SE Lab 10 – Software Deployment - Contents

I. Software Deployment

Introduction to Software Deployment

Software deployment is the process of delivering a software application or system to its intended users and making it operational in a specific environment. This critical phase in the software development lifecycle ensures that the software is correctly installed, configured, and ready for use. Deployment can range from simple installations on a single machine to complex rollouts across multiple servers and environments.

Evolution to Containerization

As software systems grew more complex, traditional deployment methods faced challenges such as dependency conflicts, inconsistent environments, and scalability issues. This led to the evolution of containerization, a technology that packages an application and its dependencies into a single, portable unit called a container.

What are Containers?

Containers are lightweight, portable, and self-sufficient units that package an application and its dependencies together. They ensure that the application runs consistently across different environments by isolating it from the underlying infrastructure. Containers are built on top of a container runtime, such as Docker, which provides the necessary tools to create, manage, and run containers.

What is Docker?

Docker is a leading containerization platform that simplifies the process of creating, deploying, and managing containers. It provides a consistent environment for applications, ensuring they run the same way regardless of where they are deployed. Docker containers are lightweight, share the host OS kernel, and include everything needed to run the application, such as code, runtime, libraries, and configuration files.

In short, **Docker** has revolutionized software deployment by providing a consistent, portable, and efficient way to package and run applications. It addresses many of the challenges associated with traditional deployment methods and supports modern development practices, making it an essential tool for today's software development and deployment processes.

II. ASP.NET Core with Docker Support

ASP.NET Core with Docker Support

ASP.NET Core is a cross-platform, high-performance framework for building modern, cloud-based, and internet-connected applications. It is designed to be lightweight and modular, making it an excellent choice for containerization. Docker support in ASP.NET Core allows developers to easily

package their applications into containers, ensuring consistent deployment and execution across various environments.

Why Use Containers with ASP.NET Core?

1. **Consistency Across Environments**: Containers encapsulate the application and its dependencies, ensuring that it runs the same way in development, testing, and production environments.
2. **Portability**: Containers can run on any system that supports the container runtime, making it easy to move applications between different environments or cloud providers.
3. **Isolation**: Containers provide process and resource isolation, ensuring that applications do not interfere with each other, which enhances security and stability.
4. **Scalability**: Containers can be easily scaled up or down to handle varying loads, making them ideal for microservices architectures and cloud-native applications.

Benefits of Using Docker with ASP.NET Core

1. **Simplified Deployment**: Docker simplifies the deployment process by packaging the application and its dependencies into a single container image. This eliminates the "it works on my machine" problem.
2. **Efficient Resource Utilization**: Containers are lightweight and share the host OS kernel, which allows for efficient use of system resources compared to traditional virtual machines.
3. **Rapid Development and Testing**: Docker enables rapid development and testing by allowing developers to quickly spin up and tear down containers. This accelerates the development cycle and improves productivity.
4. **Version Control and Rollbacks**: Docker images can be versioned, allowing for easy rollbacks to previous versions if needed. This enhances the ability to manage application updates and deployments.
5. **Integration with CI/CD Pipelines**: Docker integrates seamlessly with continuous integration and continuous deployment (CI/CD) pipelines, automating the build, test, and deployment processes. This ensures faster and more reliable releases.

By leveraging Docker with ASP.NET Core, developers can build, ship, and run applications more efficiently, ensuring consistency, portability, and scalability across different environments

Exercise 1: Setting up ASP.NET Core applications (both Razor Pages and MVC) to support Docker deployments

Step 1: Open ASP.NET Core Application

- Open Visual Studio.

- Open either "Web Application (Model-View-Controller)" or "Web Application" (for Razor Pages) (Lab 7, Lab 8)

Step 2: Add Docker Support:

- Right-click on the project in Solution Explorer and select "Add" > "Docker Support".
- Choose "Linux" as the target OS.
- Visual Studio will add a Dockerfile to your project.

Note: Understanding the Dockerfile

- Open the Dockerfile and review its contents. It typically includes instructions to:
  - Use a base image (e.g., mcr.microsoft.com/dotnet/aspnet:6.0).
  - Copy the application files.
  - Restore dependencies and build the application.
  - Expose a port and define the entry point.

Step 3: Building and Running the Docker Image

1. **Build the Docker Image**:
   - Open a terminal in the project directory.
   - Run the following command to build the Docker image:

     ```
     docker build -t myaspnetcoreapp
     ```

2. **Run the Docker Container**:
   - Run the following command to start a container from the image:

     ```
     docker run -d -p 8080:80 --name myaspnetcoreappcontainer myaspnetcoreapp
     ```

   - Open a web browser and navigate to **http://localhost:8080** to see your application running in Docker.

Step 4: Deploying to Docker Hub

1. **Tag the Docker Image**:
   - Tag the image with your Docker Hub username and repository name:

     docker tag myaspnetcoreapp yourdockerhubusername/myaspnetcoreapp

2. **Push the Image to Docker Hub**:
   - Push the image to Docker Hub:

     docker push yourdockerhubusername/myaspnetcoreapp

Exercise 2: Deploy your exercises with Docker Desktop on MS Window 10/11

These steps should help you get started with running your Docker files in Docker Desktop on Windows

Step 1: Install Docker Desktop

1. **Download and Install Docker Desktop**:
    - Go to the Docker Desktop website and download the installer for Windows.
    - Run the installer and follow the instructions to complete the installation.
    - After installation, start Docker Desktop and ensure it is running.

Step 2: Verify Docker Installation

1. **Open Command Prompt or PowerShell**:
    - Open Command Prompt or PowerShell.
    - Run the following command to verify Docker is installed correctly:
    - docker --version
    - You should see the Docker version information.

Step 3: Build the Docker Image

1. **Navigate to Your Project Directory**:
    - Use Command Prompt or PowerShell to navigate to the directory containing your Dockerfile.
2. **Build the Docker Image**:
    - Run the following command to build the Docker image:

    docker build -t myaspnetcoreapp .

Step 4: Run the Docker Container

1. **Run the Docker Container**:
    - Run the following command to start a container from the image:

    docker run -d -p 8080:80 --name myaspnetcoreappcontainer myaspnetcoreapp

2. **Verify the Application is Running**:
    - Open a web browser and navigate to http://localhost:8080 to see your application running in Docker.

Step 5: Managing Docker Containers

1. **List Running Containers**:
    - Run the following command to list all running containers:

    ```
    docker ps
    ```

2. **Stop a Running Container**:
    - Run the following command to stop the container:

    ```
    docker stop myaspnetcoreappcontainer
    ```

3. **Remove a Container**:
    - Run the following command to remove the container:

    ```
    docker rm myaspnetcoreappcontainer
    ```

Step 6: Pushing to Docker Hub

1. **Log in to Docker Hub**:
    - Run the following command to log in to Docker Hub:

    ```
    docker login
    ```

    - Enter your Docker Hub username and password.

2. **Tag the Docker Image**:
    - Tag the image with your Docker Hub username and repository name:
    - docker tag myaspnetcoreapp yourdockerhubusername/myaspnetcoreapp

3. **Push the Image to Docker Hub**:
    - Push the image to Docker Hub:

    ```
    docker push yourdockerhubusername/myaspnetcoreapp
    ```

Step 7: Pulling and Running from Docker Hub

1. **Pull the Docker Image**:
    - Run the following command to pull the image from Docker Hub:

    ```
    docker pull yourdockerhubusername/myaspnetcoreapp
    ```

2. **Run the Docker Container**:
    - Run the following command to start a container from the pulled image:

    ```
    docker run -d -p 8080:80 --name myaspnetcoreappcontainer
    yourdockerhubusername/myaspnetcoreapp
    ```

Exercise 3: Deploy a Docker container to an Ubuntu server

This exercise should help you deploy your Docker container to an Ubuntu server.

Step 1: Setting Up the Ubuntu Server

1. **Install Docker on Ubuntu**:
    - SSH into your Ubuntu server.
    - Update the package index:

      sudo apt-get update
    - Install Docker:

      sudo apt-get install -y docker.io
    - Start Docker and enable it to start on boot:

      sudo systemctl start docker

      sudo systemctl enable docker
2. **Verify Docker Installation**:
    - Run the following command to verify Docker is installed correctly:

      docker --version

Step 2: Transfer Docker Image to Ubuntu Server

1. **Save the Docker Image**:
    - On your local machine, save the Docker image to a tar file:

      docker save -o myaspnetcoreapp.tar myaspnetcoreapp
2. **Transfer the Docker Image**:
    - Use scp to transfer the tar file to your Ubuntu server:
    - scp myaspnetcoreapp.tar username@your_ubuntu_server_ip:/path/to/destination

Step 3 : Load and Run the Docker Image on Ubuntu

1. **Load the Docker Image**:
    - SSH into your Ubuntu server.
    - Load the Docker image from the tar file:

      docker load -i /path/to/destination/myaspnetcoreapp.tar
2. **Run the Docker Container**:

- Run the following command to start a container from the image:

```
docker run -d -p 8080:80 --name myaspnetcoreappcontainer myaspnetcoreapp
```

3. **Verify the Application is Running**:
    - Open a web browser and navigate to http://your_ubuntu_server_ip:8080 to see your application running in Docker.

Special Case: Automating Deployment with Docker Compose

1. **Create a docker-compose.yml File**:
    - On your Ubuntu server, create a docker-compose.yml file:

```
version: '3.8'

services:

 web:

  image: myaspnetcoreapp

  ports:

   - "8080:80"
```

2. **Deploy with Docker Compose**:
    - Run the following command to start the services defined in the docker-compose.yml file:

```
sudo docker-compose up -d
```

Step 4: Managing Docker Containers

1. **List Running Containers**:
    - Run the following command to list all running containers:

```
docker ps
```

2. **Stop a Running Container**:
    - Run the following command to stop the container:

```
docker stop myaspnetcoreappcontainer
```

3. **Remove a Container**:
    - Run the following command to remove the container:

```
docker rm myaspnetcoreappcontainer
```

Exercise 4: Automating deployment with CI/CD

Automating deployment with CI/CD (Continuous Integration/Continuous Deployment) helps streamline the process of building, testing, and deploying applications. Here's a step-by-step guide to setting up CI/CD for deploying Docker containers:

Step 1: Set Up a Version Control System

1. **Use GitHub, GitLab, or Bitbucket**:
    - Ensure your project is hosted on a version control platform like GitHub, GitLab, or Bitbucket.

Step 2: Create a CI/CD Pipeline

1. **GitHub Actions**:
    - Create a .github/workflows directory in your project.
    - Add a workflow file (e.g., docker-image.yml) to define the CI/CD pipeline.

```
name: Docker Image CI

on:
 push:
  branches: [ main ]
 pull_request:
  branches: [ main ]

jobs:
 build:
  runs-on: ubuntu-latest

  steps:
  - name: Checkout code
   uses: actions/checkout@v2

  - name: Set up Docker Buildx
```

```
      uses: docker/setup-buildx-action@v1

    - name: Log in to Docker Hub
      uses: docker/login-action@v1
      with:
        username: ${{ secrets.DOCKER_USERNAME }}
        password: ${{ secrets.DOCKER_PASSWORD }}

    - name: Build and push
      uses: docker/build-push-action@v2
      with:
        push: true
        tags: yourdockerhubusername/myaspnetcoreapp:latest
```

2. **GitLab CI/CD**:
   - Create a .gitlab-ci.yml file in your project root.

```
stages:
  - build
  - deploy

build:
  stage: build
  script:
    - docker build -t yourdockerhubusername/myaspnetcoreapp:latest .
    - docker login -u "$CI_REGISTRY_USER" -p "$CI_REGISTRY_PASSWORD"
    - docker push yourdockerhubusername/myaspnetcoreapp:latest

deploy:
  stage: deploy
  script:
```

- docker pull yourdockerhubusername/myaspnetcoreapp:latest

- docker run -d -p 8080:80 --name myaspnetcoreappcontainer yourdockerhubusername/myaspnetcoreapp:latest

Step 3: Configure Secrets

1. **Store Docker Hub Credentials**:
    - In GitHub, go to your repository settings and add secrets for DOCKER_USERNAME and DOCKER_PASSWORD.
    - In GitLab, add variables for CI_REGISTRY_USER and CI_REGISTRY_PASSWORD in your project settings.

Step 4: Trigger the Pipeline

1. **Push Code to Repository**:
    - Push your code changes to the main branch or create a pull request.
    - The CI/CD pipeline will automatically trigger, build the Docker image, push it to Docker Hub, and deploy it to your server.

Step 5: Monitor and Maintain

1. **Monitor Pipeline Runs**:
    - Check the status of your pipeline runs in GitHub Actions or GitLab CI/CD.
    - Address any build or deployment failures promptly.
2. **Update and Maintain**:
    - Regularly update your CI/CD pipeline configuration as your project evolves.
    - Ensure your Docker images and dependencies are up-to-date.

By setting up a CI/CD pipeline, you can automate the process of building, testing, and deploying your Docker containers, ensuring a consistent and reliable deployment process.

Exercise 5: Deploying to Microsoft Azure

Step 1: Set Up Azure Account

1. **Create an Azure Account**:
    - If you don't have an Azure account, sign up at Azure.

Step 2: Create an Azure Container Registry

1. **Create a Container Registry**:
    - Go to the Azure portal.
    - Click on "Create a resource" and search for "Container Registry".
    - Click "Create" and fill in the required details (e.g., resource group, registry name).

Step 3: Push Docker Image to Azure Container Registry

1. **Log in to Azure**:
    - Use the Azure CLI to log in:
    - az login
2. **Tag and Push the Docker Image**:
    - Tag your Docker image for the Azure Container Registry:
    - docker tag myaspnetcoreapp <your_registry_name>.azurecr.io/myaspnetcoreapp:latest
    - Push the image to the registry:
    - docker push <your_registry_name>.azurecr.io/myaspnetcoreapp:latest

Step 4: Deploy to Azure App Service

1. **Create an App Service**:
    - In the Azure portal, click on "Create a resource" and search for "App Service".
    - Click "Create" and fill in the required details.
    - Under "Docker" settings, select "Single Container" and configure it to use the image from your Azure Container Registry.
2. **Configure and Deploy**:
    - Complete the configuration and click "Review + create".
    - Once the App Service is created, it will automatically pull the Docker image and deploy it.

Exercise 6: Deploying to Amazon Web Services (AWS)

Step 1: Set Up AWS Account

1. **Create an AWS Account**:

- If you don't have an AWS account, sign up at AWS.

Step 2: Create an Amazon ECR Repository

1. **Create a Repository**:
    - Go to the AWS Management Console.
    - Navigate to the Amazon **ECR (Elastic Container Registry)** service.
    - Click "Create repository" and fill in the required details.

Step 3: Push Docker Image to Amazon ECR

1. **Log in to AWS**:
    - Use the AWS CLI to log in:
    - aws configure
2. **Tag and Push the Docker Image**:
    - Tag your Docker image for the ECR repository:
    - docker tag myaspnetcoreapp <aws_account_id>.dkr.ecr.<region>.amazonaws.com/myaspnetcoreapp:latest
    - Push the image to the repository:
    - aws ecr get-login-password --region <region> | docker login --username AWS --password-stdin <aws_account_id>.dkr.ecr.<region>.amazonaws.com
    - docker push <aws_account_id>.dkr.ecr.<region>.amazonaws.com/myaspnetcoreapp:latest

Step 4: Deploy to AWS Elastic Beanstalk

1. **Create an Elastic Beanstalk Environment**:
    - In the AWS Management Console, navigate to the Elastic Beanstalk service.
    - Click "Create a new environment" and select "Web server environment".
    - Choose "Docker" as the platform and configure it to use the image from your ECR repository.
2. **Configure and Deploy**:
    - Complete the configuration and click "Create environment".
    - Elastic Beanstalk will automatically pull the Docker image and deploy it.

Special exercise: Deploying a Website to a Production Environment

Objective: Deploy any website **(Lab Assignment 2, or Lab7/8)** to a production environment using a free or paid hosting service.

Instructions:

1. **Choose a Hosting Service**:
    - Select a hosting service that fits your needs. Some popular options include:
        - **Free Hosting**: GitHub Pages, Netlify, Vercel, Heroku (free tier)
        - **Paid Hosting**: AWS, Azure, Google Cloud, DigitalOcean, Bluehost
2. **Prepare Your Website**:
    - Ensure your website is ready for deployment. This includes:
        - Finalizing the design and functionality.
        - Testing the website locally to ensure it works as expected.
        - Optimizing the website for performance (e.g., minifying CSS/JS, optimizing images).
3. **Deploy the Website**:
    - Follow the hosting service's documentation to deploy your website. This typically involves:
        - Creating an account on the hosting platform.
        - Setting up a new project or site.
        - Uploading your website files or connecting your repository (for services like GitHub Pages, Netlify, Vercel).
        - Configuring any necessary settings (e.g., environment variables, build settings).
4. **Verify Deployment**:
    - After deployment, verify that your website is live and functioning correctly.
    - Test the website on different devices and browsers to ensure compatibility.
5. **Document the Process**:
    - Create a detailed report documenting the deployment process. Include:
        - The hosting service you chose and why.

- Steps you followed to deploy the website.
- Any challenges you faced and how you resolved them.
- Screenshots of key steps and the final deployed website.

6. **Submit Your Work**:
    - Submit the following together with Lab10 Report:
        - A link to the live website.
        - The detailed report in PDF format.
        - A zip file containing the website's source code.

Evaluation Criteria to add point/ replace points of lacking submitting the previous Lab:

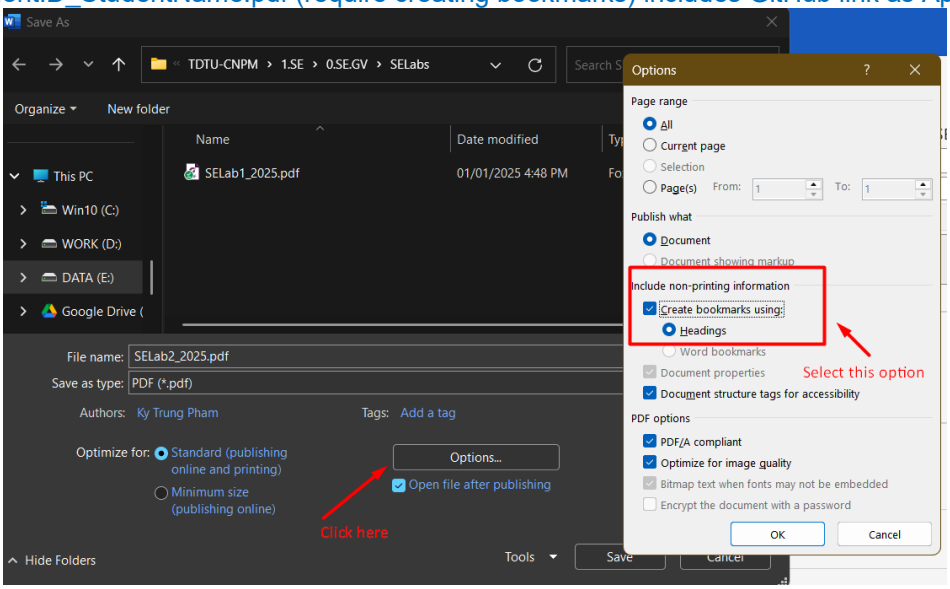
- **Functionality**: The website should be fully functional and accessible.
- **Documentation**: The report should be clear, detailed, and well-organized.
- **Problem-Solving**: Demonstrate how you addressed any challenges during the deployment process.
- **Presentation**: The final website should be visually appealing and user-friendly.

Submit on eLearning: Summary Report

- **Introduction**: Briefly describe the purpose of the exercises.
- **Exercises**: List each exercise with a brief description of what you did and learned.
- **Screenshots**: Include the screenshots of every step for each exercise and steps (both Visual Studio and Github) (Index each step with sequential number) and the output when press F5.
- **Conclusion**: Summarize your overall learning experience and any challenges you faced.

Submit 2 files on eLearning:

- StudentID_StudentName.zip (contain Windows Web Project & .sql file) for both Class Activity and Homework.
- StudentID_StudentName.pdf (require creating bookmarks) includes GitHub link as Appendix



Notes: There may be a typo/mistake during creating this document. Please correct it by yourself.