Неделя 3. Лаба 2. Docker: упаковываем сервис задач и запускаем на Docker Desktop

Часть 1 из 2: разделы «Подготовка: Docker Desktop и PowerShell 7» — «5. Сборка и кэш».

Цель: написать multi-stage Dockerfile для сервиса задач, собрать образ, запустить сервис на Docker Desktop и поработать с ним — из терминала и из окна Docker Desktop. **Время:** 90 мин. **Теория:** лекция 2. **Инструменты:** Docker Desktop, Visual Studio 2026, PowerShell 7.

Подготовка: Docker Desktop и PowerShell 7

С этой лабы сервис запускается в контейнере, поэтому нужны две новые программы. Установка с загрузкой занимает 20–40 минут — лучше сделать её до занятия.

Программа	Зачем
Docker Desktop	Собирает образы и запускает контейнеры на вашем ПК. Контейнеры — это Linux-процессы, поэтому Docker Desktop работает поверх **WSL 2** — встроенной в Windows лёгкой виртуальной машины с Linux-ядром. Установщик включит её сам
PowerShell 7	Терминал, в котором выполняются команды курса. Встроенный Windows PowerShell 5.1 старее и в ряде команд ведёт себя иначе

Установка — PowerShell **от администратора** (или скачайте установщики с docker.com и learn.microsoft.com/powershell):

```
winget install -e --id Docker.DockerDesktop
winget install -e --id Microsoft.PowerShell
```

Перезагрузите ПК, запустите Docker Desktop и примите условия. Проверьте настройки:

- *Settings → General:* **Use the WSL 2 based engine** включено;
- *Settings → Resources:* Docker доступно не меньше 4 ГБ памяти (иначе в лабе 7 не хватит места мониторингу).

Если Docker Desktop пишет, что виртуализация выключена, — её включают в BIOS/UEFI (пункт *Intel VT-x* или *AMD SVM*); обратитесь к преподавателю.

Дальше **все команды — в PowerShell 7**: в меню Пуск «PowerShell 7», или `pwsh`. Один раз настройте кириллицу в выводе и откройте терминал заново:

```
if (-not (Test-Path $PROFILE)) { New-Item -ItemType File -Path $PROFILE -Force | Out-Null }
Add-Content $PROFILE '[Console]::OutputEncoding = [Text.Encoding]::UTF8'
```

Проверка:

```
docker version              # есть блоки Client и Server — Docker Desktop запущен
docker run --rm hello-world  # скачал образ и запустил первый контейнер
pwsh --version              # 7.x
```

> В PowerShell `curl` может быть псевдонимом `Invoke-WebRequest`. **В курсе пишем `curl.exe`**, а для JSON-запросов — `Invoke-RestMethod` (сокращённо `irm`).

Начало

```
cd C:\dev\demo-api           # корень репозитория, рядом с DemoApi.slnx
```

Остановите сервис в Visual Studio (Shift+F5): порт 8080 понадобится контейнеру.

1. Базовый образ

```
docker pull mcr.microsoft.com/dotnet/aspnet:10.0
docker image ls mcr.microsoft.com/dotnet/aspnet
docker image history mcr.microsoft.com/dotnet/aspnet:10.0
```

Что увидеть: это основа нашего будущего образа — Linux и рантайм ASP.NET. `history` показывает слои, из которых он собран.

2. `.dockerignore`

Создайте в корне репозитория два пустых файла и откройте первый:

```
New-Item -ItemType File .dockerignore, Dockerfile
notepad .dockerignore
```

У файлов не должно быть расширения `.txt`, иначе Docker их не найдёт. Проверка: `Get-ChildItem .dockerignore, Dockerfile`.

Содержимое `.dockerignore`:

```
**/bin/
**/obj/
**/TestResults/
.git
.github
.vs
.vscode
.idea
tests/
monitoring/
scripts/
db/
*.md
Dockerfile
compose*.yaml
.env
```

Зачем: в контекст сборки не попадут Windows-артефакты из `bin/obj` (они ломают Linux-сборку), история git, тесты и секреты из `.env`. Сборка быстрее, образ чище.

3. Dockerfile: этап сборки

Откройте `Dockerfile` (`notepad Dockerfile` или в Visual Studio *Файл → Открыть → Файл*) и вставьте:

```
# syntax=docker/dockerfile:1

# ---------- Этап 1: сборка ----------
FROM mcr.microsoft.com/dotnet/sdk:10.0 AS build
WORKDIR /src

COPY src/DemoApi/DemoApi.csproj src/DemoApi/
RUN dotnet restore src/DemoApi/DemoApi.csproj

COPY src/ src/
RUN dotnet publish src/DemoApi/DemoApi.csproj -c Release -o /app --no-restore
```

Строки	Зачем
`# syntax=docker/dockerfile:1`	Версия синтаксиса Dockerfile; оставьте как есть
`FROM ...sdk:10.0 AS build`	SDK с компилятором. Имя этапа нужно, чтобы потом забрать из него результат
`COPY ...csproj` + `RUN dotnet restore`	Слой с NuGet-пакетами кэшируется, пока не меняется `.csproj`
`COPY src/ src/`	Код копируется **после** restore: правка кода не сбрасывает кэш пакетов
`dotnet publish -c Release -o /app`	То же, что публикация из Visual Studio: готовые к запуску DLL в `/app`

4. Dockerfile: этап запуска

Допишите в тот же файл:

```
# ---------- Этап 2: рантайм ----------
FROM mcr.microsoft.com/dotnet/aspnet:10.0 AS runtime

RUN apt-get update \
 && apt-get install -y --no-install-recommends curl \
 && rm -rf /var/lib/apt/lists/*

WORKDIR /app
COPY --from=build /app .

ARG APP_VERSION=dev
ENV APP_VERSION=$APP_VERSION

USER $APP_UID
EXPOSE 8080

HEALTHCHECK --interval=10s --timeout=3s --start-period=15s --retries=3 \
  CMD curl -fsS http://localhost:8080/health/live || exit 1

ENTRYPOINT ["dotnet", "DemoApi.dll"]
```

Строки	Зачем
`FROM ...aspnet:10.0`	Только рантайм, без SDK и исходников: образ меньше и безопаснее
`apt-get install curl`	Нужен для `HEALTHCHECK`; очистка списков пакетов уменьшает слой
`COPY --from=build /app .`	Из первого этапа берём только результат publish
`ARG` / `ENV APP_VERSION`	Версия задаётся при сборке (в CI — sha коммита) и видна в `GET /`
`USER $APP_UID`	Процесс работает не от root. `APP_UID` — переменная базового образа .NET: id встроенного пользователя `app`
`EXPOSE 8080`	Документирует порт ASP.NET по умолчанию
`HEALTHCHECK`	Docker проверяет `/health/live` и показывает `healthy`/`unhealthy`
`ENTRYPOINT`	Главный процесс контейнера — наш сервис

5. Сборка и кэш

```
docker build -t demo-api:1.0 .
docker image ls demo-api                  # размер образа
```

Измените текст любого сообщения лога в `src/DemoApi/Services/TaskService.cs` и соберите снова:

```
docker build --progress=plain -t demo-api:1.1 .
```

`--progress=plain` выводит лог сборки обычным текстом, его удобно копировать. Найдите шаги с пометкой `CACHED`. **Шаг `dotnet restore` должен быть из кэша** — пакеты не менялись.