

# Ollama

Ve výchozím nastavení naslouchá na adrese: <http://127.0.0.1:11434/>

## ⊗ Important

Pro naslouchání aplikace je důležité mít spuštění ollama soubor. (`ollama app.exe`)

## Změna v naslouchání adresy

- Proměnné prostředí ve Windows

V `System variables`, nastavte `OLLAMA_HOST` s hodnotou adresy, kde má ollama naslouchat.

- Následně restartujte aplikaci `ollama app.exe`.

## Vypnout automatické spuštění

### i Note

Aplikace ollama je ve výchozím stavu zapnuta při spuštění počítače.

Vypnout lze provést tímto způsobem:

- `Windows` + `R`, následně napsat: `shell:startup` -> odkliknout `OK`
- Odstranit zástupce na Ollama.

## Příkazy

- ▶ [Nainstalované moduly](#)
- ▶ [Stažení modelu](#)
- ▶ [Odstranění modelu](#)

# Uživatelská konfigurace

- [Porovnání souborů přes aplikaci Meld](#)

## Klávesové zkratky

- ▶ [Parametry metody](#)
- ▶ [Procházení seznamu](#)
- ▶ [XML komentáře - zalomení řádku](#)

# Modifikátory přístupu

Určuje přístup k danému prvku.

Caller's location	public	protected internal	protected	internal	private protected	private	file
Within the file	✓	✓	✓	✓	✓	✓	✓
Within the class	✓	✓	✓	✓	✓	✓	✗
Derived class (same assembly)	✓	✓	✓	✓	✓	✗	✗
Non-derived class (same assembly)	✓	✓	✗	✓	✗	✗	✗
Derived class (different assembly)	✓	✓	✓	✗	✗	✗	✗
Non-derived class (different assembly)	✓	✗	✗	✗	✗	✗	✗

Více podrobností [zde](#).

## Složka 'runtimes' a multiplatformní nasazení

Slouží k ukládání **platformově specifických knihoven a binárních souborů**, které jsou nezbytné pro správné spuštění aplikace na různých operačních systémech a architekturách.

Používá se v aplikacích:

- WPF aplikace
- Konzolové aplikace
- ASP.NET Core aplikace
- WinForms aplikace
- .NET knihovny a služby, které se nasazují na různé platformy (Windows, Linux, macOS, atd.)

### Tip

V unity se používá složka `'Plugins'` k umístění knihoven (DLL), které jsou platformově specifické

Tato složka může obsahovat nativní kód pro různé platformy (Windows, Android, iOS, macOS, atd.)

Tato složka `'runtime'` se automaticky generuje při publikaci aplikace a hraje zásadní roli zejména při použití dvou typů nasazení.

Každý z těchto typů nasazení určuje, jakým způsobem aplikace zajišťuje dostupnost .NET runtime a dalších závislostí:

## self-contained deployment

Aplikace je distribuována společně s **kompletním .NET runtime**

To znamená, že aplikace si nese vlastní runtime a může běžet nezávisle na tom, zda má uživatel na svém systému nainstalovaný správný .NET runtime.

### Tip

Toto nasazení je vhodné, pokud chcete zajistit, že aplikace poběží na jakémkoliv počítači, bez ohledu na její aktuální stav.

Výsledkem je větší velikost aplikace, protože obsahuje kompletní runtime pro cílové platformy, které jsou zahrnuty ve složce `'runtimes'`.

V praxi složka `'runtimes'` obsahuje potřebné knihovny a binární soubory pro různé platformy, jako jsou Windows, Linux, macOS, nebo různé architektury (x64, x86, ARM).

Díky tomu může aplikace běžet **out-of-the-box** bez nutnosti další instalace runtime na cílovém systému.

## Použití

Nastavit v souboru projektu (.csproj):

```
<PropertyGroup>
  <SelfContained>true</SelfContained>
  <RuntimeIdentifier>win-x64</RuntimeIdentifier> <!-- Nebo jiný RID podle
platformy -->
</PropertyGroup>
```

nebo s více Runtime Identifiers (RID)

```
<PropertyGroup>
  <SelfContained>true</SelfContained>
  <RuntimeIdentifiers>win-x64;linux-x64;osx-x64</RuntimeIdentifiers>
</PropertyGroup>
```

### Note

**Runtime Identifiers (RID)** Runtime Identifiers (RID) jsou klíčovou součástí procesu nasazení, protože určují, pro jaké platformy a architektury bude aplikace kompilována.

Můžete specifikovat různé RID podle cílové platformy:

- win-x64 (Windows 64-bit)
- linux-x64 (Linux 64-bit)
- osx-x64 (macOS 64-bit)
- win-arm (Windows na ARM procesorech)

## framework-dependent deployment (runtime-specific deployment)

Aplikace **závisí na přítomnosti .NET runtime** na cílovém systému.

Aplikace neobsahuje runtime v sobě, což zmenšuje její velikost, ale předpokládá, že uživatel má již správnou verzi .NET runtime nainstalovanou.

### Important

Pokud runtime není přítomen, aplikace nefunguje, dokud uživatel runtime nedoinstaluje.

V tomto případě složka `"runtimes"` může obsahovat pouze platformově specifické knihovny a závislosti, které nejsou součástí základního .NET runtime, a zajistit kompatibilitu aplikace s různými platformami.

## Použití

### Note

Nastavte `SelfContained` na `false`, nebo tuto vlastnost úplně vynechte (výchozí nastavení je totiž `framework-dependent`).

Nastavit v souboru projektu (.csproj):

```
<PropertyGroup>
  <SelfContained>false</SelfContained>
</PropertyGroup>
```

## Uvolnění zdrojů

- **Řízené zdroje**

= Objekty, které jsou spravovány garbage collectorem.

Zahrnuje všechny objekty vytvořené pomocí klíčového slova `new`.

Garbage collector automaticky sleduje tyto objekty a uvolňuje jejich paměť, když již nejsou potřebné.

### Tip

Programátoři nemusí explicitně uvolňovat paměť pro tyto objekty.

- **Neřízené zdroje**

= Objekty, které nejsou spravovány garbage collectorem.

Zahrnuje soubory, databázové připojení, síťové zdroje, atd...

### Note

Programátoři musí explicitně uvolnit tyto zdroje, aby zabránili úniku paměti.

Uvolnění zdrojů je důležité pro správné fungování aplikace.

## Destruktor

- Automaticky volán, když je objekt zničen garbage collectorem.

- Definuje se pomocí syntaxe `~ClassName()`.
- Uvolňuje neřízené zdroje, které třída drží.
- Destruktory nejsou deterministické.

**i Tip**

Znamená, že nevíme přesně, kdy budou volány.

- Jsou volány, když **garbage collector rozhodne**, že je objekt vhodný ke zničení.

**i Note**

Může to být kdykoliv po tom, co objekt přestane být používán.

## Dispose

- Součástí rozhraní `IDisposable` a je určen pro explicitní uvolnění zdrojů.
- Metoda `Dispose` je volána programátorem, když je známo, že objekt již nebude potřebný.

**i Note**

To umožňuje okamžité uvolnění zdrojů a zajišťuje, že nebudou drženy déle, než je nutné.

- Metoda `Dispose` je určena pro uvolnění jak řízených, tak neřízených zdrojů.

## Volání funkcí z externích DLL

**i Note**

**Platform Invocation Services (PInvoke)** se používá pro volání knihoven z nativního kódu.

- Nativní kód = Kód kompilován do strojového kódu pro konkrétní platformu.
- Strojový kód = Kód přímo spustitelný na hardware dané platformy.

Postup:

### 1. Importování funkce z DLL



Atribut `DllImport` z `System.Runtime.InteropServices` k importování funkce z DLL.

Například:

```
using System.Runtime.InteropServices;

public class MyProgram
{
    [DllImport("User32.dll")]
    public static extern int MessageBox(IntPtr h, string m, string c, int type);
}
```

#### Note

`MessageBox` je funkce definovaná v knihovně: `User32.dll`.

Tato funkce je nyní dostupná v rámci tohoto .NET kódu.

## Příklad knihovny v c++

Knihovna v c++ by mohla vypadat takto:

```
#include <windows.h>

extern "C" __declspec(dllexport) int MessageBox(HWND h, LPCSTR m, LPCSTR c,
int type)
{
    return MessageBoxA(h, m, c, type);
}
```

## Metody a argumenty v c++

- `extern "C"`

#### Warning

Zajistí, že funkce jsou kompatibilní s C jazykem. (To je důležité pro interoperabilitu mezi C++ a jinými jazyky, jako je C#.)

### Tip

Když kompilátor narazí na funkci, změní její název na něco, co jednoznačně identifikuje nejen název funkce, ale také typy jejích parametrů.

To znamená, že název funkce, jak je viděn v DLL, nebude stejný jako název funkce v původním kódu.

Když použijete `extern "C"`, říkáte kompilátoru, aby tuto funkci nezměnil a zachoval její název tak, jak je. To umožňuje jiným jazykům, jako je C#, najít a volat tuto funkci správným názvem.

#### ◦ `__declspec(dllexport)`

Říká kompilátoru C++, že tato funkce nebo proměnná bude exportována z DLL, takže ji může volat jiný kód, který tuto DLL používá.

### Warning

Důležité k viditelnosti a dostupnosti pro `PInvoke`

#### ◦ `HWND`

"handle to a window" (rukojeť okna)

#### ◦ `LPCTSTR`

"Long Pointer Constant String"

Item	8-bitů (Ansi)	16-bitů (Wide)	Různé
character	CHAR	WCHAR	TCHAR
string	LPSTR	LPWSTR	LPTSTR
string (const)	LPCSTR	LPCWSTR	LPCTSTR

Odkaz [zde](#) 

## 2. Volání importované funkce

```
public class MyProgram
{
```

```
public static void Main()
{
    MessageBox(IntPtr.Zero, "Hello, World!", "Test MessageBox", 0);
}
```

`IntPtr.Zero`

Konstanta, která reprezentuje nulový ukazatel.

Je to ekvivalent NULL v C++.

## `__Internal`

Pokud se používá `__Internal` jako název DLL v atributu `DllImport`, znamená to, že funkce se hledá přímo v hlavním spustitelném souboru.

## Tip

Hledá tedy v samotné aplikaci, pokud je to nativní kód, nebo v jedné z knihoven, na které aplikace odkazuje.

- Příklad použití v C#:

```
public class MyProgram
{
    [DllImport("__Internal")]
    public static extern int MyFunction();

    public static void Main()
    {
        MyFunction();
    }
}
```

- Příklad definice v C++

```
extern "C" __declspec(dllexport) int MyFunction()
{
    // Implementace vaší funkce
    return 0;
}
```

V tomto příkladu `MyFunction` je funkce definovaná v nativním kódu.

Je tedy součástí aplikace nebo jedné z jejích závislostí.

## Tipy

- `PInvoke`

= **Platform Invocation Services**, což je technika v `.NET`, která umožňuje volání funkcí, které jsou implementovány v neřízeném kódu.

## Tip

To je obvykle používáno pro volání C API funkcí, které jsou definovány v DLL.

Příklad:

```
private static class PInvoke
{
    #if UNITY_IOS || UNITY_TVOS
        private const string DllName = "__Internal";
    #elif UNITY_STANDALONE_OSX
        private const string DllName = "MacOSAppleAuthManager";
    #endif

    public delegate void NativeMessageHandlerCallbackDelegate(uint
requestId, string payload);

    [AOT.MonoPInvokeCallback(typeof(NativeMessageHandlerCallbackDelegate))]
    public static void NativeMessageHandlerCallback(uint requestId,
string payload)
    {
        try
        {
            CallbackHandler.ScheduleCallback(requestId, payload);
        }
        catch (Exception exception)
        {
            Console.WriteLine("Received exception while scheduling a
callback for request ID " + requestId);
            Console.WriteLine("Detailed payload:\n" + payload);
            Console.WriteLine("Exception: " + exception);
        }
    }

    [System.Runtime.InteropServices.DllImport(DllName)]
    public static extern bool AppleAuth_IsCurrentPlatformSupported();

    [System.Runtime.InteropServices.DllImport(DllName)]
    public static extern void
AppleAuth_SetupNativeMessageHandlerCallback(NativeMessageHandlerCallbackDelegate
callback);

    [System.Runtime.InteropServices.DllImport(DllName)]
```

```

        public static extern void AppleAuth_GetCredentialState(uint requestId,
string userId);

[System.Runtime.InteropServices.DllImport(DllName)]
        public static extern void AppleAuth_LoginWithAppleId(uint requestId,
int loginOptions, string nonceCStr, string stateCStr);

[System.Runtime.InteropServices.DllImport(DllName)]
        public static extern void AppleAuth_QuickLogin(uint requestId, string
nonceCStr, string stateCStr);

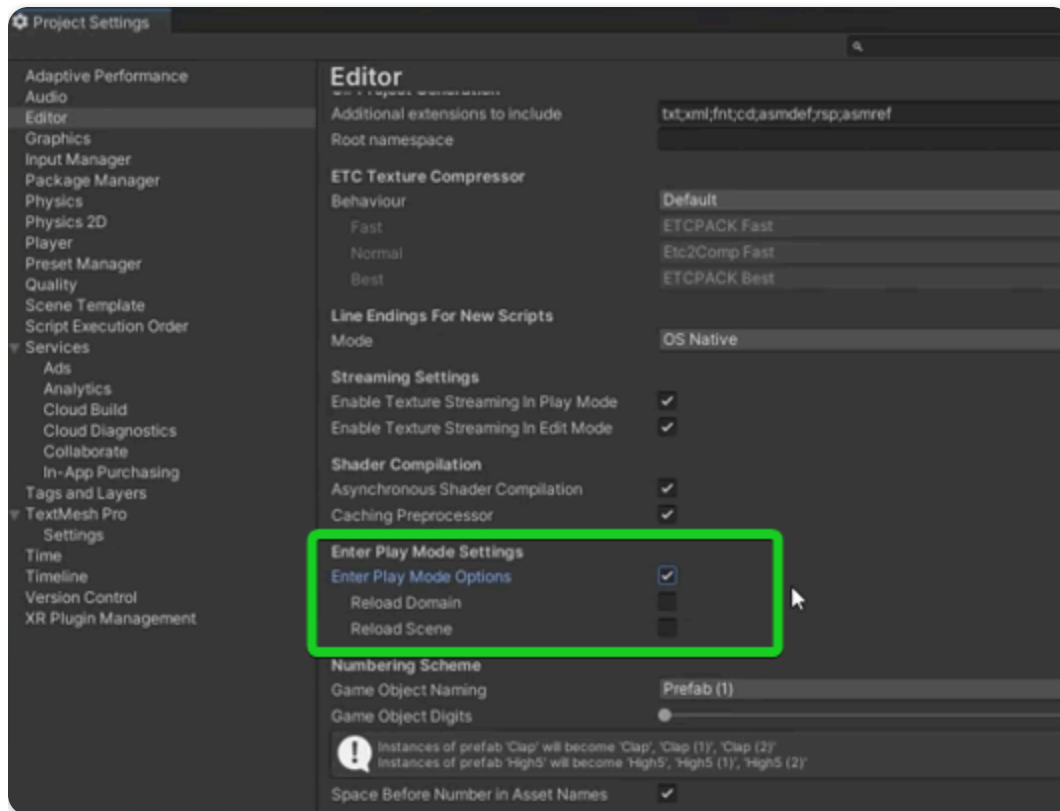
[System.Runtime.InteropServices.DllImport(DllName)]
        public static extern void
AppleAuth_RegisterCredentialsRevokedCallbackId(uint callbackId);

[System.Runtime.InteropServices.DllImport(DllName)]
        public static extern void AppleAuth_LogMessage(string messageCStr);
    }

```

Více info [zde](#).

# Rychlejší spuštění



- Reload Domain

Když je tato možnost povolena, všechny skripty se znovu načtou, což může trvat déle, ale zajišťuje, že se všechny změny v kódu projeví.

- Reload Scene

Když je tato možnost povolena, Unity znovu načte aktuální scénu, což může být užitečné, pokud chceš začít s "čistým" stavem.

Pokud tyto možnosti zakážeš, můžeš zrychlit vstup do režimu hry, protože Unity se vyhne některým časově náročným procesům.

## Výběr hry 2D či 3D

2D or 3D Unity Dev - What's better??



## Rychlé prototypování

Prototyping Games in Unity?



## Klíčová slova



2 Keywords I didn't fully understand when starting gamedev...



## Vývojové vzory

3 Game Programming Patterns WE ACTUALLY NEED.



# Microsoft SQL

## Kód

### Získat informace

---

- ▶ [Informace ze Serveru](#)
- ▶ [Informace z Tabulky](#)
- ▶ [Velikost Tabulek](#)
- ▶ [Informace o indexech na sloupcích](#)

### Hledat

---

- ▶ [Sloupec a zjistit v jaké Tabulce se nachází](#)
- ▶ [Datový typ Sloupce z Tabulky](#)
- ▶ [Hodnoty ve všech textových a číselných sloupcích databáze](#)
- ▶ [Nejnovější a Nejstarší záznam](#)
- ▶ [Nejčastěji se vyskytující hodnoty](#)
- ▶ [Port na kterém je spuštěn Server](#)

### Výkon

---

- ▶ [Efektivita dotazů](#)

### Konfigurace

---

- ▶ [Vzdálený přístup](#)

Izoluje aplikace se všemi jejími knihovnami, konfiguračními soubory a dalšími závislými soubory do kontejnerů.

### *i* Note

Kontejnery zajišťují, že aplikace mohou být spuštěny v jakémkoli prostředí.

Docker se stará o celý životní cyklus kontejnerů.

**Kontejner → Vytvoření → Spuštění → Zastavení**

### *i* Tip

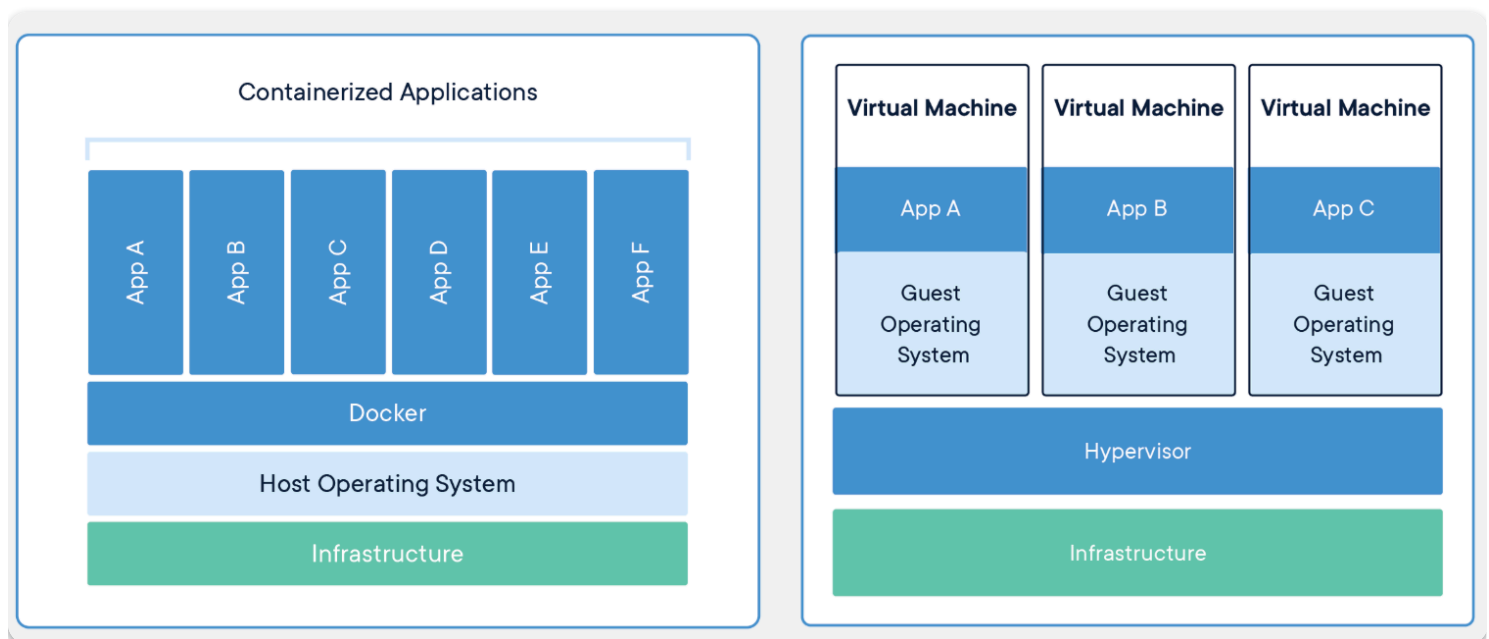
- Kontejnerizace je virtualizací jádra operačního systému.

Všechny kontejnery běží v rámci jednoho operačního systému a sdílejí paměť, knihovny a další zdroje.

- Zdroje se tímto způsobem využívají efektivněji než klasická virtualizace.

Spuštění kontejneru je navíc mnohem rychlejší než spuštění virtuálního stroje s instalací operačního systému.

- Malá režie a na stejném hardwaru můžete spustit více docker kontejnerů než virtuálních strojů



## Soubory dockeru

## **dockerd.exe a docker.exe**

- dockerd.exe:

Spouští Docker Daemon, což je hlavní služba, která spravuje kontejnery a poslouchá na socketu či TCP portu.

- docker.exe

Klientský nástroj, který posílá příkazy daemonu (např. `docker run`, `docker ps`).

## **docker-compose.exe a docker-compose.yml**

- docker-compose.exe:

Umožňuje definovat a spouštět více Docker kontejnerů jako součásti jedné aplikace.

Pomocí souboru `docker-compose.yml` můžete definovat všechny služby (kontejnery), které mají běžet, včetně jejich konfigurací, závislostí a propojení mezi nimi.

- docker-compose.yml:

Konfigurační soubor, který popisuje, jaké kontejnery (služby) mají být spuštěny, jaké obrazové soubory mají používat, jaké porty mají být mapovány a jaké další nastavení kontejnery potřebují.

Tento soubor je zpracován příkazem `docker-compose up`, který automaticky spustí všechny definované kontejnery.

## **Základní pojmy**

### **dockerfile**

Textový soubor s instrukcemi k vytvoření `Docker image`.

Specifikuje operační systém, na kterém bude běžet kontejner, jazyky, lokace, porty a další komponenty.

### **docker image**

Komprimovaná, samostatná část softwaru vytvořená příkazy v `Dockerfile`.

Je to "šablona" (aplikace plus požadované knihovny a binární soubory) potřebná k vytvoření a spuštění Docker kontejneru.

### **docker run**

Příkaz, který spouští kontejnery.

#### Note

Každý kontejner je instancí jednoho image.

## docker hub

Oficiální úložiště pro sdílení `docker image`.

#### Tip

Obsahuje oficiální `docker image` z open-source projektů a neoficiální od komunity.

Možnost pracovat i s lokálními docker úložišti.

## docker engine

Jádro softwaru docker.

Technologie na principu klient-server, která vytváří a provozuje kontejnery.

## docker compose

Definice ke spuštění více kontejnerů.

# Příkazy

## Stáhnout docker image

```
docker pull <Image name>
```

#### Note

Je název `docker image`.

Například: `mcr.microsoft.com/dotnet/core/sdk:3.1`

Umístění docker image po stažení:

- Linux:

```
/var/lib/docker/
```

- Windows:

```
C:\ProgramData\DockerDesktop
```

- macOS:

```
~/Library/Containers/com.docker.docker/Data/vms/0/
```

## Sestavení docker image

- `docker build [-t] customFolder`

Sestaví kontejner pro docker image ve vybraném adresáři.

### Note

`customFolder`

Název vybraného adresáře k sestavení docker image.

Může být například aktuální adresář: `.`, nebo jakkoli jinak.

`-t`

Pojmenování image a tagu. (Pokud není zadán parametr `-t`, použije se tag: `latest`)

- Příklad:

```
docker build -t myapp .
```

**i Tip**

`myapp`

Název pro nově sestavený kontejner. (Může být jakýkoli.)

.

Pracovní adresář v dockeru. (V tomto příkladu kořenový adresář.)

## Spuštění kontejneru z docker image

- `docker run <docker image>`

Spustí kontejner pro docker image.

```
docker run kitematic/hello-world-nginx
```

**i Tip**

Spustí docker kontejner s docker image: `kitematic/hello-world-nginx`

## Spustit na jiném portu

```
docker run -p 70:80 kitematic/hello-world-nginx
```

### Note

`-p`

Mapuje port 70 na hostitelském stroji na port 80 uvnitř kontejneru. (To znamená, že pokud aplikace uvnitř kontejneru poslouchá na portu 80, bude přístupná na portu 70 hostitelského stroje.)

`kitematic/hello-world-nginx`

Název docker image ke spuštění.

## Spustit v interaktivním módu

```
docker run -it kitematic/hello-world-nginx
```

### Note

Užitečné, pokud chcete spustit kontejner a poté v něm spustit další příkazy, například při ladění nebo vývoji.

## Odstranit po zastavení

```
docker run --rm kitematic/hello-world-nginx
```

### Note

`--rm`

Docker automaticky odstraní kontejner, když je běh kontejneru přerušeno.

### Tip

Užitečné, pokud nechcete, aby se vaše lokální úložiště naplnilo zastavenými kontejnery.



## Spuštění více kontejnerů z docker image najednou

### Note

Musíte použít soubor YAML k definování služeb vaší aplikace.

Následně pomocí jediného příkazu `docker-compose up` můžete vytvořit a spustit všechny služby definované ve vašem souboru `docker-compose.yml`.

### Tip

Automaticky použije lokální `docker image`, pokud je k dispozici.

Příklad souboru `docker-compose.yml`:

```
# Verze Docker Compose souboru
version: '3.4'

# Definice služeb
services:
  # Název služby
  webapp:
    # Obraz, který se má použít pro tuto službu
    # Tento obraz je vzorová aplikace ASP.NET Core od Microsoftu
    image: mcr.microsoft.com/dotnet/core/samples:aspnetapp
    # Instrukce pro sestavení obrazu
    build:
      # Kontext pro sestavení, obvykle je to adresář obsahující Dockerfile
      context: .
      # Cesta k Dockerfile
      dockerfile: Dockerfile
    # Mapování portů mezi hostitelem a kontejnerem
    # Formát je "host:kontejner"
    # Toto nastavení říká Dockeru, aby přesměroval port 8000 na hostiteli na port 80
    # v kontejneru
    ports:
      - "8000:80"
```

## Dockerfile

V dockeru **není žádná výchozí složka**.

#### Tip

Když vytváříte Dockerfile, můžete nastavit pracovní adresář v kontejneru pomocí příkazu `WORKDIR`.

#### Tip

Pokud není nastaven `WORKDIR`, vztahuje se vše na kořenový adresář (/) kontejneru.

## Příklad pro .NET Core

```
# Používáme oficiální .NET Core runtime image z Docker Hub
# 'dotnet' je jméno image a '3.1' je tag, který specifikuje verzi
FROM mcr.microsoft.com/dotnet/core/runtime:3.1

# Nastavíme pracovní adresář v kontejneru na /app
# Pokud tento adresář neexistuje, docker ho vytvoří
WORKDIR /app

# Kopírujeme výstup buildu z našeho stroje do kontejneru
# 'publish' je cesta k výstupu buildu na našem stroji
# '.' znamená aktuální (pracovní) adresář v kontejneru
COPY ./publish .

# Nastavíme spustitelný soubor pro kontejner
# 'myapp.dll' je název naší aplikace
ENTRYPOINT ["dotnet", "myapp.dll"]
```

#### Note

Vytvoří `docker image` pro vaši aplikaci .NET Core.

Když spustíte kontejner z této image, vaše aplikace se automaticky spustí.

## Příklad pro C# Aplikaci

```
# Používáme oficiální .NET Core SDK image z Docker Hub
# 'dotnet' je jméno image a '3.1' je tag, který specifikuje verzi
```

```
FROM mcr.microsoft.com/dotnet/core/sdk:3.1

# Nastavíme pracovní adresář v kontejneru na /app
# Pokud tento adresář neexistuje, docker ho vytvoří
WORKDIR /app

# Kopírujeme všechny soubory z našeho stroje do kontejneru
# '.' znamená aktuální adresář na našem stroji
# '.' znamená aktuální (pracovní) adresář v kontejneru
COPY . .

# Spustíme příkaz 'dotnet restore', který stáhne všechny potřebné NuGet balíčky
RUN dotnet restore

# Spustíme příkaz 'dotnet publish', který vytvoří výstup buildu naší aplikace
RUN dotnet publish -c Release -o out

# Nastavíme spustitelný soubor pro kontejner
# 'myapp.dll' je název naší aplikace
ENTRYPOINT ["dotnet", "out/myapp.dll"]
```

#### Note

Tento Dockerfile vytvoří docker image pro vaši aplikaci C#.

Když spustíte kontejner z této image, vaše aplikace se automaticky spustí.

## Příklad .NET Core a lokálních NuGet balíčků

```
# Používáme oficiální .NET Core SDK image z Docker Hub
FROM mcr.microsoft.com/dotnet/core/sdk:3.1

# Nastavíme pracovní adresář v kontejneru na /app
WORKDIR /app

# Kopírujeme všechny soubory z našeho stroje do kontejneru
COPY . .

# Spustíme příkaz 'dotnet restore', který načte všechny potřebné NuGet balíčky z
lokálního úložiště
# Předpokládáme, že všechny potřebné NuGet balíčky jsou uloženy v adresáři 'nuget'
našeho projektu
RUN dotnet restore --source ./nuget
```

```
# Spustíme příkaz 'dotnet publish', který vytvoří výstup buildu naší aplikace
RUN dotnet publish -c Release -o out

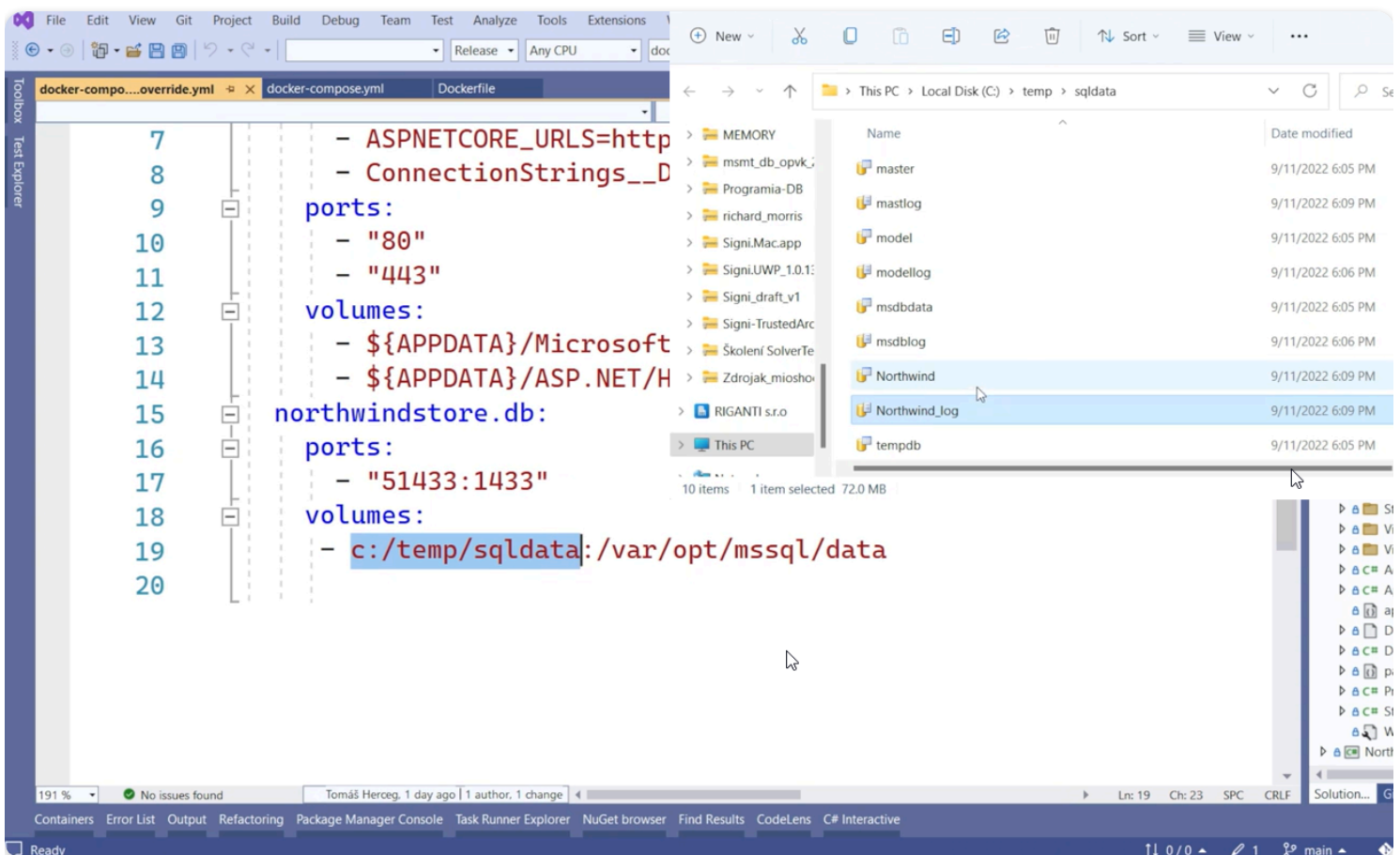
# Nastavíme spustitelný soubor pro kontejner
ENTRYPOINT ["dotnet", "out/myapp.dll"]
```

## Tip

V tomto příkladu předpokládáme, že všechny potřebné NuGet balíčky jsou uloženy v adresáři `nuget` vašeho projektu.

Příkaz `dotnet restore --source ./nuget` pak načte tyto balíčky z lokálního úložiště místo stahování z internetu.

## Zachování dat z kontejneru na lokálním disku



The screenshot displays the Visual Studio IDE with a Docker Compose configuration file open in the editor. The configuration is for a service named `northwindstore.db` and includes the following settings:

```
7 - ASPNETCORE_URLS=http
8 - ConnectionStrings__D
9
10 ports:
11 - "80"
12 - "443"
13
14 volumes:
15 - ${APPDATA}/Microsoft
16 - ${APPDATA}/ASP.NET/H
17
18 northwindstore.db:
19 ports:
20 - "51433:1433"
21
22 volumes:
23 - c:/temp/sqldata:/var/opt/mssql/data
```

On the right side of the IDE, a file explorer window shows the contents of the `c:/temp/sqldata` directory. The directory contains several files and folders, including `master`, `mastlog`, `model`, `modellog`, `msdbdata`, `msdblog`, `Northwind`, `Northwind_log`, and `tempdb`. The `Northwind_log` file is currently selected.

## Použití mřížky (grid)

1. View -> Show Grid (zobrazí mřížku)
2. View -> Grid and Axis... (nastavení mřížky)

- [Výpočet obrázku pro zarovnání na střed](#)

## FAQ

- [Modrý čtverec uvnitř stránky](#)

# OBS (Open Broadcast Software)

► [Argumenty](#)

# SSH

SSH je bezpečnější než používání uživatelského jména a hesla, protože využívá veřejný a soukromý klíč.

► [Přípojení na GitHub](#)

# Zobrazení

- [Sloučení/Oddělení panelů kalendáře](#)



## Instalace

- ▶ [Řešení problému s neviditelným diskem při instalaci Windows](#)

## Základní nastavení

- ▶ [Zobrazení sekund v dolním panelu](#)

## Klávesnicové zkratky

- ▶ [Minimalizace/Maximalizace všech oken](#)
- ▶ [Skočení na adresní řádek](#)

## Chybějící klávesy na klávesnici

- ▶ [Kontextová klávesa](#)