Лекция 17
Основы Express.js для разработки бэкенда

Содержание:

1. Введение в Express.js.
2. Маршрутизация в Express.js

1. Введение в Express.js.

1.1 Познакомьтесь с Express.js, его назначением и тем, как он вписывается в экосистему Node.js.

Express.js — это легковесный веб-фреймворк для Node.js, используемый для создания веб-серверов, API и бэкенд-приложений. Он упрощает разработку на стороне сервера, предоставляя инструменты для маршрутизации, промежуточного ПО, обработки запросов и управления ответами.

Без Express.js разработчикам пришлось бы работать напрямую с HTTP-модулями Node.js, что часто требует написания большего количества повторяющегося кода и ручной настройки.

Пример базового сервера Express.js:

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

const app = express();

app.get('/', (req, res) => {
  res.send('Hello from Express!');
});

app.listen(3000, () => {
  console.log('Server running on port 3000');
});
```

В этом примере:

- **express()** - создает экземпляр приложения;
- **app.get()** - определяет маршрут;
- **req** - представляет собой объект запроса;
- **res** - представляет собой объект ответа;
- **app.listen()** - запускает сервер.

Почему разработчики используют Express.js

Express.js популярен благодаря своей простоте, гибкости и хорошей совместимости как с небольшими, так и с крупными приложениями.

Разработчики обычно используют Express.js для:

- REST API;
- Сервисы бэкэнда;
- Полнофункциональные веб-приложения;
- Системы аутентификации;

– Real-time applications - приложения, работающие в режиме реального времени.

Express.js также легко интегрируется с базами данных, системами шаблонизации и фронтенд-фреймворками.

Express.js и Node.js

Express.js работает поверх Node.js и расширяет его функциональность, обеспечивая более чистый и организованный процесс разработки.

Node.js предоставляет среду выполнения, а Express.js — инструменты для более эффективной обработки маршрутов, промежуточного ПО и серверной логики.

1.2 Структура приложения Express.js

Для начала работы с Express.js обычно создается минимальное приложение в одном файле с именем `.js` app.js. Этот файл является точкой входа в ваш проект Express.js. Вот как выглядит минимальное приложение Express.js:

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

app.get('/', (req, res) => {
  res.send('Hello, Express!');
});

app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

Каждая часть этого файла имеет определенное назначение.

Первая строка

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

-импортирует модуль **Express.js**, чтобы вы могли использовать его возможности в своем приложении.

Следующая строка

```
const app = express();
```

-создает экземпляр приложения **Express** . Этот арробъект используется для определения маршрутов и настройки вашего сервера.

Строка -

```
app.get('/', (req, res) => {
  res.send('Hello, Express!');
});
```

определяет **обработчик маршрута** для HTTP GET-запросов к корневому URL (/). Когда кто-то посещает домашнюю страницу вашего сервера, эта функция запускается и отправляет обратно текст **"Hello, Express!"**.

Наконец строка –

```
app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

- запускается сервер и дается команда прослушивать входящие соединения на порту 3000. После запуска сервера выполняется функция обратного вызова, которая выводит сообщение в консоль.

Приложения Express.js обычно организуются вокруг основного **app.js** файла. По мере роста приложения вы можете разделить код на несколько файлов, но этот **app.js** файл всегда служит главным центром, который создает экземпляр **Express**, настраивает промежуточное ПО, определяет маршруты и запускает сервер.

1.3. Жизненный цикл и поток запросов Express.js

Когда клиент отправляет запрос приложению Express.js, запрос проходит несколько этапов, прежде чем будет получен ответ. Этот процесс называется жизненным циклом запроса или потоком запроса.

Понимание этого процесса важно, поскольку приложения Express.js построены на обработке запросов и ответов.

Как работает поток запросов

Жизненный цикл обычно проходит следующие этапы:

1. Клиент отправляет HTTP-запрос на сервер;
2. Express.js получает запрос;
3. Функции промежуточного программного обеспечения обрабатывают запрос;
4. Запускается обработчик соответствующего маршрута;
5. Ответ отправляется клиенту.

Пример:

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

// Middleware
app.use((req, res, next) => {
  console.log('Request received');
  next();
});

// Route handler
app.get('/', (req, res) => {
  res.send('Home Page');
});

app.listen(3000);
```

В этом примере сначала запускается промежуточное ПО и выводит сообщение в лог. Функция next() передает управление следующему промежуточному ПО или обработчику маршрута. Затем обработчик маршрута отправляет окончательный ответ.

Объекты запроса и ответа

Express.js предоставляет два важных объекта:

– req - Содержит информацию о входящем запросе;

– res - Используется для отправки ответа клиенту.

```
app.get('/users', (req, res) => {
  res.send('Users route');
});
```

Здесь:

– **req** - содержит данные запроса, такие как заголовки, параметры и строки запроса;

– **res.send()** - Отправляет данные обратно в браузер или API-клиент.

Поток промежуточного программного обеспечения

Функции промежуточного программного обеспечения выполняются в порядке их регистрации. Каждое промежуточное программное обеспечение может:

– Измените запрос;

– Выполнить логику;

– Завершите ответ;

– Передайте управление с помощью **next()**.

Если промежуточное программное обеспечение не вызывает функцию **next()** или не отправляет ответ, запрос может остаться незавершенным.

2. Маршрутизация в Express.js

Изучите, как Express.js обрабатывает маршрутизацию и управляет различными HTTP-методами и конечными точками.

2.1 Определение маршрутов

При создании веб-приложений с использованием Express.js вы определяете **маршруты**, которые определяют, как ваше приложение реагирует на запросы клиентов к определенным конечным точкам и методам HTTP. **Маршрут** по сути представляет собой комбинацию пути URL и метода HTTP (например, GET или POST), а также **функции-обработчика**, которая выполняется при совпадении запроса.

Для определения маршрута в Express используются такие методы, как **app.get()** и **app.post()**. Эти методы принимают два основных аргумента: путь и функцию обработчика. Путь — это конечная точка, на которую вы хотите ответить, а функция обработчика получает объекты запроса и ответа.

Вот как можно определить простой GET-запрос для главной страницы и POST-запрос для точки входа в систему:

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

// Define a GET route for the root path "/"
app.get('/', (req, res) => {
  res.send('Welcome to the homepage!');
});

// Define a POST route for "/login"
app.post('/login', (req, res) => {
  res.send('Login form submitted!');
});
```

В этом примере:

– Когда клиент отправляет **GET**- запрос к корневому **пути** /, Express сопоставляет его с первым маршрутом и выполняет его обработчик, отправляя "Welcome to the homepage!" в качестве ответа;

– Когда клиент отправляет **POST**- запрос на адрес **/login**, Express сопоставляет его со вторым маршрутом и отправляет обратно "Login form submitted!".

Каждый маршрут в Express состоит из трех основных частей:

– Методы HTTP: например , **GET** , **POST** , **PUT** , **DELETE** ;

– Путь: конкретная строка конечной точки, например **/, /login,** или **/users** ;

– Функция обработчика: функция обратного вызова, которая получает объекты запроса (**req**) и ответа (**res**) и отправляет ответ.

Express сопоставляет входящие запросы с вашими маршрутами, проверяя как HTTP-метод, так и путь. Если оба параметра совпадают, запускается соответствующий обработчик. Если ни один маршрут не соответствует действительности, Express продолжит поиск или вернет ошибку 404, если маршрут не найден.

Вы также можете определять маршруты для других методов HTTP, таких как **app.put()** и **app.delete()**. Это упрощает создание RESTful API или веб-приложений, которые реагируют на различные действия.

2.2 Параметры маршрута и строки запроса

При разработке веб-приложений с использованием Express.js часто возникает необходимость обработки URL-адресов, содержащих динамические значения или принимающих необязательные входные данные от клиента. Express.js предоставляет для этого два основных механизма: **параметры маршрута** и **строки запроса**.

Параметр **маршрута** — это переменная часть пути URL. Он определяется в пути маршрута с помощью двоеточия **(:)** с последующим именем параметра. Например, вы можете определить маршрут, подобный этому, **/users/:id** для обработки запросов от разных пользователей на основе их уникального идентификатора. Когда запрос соответствует этому шаблону, Express.js делает значение доступным в **req.params** объекте.

Строка **запроса** состоит из пар ключ-значение, добавляемых в конец URL-адреса после вопросительного знака **(?)**. Строки запроса обычно используются для предоставления серверу необязательной информации, такой как фильтры поиска или настройки пагинации. В Express.js значения строк запроса доступны в объекте **req.query**.

Вот пример, демонстрирующий работу параметров маршрута и строк запроса:

```
// Example route with a route parameter
app.get('/users/:id', (req, res) => {
  // Access route parameter
  const userId = req.params.id;
  // Access query string parameters
  const showDetails = req.query.details;
  res.send(`User ID: ${userId}, Show details: ${showDetails}`);
```

```
});
```

В этом примере, когда клиент отправляет GET-запрос на адрес **/users/42?details=true**, Express.js извлекает параметр маршрута id со значением "42" и параметр строки запроса **details** со значением "true". Обработчик использует **req.params.id** для доступа к параметру маршрута и **req.query.details** для доступа к значению строки запроса.

Чтобы извлечь и использовать параметры маршрута и строки запроса в обработчиках запросов, выполните следующие действия:

1. Для каждого динамического сегмента, который вы хотите захватить, укажите путь маршрута с двоеточием и именем параметра;
2. Используйте этот **req.params** объект для доступа к параметрам маршрута внутри вашей функции-обработчика;
3. Используйте этот **req.query** объект для доступа к параметрам строки запроса, которые всегда являются строками или неопределенными, если не указаны;
4. При необходимости комбинируйте оба механизма для создания гибких и динамичных конечных точек.

Например, если вы определите маршрут как **/products/:productId**, и клиент запросит **/products/123?color=red&size=large**, вы найдете **req.params.productId === "123"**, req.query.color === "red", и req.query.size === "large". Это упрощает написание обработчиков, которые могут получать ресурсы по ID, а также фильтровать или изменять ответ на основе необязательного ввода клиента.

2.3 Организация маршрутов с помощью маршрутизатора

По мере роста вашего приложения Express.js управление всеми маршрутами в одном файле быстро становится неудобным. Чтобы поддерживать чистоту и удобство сопровождения кода, вы можете использовать **Express Router** для модульной организации маршрутов. Router позволяет разделить связанные маршруты на отдельные модули, каждый из которых обрабатывает определенную часть вашего приложения. Такой подход не только улучшает читаемость, но и упрощает повторное использование и тестирование логики маршрутов.

Предположим, вы хотите организовать все маршруты, связанные с пользователями, в отдельном файле. Вы создадите новый файл, например users.js, чтобы определить эти маршруты с помощью экземпляра Express Router, а затем импортируете этот маршрутизатор в основной файл вашего приложения (app.js). Вот как можно структурировать ваш код:

users.js (router module):

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

// GET /users/
router.get('/', (req, res) => {
  res.send('List of users');
});
```

```
// GET /users/:id
router.get('/:id', (req, res) => {
  res.send(`User with ID: ${req.params.id}`);
});

module.exports = router;
```

app.js (main application file):

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

app.use('/users', usersRouter);

app.listen(3000, () => {
  console.log('Server running on port 3000');
});
```

В этой конфигурации users.js экспортируется экземпляр маршрутизатора, определяющий все маршруты, связанные с пользователем. Основной файл приложения импортирует этот маршрутизатор и монтирует его по указанному **/users** пути с помощью команды **app.use('/users', usersRouter)**`. В результате запросы к **/users** или **/users/:id** обрабатываются маршрутизатором, определенным в файле ` **users.js**. Такой модульный подход позволяет упорядочить логику маршрутизации и упрощает масштабирование приложения.

Express Router — это мини-приложение Express, которое можно подключить к основному приложению. Оно позволяет организовывать маршруты в отдельные модули на основе функций или ресурсов, таких как пользователи, товары или заказы. Основные преимущества модульной маршрутизации:

– Улучшена организация кода за счет группировки связанных маршрутов;

– Упрощенное обслуживание и масштабируемость по мере роста вашего приложения;

– Упрощенное тестирование и повторное использование логики маршрутизации;

– Четкое разделение задач между различными частями вашего приложения.

Вы также можете использовать **промежуточное ПО на уровне маршрутизатора (router-level middleware)** с Express Router. Это означает, что вы можете добавлять функции промежуточного ПО, которые применяются только к маршрутам в рамках определенного модуля маршрутизатора, например, аутентификацию или логирование для пользовательских маршрутов. Чтобы использовать промежуточное ПО на уровне маршрутизатора, просто добавьте его в свой экземпляр маршрутизатора перед обработчиками маршрутов:

```
router.use((req, res, next) => {
  console.log('Request made to user routes');
  next();
});
```

Использование Express Router и промежуточного программного обеспечения на уровне маршрутизатора позволяет создать более структурированную и поддерживаемую кодовую базу, особенно по мере усложнения вашего приложения.

Вопросы

1) Что такое Express.js?
2) Как выглядит минимальное приложение Express.js?
3) Жизненный цикл и поток запросов Express.js. Как работает поток запросов?
4) Объекты запроса и ответа.
5) Как Express.js обрабатывает маршрутизацию?
6) Вот как можно определить простой GET и POST-запросы?
7) Параметры маршрута и строки запроса
8) Организация маршрутов с помощью маршрутизатора

Ссылка
https://codefinity.com/courses/v2/fc82fa76-7239-478a-bd4e-99ee5e48c1c9/6a53790c-f660-4624-934a-1bf1c5b39341/6e00164f-b968-41ca-bfb4-99f4f909dad4