1. **Основы Node.js**:
   * Что такое Node.js и его архитектура.

Node.js - это программная платформа, которая предоставляет среду для выполнения кода JavaScript на стороне сервера.

Node.js это однопоточная событийно-ориентированная система.

* + Установка Node.js и npm (Node Package Manager).

Установка Node.js занимает всего несколько минут независимо от операционной системы. Сперва перейдите на официальный сайт платформы, где автоматически определяется тип и разрядность вашей операционной системы. На сайте предлагается установить Node.js либо самой последней (Current), либо стабильной версии (LTS).

Установка Node.js занимает всего несколько минут независимо от операционной системы. Сперва перейдите на официальный сайт платформы, где автоматически определяется тип и разрядность вашей операционной системы. На сайте предлагается установить Node.js либо самой последней (Current), либо стабильной версии (LTS).

* + Основы работы с REPL (Read-Eval-Print Loop).

1. **Модули и пакеты**:
   * Импорт и экспорт модулей в Node.js.

Для загрузки модулей применяется функция require(), в которую передается название модуля.

* + Использование сторонних пакетов с помощью npm.

Для установки модулей используется команда npm install

* + Понимание разницы между глобальными и локальными пакетами.

Глобальные пакеты в Node.js устанавливаются глобально на компьютере и доступны из любой директории. Они обычно содержат инструменты командной строки, такие как ноды или менеджеры пакетов, которые нужны для работы с проектами Node.js.

Локальные пакеты, как следует из названия, устанавливаются локально внутри конкретного проекта. Они перечислены в файле package.json и устанавливаются с помощью менеджера пакетов, такого как npm или yarn. Локальные пакеты содержат зависимости, необходимые для функционирования конкретного проекта, такие как фреймворки, библиотеки или вспомогательные утилиты.

**Система файлов и потоки (File System and Streams)**:

* + Работа с файлами и директориями в Node.js.

fs.ReadFile() для чтения файла

* + Использование потоков для обработки данных.

Обмен данными осуществляется частями, не дожидаясь полной загрузки, что позволяет эффективно использовать оперативную память. Например, вам нужно найти определенную строку в файле большого объема. Использование потока позволяет получать данные по частям и искать необходимую строку в этих частях, при совпадении дальнейшее чтение файла можно прекратить, сэкономив при этом время и производительные ресурсы.

1. **Асинхронное программирование**:
   * Обзор колбэков (callbacks) и промисов (Promises).

Чтобы избежать блокировки event loop и тем самым создавать высокопроизводительные Node.js приложения, используйте асинхронные методы. Почти наверняка каждому синхронному блокирующему методу соответствует метод с точно таким же функционалом, но асинхронный.

const fs = require('fs');

fs.readFile('./file.txt', (err, data) => {

if (err) throw err;

console.log(data);

});

console.log('Before reading file');

В этом примере процесс не будет дожидаться окончания чтения файла, а продолжит выполнять следующий код. А когда содержимое файла будет получено - вызовет переданную методу readFile() callback-функцию.

* + Применение async/await для управления асинхронным кодом.

Async – применяется для объявления ассинхронной функции. Await для ожидания выполнения асинхронной функции.

1. **Express.js и веб-разработка**:

Express - это фреймворк для Node.js, который реализовывает слой функций, необходимых для создания эффективных приложений и API. Его использование значительно сокращает написание кода, а, значит, уменьшается затрачиваемое на разработку время.

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

const host = '127.0.0.1';

const port = 7000;

app.get('/home', (req, res) => {

res.status(200).type('text/plain');

res.send('Home page');

});

app.get('/about', (req, res) => {

res.status(200).type('text/plain');

res.send('About page');

});

app.post('/api/admin', (req, res) => {

res.status(200).type('text/plain');

res.send('Create admin request');

});

app.post('/api/user', (req, res) => {

res.status(200).type('text/plain');

res.send('Create user request');

});

app.use((req, res, next) => {

res.status(404).type('text/plain');

res.send('Not found');

});

app.listen(port, host, function () {

console.log(`Server listens http://${host}:${port}`);

});

1. **WebSocket и реальное время**:

WebSocket - это протокол передачи данных, основанный на протоколе TCP обеспечивающий обмен сообщениями между клиентом и сервером в режиме реального времени.

Для подключения WebSocket на клиентской стороне используется модуль socket.io-client, экземпляру которого передается адрес сервера, с которым необходимо установить соединение по WebSocket.

При установке соединения между клиентом и сервером Node.js по WebSocket генерируется событие connection, которое обрабатывается с помощью метода on() модуля socket.io. Передаваемая вторым параметром методу on() callback-функция единственным параметром принимает экземпляр соединения

**Базы данных**:

Рассмотрим подключение к базе данных PostgreSQL.

*connection.js*

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | const Sequelize = require('sequelize');  const sequelize = new Sequelize(  'db\_name',  'user',  'password',  {  dialect: 'postgres',  }  );  sequelize  .authenticate()  .then(() => console.log('Connected.'))  .catch((err) =>  console.error('Connection error: ', err)  ); |

Модели используются для описания структуры таблицы. Одна модель описывает одну таблицу.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | const Sequelize = require('sequelize');  class Book extends Sequelize.Model {}  Book.init(  {  id: {  type: Sequelize.NUMBER,  primaryKey: true,  },  title: {  type: Sequelize.STRING,  allowNull: false,  comment: "Book 's title",  },  author: {  type: Sequelize.STRING,  field: '\_author',  },  description: {  type: Sequelize.TEXT,  },  publishDate: {  type: Sequelize.DATE,  defaultValue: Sequelize.NOW,  },  },  { sequelize, modelName: 'book' }  ); |

Модель создается с помощью метода init() класса, который является дочерним по отношению к Sequelize.Model. Метод init() принимает два объекта:

* объект с описанием полей таблицы;
* конфигурация создаваемой модели и соответствующей ей таблицы.

Описание поля задается объектом со следующими свойствами:

* type - тип поля;
* defaultValue - значение поля по умолчанию;
* primaryKey - булевое значение, если true, то поле является первичным ключом (по умолчанию false);
* autoIncrement - булевое значение, если true, то при добавлении новой записи значение поля будет значение предыдущей записи этого поля плюс единица (по умолчанию false);
* allowNull - булевое значение, если false, запрещает создавать новую запись с этим пустым полем;
* unique - булевое значение, если true, то значение указанное для этого поля в записи должно быть уникальным в пределах таблицы (по умолчанию false);
* comment - комментарий к полю;
* field - если указано, то в качестве названия поле будет использоваться именно это значение, а не ключ;
* validate - объект с заданием для поля валидаторов, с полным списком можно ознакомиться в [документации](http://docs.sequelizejs.com/manual/models-definition.html#validations);
* get() - функция, которая модифицирует значение поля при чтении записи;
* set() - функция, преобразующая передаваемое значение при сохранении записи.

1. **Аутентификация и безопасность**:

В Node.js **аутентификация** чаще всего реализуется с использованием функций промежуточной обработки, предоставляемых библиотекой passport.js, которая устанавливается с помощью npm.

Пример

const express = require('express'),

app = express(),

session = require('express-session'),

passport = require('passport'),

localStrategy = require('passport-local').Strategy,

flash = require('connect-flash');

const host = '127.0.0.1';

const port = 7000;

function checkAuth() {

return app.use((req, res, next) => {

if (req.user) next();

else res.redirect('/login');

});

}

passport.serializeUser((user, done) => done(null, user));

passport.deserializeUser((user, done) => done(null, user));

app.use(express.json());

app.use(express.urlencoded({ extended: true }));

app.use(session({ secret: 'you secret key' }));

app.use(flash());

app.use(passport.initialize());

app.use(passport.session());

passport.use(

new localStrategy((user, password, done) => {

if (user !== 'test\_user')

return done(null, false, {

message: 'User not found',

});

else if (password !== 'test\_password')

return done(null, false, {

message: 'Wrong password',

});

return done(null, { id: 1, name: 'Test', age: 21 });

})

);

app.get('/login', (req, res) => {

res.send('Login page. Please, authorize.');

});

app.use((req, res, next) => {

if (req.user) next();

else res.redirect('/login');

});

app.post(

'/login',

passport.authenticate('local', {

successRedirect: '/home',

failureRedirect: '/login',

failureFlash: true,

})

);

app.get('/home', checkAuth(), (req, res) => {

res.send("Home page. You're authorized.");

});

app.listen(port, host, function () {

console.log(`Server listens http://${host}:${port}`);

});

**Тестирование**:

**Пример теста**

var operations = require('./operations');

it('should multiply two numbers', function () {

var expectedResult = 15;

var result = operations.multiply(3, 5);

if (result !== expectedResult) {

throw new Error(

`Expected ${expectedResult}, but got ${result}`

);

}

});

1. **Деплоймент и управление серверами**:

Для обеспечения непрерывной работоспособности Node.js сервера нужно либо постоянно держать открытой консоль, либо использовать менеджер процессов pm2. Он имеет встроенный балансировщик нагрузки, позволяет следить за потребляемыми ресурсами запущенных процессов, автоматически перезапускать процессы после системного сбоя и т. д.

Менеджер процессов pm2 имеется в репозитории npm и должен быть установлен в системе глобально.

**Микросервисная архитектура**:

* + [Введение](http://m.vksit.ru/mod/page/view.php?id=5787) в микросервисы и их реализация с использованием Node.js.
  + Взаимодействие между микросервисами.

1. **Разработка API и RESTful сервисы**:
2. Основная идеи REST API заключается в разделении разных операций (чаще всего CRUD) при обращении к одному и тому же URL с помощью HTTP методов, основные из которых:
3. GET - используется для получения данных;
4. POST - используется для создания новой записи(ей);
5. PUT - используется для обновления уже существующей записи(ей);
6. PATCH - используется для обновления, но только тогда, когда изменяется идентификатор записи(ей);
7. DELETE - используется для удаления записи(ей).
8. process.env.NODE\_ENV = 'development';
9. const express = require('express'),
10. app = express(),
11. fs = require('fs');
12. const host = '127.0.0.1';
13. const port = 7000;
14. app.use(express.json());
15. app.use(express.urlencoded({ extended: true }));
16. let file = 'data.json';
17. if ((process.env.NODE\_ENV = 'test'))
18. file = 'data-test.json';
19. app.use((req, res, next) => {
20. fs.readFile(file, (err, data) => {
21. if (err)
22. return res.status(500).send({
23. message: 'Error while getting users',
24. });
25. req.users = JSON.parse(data);
26. next();
27. });
28. });
29. app.route('/api/users')
30. .get((req, res) => {
31. if (req.query.id) {
32. if (req.users.hasOwnProperty(req.query.id))
33. return res.status(200).send({
34. data: req.users[req.query.id],
35. });
36. else
37. return res
38. .status(404)
39. .send({ message: 'User not found.' });
40. } else if (!req.users)
41. return res
42. .status(404)
43. .send({ message: 'Users not found.' });
44. return res.status(200).send({ data: req.users });
45. })
46. .post((req, res) => {
47. if (req.body.user && req.body.user.id) {
48. if (req.users.hasOwnProperty(req.body.user.id))
49. return res.status(409).send({
50. message: 'User already exists.',
51. });
52. req.users[req.body.user.id] = req.body.user;
53. fs.writeFile(
54. file,
55. JSON.stringify(req.users),
56. (err, response) => {
57. if (err)
58. return res.status(500).send({
59. message: 'Unable create user.',
60. });
61. return res
62. .status(200)
63. .send({ message: 'User created.' });
64. }
65. );
66. } else
67. return res
68. .status(400)
69. .send({ message: 'Bad request.' });
70. })
71. .put((req, res) => {
72. if (req.body.user && req.body.user.id) {
73. if (!req.users.hasOwnProperty(req.body.user.id))
74. return res
75. .status(404)
76. .send({ message: 'User not found.' });
77. req.users[req.body.user.id] = req.body.user;
78. fs.writeFile(
79. file,
80. JSON.stringify(req.users),
81. (err, response) => {
82. if (err)
83. return res.status(500).send({
84. message: 'Unable update user.',
85. });
86. return res
87. .status(200)
88. .send({ message: 'User updated.' });
89. }
90. );
91. } else
92. return res
93. .status(400)
94. .send({ message: 'Bad request.' });
95. })
96. .delete((req, res) => {
97. if (req.query.id) {
98. if (req.users.hasOwnProperty(req.query.id)) {
99. delete req.users[req.query.id];
100. fs.writeFile(
101. file,
102. JSON.stringify(req.users),
103. (err, response) => {
104. if (err)
105. return res.status(500).send({
106. message:
107. 'Unable delete user.',
108. });
109. return res.status(200).send({
110. message: 'User deleted.',
111. });
112. }
113. );
114. } else
115. return res
116. .status(404)
117. .send({ message: 'User not found.' });
118. } else
119. return res
120. .status(400)
121. .send({ message: 'Bad request.' });
122. });
123. app.listen(port, host, () =>
124. console.log(`Server listens http://${host}:${port}`)
125. );
126. **Работа с**[**события**](http://m.vksit.ru/mod/page/view.php?id=6116)**ми**:
127. Подавляющее большинство функционала Node.js применяет асинхронную событийную архитектуру, которая использует специальные объекты - эмиттеры для генерации различных событий, которые обрабатываются специальными функциями - обработчиками или слушателями событий. Все объекты, которые генерируют события, представляют экземпляры класса EventEmitter.
128. С помощью функции eventEmitter.on() к определенному событию по имени цепляется функция обработчика. Причем для одного события можно указать множество обработчиков. Когда объект EventEmitter генерирует событие, происходит выполнение всех этих обработчиков.
129. Рассмотрим применение объекта EventEmitter и событий. Для этого определим следующий файл app.js:
130. const Emitter = require('events');
131. let emitter = new Emitter();
132. let eventName = 'greet';
133. emitter.on(eventName, function () {
134. console.log('Hello all!');
135. });
136. emitter.on(eventName, function () {
137. console.log('Привет!');
138. });
139. emitter.emit(eventName);

**Дополнительные инструменты и библиотеки**:

* + Изучение других библиотек и инструментов, таких как GraphQL, Redis и др.

1. **Оптимизация и масштабирование**:

Обеспечение высокого уровня производительности и надежности Node.js приложений достигается соблюдением простых рекомендаций и правил.

1. Минимизируйте использование синхронных функций, особенно при выполнении операций ввода/вывод, которые блокируют выполнение цикла событий.

fs.readFileSync('data.txt'); //избегайте

fs.readFile('data.txt', (err, data) => {}); //так правильно

2. Обрабатывайте отдачу статического контента с помощью функций промежуточной обработки.

app.use(express.static(`${\_\_dirname}/assets`));

3. Предусматривайте обработку ошибок при выполнении каждой операции, чтобы предотвратить аварийное завершение процесса и дать пользователю подробное сообщение о том, что он сделал не так и что привело к сбою в работе приложения.

//для синхронных операций

try {

} catch (e) {}

//для асинхронных операций в callback-функции

(err, f) => {

if (err) {

}

};

//при работе с объектами Promise

Promise.then(() => {}).catch((err) => {});

4. Настройте сжатие gzip, которое уменьшает размер передаваемых данных любого формата, что увеличивает скорость обмена данными между клиентом и сервером.

const compression = require('compression');

app.use(compression());

5. Указывайте при запуске Node.js приложения на сервере значение production для переменной среды окружения process.env.NODE\_ENV. В связке с Express активация рабочей среды инициирует сохранение файлов шаблонов и стилей в кэш и генерацию сообщений об ошибке без полного StackTrace. Экспериментально выявлено, что все перечисленное увеличивает быстродействие приложения примерно в три раза. Переменные окружения Node.js задаются в операционной системе.

6. Используйте кластеризацию при запуске приложения для распределения нагрузки между дополнительными экземплярами.

7. Управляйте приложением с помощью менеджера процессов, например, pm2. Использование pm2 позволит вам настроить автоматический перезапуск процесса приложения при возникновении необработанной ошибки и тем самым обеспечит его непрерывную работу. Также менеджер процессов поможет вам отслеживать количество потребляемых ресурсов.