МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский технологический университет»
МИРЭА

Институт Информационных технологий
Кафедра Инструментального и прикладного программного обеспечения

Отчёт по лабораторной работе № _
по дисциплине
«Веб-программирование»

Выполнил Карих Дмитрий Степанович
студент группы ИСБОп-01-14

Принял _____

2017 год

Разработка бэкенда мобильного приложения «Wi-Fi в метро»

По мере роста аудитории приложения «Wi-Fi в метро», старый бэкенд, написанный на языке PHP, постепенно перестал справляться с нагрузкой. По этой причине было решено переписать проект на более удобный и современный язык Python, используя фреймворк Flask.

Постановка задачи

Основные требования к проекту остались прежними:

1. Разбор информации о новых сборках приложения из системы непрерывной интеграции и передача этих сведений пользователям;
2. Сбор анонимной статистики использования приложения (главная причина роста нагрузки на сервер);
3. Предоставление зеркальных копий сборок с других серверов для обеспечения их повышенной доступности.

При этом новый проект должен быть обратно совместим со старым, чтобы пользователи, не успевшие обновиться до новой версии приложения, не потеряли такую возможность.

Дополнительные требования:

1. Воспроизводимость окружения

   Все компоненты должны быть упакованы в контейнеры, которые будут одинаковыми для любой системы. Связи между контейнерами также должны быть задокументированы в рамках проекта.

2. Использование системы контроля версий
3. Автоматическая сборка и непрерывная интеграция

   При любом изменении в репозитории проекта сборка должна производиться автоматически. В процесс сборки входят подготовка контейнеров и их развёртывание на подготовленные для этого сервера.

Реализация проекта

Исходный проект был разбит на две независимые части и один служебный модуль. Каждая часть размещается в своём python-модуле (директории с файлом __init__.py).

Модуль «branches»

Данный модуль получает сведения о сборках и релизах приложения из различных источников и отдаёт их другим компонентам в формате вложенного словаря (dictionary).

В текущей реализации поддерживаются два источника:

1. Репозиторий проекта на GitHub

   Информация о релизах заполняется вручную владельцем репозитория. Одновременно могут существовать только две ветки обновления (Release и Pre-Release). Информацию о релизах можно получить через JSON API без предварительной аутентификации.

2. Директория проекта в системе непрерывной интеграции Jenkins

   Jenkins осуществляет автоматическую сборку и тестирование всех ответвлений исходного кода с GitHub и позволяет запросить сведения о ветках обновления с помощью аналогичного JSON API. Единственное отличие — сведения о сборках генерируются автоматически на основе порядкового номера и списка изменений.

Чтобы снизить нагрузку на другие серверы, модуль «branches» применяет кэширование на основе сетевого хранилища Redis (**re**mote **di**ctionary **s**erver). Такой подход позволяет запрашивать сведения только в том случае, если система сборки или мейнтейнер репозитория сообщают о наличии изменений. Redis хранит все данные в оперативной памяти, периодически синхронизируя их с жёстким диском, что обеспечивает сохранность данных и высокую скорость доступа к ним. Для использования Redis вместе с Python был выбран модуль «redis» из репозитория PyPI (Python Package Index).

Модуль «api»

Данный модуль непосредственно участвует в клиент-серверном взаимодействии. Через него пользователи и приложение получают информацию о сборках, отправляют данные для статистики и т.д.

Модуль был разделён на несколько логических частей, называемых в рамках фреймворка Flask чертежами (Blueprints):

1. /api/v1 — обратно совместимая версия API;
2. /releases — зеркалирование сборок приложения и предоставление доступа к ним;
3. /admin — набор административных команд (защищён паролем).

В главном файле модуля присутствует функция, которая предназначена для регистрации Blueprints во фреймворке.

```
def register(app):
    if config['admin'] == "true":
        app.register_blueprint(admin, url_prefix='/api/admin')
    app.register_blueprint(v1, url_prefix='/api/v1')
    app.register_blueprint(releases, url_prefix='/releases')
```

Эта функция вызывается из главного файла проекта, активируя таким образом все перечисленные выше пути.

```
app = Flask(__name__)
api.register(app)  # Подключение модуля «api»
```

Модуль «util»

В этом модуле находятся некоторые служебные классы и процедуры, которые часто используются в остальных модулях.

1. **config** — абстракция для конфигурирования сервера. Текущая реализация загружает настройки из переменных окружения, но можно без особых проблем сделать и стандартный файл конфигурации.

2. **requests** описывает объект CachedRequests, который позволяет кэшировать исходящие HTTP запросы сервера. Ответы удалённых серверов сохраняются в Redis и подгружаются оттуда при повторении запроса вместо его реальной отправки. Этот объект позволяет в одну строку кода скомбинировать модули «requests» и «requests_cache» из PyPI.

   Например, код для отправки запроса и кэширования ответа на 7 дней будет выглядеть так:

   ```
   with CachedRequests(ttl=7*24*60*60):
       res = requests.get('url')
   ```

   Первый вызов этого участка кода займёт некоторое время, а последующие выполнятся практически мгновенно.

3. **stats** — абстрация для сбора и записи статистики в агрегатор метрик StatsD. Взаимодействие с сервером StatsD производится при помощи стандартного модуля «statsd» из PyPI.

Движок

Если для проектов на PHP используется интерпретатор PHP, то для проектов на Python существует множество альтернатив, различных по назначению и проиводительности.

- В ходе разработки может быть полезно запустить сервер, который будет автоматически отслеживать изменения исходного кода и применять их. Такой подход позволяет сразу же проверить эффективность кода и найти большую часть ошибок. Эту роль отлично выполняет сам фреймворк Flask, который включает в себя однопоточный отладочный сервер.

- Однако однопоточный сервер неприемлем для «боевых» условий, когда сервер должен обработать десятки запросов в секунду. Для такой цели отлично подойдёт связка NGINX и uWSGI, где первый работает с соединениями, а второй исполняет код. NGINX – это веб-сервер, поэтому он отлично оптимизирован для работы с тысячами клиентов. uWSGI – сервер приложений, который может запускать множество процессов и исполнять в них Python-код.

Логично, что стоит обеспечить поддержку обоих вариантов для упрощения разработки и достижения максимальной производительности.

Docker-контейнер

Для получения одинакового окружения сервера на всех системах было решено использовать проект Docker, который позволяет собирать образы, включающие все необходимые компоненты, а затем создавать неограниченное число одинаковых контейнеров на базе образа.

Контейнеры в данном случае практически не отличаются от виртуальных машин: у них есть IP-адрес, дисковое пространство и оперативная память. Важное отличие заключается в том, что контейнеру выделяются только необходимые ресурсы, в отличие от виртуальных машин, которые запрашивают ресурсы на запуск целой операционной системы. Это позволяет нам запускать десятки контейнеров даже на слабых машинах, в том числе и ноутбуках.

Создание образа тесно связано с файлом Dockerfile, который содержит набор инструкций для настройки контейнера. При генерации образа эти команды последовательно выполняются, а их результат накладывается в виде очередного слоя файловой системы. Ниже приведён Dockerfile, созданный в процессе разработки этого проекта. В нём последовательно устанавливаются uWSGI, NGINX и все зависимости проекта, указанные в файле requirements.txt.

```
# Контейнер основан на дистрибутиве Alpine Linux
FROM python:3-alpine

# Устанавливаем uWSGI из PyPI
RUN apk --no-cache add gcc linux-headers musl-dev \
 && pip install uwsgi \
 && apk del gcc linux-headers musl-dev
```

```
# Добавляем файлы проекта и устанавливаем зависимости
ADD app /app
RUN pip install -r /app/requirements.txt

# Устанавливаем NGINX и supervisor
RUN apk --no-cache add nginx supervisor
ADD container /

# Настраиваем контейнер
EXPOSE 80
CMD ["/usr/bin/supervisord"]
```

Следует отдельно отметить наличие такого компонента, как supervisor. Его роль заключается в параллельном запуске нескольких процессов (NGINX и uWSGI). Дело в том, что Docker изначально расчитан на запуск одного процесса на контейнер, но в нашем случае это не очень удобно: NGINX по умолчанию не умеет отслеживать изменение IP-адреса сервера uWSGI, что может привести к потере работоспособности всего проекта. В нашем же случае у uWSGI всегда будет один и тот же адрес (127.0.0.1).

Обработка и хранение метрик

На текущей стадии своего развития, приложение «Wi-Fi в метро» генерирует десятки тысяч запросов в день. В моменты пиковой нагрузки количество запросов достигает 1100 в минуту. Чтобы обработать и сохранить такой набор данных без мощного сервера требуется несколько этапов:

1. Сервер получает POST-запрос с метриками, разбирает и отправляет их в StatsD;
2. StatsD каждые 10 секунд подводит итоги по каждой метрике и отправляет сумму и частоту в базу данных InfluxDB;
3. InfluxDB сохраняет эти данные в течение некоторого времени (1 месяц);
4. Grafana или любой другой клиент запрашивают данные у InfluxDB и строят наглядные графики.

Таким образом, нам требуется связать как минимум 4 контейнера:

1. Основной контейнер проекта
2. Redis
3. StatsD
4. InfluxDB

Также для расширения функциональности Grafana был добавлен контейнер с сервисом InfluxDB Timeshift Proxy, который позволяет получать данные со смещением во времени. Он проксирует все запросы от Grafana к InfluxDB, то есть напрямую InfluxDB взаимодействует только со StatsD и прокси.

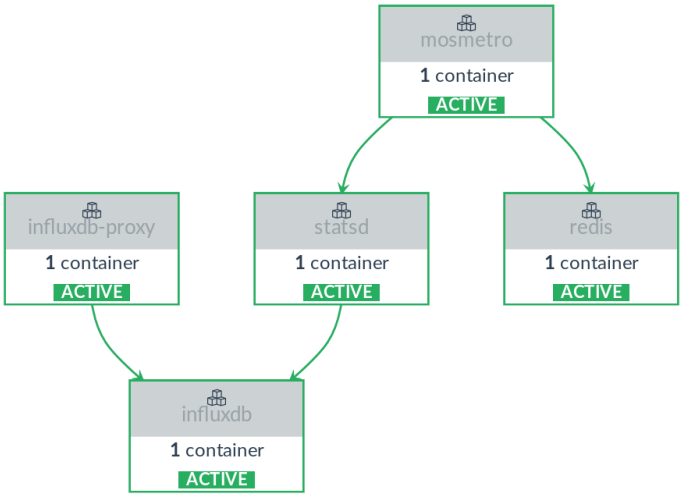

Рис. 1: Связи контейнеров

Полученная структура отлично справляется с нагрузкой, а также может быть свободно разделена на несколько серверов.

Docker Compose

Чтобы не создавать каждый раз структуру с Рис. 1 вручную, воспользуемся инструментом docker-compose. Настройки контейнеров прописываются в специальном файле docker-compose.yml, который можно включить в репозиторий проекта и использовать на любом компьютере.

Так как у нас должна быть возможность создавать две различные среды (для разработки и окончательного тестирования), создадим четыре файла Compose:

1. 1-common.yml (обязательно) – описание общих для всех конфигураций компонентов: InfluxDB, StatsD и Redis;

2. 2-development.yml – описание главного контейнера, в котором запустится однопоточный сервер на базе Flask;

3. 2-production.yml – описание главного контейнера, в котором запустятся NGINX и uWSGI;

4. 3-statistics.yml (необязательно) – окружение для тестирования обработки метрик: InfluxDB Timeshift Proxy, Grafana и Chronograf (аналог PhpMyAdmin для InfluxDB).

Из двух файлов с цифрами 2 нужно выбрать только один. Применить конфигурацию можно следующим образом:

```
docker-compose -f файл-1.yml -f файл-2.yml -f файл-3.yml <операция>
```

Для запуска нужно вместо <операция> подставить «up -d», а для остановки — «down».

Rancher Compose

Одним из требований к этому проекту была возможность автоматического развёртывания серверов. В качестве одного из возможных вариантов рассматривается развёртывание на кластер серверов, находящийся под управлением Rancher. Rancher позволяет управлять десятками и сотнями контейнеров на одном или нескольких серверах.

Для автоматизации работы с Rancher был создан инструмент rancher-compose, который во многом напоминает docker-compose. Основные отличия заключаются в том, что он использует немного другой формат файла docker-compose.yml, дополнительный файл rancher-compose.yml, а также управляет контейнерами не на локальной машине, а на виртуальном кластере.

Таким образом, имея адаптированный файл docker-compose.yml и дополнительный rancher-compose.yml можно без проблем запустить обновление сервисов в соответствии со структурой проекта.

Сборка и развёртывание

Чтобы сборка проекта происходила при каждом его изменении, воспользуемся системой непрерывной интеграции Jenkins, а именно её механизмом Pipeline. Данный механизм позволяет управлять процессом сборки с помощью файла Jenkinsfile (по аналогии с Dockerfile).

При каждом запуске сборки Jenkins загрузит Jenkinsfile из репозитория проекта и выполнит все необходимые операции. В нашем случае сборка будет состоять из нескольких шагов:

1. **Pull**: Получение исходного кода с GitHub;
2. **Build**: Сборка образа на основе Dockerfile;
3. **Push**: Отправка образа на Docker Hub – публичное хранилище образов Docker;
4. **Deploy**: Запуск Rancher Compose и обновление всех контейнеров в стеке;
5. **Notify**: Отправка отчёта о сборке на GitHub.

Docker Hub в данном случае необходим, так как Rancher может использовать только готовые образы.

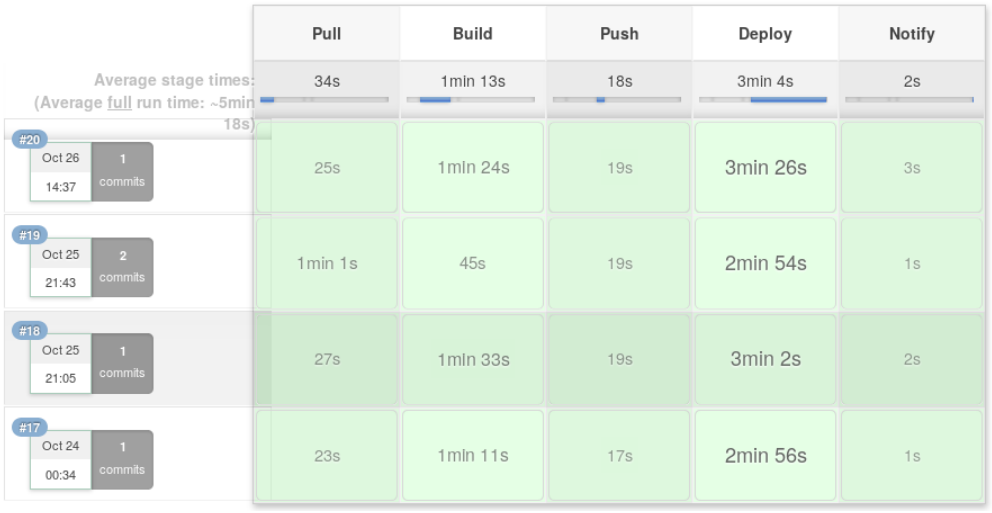

Рис. 2: Отчёт по последним 4-м сборкам проекта

Заключение

Использование связки NGINX + uWSGI + Python + Flask позволило добиться значительно большей производительности проекта, чем NGINX + PHP. Важную роль здесь играет не только производительность сервера приложений uWSGI, но и удобство управления проектом на более современном языке программирования.

Применив Docker при разработке проекта нам удалось обеспечить практически полную воспроизводимость окружения, что гарантирует одинаковую работу программного обеспечения на любом оборудовании и операционной системе (главное, чтобы процессор поддерживал архитектуру x86_64). Также мы получили возможность свободно перемещать контейнеры между серверами (кроме баз данных).

Непрерывная интеграция позволила сократить временные затраты на сборку и обновление контейнеров. А rancher-compose также обеспечил возможность обновлять все сервисы незаметно для пользователя, сначала запуская новые контейнеры и уже затем останавливая старые.

Список использованных проектов

1. Flask - http://flask.pocoo.org/
2. Python - https://www.python.org/
3. uWSGI - https://uwsgi-docs.readthedocs.io/en/latest/
4. NGINX - http://nginx.org/
5. Supervisor - http://supervisord.org/
6. Alpine Linux - https://www.alpinelinux.org/
7. Docker - https://www.docker.com/
8. Docker Compose - https://docs.docker.com/compose/
9. Rancher - http://rancher.com/
10. Rancher Compose - http://rancher.com/docs/rancher/latest/en/cattle/rancher-compose/
11. Redis - https://redis.io/
12. StatsD - https://github.com/etsy/statsd
13. InfluxDB - https://www.influxdata.com/
14. InfluxDB Timeshift Proxy - https://github.com/maxsivanov/influxdb-timeshift-proxy
15. Grafana - https://grafana.com/
16. Jenkins - https://jenkins.io/

Список использованных сервисов

1. GitHub - https://github.com/mosmetro-android/backend
2. Docker Hub - https://hub.docker.com/r/thedrhax/mosmetro-backend/