STEP ACADEMY

Занятие № 12

Backend архитектура

Введение

Проекты Backend представляют собой основу для функционирования веб-приложений. Правильная архитектура является ключевым аспектом для создания надежного, масштабируемого и удобного в поддержке приложения.

Основной принцип эффективной архитектуры - разделение на слои.

Верхние архитектурные слои

1. Маршрутизация (Routing Layer)

- Обработка запросов клиента
- Определение путей и HTTP-методов
- Направление запросов контроллерам

2. Контроллеры (Controller Layer)

- Обработка бизнес-логики
- Взаимодействие с сервисами
- Формирование ответов для клиента

Нижние архитектурные слои

3. Сервисы (Service Layer)

- Реализация бизнес-логики
- Взаимодействие с базой данных
- Управление данными и состоянием приложения

4. Хранилище данных (Data Storage Layer)

- Взаимодействие с базой данных
- Использование ORM или ODM для удобной работы с данными

1. Слой маршрутизации

Слой маршрутизации - это часть приложения, которая определяет, как обрабатывать входящие HTTP-запросы, какие пути запросов существуют и какие обработчики вызывать для каждого пути. Он является основным механизмом определения, как клиентские запросы должны быть обработаны и каким образом должен быть сформирован ответ.

1.1. Определение маршрутов

В Express.js определение маршрутов осуществляется с помощью методов, таких как `app.get()`, `app.post()`, `app.put()` и других. Каждый метод принимает два параметра: путь запроса и обработчик, который будет вызываться при совпадении пути запроса.

```
// Пример определения маршрута GET
app.get('/users', (req, res) => {
  // Обработка запроса для пути '/users'
  res.send('Список пользователей');
});
```

1.2. Параметры маршрутов

Маршруты могут содержать параметры, которые позволяют передавать переменные значения в пути запроса. Параметры задаются в виде двоеточия (`:`) с именем параметра.

```
// Пример маршрута с параметром
app.get('/users/:id', (req, res) => {
  const userId = req.params.id;
  // Использование значения параметра
  res.send(`Получен пользователь с ID ${userId}`);
});
```

1.3. Middleware в маршрутизации

Middleware - это функции, которые выполняются перед или после обработки маршрута. Они позволяют добавлять дополнительную логику и функциональность к обработке запросов.

```
// Пример использования Middleware перед обработкой маршрута
app.get('/users', checkAuth, (req, res) => {

  res.send('Список пользователей');
});
```

1.4. Пример Middleware-функции

```
const checkAuth = (req, res, next) => {
  // Проверка авторизации
  if (req.isAuthenticated()) {
    // Пользователь авторизован, передаём управление
    // следующему Middleware или обработчику маршрута
    next();
  } else {
    // Пользователь не авторизован, отправляем ошибку
    res.status(401).send('Ошибка авторизации');
  }
};
```

1.5. Группировка маршрутов

Express.js предоставляет возможность группировать маршруты, что упрощает организацию и поддержку кода. Группировка позволяет определить общий префикс для нескольких маршрутов.

Это позволяет разделить слой маршрутизации на модули

1.6. Пример группировки маршрутов

```
const usersRouter = express.Router();

// Маршруты, относящиеся к пользователям
usersRouter.get('/', (req, res) => {
  res.send('Список пользователей');
});

usersRouter.get('/:id', (req, res) => {
  const userId = req.params.id;
  res.send(`Получен пользователь с ID ${userId}`);
});

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

2. Слой контроллеров

Слой контроллеров - это часть архитектуры проекта Backend, который отвечает за обработку входящих запросов от клиентов. Контроллеры обрабатывают запросы, извлекают необходимые данные, применяют бизнес-логику и возвращают соответствующий ответ.

2.1. Организация контроллеров

Контроллеры обычно организованы в виде отдельных модулей или файлов. Каждый контроллер отвечает за обработку конкретного типа запросов или ресурса. Организация контроллеров позволяет логически разделить функциональность приложения и повысить его модульность.

2.2. Пример модульности контроллеров

```
// usersController.js
const getUsers = (req, res) => {
  // Логика получения списка пользователей
  res.send('Список пользователей');
};

// postsController.js
const getPosts = (req, res) => {
  // Логика получения списка постов
  res.send('Список постов');
};
```

2.3. Маршрутизация на контроллеры

Слой контроллеров связывается с слоем маршрутизации для определения, какой контроллер будет вызываться при обработке конкретного маршрута. Маршруты указываются в слое маршрутизации, а каждый маршрут связывается с соответствующим контроллером.

```
// app.js
app.get('/users', usersController.getUsers);
app.get('/posts', postsController.getPosts);
```

2.4. Применение бизнес-логики

Контроллеры выполняют бизнес-логику приложения, такую как валидация данных, вызов сервисов и другие операции. Контроллеры могут получать данные из запроса, обрабатывать их, вызывать соответствующие сервисы и формировать ответ.

```
// usersController.js
const createUser = (req, res) => {
  const { name, email } = req.body;

  // Валидация данных
  if (!name || !email) {
    return res.status(400).send('Имя и email обязательны');
  }

  // Вызов сервиса для создания пользователя
  const user = userService.createUser(name, email);

  // Отправка ответа
  res.send(user);
};
```

2.5. Обработка ошибок

Контроллеры также отвечают за обработку ошибок и отправку соответствующих ответов клиенту. При возникновении ошибки контроллер может отправить код ошибки и сообщение об ошибке клиенту.

```
// usersController.js
const getUserById = (req, res) => {
  const userId = req.params.id;

  // Поиск пользователя по ID
  const user = userService.getUserById(userId);

  // Проверка наличия пользователя
  if (!user) {
    return res.status(404).send('Пользователь не найден');
  }

  // Отправка пользователя
  res.send(user);
};
```

3. Слой сервисов

Слой сервисов обеспечивает разделение ответственностей и повышает модульность приложения. Он отвечает за бизнес-логику и управление данными, тогда как слой контроллеров отвечает за обработку запросов и взаимодействие с клиентами. Разделение ответственностей помогает создать более чистый и управляемый код.

Каждый сервис отвечает за определенную функциональность и содержит набор методов, которые реализуют необходимую бизнес-логику.

3.1. Пример сервиса

```
// userService.js
export const createUser = (userData) => {
  // Логика создания пользователя
};

export const getUserById = (userId) => {
  // Логика получения пользователя по ID
};
```

3.2. Пример сервиса

```
// postService.js
export const createPost = (postData) => {
  // Логика создания поста
};

export const getPosts = () => {
  // Логика получения списка постов
};
```

3.3. Использование сервисов

Слой контроллеров использует сервисы для выполнения бизнес-логики и обработки данных. Контроллеры вызывают соответствующие методы сервисов для выполнения нужных операций. Это позволяет логически разделить функциональность и упрощает поддержку и тестирование кода.

3.4. Пример использования сервисов

```
// usersController.js
import * as userService from '../services/userService.js';

export const createUser = (req, res) => {
  const userData = req.body;

  // Вызов сервиса для создания пользователя
  const user = userService.createUser(userData);

  // Отправка ответа
  res.send(user);
};
```

3.5. Польза от слоя сервисов

Слой сервисов в архитектуре Backend приносит ряд преимуществ:

- Чистота кода и разделение ответственностей.
- Модульность и повторное использование кода.
- Упрощение тестирования и отладки.
- Легкость внесения изменений и поддержки проекта.

4. Слой Data Storage

Слой Data Storage представляет собой компонент, отвечающий за управление и хранение данных приложения. Он обеспечивает взаимодействие с базой данных, файловой системой или другими механизмами хранения данных.

4.1. Роль слоя Data Storage

Data Storage выполняет следующие функции:

- Управление подключением к базе данных.
- Выполнение операций чтения, записи, обновления и удаления данных.
- Моделирование данных и их отображение на объекты в коде.
- Обработка запросов к базе данных и преобразование данных для передачи другим слоям приложения.

```
// users.js - класс для работы с таблицей "users"
export class UsersDataStorage {
  constructor(db) {
    this.db = db;
  }

  getAllUsers(callback) {
    this.db.all(`SELECT * FROM users`, (err, rows) => {
      if (err) console.error(err);
      else callback(rows);
    });
  }

  addUser(user) {
    this.db.run(`INSERT INTO users (id, name) VALUES (?, ?)`,
                [user.id, user.name]);
  }
}
```

```
// products.js - класс для работы с таблицей "products"
export class ProductsDataStorage {
  constructor(db) {
    this.db = db;
  }

  getAllProducts(callback) {
    this.db.all('SELECT * FROM products', (err, rows) => {
      if (err) console.error(err);
      else callback(rows);
    });
  }

  addProduct(product) {
    this.db.run('INSERT INTO products (id, name, price) VALUES (?, ?, ?)',
                [product.id, product.name, product.price]);
  }
}
```

4.2. Композиция классов Data Storage

```
import { UsersDataStorage } from './users.js';
import { ProductsDataStorage } from './products.js';
const db = new sqlite3.Database('./database.sdb');

// Создание экземпляров классов Data Storage
const userStorage = new UsersDataStorage(db);
const productStorage = new ProductsDataStorage(db);

userStorage.getAllUsers((users) => {
  console.log(users);
});

productStorage.addProduct({ id: 1, name: 'Product A', price: 10.99 });
db.close();
```

Конец