Неделя 12. Лекция 6. Жизнь контейнера: логи, здоровье, рестарты, ресурсы

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

7. Логи

Куда пишет приложение. Контейнерное приложение пишет логи в **stdout/stderr**, дальше их забирает драйвер логирования Docker. По умолчанию это `json-file`: каждая строка вывода сохраняется в файл на хосте (у Docker Desktop — внутри его виртуальной машины) вместе с временем и потоком. Команда `docker logs` читает именно этот файл. Писать логи в файлы внутри контейнера не нужно: их никто не соберёт, а при удалении контейнера они пропадут.

Файл логов принадлежит контейнеру. Перезапуск его сохраняет, поэтому после крэша `docker logs` покажет и последние слова упавшего процесса, и старт нового. А `docker compose down` или пересоздание контейнера удаляют логи вместе с контейнером.

Ротация обязательна. Без неё `json-file` растёт бесконечно. Классическая история: сервис годами работает на сервере, в какой-то момент начинает писать ошибку на каждый запрос, и за выходные лог занимает весь диск. После этого падает уже не один сервис, а все, кому нужно место на диске, включая БД. В нашем `compose.yaml`:

```
        logging:
          driver: json-file
          options: { max-size: "10m", max-file: "3" }
```

Docker хранит не больше трёх файлов по 10 МБ и удаляет самый старый. Настройки применяются к новым контейнерам, поэтому после правки контейнер пересоздают.

Структурные логи. Вне Development сервис задач пишет логи в JSON (`AddJsonConsole()` в `Program.cs`), одна запись — одна строка. Вот что попадает в лог при `/api/chaos/crash` (разбито на строки для читаемости):

```
        {"EventId":0,"LogLevel":"Critical","Category":"DemoApi.Controllers.ChaosController",
         "Message":"Chaos: аварийное завершение процесса","State":{...}}
```

Такие записи легко фильтровать по полю (`LogLevel`, `Category`) и отправлять в системы сбора. Параметры сообщения становятся отдельными полями в `State`: `log.LogInformation("Задача {TaskId} создана", id)` даст поле `TaskId`, по которому потом ищут все события одной задачи. Метки времени в наших JSON-записях нет, её добавляет Docker: `docker logs -t`.

Уровни:

Уровень	Когда	Пример
`Information`	Обычные события	`Хранилище: PostgreSQL` при старте
`Warning`	Подозрительное, но работаем	Повторная попытка подключения
`Error`	Сбой одного запроса	Каждый запрос, упавший с 500, через `UseExceptionHandler`
`Critical`	Сбой всего сервиса	`Chaos: аварийное завершение процесса`

Корреляция. Ответ 500 содержит поле `traceId` в ProblemDetails, например `00-4bf92f3577b34da6a3ce929d0e0e4736-00f067aa0ba902b7-00`. Средняя часть — идентификатор трассировки запроса. Если пользователь прислал его в обращении, по нему находят в логах именно его запрос среди тысяч других. В нашем проекте JSON-логи пишутся без scopes, поэтому `traceId` в записи не попадает; включается это опцией `IncludeScopes = true` в `AddJsonConsole`, и тогда у каждой записи появляются `TraceId`, `SpanId` и путь запроса.

Сколько писать. Лог, в котором на каждый запрос приходится десяток строк `Information`, так же бесполезен, как пустой: нужная запись тонет, а ротация успевает удалить вчерашние события раньше, чем до них дойдут руки. Поэтому в `appsettings.json` сервиса задач для категорий `Microsoft.AspNetCore` и `Npgsql` стоит уровень `Warning`: служебные сообщения фреймворка и драйвера БД в лог не попадают. Уровень меняют без пересборки, через конфигурацию: переменная окружения `Logging__LogLevel__Microsoft.AspNetCore=Information` временно вернёт подробности для расследования, после чего её убирают.

Чего не писать в логи. Пароли, токены, строки подключения, номера карт, персональные данные пользователей. Логи читает больше людей, чем базу, они копируются в системы сбора и хранятся дольше, чем кажется. Строка `ConnectionStrings__Db` с паролем, записанная «для отладки», становится утечкой секрета.

Команды. `docker logs demo-api-1 --since 10m --tail 100 -f`: `--since` ограничивает по времени (удобно, когда известно время сбоя), `--tail` — по числу строк, `-f` продолжает выводить новые. В compose то же самое: `docker compose logs api --since 10m`. Смотреть логи всех контейнеров в браузере удобнее в **Dozzle**, он появится в лабе 7.

8. Ресурсы: память и CPU

```
        memswap_limit: 256M            # Docker Desktop: без этого лишняя память уходит в swap
        deploy:
          resources:
            limits: { cpus: "1.0", memory: 256M }
```

Лимиты реализованы механизмом ядра Linux **cgroups**: процессы контейнера объединены в группу, и ядро следит, чтобы группа не превысила выделенное. Лимит видно изнутри контейнера: `docker exec demo-api-1 cat /sys/fs/cgroup/memory.max` покажет `268435456`, то есть 256 МБ. Снаружи использование против лимита показывает `docker stats`.

Зачем лимиты. Без них один контейнер с утечкой займёт всю память сервера, и ядро начнёт убивать процессы уже по своему выбору, в том числе БД. Лимит превращает «упал сервер» в «перезапустился один сервис». `memswap_limit` равный лимиту памяти запрещает контейнеру уходить в swap: иначе вместо быстрого и понятного OOM вы получите медленную агонию, когда сервис вроде жив, но отвечает секундами.

Как .NET видит лимит. Среда выполнения .NET читает лимиты cgroups и подстраивается под них. Куча сборщика мусора (GC) по умолчанию ограничена примерно **75% лимита контейнера**, у нас это около 190 МБ. Отсюда два разных сценария нехватки памяти:

	Управляемая память (объекты .NET)	**Неуправляемая память (нативный код, `Marshal.AllocHGlobal`)**
Кто следит	GC	Только ядро
Что происходит	GC упирается в свой лимит, бросается `OutOfMemoryException`	Группа превышает 256 МБ, ядро посылает SIGKILL
Для пользователя	500 на запросах, которым не хватило памяти	Обрыв всех соединений
Для Docker	Контейнер жив	`oom`, `die` с кодом 137, рестарт
Где искать	Логи: `Error` с `OutOfMemoryException`	События Docker

Какой лимит поставить. Лимит подбирают по наблюдению, а не наугад: запускают сервис под типичной нагрузкой, смотрят в `docker stats`, сколько он занимает после прогрева, и добавляют запас на пики. Слишком тесный лимит даёт регулярные OOM на ровном месте, слишком свободный перестаёт защищать соседей. Если GC нужно дать другую долю памяти, это настраивается переменной окружения `DOTNET_GCHeapHardLimitPercent`, но в учебном проекте хватает значения по умолчанию.

Утечка памяти — память, которую программа занимает и больше не освобождает. В .NET это обычно не «забытый free», а живые ссылки: статическая коллекция, которая только растёт, кэш без ограничения размера, подписка на событие, которую никто не снимает. Признак в `docker stats` один и тот же: память растёт при нагрузке и не возвращается, когда нагрузка уходит. Растущая «пила» с падениями до базового уровня в момент рестартов — почти всегда утечка.

CPU-лимит не убивает, а троттлит. `cpus: "1.0"` означает: за каждые 100 мс контейнер может использовать не больше 100 мс процессорного времени. Израсходовал квоту раньше — ждёт начала следующего периода. Ошибок нет, память в порядке, в `docker stats` CPU около 100%, а ответы медленные. Увидеть троттлинг можно в `docker exec demo-api-1 cat /sys/fs/cgroup/cpu.stat`: растут `nr_throttled` и `throttled_usec`. Так что «сервис стал медленным» — это повод посмотреть не только в код, но и на CPU против лимита.

9. Разбор на нашем проекте: `/api/chaos`

Контроллер `ChaosController` включается настройкой `Chaos__Enabled` (в нашем `compose.yaml` она `true`) и воспроизводит типовые сбои. Для каждого стоит заранее понимать, что именно увидит инженер.

`GET /api/chaos/error` бросает `InvalidOperationException`. `UseExceptionHandler` ловит исключение, пишет его в лог с уровнем `Error` и возвращает клиенту 500 без подробностей:


```
        {"type":"https://tools.ietf.org/html/rfc9110#section-15.6.1","title":"An error occurred while processing your request.","status":500,"traceId":"00-…-…-00"}
```


В логе — запись `"LogLevel":"Error"` из категории `Microsoft.AspNetCore.Diagnostics.ExceptionHandlerMiddleware` с полем `Exception`, где лежит текст `Chaos: искусственная ошибка` и стек вызовов. Событий Docker нет, `RestartCount` не меняется. Большинство реальных ошибок выглядят именно так.

`POST /api/chaos/crash` отвечает 202, пишет `Critical` и через 200 мс вызывает `Environment.Exit(1)`. Задержка нужна, чтобы клиент успел получить ответ. Дальше:

```
container die … (exitCode=1, name=demo-api-1)
container start … (name=demo-api-1)
container health_status: healthy … (name=demo-api-1)
```

```
        docker ps --filter name=demo-api --format "table {{.Names}}\t{{.Status}}"
        # NAMES        STATUS
        # demo-api-1   Up 6 seconds (health: starting)
```

Сервис был недоступен несколько секунд, пока .NET запускался заново. Короткий аптайм при давно запущенном стенде — первый признак, что контейнер перезапускался.

`POST /api/chaos/leak?mb=80` занимает 80 МБ неуправляемой памяти и не освобождает её, ответ — `{"leakedMb":80}`, затем `160`. GC эту память не видит и не ограничивает. В `docker stats` колонка `MEM USAGE / LIMIT` растёт к `256MiB`, на третьем-четвёртом запросе ядро убивает процесс: соединение обрывается, в событиях `oom` → `die` (`exitCode=137`) → `start`. В логах перед падением нет ничего: SIGKILL не даёт приложению записать ни строчки. Если бы память занимали обычные массивы .NET, процесс остался бы жив, а запрос получил бы 500 с `OutOfMemoryException`.

`GET /api/chaos/slow?ms=3000` ждёт через `Task.Delay` и отвечает `{"delayedMs":3000}`. Ожидание асинхронное, поток не блокируется, поэтому healthcheck продолжает проходить, CPU не растёт, ошибок нет. Ни `docker ps`, ни логи, ни события эту проблему не покажут. Её видно только по времени ответа, то есть в метриках (тема 7).

> **Демонстрация для преподавателя.** Разделите экран на три терминала, как в лабе: `docker events` с фильтрами, `docker stats demo-api-1` и командный. Отправьте `leak?mb=80` четыре раза, проговаривая, что видно в каждом окне. После рестарта выполните `docker inspect … OOMKilled` и покажите `false`. Спросите аудиторию, где теперь доказательство OOM, и верните их к окну с событиями.

10. Инструменты диагностики

Команда	Что даёт
`docker compose ps -a`	Статус, health, аптайм, exit-код остановленных
`docker logs --since 10m --tail 100`	Последние слова перед падением
`docker events --since 10m --until 0s`	Хронология: `die exitCode`, `oom`, `health_status`
`docker inspect -f '{{json .State}}'`	Состояние, `ExitCode`, `OOMKilled`, `Health.Log`
`docker inspect -f '{{.RestartCount}}'`	Сколько раз контейнер перезапускался
`docker stats --no-stream`	Память и CPU сейчас, против лимита
`docker exec -it X sh`	Оболочка внутри: переменные, файлы, `/sys/fs/cgroup`
`docker top X`	Процессы контейнера, видимые с хоста
`docker cp X:/app/appsettings.json .`	Достать файл из контейнера, в том числе остановленного
Docker Desktop → Containers	Логи, inspect, exec, stats в UI
Dozzle (лаба 7)	Логи всех контейнеров в браузере

`docker exec` работает только с запущенным контейнером и под тем же пользователем, что и приложение (у нас непривилегированный `app`). Он полезен, чтобы проверить, что реально видит процесс: `docker exec demo-api-1 printenv Storage` покажет, какая настройка хранилища попала в контейнер на самом деле.

11. Алгоритм «контейнер упал»

```
flowchart TD
  A["Сервис недоступен или сбоит"] --> B["docker compose ps -a<br/>статус, аптайм, health"]
  B --> C["docker logs --since<br/>исключения, Critical"]
  C --> D["docker events --since<br/>exitCode, oom, kill"]
  D --> E["docker stats<br/>память и CPU против лимита"]
  E --> F["Что менялось:<br/>деплой, конфигурация, БД"]
  F --> G["Гипотеза и проверка"]
```

1. **Состояние.** `docker compose ps -a`. `Restarting` — цикл падений; `Up 40 seconds` на стенде, который работает с утра, — был рестарт; `unhealthy` — процесс жив, но не отвечает; `Exited (1)` — упал и не поднят (политика `no` или остановлен вручную).
2. **Логи.** `docker logs demo-api-1 --since 15m`. Ищите `Critical`, `Error`, `Unhandled exception`, сообщения старта. Пустота перед рестартом — важная улика: процесс не успел ничего сказать, значит, его убили снаружи.
3. **События.** `docker events --since 15m --until 0s --filter container=demo-api-1`. Код 1 или 134 — ошибка приложения, смотрите логи; 137 с `oom` — память; 137 с `kill` — не уложился в остановку; 0 — кто-то остановил штатно.
4. **Ресурсы.** `docker stats --no-stream`. Память у лимита и растёт — утечка или слишком маленький лимит; CPU упирается в 100% — троттлинг.
5. **Изменения.** Что деплоили последним, когда, кто менял конфигурацию (лекция 5). Совпадение времени сбоя с деплоем — самая сильная подсказка, и откат часто быстрее поиска причины.

Пример разбора. Пользователи жалуются, что сервис «иногда отваливается». Первая команда показывает `demo-api-1   Up 3 minutes (healthy)`, хотя стенд запущен с утра, и `RestartCount` равен 4. Значит, контейнер падает, а restart policy каждый раз его поднимает. В логах перед каждым новым стартом нет ни `Critical`, ни исключений: после последней будничной записи сразу идёт стартовое `Хранилище: PostgreSQL`. Приложение не успело ничего сказать, значит, его убили снаружи. События подтверждают: `oom`, затем `die` с кодом 137. В `docker stats` память после рестарта снова ползёт вверх. Вывод: утечка неуправляемой памяти, и вопрос для следующего шага — какой релиз её принёс. Обратите внимание, что ни одна команда по отдельности ответа не дала; его дала последовательность.

Чего не делать: сразу выполнять `docker compose down` и `up`. Это возвращает сервис, но удаляет контейнер вместе с логами и состоянием. Если сервис нужно поднять срочно, сначала сохраните улики: `docker logs demo-api-1 > crash.log`, вывод `docker inspect` и `docker events`.

> **Демонстрация для преподавателя.** Попросите кого-то из студентов незаметно выполнить один из сценариев лабы (`crash`, `leak` или `docker compose stop db`), пока вы отвернулись. Затем вслух пройдите алгоритм и определите сценарий, объясняя каждый вывод. Это хорошо показывает, что порядок команд важнее памяти на команды.

12. После инцидента: постмортем

Когда сервис восстановлен, работа не закончена. **Постмортем** — короткий документ о том, что произошло: хронология с временем, влияние на пользователей, причина, как обнаружили, что сделали и что изменим, чтобы не повторилось. Хронологию для него дают как раз те источники, которые мы разобрали: события, логи, записи о деплоях.

Постмортем пишут **без поиска виноватых**. Вопрос не «кто выложил плохой релиз», а «почему система позволила плохому релизу дойти до пользователей и почему мы узнали об этом так поздно». Если за ошибку наказывают, люди начинают скрывать ошибки, и следующая разборка будет опираться на неполные данные. Результат хорошего постмортема — конкретные действия: алерт на `RestartCount`, лимит памяти, тест, проверка в пайплайне. Полный цикл с постмортемом вы пройдёте в лабе 8.

Типичные ошибки

- **Ожидать, что Docker перезапустит `unhealthy`.** Не перезапустит, это делают оркестраторы.
- **Проверять БД в liveness:** при сбое БД перезапускаются все экземпляры сервиса, короткий сбой превращается в каскадный отказ.
- **Считать рестарт решением.** Растущий `RestartCount` — инцидент, а не норма.
- **Shell-форма `ENTRYPOINT`:** приложение не получает SIGTERM, каждая остановка заканчивается SIGKILL и кодом 137.
- **Логи в файлы внутри контейнера** или без ротации.
- **Секреты и персональные данные в логах.**

- **Нет лимитов памяти:** один контейнер кладёт весь хост.
- **Верить `OOMKilled=false` после рестарта:** поле уже сброшено.
- **`docker compose down/up` вместо разбора причины:** улики уничтожены.

Вопросы для самопроверки

1. Контейнер рестартится с exit 137. Две возможные причины и как их различить?
2. Почему `HEALTHCHECK` смотрит на `/health/live`, а не на `/health/ready`? Что случится, если проверять БД в liveness?
3. Чем `always` отличается от `unless-stopped`? Почему после `docker stop` контейнер не поднимается ни при одной политике?
4. Почему `docker inspect` после рестарта показывает `OOMKilled=false`? Где искать правду?
5. Почему `/api/chaos/leak` приводит к OOM kill, а массивы .NET того же размера — к `OutOfMemoryException`?
6. Контейнер при каждом деплое завершается с кодом 137 вместо 0. Какие две причины вы проверите первыми?
7. Сервис медленный, ошибок в логах нет, контейнер healthy. Что посмотрите в первую очередь?
8. Какие сбои из таблицы п. 1 не видны по `docker compose ps`?

Практика: лаба 6

Будете ломать сервис задач через `/api/chaos`: ошибка в запросе, падение процесса, утечка памяти до OOM. Остановите БД, сломаете healthcheck и по событиям, логам и exit-кодам определите, что произошло в каждом случае.

Работать будете в трёх терминалах: команды, `docker events` с фильтрами и `docker stats`. Для каждого сценария заранее сформулируйте, что ожидаете увидеть в каждом окне, и сверяйте с тем, что увидели на самом деле: расхождение ожидания и факта и есть то место, где понимание пока неполное. Таблица сценариев, которую вы заполните, пригодится в лабе 7, когда те же сбои нужно будет увидеть уже на дашборде и в алертах.