Неделя 4. Лекция 2. Docker: сервис задач в контейнере

Часть 2 из 2. Начало лекции — в материалах предыдущей недели.

8. Наш Dockerfile построчно

Это `Dockerfile` из корня репозитория, который вы напишете в лабе.

```
# syntax=docker/dockerfile:1
```

Указывает BuildKit, какую версию синтаксиса Dockerfile использовать. Сборщик скачает актуальный парсер версии 1.x, и новые возможности синтаксиса работают независимо от версии Docker на машине.

```
FROM mcr.microsoft.com/dotnet/sdk:10.0 AS build
WORKDIR /src
```

Первый этап на SDK-образе, имя `build` нужно, чтобы потом забрать из него результат. `WORKDIR` создаёт папку, если её нет, и делает её текущей для следующих инструкций — вместо `RUN cd`, который действует только внутри одной команды.

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

Сначала только `.csproj` и restore: слой с NuGet-пакетами живёт в кэше, пока не меняются зависимости (п. 6). Путь `src/DemoApi/` повторяет структуру репозитория — на неё рассчитаны относительные пути в проекте.

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

Затем весь код и публикация Release-сборки в `/app`. `--no-restore` не даёт повторно ходить в NuGet: пакеты уже восстановлены слоем выше. Тесты и `db/` сюда не попадают — их отрезал `.dockerignore`.

```
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/*
```

Второй этап начинается с чистого листа: от первого в нём нет ничего, пока мы явно не скопируем. `curl` нужен только для `HEALTHCHECK`. Все три команды — в одном `RUN`: списки пакетов удаляются в том же слое, где скачаны, а `--no-install-recommends` не тянет необязательные зависимости.

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

Из этапа `build` берём только результат `publish`: DLL, `appsettings*.json`, зависимости. Файлы принадлежат root, и процесс сервиса может их читать, но не менять — подменить код изнутри контейнера нельзя.

```
ARG APP_VERSION=dev
ENV APP_VERSION=$APP_VERSION
```

`ARG` существует только во время сборки, `ENV` остаётся в образе и видна запущенному процессу. Связка переносит версию из команды сборки (`--build-arg APP_VERSION=...`, в CI — sha коммита) в переменную окружения, которую читает `GET /`. Без аргумента версия будет `dev`.

```
USER $APP_UID
EXPOSE 8080
```

Базовые образы .NET 8+ содержат пользователя `app` и переменную `APP_UID` с его числовым id. Числовой id предпочтительнее имени: оркестраторы вроде Kubernetes по нему проверяют, что контейнер не запущен от root. `EXPOSE` ничего не открывает, это документация для людей и инструментов.

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

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

Проверка здоровья (п. 10) и главный процесс контейнера в exec-форме — массивом (п. 11).

9. ASP.NET в контейнере: что важно

- **Порт 8080.** В образах .NET 8+ приложение слушает 8080: базовый образ задаёт переменную `ASPNETCORE_HTTP_PORTS=8080`. Раньше был порт 80, но в Linux порты ниже 1024 может открыть только root, а образы перешли на непривилегированного пользователя. Наружу порт публикуют флагом `-p`.
- **HTTPS внутри контейнера обычно не нужен:** шифрование делает прокси или балансировщик перед сервисом. Внутри остаётся простой HTTP, и не нужно раскладывать сертификаты по образам.
- **Окружение по умолчанию — `Production`.** `appsettings.Development.json` не применяется, страницы `/swagger` нет. `launchSettings.json`, где Visual Studio задаёт `Development`, в образ не попадает. Окружение меняют переменной `ASPNETCORE_ENVIRONMENT`.
- **Настройки — через переменные окружения** (`-e`), не пересобирая образ: `-e Storage=Postgres`. Вложенные ключи пишутся через двойное подчёркивание: `ConnectionStrings__Db`. Это принцип «один образ — все окружения»: тестовый стенд и прод запускают одни и те же байты с разными переменными.
- **Непривилегированный пользователь.** Процесс работает от `app`, а не от root. Если через уязвимость в приложении кто-то выполнит команду внутри контейнера, у него не будет прав менять систему, ставить пакеты и перезаписывать `/app`.
- **Логи — в stdout.** Docker перехватывает вывод главного процесса и сохраняет его (`docker logs`). Писать в файлы внутри контейнера бессмысленно: они пропадут вместе с контейнером. В `Production` наш сервис пишет логи в JSON — по одной строке на событие, их удобно фильтровать инструментами (тема 6).
- **Состояние в памяти живёт, пока жив процесс.** Остановили контейнер — данные хранилища `InMemory` пропали. Это главная тема следующей лекции.

При старте в `docker logs api` видно, что сервис поднялся в `Production` на порту 8080 (строки сокращены):

```
{"EventId":14,"LogLevel":"Information","Category":"Microsoft.Hosting.Lifetime","Message":"Now listening on:
http://[::]:8080",...}
{"EventId":0,"LogLevel":"Information","Category":"Microsoft.Hosting.Lifetime","Message":"Hosting environment:
Production",...}
{"EventId":0,"LogLevel":"Information","Category":"Microsoft.Hosting.Lifetime","Message":"Content root path: /app",...}
```

10. HEALTHCHECK

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

Docker каждые 10 секунд выполняет команду внутри контейнера. Статусы видны в `docker ps` и в Docker Desktop: `starting` → `healthy` / `unhealthy`. В базовом aspnet-образе нет `curl`, поэтому его ставят отдельным `RUN`. Почему проверяется именно `live`, — в лекции 6.

Параметр	Смысл
`--interval=10s`	Как часто проверять
`--timeout=3s`	Проверка дольше 3 секунд считается неудачной
`--start-period=15s`	Время на старт: неудачи в этот период не считаются
`--retries=3`	После трёх неудач подряд — `unhealthy`

`curl -f` возвращает ошибку на HTTP-коде 4xx/5xx, а `|| exit 1` превращает любую ошибку в код 1, который Docker понимает как «нездоров». Сам Docker **не перезапускает** unhealthy-контейнер — он только ставит статус. Этим статусом пользуются другие: Compose ждёт `healthy` перед стартом зависимых сервисов (тема 3), скрипт деплоя проверяет его после выкладки (тема 5).

11. Жизненный цикл контейнера

```
stateDiagram-v2
  [*] --> created: docker create
  created --> running: docker start
  running --> exited: docker stop / процесс завершился
  exited --> running: docker start / restart policy
  exited --> [*]: docker rm
```

Контейнер живёт, пока жив его главный процесс (PID 1). Для нашего сервиса это `dotnet DemoApi.dll`. Завершился процесс — контейнер становится `exited` с кодом выхода процесса. Контейнер при этом не удаляется: его записываемый слой и настройки остаются, его можно снова запустить (`docker start`) или удалить (`docker rm`). `docker run` — это `create` и `start` одной командой.

ENTRYPOINT и CMD. Обе инструкции задают, что запустить. `ENTRYPOINT` — сама программа, `CMD` — аргументы по умолчанию, которые заменяются всем, что написано после имени образа в `docker run`. Если есть только `CMD`, аргументы `docker run` заменяют команду целиком. Для сервиса удобнее `ENTRYPOINT`: контейнер `demo-api` всегда запускает сервис, а не что-то случайное.

Exec-форма и сигналы. `ENTRYPOINT ["dotnet", "DemoApi.dll"]` запускает `dotnet` напрямую, и он получает PID 1. Строковая форма `ENTRYPOINT dotnet DemoApi.dll` запускает команду через `/bin/sh -c`: главным процессом становится оболочка, и документация Docker прямо предупреждает, что сигналы до приложения тогда не доходят. Внешне всё работает, пока не понадобится остановить контейнер.

Что происходит при `docker stop`. Docker посылает PID 1 сигнал **SIGTERM** и ждёт 10 секунд. ASP.NET обрабатывает SIGTERM: перестаёт принимать новые запросы, дожидается текущих и завершается с кодом 0 — в логе появляется `Application is shutting down...`. Если за 10 секунд процесс не завершился, Docker посылает **SIGKILL**, который перехватить нельзя, и код выхода будет 137. Коды выхода подробно разберём в теме 6.

```
sequenceDiagram
  participant U as docker stop api
  participant E as Docker Engine
  participant P as PID 1<br/>dotnet DemoApi.dll
  U->>E: остановить контейнер
  E->>P: SIGTERM
  P-->>P: не принимать новые запросы,<br/>дождаться текущих
  P-->>E: процесс завершился, код 0
  E-->>U: Exited (0)
  Note over E,P: не успел за 10 с → SIGKILL, код 137
```

У PID 1 в Linux есть особенность: ядро не применяет к нему действия сигналов по умолчанию. Обычный процесс без обработчика SIGTERM просто завершится, а PID 1 без обработчика сигнал проигнорирует. .NET обработчик ставит, поэтому наш сервис останавливается сразу. Для программ, которые этого не умеют, в `docker run` есть флаг `--init`: он запускает маленький init-процесс, который становится PID 1 и пересылает сигналы приложению.

Типичная история из практики: сервис запускали через shell-скрипт в `ENTRYPOINT`, и каждый деплой «зависал» на 10 секунд на остановке старой версии, а запросы, которые она обслуживала, обрывались по SIGKILL. Проблему годами списывали на медленный сервер, пока кто-то не посмотрел на код выхода 137 вместо 0. Лечится одной строкой: `exec` в конце скрипта или exec-форма `ENTRYPOINT`.

Демонстрация для преподавателя. Показать корректную остановку и разницу между `stop` и `kill`:

```
        docker logs -f api                      # во втором окне
        docker stop api                         # в логе: Application is shutting down...
        docker ps -a --filter name=api          # Exited (0)
        docker start api; docker kill api       # SIGKILL сразу
        docker ps -a --filter name=api          # Exited (137)
```

12. Размер и безопасность образа

Размер образа — это сумма всех его слоёв, и в основном он определяется базовым образом. Наш код и DLL пакетов — это единицы мегабайт поверх рантайма .NET и Ubuntu. Что влияет на размер:

Фактор	Что делать
Базовый образ	Runtime вместо SDK (multi-stage); при необходимости `alpine` или `chiseled`
Лишние файлы в контексте	`.dockerignore`

Мусор пакетных менеджеров	Установка и уборка в одном `RUN`, `--no-install-recommends`
Удаление в отдельном слое	Не помогает — файл остаётся в предыдущем слое

Размер важен не ради экономии диска. Образ скачивается на каждый сервер при деплое и при переезде контейнера, и маленький образ — это быстрый старт новой версии и быстрый откат.

Безопасность образа складывается из нескольких простых правил:

- **Меньше содержимого — меньше уязвимостей.** Каждый пакет в образе — потенциальная CVE. SDK, компилятор и исходники в рантайм-образе не нужны.
- **Не root.** `USER $APP_UID` ограничивает последствия взлома приложения.
- **Секретов в образе нет.** Всё, что попало в слой или в `ENV`, достаётся любым, кто может скачать образ: `docker image history` и `docker inspect` показывают команды сборки и переменные. Пароли передают при запуске через переменные окружения или файлы (тема 3), а для секретов на этапе сборки у BuildKit есть отдельный механизм `RUN --mount=type=secret`.
- **Регулярная пересборка.** Базовый образ обновляется с исправлениями безопасности, но ваш образ получит их, только если его пересобрать. Поэтому в индустрии образы пересобирают по расписанию, а не только при изменении кода, и сканируют на известные уязвимости — например, встроенным в Docker Desktop `docker scout cves demo-api:1.1`.

13. Команды

Команда	Зачем
`docker build -t demo-api:1.0 .`	Собрать образ из Dockerfile в текущей папке
`docker image ls`, `docker image history demo-api:1.0`	Образы, слои образа
`docker run -d --name api -p 8080:8080 demo-api:1.0`	Создать и запустить контейнер
`docker ps` / `docker ps -a`	Работающие / все контейнеры
`docker logs -f api`	Логи процесса
`docker exec -it api sh`	Зайти внутрь контейнера
`docker inspect api`	Полное описание: настройки, порты, health
`docker stats`	CPU и память
`docker stop api` / `docker start api`	Остановить / запустить
`docker rm -f api`	Удалить (с остановкой)

Флаги `docker run`:

Флаг	Что делает
`-d`	Запустить в фоне
`--name api`	Имя контейнера
`-p 8080:8080`	Опубликовать порт `хост:контейнер`
`-e KEY=value`	Переменная окружения
`-it`	Интерактивно, с терминалом (для `sh`)
`--rm`	Удалить контейнер после завершения

Что вы увидите в `docker ps` через полминуты после запуска (часть колонок опущена):

```
CONTAINER ID   IMAGE          COMMAND                  STATUS                    PORTS                    NAMES
3f2a9c1b7d4e   demo-api:1.1   "dotnet DemoApi.dll"   Up 25 seconds (healthy)   0.0.0.0:8080->8080/tcp   api
```

`COMMAND` — это `ENTRYPOINT` из Dockerfile, в `STATUS` — результат `HEALTHCHECK`, в `PORTS` — публикация порта: все интерфейсы Windows на 8080 ведут в порт 8080 контейнера. Первые 12 символов `CONTAINER ID` — это и есть `host` в ответе `GET /`.

Порядок в `-p` — слева хост, справа контейнер. `-p 8082:8080` значит «на Windows слушать 8082 и пересылать в 8080 контейнера». Так в лабе рядом работают два контейнера из одного образа: внутри оба слушают 8080, но снаружи у них разные порты. Правая часть должна совпадать с портом, который слушает приложение, иначе запросы уйдут в пустоту.

То же доступно в окне **Docker Desktop → Containers**: логи, `Inspect`, `Exec` (терминал внутри), `Stats`.

Типичные ошибки

- **`COPY . .` до `restore`:** кэш бесполезен, каждая сборка качает пакеты заново.
- **Нет `.dockerignore`:** папки `bin/obj` из Windows попадают в сборку и ломают её.
- **Секреты в `ARG`/`ENV` Dockerfile:** они видны в `docker image history` и `docker inspect`.
- **Процесс работает от root.**
- **Перепутан порядок в `-p`:** слева порт хоста, справа — контейнера.
- **`latest` в проде:** непонятно, какая версия запущена, и откатиться нельзя.
- **Строковая форма `ENTRYPOINT`:** сигналы не доходят до приложения, остановка ждёт 10 секунд и заканчивается SIGKILL.
- **Уборка в отдельном `RUN`:** образ не уменьшается, удалённые файлы остаются в предыдущем слое.
- **Ожидать, что `HEALTHCHECK` перезапустит сервис:** Docker только выставляет статус `unhealthy`.
- **Ожидать, что данные в памяти переживут перезапуск контейнера.**

Вопросы для самопроверки

1. Почему контейнер стартует быстрее виртуальной машины? Какие механизмы ядра делают процесс «контейнером»?
2. Почему сначала копируется `.csproj`, а потом код? Что попадёт в кэш, если изменить только `TaskService.cs`?
3. Что попадёт в итоговый образ после multi-stage сборки, а что нет?
4. Почему в контейнере нет страницы `/swagger`, хотя в Visual Studio она была? Как её включить без пересборки?
5. Почему ASP.NET в контейнере слушает 8080, а не 80?
6. Чем тег отличается от digest? Почему `aspnet:10.0` через месяц может оказаться другим образом?
7. Что произойдёт при `docker stop`, если `ENTRYPOINT` записан строкой, а не массивом?
8. Что случится с проектами, созданными через API, после `docker stop` и `docker start`? Почему?

Практика: лаба 2

Напишете Dockerfile сервиса задач по шагам, соберёте образ `demo-api` и проверите кэш слоёв. Запустите сервис на Docker Desktop, поработаете с ним из терминала и из окна Docker Desktop, увидите, что данные в памяти не переживают перезапуск, и включите страницу Swagger в контейнере через переменную окружения.