**От новичка, для новичков: «Event-driven programming. EventEmitter.»**

**Описание**

Вдохновившись [документацией MDN](https://developer.mozilla.org/ru/docs/Web/Events), которая позиционирует себя как - ***Resources for Developers, by Developers***, я решил сделать серию статей «От новичка, для новичков», где попытаюсь максимально простым языком рассказывать о разных концепциях в программировании, в основном связанных с JavaScript. Я буду стараться избегать технически-сложного контента, или буду максимально его упрощать.

Это "первая статья серии, будет посвящена событийно-ориентированной парадигме применительно к Node.js, а также паттерну «EventEmitter».

P.S. Рассчитываю на конструктивную критику читателей! Всегда буду рад обратной связи. Не стесняйтесь указывать на ошибки или хвалить, пишите в комментариях и в личку. Особенно буду благодарен если вы предложите темы для статей!

**Введение**

Когда я только-только начинал изучать программирование то постоянно сталкивался с нехваткой материалов для абсолютных новичков. Особенно сильно мешало восприятию материалов непонимание терминологии и разных базовых концепций. А поскольку я осваивал **Node.js**, что бы я не изучал и в чём бы не пытался разобраться – везде меня преследовал ужасный и непонятный **EventEmitter**.

В Node.js чуть ли не каждая первая сущность наследуется и расширяет класс EventEmitter. При просмотре видеоуроков и туториалов, лекторы постоянны говорили о каком-то «прослушивании» (*кто кого прослушивает?*), каких-то «листенерах» (*это вообще, что за зверь?*), «эвентах» (*это как-то связано с эвент-агентствами?*), «подписках» (*причем тут ютуб и нетфликс?*), «генерации событий» (*это мне ещё и генератор нужен?*) и др.

Сейчас концепция EventEmitter для меня более-менее понятна, но в начале пути, казалось, что речь идёт о каком-то вездесущем монстре. Так давайте же укротим монстра!

**Событийно-ориентированное программирование или «А как же ООП?»**

На своём официальном сайте Node.js описана как «асинхронная среда выполнения JavaScript, **управляемая событиями** (event-driven)». Таким образом, значительная часть программирования на Node.js построена на событийно-ориентированной модели. А сердцем Node.js является класс «EventEmitter». Давайте разберёмся как это всё работает?

**Событийно-ориентированное программирование** (***event-driven programming***) - парадигма программирования, в которой выполнение программы определяется событиями. ***Event-driven programming (EDP)*** ещё переводят как ***программирование, управляемое событиями.***

На фронтенде событиями будут являться, например: действия пользователя - клик мышкой по кнопке, отправка данных через форму, нажатие пальцем по сенсорному экрану и т.д.

На бэкенде событиями будут являться, например: установление TCP-соединения между клиентом и сервером, обращение к базе данных, конец чтения файла, и т.д.

Событийно-ориентированная парадигма строится на том, что окружающий мир можно представить в виде событий. И объекты взаимодействуют друг с другом через события.

**«А как же объектно-ориентированное программирование?»** —спросите вы. И на самом деле многие интересуется, как сосуществуют друг с другом **ООП** (***object-oriented programming***) и **EDP** (***event-driven programming***). Ведь все говорят, что на собеседованиях спрашивают про ООП, что JavaScript код нужно писать с соблюдением принципов ООП… А мы тут задвигаем за какое-то объектно-ориентированное программирование.

На эту тему написано много чего умного, а вот мне больше всего понравился ответ на вопрос [«What is the relation of 'Event Driven' and 'Object Oriented' programming?»](https://stackoverflow.com/questions/22101731/what-is-the-relation-of-event-driven-and-object-oriented-programming) с сайта stackoverflow.com.

***Примечание:*** *на данный момент, приведенный ниже «ответ на вопрос» скорее всего будет непонятен для новичков, поэтому рекомендую ещё раз вернуться к этой части после прочтения данной статьи.*

**Вопрос:** Какова связь между "управляемым событиями" и "объектно-ориентированным" программированием?

**Ответ:**

1. Объектно-ориентированное программирование (ООП) и программирование, управляемое событиями (EDP) **совместимы**, т.е. их можно использовать вместе.

2. В ООП + EDP все принципы ООП (инкапсуляция, наследование и полиморфизм) остаются неизменными.

3. В ООП + EDP стандартные объекты приобретают новый механизм взаимодействия с друг с другом на основе «*публикации уведомлений о событиях*» и «*подписки на уведомления о событиях*» от других объектов.

4. Разница между ООП «с» и «без» EDP заключается в потоке управления между объектами.

В **ООП без EDP** контроль над исполнением кода переходит от одного объекта к другому при вызове методов. Обычно один объект вызывает метод другого объекта.

В **ООП с EDP** контроль над исполнением кода переходит от одного объекта к другому при уведомлении о событии. Один объект подписывается на уведомление от других объектов, затем ожидает уведомлений от объектов, на которые он подписан, выполняет некоторую работу на основе уведомлений и публикует свои собственные уведомления.

**Вывод:** ООП с использованием EDP – это всё тот же старый ООП, но с **инвертированным** (перевернутым) потоком управления, благодаря механизму управления событиями.

**Весь мир – программа, а люди в нём – события, или «Один день из жизни Жени!»**

Представим спящую девушку по имени Женя. Она спит до тех пор, пока будильник не ***сгенерирует событие*** «звон». Женя, услышав звон будильника, проснется и начнёт своё обычное утро. Женя знает, что до работы ей идти пол часа, а значит ровно в 08:30 нужно выходить из дома на работу. До этого времени можно почитать статейки на Хабре или позалипать на видосики в интернете. Женя ***находится в состоянии ожидания***, пока настенные часы не ***сгенерируют событие*** «отобразить на циферблате время 08:30». Женя отслеживает (***подписана на***) это событие, чтобы не опоздать на работу. Когда часы отобразят 08:30 она это увидит (***отловит***) и пойдёт на работу.

В 09:00 утра Женя уже будет сидеть за монитором рабочего компьютера в хорошо освещаемом тёплом офисе, попивая ароматный латте, купленный в кофейне на первом этаже офисного здания. Женя снова ***находится в состояния ожидания***, она ждёт письма от руководителя со списком дел на день. И вот, руководитель наконец-то ***сгенерировал событие*** «рассылка сотрудникам писем со списками дел». Письмо придёт на электронную почту Жени, а почтовый менеджер на рабочем компьютере Жени ***сгенерирует событие*** «поступило новое письмо». Женя откроет электронную почту, прочитает письмо и начнет поочередно выполнять задания руководителя. В конце рабочего дня Женя напишет отчёт о проделанной работе и направит его своему руководителю, тем самым ***сгенерировав*** для руководителя ***событие*** «ежедневный отчёт сотрудника», чтобы тот смог подготовить задания на завтра. Руководитель в свою очередь отслеживает события «ежедневный отчёт» от каждого сотрудника, т.е. ***подписан*** на событие.

После окончания рабочего, возвращаясь домой Женя будет стоять на перекрестке, в ожидании зеленого сигнала светофора. Когда светофор ***сгенерирует событие*** «зелёный свет для пешехода», Женя начнёт переходить дорогу, однако другие участники дорожного движения, например водители будут стоять на месте. Потому что только Женя и другие пешеходы ***ожидают (подписаны на)*** ***событие*** «зелёный свет для пешехода», водители же в свою очередь ***ожидают (подписаны на) событие*** «зелёный свет для водителей». Женя будет игнорировать события для водителей, потому что она пешеход и это событие ей не интересно, т.е. она ***не подписана на это событие***.

В описанной модели, Женя находится в режиме ожидания до тех пор, пока не будет сгенерировано событие, на которое она подписана. После того как событие будет сгенерировано, Женя приступит к выполнению задачи, которая соответствует сгенерированному событию. В процессе выполнения задачи, Женя так же может генерировать события, на которые могут быть подписаны другие сущности.

Это есть событийно-ориентированная модель в действии.

**Введение в терминологию или «Погружение на глубину!»**

Теперь, когда мы на базовом уровне понимаем, что из себя представляет событийно-ориентированная модель, попробуем дать определение её составным частям.

**Событие *(event)*** – какое-то значимое событие, возникающее в программируемой среде (среде выполнения кода) и на которые можно отреагировать (обработать событие). Таким образом, программист может явно задать конкретное поведении программы при наступлении конкретного события.

Основные среды выполнения JavaScript – это Node.js и браузеры. События можно разделить на системные (system) и пользовательскими (custom).

Системные события ***(system events)* –** это низкоуровневые события, информацию о которых Node.js получает из операционной системы. За обработку системных событий отвечает лежащая в основе ядра Node.js библиотека libuv, написанная на низкоуровневом языке программирования «C». Сам язык JavaScript слишком высокоуровневый и не имеет возможностей для обработки системных событий.

Пользовательские события ***(custom events)*** – это события, которые человек создает сам. Пользовательские события в Node.js создаются и обрабатываются с помощью специальной сущности – JavaScript класса ***EventEmitter***. Имеется виду, что пользовательские события – это не те события, которые вызываются действиями человека, а те события, который созданы программистами с использованием языка JavaScript.

**EventEmitter –** это в первую очередь паттерн проектирования, суть которого заключается в том, чтобы дать возможность с любого места в нашем приложении сообщить о каком-либо событии. Все, кто были «подписаны» на это событие сразу же об этом узнают и как-то отреагируют.

**Генератор событий** (***event emitter***) – это сущность, которая генерирует событие. Вместо слова *генератор* (***emitter)*** ещё можно перевести как ***транслятор***, ***излучатель***, а иногда – «***отправитель»*** или «***издатель».*** Это всё одно и то же.

Нужно понимать, что событие будет транслироваться в любом случае, независимо от того, *подписан* ли кто-то на это событие или нет.

В Node.js, сущности, которые генерируют пользовательские (***custom***) события являются JavaScript классами, которые наследуются от ***класса «EventEmitter».***

*Примечание: просто для справки, в Node.js наследоваться от класса можно не только с помощью ключевого слова «extends», а ещё и с помощью встроенных в Node.js «утилит» (utils). Использовать утилиты для наследования сейчас не рекомендуется!*

*Если столкнетесь где-то в коде с функцией* ***util.inherits()****, не удивляйтесь, это раньше в Node.js так наследовались от классов.*

**Class EventEmitter –** это обычный JavaScript класс, реализующий паттерн EventEmitter. Класс **EventEmitter** содержит в себе методы по работе с событиями. Он помогает разным частям приложения взаимодействовать между собой. Как выглядит класс EventEmitter мы разберём в конце статьи.

**Обработка событий *(event handling*)** – это процесс управления событием, путем *подписки* на событие и вызова *обработчика* при генерации события. Когда говорят, что какое-то событие нужно обработать, просто имеют ввиду что хотят задать поведение программы при наступлении этого события. А как это делается на уровне кода, мы ещё рассмотрим.

**Обработчик события** (***event handler***) – это блок кода, обычно функция (function), которая будет вызвана при генерации события, т.е. это написанный программистом кусок кода, который будет исполнен при срабатывании события. Теперь если вы услышите на YouTube умные фразочки по типу *нужно написать «error handler»,* то будете понимать, что речь идёт просто о написании кода для обработки события «error» (ошибка).

**Слушатель события** (***event listener***) – это процедура или функция, которая ожидает наступления события. Здесь имеется ввиду не столько код, который будет выполняться, а некий механизм, который отслеживает событие, и при его наступлении – вызывает этот самый код «***обработчика***» событий.

Есть разница между «*слушателем события*» (***event listener***) и «*обработчиком события*» (***event handler***). При разработке клиентской (frontend) части приложения эта разница достаточно очевидна, но при разработке серверной (backend) части – сложно уловить разницу.

О разнице между «слушателем» и «обработчиком» на клиентской части приложения можно почитать в англоязычной статье [«What’s the difference between Event Handlers & addEventListener in JS?»](https://www.thatsanegg.com/blog/what%E2%80%99s-the-difference-between-event-handlers-addeventlistener-in-js/), или в русскоязычном учебнике LearnJS в разделе [«Введение в браузерные события»](https://learn.javascript.ru/introduction-browser-events).

Когда же дело касается серверной части приложения, то в англоязычных и русскоязычных источниках понятия «слушателя» и «обработчика» постоянно смешивают.

Событийно-ориентированная модель в Node.js реализована через класс EventEmitter, и он устроен таким образом, что *обработчик (handler)* и *слушатель (listener)* работают в очень тесной связке. Поэтому обычно, применительно к Node.js, понятие слушателя и обработчика будут взаимозаменяемыми синонимами. Обработчика (handler) и слушателя (listener) так же можно называть **«*подписчиками***».

**Подписка на событие** – это связывающая процедура, указывающая, что конкретное событие является значимым для конкретного слушателя или обработчика. Здесь такая же логика как с подпиской на YouTube-канал. Когда на YouTube-канале выходит новое видео, вам приходит уведомление. Ещё один пример – push уведомления от банка – когда с вашей карты списываются деньги, вам сразу приходит уведомление.

*Примечание: на самом деле, такие понятия как* ***подписка****,* ***отписка****,* ***подписчик****,* ***публикация, издатель*** *– это не совсем относится к паттерну EventEmitter. Эти понятия используются в других популярных паттернах проектирования, таких как «****Observer****» (наблюдатель), «****Publish-Subscribe****» (издатель-подписчик) и др. Однако из-за очень схожей логики работы, эти понятия так же можно применять и к паттерну «EventEmitter».*

При генерации события подписанный на это событие слушатель или обработчик узнают о наступлении события и после чего вызывается код обработчика, т.е. путём подписки мы указываем что при наступлении конкретного события должен сработать конкретный программный код.

На одно событие может подписаться неограниченное количество слушателей. В этом случае все обработчики будут вызваны последовательно, друг за другом. Также, один и тот же слушатель может несколько раз подписаться на одно и тоже событие, в этом случае код обработчика будет выполнен несколько раз.

В разговорной речи и в интернете вместо словосочетания *подписка на событие*могут использовать выражения «***зарегистрировать слушателя/обработчик***», «***повесить на событие слушателя/обработчик***» или «***назначить на событие слушателя/обработчик***». Эти выражения имеют одинаковое значение.

**Отписка от события** – процесс, противоположный подписке. После отписки от события мы больше не связываем событие и слушателя.

Такой механизм нужен, например, чтобы код обработчика сработал только один раз после самой первой генерации события, а при повторной генерации обработчик не вызывался. Так же рекомендуют использовать отписку от событий чтобы избежать «утечек памяти».

***Примечание:*** *наткнулся на одном форуме на интересное обсуждение 10-летней давности на тему «*[*Нужно ли удалять события в js?»*](https://javascript.ru/forum/offtopic/33507-nuzhno-li-udalyat-sobytiya-v-js.html)*. Человек из-за непонимания терминологии и непонимания того, как работают события, вместо отписки от события пытается удалить само событие. Рекомендую ознакомиться для общего развития.*

Примерно так хранятся события на уровне кода

class MyEventEmitter {

// Так хранятся события и слушатели внутри класса EventEmitter

  events = {

    close: [func1, func2, ... funcN],

    drop: [func1, func2, ... funcN],

  error: [func1, func2, ... funcN],

...

  }

**События (events)** **Слушатели (listeners)**

...

// А здесь код, который реализует функционал по работе с событиями

}

Как видно, все *слушатели* (***listeners***) хранятся в массиве ([ … ]). Когда мы *подписываем* слушателей на событие, мы добавляем их в массив ([ func1, func2, func3 ]), а когда отписываем одного слушателя от события – мы фактически удаляем его из массива ([ func1, func3 ]). А когда события генерируется, последовательно вызываются все слушатели из массива.

**Всё есть "событие" или «Как устроена вселенная»!**

Вернемся к наше новой знакомой Евгении. В предыдущем примере, Женя – это всего лишь один человек на большой планете Земля. Помимо неё существуют и иные сущности - люди, животные, предметы, природные явления. И они тоже генерирую и отлавливают события: кошка перешла дорогу, температура воздуха поднялась до 25 градусов, тучи закрыли солнце, лифт сломался на 4 этаже, машина посигналила — всё это события.

Теперь, попробуем осмыслить простую идею – **Всё есть событие**!

Представьте любую сущность, любой физический объект или живое существо. Появление сущности — это уже событие, гибель сущности – это тоже событие. А между появлением и гибелью сущность что-то делает и каждое действие генерирует события.

Таким образом, в событийно-ориентированной парадигме мы должны мыслить не столько понятием физического объекта, сколько понятием события, которое генерируется этим объектом.

Схожая концепция в юриспруденции. Юристы мыслят не понятием объекта, а понятием «права на объект». С точки зрения юриста, люди торгуют не объектами, а правами на объект. У любого объекта, даже у пакета с мусором должен быть собственник, обладающий правами владения, пользования и распоряжения на объект.

А теперь, наложим получившуюся картину мира на работу приложения. Код всего приложения – это как окружающий нас мир, а язык программирования (JavaScript) и среда выполнения (Node.js или браузер) – это как законы физики. Наш код может делать только то, что заложено в возможностях самого языка программирования и среды, в которой выполняется код. Код написанный для Node.js, но запущенный в браузере может вызвать ошибку.

В программировании мы управляем абстрактными сущностями – переменными, массивами, функциями, объектами, html-элементами DOM дерева, модулями, библиотеками и т.д. И все эти сущности в теории могут генерировать события прямо с момента их возникновения в нашем коде.

**Обработка событий или «От хаоса к порядку»!**

В итоге, у нас есть огромное количество самых разных «событий», которые мы пока не умеем обрабатывать. Благородная миссия разработчика – подчинить событийный хаос и привнести порядок в эту вселенную!

Первое что мы должны сделать – это отловить «событие» (***event***). Чтобы забить гвоздь, нужно хотя бы схватить молоток. Однако как схватить «событие»? Хорошие новости – всё уже придумано до нас. Молоток в нашу руку вкладывает среда выполнения кода (Node.js или браузер), а точнее их API.

Применительно к JavaScript, для работы на фронтенде нужно смотреть в сторону Web API, а для работы на бэкенде – в сторону API Node.js. По сути API – это документ для разработчиков, который техническим языком объясняет, как и какой код нужно писать, чтобы заставить программу сделать то, что нам от неё нужно. «События», которые предоставляет нам Web API, можно посмотреть, например в документации MDN в разделе «[Справочник по событиям](https://developer.mozilla.org/ru/docs/Web/Events)».

Хотя в теории, любая сущность может генерировать (***emitted***) события (***events***), но на практике, разработчики добавляют в API возможность работы только с теми с событиями, которые решают конкретные полезные задачи на бэкенде или фронтенде. Разработчики постоянно усовершенствуют API, добавляя в неё новые «события» (***events***), которые можно отлавливать и обрабатывать.

Также API предоставляет программистам возможность создавать свои кастомные события, подписываться на них и генерировать события в любой части приложения. Поскольку эти новые события будут реализованы через класс EventEmitter, то это будут пользовательские события (***custom events***).

Обработка «событий» в Node.js и Браузере во многом похожа, но из-за разницы в контексте и внутреннем устройстве, всё-таки отличается.

*Примечание: В одной англоязычной статье автор* ***утверждал****, что в браузере используется паттерн проектирования «****Observer****», а в Node.js – используется паттерн «****EventEmitter****». Если у вас есть свободное время, предлагаю вам углубиться в паттерны, изучить данный вопрос и выяснить был ли прав автор статьи.*

В интернете огромное количество статей с примерами обработки событий на фронтенде. Мне же рассмотрим работу с событиями в Node.js. Мы посмотрим, как работает событие **«drop»** для класса «**net.Server»**, введенное в API Node.js в версии 18.6.0 (в июле 2022 года).

Согласно [документации Node.js](https://nodejs.org/api/net.html#event-drop), с помощью параметра «**server.maxConnections»** мы можем ограничить для сервера количество активных соединений. Например, мы можем разрешить серверу только 1 активное соединение, а все остальные попытки подключения сервер будет отбрасывать (**drop**).

Таким образом, сервер генерирует событие **«drop»,** когда количество подключений достигает порога и сервер отбрасывает каждое новое подключения и вместо подключения каждый раз генерирует событие **«drop»** (отбрасывание).

**Сложновато, не правда ли? Но не переживайте, разобраться в работе сервера нам поможет наша старая знакомая Женя!**

Перед тем как идти дальше, установите ***версию Node.js v19.0.0 или выше*** с официального сайта [Node.js](https://nodejs.org/en/). В интернете в т.ч. на YouTube есть много инструкций [как скачать и установить Node.js](https://www.youtube.com/results?search_query=%D1%83%D1%81%D1%82%D0%B0%D0%BD%D0%BE%D0%B2%D0%B8%D1%82%D1%8C+node.js).

**Немного о TCP-серверах или «Один вечер пятницы из жизни Жени»**

В пятницу вечером Женя пошла с подругами в кафе, где её увидели два парня – Антон и Стас. Оба парня хотят с ней познакомиться. Женя – порядочная девушка, поэтому познакомится только с тем парнем, который первым проявит инициативу. Предложение второго парня Женя отвергнет, потому что она уже будет занята.

**Ну что, у вы уже видите сходство с работой сервера? Не страшно если еще нет, дальше всё станет очевиднее.**

Создадим папку и назовём её как-нибудь по-английски, например «dating» (знакомство, свидание). В данной папке создадим 3 (три) файла: ***Zhenya.js*, *Anton.js*** и ***Stas.js*.**

*Примечание: Весь код можно* [*посмотреть и скачать с GitHub*](https://github.com/GeorgiiGalechyan/code-for-blogs/tree/main/event-driven-programming)*.*

**-** Z***henya.js* –** это наш TCP-сервер (server) – он будет моделировать поведение Жени, которая скучает в кафе и хочет с кем-то познакомиться. Мы разрешим Жене знакомиться только с одним парнем., т.е. разрешим нашему серверу поддерживать только одно активное соединение.

- **A*nton.js*** и ***Stas.js*** – это клиенты (clients) для нашего TCP-сервера – они моделируют поведение Антона и Стаса, которые хотят с познакомиться с Женей. Однако, с Женей сможет познакомиться только самый быстрый из двух, второму Женя откажет, т.е. клиент, который первым попытается установить соединение с сервером - сможет это сделать, а второй клиент не сможет подключиться к серверу т.к. сервер может поддерживать только одно активное соединение, второе и все последующие попытки подключится сервер будет отбрасывать (**drop**). При каждом отбрасывании генерируется событие **«drop».**

Теперь всё стало проще, не так ли? Теперь смоделируем описанную выше модель поведения, прописав соответствующий код в каждый из файлов.

Для разработки я использую бесплатную среду для разработки (***IDE***) - [***Visual Studio Code***](https://code.visualstudio.com/Download), вы же можете писать код хоть в стандартном блокноте Windows, но настоятельно рекомендую использовать любую популярную IDE.

Файл Z***henya.js:***

***```***

***```***

Файл **A*nton.js:***

***```***

***```***

Файл ***Stas.js:***

***```***

***```***

Теперь осталось запустить нашу программу. Для того чтобы понять, как это всё работает на практике, нужно открыть три (3) терминала, и последовательно запустить наши файлы, ***наблюдая за сообщениями, которые будут выводятся в терминалах***. Терминал ещё называют «***консолью»***, но здесь я буду использовать название «терминал***»***.

Я открываю терминалы прямо в Visual Studio Code, но можно воспользоваться и стандартной командной строкой Windows или его аналогом в зависимости от вашей ОС.

[Как запустить три терминала с помощью Visual Studio Code](https://github.com/GeorgiiGalechyan/code-for-blogs/tree/main/event-driven-programming/how-split-terminals) можно посмотреть по [ссылке](https://github.com/GeorgiiGalechyan/code-for-blogs/tree/main/event-driven-programming/how-split-terminals).

Сначала, обязательно нужно запустить сервер, т.е. Женю (файл Z***henya.js***). После чего сервер запуститься, а ***терминал заблокируется***.

Далее, **в другом терминале**, нужно запустить один из оставшихся файлов: ***Anton.js*** или ***Stas.js***. Но имейте ввиду, судьба Жени в ваших руках! С Женей сможет познакомиться только один из парней – Антон или Стас. При запуске первого клиента, установится соединение и ***второй терминал также заблокируется***.

Далее, запустите в третьем терминале оставшийся файл, при этом попытка подключения к серверу провалится и в терминал выведется ошибка.

*Примечание: без спешки повторите последовательный запуск файлов, посмотрите какие сообщения выводятся в консоль. Попробуйте поиграться с кодом чтобы лучше прочувствовать как это работает. Как знать, возможно вы поможете Жене познакомиться с обоими парнями ;-).*

Итак, мы только что посмотрели, как можно обработать событие **«drop»**. Сначала сервер сгенерировал событие «**drop**», мы отловили это событие и заставили программу реагировать так, как нам это было нужно. По этой же схеме обрабатываются и все остальные события.

Но как это всё реализовано на уровне кода?

**Вездесущий EventEmitter или «Раскапываем скелет динозавра»!**

В предыдущем примере мы создавали TCP-сервер с помощью класса «**net.Server».** В документации Node.js указано, что класс «**net.Server»** расширяет (наследует) класс «**EventEmitter»**.

В контексте Node.js, **EventEmitter –** это обычный JavaScript класс, основанный на одноименном паттерне и обеспечивающий работу с событиями.

Паттерн EventEmitter можно реализовать по-разному. Мы не будем изучать [исходный код встроенного в Node.js класса EventEmitter](https://github.com/nodejs/node/blob/main/lib/events.js), потому что он слишком сложен. В интернете так же есть много статей на тему как написать простой EventEmitter с нуля. Здесь, же я покажу «скелет» класса EventEmitter – напишу EventEmitter без реализации методов класса.

```

```

Как именно реализуется тот или иной метод внутри класса – это уже отдельная история. **Хороший пример реализации методов можно посмотреть** [**здесь**](https://gist.github.com/mudge/5830382)**.**

**Весь код, приведенный в статье можно скачать с моего гитхаба.**

Теперь вы знаете, как работают и обрабатываются события в Node.js и как выглядит простой класс EventEmitter изнутри. Реализация класса EventEmitter внутри Node.js сильно сложнее, но нам и не нужно её понимать, достаточно уметь работать с классом EventEmitter через API Node.js

Теперь можете смело открывать [документацию Node.js](https://nodejs.org/api/events.html) и осваивать работу с событиями на практике.

На этом всё!

P.S. Рассчитываю на конструктивную критику читателей! Всегда буду рад обратной связи. Не стесняйтесь указывать на ошибки или хвалить, лучше пишите в комментариях. Особенно буду благодарен если вы предложите темы для статей!

Пока в планах следующие статьи из серии «От новичка, для новичков»:

- Слои абстракции или «Осмыслить неосязаемое!»

- Ввод-вывод (I/O) или «Такому на информатике нас не учили!»;

- Введение в ECMAScript или «Когда бежать впереди паровоза полезно!»

- Операционные системы или «Под капотом невидимого автомобиля!»