**Вопросы к экзамену по дисциплине**
**«Разработка приложений для Интернет»**

1. JavaScript. Набор символов. Комментарии, литералы, идентификаторы, зарезервированные слова.
2. JavaScript. Основные типы данных.
3. JavaScript. Базовые операторы. Арифметические выражения.
4. JavaScript. Условные операторы. Логические выражения.
5. JavaScript. Операторы циклов.
6. JavaScript. Функции. Стрелочные функции. замыкание.
7. JavaScript. Объекты. Объектный литерал. Функция-конструктор. Прототипное наследование.
8. JavaScript. ES5. Классы. Геттеры, сеттеры. Наследование.
9. JavaScript. Встроенный объект Object. Основные свойства.
10. JavaScript. Встроенный объект Math. Основные свойства.
11. JavaScript. Встроенный объект Date. Основные свойства.
12. JavaScript. Массивы. Встроенный объект Array. Основные свойства.
13. JavaScript. Строки. Встроенный объект String. Основные свойства.
14. JavaScript. Определение регулярных выражений. Встроенный объект RegExp. Основные свойства объекта String для поиска и замены по шаблону.
15. JavaScript. Встраивание JavaScript-кода в разметку HTML. Выполнение сценария JavaScript.
16. JavaScript. Объект Window. Таймеры.
17. JavaScript. Объект Window. История посещений. Информация о браузере и об экране.
18. JavaScript. Объект Window. Работа с окнами.
19. JavaScript. Объектная модель документа (DOM).
20. JavaScript. DOM. Доступ к элементам документа.
21. JavaScript. DOM. Структура документа и навигация.
22. JavaScript. DOM. Работа с контентом элемента. Управление стилями элемента.
23. JavaScript. DOM. Обработка событий.
24. JavaScript. DOM. Обработка форм.
25. JavaScript. Сериализация и десериализация объектов. Формат JSON.
26. JavaScript. Работа с протоколом HTTP.
27. JavaScript. Сохранение данных на стороне клиента.
28. JavaScript. Библиотека jQuery. Доступ к элементам документа. Навигация.
29. JavaScript. Библиотека jQuery. Обработка событий.
30. JavaScript. Библиотека jQuery. Анимационные эффекты.
31. JavaScript. Библиотека jQuery. Реализация AJAX.
32. JavaScript. Библиотека React. Настройка окружения.
33. JavaScript. Библиотека React. Функциональные и классовые компоненты.
34. JavaScript. Библиотека React. Передача данных в компонент. Props.
35. JavaScript. Библиотека React. Состояние компонента.
36. JavaScript. Библиотека React. Хуки. UseState, UseEffect, UseContext.

37. JavaScript. Библиотека React. Маршрутизация.
38. NodeJS. Обзор платформы. Окружение.
39. NodeJS. Различия между платформой Node.js и браузером.
40. NodeJS. Понятие модуля. Экспорт данных.
41. NodeJS. NPM. Файлы package.json, package-lock.json.
42. NodeJS. Утилита NPX.
43. NodeJS. Работа в консольном режиме. Аргументы командной строки.
44. NodeJS. Модуль readline.
45. NodeJS. Цикл событий, стек вызовов, таймеры
46. NodeJS. Работа с файловой системой. Модуль fs.
47. NodeJS. Асинхронное программирование. Промисы (Promises).
48. NodeJS. Модуль http/https. Создание сервера.
49. NodeJS. Статические сайт.
50. NodeJS. Фреймворк Express.
51. NodeJS. Фреймворк Express. Конвейер обработки запроса и middleware.
52. NodeJS. Фреймворк Express. Статические файлы.
53. NodeJS. Фреймворк Express. Маршрутизация.
54. NodeJS. Фреймворк Express. Разработка REST API.
55. NodeJS. Тестирование приложения.
56. NodeJS. Фреймворк Express. Взаимодействие с базами данных.
57. NodeJS. Фреймворк Express. Взаимодействие с базами данных с использование ORM-библиотек.
58. NodeJS. Авторизация и аутентификация.

**1. JavaScript. Набор символов. Комментарии, литералы, идентификаторы, зарезервированные слова.**

Набор символов в JavaScript включает в себя буквы, цифры, знаки препинания и специальные символы.

Комментарии — это текст, который игнорируется интерпретатором JavaScript. Комментарии могут быть однострочными (начинаются с //) или многострочными (начинаются с /* и заканчиваются */).

Литералы — это значения, которые напрямую записываются в коде. Например, числовой литерал 42 или строковый литерал "Hello, world!".

Идентификаторы — это имена переменных, функций и других элементов программы. Идентификаторы могут содержать буквы, цифры и символы $ и _, но не могут начинаться с цифры.

Зарезервированные слова — это слова, которые имеют специальное значение в JavaScript и не могут использоваться как идентификаторы. Например, if, else, for, while, function и т.д.

**2. JavaScript. Основные типы данных.**

JavaScript имеет 7 основных типов данных:

1. Числа (Numbers) — это числовые значения, которые могут быть целыми или десятичными.

2. Строки (Strings) — это последовательность символов, заключенных в кавычки. Они могут содержать буквы, цифры, знаки препинания и другие символы.

3. Булевые значения (Booleans) — это логические значения true или false.

4. null — это специальное значение, которое означает отсутствие значения.

5. undefined — это специальное значение, которое означает, что переменная не имеет значения.

6. Объекты (Objects) — это коллекция ключ-значение, где ключом может быть любая строка, а значением может быть любой тип данных.

7. Массивы (Arrays) — это упорядоченная коллекция элементов, которые могут быть любого типа данных. Элементы массива нумеруются, начиная с 0.

**3. JavaScript. Базовые операторы. Арифметические выражения.**

Любая программа на JavaScript начинается с объявления переменных, которые могут содержать числа, строки, логические значения и другие типы данных. Для работы с этими переменными используются базовые операторы, такие как арифметические выражения.

Арифметические выражения включают в себя операции сложения (+), вычитания (-), умножения (*) и деления (/). Например, для сложения двух чисел можно использовать следующий код:

```
let a = 5;
let b = 3;
let c = a + b;
console.log(c); // выведет 8
```

Также доступны операции инкремента (++) и декремента (--), которые увеличивают или уменьшают значение переменной на 1. Например:

```
let a = 5;
a++; // увеличиваем на 1
console.log(a); // выведет 6
```

Для выполнения более сложных операций можно использовать скобки для задания порядка выполнения операций. Например:

```
let a = 5;
let b = 3;
let c = 2;
let d = (a + b) * c;
console.log(d); // выведет 16
```

Также доступны операции сравнения (>, <, >=, <=, ==, !=), которые возвращают логическое значение true или false в зависимости от результата сравнения. Например:

```
let a = 5;
let b = 3;
console.log(a > b); // выведет true
console.log(a == b); // выведет false
```

**4. JavaScript. Условные операторы. Логические выражения.**

В JavaScript доступны условные операторы, которые позволяют выполнять различные действия в зависимости от выполнения определенных условий. Один из наиболее распространенных условных операторов - if-else.

Оператор if позволяет выполнить определенный блок кода, если заданное условие истинно. Например:

```
let a = 5;
if (a > 3) {
  console.log("a больше 3");
}
```

Оператор else позволяет выполнить другой блок кода, если условие не выполняется.

Также доступны операторы else if и switch, которые позволяют задавать несколько условий для выполнения различных действий.

Для проверки логических выражений в JavaScript используются логические операторы (&&, ||, !). Например:

```
let a = 5;
let b = 3;
if (a > 3 && b < 5) {
  console.log("a больше 3 и b меньше 5");
}
```

Логический оператор || позволяет задавать условия, которые будут выполнены, если хотя бы одно из них истинно.

Логический оператор ! позволяет инвертировать логическое значение.

**5. JavaScript. Операторы циклов.**

JavaScript также поддерживает операторы циклов, которые позволяют выполнять определенный блок кода несколько раз в зависимости от заданных условий. Один из наиболее распространенных операторов цикла - for.

Оператор for позволяет выполнить определенный блок кода заданное количество раз.

Оператор while позволяет выполнять блок кода до тех пор, пока заданное условие истинно.

Оператор do-while позволяет выполнить блок кода хотя бы один раз, а затем продолжать выполнение до тех пор, пока заданное условие истинно.

Также доступны операторы break и continue, которые позволяют прерывать или пропускать выполнение цикла в определенных условиях.

**6. JavaScript. Функции. Стрелочные функции. замыкание.**

Функции в JavaScript — это блоки кода, которые могут быть вызваны из других частей программы. Они могут принимать аргументы и возвращать значения. Функции могут быть определены как отдельные блоки кода, а также использоваться встроенные функции, такие как console.log() или alert().

Стрелочные функции — это новый способ определения функций в ES6. Они имеют более короткий синтаксис и не имеют своего контекста this.

Пример определения стрелочной функции:

```
const sum = (a, b) => a + b;
console.log(sum(2, 3)); // выведет 5
```

Замыкание — это особенность JavaScript, которая позволяет функции сохранять ссылку на переменные из родительской области видимости, даже после того, как родительская функция завершила свое выполнение.

Пример использования замыкания:

```
function outer() {
  const name = "John";
  function inner() {
    console.log(name);
  }
  return inner;
}
const func = outer();
func(); // выведет "John"
```

**7. JavaScript. Объекты. Объектный литерал. Функция-конструктор. Прототипное наследование.**

Объекты в JavaScript — это коллекции свойств и методов, которые могут быть использованы для представления реальных объектов или абстракций.

Объектный литерал — это способ создания объектов в JavaScript путем определения свойств и методов в фигурных скобках.

Пример создания объекта с помощью объектного литерала:

```
const person = {
  name: "John",
  age: 30,
  sayHello: function() {
    console.log("Hello!");
  }
};
console.log(person.name); // выведет "John"
person.sayHello(); // вызов метода объекта
```

Функция-конструктор — это способ создания объектов путем определения функции, которая будет использоваться для создания новых экземпляров объектов.

Пример создания функции-конструктора:

```
function Person(name, age) {
  this.name = name;
  this.age = age;
  this.sayHello = function() {
    console.log("Hello!");
  }
}
const person = new Person("John", 30);
console.log(person.name); // выведет "John"
person.sayHello(); // вызов метода объекта
```

Прототипное наследование - это особенность JavaScript, которая позволяет объектам наследовать свойства и методы от других объектов, называемых прототипами.

Пример использования прототипного наследования:

```
function Animal(name) {
  this.name = name;
}
Animal.prototype.sayHello = function() {
  console.log("Hello!");
}
function Dog(name) {
  Animal.call(this, name);
}
Dog.prototype = Object.create(Animal.prototype);
const dog = new Dog("Buddy");
dog.sayHello(); // вызов метода унаследованного объекта
```

**8. JavaScript. ES5. Классы. Геттеры, сеттеры. Наследование.**

В ES5 классы создавались с помощью функций-конструкторов и прототипного наследования. Однако, в ES6 появился новый синтаксис для создания классов.

Пример создания класса в ES6:

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
    sayHello() {
    console.log("Hello!");
  }
}
const person = new Person("John", 30);
console.log(person.name); // выведет "John"
person.sayHello(); // вызов метода объекта
```

Геттеры и сеттеры — это специальные методы класса, которые позволяют получать и устанавливать значения свойств объекта.

Пример использования геттеров и сеттеров:

```
class Person {
  constructor(name, age) {
    this._name = name;
    this._age = age;
  }
    get name() {
    return this._name;
  }
  set name(value) {
    this._name = value;
  }
  sayHello() {
    console.log("Hello!");
  }
}
const person = new Person("John", 30);
console.log(person.name); // вызов геттера
person.name = "Bob"; // вызов сеттера
console.log(person.name); // вызов геттера
```

Наследование — это механизм, который позволяет классам наследовать свойства и методы от других классов.

Пример использования наследования:

```
class Animal {
  constructor(name) {
```

```
    this.name = name;
  }
    sayHello() {
    console.log("Hello!");
  }
}
class Dog extends Animal {
  constructor(name) {
    super(name);
  }
  bark() {
    console.log("Woof!");
  }
}
const dog = new Dog("Buddy");
dog.sayHello(); // вызов метода унаследованного класса
dog.bark(); // вызов метода класса Dog
```

**9. JavaScript. Встроенный объект Object. Основные свойства.**

В JavaScript объект Object является встроенным объектом, который представляет собой коллекцию ключ-значение. Каждый объект Object имеет свойства и методы, которые могут быть использованы для работы с объектами.

Основные свойства объекта Object:

1. Object.prototype — это свойство объекта Object, которое содержит методы и свойства, которые наследуются всеми объектами JavaScript.

2. Object.length — это свойство объекта Object, которое содержит количество аргументов, которые ожидаются при вызове конструктора Object.

3. Object.create() — это метод объекта Object, который создает новый объект с указанным прототипом и свойствами.

4. Object.keys() — это метод объекта Object, который возвращает массив строк, представляющих имена перечисляемых свойств объекта.

5. Object.values() — это метод объекта Object, который возвращает массив значений перечисляемых свойств объекта.

6. Object.entries() — это метод объекта Object, который возвращает массив массивов, каждый из которых содержит пару ключ-значение перечисляемых свойств объекта.

7. Object.freeze() — это метод объекта Object, который замораживает объект, предотвращая изменение его свойств.

8. Object.is() — это метод объекта Object, который определяет, являются ли два значения равными. Он возвращает true, если значения равны, и false в противном случае.

**10. JavaScript. Встроенный объект Math. Основные свойства.**

В JavaScript объект Math является встроенным объектом, который предоставляет математические функции и константы. Каждый объект Math не может быть изменен и имеет только свойства и методы для чтения.

Основные свойства объекта Math:

1. Math.PI — значение числа π
2. Math.E — значение числа e
3. Math.LN2 — натуральный логарифм числа 2
4. Math.LN10 — натуральный логарифм числа 10
5. Math.SQRT2 — квадратный корень из 2

Основные методы объекта Math:

1. Math.abs()
2. Math. round () — округляет число до ближайшего целого числа.
3. Math.max() — наибольшее число из заданных аргументов.
4. Math.min() — наименьшее число из заданных аргументов.
5. Math.pow() — значение числа, возведенного в степень.
6. Math.random() — возвращает случайное число
7. Math.sqrt() — квадратный корень числа.

**11. JavaScript. Встроенный объект Date. Основные свойства.**

В JavaScript объект Date является встроенным объектом, который представляет дату и время. Он позволяет работать с датами и временем, выполнять операции с ними и отображать их в различных форматах.

Основные свойства объекта Date:

1. Date.prototype — это свойство объекта Date, которое представляет собой прототип для всех объектов Date.
2. Date.now() — это статическое свойство объекта Date, которое возвращает количество миллисекунд, прошедших с 1 января 1970 года по UTC.
3. Date.parse() — это статическое свойство объекта Date, которое преобразует строку, содержащую дату и время, в число миллисекунд, прошедших с 1 января 1970 года по UTC.
4. Date.UTC() — это статическое свойство объекта Date, которое возвращает количество миллисекунд, прошедших с 1 января 1970 года по UTC для заданной даты и времени.
5. date.getDate() — это метод объекта Date, который возвращает день месяца (от 1 до 31).
6. date.getDay() — это метод объекта Date, который возвращает день недели (от 0 до 6, где 0 - воскресенье).
7. date.getFullYear() — это метод объекта Date, который возвращает год (4 цифры).
8. date.getHours() — это метод объекта Date, который возвращает часы (от 0 до 23).
9. date.getMilliseconds() — это метод объекта Date, который возвращает миллисекунды (от 0 до 999).
10. date.getMinutes() — это метод объекта Date, который возвращает минуты (от 0 до 59).

11. date.getMonth() — это метод объекта Date, который возвращает месяц (от 0 до 11).

12. date.getSeconds() — это метод объекта Date, который возвращает секунды (от 0 до 59).

13. date.getTime() — это метод объекта Date, который возвращает количество миллисекунд, прошедших с 1 января 1970 года по UTC.

**12. JavaScript. Массивы. Встроенный объект Array. Основные свойства.**

Основные свойства

1. length — количество элементов в массиве.

2. constructor — ссылается на функцию-конструктор Array.

3. prototype — позволяет добавлять новые методы и свойства в прототип массива.

4. isArray() — это статический метод объекта Array, который проверяет, является ли переданный аргумент массивом.

5. name — это свойство объекта Function, которое возвращает имя функции (в данном случае "Array").

6. buffer — это свойство объекта TypedArray, которое содержит буфер, используемый для хранения элементов массива.

**13. JavaScript. Строки. Встроенный объект String. Основные свойства.**

1. length — это свойство объекта String, которое возвращает количество символов в строке.

2. fromCharCode() — это статический метод объекта String, который создает строку из указанных кодов символов Unicode.

3. name — это свойство объекта Function, которое возвращает имя функции (в данном случае "String").

4. Symbol.match — это символическое свойство объекта String, которое содержит функцию-предикат для проверки соответствия строки регулярному выражению.

5. Symbol.replace — это символическое свойство объекта String, которое содержит функцию для замены подстрок в строке с помощью регулярного выражения.

6. Symbol.search — это символическое свойство объекта String, которое содержит функцию для поиска подстроки в строке с помощью регулярного выражения.

7. split() — это метод объекта String, который разбивает строку на массив подстрок с помощью разделителя и возвращает новый массив.

8. charAt() — это метод объекта String, который возвращает символ строки по указанному индексу.

9. concat() — это метод объекта String, который объединяет две или более строк и возвращает новую строку.

10. includes() — это метод объекта String, который проверяет, содержит ли строка указанную подстроку, и возвращает true или false.

11. indexOf() — это метод объекта String, который возвращает индекс первого вхождения указанной подстроки в строку или -1, если подстрока не найдена.

12. lastIndexOf() — это метод объекта String, который возвращает индекс последнего вхождения указанной подстроки в строку или -1, если подстрока не найдена.

13. replace() — это метод объекта String, который заменяет подстроки в строке с помощью регулярного выражения и возвращает новую строку.

14. slice() - это метод объекта String, который возвращает часть строки, начиная с указанного индекса и до конца строки или до указанного индекса.

15. startsWith() — это метод объекта String, который проверяет, начинается ли строка с указанной подстроки, и возвращает true или false.

16. toLowerCase() — это метод объекта String, который преобразует все символы строки в нижний регистр и возвращает новую строку.

17. toUpperCase() — это метод объекта String, который преобразует все символы строки в верхний регистр и возвращает новую строку.

18. trim() — это метод объекта String, который удаляет пробельные символы в начале и конце строки и возвращает новую строку.

**14. JavaScript. Определение регулярных выражений. Встроенный объект RegExp. Основные свойства объекта String для поиска и замены по шаблону.**

Регулярные выражения (RegExp) — это шаблоны, используемые для поиска и замены текста в строке. Они состоят из символов, которые определяют правила поиска и замены.

В JavaScript, объект RegExp используется для работы с регулярными выражениями. Он имеет два способа создания: с помощью литерала регулярного выражения (/pattern/) или с помощью конструктора new RegExp("pattern").

Основные свойства объекта String для работы с регулярными выражениями:

1. search() - метод, который ищет заданный шаблон в строке и возвращает индекс первого вхождения или -1, если шаблон не найден.

2. match() - метод, который ищет заданный шаблон в строке и возвращает массив с найденными совпадениями.

3. replace() - метод, который заменяет заданный шаблон в строке на указанную подстроку и возвращает новую строку.

4. split() - метод, который разбивает строку на массив подстрок по заданному шаблону.

Пример использования регулярных выражений:

```
var str = "The quick brown fox jumps over the lazy dog.";
var pattern = /fox/;
var result = str.search(pattern); // результат: 16

var pattern1 = /the/gi;
var result1= str.match(pattern1); // результат: ["The", "the"]

var pattern2 = /the/gi;
var result2= str.replace(pattern2, "a"); // результат: "a quick brown fox jumps over a lazy dog."

var pattern3 = / /;
var arr = str.split(pattern); // результат: ["The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog."]
```

**15. JavaScript. Встраивание JavaScript-кода в разметку HTML. Выполнение сценария JavaScript.**

JavaScript может быть встроен в HTML-документы с помощью тега <script>. Этот тег может быть размещен внутри <head> или <body> элементов и содержит JavaScript-код.

Пример:

```
<!DOCTYPE html>
<html>
<head>
 <title>Пример встраивания JavaScript</title>
 <script>
  function showMessage() {
   alert("Привет, мир!");
  }
 </script>
</head>
<body>
 <button onclick="showMessage()">Нажми меня</button>
</body>
</html>
```

JavaScript-код также может быть размещен в отдельном файле с расширением .js и подключен к HTML-документу с помощью атрибута src тега <script>.

Пример:

```
<!DOCTYPE html>
<html>
<head>
 <title>Пример подключения файла JavaScript</title>
 <script src="script.js"></script>
</head>
<body>
 <button onclick="showMessage()">Нажми меня</button>
</body>
</html>
```

**16. JavaScript. Объект Window. Таймеры.**

Объект Window является главным объектом в браузере и представляет окно браузера или вкладку. Он содержит множество свойств и методов, которые позволяют взаимодействовать с браузером и документом.

Например, свойство window.location содержит информацию о текущем URL-адресе страницы, а метод window.alert() выводит модальное окно с сообщением для пользователя.

Таймеры в JavaScript позволяют запускать функции через определенный промежуток времени или с задержкой. Для этого используются функции setInterval() и setTimeout().

Функция setInterval() запускает функцию через определенный интервал времени и продолжает ее вызывать снова и снова, пока не будет вызван метод clearInterval() для остановки таймера.

Пример:

```
var intervalId = setInterval(function() {
 console.log("Прошло 1 секунда");
}, 1000);
setTimeout(function() {
 clearInterval(intervalId);
}, 5000);
```

Функция setTimeout() запускает функцию через определенную задержку времени и вызывает ее только один раз.

```
setTimeout(function() {
 console.log("Прошла 1 секунда");
}, 1000);
```

**17. JavaScript. Объект Window. История посещений. Информация о браузере и об экране.(можно еще 18 вопрос посмотреть)**

Объект Window в JavaScript представляет главное окно браузера, в котором отображается документ. История посещений хранится в свойстве window.history. Это позволяет переходить назад и вперед по истории браузера с помощью методов back() и forward().

Для получения информации о браузере, можно использовать свойства объекта window.navigator. Например, свойство navigator.userAgent содержит информацию о браузере и операционной системе. Также можно получить информацию об экране с помощью свойств window.screen.width и window.screen.height, которые содержат ширину и высоту экрана в пикселях.

Свойство window.screen содержит информацию о размерах экрана пользователя, его разрешении и глубине цвета.

Также объект Window содержит множество других полезных свойств и методов, которые позволяют управлять окном браузера, открывать новые окна, работать с cookie и многое другое.

**18. JavaScript. Объект Window. Работа с окнами.**

JavaScript позволяет работать с окнами браузера через объект Window. С помощью методов этого объекта можно открывать новые окна, закрывать текущее окно, изменять его размеры и положение на экране.

Метод window.open(url, name, specs) открывает новое окно браузера и загружает в него страницу по указанному URL-адресу. Параметр name задает имя окна, а параметр specs определяет его характеристики, такие как размеры, положение, наличие панелей инструментов и т.д.

Метод window.close() закрывает текущее окно браузера.

Методы window.resizeTo(width, height) и window.moveTo(x, y) изменяют размеры и положение окна на экране соответственно.

Также объект Window содержит методы для работы с всплывающими окнами (pop-up windows), которые могут быть блокированы браузером пользователя. Для открытия всплывающего окна рекомендуется использовать метод window.open() с параметрами, которые не вызовут блокировку.

Например, следующий код откроет новое окно браузера с загруженной страницей Google:

```
window.open("https://www.google.com", "google", "width=800,height=600");
```

**19. JavaScript. Объектная модель документа (DOM).**

Объектная модель документа (Document Object Model, DOM) представляет собой структуру документа в виде дерева объектов, которые могут быть изменены с помощью JavaScript. DOM позволяет обращаться к элементам HTML-страницы, изменять их содержимое, стили, атрибуты и т.д.

Каждый элемент HTML-страницы представлен объектом в DOM, который имеет свойства и методы для работы с ним. Например, свойство innerHTML позволяет получить или изменить содержимое элемента HTML, а методы getAttribute() и setAttribute() позволяют получить и изменить значения атрибутов элемента.

DOM также позволяет работать с событиями на странице. События – это действия пользователя или браузера, такие как клик мышью, нажатие клавиши на клавиатуре, загрузка страницы и т.д. С помощью JavaScript можно создавать обработчики событий, которые будут реагировать на определенные действия пользователя или браузера.

Например, следующий код добавляет обработчик события клика на кнопку:

```
var button = document.getElementById("myButton");
button.addEventListener("click", function() {
  alert("Button clicked!");
});
```

DOM также позволяет создавать новые элементы на странице и добавлять их в документ. Например, следующий код создает новый элемент div, добавляет ему текстовое содержимое и добавляет его в конец элемента body:

```
var newDiv = document.createElement("div");
newDiv.innerHTML = "New div element";
document.body.appendChild(newDiv);
```

**20. JavaScript. DOM. Доступ к элементам документа.**

Для работы с элементами документа в DOM используются методы и свойства объектов. Например, чтобы изменить содержимое элемента, можно использовать свойство innerHTML:

```
document.getElementById("myElement").innerHTML = "Новый текст";
```

Здесь "innerHTML" – это свойство объекта, которое позволяет изменить содержимое элемента.

Чтобы изменить атрибут элемента, можно использовать метод setAttribute:

document.getElementById("myElement").setAttribute("class", "newClass");

Здесь "setAttribute" – это метод объекта, который позволяет изменить значение атрибута элемента.

Для добавления нового элемента на страницу можно использовать метод createElement:

var newElement = document.createElement("div");

Здесь "createElement" – это метод объекта, который создает новый элемент.

Чтобы добавить созданный элемент на страницу, нужно использовать метод appendChild:

document.getElementById("myContainer").appendChild(newElement);

Здесь "appendChild" – это метод объекта, который добавляет новый элемент в конец контейнера.

**21. JavaScript. DOM. Структура документа и навигация.**

Документ Object Model (DOM) – это стандарт, который определяет структуру и свойства объектов, представляющих веб-страницу. DOM представляет веб-страницу в виде дерева объектов, где каждый элемент страницы является объектом с определенными свойствами и методами.Структура документа в DOM представляется в виде объекта document, который содержит все элементы страницы. Например, чтобы получить доступ к элементу с идентификатором "myElement", можно использовать следующий код:

var element = document.getElementById("myElement");

Здесь "getElementById" – это метод объекта document, который позволяет получить доступ к элементу страницы по его идентификатору.

Навигация по структуре документа в DOM осуществляется с помощью свойств parentNode, childNodes и siblings. Например, чтобы получить доступ к родительскому элементу элемента "myElement", можно использовать следующий код:

var parentElement = document.getElementById("myElement").parentNode;

Здесь "parentNode" – это свойство объекта, которое позволяет получить доступ к родительскому элементу.

Чтобы получить доступ к дочерним элементам элемента, можно использовать свойство childNodes. Например, чтобы получить доступ к первому дочернему элементу элемента "myElement", можно использовать следующий код:

var firstChild = document.getElementById("myElement").childNodes[0];

Здесь "childNodes" – это свойство объекта, которое возвращает коллекцию дочерних элементов.

Для доступа к соседним элементам можно использовать свойства previousSibling и nextSibling. Например, чтобы получить доступ к предыдущему элементу от элемента "myElement", можно использовать следующий код:

var previousElement = document.getElementById("myElement").previousSibling;

Здесь "previousSibling" – это свойство объекта, которое возвращает предыдущий соседний элемент.

**22. JavaScript. DOM. Работа с контентом элемента. Управление стилями элемента.**

JavaScript позволяет изменять контент и стили элементов на веб-страницах с помощью объекта Document Object Model (DOM).Для изменения контента элемента можно использовать свойства innerHTML или textContent. Например, чтобы изменить текст элемента с идентификатором "myElement", можно использовать следующий код:

```
document.getElementById("myElement").innerHTML = "Новый текст";
```

Здесь "innerHTML" – это свойство, которое позволяет изменять HTML-код внутри элемента, а "Новый текст" – это новый текст, который будет отображаться в элементе.

Альтернативно, можно использовать свойство textContent, которое изменяет только текстовое содержимое элемента:

```
document.getElementById("myElement").textContent = "Новый текст";
```

Для изменения стилей элемента можно использовать свойство style. Например, чтобы изменить цвет фона элемента с идентификатором "myElement", можно использовать следующий код:

```
document.getElementById("myElement").style.backgroundColor = "red";
```

Здесь "backgroundColor" – это свойство, которое позволяет изменять цвет фона элемента, а "red" – это новый цвет фона.

Также можно использовать методы классов для управления стилями элемента. Например, чтобы добавить класс "active" к элементу с идентификатором "myElement", можно использовать следующий код:

```
document.getElementById("myElement").classList.add("active");
```

Здесь "classList" – это свойство, которое предоставляет доступ к классам элемента, а "add" – это метод, который добавляет новый класс к элементу.

**23. JavaScript. DOM. Обработка событий.**

Для добавления обработчика события к элементу на странице можно использовать метод addEventListener(). Например, чтобы обработать клик на кнопке с идентификатором "myButton", можно использовать следующий код:

```
document.getElementById("myButton").addEventListener("click", function() {
  // выполнить действие при клике на кнопку
});
```

Здесь "click" – это тип события, которое происходит при клике на кнопку, а функция внутри addEventListener() выполняет необходимые действия.Также можно использовать атрибуты HTML для добавления обработчиков событий. Например, чтобы обработать отправку формы, можно использовать следующий код:

```
<form onsubmit="myFunction()">
  // содержимое формы
</form>
```

Здесь "myFunction()" – это функция, которая будет вызвана при отправке формы.JavaScript также позволяет удалять обработчики событий с помощью метода removeEventListener(). Например, чтобы удалить обработчик клика на кнопке, можно использовать следующий код:

```
var button = document.getElementById("myButton");
var myFunction = function() {
```

```
  // выполнить действие при клике на кнопку
};
button.addEventListener("click", myFunction);
// удалить обработчик клика на кнопке
button.removeEventListener("click", myFunction);
```

**24. JavaScript. DOM. Обработка форм.**

Одним из наиболее распространенных случаев использования JavaScript на веб-страницах является обработка форм. Формы позволяют пользователю вводить данные, которые затем можно отправить на сервер для обработки. JavaScript позволяет добавлять дополнительную функциональность к формам, например, проверку правильности заполнения полей или изменение содержимого страницы в зависимости от выбранных пользователем опций.Для доступа к элементам формы на странице используется объект Document, который представляет текущий документ. Например, чтобы получить доступ к элементу формы с именем "username", можно использовать следующий код:

```
var usernameInput = document.forms["myForm"].elements["username"];
```

Здесь "myForm" – это имя формы, а "username" – имя элемента формы.

Для обработки событий формы, таких как отправка данных на сервер или изменение значения поля, используются обработчики событий. Например, чтобы выполнить определенное действие при отправке формы, можно использовать следующий код:

```
document.forms["myForm"].addEventListener("submit", function(event) {
  // выполнить действие при отправке формы
});
```

Здесь "submit" – это тип события, которое происходит при отправке формы, а функция внутри addEventListener() выполняет необходимые действия.

JavaScript также позволяет изменять содержимое элементов формы, например, устанавливать значение поля или добавлять новый элемент. Например, чтобы установить значение поля с именем "username", можно использовать следующий код:

```
document.forms["myForm"].elements["username"].value = "John";
```

Здесь "John" – это новое значение поля.

**25. JavaScript. Сериализация и десериализация объектов. Формат JSON.**

Сериализация объектов в JavaScript означает преобразование объекта в строку, которую можно отправить на сервер или сохранить в локальном хранилище. Десериализация – это обратный процесс, когда строка преобразуется обратно в объект.

В JavaScript для сериализации и десериализации объектов используется формат JSON (JavaScript Object Notation). JSON – это легкий формат обмена данными, основанный на синтаксисе объектов JavaScript. Он используется для передачи данных между клиентом и сервером в веб-приложениях.Для сериализации объекта в формат JSON используется метод JSON.stringify(), который принимает объект в качестве аргумента и возвращает строку в формате JSON. Например:

```
var obj = {name: "John", age: 30};
var json = JSON.stringify(obj);
```

Для десериализации строки JSON в объект используется метод JSON.parse(), который принимает строку в формате JSON и возвращает объект. Например:

```
var json = '{"name": "John", "age": 30}';
var obj = JSON.parse(json);
```

Объекты, которые могут быть сериализованы в формат JSON, должны быть простыми (не содержать функций или циклических ссылок) и состоять только из примитивных типов данных (строк, чисел, логических значений, массивов и других объектов, которые также могут быть сериализованы в формат JSON).JSON является удобным и распространенным форматом для обмена данными между клиентом и сервером в веб-приложениях. Он позволяет легко преобразовывать объекты в строки и обратно, что упрощает передачу данных и их обработку на стороне сервера или клиента.

**26. JavaScript. Работа с протоколом HTTP.**

JavaScript позволяет отправлять запросы на сервер с помощью протокола HTTP (Hypertext Transfer Protocol). Для этого используется объект XMLHttpRequest (XHR), который позволяет асинхронно отправлять и получать данные с сервера.

Для отправки запроса на сервер с помощью XHR используется метод open(), который принимает три аргумента: метод запроса (GET, POST, PUT, DELETE и т.д.), URL сервера и флаг асинхронности (true или false). Например, чтобы отправить GET-запрос на сервер, можно использовать следующий код:

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "http://example.com/api/data", true);
```

После того, как запрос был открыт, можно отправить его на сервер с помощью метода send(), который принимает данные запроса в виде строки. Например, чтобы отправить POST-запрос на сервер с данными в формате JSON, можно использовать следующий код:

```
var xhr = new XMLHttpRequest();
xhr.open("POST", "http://example.com/api/data", true);
xhr.setRequestHeader("Content-Type", "application/json");
var data = {name: "John", age: 30};
xhr.send(JSON.stringify(data));
```

После отправки запроса можно получить ответ от сервера с помощью свойства responseText, которое содержит ответ в виде строки. Например, чтобы получить ответ от сервера после отправки GET-запроса, можно использовать следующий код:

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "http://example.com/api/data", true);
xhr.onreadystatechange = function() {
  if (xhr.readyState === 4 && xhr.status === 200) {
    console.log(xhr.responseText);
  }
};
xhr.send();
```

XHR также предоставляет возможность отслеживать прогресс выполнения запроса с помощью свойства onprogress, которое вызывается во время отправки и получения данных.

Например, чтобы отслеживать прогресс выполнения загрузки файла, можно использовать следующий код:

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "http://example.com/files/file.pdf", true);
xhr.onprogress = function(event) {
  if (event.lengthComputable) {
    var percentComplete = event.loaded / event.total * 100;
    console.log(percentComplete + "%");
  }
};
xhr.send();
```

**27. JavaScript. Сохранение данных на стороне клиента.**

JavaScript позволяет сохранять данные на стороне клиента с помощью объекта localStorage. Этот объект предоставляет доступ к хранилищу данных браузера, в котором можно хранить данные в виде пар ключ-значение.

Для сохранения данных в localStorage используется метод setItem(), который принимает два аргумента: ключ и значение. Например, чтобы сохранить имя пользователя, можно использовать следующий код:

```
localStorage.setItem("username", "John");
```

Для получения сохраненных данных используется метод getItem(), который принимает ключ и возвращает значение. Например, чтобы получить имя пользователя, можно использовать следующий код:

```
var username = localStorage.getItem("username");
```

Данные в localStorage хранятся без ограничения по времени, то есть они будут доступны даже после перезагрузки страницы или закрытия браузера. Чтобы удалить данные из localStorage, можно использовать метод removeItem(), который принимает ключ. Например, чтобы удалить сохраненное имя пользователя, можно использовать следующий код:

```
localStorage.removeItem("username");
```

LocalStorage может быть полезен для сохранения настроек пользователя, последних запросов к серверу или других данных, которые могут быть полезны для повторного использования в будущем. Однако не следует хранить в localStorage конфиденциальные данные, так как они могут быть доступны злоумышленникам при возможном взломе браузера.

**28. JavaScript. Библиотека jQuery. Доступ к элементам документа. Навигация.**

Для доступа к элементам документа в jQuery используется селекторы, которые позволяют выбирать элементы по различным критериям, таким как id, класс, тип элемента и другие атрибуты.Например, чтобы выбрать элемент с id="myElement", можно использовать следующий код:

```
$("#myElement")
```

А чтобы выбрать все элементы с классом "myClass", можно использовать следующий код:

```
$(".myClass")
```

Также в jQuery есть методы для навигации по дереву элементов документа. Например, методы .parent(), .children() и .siblings() позволяют получить родительский элемент, дочерние элементы и соседние элементы соответственно.Например, чтобы получить все дочерние элементы элемента с id="myElement", можно использовать следующий код:

```
$("#myElement").children()
```

А чтобы получить все соседние элементы элемента с классом "myClass", можно использовать следующий код:

```
$(".myClass").siblings()
```

**29. JavaScript. Библиотека jQuery. Обработка событий.**

JavaScript позволяет обрабатывать события на веб-странице, такие как клики мыши, нажатия клавиш и другие действия пользователя. Библиотека jQuery упрощает этот процесс, предоставляя удобные методы для обработки событий.

Для привязки обработчика событий к элементу на странице можно использовать методы .click(), .keydown(), .submit() и другие. Например:

```
$("#myButton").click(function() {
  alert("Кнопка нажата");
});
```

Этот код добавляет обработчик клика на элемент с id="myButton", который вызывает функцию, выводящую сообщение.

Также можно использовать метод .on() для привязки нескольких обработчиков к одному элементу или для привязки обработчика к нескольким событиям. Например:

```
$("#myElement").on({
  click: function() {
    alert("Кликнули на элементе");
  },
  mouseenter: function() {
    $(this).css("background-color", "yellow");
  },
  mouseleave: function() {
    $(this).css("background-color", "");
  }
});
```

Этот код добавляет обработчики клика, наведения мыши и ухода мыши с элемента с id="myElement".

Также можно использовать методы .off() для отвязки обработчиков событий и .trigger() для программного вызова событий.

30. JavaScript. Библиотека jQuery. Анимационные эффекты.

Библиотека jQuery - это набор функций JavaScript, которые упрощают работу с DOM-деревом и обработку событий на странице. Она также позволяет создавать анимационные эффекты на странице.

Для создания анимаций с помощью jQuery можно использовать методы .animate() и .fadeIn()/.fadeOut(). Метод .animate() позволяет изменять значения CSS свойств элемента в течение определенного времени, что создает эффект движения. Например:

$("#myElement").animate({left: "300px"}, 1000);

Этот код изменит значение свойства left элемента с id="myElement" до 300px за 1 секунду.

Методы .fadeIn() и .fadeOut() позволяют плавно появляться или исчезать элементу на странице. Например:

$("#myElement").fadeIn(1000);

Этот код плавно появит элемент с id="myElement" за 1 секунду.

Также можно использовать другие методы для создания различных анимационных эффектов, таких как .slideDown()/.slideUp() для плавного раскрытия/скрытия элемента, .toggleClass() для изменения класса элемента с анимацией и многие другие.

**31. JavaScript. Библиотека jQuery. Реализация AJAX.**

Библиотека jQuery - это набор функций JavaScript, которые упрощают работу с DOM-деревом и обработку событий на странице. Она позволяет создавать анимации, изменять стили элементов, выполнять AJAX-запросы и многое другое.

AJAX - это технология, которая позволяет обновлять часть страницы без перезагрузки всей страницы. Она использует JavaScript для отправки запросов на сервер и получения данных в формате XML, JSON или HTML. Это позволяет создавать более быстрые и удобные веб-приложения.

Для реализации AJAX-запросов с помощью jQuery необходимо использовать методы $.ajax(), $.get() или $.post(). Например, чтобы получить данные из файла data.json, можно использовать следующий код:

```
$.ajax({
  url: "data.json",
  dataType: "json",
  success: function(data) {
    console.log(data);
  }
});
```

Этот код отправит GET-запрос на сервер по адресу "data.json" и ожидает получить данные в формате JSON. Если запрос успешен, то данные будут выведены в консоль браузера.

Также можно использовать методы $.get() или $.post() для более простой реализации AJAX-запросов. Например:

```
$.get("data.json", function(data) {
  console.log(data);
}, "json");
```

Этот код отправит GET-запрос на сервер по адресу "data.json" и ожидает получить данные в формате JSON. Если запрос успешен, то данные будут выведены в консоль браузера.

В целом, jQuery позволяет упростить работу с AJAX-запросами и создать более интерактивные и удобные веб-приложения.

**32. JavaScript. Библиотека React. Настройка окружения.**

Для начала работы с React необходимо настроить окружение. Вот основные шаги:

1. Установить Node.js и npm (Node Package Manager), если они еще не установлены на компьютере.

2. Создать новый проект с помощью команды "npx create-react-app my-app" в командной строке. "my-app" — это название вашего проекта, которое можно изменить на свое усмотрение.

3. Перейти в папку проекта с помощью команды "cd my-app".

4. Запустить проект с помощью команды "npm start". Она автоматически откроет браузер и запустит приложение на локальном сервере.

Теперь вы можете начать создавать свои компоненты и разрабатывать приложение с помощью React.

**33. JavaScript. Библиотека React. Функциональные и классовые компоненты.**

React — это библиотека JavaScript для создания пользовательских интерфейсов. Она использует компонентный подход, где каждый элемент интерфейса представлен в виде компонента.

Компоненты в React могут быть функциональными или классовыми. Функциональные компоненты — это простые функции, которые принимают некоторые данные (props) и возвращают React-элементы. Они обычно используются для создания простых компонентов, которые не имеют состояния.

```
function MyComponent(props) {
  return <div>Hello {props.name}!</div>;
}
```

Классовые компоненты — это классы, которые наследуются от базового класса React.Component. Они могут иметь состояние и методы жизненного цикла, что позволяет им управлять своим поведением и отображением.

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }
  handleClick() {
    this.setState({ count: this.state.count + 1 });
  }

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={() => this.handleClick()}>Increment</button>
      </div>
```

```
    );
  }
}
```

Здесь мы определили класс "MyComponent", который имеет состояние "count" и метод "handleClick", который изменяет это состояние. Метод "render" возвращает React-элемент, который отображает текущее значение состояния и кнопку для увеличения его значения при нажатии.

**34. JavaScript. Библиотека React. Передача данных в компонент. Props.**

В библиотеке React данные передаются в компоненты с помощью props (properties - свойства). Props - это объект, который содержит все переданные компоненту свойства.

Пример использования props:

```
function Greeting(props) {
  return <h1>Hello, {props.name}!</h1>;
}
const element = <Greeting name="John" />;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

В этом примере мы создаем компонент Greeting, который принимает свойство name. Затем мы создаем элемент, передавая свойство name со значением "John". Наконец, мы рендерим элемент в DOM.

В компоненте мы можем обращаться к свойствам с помощью объекта props. В нашем примере мы выводим приветствие с именем, переданным через свойство name.

Props могут быть любого типа данных - строки, числа, объекты и т.д. Мы также можем передавать функции в качестве свойств, чтобы компоненты могли взаимодействовать друг с другом.

**35. JavaScript. Библиотека React. Состояние компонента.**

Кроме props, компоненты в React могут иметь состояние (state). Состояние - это объект, который содержит данные, управляемые компонентом. Когда состояние изменяется, React автоматически перерисовывает компонент, чтобы отобразить новое состояние.

Пример использования состояния:

```
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }
  handleClick = () => {
    this.setState({ count: this.state.count + 1 });
  }
  render() {
```

```
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={this.handleClick}>Click me</button>
      </div>
    );
  }
}
ReactDOM.render(
  <Counter />,
  document.getElementById('root')
);
```

В этом примере мы создаем компонент Counter, который имеет состояние count со значением 0. Когда пользователь кликает на кнопку, вызывается метод handleClick, который увеличивает значение count на 1 и обновляет состояние с помощью метода setState.

В методе render мы выводим текущее значение count и кнопку, которая вызывает метод handleClick при клике.

Состояние не должно изменяться напрямую, только через метод setState. Это гарантирует правильную работу React и обновление компонента при изменении состояния.

Использование состояния позволяет создавать динамические и интерактивные компоненты, которые могут изменять свое поведение в зависимости от пользовательского ввода или других факторов.

**36. JavaScript. Библиотека React. Хуки. UseState, UseEffect, UseContext.**

Хуки (hooks) в React — это функции, которые позволяют использовать состояние и другие возможности React в функциональных компонентах. Ранее, до появления хуков, для работы с состоянием и жизненным циклом компонентов в функциональных компонентах использовались различные паттерны и библиотеки.

useState

Хук useState позволяет добавить состояние в функциональный компонент. Он принимает начальное значение состояния и возвращает массив, содержащий текущее значение состояния и функцию для его обновления.

Пример использования useState:

```
import React, { useState } from 'react';
const Counter = () => {
  const [count, setCount] = useState(0);
  const handleClick = () => {
    setCount(count + 1);
  };
  return (
    <div>
      <p>Count: {count}</p>
```

```
      <button onClick={handleClick}>Increment</button>
    </div>
  );
};
```

В этом примере мы создаем компонент Counter, который использует хук useState для добавления состояния count. Мы также определяем функцию handleClick, которая будет вызываться при клике на кнопку и увеличивать значение count на 1.

Возвращаемый массив из useState содержит текущее значение состояния count и функцию setCount, которая позволяет обновлять это значение.

UseEffect

Хук useEffect позволяет выполнять побочные эффекты в функциональных компонентах. Он принимает функцию, которая будет выполняться после каждого рендеринга компонента.

Пример использования useEffect:

```
import React, { useState, useEffect } from 'react';
const Timer = () => {
  const [seconds, setSeconds] = useState(0);
  useEffect(() => {
    const intervalId = setInterval(() => {
      setSeconds(seconds + 1);
    }, 1000);

    return () => {
      clearInterval(intervalId);
    };
  }, [seconds]);
  return (
    <div>
      <p>Seconds: {seconds}</p>
    </div>
  );
};
```

В этом примере мы создаем компонент Timer, который использует хук useEffect для запуска таймера. Мы определяем функцию setInterval, которая будет вызываться каждую секунду и обновлять значение seconds с помощью функции setSeconds.

Функция useEffect принимает второй параметр - массив зависимостей. Если этот массив не указан, то функция будет вызываться после каждого рендеринга компонента. Если указан массив зависимостей, то функция будет вызываться только при изменении значений в этом массиве.

В нашем примере мы указали зависимость [seconds], чтобы функция useEffect вызывалась только при изменении значения seconds.

UseContext

Хук useContext позволяет использовать контекст (context) в функциональных компонентах. Контекст - это механизм передачи данных через дерево компонентов без необходимости передавать пропсы вручную на каждом уровне.

Пример использования useContext:

```
import React, { useContext } from 'react';
const ThemeContext = React.createContext('light');
const Header = () => {
  const theme = useContext(ThemeContext);
  return (
    <div>
      <p>Theme: {theme}</p>
    </div>
  );
};
const App = () => (
  <ThemeContext.Provider value="dark">
    <Header />
  </ThemeContext.Provider>
);
```

В этом примере мы создаем контекст ThemeContext с начальным значением "light". Мы также создаем компонент Header, который использует хук useContext для получения значения контекста ThemeContext.

Компонент App оборачивает компонент Header в провайдер (Provider) контекста ThemeContext с новым значением "dark". Это позволяет компоненту Header получить значение контекста из провайдера, а не из начального значения.

Хуки useState, useEffect и useContext — это лишь некоторые из возможностей, которые предоставляют хуки в React. Они позволяют упростить написание функциональных компонентов и сделать их более гибкими и мощными.

**37. JavaScript. Библиотека React. Маршрутизация.**

Кроме работы с компонентами, React также предоставляет средства для маршрутизации (routing) веб-приложений. Маршрутизация позволяет определять, какой компонент должен быть отображен на странице в зависимости от URL.

Для работы с маршрутизацией в React используется библиотека react-router. Она предоставляет компоненты для определения маршрутов и отображения соответствующих компонентов.

Пример использования react-router:

```
import { BrowserRouter as Router, Route, Link } from 'react-router-dom';
const Home = () => <h1>Home</h1>;
const About = () => <h1>About</h1>;
const Contact = () => <h1>Contact</h1>;
const App = () => (
  <Router>
```

```
      <div>
        <nav>
          <ul>
            <li><Link to="/">Home</Link></li>
            <li><Link to="/about">About</Link></li>
            <li><Link to="/contact">Contact</Link></li>
          </ul>
        </nav>
        <Route exact path="/" component={Home} />
        <Route path="/about" component={About} />
        <Route path="/contact" component={Contact} />
      </div>
    </Router>
  );
  ReactDOM.render(
    <App />,
    document.getElementById('root')
  );
```

В этом примере мы создаем три компонента - Home, About и Contact, которые будут отображаться на разных страницах. Для определения маршрутов мы используем компоненты Router, Route и Link из библиотеки react-router-dom.

Компонент Router определяет контейнер для маршрутизации. Внутри него мы создаем навигационное меню с помощью компонента Link, который представляет собой ссылку на другую страницу. Компоненты Route определяют соответствие между URL и компонентами, которые должны быть отображены на странице.

В примере мы использовали атрибут exact для маршрута "/", чтобы указать, что он должен соответствовать только точному совпадению URL. Без этого атрибута маршрут "/" также будет соответствовать любому другому URL, начинающемуся с "/".

Маршрутизация в React позволяет создавать более сложные веб-приложения с несколькими страницами и переходами между ними. Она также упрощает работу с URL и позволяет создавать более удобную навигацию для пользователей.

**38. NodeJS. Обзор платформы. Окружение.**

NodeJS - это среда выполнения JavaScript, которая позволяет запускать код на стороне сервера. Она основана на движке V8 от Google, который используется в браузере Chrome для выполнения JavaScript.

NodeJS позволяет разработчикам использовать JavaScript как язык для создания серверных приложений и инструментов командной строки. Она также предоставляет доступ к множеству модулей и библиотек, которые упрощают разработку приложений.

Окружение NodeJS включает в себя:

1. Модули - NodeJS предоставляет множество встроенных модулей, таких как http, fs, path и другие, которые позволяют работать с сетью, файловой системой, путями и другими аспектами серверной разработки.

2. Пакетный менеджер npm - npm позволяет управлять зависимостями приложения и устанавливать сторонние модули и библиотеки.

3. REPL - REPL (Read-Eval-Print Loop) — это интерактивная консоль, которая позволяет выполнять JavaScript код в реальном времени.

4. Debugger - NodeJS предоставляет встроенный отладчик, который позволяет отлаживать приложения и находить ошибки.

5. Event Loop - NodeJS использует Event Loop для обработки асинхронного кода. Это позволяет создавать высокопроизводительные приложения, которые могут обрабатывать большое количество запросов одновременно.

6. Сервер - NodeJS может быть использован для создания сервера, который может обрабатывать запросы HTTP и WebSocket.

NodeJS является популярной платформой для разработки серверных приложений и инструментов командной строки. Она позволяет использовать JavaScript как язык для создания серверных приложений и имеет множество встроенных модулей и библиотек, которые упрощают разработку.

**39. NodeJS. Различия между платформой Node.js и браузером.**

NodeJS и браузер имеют схожий движок V8, но существуют значительные различия между этими платформами:

1. Среда выполнения - NodeJS запускается на сервере, в то время как браузер запускается на клиентской машине.

2. Доступ к ресурсам - NodeJS имеет доступ к ресурсам сервера, таким как файловая система, сеть и т.д., в то время как браузер имеет ограниченный доступ к ресурсам клиентской машины.

3. API - NodeJS предоставляет API для работы с сервером, файловой системой, сетью и другими аспектами серверной разработки, в то время как браузер предоставляет API для работы с DOM, событиями и другими аспектами клиентской разработки.

4. Модули - NodeJS предоставляет модули для работы с сервером, файловой системой, сетью и другими аспектами серверной разработки, в то время как браузер предоставляет модули для работы с DOM, событиями и другими аспектами клиентской разработки.

5. Асинхронность - NodeJS использует Event Loop для обработки асинхронного кода, что позволяет создавать высокопроизводительные приложения, которые могут обрабатывать большое количество запросов одновременно. Браузер также поддерживает асинхронность, но в основном для работы с DOM.

6. Модульность - NodeJS позволяет использовать модули и библиотеки из NPM, что упрощает разработку приложений. Браузер также поддерживает модульность, но требует использования инструментов сборки, таких как Webpack или Browserify.

В целом, NodeJS и браузер имеют различные цели и области применения, но оба используют JavaScript и имеют схожий движок V8.

**40. NodeJS. Понятие модуля. Экспорт данных.**

NodeJS позволяет использовать модули для организации кода и упрощения разработки приложений. Модуль — это набор функций, переменных и классов, которые могут быть использованы в других частях приложения.

Для экспорта данных из модуля в NodeJS используется объект module.exports. Этот объект может содержать любые данные, включая функции, переменные и объекты. Например, чтобы экспортировать функцию add, которая складывает два числа, из модуля, можно написать следующий код:

```
function add(a, b) {
  return a + b;
}
module.exports = add;
```

Этот код экспортирует функцию add из модуля. Теперь эту функцию можно использовать в других частях приложения, подключив модуль с помощью функции require:

```
const add = require('./add');
console.log(add(2, 3)); // 5
```

В этом примере мы подключаем модуль add и вызываем функцию add, передавая ей два аргумента. Результат выполнения функции выводится в консоль.

Также можно экспортировать несколько функций, переменных или объектов из модуля, объединив их в объекте module.exports:

```
function add(a, b) {
  return a + b;
}
function subtract(a, b) {
  return a - b;
}
module.exports = {
  add: add,
  subtract: subtract
};
```

В этом примере мы экспортируем две функции add и subtract из модуля, объединив их в объекте module.exports. После подключения модуля с помощью функции require мы можем использовать эти функции:

```
const math = require('./math');
console.log(math.add(2, 3)); // 5
console.log(math.subtract(5, 2)); // 3
```

**41. NodeJS. NPM. Файлы package.json, package-lock.json.**

NodeJS - это среда выполнения JavaScript на стороне сервера, которая позволяет разрабатывать высокопроизводительные и масштабируемые приложения. Одним из основных преимуществ NodeJS является использование модульной системы для организации кода.

NPM (Node Package Manager) - это менеджер пакетов для NodeJS, который позволяет управлять зависимостями проекта и устанавливать необходимые модули. NPM позволяет

установить любой модуль, опубликованный в реестре npmjs.com, а также использовать собственные модули.

Файл package.json - это файл конфигурации проекта, который содержит информацию о зависимостях, скриптах и других настройках проекта. Этот файл создается автоматически при инициализации проекта с помощью команды npm init.

Файл package-lock.json - это файл, который создается автоматически после установки зависимостей проекта с помощью команды npm install. Этот файл содержит информацию о версиях установленных модулей и их зависимостях, чтобы гарантировать одинаковую установку модулей на всех устройствах.

**42. NodeJS. Утилита NPX.**

NPX — это утилита, входящая в состав NodeJS, которая позволяет запускать команды из установленных модулей без необходимости устанавливать их глобально на компьютере. Это особенно удобно для запуска одноразовых скриптов или для использования модулей, которые не нужны в каждом проекте.

Например, чтобы запустить локально установленный модуль "eslint" для проверки кода, можно использовать команду "npx eslint index.js". NPX найдет и запустит установленный модуль "eslint" без необходимости устанавливать его глобально на компьютере.

Также NPX позволяет использовать различные версии NodeJS для запуска команд. Например, чтобы запустить команду с использованием NodeJS версии 10, можно использовать команду "npx -p node@10 command".

**43. NodeJS. Работа в консольном режиме. Аргументы командной строки.**

NodeJS — это среда выполнения JavaScript, которая позволяет запускать JavaScript-код вне браузера. В NodeJS можно работать как в браузерной консоли, так и в консольном режиме.

Консольный режим NodeJS позволяет выполнять JavaScript-код из командной строки. Для запуска скрипта нужно открыть терминал, перейти в директорию с файлом скрипта и выполнить команду "node имя_файла.js". NodeJS выполнит скрипт и выведет результат в консоль.

Аргументы командной строки — это параметры, передаваемые при запуске скрипта из командной строки. Они могут быть использованы для передачи данных в скрипт или для настройки его работы. Аргументы командной строки передаются в виде строковых значений и могут быть доступны в скрипте через объект process.argv.

Например, чтобы передать аргументы "hello" и "world" при запуске скрипта, нужно выполнить команду "node имя_файла.js hello world". В скрипте можно получить эти аргументы следующим образом:

```
const args = process.argv.slice(2); // удаляем первые два элемента (node и имя файла)
console.log(args); // выведет ["hello", "world"]
```

**44. NodeJS. Модуль readline.**

Модуль readline в NodeJS предоставляет интерфейс для чтения данных из потока ввода (например, из консоли) построчно. Он может быть использован для создания интерактивных консольных приложений.

Для использования модуля readline нужно его подключить:

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

Затем можно создать интерфейс чтения данных из потока ввода:

```
const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});
```

Этот интерфейс будет читать данные из стандартного потока ввода (process.stdin) и выводить результаты в стандартный поток вывода (process.stdout).

Чтение данных из потока ввода происходит асинхронно, поэтому для получения данных нужно использовать колбэк-функцию:

```
rl.question('What is your name? ', (answer) => {
  console.log(`Hello, ${answer}!`);
  rl.close();
});
```

В этом примере мы задаем вопрос пользователю и ждем ответа. После получения ответа мы выводим приветствие и закрываем интерфейс чтения данных.

Модуль readline также предоставляет другие методы для работы с потоком ввода, например, для чтения данных построчно или для установки курсора в консоли.

**45. NodeJS. Цикл событий, стек вызовов, таймеры**

NodeJS работает в едином потоке, который обрабатывает события и выполняет задачи в цикле событий (event loop). Цикл событий состоит из нескольких фаз, каждая из которых обрабатывает определенный тип задач.

Стек вызовов (call stack) используется для хранения вызовов функций и их контекста выполнения. Каждый новый вызов функции добавляется в вершину стека, а при завершении функции она удаляется из стека.

Таймеры в NodeJS позволяют запускать функции через определенный промежуток времени или с задержкой. Для этого можно использовать функции setTimeout и setInterval:

```
setTimeout(() => {
  console.log('Hello after 1 second');
}, 1000);

setInterval(() => {
  console.log('Hello every 2 seconds');
}, 2000);
```

Первый параметр функций setTimeout и setInterval - это функция, которую нужно выполнить. Второй параметр - это время задержки в миллисекундах (для setTimeout) или интервал между запусками функции в миллисекундах (для setInterval).

Таймеры в NodeJS работают асинхронно, поэтому они не блокируют выполнение кода. Когда таймер срабатывает, соответствующая функция добавляется в очередь задач (task queue) цикла событий, и ее выполнение начнется только после того, как стек вызовов будет пустым.

Также в NodeJS есть другие типы таймеров, например, setImmediate, который позволяет запустить функцию в конце текущей фазы цикла событий, и process.nextTick, который позволяет добавить функцию в начало очереди задач цикла событий.

46. **NodeJS. Работа с файловой системой. Модуль fs.**

NodeJS позволяет работать с файловой системой с помощью модуля fs. Данный модуль предоставляет различные методы для чтения, записи, создания, удаления и переименования файлов и директорий.

Например, чтобы прочитать содержимое файла, можно использовать метод fs.readFile:

```
const fs = require('fs');
fs.readFile('file.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

Первый параметр метода — это имя файла, второй параметр — это кодировка файла (если не указана, то данные будут прочитаны в буфер), третий параметр — это функция обратного вызова, которая будет выполнена после завершения операции чтения файла. В случае ошибки в функцию обратного вызова будет передан объект ошибки.

Аналогично можно использовать методы fs.writeFile и fs.appendFile для записи и добавления данных в файл:

```
fs.writeFile('file.txt', 'Hello, world!', (err) => {
  if (err) throw err;
  console.log('Data has been written to file');
});
fs.appendFile('file.txt', '\nNew data', (err) => {
  if (err) throw err;
  console.log('Data has been appended to file');
});
```

Методы fs.mkdir и fs.rmdir позволяют создавать и удалять директории:

```
fs.mkdir('newDir', (err) => {
  if (err) throw err;
  console.log('Directory has been created');
});
fs.rmdir('newDir', (err) => {
  if (err) throw err;
  console.log('Directory has been deleted');
});
```

Метод fs.rename позволяет переименовывать файлы и директории:

```
fs.rename('oldName.txt', 'newName.txt', (err) => {
```

```
  if (err) throw err;
  console.log('File has been renamed');
});
```

**47. NodeJS. Асинхронное программирование. Промисы (Promises).**

NodeJS позволяет работать с асинхронным кодом с помощью колбеков и промисов (Promises). Промисы позволяют написать более чистый и понятный код, упрощают обработку ошибок и позволяют использовать цепочки вызовов.

Промис — это объект, который представляет результат асинхронной операции. Он имеет три состояния: ожидание (pending), выполнено (fulfilled) и отклонено (rejected). Когда промис создается, он находится в состоянии ожидания. Когда операция завершается успешно, промис переходит в состояние выполнено и возвращает результат операции. Если операция завершается с ошибкой, промис переходит в состояние отклонено и возвращает объект ошибки.

Пример использования промиса для чтения файла:

```
const fs = require('fs');
const readFile = (filename) => {
  return new Promise((resolve, reject) => {
    fs.readFile(filename, 'utf8', (err, data) => {
      if (err) {
        reject(err);
      } else {
        resolve(data);
      }
    });
  });
};
readFile('file.txt')
  .then((data) => {
    console.log(data);
  })
  .catch((err) => {
    console.error(err);
  });
```

В данном примере функция readFile возвращает новый промис. Внутри промиса выполняется операция чтения файла. Если операция завершается успешно, вызывается функция resolve с результатом операции. Если операция завершается с ошибкой, вызывается функция reject с объектом ошибки.

Для обработки результатов операции используются методы then и catch. Метод then вызывается при успешном выполнении операции и принимает в качестве параметра функцию, которая будет вызвана с результатом операции. Метод catch вызывается при возникновении ошибки и принимает в качестве параметра функцию, которая будет вызвана с объектом ошибки.

Пример использования цепочки промисов для чтения и записи файла:

```
const fs = require('fs');
const readFile = (filename) => {
  return new Promise((resolve, reject) => {
    fs.readFile(filename, 'utf8', (err, data) => {
      if (err) {
        reject(err);
      } else {
        resolve(data);
      }
    });
  });
};
const writeFile = (filename, data) => {
  return new Promise((resolve, reject) => {
    fs.writeFile(filename, data, (err) => {
      if (err) {
        reject(err);
      } else {
        resolve();
      }
    });
  });
};
readFile('file.txt')
  .then((data) => {
    return writeFile('newFile.txt', data);
  })
  .then(() => {
    console.log('Data has been written to file');
  })
  .catch((err) => {
    console.error(err);
  });
```

В данном примере функции readFile и writeFile возвращают новые промисы. Цепочка вызовов начинается с чтения файла. Если операция завершается успешно, вызывается метод then, который возвращает новый промис для записи файла. Если операция записи завершается успешно, вызывается еще один метод then, который выводит сообщение об успешной записи. Если возникает ошибка, вызывается метод catch.

**48. NodeJS. Модуль http/https. Создание сервера.**

Модуль http/https — это встроенный модуль NodeJS, который позволяет создавать серверы и обрабатывать HTTP/HTTPS запросы.

Для создания сервера необходимо использовать метод createServer, который принимает в качестве параметра функцию-обработчик запросов. Функция-обработчик должна принимать два параметра: объект запроса (request) и объект ответа (response).

Пример создания HTTP сервера:

```
const http = require('http');
const server = http.createServer((request, response) => {
  response.writeHead(200, {'Content-Type': 'text/plain'});
  response.end('Hello World!');
});
server.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

В данном примере создается HTTP сервер, который отвечает на все запросы сообщением "Hello World!". Метод listen указывает на порт, на котором будет запущен сервер.

Для создания HTTPS сервера необходимо использовать модуль https и передать в метод createServer параметры для создания сертификата и приватного ключа.

Пример создания HTTPS сервера:

```
const https = require('https');
const fs = require('fs');
const options = {
  key: fs.readFileSync('server.key'),
  cert: fs.readFileSync('server.crt')
};
const server = https.createServer(options, (request, response) => {
  response.writeHead(200, {'Content-Type': 'text/plain'});
  response.end('Hello World!');
});
server.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

В данном примере создается HTTPS сервер, который использует сертификат и приватный ключ, указанные в параметрах options. Сертификат и приватный ключ должны быть сохранены в файлы server.crt и server.key соответственно.

Для обработки запросов на сервере можно использовать различные модули, например, модуль express. Он позволяет упростить создание маршрутов и обработчиков запросов.

**49. NodeJS. Статические сайт.**

Статический сайт — это сайт, который состоит из HTML, CSS и JavaScript файлов, которые не изменяются динамически на сервере. Такой сайт может быть размещен на любом веб-сервере и не требует установки дополнительных программ.

Для создания статического сайта на NodeJS можно использовать модуль http-server. Он позволяет запустить локальный веб-сервер, который будет отдавать статические файлы.

Установка модуля http-server:

npm install http-server -g

После установки можно запустить локальный сервер командой:

http-server [path] [options]

где [path] - путь к папке с файлами сайта, [options] - дополнительные параметры запуска сервера (например, порт).

Пример запуска сервера:

http-server ./public -p 3000

В данном примере запускается сервер на порту 3000, который отдает файлы из папки ./public.

Для более продвинутого управления статическим сайтом можно использовать модули express или koa. Они позволяют создавать маршруты, обрабатывать запросы и использовать шаблонизаторы для генерации HTML страниц.

**50. NodeJS. Фреймворк Express.**

Express - это минималистичный и гибкий веб-фреймворк для NodeJS, который позволяет создавать веб-приложения и API. Он предоставляет удобный интерфейс для работы с HTTP-запросами, маршрутизацией, обработкой ошибок и многим другим.

Установка Express:

npm install express

Пример создания сервера с использованием Express:

```
const express = require('express');
const app = express();
app.get('/', (req, res) => {
  res.send('Hello World!');
});
app.listen(3000, () => {
  console.log('Server started on port 3000');
});
```

В данном примере создается сервер на порту 3000, который отвечает на GET-запросы по пути '/'. Функция обратного вызова (callback) выполняется при запуске сервера.

Express также позволяет использовать маршруты для обработки запросов:

```
app.get('/users', (req, res) => {
  res.send('List of users');
});
app.post('/users', (req, res) => {
  res.send('Create a new user');
});
app.put('/users/:id', (req, res) => {
  const id = req.params.id;
  res.send(`Update user with ID ${id}`);
});
app.delete('/users/:id', (req, res) => {
  const id = req.params.id;
```

```
  res.send(`Delete user with ID ${id}`);
});
```

В данном примере создаются маршруты для работы с пользователями: получение списка пользователей (GET), создание нового пользователя (POST), обновление пользователя по ID (PUT) и удаление пользователя по ID (DELETE).

Express также позволяет использовать шаблонизаторы для генерации HTML страниц. Для этого нужно установить соответствующий модуль (например, EJS или Pug) и настроить Express на использование этого шаблонизатора:

```
const express = require('express');
const app = express();
const ejs = require('ejs');
app.set('view engine', 'ejs');
app.get('/', (req, res) => {
  const data = { title: 'Home page' };
  res.render('index', data);
});
app.listen(3000, () => {
  console.log('Server started on port 3000');
});
```

В данном примере устанавливается шаблонизатор EJS, настраивается Express на его использование и создается маршрут для главной страницы, которая генерируется из шаблона index.ejs с передачей данных в виде объекта data.

**51. NodeJS. Фреймворк Express. Конвейер обработки запроса и middleware.**

Конвейер обработки запроса (request pipeline) - это последовательность middleware функций, которые обрабатывают запросы в Express. Middleware - это функции, которые имеют доступ к объектам запроса (request) и ответа (response) и могут выполнять различные операции, например, обработку данных, проверку аутентификации, логирование и т.д.

Пример middleware функции для логирования запросов:

```
const express = require('express');
const app = express();
app.use((req, res, next) => {
  console.log(`${req.method} ${req.url}`);
  next();
});
app.get('/', (req, res) => {
  res.send('Hello World!');
});
app.listen(3000, () => {
  console.log('Server started on port 3000');
});
```

В данном примере создается middleware функция, которая выводит в консоль метод и URL запроса. Функция передается в метод use() и будет вызываться для каждого запроса перед выполнением соответствующего маршрута.

Middleware функции могут быть созданы как отдельные модули и подключены в приложение:

```
// middleware.js
module.exports = (req, res, next) => {
  console.log(`${req.method} ${req.url}`);
  next();
}
// app.js
const express = require('express');
const app = express();
const middleware = require('./middleware');
app.use(middleware);
app.get('/', (req, res) => {
  res.send('Hello World!');
});
app.listen(3000, () => {
  console.log('Server started on port 3000');
});
```

В данном примере middleware функция вынесена в отдельный модуль middleware.js и подключается в приложение через метод use().

Middleware функции могут иметь различные параметры и вызываться в определенном порядке:

```
const express = require('express');
const app = express();
const middleware1 = (req, res, next) => {
  console.log('Middleware 1');
  next();
}
const middleware2 = (req, res, next) => {
  console.log('Middleware 2');
  next();
}
app.use(middleware1);
app.use(middleware2);
app.get('/', (req, res) => {
  res.send('Hello World!');
});
app.listen(3000, () => {
  console.log('Server started on port 3000');
});
```

В данном примере создаются две middleware функции, которые выводят в консоль свое название. Функции передаются в метод use() в порядке их вызова. При выполнении запроса сначала будет вызвана middleware1, затем middleware2, а затем выполнится соответствующий маршрут.

**52. NodeJS. Фреймворк Express. Статические файлы.**

Статические файлы — это файлы, которые не изменяются в процессе работы приложения и могут быть отправлены клиенту напрямую без обработки на сервере. К таким файлам относятся, например, изображения, стили CSS, скрипты JavaScript и т.д.

В Express для работы со статическими файлами используется встроенный middleware функция express.static(). Она принимает путь к директории, в которой хранятся статические файлы, и возвращает middleware функцию для обработки запросов на эти файлы.

Пример использования express.static():

```
const express = require('express');
const app = express();
app.use(express.static('public'));
app.get('/', (req, res) => {
  res.send('Hello World!');
});
app.listen(3000, () => {
  console.log('Server started on port 3000');
});
```

В данном примере создается директория public, в которой хранятся статические файлы. Middleware функция express.static() передается методу use() и указывается путь к этой директории. Теперь все запросы на статические файлы будут обрабатываться автоматически.

Например, если в директории public есть файл index.html, то он будет доступен по адресу http://localhost:3000/index.html.

Также можно указать путь к директории относительно корневой директории приложения:

```
app.use(express.static(__dirname + '/public'));
```

В данном случае используется переменная __dirname, которая содержит абсолютный путь к директории, в которой находится текущий исполняемый файл.

Можно также указать путь к директории с помощью объекта:

```
app.use('/static', express.static('public'));
```

В данном примере указывается, что все запросы на статические файлы должны начинаться с пути /static. Например, файл index.html будет доступен по адресу http://localhost:3000/static/index.html.

**53. NodeJS. Фреймворк Express. Маршрутизация.**

Express — это минималистичный, гибкий и быстрый фреймворк для Node.js, который позволяет создавать веб-приложения и API. Маршрутизация в Express предназначена для определения того, как приложение должно обрабатывать запросы клиента на определенные URL-адреса.

Для создания маршрута в Express используется метод app.METHOD(PATH, HANDLER), где METHOD - это HTTP-метод (get, post, put, delete и т.д.), PATH - это путь на сервере, а HANDLER - это функция, которая будет вызываться при обработке запроса. Например, чтобы создать маршрут для GET-запроса на путь '/', можно использовать следующий код:

```
app.get('/', function(req, res) {
  res.send('Hello, World!');
});
```

Этот код создает маршрут для GET-запроса на путь '/', который возвращает ответ 'Hello, World!'.

Вы также можете использовать параметры маршрута, чтобы обрабатывать динамические URL-адреса. Например, чтобы создать маршрут для GET-запроса на путь '/users/:userId', где userId является параметром, можно использовать следующий код:

```
app.get('/users/:userId', function(req, res) {
  res.send('User ID: ' + req.params.userId);
});
```

Этот код создает маршрут для GET-запроса на путь '/users/:userId', который возвращает ответ 'User ID: ' и значение параметра userId, переданного в URL-адресе.

Маршрутизация в Express очень мощный инструмент, который позволяет создавать гибкие и масштабируемые приложения и API.

**54. NodeJS. Фреймворк Express. Разработка REST API.**

Express является одним из наиболее популярных фреймворков Node.js, который позволяет создавать REST API. REST API представляет собой программный интерфейс, который позволяет взаимодействовать с приложением или веб-сайтом через HTTP-запросы.

Для создания REST API с помощью Express, вы можете использовать метод app.METHOD(PATH, HANDLER), где METHOD - это HTTP-метод (GET, POST, PUT, DELETE и т.д.), PATH - это путь запроса, а HANDLER - функция, которая обрабатывает запрос и возвращает ответ.

Например, чтобы создать маршрут для GET-запроса, вы можете использовать следующий код:

```
app.get('/users', (req, res) => {
  res.send('Список всех пользователей');
});
```

В этом примере мы создали маршрут для GET-запроса по пути /users, который возвращает строку "Список всех пользователей".

Вы также можете создавать маршруты с параметрами. Например, чтобы получить информацию о конкретном пользователе, вы можете использовать следующий код:

```
app.get('/users/:id', (req, res) => {
  const userId = req.params.id;
  res.send(`Информация о пользователе с ID ${userId}`);
});
```

В этом примере мы создали маршрут для GET-запроса по пути /users/:id, где :id - это параметр пути, который содержит идентификатор пользователя. Мы извлекаем значение параметра id из объекта req.params и используем его для формирования ответа.

Использование маршрутизации является важным компонентом создания гибких и масштабируемых приложений и API. Он позволяет управлять различными запросами и параметрами, обрабатывать ошибки и возвращать соответствующие ответы.

**55. NodeJS. Тестирование приложения.**

Тестирование - важная часть создания любого приложения, и приложения NodeJS не являются исключением. Существует несколько инструментов и фреймворков, доступных для тестирования приложений NodeJS, включая Mocha, Jest и Cypress.

Mocha — это популярный фреймворк тестирования для NodeJS, который предоставляет гибкую и настраиваемую среду тестирования. Mocha позволяет разработчикам писать тесты, используя различные стили, такие как BDD (разработка, основанная на поведении) и TDD (разработка, основанная на тестировании), и предоставляет ряд встроенных утверждений для тестирования различных аспектов приложения.

Jest - еще один популярный фреймворк тестирования для NodeJS, который предоставляет простую и удобную в использовании среду тестирования. Jest включает встроенную поддержку для издевательства и слежки за функциями, а также ряд встроенных средств сопоставления для тестирования различных аспектов приложения.

Cypress — это платформа тестирования для NodeJS, которая фокусируется на сквозном тестировании веб-приложений. Cypress предоставляет ряд инструментов для моделирования пользовательских взаимодействий

**56. NodeJS. Фреймворк Express. Взаимодействие с базами данных.**

NodeJS — это среда выполнения JavaScript, которая позволяет разработчикам создавать серверные приложения с использованием JavaScript. Express - один из самых популярных фреймворков для создания веб-приложений на NodeJS.

Express предоставляет множество инструментов для работы с базами данных, включая библиотеки объектно-реляционного отображения (ORM). Библиотеки ORM позволяют разработчикам работать с базами данных на более высоком уровне абстракции, используя объекты и методы вместо написания SQL-запросов.

Некоторые из наиболее популярных ORM-библиотек для NodeJS и Express включают Sequelize, TypeORM и Prisma.

Sequelize — это ORM-библиотека для NodeJS, которая поддерживает работу с различными базами данных, включая MySQL, PostgreSQL и SQLite. Sequelize позволяет разработчикам определять модели данных и взаимосвязи между ними, а также выполнять операции CRUD (создавать, читать, обновлять, удалять) с использованием объектов и методов.

TypeORM — это библиотека ORM для NodeJS и TypeScript, которая также поддерживает работу с различными базами данных. TypeORM позволяет разрабатывать

**57. NodeJS. Фреймворк Express. Взаимодействие с базами данных с использование ORM-библиотек.**

NodeJS — это среда выполнения JavaScript, которая позволяет разрабатывать серверные приложения на языке JavaScript. Фреймворк Express — это один из наиболее популярных фреймворков для разработки веб-приложений на NodeJS.

Express предоставляет множество инструментов для работы с базами данных, включая ОРМ-библиотеки (Object-Relational Mapping). ОРМ-библиотеки позволяют разработчикам работать с базами данных на более высоком уровне абстракции, используя объекты и методы, вместо написания SQL-запросов.

Некоторые из наиболее популярных ОРМ-библиотек для NodeJS и Express включают Sequelize, TypeORM и Prisma.

Sequelize — это ORM-библиотека для NodeJS, которая поддерживает работу с различными базами данных, включая MySQL, PostgreSQL и SQLite. Sequelize позволяет разработчикам определять модели данных и связи между ними, а также выполнять операции CRUD (Create, Read, Update, Delete) с помощью объектов и методов.

TypeORM — это ORM-библиотека для NodeJS и TypeScript, которая также поддерживает работу с различными базами данных. TypeORM позволяет разработчикам определять модели данных с помощью классов и декораторов, а также выполнять операции CRUD с помощью объектов и методов.

Prisma — это ORM-библиотека для NodeJS, которая предоставляет более высокий уровень абстракции для работы с базами данных. Prisma позволяет разработчикам определять модели данных и связи между ними, а также выполнять операции CRUD с помощью методов, которые генерируются автоматически на основе определения моделей данных.

В целом, использование ОРМ-библиотек в NodeJS и Express позволяет разработчикам упростить работу с базами данных, повысить производительность и безопасность приложений. Разработчики должны выбирать подходящую ОРМ-библиотеку в зависимости от конкретных потребностей своего приложения и базы данных.

**58. NodeJS. Авторизация и аутентификация.**

Авторизация и аутентификация — это процессы, которые используются для обеспечения безопасности веб-приложений.

Аутентификация — это процесс проверки подлинности пользователя. Когда пользователь пытается войти в систему, он должен предоставить правильные учетные данные (логин и пароль). Система затем проверяет эти данные на соответствие с данными, хранящимися в базе данных, и, если они совпадают, пользователь получает доступ к системе.

Авторизация — это процесс определения того, что пользователь может делать в системе после того, как он был аутентифицирован. Авторизация определяет права доступа пользователя к определенным функциям и ресурсам системы. Например, пользователь может иметь право только на чтение информации, но не на ее изменение или удаление.

NodeJS предоставляет множество инструментов для реализации авторизации и аутентификации в веб-приложениях. Некоторые из наиболее распространенных инструментов включают Passport, JWT (JSON Web Tokens), OAuth и Firebase Authentication.

Passport — это middleware для NodeJS, который позволяет легко реализовать аутентификацию на основе различных стратегий (например, локальной аутентификации, аутентификации через социальные сети и т.д.). Passport также обеспечивает механизмы для управления сеансами пользователей и авторизации.

JSON Web Tokens (JWT) — это открытый стандарт (RFC 7519) для создания токенов доступа, которые могут использоваться для аутентификации и авторизации в веб-приложениях. JWT состоит из трех частей: заголовка, полезной нагрузки и подписи. Полезная нагрузка содержит информацию о пользователе, которая может быть использована для авторизации в системе.

OAuth — это протокол авторизации, который позволяет пользователям дать разрешение на доступ к своим данным на сторонних сайтах без необходимости предоставлять свои учетные данные. OAuth используется многими крупными компаниями, такими как Facebook, Google и Twitter, для авторизации пользователей на своих платформах.

Firebase Authentication — это сервис аутентификации, предоставляемый Google Firebase. Firebase Authentication обеспечивает простой и безопасный способ аутентификации пользователей в веб-приложениях с использованием различных методов (например, электронной почты и пароля, аутентификации через социальные сети и т.д.). Firebase Authentication также обеспечивает механизмы для управления сеансами пользователей и авторизации.

В целом, NodeJS предоставляет множество инструментов для реализации безопасности веб-приложений, включая авторизацию и аутентификацию. Разработчики должны выбрать подходящий инструмент в зависимости от конкретных потребностей своего приложения.