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.

Тест: Определи свой уровень знаний по основам PHP

Приветствую вас, коллеги! Предлагаю вам проверить Ваш уровень знаний по основам PHP! Что такое PHP, для чего он используется и так далее. Этот тест не был направлен на знание языка PHP.

Вам нужно ответить всего на 7 вопросов. Жду ваших комментариев. Пройдя весь тест вы сможете увидеть свои пробелы, если их нет, то здорово! Ну, а если есть, вы молниеносно можете исправить положение!

Если Вы прошли, то уже добились результатов. Почему? Так или иначе Вы получили знания и закрепили их. Ну, а для того чтобы пройти тест на все 100% прочитайте статью там вы найдете все ответы.

Если ваш результат не ахти какой, то не расстраивайтесь, прочитайте статью и вы найдете правильные ответы и затем заново пройдите тест. Я уверен, что у вас получиться улучшить результат!

Успехов!

Начать тест

Таблица лучших: Основы PHP

максимум из 7 баллов

Место	Имя	Записано	Баллы	Результат
Таблица загружается				
Нет данных				

Настройка Multiseat на Ubuntu 14.04

Доброго времени суток, уважаемые читатели нашего блога! Недавно столкнулся с вопросом **установки нескольких рабочих мест на один компьютер** в операционной системе Ubuntu 14.04. У меня получилось настроить режим Multiseat, как я и хотел, три рабочих места с одним системным блоком компьютера.

Конечно, эта заметка никак не связана с программированием в 1С, PHP или Delphi, но все же чуть-чуть программирования вы увидите в конце этой статьи. Думаю, что мои записи будут полезны тем, кто хочет настроить на своем компьютере режим Multiseat. В прошлой заметке я делился мыслями как настроить XAMPP на Mac OS X, а сегодня разберемся с Multiseat. Поехали!

Содержание

Подбор видеокарт

Для начала нужно определится с видеокартами. Желательно, чтобы они были одного производителя, например Nvidia. У меня было их три: GeForce 7300 GS, GeForce 210, GeForce GT 220. На материнской плате было три PCI-слота, в них я и поместил эти видеокарты.

Пробовал другую материнскую плату с двумя PCI-слотами и одной встроенной видеокартой, но биос не позволил сделать так, чтобы совместно работали и встроенная и внешняя видеокарта, поэтому остановился на варианте с тремя внешними видеокартами.

Чтобы убедится, что все три видеокарты видны системе можно воспользоваться следующей командой консоли Linux:

```
$ lspci |grep VGA
```

Определение значений видеокарт и USB-портов среди устройств, зарегистрированных в Ubuntu

Чтобы правильно настроить Multiseat в Ubuntu 14.04 нужно знать уникальный идентификатор каждого устройства или порта. Для этого нужно воспользоваться командой консоли:

```
$ udevadm info --export-db > /home/user/udevadm.txt
```

Эта команда создаст файл, по указанному пути и запишет информацию об устройствах. Открыв этот файл в текстовом редакторе, можно с помощью поиска найти ваши порты и устройства. Файл получается большой, поэтому в диалоге поиска можно ввести, например: «GeForce» для поиска видекарт или «usb7» для поиска идентификатора USB-порт. Привожу часть информации из моего файла:

```
...
E: DEVPATH=/devices/pci0000:00/0000:00:1c.0/0000:05:00.0
E: DRIVER=nouveau
E: ID_MODEL_FROM_DATABASE=G72 [GeForce 7300 GS]
E: ID_PCI_CLASS_FROM_DATABASE=Display controller
E: ID_PCI_INTERFACE_FROM_DATABASE=VGA controller
E: ID_PCI_SUBCLASS_FROM_DATABASE=VGA compatible controller
E: ID_VENDOR_FROM_DATABASE=NVIDIA Corporation
...
N: bus/usb/007/001
E: BUSNUM=007
E: DEVNAME=/dev/bus/usb/007/001
E: DEVNUM=001
E: DEVPATH=/devices/pci0000:00/0000:00:1d.1/usb7
E: DEVTYPE=usb_device
...
```

Значения DEVPATH, которые выделены жирным шрифтом — «/devices/pci0000:00/0000:00:1c.0/0000:05:00.0» и «/devices/pci0000:00/0000:00:1d.1/usb7», как раз, то, что нам нужно. Они нам понадобятся на следующем шаге.

Чтобы посмотреть подключенные USB устройства воспользуйтесь командой:

```
$ lsusb
```

С помощью этой команды можно увидеть какому физическому порту соответствует идентификатор из файла «udevadm.txt». Чтобы понять это, вытащите одно из устройств, (например, мышку или клавиатуру) из USB-порта и введите команду вновь, затем сверьте изменения, которые произошли после повторного ввода

команды «lsusb».

Настройка правил для режима Multiseat

Давайте настроим файл правил для Multiseat, воспользовавшись информацией из предыдущего шага. Для этого создадим файл:

```
$ sudo touch /etc/udev/rules.d/99-multiseat.rules
```

В нем запишем следующую информацию:

```
# ********************** SEAT-1 **********************

# назначение USB порта мыши для seat-1
TAG==»seat»,
DEVPATH==»/devices/pci0000:00/0000:00:1a.2/usb5*»,
ENV{ID_SEAT}=»seat-1″, TAG+=»seat-1″

# назначение USB порта клавиатуры для seat-1
TAG==»seat»,
DEVPATH==»/devices/pci0000:00/0000:00:1d.0/usb6*»,
ENV{ID_SEAT}=»seat-1″, TAG+=»seat-1″

# назначение видеокарты GeForce 210 для seat-1
TAG==»seat»,
DEVPATH==»/devices/pci0000:00/0000:00:06.0/0000:02:00.0*»,
ENV{ID_SEAT}=»seat-1″, TAG+=»seat-1″

# ********************** SEAT-2 **********************

# назначение USB порта мыши для seat-2
TAG==»seat»,
DEVPATH==»/devices/pci0000:00/0000:00:1d.1/usb7*»,
ENV{ID_SEAT}=»seat-1″, TAG+=»seat-1″

# назначение USB порта клавиатуры для seat-2
TAG==»seat»,
```

```
DEVPATH==»/devices/pci0000:00/0000:00:1d.2/usb8*»,
ENV{ID_SEAT}=»seat-1″, TAG+=»seat-2″

# назначение видеокарты GeForce 7300 GS для seat-1
TAG==»seat»,
DEVPATH==»/devices/pci0000:00/0000:00:1c.0/0000:05:00.0*»,
ENV{ID_SEAT}=»seat-2″, TAG+=»seat-2″
```

Как вы могли догадаться, значения, которые выделены жирным шрифтом взяты из файла «udevadm.txt» (значения без звездочки). Мы настраиваем три рабочих места, а в файле правил нужно прописать только два. Почему? Дело в том, что одна из видеокарт и не прописанные в файле правил USB-порты будут автоматически присвоены рабочему месту с именем «seat0» (без тире в названии, таковы правила именования рабочих мест). После выполнения всех шагов нужно перезагрузить компьютер, чтобы изменения вступили в силу. После перезагрузки в консоли можно увидеть список подключенных рабочих мест, выполнив команду:

```
$ loginctl list-seats
```

Еще нужно проверить файл «/etc/lightdm/lightdm.conf» в нем должна быть секция с соответствующим идентификатором и значением:

```
[LightDM]
logind-load-seats=true
```

Автозапуск Firefox

Для полного решения нашей задачи необходимо было запускать Firefox для каждого рабочего места. В автостарте был прописан запуск Firefox, но при запуске Firefox запускался только на одном рабочем месте, а на других выводилась ошибка:

```
Firefox is already running, but is not responding.
To open a new window, you must first close the existing
Firefox process,
```

or restart your system.

Дело в том, что рабочие места инициализируются от имени одного пользователя, поэтому выпадала такая ошибка. Решилась эта проблема созданием нескольких профилей для Firefox:

1. Запустим менеджера профилей командой консоли:

   ```
   $ firefox -P
   ```

2. В окне менеджера профилей создадим три профиля FullScreenStorage0, FullScreenStorage1, FullScreenStorage2.
3. Создадим исполняемый файл «firefox.multiseat»:

   ```
   $ sudo touch /home/user/firefox.multiseat
   ```

4. В нем запишем запуск для каждого рабочего места:

   ```
   #!/bin/bash
   firefox -P FullScreenStorage`echo $DISPLAY | cut -f2 -d\:`
   ```

5. Пропишем запуск файла «firefox.multiseat» в автозагрузке.

Что такое пространство имен –

namespaces – в PHP?

Доброго времени суток, уважаемые читатели нашего блога! В прошлой статье мы начали тему, связанную с объектно-ориентированным программированием в PHP. Сегодня мы копнем поглубже и рассмотрим **пространство имен – namespaces** – в языке PHP, используя простые примеры программирования. И первая хорошая новость – Namespaces прост в изучении. Начнем!
Создадим класс в PHP

```
<?php
class Foo
{
  public function doAwesomeFooThings ()
  {
   // здесь должен быть ваш код
  }
}
?>
```

Мы заполняем PHP 5.2 класс, который делает много важных вещей. Например, скажем «Привет» читателям:

```
<?php
  class Foo
  {
  public function doAwesomeFooThings ()
    {
     echo “Привет, читатели!”;
  }
  }
  ?>
```

Как использовать класс *Foo*? Использование *Foo* простое, напишем:

```
<?php
  require ‘foo.php’;
  $foo = new Foo();
```

```
  ?>
```

Для того, чтобы идти в ногу со временем, давайте используем новое понятие, которое появилось в PHP 5.3 – **пространство имен Namespace**. Пространство имен (namespace) используется подобно вложенным папкам, добавим это пространство, расположенное в *Acme\Tools*.

```
<?php
// это файл foo.php
namespace Acme\Tools;
class Foo
{
  public function doAwesomeFooThings ()
  {
    echo “Привет, читатели”;
  }
}
?>
```

Для использования *Foo* нужно назвать его по-новому. Это будет ссылка на файл, представляющая собой полный путь.

```
<?php
require ‘foo.php’;
$foo = new \Acme\Tools\Foo();
?>
```

Вот и все. Добавляя пространство имен в класс – это словно организация файлов из одной директории в несколько поддиректорий. Для ссылки на класс используйте его имя, начинающееся со слеша «\». Большое имя воспринимается тяжело, поэтому давайте добавим короткое. Для этого дадим классу *\Acme\Tools\Foo()* псевдоним.

```
<?php
require ‘foo.php’;
use \Acme\Tools\Foo as SomeFooClass;
$foo = new SomeFooClass();
?>
```

Можно как-нибудь назвать класс, либо использовать имя по

умолчанию *Foo*.

```
<?php
require ‘foo.php’;
use \Acme\Tools\Foo;
$foo = new Foo();
?>
```

Хорошо. Как насчет старых классов PHP без имени пространства? Для этого перейдем к подходящему для нашего случая классу *datetime*. И возьмем несколько новых фишек от PHP 5.3. Создание нового объекта даты *datetime* выглядит аналогично:

```
<?php
require ‘foo.php’;
use \Acme\Tools\Foo;
$foo = new Foo();
$dt = new DateTime();
?>
```

Для обычного файла это все еще работает. А вот в файле с объявлением пространства namespace PHP считает, что речь идет о файле в директории *Acme**Tools*:

```
<?php
namespace Acme\Tools;
class Foo
{
  public function doAwesomeFooThings ()
  {
     echo “Hi listeners”;
     // здесь поиск будет вестись в директории \Acme\Tools
     $dt = new DateTime();
  }
}
?>
```

Или вы можете сослаться на класс в глобальном пространстве, используя полное имя *Datetime*:

```
<?php
```

```
namespace Acme\Tools;
class Foo
{
  public function doAwesomeFooThings ()
  {
     echo “Hi listeners”;
     // здесь поиск будет вестись в глобальном пространстве
     $dt = new \DateTime();
  }
}
?>
```

Предыдущий фрагмент кода показывает, что мы обращаемся к глобальному классу *DateTime*

Чтобы обратиться глобальному классу *DateTime* можно это сделать с помощью *use*:

```
<?php
namespace Acme\Tools;
use \DateTime;
class Foo
{
  public function doAwesomeFooThings ()
    {
     echo “Hi listeners”;
     // здесь поиск будет вестись в глобальном пространстве
     $dt = new DateTime();
  }
}
?>
```

Использование *use* выглядит глупо, но оно говорит PHP, что когда вы вызываете DateTime, вы имеете в виду класс без имени, указывающий на время. Если убрать слэш перед DateTime, то все будет работать точно так же.

```
<?php
namespace Acme\Tools;
use DateTime;
class Foo
```

```
{
  public function doAwesomeFooThings ()
  {
     echo “Hi listeners”;
     // здесь поиск будет вестись в глобальном пространстве
     $dt = new DateTime();
  }
}
?>
```

И вы даже не заметите этого. ОК. Пока!

P.S.

Этот блог читают уже много людей
- читай и ТЫ!
Да, Я тоже хочу читать статьи!

Как удалить спам на форуме phpBB

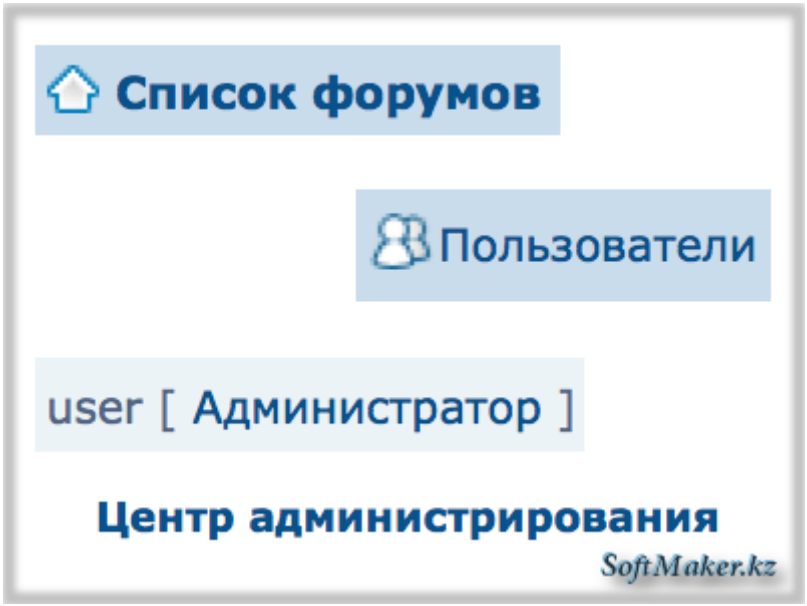
Доброго времени суток, уважаемые читатели нашего блога! Как показывает практика, многие люди устанавливают и активно используют популярный форум phpBB. Для них очень важно знать, как с ним правильно работать. Например, их интересует как изменить шапку форума phpBB.А сегодня речь пойдет об удалении всех нежелательных сообщений с вашего форума phpBB с помощью панели администрирования. Давайте посмотрим какие шаги необходимо предпринять, чтобы правильно это сделать.

Содержание

Панель администрирования phpBB

Удаление спама на форуме phpBB – это рутинная работа владельца онлайн-форума. Панель форума phpBB обеспечивает **возможность удалять все сообщения от конкретного пользователя**. Этот метод удобен, когда спамер может разместить десятки сообщений на вашем форуме.

Войдите в панель администрирования phpBB. Панель администрирования доступна только для учетной записи администратора, которая является главной учетной записью для установки системы. Также вы можете войти в эту панель, если ваша учетная запись входит в группу администраторов. Ссылка для входа в панель администрирования находится в нижней части вашего форума и называется «Центр администрирования».



Нажмите «Управление пользователями» в левой части панели администрирования в закладке ОБЩИЕ. Введите имя пользователя, которого вы хотите **удалить как спамера**, и нажмите «Ввод».

Кроме того, можно нажать ссылку «Администратор» в профиле пользователя. Эта ссылка доступна только для модераторов и администраторов. Ссылку можно увидеть, если перейти в «Список форумов» по ссылке «Пользователи» (находится в правой части форума) и выбрать нужного пользователя.

Обзор информации о пользователе форума

Если вы не уверены в том, что пользователь является спамером, нажмите кнопку «Whois» под IP-адресом. Открытое окно сообщит вам, в какой стране зарегистрирован пользователь. Если страна RU, UA или SE, возможно, пользователь использовал прокси-сервер. Чтобы проверить, сообщалось ли о пользователе как о спамере, скопируйте и вставьте его адрес электронной почты в поисковую систему. Обычно адреса эл. почты спамеров находятся на сайтах, которые сообщают о таких адресах, так что вы можете быстро узнать адреса спамеров.

Прокрутите вниз до раздела «Основные инструменты». Выберите «Запретить e-mail адрес» и нажмите кнопку «Отправить». Это позволит прекратить рассылку и перерегистрацию с данного адреса электронной почты. Повторите этот процесс за исключением выбора «Запретить IP-адрес» в раскрывающемся списке. Этот пункт используется, чтобы блокировать спамера по IP-адресу.

И наконец, прокрутите вниз до раздела «Удалить пользователя». Выберите «Удалить сообщения» из раскрывающегося списка и нажмите кнопку «Отправить». Данный процесс устраняет учетную запись и все связанные с ней сообщения, так что вам не нужно удалять их по одному.

Вы можете использовать этот процесс для каждой учетной записи, в которой размещается спам на вашем форуме phpBB, чтобы свести к минимуму время, необходимое для **удаления больших объемов**

спама. Этот процесс удаляет десятки сообщений, которые спамер может автоматически размещать на Вашем форуме. Поскольку спамер использует программное обеспечение для размещения сообщений на форуме, **вы можете удалить все сообщения** и не пропустить ни одного в более ранних темах на форуме. Отсутствие спамеров на вашем форуме позволит вам избежать того, чтобы поисковые системы обозначали ваш сайт как спамерский, а этот метод поможет вам навести порядок за нескольких минут.