JavaScript – Асинхронные функции (async/await)

Главным нововведением java script 2017 году стали асинхронные функции, которые призваны навсегда изменить, то как мы пишем асинхронный код. В этой статье будет детально разбираться, что из себя представляет асинхронные функции и как они работают. Перед этим давайте рассмотрим проблему, которую они пытаются решить. У нас имеются три функции ***getUser***, ***getPosts*** и ***getComments***. Исходя из названия ***getUser*** получает пользователя, ***getPosts*** получает посты и **getComments** получает комментарий поста.

```
const { getUser, getPosts, getComments } = require('./db');

getUser(1, (error, user) => {

    if (error) return console.error(error);

    getPosts(user.id, (error, posts) => {

        if (error) return console.error(error);

        getComments(posts[0].id, (error, comments) => {

            if (error) return console.error(error);
```

```
            console.log(comments);

        });

    });

});
```

Первой вызывается функция ***getUser***, куда мы отправляем id пользователя и функцию обратного вызова. Первое, что мы делаем в функции обратного вызова это проверяем наличие ошибки, если она есть то мы просто выводим ее в консоль. Если все хорошо, то мы вызываем функцию ***getPosts***, куда отправляем id пользователя и функцию обратного вызова, в которой вначале также проверяем на ошибку. Если все хорошо то вызываем функцию ***getComments***, куда отправляем id первого поста и в функции обратно вызову также проверяем на ошибку и если все хорошо, то выводим в консоль комментарии. В данном примере трудно не заметить пирамидальную фигуру, которая увеличивается с добавлением вложенных функций. Это, так называемый, ***callback hell*** или ад функций обратного вызова.

Отчасти, решением проблемы вложенности функции обратного вызова является использование обещаний, который эту вложенность убирают. Также, обещание предоставляют более удобный способ обработки ошибок. Лично мне кажется, что обещание являются очень элегантным способом написания синхронного кода. Но многим они пришлись не по душе особенно тем, кто пришел в java script из других языков, в которых асинхронный код пишется синхронно.

```
const { getUser, getPosts, getComments } = require('./db');

getUser(1)
    .then(user => getPosts(user.id))
    .then(posts => getComments(posts[0].id))
    .then(comments => console.log(comments))
    .catch(error => console.error(error));
```

Альтернативой обещаниям стали генераторы, которые были добавлены в язык вместе с обещаниями. Генераторы это особый тип функций, которые можно поставить на паузу, вернуть промежуточный результат получить что-либо обратно и продолжить выполнение функции. Сами по себе генераторы не приспособлены для написания синхронного кода, но если их использовать вместе с обещаниями, то в результате мы получаем нечто уникальное: асинхронный код, который выглядит синхронно. И при этом генераторы, также предоставляют очень удобный и знакомый механизм обработки ошибок с помощью конструкций ***try\catch***. Однако, у генераторов есть один большой недостаток, для того чтобы их использовать с обещаниями вам понадобится отдельная функция, которая будет управлять процессом работы генератора. Эту функцию вы можете написать сами или же использовать стороннюю библиотеку типа ***co***. В данном примере я написал свою реализацию такой функции.

```
const { getUser, getPosts, getComments } = require('./db');
```

```
function co(generator) {

    const iterator = generator();

    return new Promise((resolve, reject) => {

        function run(prev) {

            const { value, done } = iterator.next(prev);

            if (done) {

                resolve(value);

            } else if (value instanceof Promise) {

                value.then(run, reject);

            } else {

                run(value);

            }

        }
```

```
        run();

    });

}

co(function* () {

    try {

        let user = yield getUser(1);

        let posts = yield getPosts(user.id);

        let comments = yield getComments(posts[0].id);

        console.log(comments);

    }

    catch (error) {

        console.error(error);

    }
```

```
});
```

У каждого из способов работы с асинхронным кодом есть свои плюсы и недостатки. Функции обратного вызова просты в использовании, но с увеличением количества вложенных вызовов читаемость и понимание кода начинает страдать обещание элегантны, но трудны в понимании. Для начинающих генераторы позволяют писать асинхронный код синхронно, но для них требуется отдельная функция и сам механизм работы генераторов весьма запутан.

Асинхронные функции были созданы на основе обещаний и генераторов для того, чтобы сделать работу с асинхронным кодом простой и понятной. Давайте разберемся так ли это для того, чтобы разобраться с тем, что из себя представляет асинхронные функции рассмотрим простой пример. У нас имеется функция ***getUser***, которая принимает ***id*** пользователя. Пока это значение никак не используется и с функцией мы возвращаем простой объект.

```
function getUser(id) {

    return { id };

}

let user = getUser(1);

console.log(user);
```

На пятой строчке мы вызываем функцию getUser и результат помещаем в переменную ***user*** и далее выводим значение переменной ***user*** в консоль. Если запустим приложение, то мы получим объект. Теперь давайте сделаем функцию ***getUser*** асинхронной. Для этого перед словом ***function*** добавим ключевое слово ***async.***

```
async function getUser(id) {

    return { id };

}

let user = getUser(1);

console.log(user);
```

Сейчас от функций ***getUser*** мы получили обещание, в котором находится значение, а именно объект пользователя.

Итак, асинхронная функция возвращает обещание для получения значения из обещания воспользуемся методом ***then***.

```
async function getUser(id) {

    return { id };

}
```

```
getUser(1).then(user => console.log(user));
```

Если снова запустим программу, то увидим объект. Если асинхронная функция возвращает обещание, то что будет если из асинхронной функции мы сами вернем обещание? Давайте посмотрим, для этого из функций мы вернём результат вызова метода ***resolve*** у объекта ***Promise.***

```
async function getUser(id) {

    return Promise.resolve({ id });

}

getUser(1).then(user => console.log(user));
```

Напомню, данный метод возвращает выполненное обещание. Если запустим программу, то увидим объект. Если асинхронная функция непосредственно возвращает обещание, то она не оборачивает его в другое обещание.

Давайте посмотрим что произойдет, если внутри асинхронный функции произойдет ошибка. Для этого вместо возврата значения вызовем ошибку.

```
async function getUser(id) {

    return new Error('Oops!');
```

```
}

getUser(1).then(user => console.log(user));
```

Если запустим программу, то ***node*** сообщит нам о том, что мы не обрабатываем отвергнутые

обещания. Для того чтобы это сделать, заменим метод ***then*** на метод ***catch*** параметр назовем ***error***.

```
async function getUser(id) {

    return new Error('Oops!');

}

getUser(1).catch(error => console.log(error));
```

Ну и соответственно выведем его в консоль. Попробуем. Теперь в терминале, мы увидим отбрасываемую ошибку.

Как видите, сами по себе синхронные функции с точки зрения написания кода мало что меняют. Они возвращают обещания, которые нам нужно обрабатывать соответствующим образом. Как я сказал в самом начале асинхронные функции создавались для того, чтобы полностью изменить то, как мы пишем асинхронный код, то есть вместо того, чтобы обрабатывать обещания и работать с функциями обратного вызова. Мы хотим писать

асинхронный код следующим образом:

```
async function getUser(id) {

    return { id };

}

let user =getUser(1);

console.log(user);
```

Объявляем переменную ***user*** и в качестве значений указываем вызов функции ***getUser*** и только после того как функция ***getUser*** вернет значение, которое мы помещаем в переменную ***user,*** мы выводим его в консоль, то есть мы вернулись обратно к синхронному коду. Для написания кода в таком виде одной синхронной функции недостаточно. Нужно написать так:

```
async function getUser(id) {

    return { id };

}

let user = await getUser(1);
```

```
console.log(user);
```

Перед вызовом асинхронной функции нам необходимо указать еще одно ключевое слово ***await***, которое с английского переводится как ждать или ожидать. В данном примере ***await*** дожидается выполнение обещания от функции ***getUser*** вытаскивает из него объект пользователя и возвращает его. Ну и далее мы помещаем его в переменную. Строчка 7 выполнится только после того, как в переменной уже будет значение, то есть весь код, который находится ниже ключевого слова ***await*** будет выполнен после того, как выполнится сама функция которую ***await*** ожидает.

Если запустить данный код, то получим синтаксическую ошибку:

Unexpected Identifier

Дело в том, что ключевое слово ***await*** можно использовать только внутри асинхронной функции. Исправить проблему мы можем следующим образом. Мы создадим еще одну асинхронную функцию:

```
async function getUser(id) {

    return { id };

}

async function main() {
```

```
   let user = await getUser(1);

   console.log(user);

}

main();
```

Обратите внимание, что ключевое слово ***await*** не обязательно ставить перед вызовом асинхронной функции. Это может быть любая функция, которая возвращает обещание.

```
function getUser(id) {

    return Promise.resolve({ id });

}

async function main() {

   let user = await getUser(1);

   console.log(user);

}
```

```
main();
```

Здесь функцию ***getUser*** я сделал простой функцией. Из неё мы возвращаем обещание с помощью ***Promise.resolve***. Если скрипт запустить, то можно увидеть тот же самый результат. Еще раз, ***await*** ставится перед вызовом любой функции, которая возвращает обещание, но это необязательно должна быть асинхронная функция.

Теперь давайте сделаем функцию ***getUser*** реалистичной, она будет получать данные от реального сервера. Первое, что нужно сделать это установить библиотеку для отправки запросов. Воспользуемся библиотекой ***node-fetch***,

она предоставит функцию ***fetch***.

npm i node-fetch

Далее импортируем ***node-fetch*** и напишем следующий код:

```
const fetch = require('node-fetch');

function getUser(id) {

                                                    return
fetch(`https://jsonplaceholder.typicode.com/users/${id}`);

}
```

```
async function main() {

   let user = await getUser(1);

   console.log(user);

}

main();
```

В функции ***getUser*** я убрал ***Promise.resolve***. Вместо этого я написал ***fetch(***`https://jsonplaceholder.typicode.com`***).*** Jsonplaceholder это сервис, который предоставляет фейковые JSON данные. Их можно использовать для тестирования.

Функция ***getUser*** получает параметр ***id*** который я вставил в url. Теперь функция ***getUser*** возвращает результат вызова функции ***fetch***. В терминале мы видим полученные данные и это явно не объект пользователя. Функция ***fetch*** обратно возвращает объект ответа у него есть свойство ***url, status, statusText,*** заголовки и другие свойства и методы. Нам нужно определить что именно должна возвращать функция ***getUser.*** Объект ответа или объект пользователя, исходя из названия конечно второе, объект пользователя. То есть мы вернем не результат вызова функции ***fetch***, а обработаем обещание.

```
const fetch = require('node-fetch');

function getUser(id) {
```

```
                                                    return
fetch(`https://jsonplaceholder.typicode.com/users/${id}`)

.then(response => response.json());

}

async function main() {

   let user = await getUser(1);

   console.log(user);

}

main();
```

Параметры функции то есть объект ответа мы называем response и далее из него возвращаем данные. Это делаем с помощью метода ***json***. Из функции, которую мы отправляем в ***then*** получаем обещание. То есть, в результате функция ***getUser***, также возвращает обещание, но это обещание уже будет оборачивать объект пользователя, который мы помещаем в переменную ***user***.

Если запустить программу, то в терминале мы увидим объект пользователя. В функции ***main*** мы используем ключевое слово ***await*** для того, чтобы дождаться выполнения

обещания получаемое от ***getUser***, благодаря этому код выглядит синхронно, но при этом работает асинхронно. В функции ***getUser*** мы по старинке используем обещания.

Вместо этого мы также можем воспользоваться ключевым словом ***await*** для того, чтобы упростить код функции ***getUser*** и сделать так, чтобы он также выглядел синхронным но при этом работал асинхронно.

```
const fetch = require('node-fetch');

async function getUser(id) {

                    let response = await fetch(`https://jsonplaceholder.typicode.com/users/${id}`);

    let response = await response.json();

    return data;

}

async function main() {

   let user = await getUser(1);

   console.log(user);
```

```
}

main();
```

Итак, функция ***getUser*** вместо того, чтобы возвращать результат вызова ***fetch*** мы его поместили в переменную ***response.*** Так как ***fetch*** возвращает обещание перед ним мы указали ключевое слово ***await***. Дальше мы получим данные ***data***. У объекта ***response*** метод ***json***, также возвращает обещание, поэтому также можем указать ключевое слово ***await.*** Так как в функции ***getUser*** мы используем ключевое ***await*** необходимо сделать ее асинхронной, поставив перед ней ключевое слово ***async.*** Если запустить код, то результат будет тот же самый, мы получаем объект пользователя. При этом код обеих функций выглядит синхронно, но работает асинхронно.

Мы уже с вами видели, что произойдет если внутри асинхронный функции произойдет ошибка, то обратно вернется отвергнутое обещание. Если мы работаем с обещанием, то обработать ошибку можно, либо с помощью второго аргумента в метод ***then***, либо с помощью метода ***catch.*** Но как обработать ошибку, если перед вызовом функции стоит ***await***? На самом деле все очень просто. Для обработки ошибки мы можем использовать старый добрый ***try/catch***:

```
const fetch = require('node-fetch');

async function getUser(id) {

                    let response = await
fetch(`https://jsonplaceholder.typicode.com/users/${id}`);
```

```
    let response = await response.json();

    return data;

}

async function main() {

    try {

        let user = await getUser(1);

        console.log(user);

    } catch (error) {

        console.error(error);

    }

}

main();
```

Итак, функция ***main*** код по получению пользователя я поместил в

блок ***try***, если произойдет ошибка то мы укажем блок ***catch*** и в консоли мы выведем ошибку.

По идее и код внутри функции ***getUser*** также можно поместить в ***try/catch*** это может быть полезно, если мы хотим предоставить пользователю более понятное сообщение об ошибке. Например, из блока ***catch*** мы выкинем еще одну ошибку, в которой напишем: «Не удалось получить данные от сервера». Если ошибка вдруг произойдёт в методе ***json*** у объекта ***response*** функция ***getUser***, также вернет ошибку: «Не удалось получить данные от сервера».

```
const fetch = require('node-fetch');

async function getUser(id) {

    try {

                            let response = await fetch(`https://jsonplaceholder.typicode.com/users/${id}`);

        let response = await response.json();

        return data;

    } catch (error) {

          throw new Error('Не удалось получить данные от сервера');
```

```
    }

}

async function main() {

    try {

        let user = await getUser(1);

        console.log(user);

    } catch (error) {

        console.error(error);

    }

}

main();
```

Разобравшись с основами работы асинхронных функций давайте посмотрим на различные варианты синтаксиса. Во-первых, объявления функции мы можем заменить на выражение функции. Вместо ***async*** зададим константу ***getUser*** в качестве значения

укажем асинхронную функцию, название можно не указывать.

```
const fetch = require('node-fetch');

const getUser = async function(id) {

    try {

                              let   response   =   await
fetch(`https://jsonplaceholder.typicode.com/users/${id}`);

        let response = await response.json();

        return data;

    } catch (error) {

          throw new Error('Не удалось получить данные от
сервера');

    }

}

async function main() {
```

```
    try {

        let user = await getUser(1);

        console.log(user);

    } catch (error) {

        console.error(error);

    }

}

main();
```

Вместо анонимной функции мы можем воспользоваться стрелочной функцией.

```
const fetch = require('node-fetch');

const getUser = async (id) => {

    try {
```

```
                    let response = await fetch(`https://jsonplaceholder.typicode.com/users/${id}`);

        let response = await response.json();

        return data;

    } catch (error) {

          throw new Error('Не удалось получить данные от сервера');

    }

}

async function main() {

    try {

        let user = await getUser(1);

        console.log(user);

    } catch (error) {
```

```
        console.error(error);

    }

}

main();
```

Асинхронная функция также может быть значением для свойства у объекта.

```
const fetch = require('node-fetch');

const dataService = {

    getUser: async function(id) {

        try {

                                let response = await fetch(`https://jsonplaceholder.typicode.com/users/${id}`);

            let response = await response.json();

            return data;
```

```
        } catch (error) {

            throw new Error('Не удалось получить данные от сервера');

        }

    }

}

async function main() {

    try {

        let user = await dataService.getUser(1);

        console.log(user);

    } catch (error) {

        console.error(error);

    }
```

```
}

main();
```

Итак, мы создали объект и в качестве значения указали асинхронную функцию. Вместо стрелочной функции мы указали простую анонимную функцию. Так как у стрелочных функций нет параметра ***this***.

Вместо свойства с функцией в качестве значения мы можем определить метод. Для этого все между скобками и «:» уберем и перед названием метода укажем ключевое слово ***async:***

```
const fetch = require('node-fetch');

const dataService = {

    async getUser(id) {

        try {

                                let response = await
fetch(`https://jsonplaceholder.typicode.com/users/${id}`);

            let response = await response.json();

            return data;
```

```
        } catch (error) {

            throw new Error('Не удалось получить данные от сервера');

        }

    }

}

async function main() {

    try {

        let user = await dataService.getUser(1);

        console.log(user);

    } catch (error) {

        console.error(error);

    }
```

```
}

main();
```

Асинхронный метод мы также можем определить и у класса:

```
const fetch = require('node-fetch');

class DataService {

    async getUser(id) {

        try {

                                    let   response   =   await
fetch(`https://jsonplaceholder.typicode.com/users/${id}`);

            let response = await response.json();

            return data;

        } catch (error) {

             throw new Error('Не удалось получить данные от
сервера');
```

```
        }

    }

}

async function main() {

    let dataService = new DataService();

    try {

        let user = await dataService.getUser(1);

        console.log(user);

    } catch (error) {

        console.error(error);

    }

}
```

Обратите внимание на функцию ***main*** мы ее создали с единственной

целью: иметь возможность использовать ключевое слово ***await***, ведь для того чтобы запустить главный код программы, функцию ***main*** нам нужно вызвать. Вместо этого мы можем создать самовызывающуюся функцию **IIFE**,которая также может быть асинхронной. Такой функции не обязательно присваивать название, более того мы можем сделать ее стрелочной.

```
const fetch = require('node-fetch');

class DataService {

    async getUser(id) {

        try {

                                let response = await fetch(`https://jsonplaceholder.typicode.com/users/${id}`);

            let response = await response.json();

            return data;

        } catch (error) {

             throw new Error('Не удалось получить данные от сервера');
```

```
        }

    }

}

(async () => {

    let dataService = new DataService();

    try {

        let user = await dataService.getUser(1);

        console.log(user);

    } catch (error) {

        console.error(error);

    }

})();
```

Давайте доработаем наш пример и в кластере сервис добавим два

метода. Первый будет получать и отправлять посты пользователя, а второй будет получать и соответственно отправлять комментарии конкретного поста.

```
const fetch = require('node-fetch');

class DataService {

    constructor(url) {

        this.url = url;

    }

    async getUser(id) {

        try {

                                let response = await fetch(`${this.url}/users/${id}`);

            let response = await response.json();

            return data;

        } catch (error) {
```

```
            throw new Error('Не удалось получить данные пользователя');

        }

    }

    async getPosts(userId) {

        try {

                                let response = await fetch(`${this.url}/posts?userId=${userId}`);

            let response = await response.json();

            return data;

        } catch (error) {

            throw new Error('Не удалось получить посты');

        }

    }
```

```
    async getComments(postId) {

        try {

                                let response = await
fetch(`${this.url}/comments?postId=${postId}`);

            let response = await response.json();

            return data;

        } catch (error) {

                throw new Error('Не удалось получить
комментарии');

        }

    }

}

(async () => {

                    let dataService = new
DataService('https://jsonplaceholder.typicode.com');
```

```
    try {

        let user = await dataService.getUser(1);

        let posts = await dataService.getPosts(user.id);

                            let comments = await
dataService.getComments(posts[0].id);

        console.log(comments);

    } catch (error) {

        console.error(error);

    }

})();
```

Еще раз хочу обратить ваше внимание на, то как выглядит код. Он выглядит синхронно. Сначала мы получаем пользователя далее мы получаем посты пользователя и дальше мы получаем комментарий конкретного поста. После того, как мы получили необходимые данные мы выводим их в консоль. Если на каком-то из этапов произойдет ошибка, то мы ее поймаем, ну и в данном случае просто выводим в консоль. И при этом механизм работы асинхронных функций основан на обещаниях и генераторах, но нас это не касается. Мы получаем удобный и понятный способ

написания асинхронного кода, то о чем java script разработчики мечтали уже очень давно.