Лабораторная работа №10

Observability

Оглавление

Теоретическая часть

Observability

Observability (наблюдаемость) — это характеристика системы, которая определяет, насколько хорошо можно понять её текущее внутреннее состояние, опираясь исключительно на данные, поступающие извне: метрики, логи, трассировки. Чем выше наблюдаемость, тем проще анализировать поведение системы, выявлять и устранять проблемы без необходимости добавления дополнительного инструментирования или доступа к внутренним компонентам.

В отличие от мониторинга, который представляет собой сбор и анализ заранее определённых метрик, observability (наблюдаемость) охватывает более широкий спектр данных, включая не только метрики, но также логи и трассировки. Наблюдаемость позволяет получить целостное представление о внутреннем состоянии системы, основываясь исключительно на внешних сигналах.

Метрики являются одной из ключевых составляющих наблюдаемости. Они представляют собой числовые значения, измеряемые на протяжении времени и описывающие характеристики работы системы. Метрики позволяют отслеживать такие параметры, как количество запросов, время отклика, использование ресурсов и частоту ошибок. Благодаря агрегируемой и сжатой форме метрики легко хранить и обрабатывать, что делает их особенно удобными для построения дашбордов, генерации алертов и проведения анализа тенденций. Основное преимущество метрик — низкое потребление ресурсов и высокая скорость обработки, что позволяет собирать данные в реальном времени и мгновенно реагировать на отклонения от нормы.

Сбор метрик

Сбор метрик в системах мониторинга может осуществляться двумя основными способами — через pull или push модель. В pull-модели инициатором сбора выступает сама система мониторинга: она регулярно опрашивает приложения и сервисы по заданному адресу, получая от них актуальные данные. Это даёт возможность централизованно управлять частотой опроса и проще контролировать доступ. Например, Prometheus по умолчанию работает именно по такой схеме, запрашивая метрики с http://localhost:8000/metrics или аналогичных endpoint'ов.

Push-модель, наоборот, предполагает, что сами приложения или их экспортёры отправляют метрики в отдельный компонент — push-шлюз. Такой подход подходит для краткоживущих сервисов или тех, что находятся за NAT и не могут быть опрошены напрямую. Пример: завершив работу, приложение отправляет накопленные метрики в push-gateway, откуда их затем может забрать Prometheus.

Обе модели находят применение в зависимости от структуры системы и её ограничений. Так, Prometheus использует pull, а такие решения как Zabbix и Graphite традиционно опираются на push-модель.

Методика выбора метрик

При выборе метрик важно исходить из цели мониторинга: что именно должно быть видно и понятно из наблюдаемых данных. На практике метрики условно делятся на бизнес-

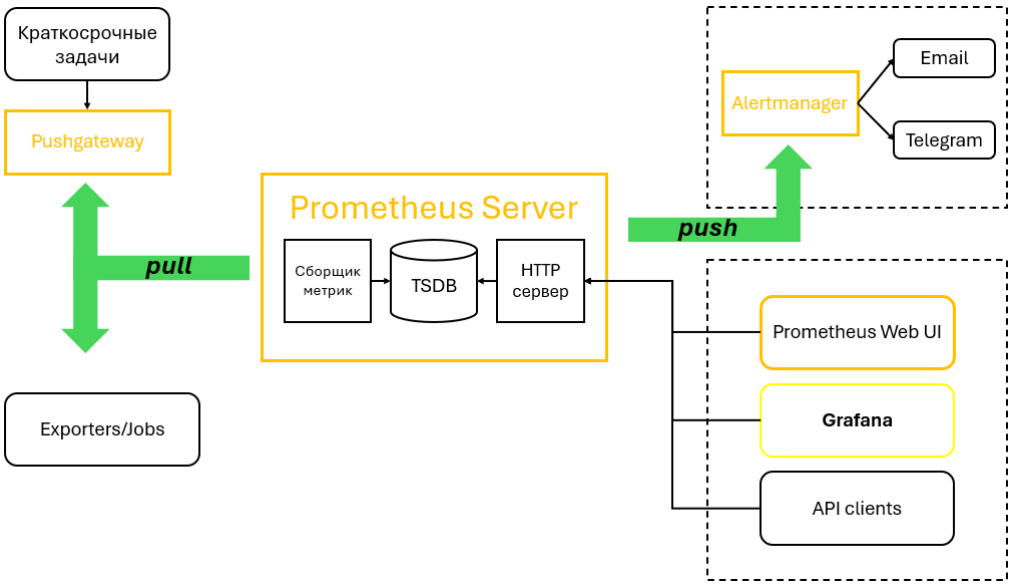
ориентированные и технические. Первые помогают оценивать, как система выглядит со стороны пользователя или заказчика — сюда относят, например, скорость ответа, частоту ошибок и стабильность работы. Эти показатели позволяют отслеживать качество сервиса и соответствие установленным нормативам. Вторые — технические — отражают внутренние характеристики компонентов: загрузку CPU, использование памяти, состояние очередей, количество потоков и другие.

Чтобы не запутаться, рекомендуется использовать структурированные подходы, такие как методики RED (Rate, Errors, Duration) — особенно полезна для HTTP-сервисов — и USE (Utilization, Saturation, Errors), применимая для анализа состояния ресурсов. Эти схемы помогают системно подойти к выбору и категоризации метрик. Например, по RED можно отследить количество запросов в секунду, долю сбоев и время отклика, а по USE — насколько интенсивно используется CPU, как часто он перегружен и сколько ошибок возникает в процессе. Методика выбора метрик должна опираться на реальные потребности команды и специфические риски системы: избыточный сбор неинформативных метрик создаёт нагрузку и затрудняет анализ, тогда как продуманный минимум покрывает все критические зоны.

Prometheus

Prometheus — это система сбора и хранения метрик. Его задача — регулярно опрашивать приложения и сохранять полученные данные. Формат хранения данных в Prometheus основан на **временных рядах** — каждый уникальный набор имя метрики + комбинация тегов образует отдельный временной ряд. При каждом опросе Prometheus сохраняет новое значение с текущей временной меткой. Таким образом, можно построить графики изменения метрик во времени, задать условия оповещений или применять агрегирующие функции.

Архитектура Prometheus представлена ниже.



Сервер Prometheus — основной компонент системы, который выполняет три задачи:

- **Сбор метрик** (scraping) по HTTP(S)-адресам.

- **Хранение данных** во встроенной time-series базе (TSDB).
- **Обработка запросов** на языке PromQL (Prometheus Query Language).

Рассмотрим компоненты подробнее:

- **TSDB (Time Series Database)** — встроенное хранилище временных рядов, оптимизированное для быстрой записи и считывания метрик. Вся база локальна — Prometheus хранит данные на диске в виде отдельных блоков, агрегированных по времени. Каждый временной ряд определяется уникальным набором меток (labels).
- **Экспортеры (Exporters)** — утилиты и библиотеки, которые преобразуют внутренние метрики различных систем (например, Linux, PostgreSQL, Nginx) в формат, понятный Prometheus. Самый распространённый — Node Exporter.
- **Push Gateway** — дополнительный компонент, позволяющий временным или batch-приложениям передавать метрики в Prometheus. Он необходим в тех случаях, когда невозможно реализовать pull-модель. Prometheus опрашивает Push Gateway как обычный endpoint.
- **Alertmanager** — компонент, отвечающий за обработку предупреждений, генерируемых на основе правил, заданных в Prometheus. Он может отправлять уведомления по email, в Slack, Telegram и другие системы оповещения, а также управлять группировкой, подавлением и маршрутизацией алертов.
- **Веб-интерфейс Prometheus** — встроенный UI, позволяющий вручную выполнять запросы на PromQL, просматривать состояние targets (целей сбора), проверять активные алерты и видеть метаданные.

В первую очередь Prometheus предназначен для мониторинга web приложений. Каждое из них должно предоставлять специальные страницы с метриками в текстовом формате. Эти страницы доступны по HTTP, и Prometheus опрашивает их с заданной периодичностью (например, раз в 60 секунд), считывая значения и сохраняя их в базе данных временных рядов (time series database).

В контексте Prometheus метрика — это числовое значение, которому сопоставлено имя и, возможно, набор тегов (labels), представляющих дополнительные атрибуты. Например, метрика http_requests_total (получить число HTTP запросов) может сопровождаться тегами method="GET" и status="200".
Например:

```
http_requests_total{status="200",method="GET"} 15423.
```

Сами значения — это числа с плавающей запятой, отражающие текущее состояние или накопленную статистику.

В Prometheus существует несколько типов метрик.

- **Counter** — счётчик, который может только расти, например, количество обработанных запросов. Если приложение перезапускается и значение счётчика сбрасывается, Prometheus может определить это и корректно интерпретировать.
- **Gauge** — переменная метрика, которая может как увеличиваться, так и уменьшаться. Примеры: использование памяти, число активных потоков.

- **Histogram** используется для подсчёта количества событий, попадающих в заданные интервалы (бакеты), например, сколько HTTP-запросов было обслужено за менее чем 50 мс, от 50 до 100 мс и так далее.
- **Summary** — это агрегирующая метрика, которая позволяет получить квантильные значения (например, 95-й процентиль), но в отличие от гистограммы, эти данные вычисляются внутри самого приложения, а не Prometheus. Это делает Summary быстрым, но проблематичным при сборе с нескольких реплик: квантиль, рассчитанный на одной ноде, не может быть корректно объединён с другим.

При опросе приложений Prometheus использует конфигурационный файл, где указаны адреса целей (targets), частота опроса и дополнительные метки, которые он добавляет к каждой метрике. Это позволяет не дублировать в коде приложения имя сервиса или его окружение (stage=prod), а задать это снаружи. Настройка частоты опроса — важный параметр, поскольку слишком частый сбор данных создает нагрузку на Prometheus, приложения и базу данных, а слишком редкий — снижает актуальность информации. Часто применяют смешанные схемы: один Prometheus собирает метрики редко для долгосрочного хранения, другой — чаще для быстрого реагирования.

С точки зрения безопасности следует учитывать, что доступ к метрикам обычно открыт только внутри внутренней сети, поэтому их нельзя рассматривать как конфиденциальные данные. В самих метриках не рекомендуется размещать чувствительную информацию. При необходимости доступ к страницам с метриками можно дополнительно ограничить внешними средствами — например, с помощью файрвола, прокси-сервера или механизмов аутентификации.

PromQL

PromQL (Prometheus Query Language) — это мощный язык запросов, предназначенный для выборки, агрегации и анализа временных рядов, собранных Prometheus. Он позволяет формировать как простые запросы, например, для отображения текущего значения метрики, так и сложные аналитические выражения с агрегацией, фильтрацией, математическими операциями и временными функциями.
Например, запрос **up** показывает состояние всех экспортёров (endpoint'ов), которые отслеживаются Prometheus. Если значение равно 1 — целевой endpoint доступен и успешно отдаёт метрики. Если 0 — Prometheus не может получить от него данные.

```
up
```

Element	Value
up{instance="localhost:8080",job="go_app"}	1
up{instance="localhost:9090",job="prometheus"}	1
up{instance="localhost:9100",job="node"}	1
up{instance="localhost:9187",job="postgres_exporter"}	1

Метки

Метки (labels) в Prometheus — это ключевые параметры, которые позволяют описывать и классифицировать метрики. Каждая метрика может иметь набор пар «ключ-значение», добавляющих контекст к данным и позволяющих гибко фильтровать, группировать и агрегировать показатели.

Например, метрика http_requests_total может содержать метки method="GET", handler="/api/user", status="200", что позволяет анализировать количество запросов по конкретным HTTP-методам, маршрутам или статусам ответов.

Использование меток делает данные в Prometheus более структурированными и информативными, позволяя создавать точечные запросы и отчёты, адаптированные под нужды конкретного приложения или компонента.

В PromQL фильтрация по меткам осуществляется в фигурных скобках, например:

```
http_requests_total{method="POST", status="500"}
```

Сопоставители

Сопоставители (matchers) в Prometheus — это операторы, используемые для фильтрации метрик по меткам (labels) в запросах PromQL. Они позволяют точно или по шаблону выбирать необходимые временные ряды, что делает запросы гибкими и точными.

Основные типы сопоставителей:

- = — точное совпадение значения метки.
  Например,

```
http_requests_total{method="GET"}
```

  выберет метрики, где method ровно "GET".
- != — значение метки не равно указанному.
  Например,

```
http_requests_total{status!="200"}
```

  выберет все метрики, где статус не равен "200".
- =~ — совпадение с регулярным выражением
  Например,

```
http_requests_total{status=~"5.."}
```

  выберет метрики с кодами ошибок HTTP 5xx.
- !~ — отрицание регулярного выражения.
  Например,

```
http_requests_total{method!~"GET|POST"}
```

  выберет метрики, где метод не равен ни "GET", ни "POST".

Мгновенный вектор

Мгновенный вектор (instant vector) в Prometheus — это набор временных рядов с их значениями на конкретный момент времени. Каждый элемент мгновенного вектора представляет собой одну метрику с определённым набором меток и её значение именно в тот момент, когда выполняется запрос.

Например, запрос *up* возвращает мгновенный вектор, состоящий из всех таргетов (экспортеров), которые Prometheus мониторит, с их текущим состоянием (1 — доступен, 0 — недоступен) на момент запроса.

Мгновенный вектор удобен для получения текущего состояния системы, отображения данных на дашбордах и формирования алертов, которые реагируют на значения метрик «здесь и сейчас».

Вектор диапазона

Вектор диапазона (range vector) в Prometheus — это набор временных рядов с их значениями за определённый временной интервал. Каждый элемент такого вектора содержит серию значений метрики, а не только одно мгновенное значение. Это позволяет анализировать изменения метрик во времени.

В PromQL вектор диапазона задаётся с помощью квадратных скобок с указанием длительности интервала, например:

```
http_requests_total[5m]
```

означает «все значения метрики http_requests_total за последние 5 минут».

Векторы диапазона широко используются в функциях для анализа трендов и динамики:

rate() — рассчитывает среднюю скорость изменения счётчика за интервал, например:

```
rate(http_requests_total[5m])
```

показывает среднее количество запросов в секунду за последние 5 минут.

increase() — вычисляет абсолютное увеличение счётчика за интервал:

```
increase(http_requests_total[10m])
```

delta() — измеряет разницу между первым и последним значением метрики за интервал (может использоваться для gauge):

```
delta(cpu_temperature_celsius[1h])
```

Смещение

Смещение (offset) в PromQL — это механизм, позволяющий сдвигать временной диапазон запроса назад во времени. Это полезно, когда нужно сравнить текущее состояние метрик с их значениями в прошлом или исключить влияние последних данных, которые могут быть неполными или некорректными.

Синтаксис смещения заключается в добавлении ключевого слова offset с указанием временного промежутка:

```
metric_name offset 1h
```

Этот запрос вернёт значение метрики metric_name, взятое ровно час назад от текущего времени.

Примеры использования:

- Сравнить нагрузку на CPU сейчас и час назад:

```
cpu_usage
cpu_usage offset 1h
```

Исключить последние 5 минут из расчёта, чтобы избежать искажений из-за задержек сбора:

```
rate(http_requests_total[5m] offset 5m)
```

Операторы

В PromQL поддерживаются стандартные арифметические и логические операторы, которые применяются к метрикам и позволяют выполнять над ними базовые вычисления и фильтрацию.

Арифметические операторы (+, -, *, /, % , ^) позволяют проводить математические операции над значениями метрик. Например:

```
http_requests_total / uptime_seconds_total
```

Этот запрос покажет среднее количество запросов в секунду.

Если в выражении участвует скаляр (например, число), он применяется ко всем значениям векторной метрики:

```
cpu_usage * 100
```

Преобразует использование CPU из долей в проценты.

Логические операторы (and, or, unless) работают с наборами меток и позволяют сравнивать и объединять временные ряды:

- and: оставляет только те ряды, которые присутствуют в обоих выражениях.
- or: объединяет ряды из двух выражений.
- unless: оставляет ряды из первого выражения, которых нет во втором.

```
up == 1 and http_requests_total
```

Оставит только те ряды, где сервис работает (up == 1) и есть данные о запросах.

Эти операторы дают возможность комбинировать и анализировать метрики более гибко, что особенно полезно при построении сложных дашбордов или оповещений.

В PromQL существует ряд **агрегирующих операторов**, которые позволяют обрабатывать множество временных рядов и сводить их к обобщённым значениям. Они используются для вычисления сумм, средних, минимальных и максимальных значений, а также других статистик по метрикам.

Основные агрегирующие операторы:

- **sum** — сумма значений всех временных рядов.
- **count** — количество временных рядов.
- **avg** — среднее значение.
- **min** — минимальное значение среди рядов.
- **max** — максимальное значение.
- **stddev** — стандартное отклонение.
- **stdvar** — дисперсия.
- **quantile(φ, ...)** — φ-квантиль (например, 0.95).

Группировка

В PromQL группировка используется для агрегирования значений метрик по определённым меткам. Это позволяет, например, посчитать общее значение по всем экземплярам или, наоборот, разбить данные по категориям. Основные агрегирующие функции применяются вместе с операторами by или without.

- by (...) — указывает, по каким меткам выполнять группировку (то есть сохранить их при агрегации).
- without (...) — наоборот, указывает, какие метки нужно убрать, чтобы сгруппировать все ряды без их учёта.

Просуммирует количество запросов по каждому job:

```
sum(http_requests_total) by (job)
```

Посчитает среднее использование CPU, исключив различие по instance — то есть среднее по всем экземплярам:

```
avg(cpu_usage) without (instance)
```

Агрегация особенно полезна при анализе поведения всей системы в целом или определённых её компонентов, а также при построении оповещений и дашбордов.

Оценка квантилей

Функция histogram_quantile() в PromQL используется для оценки квантилей (например, 0.5 для медианы, 0.95 для 95-го перцентиля) по данным, собранным в виде **гистограммы**. Гистограмма в Prometheus представлена как набор метрик вида:

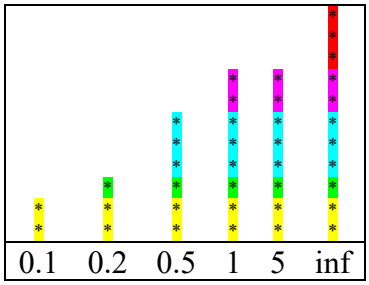
```
<metric_name>_bucket{le="0.1"} 18
<metric_name>_bucket{le="0.3"} 66
<metric_name>_bucket{le="0.5"} 107
...
<metric_name>_count 235
<metric_name>_sum   102.4
```

Каждая *_bucket метрика указывает, сколько значений попало **включительно** в этот "ведёрный" предел le.

Рассмотрим более подробно. Накопленные значения распределяются между интервалами – корзинами (bucket). Напомним, что в гистограмме значения накапливаются.

В следующем примере значения:
[0.01, 0.09, 0.12, 0.23, 0.3, 0.45, 0.75, 0.77, 7.1, 10, 35]

Были распределены следующим образом:



Все что превышает значения максимальной корзины, попадет в интервал (X; +inf), где X значение последнего интервала.

Иногда бывает полезно оценить **произвольную квантиль** (например, 50% или 95%) — это приближённое значение, ниже которого находится заданный процент всех наблюдений. Такой подход используется, в функции histogram_quantile() в Prometheus.

Рассмотрим подробно алгоритм ее работы.
Пусть:

q — квантиль (от 0 до 1)
buckets — список пар (u_i, c_i):

- u_i — верхняя граница бакета
- c_i — накопленное (cumulative) число значений до u_i

Алгоритм:

1. Если q < 0, вернуть -inf; если q > 1, вернуть +inf
2. total = c_n — последнее накопленное значение (в бакете с u_n = +inf)
3. Если total == 0, вернуть NaN
4. Вычисляем: rank = q * total
5. Находим первый индекс b, для которого: c_b >= rank
6. Если u_b == +inf, вернуть: u_{b - 1}
7. Определяем:
   u_start = u_{b - 1} (если b > 0, иначе 0.0)
   u_end = u_b

   c_start = c_{b - 1} (если b > 0, иначе 0.0)
   c_end = c_b

   count_in_bucket = c_end - c_start
   rank_in_bucket = rank - c_start
8. Если count_in_bucket == 0, вернуть u_end
9. Иначе интерполируем:
   value = u_start + (u_end - u_start) * (rank_in_bucket / count_in_bucket)

10. Вернуть value

Рассмотрим пример.
Пусть:

```
q = 0.95
buckets = {
    0.1: 18,
    0.3: 66,
    0.5: 107,
    1.0: 232,
    2.0: 232,
    5.0: 232,
    float('inf'): 235
}
```

Тогда:

1. Проверка границ квантиля: q = 0.95 → OK
2. total = c_n = 235
3. total ≠ 0 → OK
4. Вычисляем: rank = q * total = 0.95 * 235 = 223.25
5. Ищем первый индекс b, где c_b >= 223.25; 232 ← подходит, т.к. 232 ≥ 223.25. Тогда промежуток [0.5, 1.0]
6. Проверка: u_b = 1.0 ≠ +inf → продолжаем
7. Вычисления по бакету
   u_start = u_{b-1} = u_2 = 0.5
   u_end = u_b = u_3 = 1.0

   c_start = c_{b-1} = c_2 = 107
   c_end = c_b = c_3 = 232

   count_in_bucket = 232 - 107 = 125
   rank_in_bucket = 223.25 - 107 = 116.25

8. Проверка: count_in_bucket = 125 ≠ 0 → OK
9. Интерполяция:
   value = u_start + (u_end - u_start) * (rank_in_bucket / count_in_bucket)
   value = 0.5 + (1.0 - 0.5) * (116.25 / 125) = 0.5 + 0.5 * 0.93 = 0.965
10. value = 0.965

Особенность вычисления перцентиля заключается в том, что если 95 перцентиль попадает в блок inf, то вернется максимальное значение предыдущего блока. Т.е. для такого распределения

```
buckets = {
    0.1: 18,
    0.3: 66,
    0.5: 107,
    1.0: 232,
```

```
    2.0: 232,
    5.0: 232,
    float('inf'): 235
}
```

95 перцентиль равен 5, а не inf.

При выборе размера интервалов обычно руководствуются подходом, когда самое первое значение – «идеальный» вариант работы, далее идут допустимый и удовлетворительный. Самое последнее значение – недопустимое.

Например, если измеряется время отклика на запрос, то можно построить гистограмму с таким распределением:

```
buckets = {
    0.1    # Идеально
    1.0    # Допустимо
    5.0    # Еще удовлетворительно, но подозрительно
    inf    # Плохо, неудовлетворительно
}
```

Практическая часть

Образовательная часть

Часть 1. Prometheus и Grafana

1. Установите программы Prometheus и Grafana. Для этого выполните следующий скрипт

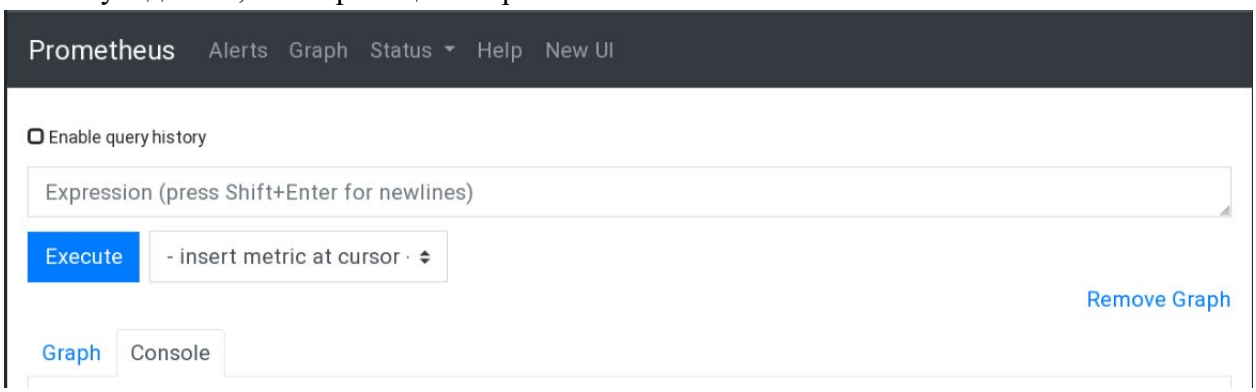
```
# Установка Grafana
echo "deb [trusted=yes] https://apt.grafana.com stable main" | sudo
tee -a /etc/apt/sources.list.d/grafana.list

sudo apt-get update
sudo apt-get install grafana
sudo apt-get install grafana-enterprise

# Установка Prometheus
sudo apt-get install Prometheus

# Проверка работы сервисов
sudo systemctl status grafana
sudo systemctl status prometheus
```

2. Добавьте в Prometheus сбор метрик с приложения. Для этого добавьте в конфигурационный файл /etc/prometheus/prometheus.yml следующие строки.

```
- job_name: ‘go_app’
  scrape_interval: 1s
```

```
  static_configs:
    - targets: [‘loclahost:8080’]
```

3. Перезагрузите Prometheus. Перейдите в браузере на страницу localhost:9090 и убедитесь, что страница отображается.



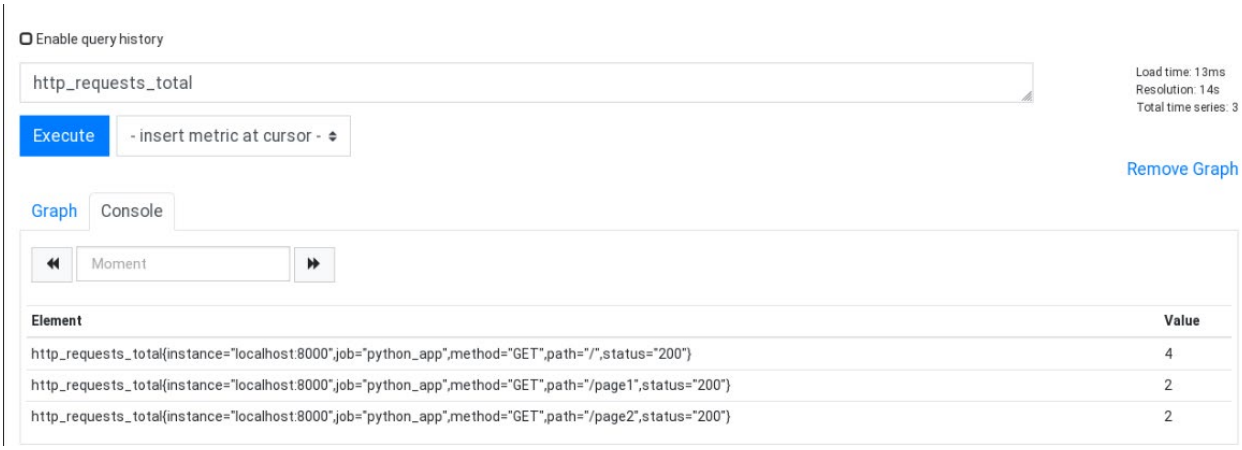
4. Изучите и запустите программу на Go. Проверьте, что на странице localhost:8000 отображаются метрики.
   Для запуска приложения воспользуйтесь командами:

```
go mod init main
go mod tide

go run main.go
```

5. Используя утилиту curl сделайте из соседней вкладки терминала несколько запросов к запущенному серверу по разным адресам страниц. Например:

```
curl http://localhost:8080
curl http://localhost:8080/page1
curl http://localhost:8080/page2
```

6. Используя командную строку Prometheus посчитайте количество запросов к серверу. Для этого запросите значение созданной в приложении метрики http_requests_total

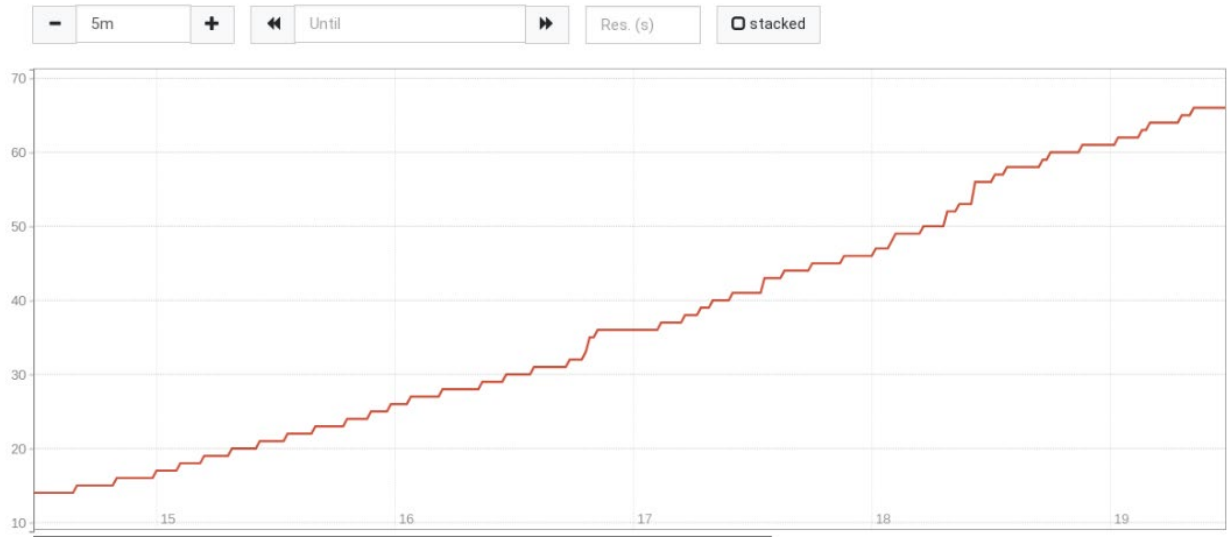


7. Запустите в отдельном терминале скрипт, который случайным образом обращается к странице.

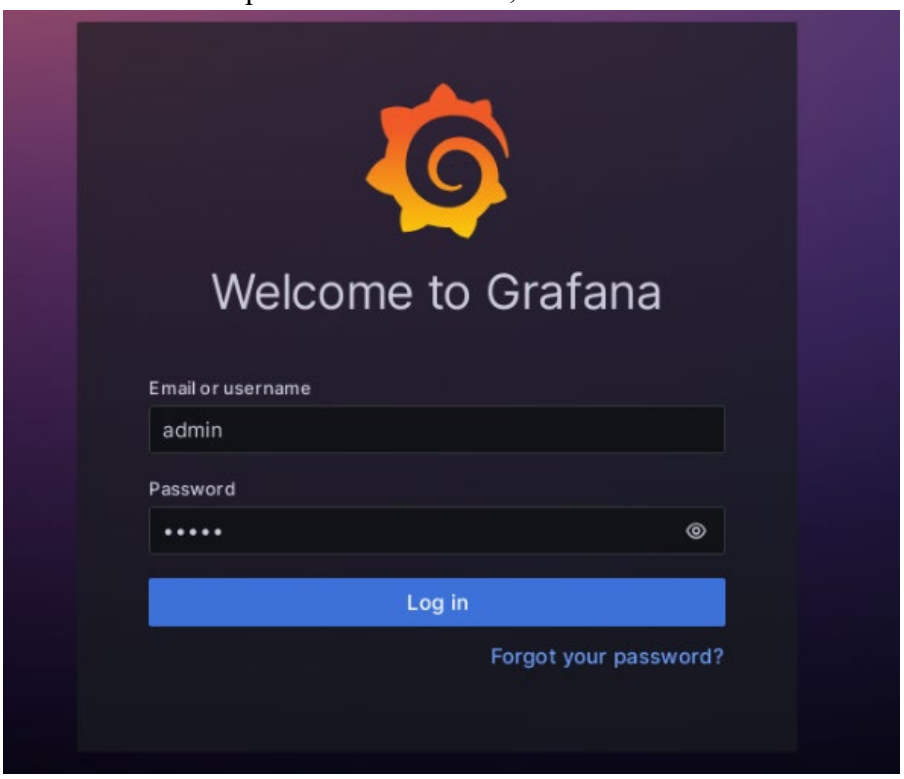
```
while true
do
curl http://localhost:8080/page2;
time=$((RANDOM %10));
sleep $time;
done
```

Запустите его приблизительно на 5 минут и после постройте график обращений к странице /page2 и с успешными кодами возврата

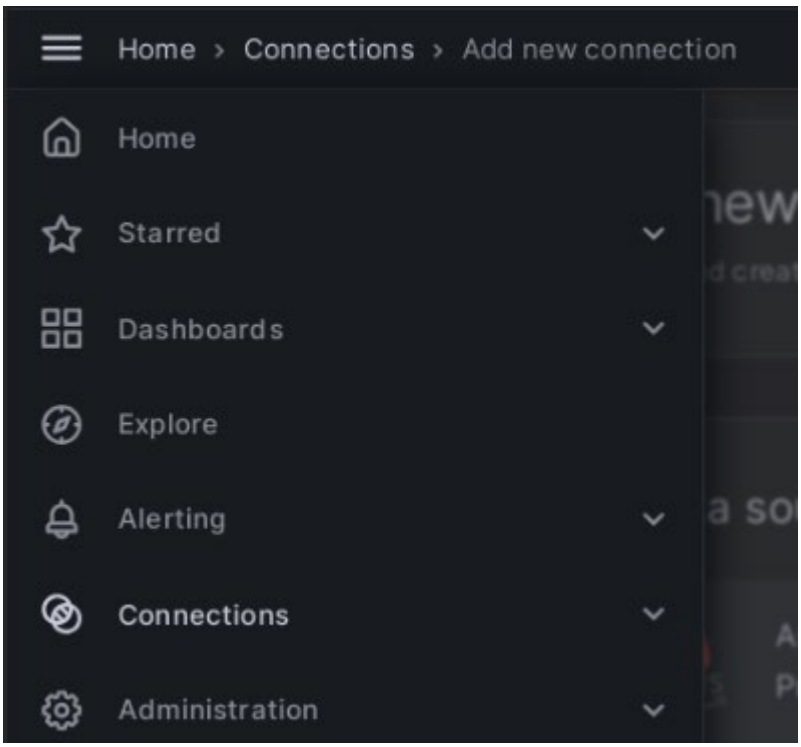
```
http_requests_total{path=”/page2”, status=”200”}
```



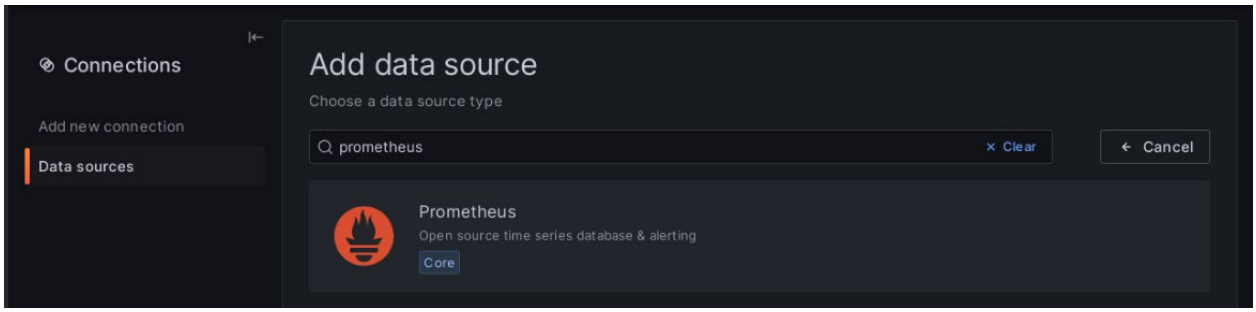
8. Запустите приложение Grafana. По умолчанию оно расположено по адресу localhost:3000. Пароль/логин – admin, admin



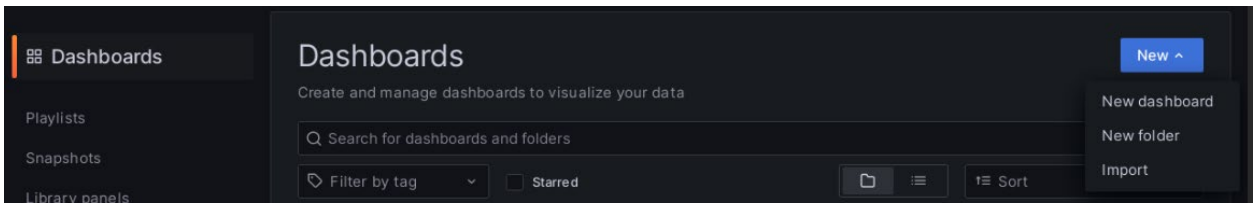
9. Добавим три графика. Для этого нажмите на три горизонтальные полосы в левой части экрана и выберете пункт “Connections”



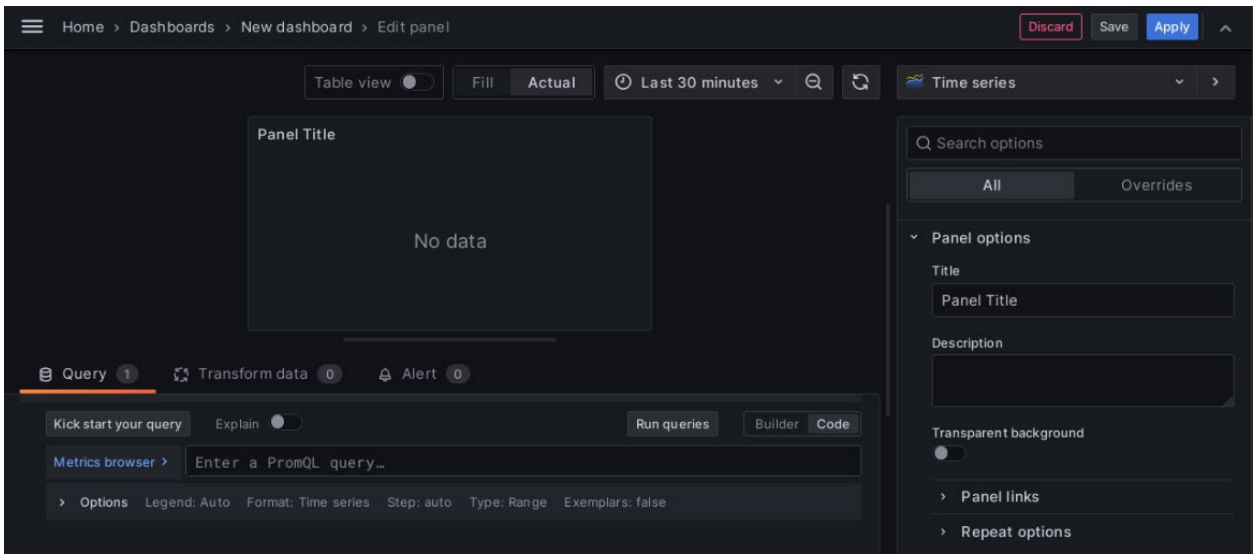
10. В разделе Data sources введите Prometheus в поле Add data source. Будет найден один источник – Prometheus, на который нужно нажать.



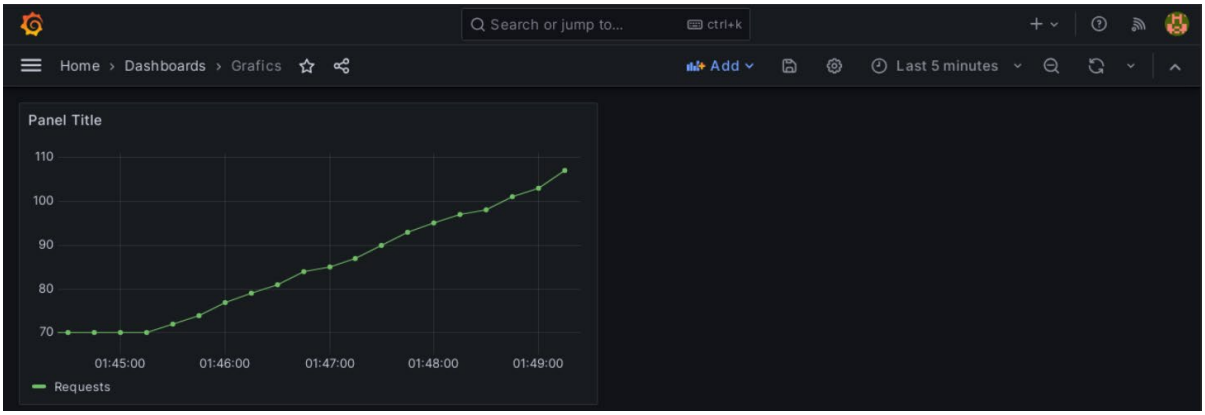
11. На странице с настройками в поле Connections укажите адрес программы. Напомним, по умолчанию она расположена по адресу http://localhost:9090. Далее в самом низу страницы нажмите синюю кнопку “Save & test”.
12. Следующим шагом добавим новый дашборд. Для этого перейдем во вкладку Dashboards -> New -> New Dashboard



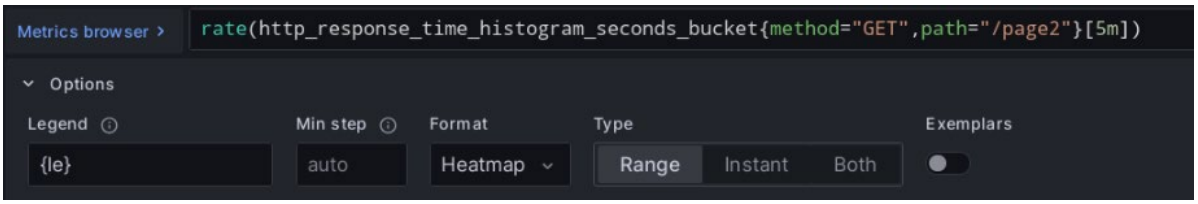
13. Добавим график. Для этого нажмем на большую синюю кнопку – «Add visualisation». В качестве источника для метрик укажите Prometheus. В правом окне выберете визуализацию – Time series. В нижней части экрана выберете пункт Code и укажите в нем запрос, приведенный выше: http_requests_total{path=”/page2”, status=”200”}



14. Чтобы подписать график, в нижнем поле «Options» в разделе Legend выберете Custom и подпишите график.
15. Чтобы сохранить график в правом верхнем углу нажмите кнопку Save. Дайте произвольное название созданному дашборду.

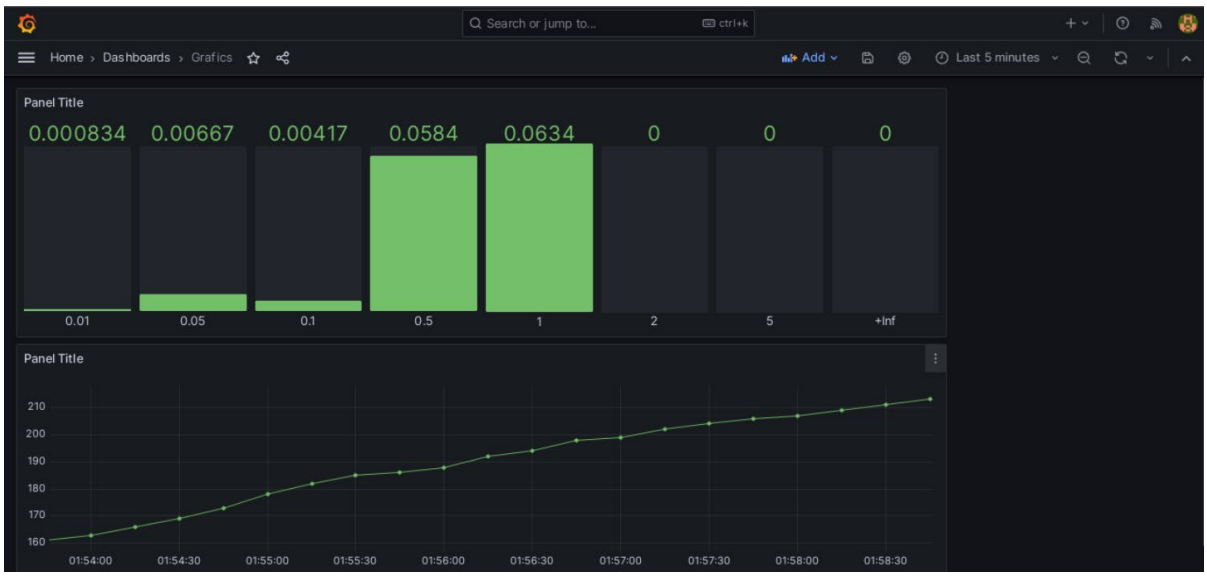


16. В качестве второго графика добавим гистограмму, отображающую частоту запросов. Для этого нажмем на синюю кнопку Add и выберем пункт visualization. Тип визуализации – Bar gauge, содержимое – rate(http_response_time_histogram_seconds_bucket{method=”GET”, path=”/page2”}[5m]), Legend – {le}, Format – Heatmap.



В данном примере мы берем значения за последние 5 минут и группируем по именам корзин. Сохраните полученную гистограмму.

Итого будет получена приблизительно следующая картинка:



Часть 2. Loki и Promtail

1. Установите программы Loki и Promtail. Для этого выполните следующие команды:
   Установка Loki:
   Скачайте последнюю версию Loki с GitHub

```
LOKI_VERSION=$(curl -s https://api.github.com/repos/grafana/loki/releases/latest | grep tag_name | cut -d '"' -f 4)
curl -LO "https://github.com/grafana/loki/releases/download/${LOKI_VERSION}/loki-linux-amd64.zip"
unzip loki-linux-amd64.zip
chmod +x loki-linux-amd64
sudo mv loki-linux-amd64 /usr/local/bin/loki
```

Скачайте пример конфигурации:

```
curl -LO "https://raw.githubusercontent.com/grafana/loki/${LOKI_VERSION}/cmd/loki/loki-local-config.yaml"
sudo mv loki-local-config.yaml /usr/local/bin/
```

После скачивания изучите конфигурацию и закомментируйте раздел querier
Создайте файл службы:
sudo nano /etc/systemd/system/loki.service

```
[Unit]
Description=Loki Log Aggregation System
After=network.target

[Service]
ExecStart=/usr/local/bin/loki -config.file=/usr/local/bin/loki-local-config.yaml
Restart=always
User=root

[Install]
WantedBy=multi-user.target
```

Запустите службу:

```
sudo systemctl daemon-reload
sudo systemctl enable loki
sudo systemctl start loki
```

Установка Promtail:

```
VERSION=$(curl -s
https://api.github.com/repos/grafana/loki/releases/latest \
  | grep tag_name | cut -d '"' -f4)
curl -LO
"https://github.com/grafana/loki/releases/download/${VERSION}/promtail-
linux-amd64.zip"
unzip promtail-linux-amd64.zip
chmod +x promtail-linux-amd64
sudo mv promtail-linux-amd64 /usr/local/bin/promtail
```

Создайте файл, например, /etc/promtail-local-config.yaml

```
server:
  http_listen_port: 9080
  grpc_listen_port: 0

positions:
  filename: /var/lib/promtail/positions.yaml

clients:
  - url: http://localhost:3100/loki/api/v1/push

scrape_configs:
  - job_name: system
    static_configs:
      - targets: [localhost]
        labels:
          job: varlogs
          __path__: /var/log/*log

  - job_name: goapp
    static_configs:
      - targets: [localhost]
        labels:
          job: goapp
          __path__: ТУТ ПУТЬ ДО ФАЙЛА С ЛОГАМИ
```

В качестве последнего параметра укажите путь до файла, где расположены логи приложения. Например, /home/USERNAME/…/…/logs.log

Настройка службы systemd для Promtail
Создайте файл /etc/systemd/system/promtail.service

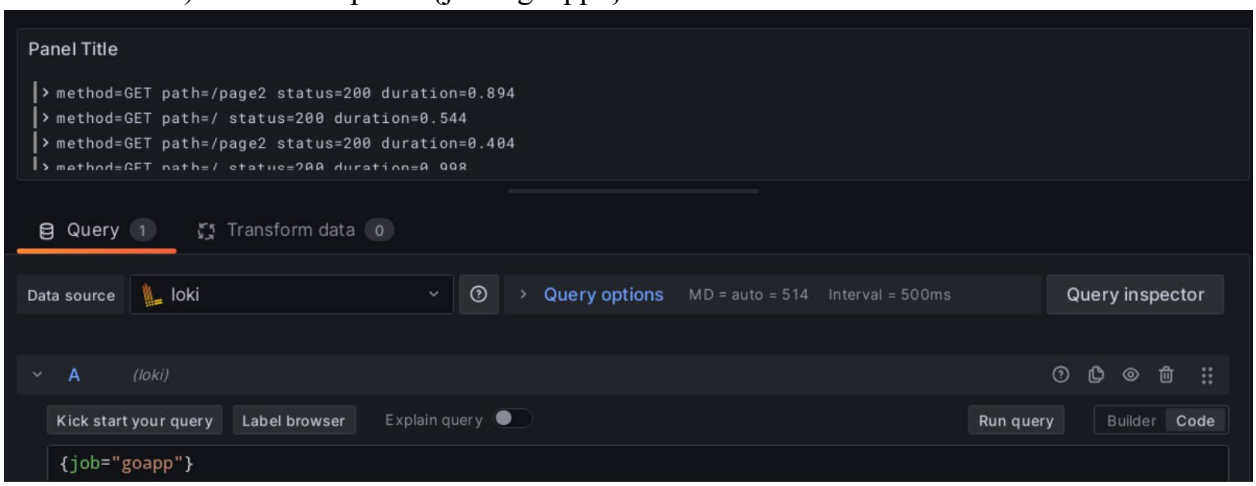
```
[Unit]
Description=Promtail service
After=network.target

[Service]
```

```
Type=simple
User=root
ExecStart=/usr/local/bin/promtail -config.file /etc/promtail-local-
config.yaml

[Install]
WantedBy=multi-user.target
```

Запустите и активируйте службу:

```
sudo systemctl daemon-reload
sudo systemctl enable --now promtail.service
sudo systemctl status promtail.service
```

2. В приложении Grafana выберете в качестве источника приложение loki (меню -> Connections -> Data Sources -> Add new -> Loki). По умолчанию адрес Loki http://localhost:3100.
3. Добавьте на дашборд новую панель – в качестве типа выберете Logs, источник (Data source) – Loki. Запрос – {job=”goapp”}



Сохраните созданный дашборд.

4. Выполните несколько запросов и убедитесь, что сообщения отображаются в логах.

Практико-ориентированная часть:

1. Настройте систему мониторинга проекта:
   - Используйте Prometheus для сбора метрик;
   - Используйте Loki для сбора логов;
   - Настройте отображение данных в Grafana.
2. Добавьте в ваш проект логирование любых действий и собирайте их с помощь Loki

Полезные ссылки

- Prometheus: https://prometheus.io/docs/
- Grafana: https://grafana.com/docs/
- Grafana Loki: https://grafana.com/docs/loki/latest/

Приложение

main.go

```
package main

import (
	"io"
	"log"
	"math/rand"
	"net/http"
	"os"
	"strconv"
	"time"

	"github.com/prometheus/client_golang/prometheus"
	"github.com/prometheus/client_golang/prometheus/promhttp"
)

var (
	// Метрики
	requestCounter = prometheus.NewCounterVec(
		prometheus.CounterOpts{
			Name: "http_requests_total",
			Help: "Total number of HTTP requests",
		},
		[]string{"method", "path", "status"},
	)

	responseTimeGauge = prometheus.NewGaugeVec(
		prometheus.GaugeOpts{
			Name: "http_response_time_seconds",
			Help: "Last HTTP response time in seconds",
		},
		[]string{"method", "path"},
	)

	requestDuration = prometheus.NewSummaryVec(
		prometheus.SummaryOpts{
			Name: "http_request_duration_seconds",
			Help: "Duration of HTTP requests in seconds",
			Objectives: map[float64]float64{
				0.5:  0.05,
				0.9:  0.01,
				0.99: 0.001,
			},
		},
		[]string{"method", "path"},
```

```
	)

	responseTimeHistogram = prometheus.NewHistogramVec(
		prometheus.HistogramOpts{
			Name:    "http_response_time_histogram_seconds",
			Help:    "Response time histogram in seconds",
			Buckets: []float64{0.01, 0.05, 0.1, 0.5, 1, 2, 5},
		},
		[]string{"method", "path"},
	)
)

func init() {
	// Регистрируем метрики
	prometheus.MustRegister(requestCounter)
	prometheus.MustRegister(responseTimeGauge)
	prometheus.MustRegister(requestDuration)
	prometheus.MustRegister(responseTimeHistogram)
}

func main() {
	rand.Seed(time.Now().UnixNano())

	// === Настройка логирования в stdout и файл ===
	logFile, err := os.OpenFile("app.log",
os.O_CREATE|os.O_WRONLY|os.O_APPEND, 0666)
	if err != nil {
		log.Fatalf("Ошибка при открытии файла для логов: %v", err)
	}
	defer logFile.Close()

	log.SetOutput(io.MultiWriter(os.Stdout, logFile))
	log.SetFlags(0) // без встроенной метки времени

	// === HTTP-хендлер ===
	http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request)
{
		start := time.Now()

		// Имитируем обработку запроса
		delay := time.Duration(rand.Intn(1000)) * time.Millisecond
		time.Sleep(delay)

		// 10% chance of error
		status := http.StatusOK
		if rand.Float64() < 0.1 {
			status = http.StatusInternalServerError
			w.WriteHeader(status)
		}

		// Метрики
```

```
            duration := time.Since(start).Seconds()
            labels := prometheus.Labels{
                  "method": r.Method,
                  "path":   r.URL.Path,
            }

            statusStr := strconv.Itoa(status)

            requestCounter.With(prometheus.Labels{
                  "method": r.Method,
                  "path":   r.URL.Path,
                  "status": statusStr,
            }).Inc()

            responseTimeGauge.With(labels).Set(duration)
            requestDuration.With(labels).Observe(duration)
            responseTimeHistogram.With(labels).Observe(duration)

            w.Write([]byte("Hello! Request took " +
time.Since(start).String()))

            // --- Простое логирование ---
            log.Printf("method=%s path=%s status=%s duration=%.3f",
                  r.Method, r.URL.Path, statusStr, duration)
      })

      // Метрики Prometheus
      http.Handle("/metrics", promhttp.Handler())

      log.Println("msg=Starting server on :8080")
      log.Fatal(http.ListenAndServe(":8080", nil))
}
```