Front-end

JavaScript

JS. Асинхронный JS, Promise

- Асинхронный JS
- Promise
- fetch
- JSON
- throw
- Error

JS. Асинхронный JS, Promise

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.

JS. Асинхронный JS, Promise

Синхронный код и его проблемы

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

```
console.log('A')
console.log('B')
console.log('C')
```

Выведется:

A

B

C

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

JS. Асинхронный JS, Promise

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:

```
function greet() {
  console.log('Hello!')
}

delay(5000)
greet()
```

JS. Асинхронный JS, Promise

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.

JS. Асинхронный JS, Promise

Асинхронный код

Теперь попробуем решить эту же задачу, но так, чтобы наш код не блокировал выполнение. Для этого мы воспользуемся функцией setTimeout():

```
setTimeout(function greet() {
  console.log('Hello!')
}, 5000)
```

5 секунд молчания, и выведется «Hello!»

Задача решена. В этот раз, однако, в эти «5 секунд молчания» мы можем выполнять другие действия.

```
setTimeout(function greet() {
  console.log('Hello!')
}, 5000)


console.log("I'm being called before greet function.")
```

JS. Асинхронный JS, Promise

Асинхронный код

Сначала выведется: «I'm being called before greet function», а через 5 секунд — «Hello!»

Возникает несколько вопросов:

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

JS. Асинхронный JS, Promise

Промис (Promise) — специальный объект JavaScript, который используется для написания и обработки асинхронного кода.

Асинхронные функции возвращают объект Promise в качестве значения. Внутри промиса хранится результат вычисления, которое может быть уже выполнено или выполнится в будущем.

Промис может находиться в одном из трёх состояний:

- pending — стартовое состояние, операция стартовала;
- fulfilled — получен результат;
- rejected — ошибка.

Поменять состояние можно только один раз: перейти из pending либо в fulfilled, либо в rejected

JS. Асинхронный JS, Promise

У промиса есть методы then() и catch(), которые позволяют использовать результат вычисления внутри промиса.

Промис создаётся с помощью конструктора.

В конструктор передаётся функция-исполнитель асинхронной операции (англ. executor). Она вызывается сразу после создания промиса.

Задача этой функции — выполнить асинхронную операцию и перевести состояние промиса в fulfilled (успех) или rejected (ошибка).

JS. Асинхронный JS, Promise

```
const promise = new Promise(function (resolve, reject) {
  const data = getData()
  resolve(data)
})
```

- первый параметр (в примере кода назван resolve) — колбэк для перевода промиса в состояние fulfilled, при его вызове аргументом передаётся результат операции;
- второй параметр (в примере кода назван reject) — колбэк для перевода промиса в состояние rejected, при его вызове аргументом передаётся информация об ошибке.

JS. Асинхронный JS, Promise

Промис решает задачу выполнения кода, который зависит от результата асинхронной операции.

Промис устроен таким образом, что рычаги управления его состоянием остаются у асинхронной функции. После создания, промис находится в состоянии ожидания pending. Когда асинхронная операция завершается, функция переводит промис в состояние успеха fulfilled или ошибки rejected.

С помощью методов then(), catch() и finally() мы можем реагировать на изменение состояния промиса и использовать результат его выполнения.

JS. Асинхронный JS, Promise

Методы

В работе мы чаще используем промисы, чем создаём. Использовать промис — значит выполнять код при изменении состояния промиса.

Существует три метода, которые позволяют работать с результатом выполнения вычисления внутри промиса:

- then()
- catch()
- finally()

JS. Асинхронный JS, Promise

Метод then() используют, чтобы выполнить код после изменения состояния промиса.

Метод принимает два аргумента:

- onFulfill — функция-колбэк, которая будет вызвана при переходе промиса в состояние «успех» fulfilled. Функция имеет один параметр, в который передаётся результат выполнения операции
- onReject — функция-колбэк, которая будет вызвана при переходе промиса в состояние «ошибка» rejected. Функция имеет один параметр, в который передаётся информация об ошибке

Всегда возвращает новый промис.

Так как then() всегда возвращает новый промис, то его удобно использовать для построения последовательностей асинхронных операций.

then() в индустрии используется только для обработки успешного завершения операции, в варианте с одним аргументом

JS. Асинхронный JS, Promise

Метод **then()** используют, чтобы выполнить код после успешного выполнения асинхронной операции.

Например, мы запросили у сервера список фильмов и хотим отобразить их на экране, когда сервер получит результат. В этом случае:

- асинхронная операция — запрос данных у сервера;
- код, который мы хотим выполнить после её завершения, — отрисовка списка.

Метод then() принимает в качестве аргумента две функции. Если промис в состоянии fulfilled то выполнится первая функция. Если в состоянии rejected — вторая.

Хорошей практикой считается не использовать второй аргумент метода then() и обрабатывать ошибки при помощи метода catch().

JS. Асинхронный JS, Promise

Метод **catch()** используют для обработки ошибки при выполнении асинхронной операции.

Метод принимает один аргумент:

- onReject — функция-колбэк, которая будет вызвана при переходе промиса в состояние «ошибка» rejected. Функция имеет один параметр, в который передаётся информация об ошибке.

Возвращает промис.

JS. Асинхронный JS, Promise

catch() выполняет переданный ему колбэк когда асинхронная операция:

- вызывает функцию reject() внутри промиса.
- выбрасывает ошибку с помощью throw.

Всегда завершайте свои цепочки промисов вызовом метода catch(). Если в одной из операций в цепочке произойдёт ошибка, и она не будет обработана, то JavaScript выведет сообщение Uncaught (in promise) Error в консоль разработчика и перестанет работать на всей странице.

JS. Асинхронный JS, Promise

Метод **catch()** используют, чтобы выполнить код в случае ошибки при выполнении асинхронной операции.

Например, мы запросили у сервера список фильмов и хотим показать экран обрыва соединения, если произошла ошибка. В этом случае:

асинхронная операция — запрос данных у сервера;

код, который мы хотим выполнить при ошибке — экран обрыва соединения.

Метод catch() принимает в качестве аргумента функцию-колбэк, которая выполняется сразу после того, как промис поменял состояние на rejected. Параметр колбэка содержит экземпляр ошибки

JS. Асинхронный JS, Promise

Метод **finally()** используют для выполнения кода при завершении промиса. Код выполнится как при переходе промиса в состояние fulfilled, так и в rejected.

Метод принимает один аргумент:

- onDone — функция-колбэк, которая будет вызвана при завершении промиса.

Возвращает новый промис.

JS. Асинхронный JS, Promise

finally() выполняет переданный ему колбэк независимо от того, как завершилась асинхронная операция.

Метод используют для того, чтобы избежать повторения кода между then() и catch(). Обычно такой код занимается уборкой после операции — скрывает индикаторы загрузки, закрывает меню и т.д.

Колбэк у finally() не содержит параметров. Это следствие того, что колбэк будет вызван как при успехе, так и при ошибке.

JS. Асинхронный JS, Promise

finally() отлично работает в случаях, когда нужно убрать лоадер со страницы или кнопки. Он сработает вне зависимости от результата промиса, поэтому можно избежать дублирования кода.

Цепочки методов

Методы then(), catch() и finally() часто объединяют в цепочки вызовов, чтобы обработать и успешный, и ошибочный сценарии

JS. Асинхронный JS, Promise

```
let isLoading = true


fetch(`https://swapi.dev/api/films/${id}/`)
  .then(function (movies) {
    renderList(movies)
  })
  .catch(function (err) {
    renderErrorMessage(err)
  })
  .finally(function () {
    isLoading = false
  })
```

JS. Асинхронный JS, Promise

Метод **all()** — это один из статических методов объекта Promise. Метод all() используют, когда нужно запустить несколько промисов параллельно и дождаться их выполнения.

Promise.all() принимает итерируемую коллекцию промисов (чаще всего — массив) и возвращает новый промис, который будет выполнен, когда будут выполнены все промисы, переданные в виде перечисляемого аргумента, или отклонён, если хотя бы один из переданных промисов завершится с ошибкой.

Метод Promise.all() возвращает массив значений всех переданных промисов, при этом сохраняя порядок оригинального (переданного) массива, но не порядок выполнения.

JS. Асинхронный JS, Promise

Метод **allSettled()** — это один из статических методов объекта Promise. Его используют, когда нужно запустить несколько промисов параллельно и дождаться их выполнения.

Promise.allSettled() очень похож на метод Promise.all(), но работает немного по-другому. В отличие от Promise.all(), Promise.allSettled() ждёт выполнения всех промисов, при этом неважно, завершились они успешно или с ошибкой.

Promise.allSettled() принимает итерируемую коллекцию промисов (чаще всего — массив) и возвращает новый промис, который будет выполнен, когда будут выполнены все переданные промисы. Полученный промис содержит массив результатов выполнения всех переданных промисов, сохраняя порядок оригинального массива, но не порядок выполнения.

JS. Асинхронный JS, Promise

Метод **any** — это один из статических методов объекта Promise. Его используют, когда нужно запустить несколько промисов параллельно и дождаться первого успешного разрешённого.

Promise.any() принимает итерируемую коллекцию промисов (чаще всего — массив) и возвращает новый промис, который будет выполнен, когда будет выполнен первый из промисов, переданных в виде перечисляемого аргумента, или отклонён, если все из переданных промисов завершатся с ошибкой.

Возвращает значение первого успешно выполнившегося промиса.

Метод полезен, когда нужно вернуть первый исполненный промис. После того как один из промисов будет исполнен, метод не будет дожидаться исполнения остальных.

JS. Асинхронный JS, Promise

Метод **race()** — это один из статических методов объекта Promise. Его используют, чтобы запустить несколько промисов и дождаться того, который выполнится быстрее.

Promise.race() принимает итерируемую коллекцию промисов (чаще всего — массив) и возвращает новый промис.

Он завершится, когда завершится самый быстрый из всех переданных. Остальные промисы будут проигнорированы.

JS. Асинхронный JS, Promise

Fetch

С помощью функции **fetch()** можно отправлять сетевые запросы на сервер — как получать, так и отправлять данные. Метод возвращает промис с объектом ответа, где находится дополнительная информация (статус ответа, заголовки) и ответ на запрос.

Браузер предоставляет глобальный API для работы с запросами и ответами HTTP. Раньше для подобной работы использовался XMLHttpRequest, однако fetch() более гибкая и мощная альтернатива, он понятнее и проще в использовании из-за того, что использует Promise.

JS. Асинхронный JS, Promise

Fetch

Функция **fetch()** принимает два параметра:

- url — адрес, по которому нужно сделать запрос;
- options (необязательный) — объект конфигурации, в котором можно настроить метод запроса, тело запроса, заголовки и многое другое.

По умолчанию вызов fetch() делает GET-запрос по указанному адресу. Базовый вызов для получения данных можно записать таким образом:

```
fetch('http://jsonplaceholder.typicode.com/posts')
```

JS. Асинхронный JS, Promise

Fetch

Результатом вызова **fetch()** будет Promise, в котором будет содержаться специальный объект ответа Response. У этого объекта есть два важных для нас поля:

- ok — принимает состояние true или false и сообщает об успешности запроса;
- json — метод, вызов которого, возвращает результат запроса в виде json.

```
fetch('http://jsonplaceholder.typicode.com/posts')
  .then((response) => response.json())
  .then((data) => data)
```

JS. Асинхронный JS, Promise

Fetch

С помощью второго аргумента options можно передать настройки запроса. Например, можно изменить метод и добавить тело запроса, если мы хотим не получать, а отправлять данные. Так же в запрос можно добавить заголовки в виде объекта или специального класса Headers

GET, POST, PUT, PATCH и DELETE - это пять наиболее распространенных HTTP-методов для получения и отправки данных на сервер.

JS. Асинхронный JS, Promise

Метод GET

Метод GET используется для получения данных с сервера. Это метод предназначен только для чтения, поэтому риск изменения или повреждения данных отсутствует.

Метод POST

Метод POST отправляет данные на сервер и создает новый ресурс. Когда этот новый ресурс помещается в родительский объект, служба API автоматически создает с ним связь, назначая свой идентификатор (URI нового ресурса). Простыми словами, этот метод используется для создания новой записи данных.

JS. Асинхронный JS, Promise

Метод PUT

Метод PUT чаще всего используется для обновления существующего ресурса. Для этого необходим URI ресурса и новая его версия.

Метод PATCH

Метод PATCH очень похож на метод PUT, поскольку он также изменяет существующий ресурс. Разница в том, что для метода PUT тело запроса содержит полную новую версию, тогда как для метода PATCH тело запроса должно содержать только конкретные изменения.

JS. Асинхронный JS, Promise

Метод DELETE

Метод DELETE используется для удаления ресурса, который указывается с помощью его URI.

JS. Асинхронный JS, Promise

Оператор throw

Оператор throw позволяет создавать пользовательские ошибки.

В техническом смысле вы можете генерировать исключения (генерировать ошибки).

Исключения могут быть строкой, числом, логическим значением или объектом JavaScript

JS. Асинхронный JS, Promise

Error и стандартные ошибки

Программа может работать правильно, только если код написан корректно и не содержит ошибок. JavaScript умеет обрабатывать некорректный код и сообщать об ошибке в коде. Существует семь встроенных видов ошибок, также можно создать свои собственные. Встроенные ошибки генерируются самим движком JavaScript при выполнении программы, а пользовательские — создаются с помощью конструктора Error. Оба типа ошибок можно ловить в конструкции try...catch.

JS. Асинхронный JS, Promise

Error

Общий конструктор ошибок.

new Error('Общая ошибка. Проверьте код')

Вызов конструктора возвращает объект ошибки со следующими свойствами:

- message представляет человекопонятное описание ошибки для встроенных типов (SyntaxError, TypeError и так далее) и переданное в конструктор значение для общего типа Error.
- name — имя типа (класса) ошибки.

```
const commonError = new Error('Общая ошибка. Проверьте код')


console.log(commonError.message)
// 'Общая ошибка. Проверьте код'
console.log(commonError.name)
// 'Error'
```

JS. Асинхронный JS, Promise

JSON (JavaScript Object Notation) — самый популярный формат обмена данными между приложениями. Этот формат очень похож на объекты JavaScript. Объекты легко трансформируются в JSON для отправки на сервер.

JS. Асинхронный JS, Promise

JSON состоит из пар ключ-значение. Пары разделяются между собой запятыми — ,, а ключ отделяется от значения через двоеточие — :. Ключом может быть только строка, обёрнутая в двойные кавычки. А вот значением — почти всё что угодно:

- Строка в двойных кавычках — "I love JSON!";
- Число — 21;
- Логическое значение — true;
- Массив — [18, true, "lost", [4, 8, 15, 16, 23, 42]];
- Объект — {"isValid": true, "isPayed": false}.

JSON основан на JavaScript, но является независимой от языка спецификацией для данных и может использоваться почти с любым языком программирования, поэтому он пропускает некоторые специфические значения объектов JavaScript:

Методы объектов (функции) — {greetings() {alert("Hello World!")}};

Ключи со значением undefined — {"value": undefined}.

Если нужно сохранить JSON в файл, то используют расширение .json.

JS. Асинхронный JS, Promise

JSON используется для того, чтобы получить данные от сервера. Типичная схема работы:

- Отправляем запрос на сервер;
- Ждём ответ;
- Получаем JSON с набором данных;
- Превращаем JSON в объект JavaScript;
- Используем данные.

JS. Асинхронный JS, Promise

Для того что бы превратить данные в JSON-код, используйте метод JSON.stringify(). Первым аргументом метод принимает значение, которое нужно преобразовать.

Преобразуем JavaScript-объект в JSON:

```
const hero = {
  nickname: "BestHealerEver",
  level: 7,
  age: 141,
  race: "Gnome",
}

console.log(typeof hero)
// object
console.log(typeof JSON.stringify(hero))
// string
console.log(JSON.stringify(hero))
```

JS. Асинхронный JS, Promise

Преобразование из JSON

Преобразовать строку с JSON в объект JavaScript можно с помощью метода JSON.parse(). Он принимает JSON-строку в качестве аргумента.

С помощью JSON.parse() мы получим стандартный объект, с которым можно взаимодействовать

JS. Асинхронный JS, Promise

```
const json = {
 "name": "Luke Skywalker",
 "height": "172",
 "mass": "77",
 "hair_color": "blond",
 "skin_color": "fair",
 "eye_color": "blue",
 "birth_year": "19BBY",
 "gender": "male"
}
const jedi = JSON.parse(json)

console.log(jedi.name)
// Luke Skywalker
console.log(jedi.gender)
// male
console.log(jedi.birth_year)
// 19BBY
```

JS. Асинхронный JS, Promise

В случае, если строка не является валидным JSON-кодом, метод JSON.parse() выбросит ошибку SyntaxError.