Неделя 11. Лекция 6. Жизнь контейнера: логи, здоровье, рестарты, ресурсы

Часть 1 из 2.

После лекции: вы понимаете, что происходит с контейнером при сбоях, и по логам, событиям и exit-коду определяете, почему он упал. Знаете, как настроить сервис, чтобы он поднимался сам и не мог положить весь сервер. **Инструменты:** Docker Desktop (CLI и вкладка Containers), PowerShell 7.

1. Сервис в эксплуатации: что может пойти не так

После темы 5 сервис задач работает в контейнере и обновляется сам. Дальше начинается эксплуатация: сервис живёт неделями, через него идут запросы, рядом работает БД, а код регулярно меняется. Сбои в такой жизни неизбежны, и они очень разные по внешним признакам:

Сбой	Как выглядит для пользователя	Что с контейнером
Ошибка в запросе	500 на одном запросе	Жив
Падение процесса	Сервис недоступен несколько секунд	Завершился, Docker перезапустил
Утечка памяти	Тормоза, затем недоступность	Убит ядром (OOM), перезапущен
Упала БД	500 на запросах к данным	Жив, но не готов
Сервис завис	Запросы висят	Жив, healthcheck не проходит
Медленные ответы	Всё работает, но долго	Жив и healthy

Обратите внимание на правую колонку: в половине случаев контейнер формально в порядке. Поэтому `docker ps` — это только первая команда в разборе, а не ответ на вопрос «что случилось».

Задача лекции — научиться **восстанавливать картину по следам**: событиям Docker, exit-коду, логам, потреблению ресурсов. И настроить контейнер так, чтобы часть проблем решалась без человека, а остальные не задевали соседей по серверу.

2. Состояния контейнера и события Docker

В лекции 2 жизненный цикл был простым: `created → running → exited`. В эксплуатации к нему добавляются состояние `restarting` и статус здоровья, который живёт внутри `running`:

```
stateDiagram-v2
  [*] --> created: docker create
  created --> running: docker start
  running --> exited: процесс завершился<br/>или docker stop
  exited --> restarting: restart policy<br/>разрешает перезапуск
  restarting --> running: пауза, новый запуск
  exited --> running: docker start
  exited --> [*]: docker rm
  state running {
    [*] --> starting
    starting --> healthy: проверка прошла
    healthy --> unhealthy: retries проверок подряд упали
    unhealthy --> healthy: проверка прошла
  }
```

Каждый переход Docker публикует как **событие**: `create`, `start`, `kill`, `die`, `oom`, `stop`, `restart`, `health_status`, `destroy`. События — это хронология, по которой разбирают сбой. Смотреть их удобно в отдельном терминале с фильтром по контейнеру:

```
docker events --filter container=demo-api-1 --filter event=die --filter event=oom --filter event=start
```

```
2026-09-19T12:03:41.51+05:00 container oom 8c1e… (…, image=demo-api:local, name=demo-api-1)
2026-09-19T12:03:41.62+05:00 container die 8c1e… (…, exitCode=137, image=demo-api:local, name=demo-api-1)
2026-09-19T12:03:41.87+05:00 container start 8c1e… (…, image=demo-api:local, name=demo-api-1)
```

Три строки рассказывают всю историю: ядро убило процесс из-за памяти, он завершился с кодом 137, Docker запустил контейнер заново. Без фильтра поток забьют события `exec_create`, `exec_start`, `exec_die`: их каждые 10 секунд порождает HEALTHCHECK, потому что проверка — это команда, выполняемая внутри контейнера.

У событий есть ограничение: Docker хранит их в памяти, небольшим буфером. После перезапуска Docker Desktop

прошлые события пропадают. Поэтому `docker events --since 30m --until 0s` полезен сразу после сбоя, а для истории за дни нужен мониторинг (тема 7).

3. Exit-коды и сигналы

Exit-код — число, с которым завершился главный процесс контейнера (PID 1). Код 0 означает успех, всё остальное — ошибку. Если процесс убит сигналом, Linux сообщает код `128 + номер сигнала`: отсюда 137 и 143.

Сигнал — уведомление, которое ядро посылает процессу. **SIGTERM** (15) означает «аккуратно заверши работу», процесс может его перехватить. **SIGKILL** (9) означает «убить немедленно», перехватить его нельзя, процесс не успевает ничего записать.

Exit code	Причина	Что обычно стоит за этим
0	Штатное завершение	`docker stop`, приложение обработало SIGTERM
1	Приложение само завершилось с ошибкой	`Environment.Exit(1)`, как в `/api/chaos/crash`
134	SIGABRT (128+6)	Процесс сам себя аварийно прервал; в .NET на Linux так обычно заканчивается необработанное исключение вне запроса
137	SIGKILL (128+9)	**OOM-killer** или процесс не уложился в grace period при остановке
139	SIGSEGV (128+11)	Обращение к недопустимой памяти, почти всегда в нативном коде
143	SIGTERM (128+15)	Процесс получил SIGTERM и не обработал его

Код 137 — самый частый и самый неоднозначный: SIGKILL посылает и ядро при нехватке памяти, и Docker, когда процесс не остановился вовремя. Различают их по событиям: перед OOM есть событие `oom`, перед принудительной остановкой — `kill` с `signal=15`, а через grace period ещё один `kill` с `signal=9`. Второй признак — поле `State.OOMKilled`:

```
docker inspect demo-api-1 --format 'Exit={{.State.ExitCode}} OOMKilled={{.State.OOMKilled}} Restarts={{.RestartCount}}'
```

Поле работает только до перезапуска. Когда restart policy снова поднимает контейнер, Docker записывает новое состояние, и `ExitCode` с `OOMKilled` сбрасываются. `RestartCount` при этом растёт, так что по нему видно, что падения были, но не видно почему. Правду о прошлом падении хранят события, логи и мониторинг.

4. Как останавливается контейнер

`docker stop` (и `docker compose stop`, и остановка при деплое) работает в три шага: посылает главному процессу **SIGTERM**, ждёт **grace period** (по умолчанию 10 секунд, в compose задаётся `stop_grace_period`), и если процесс ещё жив — посылает **SIGKILL**.

ASP.NET обрабатывает SIGTERM сам: Kestrel перестаёт принимать новые соединения, приложение дожидается текущих запросов, останавливает фоновые службы и завершается с кодом 0. В логе это видно по записи `Application is shutting down...`. Такое завершение называют **graceful shutdown**: пользователь, чей запрос был в работе, получит нормальный ответ.

Здесь есть две ловушки:

- **Сигнал должен дойти до приложения.** В нашем Dockerfile `ENTRYPOINT ["dotnet", "DemoApi.dll"]` записан в exec-форме, и `dotnet` становится PID 1. В shell-форме (`ENTRYPOINT dotnet DemoApi.dll`) PID 1 была бы оболочка `sh`, которая не передаёт SIGTERM дальше. Приложение сигнала не увидит, через 10 секунд Docker его убьёт, и каждый деплой будет заканчиваться кодом 137 и оборванными запросами.
- **Таймауты должны сходиться.** ASP.NET по умолчанию готов ждать завершения запросов до 30 секунд (`HostOptions.ShutdownTimeout`), а Docker — только 10. Если у сервиса бывают долгие запросы, `stop_grace_period` увеличивают, иначе Docker убьёт процесс посреди работы.

Штатная остановка оставляет в событиях характерный след, по которому её легко отличить от падения:

```
container kill … (name=demo-api-1, signal=15)
container die … (exitCode=0, name=demo-api-1)
container stop … (name=demo-api-1)
```

`kill` с сигналом 15 — это и есть SIGTERM от `docker stop`, код 0 означает, что приложение успело завершиться само. Если между `kill signal=15` и `die` появился ещё `kill signal=9`, а код стал 137, приложение не уложилось в grace period.

5. Политики перезапуска

restart	Поведение
`no`	Никогда (по умолчанию)
`on-failure[:N]`	Только при exit ≠ 0, до N раз
`always`	Всегда; после перезапуска Docker поднимет даже остановленный вручную
`unless-stopped`	Всегда, кроме остановленных вручную. **Выбор для нашего стенда**

Остановка вручную — отдельный случай для всех политик. После `docker stop` Docker не поднимает контейнер сам, иначе его было бы невозможно выключить. Разница между `always` и `unless-stopped` проявляется только при перезапуске самого Docker, например после перезагрузки компьютера: `always` поднимет и то, что вы остановили специально.

Между перезапусками Docker делает паузу и удваивает её после каждой неудачи: 100 мс, 200 мс, 400 мс и так далее, но не больше минуты. Если контейнер после старта проработал хотя бы 10 секунд, пауза сбрасывается к начальной. В `docker ps` такой контейнер выглядит как `Restarting (1) 3 seconds ago`, а число попыток видно в `RestartCount`.

Рестарт лечит **случайные** сбои: редкий крэш, разовую утечку, зависшее соединение. Если сервис падает на старте каждый раз (нет БД, ошибка в конфигурации, неверная строка подключения), рестарты ничего не исправят — нужно читать логи.

Есть и обратная сторона: restart-цикл **скрывает проблему**. Типичная история: после релиза сервис начинает понемногу терять память, раз в час упирается в лимит и перезапускается. Контейнер всегда в статусе `Up`, дашборд зелёный, а у пользователей раз в час обрываются запросы и пропадает сессия. Замечают это через неделю, когда кто-то случайно смотрит на `RestartCount = 170`. Вывод: растущий счётчик рестартов — это инцидент, даже если сервис «работает». В теме 7 на него ставится алерт.

6. Здоровье: liveness, readiness, startup

Процесс может быть жив, но бесполезен: завис, потерял БД, ещё не прогрелся. Поэтому здоровье сервиса проверяют разными вопросами:

Проверка	Вопрос	Что проверяет	Что делают при провале	У нас
Liveness	Процесс жив и отвечает?	Только сам процесс	Перезапускают	`/health/live`
Readiness	Готов принимать трафик?	Зависимости: БД и т.п.	Временно не шлют запросы	`/health/ready`
Startup	Закончил запуск?	Что приложение стартовало	Не проверяют liveness до конца старта	`start_period` в HEALTHCHECK

В `Program.cs` сервиса задач (лекция 1):

```
        healthChecks.AddNpgSql(connectionString, name: "db", tags: ["ready"]);
        app.MapHealthChecks("/health/live",  new() { Predicate = _ => false });
        app.MapHealthChecks("/health/ready", new() { Predicate = c => c.Tags.Contains("ready") });
```

`Predicate = _ => false` означает «не выполнять ни одной проверки»: `/health/live` отвечает `Healthy`, если процесс вообще способен обработать HTTP-запрос. Этого и достаточно: зависший процесс не ответит за 3 секунды таймаута.

Почему БД нельзя проверять в liveness. Представьте несколько экземпляров сервиса и оркестратор, который перезапускает их по liveness. БД на минуту становится недоступна. Если liveness проверяет БД, «больными» разом становятся все экземпляры, и все одновременно перезапускаются. Рестарт БД не чинит, зато после её возвращения все экземпляры стартуют одновременно, заново открывают соединения и прогреваются. Минутная проблема в БД превращается в полную недоступность сервиса, которая длится дольше самого сбоя. Это называют **каскадным отказом**. Правильное поведение — остаться живым, отвечать 503 на `/health/ready` и дождаться БД.

HEALTHCHECK в Dockerfile описывает, как Docker проверяет контейнер:

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

Параметр	Значение у нас	Смысл
`interval`	10 с	Как часто запускать проверку
`timeout`	3 с	Сколько ждать ответа; дольше — провал
`start-period`	15 с	Время на запуск: провалы в этот период не засчитываются

`retries`	3	Сколько провалов подряд нужно для `unhealthy`

Команда проверки возвращает 0 (успех) или 1 (провал); `curl -f` возвращает ошибку на любой код 4xx/5xx. Статусы: `starting` сразу после запуска, `healthy` после первой успешной проверки, `unhealthy` после трёх провалов подряд, то есть примерно через 30 секунд проблем. Последние результаты проверок с выводом команды хранятся в `State.Health.Log`.

Как это выглядит на нашем стенде, если остановить БД (`docker compose stop db`):

Запрос	Ответ	Почему
`/health/live`	200 `Healthy`	Процесс жив, зависимости не проверяются
`/health/ready`	503 `Unhealthy`	Проверка `db` не проходит
`/api/tasks`	500	Запрос к БД падает с исключением, в логе `Error`
`docker compose ps`	`Up … (healthy)`	HEALTHCHECK смотрит на live

Контейнер не перезапускается, и это правильно: рестарт не вернёт БД. Как только `db` снова запустится, `/health/ready` станет 200, а запросы к данным пойдут без вмешательства человека.

Главное, что нужно помнить: **Docker сам НЕ перезапускает `unhealthy` контейнеры.** Статус используют `depends_on: condition: service_healthy`, `docker compose up --wait` при деплое, мониторинг. Перезапуск по liveness и исключение из балансировки по readiness делают оркестраторы, например Kubernetes, у которого для этого свои проверки (probes).