API Reference

API overview (https://github.com/pouchdb/pouchdb/edit/master/docs/_i

PouchDB has an asynchronous API, supporting callbacks (http://docs.nodejitsu.com/articles/getting-started/control-flow/what-are-callbacks), promises (https://www.promisejs.org/), and async functions (https://jakearchibald.com/2014/es7-async-functions/). For beginners, we recommend promises, although you are free to use whatever format you prefer. If you are unsure, check out our guide to asynchronous code (/guides/async-code.html).

Most of the API is exposed as:

```
db.doSomething(args..., [options], [callback])
```

… where both the `options` and `callback` are optional.

Callbacks

Callbacks use the standard Node.js idiom of:

```
function(error, result) { /* ... */ }
```

... where the `error` will be undefined if there's no error.

Promises

If you don't specify a `callback`, then the API returns a promise (https://www.promisejs.org/). In supported browsers (http://caniuse.com/#feat=promises) or Node.js, native promises are used, falling back to the minimal library lie (https://github.com/calvinmetcalf/lie) as needed.

i **Using Ionic/AngularJS?** You can wrap PouchDB promises in `$q.when()` (https://docs.angularjs.org/api/ng/service/$q#when). This will notify AngularJS to update the UI when the PouchDB promise has resolved.

To use a custom promise implementation with PouchDB, you must redefine a global `Promise` object before loading PouchDB:

```
<script>window.Promise = MyCustomPromiseLibrary;</script>
<script src="path/to/pouchdb.js"></script>
```

Async functions

If you are using a transpiler like Babel (http://babeljs.io/), you can enable async functions (https://github.com/tc39/ecmascript-asyncawait), which are an experimental API tentatively slated for release in ES7 (ES2016). This allows you to use the `async` / `await` keywords when consuming promise-based APIs like PouchDB's.

i **How to configure Babel?** To use async functions, you will need the syntax-async-functions plugin (http://babeljs.io/docs/plugins/syntax-async-functions/), as well as the transform-regenerator plugin (https://babeljs.io/docs/plugins/transform-regenerator/) or Kneden (https://github.com/marten-de-vries/kneden) (which is experimental as of this writing). For a full working example, see async-functions-with-regenerator (https://github.com/nolanlawson/async-functions-with-regenerator).

Note that the samples for `async` / `await` in the API documentation assume that your code is inside an async function. So for instance:

```
async function myFunction() {
  // your code goes in here
}
```

Any `await` not inside of an async function is a syntax error. For more information about `async` / `await`, read our introductory blog post (http://pouchdb.com/2015/03/05/taming-the-async-beast-with-es7.html).

Create a database (https://github.com/pouchdb/pouchdb/edit/master/do

```
new PouchDB([name], [options])
```

This method creates a database or opens an existing one. If you use a URL like `'http://domain.com/dbname'`, then PouchDB will work as a client to an online CouchDB instance. Otherwise it will create a local database using whatever backend is present (/adapters.html).

Options

- `name` : You can omit the `name` argument and specify it via `options` instead. Note that the name is required.

Options for local databases:

- `auto_compaction` : This turns on auto compaction, which means `compact()` is called after every change to the database. Defaults to `false`.
- `adapter` : One of `'idb'`, `'leveldb'`, or `'http'`.
- `revs_limit` : Specify how many old revisions we keep track (not a copy) of. Specifying a low value means Pouch may not be able to figure out whether a new revision received via replication is related to any it currently has which could result in a conflict. Defaults to `1000`.
- `deterministic_revs` : Use a md5 hash to create a deterministic revision number for documents. Setting it to false will mean that the revision number will be a random UUID. Defaults to true.

Options for remote databases:

- `fetch(url, opts)` : Intercept or override the HTTP request, you can add or modify any headers or options relating to the http request then return a new fetch Promise.
- `auth.username + auth.password` : You can specify HTTP auth parameters either by using a database with a name in the form `http://user:pass@host/name` or via the `auth.username + auth.password` options.
- `skip_setup` : Initially PouchDB checks if the database exists, and tries to create it, if it does not exist yet. Set this to `true` to skip this setup.

Notes:

1. In IndexedDB PouchDB will use `_pouch_` to prefix the internal database names. Do not manually create databases with the same prefix.
2. When acting as a client on Node, any other options given will be passed to request (https://github.com/mikeal/request).
3. When using the `'leveldb'` adapter (the default on Node), any other options given will be passed to levelup (https://github.com/rvagg/node-levelup).

Example Usage:

```
var db = new PouchDB('dbname');
// or
var db = new PouchDB('http://localhost:5984/dbname');
```

Create an in-memory Pouch (must install `pouchdb-adapter-memory` first):

```
var db = new PouchDB('dbname', {adapter: 'memory'});
```

Create a remote PouchDB with special fetch options:

```
var db = new PouchDB('http://example.com/dbname', {
  fetch: function (url, opts) {
    opts.headers.set('X-Some-Special-Header', 'foo');
    return PouchDB.fetch(url, opts);
  }
});
```

For more info, check out adapters (/adapters.html).

Delete a database (https://github.com/pouchdb/pouchdb/edit/master/do

```
db.destroy([options], [callback])
```

Delete the database. Note that this has no impact on other replicated databases.

Example Usage

Callbacks Promises Async functions

```
db.destroy().then(function (response) {
  // success
}).catch(function (err) {
  console.log(err);
});
```

Example Response:

```
{
  "ok" : true
}
```

Create/update a document (https://github.com/pouchdb/pouchdb/edit/m

Using db.put()

```
db.put(doc, [options], [callback])
```

Create a new document or update an existing document. If the document already exists, you must specify its revision `_rev` , otherwise a conflict will occur.

If you want to update an existing document even if there's conflict, you should specify the base revision `_rev` and use `force=true` option, then a new conflict revision will be created.

There are some restrictions on valid property names of the documents (http://wiki.apache.org/couchdb/HTTP_Document_API#Special_Fields). If you try to store non-JSON data (for instance `Date` objects) you may see inconsistent results (http://pouchdb.com/errors.html#could_not_be_cloned).

Example Usage:

Create a new doc with an `_id` of `'mydoc'` :

Callbacks Promises Async functions

```
db.put({
  _id: 'mydoc',
  title: 'Heroes'
}).then(function (response) {
  // handle response
}).catch(function (err) {
  console.log(err);
});
```

You can update an existing doc using `_rev` :

Callbacks Promises Async functions

```
db.get('mydoc').then(function(doc) {
  return db.put({
    _id: 'mydoc',
    _rev: doc._rev,
    title: "Let's Dance"
  });
}).then(function(response) {
  // handle response
}).catch(function (err) {
  console.log(err);
});
```

Example Response:

```
{
  "ok": true,
  "id": "mydoc",
  "rev": "1-A6157A5EA545C99B00FF904EEF05FD9F"
}
```

The response contains the `id` of the document, the new `rev` , and an `ok` to reassure you that everything is okay.

Using db.post()

```
db.post(doc, [options], [callback])
```

Create a new document and let PouchDB auto-generate an `_id` for it.

Example Usage:

Callbacks Promises Async functions

```
db.post({
  title: 'Ziggy Stardust'
}).then(function (response) {
  // handle response
}).catch(function (err) {
  console.log(err);
});
```

Example Response:

```
{
  "ok" : true,
  "id" : "8A2C3761-FFD5-4770-9B8C-38C33CED300A",
  "rev" : "1-d3a8e0e5aa7c8fff0c376dac2d8a4007"
}
```

Put vs. post: The basic rule of thumb is: `put()` new documents with an `_id`, `post()` new documents without an `_id`.

You should also prefer `put()` to `post()`, because when you `post()`, you are missing an opportunity to use `allDocs()` to sort documents by `_id` (because your `_id`s are random). For more info, read the PouchDB pro tips (/2014/06/17/12-pro-tips-for-better-code-with-pouchdb.html).

Fetch a document (https://github.com/pouchdb/pouchdb/edit/master/do

```
db.get(docId, [options], [callback])
```

Retrieves a document, specified by `docId`.

Options

All options default to `false` unless otherwise specified.

- `options.rev` : Fetch specific revision of a document. Defaults to winning revision (see the CouchDB guide (http://guide.couchdb.org/draft/conflicts.html)).
- `options.revs` : Include revision history of the document.
- `options.revs_info` : Include a list of revisions of the document, and their availability.
- `options.open_revs` : Fetch all leaf revisions if `open_revs="all"` or fetch all leaf revisions specified in `open_revs` array. Leaves will be returned in the same order as specified in input array.
- `options.conflicts` : If specified, conflicting leaf revisions will be attached in `_conflicts` array.
- `options.attachments` : Include attachment data.
  - `options.binary` : Return attachment data as Blobs/Buffers, instead of as base64-encoded strings.
- `options.latest` : Forces retrieving latest "leaf" revision, no matter what rev was requested. Default is `false`.

Example Usage:

Callbacks Promises Async functions

```
db.get('mydoc').then(function (doc) {
  // handle doc
}).catch(function (err) {
  console.log(err);
});
```

Example Response:

```
{
  "_id": "mydoc",
  "_rev": "1-A6157A5EA545C99B00FF904EEF05FD9F"
  "title": "Rock and Roll Heart",
}
```

The response contains the document as it is stored in the database, along with its `_id` and `_rev`.

Delete a document (https://github.com/pouchdb/pouchdb/edit/master/do

```
db.remove(doc, [options], [callback])
```

Or:

```
db.remove(docId, docRev, [options], [callback])
```

Deletes the document. `doc` is required to be a document with at least an `_id` and a `_rev` property. Sending the full document will work as well.

See filtered replication (http://pouchdb.com/api.html#filtered-replication) for why you might want to use `put()` with `{_deleted: true}` instead.

Example Usage:

Callbacks Promises Async functions

```
db.get('mydoc').then(function(doc) {
  return db.remove(doc);
}).then(function (result) {
  // handle result
}).catch(function (err) {
  console.log(err);
});
```

Example Response:

```
{
  "ok": true,
  "id": "mydoc",
  "rev": "2-9AF304BE281790604D1D8A4B0F4C9ADB"
}
```

You can also delete a document by just providing an `id` and `rev`:

Callbacks Promises Async functions

```
db.get('mydoc').then(function(doc) {
  return db.remove(doc._id, doc._rev);
}).then(function (result) {
  // handle result
}).catch(function (err) {
  console.log(err);
});
```

You can also delete a document by using `put()` with `{_deleted: true}`:

Callbacks Promises Async functions

```
db.get('mydoc').then(function(doc) {
  doc._deleted = true;
  return db.put(doc);
}).then(function (result) {
  // handle result
}).catch(function (err) {
  console.log(err);
});
```

Create/update a batch of documents (https://github.com/pouchdb/pouch

```
db.bulkDocs(docs, [options], [callback])
```

Create, update or delete multiple documents. The `docs` argument is an array of documents.

If you omit an `_id` parameter on a given document, the database will create a new document and assign the ID for you. To update a document, you must include both an `_id` parameter and a `_rev` parameter, which should match the ID and revision of the document on which to base your updates. Finally, to delete a document, include a `_deleted` parameter with the value `true`.

Example Usage:

Put some new docs, providing the `_id` s:

Callbacks Promises Async functions

```
db.bulkDocs([
  {title : 'Lisa Says', _id: 'doc1'},
  {title : 'Space Oddity', _id: 'doc2'}
]).then(function (result) {
  // handle result
}).catch(function (err) {
  console.log(err);
});
```

Post some new docs and auto-generate the `_id` s:

Callbacks Promises Async functions

```
db.bulkDocs([
  {title : 'Lisa Says'},
  {title : 'Space Oddity'}
]).then(function (result) {
  // handle result
}).catch(function (err) {
  console.log(err);
});
```

Example Response:

```
[
  {
    "ok": true,
    "id": "doc1",
    "rev": "1-84abc2a942007bee7cf55007cba56198"
  },
  {
    "ok": true,
    "id": "doc2",
    "rev": "1-7b80fc50b6af7a905f368670429a757e"
  }
]
```

The response contains an array of the familiar `ok` / `rev` / `id` from the put()/post() API. If there are any errors, they will be provided individually like so:

```
[
  { status: 409,
    name: 'conflict',
    message: 'Document update conflict',
    error: true
  }
]
```

The results are returned in the same order as the supplied “docs” array.

Note that `bulkDocs()` is not transactional, and that you may get back a mixed array of errors/non-errors. In CouchDB/PouchDB, the smallest atomic unit is the document.

Bulk update/delete:

You can also use `bulkDocs()` to update/delete many documents at once:

Callbacks Promises Async functions

```
db.bulkDocs([
  {
    title  : 'Lisa Says',
    artist : 'Velvet Underground',
    _id    : "doc1",
    _rev   : "1-84abc2a942007bee7cf55007cba56198"
  },
  {
    title  : 'Space Oddity',
    artist : 'David Bowie',
    _id    : "doc2",
    _rev   : "1-7b80fc50b6af7a905f368670429a757e"
  }
]).then(function (result) {
  // handle result
}).catch(function (err) {
  console.log(err);
});
```

Or delete them:

Callbacks | Promises | Async functions

```
db.bulkDocs([
  {
    title    : 'Lisa Says',
    _deleted : true,
    _id      : "doc1",
    _rev     : "1-84abc2a942007bee7cf55007cba56198"
  },
  {
    title    : 'Space Oddity',
    _deleted : true,
    _id      : "doc2",
    _rev     : "1-7b80fc50b6af7a905f368670429a757e"
  }
]).then(function (result) {
  // handle result
}).catch(function (err) {
  console.log(err);
});
```

Note: You can also specify a `new_edits` property on the options object that when set to `false` allows you to post existing documents from other databases (http://wiki.apache.org/couchdb/HTTP_Bulk_Document_API#Posting_Existing_Revisions). This will not allow you to edit existing local documents and normally only the replication algorithm needs to do this.

Fetch a batch of documents (https://github.com/pouchdb/pouchdb/edit/m

```
db.allDocs([options], [callback])
```

Fetch multiple documents, indexed and sorted by the `_id`. Deleted documents are only included if `options.keys` is specified.

Options

All options default to `false` unless otherwise specified.

- `options.include_docs` : Include the document itself in each row in the `doc` field. Otherwise by default you only get the `_id` and `_rev` properties.
- `options.conflicts` : Include conflict information in the `_conflicts` field of a doc.
- `options.attachments` : Include attachment data as base64-encoded string.
- `options.binary` : Return attachment data as Blobs/Buffers, instead of as base64-encoded strings.
- `options.startkey` & `options.endkey` : Get documents with IDs in a certain range (inclusive/inclusive).
- `options.inclusive_end` : Include documents having an ID equal to the given `options.endkey`. Default: `true`.
- `options.limit` : Maximum number of documents to return.
- `options.skip` : Number of docs to skip before returning (warning: poor performance on IndexedDB/LevelDB!).
- `options.descending` : Reverse the order of the output documents. Note that the order of `startkey` and `endkey` is reversed when `descending`:`true`.

- `options.key` : Only return documents with IDs matching this string key.
- `options.keys` : Array of string keys to fetch in a single shot.
  - Neither `startkey` nor `endkey` can be specified with this option.
  - The rows are returned in the same order as the supplied `keys` array.
  - The row for a deleted document will have the revision ID of the deletion, and an extra key `"deleted":true` in the `value` property.
  - The row for a nonexistent document will just contain an `"error"` property with the value `"not_found"` .
  - For details, see the CouchDB query options documentation (http://wiki.apache.org/couchdb/HTTP_view_API#Querying_Options).
- `options.update_seq` : Include an `update_seq` value indicating which sequence id of the underlying database the view reflects.

Notes: For pagination, `options.limit` and `options.skip` are also available, but the same performance concerns as in CouchDB apply. Use the startkey/endkey pattern (http://docs.couchdb.org/en/latest/couchapp/views/pagination.html) instead.

Example Usage:

Callbacks Promises Async functions

```
db.allDocs({
  include_docs: true,
  attachments: true
}).then(function (result) {
  // handle result
}).catch(function (err) {
  console.log(err);
});
```

Example Response:

```
{
  "offset": 0,
  "total_rows": 1,
  "rows": [{
    "doc": {
      "_id": "0B3358C1-BA4B-4186-8795-9024203EB7DD",
      "_rev": "1-5782E71F1E4BF698FA3793D9D5A96393",
      "title": "Sound and Vision",
      "_attachments": {
        "attachment/its-id": {
          "content_type": "image/jpg",
          "data": "R0lGODlhAQABAIAAAP7//wAAACH5BAAAAAAALAAAAAABAAEAAAICRAEAOw==",
          "digest": "md5-57e396baedfe1a034590339082b9abce"
        }
      }
    },
   "id": "0B3358C1-BA4B-4186-8795-9024203EB7DD",
   "key": "0B3358C1-BA4B-4186-8795-9024203EB7DD",
   "value": {
    "rev": "1-5782E71F1E4BF698FA3793D9D5A96393"
   }
 }]
}
```

In the response, you have three things:

- `total_rows` the total number of non-deleted documents in the database
- `offset` the `skip` if provided, or in CouchDB the actual offset
- `rows` : rows containing the documents, or just the `_id` / `_revs` if you didn't set `include_docs` to `true` .

- You may optionally also have `update_seq` if you set `update_seq` to `true`

You can use `startkey` / `endkey` to find all docs in a range:

Callbacks Promises Async functions

```
db.allDocs({
  include_docs: true,
  attachments: true,
  startkey: 'bar',
  endkey: 'quux'
}).then(function (result) {
  // handle result
}).catch(function (err) {
  console.log(err);
});
```

This will return all docs with `_id` s between `'bar'` and `'quux'` .

Prefix search

You can do prefix search in `allDocs()` – i.e. "give me all the documents whose `_id` s start with `'foo'` " – by using the special high Unicode character `'\ufff0'` :

Callbacks Promises Async functions

```
db.allDocs({
  include_docs: true,
  attachments: true,
  startkey: 'foo',
  endkey: 'foo\ufff0'
}).then(function (result) {
  // handle result
}).catch(function (err) {
  console.log(err);
});
```

This works because CouchDB/PouchDB `_id` s are sorted lexicographically (http://docs.couchdb.org/en/latest/couchapp/views/collation.html).

Listen to database changes (https://github.com/pouchdb/pouchdb/edit/m

```
db.changes(options)
```

A list of changes made to documents in the database, in the order they were made. It returns an object with the method `cancel()` , which you call if you don't want to listen to new changes anymore.

It is an event emitter (http://nodejs.org/api/events.html#events_class_events_eventemitter) and will emit a `'change'` event on each document change, a `'complete'` event when all the changes have been processed, and an `'error'` event when an error occurs. Calling `cancel()` will unsubscribe all event listeners automatically.

Options

All options default to `false` unless otherwise specified.

- `options.live` : Will emit change events for all future changes until cancelled.
- `options.include_docs` : Include the associated document with each change.
  - `options.conflicts` : Include conflicts.
  - `options.attachments` : Include attachments.
    - `options.binary` : Return attachment data as Blobs/Buffers, instead of as base64-encoded strings.
- `options.descending` : Reverse the order of the output documents.
- `options.since` : Start the results from the change immediately after the given sequence number. You can also pass `'now'` if you want only new changes (when `live` is `true`).
- `options.limit` : Limit the number of results to this number.
- `options.timeout` : Request timeout (in milliseconds), use `false` to disable.
- `options.heartbeat` : For http adapter only, time in milliseconds for server to give a heartbeat to keep long connections open. Defaults to 10000 (10 seconds), use `false` to disable the default.

Filtering Options:

- `options.filter` : Reference a filter function from a design document to selectively get updates. To use a view function, pass `_view` here and provide a reference to the view function in `options.view` . See filtered changes for details.
- `options.doc_ids` : Only show changes for docs with these ids (array of strings).

- `options.query_params` : Object containing properties that are passed to the filter function, e.g. `{"foo:"bar"}` , where `"bar"` will be available in the filter function as `params.query.foo` . To access the `params` , define your filter function like `function (doc, params) {/* ... */}` .
- `options.view` : Specify a view function (e.g. `'design_doc_name/view_name'` or `'view_name'` as shorthand for `'view_name/view_name'`) to act as a filter. Documents counted as "passed" for a view filter if a map function emits at least one record for them. **Note**: `options.filter` must be set to `'_view'` for this option to work.
- `options.selector` : Filter using a query/pouchdb-find selector (http://docs.couchdb.org/en/2.0.0/api/database/find.html#selector-syntax). **Note**: Selectors are not supported in CouchDB 1.x. Cannot be used in combination with the filter option.

Advanced Options:

- `options.return_docs` : Defaults to `true` except when `opts.live = true` then it defaults to `false` . Passing `false` prevents the changes feed from keeping all the documents in memory – in other words complete always has an empty results array, and the `change` event is the only way to get the event. Useful for large change sets where otherwise you would run out of memory.
- `options.batch_size` : Only available for http databases, this configures how many changes to fetch at a time. Increasing this can reduce the number of requests made. Default is 25.
- `options.style` : Specifies how many revisions are returned in the changes array. The default, `'main_only'` , will only return the current "winning" revision; `'all_docs'` will return all leaf revisions (including conflicts and deleted former conflicts). Most likely you won't need this unless you're writing a replicator.
- `options.seq_interval` : Only available for http databases. Specifies that seq information only be generated every N changes. Larger values can improve changes throughput with CouchDB 2.0 and later. Note that `last_seq` is always populated regardless.

Example Usage:

```
var changes = db.changes({
  since: 'now',
  live: true,
  include_docs: true
}).on('change', function(change) {
  // handle change
}).on('complete', function(info) {
  // changes() was canceled
}).on('error', function (err) {
  console.log(err);
});

changes.cancel(); // whenever you want to cancel
```

Example Response:

```
{
  "id":"somestuff",
  "seq":21,
  "changes":[{
    "rev":"1-8e6e4c0beac3ec54b27d1df75c7183a8"
  }],
  "doc":{
    "title":"Ch-Ch-Ch-Ch-Changes",
    "_id":"someDocId",
    "_rev":"1-8e6e4c0beac3ec54b27d1df75c7183a8"
  }
}
```

Change events

- **`change`** (`info`) - This event fires when a change has been found. `info` will contain details about the change, such as whether it was deleted and what the new `_rev` is. `info.doc` will contain the doc if you set `include_docs` to `true` . See below for an example response.
- **`complete`** (`info`) - This event fires when all changes have been read. In live changes, only cancelling the changes should trigger this event. `info.results` will contain the list of changes. See below for an example.
- **`error`** (`err`) - This event is fired when the changes feed is stopped due to an unrecoverable failure.

Example response

Example response in the `'change'` listener (using `{include_docs: true}`):

```
{ id: 'doc1',
  changes: [ { rev: '1-9152679630cc461b9477792d93b83eae' } ],
  doc: {
    _id: 'doc1',
    _rev: '1-9152679630cc461b9477792d93b83eae'
  },
  seq: 1
}
```

Example response in the `'change'` listener when a doc was deleted:

```
{ id: 'doc2',
  changes: [ { rev: '2-9b50a4b63008378e8d0718a9ad05c7af' } ],
  doc: { _id: 'doc2',
    _rev: '2-9b50a4b63008378e8d0718a9ad05c7af',
    _deleted: true
  },
  deleted: true,
  seq: 3
}
```

Example response in the `'complete'` listener:

```
{
  "results": [
    {
      "id": "doc1",
      "changes": [ { "rev": "1-9152679630cc461b9477792d93b83eae" } ],
      "doc": {
        "_id": "doc1",
        "_rev": "1-9152679630cc461b9477792d93b83eae"
      },
      "seq": 1
    },
    {
      "id": "doc2",
      "changes": [ { "rev": "2-9b50a4b63008378e8d0718a9ad05c7af" } ],
      "doc": {
        "_id": "doc2",
        "_rev": "2-9b50a4b63008378e8d0718a9ad05c7af",
        "_deleted": true
      },
      "deleted": true,
      "seq": 3
    }
  ],
  "last_seq": 3
}
```

`seq` and `last_seq` correspond to the overall sequence number of the entire database, and it's what is passed in when using `since` (except for the special `'now'`). It is the primary key for the changes feed, and is also used as a checkpointer by the replication algorithm.

Single-shot

If you don't specify `{live: true}`, then you can also use `changes()` in the standard callback/promise style, and it will be treated as a single-shot request, which returns a list of the changes (i.e. what the `'complete'` event emits):

Callbacks Promises Async functions

```
db.changes({
  limit: 10,
  since: 0
}).then(function (result) {
  // handle result
}).catch(function (err) {
  console.log(err);
});
```

Example Response:

```
{
  "results": [{
    "id": "0B3358C1-BA4B-4186-8795-9024203EB7DD",
    "seq": 1,
    "changes": [{
      "rev": "1-5782E71F1E4BF698FA3793D9D5A96393"
    }]
  }, {
    "id": "mydoc",
    "seq": 2,
    "changes": [{
      "rev": "1-A6157A5EA545C99B00FF904EEF05FD9F"
    }]
  }, {
    "id": "otherdoc",
    "seq": 3,
    "changes": [{
      "rev": "1-3753476B70A49EA4D8C9039E7B04254C"
    }]
  }, {
    "id": "828124B9-3973-4AF3-9DFD-A94CE4544005",
    "seq": 4,
    "changes": [{
      "rev": "1-A8BC08745E62E58830CA066D99E5F457"
    }]
  }],
  "last_seq": 4
}
```

When `live` is `false`, the returned object is also an event emitter as well as a promise, and will fire the `'complete'` event when the results are ready.

Note that this `'complete'` event only fires when you aren't doing live changes.

Filtered changes

As with replicate(), you can filter using:

- an ad-hoc `filter` function
- an array of `doc_ids`
- a `filter` function inside of a design document
- a `filter` function inside of a design document, with `query_params`
- a `view` function inside of a design document

If you are running `changes()` on a remote CouchDB, then the first method will run client-side, whereas the last four will filter on the server side. Therefore the last four should be preferred, especially if the database is large, because you want to send as few documents over the wire as possible.

If you are running `changes()` on a local PouchDB, then obviously all five methods will run client-side. There are also no performance benefits to using any of the five, so can also just filter yourself, in your own `on('change')` handler. These methods are implemented in PouchDB purely for consistency with CouchDB.

The named functions can either be specified with `'designdoc_id/function_name'` or (if both design doc id and function name are equal) as `'fname'` as shorthand for `'fname/fname'`.

Filtering examples

In these examples, we'll work with some mammals. Let's imagine our docs are:

```
[
  {_id: 'a', name: 'Kangaroo', type: 'marsupial'},
  {_id: 'b', name: 'Koala', type: 'marsupial'},
  {_id: 'c', name: 'Platypus', type: 'monotreme'}
]
```

Here are 5 examples using the 5 different systems.

Example 1: Ad-hoc `filter` function

Warning: this runs client-side, if the database is remote.

Filter by `type === 'marsupial'`:

```
db.changes({
  filter: function (doc) {
    return doc.type === 'marsupial';
  }
});
```

Example 2: Array of `doc_ids`

Filter documents with `_id`s `['a', 'c']`.

```
db.changes({
  doc_ids: ['a', 'c']
});
```

Example 3: `filter` function inside of a design document

First `put()` a design document:

```
{
  _id: '_design/mydesign',
  filters: {
    myfilter: function (doc) {
      return doc.type === 'marsupial';
    }.toString()
  }
}
```

Then filter by `type === 'marsupial'`:

```
db.changes({
  filter: 'mydesign/myfilter'
});
```

Example 4: `filter` function inside of a design document, with `query_params`

This is the most powerful way to filter, because it allows you to pass in arbitrary options to your filter function.

First `put()` a design document:

```
{
  _id: '_design/myfilter',
  filters: {
    myfilter: function (doc, req) {
      return doc.type === req.query.type;
    }.toString()
  }
}
```

Then filter by `type === 'marsupial'`:

```
db.changes({
  filter: 'myfilter',
  query_params: {type: 'marsupial'}
});
```

Since both the design document and the filter function have the same name, we can shorten the function name to `'myfilter'`.

Example 5: `view` function inside of a design document

This doesn't really offer any advantages compared to the previous two methods, unless you are already using a `view` for map/reduce queries, and you want to reuse it.

Any documents that `emit()` anything will be considered to have passed this filter method.

First `put()` a design document:

```
{
  _id: '_design/mydesign',
  views: {
    myview: function (doc) {
      if (doc.type === 'marsupial') {
        emit(doc._id);
      }
    }.toString()
  }
}
```

Then filter by `type === 'marsupial'`:

```
db.changes({
  filter: '_view',
  view: 'mydesign/myview'
});
```

Replicate a database (https://github.com/pouchdb/pouchdb/edit/master/

```
PouchDB.replicate(source, target, [options])
```

Replicate data from `source` to `target`. Both the `source` and `target` can be a PouchDB instance or a string representing a CouchDB database URL or the name of a local PouchDB database. If `options.live` is `true`, then this will track future changes and also replicate them automatically. This method returns an object with the method `cancel()`, which you call if you want to cancel live replication.

Replication is an event emitter (http://nodejs.org/api/events.html#events_class_events_eventemitter) like changes() and emits the `'complete'`, `'active'`, `'paused'`, `'change'`, `'denied'` and `'error'` events.

Options

All options default to `false` unless otherwise specified.

- `options.live`: If `true`, starts subscribing to future changes in the `source` database and continue replicating them.
- `options.retry`: If `true` will attempt to retry replications in the case of failure (due to being offline), using a backoff algorithm that retries at longer and longer intervals until a connection is re-established, with a maximum delay of 10 minutes. Only applicable if `options.live` is also `true`.

Filtering Options:

- `options.filter`: Reference a filter function from a design document to selectively get updates. To use a view function, pass `_view` here and provide a reference to the view function in `options.view`. See filtered replication for details.
- `options.doc_ids`: Only show changes for docs with these ids (array of strings).
- `options.query_params`: Object containing properties that are passed to the filter function, e.g. `{"foo":"bar"}`, where `"bar"` will be available in the filter function as `params.query.foo`. To access the `params`, define your filter function like `function (doc, params) {/* ... */}`.
- `options.view`: Specify a view function (e.g. `'design_doc_name/view_name'` or `'view_name'` as shorthand for `'view_name/view_name'`) to act as a filter. Documents counted as "passed" for a view filter if a map function emits at least one record for them. **Note**: `options.filter` must be set to `'_view'` for this option to work.
- `options.selector`: Filter using a query/pouchdb-find selector (http://docs.couchdb.org/en/2.0.0/api/database/find.html#selector-syntax). **Note**: Selectors are not supported in CouchDB 1.x.

Advanced Options:

- `options.since`: Replicate changes after the given sequence number.
- `options.heartbeat`: Configure the heartbeat supported by CouchDB which keeps the change connection alive.
- `options.timeout`: Request timeout (in milliseconds).
- `options.batch_size`: Number of change feed items to process at a time. Defaults to 100. This affects the number of docs and attachments held in memory and the number sent at a time to the target server. You may need to adjust downward if targeting devices with low amounts of memory (e.g. phones) or if the documents and/or attachments are large in size or if there are many conflicted revisions. If your documents are small in size, then increasing this number will probably speed replication up.
- `options.batches_limit`: Number of batches to process at a time. Defaults to 10. This (along wtih `batch_size`) controls how many docs are kept in memory at a time, so the maximum docs in memory at once would equal `batch_size × batches_limit`.
- `options.back_off_function`: backoff function to be used in `retry` replication. This is a function that takes the current backoff as input (or 0 the first time) and returns a new backoff in milliseconds. You can use this to tweak when and how replication will try to reconnect to a remote database when the user goes offline. Defaults to a function that chooses a random backoff between 0 and 2

seconds and doubles every time it fails to connect. The default delay will never exceed 10 minutes. (See Customizing retry replication below.)

- `options.checkpoint` : Can be used if you want to disable checkpoints on the source, target, or both. Setting this option to `false` will prevent writing checkpoints on both source and target. Setting it to `source` will only write checkpoints on the source. Setting it to `target` will only write checkpoints on the target.

Example Usage:

```
var rep = PouchDB.replicate('mydb', 'http://localhost:5984/mydb', {
  live: true,
  retry: true
}).on('change', function (info) {
  // handle change
}).on('paused', function (err) {
  // replication paused (e.g. replication up to date, user went offline)
}).on('active', function () {
  // replicate resumed (e.g. new changes replicating, user went back online)
}).on('denied', function (err) {
  // a document failed to replicate (e.g. due to permissions)
}).on('complete', function (info) {
  // handle complete
}).on('error', function (err) {
  // handle error
});

rep.cancel(); // whenever you want to cancel
```

There are also shorthands for replication given existing PouchDB objects. These behave the same as `PouchDB.replicate()` :

```
db.replicate.to(remoteDB, [options]);
// or
db.replicate.from(remoteDB, [options]);
```

The `remoteDB` can either be a string or a `PouchDB` object. If you have a fetch override on a remote database, you will want to use `PouchDB` objects instead of strings, so that the options are used.

Replication events

- **`change`** (`info`) - This event fires when the replication has written a new document. `info` will contain details about the change. `info.docs` will contain the docs involved in that change. See below for an example response.
- **`complete`** (`info`) - This event fires when replication is completed or cancelled. In a live replication, only cancelling the replication should trigger this event. `info` will contain details about the replication. See below for an example response.
- **`paused`** (`err`) - This event fires when the replication is paused, either because a live replication is waiting for changes, or replication has temporarily failed, with `err` , and is attempting to resume.
- **`active`** - This event fires when the replication starts actively processing changes; e.g. when it recovers from an error or new changes are available.
- **`denied`** (`err`) - This event fires if a document failed to replicate due to validation or authorization errors.
- **`error`** (`err`) - This event is fired when the replication is stopped due to an unrecoverable failure. If `retry` is `false` , this will also fire when the user goes offline or another network error occurs (so you can handle retries yourself, if you want).

Single-shot

As with changes(), you can also omit `live` , in which case you can use `replicate()` in the callback/promise style and it will be treated as a single-shot operation.

Callbacks | Promises | Async functions

```
db.replicate.to(remote).then(function (result) {
  // handle 'completed' result
}).catch(function (err) {
  console.log(err);
});
```

For non-live replications, the returned object is also an event emitter as well as a promise, and you can use all the events described above (except for `'paused'` and `'active'` , which only apply to `retry` replications).

Example Response:

Example response in the `'change'` listener:

```
{
  "doc_write_failures": 0,
  "docs_read": 1,
  "docs_written": 1,
  "errors": [],
  "last_seq": 1,
  "ok": true,
  "start_time": "Fri May 16 2014 18:23:12 GMT-0700 (PDT)",
  "docs": [
    { _id: 'docId',
      _rev: '1-e798a1a7eb4247b399dbfec84ca699d4',
      and: 'data' }
  ]
}
```

Example response in the `'complete'` listener:

```
{
  "doc_write_failures": 0,
  "docs_read": 2,
  "docs_written": 2,
  "end_time": "Fri May 16 2014 18:26:00 GMT-0700 (PDT)",
  "errors": [],
  "last_seq": 2,
  "ok": true,
  "start_time": "Fri May 16 2014 18:26:00 GMT-0700 (PDT)",
  "status": "complete"
}
```

Note that replication is supported for both local and remote databases. So you can replicate from local to local or from remote to remote.

However, if you replicate from remote to remote, then the changes will flow through PouchDB. If you want to trigger a server-initiated replication, please use regular ajax to POST to the CouchDB `_replicate` endpoint, as described in the CouchDB docs (https://wiki.apache.org/couchdb/Replication).

Filtered replication

As with changes(), you can filter from the source database using:

- an ad-hoc `filter` function
- an array of `doc_ids`
- a `filter` function inside of a design document
- a `filter` function inside of a design document, with `query_params`
- a `view` function inside of a design document

If you are replicating from a remote CouchDB, then the first method will run client-side, whereas the last four will filter on the server side. Therefore the last four should be preferred, especially if the database is large, because you want to send as few documents over the wire as possible.

You should also beware trying to use filtered replication to enforce security, e.g. to partition a database per user. A better strategy is the “one database per user” method (https://github.com/nolanlawson/pouchdb-authentication#couchdb-authentication-recipes).

! **Deleting filtered docs**: When you use filtered replication, you should avoid using `remove()` to delete documents, because that removes all their fields as well, which means they might not pass the filter function anymore, causing the deleted revision to not be replicated. Instead, set the `doc._deleted` flag to `true` and then use `put()` or `bulkDocs()`.

Filtering examples

In these examples, we’ll work with some mammals. Let’s imagine our docs are:

```
[
  {_id: 'a', name: 'Kangaroo', type: 'marsupial'},
  {_id: 'b', name: 'Koala', type: 'marsupial'},
  {_id: 'c', name: 'Platypus', type: 'monotreme'}
]
```

Here are 5 examples using the 5 different systems.

Example 1: Ad-hoc `filter` function

Warning: this runs client-side, if you are replicating from a remote database.

Filter by `type === 'marsupial'`:

```
remote.replicate.to(local, {
  filter: function (doc) {
    return doc.type === 'marsupial';
  }
});
```

Example 2: Array of `doc_ids`

Filter documents with `_id`s `['a', 'c']`.

```
remote.replicate.to(local, {
  doc_ids: ['a', 'c']
});
```

Example 3: `filter` function inside of a design document

First `put()` a design document in the remote database:

```
{
  _id: '_design/mydesign',
  filters: {
    myfilter: function (doc) {
      return doc.type === 'marsupial';
    }.toString()
  }
}
```

Then filter by `type === 'marsupial'`:

```
remote.replicate.to(local, {
  filter: 'mydesign/myfilter'
});
```

Example 4: `filter` function inside of a design document, with `query_params`

This is the most powerful way to filter, because it allows you to pass in arbitrary options to your filter function.

First `put()` a design document in the remote database:

```
{
  _id: '_design/mydesign',
  filters: {
    myfilter: function (doc, req) {
      return doc.type === req.query.type;
    }.toString()
  }
}
```

Then filter by `type === 'marsupial'`:

```
remote.replicate.to(local, {
  filter: 'mydesign/myfilter',
  query_params: {type: 'marsupial'}
});
```

Example 5: `view` function inside of a design document

This doesn't really offer any advantages compared to the previous two methods, unless you are already using a `view` for map/reduce queries, and you want to reuse it.

Any documents that `emit()` anything will be considered to have passed this filter method.

First `put()` a design document in the remote database:

```
{
  _id: '_design/mydesign',
  views: {
    myview: {
      map: function(doc) {
        if (doc.type === 'marsupial') {
          emit(doc._id);
        }
      }.toString()
    }
  }
}
```

Then filter by `type === 'marsupial'`:

```
remote.replicate.to(local, {
  filter: '_view',
  view: 'mydesign/myview'
});
```

Customizing retry replication

During `retry` replication, you can customize the backoff function that determines how long to wait before reconnecting when the user goes offline.

Here's a simple backoff function that starts at 1000 milliseconds and triples it every time a remote request fails:

```
db.replicate.to(remote, {
  live: true,
  retry: true,
  back_off_function: function (delay) {
    if (delay === 0) {
      return 1000;
    }
    return delay * 3;
  }
});
```

The first time a request fails, this function will receive 0 as input. The next time it fails, 1000 will be passed in, then 3000, then 9000, etc. When the user comes back online, the `delay` goes back to 0.

By default, PouchDB uses a backoff function that chooses a random starting number between 0 and 2000 milliseconds and will roughly double every time, with some randomness to prevent client requests from occurring simultaneously.

Sync a database (https://github.com/pouchdb/pouchdb/edit/master/docs

```
var sync = PouchDB.sync(src, target, [options])
```

Sync data from `src` to `target` and `target` to `src`. This is a convenience method for bidirectional data replication.

In other words, this code:

```
PouchDB.replicate('mydb', 'http://localhost:5984/mydb');
PouchDB.replicate('http://localhost:5984/mydb', 'mydb');
```

is equivalent to this code:

```
PouchDB.sync('mydb', 'http://localhost:5984/mydb');
```

Options

- `options.push + options.pull`: Allows you to specify separate replication options (api.html#replication) for the individual replications.

Replication options such as `filter` passed to sync directly will be passed to both replications. Please refer to replicate() (api.html#replication) for documentation on those options.

Example Usage:

```
var sync = PouchDB.sync('mydb', 'http://localhost:5984/mydb', {
  live: true,
  retry: true
}).on('change', function (info) {
  // handle change
}).on('paused', function (err) {
  // replication paused (e.g. replication up to date, user went offline)
}).on('active', function () {
  // replicate resumed (e.g. new changes replicating, user went back online)
}).on('denied', function (err) {
  // a document failed to replicate (e.g. due to permissions)
}).on('complete', function (info) {
  // handle complete
}).on('error', function (err) {
  // handle error
});

sync.cancel(); // whenever you want to cancel
```

There is also a shorthand for syncing given existing PouchDB objects. This behaves the same as `PouchDB.sync()`:

```
db.sync(remoteDB, [options]);
```

It is also possible to combine “one-way” replication and sync for performance reasons. When your PouchDB application starts up it could perform a one-off, one-way replication to completion and then initiate the two-way, continuous retryable sync:

```
var url = 'http://localhost:5984/mydb';
var opts = { live: true, retry: true };

// do one way, one-off sync from the server until completion
db.replicate.from(url).on('complete', function(info) {
  // then two-way, continuous, retriable sync
  db.sync(url, opts)
    .on('change', onSyncChange)
    .on('paused', onSyncPaused)
    .on('error', onSyncError);
}).on('error', onSyncError);
```

The above technique results in fewer HTTP requests being used and better performance than just using `db.sync` on its own.

Example Response:

Change events in `sync` have an extra property `direction` which refers to the direction the change was going. Its value will either be `push` or `pull`.

```
{ direction: 'push',
  change:
   { ok: true,
     start_time: '2015-10-21T15:26:51.151Z',
     docs_read: 1,
     docs_written: 1,
     doc_write_failures: 0,
     errors: [],
     last_seq: 1,
     docs: [ [Object] ] } }
```

For any further details, please refer to replicate() (api.html#replication).

Save an attachment (https://github.com/pouchdb/pouchdb/edit/master/d

```
db.putAttachment(docId, attachmentId, [rev], attachment, type, [callback]);
```

Attaches a binary object to a document.

This method will update an existing document to add the attachment, so it requires a `rev` if the document already exists. If the document doesn't already exist, then this method will create an empty document containing the attachment.

What's the point of attachments? If you're dealing with large binary data (such as PNGs), you may incur a performance or storage penalty if you naïvely include them as base64- or hex-encoded strings inside your documents. But if you insert the binary data as an attachment, then PouchDB will attempt to store it in the most efficient way possible (http://pouchdb.com/faq.html#data_types).

For details, see the CouchDB documentation on attachments (https://wiki.apache.org/couchdb/HTTP_Document_API#Attachments).

Example Usage:

Callbacks Promises Async functions

```
var attachment = new Blob(['Is there life on Mars?'], {type: 'text/plain'});
db.putAttachment('doc', 'att.txt', attachment, 'text/plain').then(function (result) {
  // handle result
}).catch(function (err) {
  console.log(err);
});
```

Example Response:

```
{
  "ok": true,
  "id": "doc",
  "rev": "2-068E73F5B44FEC987B51354DFC772891"
}
```

Within Node, you must use a `Buffer` instead of a `Blob`:

```
var attachment = new Buffer('Is there life on Mars?');
```

For details, see the Mozilla docs on `Blob` (https://developer.mozilla.org/en-US/docs/Web/API/Blob) or the Node docs on `Buffer` (http://nodejs.org/api/buffer.html).

If you need a shim for older browsers that don't support the `Blob` constructor, or you want some convenience methods for Blobs, you can use blob-util (https://github.com/nolanlawson/blob-util).

Save a base64 attachment

If you supply a string instead of a `Blob`/`Buffer`, then it will be assumed to be a base64-encoded string, and will be processed accordingly:

Callbacks Promises Async functions

```
var attachment =
        "TGVnZW5kYXJ5IGhlYXJ0cywgdGVhciB1cyBhbGwgYXBhcnQKTWFrZS" +
        "BvdXIgZW1vdGlvbnMgYmxlZWQsIGNyeWluZyBvdXQgaW4gbmVlZA==";
db.putAttachment('doc', 'att.txt', attachment, 'text/plain').then(function (result) {
  // handle result
}).catch(function (err) {
  console.log(err);
});
```

Save an inline attachment

You can also inline attachments inside the document. The attachment data may be supplied as a base64-encoded string with the `content_type`:

Callbacks Promises Async functions

```
var doc = {
  "_id": "doc",
  "title": "Legendary Hearts",
  "_attachments": {
    "att.txt": {
      "content_type": "text/plain",
      "data": "TGVnZW5kYXJ5IGhlYXJ0cywgdGVhciB1cyBhbGwgYXBhcnQKTWFrZS" +
              "BvdXIgZW1vdGlvbnMgYmxlZWQsIGNyeWluZyBvdXQgaW4gbmVlZA=="
    }
  }
};
db.put(doc).then(function (result) {
  // handle result
}).catch(function (err) {
  console.log(err);
});
```

Save an inline Blob/Buffer attachment

You can also inline `Blob` s/ `Buffer` s:

Callbacks Promises Async functions

```
var doc = {
  "_id": "doc",
  "title": "Legendary Hearts",
  "_attachments": {
    "att.txt": {
      "content_type": "text/plain",
      "data": new Blob(['Is there life on Mars?'], {type: 'text/plain'})
    }
  }
};
db.put(doc).then(function (result) {
  // handle result
}).catch(function (err) {
  console.log(err);
});
```

Save many attachments at once

The inline approach allows you to save multiple attachments to the same document in a single shot:

Callbacks Promises Async functions

```
var doc = {
  "_id": "doc",
  "title": "Legendary Hearts",
  "_attachments": {
    "att.txt": {
      "content_type": "text/plain",
      "data": new Blob(
        ["And she's hooked to the silver screen"],
        {type: 'text/plain'})
    },
    "att2.txt": {
      "content_type": "text/plain",
      "data": new Blob(
        ["But the film is a saddening bore"],
        {type: 'text/plain'})
    },
    "att3.txt": {
      "content_type": "text/plain",
      "data": new Blob(
        ["For she's lived it ten times or more"],
        {type: 'text/plain'})
    }
  }
};
db.put(doc).then(function (result) {
  // handle result
}).catch(function (err) {
  console.log(err);
});
```

See Inline Attachments (http://wiki.apache.org/couchdb/HTTP_Document_API#Inline_Attachments) on the CouchDB wiki for details.

Get an attachment (https://github.com/pouchdb/pouchdb/edit/master/dc

```
db.getAttachment(docId, attachmentId, [options], [callback])
```

Get attachment data.

Options

- `options.rev` : as with get(), you can pass a `rev` in and get back an attachment for the document at that particular revision.

Example Usage:

Get an attachment with filename `'att.txt'` from document with ID `'doc'` :

Callbacks Promises Async functions

```
db.getAttachment('doc', 'att.txt').then(function (blobOrBuffer) {
  // handle result
}).catch(function (err) {
  console.log(err);
});
```

Get an attachment with filename `'att.txt'` from document with ID `'doc'` , at the revision `'1-abcd'` :

Callbacks Promises Async functions

```
db.getAttachment('doc', 'att.txt', {rev: '1-abcd'}).then(function (blobOrBuffer) {
  // handle result
}).catch(function (err) {
  console.log(err);
});
```

Response type:

The response will be a `Blob` object in the browser, and a `Buffer` object in Node.js. See blob-util (https://github.com/nolanlawson/blob-util) for utilities to transform `Blob` s to other formats, such as base64-encoded strings, data URLs, array buffers, etc.

Inline base64 attachments

You can specify `{attachments: true}` to most "read" operations, such as `get()`, `allDocs()`, `changes()`, and `query()`. The attachment data will then be included inlined in the resulting doc(s). However, it will always be supplied as base64. For example:

```
{
  "_attachments": {
    "att.txt": {
      "content_type": "text/plain",
      "digest": "d5ccfd24a8748bed4e2c9a279a2b6089",
      "data": "SXMgdGhlcmUgbGlmZSBvbiBNYXJzPw=="
    }
  },
  "_id": "mydoc",
  "_rev": "1-e147d9ec9c85139dfe7e93bc17148d1a"
}
```

For such APIs, when you don't specify `{attachments: true}`, you will instead get metadata about the attachments. For example:

```
{
  "_attachments": {
    "att.txt": {
      "content_type": "text/plain",
      "digest": "d5ccfd24a8748bed4e2c9a279a2b6089",
      "stub": true
    }
  },
  "_id": "mydoc",
  "_rev": "1-e147d9ec9c85139dfe7e93bc17148d1a"
}
```

This "summary" operation may be faster in some cases, because the attachment itself does not need to be read from disk.

Delete an attachment (https://github.com/pouchdb/pouchdb/edit/master

```
db.removeAttachment(docId, attachmentId, rev, [callback])
```

Delete an attachment from a doc. You must supply the `rev` of the existing doc.

Example Usage:

Callbacks Promises Async functions

```
var rev = '1-068E73F5B44FEC987B51354DFC772891';
db.removeAttachment('doc', 'att.txt', rev).then(function (result) {
  // handle result
}).catch(function (err) {
  console.log(err);
});
```

Example Response:

```
{
  "ok": true,
  "rev": "2-1F983211AB87EFCCC980974DFC27382F"
}
```

Create index (https://github.com/pouchdb/pouchdb/edit/master/docs/_ir

```
db.createIndex(index [, callback])
```

Create an index if it doesn't exist, or do nothing if it already exists.

pouchdb-find plugin needed: This API requires the `pouchdb-find` plugin. See Mango queries (/guides/mango-queries.html) for installation instructions.

Example Usage:

Callbacks Promises Async functions

```
db.createIndex({
  index: {
    fields: ['foo']
  }
}).then(function (result) {
  // handle result
}).catch(function (err) {
  console.log(err);
});
```

Example Response:

If the index was created, you'll see:

```
{ "result": "created" }
```

Or if the index already exists:

```
{ "result": "exists" }
```

You can also create an index on multiple fields:

Callbacks Promises Async functions

```
db.createIndex({
  index: {
    fields: ['foo', 'bar', 'baz']
  }
}).then(function (result) {
  // handle result
}).catch(function (err) {
  console.log(err);
});
```

Or an index on deep fields:

Callbacks Promises Async functions

```
db.createIndex({
  index: {
    fields: ['person.address.zipcode']
  }
}).then(function (result) {
  // handle result
}).catch(function (err) {
  console.log(err);
});
```

You can also specify additional options, if you want more control over how your index is created:

Callbacks Promises Async functions

```
db.createIndex({
  index: {
    fields: ['foo', 'bar'],
    name: 'myindex',
    ddoc: 'mydesigndoc'
    type: 'json',
  }
}).then(function (result) {
  // handle result
}).catch(function (err) {
  console.log(err);
});
```

Options

- `fields` : a list of fields to index
- `name` (optional): name of the index, auto-generated if you don't include it
- `ddoc` (optional): design document name (i.e. the part after `'_design/'`), auto-generated if you don't include it
- `type` (optional): only supports `'json'` , which is also the default

Query index (https://github.com/pouchdb/pouchdb/edit/master/docs/_in

```
db.find(request [, callback])
```

Query an index and return the list of documents that match the request.

pouchdb-find plugin needed: This API requires the `pouchdb-find` plugin. See Mango queries (/guides/mango-queries.html) for installation instructions.

Example Usage:

Callbacks | Promises | Async functions

```
db.find({
  selector: {name: 'Mario'},
  fields: ['_id', 'name'],
  sort: ['name']
}).then(function (result) {
  // handle result
}).catch(function (err) {
  console.log(err);
});
```

Example Response:

```
{
  "docs": [
    {
      "_id": "mario",
      "name": "Mario"
    }
  ]
}
```

The above is a simple example. For an in-depth tutorial, please refer to the Mango guide (/guides/mango-queries.html).

Options

- `selector` Defines a selector to filter the results. Required.
  - `$lt` Match fields "less than" this one.
  - `$gt` Match fields "greater than" this one.
  - `$lte` Match fields "less than or equal to" this one.
  - `$gte` Match fields "greater than or equal to" this one.
  - `$eq` Match fields equal to this one.

  - `$ne` Match fields not equal to this one.
  - `$exists` True if the field should exist, false otherwise.
  - `$type` One of: "null", "boolean", "number", "string", "array", or "object".
  - `$in` The document field must exist in the list provided.
  - `$and` Matches if all the selectors in the array match.
  - `$nin` The document field must not exist in the list provided.
  - `$all` Matches an array value if it contains all the elements of the argument array.
  - `$size` Special condition to match the length of an array field in a document.
  - `$or` Matches if any of the selectors in the array match. All selectors must use the same index.
  - `$nor` Matches if none of the selectors in the array match.
  - `$not` Matches if the given selector does not match.
  - `$mod` Matches documents where (field % Divisor == Remainder) is true, and only when the document field is an integer.
  - `$regex` A regular expression pattern to match against the document field.
  - `$elemMatch` Matches all documents that contain an array field with at least one element that matches all the specified query criteria.
- `fields` (Optional) Defines a list of fields that you want to receive. If omitted, you get the full documents.
- `sort` (Optional) Defines a list of fields defining how you want to sort. Note that sorted fields also have to be selected in the `selector`.
- `limit` (Optional) Maximum number of documents to return.
- `skip` (Optional) Number of docs to skip before returning.
- `use_index` (Optional) Set which index to use for the query. It can be "design-doc-name" or "['design-doc-name', 'name']".

If there's no index that matches your `selector` / `sort`, then this method will issue a warning:

```
{
  "docs": [ /* ... */ ],
  "warning": "no matching index found, create an index to optimize query time"
}
```

The best index will be chosen automatically. If you want to see the query plan for your query, then turn on debugging:

```
PouchDB.debug.enable('pouchdb:find');
```

See the CouchDB `_find` documentation (http://docs.couchdb.org/en/2.0.0/api/database/find.html) for more details on selectors and the Mango query language.

More examples

Use `$eq` for "equals":

Callbacks Promises Async functions

```
db.find({
  selector: {name: {$eq: 'Mario'}}
}).then(function (result) {
  // handle result
}).catch(function (err) {
  console.log(err);
});
```

This is equivalent to:

Callbacks Promises Async functions

```
db.find({
  selector: {name: 'Mario'}
}).then(function (result) {
  // handle result
}).catch(function (err) {
  console.log(err);
});
```

You can also do selections on multiple fields. For instance, to find all docs where `series` is `'Mario'` and `debut` is greater than `1990`:

Callbacks Promises Async functions

```
db.find({
  selector: {
    series: 'Mario',
    debut: { $gt: 1990 }
  }
}).then(function (result) {
  // handle result
}).catch(function (err) {
  console.log(err);
});
```

This is equivalent to:

Callbacks Promises Async functions

```
db.find({
  selector: {
    $and: [
      { series: 'Mario' },
      { debut: { $gt: 1990 } }
    ]
  }
}).then(function (result) {
  // handle result
}).catch(function (err) {
  console.log(err);
});
```

You can also sort the returned documents. For instance, to find all docs sorted by `debut` descending:

Callbacks Promises Async functions

```
db.find({
  selector: {
    debut: {'$gte': null}
  },
  sort: [{debut: 'desc'}]
}).then(function (result) {
  // handle result
}).catch(function (err) {
  console.log(err);
});
```

Explain index (https://github.com/pouchdb/pouchdb/edit/master/docs/_i

```
db.explain(request [, callback])
```

Explain the query plan for a given query

i **pouchdb-find plugin needed:** This API requires the `pouchdb-find` plugin. See Mango queries (/guides/mango-queries.html) for installation instructions.

Example Usage:

Callbacks Promises Async functions

```
db.explain({
  selector: {
    name: 'Mario',
    series: "mario"
  },
  fields: ['_id', 'name'],
  sort: ['name']
}).then(function (explanation) {
  // view explanation
}).catch(function (err) {
  console.log(err);
});
```

Example Response:

```
{
  "dbname": "database-name"
  "index": {
    "ddoc": "_design/design-doc-name",
    "name": "index-name",
    "type": "json",
    "def": {
      "fields": [
        {
          "name": "asc"
        },
        {
          "series": "asc"
        }
      ]
    }
  },
  "selector": {
    "$and": [
      {
        "name": {
          "$eq": "mario"
        }
      },
      {
        "series": {
          "$eq": "mario"
        }
      }
    ]
  },
  "opts": {
    "use_index": [],
    "bookmark": "nil",
    "limit": 25,
    "skip": 0,
    "sort": {"name": "asc"},
    "fields": [
      "_id"
    ],
    "r": [
      49
    ],
    "conflicts": false
  },
  "limit": 10,
  "skip": 1,
  "fields": [
    "_id"
  ],
  "range": {
    "start_key": [
      "mario",
      "mario"
    ],
    "end_key": [
      "mario",
      "mario",
      {}
    ]
  }
};
```

List indexes (https://github.com/pouchdb/pouchdb/edit/master/docs/_inc

```
db.getIndexes([callback])
```

Get a list of all the indexes you've created. Also tells you about the special `_all_docs` index, i.e. the default index on the `_id` field.

pouchdb-find plugin needed: This API requires the `pouchdb-find` plugin. See Mango queries (/guides/mango-queries.html) for installation instructions.

Example Usage:

Callbacks Promises Async functions

```
db.getIndexes().then(function (result) {
  // handle result
}).catch(function (err) {
  console.log(err);
});
```

Example Response:

```
{
  "indexes": [
    {
      "ddoc": null,
      "name": "_all_docs",
      "type": "special",
      "def": {
        "fields": [
          {
            "_id": "asc"
          }
        ]
      }
    },
    {
      "ddoc": "_design/idx-0f3a6f73110868266fa5c688caf8acd3",
      "name": "idx-0f3a6f73110868266fa5c688caf8acd3",
      "type": "json",
      "def": {
        "fields": [
          {
            "foo": "asc"
          },
          {
            "bar": "asc"
          }
        ]
      }
    }
  ]
}
```

Delete index (https://github.com/pouchdb/pouchdb/edit/master/docs/_in

```
db.deleteIndex(index [, callback])
```

Delete an index, remove any orphaned design documents, and clean up any leftover data on disk.

pouchdb-find plugin needed: This API requires the `pouchdb-find` plugin. See Mango queries (/guides/mango-queries.html) for installation instructions.

Example Usage:

Callbacks Promises Async functions

```
db.deleteIndex({
  "ddoc": "_design/idx-0f3a6f73110868266fa5c688caf8acd3",
  "name": "idx-0f3a6f73110868266fa5c688caf8acd3",
  "type": "json",
  "def": {
    "fields": [
      { "foo": "asc" },
      { "bar": "asc" }
    ]
  }
}).then(function (result) {
  // handle result
}).catch(function (err) {
  console.log(err);
});
```

Example Response:

```
{ "ok": true }
```

Note that the easiest way to do this is to locate the index you want to delete using `getIndexes()`. For instance, here is how you would delete the second index from that list (which should be the one after the built-in `_all_docs` index):

Callbacks Promises Async functions

```
db.findIndexes().then(function (indexesResult) {
  return db.deleteIndex(indexesResult.indexes[1]);
}).then(function (result) {
  // handle result
}).catch(function (err) {
  console.log(err);
});
```

Notes:

- You don't need to provide a `_rev` when deleting an index.
- The associated design doc is automatically deleted, assuming it only contains one index.
- There is no need to call `viewCleanup` to clean up any leftover data. `deleteIndex()` does this automatically for you.

Map/reduce queries (https://github.com/pouchdb/pouchdb/edit/master/

```
db.query(fun, [options], [callback])
```

Invoke a map/reduce function, which allows you to perform more complex queries on PouchDB than what you get with allDocs(), changes(), or find(). The CouchDB documentation for map/reduce (http://docs.couchdb.org/en/latest/couchapp/views/intro.html) applies to PouchDB.

Since views perform a full scan of all documents, this method may be slow, unless you first save your view in a design document. Read the query guide (/guides/queries.html) for a good tutorial.

! **Warning: advanced API.** The map/reduce API is designed for cases that are too complex for Mango queries, which are described in createIndex(), find(), listIndexes(), and deleteIndex().

Under the hood, Mango indexes are the same as map/reduce indexes. The Mango API is just a simplified user-facing API on top of map/reduce.

Options

All options default to `false` unless otherwise specified.

- `fun` : Map/reduce function, which can be one of the following:
  - A full CouchDB-style map/reduce view: `{map : ..., reduce: ...}`.
  - A map function by itself (no reduce).
  - The name of a view in an existing design document (e.g. `'mydesigndoc/myview'`, or `'myview'` as a shorthand for `'myview/myview'`).
- `options.reduce` : Reduce function, or the string name of a built-in function: `'_sum'`, `'_count'`, or `'_stats'`. Defaults to `true` when a reduce function is defined, or `false` otherwise.

  - Tip: if you’re not using a built-in, you’re probably doing it wrong (http://youtu.be/BKQ9kXKoHS8?t=865s).
  - PouchDB will always call your reduce function with rereduce == false. As for CouchDB, refer to the CouchDB documentation (http://docs.couchdb.org/en/1.6.1/couchapp/views/intro.html).
- `options.include_docs` : Include the document in each row in the `doc` field.
  - `options.conflicts` : Include conflicts in the `_conflicts` field of a doc.
  - `options.attachments` : Include attachment data.
    - `options.binary` : Return attachment data as Blobs/Buffers, instead of as base64-encoded strings.
- `options.startkey` & `options.endkey` : Get rows with keys in a certain range (inclusive/inclusive).
- `options.inclusive_end` : Include rows having a key equal to the given `options.endkey` . Default: `true` .
- `options.limit` : Maximum number of rows to return.
- `options.skip` : Number of rows to skip before returning (warning: poor performance on IndexedDB/LevelDB!).
- `options.descending` : Reverse the order of the output rows.
- `options.key` : Only return rows matching this key.
- `options.keys` : Array of keys to fetch in a single shot.
  - Neither `startkey` nor `endkey` can be specified with this option.
  - The rows are returned in the same order as the supplied `keys` array.
  - The row for a deleted document will have the revision ID of the deletion, and an extra key `"deleted":true` in the `value` property.
  - The row for a nonexistent document will just contain an `"error"` property with the value `"not_found"` .
- `options.group` : True if you want the reduce function to group results by keys, rather than returning a single result. Defaults to `false` .
- `options.group_level` : Number of elements in a key to group by, assuming the keys are arrays. Defaults to the full length of the array.
- `options.stale` : One of `'ok'` or `'update_after'` . Only applies to saved views. Can be one of:
  - unspecified (default): Returns the latest results, waiting for the view to build if necessary.
  - `'ok'` : Returns results immediately, even if they’re out-of-date.
  - `'update_after'` : Returns results immediately, but kicks off a build afterwards.
- `options.update_seq` : Include an `update_seq` value indicating which sequence id of the underlying database the view reflects.

For details, see the CouchDB query options documentation (http://docs.couchdb.org/en/stable/api/ddoc/views.html#get--db-_design-ddoc-_view-view).

Example Usage:

Callbacks Promises Async functions

```
// create a design doc
var ddoc = {
  _id: '_design/index',
  views: {
    index: {
      map: function mapFun(doc) {
        if (doc.title) {
          emit(doc.title);
        }
      }.toString()
    }
  }
}

// save the design doc
db.put(ddoc).catch(function (err) {
  if (err.name !== 'conflict') {
    throw err;
  }
  // ignore if doc already exists
}).then(function () {
  // find docs where title === 'Lisa Says'
  return db.query('index', {
    key: 'Lisa Says',
    include_docs: true
  });
}).then(function (result) {
  // handle result
}).catch(function (err) {
  console.log(err);
});
```

Example Response:

```
{
  "offset" : 0,
  "rows": [{
    "id": "doc3",
    "key": "Lisa Says",
    "value": null,
    "doc": {
      "_id": "doc3",
      "_rev": "1-z",
      "title": "Lisa Says"
    }
  }],
  "total_rows" : 4
}
```

In the result, `total_rows` is the total number of possible results in the view. The response is very similar to that of `allDocs()`.

The result may also have `update_seq` if you set `update_seq` to `true`

Note: you can also pass in the map function instead of saving a design doc first, but this is slow because it has to do a full database scan. The following examples will use this pattern for simplicity's sake, but you should normally avoid it.

Complex keys

You can also use complex keys (https://wiki.apache.org/couchdb/Introduction_to_CouchDB_views#Complex_Keys) for fancy ordering:

Callbacks Promises Async functions

```
function map(doc) {
  // sort by last name, first name, and age
  emit([doc.lastName, doc.firstName, doc.age]);
}
db.query(map).then(function (result) {
  // handle result
}).catch(function (err) {
  console.log(err);
});
```

Example Response:

```
{
  "offset": 0,
  "rows": [{
      "id"  : "bowie",
      "key" : ["Bowie", "David", 67]
    }, {
      "id"  : "dylan",
      "key" : ["Dylan", "Bob", 72]
    }, {
      "id"  : "younger_dylan",
      "key" : ["Dylan", "Jakob", 44]
    }, {
      "id"  : "hank_the_third",
      "key" : ["Williams", "Hank", 41]
    }, {
      "id"  : "hank",
      "key" : ["Williams", "Hank", 91]
    }],
  "total_rows": 5
}
```

Tips:

- The sort order is `[nulls, booleans, numbers, strings, arrays, objects]`, so `{startkey: ['Williams'], endkey: ['Williams', {}]}` would return all people with the last name `'Williams'` because objects are higher than strings. Something like `'zzzzz'` or `'\ufff0'` would also work.

- `group_level` can be very helpful when working with complex keys. In the example above, you can use `{group_level: 1}` to group by last name, or `{group_level: 2}` to group by last and first name. (Be sure to set `{reduce: true, group: true}` as well.)

Linked documents

PouchDB fully supports linked documents (https://wiki.apache.org/couchdb/Introduction_to_CouchDB_views#Linked_documents). Use them to join two types of documents together, by simply adding an `_id` to the emitted value:

Callbacks Promises Async functions

```
function map(doc) {
  // join artist data to albums
  if (doc.type === 'album') {
    emit(doc.name, {_id : doc.artistId, albumYear : doc.year});
  }
}
db.query(map, {include_docs : true}).then(function (result) {
  // handle result
}).catch(function (err) {
  console.log(err);
});
```

Example response:

```
{
    "offset": 0,
    "rows": [
        {
            "doc": {
                "_id": "bowie",
                "_rev": "1-fdb234b78904a5c8293f2acf4be70d44",
                "age": 67,
                "firstName": "David",
                "lastName": "Bowie"
            },
            "id": "album_hunkeydory",
            "key": "Hunky Dory",
            "value": {
                "_id": "bowie",
                "albumYear": 1971
            }
        },
        {
            "doc": {
                "_id": "bowie",
                "_rev": "1-fdb234b78904a5c8293f2acf4be70d44",
                "age": 67,
                "firstName": "David",
                "lastName": "Bowie"
            },
            "id": "album_low",
            "key": "Low",
            "value": {
                "_id": "bowie",
                "albumYear": 1977
            }
        },
        {
            "doc": {
                "_id": "bowie",
                "_rev": "1-fdb234b78904a5c8293f2acf4be70d44",
                "age": 67,
                "firstName": "David",
                "lastName": "Bowie"
            },
            "id": "album_spaceoddity",
            "key": "Space Oddity",
            "value": {
                "_id": "bowie",
                "albumYear": 1969
            }
        }
    ],
    "total_rows": 3
}
```

Closures

If you pass a function to `db.query` and give it the `emit` function as the second argument, then you can use a closure. (Since PouchDB has to use `eval()` to bind `emit` .)

Callbacks | Promises | Async functions

```
// BAD! will throw error
var myId = 'foo';
db.query(function(doc) {
  if (doc._id === myId) {
    emit(doc);
  }
}).catch(function (err) {
  // you'll get an error here
}

// will be fine
var myId = 'foo';
db.query(function(doc, emit) {
  if (doc._id === myId) {
    emit(doc);
  }
}).then(function (result) {
  // handle result
}).catch(function (err) {
  console.log(err);
});
```

Note that closures are only supported by local databases with temporary views. So if you are using closures, then you must use the slower method that requires a full database scan.

View cleanup (https://github.com/pouchdb/pouchdb/edit/master/docs/_i

```
db.viewCleanup([callback])
```

Cleans up any stale map/reduce indexes.

As design docs are deleted or modified, their associated index files (in CouchDB) or companion databases (in local PouchDBs) continue to take up space on disk. `viewCleanup()` removes these unnecessary index files.

See the CouchDB documentation on view cleanup (http://couchdb.readthedocs.org/en/latest/maintenance/compaction.html#views-cleanup) for details.

Example Usage:

Callbacks Promises Async functions

```
db.viewCleanup().then(function (result) {
  // handle result
}).catch(function (err) {
  console.log(err);
});
```

Example Response:

```
{ "ok" : "true" }
```

Get database information (https://github.com/pouchdb/pouchdb/edit/ma

```
db.info([callback])
```

Get information about a database.

Example Usage:

Callbacks Promises Async functions

```
db.info().then(function (result) {
  // handle result
}).catch(function (err) {
  console.log(err);
});
```

Example Response:

```
{
  "db_name": "test",
  "doc_count": 4,
  "update_seq": 5
}
```

Response object:

- `db_name` is the name of the database you gave when you called `new PouchDB()`, and also the unique identifier for the database.
- `doc_count` is the total number of non-deleted documents in the database.
- `update_seq` is the sequence number of the database. It starts at 0 and gets incremented every time a document is added or modified.

There are also some details you can use for debugging. These are unofficial and may change at any time:

- `adapter`: The name of the adapter being used (idb, leveldb, …).
- `idb_attachment_format`: (IndexedDB) either `'base64'` or `'binary'`, depending on whether the browser supports binary blobs (/faq.html#data_types).
- `backend_adapter`: (Node.JS) the backend *DOWN adapter being used (MemDOWN, RiakDOWN, …).

Compact the database (https://github.com/pouchdb/pouchdb/edit/maste

```
db.compact([options], [callback])
```

Triggers a compaction operation in the local or remote database. This reduces the database's size by removing unused and old data, namely non-leaf revisions and attachments that are no longer referenced by those revisions. Note that this is a separate operation from `viewCleanup()`.

For remote databases, PouchDB checks the compaction status at regular intervals and fires the callback (or resolves the promise) upon completion. Consult the compaction section of CouchDB's maintenance documentation (http://couchdb.readthedocs.org/en/latest/maintenance/compaction.html) for more details.

Also see auto-compaction, which runs compaction automatically (local databases only).

- `options.interval`: Number of milliseconds to wait before asking again if compaction is already done. Defaults to 200. (Only applies to remote databases.)

Example Usage:

Callbacks Promises Async functions

```
db.compact().then(function (result) {
  // handle result
}).catch(function (err) {
  console.log(err);
});
```

Example Response:

```
{ "ok" : "true" }
```

Document revisions diff (https://github.com/pouchdb/pouchdb/edit/mast

```
db.revsDiff(diff, [callback])
```

Given a set of document/revision IDs, returns the subset of those that do not correspond to revisions stored in the database. Primarily used in replication.

Example Usage:

Callbacks Promises Async functions

```
db.revsDiff({
  myDoc1: [
    "1-b2e54331db828310f3c772d6e042ac9c",
    "2-3a24009a9525bde9e4bfa8a99046b00d"
  ]
}).then(function (result) {
  // handle result
}).catch(function (err) {
  console.log(err);
});
```

Example Response:

```
{
  "myDoc1": {
    "missing": ["2-3a24009a9525bde9e4bfa8a99046b00d"]
  }
}
```

Document bulk get (https://github.com/pouchdb/pouchdb/edit/master/d

```
db.bulkGet(options, [callback])
```

Given a set of document/revision IDs, returns the document bodies (and, optionally, attachment data) for each ID/revision pair specified.

Options

- `options.docs` : An array of `id` and `rev` pairs representing the revisions to fetch.
  - `id` : ID of the document to fetch.
  - `rev` : Revision of the document to fetch. If this is not specified, all available revisions are fetched.
  - `atts_since` : Optional and supported by the http adapter only. Includes attachments only since specified revisions. Doesn't includes attachments for specified revisions.
- `options.revs` : Each returned revision body will include its revision history as a `_revisions` property. Default is `false` .
- `options.attachments` : Include attachment data in the response. Default is `false` , resulting in only stubs being returned.
- `options.binary` : Return attachment data as Blobs/Buffers, instead of as base64-encoded strings. Default is `false` .

Example Usage:

Callbacks Promises Async functions

```
db.bulkGet({
    docs: [
      { id: "doc-that-exists", rev: "1-967a00dff5e02add41819138abb3284d"},
      { id: "doc-that-does-not-exist", rev: "1-3a24009a9525bde9e4bfa8a99046b00d"},
      { id: "doc-that-exists", rev: "1-bad_rev"}
    ]
}).then(function (result) {
  // handle result
}).catch(function (err) {
  console.log(err);
});
```

Example Response:

```
{
  "results": [
  {
    "docs": [
    {
      "ok": {
        "_id": "doc-that-exists",
        "_rev": "1-967a00dff5e02add41819138abb3284d",
        "_revisions": {
          "ids": [
            "967a00dff5e02add41819138abb3284d"
          ],
          "start": 1
        }
      }
    }],
    "id": "doc-that-exists"
  },
  {
    "docs": [
    {
      "error": {
        "error": "not_found",
        "id": "doc-that-does-not-exist",
        "reason": "missing",
        "rev": "undefined"
      }
    }],
    "id": "doc-that-does-not-exist"
  },
  {
    "docs": [
    {
      "error": {
        "error": "not_found",
        "id": "doc-that-exists",
        "reason": "missing",
        "rev": "1-badrev"
      }
    }
    ],
    "id": "doc-that-exists"
  }]
}
```

Close a database (https://github.com/pouchdb/pouchdb/edit/master/docs

```
db.close([callback])
```

Close the database, this closes any open connection to the underlying storage and frees memory (event listeners) the database may be using.

Example Usage

Callbacks Promises Async functions

```
db.close().then(function () {
  // success
});
```

Events (https://github.com/pouchdb/pouchdb/edit/master/docs/_include

PouchDB is an event emitter (http://nodejs.org/api/events.html#events_class_events_eventemitter) and will emit a `'created'` event when a database is created. A `'destroyed'` event is emitted when a database is destroyed.

```
PouchDB.on('created', function (dbName) {
  // called whenever a db is created.
});
PouchDB.on('destroyed', function (dbName) {
  // called whenever a db is destroyed.
});
```

Default settings (https://github.com/pouchdb/pouchdb/edit/master/docs

If you find yourself using the same constructor options repeatedly, you can simplify your code with `PouchDB.defaults()` :

```
PouchDB.defaults({
  option1: 'foo',
  option2: 'value'
});
```

The returned object is a constructor function that works the same as `PouchDB` , except that whenever you invoke it (e.g. with `new`), the given options will be passed in by default.

Example Usage:

```
var MyMemPouch = PouchDB.defaults({
  adapter: 'memory'
});
// In-memory PouchDB
var myMemPouch = new MyMemPouch('dbname');

var MyPrefixedPouch = PouchDB.defaults({
  prefix: '/path/to/my/db/'
});
// db will be named '/path/to/my/db/dbname', useful for LevelDB
var myPrefixedPouch = new MyPrefixedPouch('dbname');

var HTTPPouch = PouchDB.defaults({
  prefix: 'http://example.org'
});

// db will be located at 'http://example.org/dbname'
var myHttpPouch = new HTTPPouch('dbname');
```

Note the special constructor option `prefix` , which appends a prefix to the database name and can be helpful for URL-based or file-based LevelDOWN path names.

All constructor options are supported. Default options can still be overriden individually.

Plugins (https://github.com/pouchdb/pouchdb/edit/master/docs/_include

i **This section covers *authoring* plugins.** For a list of third-party plugins, see Plugins (/external.html), or for a list of first-party plugins that you can use to customize the PouchDB build, see Custom Builds (/custom.html).

Writing a plugin is easy! The API is:

```
PouchDB.plugin({
  methodName: myFunction
});
```

This will add a `db.methodName()` to all databases, which runs `myFunction` .It will always be called in context, so that within the function, `this` refers to the database object.

There is a PouchDB Plugin Seed project (https://github.com/pouchdb/plugin-seed), which is the fastest way to get started writing, building and testing your very own plugin.

Example Usage:

```
PouchDB.plugin({
  sayHello : function () {
    console.log("Hello!");
  }
});
new PouchDB('foobar').sayHello(); // prints "Hello!"
```

Alternatively, instead of passing in an object to `.plugin()`, you can pass in a function that takes the `PouchDB` object and performs whatever operations you want on it. You can use this to load multiple plugins, add adapters, or attach event listeners to the `PouchDB` object.

Example Usage:

```
PouchDB.plugin(function (PouchDB) {
  PouchDB.hello = 'world';
});
console.log(PouchDB.hello); // prints "world"
```

(Most likely, if you are writing a PouchDB plugin, you will export either the object-style or the function-style plugin, so that your users can then attach it to their PouchDB object.)

Load Plugins from require()

You can load plugins into PouchDB when you load it via `require()`.

```
var greet = {sayHello: function() { console.log("Hello!"); }};

var PouchDB = require('pouchdb').plugin(greet);

var db = new PouchDB('foobar');
db.sayHello(); // prints "Hello!"
```

You can chain plugins, as well:

```
var greet = {sayHello: function() { console.log("Hello!"); }};
var manners = {thank: function(name) { console.log("Thank you, " + name); }};

var PouchDB = require('pouchdb')
  .plugin(greet)
  .plugin(manners);

var db = new PouchDB('foobar');
db.sayHello(); // prints "Hello!"
db.thank('Mom'); // prints "Thank you, Mom"
```

Example Plugin: Intercept Updates

A useful feature of plugins is to intercept updates before they are stored in PouchDB. In this way, a plugin might validate that the data is correct for the application, or even alter documents before they are committed to the database.

The best way to intercept all updates to a PouchDB database is to **override the `bulkDocs()` method**. All changes to PouchDB documents ultimately pass through the `bulkDocs()` method. For example, a call to `put()` will become a `bulkDocs()` call with a “batch” of one document.

Because PouchDB guarantees to plugin authors that all data changes ultimately happen via `bulkDocs()`, it is the ideal place for an application or plugin to intercept updates.

```
// Keep a reference to the "upstream" function.
var pouchBulkDocs = PouchDB.prototype.bulkDocs;
PouchDB.plugin({bulkDocs: validBulkDocs});

function validBulkDocs(body, options, callback) {
  if (typeof options == 'function') {
    callback = options
    options = {}
  }

  if (Array.isArray(body)) {
    var docs = body;
  } else {
    var docs = body.docs;
  }

  // All documents must have a .name field.
  for (var i = 0; i < docs.length; i++) {
    if (!docs[i].name) {
      var id = doc._id || '(no _id given)';
      return callback(new Error('Document is missing .name field: ' + id));
    }
  }

  // All documents check out. Pass them to PouchDB.
  return pouchBulkDocs.call(this, docs, options, callback);
}
```

The above plugin would return an error if anything ever attempts to store an unnamed document, including documents which change during replication.

Note: this is a very, very simple validation example. It does not behave, for example, like the Apache CouchDB `validate_doc_update()` API.

Debug mode (https://github.com/pouchdb/pouchdb/edit/master/docs/_ir

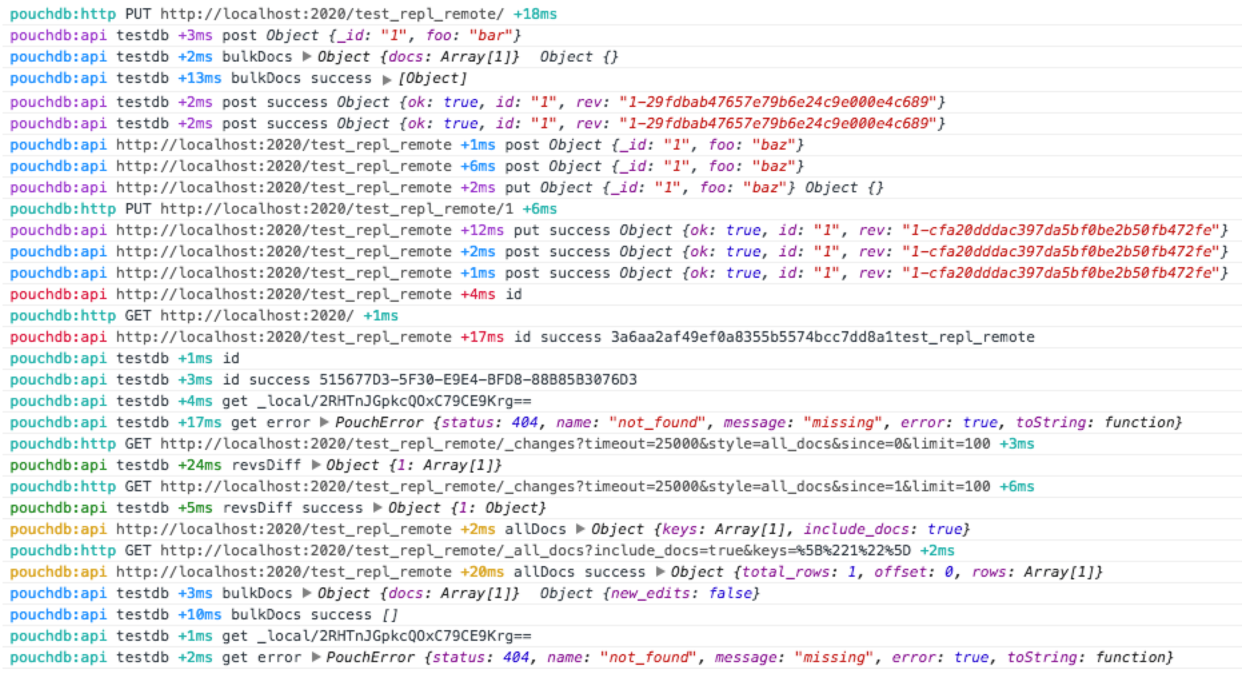
PouchDB uses the debug (https://www.npmjs.org/package/debug) module for fine-grained debug output.

! This `debug()` API is currently part of PouchDB core. However, in PouchDB v7.0.0 it will be moved to a separate plugin.

To enable debug mode, just call:

```
PouchDB.debug.enable('*');
```

In your browser console, you should then see something like this:



```
pouchdb:http PUT http://localhost:2020/test_repl_remote/ +18ms
pouchdb:api testdb +3ms post Object {_id: "1", foo: "bar"}
pouchdb:api testdb +2ms bulkDocs ▶ Object {docs: Array[1]}  Object {}
pouchdb:api testdb +13ms bulkDocs success ▶ [Object]
pouchdb:api testdb +2ms post success Object {ok: true, id: "1", rev: "1-29fdbab47657e79b6e24c9e000e4c689"}
pouchdb:api testdb +2ms post success Object {ok: true, id: "1", rev: "1-29fdbab47657e79b6e24c9e000e4c689"}
pouchdb:api http://localhost:2020/test_repl_remote +1ms post Object {_id: "1", foo: "baz"}
pouchdb:api http://localhost:2020/test_repl_remote +6ms post Object {_id: "1", foo: "baz"}
pouchdb:api http://localhost:2020/test_repl_remote +2ms put Object {_id: "1", foo: "baz"} Object {}
pouchdb:http PUT http://localhost:2020/test_repl_remote/1 +6ms
pouchdb:api http://localhost:2020/test_repl_remote +12ms put success Object {ok: true, id: "1", rev: "1-cfa20dddac397da5bf0be2b50fb472fe"}
pouchdb:api http://localhost:2020/test_repl_remote +2ms post success Object {ok: true, id: "1", rev: "1-cfa20dddac397da5bf0be2b50fb472fe"}
pouchdb:api http://localhost:2020/test_repl_remote +1ms post success Object {ok: true, id: "1", rev: "1-cfa20dddac397da5bf0be2b50fb472fe"}
pouchdb:api http://localhost:2020/test_repl_remote +4ms id
pouchdb:http GET http://localhost:2020/ +1ms
pouchdb:api http://localhost:2020/test_repl_remote +17ms id success 3a6aa2af49ef0a8355b5574bcc7dd8a1test_repl_remote
pouchdb:api testdb +1ms id
pouchdb:api testdb +3ms id success 515677D3-5F30-E9E4-BFD8-88B85B3076D3
pouchdb:api testdb +4ms get _local/2RHTnJGpkcQ0xC79CE9Krg==
pouchdb:api testdb +17ms get error ▶ PouchError {status: 404, name: "not_found", message: "missing", error: true, toString: function}
pouchdb:http GET http://localhost:2020/test_repl_remote/_changes?timeout=25000&style=all_docs&since=0&limit=100 +3ms
pouchdb:api testdb +24ms revsDiff ▶ Object {1: Array[1]}
pouchdb:http GET http://localhost:2020/test_repl_remote/_changes?timeout=25000&style=all_docs&since=1&limit=100 +6ms
pouchdb:api testdb +5ms revsDiff success ▶ Object {1: Object}
pouchdb:api http://localhost:2020/test_repl_remote +2ms allDocs ▶ Object {keys: Array[1], include_docs: true}
pouchdb:http GET http://localhost:2020/test_repl_remote/_all_docs?include_docs=true&keys=%5B%221%22%5D +2ms
pouchdb:api http://localhost:2020/test_repl_remote +20ms allDocs success ▶ Object {total_rows: 1, offset: 0, rows: Array[1]}
pouchdb:api testdb +3ms bulkDocs ▶ Object {docs: Array[1]}  Object {new_edits: false}
pouchdb:api testdb +10ms bulkDocs success []
pouchdb:api testdb +1ms get _local/2RHTnJGpkcQ0xC79CE9Krg==
pouchdb:api testdb +2ms get error ▶ PouchError {status: 404, name: "not_found", message: "missing", error: true, toString: function}
```

(/static/img/debug_mode.png)

In Node.js, you can also set a command-line flag:

```
DEBUG=pouchdb:* node myscript.js
```

You can also enable debugging of specific modules. Currently we only have `pouchb:api` (API-level calls) and `pouchdb:http` (HTTP requests):

```
PouchDB.debug.enable('pouchdb:api'); // or
PouchDB.debug.enable('pouchdb:http');
```

These settings are saved to the browser's LocalStorage. So to disable them, you must call:

```
PouchDB.debug.disable();
```

Your users won't see debug output unless you explicitly call `PouchDB.debug.enable()` within your application code.

(https://twitter.com/pouchdb)

(https://github.com/rvagg/node-levelup)

(https://github.com/pouchdb/pouchdb)

(https://travis-ci.org/pouchdb/pouchdb)

(http://couchdb.apache.org/)

(https://saucelabs.com)

Learn

Getting Started (/getting-started.html)
API Guide (/api.html)
Wiki (https://github.com/pouchdb/pouchdb/wiki)

Discuss

Mailing List (https://groups.google.com/forum/#!forum/pouchdb)
IRC (irc://freenode.net/#pouchdb)
Twitter (http://twitter.com/pouchdb)
StackOverflow (http://stackoverflow.com/questions/tagged/pouchdb)

Contribute

Contributing (https://github.com/pouchdb/pouchdb/blob/master/CONTRIBUTING.md)
Source (https://github.com/pouchdb/pouchdb)
Issues (https://github.com/pouchdb/pouchdb/issues)