Неделя 3. Лекция 2. Docker: сервис задач в контейнере

Часть 1 из 2.

После лекции: вы понимаете, что такое контейнер и образ и как они устроены внутри, как Docker Desktop работает на Windows, что происходит при `docker build` и `docker run`, и можете написать production-пригодный Dockerfile для ASP.NET — чтобы сервис задач одинаково запускался на любом компьютере. **Инструменты:** Docker Desktop, Visual Studio 2026, PowerShell 7.

1. Какую проблему решают контейнеры

В теме 1 сервис запускался из Visual Studio. Чтобы запустить его на другом компьютере, там нужны: .NET нужной версии, собранные DLL, правильные настройки. На сервере — то же самое, плюс ещё десяток сервисов со своими версиями .NET, Java, Python. Отсюда вечное «у меня работает».

Классический сценарий до контейнеров выглядит так. Сервис собирают на машине разработчика, архив с DLL копируют на сервер, а там уже стоит рантайм другой патч-версии, потому что соседний сервис попросил обновление. Через месяц никто не может сказать, чем прод отличается от тестового стенда: на них годами вручную ставили пакеты и правили конфиги. Каждый деплой превращается в ручную операцию со своей инструкцией.

Контейнер упаковывает приложение вместе со всем, что нужно для запуска: рантаймом .NET, библиотеками ОС, нашими DLL и настройками по умолчанию. Такая упаковка называется **образ**. Образ запускается одинаково:

- у вас на Docker Desktop (эта тема);
- у коллеги — ему не нужно ставить .NET;
- на сервере GitHub в CI и на сервере деплоя (тема 5).

Плюсы: **один артефакт** от ноутбука до прода, **старт за секунды**, сервисы изолированы друг от друга и не конфликтуют версиями. Сервер при этом становится «тупым»: на нём нужен только Docker, а всё, от чего зависит приложение, приезжает внутри образа. Поэтому в индустрии контейнер стал стандартной единицей поставки: CI собирает образ, а дальше его одинаково запускают Docker, Compose или Kubernetes.

2. Контейнер ≠ виртуальная машина

	Виртуальная машина	Контейнер
Что внутри	Целая ОС со своим ядром	Только приложение и его файлы
Ядро ОС	своё	общее с хостом
Старт	минуты	секунды
Размер	гигабайты	десятки–сотни МБ
Изоляция	гипервизор	механизмы ядра Linux

Контейнер — это **обычный процесс Linux**. Никакой «маленькой ОС» внутри не запускается: ядро одно, и оно же выполняет процессы контейнеров. Отличие от обычного процесса в том, что ядро при запуске ограничивает ему три вещи.

Видимость — namespaces. Пространство имён — это отдельная «копия» какого-то ресурса ядра для группы процессов:

Namespace	Что изолирует	Что видит наш сервис
PID	Номера процессов	Себя под номером 1 и больше никого
NET	Сетевые интерфейсы, порты	Свой IP и свой порт 8080; порт 8080 соседнего контейнера ему не мешает
MNT	Точки монтирования, файловая система	Файлы образа, а не диск хоста
UTS	Имя хоста	Hostname = id контейнера (его возвращает `GET /`)
IPC	Общую память между процессами	Только свою

Ресурсы — cgroups. Control groups считают и ограничивают, сколько CPU, памяти и операций ввода-вывода

потребляет группа процессов. По умолчанию лимитов нет, контейнер может съесть всю память VM. Лимиты и то, что бывает при их превышении, — тема 6.

Файловая система — слои. Корневая файловая система контейнера собирается из слоёв образа через **overlay FS**: нижние слои только для чтения, сверху — тонкий записываемый слой контейнера. Приложение видит обычное дерево `/`, а на деле читает файл из того слоя, где он лежит выше всего. Запись попадает только в верхний слой, поэтому десять контейнеров из одного образа не копируют его десять раз и не мешают друг другу.

Отсюда и скорость старта: запустить контейнер — значит создать namespaces, cgroup и смонтировать готовые слои. Ядро уже работает, загружать нечего, поэтому время старта — это в основном время старта самого `dotnet`. Цена — изоляция слабее, чем у VM: уязвимость в ядре затрагивает все контейнеры на хосте, поэтому процессы внутри не должны работать от root (п. 8).

Наш сервис в контейнере — это процесс `dotnet DemoApi.dll`, который видит только свои файлы и свою сеть.

3. Docker Desktop на Windows

```
flowchart LR
  CLI["docker CLI<br/>PowerShell"] -->|Docker API| Engine
  UI["Docker Desktop<br/>окно приложения"] --> Engine
  subgraph VM["Linux VM в WSL 2"]
    Engine["Docker Engine"] --> C1["контейнер api<br/>dotnet DemoApi.dll"]
  end
  Engine <-->|pull / push| Reg["Реестры<br/>MCR · Docker Hub · GHCR"]
```

- Linux-контейнерам нужно ядро Linux, а у Windows его нет. Поэтому Docker Desktop держит лёгкую VM в **WSL 2** — это настоящее ядро Linux от Microsoft, запущенное под Hyper-V. В `wsl -l -v` она видна как дистрибутив `docker-desktop`.
- **Docker Engine** внутри VM создаёт контейнеры, хранит образы и собирает их. Команда `docker` в PowerShell и окно Docker Desktop — два клиента одного движка: всё, что вы делаете в терминале, видно в окне, и наоборот.
- Опубликованные порты контейнеров доступны на `localhost` Windows: Docker Desktop сам пробрасывает их из VM.
- Образы и контейнеры лежат не на диске C: как обычные файлы, а внутри виртуального диска VM. Поэтому место после экспериментов освобождают командами Docker (`docker system df`, `docker image prune`), а не в Проводнике.

Разделение на клиент и сервер хорошо видно в `docker version`:

```
Client:
 Version:           28.5.x
 OS/Arch:           windows/amd64
...
Server: Docker Desktop
 Engine:
  Version:          28.5.x
  OS/Arch:          linux/amd64
```

Клиент — Windows-программа, движок — Linux. Когда вы пишете `docker run`, клиент лишь отправляет HTTP-запрос к Docker API, а всю работу делает движок в VM. Тот же API использует окно Docker Desktop, а в теме 5 — раннер GitHub Actions.

Что происходит при `docker run -d --name api -p 8080:8080 demo-api:1.0`:

1. Движок ищет образ `demo-api:1.0` локально. Если его нет — скачивает из реестра, причём только те слои, которых ещё нет на машине.
2. Создаёт контейнер: поверх слоёв образа монтирует пустой записываемый слой, записывает настройки (имя, переменные окружения, порты).
3. Подключает контейнер к сети: создаёт ему сетевой интерфейс с внутренним IP и правило, по которому порт 8080 хоста ведёт в порт 8080 контейнера. Docker Desktop дополнительно пробрасывает этот порт из VM на `localhost` Windows.
4. Создаёт namespaces и cgroup и запускает в них `ENTRYPOINT` образа — процесс `dotnet DemoApi.dll`, который получает PID 1.
5. Начинает собирать stdout процесса в лог и выполнять `HEALTHCHECK` по расписанию.

С флагом `-d` клиент печатает id контейнера и сразу возвращает управление, а контейнер продолжает работать в VM и после закрытия терминала.

Демонстрация для преподавателя. Показать, что контейнер — это процесс с ограниченной видимостью:

```
docker version                          # клиент windows, сервер linux
docker run -d --name api -p 8080:8080 demo-api:1.0
docker top api                          # процесс dotnet DemoApi.dll, видимый со стороны движка
docker exec api hostname                # id контейнера — отдельный UTS namespace
docker exec api ls /                    # корень из слоёв образа, а не диск хоста
```

4. Образ, контейнер, реестр

Понятие	Суть	В нашем проекте
Образ	Неизменяемый шаблон: файловая система из слоёв + команда запуска	`demo-api:1.0` — наш сервис; `mcr.microsoft.com/dotnet/aspnet:10.0` — основа, на которой он построен
Слой	Изменение файловой системы от одной инструкции Dockerfile. Слои переиспользуются между образами	Слой с рантаймом .NET скачивается один раз и общий для всех .NET-образов
Тег	Имя версии образа. `latest` — просто тег по умолчанию, **не** «самый свежий»	`demo-api:1.0`, `demo-api:1.1`; в теме 5 — `sha-abc1234`
Контейнер	Запущенный экземпляр образа: слои образа (только чтение) + тонкий записываемый слой + процесс	`api` — работающий сервис
Реестр	Хранилище образов	MCR (образы .NET от Microsoft), Docker Hub (postgres), GHCR (наш образ, тема 5)

Из одного образа можно запустить сколько угодно контейнеров — как из одного класса много объектов.

Полное имя образа читается слева направо: `mcr.microsoft.com/dotnet/aspnet:10.0` — это реестр `mcr.microsoft.com`, репозиторий `dotnet/aspnet`, тег `10.0`. Если реестр не указан (`postgres:18-alpine`), Docker идёт на Docker Hub; если не указан тег — подставляет `latest`.

Откуда берётся `aspnet:10.0`. Образы .NET собирает Microsoft, их Dockerfile открыты в репозитории `dotnet/dotnet-docker` на GitHub. Они выстроены цепочкой: `runtime-deps` (Ubuntu и системные библиотеки, нужные .NET) → `runtime` (+ рантайм .NET) → `aspnet` (+ ASP.NET Core). SDK-образ строится на них же и добавляет компилятор и NuGet. Поэтому слои нижних образов общие, и после `docker pull` рантайма сборка на SDK скачивает только недостающее.

Тег и digest. Тег — подвижная ссылка: Microsoft регулярно пересобирает образы с исправлениями безопасности и патч-версиями .NET, и `aspnet:10.0` сегодня и через месяц — разные образы. **Digest** (`sha256:...`) — хэш содержимого, он никогда не меняется. Пулл по digest гарантирует ровно те же байты:

```
docker image ls --digests mcr.microsoft.com/dotnet/aspnet
docker pull mcr.microsoft.com/dotnet/aspnet:10.0@sha256:<digest>
```

Один тег может указывать сразу на несколько вариантов образа под разные архитектуры (amd64, arm64). Docker сам выбирает подходящий вашему процессору, поэтому одна и та же строка `FROM` работает и на ноутбуке с Intel, и на ARM-сервере.

Почему `latest` — плохо. Тег `latest` ставится по умолчанию, когда тег забыли, и ничего не говорит о версии. Типичная история: сервер перезапустили, Docker скачал `latest`, а это уже другая сборка — сервис ведёт себя иначе, и никто не может сказать, что было запущено вчера. Откатиться тоже некуда: предыдущий `latest` перезаписан. Поэтому в индустрии свои образы тегируют неизменяемыми версиями — номером релиза или sha коммита (так сделаем в теме 5), а базовые образы — как минимум мажорной версией, а в строгих командах — digest, который обновляет бот вроде Dependabot (тема 4).

5. Dockerfile: рецепт образа

Dockerfile — текстовый файл с инструкциями, по которым `docker build` собирает образ. Он лежит в git рядом с кодом, поэтому образ одинаково собирается у вас, у коллеги и в CI.

Инструкция	Что делает
`FROM image AS name`	Базовый образ, на котором строим, и имя этапа

`WORKDIR /app`	Рабочая папка внутри образа
`COPY src dst`	Скопировать файлы из **контекста сборки** (папки, указанной в `docker build`)
`RUN cmd`	Выполнить команду при сборке → новый слой
`ARG` / `ENV`	Переменная сборки / переменная окружения для запущенного контейнера
`USER`	От чьего имени работает процесс
`EXPOSE 8080`	Документация: какой порт слушает приложение
`HEALTHCHECK`	Команда, которой Docker проверяет, что приложение живо
`ENTRYPOINT`	Главный процесс контейнера

Контекст сборки. В `docker build -t demo-api:1.0 .` точка — это папка, файлы которой отправляются движку. `COPY` видит только их. Лишнее исключают через `.dockerignore`.

Что происходит при `docker build`. Сборку выполняет не клиент, а сборщик **BuildKit** внутри движка. Клиент упаковывает контекст и отправляет его в VM — первая строка вывода так и называется: `transferring context`. Дальше BuildKit читает Dockerfile, строит граф шагов и выполняет их: для каждой `RUN` поднимает временный контейнер из предыдущих слоёв, выполняет команду и сохраняет изменения файлов как новый слой. Независимые этапы multi-stage сборки он умеет выполнять параллельно, а этапы, которые не нужны итоговому образу, пропускает.

Зачем `.dockerignore`. Без него в контекст уедут `bin/obj`, папка `.git` и `.vs` — это десятки мегабайт, которые каждый раз передаются в VM. Хуже того, `COPY src/ src/` принесёт в Linux-сборку папки `obj` из Windows: в них лежит `project.assets.json` с путями к пакетам вида `C:\Users\...\.nuget\packages`. Он перезапишет результат `restore`, сделанного внутри образа, и `publish --no-restore` упадёт с ошибкой про ненайденные пакеты. Наш `.dockerignore` также отрезает `tests/`, `db/`, `monitoring/`, `*.md` и `.env`: секреты из `.env` не должны попасть даже во временный слой сборки.

6. Слои и кэш

Каждая `COPY` или `RUN` создаёт слой. При повторной сборке Docker берёт слой из кэша, если не изменились ни инструкция, ни её входные файлы. **Изменился слой — пересобираются он и все слои после него.**

Для `COPY` Docker сравнивает контрольные суммы копируемых файлов, для `RUN` — только текст команды. Это важно: `RUN apt-get update` с тем же текстом будет браться из кэша неделями, даже если в репозитории пакетов уже вышли обновления. Поэтому время от времени образы пересобирают без кэша (`--no-cache`), а в CI — с явным обновлением базового образа (`--pull`).

```
# Плохо: любое изменение кода → заново restore всех NuGet-пакетов
COPY . .
RUN dotnet restore

# Хорошо: restore кэшируется, пока не меняется .csproj
COPY src/DemoApi/DemoApi.csproj src/DemoApi/
RUN dotnet restore src/DemoApi/DemoApi.csproj
COPY src/ src/
```

Правило: что меняется редко — выше, что часто — ниже. Код правится каждый день, список пакетов — раз в неделю. Правильный порядок превращает сборку из минут в секунды.

Так выглядит повторная сборка после правки кода (`docker build --progress=plain`, фрагмент; номера шагов у вас будут другими):

```
#9 [build 3/6] COPY src/DemoApi/DemoApi.csproj src/DemoApi/
#9 CACHED

#10 [build 4/6] RUN dotnet restore src/DemoApi/DemoApi.csproj
#10 CACHED

#11 [build 5/6] COPY src/ src/
#11 DONE 0.1s

#12 [build 6/6] RUN dotnet publish src/DemoApi/DemoApi.csproj -c Release -o /app --no-restore
```

`csproj` не менялся, поэтому restore взят из кэша. Изменились файлы в `src/` — с этого шага и до конца всё выполняется заново.

Слой нельзя «похудеть» следующим слоем. Если скачать файл в одном `RUN`, а удалить в следующем, файл останется в предыдущем слое и в размере образа — верхний слой лишь пометит его как удалённый. Поэтому установка и уборка делаются в одном `RUN` через `&&`, как с `apt-get` в нашем Dockerfile. Посмотреть, какой слой сколько весит, можно командой `docker image history demo-api:1.0`.

7. Multi-stage build

```
flowchart LR
  A["Этап build<br/>dotnet/sdk:10.0<br/>restore → publish"] -->|COPY --from=build /app| B["Этап
runtime<br/>dotnet/aspnet:10.0<br/>только DLL"]
  B --> I["Итоговый образ"]
```

- **SDK-образ** содержит компилятор, NuGet и всё для сборки — он большой.
- **Runtime-образ** содержит только ASP.NET Core runtime.
- В итоговый образ попадает **только последний этап**. Исходники, SDK и промежуточные файлы остаются за бортом.

Зачем: образ в несколько раз меньше — быстрее скачивается при деплое. В нём нет компилятора и исходников — меньше возможностей для атаки. Сравните сами: `docker image ls` покажет размеры `sdk:10.0`, `aspnet:10.0` и `demo-api` рядом.

Базовый runtime-образ	Особенности
`aspnet:10.0`	Ubuntu, есть shell и apt: удобно учиться и отлаживать. **Наш выбор**
`aspnet:10.0-alpine`	Меньше, другая стандартная библиотека C (musl)
`aspnet:10.0-noble-chiseled`	Без shell и пакетного менеджера: меньше и безопаснее, но `docker exec ... sh` не сработает