Сетевые запросы. Колбэки, промисы, async/await

Колбэки - функции обратного вызова

Функции называют «асинхронными», потому что действие (загрузка скрипта) будет завершено не сейчас, а потом.
Аргументом в такие функции передаётся функция (обычно анонимная), которая выполняется по завершении действия. Это и есть функция обратного вызова.

Колбэки - функции обратного вызова

callback — функция, которая будет вызвана по завершению асинхронного действия.

Обработка ошибок:

```
function load(scriptUrl, callback) {
    let script = document.createElement('script');
    script.src = scriptUrl;

    script.onload = () => callback(null, script);
    script.onerror = () => callback(new Error(`Ошибка загрузки`));

    document.head.append(script);
}
```

Адская пирамида колбэков

```
loadScript('1.js', function(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', function(error, script) {
      if (error) {
        handleError(error);
      } else {
        // ...
        loadScript('3.js', function(error, script) {
          if (error) {
            handleError(error);
          } else {
            // ...
          }
        });
      }
    })
  }
});
```

Промисы

Объект Promise (промис) используется для отложенных и асинхронных вычислений.

```
const newPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('test');
  }, 300);
});

newPromise.then((value) => {
  console.log(value);
  // "test"
});
```

Промисы

Функции-потребители могут быть зарегистрированы (подписаны) с помощью методов .then, .catch и .finally.

```
promise.then(
    function(result) { /* успешное выполнение */ },
    function(error) { /* ошибка */ }
);
```

Первый аргумент метода .then – функция, которая выполняется, когда промис переходит в состояние «выполнен успешно», и получает результат.

Второй аргумент .then – функция, которая выполняется, когда промис переходит в состояние «выполнен с ошибкой», и получает ошибку.

Промисы

new Promise(function(resolve, reject) { ... });

Функция, переданная в конструкцию new Promise, называется исполнитель (executor). Когда Promise создаётся, она запускается автоматически. Она должна содержать «создающий» код, который когда-нибудь создаст результат.

Её аргументы resolve и reject – это колбэки, которые предоставляет сам JavaScript. Наш код – только внутри исполнителя.
Когда он получает результат, сейчас или позже – не важно, он должен вызвать один из этих колбэков:

- resolve(value) — если работа завершилась успешно, с результатом value.
- reject(error) — если произошла ошибка, error – объект ошибки.

Промисы

Если мы хотели бы только обработать ошибку, то можно использовать null в качестве первого аргумента: .then(null, errorFunc). Или можно воспользоваться методом .catch(errorFunc), который сделает тоже самое

.finally(() => alert("Промис завершён"))

функция в аргументе finally выполнится в любом случае, когда промис завершится: успешно или с ошибкой.

Promise.all

```
Promise.all([
  new Promise(resolve => setTimeout(() => resolve(1), 3000)), // 1
  new Promise(resolve => setTimeout(() => resolve(2), 2000)), // 2
  new Promise(resolve => setTimeout(() => resolve(3), 1000))  // 3
]).then(alert);
```

Async/await

Специальный синтаксис для работы с промисами - «async/await»

```
async function newFunc() {
  return 1;
}
// эта функция всегда
возвращает промис
```

Async/await

Ключевое слово await заставит интерпретатор JavaScript ждать до тех пор, пока промис справа от await не выполнится. После чего оно вернёт его результат, и выполнение кода продолжится.

Async/await

```
async function newFunc() {

  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve(1), 1000)
  });

  let result = await promise; // будет ждать, пока промис не выполнится (*)

  alert(result); // 1
}

newFunc();
```

Сетевые запросы

Для сетевых запросов в JavaScript можно использовать методы fetch (для современных браузеров) или XMLHttpRequest

Fetch

let response = fetch(url, [options])

- url – URL для отправки запроса.
- options – дополнительные параметры: метод, заголовки и так далее.

Свойства response -

- status – код статуса HTTP-запроса, например 200.
- ok – логическое значение: будет true, если код HTTP-статуса в диапазоне 200-299.

Fetch

Методы для response:

- response.text() – читает ответ и возвращает как обычный текст,
- response.json() – декодирует ответ в формате JSON,
- response.formData() – возвращает ответ как объект FormData
- response.blob() – возвращает объект как Blob (бинарные данные с типом),
- response.arrayBuffer() – возвращает ответ как ArrayBuffer (низкоуровневое представление бинарных данных),
- помимо этого, response.body – можно считывать тело запроса по частям

Fetch

```
fetch('https://api.github.com/repos/javascript-tutorial/en.javascript.info/commits')
  .then(response => response.json())
  .then(commits => console.log(commits));
```

Fetch

Получить хедеры ответа:
response.headers

Получить хедер Content-Type ответа:
response.headers.get('Content-Type')

Fetch

Установить хедеры запроса:

```
let response = fetch(URL, {
  headers: {
    Authentication: 'secret'
  }
});
```

Fetch

Для отправки POST-запроса или запроса с другим методом, нам необходимо использовать fetch параметры:

- method – HTTP метод, например POST,
- body – тело запроса

```
let response = await fetch('/article/fetch/post/user', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json;charset=utf-8'
  },
  body: JSON.stringify(user)
});
```

XMLHttpRequest

1. Создать XMLHttpRequest.

```
let xhr = new XMLHttpRequest();
```

2. Инициализировать его.

```
xhr.open(method, URL, [async, user, password])
```

3. Послать запрос.

```
xhr.send([body])
```

XMLHttpRequest

4. Слушать события на xhr, чтобы получить ответ.

```
xhr.onload = function() {
  alert(`Загружено: ${xhr.status} ${xhr.response}`);
};

xhr.onerror = function() { // происходит, только когда запрос совсем не получилось выполнить
  alert(`Ошибка соединения`);
};
```

IT SHATLE

Задача 1

Создайте асинхронную функцию getTasks(), которая возвращает массив объектов категорий. Данные получать по ссылке: https://test-todoist.herokuapp.com/api/tasks

Необходимо вывести имена описание этих задач на странице в виде списка

Задача 2

Создайте асинхронную функцию getUsers(names), которая получает на вход массив логинов пользователей GitHub, запрашивает у GitHub информацию о них и возвращает массив объектов-пользователей.

Информация о пользователе GitHub с логином USERNAME доступна по ссылке: https://api.github.com/users/USERNAME.

IT SHATLE

Задача 3

Необходимо создать страницу

На странице должен выводиться:

- список категорий, полученных по этому GET запросу: https://test-todoist.herokuapp.com/api/categories
- кнопка добавить категорию (POST запрос - https://test-todoist.herokuapp.com/api/categories) - после успешной отправки этого запроса - добавленная категория появляется в списке категорий

Вид тела (body) для POST запроса:

```
{
  "id": 0,
  "name": "string"
}
```