**Creating Microservices Architecture**

We are creating a very simple Microservices Architecture project

1. Create the projects as below:
   1. 2 .Net core Web API Project (Project names: UserService, Inventory)
   2. 1 ASP.Net core Web App (Razor Pages) project (Project name: UI)
2. Install these components in UserService Project

* Microsoft.EntityFrameworkCore
* Microsoft.EntityFrameworkCore.SqlServer
* Microsoft.EntityFrameworkCore.Tools

1. Codes for UserService Project

**Model**

**namespace UserService.Models**

namespace UserService.Models

{

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public string? Email { get; set; }

}

}

**DBContext**

using Microsoft.EntityFrameworkCore;

using UserService.Models;

namespace UserService.Data

{

public class UserDbContext : DbContext

{

public UserDbContext(DbContextOptions<UserDbContext> options) : base(options) { }

public DbSet<User> Users { get; set; };

}

}

**appSettings.json**

{

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft.AspNetCore": "Warning"

}

},

"AllowedHosts": "\*",

"ConnectionStrings": {

"UserDb": "Server=localhost;Database=UserServiceDb;Trusted\_Connection=True;TrustServerCertificate=True;"

}

}

**Program.cs**

using Microsoft.EntityFrameworkCore;

using UserService.Data;

var builder = WebApplication.CreateBuilder(args);

// 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();

// Add DB Context

builder.Services.AddDbContext<UserDbContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("UserDb")));

// Add services

builder.Services.AddControllers();

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();

**Controller**

using Microsoft.AspNetCore.Mvc;

using UserService.Data;

using UserService.Models;

namespace UserService.Controllers

{

[ApiController]

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

public class UsersController : ControllerBase

{

private readonly UserDbContext \_context;

public UsersController(UserDbContext context)

{

\_context = context;

}

[HttpGet]

public IActionResult GetAll()

{

var users = \_context.Users.ToList();

return Ok(users);

}

[HttpGet("{id}")]

public IActionResult GetById(int id)

{

var user = \_context.Users.Find(id);

if (user == null) return NotFound();

return Ok(user);

}

[HttpPost]

public IActionResult Create(User user)

{

\_context.Users.Add(user);

\_context.SaveChanges();

return CreatedAtAction(nameof(GetById), new { id = user.Id }, user);

}

[HttpPut("{id}")]

public IActionResult Update(int id, User updatedUser)

{

var user = \_context.Users.Find(id);

if (user == null) return NotFound();

user.Name = updatedUser.Name;

user.Email = updatedUser.Email;

\_context.SaveChanges();

return NoContent();

}

[HttpDelete("{id}")]

public IActionResult Delete(int id)

{

var user = \_context.Users.Find(id);

if (user == null) return NotFound();

\_context.Users.Remove(user);

\_context.SaveChanges();

return NoContent();

}

}

}

Run migration script from Nuget Package Manager 🡪 Package Manager Console

Add-Migration Initial

Update-Database

1. Codes **for InventoryServices**:

NOTE: In the Inventory Services we are following a good structure.

**Models**

namespace InventoryService.Models

{

public class InventoryItem

{

public int Id { get; set; }

public string ProductName { get; set; } = string.Empty;

public int Quantity { get; set; }

public decimal Price { get; set; }

}

}

**DBContext**

using Microsoft.EntityFrameworkCore;

using InventoryService.Models;

namespace InventoryService.Data

{

public class InventoryDbContext : DbContext // Fix CS0311 by inheriting from DbContext

{

public InventoryDbContext(DbContextOptions<InventoryDbContext> options): base(options)

{

}

public DbSet<InventoryItem> Inventories { get; set; }

}

}

**appSettings.json**

{

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft.AspNetCore": "Warning"

}

},

"AllowedHosts": "\*",

"ConnectionStrings": {

"DefaultConnection": "Server=localhost;Database=InventoryDb;Trusted\_Connection=True;TrustServerCertificate=True;"

}

}

**Interface folders**

// Interfaces/IInventoryRepository.cs

using InventoryService.Models;

namespace InventoryService.Interfaces

{

public interface IInventoryRepository

{

Task<IEnumerable<InventoryItem>> GetAllAsync();

Task<InventoryItem?> GetByIdAsync(int id);

Task<InventoryItem> AddAsync(InventoryItem item);

Task<InventoryItem?> UpdateAsync(InventoryItem item);

Task<bool> DeleteAsync(int id);

}

}

// Interfaces/IInventoryService.cs

using InventoryService.Models;

namespace InventoryService.Interfaces

{

public interface IInventoryService

{

Task<IEnumerable<InventoryItem>> GetAllAsync();

Task<InventoryItem?> GetByIdAsync(int id);

Task<InventoryItem> AddAsync(InventoryItem item);

Task<InventoryItem?> UpdateAsync(InventoryItem item);

Task<bool> DeleteAsync(int id);

}

}

**Repositories Folder**

// Repositories/InventoryRepository.cs

using InventoryService.Data;

using InventoryService.Interfaces;

using InventoryService.Models;

using Microsoft.EntityFrameworkCore;

namespace InventoryService.Repositories

{

public class InventoryRepository : IInventoryRepository

{

private readonly InventoryDbContext \_context;

public InventoryRepository(InventoryDbContext context)

{

\_context = context;

}

public async Task<IEnumerable<InventoryItem>> GetAllAsync()

{

return await \_context.Inventories.ToListAsync();

}

public async Task<InventoryItem?> GetByIdAsync(int id)

{

return await \_context.Inventories.FindAsync(id);

}

public async Task<InventoryItem> AddAsync(InventoryItem item)

{

\_context.Inventories.Add(item);

await \_context.SaveChangesAsync();

return item;

}

public async Task<InventoryItem?> UpdateAsync(InventoryItem item)

{

var existing = await \_context.Inventories.FindAsync(item.Id);

if (existing == null) return null;

existing.ProductName = item.ProductName;

existing.Quantity = item.Quantity;

await \_context.SaveChangesAsync();

return existing;

}

public async Task<bool> DeleteAsync(int id)

{

var item = await \_context.Inventories.FindAsync(id);

if (item == null) return false;

\_context.Inventories.Remove(item);

await \_context.SaveChangesAsync();

return true;

}

}

}

**Services folder**

// Services/InventoryService.cs

using InventoryService.Interfaces;

using InventoryService.Models;

namespace InventoryService.Services

{

public class InventoryService : IInventoryService

{

private readonly IInventoryRepository \_repo;

public InventoryService(IInventoryRepository repo)

{

\_repo = repo;

}

public Task<IEnumerable<InventoryItem>> GetAllAsync() => \_repo.GetAllAsync();

public Task<InventoryItem?> GetByIdAsync(int id) => \_repo.GetByIdAsync(id);

public Task<InventoryItem> AddAsync(InventoryItem item) => \_repo.AddAsync(item);

public Task<InventoryItem?> UpdateAsync(InventoryItem item) => \_repo.UpdateAsync(item);

public Task<bool> DeleteAsync(int id) => \_repo.DeleteAsync(id);

}

}

**Controllers folder**

// Controllers/InventoryController.cs

using InventoryService.Interfaces;

using InventoryService.Models;

using Microsoft.AspNetCore.Mvc;

namespace InventoryService.Controllers

{

[ApiController]

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

public class InventoryController : ControllerBase

{

private readonly IInventoryService \_service;

public InventoryController(IInventoryService service)

{

\_service = service;

}

[HttpGet]

public async Task<IActionResult> GetAll()

{

var items = await \_service.GetAllAsync();

return Ok(items);

}

[HttpGet("{id}")]

public async Task<IActionResult> GetById(int id)

{

var item = await \_service.GetByIdAsync(id);

return item is null ? NotFound() : Ok(item);

}

[HttpPost]

public async Task<IActionResult> Add([FromBody]InventoryItem item)

{

var added = await \_service.AddAsync(item);

return CreatedAtAction(nameof(GetById), new { id = added.Id }, added);

}

[HttpPut("{id}")]

public async Task<IActionResult> Update(int id,[FromBody]InventoryItem item)

{

if (id != item.Id) return BadRequest("Mismatched ID");

var updated = await \_service.UpdateAsync(item);

return updated is null ? NotFound() : Ok(updated);

}

[HttpDelete("{id}")]

public async Task<IActionResult> Delete(int id)

{

var deleted = await \_service.DeleteAsync(id);

return deleted ? NoContent() : NotFound();

}

}

}

**Program.cs**

using InventoryService.Data;

using InventoryService.Interfaces;

using InventoryService.Repositories;

using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

// 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();

builder.Services.AddDbContext<InventoryDbContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

builder.Services.AddScoped<IInventoryRepository, InventoryRepository>();

builder.Services.AddScoped<IInventoryService, InventoryService.Services.InventoryService>();

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();

So we are following the current structure as:

**Client → Controller → Service Interface → Service Implementation**

**→ Repository Interface → Repository Implementation → DbContext**

For our above implementation

Client 🡪 Controller → IInventoryService → InventoryService → IInventoryRepository → InventoryRepository → DbContext

1. Validating InventoryService using **FluentValidation** (just testing one method for the sake of learning)

**Install FluentValidation from Nuget Package**

**FluentValidation** (This package by jskinner. There are other packages also, don’t install them)

**Create a Validator**

Create a new folder Validators and add a class called InventoryItemValidator.cs

using FluentValidation;

using InventoryService.Models;

public class InventoryItemValidator : AbstractValidator<InventoryItem>

{

public InventoryItemValidator()

{

RuleFor(x => x.ProductName).NotEmpty().WithMessage("Name is required.").MaximumLength(100);

RuleFor(x => x.Quantity).GreaterThanOrEqualTo(0).WithMessage("Quantity must be non-negative.");

RuleFor(x => x.Price).GreaterThan(0).WithMessage("Price must be greater than 0.");

}

}

**For Sample, just change the Add method to incorporate FluentValidation as below:**

[HttpPost]

public async Task<IActionResult> Add([FromBody] InventoryItem item)

{

var validator = new InventoryItemValidator();

var validationResult = await validator.ValidateAsync(item);

if (!validationResult.IsValid)

{

var errors = validationResult.Errors

.Select(e => e.ErrorMessage)

.ToList();

return BadRequest(errors);

}

var addedItem = await \_service.AddAsync(item);

return CreatedAtAction(nameof(GetById), new { id = addedItem.Id },

addedItem);

}

1. Global error checking for other methods:

**Create a folder Middleware and write Global Error Handler class in the folder as below:**

using System.Net;

using System.Text.Json;

namespace InventoryService.Middleware;

public class ErrorHandlerMiddleware

{

// Reference to the next middleware in the pipeline

private readonly RequestDelegate \_next;

// Logger to log any unhandled exceptions

private readonly ILogger<ErrorHandlerMiddleware> \_logger;

// Constructor to inject dependencies: next middleware and logger

public ErrorHandlerMiddleware(RequestDelegate next, ILogger<ErrorHandlerMiddleware> logger)

{

\_next = next;

\_logger = logger;

}

// This method is called automatically for each HTTP request

public async Task Invoke(HttpContext context)

{

try

{

// Pass control to the next middleware component or the endpoint (controller)

await \_next(context);

}

catch (Exception ex)

{

// Log the exception details with a custom message

\_logger.LogError(ex, "An unhandled exception occurred");

// Set the HTTP status code to 500 (Internal Server Error)

context.Response.StatusCode = StatusCodes.Status500InternalServerError;

// Set the response content type to JSON so that the client receives a proper error object

context.Response.ContentType = "application/json";

// Create a simple error response object to return to the client

var result = new

{

message = "An unexpected error occurred.", // Generic error message

details = ex.Message // Include actual exception message (hide in production if needed)

};

// Serialize and write the error response as JSON to the response body

await context.Response.WriteAsJsonAsync(result);

}

}

}

**Line by Line Explanation of the middleware**

**🔍 Line-by-Line Explanation**

**public class ErrorHandlerMiddleware**

* This declares a middleware class named ErrorHandlerMiddleware.
* Middleware is a component that runs in the HTTP pipeline and can inspect, modify, or handle HTTP requests/responses.

**private readonly RequestDelegate \_next;**

* This represents the next middleware in the pipeline.
* Middleware components are chained, and \_next lets this component forward the request down the pipeline.

**private readonly ILogger<ErrorHandlerMiddleware> \_logger;**

* This allows you to log messages (errors, warnings, info).
* The ILogger<T> is injected automatically by ASP.NET Core’s built-in dependency injection.

**public ErrorHandlerMiddleware(RequestDelegate next, ILogger<ErrorHandlerMiddleware> logger)**

* Constructor that accepts dependencies:
  + next: the next middleware delegate
  + logger: the logging service for logging errors

**public async Task Invoke(HttpContext context)**

* This method is **mandatory** in middleware.
* It gets called for **every HTTP request**.
* HttpContext context contains request/response details like headers, status code, body, etc.

**try { await \_next(context); }**

* It forwards the request to the **next middleware or controller**.
* If no exception occurs, everything runs normally.

**catch (Exception ex)**

* If any **unhandled exception** occurs in the next middleware or controller, it will be caught here.
* This is where **global error handling** happens.

**\_logger.LogError(ex, "Unhandled exception occurred");**

* Logs the exception using the built-in logger.
* You can later view this in the console, log files, or application monitoring tools.

**context.Response.StatusCode = StatusCodes.Status500InternalServerError;**

* Sets the HTTP response status code to **500**, which means **Internal Server Error**.

**context.Response.ContentType = "application/json";**

* Ensures the response is sent back as **JSON**, so Postman/clients can parse it properly.

**var result = new { message = ..., details = ex.Message };**

* Creates an **anonymous object** for the JSON response.
* message: a generic user-facing error message.
* details: the actual exception message. (In production, you might remove this for security.)

**await context.Response.WriteAsJsonAsync(result);**

* Serializes the result object to JSON and writes it to the HTTP response body.

**Register the middle ware in Program.cs**

The complete Program.cs including the middleware is given below:

using InventoryService.Data;

using InventoryService.Interfaces;

using InventoryService.Repositories;

using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

// 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();

builder.Services.AddDbContext<InventoryDbContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

builder.Services.AddScoped<IInventoryRepository, InventoryRepository>();

builder.Services.AddScoped<IInventoryService, InventoryService.Services.InventoryService>();

var app = builder.Build();

// Add custom middleware for error handling

app.UseMiddleware<InventoryService.Middleware.ErrorHandlerMiddleware>();

// Configure the HTTP request pipeline.

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI();

}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();

1. Capture logs using Serilog

**To capture logs using Serilog for Console and Files, we have to install the following**:

Serilog.AspNetCore

Serilog.Sinks.Console

Serilog.Sinks.File

**Then make changes in Program.cs file**

using InventoryService.Data;

using InventoryService.Interfaces;

using InventoryService.Repositories;

using Microsoft.EntityFrameworkCore;

using Serilog;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

// Configure Serilog

Log.Logger = new LoggerConfiguration()

.WriteTo.Console()

.WriteTo.File("Logs/log.txt", rollingInterval: RollingInterval.Day)

.Enrich.FromLogContext()

.CreateLogger();

builder.Host.UseSerilog(); // Replace built-in logger with Serilog

builder.Services.AddControllers();

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

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

builder.Services.AddDbContext<InventoryDbContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

builder.Services.AddScoped<IInventoryRepository, InventoryRepository>();

builder.Services.AddScoped<IInventoryService, InventoryService.Services.InventoryService>();

var app = builder.Build();

// Add custom middleware for error handling

app.UseMiddleware<InventoryService.Middleware.ErrorHandlerMiddleware>();

// Configure the HTTP request pipeline.

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI();

}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();

1. Suppress message in Production.

In case, if we want to suppress the messages received by user in Production, we need to change our Custom Error Handler as below:

using System.Net;

using System.Text.Json;

namespace InventoryService.Middleware;

public class ErrorHandlerMiddleware

{

// Reference to the next middleware in the pipeline

private readonly RequestDelegate \_next;

// Logger to log any unhandled exceptions

private readonly ILogger<ErrorHandlerMiddleware> \_logger;

private readonly IWebHostEnvironment \_env;

// Constructor to inject dependencies: next middleware and logger

public ErrorHandlerMiddleware(RequestDelegate next, ILogger<ErrorHandlerMiddleware> logger, IWebHostEnvironment env)

{

\_next = next;

\_logger = logger;

\_env = env;

}

// This method is called automatically for each HTTP request

public async Task Invoke(HttpContext context)

{

try

{

// Pass control to the next middleware component or the endpoint (controller)

await \_next(context);

}

catch (Exception ex)

{

// Log the exception details with a custom message

\_logger.LogError(ex, "An unhandled exception occurred");

// Set the HTTP status code to 500 (Internal Server Error)

context.Response.StatusCode = StatusCodes.Status500InternalServerError;

// Set the response content type to JSON so that the client receives a proper error object

context.Response.ContentType = "application/json";

// Create a simple error response object to return to the client

//var result = new

//{

// message = "An unexpected error occurred.", // Generic error message

// details = ex.Message // Include actual exception message (hide in production if needed)

//};

var result = \_env.IsDevelopment()

? new

{

message = "An unexpected error occurred.",

details = ex.Message // Show detailed error in development

}

: new

{

message = "An unexpected error occurred. Please contact support.", // No internal details in production

details = string.Empty

};

// Serialize and write the error response as JSON to the response body

await context.Response.WriteAsJsonAsync(result);

}

}

}

1. Create the docker file as below:

# 1. Use the official .NET SDK image to build the app

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

# Explanation:

# This sets the base image to the .NET SDK 8.0, which includes tools for building .NET apps.

# The alias `AS build` lets us refer to this stage later (multi-stage build).

# This image is temporary and won't be part of the final image — only used to compile the app.

# 2. Set the working directory inside the container for this build stage

WORKDIR /app

# Explanation:

# WORKDIR creates (if not exists) and moves into the /app directory inside the container.

# All subsequent RUN, COPY, etc., commands will be relative to this folder inside the container.

# 3. Copy only the .csproj file to the container (from build context)

COPY \*.csproj ./

# Explanation:

# This copies your .csproj file from your host machine to the /app folder in the container.

# Wildcards like \*.csproj are supported only for local matches — not inside nested folders.

# This step helps us restore NuGet packages without copying unnecessary files early.

# 4. Restore NuGet packages (dependencies)

RUN dotnet restore

# Explanation:

# Runs `dotnet restore` to download dependencies defined in the .csproj file.

# Since only .csproj was copied, Docker can cache this layer for faster builds,

# as long as you don’t change the project file.

# 5. Copy the rest of the source code to the container

COPY . ./

# Explanation:

# This copies everything else (controllers, Program.cs, etc.) to /app inside the container.

# This must be done \*after\* restore to avoid triggering restore again on every small change.

# 6. Build the application and publish it to a folder (in release mode)

RUN dotnet publish -c Release -o /app/publish

# Explanation:

# Publishes the compiled app to the `/app/publish` directory inside the container.

# `-c Release` ensures production-grade optimizations are used.

# 7. Use a smaller runtime image for the final container (no SDK)

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

# Explanation:

# This is a smaller image (ASP.NET runtime only — no SDK/tools).

# This keeps the image size minimal and secure for deployment.

# 8. Set working directory for the final runtime container

WORKDIR /app

# Explanation:

# Again, sets the working directory to /app (fresh context in runtime image).

# The publish folder will be copied into this.

# 9. Copy published output from the build stage to the runtime container

COPY --from=build /app/publish .

# Explanation:

# Copies the published output files from the build stage to the final image’s /app directory.

# Only the binaries and DLLs needed to run the app are copied (no source code).

# 10. Run the application

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

# Explanation:

# This tells the container what command to run when started.

# It launches the compiled DLL using the dotnet runtime — the main entry point of your API.

1. What is a build context?

When we run the below command to build the docker image:

**docker build -t inventory-service .**

 The . is your **build context** — usually the folder where Dockerfile and .csproj are located.

 Docker will look inside this folder for files to copy (e.g., .csproj, code, etc.).

1. If we do not have Dockers installed, need to install Dockers. I am installing Dockers on my laptop. Hence do the following:
   1. Visit the site: <https://www.docker.com/products/docker-desktop/>
   2. Click Download Docker Desktop
   3. Choose version “Download for Windows-AMD64
   4. Once downloaded, install the docker.
2. To create the docker image, from Solutions Explorer do the following:
   1. Right click on the project, in our case InventoryService
   2. Click on **Open in Terminal**
   3. Run the below command to create the docker image:

**docker build -t inventory-service .**

**Upon executing the above command, Docker looks for the Dockerfile in the current directory(.) and builds the image.**

1. To Run the Docker Container (Creates and starts a container from the image), run the following and command:

**docker run -d -p 8080:8080 --name inventory-api inventory-service**

| **Option** | **Meaning** |
| --- | --- |
| -d | Run in detached mode (in background) |
| -p 8080:8080 | Maps **host port 8080** → **container port 8080** (change if needed) |
| --name inventory-api | Gives a name to the running container |
| inventory-service | The image name you built earlier |

1. To check if the container is running, run the below command:

**docker ps**

1. To see the logs for the container, for example our container name is inventory-api, run the following command:

**docker logs inventory-api**

1. Note: We will be creating separate containers for each service and their database. Each service and each database will have their own containers. Hence in order for the services in a container to communicate with a database in a container, we need to create a docker network. And then when creating the container, we need to specify the network.
2. Now we will create a network as below:

**docker network create microservice-net**

This creates a network by name microservice-net

1. Now let us pull and run the official Microsoft SQL Server 2022 image from the Terminal and run SQL Server container inside the network we created above (microservice-net).

**docker run -e "ACCEPT\_EULA=Y" -e "SA\_PASSWORD=Your\_strong\_password123" -p 1433:1433 --name inventory-sqlserver --network microservice-net -d mcr.microsoft.com/mssql/server:2022-latest**

Here my password for SA is Your\_strong\_password123

### What this does:

* Pulls the **SQL Server 2022 Linux image**
* Sets the sa password
* Maps port 1433 (SQL default)
* Names the container inventory-sqlserver
* Runs inside the network microservice-net
* Runs it in detached mode (-d)

1. Now we are updating the connection string to point to the Docker SQL Server as follows: For now we are using only one connection string. Later we have to handle multiple connection strings.

{

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft.AspNetCore": "Warning"

}

},

"AllowedHosts": "\*",

//"ConnectionStrings": {

// "DefaultConnection": "Server=localhost;Database=InventoryDb;Trusted\_Connection=True;TrustServerCertificate=True;"

"ConnectionStrings": {

"DefaultConnection": "Server=inventory-sqlserver,1433;Database=InventoryDb;User Id=sa;Password=Your\_strong\_password123;TrustServerCertificate=True;"

}

}

1. Earlier when we created the inventory-api container (i.e. when we earlier ran the command **docker run -d -p 8080:8080 --name inventory-api inventory-service)**, we did not include under which network should this container run. Only when the inventory-api container and the inventory-service container run in the same network, both these containers will be able to communicate with each other. Since the inventory-api container is already running, we need to stop it, remove and then recreate it by indicating the network. Hence we will use the following set of commands:

docker stop inventory-api (Stop the container)

docker rm inventory-api (Remove the container)

docker build --no-cache -t inventory-service . (Build image “inventor-service w/o using cache)

**docker run -d --name inventory-api --network microservice-net -p 8080:8080 inventory-service** (this command will create the container in the network microservice-net in port 8080 in the container, which will be referenced in the host in port 8080)

1. To check which containers are running on the network microservice-net, which we created above run the below command:

**docker network inspect microservice-net**

1. Since we have already written the controller, repository, models and service we just have to run the migration as below: If there are multiple DB context, we have to specify at the last -Context <DBContextName> Eg. Add-migration InitialCreate -Context InventoryDbContext and Update-database -Context InventoryDbContext

Add-Migration InitialCreate -Context InventoryDbContext

Update-Database -Context InventoryDbContext

1. Now let us write the dockerfile for UserServices:

# 1. Use the official .NET SDK image to build the app

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

# Explanation:

# This sets the base image to the .NET SDK 8.0, which includes tools for building .NET apps.

# The alias `AS build` lets us refer to this stage later (multi-stage build).

# This image is temporary and won't be part of the final image — only used to compile the app.

# 2. Set the working directory inside the container for this build stage

WORKDIR /app

# Explanation:

# WORKDIR creates (if not exists) and moves into the /app directory inside the container.

# All subsequent RUN, COPY, etc., commands will be relative to this folder inside the container.

# 3. Copy only the .csproj file to the container (from build context)

COPY \*.csproj ./

# Explanation:

# This copies your .csproj file from your host machine to the /app folder in the container.

# Wildcards like \*.csproj are supported only for local matches — not inside nested folders.

# This step helps us restore NuGet packages without copying unnecessary files early.

# 4. Restore NuGet packages (dependencies)

RUN dotnet restore

# Explanation:

# Runs `dotnet restore` to download dependencies defined in the .csproj file.

# Since only .csproj was copied, Docker can cache this layer for faster builds,

# as long as you don’t change the project file.

# 5. Copy the rest of the source code to the container

COPY . ./

# Explanation:

# This copies everything else (controllers, Program.cs, etc.) to /app inside the container.

# This must be done \*after\* restore to avoid triggering restore again on every small change.

# 6. Build the application and publish it to a folder (in release mode)

RUN dotnet publish -c Release -o /app/publish

# Explanation:

# Publishes the compiled app to the `/app/publish` directory inside the container.

# `-c Release` ensures production-grade optimizations are used.

# 7. Use a smaller runtime image for the final container (no SDK)

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

# Explanation:

# This is a smaller image (ASP.NET runtime only — no SDK/tools).

# This keeps the image size minimal and secure for deployment.

# 8. Set working directory for the final runtime container

WORKDIR /app

# Explanation:

# Again, sets the working directory to /app (fresh context in runtime image).

# The publish folder will be copied into this.

# 9. Copy published output from the build stage to the runtime container

COPY --from=build /app/publish .

# Explanation:

# Copies the published output files from the build stage to the final image’s /app directory.

# Only the binaries and DLLs needed to run the app are copied (no source code).

# 10. Run the application

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

# Explanation:

# This tells the container what command to run when started.

# It launches the compiled DLL using the dotnet runtime — the main entry point of your API.

1. Now since we want the app to run in port 8081 in the container, in program.cs file , we add the below statement after the line var builder = WebApplication.CreateBuilder(args); Hence a part of Program.cs will look like as below:

var builder = WebApplication.CreateBuilder(args);

// Make the app listen on port 8081 inside the container

builder.WebHost.UseUrls("http://\*:8081");

Note: Here we are using http because, if we use https for containers, we need to install

certificates. Hence since we are only learning to make it simple we have used http.

1. Now, to create the docker image for UserService, from Solutions Explorer do the following:
   1. Right click on the project, in our case UserService
   2. Click on **Open in Terminal**
   3. Run the below command to create the docker image:

**docker build -t user-service .**

**Upon executing the above command, Docker looks for the Dockerfile in the current directory(.) and builds the image.**

1. To Run the Docker Container (Creates and starts a container from the image), run the following and command: This maps the port 8081 on the host to listen to the port 8081 on the container. The first 8081 to the left of “:” denotes the port of the host and the second 8081 to the right of “:” denotes the port of the container. So, when port 8081 is specified on the host, it points to port 8081 on the container.

**docker run -d -p 8081:8081 --name user-api user-service**

1. Now let us pull and run the official Microsoft SQL Server 2022 image from the Terminal for the UserService. Earlier we created a container for InventoryService. Now we will create a container for UserService.

**docker docker run -e "ACCEPT\_EULA=Y" -e "SA\_PASSWORD=Your\_strong\_password123" -p 1434:1433 --name user-sqlserver -d mcr.microsoft.com/mssql/server:2022-latest**

Here my password for SA is Your\_strong\_password123

This maps the localhost port 1434 to the port 1433 on the container, which is the default port for SQL Server.

**Now whatever we have done earlier, we will do it with docker-compose.yml**

### 🧱 ****What is**** docker-compose****?****

**docker-compose.yml** is a configuration file used by the Docker Compose tool to **define and manage multiple containers** as a single application.

#### ✅ ****Why use it?****

* To **orchestrate** your microservices (Inventory, User, etc.) with their own containers.
* Start, stop, and manage **multiple containers with a single command**.
* Define how services **communicate**, share **networks**, and persist **data**.
* Replace repetitive docker run commands with a **declarative file**.

#### 🛠️ ****Example Use Cases in Your Project:****

* Spin up inventory-api and inventory-sqlserver in one go.
* Ensure user-api only starts **after** user-sqlserver is up (depends\_on).
* Simplify deployment to CI/CD or Azure Kubernetes later.

### 🧾 ****What is**** .env ****File?****

The **.env file** is used to **store environment-specific configuration variables** (like secrets, passwords, connection strings) outside of your Compose file.

#### ✅ ****Why use it?****

* Avoid **hardcoding sensitive values** (e.g., passwords, ports) in docker-compose.yml.
* Simplify management across **multiple environments** (dev, staging, prod).
* Makes the Compose file **cleaner and more reusable**.

#### 🛠️ ****Example Use Cases in Your Project:****

* SA\_PASSWORD, ConnectionStrings\_\_DefaultConnection, etc., are pulled from .env.
* Can commit docker-compose.yml to source control, but **exclude .env** to keep credentials private.

**Docker Compose → Lightweight Container Orchestration Tool**

**✅ Purpose:**

Docker Compose is a **developer-friendly orchestration tool** used to:

* Define **multi-container applications** in a single YAML file.
* Automate the **startup order**, **networking**, **volumes**, and **environment variables**.
* Simplify **local development**, testing, and even CI setups.

1. Following are the commands and their purpose:

Check what is taking space

**docker system df**

Remove stopped containers

**docker container prune**

Remove dangling and also unused images

**docker image prune**

All unused resources ( aggressive)

This removes **everything that Docker considers “unused”** across **images**, **containers**, **networks**, **volumes**, and **build cache**

**By default (docker system prune without any flags), it removes:**

✅ **Stopped containers**  
✅ **Dangling images** (images with <none>:<none> not used by any container)  
✅ **Unused networks** (not used by any containers)  
✅ **Build cache** (intermediate build layers no longer needed)

**docker system prune**

Force remove of containers (even removes running containers)

**docker rm -f <container-name-1> <container-name-2> …**

1. Now let us start removing old containers we have already:

**docker rm -f inventory-api inventory-sqlserver user-api user-sqlserver**

1. (Optional) Clean up old unused images to free disk space:

**docker image prune -a**

This deletes **all unused images** not tied to running containers — useful if you're starting fresh or storage is an issue.

1. We will write the Dockerfile for InventoryService

# ===============================================================

# Multi-stage Dockerfile for InventoryService (.NET 8 Web API)

# Stage 1: Build and publish the application

# Stage 2: Use lightweight runtime for final image

# ===============================================================

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

# Stage 1: Build stage

# Uses full SDK image to compile and publish the app

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

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

WORKDIR /src

# Copy the project file and restore dependencies

COPY InventoryService.csproj ./

RUN dotnet restore

# Copy the entire source and build the app in Release mode

COPY . ./

RUN dotnet publish -c Release -o /app/publish

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

# Stage 2: Runtime stage

# Uses smaller ASP.NET Core runtime-only image for deployment

# Keeps the image size minimal and secure

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

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

WORKDIR /app

# Enable support for TLS or plaintext HTTP across containers

# Workaround for internal HTTP communication issues in some cases

ENV DOTNET\_SYSTEM\_NET\_HTTP\_USESOCKETSHTTPHANDLER=0

ENV DOTNET\_SYSTEM\_NET\_SECURITY\_ALLOWUNENCRYPTED=true

# Copy published output from the build stage

COPY --from=build /app/publish .

# Set the startup command for the container

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

1. Now we will write Dockerfile for UserService

# ===============================================================

# Multi-stage Dockerfile for UserService (.NET 8 Web API)

# Stage 1: Build and publish the application

# Stage 2: Use lightweight runtime for final image

# ===============================================================

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

# Stage 1: Build stage

# Starts from the full .NET 8 SDK image for building the app

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

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

WORKDIR /src

# Copy only the project file first to leverage Docker caching

# This allows cached layer reuse if dependencies haven’t changed

COPY UserService.csproj ./

RUN dotnet restore

# Copy the rest of the source and publish to a folder

# Output goes to /app/publish in Release configuration

COPY . ./

RUN dotnet publish -c Release -o /app/publish

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

# Stage 2: Final runtime stage

# Uses ASP.NET Core runtime image (no SDK) for leaner deployment

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

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

WORKDIR /app

# Environment variables to allow HTTP communication between services

# These are essential for avoiding TLS-related errors in Docker networks

ENV DOTNET\_SYSTEM\_NET\_HTTP\_USESOCKETSHTTPHANDLER=0

ENV DOTNET\_SYSTEM\_NET\_SECURITY\_ALLOWUNENCRYPTED=true

# Copy the published output from the build stage

COPY --from=build /app/publish .

# Define the container startup command

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

1. We need to place **docker-compose.yml** in the **solutions** folder. The contents of the file is as below:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Note 1: Here we have defined environment variables like ASPNETCORE\_ENVIRONMENT. But for them to be successfully passed to the app in the container, they have to be properly referenced in the application. For example, see the below code.

builder.Host.ConfigureAppConfiguration((hostingContext, config) =>

{

var env = hostingContext.HostingEnvironment;

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

.AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true, reloadOnChange: true);

});

But we have not used the above code in Program.cs. Hence this is just for knowledge only.

Note 2: In the code the placeholder for SQL Server initialization and mount volume is commented for knowledge purpose. For example in our docker-compose.yml file in the below code segment, the commented portions persists the container data on the volume name inventory-sqldata.

Similarly we can have initialization script files (in the example below the .sql file name is inventory-init.sql). This file has to be placed in the same location as that of .docker-compose.yml. Actually we have not created inventory-init.sql file. This is just for knowledge purpose only.

*inventory-sqlserver:*

*image: mcr.microsoft.com/mssql/server:2022-latest*

*container\_name: inventory-sqlserver*

*environment:*

*- ACCEPT\_EULA=Y*

*- SA\_PASSWORD=Your\_strong\_password123*

*ports:*

*- "1433:1433"*

*networks:*

*- microservice-network*

*# Mount persistent volume and init SQL script*

*# volumes:*

*# - inventory-sqldata:/var/opt/mssql*

*# - ./inventory-init.sql:/init/inventory-init.sql*

And at the bottom of the docker-compose.yml file, we have commented and included this portion:

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

# Volumes for persistent SQL Server data

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

# volumes:

# inventory-sqldata:

# user-sqldata:

We need to have this to persist the volumes

.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The code starts here:

version: '3.8'

# ============================================================

# Docker Compose setup for Inventory and User microservices.

# Each service runs its own DB container and ASP.NET backend.

# Includes commented helper container to run init SQL scripts.

# Ready for extension to other DB engines (MySQL/PostgreSQL).

# ============================================================

services:

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

# SQL Server for InventoryService

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

inventory-sqlserver:

image: mcr.microsoft.com/mssql/server:2022-latest

container\_name: inventory-sqlserver

environment:

- ACCEPT\_EULA=Y

- SA\_PASSWORD=Your\_strong\_password123

ports:

- "1433:1433"

networks:

- microservice-network

# Mount persistent volume and init SQL script

# volumes:

# - inventory-sqldata:/var/opt/mssql

# - ./inventory-init.sql:/init/inventory-init.sql

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

# InventoryService - ASP.NET Core Web API

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

inventoryservice:

build:

context: ./InventoryService

dockerfile: Dockerfile

ports:

- "8080:8080"

depends\_on:

- inventory-sqlserver

environment:

- ASPNETCORE\_ENVIRONMENT=Production

networks:

- microservice-network

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

# SQL Server for UserService

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

user-sqlserver:

image: mcr.microsoft.com/mssql/server:2022-latest

container\_name: user-sqlserver

environment:

- ACCEPT\_EULA=Y

- SA\_PASSWORD=Your\_strong\_password123

ports:

- "1434:1433"

networks:

- microservice-network

# Mount persistent volume and init SQL script

# volumes:

# - user-sqldata:/var/opt/mssql

# - ./user-init.sql:/init/user-init.sql

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

# UserService - ASP.NET Core Web API

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

userservice:

build:

context: ./UserService

dockerfile: Dockerfile

ports:

- "8081:8081"

depends\_on:

- user-sqlserver

environment:

- ASPNETCORE\_ENVIRONMENT=Production

networks:

- microservice-network

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

# (Commented) Helper container to run init scripts using sqlcmd

#

# This container runs user-init.sql after user-sqlserver is up.

# Similarly, inventory-init.sql can be run for inventory-sqlserver.

# Both SQL script files must be located in the same directory

# as this docker-compose.yml file.

#

# Example script content:

# user-init.sql -> setup for user DB (tables, seed data)

# inventory-init.sql -> (sample) setup for inventory DB

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

# init-user-db:

# image: mcr.microsoft.com/mssql-tools

# depends\_on:

# - user-sqlserver

# entrypoint: ["/bin/bash", "-c"]

# command:

# - |

# echo "Waiting for SQL Server to be available...";

# sleep 20;

# /opt/mssql-tools/bin/sqlcmd -S user-sqlserver -U sa -P 'Your\_strong\_password123' -i /init/user-init.sql;

# echo "User database initialized.";

# volumes:

# - ./user-init.sql:/init/user-init.sql

# networks:

# - microservice-network

# init-inventory-db:

# image: mcr.microsoft.com/mssql-tools

# depends\_on:

# - inventory-sqlserver

# entrypoint: ["/bin/bash", "-c"]

# command:

# - |

# echo "Waiting for SQL Server to be available...";

# sleep 20;

# /opt/mssql-tools/bin/sqlcmd -S inventory-sqlserver -U sa -P 'Your\_strong\_password123' -i /init/inventory-init.sql;

# echo "Inventory database initialized.";

# volumes:

# - ./inventory-init.sql:/init/inventory-init.sql

# networks:

# - microservice-network

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

# Custom bridge network for all services

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

networks:

microservice-network:

driver: bridge

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

# Volumes for persistent SQL Server data

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

# volumes:

# inventory-sqldata:

# user-sqldata:

1. Now run the following command:

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

**Running this command does the following:**

**-d will run in the background**

 You spin up **all containers in one command**: docker-compose up --build.

 Networking, volumes, and environment variables are **automatically handled**.

 You can **bring it all down and up again with state preserved**, which is excellent for development

1. Step by Step clean-up commands:

**Stop and remove all containers**

docker container stop $(docker container ls -aq)

docker container rm $(docker container ls -aq)

If there are no containers, you’ll see nothing removed — that’s fine.

**Remove all images**

docker rmi -f $(docker images -q)

This removes **all Docker images**, including .NET SDK, SQL Server, etc. They’ll be re-pulled as needed.

**(Optional) Prune Docker system**

Removes **networks, volumes, build cache** not used by any container:

docker system prune -a --volumes

It will ask for confirmation (y). This is the most aggressive cleanup.

1. Once the clean up is done, docker-compose.yml can be run by any of these commands:

**✅ docker-compose up -d**

* Runs your containers in **detached mode** (in the background).
* **Does NOT rebuild images** if Docker thinks they are up to date.
* Fastest if nothing has changed.

**docker-compose up --build**

* Builds **fresh Docker images** before running the containers.
* Useful when:
  + You've changed the Dockerfile
  + You've changed dependencies
  + You want to be sure everything is rebuilt cleanly
* Runs in **foreground** unless you add -d.

1. How to run a command inside a container?

Suppose that my container name is microservicestarter-userservice-1, I will run the below command:

docker exec -it microservicestarter-userservice-1 /bin/sh

| **Part** | **Meaning** |
| --- | --- |
| docker | You are calling Docker CLI. |
| exec | Execute a command in a running container. |
| -it | Combine two flags:**-i** = interactive mode (keep STDIN open)**-t** = allocate a pseudo-TTY (you get a terminal-like experience). |
| microservicestarter-userservice-1 | Name (or ID) of the running container you want to access. |
| /bin/sh | The shell program to run inside the container. (A lightweight shell like Bash). |

**So this command:**

Opens a terminal shell **inside the userservice container**, so you can run commands there interactively.

1. Some Commands

**docker-compose stop userservice inventoryservice.** These commands stop the userservice and inventoryservice. These services correspond to the containers created in the docker. Note these are the names specified under the services: in the docker-compose.yml file

**docker-compose build --no-cache userservice inventoryservice.** This command re-builds the images without the cache.

**docker-compose up -d userservice inventoryservice**. Brings up the service

1. **Explanations for Mounts and Volumes**

**🔹 Concept: Mounting and Volumes in Docker**

There are two main ways to persist data in Docker:

**1. Bind Mounts (host\_path:/container\_path)**

* You manually bind a folder from the host machine into the container.
* Example: C:/data:/var/opt/mssql
* Changes are reflected both ways: host ↔ container.
* Useful during development, but **less portable**.

**2. Named Volumes (volume\_name:/container\_path)**

* Docker manages the storage location.
* Defined in the volumes: section of docker-compose.yml.
* **Persist across container rebuilds/removals**.
* Safer, preferred for production.

**🔹 How Volumes Work for SQL Server**

SQL Server stores its data under /var/opt/mssql.

To persist the data:

volumes:

- inventory-sqldata:/var/opt/mssql

This mounts a Docker-managed volume named inventory-sqldata to SQL Server’s data directory.  
When the container is destroyed and re-created, the volume survives and reattaches.

**🔹 1. What is a Volume?**

A **Docker volume** is a special type of **persistent storage managed by Docker**.

* It is **not stored inside the container filesystem**, but in a **separate part of the host's filesystem**, managed by Docker.
* Even if you **delete the container**, the volume still exists on the host, and **you can reuse it**.

Think of a **volume** as an **external hard drive managed by Docker**, automatically connected to your container.

**🔹 2. Types of Storage in Docker**

| **Type** | **Example** | **Where is the data stored?** | **Managed by** | **Suitable for** |
| --- | --- | --- | --- | --- |
| **Volume** | inventory-sqldata:/var/opt/mssql | In host’s Docker volume directory (e.g., Linux: /var/lib/docker/volumes) | Docker | Production use |
| **Bind mount** | C:/data:/var/opt/mssql | Host machine's C:\data directory | You (host user) | Dev (local editing) |
| **Container layer** | Default (no volume) | Inside container’s internal FS | Docker (ephemeral) | Not persistent (gone when container is deleted) |

**🔹 3. What Happens Without a Volume?**

If you don’t mount anything:

* SQL Server writes to /var/opt/mssql (inside the container).
* When you remove the container → 💥 **all data is lost** (because the container’s filesystem is deleted).

**🔹 4. What Happens With a Volume?**

If you mount a volume like this:

volumes:

- inventory-sqldata:/var/opt/mssql

Then:

* /var/opt/mssql in the container is **mapped to a volume** called inventory-sqldata.
* Docker stores that data **on the host**, usually under:
  + Linux: /var/lib/docker/volumes/inventory-sqldata/\_data
  + Windows: C:\ProgramData\Docker\volumes\inventory-sqldata\\_data
* Even if you delete the container, the volume is still there.
* If you run the container again and map the same volume, it gets the **same data** back.

This means, the volume named inventory-sqldata is **logically linked** to the container path /var/opt/mssql, and **physically stored** on the host (Linux: /var/lib/docker/volumes/inventory-sqldata/\_data).

SQL Server writes to /var/opt/mssql inside the container, but the actual files are **persisted on the host**, so data **survives even if the container is deleted**.

**🔹 5. Volume vs Bind Mount**

| **Feature** | **Volume** | **Bind Mount** |
| --- | --- | --- |
| Managed by Docker? | ✅ Yes | ❌ No (you manage the folder) |
| Portability | ✅ High (volume recreated by Docker) | ❌ Host-dependent path (like C:/data) |
| Use in Production? | ✅ Recommended | ❌ Not recommended |
| Use in Development? | ✅ Good (for data) | ✅ Good (for live code edits) |

**🔹 6. So Is Volume Physical or Logical?**

It is:

* **Logically defined** in docker-compose.yml.
* **Physically stored on host’s disk** in a hidden Docker-managed directory.

It’s not stored inside the container — that’s why it **survives even if the container is deleted**.

**🔹 7. Visual Illustration**

Host Machine

│

├─ Docker Volumes

│ └─ inventory-sqldata

│ └─ (holds .mdf, .ldf files from SQL Server)

│

├─ Containers

│ └─ inventory-sqlserver

│ └─ /var/opt/mssql → points to → inventory-sqldata

**✅ In Short**

* Volumes are **safe places** on your host machine managed by Docker.
* They are **used for persisting data** (like databases).
* When you write:

yaml

CopyEdit

volumes:

- inventory-sqldata:/var/opt/mssql

it means: “Store this folder’s contents in a special persistent area called inventory-sqldata”.

NOTE: To get a complete list of the Docker commands, refer Docker\_Command\_Cheat\_Sheet.pdf

**1. Ways to Set Environment Variables in Docker**

**A. Directly via docker run**

**docker run -e "DB\_SERVER=mydb" -e "DB\_USER=sa" my-image**

* -e sets an environment variable inside the container.
* Useful for quick overrides in development.
* Downside: Not good for secrets in production (shows up in command history).
* These variables **exist only inside the container** while it is running.
* Your **application code** inside the container can read them using its standard environment variable API.

For example:  
**In .NET Core**

var dbServer = Environment.GetEnvironmentVariable("DB\_SERVER");

var dbUser = Environment.GetEnvironmentVariable("DB\_USER");

**B. In docker-compose.yml**

yaml

CopyEdit

version: "3.9"

services:

inventory-service:

image: my-inventory-image

**environment:**

**- DB\_SERVER=inventory-db**

**- DB\_USER=sa**

**- DB\_PASSWORD=StrongPass123**

* Keeps variables grouped per service.
* Easier to maintain for multiple containers.

**C. Using .env Files**

docker-compose automatically reads a .env file in the same directory.

# .env

SERVER=inventory-db

USER=sa

PASSWORD=StrongPass123

docker-compose.yml:

yaml

CopyEdit

services:

inventory-service:

image: my-inventory-image

environment:

- DB\_SERVER=${SERVER}

- DB\_USER=${USER}

- DB\_PASSWORD=${PASSWORD}

**In .NET Core, this can be referenced using the environment variable set in docker-compose**

var dbServer = Environment.GetEnvironmentVariable("DB\_SERVER");

var dbUser = Environment.GetEnvironmentVariable("DB\_USER");

* Good for local development.
* **Don’t commit .env with secrets** to Git.

**D. Inside the Dockerfile**

ENV DB\_SERVER=inventory-db

ENV DB\_USER=sa

* Hardcodes default values into the image.
* Can still override at runtime with -e or docker-compose.yml.

## Setting defaults in the Dockerfile

When you write in your **Dockerfile**:

ENV DB\_SERVER=inventory-db

ENV DB\_USER=sa

You are **baking default values into the image**.  
This means — if nobody overrides these values at runtime, the container will have:

DB\_SERVER=inventory-db

DB\_USER=sa

## Overriding from docker-compose.yml

If your **docker-compose.yml** says:

environment:

- DB\_SERVER=${DATABASE}

- DB\_USER=${USER}

and .env contains:

DATABASE=prod-db

USER=admin

When you run:

docker-compose up

the container will get:

DB\_SERVER=prod-db

DB\_USER=admin

💡 The Dockerfile defaults are ignored because docker-compose explicitly set them

**Overriding from docker run**

If you start it with:

docker run -e DB\_SERVER=dev-db my-image

then DB\_SERVER=dev-db will override both the Dockerfile default and the .env file.

## Rule of precedence

Environment variable values are taken in this order (highest priority first):

1. **docker run -e** or **environment: in docker-compose**
2. Defaults in .env file (if referenced in docker-compose)
3. **ENV in Dockerfile** (lowest priority)

**Best Practices**

* Use .env for development; use secrets managers (Azure Key Vault, AWS Secrets Manager) for production.
* Avoid hardcoding secrets into images or Dockerfiles.
* Use docker-compose.override.yml for environment-specific overrides

**docker-compose.override.yml file**

It’s basically:

* Your **docker-compose.yml** is the base configuration (shared across all environments).
* **docker-compose.override.yml** automatically gets merged with it when you run docker-compose up — you don’t even have to specify it.
* It’s mainly used for:
  + Overriding environment variables (dev-specific values)
  + Changing volumes or ports for local dev
  + Adding debug tools only in dev
* Keeps your base config clean and production-ready, while letting you tweak it for development without touching the main file.

**Example**

**docker-compose.yml**

services:

web:

image: my-app

environment:

- NODE\_ENV=production

ports:

- "80:80"

**docker-compose.override.yml**

services:

web:

environment:

- NODE\_ENV=development

ports:

- "3000:80"

volumes:

- ./src:/app

When you run:

**docker-compose up**

the override file replaces and adds settings on top of the base file.

**Sharing and Tagging Docker Images**

## Why Share and Tag Docker Images?

When you build a Docker image on your local machine, **it only exists on your machine**.  
If you want that same image to run:

* On **another developer’s machine**
* On a **server** (staging, production, etc.)
* In a **cloud environment** (Kubernetes, AWS, Azure, etc.)

…you have to **share** it somehow.

**1. Sharing = Making the Image Available Elsewhere**

There are three main ways:

| **Method** | **When to Use** | **Example** |
| --- | --- | --- |
| **Push to a registry** (Docker Hub, AWS ECR, Azure ACR, etc.) | Common for teams, CI/CD, cloud deployment | docker push to Docker Hub |
| **Save & load offline** | No internet / secure environments | docker save → USB / file transfer |
| **Export container** | You want just the container’s state, not the build layers | docker export / docker import |

**2. Tagging = Giving the Image a Proper Name & Address**

When you **tag** an image, you’re essentially giving it:

* **A registry location** (Docker Hub, AWS, Azure…)
* **A repository name** (like a folder in that registry)
* **A version** (so you know which build it is)

**Without tagging**, your image will have a local name like:

microservicestarter-inventoryservice:latest

That name **means nothing** outside your PC.

But if you tag it like:

sriramdockers/inventoryservice:1.0

Docker now knows:

* Registry = Docker Hub (default if none specified)
* User = sriramdockers
* Repo = inventoryservice
* Version = 1.0

Now when you run the below, it knows where to send:

docker push sriramdockers /inventoryservice:1.0

**Steps to push the images in the Docker to the Docker Hub**

1. Create a Windows Powershell file (extension .ps1) . Note the commands in the file can also run independently, but we can efficiently run from file.

# ============================================================

# push-images.ps1

# PowerShell script to tag and push Docker images for

# 1. InventoryService (.NET Core Web API)

# 2. UserService (.NET Core Web API)

# 3. Inventory SQL Server (with DB + tables)

# 4. User SQL Server (with DB + tables)

#

# Images are tagged with your Docker Hub username and a

# dynamically generated version number for uniqueness.

# ============================================================

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

# Step 1 Set Docker Hub username

# Replace with your actual Docker Hub username.

# This will be used in image names like

# sriramdockersinventoryserviceversion

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

$DOCKER\_HUB\_USER = "sriramdockers"

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

# Step 2 Generate dynamic version tag based on date & time

# Format yyyyMMdd-HHmm

# Example 20250811-1420 = Aug 11, 2025 at 1420

# This ensures each push is uniquely tagged.

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

$VERSION = (Get-Date -Format "yyyyMMdd-HHmm")

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

# Step 3 Tag & push InventoryService image

# docker tag creates a new tag for an existing local image.

# Here we take the image built by docker-compose

# (microservicestarter-inventoryservice) and tag it for Docker Hub.

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

Write-Host Tagging InventoryService image...

docker tag microservicestarter-inventoryservice "$DOCKER\_HUB\_USER/inventoryservice:$VERSION"

Write-Host Pushing InventoryService image to Docker Hub...

docker push "$DOCKER\_HUB\_USER/inventoryservice:$VERSION"

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

# Step 4 Tag & push UserService image

# Same approach as InventoryService — tag locally, then push.

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

Write-Host Tagging UserService image...

docker tag microservicestarter-userservice "$DOCKER\_HUB\_USER/userservice:$VERSION"

Write-Host Pushing UserService image to Docker Hub...

docker push "$DOCKER\_HUB\_USER/userservice:$VERSION"

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

# Step 5 Commit & push Inventory SQL Server container

# docker commit takes a running container and saves its

# entire current state as a new image. This includes

# - SQL Server binaries

# - Databases and tables already created

# - Any other changes made inside the container

# We then tag that committed image for Docker Hub and push it.

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

Write-Host Tagging Inventory SQL Server container as a new image...

docker commit inventory-sqlserver "$DOCKER\_HUB\_USER/inventory-sqlserver:$VERSION"

Write-Host Pushing Inventory SQL Server image to Docker Hub...

docker push "$DOCKER\_HUB\_USER/inventory-sqlserver:$VERSION"

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

# Step 6 Commit & push User SQL Server container

# Same logic as Step 5, but for the user-sqlserver container.

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

Write-Host Tagging User SQL Server container as a new image...

docker commit user-sqlserver "$DOCKER\_HUB\_USER/user-sqlserver:$VERSION"

Write-Host Pushing User SQL Server image to Docker Hub...

docker push "$DOCKER\_HUB\_USER/user-sqlserver:$VERSION"

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

# Step 7 Completion message

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

Write-Host ✅ All images have been tagged with version $VERSION and pushed to Docker Hub.

1. Save the file (in our case push-images.ps1) in the folder where docker-compose.yml resides
2. Open Windows Powershell
3. Go to the folder where the .ps1 command is placed
4. Login to Docker Hub from Powershell as below:

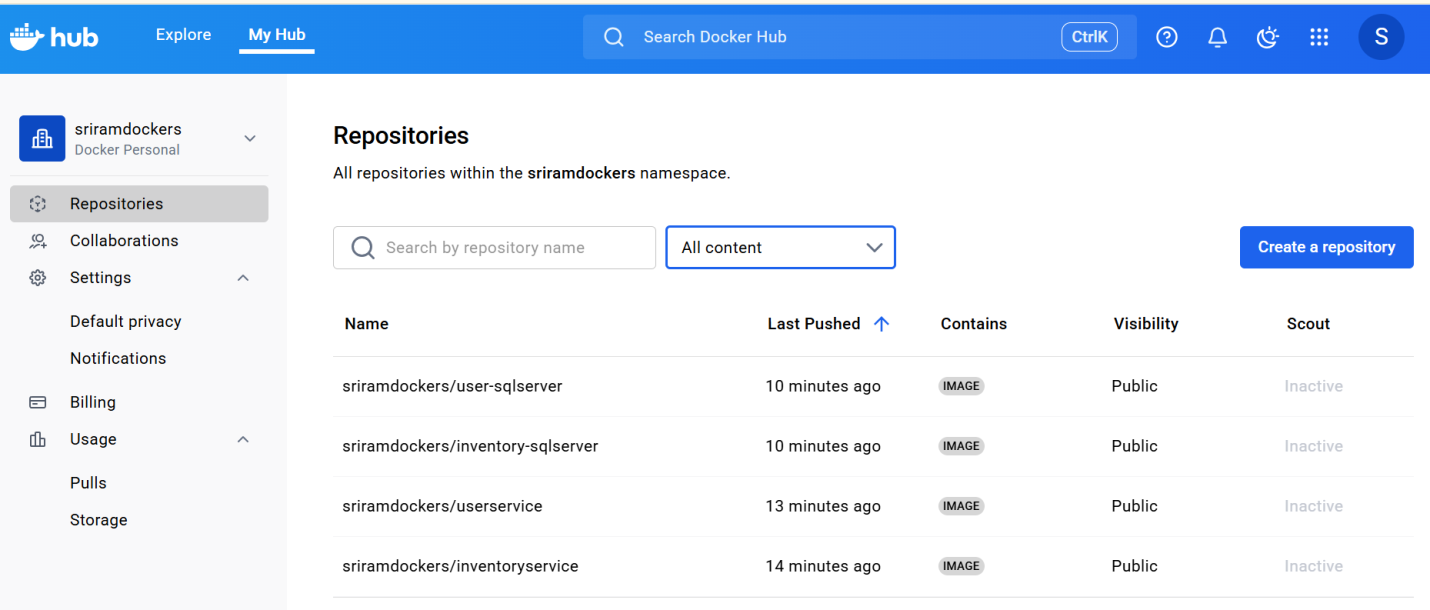
docker login

If already logged in, it will authenticate and login will succeed. Otherwise, follow the instructions on the screen.

1. Now run the Windows Powershell script from the folder where it is located. Our file name is push-images .ps1. Hence run the command as below:

.\push-images.ps1

1. Once the images are successfully pushed to Docker Hub, they will be available under Repositories in Docker Hub as shown in screen shot below:



1. The images that we pushed are in the Docker Hub repositories. These images can be pulled and run by anyone who has access to it.
2. The advantage of pushing to Docker Hub are:

 **Sharing** → Others can quickly get the exact same app & environment.

 **Deployment** → You can deploy to any cloud or server that has Docker installed.

 **Versioning** → You can keep multiple versions (e.g., :1.0, :1.1, :prod, :staging) without overwriting old ones.

So in short — **Docker Hub is your central storage for Docker images**, and pushing to it is like publishing your “app package” so it can be downloaded anywhere

**How to pull and run the images from the Docker Hub**

Once the images are pushed to Docker Hub, any user can pull the images from Docker Hub and can spin the images to get the containers:

Note: Here, the below commands are based on the public repository and for the images we have pushed to the Docker Hub above. :20250811-1624 is the version number we created for the image, as per the below comments we gave while running the script

*$VERSION = (Get-Date -Format "yyyyMMdd-HHmm")*

*docker tag microservicestarter-inventoryservice "$DOCKER\_HUB\_USER/inventoryservice:$VERSION"*

*docker push "$DOCKER\_HUB\_USER/inventoryservice:$VERSION"*

**Step 1 – Pull your image from Docker Hub**

docker pull sriramdockers/inventoryservice:20250811-1624

docker pull sriramdockers/userservice:20250811-1624

docker pull sriramdockers/inventory-sqlserver:20250811-1624

docker pull sriramdockers/user-sqlserver:20250811-1624

**Step 2 – Run the containers**

Since your images already include the application or committed SQL Server state, you can spin them up directly:

# Run Inventory SQL Server

docker run -d --name inventory-sqlserver -p 1433:1433 sriramdockers/inventory-sqlserver:20250811-1602

# Run Inventory Service (depends on the DB above)

docker run -d --name inventoryservice -p 8080:8080 sriramdockers/inventoryservice:20250811-1602

# Run User SQL Server

docker run -d --name user-sqlserver -p 1434:1433 sriramdockers/user-sqlserver:20250811-1602

# Run User Service

docker run -d --name userservice -p 8081:8081 sriramdockers/userservice:20250811-1602

**Step 3 – Verify containers are running**

docker ps

**How to push images to another Windows Server instead of Docker Hub?**

**Choose a transfer method. There are 2 methods here:**

1**. Running a private registry on the remote Windows server** (so you docker push / docker pull like Docker Hub)

**a. What is Docker Registry Container?**

* The **Docker Registry** is a service that stores and serves your Docker images.
* Docker Hub is a **public** registry, but you can also run your **own private registry** locally or on a server.
* When you run it **as a container**, it’s just a normal Docker image (called registry) that implements the registry API, so you can push and pull images to it just like Docker Hub, but it’s only accessible where you set it up (unless you make it public)

The illustration is as below:

(1) Windows Server

|-- Must have Docker Engine installed

|

(2) Pull "registry:2" image from Docker Hub

|

(3) Run "registry:2" image → Creates Docker Registry container

|-- This container stores images (private registry)

|

(4) Push user-created image (e.g., "inventory-service") to registry

|

(5) Pull image (e.g., "inventory-service") from registry when needed

|

(6) Run pulled image → Creates application container (e.g., inventory-service container)

**Testing Registry without Persistance**

**Step 1: Run local Docker registry container**

**docker run -d -p 5000:5000 --name registry registry:2**

 -p 5000:5000 → maps registry’s port 5000 to your host, so you can reach it via localhost:5000.

 --name registry → names the container "registry" for easy management.

 registry:2 → the official Docker Registry image (Version 2).

**Step 2: Tag an existing image for the registry**

Check the existing images, by running the command:

**docker images**

Choose the image from the list and tag and tag it so the registry knows it’s for localhost:5000. I am using the image microservicestarter-userservice:latest

**docker tag microservicestarter-userservice:latest localhost:5000/microservicestarter-userservice:1.0**

**Step 3: Push the image to the local registry**

**docker push localhost:5000/microservicestarter-userservice:1.0**

**Step 4: Verify the image is in your registry**

Open this URL in your browser:

<http://localhost:5000/v2/_catalog>

We can see as below:

{"repositories":["microservicestarter-userservice"]}

**Step 5: Pull the image from the registry**

**docker pull localhost:5000/microservicestarter-userservice:1.0** # Pull User Service v1.0 from local Docker registry at port 5000

 **localhost:5000** → Local Docker registry

 **microservicestarter-userservice** → Image name

 **:1.0** → Tag (version)

**Step 6: Run image and create controller**

**docker run -d --name userservice -p 8081:8081 localhost:5000/microservicestarter-userservice:1.0**

1. **direct file transfer** (docker save → copy → docker load). Private registries are great for ongoing team use; save/load is simplest for one-off transfers or air-gapped environments.

**Step 1:**

**docker save -o microservicestarter-inventoryservice.tar microservicestarter-inventoryservice:latest**

**This command create a tar file** microservicestarter-inventoryservice.tar of the image microservicestarter-inventoryservice from the source system

**Step 2:**

Transfer the .tar files created above to **External Drive/Shared Folder or WinSCP/PSCP for windows and SCP for Linux/Mac**

**Step 3:**

**Load the docker image on the target machine**

**docker load -i microservicestarter-inventoryservice.tar**

**Step 4:**

**Run containers on the target:**

**docker run -d --name microservicestarter-inventoryservice microservicestarter-inventoryservice:latest**

All the above commands that are run independently can be written in a Windows Powershell script file and executed.

# =====================================================================

# Script Name : push-pull-userservice.ps1

# Description : Push local UserService image to local registry, pull it, and run it from

# the registry.

# Author : Your Name

# Date : 2025-08-14

# =====================================================================

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

# Step 1: Start local registry (persistant storage of the images created in the Docker

# ContainerRegistry)

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

# Create a Docker volume to store registry data outside the container filesystem

docker volume create registry-data

# Remove any old registry container (if exists) so we can recreate it with the volume

docker rm -f registry 2>$null

# docker run:

# -d → Run container in detached mode (background)

# -p 5000:5000 → Map host port 5000 to container port 5000

# --name registry → Name the container "registry"

# -v registry-data:/var/lib/registry → Mount named volume to registry's internal storage path

# registry:2 → Use the official Docker registry image version 2

Write-Host "Starting local registry on port 5000..."

docker run -d -p 5000:5000 --name registry `

-v registry-data:/var/lib/registry `

registry:2

# Wait for registry to start (prevents "connection refused" when pushing immediately)

Write-Host "Waiting for registry container to initialize..."

Start-Sleep -Seconds 5 # Adjust if your system is slower

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

# Step 2: Tag local image for registry

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

# docker tag:

# Format: docker tag <source\_image>:<tag> <registry>/<image\_name>:<tag>

# Here we re-tag the existing local image so it can be pushed to our local registry

Write-Host "Tagging local microservicestarter-userservice:latest image..."

docker tag microservicestarter-userservice:latest localhost:5000/microservicestarter-userservice:1.0

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

# Step 3: Push image to registry

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

# docker push sends the image to the registry we specified in the tag

Write-Host "Pushing image to registry..."

docker push localhost:5000/microservicestarter-userservice:1.0

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

# Step 4: Verify registry contents

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

# Invoke-WebRequest:

# -Uri → The HTTP endpoint to call (here the registry API endpoint /v2/\_catalog)

# -UseBasicParsing → In older PowerShell versions (< 6.0), prevents Internet Explorer rendering and returns raw HTML/JSON.

#

# This specific registry API endpoint:

# http://localhost:5000/v2/\_catalog

# Returns JSON showing available repositories in the registry, e.g.:

# {"repositories":["microservicestarter-userservice"]}

#

# Select-Object -ExpandProperty Content:

# Extracts only the "Content" property from the HTTP response object

# so we see just the JSON text in the console.

Write-Host "Verifying registry contents..."

Invoke-WebRequest -Uri "http://localhost:5000/v2/\_catalog" -UseBasicParsing |

Select-Object -ExpandProperty Content

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

# Step 5: Pull image from registry

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

# docker pull fetches the image from the registry to ensure it can be retrieved successfully.

Write-Host "Pulling image back from registry..."

docker pull localhost:5000/microservicestarter-userservice:1.0

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

# Step 6: Run container from registry image

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

# docker run:

# -d → Run container in detached mode

# --name userservice → Name the container "userservice"

# -p 8081:8081 → Map host port 8081 to container port 8081

# localhost:5000/...:1.0 → Image location in the local registry

Write-Host "Running UserService container from registry image..."

docker run -d --name userservice -p 8081:8081 localhost:5000/microservicestarter-userservice:1.0

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

# Completion message

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

Write-Host "`nAll steps completed. UserService should now be running at <http://localhost:8081>"

To run the above script, go to the folder where the script is saved from Windows Powershell. Then type the below to execute the script file:

**.\push-pull-userservice.ps1**

**Set up the remote registry (optional)**  
On the Windows server you can run the official registry:2 container (or Harbor/ACR) and secure it with TLS and authentication. Once the registry is ready, tag images with the registry host (e.g., myserver:5000/myimage:tag) and push them just like to Docker Hub.

**Save & transfer images (offline/direct)**  
Locally run docker save to write the image(s) to a tar file, then transfer via SCP/SMB/USB to the remote server and run docker load there. This method transfers images exactly, requires no registry, and is useful when networks or policies prevent direct pushes.

**Handling DB-containing containers**  
For containers with live DB state you can either docker commit to snapshot the container (quick but large and less reproducible) or **export the DB** (e.g., sqlcmd dump) and recreate it on the remote server (cleaner and versionable). Prefer DB dumps + init scripts for reproducibility; use commit/save only for exact environment snapshots.

**Networking, ports, and security**  
If you use a registry, open and optionally firewall the registry port (commonly 5000) and configure TLS certs; if you skip TLS, configure the Docker daemon’s insecure-registries on the client/host. For direct transfers you just need an authenticated copy mechanism (SCP/SMB) and proper permissions on the Windows server.

**Run the images on the remote server**  
After the image is on the remote host (via push/pull or load), run containers with the same docker run/docker-compose up commands, ensuring environment variables, port mappings, and volumes match what the services expect. If you used a registry, update your docker-compose.yml to reference the registry hostname or the loaded local image names.

**Best practices & cleanup**  
Tag every image with meaningful version tags (build or timestamp), keep images small (multi-stage builds, remove dev deps), and prefer DB dumps + init scripts for portability. Automate recurring transfers (scripts or CI) and remove large intermediate images when done to save disk space.