

Раздел

[Хранение данных в браузере](#)

Навигация по уроку

Открыть базу данных

Хранилище объектов

Транзакции

Автоматическая фиксация транзакций

Обработка ошибок

Поиск по ключам

Поиск по индексированному полю

Удаление из хранилища

Курсоры

Обёртка для промисов


Итого

Комментарии

Поделиться



Редактировать на GitHub

 → [Хранение данных в браузере](#) 1-го сентября 2020

# IndexedDB

IndexedDB – это встроенная база данных, более мощная, чем `localStorage`.

- Хранилище ключей/значений: доступны несколько типов ключей, а значения могут быть (почти) любыми.
- Поддерживает транзакции для надёжности.
- Поддерживает запросы в диапазоне ключей и индексы.
- Позволяет хранить больше данных, чем `localStorage`.

Для традиционных клиент-серверных приложений эта мощность обычно чрезмерна. IndexedDB предназначена для офлайн приложений, можно совмещать с ServiceWorkers и другими технологиями.

Интерфейс для IndexedDB, описанный в спецификации <https://www.w3.org/TR/IndexedDB>, основан на событиях.

Мы также можем использовать `async/await` с помощью обёртки, которая основана на промисах, например <https://github.com/jakearchibald/idb>. Это очень удобно, но обёртка не идеальна, она не может полностью заменить события. Поэтому мы начнём с событий, а затем, когда разберёмся в IndexedDB, рассмотрим и обёртку.

## Открыть базу данных

Для начала работы с IndexedDB нужно открыть базу данных.

Синтаксис:

```
1 let openRequest = indexedDB.open(name, version);
```

- `name` – название базы данных, строка.
- `version` – версия базы данных, положительное целое число, по умолчанию 1 (объясняется ниже).

У нас может быть множество баз данных с различными именами, но все они существуют в контексте текущего источника (домен/протокол/порт). Разные сайты не могут получить доступ к базам данных друг друга.

После этого вызова необходимо назначить обработчик событий для объекта `openRequest`:

- `success`: база данных готова к работе, готов «объект базы данных» `openRequest.result`, его следует использовать для дальнейших вызовов.
- `error`: не удалось открыть базу данных.
- `upgradeneeded`: база открыта, но её схема устарела (см. ниже).

**IndexedDB имеет встроенный механизм «версионирования схемы», который отсутствует в серверных базах данных.**

В отличие от серверных баз данных, IndexedDB работает на стороне клиента, в браузере, и у нас нет прямого доступа к данным. Но когда мы публикуем новую версию нашего приложения, возможно, нам понадобится обновить базу данных.

Если локальная версия базы данных меньше, чем версия, определённая в `open`, то сработает специальное событие `upgradeneeded`, и мы сможем сравнить версии и обновить структуры данных по мере необходимости.

Это событие также сработает, если базы данных ещё не существует, так что в этом обработчике мы можем выполнить инициализацию.

Например, когда мы впервые публикуем наше приложение, мы открываем базу данных с версией 1 и выполняем инициализацию в обработчике

upgradeneeded :

Раздел

[Хранение данных в браузере](#)

Навигация по уроку

Открыть базу данных

Хранилище объектов

Транзакции

Автоматическая фиксация транзакций

Обработка ошибок

Поиск по ключам

Поиск по индексированному полю

Удаление из хранилища

Курсоры

Обёртка для промисов

Итого

Комментарии

Поделиться



Редактировать на GitHub



```
1 let openRequest = indexedDB.open("store", 1);
2
3 openRequest.onupgradeneeded = function() {
4   // срабатывает, если на клиенте нет базы данных
5   // ...выполнить инициализацию...
6 };
7
8 openRequest.onerror = function() {
9   console.error("Error", openRequest.error);
10 };
11
12 openRequest.onsuccess = function() {
13   let db = openRequest.result;
14   // продолжить работу с базой данных, используя объект
15 };
```

Когда мы публикуем вторую версию:

```
1 let openRequest = indexedDB.open("store", 2);
2
3 // проверить существование указанной версии базы данных
4 openRequest.onupgradeneeded = function() {
5   // версия существующей базы данных меньше 2 (или база
6   let db = openRequest.result;
7   switch(db.version) { // существующая (старая) версия
8     case 0:
9       // версия 0 означает, что на клиенте нет базы дан
10      // выполнить инициализацию
11     case 1:
12       // на клиенте версия базы данных 1
13       // обновить
14   }
15 };
```

Таким образом, в `openRequest.onupgradeneeded` мы обновляем базу данных. Скоро подробно увидим, как это делается. А после того, как этот обработчик завершится без ошибок, сработает `openRequest.onsuccess`.

После `openRequest.onsuccess` у нас есть объект базы данных в `openRequest.result`, который мы будем использовать для дальнейших операций.

Удалить базу данных:

```
1 let deleteRequest = indexedDB.deleteDatabase(name)
2 // deleteRequest.onsuccess/onerror отслеживает результа
```

#### ⚠ А что, если открыть предыдущую версию?

Что если мы попробуем открыть базу с более низкой версией, чем текущая? Например, на клиенте база версии 3, а мы вызываем `open(...2)`.

Возникнет ошибка, сработает `openRequest.onerror`.

Такое может произойти, если посетитель загрузил устаревший код, например, из кеша прокси. Нам следует проверить `db.version` и предложить ему перезагрузить страницу. А также проверить наши кеширующие заголовки, убедиться, что посетитель никогда не получит устаревший код.

## Проблема параллельного обновления

Раздел

[Хранение данных в браузере](#)

Навигация по уроку

Открыть базу данных

Хранилище объектов

Транзакции

Автоматическая фиксация транзакций

Обработка ошибок

Поиск по ключам

Поиск по индексированному полю

Удаление из хранилища

Курсоры

Обёртка для промисов

Итого

Комментарии

Поделиться



Редактировать на GitHub



Раз уж мы говорим про версионирование, рассмотрим связанную с этим небольшую проблему.

Допустим, посетитель открыл наш сайт во вкладке браузера, с базой версии 1.

Затем мы выкатили обновление, и тот же посетитель открыл наш сайт в другой вкладке. Так что есть две вкладки, на которых открыт наш сайт, но в одной открыто соединение с базой версии 1, а другая пытается обновить версию базы в обработчике `upgradeneeded`.

Проблема заключается в том, что база данных всего одна на две вкладки, так как это один и тот же сайт, один источник. И она не может быть одновременно версии 1 и 2. Чтобы обновить на версию 2, все соединения к версии 1 должны быть закрыты.

Чтобы это можно было организовать, при попытке обновления на объекте базы возникает событие `versionchange`. Нам нужно слушать его и закрыть соединение к базе (а также, возможно, предложить пользователю перезагрузить страницу, чтобы получить обновлённый код).

Если мы его не закроем, то второе, новое соединение будет заблокировано с событием `blocked` вместо `success`.

Код, который это делает:

```
1 let openRequest = indexedDB.open("store", 2);
2
3 openRequest.onupgradeneeded = ...;
4 openRequest.onerror = ...;
5
6 openRequest.onsuccess = function() {
7   let db = openRequest.result;
8
9   db.onversionchange = function() {
10     db.close();
11     alert("База данных устарела, пожалуйста, перезагрузи");
12   };
13
14   // ...база данных доступна как объект db...
15 };
16
17 openRequest.onblocked = function() {
18   // есть другое соединение к той же базе
19   // и оно не было закрыто после срабатывания на нём db
20 };
```

Здесь мы делаем две вещи:

1. Добавляем обработчик `db.onversionchange` после успешного открытия базы, чтобы узнать о попытке параллельного обновления.
2. Добавляем обработчик `openRequest.onblocked` для ситуаций, когда старое соединение не было закрыто. Такого не произойдёт, если мы закрываем его в `db.onversionchange`.

Есть и другие варианты. Например, мы можем более «мягко» закрыть соединение в `db.onversionchange`, предложить пользователю сохранить данные перед этим. Новое обновляющее соединение будет заблокировано сразу после того как обработчик `db.onversionchange` завершится, не закрыв соединение, и мы можем в новой вкладке попросить посетителя закрыть старые для обновления.

Такой конфликт при обновлении происходит редко, но мы должны как-то его обрабатывать, хотя бы поставить обработчик `onblocked`, чтобы наш скрипт не «умирал» молча, удивляя посетителя.

## Хранилище объектов

Чтобы сохранить что-то в IndexedDB, нам нужно *хранилище объектов*.

Хранилище объектов – это основная концепция IndexedDB. В других базах данных это «таблицы» или «коллекции». Здесь хранятся данные. В базе

Раздел

[Хранение данных в браузере](#)

Навигация по уроку

Открыть базу данных

Хранилище объектов

Транзакции

Автоматическая фиксация транзакций

Обработка ошибок

Поиск по ключам

Поиск по индексированному полю

Удаление из хранилища

Курсоры

Обёртка для промисов

Итого

Комментарии

Поделиться



Редактировать на GitHub



данных может быть множество хранилищ: одно для пользователей, другое для товаров и так далее.

Несмотря на то, что название – «хранилище объектов», примитивы тоже могут там храниться.

**Мы можем хранить почти любое значение, в том числе сложные объекты.**

IndexedDB использует [стандартный алгоритм сериализации](#) для клонирования и хранения объекта. Это как `JSON.stringify`, но более мощный, способный хранить гораздо больше типов данных.

Пример объекта, который нельзя сохранить: объект с циклическими ссылками. Такие объекты не сериализуемы. `JSON.stringify` также выдаст ошибку при сериализации.

**Каждому значению в хранилище должен соответствовать уникальный ключ.**

Ключ должен быть одним из следующих типов: `number`, `date`, `string`, `binary` или `array`. Это уникальный идентификатор: по ключу мы можем искать/удалять/обновлять значения.

### Database

#### objectStore

key1: value1
key2: value2
key3: value3
key4: value4
key5: value5

#### objectStore

key1: value1
key2: value2
key3: value3
key4: value4
key5: value5

#### objectStore

key1: value1
key2: value2
key3: value3
key4: value4
key5: value5

Как мы видим, можно указать ключ при добавлении значения в хранилище, аналогично `localStorage`. Но когда мы храним объекты, IndexedDB позволяет установить свойство объекта в качестве ключа, что гораздо удобнее. Или мы можем автоматически сгенерировать ключи.

Но для начала нужно создать хранилище.

Синтаксис для создания хранилища объектов:

```
1 db.createObjectStore(name[, keyOptions]);
```

Обратите внимание, что операция является синхронной, использование `await` не требуется.

- `name` – это название хранилища, например `"books"` для книг,
- `keyOptions` – это необязательный объект с одним или двумя свойствами:
  - `keyPath` – путь к свойству объекта, которое IndexedDB будет использовать в качестве ключа, например `id`.
  - `autoIncrement` – если `true`, то ключ будет формироваться автоматически для новых объектов, как постоянно увеличивающееся число.

Если при создании хранилища не указать `keyOptions`, то нам потребуется явно указать ключ позже, при сохранении объекта.

Например, это хранилище объектов использует свойство `id` как ключ:

```
1 db.createObjectStore('books', {keyPath: 'id'});
```

**Хранилище объектов можно создавать/изменять только при обновлении версии базы данных в обработчике `upgradeneeded`.**

Это техническое ограничение. Вне обработчика мы сможем добавлять/удалять/обновлять данные, но хранилища объектов могут быть созданы/удалены/изменены только во время обновления версии базы данных.

Раздел

[Хранение данных в браузере](#)

Навигация по уроку

Открыть базу данных

Хранилище объектов

Транзакции

Автоматическая фиксация транзакций

Обработка ошибок

Поиск по ключам

Поиск по индексированному полю

Удаление из хранилища

Курсоры

Обёртка для промисов

Итого

Комментарии

Поделиться



Редактировать на GitHub



Для обновления версии базы есть два основных подхода:

1. Мы можем реализовать функции обновления по версиям: с 1 на 2, с 2 на 3 и т.д. Потом в `onupgradeneeded` сравнить версии (например, была 2, сейчас 4) и запустить операции обновления для каждой промежуточной версии (2 на 3, затем 3 на 4).
2. Или мы можем взять список существующих хранилищ объектов, используя `db.objectStoreNames`. Этот объект является [DOMStringList](#), в нём есть метод `contains(name)`, используя который можно проверить существование хранилища. Посмотреть, какие хранилища есть и создать те, которых нет.

Для простых баз данных второй подход может быть проще и предпочтительнее.

Вот демонстрация второго способа:

```
1 let openRequest = indexedDB.open("db", 2);
2
3 // создаём хранилище объектов для books, если ещё не су
4 openRequest.onupgradeneeded = function() {
5   let db = openRequest.result;
6   if (!db.objectStoreNames.contains('books')) { // если
7     db.createObjectStore('books', {keyPath: 'id'}); //
8   }
9   };
```

Чтобы удалить хранилище объектов:

```
1 db.deleteObjectStore('books')
```

## Транзакции

Термин «транзакция» является общеизвестным, транзакции используются во многих видах баз данных.

Транзакция – это группа операций, которые должны быть или все выполнены, или все не выполнены (всё или ничего).

Например, когда пользователь что-то покупает, нам нужно:

1. Вычесть деньги с его счёта.
2. Отправить ему покупку.

Будет очень плохо, если мы успеем завершить первую операцию, а затем что-то пойдёт не так, например отключат электричество, и мы не сможем завершить вторую операцию. Обе операции должны быть успешно завершены (покупка сделана, отлично!) или необходимо отменить обе операции (в этом случае пользователь сохранит свои деньги и может попытаться купить ещё раз).

Транзакции гарантируют это.

**Все операции с данными в IndexedDB могут быть сделаны только внутри транзакций.**

Для начала транзакции:

```
1 db.transaction(store[, type]);
```

- `store` – это название хранилища, к которому транзакция получит доступ, например, `"books"`. Может быть массивом названий, если нам нужно предоставить доступ к нескольким хранилищам.
- `type` – тип транзакции, один из:
  - `readonly` – только чтение, по умолчанию.
  - `readwrite` – только чтение и запись данных, создание/удаление самих хранилищ объектов недоступно.

Раздел

[Хранение данных в браузере](#)

Навигация по уроку

Открыть базу данных

Хранилище объектов

Транзакции

Автоматическая фиксация транзакций

Обработка ошибок

Поиск по ключам

Поиск по индексированному полю

Удаление из хранилища

Курсоры

Обёртка для промисов

Итого

Комментарии

Поделиться



[Редактировать на GitHub](#)



Есть ещё один тип транзакций: `versionchange`. Такие транзакции могут делать любые операции, но мы не можем создать их вручную. IndexedDB автоматически создаёт транзакцию типа `versionchange`, когда открывает базу данных, для обработчика `updateneeded`. Вот почему это единственное место, где мы можем обновлять структуру базы данных, создавать/удалять хранилища объектов.

### ❗ Почему существует несколько типов транзакций?

Производительность является причиной, почему транзакции необходимо помечать как `readonly` или `readwrite`.

Несколько `readonly` транзакций могут одновременно работать с одним и тем же хранилищем объектов, а `readwrite` транзакций – не могут. Транзакции типа `readwrite` «блокируют» хранилище для записи. Следующая такая транзакция должна дожидаться выполнения предыдущей, перед тем как получит доступ к тому же самому хранилищу.

После того, как транзакция будет создана, мы можем добавить элемент в хранилище, вот так:

```
1 let transaction = db.transaction("books", "readwrite");
2
3 // получить хранилище объектов для работы с ним
4 let books = transaction.objectStore("books"); // (2)
5
6 let book = {
7   id: 'js',
8   price: 10,
9   created: new Date()
10 };
11
12 let request = books.add(book); // (3)
13
14 request.onsuccess = function() { // (4)
15   console.log("Книга добавлена в хранилище", request.re
16 };
17
18 request.onerror = function() {
19   console.log("Ошибка", request.error);
20 };
```

Мы сделали четыре шага:

1. Создать транзакцию и указать все хранилища, к которым необходим доступ, строка (1).
2. Получить хранилище объектов, используя `transaction.objectStore(name)`, строка (2).
3. Выполнить запрос на добавление элемента в хранилище объектов `books.add(book)`, строка (3).
4. ...Обработать результат запроса (4), затем мы можем выполнить другие запросы и так далее.

Хранилища объектов поддерживают два метода для добавления значений:

- **put(value, [key])** Добавляет значение `value` в хранилище. Ключ `key` необходимо указать, если при создании хранилища объектов не было указано свойство `keyPath` или `autoIncrement`. Если уже есть значение с таким же ключом, то оно будет заменено.
- **add(value, [key])** То же, что `put`, но если уже существует значение с таким ключом, то запрос не выполнится, будет сгенерирована ошибка с названием `"ConstraintError"`.

Аналогично открытию базы, мы отправляем запрос: `books.add(book)` и после ожидаем события `success/error`.

- `request.result` для `add` является ключом нового объекта.

- Ошибка находится в `request.error` (если есть).

## Автоматическая фиксация транзакций

В примере выше мы запустили транзакцию и выполнили запрос `add`. Но, как говорилось ранее, транзакция может включать в себя несколько запросов, которые все вместе должны либо успешно завершиться, либо нет. Как нам закончить транзакцию, обозначить, что больше запросов в ней не будет?

Короткий ответ: этого не требуется.

В следующей 3.0 версии спецификации, вероятно, будет возможность вручную завершить транзакцию, но сейчас, в версии 2.0, такой возможности нет.

**Когда все запросы завершены и очередь микрзадач пуста, тогда транзакция завершится автоматически.**

Как правило, это означает, что транзакция автоматически завершается, когда выполнены все её запросы и завершился текущий код.

Таким образом, в приведённом выше примере не требуется никакой специальный вызов, чтобы завершить транзакцию.

Такое автозавершение транзакций имеет важный побочный эффект. Мы не можем вставить асинхронную операцию, такую как `fetch` или `setTimeout` в середину транзакции. IndexedDB никак не заставит транзакцию «висеть» и ждать их выполнения.

В приведённом ниже коде в запросе `request2` в строке с (\*) будет ошибка, потому что транзакция уже завершена, больше нельзя выполнить в ней запрос:

```
1 let request1 = books.add(book);
2
3 request1.onsuccess = function() {
4   fetch('/').then(response => {
5     let request2 = books.add(anotherBook); // (*)
6     request2.onerror = function() {
7       console.log(request2.error.name); // TransactionI
8     };
9   });
10  };
```

Всё потому, что `fetch` является асинхронной операцией, макрозадачей. Транзакции завершаются раньше, чем браузер приступает к выполнению макрозадач.

Авторы спецификации IndexedDB из соображений производительности считают, что транзакции должны завершаться быстро.

В частности, `readwrite` транзакции «блокируют» хранилища от записи. Таким образом, если одна часть приложения инициирует `readwrite` транзакцию в хранилище объектов `books`, то другая часть приложения, которая хочет сделать то же самое, должна ждать: новая транзакция «зависает» до завершения первой. Это может привести к странным задержкам, если транзакции слишком долго выполняются.

Что же делать?

В приведённом выше примере мы могли бы запустить новую транзакцию `db.transaction` перед новым запросом (\*).

Но ещё лучше выполнять операции вместе, в рамках одной транзакции: отделить транзакции IndexedDB от других асинхронных операций.

Сначала сделаем `fetch`, подготовим данные, если нужно, затем создадим транзакцию и выполним все запросы к базе данных.

Чтобы поймать момент успешного выполнения, мы можем повесить обработчик на событие `transaction.oncomplete`:

```
1 let transaction = db.transaction("books", "readwrite");
2
```

Раздел

[Хранение данных в браузере](#)

Навигация по уроку

Открыть базу данных

Хранилище объектов

Транзакции

Автоматическая фиксация транзакций

Обработка ошибок

Поиск по ключам

Поиск по индексированному полю

Удаление из хранилища

Курсоры

Обёртка для промисов

Итого

Комментарии

Поделиться



Редактировать на GitHub

Раздел

[Хранение данных в браузере](#)

Навигация по уроку

Открыть базу данных

Хранилище объектов

Транзакции

Автоматическая фиксация транзакций

Обработка ошибок

Поиск по ключам

Поиск по индексированному полю

Удаление из хранилища

Курсоры

Обёртка для промисов

Итого

Комментарии

Поделиться



[Редактировать на GitHub](#)



```
3 // ...выполнить операции...
4
5 transaction.oncomplete = function() {
6   console.log("Транзакция выполнена");
7 };
```

Только `complete` гарантирует, что транзакция сохранена целиком. По отдельности запросы могут выполняться, но при финальной записи что-то может пойти не так (ошибка ввода-вывода, проблема с диском, например).

Чтобы вручную отменить транзакцию, выполните:

```
1 transaction.abort();
```

Это отменит все изменения, сделанные запросами в транзакции, и сгенерирует событие `transaction.onabort`.

## Обработка ошибок

Запросы на запись могут выполняться неудачно.

Мы должны быть готовы к этому, не только из-за возможных ошибок на нашей стороне, но и по причинам, которые не связаны с транзакцией. Например, размер хранилища может быть превышен. И мы должны быть готовы обработать такую ситуацию.

**При ошибке в запросе соответствующая транзакция отменяется полностью, включая изменения, сделанные другими её запросами.**

Если мы хотим продолжить транзакцию (например, попробовать другой запрос без отмены изменений), это также возможно. Для этого в обработчике `request.onerror` следует вызвать `event.preventDefault()`.

В примере ниже новая книга добавляется с тем же ключом (`id`), что и существующая. Метод `store.add` генерирует в этом случае ошибку `"ConstraintError"`. Мы обрабатываем её без отмены транзакции:

```
1 let transaction = db.transaction("books", "readwrite");
2
3 let book = { id: 'js', price: 10 };
4
5 let request = transaction.objectStore("books").add(book);
6
7 request.onerror = function(event) {
8   // ConstraintError возникает при попытке добавить объект
9   if (request.error.name === "ConstraintError") {
10     console.log("Книга с таким id уже существует"); // ...
11     event.preventDefault(); // предотвращаем отмену транзакции
12     // ...можно попробовать использовать другой ключ...
13   } else {
14     // неизвестная ошибка
15     // транзакция будет отменена
16   }
17 };
18
19 transaction.onabort = function() {
20   console.log("Ошибка", transaction.error);
21 };
```

## Делегирование событий

Нужны ли обработчики `onerror`/`onsuccess` для каждого запроса? Не всегда. Мы можем использовать делегирование событий.

**События IndexedDB всплывают: запрос → транзакция → база данных.**

Все события являются DOM-событиями с фазами перехвата и всплытия, но обычно используется только всплытие.



Раздел

[Хранение данных в браузере](#)

Навигация по уроку

Открыть базу данных

Хранилище объектов

Транзакции

Автоматическая фиксация транзакций

Обработка ошибок

Поиск по ключам

Поиск по индексированному полю

Удаление из хранилища

Курсоры

Обёртка для промисов

Итого

Комментарии

Поделиться



[Редактировать на GitHub](#)



Поэтому мы можем перехватить все ошибки, используя обработчик `db.onerror`, для оповещения пользователя или других целей:

```
1 db.onerror = function(event) {
2   let request = event.target; // запрос, в котором прои
3
4   console.log("Ошибка", request.error);
5 };
```

...А если мы полностью обработали ошибку? В этом случае мы не хотим сообщать об этом.

Мы можем остановить всплытие и, следовательно, `db.onerror`, используя `event.stopPropagation()` в `request.onerror`.

```
1 request.onerror = function(event) {
2   if (request.error.name == "ConstraintError") {
3     console.log("Книга с таким id уже существует"); //
4     event.preventDefault(); // предотвращаем отмену тра
5     event.stopPropagation(); // предотвращаем всплытие
6   } else {
7     // ничего не делаем
8     // транзакция будет отменена
9     // мы можем обработать ошибку в transaction.onabort
10  }
11 };
```

## Поиск по ключам

Есть два основных вида поиска в хранилище объектов:

1. По ключу или по диапазону ключей. То есть: по `book.id` в хранилище «books».
2. По полям объекта, например, `book.price`.

Сначала давайте разберёмся с ключами и диапазоном ключей (1).

Методы поиска поддерживают либо точные ключи, либо так называемые «запросы с диапазоном» – `IDBKeyRange` объекты, которые задают «диапазон ключей».

Диапазоны создаются с помощью следующих вызовов:

- `IDBKeyRange.lowerBound(lower, [open])` означает:  $>lower$  (или  $\geq lower$ , если `open` это `true`)
- `IDBKeyRange.upperBound(upper, [open])` означает:  $<upper$  (или  $\leq upper$ , если `open` это `true`)
- `IDBKeyRange.bound(lower, upper, [lowerOpen], [upperOpen])` означает: между `lower` и `upper`, включительно, если соответствующий `open` равен `true`.
- `IDBKeyRange.only(key)` – диапазон, который состоит только из одного ключа `key`, редко используется.

Все методы поиска принимают аргумент `query`, который может быть либо точным ключом, либо диапазоном ключей:

- `store.get(query)` – поиск первого значения по ключу или по диапазону.
- `store.getAll([query], [count])` – поиск всех значений, можно ограничить, передав `count`.
- `store.getKey(query)` – поиск первого ключа, который удовлетворяет запросу, обычно передаётся диапазон.
- `store.getAllKeys([query], [count])` – поиск всех ключей, которые удовлетворяют запросу, обычно передаётся диапазон, возможно ограничить поиск, передав `count`.
- `store.count([query])` – получить общее количество ключей, которые удовлетворяют запросу, обычно передаётся диапазон.

Раздел

[Хранение данных в браузере](#)

Навигация по уроку

Открыть базу данных

Хранилище объектов

Транзакции

Автоматическая фиксация транзакций

Обработка ошибок

Поиск по ключам

Поиск по индексированному полю

Удаление из хранилища

Курсоры

Обёртка для промисов

Итого

Комментарии

Поделиться



[Редактировать на GitHub](#)



Например, в хранилище у нас есть множество книг. Помните, поле `id` является ключом, поэтому все эти методы могут искать по ключу `id`.

Примеры запросов:

```
1 // получить одну книгу
2 books.get('js')
3
4 // получить все книги с 'css' < id < 'html'
5 books.getAll(IDBKeyRange.bound('css', 'html'))
6
7 // получить книги с 'html' <= id
8 books.getAll(IDBKeyRange.lowerBound('html', true))
9
10 // получить все книги
11 books.getAll()
12
13 // получить все ключи: id >= 'js'
14 books.getAllKeys(IDBKeyRange.lowerBound('js', true))
```

#### Хранилище объектов всегда отсортировано

Хранилище объектов внутренне сортирует значения по ключам.

Поэтому запросы, которые возвращают много значений, всегда возвращают их в порядке сортировки по ключу.

## Поиск по индексированному полю

Для поиска по другим полям объекта нам нужно создать дополнительную структуру данных, называемую «индекс» (index).

Индекс является «расширением» к хранилищу, которое отслеживает данное поле объекта. Для каждого значения этого поля хранится список ключей для объектов, которые имеют это значение. Ниже будет более подробная картина.

Синтаксис:

```
1 objectStore.createIndex(name, keyPath, [options]);
```

- **name** – название индекса,
- **keyPath** – путь к полю объекта, которое индекс должен отслеживать (мы собираемся сделать поиск по этому полю),
- **option** – необязательный объект со свойствами:
  - **unique** – если `true`, тогда в хранилище может быть только один объект с заданным значением в `keyPath`. Если мы попытаемся добавить дубликат, то индекс сгенерирует ошибку.
  - **multiEntry** – используется только, если `keyPath` является массивом. В этом случае, по умолчанию, индекс обрабатывает весь массив как ключ. Но если мы укажем `true` в `multiEntry`, тогда индекс будет хранить список объектов хранилища для каждого значения в этом массиве. Таким образом, элементы массива становятся ключами индекса.

В нашем примере мы храним книги с ключом `id`.

Допустим, мы хотим сделать поиск по полю `price`.

Сначала нам нужно создать индекс. Индексы должны создаваться в `upgradeneeded`, как и хранилище объектов:

```
1 openRequest.onupgradeneeded = function() {
2   // мы должны создать индекс здесь, в versionchange тр.
3   let books = db.createObjectStore('books', {keyPath: '
4   let index = inventory.createIndex('price_idx', 'price
5 };
```

Раздел

[Хранение данных в браузере](#)

Навигация по уроку

Открыть базу данных

Хранилище объектов

Транзакции

Автоматическая фиксация транзакций

Обработка ошибок

Поиск по ключам

Поиск по индексированному полю

Удаление из хранилища

Курсоры

Обёртка для промисов

Итого

Комментарии

Поделиться

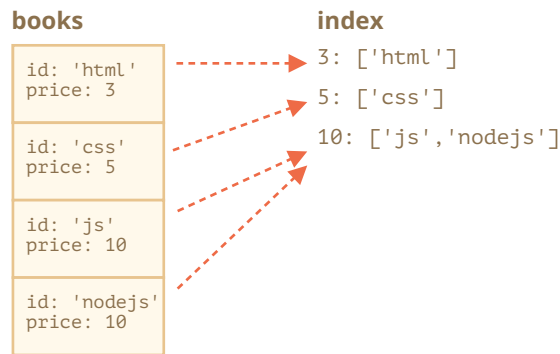


[Редактировать на GitHub](#)



- Индекс будет отслеживать поле `price`.
- Поле `price` не уникальное, у нас может быть несколько книг с одинаковой ценой, поэтому мы не устанавливаем опцию `unique`.
- Поле `price` не является массивом, поэтому флаг `multiEntry` не применим.

Представим, что в нашем `inventory` есть 4 книги. Вот картинка, которая показывает, что такое «индекс».



Как уже говорилось, индекс для каждого значения `price` (второй аргумент) хранит список ключей, имеющих эту цену.

Индексы автоматически обновляются, нам не нужно об этом заботиться.

Сейчас, когда мы хотим найти объект по цене, мы просто применяем те же методы поиска к индексу:

```
1 let transaction = db.transaction("books"); // readonly
2 let books = transaction.objectStore("books");
3 let priceIndex = books.index("price_idx");
4
5 let request = priceIndex.getAll(10);
6
7 request.onsuccess = function() {
8   if (request.result !== undefined) {
9     console.log("Книги", request.result); // массив книг
10  } else {
11    console.log("Нет таких книг");
12  }
13 };
```

Мы также можем использовать `IDBKeyRange`, чтобы создать диапазон и найти дешёвые/дорогие книги:

```
1 // найдём книги, где цена < 5
2 let request = priceIndex.getAll(IDBKeyRange.upperBound(
```

Индексы внутренне отсортированы по полю отслеживаемого объекта, в нашем случае по `price`. Поэтому результат поиска будет уже отсортированный по полю `price`.

## Удаление из хранилища

Метод `delete` удаляет значения по запросу, формат вызова такой же как в `getAll`:

- **`delete(query)`** – производит удаление соответствующих запросу значений.

Например:

Раздел

[Хранение данных в браузере](#)

Навигация по уроку

Открыть базу данных

Хранилище объектов

Транзакции

Автоматическая фиксация транзакций

Обработка ошибок

Поиск по ключам

Поиск по индексированному полю

Удаление из хранилища

Курсоры

Обёртка для промисов

Итого

Комментарии

Поделиться



Редактировать на GitHub

```
1 // удалить книгу с id='js'
2 books.delete('js');
```



Если нам нужно удалить книги, основываясь на цене или на любом другом поле, сначала нам надо найти ключ в индексе, а затем выполнить `delete`:



```
1 // найдём ключ, где цена = 5
2 let request = priceIndex.getKey(5);
3
4 request.onsuccess = function() {
5   let id = request.result;
6   let deleteRequest = books.delete(id);
7 };
```

Чтобы удалить всё:

```
1 books.clear(); // очищаем хранилище.
```

## Курсоры

Такие методы как `getAll/getAllKeys` возвращают массив ключей/значений.

Но хранилище объектов может быть огромным, больше, чем доступно памяти.

Тогда метод `getAll` вернёт ошибку при попытке получить все записи в массиве.

Что делать?

Курсоры предоставляют возможности для работы в таких ситуациях.



**Объект *cursor* идёт по хранилищу объектов с заданным запросом (query) и возвращает пары ключ/значение по очереди, а не все сразу. Это позволяет экономить память.**



Так как хранилище объектов внутренне отсортировано по ключу, курсор проходит по хранилищу в порядке хранения ключей (по возрастанию по умолчанию).

Синтаксис:

```
1 // как getAll, но с использованием курсора:
2 let request = store.openCursor(query, [direction]);
3
4 // чтобы получить ключи, не значения (как getAllKeys):
```

- **query** ключ или диапазон ключей, как для `getAll`.
- **direction** необязательный аргумент, доступные значения:
  - "next" – по умолчанию, курсор будет проходить от самого маленького ключа к большему.
  - "prev" – обратный порядок: от самого большого ключа к меньшему.
  - "nextunique", "prevunique" – то же самое, но курсор пропускает записи с тем же ключом, что уже был (только для курсоров по индексам, например, для нескольких книг с `price=5`, будет возвращена только первая).

**Основным отличием курсора является то, что `request.onsuccess` генерируется многократно: один раз для каждого результата.**

Вот пример того, как использовать курсор:

```
1 let transaction = db.transaction("books");
2 let books = transaction.objectStore("books");
3
4 let request = books.openCursor();
```

Раздел

[Хранение данных в браузере](#)

Навигация по уроку

Открыть базу данных

Хранилище объектов

Транзакции

Автоматическая фиксация транзакций

Обработка ошибок

Поиск по ключам

Поиск по индексированному полю

Удаление из хранилища

Курсоры

Обёртка для промисов

Итого

Комментарии

Поделиться



[Редактировать на GitHub](#)



```
5
6 // вызывается для каждой найденной курсором книги
7 request.onsuccess = function() {
8   let cursor = request.result;
9   if (cursor) {
10    let key = cursor.key; // ключ книги (поле id)
11    let value = cursor.value; // объект книги
12    console.log(key, value);
13    cursor.continue();
14  } else {
15    console.log("Книг больше нет");
16  }
17 };
```

Основные методы курсора:

- `advance(count)` – продвинуть курсор на `count` позиций, пропустив значения.
- `continue([key])` – продвинуть курсор к следующему значению в диапазоне соответствия (или до позиции сразу после ключа `key`, если указан).

Независимо от того, есть ли ещё значения, соответствующие курсору или нет – вызывается `onsuccess`, затем в `result` мы можем получить курсор, указывающий на следующую запись или равный `undefined`.

В приведённом выше примере курсор был создан для хранилища объектов.

Но мы также можем создать курсор для индексов. Как мы помним, индексы позволяют искать по полю объекта. Курсоры для индексов работают так же, как для хранилищ объектов – они позволяют экономить память, возвращая одно значение в единицу времени.

Для курсоров по индексам `cursor.key` является ключом индекса (например `price`), нам следует использовать свойство `cursor.primaryKey` как ключ объекта:

```
1 let request = priceIdx.openCursor(IDBKeyRange.upperBoun
2
3 // вызывается для каждой записи
4 request.onsuccess = function() {
5   let cursor = request.result;
6   if (cursor) {
7     let key = cursor.primaryKey; // следующий ключ в хр
8     let value = cursor.value; // следующее значение в х
9     let key = cursor.key; // следующий ключ индекса (pr
10    console.log(key, value);
11    cursor.continue();
12  } else {
13    console.log("Книг больше нет");
14  }
15 };
```

## Обёртка для промисов

Добавлять к каждому запросу `onsuccess/onerror` немного громоздко. Мы можем сделать нашу жизнь проще, используя делегирование событий, например, установить обработчики на все транзакции, но использовать `async/await` намного удобнее.

Давайте далее в главе использовать небольшую обёртку над промисами <https://github.com/jakearchibald/idb>. Она создаёт глобальный `idb` объект с промисифицированными `IndexedDB` методами.

Тогда вместо `onsuccess/onerror` мы можем писать примерно так:

```
1 let db = await idb.openDb('store', 1, db => {
2   if (db.oldVersion == 0) {
3     // выполняем инициализацию
4     db.createObjectStore('books', {keyPath: 'id'});
```

Раздел

[Хранение данных в браузере](#)

Навигация по уроку

Открыть базу данных

Хранилище объектов

Транзакции

Автоматическая фиксация транзакций

Обработка ошибок

Поиск по ключам

Поиск по индексированному полю

Удаление из хранилища

Курсоры

Обёртка для промисов

Итого

Комментарии

Поделиться



[Редактировать на GitHub](#)



```
5   }
6   });
7
8   let transaction = db.transaction('books', 'readwrite');
9   let books = transaction.objectStore('books');
10
11  try {
12    await books.add(...);
13    await books.add(...);
14
15    await transaction.complete;
16
17    console.log('сохранено');
18  } catch(err) {
19    console.log('ошибка', err.message);
20  }
```

Теперь у нас красивый «плоский асинхронный» код и, конечно, будет работать `try..catch`.

## Обработка ошибок

Если мы не перехватим ошибку, то она «вывалится» наружу, вверх по стеку вызовов, до ближайшего внешнего `try..catch`.

Необработанная ошибка становится событием «unhandled promise rejection» в объекте `window`.

Мы можем обработать такие ошибки вот так:

```
1 window.addEventListener('unhandledrejection', event =>
2   let request = event.target; // объект запроса IndexedDB
3   let error = event.reason; // Необработанный объект об
4   ...сообщить об ошибке...
5 });
```

## Подводный камень: «Inactive transaction»

Как мы уже знаем, транзакции автоматически завершаются, как только браузер завершает работу с текущим кодом и макрозадачу. Поэтому, если мы поместим *макрозадачу* наподобие `fetch` в середину транзакции, транзакция не будет ожидать её завершения. Произойдёт автозавершение транзакции. Поэтому при следующем запросе возникнет ошибка.

Для промисифицирующей обёртки и `async/await` поведение такое же.

Вот пример `fetch` в середине транзакции:

```
1 let transaction = db.transaction("inventory", "readwrite");
2 let inventory = transaction.objectStore("inventory");
3
4 await inventory.add({ id: 'js', price: 10, created: new
5
6 await fetch(...); // (*)
7
8 await inventory.add({ id: 'js', price: 10, created: new
```

Следующий `inventory.add` после `fetch (*)` не работает, сгенерируется ошибка «inactive transaction», потому что транзакция уже завершена и закрыта к этому времени.

Решение такое же, как при работе с обычным IndexedDB: либо создать новую транзакцию, либо разделить задачу на части.

1. Подготовить данные и получить всё, что необходимо.
2. Затем сохранить в базу данных.

## Получение встроенных объектов

Раздел

[Хранение данных в браузере](#)

Навигация по уроку

Открыть базу данных

Хранилище объектов

Транзакции

Автоматическая фиксация транзакций

Обработка ошибок

Поиск по ключам

Поиск по индексированному полю

Удаление из хранилища

Курсоры

Обёртка для промисов

Итого

Комментарии

Поделиться



Редактировать на GitHub



Внутренне обёртка выполняет встроенные IndexedDB запросы, добавляя к ним `onerror/onsuccess`, и возвращает промисы, которые отклоняются или выполняются с переданным результатом.

Это работает в большинстве случаев. Примеры можно увидеть на странице библиотеки <https://github.com/jakearchibald/idb>.

В некоторых редких случаях, когда нам нужен оригинальный объект `request`, мы можем получить в нему доступ, используя свойство `promise.request`:

```
1 let promise = books.add(book); // получаем промис (без
2
3 let request = promise.request; // встроенный объект зап
4 let transaction = request.transaction; // встроенный об
5
6 // ...работаем с IndexedDB...
7
8 let result = await promise; // если ещё нужно
```

## Итого

IndexedDB можно рассматривать как «localStorage на стероидах». Это простая база данных типа ключ-значение, достаточно мощная для оффлайн приложений, но простая в использовании.

Лучшим руководством является спецификация, [текущая версия 2.0](#), но также поддерживаются несколько методов из 3.0 (не так много отличий) версии.

Использование можно описать в нескольких фразах:

1. Подключить обёртку над промисами, например [idb](#).
2. Открыть базу данных: `idb.openDb(name, version, onupgradeneeded)`
  - Создание хранилищ объектов и индексов происходит в обработке `onupgradeneeded`.
  - Обновление версии – либо сравнивая номера версий, либо можно проверить что существует, а что нет.
3. Для запросов:
  - Создать транзакцию `db.transaction('books')` (можно указать `readwrite`, если надо).
  - Получить хранилище объектов `transaction.objectStore('books')`.
4. Затем для поиска по ключу вызываем методы непосредственно у хранилища объектов.
  - Для поиска по любому полю объекта создайте индекс.
5. Если данные не помещаются в памяти, то используйте курсор.

Демо-приложение:

Результат index.html



```
<!doctype html>
<script src="https://cdn.jsdelivr.net/npm/idb@3.0.2/build/idb"></script>

<button onclick="addBook()">Добавить книгу</button>
<button onclick="clearBooks()">Очистить хранилище</button>

<p>Список книг:</p>
```



Раздел

[Хранение данных в браузере](#)

Навигация по уроку

Открыть базу данных

Хранилище объектов

Транзакции

Автоматическая фиксация транзакций

Обработка ошибок

Поиск по ключам

Поиск по индексированному полю

Удаление из хранилища

Курсоры

Обёртка для промисов

Итого

Комментарии

Поделиться



Редактировать на GitHub

