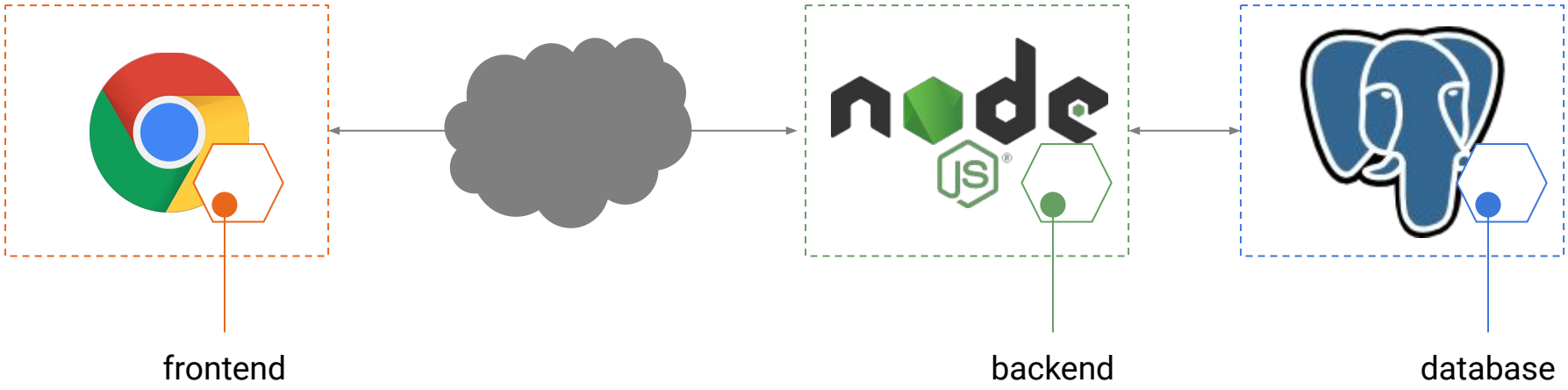
JS Level 3

Node.js

Предисловие

На прошлых лекциях мы написали собственное серверное приложение поверх модуля http из Node.js и научились работать с СУБД PostgreSQL.

Давайте теперь объединим всё в одну картину:



Это самая распространённая схема организации приложений на сегодняшний день

Фреймворки и библиотеки

Соответственно, frontend мы писать уже умеем (и с помощью Vanilla JS, и с помощью библиотеки React), а что с backend и database?

Конечно же мы можем дальше использовать самописный наш сервер (и он даже будет быстрее большинства готовых решений, т.к. в нём нет ничего лишнего, но не всегда), но с этим есть проблема: в большинстве проектов мы будем работать в команде и неплохо бы, чтобы если новый человек приходит в команду, ему не приходилось с нуля всё объяснять. Поэтому достаточно часто вместо самописных решений используют уже готовые.

Кроме того, готовые решения уже оттестированы сообществом и для них написано большое количество совместимых расширений (минусы, конечно же, тоже есть – их чаще ломают "ботами", их необходимо изучать, просто так не изменишь логику работы, в зависимостях готовых решений чаще находят уязвимости и т.д.)

Фреймворки и библиотеки

С базами данных примерно всё то же самое: мы, конечно, можем изучить TCP-протокол работы с PostgreSQL, написать свою библиотеку и использовать, но зачем, если есть уже готовые?

Поэтому, условно, мы можем разделить всех разработчиков на несколько уровней:

1. Разработчики бизнес-логики <- мы с вами
2. Разработчики библиотек и фреймворков
3. Разработчики платформы Node.js
4. Системные разработчики

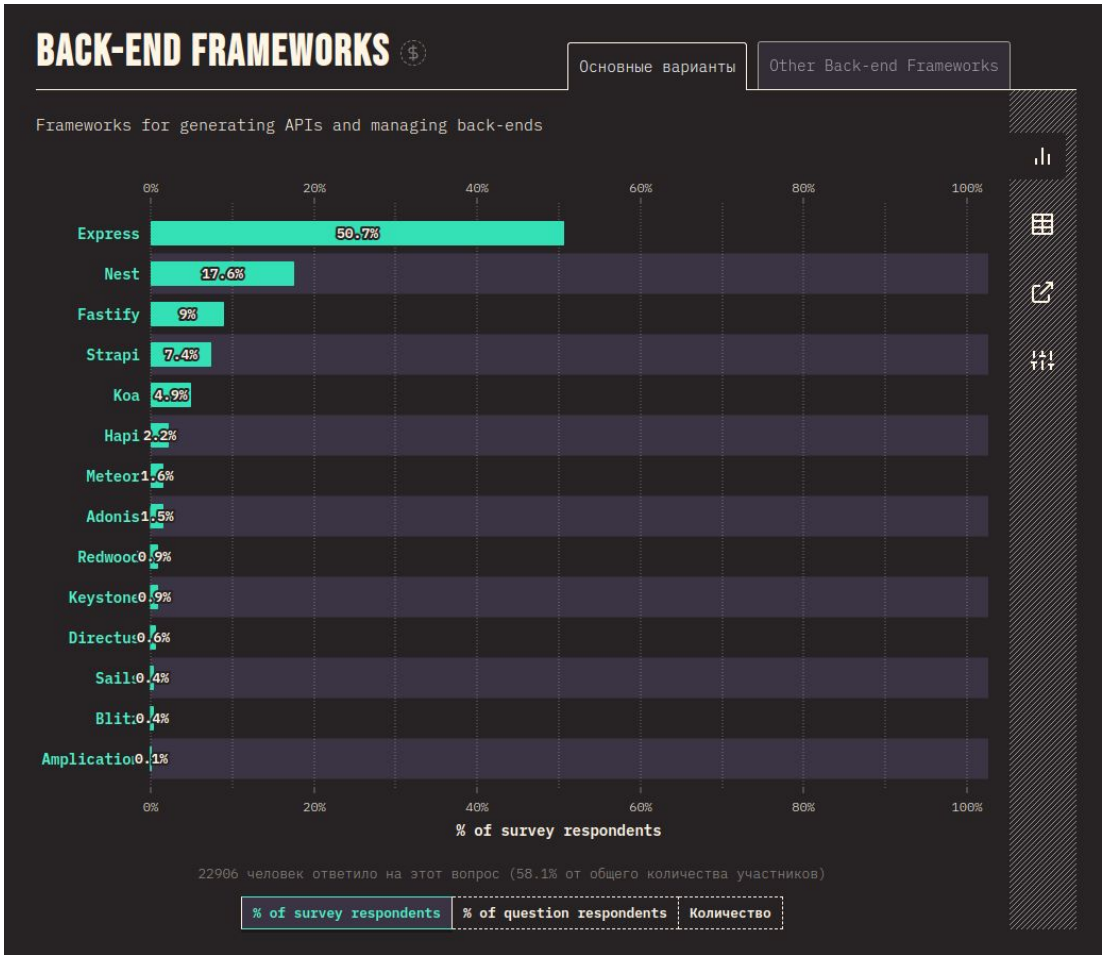
Open Source Components

На уровне разработчиков бизнес-логики принято по максимуму использовать уже готовые решения, поскольку это позволяет буквально собирать готовое приложение из кубиков, дописывая лишь специфичные для нашего приложения кусочки кода.

И только когда готовые решения не устраивают из соображений скорости, безопасности и т.д., пишут уже собственные решения.

Open Source Components

В части backend'а на список самых популярных библиотек/фреймворков можно посмотреть на результаты исследования State of JS:



Open Source Components

С базами данных всё немного проще: обычно есть 1-2 ключевых библиотеки, которые позволяют из Node.js работать с выбранной СУБД (в нашем случае – PostgreSQL). Мы будем использовать node-postgres.

NPM

npm

В состав Node.js входит специальный инструмент – npm, который позволяет устанавливать библиотеки из репозитория npmjs.com (именно там публикуется большинство библиотек):

```
  npm --help

Usage: npm <command>

where <command> is one of:
    access, adduser, audit, bin, bugs, c, cache, ci, cit,
    clean-install, clean-install-test, completion, config,
    create, ddp, dedupe, deprecate, dist-tag, docs, doctor,
    edit, explore, fund, get, help, help-search, hook, i, init,
    install, install-ci-test, install-test, it, link, list, ln,
    login, logout, ls, org, outdated, owner, pack, ping, prefix,
    profile, prune, publish, rb, rebuild, repo, restart, root,
    run, run-script, s, se, search, set, shrinkwrap, star,
    stars, start, stop, t, team, test, token, tst, un,
    uninstall, unpublish, unstar, up, update, v, version, view,
    whoami

npm <command> -h  quick help on <command>
npm -l            display full usage info
npm help <term>   search for help on <term>
npm help npm      involved overview
```

npm

npm – это:

- Node.js package manager (менеджер пакетов для Node.js)
- npm Registry (реестр пакетов)
- npm CLI (инструмент командной строки, поставляемый вместе с Node.js)

В терминах этой системы все внешние библиотеки и инструменты называются пакетами. Эта же система хранит онлайн большой реестр пакетов (почти все – бесплатные), которые вы можете скачать и установить для собственного использования.

Важно: в организациях обычно поднимают собственные корпоративные репозитории и npm устанавливает пакеты уже из них, а не с npmjs.com.

npm

С npm вы уже знакомы по предыдущему курсу, поэтому при возникновении вопросов, обращайтесь к нему.

NPM

Инициализация пакета

До этого npm-пакет для нас инициализировал CRA (Create React App): создавал package.json, прописывал в него необходимые скрипты, зависимости и т.д.

Сейчас нам предстоит это сделать вручную.

Инициализация пакета

Мы находимся в каталоге C:\projects\social и в нём будем создавать пакет.

С точки зрения npm пакет – это набор файлов + файл описывающий пакет (это package.json). Чтобы создать файл package.json нужно ввести команду npm init:

```
C:\projects\social>npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
```

Инициализация пакета

Вам зададут ряд вопросов о вашем проекте, достаточно просто жать Enter (тогда будут использованы значения по умолчанию):

```
package name: (social)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to C:\projects\social\package.json:

{
  "name": "social",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}


Is this OK? (yes)
```

package.json

После описанных действий в вашем каталоге должен появится файл package.json вот с таким содержимым:

{} package.json > ...

```
{
  "name": "social",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  ▷ Debug
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Изменять мы его пока не будем, идём дальше.

Установка и удаление

С точки зрения управления зависимостями у npm есть две ключевые команды:

- npm install <имя пакета> – установка
- npm uninstall <имя пакета> – удаление

Все толковые программисты – ленивые, поэтому чтобы целиком не писать npm install/uninstall, есть сокращения:

- npm i – сокращение для npm install
- npm un – аналог npm uninstall

Под зависимостью понимается что-то, что нужно вашему проекту для работы.

Установка и удаление

Давайте установим пакет для работы с PostgreSQL: npm i pg

Появится каталог node_modules (туда и будет установлен нужный нам пакет), и файл package-lock.json. Кроме того, в файле package.json появится секция:

```
"dependencies": {
  "pg": "^8.11.2"
}
```

Установка и удаление

Теперь попробуем удалить: npm un pg

Секция dependencies в package.json станет пустой и, поскольку зависимостей у нас совсем нет, каталог node_modules удалится.

Установим снова: npm i pg

ПОДКЛЮЧЕНИЕ К БД

Подключение к БД

Полная документация по работе с пакетом находится по ссылке. Мы же по шагам пройдём основные моменты.

Общая схема работы выглядит следующим образом:

1. При старте приложения подключаемся к базе данных
2. В обработчиках http-запросов, выполняем запросы к БД

Чтобы подключиться к базе данных, нужно, чтобы PostgreSQL был запущен. Удостоверьтесь, что вы можете к нему подключиться с помощью SqlTools и только потом пишите код.

Promises API

API основано на Promise, а значит нам придётся писать либо цепочки .then/.catch/.finally, либо использовать async/await (напоминаем, что await можно использовать только внутри async-функций*).

Использовать async/await гораздо приятнее, поэтому мы пойдём по этому пути.

Примечание*: чуть позже увидим, что если используем ESM-модули, то поддерживается top-level await.

API

Давайте создадим каталог sql, в котором и будем размещать наши тестовые запросы, которые мы потом перенесём в приложение:

sql > JS all.js > ...

```
1  const {Pool} = require('pg');
2  const dsn = process.env.DSN ?? 'postgres://app:pass@localhost:5432/db';
3  const pool = new Pool({connectionString: dsn});
4
5  const all = async () => {
6    // TODO:
7  };
8
9  all();
10
```

Здесь мы делаем следующее:

1. Получаем тип Pool из пакета pg
2. Получаем строку подключения (либо из переменных окружения, либо хардкодим тестовую строку)
3. Создаём объект для работы с БД

API

Как вы видите, работать не особо удобно, т.к. приходится создавать функцию и потом её вызывать, поскольку мы не сможем await писать на верхнем уровне (а на .then/.catch программировать неудобно). Поэтому давайте переключимся на режим ESM, для этого нужно выполнить два действия:

1. В package.json добавить "type": "module":

{} package.json > ...

```
{
  "name": "social",
  "version": "1.0.0",
  "description": "",
  "type": "module",
  "main": "index.js",
```

2. В js-файлах везде require заменить на import (конкретно для node-postgres это будет выглядеть так):

sql > JS all.js > ...

```
import pg from 'pg';
const { Pool } = pg;
const dsn = process.env.DSN ?? 'postgres://app:pass@localhost:5432/db';
const pool = new Pool({ connectionString: dsn });
```

API

Теперь сами запросы: всё достаточно просто, для у нас есть метод .query, в которой можно передать сам запрос (SQL) и массив параметров (про них – чуть позже):

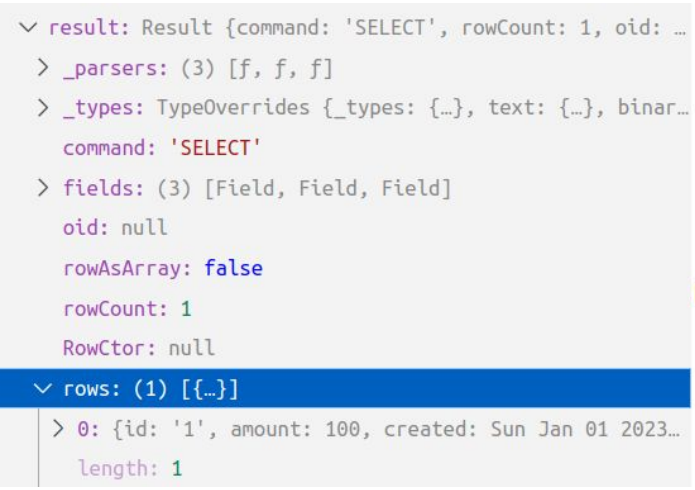
sql > JS all.js > ...

```
1  import pg from 'pg';
2  const { Pool } = pg;
3  const dsn = process.env.DSN ?? 'postgres://app:pass@localhost:5432/db';
4  const pool = new Pool({ connectionString: dsn });
5
6  const result = await pool.query(
7    `SELECT id, amount, created FROM payments ORDER BY id LIMIT 5`
8  );
9
10 pool.end();
```

pool.end() закрывает ресурсы, связанные с подключением.

API

Запустимся под отладкой и увидим следующее:




```
2  const { Pool } = pg;
3  const dsn = process.env.DSN ?? 'postgres://app:pass@localhost:5432/db';
4  const pool = new Pool({ connectionString: dsn });
5
6  const result = await pool.query(
7    `SELECT id, amount, created FROM payments ORDER BY id LIMIT 5`
8  );
9
10 pool.end();
11
12
```

Т.е. в result есть массив rows, который и содержит список строк, возвращённых нашим SELECT'ом.

API

Это значит, что мы можем переписать наш запрос для удобства следующим образом:

sql > JS all.js > ...

```
1  import pg from 'pg';
2  const { Pool } = pg;
3  const dsn = process.env.DSN ?? 'postgres://app:pass@localhost:5432/db';
4  const pool = new Pool({ connectionString: dsn });
5
6  const {rows} = await pool.query(
7    `SELECT id, amount, created FROM payments ORDER BY id LIMIT 5`
8  );
9
10 console.log(rows);
11
12 await pool.end();
```

Как видно из примера, всё достаточно просто: осталось разобраться только с тем, как передавать параметры и когда вызывать pool.end().

API

С параметрами достаточно просто:

1. В самом запросе мы их обозначаем placeholder'ами – $1, $2, $3
2. В запрос мы их передаём в виде массива, где первый элемент массива соответствует $1, второй – $2 и т.д.:

sql > JS all.js > ...

```
1  import pg from 'pg';
2  const { Pool } = pg;
3  const dsn = process.env.DSN ?? 'postgres://app:pass@localhost:5432/db';
4  const pool = new Pool({ connectionString: dsn });
5
6  const {rows} = await pool.query(
7    `SELECT id, amount, created FROM payments ORDER BY id LIMIT $1`,
8    [5]
9  );
```

API

Давайте для интереса добавим ещё столбец, чтобы продемонстрировать на примере INSERT, как можно передавать несколько параметров:

Run on active connection | Select block

```
DROP TABLE IF EXISTS payments;
CREATE TABLE payments (
  id BIGSERIAL PRIMARY KEY,
  amount INT NOT NULL CHECK (amount > 0),
  status TEXT NOT NULL CHECK (status IN ('INPROGRESS', 'SUCCESS', 'FAIL')),
  created TIMESTAMPTZ NOT NULL DEFAULT CURRENT_TIMESTAMP
);
```

API

В данном примере: const {rows: [row]} – это деструктуризация в переменную row первого элемента из массива rows:

sql > JS add.js > ...

```
1  import pg from 'pg';
2  const { Pool } = pg;
3  const dsn = process.env.DSN ?? 'postgres://app:pass@localhost:5432/db';
4  const pool = new Pool({ connectionString: dsn });
5
6  const {rows: [row]} = await pool.query(
7    `
8    INSERT INTO payments(amount, status)
9    VALUES ($1, $2)
10   RETURNING id, amount, status, created
11   `,
12   [1000, 'INPROGRESS']
13 );
14
15 console.log(row);
16
17 await pool.end();
```

API

Но есть одна проблема – если мы посмотрим на вывод, то увидим (у вас дата будет другая):

```
{
  id: '1',
  amount: 1000,
  status: 'INPROGRESS',
  created: 2023-01-01T00:00:00.000Z
}
```

Интересно здесь то, что id – это строка, а не число, как бы нам хотелось.

API

Всё дело в том, разработчики пакета pg крайне аккуратно относятся к совместимости с предыдущими версиями Node.js (в которых когда-то давно не было типа BigInt, который бы соответствовал BIGINT/BIGSERIAL в PostgreSQL):

sql > JS add.js > ...

```
1  import pg from 'pg';
2  const { Pool } = pg;
3  const dsn = process.env.DSN ?? 'postgres://app:pass@localhost:5432/db';
4  const pool = new Pool({ connectionString: dsn });
5
6  pg.types.setTypeParser(20, BigInt);
7  pg.types.setTypeParser(1016, o => pg.types.getTypeParser(1016)(o).map(BigInt));
8
```

Теперь всё будет хорошо до тех пор, пока мы не решим всё сериализовать в JSON, BigInt там не поддерживается (привыкайте: это реальный мир, где многие вещи не работают и не известно, когда заработают).

API

Поэтому мы можем сделать небольшой хак и будем переводить всё в Number, при этом имея в виду, что когда транзакций у нас станет очень-очень много, мы столкнёмся с проблемами:

sql > JS add.js > ...

```
1  import pg from 'pg';
2  const { Pool } = pg;
3  const dsn = process.env.DSN ?? 'postgres://app:pass@localhost:5432/db';
4  const pool = new Pool({ connectionString: dsn });
5
6  pg.types.setTypeParser(20, Number);
7  pg.types.setTypeParser(1016, o => pg.types.getTypeParser(1016)(o).map(Number));
8
```

Это один из вариантов (наряду с тем, чтобы оставить id строкой, который вы встретите).

API

Но есть ли другие варианты? Да, можно опять вернуться к строке, но в качестве строки использовать некоторые специально-генерируемые строки – UUID:

```
Run on active connection | Select block
1  DROP TABLE IF EXISTS payments;
2  CREATE EXTENSION pgcrypto;
3  CREATE TABLE payments (
4    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
5    amount INT NOT NULL CHECK (amount > 0),
6    status TEXT NOT NULL CHECK (status IN ('INPROGRESS', 'SUCCESS', 'FAIL')),
7    created TIMESTAMPTZ NOT NULL DEFAULT CURRENT_TIMESTAMP
8  );
```

Теперь данные будут вида:

```
{
  id: 'eeaf6390-c523-429c-9faa-43667934ec3b',
  amount:1000,
  status: 'INPROGRESS',
  created: 2023-01-01T00:00:00.000Z
}
```

BIGINT vs UUID

Нужно понимать, что не существует идеального способа: UUID работает медленнее, BIGINT/BIGSERIAL – быстрее, UUID можно безопасно генерировать на разных серверах (т.е. разделять данные по разным базам данных), BIGINT/BIGSERIAL придётся "накручивать" вручную распределяя диапазоны значений, по UUID бессмысленно сортировать, а по BIGSERIAL – вполне логично.

Поэтому выбор конкретного типа будет зависеть от вашего приложения и вашей команды. Мы же, для упрощения, остановимся на варианте с 33 слайда, понимая всю ограниченность нашей реализации.

API

Оставшиеся запросы (byId, edit, remove, restore) реализовывать мы не будем, поскольку вам пора привыкать к общему формату общения разработчиков (так будет и в документации, и в чатах, и на StackOverflow): вам показывают базовый пример и дают ссылку на документацию (либо просто дают ссылку на документацию), а дальше вы адаптируете пример под свои задачи.

GRACEFUL SHUTDOWN

Graceful Shutdown

Когда мы пишем серверные приложения, взаимодействующие с внешними сервисами (например, СУБД, клиентами и т.д.), крайне важно корректно завершать (по возможности) работу приложения и закрывать все ресурсы (например, вызывать тот же pool.end()).

Когда приложение работает на сервере, то операционная система при завершении работы (например, вы обновляете систему и перезапускаете все необходимые сервисы) может уведомлять приложение, о том, что необходимо завершить работу.

Делается это с помощью механизма сигналов (это из мира Linux).

Graceful Shutdown

Сигналы бывают разные, нас будут интересовать следующие:

- SIGINT – например, когда мы запускаем node app.js и нажимаем Ctrl + C
- SIGTERM – например, когда ОС решает сообщить приложению, что нужно корректно завершить работу (при плановом обслуживании ОС, нормальной перезагрузке* и т.д.)
- SIGKILL – например, ОС "требует" приложение завершить свою работу

Важным является то, что приложение может отреагировать на SIGINT и SIGTERM и выполнить какие-то действия, например, закрыть подключения к СУБД. На SIGKILL ничего сделать нельзя – ОС просто "убивает" ваш процесс.

Примечание*: в противоположность "жёсткой" перезагрузке, когда вы нажимаете кнопку перезагрузки на системном блоке.

Graceful Shutdown

В простейшем варианте это можно сделать следующим образом:

sql > JS add.js > ...

```
1  import pg from 'pg';
2  const { Pool } = pg;
3  const dsn = process.env.DSN ?? 'postgres://app:pass@localhost:5432/db';
4  const pool = new Pool({ connectionString: dsn });
5
6  pg.types.setTypeParser(20, Number);
7  pg.types.setTypeParser(1016, o => pg.types.getTypeParser(1016)(o).map(Number));
8
9  const {rows: [row]} = await pool.query(
10   `
11   INSERT INTO payments(amount, status, created)
12   VALUES ($1, $2)
13   RETURNING id, amount, status, created
14   `,
15   [1000, 'INPROGRESS']
16 );
17
18 console.log(JSON.stringify(row));
19
20 const shutdown = () => {
21
22   pool.end(() => console.log('pool closed'));
23 };
24
25 process.on('SIGTERM', shutdown);
26 process.on('SIGINT', shutdown);
```

Graceful Shutdown

В случае использования http-сервера всё станет немного сложнее:

```
54  const shutdown = () => {
55    server.close(() => {
56      console.log('server closed');
57      pool.end(() => console.log('pool closed'));
58    });
59  };
60
61  process.on('SIGTERM', shutdown);
62  process.on('SIGINT', shutdown);
```

Мы сначала остановим сервер, а потом будет уже останавливать пул.

Graceful Shutdown

Теперь, если вы запустите приложение, то оно не завершит свою работу, пока вы в терминале не нажмёте Ctrl + C или не завершите его работу через диспетчер задач.

При этом в терминале вы увидите сообщение: pool closed.

ИТОГИ

Итоги

В этой лекции мы обсудили некоторые ключевые моменты, связанные с работой с СУБД PostgreSQL из Node.js.

ДОМАШНЕЕ ЗАДАНИЕ

Как сдавать ДЗ

В рамках этой лекции вы должны сделать один большой проект, в рамках которого нужно реализовать все требования (бот **сразу** будет проверять реализацию **всех требований ДЗ**). Вам нужно переделать наш предыдущий проект таким образом, чтобы теперь все посты хранились не в памяти, а в БД (это достаточно частая задача на реальных проектах – переписать часть проекта на использование новой библиотеки, нового фреймворка, формата или способа хранения данных).

Важно: архивируйте всё, кроме каталога node_modules!

Таблицу постов создавать не надо! Бот её создаст за вас следующим образом:

```
CREATE TABLE posts (
  id BIGSERIAL PRIMARY KEY,
  content TEXT NOT NULL,
  removed BOOLEAN NOT NULL DEFAULT FALSE,
  created TIMESTAMPTZ NOT NULL DEFAULT CURRENT_TIMESTAMP
);
```

ДЗ №0: getAll

Напишите реализацию функции get, которая будет возвращать список всех постов.

Бот будет присылать запрос следующего вида:

http://localhost:9999/posts.getAll

Вы должны: отдавать код 200 и ответ в виде JSON (не забудьте заголовок)

ДЗ №1: getById

Напишите реализацию функции getById, которая будет возвращать пост по его id.

Бот будет присылать запрос следующего вида:

http://localhost:9999/posts.getById?id=1 (не обязательно 1).

Вы должны:

1. Отдавать код 200 и ответ в виде JSON (не забудьте заголовок), если пост найден
2. Отдавать код 400, если пользователь прислал "плохой" запрос (см. следующий слайд)
3. Отдавать код 404, если пост с таким id не найден

ДЗ №1: getById

Поскольку это клиент присылает вам запрос (другое приложение - мы просто тестируем через браузер), то он может прислать плохой запрос:

1. Не указать id
2. Указать id неправильно, например, id=post (т.е. не число, а строка)

Вы должны обрабатывать эти ситуации и возвращать код 400.

ДЗ №2: add

Напишите реализацию функции add, которая будет добавлять пост.

Бот будет присылать запрос следующего вида:

http://localhost:9999/posts.post?content=Fun

Вы должны:

1. Отдавать код 200 и ответ в виде JSON (обновлённый пост), если пост найден
2. Отдавать код 400, если пользователь прислал "плохой" запрос (нет content)

ДЗ №3: edit

Напишите реализацию функции edit, которая будет обновлять пост (т.е. изменять свойство content).

Бот будет присылать запрос следующего вида:

http://localhost:9999/posts.edit?id=1&content=Updated (необязательно 1 и Updated).

Вы должны:

1. Отдавать код 200 и ответ в виде JSON (обновлённый пост), если пост найден
2. Отдавать код 400, если пользователь прислал "плохой" запрос (нет id, id – не число или нет content)
3. Отдавать код 404, если пост с таким id не найден

ДЗ №4: safe delete

Вам также необходимо реализовать safe delete. При этом, если пост помечен как удалённый, то get не должен возвращать его в общем массиве (реализуйте средствами SELECT), а getById, edit, delete должны возвращать 404.

ДЗ №5: restore

Вам нужно сделать функцию posts.restore (аналог Vk wall.restore), которая "восстанавливает" пост (фактически просто меняет removed на false).

Бот будет отправлять такой запрос:

http://localhost:9999/posts.restore?id=1 (не обязательно 1)

Вы должны:

1. Отдавать код 200 и ответ в виде JSON (обновлённый пост), если пост найден и был удалён
2. Отдавать код 400, если пользователь прислал "плохой" запрос (нет id, id – не число или пост не был удалён)
3. Отдавать код 404, если пост с таким id не найден

Спасибо за внимание

alif skills

2023г.