Неделя 13. Лекция 7. Мониторинг: метрики, дашборды, алерты

Часть 1 из 2.

После лекции: вы понимаете, зачем сервису мониторинг, какие метрики снимать с сервиса и контейнера, как работает Prometheus, умеете писать базовые запросы PromQL и алерты, которые будят только по делу.
Инструменты: Prometheus, Grafana, cAdvisor, Dozzle (все в контейнерах, ставить не нужно), браузер.

1. Зачем мониторинг

В лабе 6 вы узнавали о сбоях, потому что сами их устраивали. В жизни о проблеме должна сообщать система — **раньше пользователей**. И не только о падениях: в лабе 6 медленные ответы не были видны ни в `docker ps`, ни в логах. Контейнер `healthy`, в логах тишина, а пользователь ждёт ответа три секунды и уходит.

Мониторинг отвечает на вопросы:

- работает ли сервис прямо сейчас;
- насколько ему плохо: сколько ошибок, как медленно;
- что изменилось и когда: после какого деплоя стало хуже;
- хватит ли ресурсов.

Мониторинг и наблюдаемость — не синонимы. Мониторинг проверяет то, о чём вы заранее догадались спросить: «доля 5xx выше 5%?». Наблюдаемость (observability) — свойство системы, по внешним данным которой можно ответить и на вопрос, который вы заранее не придумали: «почему тормозит только смена статуса и только после обеда?». Для этого нужны не только правила, но и подробные данные с измерениями: какой метод, какой код ответа, какой контейнер. В курсе мы строим мониторинг, но метрики снимаем так, чтобы из них можно было выжать и второе.

2. Три сигнала наблюдаемости

Сигнал	Отвечает на вопрос	Инструмент в курсе
Метрики	*Что* происходит и насколько плохо? (числа во времени)	Prometheus + Grafana
Логи	*Почему* это произошло? (события с деталями)	`docker logs`, Dozzle
Трейсы	*Где* тормозит запрос в цепочке сервисов?	OpenTelemetry (за рамками курса)

Метрики сообщают о проблеме, логи объясняют её причину.

Метрики дёшевы: одно число на ряд раз в несколько секунд, сколько бы запросов ни пришло. Поэтому их хранят неделями, по ним рисуют графики и считают алерты. Логи дороже, зато в каждой записи есть детали: текст исключения, id задачи, стек вызовов. Трейс показывает путь одного запроса через несколько сервисов и БД с временем каждого шага; у нас сервис один, и трейсы ничего не добавят.

Типичный порядок работы в инциденте: алерт по метрике → дашборд сужает проблему до действия и времени → логи этого контейнера за это время объясняют причину.

На нашем сервисе это выглядит так. Вызов `/api/chaos/error` на дашборде даёт линию с кодом `500` у действия `Chaos/Error` — это всё, что умеет сказать метрика. В Dozzle у `demo-api-1` в то же время видна JSON-запись уровня Error с исключением `InvalidOperationException` и текстом «Chaos: искусственная ошибка». Метрика подсказала, где и когда искать, лог — что именно сломалось. Обратное тоже верно: по логам трудно понять масштаб, одна ошибка в минуту или тысяча.

3. Стек мониторинга

```
flowchart TB
  subgraph demo["проект demo"]
    API["DemoApi<br/>/metrics"]
    DB["PostgreSQL"]
  end
  subgraph mon["проект monitoring"]
    P["Prometheus<br/>:9090<br/>правила алертов"]
    G["Grafana<br/>:3000"]
    C["cAdvisor<br/>:8081"]
    DZ["Dozzle<br/>:8888"]
  end
  P -->|scrape host.docker.internal:8080| API
  P -->|scrape| C
  C -.->|cgroups| API
  C -.-> DB
  G -->|PromQL| P
  DZ -.->|docker.sock| API
```

Компонент	Роль
Сервис (`/metrics`)	Отдаёт свои метрики текстом: запросы, время ответа, память процесса
cAdvisor	Читает cgroups и отдаёт метрики **каждого контейнера**: CPU, память, время старта
Prometheus	Раз в N секунд **сам забирает** метрики (pull-модель), хранит их во времени, вычисляет правила алертов
Grafana	Строит дашборды по запросам к Prometheus
Dozzle	Показывает логи всех контейнеров в браузере

Мониторинг — **отдельный compose-проект**: деплой сервиса его не трогает. Сервис доступен Prometheus по адресу `host.docker.internal` (Docker Desktop). Настройки Prometheus, правила алертов и дашборды Grafana лежат в git — это «мониторинг как код».

Версии в `monitoring/compose.yaml` зафиксированы: `prom/prometheus:v3.14.0`, `grafana/grafana:13.2.1`, `ghcr.io/google/cadvisor:v0.60.5`, `amir20/dozzle:v11.0.1`. Тег `latest` в мониторинге так же опасен, как в приложении: после обновления может поменяться формат конфигурации или имена метрик, и дашборд молча опустеет.

4. Как Prometheus собирает метрики

Prometheus работает по **pull-модели**: сервис ничего никуда не отправляет, а только держит страницу `/metrics`. Prometheus сам ходит за ней раз в `scrape_interval` — это называется **scrape**. Наш `prometheus.yml`:

```yaml
global:
  scrape_interval: 5s        # для учебного стенда; в проде обычно 15–30s
  evaluation_interval: 5s

scrape_configs:
  - job_name: demo-api
    static_configs:
      - targets: ["host.docker.internal:8080"]
```

Каждый адрес в `targets` — **цель** (target). Ко всем метрикам цели Prometheus добавляет два лейбла: `job` (имя из `job_name`) и `instance` (адрес цели). Если бы сервис работал в трёх экземплярах, у одного `job="demo-api"` было бы три `instance`.

После каждого scrape Prometheus записывает служебную метрику **`up`**: 1 — страница получена, 0 — нет. На ней построен алерт `ApiDown`, и это самый честный признак «сервис недоступен»: он не зависит от кода сервиса.

Pull удобен тем, что Prometheus сам знает список целей и отличает «сервис молчит, потому что нет запросов» от «сервис недоступен». Недостаток — Prometheus должен дотянуться до сервиса по сети; поэтому в `compose.yaml` мониторинга есть `host.docker.internal`. Для коротких задач, которые завершаются раньше, чем их успеют опросить (ночной скрипт, задача в пайплайне), есть обходной путь — Pushgateway: задача отправляет туда метрики, а Prometheus забирает их оттуда. Для долгоживущего веб-сервиса он не нужен.

Если цель не ответила, Prometheus ставит `up` в 0, а её остальные ряды помечает устаревшими. На графиках это выглядит как разрыв линии, а не как ноль: Prometheus не знает, сколько было запросов, и честно ничего не рисует. Когда сервис вернётся, счётчики начнутся с нуля, и `rate` это переживёт (раздел 7).

Данные хранятся в локальной базе временных рядов (TSDB) в томе `prometheus-data`, срок хранения задан флагом `--storage.tsdb.retention.time=7d`. Флаг `--web.enable-lifecycle` разрешает перечитать конфигурацию и правила без перезапуска: `curl.exe -X POST http://localhost:9090/-/reload`.

5. Как сервис отдаёт метрики

Одна строка в `Program.cs` — `app.UseHttpMetrics()` из библиотеки prometheus-net — считает каждый HTTP-запрос. `app.MapMetrics()` публикует результат на `/metrics`. Порядок строк важен:

```
        app.UseHttpMetrics();        // снаружи обработчика ошибок: в метрики попадает итоговый код ответа (500)
        app.UseExceptionHandler();
```

Конвейер ASP.NET — цепочка middleware, где каждое звено оборачивает следующие (лекция 1). Метрики стоят снаружи обработчика ошибок и видят ответ уже после того, как исключение превратилось в 500. Поставьте их внутрь — и они увидят не тот ответ, который получил клиент, а исключение, пролетающее мимо.

Фрагмент реального `/metrics` нашего сервиса (строки укорочены):

```
# HELP http_requests_received_total Provides the count of HTTP requests that have been processed by the ASP.NET Core
pipeline.
# TYPE http_requests_received_total counter
http_requests_received_total{code="200",method="GET",controller="Tasks",action="List",endpoint="api/tasks"} 1
http_requests_received_total{code="404",method="GET",controller="Tasks",action="Get",endpoint="api/tasks/{id:int}"} 1
# TYPE http_request_duration_seconds histogram
http_request_duration_seconds_sum{code="200",method="GET",controller="Tasks",action="List",endpoint="api/tasks"}
0.0661079
http_request_duration_seconds_count{code="200",method="GET",controller="Tasks",action="List",endpoint="api/tasks"} 1
http_request_duration_seconds_bucket{...,action="List",endpoint="api/tasks",le="0.064"} 0
http_request_duration_seconds_bucket{...,action="List",endpoint="api/tasks",le="0.128"} 1
http_request_duration_seconds_bucket{...,action="List",endpoint="api/tasks",le="+Inf"} 1
# TYPE process_working_set_bytes gauge
process_working_set_bytes 8.1e+07
```

Это **текстовый формат экспозиции**: строки `# HELP` и `# TYPE` описывают метрику, дальше идут строки «имя{лейблы} значение». Время не пишется — его ставит Prometheus в момент scrape. Кроме HTTP-метрик библиотека отдаёт метрики процесса и .NET: `process_cpu_seconds_total`, `process_working_set_bytes`, `dotnet_collection_count_total` (сборки мусора) и другие.

Лейблы `controller` и `action` — это класс и метод вашего кода: `Tasks` / `ChangeStatus` — метод `TasksController.ChangeStatus`. По ним видно, какая именно операция тормозит или сыпет ошибками. Обратите внимание на `endpoint="api/tasks/{id:int}"`: в лейбл попадает **шаблон маршрута**, а не конкретный путь `/api/tasks/5`. Почему это важно, объясняет следующий раздел.

> Для ответов 500, которые сформировал обработчик исключений, лейбл `endpoint` пустой. **Группируйте по `controller` и `action`** — они заполнены всегда.

6. Временной ряд и кардинальность

Временной ряд — это имя метрики плюс конкретный набор значений лейблов. `http_requests_received_total{code="200",action="List",...}` и то же с `code="404"` — два разных ряда, у каждого своя история значений. Лейблы — это измерения, по которым потом можно фильтровать и группировать.

Кардинальность — число рядов. Она растёт как произведение числа значений лейблов: 5 кодов × 4 метода × 15 действий — в худшем случае уже сотни рядов у одного счётчика, а гистограмма умножает это ещё на число корзин (у нас 17). Prometheus держит свежие данные каждого ряда в памяти и строит по рядам индекс, поэтому его стоимость определяется числом рядов, а не числом запросов.

Отсюда правило: **не кладите в лейблы userId, email, id задачи, полный URL**. Каждый новый пользователь создаст новые ряды, через неделю их будут миллионы, и Prometheus упрётся в память. Для таких деталей есть логи. Если бы prometheus-net писал в `endpoint` путь `/api/tasks/5` вместо шаблона, каждая задача порождала бы свой набор рядов.

Оценить кардинальность своего сервиса можно двумя способами. Сам prometheus-net отдаёт метрику `prometheus_net_metric_timeseries` — сколько рядов он сейчас генерирует. В Prometheus тот же вопрос задаёт запрос `count({job="demo-api"})`. У учебного сервиса это сотни рядов, и так должно оставаться: число рядов

должно зависеть от размера кода (сколько действий и кодов ответа), а не от объёма данных (сколько задач и пользователей).

7. Типы метрик

Тип	Пример	Как читать
Counter — только растёт	`http_requests_received_total`	Через `rate()` / `increase()`
Gauge — текущее значение	`container_memory_working_set_bytes`	Как есть
Histogram — распределение по корзинам	`http_request_duration_seconds_bucket`	`histogram_quantile()`

Counter считает события с момента старта процесса и только растёт; при рестарте сбрасывается в ноль. Само значение «12 345 запросов» бесполезно: оно зависит от того, когда стартовал сервис. Полезна скорость роста. `rate(x[1m])` берёт точки ряда за последнюю минуту и считает прирост в секунду; сброс в ноль после рестарта функция распознаёт и не показывает отрицательный провал. Поэтому счётчику не нужно уметь уменьшаться: всё полезное извлекается из разности.

Gauge — значение «сейчас»: память, число запросов в обработке (`http_requests_in_progress`), температура. Его читают как есть, усредняют, сравнивают с порогом. `rate()` к gauge бессмыслен.

Histogram решает задачу «как распределено время ответа», не храня каждый запрос. Заданы границы корзин; у prometheus-net по умолчанию это 1 мс, 2 мс, 4 мс и так далее удвоением до 32,768 с, плюс `+Inf`. На каждый запрос библиотека увеличивает счётчики всех корзин, в которые он уложился: запрос за 70 мс попадёт в `le="0.128"`, `le="0.256"` и все следующие. Поэтому корзины **накопительные**: `le="0.128"` — сколько запросов выполнилось не дольше 128 мс. Рядом лежат `_sum` (суммарное время) и `_count` (число запросов); `_sum / _count` даёт среднее.

Свои метрики называют по соглашениям Prometheus: префикс приложения, базовые единицы (секунды, байты), суффикс `_total` у счётчиков. Отсюда `http_request_duration_seconds`, а не `..._ms`, и `demo_tasks_created_total` в дополнительном задании лабы. Бизнес-счётчик считает событие предметной области, а не HTTP-запрос, поэтому его показания не обязаны совпадать с числом ответов 201.

Summary — четвёртый тип: перцентили считает сам сервис и отдаёт готовые. Их нельзя честно сложить между экземплярами (среднее двух p95 — не p95), поэтому для HTTP-латентности обычно выбирают гистограмму.

8. Что смотреть: RED, USE, Golden Signals

- **RED** — для сервисов: **R**ate (запросы в секунду), **E**rrors (доля 5xx), **D**uration (p95/p99 времени ответа).
- **USE** — для ресурсов: **U**tilization (память/лимит), **S**aturation (троттлинг, очереди), **E**rrors.
- **Golden Signals** из книги Google SRE: latency, traffic, errors, saturation — по сути RED плюс насыщение.

Для сервиса задач RED — это панели дашборда, USE — метрики контейнера из cAdvisor. Сначала смотрят RED: это то, что чувствует пользователь. USE объясняет, почему RED испортился.

Перцентили, а не среднее. p95 = 1 с означает, что 95% запросов выполнились быстрее секунды. Пример: из 100 запросов 94 выполнились за 50 мс, а 6 — за 3 с. Среднее: (94 × 0,05 + 6 × 3) / 100 ≈ 0,23 с — выглядит прилично. p95 при этом 3 с: больше чем каждый двадцатый запрос ждёт три секунды. Среднее размывает хвост, а жалуются как раз из хвоста.

`histogram_quantile` считает перцентиль приблизительно. В примере выше 95-й запрос попадает в корзину от 2,048 до 4,096 с; функция считает, что запросы распределены внутри корзины равномерно, и выдаёт около 2,4 с. Точность ограничена границами корзин, но для ответа на вопрос «больше секунды или нет» её хватает.

9. PromQL: язык запросов

```
# RPS по кодам
sum by (code) (rate(http_requests_received_total{job="demo-api"}[1m]))

# Доля ошибок сервера
sum(rate(http_requests_received_total{job="demo-api", code=~"5.."}[1m]))
  / sum(rate(http_requests_received_total{job="demo-api"}[1m]))

# p95 времени ответа по действиям контроллеров
histogram_quantile(0.95,
  sum by (le, controller, action) (rate(http_request_duration_seconds_bucket{job="demo-api"}[1m])))

# Как часто клиенты натыкаются на бизнес-ошибки (409) — по действиям
sum by (action) (rate(http_requests_received_total{job="demo-api", code="409"}[5m]))

# Память контейнера в % от лимита
container_memory_working_set_bytes{name="demo-api-1"} / container_spec_memory_limit_bytes{name="demo-api-1"} * 100
```

Функция	Суть
`rate(x[1m])`	Скорость роста счётчика в секунду за последнюю минуту
`increase(x[5m])`	Прирост счётчика за окно
`sum by (label)`	Сложить ряды, сгруппировав по лейблу
`histogram_quantile(q, ...)`	Перцентиль из гистограммы
`changes(x[5m])`	Сколько раз менялось значение

Обозначения:

- `[1m]` — окно: значения за последнюю минуту;
- `{code=~"5.."}` — фильтр по регулярному выражению: все коды 5xx;
- `le` (less or equal) — верхняя граница корзины гистограммы; её сохраняют в `sum by (le, ...)`, иначе перцентиль не посчитать;
- `job` — лейбл, который Prometheus ставит по `job_name` из `prometheus.yml`.

Как читать p95 по действиям — изнутри наружу. `http_request_duration_seconds_bucket{job="demo-api"}` выбирает все ряды корзин сервиса. `rate(...[1m])` превращает накопительные счётчики в «сколько запросов в секунду попадало в каждую корзину за минуту». `sum by (le, controller, action)` схлопывает лишние измерения (`code`, `method`, `endpoint`), оставляя границу корзины и действие. `histogram_quantile(0.95, ...)` по корзинам каждого действия находит границу, ниже которой 95% запросов. Если выкинуть `le` из `by`, корзины сольются в одно число, и функции будет не из чего считать.

Окно `[1m]` или `[5m]`. Короткое окно быстро реагирует, но график дёрганый: одна медленная секунда заметно двигает p95. Длинное сглаживает, но сбой становится виден позже. На стенде scrape раз в 5 с, и в `[1m]` попадает 12 точек — этого достаточно. Практическое правило: окно не меньше четырёх интервалов scrape, иначе при одном пропущенном сборе `rate` может вернуть пустоту. В проде со scrape 15–30 с обычно берут `[5m]`.

`increase(x[1m])` — это `rate` × 60 с, поэтому он может вернуть дробное число: Prometheus экстраполирует прирост на всё окно.

Почему в доле ошибок только 5xx. Коды 4xx означают, что сервис правильно отказал неправильному запросу: пустое имя проекта — 400, недопустимый переход статуса — 409. Это нормальное поведение, и если включить 4xx в алерт, любой клиент с ошибкой в коде разбудит дежурного по сервису. Но и игнорировать 4xx нельзя: резкий рост 409 на `ChangeStatus` сразу после релиза фронтенда или самого сервиса — повод разобраться, кто изменил правила игры. Поэтому 4xx смотрят на дашборде в разбивке по действиям, а алерт ставят на 5xx — это ошибки, за которые отвечает сервис.

Деление двух `sum` в запросе доли ошибок работает потому, что после `sum` без `by` с каждой стороны остаётся по одному ряду без лейблов, и PromQL сопоставляет их друг с другом. Если оставить `by (action)` только в числителе, ряды не совпадут по лейблам, и результат окажется пустым. По умолчанию операнды бинарных операций сопоставляются по полному набору лейблов (ключевые слова `on` и `ignoring` это меняют) — ошибка с несовпадающими лейблами часто встречается при первых попытках писать запросы.

10. Метрики контейнера: cAdvisor

Сервис знает о себе не всё. Когда процесс упал, его `/metrics` молчит; лимит памяти контейнера процесс не видит; про контейнер PostgreSQL DemoApi не знает ничего. Эти пробелы закрывает cAdvisor: он читает

cgroups ядра и отдаёт метрики каждого контейнера с лейблом `name`.

Метрика	Тип	Что значит
`container_memory_working_set_bytes`	gauge	Память, которую ядро не может быстро освободить. Ближе всего к риску OOM
`container_spec_memory_limit_bytes`	gauge	Лимит из `compose.yaml` (у `api` — 256M); 0, если лимита нет
`container_cpu_usage_seconds_total`	counter	Секунды CPU; `rate` даёт занятые ядра: 0,5 — половина ядра
`container_start_time_seconds`	gauge	Время старта; меняется при каждом рестарте

Почему working set, а не `container_memory_usage_bytes`: usage включает файловый кэш, который ядро сбросит при нехватке памяти, и завышает картину. Отношение working set к лимиту — главный USE-показатель для памяти: при приближении к 100% сработает OOM-killer, и вы получите exit 137 из лекции 6.

Сравните два взгляда на одну память: `process_working_set_bytes` отдаёт сам сервис, `container_memory_working_set_bytes{name="demo-api-1"}` — cAdvisor снаружи. Числа близки, но различаются по смыслу. Первое пропадает вместе с процессом и ничего не знает о лимите. Второе продолжает писаться, пока контейнер существует, и стоит рядом с лимитом на одной панели. Поэтому алерты на ресурсы строят по cAdvisor, а метрики процесса оставляют для разбора, например чтобы сопоставить рост памяти со сборками мусора в `dotnet_collection_count_total`.

CPU читается иначе, чем память. `rate(container_cpu_usage_seconds_total[1m])` равное 1,0 при `cpus: "1.0"` в `compose.yaml` означает, что контейнер упёрся в свою квоту: ядро притормаживает процесс (троттлинг), и запросы начинают ждать процессора. Это насыщение из USE; снаружи оно проявится ростом p95, а причину покажет панель CPU.