#noBackend,

или Как выжить

в эпоху толстеющих клиентов

BC 2016 Backend Conf

Николай Самохвалов
@samokhvalov.com
twitter: **@postgresmen**

Часть 1

Эпоха толстеющих клиентов

4 очень популярных языка для интернет-проектов

Год рождения — 1995

«Гадкий утёнок» JavaScript

1995: 1-я версия — за 10 дней

1990-е: второстепенные роли в веб-проектах

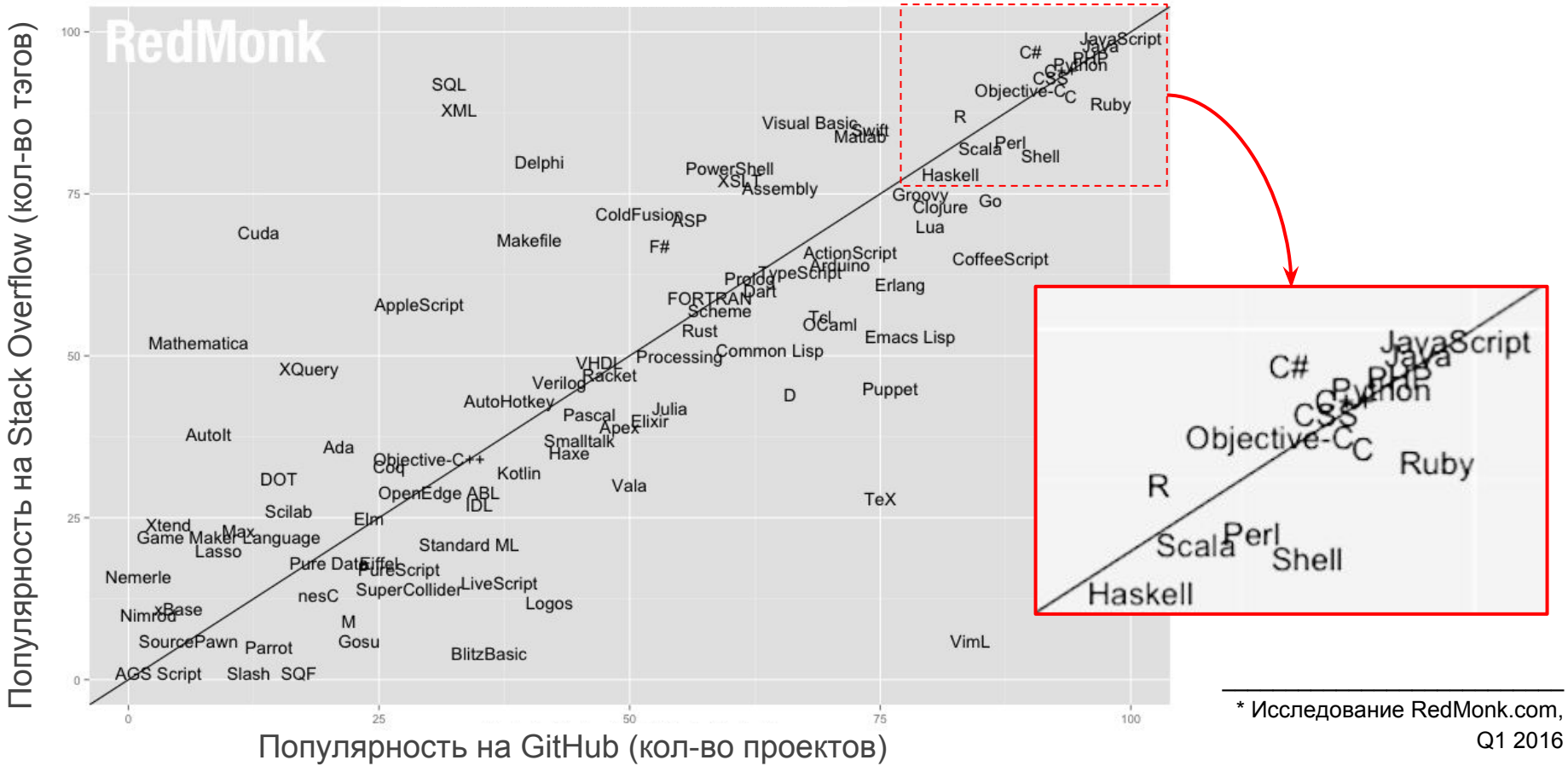
2000-е: Web 2.0, резкое усиление важности

2010-е: доминирование в самых различных областях

- Web apps
- Node.js, npm
- React Ecosystem

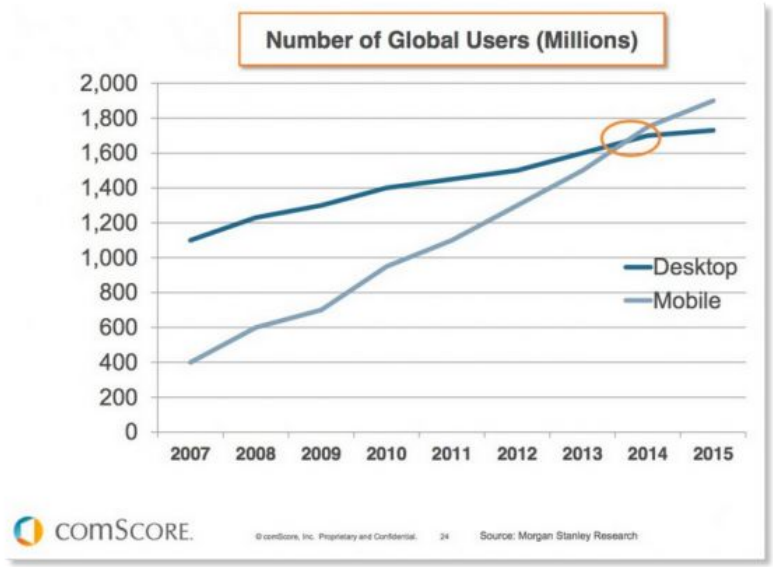
...Типичная история о силе дистрибуции и монополии

Популярность языков программирования*



* Исследование RedMonk.com, Q1 2016

«Мобильные» пользователи преобладают ещё с 2014



- декабрь 2014: 51% трафика в Рунете — мобильные устройства
- май 2015: поиском Google.com пользуются больше с мобильных, чем с десктоп

#noBackend

a.k.a. Serverless

«Толстые» клиенты в вебе
Множество фреймворков
React Ecosystem

iOS, Android
Mobile First

Основной фокус разработки продукта — **FRONTEND**

backend — просто должен работать
(безопасный, надёжный, производительный)

no, BACKEND!

Если у вас «зоопарк»

(web, mobile apps, browser extensions, desktop apps, smartTV, IoT, etc),

то единственный путь жить и развиваться —

простой/надёжный/безопасный **BACKEND** с API

(например, REST API или новомодный GraphQL)

Часть 2

Как выжить

“true noBackend” – используем облака

Кладбище mBaaS (Mobile Backend as a Service)

StackMob.com:

2013: PayPal покупает
2014: PayPal закрывает, R.I.P.

Parse.com:

2013: Facebook покупает
2016: Facebook закрывает, R.I.P.

...полные аналоги Parse использовать теперь как-то «стрёмно»*

— остаётся использовать AWS (с их Cognito, Lambda), Google Cloud

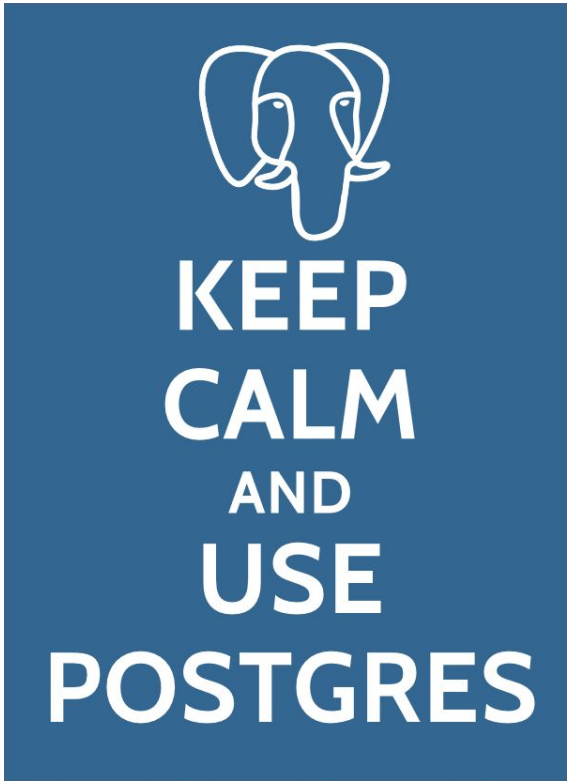
---

* но если очень хочется, есть много всякой всячины:
https://github.com/anaibol/awesome-serverless

...или ...а давайте всё же сами!

...например, PostgreSQL

...например, PostgREST (http://postgrest.com)



PostgreSQL + PostgREST

Serve a RESTful API from any Postgres database

— путь настоящего джедая: сможем не только выжить,

но и использовать *силу*!

PostgREST — API для вашей БД

PostgreSQL

http://postgrest.com

Написан на Haskell
MIT license
Активно развивается
чат: https://gitter.im/begriffs/postgrest

```
CREATE VIEW v1.person
  AS SELECT * FROM public.person;
```

→ /person

GET	→ SELECT
POST	→ INSERT
PATCH	→ UPDATE
DELETE	→ DELETE

```
CREATE FUNCTION v1.myfunc(...) …
LANGUAGE ...;
```

→ /rpc/myfunc

Только POST

(функции на языках: SQL, plpgsql, plpython, plr, plv8 и т.д.!)

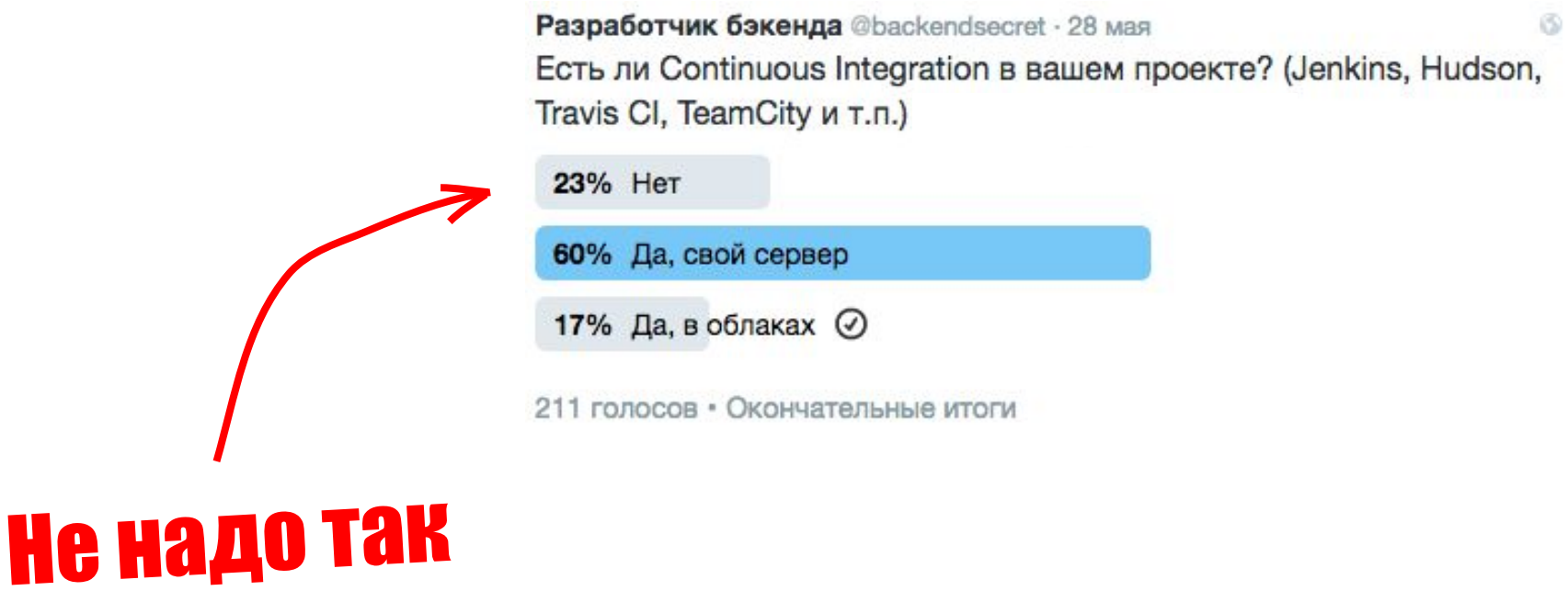
Урок 1: качество

Обязательно, обязательно автоматизируем тестирование API

- дома (Jenkins, TeamCity, etc)
- или в облаках (Circle CI, Travis CI, etc)

Тестируем:

- не только «хорошее», но и **«плохое»**
- быстродействие


Разработчик бэкенда @backendsecret · 28 мая
Есть ли Continuous Integration в вашем проекте? (Jenkins, Hudson, Travis CI, TeamCity и т.п.)
23% Нет
60% Да, свой сервер
17% Да, в облаках
211 голосов • Окончательные итоги

Не надо так

Проверка работы REST API

Как работать с REST API:

- Старый добрый Firefox умеет «Edit and Resend» (разовые проверки)

- **cURL**
- **HTTPie** (http://httpie.org)

- **Postman** (пачка тестов, environments — отгружаем в файлы)
- **newman** — для консольного запуска пачки Postman-тестов

Урок 2: безопасность. Обязательно тестируем!

DB level: используем `has_***_privilege` в тестах sqitch, pg_tap или аналогах

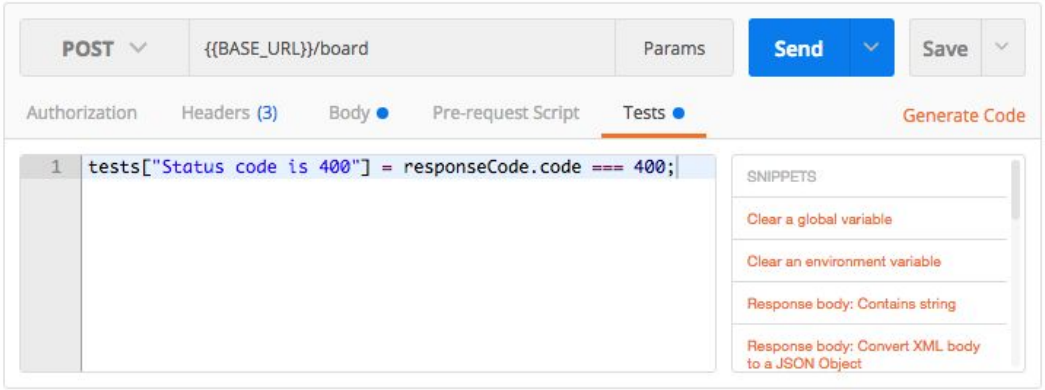
```
SELECT 1 / (not has_table_privilege)::int4
FROM has_table_privilege('anonymous', 'v1.post', 'select');
```

Деление на 0, если *есть* привилегия!

API level: пишем тесты для «плохих» случаев и проверяем:

- `401 Unauthorized`
- `403 Forbidden`
- `400 Bad Request`

и т.п.



Урок 2-а: анонимные запросы и безопасность

«а» — анонимы

Изучаем и работаем с JWT (см., например, http://jwt.io)

Урезанная DB role для анонимных доступов
(`--anonymous` у PostgREST)

Этой роли — никаких прав, кроме 2-3 методов в схеме `v1`
(можно с `SECURITY DEFINER`),
например: `register`, `login`, `password_reset`

На другие методы прав у аноним-роли быть не должно!
(**тестируем** API-вызовы)

Урок 2-b: права на столбцы

PostgreSQL

1) Совсем не светим секретное — не включаем в выборку представления:

```
CREATE VIEW v1.post AS
    SELECT id, title, body, moderated -- Никаких email, password и т.п.
    FROM public.post;
```

2) Управляем доступом с помощью прав на столбцы:

```
GRANT SELECT,DELETE ON v1.post TO apiuser;
GRANT INSERT,UPDATE (title, body) ON v1.post TO apiuser; -- moderated - R/O
```

Урок 2-с: запрет доступа к «чужим» строкам

Нельзя позволять юзеру менять чужие данные => нужно проверять `user_id`

PostgreSQL до 9.4 включительно:

используем триггеры и `current_setting('postgrest.claims.XXXXX')`

PostgreSQL 9.5+: используем Row-Level Security

...и опять не забываем **тестировать**

Урок 3-а: производительность. CPU *vs* network

Типичный пример из social media: выбор самых свежих материалов из коллекций, на которые подписан юзер. Главное — round-trip time (RTT)!

Таблицы: `person`, `post`, `collection`, `person2collection`

Вариант 1:

```
select …
from post p … join person2collection p2c ..
where p2c.collection_id in (...list…)
  And p.id < :prev_id
order by p.id desc limit :per_page;

select … from collection where id in (...);

select … from person … where id in (...);
```

– 3 API-вызова, RTT*3

Вариант 2:

```
select
  p.*
  row_to_json(c.*),
  row_to_json(p.*)
from post p
  join person2collection p2c ..
  join collection c ..
  join person p ..
where p.id < :prev_id
order by p.id desc limit :per_page;
```

- 1 API-вызов, RTT*1, ура!

Урок 3-b: «продвинутый» SQL

10 млн юзеров, 100 млн постов, 1 млн коллекций => оба варианта ~1-10 sec

Можно ли быстре?!
Да. Изучаем джедайские техники:

> Максим Богук, pgDay'14 «Неклассические приёмы оптимизации запросов»
> http://pgday.ru/files/pgmaster14/max.boguk.query.optimization.pdf

Придётся освоить и применить в комлексе:

- Recursive CTE (`WITH RECURSIVE ...`)
- работа с массивами
- свёртка строк
- Window functions
- `generate_series, generate_subscripts`
- Loose indexscan https://wiki.postgresql.org/wiki/Loose_indexscan

```
WITH RECURSIVE -- HUGESQL NS: see #1704
uids AS (SELECT ARRAY(SELECT user_id UNION SELECT "followuser_id" FROM public.user2user WHERE "user_id" = :user_id LIMIT 1000) AS _uids), -- uid list (user IDs); LIMIT 1000 because of performance :-/
bids AS (SELECT ARRAY(SELECT box_id FROM public.user2box WHERE "user_id" = user_id LIMIT 1000) AS _bids), -- bid list (board IDs); LIMIT 1000 because of performance :-/
ubids AS (SELECT array_length(_uids, 1) as _uids_size, array_length(_bids, 1) as _bids_size, array_cat(_uids, _bids) as _ubids FROM uids JOIN bids ON 1=1),
gs AS (SELECT _pos FROM generate_subscripts((SELECT _ubids FROM ubids), 1) AS gs(_pos)), --pregenerated iterator array for user IDs
t AS (
    SELECT
        NULL::integer as _p,
        _uids_size,
        NULL::public.postt AS result,
        0::integer AS _rows_found, (
            SELECT ARRAY(
                SELECT
                    CASE WHEN row_number() OVER (ORDER BY 1) <= _uids_size THEN ( -- assume that order of concat arrays hasn't changed
                        SELECT pp.id FROM public.post pp --JOIN "pnct_Board" b on box_id = b.id JOIN "pnct_User" u ON pp."user_id" = u.id AND role IN ('user', 'administrator')
                        WHERE pp."user_id" = _ubid AND (pic_id IS NOT NULL OR image_original IS NOT NULL)
                            AND (boundary_post_id IS NULL OR pp.id < boundary_post_id) /*e.g.: AND pp.id < 1113340385*/ -- PAGING (1 of 2)
                        ORDER BY pp.id DESC LIMIT 1
                    ) ELSE (
                        SELECT pp.id FROM public.post pp --JOIN "pnct_Board" b on box_id = b.id JOIN "pnct_User" u ON pp."user_id" = u.id AND role IN ('user', 'administrator')
                        WHERE box_id = _ubid AND (pic_id IS NOT NULL OR image_original IS NOT NULL)
                            AND (boundary_post_id IS NULL OR pp.id < boundary_post_id) /*e.g.: AND pp.id < 1113340385*/ -- paging (2 of 2)
                        ORDER BY pp.id DESC LIMIT 1
                    ) END
                FROM unnest(_ubids) u(_ubid)
            )
            FROM ubids
        ) AS _pids_ts --INITIAL ARRAY of latest posts for uids & bids
    FROM ubids
    UNION ALL -- ... and now iterations begin
    SELECT
        _pos,
        _uids_size,
        CASE WHEN _rows_found >= 0 THEN (
            SELECT post
            FROM public.post
            WHERE id = _pids_ts[_pos]--return row to the result set if we already go through _offset or more entries
        )
        ELSE NULL END,
        _rows_found + 1, --increase found rows count
        _pids_ts[1:_pos-1] || (
            CASE WHEN _pos <= _uids_size  THEN(
                SELECT pp.id FROM public.post pp --JOIN "pnct_Board" b on box_id = b.id JOIN "pnct_User" u ON pp."user_id" = u.id AND role IN ('user', 'administrator')
                WHERE pp."user_id" = (SELECT _ubids[_pos] FROM ubids) AND pp.id < _pids_ts[_pos] AND (pic_id IS NOT NULL OR image_original IS NOT NULL)
                ORDER BY pp.id DESC LIMIT 1
            ) ELSE (
                SELECT pp.id FROM public.post pp --JOIN "pnct_Board" b on box_id = b.id JOIN "pnct_User" u ON pp."user_id" = u.id AND role IN ('user', 'administrator')
                WHERE box_id = (SELECT _ubids[_pos] FROM ubids) AND pp.id < _pids_ts[_pos] AND (pic_id IS NOT NULL OR image_original IS NOT NULL)
                ORDER BY pp.id DESC LIMIT 1
            ) END
        ) || _pids_ts[_pos+1:array_length(_pids_ts,1)]  --replace current entry of latest post for the uid with previous by post.id for the same uid
    FROM (
        SELECT
            *,
            (SELECT _pos FROM gs ORDER BY _pids_ts[_pos] DESC NULLS LAST LIMIT 1) AS _pos --find position of the latest entry in the ubids list
        FROM t OFFSET 0
    ) AS t2
    WHERE _rows_found < 0 + COALESCE(limit_results, 100) --we had found the required amount of rows (offset+limit done) // NS: we don't use offset here; instead look for PAGING above
),
SELECT (result).id, _rows  Google Chrome  esult IS NULL ORDER BY _rows_found
;
```

PostgreSQL

~ 1-10ms !!!

BackendConf

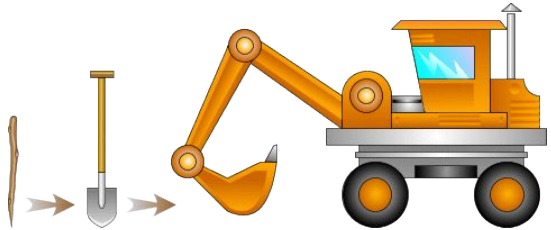
Урок 3-с: масштабируемость. Три уровня

Уровень 1: много экземпляров PostgREST
Уровень 2: часть GET-запросов → readonly PostgREST, подключённые к Postgres-реплике(ам) (направляем с помощью nginx). Принудительный мастер — с помощью custom http header.

Уровень 3: как масштабировать мастер? Сходу — пока никак.
Варианты:
3.1) микросервисная архитектура
(например, spilo и patroni от Zalando
– см. Highload++2015 http://www.highload.ru/2015/abstracts/1846.html)

3.2) CitusDB?
Пока нет, мешают `WITH` в PostgREST (возможно, уберут)

3.3) Можно просто подождать (см. планы Postgres по кластеру)



3.4) **А надо ли *вам*?** Миллион TPS на 1 мастере — может быть выгоднее «webscale» с неэффективными узлами, как 1 трактор может быть выгоднее 100 людей с лопатами!

Урок 4: что не стоит делать внутри

PostgreSQL

Не стоит делать внутри:

если непредсказумое время работы,
если есть работа с внешними ресурсами (http-запросы, API)

Примеры:

- Отправка почты, СМС
- Сохранение картинок в хранилище
- Вызовы внешних API

Урок 5: асинхронная работа

Обрабатываем асинхронно (демоны на python/ruby/nodejs/etc), отправляя сообщения с помощью:

- `LISTEN`/`NOTIFY` (пример есть в документации по PostgREST http://postgrest.com/examples/users/#password-reset)
- mbus (Р. Друзягин, И. Фролков, доклад PgDay'15 http://pgday.ru/files/papers/23/pgday.2015.messaging.frolkov.druzyagin.pdf)
- Сообщения в AMPQ (например, PostgreSQL LISTEN Exchange: https://github.com/aweber/pgsql-listen-exchange)

Памятка #noBackend-разработчика

1. Ни шагу без CI (прежде всего, *негативные* проверки API)
2. Проверить (и проверять тестами!) все двери:
   a. что может аноним?
   b. защищены (чтение, модификация) ли важные поля?
   c. можно ли поменять «чужие» записи? а заставить хозяина поменять свои собственные?
3. Проверить эффективность и быстродействие:
   a. можно ли сделать ещё меньше HTTP-запросов?
   b. насколько они эффективны сложные запросы и есть ли пути оптимизации?
   c. легко ли «положить» backend неожиданными фильтрами, сортировками, соединениями?
   d. легко ли «положить» его большим количеством запросов?
   e. какие пути смасштабироваться есть в арсенале?
4. Дополнительно:
   a. как делаются асинхронные задачи (скорость, надёжность, блокировки)?
   b. есть ли зависимость от внешних ресурсов? Если да, сможем ли сделать асинхронную работу?

THE END

Спасибо!

https://twitter.com/postgresmen

http://PostgreSQLRussia.org

http://Postgres.CHAT