



shanhaichik 26 апреля 2016 в 12:55 Разработка

Async/Await в javascript. Взгляд со стороны

Программирование, JavaScript

Из песочницы



В последнее время все больше моих друзей, коллег и людей из сообщества говорят про работу с асинхронными функциями и в частности использование `async/await` на своих проектах. Я решил разобраться для себя, что это за зверь и стоит ли его использоваться при разработ боевых проектов.

Первое что хочется развеять, это распространенное заблуждение о том, что `async/await` — это фича ES7.

По моему мнению, использование терминов ES6 и ES7 само по себе не очень верное и может ввести разработчиков в заблуждение. После удачного релиза спецификации ES2015, называемой ES6, у многих людей сложилось ошибочное мнение, что все в нее не вошло и заполни через `babel` — это фичи ES7. Это не так. Вот список того что появится с релизом спецификации ES2016. Как видите он не такой большой и `async/await` в нем никак не значится.

Я хочу, чтобы мы говорили правильно. И говоря о той, или иной фиче, ссылались на конкретную спецификацию в рамках которой она описана, а не мифические **ES6, ES7 ... ESN**.

Двигаемся дальше. Так что же такое `async/await` простыми словами?

Говоря общедоступным языком **`async/await`—это `Promise`**.

Когда вы объявляете функцию как асинхронную, через волшебное слово **`async`**, вы говорите, что данная функция возвращает `Promise`. Как вещь которую вы ожидаете внутри этой функции, используя волшебное слово **`await`**, то же возвращает `Promise`.

Это очень важный момент для понимания, как такие функции работают и чего ожидать при работе с ними.

Давайте посмотрим, как же выглядит наш единорог и разберемся как он работает.

Вот простой пример асинхронного `Redux` экшена для выхода из кабинета:

```
export function logout(router) {
  return async (dispatch) => {
    try {
      const {data: {success, message}} = await axios.get('/logout');
```

```
(success)
  ? dispatch({ type: LOGOUT_SUCCESS })
  : dispatch({ type: LOGOUT_FAILURE, message });

} catch (e) {
  dispatch({ type: LOGOUT_FAILURE, e.data.message });
}
};
}
```

А теперь идем от общего к частному

После прочтения ряда статей и самостоятельно поигравшись, я составил для себя небольшой бриф, отвечающий на основные вопросы, с небольшими примерами.

Что нужно сделать чтобы начать работу?

Если не использовать никакой системы сборки, то достаточно установить babel и babel-runtime.

```
babel test.js -o test-compile.js --optional runtime --experimental
```

В остальных случаях, лучше смотреть настройки исходя из системы сборки и версии babel. Это очень важно, так как настройки в версии babel6 сильно различаются.

Как создается асинхронная функция?

```
async function unicorn() {
  let rainbow = await getRainbow();
  return rainbow.data.colors
}
```

Создание асинхронной функции состоит из двух основных частей:

1. Использования слова `async` перед объявлением функции.

Как мы видим из примера с `logout()`, это так же работает при использовании стрелочных функций. Еще это работает для функций классов и статических функций. В последнем случае `async` пишется после `static`.

2. В теле самой функции мы должны использовать слово `await`.

Использование слова `await` сигнализирует о том, что бы основной код ждал и не возвращал ответ, пока не выполниться какое-то действие. Оно просто обрабатывает Promise для нас и ждет пока он вернет `resolve` или `reject`. Таким образом, создается впечатление, что код выполняется синхронно.

** Для работы с `await` функция должна быть асинхронной и объявлена с помощью ключевого слова `async`. В противном случае это просто не будет работать.*

Как работает `await` и какую функцию выполняет?

Как говорилось ранее, `await` ожидает любой Promise. Проводя аналогию с работой объекта Promise, можно сказать, что `await` выполняет такую же функцию что и его метод `.then()`. Единственная существенная разница в том, что она не требует никаких callback функций для получения и обработки результата. Собственно за счет этого и создается впечатление что код выполняется синхронно.

Хорошо, если `await` это аналог `.then()` у Promise, как же мне тогда поймать и обработать исключения?

```
async function unicorn() {
  try {
    let rainbow = await getRainbow();
  }
}
```

```
    return rainbow.data.colors;
  } catch(e) {
    return {
      message: e.data.message,
      somaText: 'Текст о не легкой жизни единорогов'
    }
  }
}
```

Так как код в синхронном стиле, по этой причине мы можем использовать старый добрый try/catch для решения подобных задач.

Дополнительно хочется акцентировать на этом внимание.



Использование try/catch это единственный способ поймать и обработать ошибку. Если по каким-то причинам вы решите его не использовать просто забыли, это может привести к отсутствию возможности обработки, а так же потере вовсе.

В какой момент происходит выполнение кода следующего за await?

```
async function unicorn() {
  let _colors = [];
  let rainbow = await getRainbow();

  if(rainbow.data.colors.length) {
    _colors = rainbow.colors.map((color) => color.toUpperCase());
  }

  return _colors;
}
```

Код следующий после await, продолжает свое выполнение только тогда когда функция используемая с await вернет **resolve** или **reject**.

Что если функция используемая с await не возвращает Promise?

Если функция используемая с await не возвращает Promise, а мы уже знаем, что await его ожидает, то выполнение кода продолжится так как бы мы не использовали await вообще.

Что если объявить функцию асинхронной, но не использовать await?

```
async function unicorn() {
  let rainbow = getRainbow();
  return rainbow;
}
```

В таком случае, на выходе мы получим просто ссылку на Promise функции getRainbow().

Что будет если я напишу несколько функций использующих await подряд?

```
async function unicorn() {
  let rainbow = await getRainbow();
  let food = await getFood();
  return {rainbow, food}
}
```

Такой код будет выполняться последовательно. Сначала отработает getRainbow(), после того как она вернет resolve или reject начнет работу getFood(). Один вызов, один результат.

А если мне нужно одновременно получить результат от нескольких вызовов?

```
async function unicorn() {
  let [rainbow, food] = await Promise.all([getRainbow(), getFood()]);
  return {rainbow, food}
}
```

Так как мы уже разобрались, что мы имеем дело с Promise. Следовательно можно использовать метод .all() объекта Promise для решения такого рода задач.

Дополнительно хочу заметить, что конструкция await * arrayOfPromises больше не актуальна и удалена из спецификации. При попытке ее использовать вы получите сообщение с любовью о том, что лучше использовать Promise.all().

Пример сообщения:

```
await* has been removed from the async functions proposal. Use Promise.all()
```

Обновил информацию по конструкции await*. Спасибо @xGromMx и @degrov.

Что еще хорошо бы знать для успешной работы?

```
async function getAllUnicorns(names) {
  return await Promise.all(names.map(async function(name) {
    var unicorn = await getUnicorn(name);
    return unicorn;
  }));
}
```

Надо помнить, что если ты начинаешь использовать async/await в своем проекте, нужно быть готовым к тому, что почти весь твой стек должен будет быть асинхронным. А это может добавить не мало проблем и неудобств.

Вроде бы все.

С основными теоретическими аспектами мы разобрались, если что-то осталось непонятным или у Вас есть что дополнить, жду в комментариях.

А для меня пришло время собрать всю информацию воедино, и решить что же мне делать с нашим асинхронным единорогом. Брать с собой в путешествие или оставить дома для своих Pet проектов.

Выводы:

На первый взгляд асинхронные функции вызывают положительные эмоции. Такая *няшная штучка* делающая твой код синхронно подобна

делает его более лаконичным и читабельным.

Но это на первый взгляд.

Да, код лаконичен и читабелен. Да, выглядит симпатично. Но как только ты пытаешься писать что-то сложнее чем обычное API или сложные взаимосвязанные скрипты, например, очередь задач для работы с базой данных, сразу сталкиваешься с проблемой асинхронного стека.

Почти все вечно зеленые браузеры, из коробки, на 93%-98% поддерживают фиши ES2015 (таблица). Для меня это означает, что начиная нс проект, исходя из требований и стека, я уже задумаюсь об необходимости babel на проекте.

Но, если я решу использовать async/await, я буду обязан использовать babel. И не могу сказать что это добавит красоты в мой код. Ведь официально async/await нет, и не известно будет ли вообще. И это для меня большой минус.

Так же мне очень не нравится тот факт, что если я забыл применить await или просто не удачный копипаст, вместо автоматического вылета ошибку, я ничего не получу, кроме ссылки на Promise. Это может быть черевато последствиями, особенно когда большой проект с несколькими разработчиками.

И последнее.

Большинство задач с использованием async/await прекрасно решаются с помощью [генераторов](#).

Во-первых, у них и поддержка лучше.

Во-вторых, работа генераторов будет более естественна и предсказуема.

В-третьих сам babel приводит такой код к генераторам при особых настройках [пример1](#), [пример2](#).

Поддержка в NodeJS

Async/await уже экспериментально попал в V8. Это значит что с версии nodejs 7 можно с ним поиграться и поработать прямо из коробки. Как это сделать:

```
NVM_NODEJS_ORG_MIRROR=https://nodejs.org/download/nightly
nvm install 7
nvm use 7

node --harmony-async-await app.js
```

Итого

Отвечаю себе на вопрос заданный в самом начале:

Если я и буду использовать асинхронные функции, то только на своих Pet проектах и не очень больших рабочих, в основном для написани: По крайней мере пока все не стандартизируется и не будет под флагом -экспериментально.

Например мне понравилось использовать их в экшенах для Redux. Выглядит все красиво и гармонично.

Этот материал я писал в первую очередь для себя, чтобы разобраться с интересующим меня вопросом. Если данный материал будет еще то полезен, я буду очень рад.

Также, в следующей статье, я бы хотел подробно сравнить разные подходы к реализации асинхронности (колбэки, промисы, генераторы, а Чтобы это было понятно не только гуру, но и людям только начинающим свой путь в javascript.

Всем спасибо за внимание. Удачи!

Метки: javascript, async

↑ +31 ↓ 223 👁 80,5k 💬 53



18,0

Карма

5,6

Рейтинг

11

Подписчики

Александр Духовняк @shanhaichik

Frontend

Поделиться публикацией

ПОХОЖИЕ ПУБЛИКАЦИИ

3 июня 2016 в 13:04

Событийная модель на основе async и await

↑ +22 👁 18,8k 📖 142 💬 18

18 сентября 2015 в 16:43

Python 3.5; async/await

↑ +27 👁 62,8k 📖 229 💬 26










2 июля 2015 в 13:40

Недопонимание про async/await и многопоточность в C#

↑ +21 👁 47,2k 📖 252 💬 16

Реклама

Комментарии 53

- 
xGromMx 26.04.16 в 13:47
🔖 📖
↑
- Синтаксиса await* не будет так как они говорят что есть уже Promise.all. Это не одно и тоже что в yield и yiled*
- 
shanhaichik 26.04.16 в 14:29
🔖 📖 📄 🔄
↑
- Привет, @xGromMx. А можно ознакомиться где это написано или сказано? Заранее спасибо.
- НЛО прилетело и опубликовало эту надпись здесь
- 
shanhaichik 26.04.16 в 14:48
🔖 📖 📄 🔄
↑
- Спасибо, @degrov. Ознакомлюсь и внесу соответствующие корректировки.
- 
xGromMx 26.04.16 в 15:26
🔖 📖 📄 🔄
↑
- Глянь это ишью <https://github.com/tc39/ecmascript-asyncawait/issues/85>
- 
arusakov 26.04.16 в 15:17
🔖 📖
↑
- А как сейчас можно отказаться от Babel, если нужна минификация для продакшена? Я недавно думал отключить Babel на одном проекте, но потом что Uglifyjs не поддерживает нормально ES6, и оставил все, как есть.
- 
vintage 26.04.16 в 15:45
🔖 📖 📄 🔄
↑
- Попробуйте меньше копипастить и минификация вам не понадобится :-)
- 
shanhaichik 26.04.16 в 15:56
🔖 📖 📄 🔄
↑
- @vintage, вы злодей :)
- 
shanhaichik 26.04.16 в 15:55
🔖 📖 📄 🔄
↑
- @arusakov, Я понимаю что с места этот барьер будет тяжело взять. Я к этому не призываю. И сам недавно столкнулся с похожей проблемой. Но на мой взгляд это не повод опускать руки. Скорость с которой браузеры вводят поддержку новых фич, не говоря уже о ES2015, заставит разработчиков, подобного рода инструментов, подтянуть свои продукты под современные реалии. Иначе придет кто-то новый, более адаптирова Например у **Uglifyjs** есть экспериментальная ветка **harmony**, как раз нацеленная на поддержку ES6.
- В данный момент, я решил пробовать разработку без babel на небольших внутренних проектах. Где можно *принебречь* некоторыми вещами, напр Uglifyjs.
- 
xGromMx 26.04.16 в 16:59
🔖 📖 📄 🔄
↑
- Частично с минификацией справляется closure compiler



Staltec 28.08.17 в 15:15



При чём тут минификация и Babel?



radist2s 26.04.16 в 16:20



Исходя из того, что компилирует babel, await перестает быть асинхронным и поток вполне себе блокируется. Так что это то еще зло, если не использовать в отдельном воркере.



f0rk 26.04.16 в 17:38



Где вы там блокировку увидели? while(1) в regeneratorRuntime используется для того чтобы перезапускать генераторы с циклами, ничего он не блокирует.



staticlab 26.04.16 в 17:54



Там вся суть в функции _asyncToGenerator(), в которую передается генератор.



Yozi 26.04.16 в 17:54



Эм, пруфы, пожалуйста. Babel переводит async/await либо в генераторы, либо использует регенератор.



radist2s 26.04.16 в 17:56



Да-да, теперь понял, ошибался.



Scf 26.04.16 в 16:53



Для меня это выглядит так, как будто async/await бесполезен в реальных проектах. Допустим, у нас есть 3 функции:

```
function authenticateUser(login, password) {} //возвращает Promise<userId> при успехе
function getUserDetails(userId) {}
function getUserAvatar(userId) {}
```

И я хочу запустить authenticateUser, а по его завершению — getUserDetails и getUserAvatar параллельно.

Через `then` это делается элементарно, включая обработку ошибок — достаточно проверить результирующий промис:

```
let detailsAndAvatar = authenticateUser(login, password).then(function(userId) {
  return Promise.all([getUserDetails(userId), getUserAvatar(userId)]);
});
```

Как это будет выглядеть на async/await?



xGromMx 26.04.16 в 17:02



```
async f() {
  const userId = await authenticateUser(login, password);
  return await Promise.all([getUserDetails(userId), getUserAvatar(userId)]);
}
```



Scf 26.04.16 в 17:50



Красота) А можно пример с тремя уровнями вложенности? Я никак не могу сообразить. Т.е.

```
f    -> f1
      |-> f11
      |-> f12
    -> f2
      |-> f21
      |-> f22
```



f0rk 26.04.16 в 18:07



Диаграммка не очень понятная, приведите пример с промисами.



ReklatsMasters 19.10.16 в 15:58



Поскольку async возвращает промис, мы спокойно можем сделать await к нему.

```
async function f1() {
  return await Promise.all([ f1(), f2() ])
}

async function f2() {
  await f21()
  return await f22()
}

async function f() {
  await f1()
  await f2()
}
```



keksmen 26.04.16 в 17:54

Простите за снобство, но это плохой пример. Если `async f()` возвращает обещание, то зачем оператор `await` после `return`? Не достаточно ли будет "вернуть" обещание? Механизм *Promise*ов ждёт обещания любой глубины. Даже если обещание `resolve`'ит другое обещание.



xGromMx 26.04.16 в 18:19

это и есть проблемы промисов у них `map` и `flatMap` ведут себя одинаково хотя было бы правильно с точки зрения функтора и монад так

```
const pf = Promise.of(42).map(v => v + 10);
const pm = Promise.of(42).flatMap(v => Promise.of(v + 10).delay(1000)) // псевдокод с delay
```



xGromMx 27.04.16 в 00:32

За что минусы?



vitalets 27.04.16 в 11:21

`Promise.of` нет в спецификации



faiwer 27.04.16 в 11:22

А @xGromMx и не утверждал обратного. Он просто привёл пример, того как, по его мнению, было бы правильнее. Код напоминает `sc`



f0rk 26.04.16 в 17:03

опоздал :)



xGromMx 26.04.16 в 17:03

где обертка в `async`?)



xGromMx 26.04.16 в 17:03

Грубо говоря `async/await` есть до монада



vintage 27.04.16 в 11:28

На программах (node-fibers) это будет выглядеть так:

```
let userId = authenticateUser( login , password ).wait()
let detailsFuture = getUserDetails( userId )
let avatarFuture = getUserAvatar( userId )
let detailsAndAvatar = [ detailsFuture.wait() , avatarFuture.wait() ]
```



vintage 27.04.16 в 11:33

А с моим велосипедом, вообще вот так:

```
let userId = authenticateUser( login , password )
let details = getUserDetails( userId )
let avatar = getUserAvatar( userId )
let detailsAndAvatar = [ details , avatar ]
```




faiwer 27.04.16 в 12:10



Вы какую-то слишком общую ссылку привели. Лучше сразу на примеры с велосипедом. По какому принципу это работает? В каком контекст Нужно ли оборачивать эти методы чем-нибудь? Нужно ли на вершине стека какой-нибудь Fiber(context) запускать?

А самое интересное, чего я пока не понял до конца, это в чём различия подхода волокон и async-await? Я так понимаю, и там и там, не созд новых потоков, а только переключаются стеки состояний, что дешевле, чем, скажем, новые потоки, но тем не менее далеко не бесплатно. И нет? Вы не могли бы объяснить в двух словах?



vintage 27.04.16 в 12:40



Ок, вот более конкретная ссылка. Да, всё приложение нужно стартовать в волокне и всё.

async-await — это те же генераторы, которые не имеют стека. Просто машина состояний. А волокна — это такие себе легковесные потоки каждый со своим стеком. В случае генераторов мы дополнительно платим за каждый вызов функции. В случае волокон — платим лишь : переключение волокон.



xGromMx 27.04.16 в 13:17



Скинь ссылку на статью про волокна, я помню у тебя где-то было на github



vintage 27.04.16 в 13:26



Речь об этой статье? Там фактически не волокна, а эмуляция async-await через генераторы.



xGromMx 27.04.16 в 15:04



да о ней



Scf 03.10.16 в 20:48



Только сейчас могу сформулировать, чем же мне не нравится async/await. Да тем, что он провоцирует программистов писать последовательный и Асинхронный, но все операции выполняются по очереди, лишая смысла основную идею — параллельные программы.

С другой стороны, код на промисах "по умолчанию" полностью параллельный, и последовательность операций определяется неявно, через вычислительные зависимости, выраженные в then

пример: сложное приложение, делающее множество Ajax вызовов, на промисах будет по возможности делать эти вызовы параллельно. На async/await... для этого придется прикладывать усилия.



Yozi 03.10.16 в 23:13



Цепочка промисов через then выполняется *последовательно*, параллельным является Promise.all, например. И он прекрасно сочетается с async/await, например,:

```
await Promise.all([ajax1, ajax2])
```

Можете привести пример кода из "сложного Ajax приложения, делающего вызовы по возможности параллельно"? Я не понимаю за счёт чего о внезапно станет "по-умолчанию" параллельным на промисах?



vintage 03.10.16 в 23:27



Следите за руками:

```
require( 'jin' ).application( function( $ ){

    function get( ){
        return $.request.getSync( "http://example.org/" + Math.random() )
    }

    console.time( 'serial' )
    console.log( get().statusCode )
    console.log( get().statusCode )
    console.timeEnd( 'serial' )

    console.time( 'parallel' )
    var resp1= get()
    var resp2= get()
    console.log( resp1.statusCode )
    console.log( resp2.statusCode )
```

```
console.timeEnd( 'parallel' )

} )
```

```
200
200
serial: 2418ms
200
200
parallel: 1189ms
```



Scf 04.10.16 в 10:01



т.е. `get()` асинхронный, но поток выполнения блокируется при вызове любого метода на результате?
это же лочит браузер?



vintage 04.10.16 в 10:47



В браузере это и не работает. Только в ноде. Лочится не весь процесс, а отдельный стек вызовов, который запускает `application`. Вот т действительно по возможности распараллеливается работа.



GeraldIstar 27.04.16 в 07:55



То есть, если я все правильно понимаю — отловить ошибки в `async/await` функциях можно только через `try/catch`? Учитывая не очень хорошую производительность `try/catch`, мне кажется что это огромный минус таких функций.

Кстати, в `redux-saga` используются генераторы, но на первый взгляд работают точно таким же образом (я не говорю про то как там внутри реализовано)



shanhaichik 27.04.16 в 07:59



@GeraldIstar, да, все верно. Не очень хороша производительность, а так же V8 не оптимизирует функции, содержащие эту конструкцию



dannyzubarev 18.05.17 в 19:01



Ошибается. :)

Turbofan нынче отлично оптимизирует `try-catch-finally`, `for..of` и прочие конструкции, которые ранее считались убийцами оптимизации (см. <http://benediktmeurer.de/2017/04/03/v8-behind-the-scenes-march-edition/>)



shanhaichik 18.05.17 в 21:54



Все развивается ;)



faiwer 27.04.16 в 08:25



А разве Promise-ы в целом не работают через `try-catch`?

```
new Promise((r,j) => { throw 1; }).catch(err => console.log('error=', err))
// error= 1
```

Т.е. это скорее общая особенность работы `Promise`, нежели особенность `await`. Проигрываем в оптимизации, зато получаем проборос ошибок.



Shannon 27.04.16 в 10:02



Видимо нет, потому что в хrome:

```
function testFunction() {
  new Promise((r,j) => { throw 1; }).catch(err => console.log('error=', err))
}
```

```
Function is optimized
```

Добавление в любом месте `try-catch` (даже пустого) приводит к:

```
Function is not optimized
```



faiwer 27.04.16 в 11:20



У вас `testFunction` is optimized, а внутри исходников `promise`, где-нибудь есть метод, внутри которого стоит `try-catch`. И вот там будет is optimized. Полагаю, что `async`-методы в первое время будут, что с `try-catch`, что без него, не оптимизированными, а уже апосля руки дойдут до них. Во всяком случае, смотря на список убийц оптимизации у меня сложилось впечатление, что нужно писать код стоя на одной ноге на ципочках, чтобы не выпасть из оптимизации. Шаг в лево, шаг в право — приехали.

 Shannon 27.04.16 в 12:46 # 📌 🔄

Вполне может быть, в реализации промисов Bluebird от Petka Antonov, которые быстрее нативных, как раз используется изолированная функция `tryCatch` для `catch`, использование которой не мешает V8 оптимизировать весь остальной ваш код

Для теоретического `async-await` пока вариант явно отказаться либо от ловли ошибок, либо от оптимизаций (интересно как с этим дела в I. Может потому и не спешат внедрять

Кстати, в V8 уже оптимизировали некоторые моменты из числа убийц:

- функции, содержащие выражение `for-of`;
- функции, содержащие составной оператор присваивания `let`;
- функции, содержащие составной оператор присваивания `const`;
- функции, содержащие объектные литералы, которые, в свою очередь, содержат объявления `__proto__`, `get` или `set`.
- бесконечные циклы
- 5.1. Ключ не является локальной переменной (первый пример с `nonLocalKey1`)
- частично 5.2. Итерируемый объект не является «простым перечисляемым»
- может что-то еще, протестировал только эти

Function is optimized by TurboFan

И вполне возможно скоро можно будет не так активно задумываться над убийцами оптимизаций

 iKVANT 27.04.16 в 15:11 # 📌

еще есть асинхронные генераторы. Пример синтаксиса:

```
async function* myFunction() {
  await yield new Promise((resolve) => {});
}
```

вот тут подробнее <https://www.youtube.com/watch?v=DqMFX91ToLw>

✱ rock 27.04.16 в 15:56 # 📌 🔄

Ваша информация сильно устарела. На текущий момент, асинхронная итерация и `Observable` разделены.

 alz72 04.10.16 в 10:57 # 📌

Спасибо — очень хорошо что объяснили про «Promise», буду курить в эту сторону.

 VladimirDeminenko 18.05.17 в 15:03 # 📌

"... Использование `try/catch` это **единственный способ** поймать и обработать ошибку..."

Но, если `async function` возвращает `Promise`, тогда, наверное, можно и так:

```
async function unicorn() {
  let rainbow = await getRainbow();
  return rainbow.data.colors;
}

unicorn()
  .catch(function(error) {
    ...
  });
```

Только полноправные пользователи могут оставлять комментарии. [Войдите, пожалуйста.](#)

ИНТЕРЕСНЫЕ ПУБЛИКАЦИИ

Необразованная молодёжь: попытка подытожить и немного личного

+9 2,1k 14 13

Как работает буфер обмена в Windows

+6 2k 30 3

Монады для Go-программистов

+14 2k 27 6

HyperX на Игромире-2017: часть 1 — SSD и DRAM GT

+14 1k 1 1

Семантическая разметка: LaTeX, DocBook или ???

+8 1,1k 10 8

Аккаунт	Разделы	Информация	Услуги	Приложения
Войти	Публикации	О сайте	Реклама	<div></div>
Регистрация	Хабы	Правила	Тарифы	
	Компании	Помощь	Контент	
	Пользователи	Соглашение	Семинары	
	Песочница	Конфиденциальность		
<div> © 2006 – 2017 «TM»</div>		Служба поддержки	Мобильная версия	