**Bind Mounting**

**What is Bind Mounting?**

* A **Bind Mount** in Docker means linking a file or folder on your **host machine** (your laptop/PC) to a file or folder inside the **container**.
* Changes on the host are **immediately reflected inside the container**, and vice versa.

Think of it as **sharing a folder** between your computer and the container.

**Why Bind Mount Config Files?**

In many applications, configs differ between **environments** (dev, test, prod). Instead of baking configs into your Docker image:

1. You keep configs **outside the image** (on the host or config repo).
2. Use **bind mounts** to inject them when running the container.

👉 Benefits:

* No need to rebuild the image if config changes.
* Easy to manage environment-specific settings.
* Keeps image **clean & portable**, while configs stay **flexible**.

**Purpose in Real Projects**

* **Developers** can mount a dev config without touching production configs.
* **Ops teams** can mount secure configs stored in a safe location.
* **CI/CD pipelines** can mount configs depending on environment (staging vs prod).

Bind mounting config files = **separating configs from code** so you can run the same image in multiple environments with different settings.

**🔎 Why Binding Mounts are Useful**

1. **No rebuild for config changes**
   * Imagine your app is deployed in Docker.
   * Tomorrow, DB password changes. If your config is baked inside the image (appsettings.json only), you must rebuild + redeploy the image just to update a password.
   * With a **bind mount**, you can just drop in a new appsettings.Docker.json on the host, restart the container → ✅ new config is live, no rebuild.
2. **Environment-specific configs**
   * You can keep separate configs like:
     + appsettings.Development.json → your laptop
     + appsettings.Staging.json → test server
     + appsettings.Docker.json → inside container
     + appsettings.Production.json → live environment
   * With Bind Mounts, you don’t hardcode these into your image. Instead, you **mount only the environment’s config** at runtime.

Example:

volumes:

- ./configs/appsettings.Production.json:/app/appsettings.Docker.json

→ Same container image runs everywhere, but picks up different configs depending on what you mount.

1. **Security & Secrets**
   * You don’t want DB passwords, API keys, or tokens sitting in your source code (which gets baked into the Docker image).
   * With Bind Mount, secrets can live outside the image (in host folder, Azure Key Vault, AWS Secrets Manager, etc.) and be mounted only at runtime.
   * If an image leaks, your secrets are still safe.
2. **Operational flexibility**
   * Ops team (or DevOps) may need to tweak logs, feature flags, connection strings without touching your code.
   * They just edit the mounted config file on the server, restart container → changes applied.

**🔎 Why multiple appsettings.\*.json versions?**

* It’s **layered configuration** → a hierarchy of overrides.
* Think of it like clothes:
  + appsettings.json → default base layer (always required).
  + appsettings.Docker.json → extra jacket if running in Docker.
  + appsettings.Production.json → heavy coat if running in production.
  + Environment Variables → final “accessory” that beats everything else.

👉 This layering means **you don’t duplicate everything**.

Example:

// appsettings.json

{

"Logging": { "Level": "Debug" },

"ConnectionStrings": { "Default": "Server=localhost;..." }

}

// appsettings.Docker.json

{

"ConnectionStrings": { "Default": "Server=sqlserver;..." }

}

Notice: only the ConnectionStrings section changes in Docker. Logging stays as-is from base file.

✅ **Purpose in one line:**  
Binding mounts + multiple appsettings.\*.json files let you **reuse one Docker image across multiple environments, apply changes without rebuilding, and keep secrets/config outside your codebase.**

### 🔹 ****Bind Mounts****

* **Purpose**: Quick & dirty way to share files between your host machine and container.
* **Where used**:
  + Development (e.g., editing code on your laptop → container sees changes instantly).
  + Debugging (temporarily overriding config or injecting logs).
* **Why not in production**:
  + Inconsistent behavior across OS/hosts.
  + Security risks (container can mess with host filesystem).
  + Hard to replicate in clusters (like Kubernetes).

Bottom of Form

**Step 1**: Create a Web API Project by name MicroServicerContainerBindMountConfigFiles

**Step 2:** Create **appSettings.Docker.json** file in the Project folder of MicroServicerContainerBindMountConfigFiles

{

"AppSettings": {

"Message": "Hello from Bind Mount!"

}

}

**Step 3**: Add a message Controller

using Microsoft.AspNetCore.Mvc;

using Microsoft.Extensions.Configuration;

namespace MicroServicerContainerBindMountConfigFiles.Controllers

{

[ApiController]

[Route("api/[controller]")]

public class MessageController : ControllerBase

{

private readonly IConfiguration \_configuration;

public MessageController(IConfiguration configuration)

{

\_configuration = configuration;

}

[HttpGet]

public IActionResult Get()

{

var message = \_configuration["AppSettings:Message"] ?? "No message found";

return Ok(new { Message = message });

}

}

}

**Step 4**: Update Program.cs

namespace MicroServicerContainerBindMountConfigFiles

{

public class Program

{

public static void Main(string[] args)

{

var builder = WebApplication.CreateBuilder(args);

builder.Configuration

.SetBasePath(Directory.GetCurrentDirectory())

.AddJsonFile("appsettings.json", optional: false, reloadOnChange: true)

.AddJsonFile("appsettings.Docker.json", optional: true, reloadOnChange: true)

.AddEnvironmentVariables();

// Add services to the container.

builder.Services.AddControllers();

// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

var app = builder.Build();

// Configure the HTTP request pipeline.

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI();

}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();

}

}

}

**Step 5**: Create Dockerfile in MicroServicerContainerBindMountConfigFiles projects folder

FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS base

WORKDIR /app

EXPOSE 80

FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build

WORKDIR /src

COPY ["MicroServicerContainerBindMountConfigFiles.csproj", "./"]

RUN dotnet restore "MicroServicerContainerBindMountConfigFiles.csproj"

COPY . .

RUN dotnet build "MicroServicerContainerBindMountConfigFiles.csproj" -c Release -o /app/build

# Note: Why /p:UseAppHost=false is used in the below command?

# Without the property UseAppHost=false (/p denotes property in Msbuild),

# this will generate an executable host (called the app host) for the project

# This will generate an .exe file <ProjectName>.exe

# Since Docker containers always run under a known runtime (dotnet command inside the base image),

# the .exe isn’t necessary and is skipped to save space and avoid cross-platform issues.

FROM build AS publish

RUN dotnet publish "MicroServicerContainerBindMountConfigFiles.csproj" -c Release -o /app/publish /p:UseAppHost=false

FROM base AS final

WORKDIR /app

COPY --from=publish /app/publish .

ENTRYPOINT ["dotnet", "MicroServicerContainerBindMountConfigFiles.dll"]

**Step 6**: Create docker-compose.yml file in MicroServicerContainerBindMountConfigFiles solutions folder:

version: '3.9'

services:

bindmount-demo:

build: ./MicroServicerContainerBindMountConfigFiles

ports:

- "8091:80"

**volumes:**

**- ./MicroServicerContainerBindMountConfigFiles/appsettings.Docker.json:/app/appsettings.Docker.json:ro**

**Step 7:** Run it from MicroServicerContainerBindMountConfigFiles solutions folder

**docker compose up -d --build**

**Note: Earlier we have used docker-compose up -d --build. Now we have used docker compose up -d --build. docker compose (with no hyphen between docker compose is the new version which is directly integrated into Docker CLI. This is written in Go (faster, better maintained, works natively with new Docker features). Installed as part of Docker Desktop or the Docker Engine CLI plugin system — no separate installation needed**

**docker-compose with hyphen between docker and compose.** This is the **old CLI tool** (Python-based). **You install it separately (e.g., sudo apt install docker-compose)**

**We will now create another example to load the json file based on the environment variables**

**appsettings.Docker.Dev.json**

{

"AppSettings": {

"Message": "Hello from Bind Mount - DEVELOPMENT!"

}

}

**appsettings.Docker.Prod.json**

{

"AppSettings": {

"Message": "Hello from Bind Mount - PRODUCTION!"

}

}

**appsettings.Docker.Staging.json**

{

"AppSettings": {

"Message": "Hello from Bind Mount - STAGING!"

}

}

**DockerFile**

FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS base

WORKDIR /app

EXPOSE 80

FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build

WORKDIR /src

COPY ["MicroServicerContainerBindMountConfigFiles.csproj", "./"]

RUN dotnet restore "MicroServicerContainerBindMountConfigFiles.csproj"

COPY . .

RUN dotnet build "MicroServicerContainerBindMountConfigFiles.csproj" -c Release -o /app/build

# Note: Why /p:UseAppHost=false is used in the below command?

# Without the property UseAppHost=false (/p denotes property in Msbuild),

# this will generate an executable host (called the app host) for the project

# This will generate an .exe file <ProjectName>.exe

# Since Docker containers always run under a known runtime (dotnet command inside the base image),

# the .exe isn’t necessary and is skipped to save space and avoid cross-platform issues.

FROM build AS publish

RUN dotnet publish "MicroServicerContainerBindMountConfigFiles.csproj" -c Release -o /app/publish /p:UseAppHost=false

FROM base AS final

WORKDIR /app

COPY --from=publish /app/publish .

ENTRYPOINT ["dotnet", "MicroServicerContainerBindMountConfigFiles.dll"]

**docker-compose.yml**

version: "3.9"

services:

bindmount-dev:

build: ./MicroServicerContainerBindMountConfigFiles

container\_name: MicroserviceDockerMount-dev

environment:

- DOCKER\_ENV=Dev # For this container, inject env. variable DOCKER\_END, value=**Dev** to be used in Program.cs

- ASPNETCORE\_URLS=http://+:80

**volumes:**

**- ./MicroServicerContainerBindMountConfigFiles/appsettings.Docker.Dev.json:/app/appsettings.Docker.Dev.json:ro # For development use a appsettings.Docker. Dev.json file**

ports:

- "5001:80"

bindmount-staging:

build: ./MicroServicerContainerBindMountConfigFiles

container\_name: MicroserviceDockerMount-staging

environment:

- DOCKER\_ENV=Staging # For this container, inject env. variable DOCKER\_END, value=**Staging** to be used in Program.cs

- ASPNETCORE\_URLS=http://+:80

**volumes:**

**- ./MicroServicerContainerBindMountConfigFiles/appsettings.Docker.Staging.json:/app/appsettings.Docker.Staging.json:ro # For staging use a appsettings.Docker.Staging.json file**

ports:

- "5002:80"

bindmount-prod:

build: ./MicroServicerContainerBindMountConfigFiles

container\_name: MicroserviceDockerMount-prod

environment:

- DOCKER\_ENV=Prod # For this container, inject env. variable DOCKER\_END, value=**Prod** to be used in Program.cs

- ASPNETCORE\_URLS=http://+:80

**volumes:**

**- ./MicroServicerContainerBindMountConfigFiles/appsettings.Docker.Prod.json:/app/appsettings.Docker.Prod.json:ro # For production use a appsettings.Docker.Prod.json file**

ports:

- "5003:80"

**Program.cs**

namespace MicroServicerContainerBindMountConfigFiles

{

public class Program

{

public static void Main(string[] args)

{

var builder = WebApplication.CreateBuilder(args);

//builder.Configuration

// .SetBasePath(Directory.GetCurrentDirectory())

// .AddJsonFile("appsettings.json", optional: false, reloadOnChange: true)

// .AddJsonFile("appsettings.Docker.json", optional: true, reloadOnChange: true)

// .AddEnvironmentVariables();

// Read DOCKER\_ENV variable which is injected into the container

// through the docker-compose.yml file (default to "Dev" if not set)

**var dockerEnv = Environment.GetEnvironmentVariable("DOCKER\_ENV") ?? "Dev";**

**// Configure Configuration Sources**

**builder.Configuration**

**.SetBasePath(Directory.GetCurrentDirectory())**

**.AddJsonFile("appsettings.json", optional: false, reloadOnChange: true)**

**.AddJsonFile($"appsettings.Docker.{dockerEnv}.json", optional: true, reloadOnChange: true) // Read which config file to choose from dockerEnv variable**

**.AddEnvironmentVariables();**

// Add services to the container.

builder.Services.AddControllers();

// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

var app = builder.Build();

// Configure the HTTP request pipeline.

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI();

}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();

}

}

}

**MessageController.cs**

using Microsoft.AspNetCore.Mvc;

using Microsoft.Extensions.Configuration;

namespace MicroServicerContainerBindMountConfigFiles.Controllers

{

[ApiController]

[Route("api/[controller]")]

public class MessageController : ControllerBase

{

private readonly IConfiguration \_configuration;

public MessageController(IConfiguration configuration)

{

\_configuration = configuration;

}

[HttpGet]

public IActionResult Get()

{

// The configuration file is fetched dynamically based on the environment variable DOCKER\_ENV in Program.cs

// This allows for different configurations to be loaded based on the environment

var message = \_configuration["AppSettings:Message"] ?? "No message found";

return Ok(new { Message = message });

}

}

}

**Now to start the container run**

**docker compose up -d**

In browser, upon running: <http://localhost:5001/api/message>, you get the response:

{"message":"Hello from Bind Mount - DEVELOPMENT!"}

which is taken from appsettings.docker.Dev.json file

If the URL is <http://localhost:5002/api/message>, the response will be:

{"message":"Hello from Bind Mount - STAGING!"}

which is taken from appsettings.docker.staging.json file

If the URL is <http://localhost:5003/api/message>, the response will be:

{"message":"Hello from Bind Mount - PRODUCTION!"}

which is taken from appsettings.docker.Prod.json file

**Another way of handing environment files**

**Now using a single docker-compose file and by passing environment files in docker compose up the environment file is dynamically chosen as below:**

**Use the same Docker File, Program.cs and MessageController.cs. (But here, the use of the Environment variable is meaningless here)**

**appsettings.Development.json**

{

"AppSettings": {

"Message": "Hello from Bind Mount - DEVELOPMENT!"

}

}

**appsettings. Production.json**

{

"AppSettings": {

"Message": "Hello from Bind Mount - PRODUCTION!"

}

}

**appsettings.Staging.json**

{

"AppSettings": {

"Message": "Hello from Bind Mount - STAGING!"

}

}

**.env.dev (Environment file for Development)**

Here we declare a variable APPSETTINGS\_FILE and specify the appsettings file for Dev Enrionment

APPSETTINGS\_FILE=./MicroServicerContainerBindMountConfigFiles/appsettings.Development.json

**.env.Prod (Environment file for Production)**

APPSETTINGS\_FILE=./MicroServicerContainerBindMountConfigFiles/appsettings.Production.json

**.env.Prod (Environment file for Staging)**

APPSETTINGS\_FILE=./MicroServicerContainerBindMountConfigFiles/appsettings.Staging.json

**docker-compose.yml**

# ---------------------------------------------------------------

# Use this to create a single service

# and dynamically mount the config files upon container creation

#

# Create a multiple .env files (.env.dev, .env.prod, .env.staging)

# In these .env files, set the the settings files to use

# in the variable APPSETTINGS\_FILE

#

# When running docker compose, specify the .env file to be used.

# For example: docker compose --env-file .env.prod up -d

# ----------------------------------------------------------------

services:

multi-configfiles-using-singleservice:

build: ./MicroServicerContainerBindMountConfigFiles

container\_name: multiconfigfiles-singleservice

volumes:

# Host file (outside) → Container file (inside)

- **${APPSETTINGS\_FILE}:/app/appsettings.json**

environment:

- ASPNETCORE\_URLS=http://+:80

# Force environment to Production, as we are always mounting

# dynamically .json file (Prod, Staging, Env) to app/appsettings.json,

- ASPNETCORE\_ENVIRONMENT=Production

ports:

- **"5004:80"**

**Now run docker compse, by passing the environment files**

**docker compose --env-file .env.staging up -d**

Here we pass .evn.staging as the environment for docker compose. Now when the docker compose command above is run, docker compose will read the environment file that we passed in the command, read the variables and the values defined in the environment files and substitute the value.

In our example, we have provided the environment file **.env staging**.

This environment file has variable **APPSETTINGS\_FILE** whose value = ./MicroServicerContainerBindMountConfigFiles/appsettings.Staging.json

Now when this line is encountered in the docker-compose.yml file

volumes:

# Host file (outside) → Container file (inside)

- **${APPSETTINGS\_FILE}:/app/appsettings.json**

the line $(APPSETTINGS\_FILE) substitutes ./MicroServicerContainerBindMountConfigFiles/appsettings.Staging.json and this file is mapped to /app/appsettings.json in the container.

Now upon running <http://localhost:5004/api/message>, you get the output {"message":"Hello from Bind Mount - STAGING!"}