**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