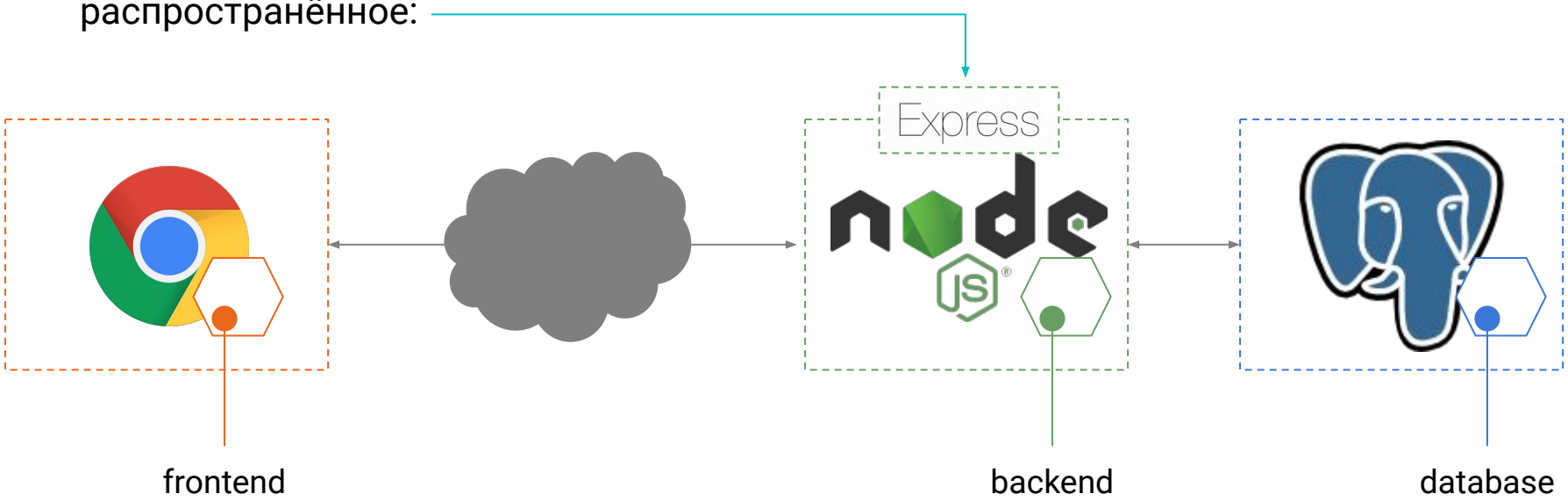
JS Level 3

Node.js

Предисловие

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

Теперь нам осталось заменить самописный сервер на что-то более распространённое:

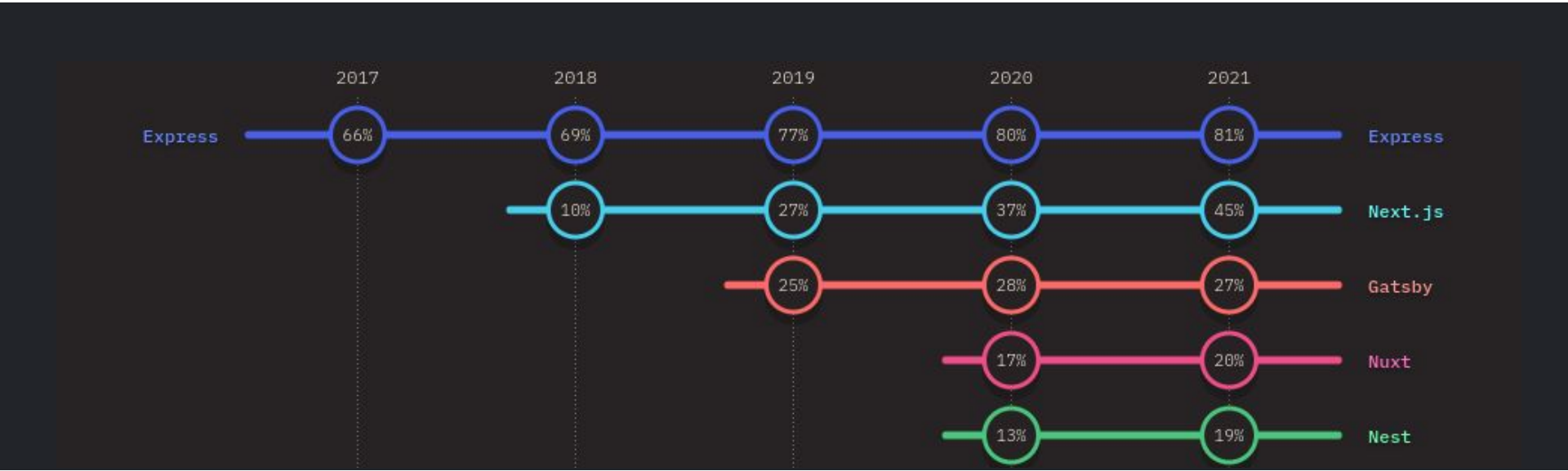


EXPRESS

Express

Писать на чистом Node.js (без использования внешний зависимостей*), конечно, хорошо. Но чаще всего всё-таки используют внешние зависимости и строят свой проект с использованием различных фреймворков, облегчающих написание веб-сервисов (конечно, не все из них можно назвать backend-фреймворками).

% в 2022 году мы уже видели (за 2023 год ещё нет), а вот так выглядит история:



Express

Установим необходимые зависимости:

npm i express cors

Текущая версия Express – 4. 5-ая пока находится в статусе Beta (но мы все ждём её релиза).

REST API

Express позволяет достаточно легко создавать REST API. REST (Representational State Transfer) – архитектурный стиль проектирования веб-сервисов. Чётких границ и правил у этого стиля нет, но есть набор "рекомендаций". Поскольку это именно рекомендации, каждая команда сама решает, в какой мере им соответствовать (а где отходить от них).

В качестве примеров можно посмотреть на описание API от:

- GitHub
- Gmail

REST API

В современной интерпретации REST чаще всего рассматривается следующим образом (конечно же, это можно оспорить):

1. HTTP в качестве транспорта взаимодействия между сервисами (т.е. все данные передаются по HTTP)
2. HTTP методы (GET, POST, DELETE и др.) используются как операции, за которыми закреплёна определённая смысловая нагрузка (получение данных, создание, удаление и т.д.)
3. HTTP статус в качестве результата успешности выполнения вызова
4. группировка URL'ов в иерархические наборы ресурсов (аналог каталогов в файловой системе)
5. stateless – клиент должен передавать все необходимые данные для выполнения запроса

REST API

В качестве формата передачи структурированных данных чаще всего используют JSON, но возможны другие варианты (например, XML).

REST API

Пример: для нашей социальной сети мы можем выделить ресурс /posts (чаще всего /api/posts) и используем HTTP-методы в качестве указания допустимых операций, а URL'ы в качестве указания того, к чему применяется операция: ко всему набору элементов или конкретному элементу:

- GET /posts – получение списка постов
- GET /posts/:id – получение конкретного продукта по id
- POST /posts – создание нового поста
- PUT /posts/:id – обновление поста по id
- PATCH /posts/:id – частичное обновление (например, одно поле)
- DELETE /posts/:id – удаление по id

Ключевое, что нужно запомнить – метод GET никогда не должен изменять сами ресурсы (для этого есть POST/PUT/PATCH/DELETE).

Кроме того, как мы видим, для restore соответствующего глагола не нашлось, поэтому на этом месте в команде возникают споры "как правильно сделать".

REST API

Мы предлагаем вам самостоятельно изучить следующие материалы (это будет частью вашей сегодняшней лекции):

- https://expressjs.com/ru/starter/hello-world.html
- https://expressjs.com/ru/starter/basic-routing.html
- https://expressjs.com/ru/guide/routing.html
- https://expressjs.com/ru/guide/using-middleware.html

JS main.js > ...

```
1   import pg from 'pg';
2   import express from 'express';
3   import cors from 'cors';
4
5   const { Pool } = pg;
6   const dsn = process.env.DSN ?? 'postgres://app:pass@localhost:5432/db';
7   const pool = new Pool({ connectionString: dsn });
8
9   pg.types.setTypeParser(20, Number);
10  pg.types.setTypeParser(1016, o => pg.types.getTypeParser(1016)(o).map(Number));
11
12  const port = process.env.PORT ?? 9999;
13
14  const app = express();
15  app.use(cors());
16  app.use(express.json());
17
18  app.get('/api/posts/:id', async (req, res) => {…
36  });
37  app.post('/api/posts', async (req, res) => {…
50  });
51
52  const server = app.listen(port);
53
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);
```

middleware, позволит отправлять запросы с frontend'а с другим Origin'ом

middleware, позволит принимать на вход JSON (будет в req.body)

```
18  app.get('/api/posts/:id', async (req, res) => {
19    const idParam = req.params.id;
20    // TODO: validate data
21    const id = Number.parseInt(idParam, 10);
22    // TODO: validate data
23    const {rows: [row]} = await pool.query(
24      `
25      SELECT id, content, created FROM posts WHERE id = $1 AND removed = FALSE
26      `,
27      [id]
28    );
29
30    if (typeof row === 'undefined') {
31      res.sendStatus(404);
32      return;
33    }
34
35    res.send(row);
36  });
37  app.post('/api/posts', async (req, res) => {
38    const {content} = req.body;
39    // TODO: validate data
40    const {rows: [row]} = await pool.query(
41      `
42      INSERT INTO posts(content)
43      VALUES ($1)
44      RETURNING id, content, created
45      `,
46      [content]
47    );
48
49    res.send(row);
50  });
```

res.send сам выставит нужные заголовки и отправит всё в JSON

REST API

Объекте req и res, которые предоставляет нам Express, отличаются от тех, что в Node.js, – Express даёт более богатое API:

- req
- res

REST API

На практике (когда приложение разрастается), обычно запросы в БД делают уже не в handler'e, а выделяют для этого отдельный класс, но поскольку приложение у нас небольшое – можем оставить пока так.

REST API

В router'е Express позволяет использовать параметры (:id), которые затем можно извлекать (например, GET /api/posts/10 "положит" в req.params.id значение '10' – именно строку, а не число):

```
18  app.get('/api/posts/:id', async (req, res) => {
19    const idParam = req.params.id;
20    // TODO: validate data
21    const id = Number.parseInt(idParam, 10);
22    // TODO: validate data
23    const {rows: [row]} = await pool.query(
24      `
25      SELECT id, content, created FROM posts WHERE id = $1 AND removed = FALSE
26      `,
27      [id]
28    );
29
30    if (typeof row === 'undefined') {
31      res.sendStatus(404);
32      return;
33    }
34
35    res.send(row);
36  });
```

приводим к числу

REST API

Middleware json позволяет нам "читать" Content-Type: application/json и складывает результаты в req.body, поэтому мы можем позволить себе "не собирать" всё из параметров (как это делали при ручной обработке):

```
37  app.post('/api/posts', async (req, res) => {
38    const {content} = req.body;
39    // TODO: validate data
40    const {rows: [row]} = await pool.query(
41      `
42      INSERT INTO posts(content)
43      VALUES ($1)
44      RETURNING id, content, created
45      `,
46      [content]
47    );
48
49    res.send(row);
50  });
```

REST API

Предоставленной информации должно быть достаточно для построения своего небольшого backend'а (для реализации полноценного приложения).

Стоит отметить, что мы не рассматриваем вопросы обеспечения безопасности и обработки ошибок (в исходных кодах вы увидите чуть усложнённую версию, где каждый handler обёрнут в try-catch).

AXIOS

Axios

Теперь самый интересный вопрос: метод GET мы можем протестировать из адресной строки браузера, а что делать с POST? Неужели писать полноценный Frontend под это?

Конечно же, Node.js позволяет делать нам http-запросы. Для этого можно воспользоваться модулем http или установить сторонний клиент, например, Axios:
npm i -D axios

Обратите внимание: мы устанавливаем Axios с опцией -D, что означает, что в нашем проекте он нужен только для разработки.

Axios

requests > JS add.js > ...

```
1   import axios from 'axios';
2
3   const client = axios.create({
4     baseURL: 'http://localhost:9999/api',
5   });
6
7   const {status, data} = await client.post('/posts', {
8     content: 'test content',
9   });
10
11  if (status !== 200) {
12    throw new Error(`bad status code: ${status}`);
13  }
14
15  console.log(data);
```

Этой информации должно быть достаточно, чтобы протестировать все запросы.

ИТОГИ

Итоги

В этой лекции мы обсудили некоторые ключевые моменты, связанные с работой с Express. Напоминаем ещё раз, что теперь вы должны привыкнуть, что бóльшую часть информации в рамках профессиональной деятельности вам придётся добывать из:

- документации
- обсуждения с коллегами
- собственных экспериментов
- ответов на StackOverflow
- и т.д.

Причём все вышеперечисленные источники могут содержать ошибки.

Не рассчитывайте, что вам всё разложат по полочкам и разъяснят на различных курсах, видео на ютубе и т.д.

ДОМАШНЕЕ ЗАДАНИЕ

Как сдавать ДЗ

В рамках этой лекции вы должны сделать один большой проект, в рамках которого нужно реализовать все требования (бот будет проверять реализацию всех требований ДЗ).

Вам нужно переделать наш предыдущий проект таким образом, чтобы использовался Express вместо нашего самописного сервера.

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 №1: getById

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

Бот будет присылать запрос следующего вида:

GET http://localhost:9999/api/posts/1 (где 1 – это id).

Вы должны:

1. Отдавать код 200 и ответ в виде JSON, 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 (в лекции есть заготовка).

Бот будет присылать запрос следующего вида:

POST http://localhost:9999/api/posts в теле будет JSON вида {"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 №2: edit

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

Бот будет присылать запрос следующего вида:

PUT http://localhost:9999/api/posts/1 в теле будет JSON вида {"content": "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 №3: safe delete

Бот будет присылать запрос вида:

DELETE http://localhost:9999/api/posts/1

Вам необходимо выставлять соответствующее состояние в БД (+ вся предыдущая логика должна сохраниться).

Спасибо за внимание

alif skills

2023г.