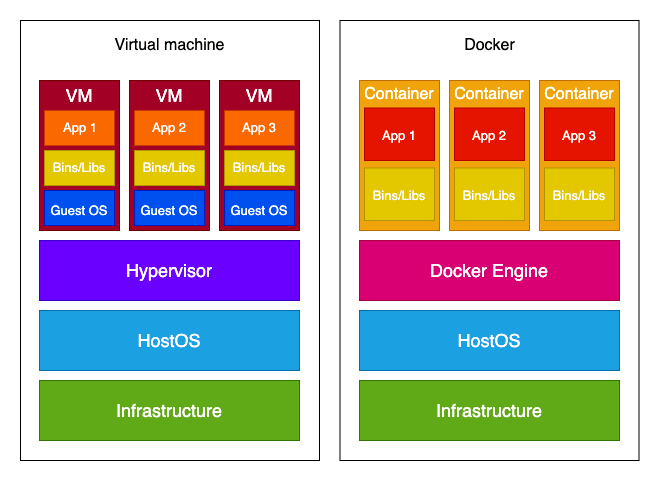
# Dockers:

# Containers and Virtual Machines:

Before jumping into Docker, it's important to understand the difference between containers and virtual machines.

Containers and virtual machines are similar in that they allow multiple apps to run on the same server with various software requirements -- e.g., different Python versions, different libraries, etc. Their main difference is in the operating system. While containers use the host's operating system, each virtual machine has its own guest operating system on top of the host's operating system.

In a now almost famous image, you can see how Docker compares to virtual machines:



So, if you have an application that needs to run on different operating systems, a virtual machine is the way to go. But if that's not a requirement, Docker has multiple advantages over a virtual machine:

* Light weight
* Faster to build
* Can be easily ported across different platforms
* Less resource intensive
* Scaling up and duplicating is easier
* All those advantages are due to Docker containers not needing their own operating system.

# Docker:

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. With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage 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 tooling and a platform to manage the lifecycle of your containers:**

* Develop your application and its supporting components using containers.
* The container becomes the unit for distributing and testing your application.
* When you’re ready, deploy your application into your production environment, as a container or an orchestrated service. This works the same whether your production environment is a local data center, a cloud provider, or a hybrid of the two.

## **Docker’s three key innovations:**

### **1. Easy app packaging**

Docker takes your application and all its software dependencies – minus the OS kernel and hardware drivers – and bundles them in a set of tarballs (similar to zip files). Docker ensures these tarballs are identical on every machine.

**The app tarball(s) and metadata are known together as a container image. This is now considered the modern way to package server apps.** The container image standard is a major improvement compared to previous packaging systems (like apt, yum, msi, npm, maven, etc.) that don’t contain application dependencies.

### **2. Easy app running**

Docker runs each container in an isolated file system. Each container gets its own networking and resource limits and can’t see anything else on the host operating system. It can do this in a single, short command.

### **3. Easy app distribution**

Docker created the idea of a “container image registry,” which allows you to store images in a central HTTP/S server and push/pull them as easily as doing so with git commits. The most popular registry is Docker Hub and you can find many open source projects with official images there.

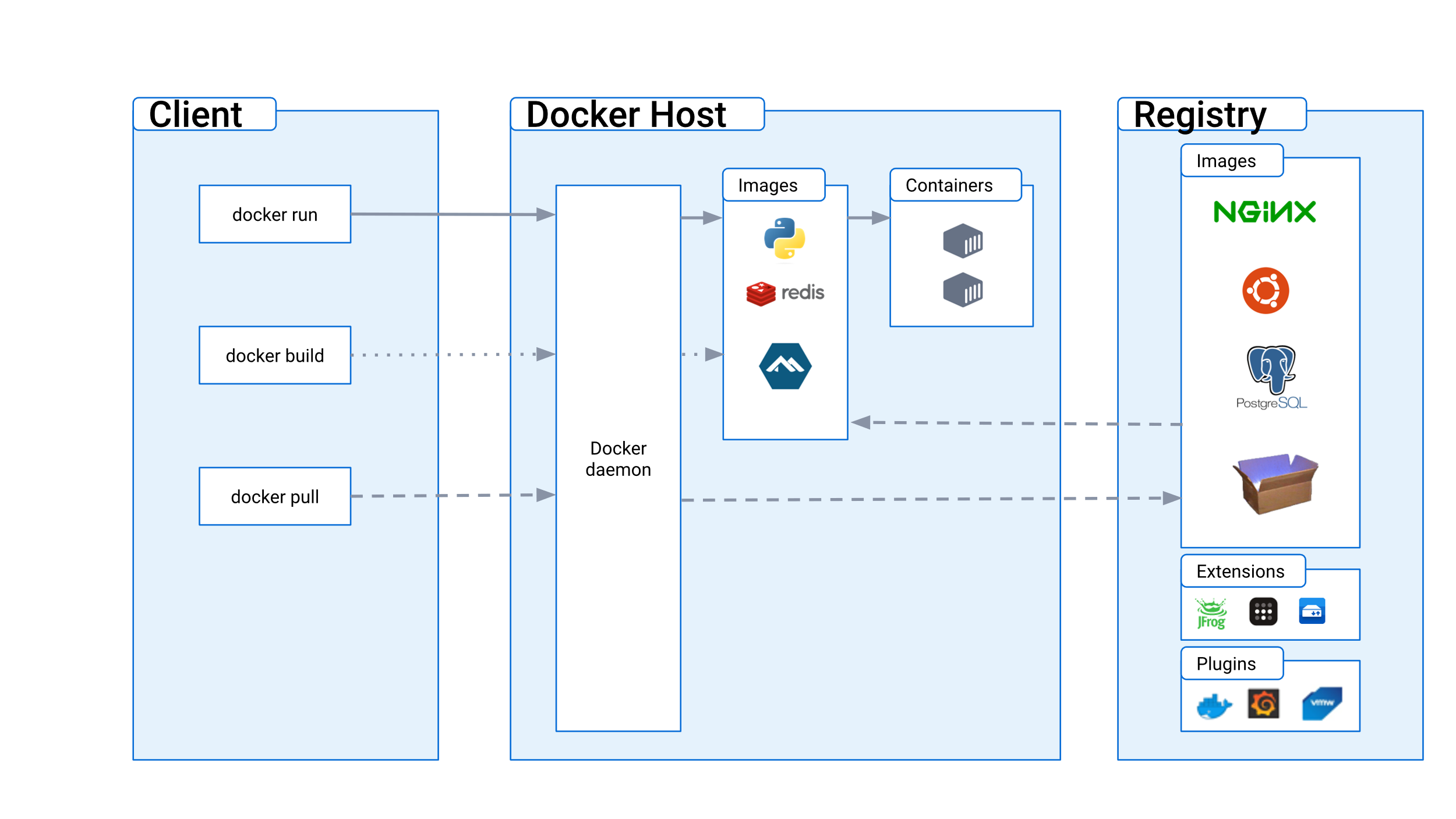
# Docker Engine

When people refer to Docker, they're typically referring to Docker Engine.

Docker Engine is the underlying open-source containerization technology for building, managing, and running containerized applications. It's a client-server application with the following components:

* **Docker daemon** (called dockerd) is a service that runs in the background that listens for Docker Engine API requests and manages Docker objects like images and containers.
* **Docker Engine API** is a RESTful API that's used to interact with Docker daemon.
* **Docker client** (called docker) is the command line interface used for interacting with Docker daemon. So, when you use a command like docker build, you're using Docker client, which in turn leverages Docker Engine API to communicate with Docker daemon.

**Docker Registry:**

A Docker registry stores Docker images. Docker Hub is a public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default. You can even run your own private registry.

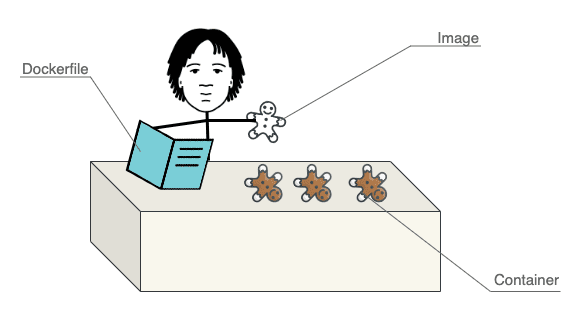
**Docker Desktop**

These days, when you try to install Docker, you'll come across Docker Desktop. While Docker Engine is included with Docker Desktop, it's important to understand that Docker Desktop is not the same as Docker Engine. Docker Desktop is an integrated development environment for Docker containers. It makes it much easier to get your operating system configured for working with Docker.

**Docker Concepts:**

At the heart of Docker, there are three core concepts:

* **Dockerfile** - a text file that serves as a blueprint for your container. In it, you define a list of instructions that Docker uses to build an image.
* **Image** - a read-only embodiment of the Dockerfile. It's built out of layers -- each layer corresponds to a single line of instructions from a Dockerfile.
* **Container** - running a Docker image produces a container, which is a controlled environment for your application. If we draw parallels with object-oriented programming, a container is an instance of a Docker image.



**Docker Engine gives us this great, easy-to-use command-line to do many things related to containerized applications, including:**

* Create container images with a built-in builder.
* Push and pull images from registries.
* Create and manage containers on a single machine.
* Create virtual networks for containers to communicate on the same host.
* Export/import images for offline transport.
* Run custom commands in existing containers.
* Centralize and access app log.

# Docker Use:

**Step 1: Install Prerequisites**

You need to install the docker desktop client based on your machine (PC) as per your operating system type. Use the following link to download the docker desktop client.

* [Docker for Windows](https://www.docker.com/docker-windows)
* [Docker for Mac](https://desktop.docker.com/mac/stable/Docker.dmg)
* [Docker Engine on Linux](https://docs.docker.com/engine/install/)

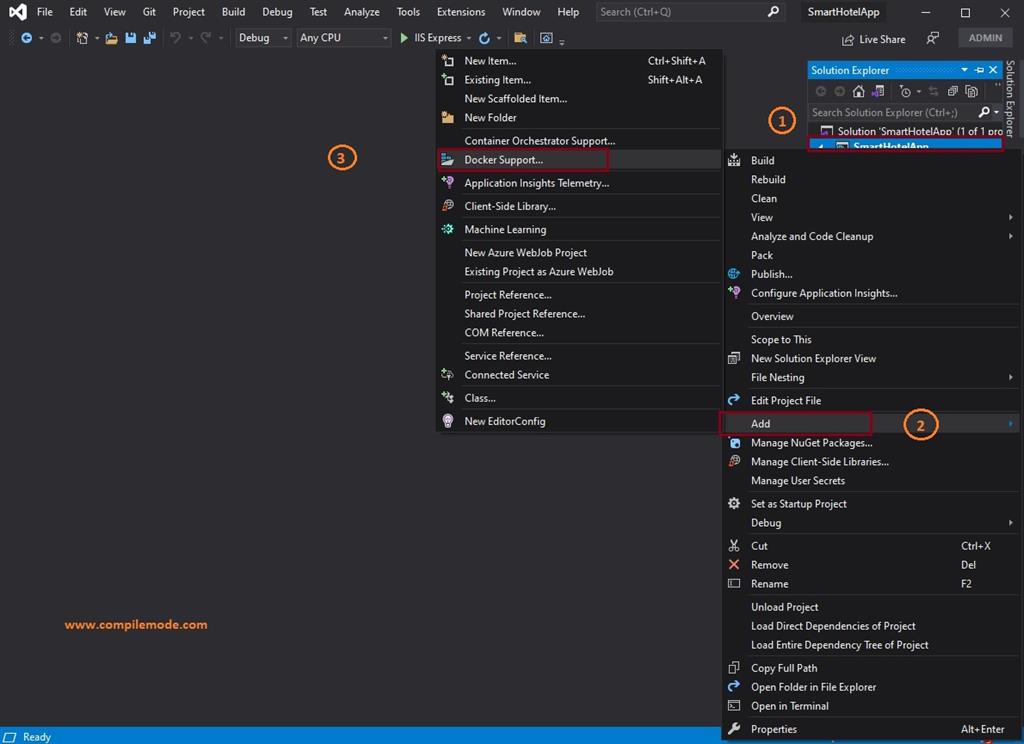
Before/after installing, open PowerShell as admin and update wsl (Windows Subsystem for Linux) kernel:

wsl --update

**Step 2: Add Docker Support to your project**

Docker support can be enabled in two ways from visual studio, at the time of creating the application and after creating the application.

Right click on the solution explorer of your existing project and follow the steps which are shown in the following image.



**Step 3: Choose the Container (Docker File)**

Choose the target operating system on which type of container you want to run the application, and it will create the docker image. Let's choose the Linux option which creates the image size smaller than compared to the windows container image.

Then, click on the OK button, it will create the Docker file in your project solution as shown in the following image.

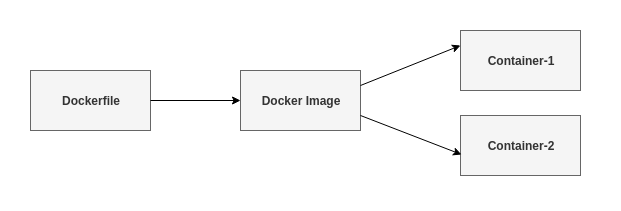
תמונה שמכילה טקסט, תוכנה, צילום מסך, תכונות מולטימדיה

התיאור נוצר באופן אוטומטי

**Step 4: Open the Docker File**

Double click on the docker file in which you will see auto generated code from visual studio which has pre-configured steps to create and run the docker image in multiple stages. The visual studio creates a docker file in your project which has multi-stages build features which make sure the final image remains smaller in size, which helps to become a container more efficient and faster. The container images were created in the following stages.

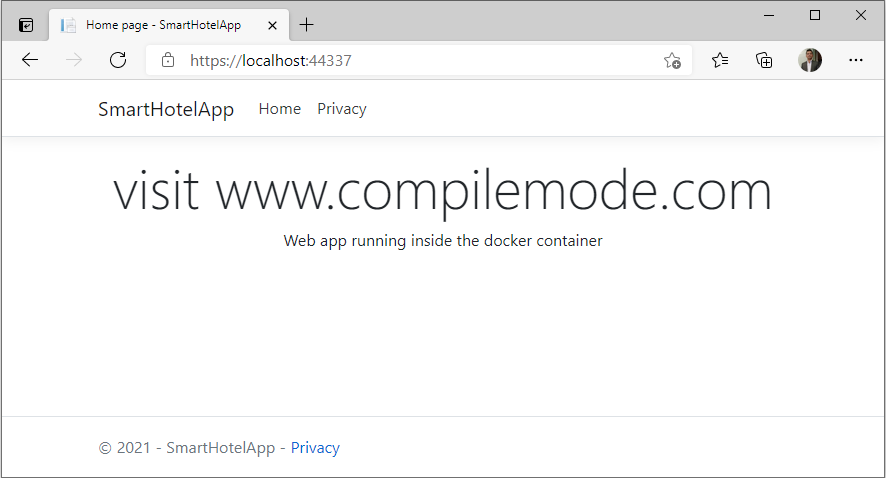
* Base
* Build
* Publish
* Final Image



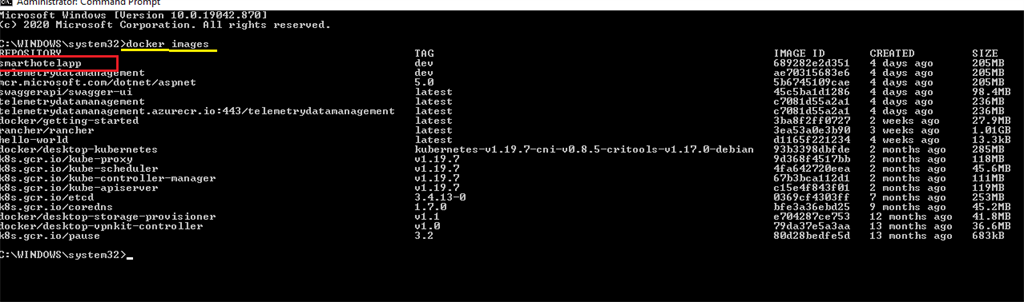
**Step 5: Run the image in Docker Container**

Choose the docker option to run the application as shown in the following image.

After clicking on the docker option, it will build code, create a docker image as well as a docker container and run the application inside the docker container without using the docker commands on the windows command prompt. The application opens the browser as follows:



An update for Visual studio might be required: https://visualstudio.microsoft.com/downloads/

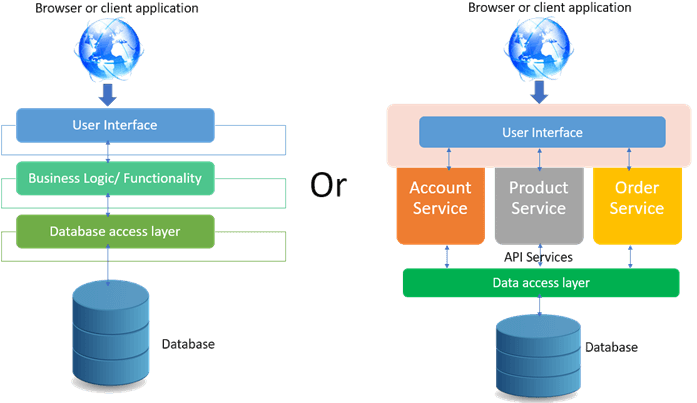
In the preceding, our application is running inside the Linux Docker container instead of IIS express. We can view the created images by using the docker command or navigating to the docker desktop dashboard. Let's view images using the docker command. Open your windows command prompt and use the following command on the command prompt and press enter.

Docker images

# **Micro Services:**

**What Are Microservices?**

A microservice system is an architectural style used by many large-scale software projects. A microservice architecture (MSA) is a collection of small components or services. The aim is to improve efficiency, reliability, performance, and scalability. Let’s compare microservices with the traditional approach, often called monolithic architecture.



**Monolithic architecture:**

A monolithic architecture is where people develop an entire system as a single unit. This is simple to manage and is an efficient strategy for small projects. The benefits of a monolithic architecture include that they are:

* **Easy to develop:** You can open the entire project in an integrated development environment (IDE) or even a text editor.
* **Easy to run:** You can hit the run button in your IDE and start testing.
* **Easy to deploy:** You can deploy the entire system by uploading a single file.

But all these advantages only apply to small systems.

As the system grows beyond a certain size, difficulties arise. And the more the system grows, the greater the difficulties. This can lead to a so-called “monolithic hell!”.

## **Problems might include the following:**

* **Deployment gets harder and riskier:** Imagine that you uncover a major bug. You need to fix it, now! So, you fix the bug. Now, the entire system needs redeployment! But right now, there are other features that are in development. So, the risk is that while you fix one bug, you introduce many others.

### **It becomes too big and too complicated:**

The monolith could become too big for a single developer to understand. Enforcing the boundaries between modules is too difficult. Over time, the modules become entangled in a mass of dependencies. a change to a module can break another module.

* **The monolith becomes difficult to scale.**

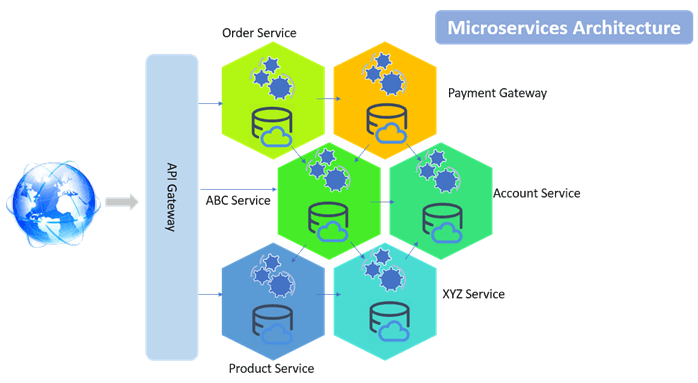
As the deployment unit grows, the only option is to increase the size of the target hardware. In a cloud environment, this means upgrading to a bigger virtual machine. This is called vertical scaling and is very inefficient because the costs do not scale linearly. For example, if you need a new machine with twice the memory, the cost will be much more than double. And, at some point, you will hit a hard limit of what is practical.

## **The alternative: a microservice architecture:**

With a microservices approach, we develop a collection of separate microservices. Each microservice would be:

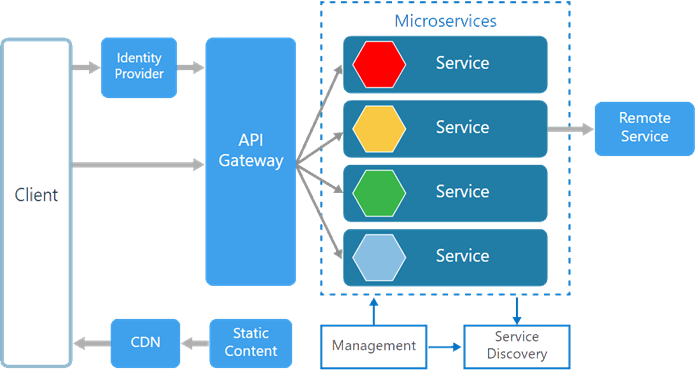
* Developed by a separate team.
* Hosted in its own repository (such as Git).
* Performing a useful business function.
* Independently deployable.
* Runnable standalone and doing something useful and testable.
* Able to collaborate with other microservices to fulfill a bigger goal.

## **Microservice architecture:**



There are various components in a microservices architecture apart from microservices themselves.

* **Management:** Maintains the nodes for the service.
* **Identity Provider:** Manages the identity information and provides authentication services within a distributed network.
* **Service Discovery**. Keeps track of services and service addresses and endpoints.
* **API Gateway:** Serves as client’s entry point. Single point of contact from the client which in turn returns responses from underlying microservices and sometimes an aggregated response from multiple underlying microservices.
* **CDN:** A content delivery network to serve static resources for e.g. pages and web content in a distributed network
* **Static Content:** The static resources like pages and web content



## **Building microservices:**

* **A microservice should be organized around business capabilities:**

For example, an e-commerce site isn’t just a “shopping cart.” It handles inventory, product catalogs, billing, customers, orders, and so on. These business functions are potential candidates for microservices. For example, inventory management sounds like a candidate microservice, if it fulfills the rest of the criteria for a microservice.

* **A single team will develop, deploy, and operate a microservice**

One team takes full ownership of a microservice. This ensures that the microservice can’t become too big. And each developer in the team should be able to understand the entire microservice.

* **A microservice should have high cohesion**

At the core of microservices is the same principle that is at the heart of any good software design. High cohesion means that a single microservice must do one thing and do that one thing well.

* **A microservice should have loose coupling**

The dependencies between the services should be as small as possible. Changing one microservice should have minimal impact on the other microservices.

* **Independent deployment of microservices**

Each microservice should be able to be redeployed at any time, with no impact on the rest of the system.

* **Automation of microservices**

You might be able to put up with the pain of manually deploying a monolith application, spinning up a few cloud instances to host it, or installing software onto those instances. But forget about scaling that up to 100 deployments or 1,000 instances. Microservices depend upon the automation of deployment, provisioning, and configuration management. Continuous integration or continuous delivery is a must with any microservice project.

* **Microservices are often distributed systems**

You could deploy all your microservices to a single server. Or, in an extreme case, you could have a single server for each microservice. This would be a very effective way of enforcing separation! Of course, for most projects, this would be very expensive.

The best solution is to use Docker to turn each microservice into a container, and then use an orchestration tool to manage the running containers. The most common orchestration tool is Kubernetes

## **Advantages of a microservice architecture:**

* You enforce good modular principles, cohesion, and loose coupling.
* You can deploy and redeploy individual services at any time. Changes and new features can be introduced quickly.
* Each microservice can be written in any language you choose. This is called polyglot programming. For example, a financial microservice might be best written in Python. On the other hand, you might choose C for a graphics processing service that needs high performance.
* If a microservice becomes a problem, it can be thrown away and redeveloped from scratch.
* As the microservices can be distributed, it is easy to scale the system horizontally. That means the microservices can be divided among many machines. This is much cheaper than scaling vertically (buying a more expensive machine).

## **Disadvantages of a microservice architecture:**

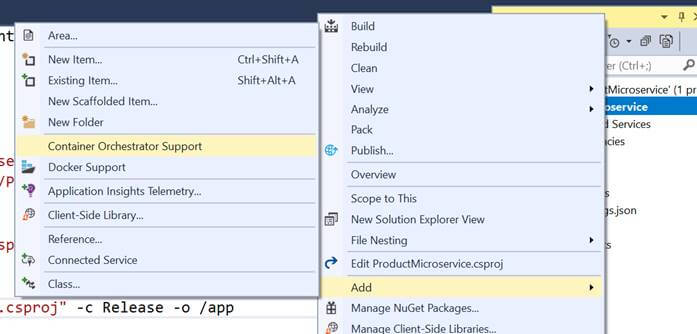
* The principles are simple but putting them into practice is hard. For example, deciding on what the microservices are, is subjective.
* A lot of tooling must be in place: automation, continuous delivery, source control, etc.
* It can be difficult to run and test a microservice in isolation.

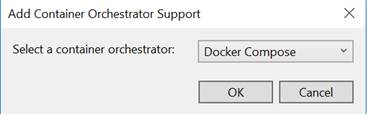
## Docker-Compose Orchestrator:

The Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application’s services. Then, with a single command, you create and start all the services from your configuration.

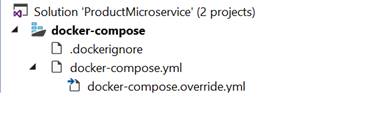
## Microservice using ASP.NET Core:

Running the service could be done via docker commands to be run in docker command prompt and using visual studio as well. Since we added the docker support, it is easy to run the service in docker container using visual studio.

1. VS uses Docker Compose Orchestrator. Add container orchestrator support in the solution as shown below:
2. This will ask for the orchestrator. Select Docker Compose and press OK.



1. Once added to the solution, the solution will look like shown below having docker-compose with dockerignore and docker-compose.yml and its override file.



As soon as the solution is saved, it builds the project under the container and creates a docker image. All the commands execution can be seen in the output window when the solution is saved.

In order to run multiple container instances:

subscriber: &subscriber

image: ${DOCKER\_REGISTRY-}subscriber

build:

context: .

dockerfile: Subscriber/Dockerfile

depends\_on:

- "productswithrabbit"

- "rabbitmq"

subscriber1:

image: ${DOCKER\_REGISTRY-}subscriber

build:

context: .

dockerfile: Subscriber/Dockerfile

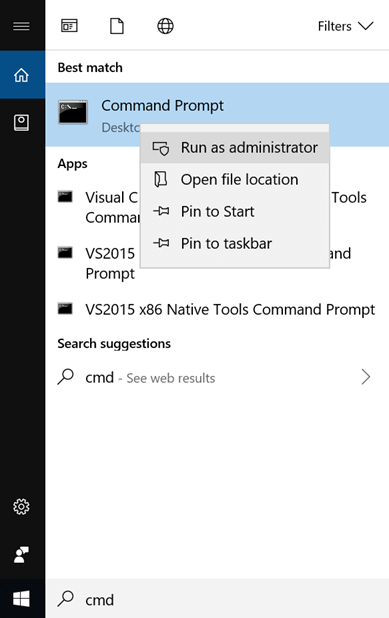
depends\_on:

- "productswithrabbit"

- "rabbitmq"

subscriber2:

<<: \*subscriber

1. Open the command prompt in admin mode and navigate to the same folder where the project files are.
2. Run the command **docker images** to see all the created images. We see the ProductMicroserviceimage the latest one.
3. Now run the application with Docker as an option as shown below.
4. Now, run the command **docker ps** to see the running containers. It shows the container is running on 32773:80 port.

Microservice Using ASP.NET Core