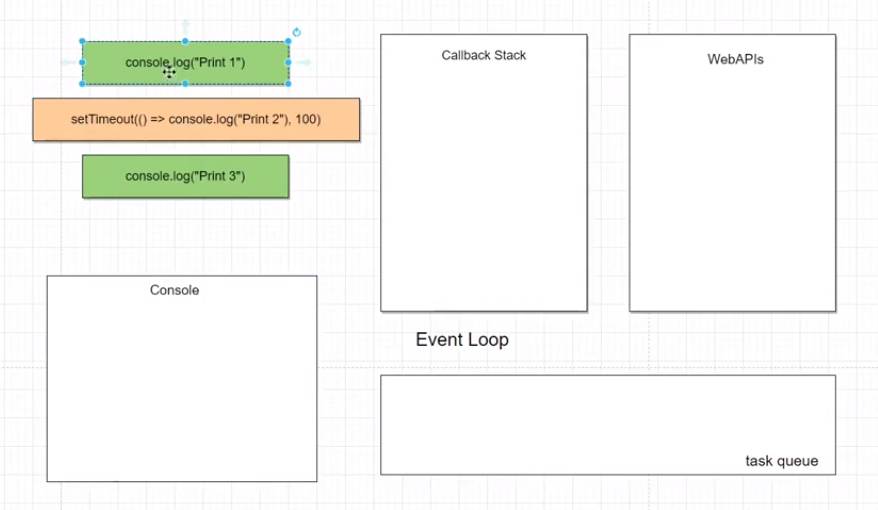
***Асинхронный JavaScript***

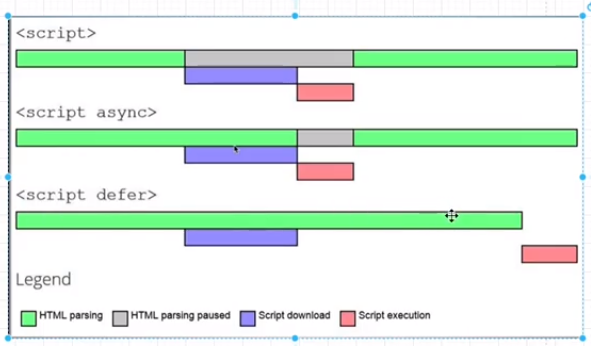
Модуль 8

Для асинхронного взаимо-действия не требуется полная перезагрузка браузера, данные подгружаются постепенно.

Пример:

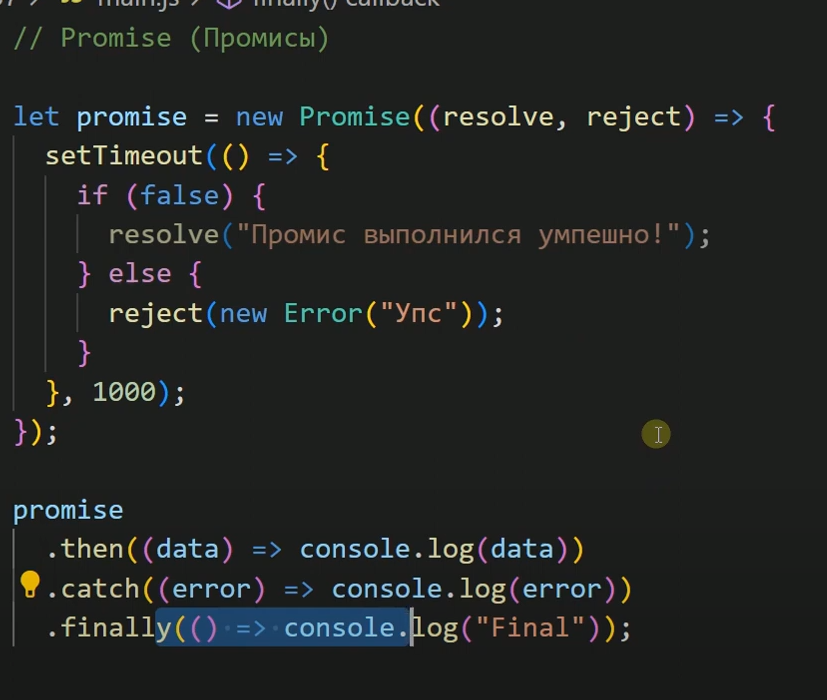
Обычные события подгружаются в callback stack. Асинхронные подгружаются в WebAPIs, а затем оттуда попадают в task queue (очередь задач). Потом из очереди задач, ***после освобождения основного стека***, – выполняются асинхронные задачи.

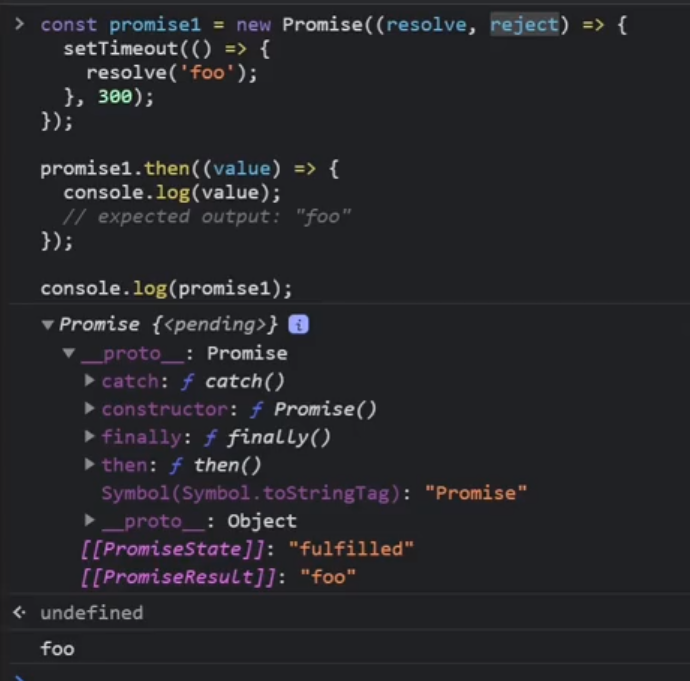
***HTTP-протокол*** позволяет работать над сайтом асинхронно; ***WebSocket*** – нужен же для постоянного обмена данными, загрузки, переброски итп. – подходит для мессенджеров.

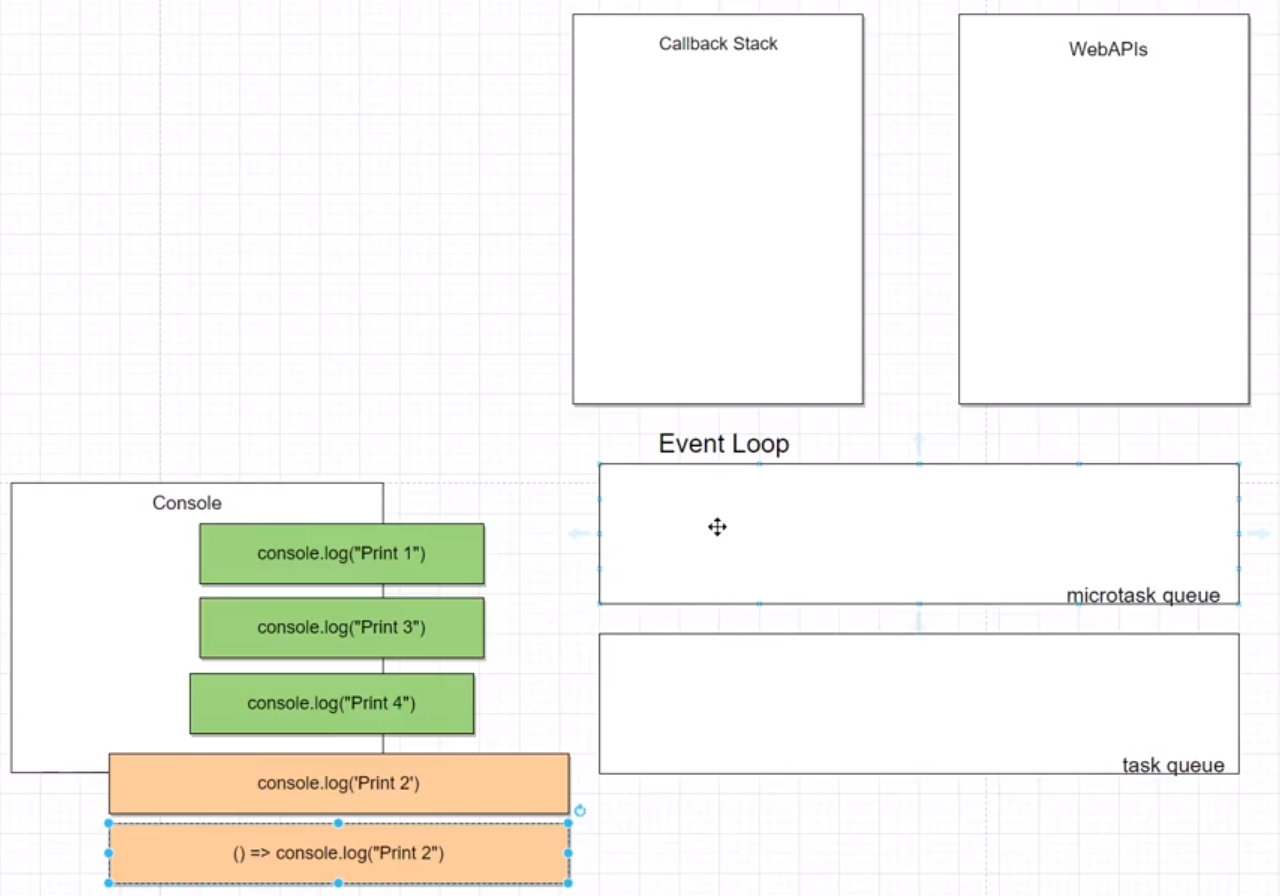
Асинхронность и загрузка скриптов

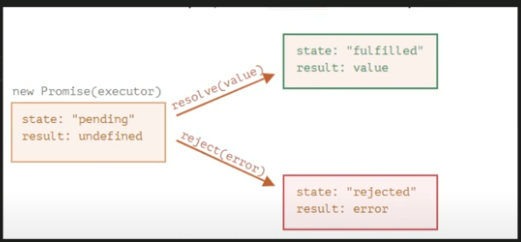
***Promise***

Это объект, для работы с асинхронными запросами. Resolve – разрешить; reject – отвергнуть.









**const** p = new Promise((resolve) **=>** {

  setTimeout(() **=>** resolve('Hello, '), 1000);

})

p.then((slovo) **=>** {

  console.log(slovo);

  return new Promise((resolve) **=>** {

    setTimeout(() **=>** resolve('world! '), 2000);

  }).then((slovo2) **=>** {

    console.log(slovo2);

    return new Promise((resolve) **=>** {

      setTimeout(() **=>** resolve('Parapam'), 2000);

    }).then((slovo3) **=>** {

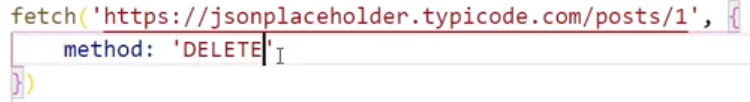
      console.log(slovo3);

    })

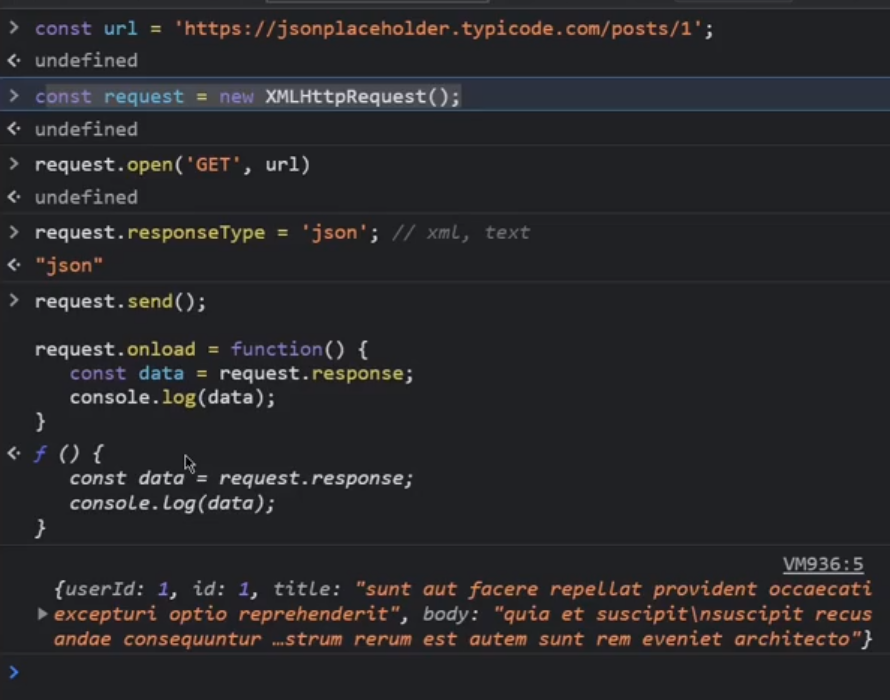
  })

}).catch((err)**=>**console.log(err));

Позволяет нам ловить ошибки (схоже с конструкцией try-catch).

* ***CRUD – create, read, update, delete.***
* ***HTTP – post, get, put | patch, delete***





*API (Application programming interface)* — это контракт, который предоставляет программа. «Ко мне можно обращаться так и так, я обязуюсь делать то и это».

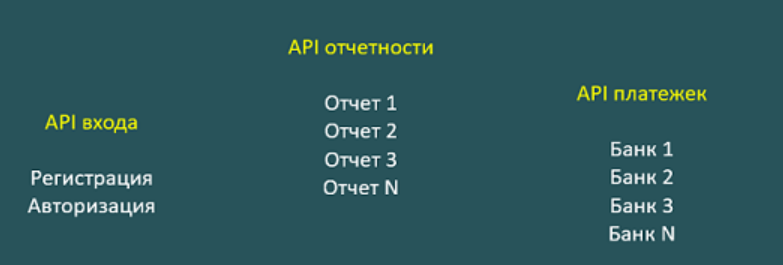
Если переводить на русский, это было бы слово «договор». Договор между двумя сторонами, как договор на покупку машины;

API — набор функций

Когда вы покупаете машину, вы составляете договор, в котором прописываете все важные для вас пункты.

Точно также и между ***программами должны составляться договоры.*** Они указывают, как к той или иной программе можно обращаться. Соответственно, API отвечает на вопрос “Как ко мне, к моей системе можно обратиться?”, и включает в себя:

* саму операцию, которую мы можем выполнить,
* данные, которые поступают на вход,
* данные, которые оказываются на выходе (контент данных или сообщение об ошибке).

Да потому что API — ***это набор функций***. Это может быть одна функция, а может быть много.

Получается, что *разработчик придумывает, какое у него будет API*. Либо делает общее, либо распределяет по функционалу или каким-то своим критериям, и в каждое апи добавляет тот набор функций, который ему необходим.

Да потому, что ***в программировании контракт — это и есть интерфейс!***

В классическом описании ООП (объектно-ориентированного программирования) есть 3 кита:

1. *Инкапсуляция*
2. *Наследование*
3. *Полиморфизм*

Есть типичная пирамида автоматизации:

* GUI-тесты — честный тест, «как это делал бы пользователь».
* API-тесты — опускаемся на уровень ниже, выкидывая лишнее.
* Unit-тесты — тесты на отдельную функцию

**GUI-тесты** — честный тест, робот делает все, что делал бы пользователь. Открывает браузер, тыкает на кнопочки… Но если что-то упадет, будете долго разбираться, где именно.  
**API-тесты** — все то же самое, только без браузера. Мы просто подаем данные на вход и проверяем данные на выходе. Например, можно внести итоговый ответ в эксельку, и пусть робот выверяет ее, правильно ли заполняются данные? Локализовать проблему становится проще.  
**Unit-тесты** — это когда мы проверяем каждую функцию отдельно. Отдельно смотрим расчет для ячейки 1, отдельно — для ячейки 2, и так далее. Такие тесты шустрее всего гоняются и баги по ним легко локализовать.

***Метод send() [XMLHttpRequest](https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest) отправляет запрос на сервер.***

Если request является асинхронным (что является значением по умолчанию), этот метод возвращается, как только *Запрос отправляется*, а результат доставляется с помощью событий. Если запрос синхронный, Этот метод не возвращается до тех пор, пока не *будет получен ответ*.

send() принимает необязательный параметр, который позволяет указать свойство запроса тело; в основном используется для таких запросов, как [PUT.](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/PUT) Если *запрос*[*GET*](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/GET) или HEAD, параметр игнорируется[,](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/HEAD) а тело запроса устанавливается в . bodynull.

* ***async*** function getUsers() {
* const response = ***await*** fetch(‘https://…’);
* const data = await response.json();
* console.log(data);
* }
* getUsers();
* console.log(1);
* console.log(2);

await – позволяет запустить функцию после остальных синхронных.

*try-catch-finally-throw*

try

{

// Обычно этот код без сбоев работает от начала до конца.

// Но в какой-то момент в нем может быть сгенерировано исключение

// либо непосредственно с помощью оператора throw, либо косвенно -

// вызовом метода, генерирующего исключение.

}

catch (ex)

{

// Инструкции в этом блоке выполняются тогда и только тогда, когда в блоке try

// возникает исключение. Эти инструкции могут использовать локальную переменную ex,

// ссылающуюся на объект Error или на другое значение, указанное в операторе throw.

// Этот блок может либо некоторым образом обработать исключение, либо

// проигнорировать его, делая что то другое, либо заново сгенерировать

// исключение с помощью оператора throw.

}

finally

{

// Этот блок содержит инструкции, которые выполняются всегда, независимо от того,

// что произошло в блоке try. Они выполняются, если блок try завершился:

// 1) как обычно, достигнув конца блока

// 2) из-за операторов break, continue или return

// 3) с исключением, обработанным приведенным в блоке catch выше

// 4) с неперехваченным исключением, которое продолжает свое

// распространение на более высокие уровни

}

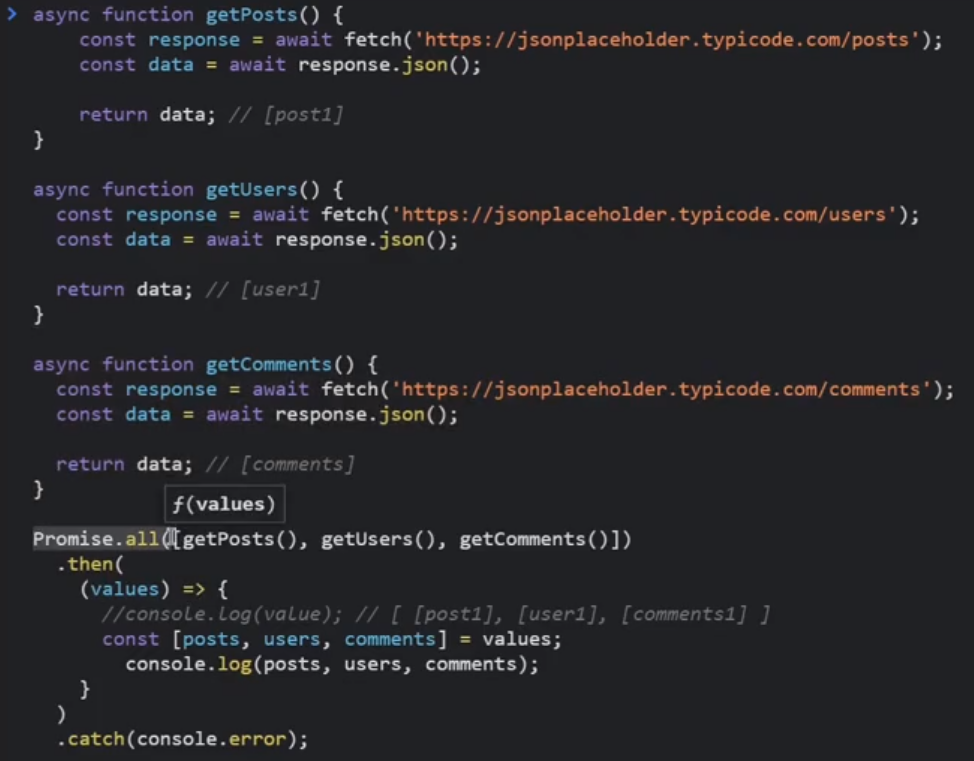
**Асинхронные функции** – альтернатива промисам. (по умолчанию возвращают промис).

* *Что такое Promise in JavaScipt?*

Promise (обещание) - это специальный объект JavaScript. Он выдает значение после успешного завершения асинхронной операции или ошибку, если она не завершается успешно из-за тайм-аута, сетевой ошибки и так далее.

Успешное завершение вызова указывается вызовом функции resolve, а ошибки - вызовом функции reject.





***Promise.all(iterable)*** – ожидает исполнения всех промисов или отклонения любого из них.

***Promise.allSettled(iterable)*** - ожидает завершения всех полученных промисов.

**async** **function** getPostBerlin() {

**const** response = await fetch('https://jsonplaceholder.typicode.com/berlin/posts');

**const** data = await response.json();

  return data; *//[post1]*

}

**async** **function** getPostLosAngeles() {

**const** response = await fetch('https://jsonplaceholder.typicode.com/losangeles/posts');

**const** data = await response.json();

  return data; *//[post1]*

}

**async** **function** getPostDefault() {

**const** response = await fetch('https://jsonplaceholder.typicode.com/default/posts');

**const** data = await response.json();

  return data; *//[post1]?*

}

Promise.race([getPostBerlin(), getPostLosAngeles(), getPostDefault()].then());

***Promise.race()*** – ожидает исполнения или отклонения одного из полученных промисов.