**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.**

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)**