Лекция 3: Создание Backend на Node.js и Express. Строим "мозг" нашего приложения

Введение: JavaScript выходит из браузера

Здравствуйте! На предыдущих лекциях мы говорили о процессах и архитектуре. Мы разобрали "языки", на которых общаются веб-приложения (REST, GraphQL, gRPC). Мы спроектировали наш "дом" на бумаге. Сегодня мы начинаем его строить.

Долгое время язык JavaScript был узником одной-единственной среды — веб-браузера. Его задачей было "оживлять" страницы: реагировать на клики, отправлять формы, создавать анимации. Но в 2009 году произошла революция: появился **Node.js**. Это технология, которая взяла невероятно быстрый движок V8 из браузера Google Chrome и позволила ему работать как самостоятельному приложению, получив доступ к файловой системе, сети и другим возможностям операционной системы. JavaScript вышел на свободу.

Сегодня мы погрузимся в мир серверной разработки на JavaScript. Мы узнаем:

- Что такое среда выполнения Node.js и почему это не язык программирования.
- Как управлять "стройматериалами" нашего проекта с помощью менеджера пакетов **npm**.
- Почему никто не строит серверы "с нуля" и какую роль в этом играет веб-фреймворк **Express.js**.

Наша цель на сегодня — пройти путь от пустой папки до полноценного, работающего веб-сервера, который умеет принимать запросы, обрабатывать их и возвращать осмысленные ответы. Мы заложим фундамент для API, которое будем развивать на протяжении всего семестра.

1. Node.js и npm: Фундамент и магазин стройматериалов

Прежде чем начать строительство, нам нужны две вещи: сама строительная площадка с базовыми инструментами (Node.js) и доступ к магазину, где можно заказать любые готовые детали (npm).

1.1. Node.js: Среда выполнения

Важно понимать: **Node.js — это не язык программирования. Это среда выполнения (runtime environment)**. Язык, на котором мы пишем, — это по-прежнему JavaScript. Node.js — это программа, которая умеет "читать" и выполнять наш JavaScript-код за пределами браузера.

Ключевая особенность Node.js, которая сделала его таким популярным для бэкенда, — это **неблокирующий, асинхронный ввод-вывод**, управляемый механизмом под названием **Event Loop (Цикл событий)**. Вспомним нашу аналогию с асинхронным поваром: пока готовится "долгое" блюдо (например, идет запрос к базе данных), "официант" (основной поток Node.js) не стоит и не ждет, а идет обслуживать других клиентов. Это позволяет Node.js очень эффективно обрабатывать тысячи одновременных соединений, что идеально подходит для создания API.

1.2. npm: Node Package Manager

Писать всё с нуля — долго, дорого и бессмысленно. Многие задачи уже были решены до нас тысячами разработчиков по всему миру. Чтобы не "изобретать велосипед", в экосистеме Node.js существует **npm (Node Package Manager)**.

npm — это две вещи в одной:

1. **Огромный онлайн-репозиторий (npmjs.com):** Библиотека из миллионов готовых к использованию модулей (пакетов). Нужен веб-сервер? Есть пакет express. Нужна работа с JWT? Есть jsonwebtoken.
2. **Инструмент командной строки (npm):** Утилита для управления этими пакетами в вашем проекте.

Каждый проект на Node.js начинается с **инициализации** и создания "паспорта" проекта — файла package.json. Он содержит метаданные (название, версия) и, самое главное, список всех внешних зависимостей.

```
# Создаем папку и переходим в нее
mkdir my-api-server && cd my-api-server
# Инициализируем проект, создавая package.json с настройками по умолчанию
npm init -y
```

Теперь мы готовы "закупать стройматериалы".

2. Express.js: Каркас нашего "дома"

Можно ли построить веб-сервер, используя только встроенные модули Node.js (например, модуль http)? Да, можно. Но это похоже на строительство дома из отдельных кирпичей. Вам придется вручную разбирать URL каждого запроса, парсить заголовки и тело, правильно форматировать ответы... Это невероятно трудоемко.

Именно поэтому никто так не делает. Вместо этого используются **веб-фреймворки** — надстройки, которые предоставляют готовый "каркас" и набор инструментов для быстрого создания приложений.

Express.js — это минималистичный, гибкий и самый популярный веб-фреймворк для Node.js. Он не навязывает вам строгую архитектуру, а дает простой и мощный API для решения основных задач: маршрутизации, обработки запросов и работы с middleware.

Давайте установим его:

```
npm install express
```

После выполнения этой команды express появится в списке dependencies в вашем package.json. Теперь мы можем начать строительство.

3. "Hello, World!": Наш первый сервер

Создадим в корне проекта файл index.js и напишем в нем минимальный код для запуска сервера:

```
// 1. Импортируем express
```

```
const express = require('express');

// 2. Создаем экземпляр приложения
const app = express();

// 3. Определяем порт, на котором будет работать сервер
const PORT = process.env.PORT || 3000;

// 4. Описываем наш первый роут (маршрут)
// Когда приходит GET-запрос на корневой URL ('/'),
// выполнить эту функцию-обработчик
app.get('/', (req, res) => {
  res.send('Hello, World! My first Express server is running.');
});

// 5. Запускаем сервер, чтобы он начал "слушать" входящие запросы
app.listen(PORT, () => {
  console.log(`Server is running on http://localhost:${PORT}`);
});
```

Теперь, запустив в терминале команду node index.js и открыв в браузере http://localhost:3000, вы увидите заветное "Hello, World!". Вы только что создали и запустили свой первый веб-сервер.

4. Роутинг и обработка запросов

Роутинг (или маршрутизация) — это процесс определения того, как приложение должно отвечать на запросы к разным URL. В Express это делается с помощью методов, названия которых совпадают с HTTP-методами: app.get(), app.post(), app.put(), app.patch(), app.delete().

Каждый такой метод принимает как минимум два аргумента:

1. **Путь (Path):** Строка, определяющая URL (например, /users, /tasks/:id).
2. **Обработчик (Handler):** Функция, которая будет вызвана при поступлении запроса.

Функция-обработчик, в свою очередь, всегда принимает два главных объекта:

- req (Request): Объект, содержащий всю информацию о входящем запросе от клиента (параметры, заголовки, тело).
- res (Response): Объект, с помощью которого мы формируем и отправляем ответ клиенту.

Давайте создадим простое API. Данные пока будем хранить в массиве:

```
let users = [{ id: 1, name: 'Alice' }, { id: 2, name: 'Bob' }];

// GET /users - получить всех пользователей
app.get('/users', (req, res) => {
  // res.json() автоматически преобразует объект/массив в JSON
  // и устанавливает правильный заголовок Content-Type
  res.json(users);
});

// GET /users/:id - получить одного пользователя по ID
app.get('/users/:id', (req, res) => {
  // :id - это динамический параметр. Его значение доступно в req.params
  const id = parseInt(req.params.id);
```

```
  const user = users.find(u => u.id === id);

  if (user) {
    res.json(user);
  } else {
    // res.status() устанавливает HTTP-статус ответа
    res.status(404).json({ message: 'User not found' });
  }
});
```

Мы также можем получать данные из **тела запроса** (для POST и PUT) и **query-параметров** (для фильтрации). Для чтения JSON-тела нам понадобится специальный "помощник".

5. Middleware: "Швейцарский нож" Express

Middleware (промежуточное ПО) — это сердце и душа Express. Это функции, которые выполняются "посередине", между получением запроса и его финальной обработкой в роуте. Они выстраиваются в конвейер, и каждый запрос проходит через них по очереди.

Middleware может:

- Выполнять любой код.
- Вносить изменения в объекты req и res.
- Завершить цикл "запрос-ответ".
- Передать управление следующему middleware в стеке.

Чтобы научить наш сервер "читать" JSON из тела запроса, мы используем встроенный в Express middleware:

```
// Эта строчка должна идти ДО определения роутов, которые работают с телом
app.use(express.json());
```

Теперь мы можем создать роут для добавления нового пользователя:

```
// POST /users - создать нового пользователя
app.post('/users', (req, res) => {
  // Благодаря express.json(), данные доступны в req.body
  const newUser = {
    id: Date.now(), // Простой способ сгенерировать ID
    name: req.body.name,
  };
  users.push(newUser);
  res.status(201).json(newUser);
});
```

Мы можем писать и свои собственные middleware, например, для логирования каждого запроса или для проверки аутентификации, что мы и будем делать в следующих лекциях.

6. Структура и надежность: Готовим сервер к реальной жизни

Когда эндпоинтов становится много, хранить их все в одном файле index.js — плохая практика. Код нужно структурировать. Для этого в Express используется объект Router, который позволяет выносить группы связанных роутов в отдельные файлы-модули.

routes/users.js:

```
const express = require('express');
const router = express.Router();

// ... здесь вся логика для /users, /users/:id ...
// Вместо app.get() пишем router.get()

module.exports = router;
```

index.js:

```
const usersRouter = require('./routes/users');

// ...
// 'Монтируем' роутер на определенный путь
app.use('/users', usersRouter);
// ...
```

Кроме того, в профессиональной разработке конфигурационные данные (порт, пароли от баз данных) никогда не "хардкодят" в коде. Их выносят в **переменные окружения** и загружают в приложение с помощью файла .env и библиотеки dotenv.

И, наконец, мы никогда не должны доверять данным от клиента. Любые данные из req.body или req.params должны проходить **валидацию** с помощью специальных библиотек (express-validator, joi), прежде чем попасть в нашу бизнес-логику.

Заключение

Сегодня мы сделали огромный шаг: от теоретических знаний о JavaScript мы перешли к созданию настоящего, работающего веб-сервера. Мы познакомились с Node.js, npm и Express — тремя столпами современной backend-разработки. Мы научились создавать роуты, обрабатывать запросы и структурировать наше приложение.

Этот простой сервер — наш "холст". В следующих лекциях мы будем добавлять на него новые "краски": подключим настоящую базу данных, реализуем систему аутентификации, покроем код тестами и снабдим его профессиональной документацией. Мы начали строить наш "дом", и теперь готовы возводить стены и крышу.