GeekBrains

CI/CD

Troubleshooting

На этом уроке

1. Рассмотрим, что такое мониторинг и для чего он нужен.
2. Узнаем, что такое логирование и уровни логирования.
3. Выясним плюсы и минусы логирования.
4. Познакомимся с инструментами отрисовки графиков.
5. Рассмотрим несколько примеров траблшутинга GitLab.

Оглавление

Мы с вами уже успели разобрать основы CI и CD. После того как приложение успешно разработано, вмержено в репозиторий, протестировано и задеплоено на продакшн, возникает вопрос: «Как понять, что всё хорошо и приложение работает как задумано?» Получается, что нужно не только развернуть приложение в продакшн, но и получать обратную связь от него во время дальнейшей эксплуатации. Для этого и необходим мониторинг приложения.

Мониторинг

Чтобы максимально полно и информативно построить мониторинг, нужно учесть несколько моментов:

1. **Инфраструктура.** В первую очередь стоит задуматься о мониторинге инфраструктуры, в которой будет развёрнуто приложение, то есть мониторить состояние серверов, сети и так далее. Это необходимо для выявления технических проблем, например выхода из строя диска, заканчивающегося места или памяти.
2. **Приложение.** Важно получать и измерять показатели работоспособности самого приложения. Это могут быть RPS (request per second — количество запросов в секунду), время ответа, количество ошибок и т. п.

Мониторинг должен осуществляться с помощью внешнего процесса. Только внешний процесс может определить, запущен ли процесс приложения, и отправить алерт, если это не так.

Кроме того, это единственный способ гарантировать доступность данных мониторинга в периоды, когда сервис работает под высокой нагрузкой. В таких ситуациях функции мониторинга в самом сервисе могли бы перестать откликаться или откликаться с большой задержкой.

Цель мониторинга — не только сбор информации, но также её обработка и представление в удобном для анализа виде. Самый простой и очевидный способ сбора информации — логирование.

Логирование

Логирование представляет собой запись информации о работе приложения. Информация записывается в файл и сопровождается временной меткой (timestamp).

Плюсы логирования

1. Запись в файлы (файлы журналов) — достаточно простой процесс. Логирование можно осуществить с использованием любой технологии. Для многих языков программирования есть большой выбор библиотек, поддерживающих операции с файлами.
2. Операция добавления записи в файл практически не влияет на производительность приложения. Операция записи — самая эффективная операция с файлами.
3. Большие объёмы информации могут обрабатываться, например, с применением ротации файлов: если размер файла превышает некоторый установленный порог, создаётся новый файл, куда продолжает записываться информация. Спустя какое-то время данные сжимаются или удаляются.
4. Файлы легко искать и анализировать. Такие инструменты, как grep, позволяют фильтровать информацию из файлов.
5. Файлы легко читаются и интерпретируются людьми.
6. Файлы журналов не зависят от конкретной технологии. Информацию, записанную приложением в журнал, можно анализировать с применением самых разных технологий.

Итак, мы определились, что логирование необходимо. Осталось определиться, что именно и как стоит логировать.

Обычно в системах журналирования есть разграничение информации по уровням важности. Это помогает определить и ограничить, что именно попадёт в файл, и помогает минимизировать объём файла.

В основном используются следующие уровни логирования:

1. **Error.** Для критических ошибок. Например, закончились коннекты, а базе данных не удалось подключиться к серверу.
2. **Warning.** Некритические ошибки. Например, это может быть сообщение, что в кеше не нашлось нужных данных. Это не критично, так как приложение пойдёт в основную систему за данными, но эта информация может быть полезной.
3. **Info.** Для информационных сообщений. Содержит информацию об успешно выполненных прикладных процедурах. Например, подключении к базе, ответе от сервиса и т. п.

4. **Debug.** Для подробных сообщений, адресованных разработчикам. Не используется на постоянной основе, включается при возникновении проблем для поиска причин их возникновения. В debug-режиме приложение пишет очень подробную информацию, которая занимает много места и не нужна в режиме нормальной работы приложения.

Если система настроена на запись информации с меткой Info, данные с более высоким приоритетом тоже будут записываться. Например, warning- и error-сообщения.

Сообщения в файле могут сопровождаться уникальным кодом. Это позволяет быстро определить, где было сгенерировано сообщение. В документации может предоставляться описание кода ошибки, информация по устранению и т. п.

При всех плюсах и простоте реализации у логирования также есть минусы.

Минусы логирования

Чем больше файлов логов, тем сложнее их исследовать и проводить анализ. К примеру, если сервис развёрнут на 10 машинах и на каждой пишется лог, то при возникновении проблем с сервисом нужно будет ходить по всем 10 машинам и смотреть логи. Это не очень удобно.

Тут на помощь приходят различные инструменты для обработки файлов журналов.

Они решают три основные задачи:

1. Сбор данных из файлов, которые могут находиться на разных серверах, в одно место.
2. Парсинг файлов.
3. Возможность анализировать данные и, соответственно, идентифицировать ошибки.

К таким инструментам можно отнести Elasticsearch, Kibana.

Графики

Также может быть интересным получение и отображение числовых характеристик работы приложения. Например, количества запросов в секунду, времени ответа.

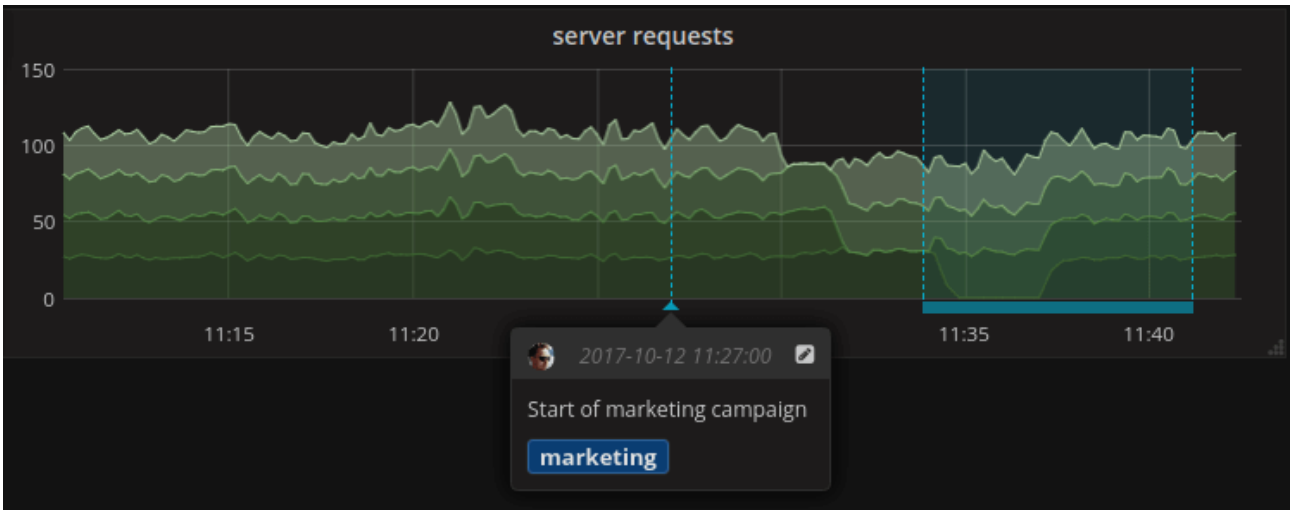
Измерение таких данных позволяет понять состояние системы и отследить, как оно меняется с течением времени. Это даёт возможность предсказать проблемы, например заканчивающееся место на диске.

Благодаря получению этих данных можно предупредить о проблеме и предотвратить её до того, как она достигнет критического уровня. Например, удалить ненужные файлы на диске.

Наиболее популярный инструмент — Grafana. Её можно использовать как для визуализации данных, так и для мониторинга и анализа данных.

Большой плюс Grafana — это open source проект. Она поддерживает достаточно большое количество источников данных: MySQL, PostgreSQL, InfluxDB, Zabbix, Prometheus, Elasticsearch и т. п.

В дашбордах Grafana можно использовать аннотации для отображения определённых событий на разных панелях.



Аннотации добавляются настраиваемыми запросами в Elasticsearch. На графике аннотация отображается вертикальной красной линией. При наведении курсора на аннотацию можно получить описание события и теги. Например, для отслеживания ответа сервера с кодом ошибки 5xx, перезапуска системы или проведения релиза. Благодаря этому можно легко сопоставить время, конкретное событие и его последствия в приложении и исследовать поведение системы.

Также Grafana имеет встроенный механизм оповещения, например по электронной почте или с помощью уведомлений через Slack, согласно определённым правилам.

Чтобы «пощупать» Grafana, не обязательно устанавливать её себе на хост, можно воспользоваться Grafana Demo.

Troubleshooting

Итак, когда у нас есть настроенный мониторинг, мы можем отслеживать состояние системы и реагировать, если что-то идёт не так или надвигается проблема.

Теперь разберёмся, как быть, если проблема уже случилась. Первый порыв — сразу начать выяснять, почему возникла проблема. Но это не совсем правильно: пока вы разбираетесь в причинах, проблемный сервис продолжает деградировать или вообще не работать и страдают пользователи.

Первым шагом при возникновении проблемы должно быть восстановление сервиса, хотя бы частичное. Например, можно увести трафик с проблемного хоста, откатить последний релиз, если проблемы начались сразу после него.

После того как работа сервиса восстановлена, можно переходить к траблшутингу, то есть поиску и устранению причины проблемы.

Troubleshooting plan

Несмотря на то, что проблемы с сервисом могут быть достаточно разнообразными, можно выделить основные универсальные шаги при поиске и устранении причин проблемы:

1. **Определить масштаб проблемы**. Проблема аффектит всех пользователей или только малую их часть? Возможно, проблема вообще не требует срочного вмешательства.
2. **Локализовать проблему**. В чём конкретно проблема? Не поднимается сервис, не открывается страница в браузере, сервис перестал отвечать на внешние запросы? Это, пожалуй, самый сложный пункт.
3. **Диагностика, исследование**. Установление причины проблемы. Что в логах, на графиках?
4. **Воспроизведение проблемы**. На третьем шаге мы делаем предположение о причинах проблемы, которое нужно проверить. Например, воспроизвести проблему в тестовой среде или на выведенном из боя хосте.
5. **Восстановление системы**. После того как причина сбоя установлена, можно приниматься за восстановление работоспособности системы.
6. **Написание post mortem**. Отчёт о том, что привело к поломке, что было сделано для устранения, что можно сделать в дальнейшем, чтобы не допустить повторения проблемы. Написание post mortem обязательно при больших факапах. Важно, что цель postmortem не в поиске виноватого, а в выявлении и исправлении ошибок.

Troubleshooting GitLab’a

Мы узнали общую схему действий при возникновении проблемы. Теперь разберём конкретные примеры.

Поскольку основным используемым инструментом у нас был GitLab, посмотрим, какие проблемы могут быть в GitLab и какие средства для траблшутинга в нём есть.

Проблемы в GitLab можно разделить на несколько видов:

1. Архитектурные проблемы.
2. Проблемы с Git.
3. Проблемы с CI/CD.

Архитектурные проблемы GitLab

Как мы уже знаем, GitLab состоит из множества компонентов. Как исследовать проблемы с различными компонентами, можно подробнее почитать в документации самого GitLab.

Сейчас мы рассмотрим только две возможные проблемы.

Проблемы с GitLab Runner’ами

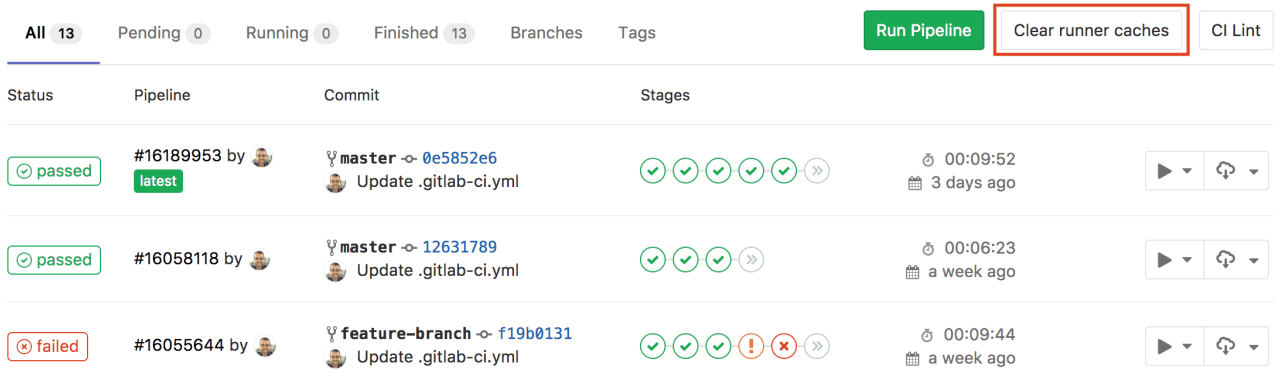
При использовании локального GitLab и GitLab Runner’ов в случае появления проблем с GitLab Runner’ами можно включить debug-режим в консоли:

```
gitlab-runner --debug run
```

Либо через правку конфига config.toml: нужно исправить `log_level` на `debug`.

Как мы уже знаем, debug-режим выдаёт максимально подробную информацию о том, что происходит в сервисе, что очень помогает при поиске причины проблемы.

Также может быть полезным выполнить очистку кеша, если он используется в gitlab_ci. Это можно сделать, если в репозитории перейти в Pipelines → Clear Runner Caches.



Проблемы с Gitaly

Gitaly — сервис GitLab, который предоставляет доступ к Git-репозиториям. Без него компоненты GitLab не могут читать и писать в Git.

При поиске проблем и неисправностей, связанных с Gitaly и Git-репозиторием, можно использовать [команду]():

```
gitaly-debug
```

Проблемы с Gitlab Container Registry

Тут самой частой проблемой оказывается невозможность запушить в registry. Рассмотрим две самые распространённые причины.

1. Отсутствие прав:

```
119 $ docker push $CI_REGISTRY_IMAGE
120 The push refers to repository [registry.gitlab.com/geekbrains-ci-cd-public/project1]
121 6af1577d8971: Preparing
122 be8b8b42328a: Preparing
123 denied: requested access to the resource is denied
125 Cleaning up file based variables
127 ERROR: Job failed: exit code 1
```

Как вы помните из второго урока, чтобы запушить что-либо (docker-контейнер, пакет и т. п.) в registry, нужно сначала авторизоваться. Если у юзера, из-под которого происходит push, нет прав на push, то ничего работать не будет. Можно посмотреть, какие права у deploy-токенов, в Repository → Deploy Tokens. У используемого токена в scopes должно быть `write_registry` или `write_package_registry`, в зависимости от того, куда вы пытаетесь запушить.

2. Попытка запушить слишком большой image. При превышении 5 минут может возникать эта ошибка, так как по умолчанию в настройках GitLab время жизни токена составляет 5 минут.

```
Unauthorized: authentication required
```

Проблемы с Git

Также разберём наиболее частые и простые проблемы, связанные с Git.

1. При пуше в репозиторий можно получить следующую ошибку:

```
remote: GitLab: 401 Unauthorized
```

Если при этом получается клонировать репозиторий по SSH и HTTPS, но не получается запушить в него, скорее всего, причина в том, что не синхронизирован файл gitlab-secrets.json с нодами GitLab.

2. При пуше в репозиторий:

```
Write failed: Broken pipe
fatal: The remote end hung up unexpectedly
```

Причин может быть несколько. Самые распространённые:

1. Недостаточный размер POST buffer в Git.
2. Ошибки в конфигурации SSH, если push проходит через SSH. Можно попробовать добавить в конфиг файл (~/.ssh/config):

```
Host your-gitlab-instance-url.com
  ServerAliveInterval 60
  ServerAliveCountMax 5
```

3. При push или pull через SSH:

```
Please make sure you have the correct access rights
and the repository exists.
...
ssh_exchange_identification: read: Connection reset by peer
fatal: Could not read from remote repository.
```

Или:

```
ssh_exchange_identification: Connection closed by remote host
fatal: The remote end hung up unexpectedly
```

Эта ошибка обычно указывает, что превышено число одновременных коннектов по SSH.

Проблему можно решить, увеличив количество коннектов на GitLab-сервере в /etc/ssh/sshd_config:

```
MaxStartups 100:30:200
```

Это значит, что после сотого коннекта 30% последующих коннектов будут дропаться, пока их количество не достигнет максимума в 200 коннектов.

Ошибки в job

Кроме ошибок в компонентах GitLab и Git, могут также встречаться ошибки в самих job GitLab'а при выполнении pipeline.

Наиболее частые проблемы:

1. Ошибки доступа и авторизации. Мы их рассмотрели выше.
2. Если в CI/CD проставлена галочка Skip outdated deployment jobs, может возникнуть такая ошибка:

```
The deployment job is older than the previously succeeded deployment job...
```

Это происходит из-за того, что job более старого пайплайна пытается запуститься после успешного выполнения более свежего пайплайна, то есть это ожидаемое поведение. Skip outdated deployment jobs как раз служит для предотвращения подобных ситуаций. Если эта функция не нужна, достаточно просто её выключить в настройках.

Если причина проблем не ясна, можно запустить job в debug-режиме. Для этого в gitlab_ci.yml в нужную job необходимо добавить строчку `CI_DEBUG_TRACE: "true"`:

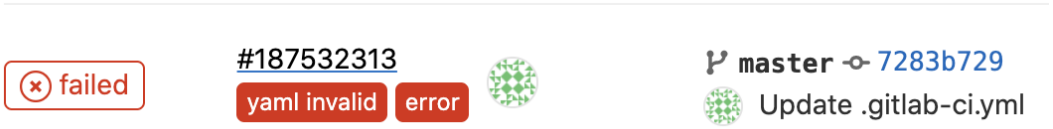
```
job_name:
  variables:
    CI_DEBUG_TRACE: "true"
```

В таком случае у нас будет более подробный лог с выводом всех имеющихся значений переменных.

Проблемы с gitlab_ci.yml

Теперь рассмотрим, какие проблемы могут возникнуть в самом gitlab_ci.yml

1. **Синтаксические ошибки**.



   Pipeline не запустится, если в файле gitlab_ci.yml будет невалидный YAML.

   В таком случае необходимо пойти и перепроверить сам файл на ошибки либо использовать встроенный CI Lint (CI/CD → Pipelines или CI/CD → Jobs)

2. **Ошибки верификации переменных**. Часто возникает ситуация, когда у используемых в gitlab_ci.yaml переменных присвоены неправильные значения или вовсе отсутствуют значения.

   Самое простое в такой ситуации — добавить вывод интересующих переменных. Либо можно включить debug-режим в job, и тогда в лог job попадут все переменные и их значения, в том числе скрытые и глобальные переменные самого GitLab.

Конечно, невозможно в одной лекции рассказать о всех проблемах, с которыми вы можете столкнуться при использовании GitLab. Проблемы и ошибки могут быть самыми разнообразными.

Самые распространённые ошибки и проблемы описаны в документации GitLab. Об известных багах и проблемах можно почитать в issue_tracker самого GitLab. А разобраться со всеми остальными проблемами помогут различные варианты debug в компонентах GitLab.

Практическое задание

См. в заданиях к лекции.

Глоссарий

Post mortem — отчёт с анализом причин возникновения проблемы, перечислением, что было сделано для устранения проблемы, что можно сделать в дальнейшем, чтобы не допустить повторения проблемы.

Troubleshooting — поиск и устранение проблемы.

Дополнительные материалы

1. Официальная документация по GitLab CI.
2. Официальная документация по Grafana.
3. Официальная документация YAML.
4. GitLab issue tracker.

Используемые источники

1. Бетси Бейер, Крис Джоунс, Дженнифер Петофф, Нейл Ричард Мёрфи «Site Reliability Engineering. Надёжность и безотказность, как в Google», 2018.
2. Официальная документация по Grafana.
3. Troubleshooting GitLab architecture.
4. Troubleshooting git.
5. Ошибки при выставленной Skip outdated jibs.
6. Job debug logging.