Тема 2

Docker: сервис задач в контейнере



Упаковываем сервис так, чтобы он одинаково запускался везде

План лекции

Сегодня

1. Зачем контейнеры
2. Контейнер против виртуальной машины
3. Образ, слой, тег, реестр
4. Dockerfile, кэш и multi-stage
5. ASP.NET в контейнере и жизненный цикл

После лекции вы сможете

- Объяснить, что такое контейнер и образ
- Написать Dockerfile для ASP.NET
- Собрать образ и запустить сервис на Docker Desktop
- Понимать, почему данные в памяти не переживают перезапуск

«У меня работает»



Без контейнеров

- На каждой машине ставить .NET нужной версии
- Копировать DLL и настройки руками
- На сервере — десяток сервисов с разными версиями
- Работает у одного, падает у другого



С контейнером

- Образ: рантайм, библиотеки, наши DLL
- Один артефакт от ноутбука до прода
- Старт за секунды
- Сервисы изолированы и не конфликтуют

Образ сервиса задач одинаково запустится у вас, у коллеги, в CI и на сервере

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 — обычный процесс Linux с ограниченной видимостью и ресурсами

Как ядро изолирует процесс



Namespaces — видимость

У процесса свои PID, сеть, файловая система и hostname. Он «думает», что один на машине



Cgroups — ресурсы

Сколько CPU и памяти можно. Превысил лимит памяти — ядро убьёт процесс (тема 6)

Docker Desktop на Windows



Вы

Команда `docker` в PowerShell или окно Docker Desktop





Docker Engine

Работает в лёгкой Linux-VM на WSL 2





Контейнеры

`api`: процесс `dotnet DemoApi.dll`





Реестры

MCR, Docker Hub, GHCR: pull и push образов

Терминал и окно Docker Desktop — два клиента одного движка. Порты контейнеров видны на localhost

Образ, слой, тег, контейнер, реестр

Образ

Шаблон: файлы из слоёв + команда запуска. `demo-api:1.0`

Слой

Изменение от одной инструкции. Общие слои скачиваются один раз

Тег

Имя версии. `latest` — просто тег по умолчанию, не «самый свежий»

Контейнер

Запущенный образ: слои + записываемый слой + процесс

Реестр

Хранилище образов: MCR, Docker Hub, GHCR (тема 5)

Один образ — много контейнеров

Как один класс и много объектов

Dockerfile: рецепт образа

Лежит в git рядом с кодом — образ одинаково собирается у всех и в CI

Инструкция	Что делает
`FROM image AS name`	Базовый образ и имя этапа
`WORKDIR /app`	Рабочая папка внутри образа
`COPY src dst`	Скопировать файлы из контекста сборки
`RUN cmd`	Выполнить команду при сборке → новый слой
`ARG` / `ENV`	Переменная сборки / переменная окружения
`USER`	От чьего имени работает процесс
`EXPOSE` / `HEALTHCHECK`	Порт приложения / проверка, что оно живо
`ENTRYPOINT`	Главный процесс контейнера

Слои и кэш: порядок решает

```
# Плохо: любое изменение кода → заново restore
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/
```

1. **Кэш по слоям**

   Не изменились инструкция и файлы — слой берётся из кэша

2. **Цепная реакция**

   Изменился слой — пересобираются он и все ниже

3. **Правило**

   Что меняется редко — выше. Код — ниже зависимостей

Multi-stage build

Этап build

`sdk:10.0`: компилятор, NuGet.
`restore → publish`

COPY --from=build

Забираем только готовые DLL

Этап runtime

`aspnet:10.0`: только рантайм и наши DLL

В образ попадает только последний этап: без SDK и исходников. Наш образ — около 240 МБ

Наш Dockerfile: этап запуска

```
FROM mcr.microsoft.com/dotnet/aspnet:10.0 AS runtime
RUN apt-get update && apt-get install -y curl
WORKDIR /app
COPY --from=build /app .
ARG APP_VERSION=dev
ENV APP_VERSION=$APP_VERSION
USER $APP_UID
EXPOSE 8080
HEALTHCHECK CMD curl -fsS http://localhost:8080/health/live
ENTRYPOINT ["dotnet", "DemoApi.dll"]
```

1. **curl**
   Нужен для HEALTHCHECK
2. **APP_VERSION**
   Версия видна в `GET /`
3. **USER $APP_UID**
   Не root: пользователь `app`
4. **HEALTHCHECK**
   Docker проверяет `/health/live`

ASP.NET в контейнере: что важно

Порт 8080

Наружу публикуем флагом `-p 8080:8080`

Окружение Production

Нет `/swagger`, не читается `appsettings.Development.json`

Настройки через `-e`

Меняем поведение без пересборки образа

Не root

Взлом приложения не даёт прав администратора

Логи в stdout

Их собирает Docker: `docker logs`

Память — временная

Остановили контейнер — данные `InMemory` пропали

Жизненный цикл контейнера



running

`docker run` / `docker start`. Работает процесс сервиса





exited

`docker stop` или процесс упал. Контейнер остался





снова running

`docker start`: тот же контейнер, новый процесс





удалён

`docker rm`: пропал вместе с записываемым слоем

Контейнер живёт, пока жив его главный процесс. HEALTHCHECK показывает `healthy` / `unhealthy`

Команды на каждый день

Команда	Зачем
`docker build -t demo-api:1.0 .`	Собрать образ
`docker run -d --name api -p 8080:8080 demo-api:1.0`	Запустить контейнер
`docker ps` / `docker ps -a`	Работающие / все контейнеры
`docker logs -f api`	Логи процесса
`docker exec -it api sh`	Зайти внутрь
`docker inspect api`	Настройки, порты, health
`docker stop api` / `docker start api`	Остановить / запустить
`docker rm -f api`	Удалить

То же — в окне Docker Desktop: Logs, Inspect, Exec, Stats

Типичные ошибки

- `COPY . .` до `restore`: кэш бесполезен
- Нет `.dockerignore`: `bin/obj` из Windows ломают сборку
- Секреты в `ARG`/`ENV` Dockerfile: видны в `docker image history`
- Процесс работает от root
- Перепутан `-p`: слева порт хоста, справа — контейнера
- `latest` в проде: непонятно, что запущено

Проверьте себя

1. Почему контейнер стартует быстрее виртуальной машины?
2. Почему сначала копируется `.csproj`, а потом код?
3. Что попадёт в итоговый образ после multi-stage, а что нет?
4. Почему в контейнере нет `/swagger`? Как включить без пересборки?
5. Что станет с проектами после `docker stop` и `docker start`?



Практика

Лаба 2. Dockerfile и сервис на Docker Desktop

Упаковываем сервис задач в образ и работаем с ним в контейнере

- Пишем Dockerfile по шагам и проверяем кэш слоёв
- Запускаем на Docker Desktop, смотрим Logs, Exec, Stats
- Убеждаемся: данные в памяти не переживают перезапуск
- Включаем `/swagger` в контейнере переменной окружения