scaled. This saves a lot of time and money.
Additionally, every monolithic application has limits in terms of scalability. However, multiplexing a service with traffic on the microservice is less inconvenient and able to handle the whole load.
This is why most projects that appeal to a large user base, have begun to adopt the microservice architecture.

Technology Diversity: Teams do not have to completely choose the technologies on which the services will be developed,
they can choose the appropriate technology for the services they will develop over time. For example, a service can be developed with the python programming language in order to use “machine learning” features next to microservices developed on .Net. The desired technology or database can be used for each microservice.

Higher level of agility: Any errors in a microservice application only affect a particular service, not the entire application. Therefore, all changes and tests are carried out with lower risks and fewer errors.

Teams can work more efficiently and quickly. Folks who are just starting the project can easily adapt without getting lost in the code base.

Intelligibility: A microservice application broken down into smaller and simpler components is easier to understand and manage.
Since each service is independent from each other and only has its own business logic, the code base of the service will be quite simple. It is easier to understand and manage.

Weaknesses of Microservice Architecture

Microservice architecture should never be preferred for small-scale applications.

Extra extra complexity : Since a microservice architecture is a distributed system, you need to configure each module and databases separately. In addition, as long as such an application includes independent services, they must all be distributed independently.

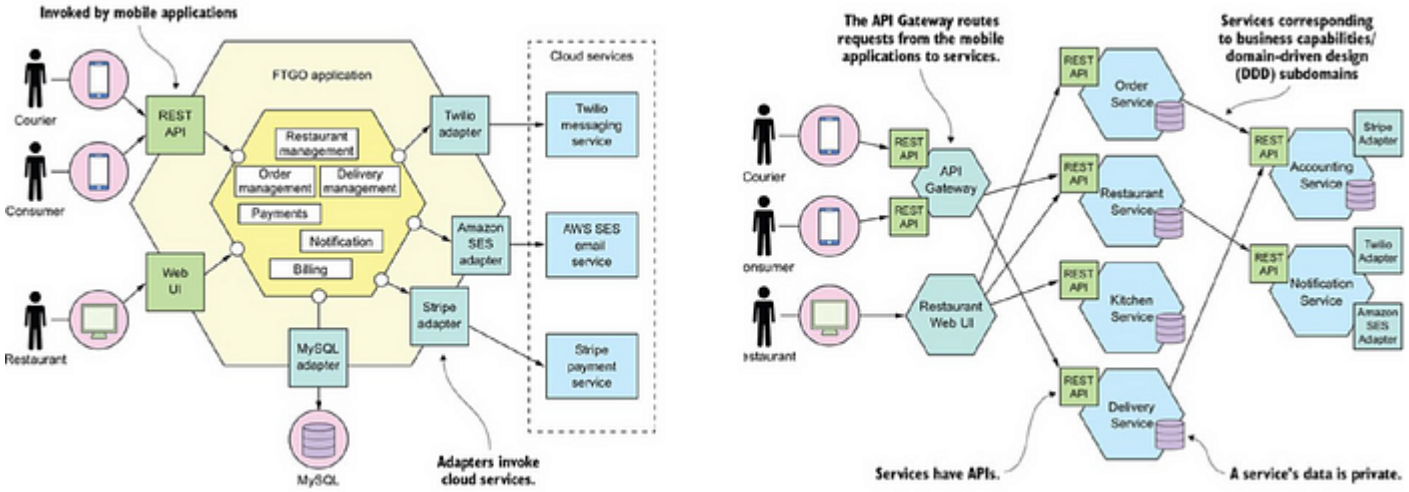
System distribution A microservice architecture is a complex system made up of multiple services and databases, so all connections need to be handled carefully. Deployment requires a separate process for each service.

The challenge of management and traceability: You will need to deal with multiple situations when creating a microservice application. It is necessary to be able to monitor external configuration, metrics, health-check mechanisms and environments where microservices can be managed.

The challenge of testing: The large number of services deployed independently of each other makes the testing process extremely difficult. Since there are more than one service and more than one database, transaction management will be difficult.

Monolithic vs Microservices Architecture Comparison

You can find monolithic architecture on left side, and on the right side is microservices architecture.



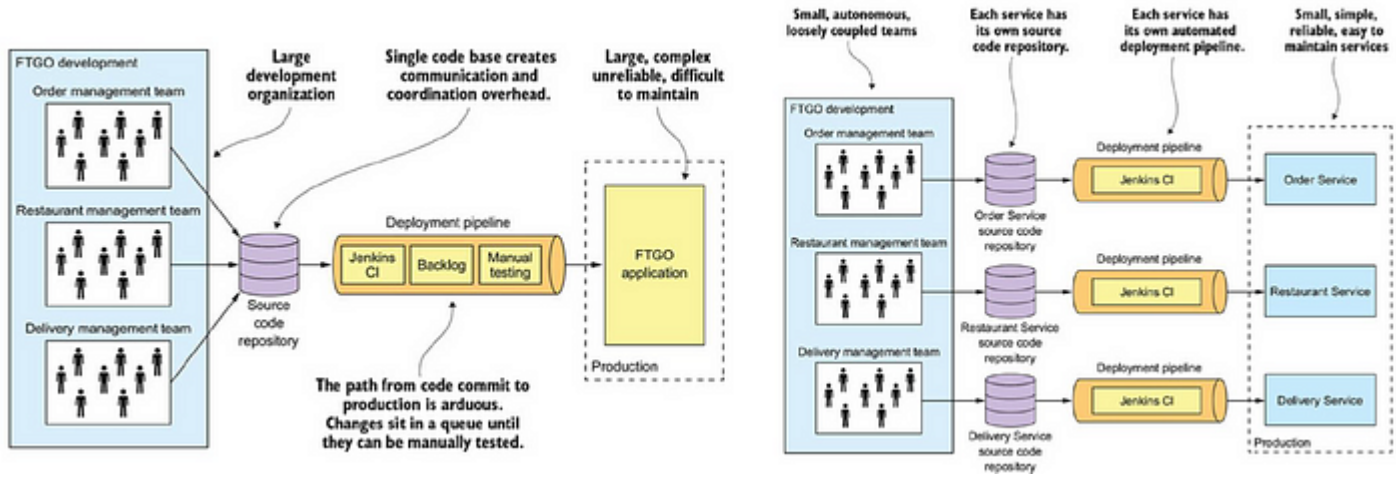
Monolithic application has a single codebase that contains multiple modules. Modules are divided according to their functional or technical characteristics. It has a single build system that builds the entire application. It also has a single executable or deployable file.

Microservices are small, independent, and loosely coupled.
A single small team of developers can write and maintain a service.
Each service is a separate codebase, which can be managed by a small development team. Services can be deployed independently. A team can update an existing service without rebuilding and redeploying the entire application.
Services are responsible for persisting their own data or external state.
This differs from the traditional model, where a separate data layer handles data persistence. Services communicate with each other by using well-defined APIs. Internal implementation details of each service are hidden from other services. Services don't need to share the same technology stack, libraries, or frameworks.

Deployment Comparison

A monolithic applications has large development organizations.
Single code base created communication overhead.

- The path from code commit to production is arduous.
- Changes sit in a queue until they can be manually tested.
- This architecture created -> Large complex unreliable difficult to maintain.

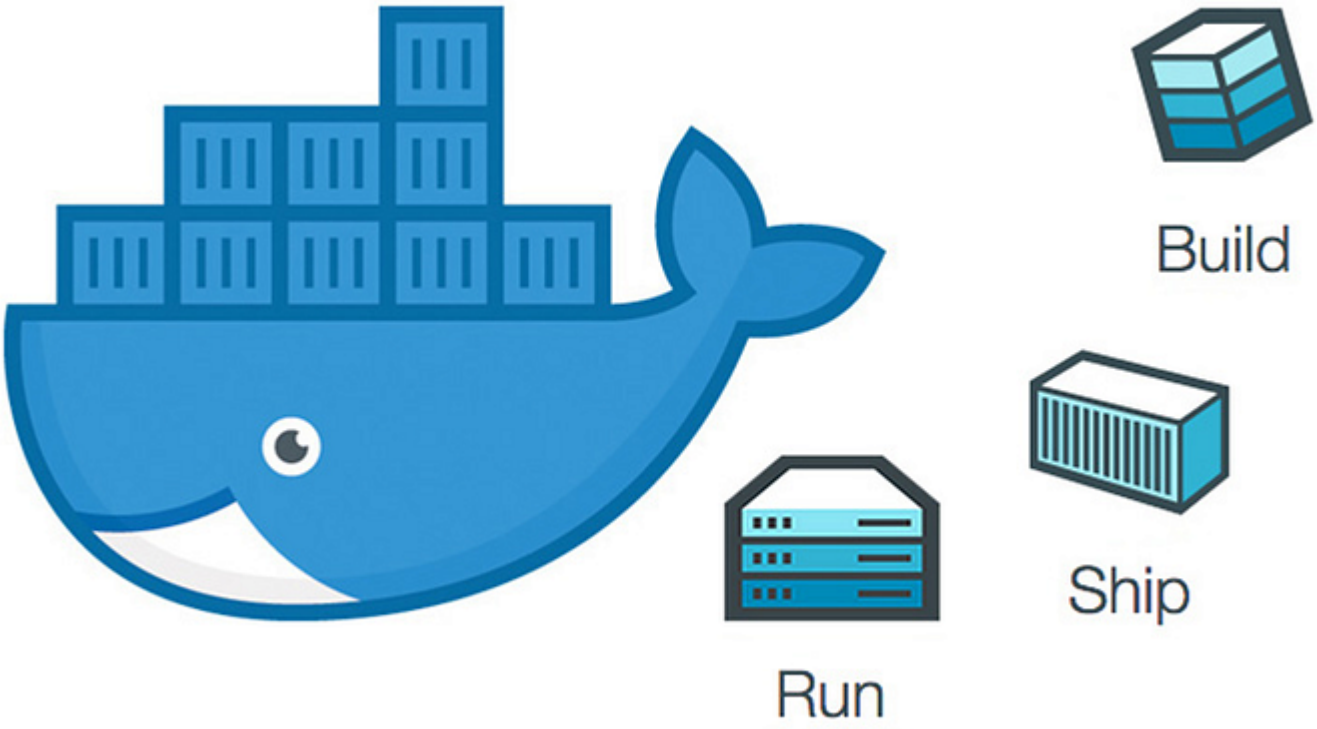


Microservices are;

- Small autonomous loosely coupled teams
- Each service has its own source code repository
- Each service has its own automated deployment pipelines.
  This architecture created -> Small simple reliable maintained services.

What is Docker and Container ?

Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly.



Advantages of Docker's methodologies for **shipping, testing,** and **deploying** code quickly, you can significantly reduce the delay between writing code and running it in production. Docker provides for automating the deployment of applications as portable, self-sufficient containers that can run on the

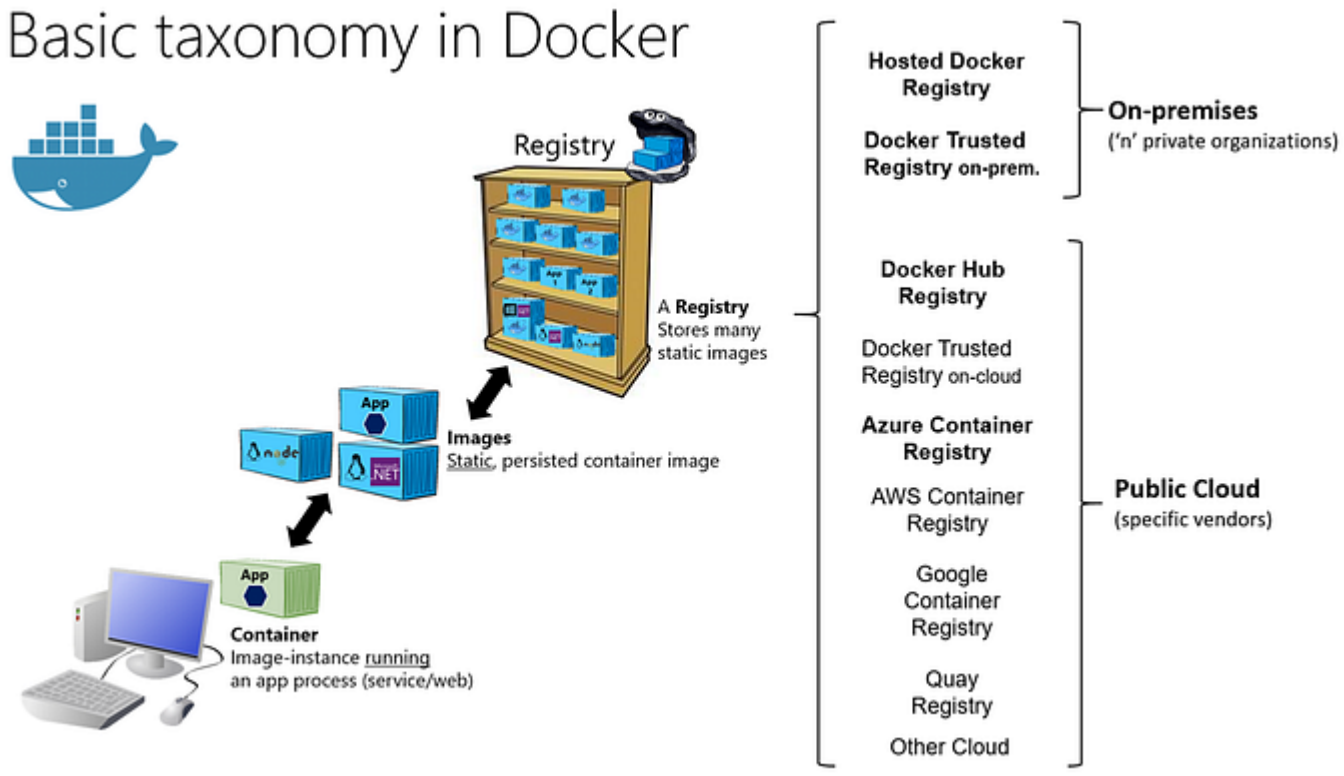
cloud or on-premises. Docker containers can run anywhere, in your local computer to the cloud. Docker image containers can run natively on Linux and Windows.

Docker Container

A container is a standard unit of software that packages up code and **all its dependencies** so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, **executable package** of software that includes everything needed to run an application.

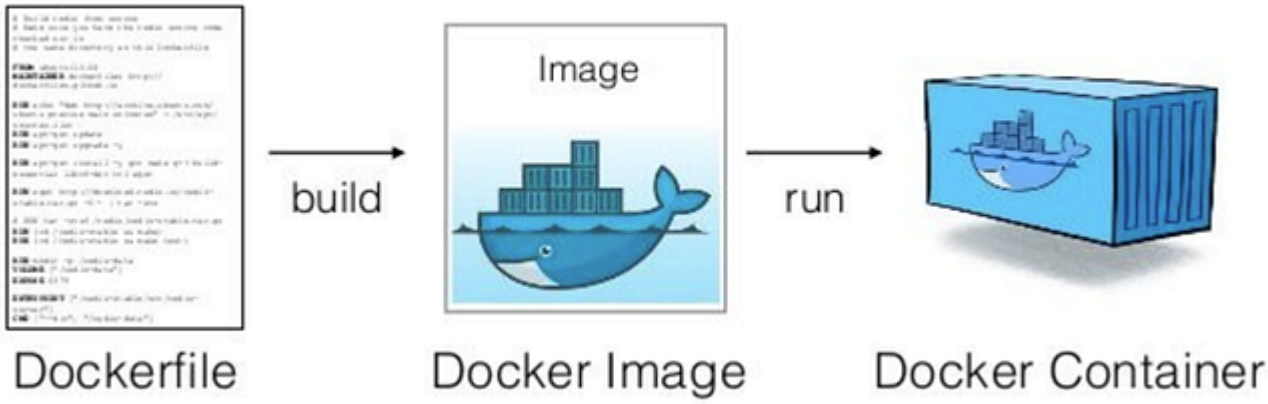
Docker containers, images, and registries

When using Docker, a developer develops an application and packages it with its dependencies into a container image. An image is a static representation of the application with its configuration and dependencies.



In order to run the application, the application's image is instantiated to create a container, which will be running on the Docker host.
Containers can be tested in a development local machines.

As you can see the images above, how docker components related each other.
Developer creates container in local and push the images the Docker Registry.
Or its possible that developer download existing image from registry and create container from image in local environment.



Developers should store images in a registry, which is a library of images and is needed when deploying to production orchestrators. Docker images are stores a public registry via Docker Hub; other vendors provide registries for different collections of images, including Azure Container Registry.
Alternatively, enterprises can have a private registry on-premises for their own Docker images.

Get Udemy Course with discounted — Microservices Architecture and Implementation on .NET.

Get the Source Code from AspnetRun Microservices Github