**Министерство Образования и Науки Республики Молдова**

**Технический университет Молдовы**

**Факультет Компьютеров, информатики и микроэлектроники**

**Кафедра компьютерных наук и системной инженерии**

**Реферат**

Веб технологии

**Тема: Асинхронность в JS**

**Выполнил: Nicolaev Eduard, группа CR-223**

**Проверил: Bargan Constantin**

**Кишинёв – 2024**

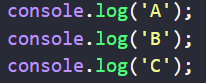
Чтобы понять, что такое асинхронность, сперва поговорим о синхронном коде и том, как в принципе JavaScript выполняет код.

Чтобы выполнить код, нам нужен JavaScript Engine (движок) — программа, которая «читает и выполняет» то, что мы написали. Самый распространённый движок среди всех — это V8, он используется в Google Chrome и Node.js.

Выполнение JS-кода — однопоточное. Это значит, что в конкретный момент времени движок может выполнять не более одной строки кода. То есть вторая строка не будет выполнена, пока не выполнится первая.

Такое выполнение кода (строка за строкой) называется синхронным.

Синхронный код понятный, его удобно читать, потому что он выполняется ровно так, как написан:



Выведется:

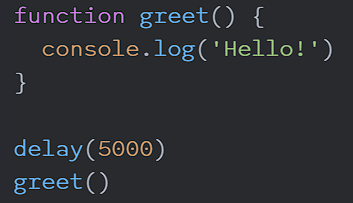
A

B

C

Никаких сюрпризов: в каком порядке команды указаны — в таком они и выполнились.

Однако с ним могут возникать некоторые проблемы. Представим, что нам нужно выполнить какую-то операцию, требующую некоторого времени — например, напечатать в консоли приветствие, но не сразу, а через 5 секунд. Ниже псевдокод — синхронная функция задержки delay() вымышленная:



Через 5 секунд бездействия вывелось бы:

Hello!

И всё вроде хорошо, приветствие бы действительно напечаталось спустя 5 секунд, однако проблема здесь в другом.

Если бы мы запустили синхронную функцию задержки delay(), то движок бы ничем другим заниматься в это время не мог.

Мы помним, что выполнение синхронного кода — строка за строкой. То есть пока delay() не выполнится до конца, к следующей строке интерпретатор не перейдёт.

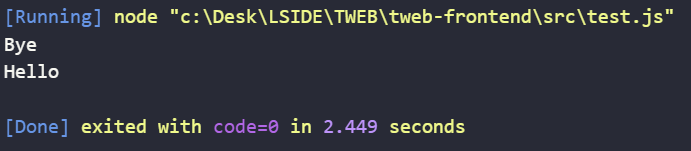
А это значит, что пока не пройдёт 5 секунд, и delay() не выполнится, мы вообще ничего сделать не сможем: ни вывести что-то в консоль ещё, ни выполнить другие функции, в особо тяжёлых случаях — даже передвинуть курсор.

Такие операции, которые не дают выполнять ничего кроме них самих, пока они не завершатся, называются блокирующими выполнение.

Решение с задержкой (async code):



Console:



Это так называемые макротаски.

Возникает несколько вопросов:

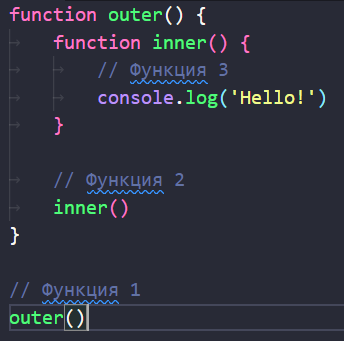
1. Почему вторая строка кода выполнилась до первой, если JS однопоточный?
2. Куда девается setTimeout() на время, пока выполняется другой код?
3. Как движок понимает, что пора выводить Hello!?

Чтобы с этим разобраться, нам надо понять, как функции вызываются «под капотом».

При вызове какой-то функции она попадает в так называемый стек вызовов.

**Stack** — это структура данных, в которой элементы упорядочены так, что последний элемент, который попадает в стек, выходит из него первым (LIFO: last in, first out). Стек похож на стопку книг: та книга, которую мы кладём последней, находится сверху.

В стеке вызовов хранятся функции, до которых дошёл интерпретатор, и которые надо выполнить.



Вызываем функцию 1 — outer(), она попадает в стек. Вызываем функцию 2 — inner(), теперь в стеке 2 функции, потому что первая ещё не выполнилась до конца: inner(); outer(). Вызываем console.log('Hello!'), теперь в стеке 3 функции: console.log('Hello!'); inner(); outer()

Как только console.log('Hello!') выполнится, она уйдёт из стека, там останется 2 функции: inner(); outer()

Выполнившись, функция inner() тоже уйдёт из стека, в нём останется лишь одна: outer()

После выполнения всего блока стек станет пустым.

В синхронном коде в стеке хранится вся цепочка вызовов. Поэтому, например, [рекурсия](https://doka.guide/js/recursion/) без базового случая может приводить к переполнению стека — в нём скапливается слишком большое количество вызовов.

Теперь посмотрим, как ведёт себя стек вызовов при работе с асинхронным кодом:



После выполнения всего блока стек снова становится пустым.

Первое, что бросается в глаза — setTimeout() завершается сразу, хотя колбэк внутри него ещё не отработал, более того, он даже ещё не был вызван! Здесь нам понадобится ещё одно понятие — цикл событий.

**Цикл событий**

 Скопировать ссылку "Цикл событий"

Сперва откроем страшную правду, setTimeout() — это не JavaScript!

Ну... не совсем так, конечно. Функция setTimeout() не является частью JavaScript-движка, это по сути *Web API,* включённое в среду браузера как дополнительная функциональность.

Эта дополнительная функциональность (Web API) берёт на себя работу с таймерами, интервалами, обработчиками событий. То есть, когда мы регистрируем обработчик клика на кнопку — он попадает в окружение Web API. Именно оно знает, когда обработчик нужно вызвать.

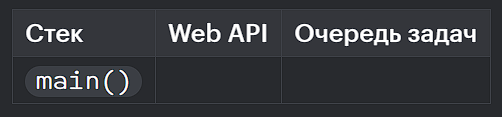
Управление тем, как должны вызываться функции Web API, берёт на себя *цикл событий* (Event loop).

**Event loop** отвечает за выполнение кода, сбор и обработку событий и выполнение подзадач из очереди.

Именно цикл событий ответственен за то, что setTimeout() пропал из стека в прошлом примере. Чтобы увидеть картину целиком, давайте включим в нашу схему все недостающие части. Возьмём тот же самый пример.

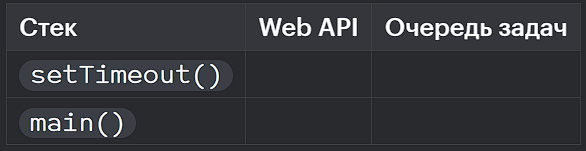
Но теперь у нас будет не только стек вызовов — также мы включим Web API и очередь задач, которую Web API использует для хранения того, что нужно выполнить.

Вызов main():

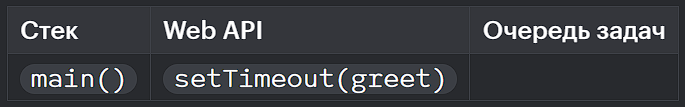


В Web API и очереди задач пока пусто.

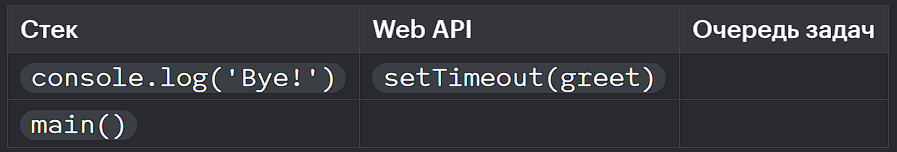
Вызов setTimeout():



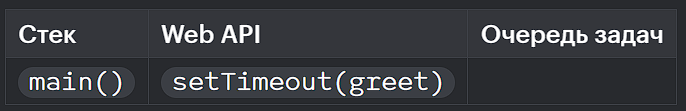
Когда setTimeout() исчезает из стека, он попадает в видимость Web API, где интерпретатор понимает, что внутри него есть функция greet(), которую надо выполнить через 2 секунды:



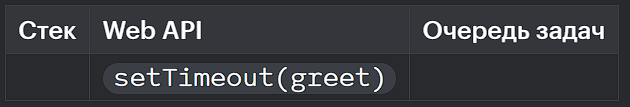
После этого выполняется вызов консоли console.log('Bye!'). В Web API находится функция setTimeout(greet). Она будет там до тех пор, пока не пройдёт 2 секунды:



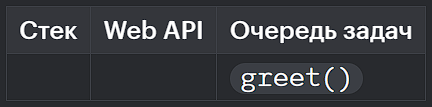
Отработал console.log(), заканчивается работа main():



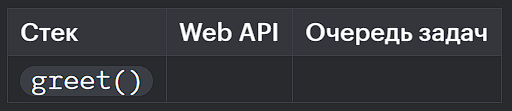
main() отработал, стек пуст. 2 секунды ещё не прошло, поэтому setTimeout(greet) все ещё в Web API:



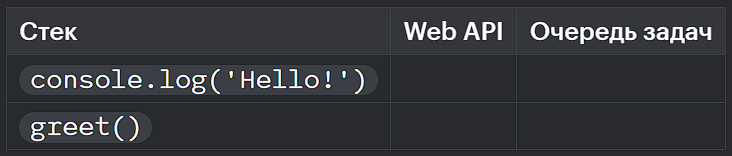
Наконец, 2 секунды прошли - функция greet() перемещается в очередь задач:



Теперь цикл событий перемещает функцию greet() из списка задач в вызов:



Затем вызов console.log('Hello!'):



И наконец стек пуст.

Заметьте, что стек вызовов и очередь задач называются именно стеком и очередью. Потому что вызовы из стека работают по принципу *«последний зашёл, первый вышел»* (LIFO: last in, first out), а в очереди — по принципу *«первый зашёл, первый вышел»* (FIFO: first in, first out).

[**Очередь**](https://ru.wikipedia.org/wiki/%D0%9E%D1%87%D0%B5%D1%80%D0%B5%D0%B4%D1%8C_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) — структура данных, в которой элементы упорядочены так, что первый попавший в очередь элемент покидает её первым.

Таким образом цикл событий работает с асинхронным кодом — то есть таким, который выполняется не построчно.

Очень хорошо работу цикла событий иллюстрирует [инструмент Loupe](http://latentflip.com/loupe/) Филипа Робертса, а также его доклад «[What the heck is the event loop anyway?](https://www.youtube.com/watch?v=8aGhZQkoFbQ)».

[Loupe](http://latentflip.com/loupe/) интерактивный, попробуйте ввести какой-нибудь код в поле слева, и справа будет показываться, что и в какой момент попадает в стек вызовов и очередь событий:

**Колбэки**

 Скопировать ссылку "Колбэки"

Пример с setTimeout(), который мы рассмотрели, показывает, как работают функции обратного вызова — колбэки.

[**Callback**](https://ru.wikipedia.org/wiki/Callback_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) (колбэк, функция обратного вызова) — функция, которая вызывается в ответ на совершение некоторого события.

В нашем случае таким событием было срабатывание таймера через 2 секунды, а колбэком — функция greet(). В целом, событием может быть что угодно:

* ответ от сервера;
* завершение какой-то длительной вычислительной задачи;
* получение доступа к каким-то API устройства, на котором выполняется код.

Таким образом *колбэк* — это первый способ обработать какое-либо асинхронное действие.

Изначально колбэки были единственным способом работать с асинхронным кодом в JavaScript. Большая часть асинхронного API Node.js была написана именно на колбэках и создана для использования с колбэками.

Это, в принципе, логично — ментальная модель достаточно простая: *«выполни эту функцию, когда случится это событие»*.

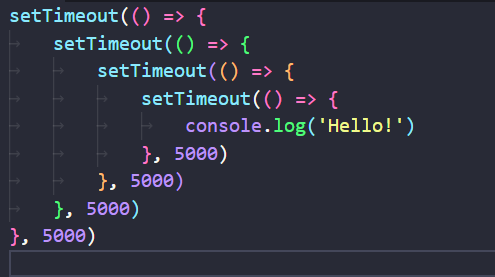
Однако у колбэков есть неприятный минус, так называемый *ад колбэков* (callback hell).

**Ад колбэков (Callback-hell)**

 Скопировать ссылку "Ад колбэков (Callback-hell)"

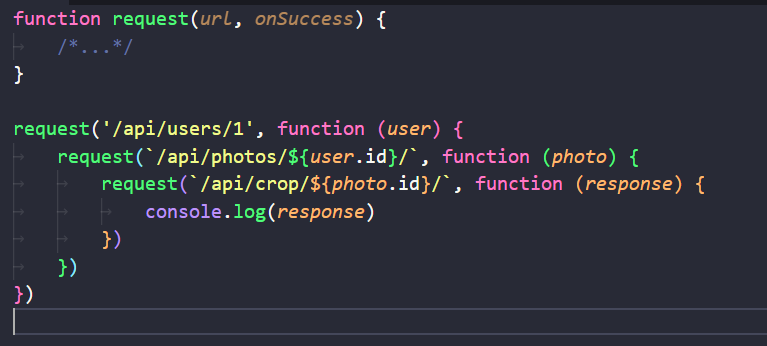
Нагляднее всего его можно показать на примере.

Допустим, у нас есть ряд асинхронных задач, которые зависят друг от друга: то есть первая задача запускает по завершении вторую, вторая — третью и т. д.



Если одна задача запускает другую, та — третью, и так далее, мы можем получить вот такую «башню» из обратных вызовов.

И такая башня может получиться где угодно. Если мы делаем несколько последовательных запросов к серверу, зависящих друг от друга, то это может выглядеть вот так:



Читать такое сложно, не говоря уже о тестировании, которое здесь становится очень накладным.

Решить эту проблему были призваны Промисы (Promise).

**Промисы (Promise)**

 Скопировать ссылку "Промисы (Promise)"

[**Промис**](https://doka.guide/js/promise/) — это объект-обёртка для асинхронного кода. Он содержит в себе состояние: вначале pending («ожидание»), затем — одно из: fulfilled («выполнено успешно») или rejected («выполнено с ошибкой»).

В понятиях цикла событий промис работает так же, как колбэк: функция, которая должна выполниться (resolve или reject), находится в окружении Web API, а при наступлении события — попадает в очередь задач, откуда потом — в стек вызова.

В асинхронных задачах есть [разделение между макрозадачами и микрозадачами](https://medium.com/javascript-in-plain-english/javascript-event-loop-y-promises-951ba6845899). Колбэки в промисах попадают в очередь микрозадач, тогда как колбэк в setTimeout() — в очередь макрозадач. Но здесь и сейчас мы в такие детали уходить не будем.

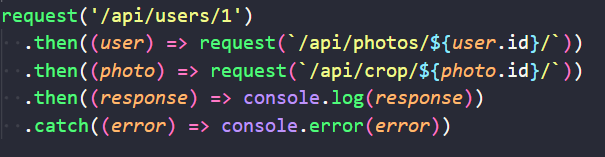
Промисы придумали, чтобы организовывать асинхронный код последовательно.

Та же последовательность запросов из прошлого примера, но переписанная с использованием промисов.



Код избавился от лишней вложенности, стал плоским и более тестируемым.

Дополнительным плюсом стала возможность обрабатывать ошибки от цепочки промисов в одном месте — последним catch:



Если что-то пошло не так, то программа не упадёт, а управление перейдёт к последней строчке с catch(), причём независимо от того, в каком из запросов ошибка появится.

Также из then() можно вернуть не промис, а обычное значение. Оно обернётся в промис самостоятельно и прокинется в следующий then():



Промисы оказались удобными, и появился даже такой термин как «[промисификация](https://nodejs.org/dist/latest-v8.x/docs/api/util.html" \l "util_util_promisify_original)» — когда асинхронную функциональность на колбэках превращали в промисы.

Однако промисы — это тоже не серебряная пуля. У них есть несколько недостатков:

* Код не такой лаконичный, как мог быть.
* В цепочке промисов, как на примере (со стрелочными функциями), невозможно выставить брейкпоинт, потому что нет тела функции. Приходится раскрывать функцию.
* Стек ошибок может содержать в себе then.then.then.then....
* Вложенные условия сильно увеличивают количество кода и ухудшают читаемость.

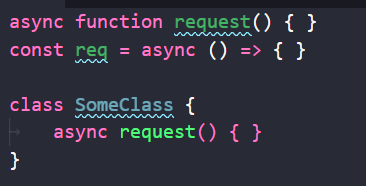
Для решения этих проблем придумали асинхронные функции.

**Асинхронные функции**

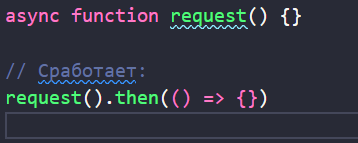
 Скопировать ссылку "Асинхронные функции"

Если коротко, [**асинхронные функции**](https://doka.guide/js/async-await/) — функции, которые возвращают промисы.

Асинхронная функция помечается специальным ключевым словом async:



Они *всегда* возвращают Промис. Даже если мы явно этого не указывали, как в примерах выше, при вызове они всё равно вернут промис.

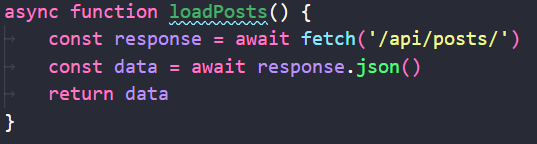


Однако с асинхронными функциями можно не обращаться с then() — есть более изящное решение.

**Связка async/await**

 Скопировать ссылку "Связка async/await"

Внутри асинхронных функций можно вызывать другие асинхронные функции, без каких-либо then() или колбэков, с помощью ключевого слова await.



В примере выше мы используем метод [fetch()](https://doka.guide/js/fetch/) внутри функции loadPosts().

Все асинхронные функции внутри мы вызываем с await — таким образом промис, который функция возвращает, автоматически разворачивается, и мы получаем значение, которое внутри промиса было.

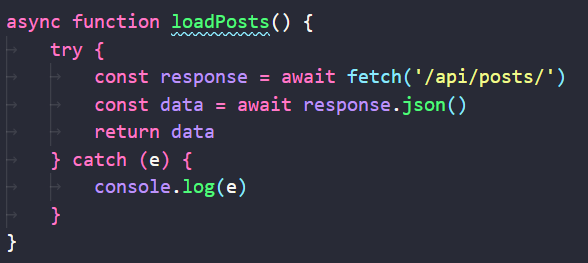
**Плюсы async/await**

 Скопировать ссылку "Плюсы async/await"

**Код чище и короче.** У нас больше нет цепочек из then(), вместо этого мы получаем плоскую структуру, которая по виду похожа на синхронный код.

Условия и вложенные конструкции становятся чище и проще читаются.

**Мы можем обрабатывать ошибки с try-catch.** Как и с синхронным кодом, обработка ошибок сводится к оборачиванию опасных операций в try-catch:



При этом в отличие от catch() промисов, try-catch поймает не только ошибки, которые были внутри асинхронных функций, но также и ошибки, которые возникли во время обычных синхронных операций.

**Можно ставить брейкпоинты (точки остановки).** Для отладки мы можем поставить брейкпоинт куда угодно, он сработает.

**Источники:**

<https://fuse8.ru/articles/async-javascript>

<https://learn.javascript.ru/async-await>