Лабораторная работа 6

Работа с API вконтакте с использованием async/await

Задание

Задание на эту лабораторную работу такое же, как и для предыдущей, однако реализовать её нужно с помощью async/await. Необходимо создать HTML страницу с встроенным скриптом, который будет в зависимости от варианта выполнять следующие действия:

- собирать список id аккаунтов, являющихся друзьями тех, кто состоит в определенной группе или встрече
- поставьте лайк верхнему посту на стене 5 тем вашим друзьям, которые были последними в онлайне
- создать пост у себя на стене. Этот пост должен содержать названия и количество участников пяти групп, в которых вы состоите.
- среди друзей ваших друзей найти аккаунт, у которого максимальное количество друзей
- взять самый последний пост в вашей новостной ленте. Вывести 10 имен друзей автора поста или 10 имен участников группы - автора поста.

Дополнительное задание

Создать поле ввода. В поле ввода вводится число, которое ограничивает общее количество аккаунтов, лайков, друзей, постов или групп, с которыми вам нужно работать.

Теория

Async/await пытается решить одну из главных головных болей языка со времен его появления: асинхронность. В большинстве случаев, для решения асинхронных задач мы полагались на коллбэки:

```
setTimeout(function() {console.log("This runs after 5 seconds");
}, 5000);
console.log("This runs first");
```

Всё это хорошо, но что если мы столкнемся с последовательностью?

```
doThingOne(function() {
      doThingTwo(function() {
            doThingThree(function() {
                  doThingFour(function() {
                        // Oh no
                  });
            });
      });
});
```

То, что вы видите выше иногда называется Pyramid of Doom и Callback Hell.

Промисы

Промисы это очень мудрый и хороший способ работы с асинхронным кодом.

Промис это объект, который представляет собой асинхронный таск, который должен завершиться. При использовании это выглядит как-то так:

```
function buyCoffee() {
      return new Promise((resolve, reject) => {
            asyncronouslyGetCoffee(function(coffee) {
                  resolve(coffee);
            });
      });
}
```

buyCoffee возвращает промис, который является процессом покупки кофе. Функция resolve указывает промису на то, что он выполнен. Он получает значение как аргумент, который будет доступен в промисе позже.

В самом экземпляре промиса есть два основных метода:

Then — запускает колбек, который вы передали, когда промис завершен.

Catch — запускает колбек, который вы передали, когда что-то идет не так, что вызывает reject вместо resolve. Reject вызывает как вручную, так и автоматически, если необработанное исключение появилось внутри кода промиса.

Важно: промисы, которые были выкинуты из-за исключения, поглотят это исключение. Это означает то, что если ваши промисы не связаны должным образом или нет вызова catch в каком-либо промисе из цепочки, то вы обнаружите, что ваш код просто втихую порушится, что может быть очень разочаровывающе, так что избегайте таких ситуаций любой ценой.

У промисов есть и другие очень интересные свойства, которые позволяют им быть связанными. Предположим, что у нас есть другие функции, которые отдают промис. Мы могли бы сделать так:

```
buyCoffee()
.then(function() {
      return drinkCoffee();
})
.then(function() {
      return doWork();
})
.then(function() {
      return getTired();
})
.then(function() {
      return goToSleep();
})
.then(function() {
      return wakeUp();
});
```

В этом случае использование колбеков было бы ужасным для поддержания кода и его чистоты.

Если вы не использовали промисы, то код выше может выглядеть непонятным, так как промисы, которые отдают промисы в своем методе then, вернут промис, который решается только когда возвращенный промис сам решается. И они сделают это со значением возвращенного промиса.

Асинхронные функции

Async функции - это функции, которые возвращают промисы. Async функции объявляются добавлением слова async, например

```
async function doAsyncStuff() { ...code }
```

Ваш код может встать на паузу в ожидании Async функции с await.

Await возвращает то, что асинхронная функция отдаёт при завершении.

Await может быть использовано только внутри async функции.

Если асинхронная функция выдает исключение, то оно поднимется к родительской функции, как в обычном JavaScript и может быть перехвачено с try/catch. Как и в промисах, исключения будут проглочены, если они не будут перехвачены где-нибудь в цепочке кода. Это говорит о том, что вы всегда должны использовать try/catch, всякий раз когда запускается цепочка вызовов Async функций. Хорошей практикой является включение хотя бы одного try/catch в каждую цепочку, если только в игнорировании этого совета нет абсолютной необходимости. Это даст одно единственное место для работы с ошибками во время работы async и сподвигнет вас правильно связать ваши запросы async функций.

Простая async функция:

```
// Async/Await version
async function helloAsync() {
      return "hello";
}


// Promises version
function helloAsync() {
      return new Promise(function (resolve) {
            resolve("hello");
      });
}
```

Обычная коллбэк-функция:

```
$.getJSON({
             url: "https://api.vk.com/method/users.get?.....",
             jsonp: "callback",
             dataType: "jsonp"
          }).done(function( data ) {

                здесь описание того, что делать с ответом

            });
```

Async функция, возвращающая промис:

```
async function getVkUser(id) {
      return $.getJSON({
            url:"https://api.vk.com/method/users.get?.....",
            jsonp: "callback",
            dataType: "jsonp"
      }).promise();
}

// здесь описание того, что делать с ответом
async function printUser(id) {
      var user = (await getVkUser(id)).response;
      console.log(user);
}
```