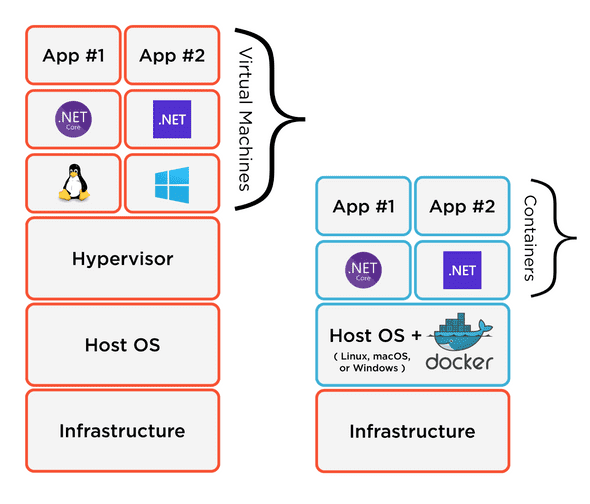
**Common Docker Terminology**

Common terms and phrases .NET developers will encounter when working with Docker.

The containerization world is no different from any technology, with its fair share of concepts and terminology. This section will explore **Docker**, how it relates to containerization, and other vital terms we might encounter on our journey into this ever-expanding application development niche.

**Containerization**

Let's start with the concept of containerization. For simplicity's sake, we can think of containers as a variation on virtual machines. Containers allow for an environment that can host our applications. While similar conceptually, containers differ from traditional virtual machines in that containers rely on the host operating system and an execution engine layer rather than having a **Hypervisor** manage isolated operating system (OS) instances.

[](https://www.jetbrains.com/dotnet/guide/static/b29889563d311a1d2f1f1062a3aefcae/913a1/docker-hosting-diagram.png)

The hosting model is a critical distinction that should be taken into account when deciding between VMs and containers.

What advantages can .NET developers hope to see when choosing containerization?

* Better resource utilization because resources are shared.
* Faster deployment and startup times because the host OS is already running.
* Simplified management of container hosts. One host OS to manage, rather than individual guest OSes.

**As we mentioned, and it's essential knowledge when thinking about containers, a container will depend on the host OS.** That means we cannot host a macOS container on a Windows host or a Windows container on a Linux host. Linux hosts and containers are ubiquitous for containerization as Linux can run on all major operating systems either natively or within a virtual machine.

There are some compromises to containerization that .NET developers should keep in mind:

* Containers may be able to interact with the Host OS, which might allow for security exposure.
* Windows dependencies require Windows-based containers, which are large, in the size range of gigabytes, compared to Linux containers having a size range of megabytes.

**Docker**

Docker has become synonymous with containerization, and many developers will use the two terms interchangeably. Docker is a toolset to make it easier for developers to start containerizing applications. Here is how Docker, the company, pitches its tools to developers.

Docker takes away repetitive, mundane configuration tasks and is used throughout the development lifecycle for fast, easy and portable application development - desktop and cloud. Docker's comprehensive end to end platform includes UIs, CLIs, APIs and security that are engineered to work together across the entire application delivery lifecycle. --[Docker](https://docker.com/)

.NET developers can think of Docker in much the same way they think about the .NET Ecosystem's toolsets and languages. In comparison, an experienced .NET developer might tell a newcomer enthusiastically to **"learn .NET!"**When what they might be recommending is to explore languages like C# or F#, build an ASP.NET application, or do some Xamarin mobile development.

For beginners, the first step of using Docker is installing [**Docker Desktop**](https://www.docker.com/get-started). The Docker Desktop product will configure an environment ready to host containerized applications. All OSes will get a Linux virtual machine and the native ability to run host-specific containers. Windows users will be able to run Windows containers, and macOS users can run macOS containers.

Docker offers a wide range of features, which is out of scope for this tutorial. If you are interested in learning Docker tooling, we recommend visiting their [**Getting Started**](https://www.docker.com/get-started) guide and then continuing with the tutorial.

At the very least, developers should install **Docker Desktop** on their development environments ([macOS](https://desktop.docker.com/mac/stable/Docker.dmg) and [Windows](https://desktop.docker.com/win/stable/Docker%20Desktop%20Installer.exe)), as we'll be using it later in the tutorial.

**Images**

Containerizing our applications first starts with choosing a base image. An image is the foundation of what we will choose to build and run our application.

An image is a read-only template with instructions for creating a Docker container. Often, an image is based on another image, with some additional customization. --[Docker](https://docs.docker.com/get-started/overview/)

Most .NET developers will start with [Microsoft's images](https://hub.docker.com/_/microsoft-dotnet/), but we can also build custom images and store them in an image registry. Base images include:

* [dotnet/sdk](https://hub.docker.com/_/microsoft-dotnet-sdk/): .NET SDK
* [dotnet/aspnet](https://hub.docker.com/_/microsoft-dotnet-aspnet/): ASP.NET Core Runtime
* [dotnet/runtime](https://hub.docker.com/_/microsoft-dotnet-runtime/): .NET Runtime
* [dotnet/runtime-deps](https://hub.docker.com/_/microsoft-dotnet-runtime-deps/): .NET Runtime Dependencies

Picking a base image can have a substantial impact on deployment, along with resource utilization in a production environment. Dependencies of our .NET applications may constrain us to specific host environments and base images. Luckily for .NET developers utilizing .NET Core and .NET 5+ runtimes, we can use Linux base images that reduce our eventual container's size. Newly created images will only contain the additional data built on top of the base image, typically our application and its footprint.

**Containers**

Containers are instances of images running on the execution engine and host OS. For .NET developers, an image is analogous to an executable on Windows or an App in macOS. A container is similar to the running process of that executable or App.

When working with containers, we must configure our instance to expose ports, attach volumes, and specify entry points. We'll see how this is done later in the tutorial.

**Ports**

Containers can be self-contained and shut-off from all external connections, but that wouldn't be very helpful. Like a physical machine, containers can communicate over ports, which are defined when creating the container. For example, a SQL Server database container might want to expose the port 1433, which is typically the port for remote database connections. For an ASP.NET container, we might want the HTTP port, the HTTPS port, or both to be accessible.

**Volumes**

Disposability is one of the greatest strengths of containers. That said, when a container is stopped and deleted, the contents of the container are gone. If our image writes to disk, there could be a legitimate concern for data loss. Have no fear! We can mount durable storage to any container, also known as [**volumes**](https://docs.docker.com/storage/volumes/).

Volumes are mounted when a container is started and persist on the host operating system, allowing us to remove one container instance and create a new container instance attached to the same volume. We can manage volumes using the Docker CLI tooling.

We can read more about volumes on the official [Docker documentation site](https://docs.docker.com/storage/volumes/).

**Multi-Container Applications**

When working with a distributed application, developers will want to create a logical application from different containers. We may have our core application talking to a database container, a caching service, and a full-text search engine. Docker ships with a tool called [**Compose**](https://docs.docker.com/compose/), which allows us to define a relationship topology between different containers.

The Compose tool has many benefits that should be immediately apparent to .NET developers:

* Codify development environments and dependencies like databases, caches, queues, and more. Getting started has never been more straightforward.
* A repeatable test environment for quality assurance teams and continuous integration pipelines.
* Provides a blueprint for production environments while allowing us to scale each container service independently.

One drawback to using the **Compose** tool is its focus on single-host deployments. Developers can use supplemental tools like **[Kompose](https://kompose.io/)** to translate Docker compose definition files to target a Kubernetes cluster.

For this tutorial, our focus is on Docker's ability to help during the .NET development process since there are other great resources for deploying containerized applications already written by others.

**Container Image Registries**

A benefit of creating images is sharing them amongst team members and environments. We can share our images by pushing them from our local development environments, and continuous integration build pipelines to a container image registry. The most notable image registry is [Docker Hub](https://hub.docker.com/), which holds most of the community's shared images and offers private registries for commercial use-cases.