Курс по Node.js

Библиотека Socket.io. Workers в Node.js

[Node.js v14.x]

© geekbrains.ru

На этом уроке

1. Узнаем, что такое веб-сокеты. Рассмотрим их применение в приложениях реального времени.
2. Выясним, как использовать эту технологию в Node.js, используя библиотеку Socket.io.
3. Познакомимся с модулем worker_threads в Node.js.
4. Научимся выполнять ресурсоёмкие вычисления в отдельных потоках, не блокируя основной.

Оглавление

Теория урока

- Введение
- Веб-сокеты
- Библиотека Socket.io
  - Серверная часть Socket.io
  - Клиентская часть Socket.io
  - Рассылка сообщений
- Однопоточность
- Модуль worker_threads
- Модуль crypto
- Заключение

Практическое задание

Глоссарий

Дополнительные материалы

Используемые источники

Теория урока

Введение

В предыдущем уроке мы познакомились с понятием http-сервер и написали собственный сервер, способный принимать различные запросы, обрабатывать их и отвечать на них. Однако есть большой класс приложений, для которых формат «клиент сделал запрос к серверу, сервер ответил, и соединение закрылось» не может обеспечить требуемую функциональность. Самые яркие представители этого класса — мессенджеры.

Чтобы мессенджер работал, а клиент получал сообщения от других клиентов, надо научить сервер передавать данные клиенту. Конечно, клиент может сам делать запросы к серверу с определённой частотой, но тогда останется некоторая задержка. А мессенджеры — это приложения реального времени.

Если частота запросов к серверу со стороны клиента будет слишком большой, то возникает вероятность наложения времени выполнения одних запросов на другие. То есть, когда клиент шлёт запрос, сервер его обрабатывает, и пока он его обрабатывает, клиент присылает новый запрос. Это приводит к тому, что до появления ответа от сервера состояние клиента уже поменяется, и данные будут не актуальны.

Однако сам сервер не инициирует соединение с клиентом. У большинства клиентов динамические IP-адреса, и до них нельзя достучаться снаружи.

В результате для реализации такой функциональности используется схема, когда клиент инициирует TCP-соединение с сервером, и оно поддерживается в течение длительного времени в отличие от обычных http-запросов.

Веб-сокеты

Веб-сокет — это стандарт двухсторонней связи клиента с сервером по TCP-соединению, который совместим с HTTP, предназначенный для обмена сообщениями в режиме реального времени.

Чтобы понять отличие веб-сокетов от http-запросов, рассмотрим две схемы применительно к задаче с мессенджером.

В случае с http-запросами схема работы будет примерно такой:

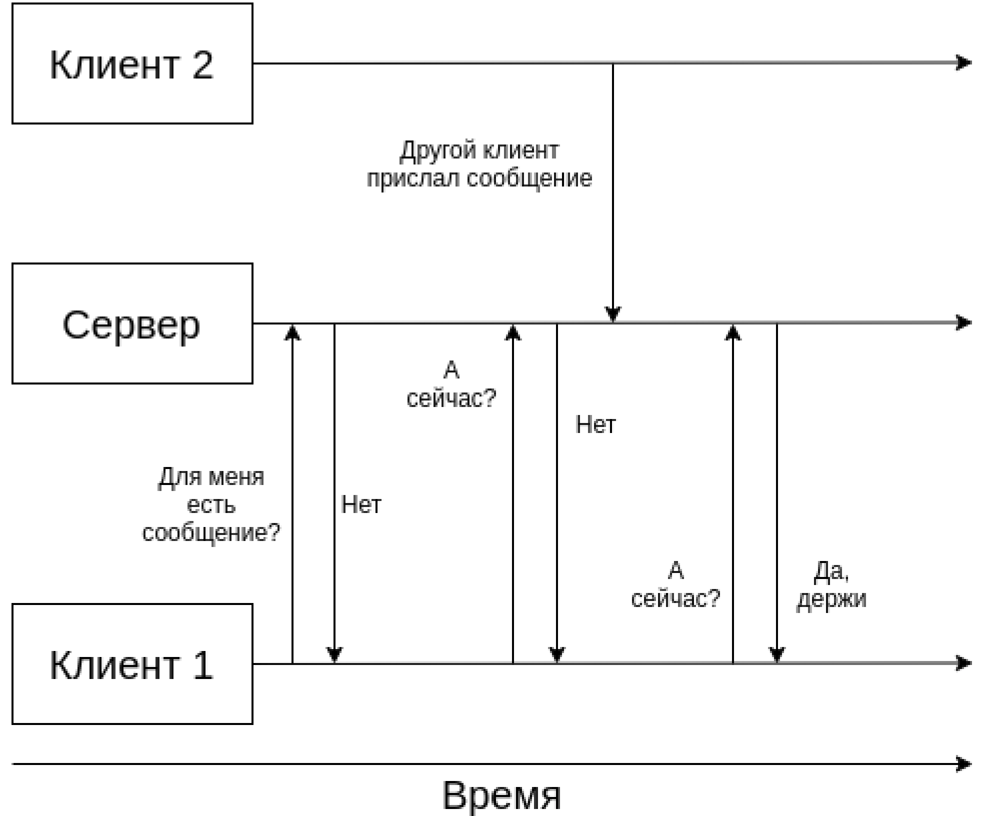

Рис. 1. Получение информации мессенджером посредством http-запросов

На этом рисунке мы видим, что Клиент 1 периодически спрашивает сервер о сообщениях для него. Когда на сервер приходит сообщение от Клиента 2 для Клиента 1 — оно передаётся ему в ответе сервера при следующем опросе. У такого подхода есть несколько недостатков:

1. Бесполезные запросы. Клиент постоянно опрашивает сервер, то есть нагружает его своими запросами, хотя никакой полезной информации в ответах не передаётся.
2. С появления на сервере сообщения для клиента и до момента, когда оно передалось клиенту, наблюдается задержка. Чем ниже частота опроса сервера, тем выше задержка.

Очевидно, что такой подход к построению подобных приложений совершенно нерационален. Взглянем на подход, который подразумевает использование веб-сокетов.

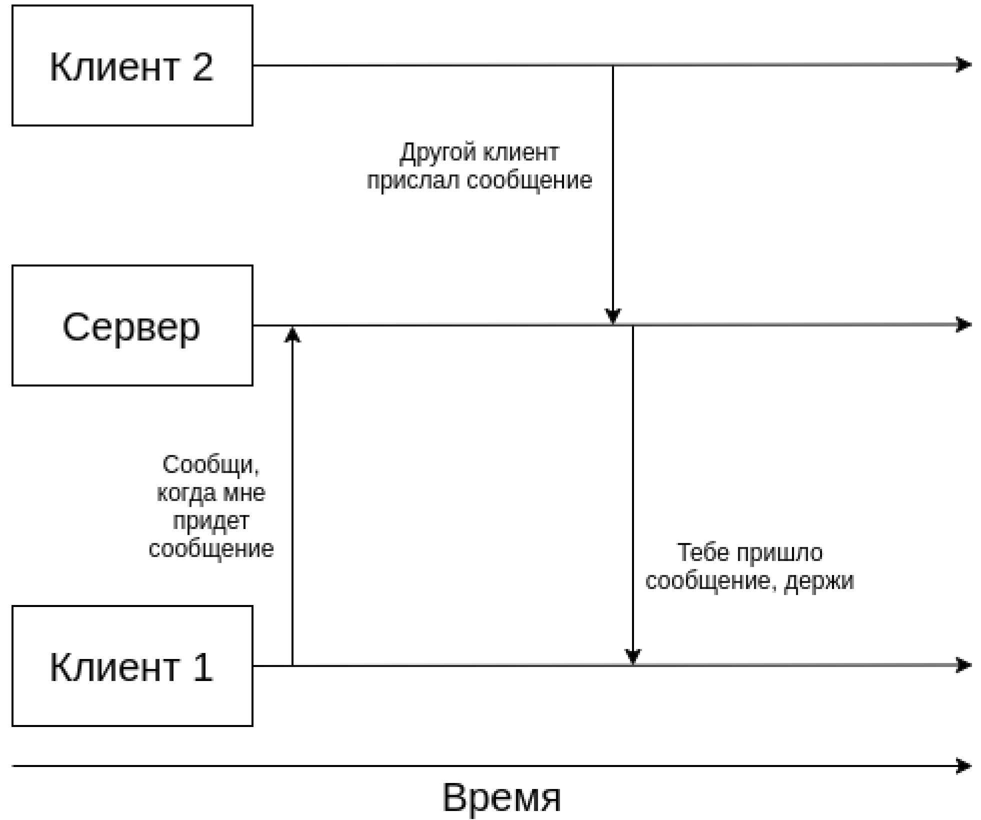

Рис. 2. Получение информации мессенджером посредством веб-сокетов

На рисунке 2 мы видим, что Клиент 1 только один раз устанавливает соединение с сервером. Дальше он просто ждёт, когда ему поступит информация от сервера. Когда на сервер приходит сообщение от Клиента 2, он сразу же отсылает его к Клиенту 1. Это происходит практически моментально, однако на рисунке всё равно изображается некоторая задержка. Её величина зависит от загрузки сервера, его вычислительных возможностей, от скорости соединения с Клиентом 1 и других технических факторов. Однако это совершенно не та задержка, когда сервер ждёт входящего запроса, чтобы передать в ответ данные, как это изображено на рисунке 1.

Таким образом, естественной сферой применения веб-сокетов становятся те приложения, которые работают в режиме реального времени:

1. Чат-приложения (мессенджеры).
2. Навигаторы.
3. Приложения для интернета вещей (IoT-приложения).
4. Многопользовательские игры.
5. Прочее.

Сегодня веб-сокеты поддерживаются всеми современными браузерами: Google Chrome, Safari, Mozilla, Opera, Microsoft Edge и т. д.

Библиотека Socket.io

Библиотека Socket.io — это событийно-ориентированная JavaScript-библиотека для веб-приложений и обмена данными в реальном времени.

Она состоит из двух частей:

- клиентской, предназначенной для использования в браузере;
- серверной, предназначенной для использования на стороне сервера Node.js.

В основном Socket.io использует веб-сокеты для своей работы. Однако иногда она применяет и другие технологии — Flash Socket, AJAX Long Polling, AJAX Multipart Stream, не меняя при этом интерфейс. Это удобно. Если пользователь запустит наше приложение на каком-нибудь очень старом браузере, который не поддерживает веб-сокеты, оно всё равно будет работать, используя эти технологии.

Библиотека даёт возможности, которые не может предоставить нативная реализация веб-сокетов, например, одновременную рассылку сообщений нескольким подключённым клиентам.

Эта библиотека используется технологическими гигантами по всему миру для создания приложений реального времени.

В этом уроке мы рассмотрим работу с этой библиотекой с обеих сторон — клиентской и серверной. Создадим базу, показывающую принцип создания чатов, основанного на базе библиотеки Socket.io.

Серверная часть Socket.io

Начнём с серверной части и инициализируем проект:

```
npm init --yes
```

Официальный сайт говорит, что для установки последней версии пакета, требуется запустить следующую команду:

```
npm install socket.io
```

В качестве сервера возьмём сервер из предыдущего урока. Для создания же сокетного соединения достаточно дописать несколько строк.

Сначала инициализируем пакет сокета:

```
const io = require('socket.io');
```

Далее — созданный http-сервер сохраняем в константу:

```
const app = http.createServer((request, response) => {
  if (request.method === 'GET') {

    const filePath = path.join(__dirname, 'index.html');

    readStream = fs.createReadStream(filePath);

    readStream.pipe(response);
  } else if (request.method === 'POST') {
    let data = '';

    request.on('data', chunk => {
      data += chunk;
    });

    request.on('end', () => {
      const parsedData = JSON.parse(data);
      console.log(parsedData);

      response.writeHead(200, { 'Content-Type': 'json'});
      response.end(data);
    });
  } else {
      response.statusCode = 405;
      response.end();
  }
});
```

Остаётся только проинициализировать точку доступа сокета, описать первые сообщения для подключившихся клиентов и запустить сервер.

```
const socket = io(app);

socket.on('connection', function (socket) {
  console.log('New connection');

  socket.on('CLIENT_MSG', (data) => {
    socket.emit('SERVER_MSG', { msg: data.msg.split('').reverse().join('')});
  });
});

app.listen(3000, 'localhost');
```

Для инициализации точки доступа сокета передадим в модуль io наш созданный http-сервер. Точка доступа сокета может существовать и без этого, однако нам требуется связать её с уже имеющимся http-сервером.

Так как библиотека socket.io событийно-ориентированная, то и работа в библиотеке ведётся по большей части с событиями.

Например, событие 'connection' возникает, когда новый клиент подключается к серверу. В нашем случае мы сообщаем об этом соединении в терминал.

Далее создаём обработчик для события 'CLIENT_MSG'. Судя по названию, событие будет ждать сообщения от клиента. В ответ на него сервер станет посылать сообщение, в которое мы записываем принятую строку в обратном порядке.

Последним шагом запускаем http-сервер, чтобы он слушал порт 3000. Так как сервер сокета связан с http-сервером, он также будет слушать порт 3000.

Клиентская часть Socket.io

Для клиентской части, как и в прошлом уроке, создадим файл index.html.

```
<html>
<head>
      <title>Socket App</title>
</head>
<body>

</body>
</html>
```

Пока этот файл пустой, но сейчас мы начнём его наполнять.

В официальной документации для клиентской части Socket.io [показано](#) несколько способов подключения библиотеки. Мы выберем самый простой способ — через CDN. Для этого в нашу html-страницу добавим строчку:

```
<script src="https://cdn.socket.io/3.1.1/socket.io.min.js"
integrity="sha384-gDaozqUvc4HTgo8iZjwth73C6dDDeOJsAgpxBcMpZYztUfjHXpzrpdrHRdVp8y
SO" crossorigin="anonymous"></script>
```

Как сказано в документации, при таком подключении io (собственно клиентская библиотека) будет доступна как глобальная переменная.

Добавим в index.html следующий код:

```html
<html>

<script src="https://cdn.socket.io/3.1.1/socket.io.min.js"
integrity="sha384-gDaozqUvc4HTgo8iZjwth73C6dDDeOJsAgpxBcMpZYztUfjHXpzrpdrHRdVp8y
SO" crossorigin="anonymous"></script>

<head>
  <title>Socket App</title>
</head>
<body>
  <input type="text" id="input" autofocus>
  <input type="submit" id="send" value="Send">
  <div id="messages"></div>
</body>
<script type="text/javascript">
  const socket = io('localhost:3000');

  const addMessage = (msg) => {
    const msgSpan = document.createElement('span').innerHTML = msg;
    document.getElementById('messages').append(msgSpan);
    document.getElementById('messages').append(document.createElement('br'));
  };

  socket.on('connect', function() {
    console.log('Successful connected to server');
  });

  socket.on('SERVER_MSG', function (data) {
    addMessage(data.msg);
  });

  document.getElementById('send').onclick = function() {
    socket.emit('CLIENT_MSG', { msg: document.getElementById('input').value });
    document.getElementById('input').value = '';
  };
</script>
</html>
```

Здесь всё происходил по аналогии с серверной частью. При инициализации указывается url сервера, который ожидает сокетное соединение. В нашем случае — ‘localhost:3000’.

Затем создаётся обработчик события ‘connect’. Это событие генерируется при установлении соединения с сервером.

Далее формируем обработчики события SERVER_MSG'. Данные, которые приходят в этом событии, выведем в специально созданный div-блок ‘messages’ посредством функции addMessage.

Мы создали простую html-форму, чтобы отправлять данные серверу. Для этого у нас есть поле ввода — input-элемент типа text — и кнопка Send. Для кнопки мы сформировали обработчик клика, в котором будем отправлять сообщения на сервер и очищать поле ввода.

Теперь запустим нашу систему и посмотрим на результат.

Сначала запускаем сервер, затем в браузере открываем [эту страницу](#).

Сервер возвращает html-страницу, и на ней мы видим поле ввода.

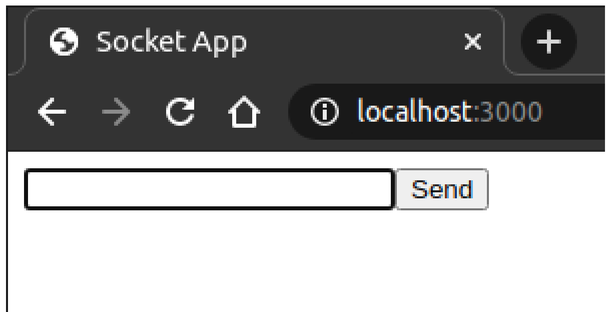

Рис. 3. Html-форма для отправки сообщений

В терминале запущенного сервера появилось успешно установленное соединение:

```
New connection
```

В консоли браузера мы также видим успешное соединение и сообщение от сервера:

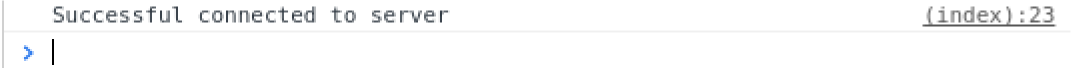

Рис. 4. Сообщение об установленном соединении с сервером в консоли браузера

Отправим сообщение Hello World!, используя созданную форму.

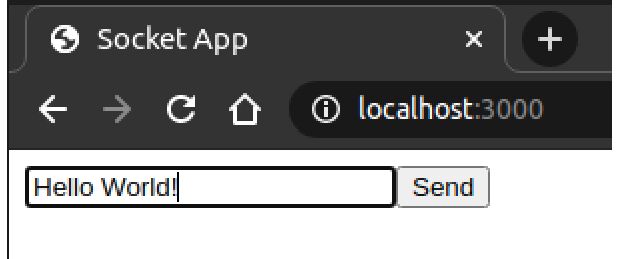

Рис. 5. Состояние страницы до нажатия кнопки Send

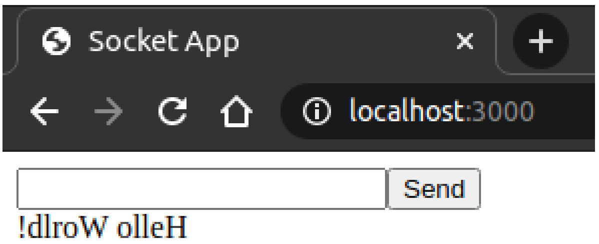

Рис. 6. Состояние страницы после нажатия кнопки Send

Как мы видим на рисунках, клиент успешно отправил сообщение, сервер его принял, инвертировал и отправил обратно. Функция addMessage, вызываемая в обработчике события, добавила сообщение на страницу.

Теперь отправим ещё одно сообщение.

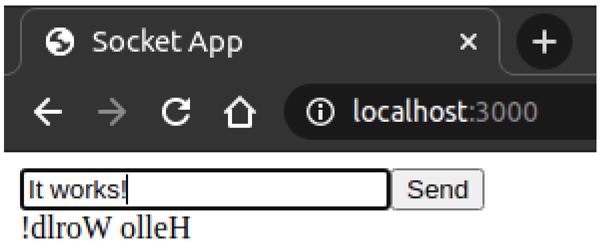

Рис. 7. Состояние страницы до отправки второго сообщения

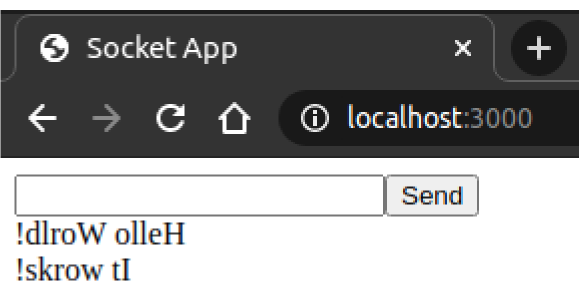

Рис. 8. Состояние страницы после отправки второго сообщения

Система сообщений работает. Теперь научимся оповещать всех клиентов о подключении нового участника. Когда новый человек войдёт в чат, это должны увидеть сразу все пользователи.

Рассылка сообщений

Поменяв всего одну строчку в серверной части, получится реализовать этот инструментарий. Для этого в отправку сообщения от сервера достаточно добавить вызов метода broadcast:

```
socket.broadcast.emit('NEW_CONN_EVENT', { msg: 'The new client connected' });
```

В клиенте добавим обработку этого сообщения по аналогии с сообщением ‘CLIENT_MSG’:

```
socket.on('NEW_CONN_EVENT', function (data) {
  addMessage(data.msg);
});
```

Теперь перезапустим сервер и откроем сначала одну страницу в браузере, а потом вторую.

На первой странице появится сообщение о подключении нового клиента:

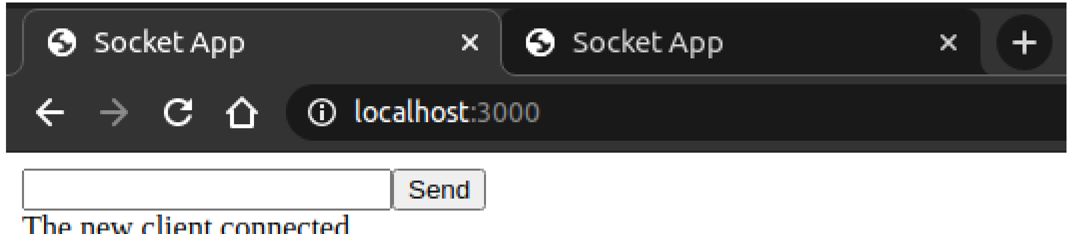

Рис. 9. Сообщение о подключении нового клиента

Здесь мы рассмотрели основной инструментарий, который даёт базовое понимание принципа работы всех чатов, мессенджеров и других приложений реального времени. Однако клиент всё ещё не получает никаких данных от других клиентов, кроме сообщения об их подключении. Реализация такого инструментария станет частью практического задания.

Далее перейдём к ещё одной важной концепции Node.js-приложений — к использованию отдельных потоков воркеров для ресурсоёмких задач.

Однопоточность

Node.js — это однопоточная среда. Как мы знаем, некоторые операции выгружаются в операционную систему, где выполняются параллельно. Это асинхронные операции ввода-вывода. Основной же JavaScript-код выполняется синхронно в один поток. И этим потоком нагружается только одно ядро процессора.

В тех случаях, когда основную часть приложения составляют асинхронные операции ввода-вывода, всё работает прекрасно.

Однако, если в коде надо выполнять какие-то тяжёлые вычисления, которые занимают много времени — это может заблокировать основной поток. То есть программа не выполнит никаких других вычислений, так как код синхронный. Например, такое вычисление займёт у веб-сервера 5 секунд. В течение этих 5 секунд сервер не примет и не обработает ни один запрос!

Дело в том, что изначально ни язык программирования JavaScript, ни среда Node.js не были предназначены для сложных вычислений, требующих больших ресурсов процессора. В браузере

выполнение таких сложных задач означало бы «тормоза» в пользовательском интерфейсе. В Node.js мы наблюдаем ограничение пропускной способности веб-сервера, ограничение запуска новых асинхронных задач ввода-вывода и задержку выполнения коллбэков, связанных с завершением выполнения асинхронных задач.

Эту задачу призван решить модуль worker_threads.

Модуль worker_threads

Этот модуль позволяет создавать отдельные потоки воркеров для выполнения трудоёмких по вычислительной мощности задач. Однако он даёт использовать для вычисления незагруженные ядра процессора. По этой причине не рекомендуется создавать больше воркеров, чем количество ядер у процессора.

Такие потоки выполняются в изолированном контексте, обмениваясь информацией с главным процессом посредством сообщений. Это позволяет избежать «состояния гонок». В многопоточных системах такая проблема возникает, когда несколько потоков одновременно обращаются к одной и той же области памяти. Последнее, как правило, приводит к утечке памяти, порче данных, уязвимостям и т. д.

Потоки воркеров функционируют в том же процессе, что и основная программа.

Важно!

В модуле worker_threads есть возможность пользоваться разделяемой памятью. Для этого предусмотрены объекты типа SharedArrayBuffer. С ними надо быть осторожными и использовать только в тех случаях, когда требуется выполнить сложную обработку больших данных. В этом случае мы можем сэкономить ресурсы на передаче данных между воркерами и основным процессом, когда приходиться много раз сериализовывать или десериализовывать данные.

В качестве примера напишем приложение-генератор паролей.

Подготовим инструментарий для инициализации и создания отдельного потока воркера.

Для этого нам понадобится два файла. Файл index.js будет отвечать за инициализацию потока воркера, а worker.js — нести в себе его инструментарий.

index.js:

```
const { Worker } = require('worker_threads')

const passwordSizeBytes = 4;

function start(workerData) {
  return new Promise((resolve, reject) => {
    const worker = new Worker('./worker.js', { workerData });
```

```
    worker.on('message', resolve);
    worker.on('error', reject);
  })
}

start(passwordSizeBytes)
  .then(result => console.log(result))
  .catch(err => console.error(err));
```

Здесь всё достаточно очевидно. Импортируя в файл класс Worker из модуля worker_threads, мы получаем возможность создать его экземпляр.

При создании воркера в конструктор передаётся путь к файлу, содержащий код воркера, а также объект с данными, которые для него предназначены.

С концепцией событий в этом курсе мы уже встречались, применима она и здесь.

Событие message предназначается для получения сообщения от воркера. В самом простом случае такое сообщение — это результат выполнения задачи.

Событие error предназначается для обработки ошибок, которые возникли в процессе выполнения задачи.

worker.js:

```
const { workerData, parentPort } = require('worker_threads');

parentPort.postMessage({ result: `You want to generate password ${workerData}
bytes size` });
```

Сейчас в этом файле есть только получение данных из главного процесса (константа workerData) и объект, отвечающий за передачу данных обратно главному процессу — объект parentPort. Используя метод postMessage объекта parentPort, мы можем передать данные выполнения программы главному процессу.

Если сейчас запустить программу, где точка входа — файл index.js, то в терминале появится следующий вывод:

```
{ result: 'You want to generate password 4 bytes size' }
```

Модуль crypto

Для создания пароля мы будем использовать ещё один стандартный модуль Node.js, который называется crypto. Это модуль шифрования, он предоставляет различные криптографические функции, например, функции хеширования, шифрования, дешифрования, подписи и прочие.

Нас интересует метод crypto.randomBytes(size[, callback]). Этот метод генерирует криптографически сильный псевдослучайный набор данных определённого размера. Переменная size — это размер данных в байтах, а callback — функция обратного вызова, которая вызывается после окончания работы метода.

Однако, если в метод crypto.randomBytes не передать параметр callback, он будет работать синхронно. Именно синхронный режим работы мы используем в примере, так как асинхронные функции отправлять в специально созданный поток воркера не имеет особого смысла. Асинхронные операции ввода-вывода будут гораздо эффективнее работать сами по себе, без «обёртки» в виде отдельного потока воркера.

Для генерации пароля достаточно добавить лишь несколько строк:

```
const { workerData, parentPort } = require('worker_threads');
const crypto = require('crypto');

const password = crypto.randomBytes(workerData).toString('hex');

parentPort.postMessage({ result: `Password was generated: ${password}` });
```

Теперь, если запустить приложение, то в терминале мы увидим следующее:

```
{ result: 'Password was generated: 72aac223' }
```

Заключение

В этом курсе мы рассмотрели базовые возможности среды Node.js. Конечно, это далеко не все её возможности.

Node.js постоянно развивается и улучшается. Программисты всего мира постоянно придумывают и публикуют новые модули в npm, разрабатывают новые фреймворки и реализуют новые концепции использования.

Например, есть фреймворк Electron, который, используя Node.js в том числе, позволяет создавать графические приложения для операционных систем. На базе Electron появились следующие:

1. Visual Studio Code — популярная среди программистов IDE.
2. Клиентское приложение Slack — этот мессенджер активно используется в корпоративной IT-среде и не только.
3. Клиентское приложение Skype.
4. Мессенджер Discord.
5. Прочие.

Или, например, фреймворк NestJS, который также обретает всё большую популярность в последнее время. Этот фреймворк создан, чтобы помогать программистам, использующим Node.js в качестве среды для разработки бэкенда, делать это максимально просто, используя лучшие подходы в построении архитектуры бэкенд-приложений прямо из коробки.

Однако все эти фреймворки «под капотом» используют всё те же базовые возможности среды Node.js — событийно-ориентированную архитектуру, цикл событий, потоки и т. д. Таким образом, владея этими основами, нам не составит труда разобраться и в новейших фреймворках.

Практическое задание

В разделе, посвящённом веб-сокетам, мы сделали базовую основу для простого чата. В ней клиент может посылать сообщения серверу и получать их обратно в виде обратно отображаемой строки. Мы сделали оповещение всех клиентов о подключении нового клиента.

Дополните это приложение следующим инструментарием:

1. Пользователи должны видеть не только сообщение о подключении нового клиента, но и об отключении клиента или переподключении.
2. На странице приложения важно, чтобы сообщения от разных клиентов различались. Для этого генерируйте ник пользователя при каждом его подключении.
3. Пользователи должны видеть сообщения к серверу от других пользователей.

Дополните вашу веб-версию файлового менеджера следующим инструментарием:

1. Добавьте счётчик посетителей на странице, который работал бы через сокеты и динамически обновлялся на всех клиентах при их подключении или отключении.
2. Вынесете инструментарий поиска по содержимому файла из заданий уроков 3 и 4 в отдельный поток воркера.

Глоссарий

1. **Веб-сокет** — это стандарт двухсторонней связи клиента с сервером по TCP-соединению, совместимый с HTTP и предназначенный для обмена сообщениями в режиме реального времени.

2. **Библиотека Socket.io** — это событийно-ориентированная JavaScript-библиотека для веб-приложений и обмена данными в реальном времени.

3. **Модуль worker_threads** — это стандартный модуль Node.js, который позволяет создавать отдельные потоки воркеров для выполнения трудоёмких по вычислительной мощности задач.

Дополнительные материалы

1. Официальная документация.

2. Официальный сайт библиотеки Socket.io.

3. Официальная документация модуля Workers.

4. Статья «Многопоточность в Node.js: модуль worker_threads».

5. Статья «Состояние гонки».

Используемые источники

1. Официальная документация.

2. Статья WebSocket.

1. Статья «Многопоточность в Node.js: модуль worker_threads».