Лекция 18: Мониторинг и логирование. Делаем приложение "прозрачным"

Введение: От "слепого полета" к полной наблюдаемости

Здравствуйте! Сегодня мы обсуждаем тему, которая отделяет "проект, который работает на ноутбуке" от "надежного сервиса, работающего в продакшене". Мы развернули наше приложение в облаке, оно доступно пользователям. Но наша работа на этом не закончена, а скорее, только начинается. Теперь мы вступаем в фазу **эксплуатации**, и наша главная задача — обеспечить стабильную и предсказуемую работу сервиса.

Без специальных инструментов наше приложение в продакшене — это "черный ящик". Мы не знаем, что происходит внутри. Если пользователи начинают жаловаться, что "сайт тормозит" или "не работает", мы не можем быстро понять причину. Мы летим вслепую.

Наблюдаемость (Observability) — это свойство системы, которое позволяет нам понимать её внутреннее состояние на основе внешних данных, которые она производит. Проще говоря, это наша способность задать системе любой вопрос о её поведении и получить на него ответ.

Аналогия: приборная панель самолета

- **Без Observability** — мы **пассажиры**. Мы просто надеемся, что все хорошо.
- **С Observability** — мы **пилоты в кабине**. Мы в реальном времени видим скорость, высоту, уровень топлива и сотни других параметров, и можем принять меры **до того**, как случится катастрофа.

Наблюдаемость строится на "трех столпах": **Логи**, **Метрики** и **Трассировка**. Сегодня мы подробно разберем каждый из них.

1. Столп №1: Логи — "Бортовой журнал" приложения

Логи отвечают на вопрос: **"ЧТО случилось?"**.
Это запись о **конкретном, дискретном событии**, произошедшем в определенный момент времени. Например: 2023-11-10T10:00:05Z INFO: User '123' successfully created task '456'.

Проблема console.log: В продакшене, где наше приложение запущено в 10 контейнерах на 3 разных серверах, console.log бесполезен. Логи пишутся в стандартный вывод контейнера и теряются при его перезапуске. Нам нужен способ **собирать все логи со всех контейнеров в одном месте**.

Решение: Централизованное логирование и структурированные логи

- **Правило №1: Логи должны быть написаны для машин.** Вместо простого текста мы пишем логи в формате **JSON**. Это позволяет легко их парсить, индексировать и искать. Для этого в Node.js используются библиотеки-логгеры, такие как **Pino** или **Winston**.
- **Правило №2: Логи нужно централизовать.** В Kubernetes специальный агент (например, **Fluentd**) автоматически собирает логи из stdout всех контейнеров и отправляет их в центральное хранилище.

Промышленный стандарт: ELK Stack

- **Elasticsearch:** Мощный поисковый движок для хранения и индексации огромных объемов логов.
- **Logstash (или Fluentd):** Инструмент для сбора, обработки и отправки логов в Elasticsearch.
- **Kibana:** Веб-интерфейс для поиска, фильтрации и визуализации логов. Это "Google для ваших логов".

2. Столп №2: Метрики — "Приборная панель"

Метрики отвечают на вопросы: **"КАК ЧАСТО?"** и **"КАК БЫСТРО?"**.
Это **числовое** измерение, снятое в определенный момент времени (например, http_requests_total = 5214). В отличие от логов, метрики легко агрегировать и на их основе строить графики.

Промышленный стандарт: Prometheus + Grafana

- **Prometheus:** Это база данных временных рядов (Time Series Database). Он работает по **Pull-модели**: периодически сам "опрашивает" наши приложения по специальному эндпоинту (/metrics) и забирает текущее состояние их метрик.
- **Grafana:** Это инструмент для визуализации. Она подключается к Prometheus и строит красивые, интерактивные дашборды, на которых мы можем видеть "пульс" нашего приложения в реальном времени.

"Золотые сигналы" Google SRE (4 метрики, которые нужно отслеживать в любом сервисе):

1. **Latency (Задержка):** Как быстро сервис отвечает на запросы? (Обычно измеряется 95-м и 99-м перцентилем).
2. **Traffic (Трафик):** Какая нагрузка на сервис? (Запросы в секунду, RPS).
3. **Errors (Ошибки):** Как часто сервис отвечает с ошибкой? (Доля 5xx ответов).
4. **Saturation (Насыщение):** Насколько "заполнен" сервис? Насколько он близок к своему пределу? (Загрузка CPU, использование памяти).

Практика: С помощью библиотеки prom-client в Node.js мы можем легко создать эндпоинт /metrics и "выставить" наружу как стандартные метрики (использование CPU/памяти), так и кастомные (количество HTTP-запросов, время ответа).

3. Столп №3: Трассировка — "Черный ящик"

Трассировка отвечает на вопрос: **"ГДЕ именно это случилось?"**.
Эта технология критически важна в мире **микросервисов**. Представьте, что один запрос от пользователя порождает цепочку из 10 вызовов между вашими внутренними сервисами. Если запрос "тормозит", как понять, какой именно сервис является "бутылочным горлышком"?

Распределенная трассировка решает эту проблему.

- **Принцип:** Когда запрос впервые попадает в систему, ему присваивается уникальный **Trace ID**. Этот ID, как эстафетная палочка, передается в заголовках **каждого** последующего вызова между сервисами.

- Каждый сервис записывает информацию о своей части работы (например, "запрос к БД занял 50мс") в виде **"спана"**, привязанного к этому Trace ID.
- Специальная система (например, **Jaeger** или **Zipkin**) собирает все спаны с одним Trace ID и строит **"водопадную" диаграмму (flame graph)**, на которой наглядно виден весь путь запроса и время, потраченное на каждом этапе.

Стандартом для "инструментации" кода (добавления кода для сбора трейсов) сегодня является **OpenTelemetry**.

4. Алертинг: Когда система "кричит" о помощи

Иметь красивые дашборды — это хорошо, но инженеры не могут смотреть на них 24/7. Мониторинг без **оповещений (алертов)** — это как пожарная сигнализация без сирены.

Алертинг — это процесс автоматического уведомления команды (в Slack, Telegram, PagerDuty), когда какая-либо метрика выходит за пределы нормы.

- **Инструмент:** В стеке Prometheus за это отвечает **Alertmanager**.
- **Процесс:** В Prometheus настраиваются **правила алертинга** на языке PromQL (например, alert if (error_rate > 5%) for 5 minutes). Когда правило срабатывает, Prometheus отправляет алерт в Alertmanager, а тот уже решает, как и куда отправить уведомление.

Главное правило хорошего алертинга: он должен быть основан на бизнес-метриках и "золотых сигналах" (влияющих на пользователя), а не на утилизации CPU. Система должна "кричать" только тогда, когда ей действительно больно.

Заключение

Сегодня мы научились делать наши приложения "прозрачными". Мы поняли, что **наблюдаемость** — это не одна технология, а комбинация трех подходов, которые дополняют друг друга:

- **Метрики** показывают общую картину и помогают выявить **аномалии**.
- **Алерты** сообщают нам об этих аномалиях.
- **Логи** помогают понять, **что именно** произошло в момент аномалии.
- **Трассировка** помогает понять, **где именно** в сложной системе это произошло.

На финальной лекции мы соберем все знания, полученные в этом семестре, в единую картину и проследим полный путь изменения в коде: от git push до работающего, отмасштабированного и, что самое главное, **наблюдаемого** сервиса в Kubernetes.