Docker für ASP.NET Core Anwendungen: Best Practices von der Entwicklung bis zur Produktion

Marc Müller
Principal Consultant

marc.mueller@4tecture.ch
@muellermarc
www.4tecture.ch

4tecture©
empower your software solutions

About me:

Marc Müller
Principal Consultant
@muellermarc

MVP
Microsoft Most Valuable Professional

4tecture©
empower your software solutions

Our Products:

Multi-Tenant OpenID Connect Identity Provider

ProAuth

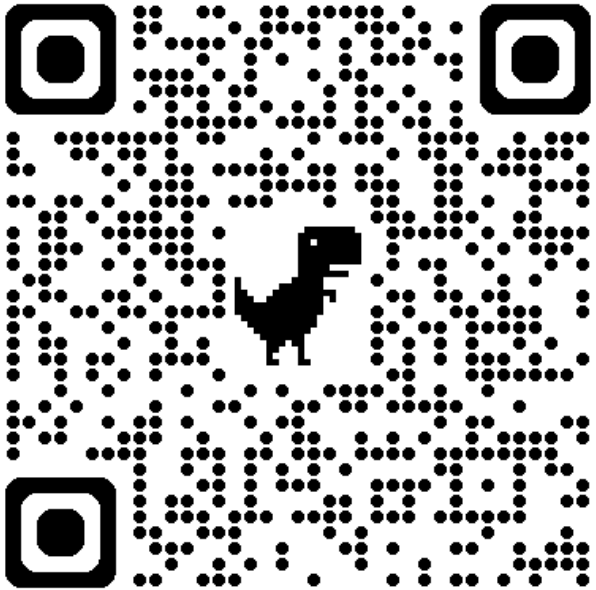
www.proauth.net

Enterprise Application Framework for .NET

ReaFx

www.reafx.net

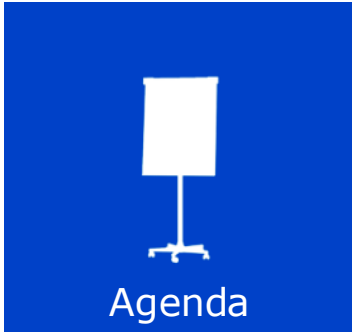
Slide Download



https://www.4tecture.ch/events/basta25springdockeraspnetcore

Agenda

- Intro
- Building Containers
- IDE Integration & Multi-Service Development
- Q&A



Docker for ASP.NET Core Developers

Intro

4tecture
empower your software solutions

Challenges with traditional development

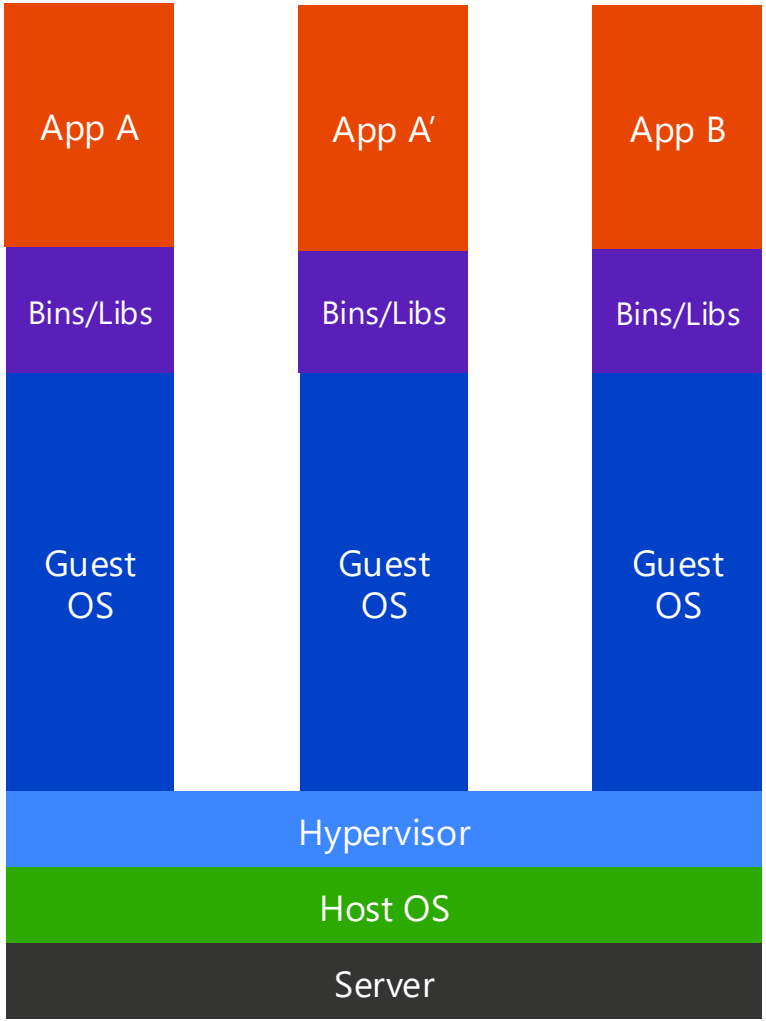
- Environment inconsistencies (dev, test, prod)
- Dependency and configuration conflicts
- Complex deployment and scaling processes
- Security and resource isolation issues

How Containers Solves These Challenges

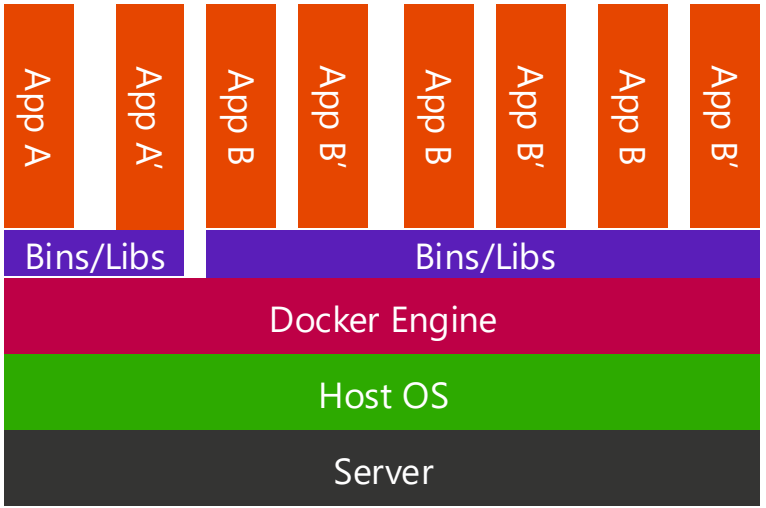
- **Consistency**
  Same container image across all environments
- **Isolation**
  Containers encapsulate dependencies and runtime
- **Portability**
  Run containers anywhere (local, cloud, hybrid)
- **Efficiency**
  Simplified scaling and continuous integration

Containers

- Versioned artifact
- Isolated deployable unit
- Container image is bit by bit identical when deployed
- Abstraction of data center resources
  - Pool of compute, network and storage
- Orchestration is “Cattle Business”
  - “Forget about naming the servers and treating them like pets.”
  - “Run it for me please”



Containers are isolated, but share OS and, where appropriate, bins/libraries



https://sec.ch9.ms/sessions/build/2016/B822.pptx

Docker Layers

My ASP.NET Core Application

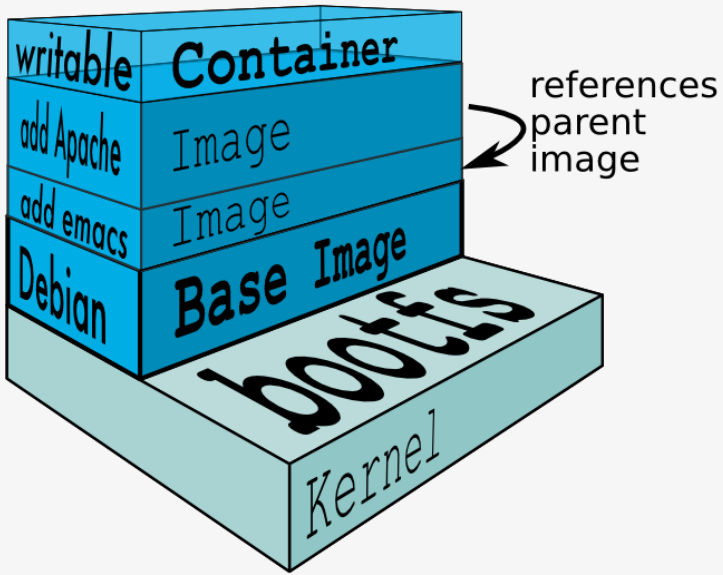
mcr.microsoft.com/dotnet/core/aspnet:9.0

mcr.microsoft.com/dotnet/core/runtime

mcr.microsoft.com/dotnet/core/runtime-deps

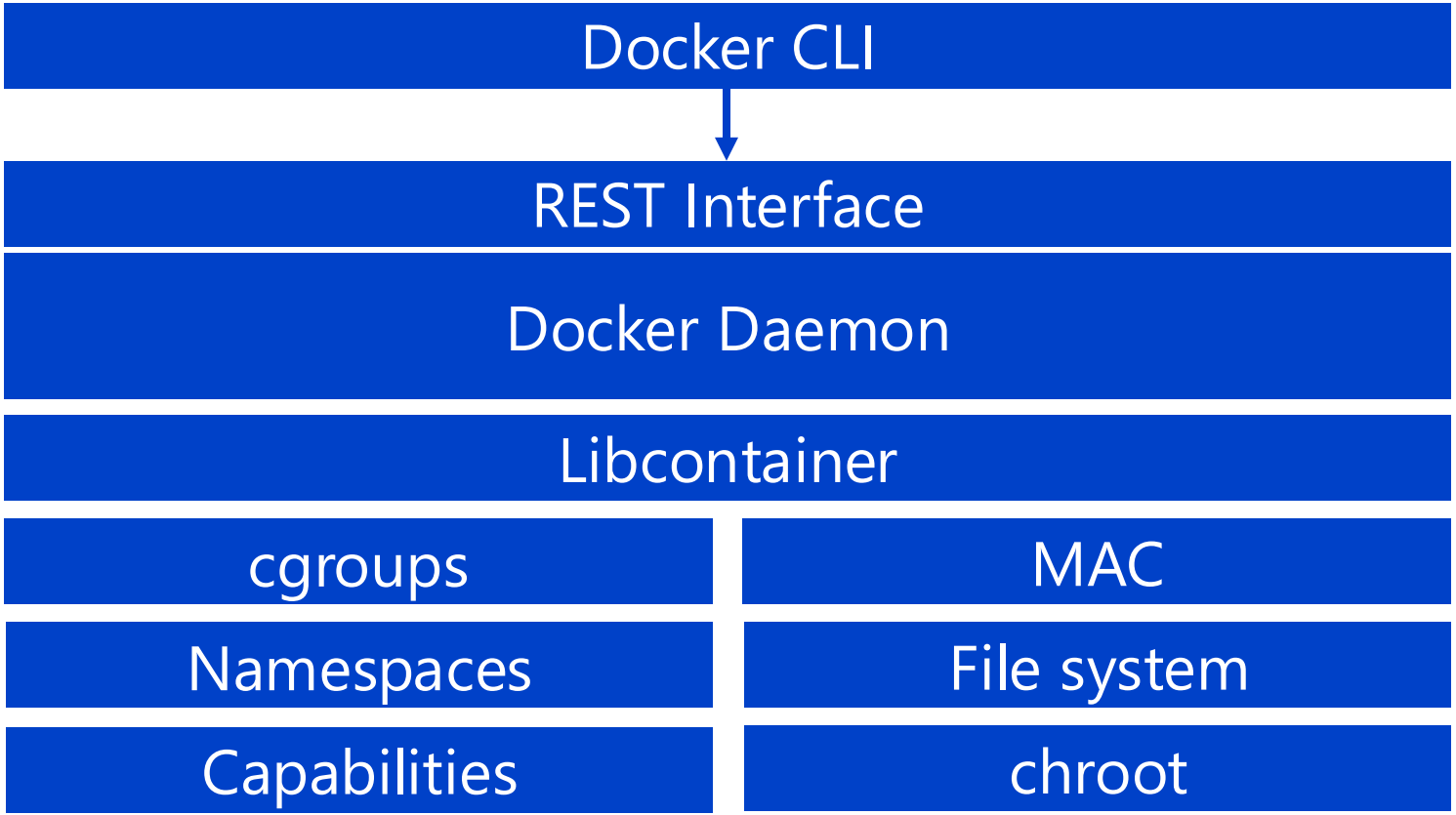
amd64/debian:bookworm

File System Layers



- Rootfs stays read-only
- Union-mount file system over the read-only file system
- Multiple file systems stacked on top of each other
- Only top-most file system is writable
- Copy-on-write

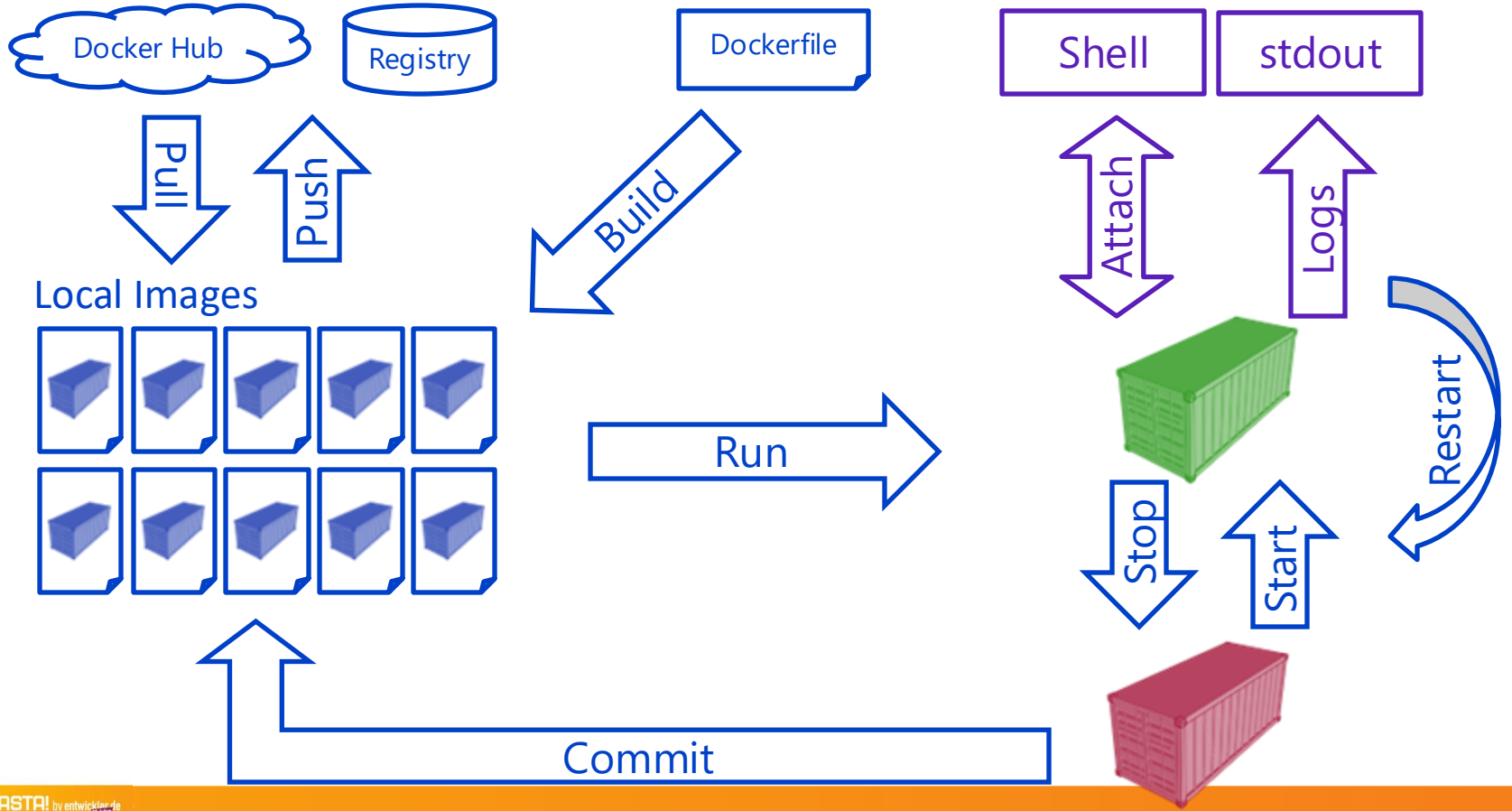
Docker Architecture



Definitions

- Container
  A container defines a software application and its dependencies wrapped in a complete filesystem including code, runtime, system tools, and libraries.
- Image
  An image is a read-only snapshot of a Docker container.

Docker Lifecycle



Docker

Build Time

- Dockerfile as definition
- Creates a read-only image
- Normally uses a base image
- Builds a docker file system by executing commands and processes

Runtime

- Instanciate a container from an image
- Has isolated CPU, RAM, Disk and Network
- Runs processes

Demo

Sample Application

Docker for ASP.NET Core Developers

Building Containers

4tecture
empower your software solutions

Challenges: Building and running Containers

- **Inconsistent Build Environments**
  Different compilers and development setups
- **Overly Large & Insecure Images**
  Unnecessary dependencies in runtime images
- **Inefficient Build Processes**
  Lack of caching and redundant steps

Layer Caching

- Docker creates images using layers
- Each command in Dockerfile creates a new layer
- Each layer contains the filesystem changes
  (diff between before and after command)
- To build images faster, docker uses layer caches
- Works with RUN, COPY, ADD

Images and Layers

```
FROM ubuntu:18.04
COPY . /app
RUN make /app
CMD python /app/app.py
```

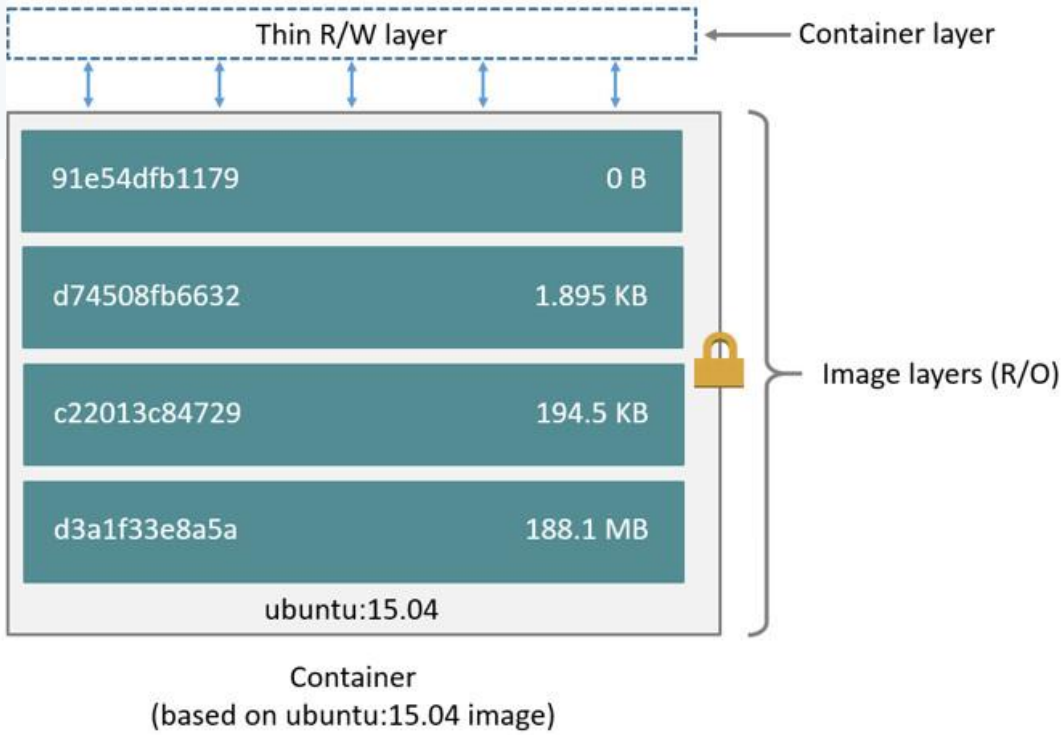


Image source: https://docs.docker.com/storage/storagedriver/#images-and-layers

Layer Caching - RUN

- RUN executes a command in the Docker image
- If the layer generated by the run command already exists, it is reused, and the command is not executed

- Cache Hash: the command itself

Layer Caching - COPY

- COPY imports one or more external files into a Docker image
- If the contents of all the external files are the same, the layer cache is used

- Cache Hash: hash on all the file contents

Layer Caching - ADD

- ADD imports one or more external files into a Docker image
- If the contents of all the external files are the same, the layer cache is used

- Cache Hash: hash on all the file contents

Note: COPY vs ADD → ADD allows external URLs and can extract tar files to the image

Taking Advantage of Layer Caching

- If a layer has a cache miss, then all subsequent layers will be built and are not cached
- Structure your Dockerfile in a way to get most out of the caching and therefore build performance

Demo

Dockerfile Basics

- Multi Stage
- Caching

Demo Recap - Dockerfile

- **Clear Separation of Concerns**
  Isolate build and runtime environments
- **Optimized Dockerfile Practices**
  Multi-stage builds to reduce image size
- **Efficient Layer Caching**
  Minimize rebuild times with smart file copying

Challenges: Image Size

- **Impact on Deployments**
  Larger images slow down downloads and startup times
  Increased resource usage during storage and transfer
- **Variability by Base Image**
  Standard, Alpine, and Chiseled images differ greatly in size


Demo
Image Size

Demo Recap – Image Size

- ## Optimization Strategies
  Tailor your base image choice to your application’s needs
- ## Balancing Trade-offs
  Consider features versus size – minimal images for runtime efficiency
  Multi-Language Support (ICU)
  AOT Compilation – no dynamic loading, no reflection
- ## Base image is also relevant for security
  Hardening, dependencies / tools, distro-less

Challenges: Dependencies of Base Images

- **Dependency Overload**
  Base images include numerous libraries and dependencies
- **Risk Exposure**
  Each dependency may have vulnerabilities
- **Need for Proactive Security**
  Continuous monitoring and tracking are essential


Demo
Security Scans

Demo Recap – Security Scan

- ## SBOM: Your Dependency Inventory
  Generate a Software Bill of Materials to list all components
- ## Automated Vulnerability Scanning
  Utilize Docker Scout and Trivy for security analysis
- ## Proactive Security Management
  Identify and remediate issues before deployment
  Integrate security scans in your CI builds as well as run them periodically.

Challenges: Container Security

- ## Challenge: Elevated Risks

  Containers running as root with open permissions
  Multiple dependencies increase the attack surface

- ## Not all base images are secure by default

  Most base images run with root user
  Large images with a lot dependencies

Demo

Container Hardening

Demo Recap – Hardening

- ## Hardened Base Image

  Improved security with adjusted file permissions and non-root user
  Offers balance: enhanced security while still allowing debugging tools

- ## Chiseled Image

  Distroless design minimizes extraneous components
  Focus on maximum security and performance for production

Challenges: Important artifacts in intermediate images

- **More artifacts than just the application**

  Build produce test results, database schemas, client packages, etc.

  The final image does not contain them, but they are needed for the process.

- **Intermediate image identification**

  We need to identify intermediate images and extract artifacts from them.


Demo
Test Results Extraction

Demo Recap – Extract artifacts

- ## Label Images

  Images – also intermediate images – can be labelled and therefore easily be identified.

  Unique label values are needed if multiple builds are performed on the same environment.

- ## Copy files between Container and Host

  Files can be copied from the container to the host.

  The host (i.e. CI build) can then work with those files.

Challenges: Secrets in Build

- ## Private Feeds

  Private feeds need authentication.
  The docker build does not know anything about the outer environment

- ## Use secrets in container build

  Make sure the secrets are not persisted in the produced image.

Demo

Secrets in Build

Demo Recap – Secrets

- ## Multi-Stage Build

  Make sure that no build related files and configs make it into the target image

- ## Use build-args

  Provide arguments for the build which could contain secrets

- ## Use Secrets

  Credentials are injected at build time, never stored in the final image.

  Use the --secret option to securely access private feeds (e.g., NuGet) without hardcoding sensitive data.

  Eliminates risk of exposing authentication details during runtime or in image layers.

  Adheres to best practices for secure CI/CD pipelines by separating code and secrets.

Docker for ASP.NET Core Developers

IDE Integration & Multi-Service Development

4tecture
empower your software solutions

IDE Integration (Visual Studio)

- Seamless Integration
- Fast Mode Debugging
- Full Developer Experience
- Configurable Options

Container Tools Build Properties

Property name	Description
ContainerDevelopmentMode	Controls whether "build-on-host" optimization ("Fast Mode" debugging) is enabled. Allowed values are **Fast** and **Regular**.
ContainerVsDbgPath	The path for VSDBG debugger.
DockerDebuggeeArguments	When debugging, the debugger is instructed to pass these arguments to the launched executable.
DockerDebuggeeProgram	When debugging, the debugger is instructed to launch this executable.
DockerDebuggeeKillProgram	This command is used to kill the running process in a container.
DockerDebuggeeWorkingDirectory	When debugging, the debugger is instructed to use this path as the working directory.
DockerDefaultTargetOS	The default target operating system used when building the Docker image.
DockerImageLabels	The default set of labels applied to the Docker image.
DockerFastModeProjectMountDirectory	In **Fast Mode**, this property controls where the project output directory is volume-mounted into the running container.
DockerfileBuildArguments	Additional arguments passed to the Docker build command.
DockerfileContext	The default context used when building the Docker image, as a path relative to the Dockerfile.
DockerfileFastModeStage	The Dockerfile stage (that is, target) to be used when building the image in debug mode.
DockerfileFile	Describes the default Dockerfile to use to build/run the container for the project. This value can be a path.
DockerfileRunArguments	Additional arguments passed to the Docker run command.
DockerfileRunEnvironmentFiles	Semicolon-delimited list of environment files applied during Docker run.
DockerfileTag	The tag to use when building the Docker image. In debugging, a ":dev" is appended to the tag.

Image Soure: https://docs.microsoft.com/en-us/visualstudio/containers/container-msbuild-properties?view=vs-2022

Sample

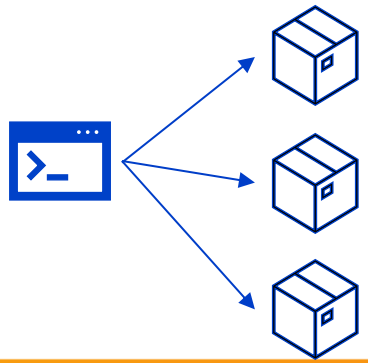
```xml
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
    <UserSecretsId>8c7ab9a5-d578-4c40-8b6d-54d174002229</UserSecretsId>
    <DockerDefaultTargetOS>Linux</DockerDefaultTargetOS>
    <!-- In CI/CD scenarios, you might need to change the context. By default, Visual Studio uses the
         folder above the Dockerfile. The path is relative to the Dockerfile, so here the context is
         set to the same folder as the Dockerfile. -->
    <DockerfileContext>.</DockerfileContext>
    <!-- Set `docker run` arguments to mount a volume -->
    <DockerfileRunArguments>-v $(MSBuildProjectDirectory)/host-folder:/container-folder:ro</DockerfileRunArguments>
    <!-- Set `docker build` arguments to add a custom tag -->
    <DockerfileBuildArguments>-t contoso/front-end:v2.0</DockerfileBuildArguments>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.VisualStudio.Azure.Containers.Tools.Targets" Version="1.20.1" />
  </ItemGroup>

</Project>
```


Image Soure: https://docs.microsoft.com/en-us/visualstudio/containers/container-msbuild-properties?view=vs-2022

Docker Compose – Multi Container Applications

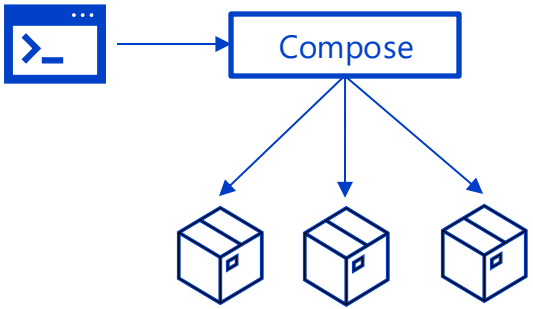
Without Compose

- Build and run one container at a time
- Must be careful with dependencies and startup order
- Manually connect containers together



With Compose

- Single file to define multi container applications
- Single command to deploy the entire application
- Compose takes care of dependencies
- Works with Docker Swarm, Networking, Volumes, Universal Control Plane



What is Docker Compose

- A tool for running multi-container applications
- A YAML file definition to configure your application
- Works on all stages: development, testing, production
- Single command to create, start and stop services based on the configuration


Demo
Docker Compose


Demo
Debugging in IDE


Demo
Networking

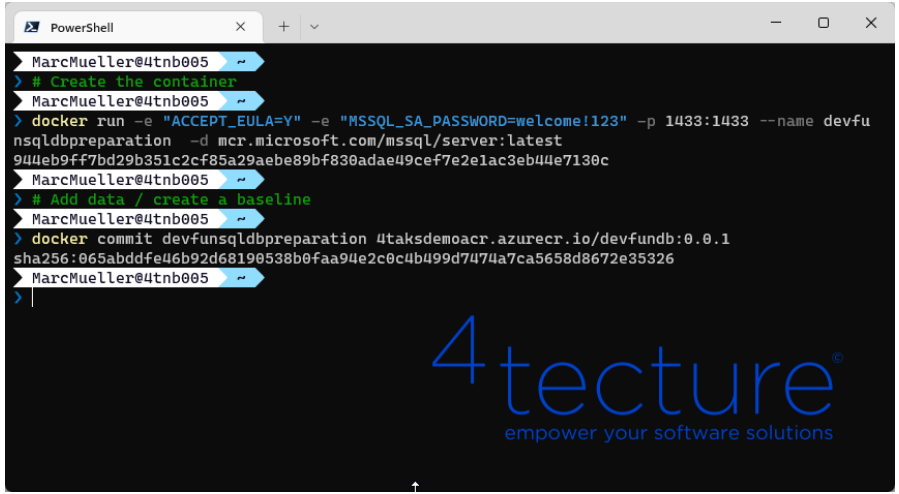

Demo
SQL Server Container

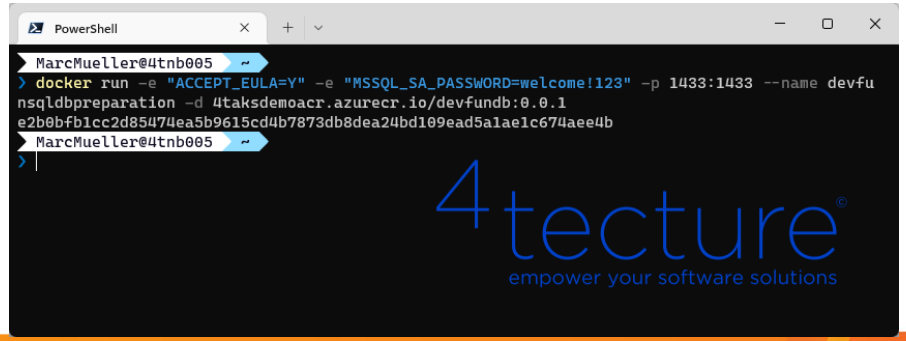
Local Development Environment

- Fast
- Multiple versions side-by-side
- External persistency

Fast Baseline for Testing

- Backup / Restore may increase execution time drastically
- Container images can be built with database baselines
- Multiple application data versions for migration testing
- Specific data baseline for corresponding tests





Docker for ASP.NET Core Developers

Q & A

4tecture
empower your software solutions

Recap

- Containers are the ideal artifact
- Multi-Stage Container Builds include your full CI tooling
- Optimize your final container images – size, dependencies, and security
- Seamless IDE integration with debugging support

Thank you for your attention!

If you have any questions do not hesitate to contact us:

4tecture GmbH
Industriestrasse 25
CH-8604 Volketswil

+41 44 508 37 00
info@4tecture.ch
www.4tecture.ch

Marc Müller
Principal Consultant

www.powerofdevops.com

4tecture
empower your software solutions

ProAuth

ReaFx


4tecture©
empower your software solutions