**КТ № 6. Работа с файлами. События в Node.js. Потоки данных. Канал Pipe.**

**Время на выполнение практической работы:** 4 часа.

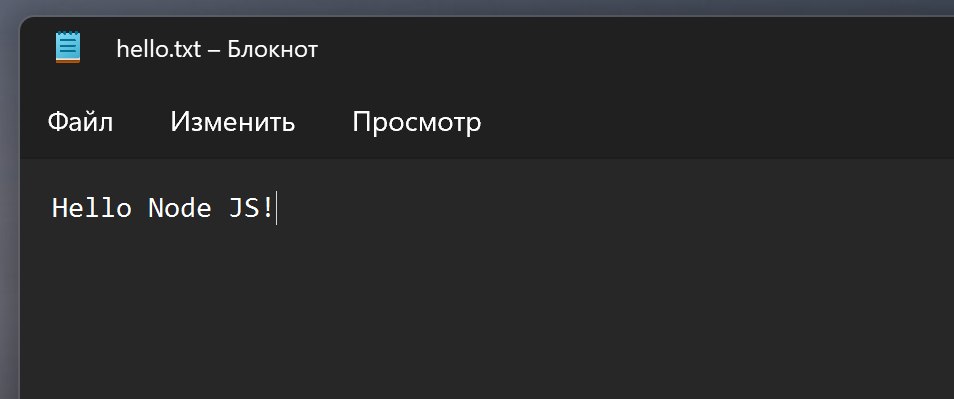
**Теоретические сведения к выполнению практической работы:**

**Работа с файлами.**

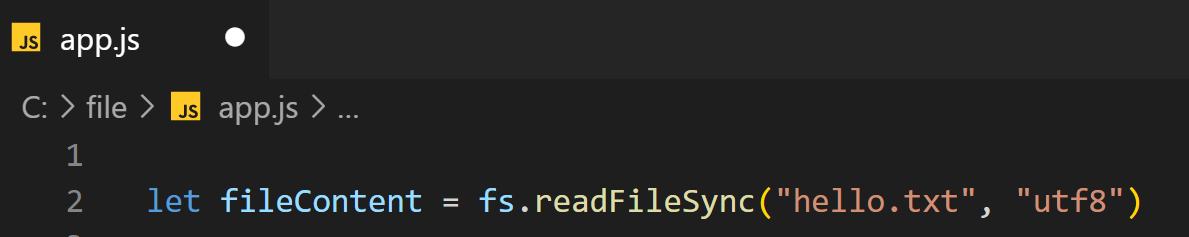
Для работы с файлами в Node.js предназначен модуль fs. Рассмотрим, как с ним работать.

**Чтение из файла**

Допустим, в одной папке с файлом приложения app.js расположен текстовый файл hello.txt с простейшим текстом, например:

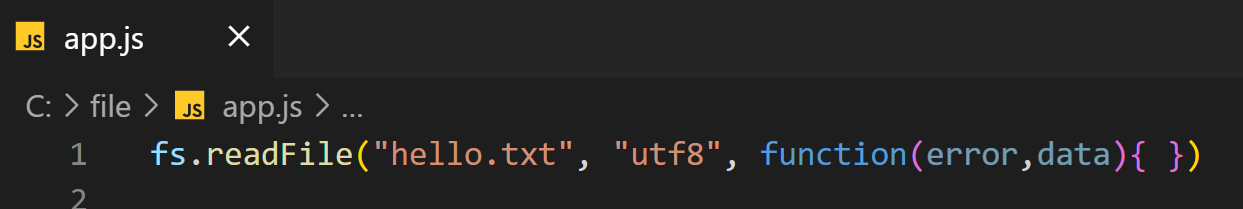


Для чтения файла в синхронном варианте применяется функция fs.readFileSync():



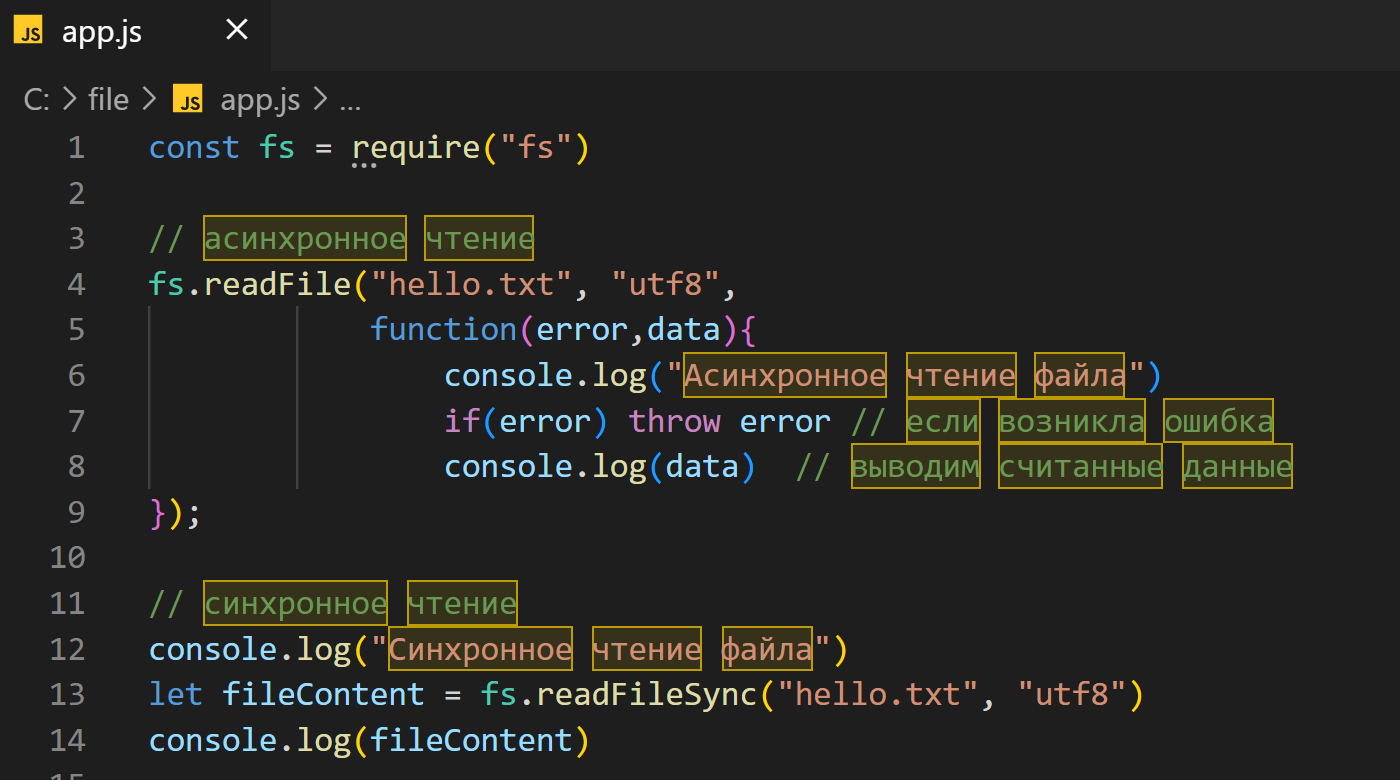
В метод передается путь к файлу относительно файла приложения app.js, а в качестве второго параметра указывается кодировка для получения текстового содержимого файла. На выходе получаем считанный текст.

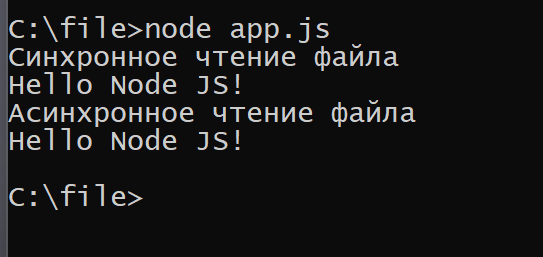
Для асинхронного чтения файла применяется функция fs.readFile:



Первый и второй параметр функции опять же соответственно путь к файлу и кодировка. А в качестве третьего параметра передается функция обратного вызова, которая выполняется после завершения чтения. Первый параметр этой функции хранит информацию об ошибке при наличии, а второй - собственно считанные данные.

Для чтения файла определим в файле app.js следующий код:





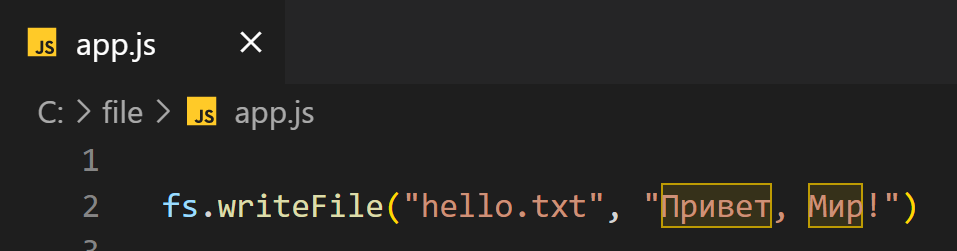
И здесь стоит обратить внимание, что несмотря на то, что функция fs.readFile() вызывается первой, но так как она асинхронная, она не блокирует поток выполнения, поэтому ее результат выводится в самом конце.

**Запись файла**

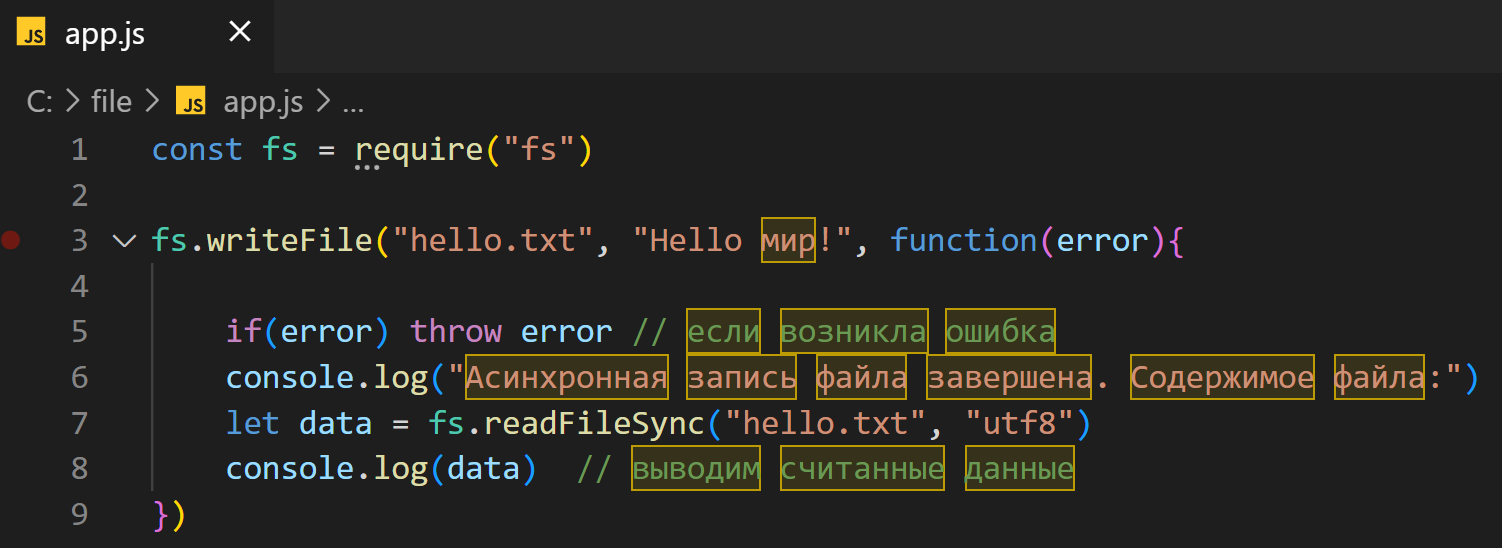
Для записи файла в синхронном варианте используется функция fs.writeFileSync(), которая в качестве параметра принимает путь к файлу и записываемые данные:

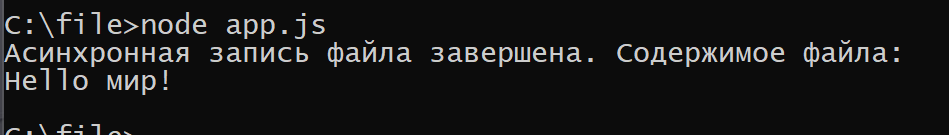


Также для записи файла можно использовать асинхронную функцию fs.writeFile(), которая принимает те же параметры:

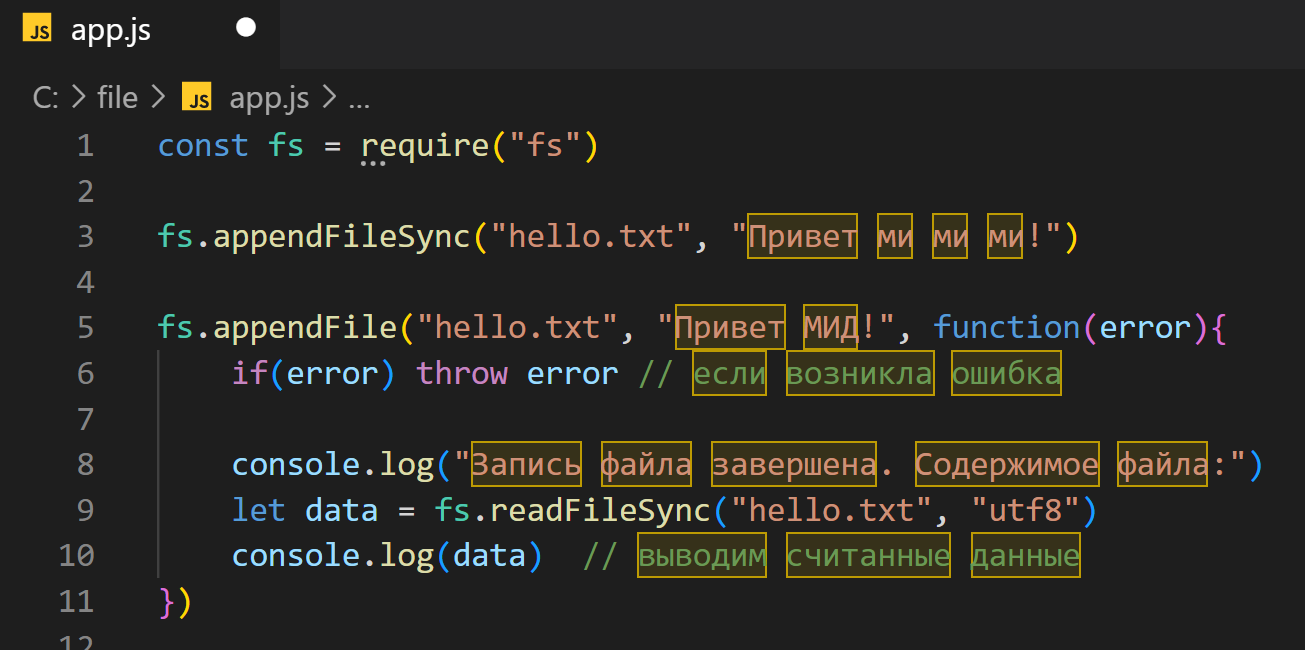


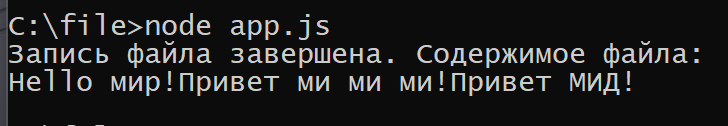
В качестве вспомогательного параметра в функцию может передаваться функция обратного вызова, которая выполняется после завершения записи:





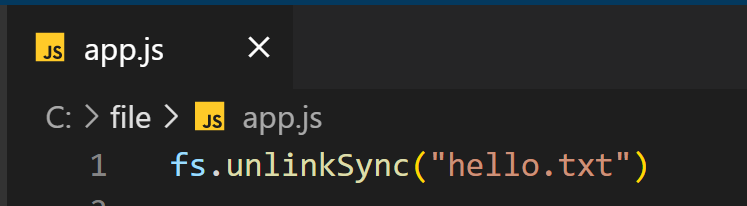
Следует отметить, что эти методы полностью перезаписывают файл. Если надо дозаписать файл, то применяются методы fs.appendFile()/fs.appendFileSync():



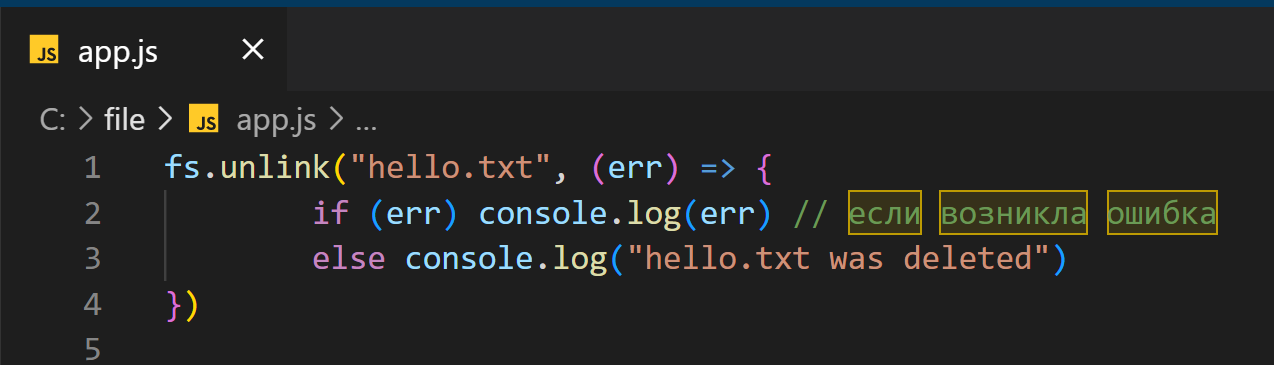


**Удаление файла**

Для удаления файла в синхронном варианте используется функция fs.unlinkSync(), которая в качестве параметра принимает путь к удаляемому файлу:



Также для удаления файла можно использовать асинхронную функцию fs.unlink(), которая принимает путь к файлу и функцию, вызываемую при завершении удаления:

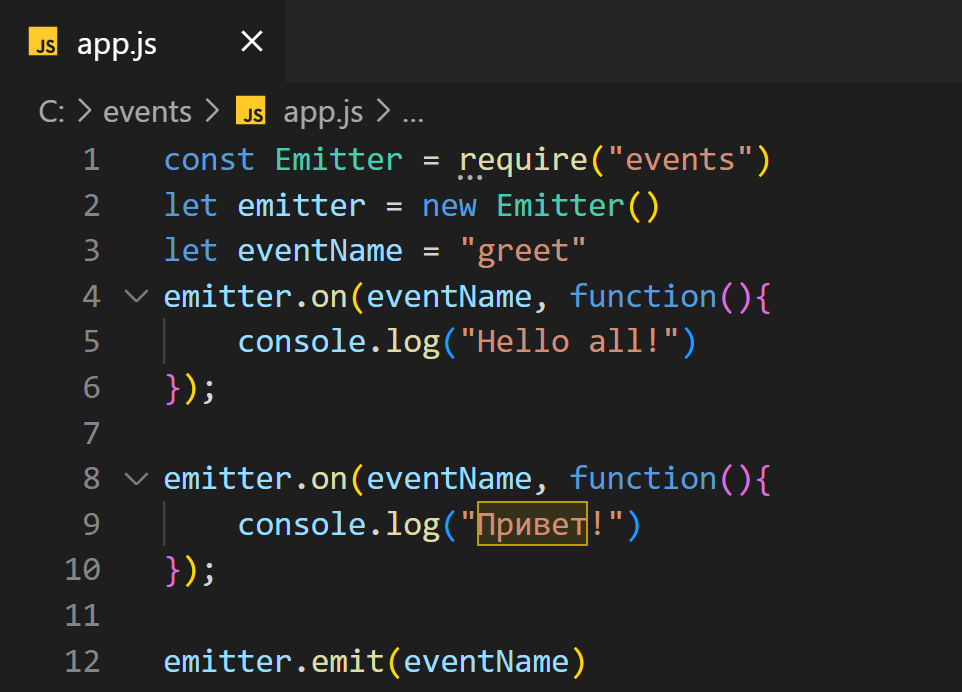


**События**

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.

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.

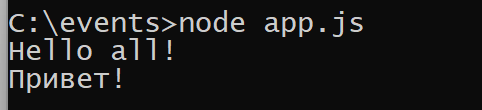
Рассмотрим применение объекта EventEmitter и событий. Для этого определим следующий файл app.js:



Весь необходимый функционал сосредоточен в модуле events, который необходимо подключить. С помощью функции on() связываем событие, которое передается в качестве первого параметра, с некоторой функцией, которая передается в качестве второго параметра. В данном случае событие называется "greet".

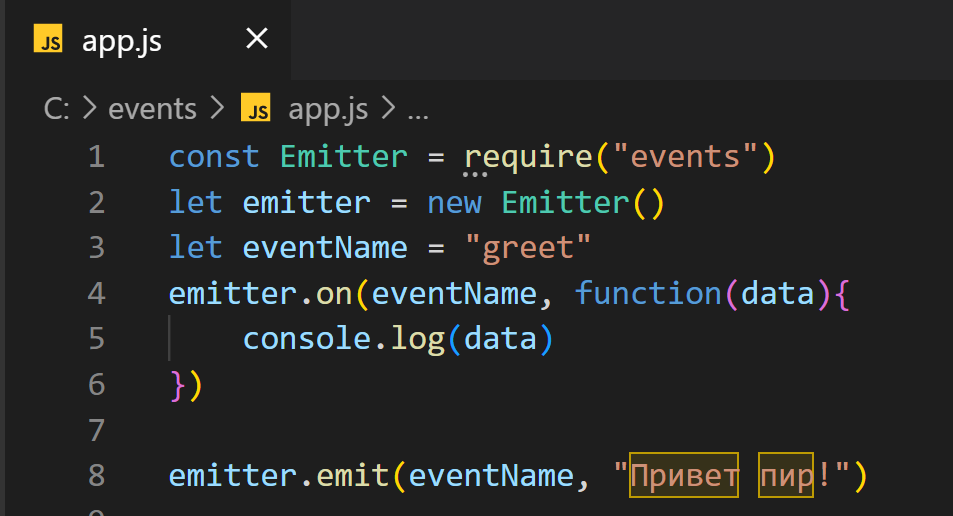
Для генерации события и вызова связанных с ним обработчиков выполняется функция emitter.emit(), в которое передается название события.

И при запуске приложения будут вызваны все обработчики:



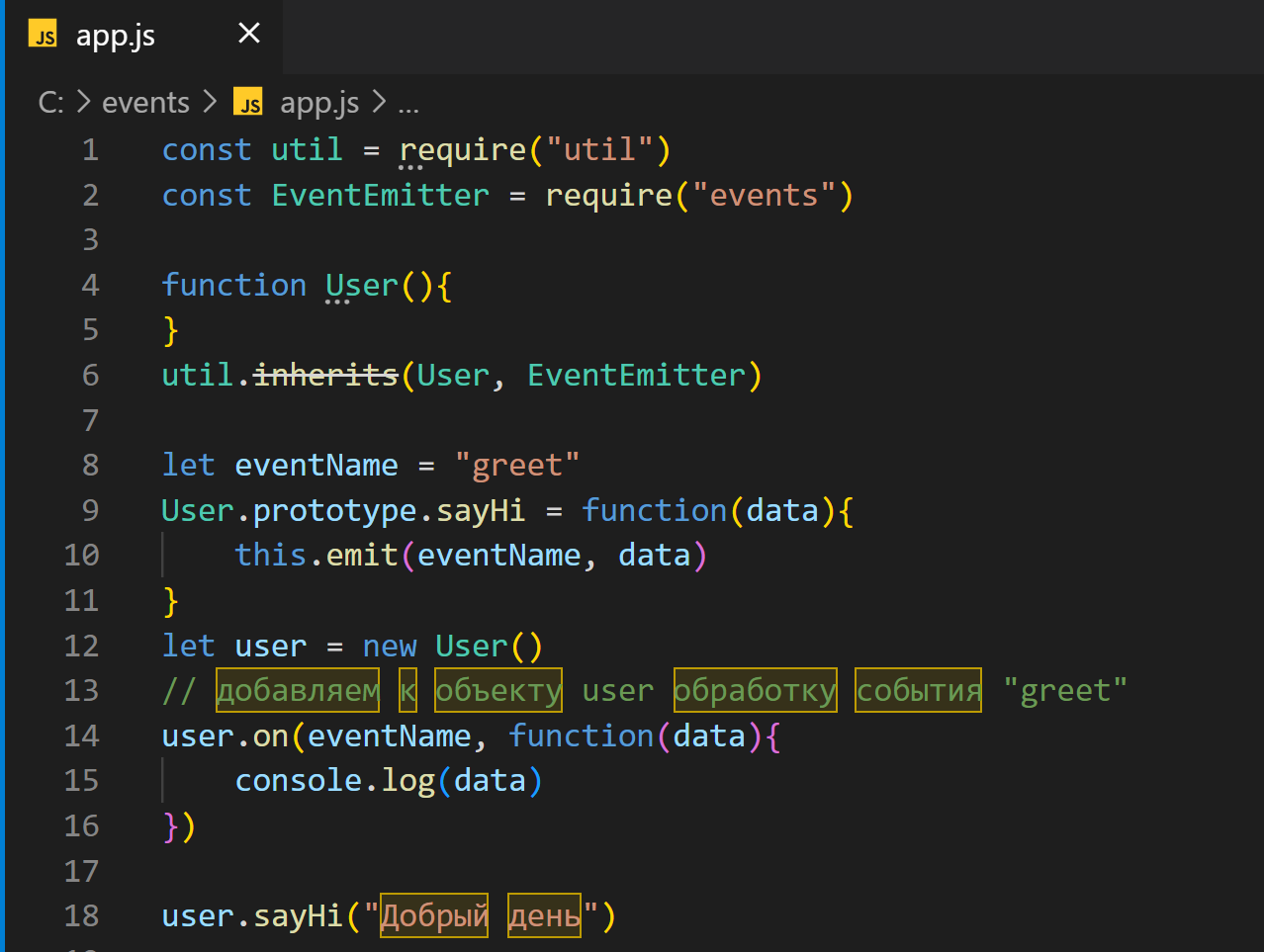
**Передача параметров событию**

При вызове события в качестве второго параметра в функцию emit можно передавать некоторый объект, который передается в функцию обработчика события:



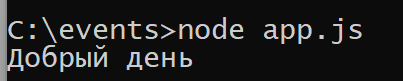
**Наследование от EventEmitter**

В приложении мы можем оперировать сложными объектами, для которых также можно определять события, но для этого их надо связать с объектом EventEmitter. Например:

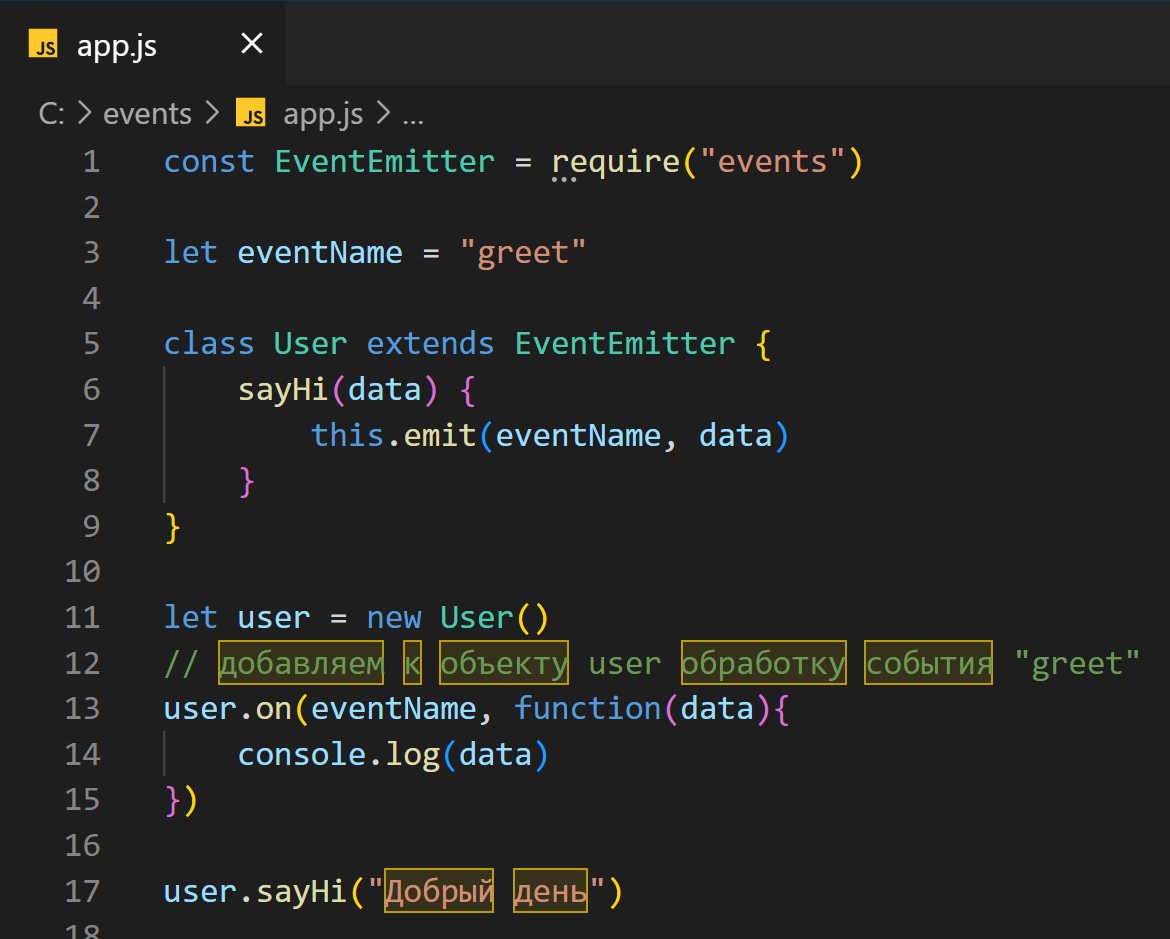


Здесь определена функция конструктора User, которая представляет пользователя. Для прототипа User определяется метод sayHi, в котором генерируется событие "greet".

Но чтобы связать объект User с EventEmitter, надо вызвать функцию util.inherits(User, EventEmitter);. Она позволяет унаследовать классу User функционал от EventEmitter. Благодаря этому мы можем через метод on() добавить к событию объекта user какой-нибудь обработчик, который будет вызван при выполнении метода user.sayHi().



С помощью возможностей ES6 мы можем упростить выше пример:



Результат будет тот же, но теперь не нужно использовать функцию util.inherits.

**Stream**

Stream представляет поток данных. Потоки бывают различных типов, среди которых можно выделить потоки для чтения и потоки для записи.

При создании сервера мы уже сталкивались с потоками:



Параметры request и response, которые передаются в функцию и с помощью которых мы можем получать данные о запросе и управлять ответом, как раз представляют собой потоки: request - поток для чтения, а response - поток для записи.

Используя потоки чтения и записи, мы можем считывать и записывать информацию в файл. Например:



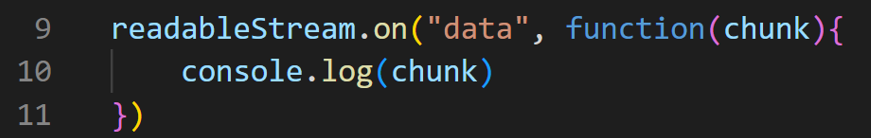
Для создания потока для записи применяется метод fs.createWriteStream(), в который передается название файла. Если вдруг в папке нет такого файла, то он создается.

Запись данных производится с помощью метода write(), в который передаются данные. Для окончания записи вызывается метод end().

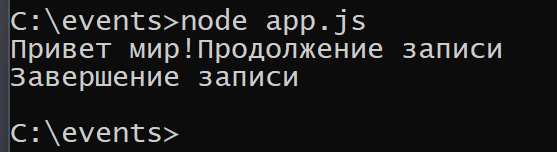
После этого в папке проекта появляется файл hello.txt, который можно открыть в любом текстовом редакторе.

Для создания потока для чтения используется метод fs.createReadStream(), в который также передается название файла. В качестве опционального параметра здесь передается кодировка, что позволит сразу при чтении кодировать считанные данные в строку в указанной кодировке.

Сам поток разбивается на ряд кусков или чанков (chunk). И при считывании каждого такого куска, возникает событие data. С помощью метода on() мы можем подписаться на это событие и вывести каждый кусок данных на консоль:



Запустим файл на выполнение:

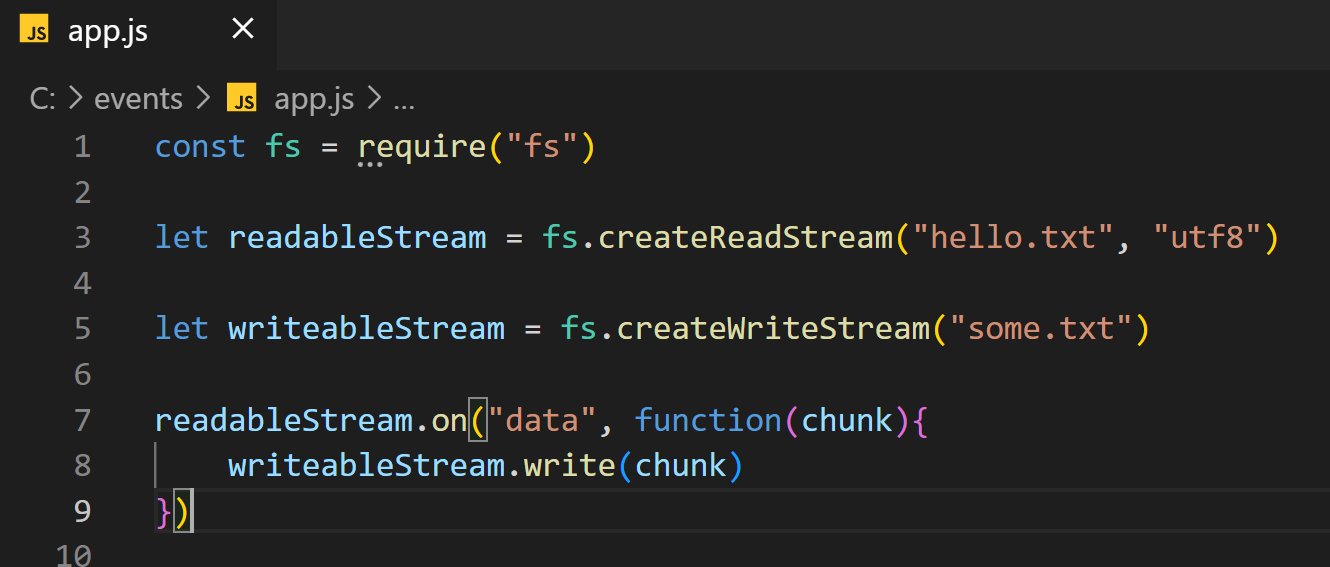


Только работой с файлами функциональность потоков не ограничивается, также имеются сетевые потоки, потоки шифрования, архивации и т.д., но общие принципы работы с ними будут те же, что и у файловых потоков.

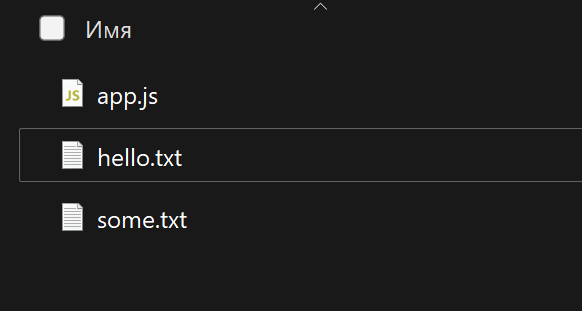
**Pipe**

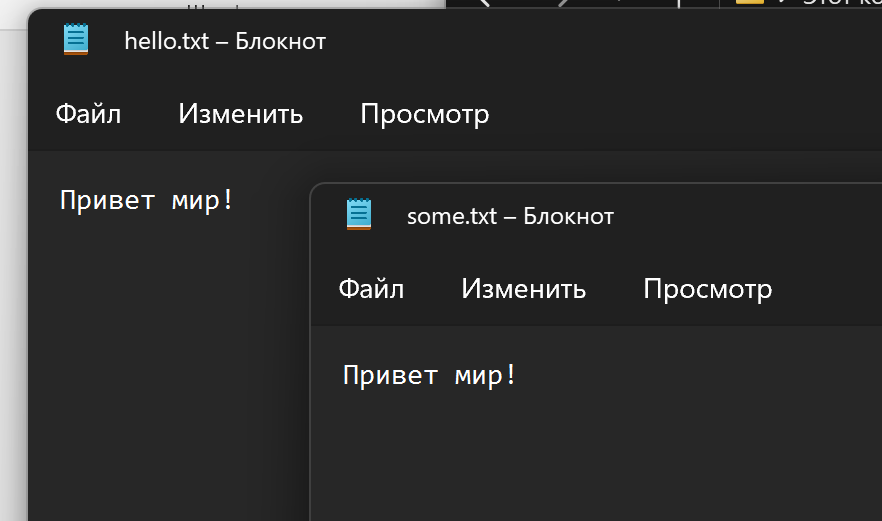
Pipe - это канал, который связывает поток для чтения и поток для записи и позволяет сразу считать из потока чтения в поток записи. Для чего они нужны? Возьмем, к примеру проблему копирования данных из одного файла в другой.

Пусть в папке проекта определен некоторый файл hello.txt. Скопируем его содержимое в новый файл some.txt:

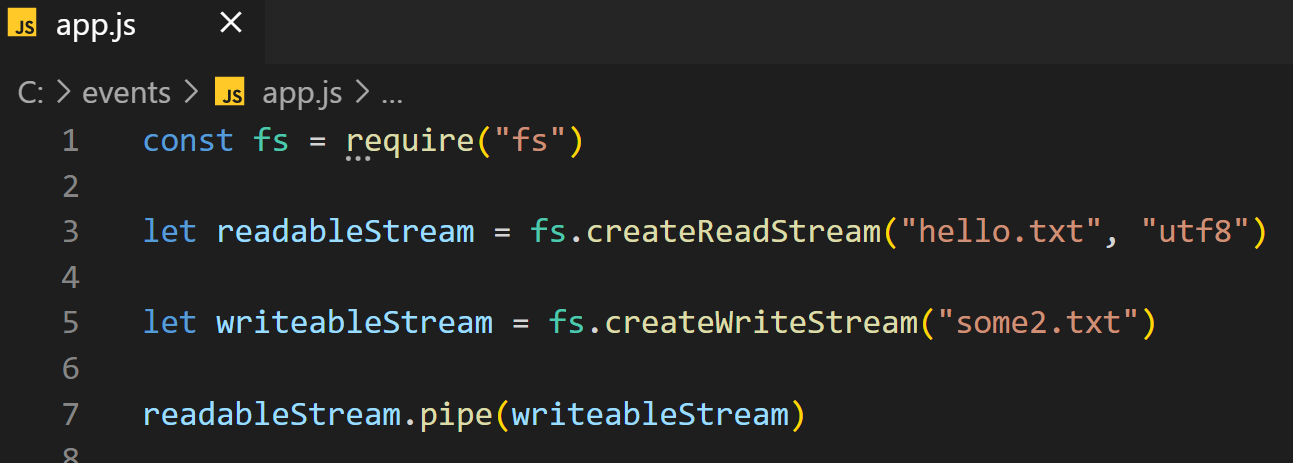


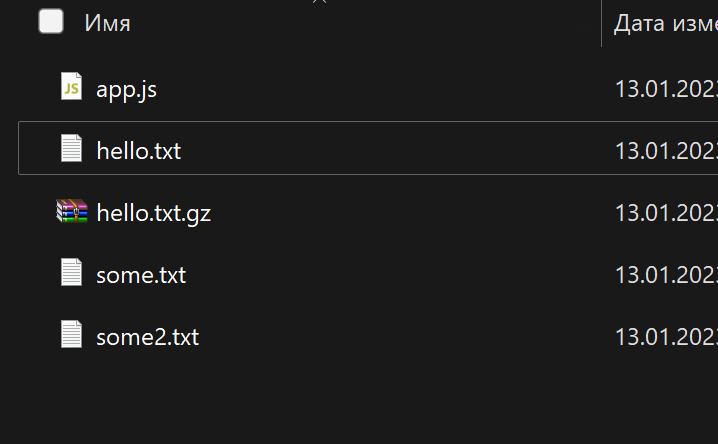
Данный код вполне работоспособен, и после запуска файла в папке проекта появится новый файл some.txt.





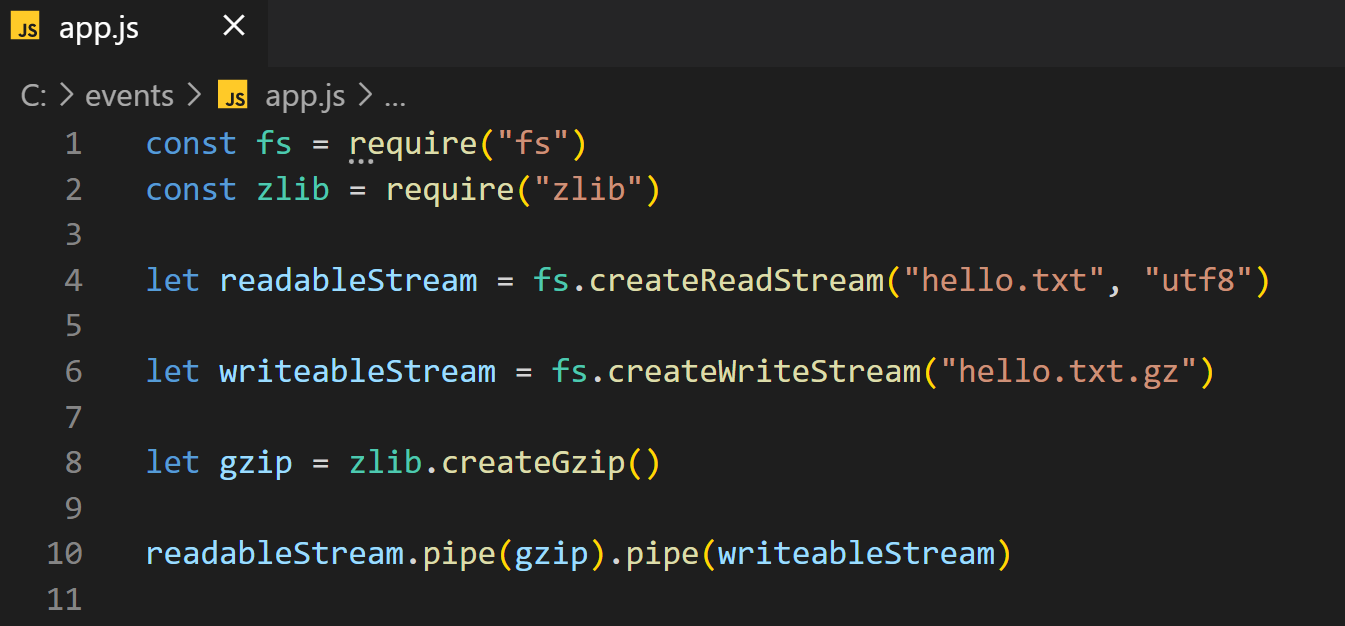
Однако задача записи в поток данных, считанных из другого потока, является довольно распространенной, и в этом случае pipes или каналы позволяют нам сократить объем кода:



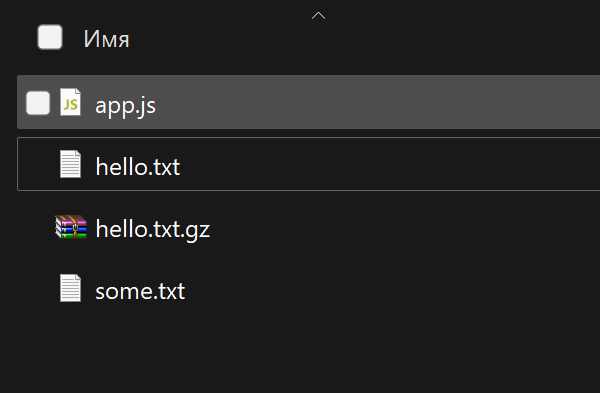


У потока чтения вызывается метод pipe(), в который передается поток для записи.

Рассмотрим другую проблему - архивацию файла. Здесь нам надо сначала считать файл, затем сжать данные и в конце записать сжатые данные в файл-архив. Pipes особенно удобно применять для подобного набора операций:



Для архивации подключается модуль zlib. Каждый метод pipe() в цепочке вызовов возвращает поток для чтения, к которому опять же можно применить метод pipe() для записи в другой поток.



**Этапы выполнения:**

1. ***Реализовать чтение из текстового файла в синхронном варианте;***
2. ***Реализовать асинхронное чтение из текстового файла;***
3. ***Результат зафиксировать при помощи командной строки;***
4. ***Реализовать запись текстового файла в синхронном и асинхронном вариантах. Результат зафиксировать при помощи командной строки;***
5. ***«Дозапишите» информацию в текстовый файл. Результат зафиксировать при помощи командной строки;***
6. ***Реализовать удаление текстового файла в синхронном и асинхронном вариантах;***
7. ***Сгенерируйте событие и вызовите связанные с ним обработчики. Результат зафиксировать при помощи командной строки;***
8. ***Передайте событию параметры;***
9. ***Реализуйте наследование двумя способами: используя функцию util.inherits и не прибегая к ней. Результат зафиксировать при помощи командной строки;***
10. ***При помощи потоков чтения и записи считайте и запишите информацию в текстовый файл. Результат зафиксировать при помощи командной строки;***
11. ***Используя канал pipe скопируйте данные из одного текстового файла в другой. Подтвердите скриншотом;***
12. ***Создайте файл архив используя канал pipe и модуль zlib. Подтвердите скриншотом.***