Minimongo

Source code of released version | Source code of development version

---

`minimongo` is reimplementation of (almost) the entire MongoDB API, against an in-memory JavaScript database. It is like a MongoDB emulator that runs inside your web browser. You can insert data into it and search, sort, and update that data. This is great if you like the MongoDB API (which you may already be using on the server), need a way to store data on the client that you've fetched and are using to render your interface, and want to use that familiar API.

Minimongo is used as a temporary data cache in the standard Meteor stack, to learn more about mini-databases and what they can do, see the project page on www.meteor.com

Internals

Minimongo implements the following features, mirroring the MongoDB features:

- Selectors
- Modifiers
- Fields projections
- Querying with `sort` and `limit`
- ObjectID generation
- Geo-positional operator `$near` with GeoJSON parsing

Internally, all documents are mapped in a single JS object from `_id` to the document. Besides this mapping, Minimongo doesn't implement any types of secondary indexes.

Also, currently Minimongo doesn't implement any aggregation features. The full list of incompatible features can be found in the NOTES file.

Besides the MongoDB features, Minimongo implements the following for a better integration with the Meteor stack:

- `observe` and `observeChanges` APIs, notification callbacks when the result of a query changes
- integration with Meteor's Tracker
- Meteor's Publish a cursor interface `_publishCursor`, that results into an `observe` call
- Meteor's interface of `pauseObservers` and `resumeObservers`, for client-side caches (allow latency compensation to change multiple objects at once w/o a flicker)
- In addition to the previous point, `saveOriginals` and `retrieveOriginals` are used to revert the local changes
- additional analysis functions for Meteor's server-side Oplog Observe Driver (only loaded for the server-side):
  - can this modifier change the result of this selector if the document didn't match before
  - can the sort order change after applying this modifier (for a given selector)
  - what is a combined projection for these selector, sorter and projection
- `find` accepts a literal function instead of a selector, and a comparison function for sorting

saveOriginals/retrieveOriginals & pauseObserver/resumeObservers

This part of the implementation is very important for a smooth Optimistic UI experience (also called "Latency Compensation") avoiding an extra flicker.

Let's review the usual Optimistic UI flow:

- User triggers an interaction on the client
- The simulation applies some mutations to the state locally
- The RPC is fired to be executed on the server
- After some time, the RPC returns with the result
- After some more time, RPC returns an "updated" message (on DDP level), meaning that all the changes from RPC have persisted
- At this point we know, that all the actual changes from the server are synced, we can throw away the simulated mutations (preserving the real changes from the server)

To implement the last step, Minimongo implements the following:

- When a simulation starts, `saveOriginals` is called, to take a snapshot of the state before the update. Internally, it is implemented in a more efficient copy-on-write style.
- After the simulation, wait, to get all the real changes from the server.
- When the server is done with giving us new changes, we apply all the new changes and throw away all the simulated changes (with `retrieveOriginals`).
- All massive replacements happen in between `pauseObservers` and `resumeObservers` calls, so any user code (such as Blaze) sees the whole change in one tick (helps to avoid the flicker).