Cheatsheet for Mongoose Query Methods

A cheatsheet of methods you can use to implement common patterns in mongoose data-fetching(eg pagination, filtering etc).

Base Example

We'll work with a couple of queries which have not been resolved. Since each method returns the overall instance, we can chain methods

```
const todosQuery = Todo.find({});
const userQuery = User.find({});
```

Filtering By Value

Using the `where()` method in conjugation with some other methods, based on value we can filter out certain results.

The following methods are usually used with the `where()` method:

- `equals()` - equal to

```
userQuery.where('age').equals(13); // results where user age is 18
```

- `gt()` - greater than

```
userQuery.where('age').equals(13); // results where user age is more than 18
```

- `lt()` - less than

```
userQuery.where('age').equals(13); // results where user age is less than 18
```

- `gte()` - greater than or equal to

```
userQuery.where('age').gte(13); // results where user age is greater than or equal to 18
```

- `lte()` - less than or equal to

```
userQuery.where('age').lte(18); // results where user age is less than or equal to 18
```

- `mod()` - reminder

```
userQuery.where('age').mod([2, 1]); // all users where add is odd
```

- `ne()` - not equal to

```
userQuery.where('age').ne(13); // results where age is not 13
```

- `size()` - size of array

```
userQuery.where('projects').size(); // 3
```

- `regex()` - use regex

```
userQuery.where('name').regex(/abc/i); // results where the name matches the pattern
```

- `slice()` - JavaScript slice on the array of results

```
// projects is an array
userQuery.where('projects').slice(2); // results after doing the slice on each result
```

- `all()` - all elements in the argument array is present

```
// projects is an array
userQuery.where('projects').all(['p1', 'p2', 'p3']); // results where the field has all the specified
```

Sorting

sort()

The field name is the key, and the value states whether it's ascending or descending. There are different ways to implement this:

```
// -1 or 1
todosQuery.sort({ title: -1, description: 1 });

// 'ascending' or 'descending'
todosQuery.sort({ title: 'descending', description: 'ascending' });

// 'asc' or 'desc'
todosQuery.sort({ title: 'desc', description: 'asc' });

// shorthand - title is ascending and description is descending
todosQuery.sort('title -description');
```

Pagination

We can use a combination of `limit` and `skip` to implement pagination easily. Before that, let's look at how these two methods work.

limit()

```
todosQuery.limit(100);
```

skip()

This will skip the first `x` results of the `find` query.

```
todosQuery.skip(20);
```

Combining To Implement Pagination

Let's say that we are on page `2` and each page contains 10 results. We'll create a variable called `page` which is `2` and `number` which is `10`.

So, we'll skip the first `(page - 1) * number` which will send the results 11 and above.

```
const page = 2;
const number = 10;

todosQuery.skip((page - 1) * number);
```

Until now, we've skipped the first 10 results, but we still show all of remaining results. Therefore, let's use the `limit()` method to limit the number of results.

```
const page = 2;
const number = 10;

// skip = 2-1 * 10 = first 10 results
// limit = 2 * 10 = first 20 results
// the results are limited to the first 11 to 20 results
todosQuery.skip((page - 1) * number).limit(page * 10);
```

Performance

explain()

This method returns execution stats instead of the data. It can be useful when comparing different approaches and analyzing which one is more performative.

```
const stats = await todosQuery.explain();
```

lean()

Lean removes all the `getters`, `setters` and the `virtuals` from the document that is returned by mongoose. The object returned is a plain JavaScript object and not a mongoose query compared to other query methods.

```
const user = await userQuery.lean();
```

> *Chaining further query methods will not work as the mongoose query is not returned, the result is. It's like calling the `exec()` method.*

cursor()

Cursors are the native way mongodb navigates through the database. With mongoose, an array is returned as a result of `find()` but the native driver returns a `Cursor`.

Mongoose allows us to get the data in the form a cursor/stream because it may be more performant in some cases.

```
// There are 2 ways to use a cursor. First, as a stream:
Thing.
  find({ name: /^hello/ }).
  cursor().
  on('data', function(doc) { console.log(doc); }).
  on('end', function() { console.log('Done!'); });

// Or you can use `.next()` to manually get the next doc in the stream.
// `.next()` returns a promise, so you can use promises or callbacks.
const cursor = Thing.find({ name: /^hello/ }).cursor();
cursor.next(function(error, doc) {
  console.log(doc);
});

// Because `.next()` returns a promise, you can use co
// to easily iterate through all documents without loading them
// all into memory.
const cursor = Thing.find({ name: /^hello/ }).cursor();
for (let doc = await cursor.next(); doc != null; doc = await cursor.next()) {
  console.log(doc);
}
```

Schema

A Schema is a central concept in Mongoose. A Schema is mapped to a MongoDB collection; it defines the shape of the documents in that collection. You can read about different types you can use in Schema here.

Base Example

You will work on a collection that stores the number of hits of a web page. Each document in the collection stores the page path (URL) and the number of hits the page got.

Example:

```
{
  pagePath: '/contacts'
  hits: 10
}
```

Define a `Schema` and export it as a `Model`:

```
// PageHits.js

const pageHitsSchema = new mongoose.Schema({
  pagePath: String,
  hits: number,
});

export { mongoose.model("PageHits", pageHitsSchema) };
```

Instantiate a `document` using a `Model`:

```
import { PageHits } from 'PageHits';

async function recordPageHits(pageHit) {
  const john = await PageHits.create(pageHit);
}

const pageHit = {
  pagePath: '/contact',
  hits: 8,
};

recordPageHits(pageHit);
```

Validation

You can set the data to be automatically validated before the save so data integrity is maintained.

Make `hits` a mandatory value:

```
const pageHitsSchema = new mongoose.Schema({
  pagePath: String,
  hits: {
    type: Number,
    required: true,
  },
});
```

Querying

You can use queries on instantiated documents.

`find()` - Find all pages with page hits higher than `n`:

```
async function getPageHitsHigherThan(n) {
  const result = await PageHits.find({
    hits: {$gt: n},
  });
}
```

`deleteOne()` - Delete the "page hit" record of a page:

```
async function deletePageHitRecord(page) {
  const result = await PageHits.deleteOne({
    pagePath: page,
  });
}
```

Instance methods

You can define methods on **document instances**.

Write an instance method to find all the pages with same number of hits as the current page:

```
const pageHitsSchema = new mongoose.Schema({
  pagePath: String,
  hits: {
    type: Number,
    required: true,
  },
  methods: {
    pagesWithSameHits() {
      return mongoose.model('PageHits').find({
        hits: {eq: this.hits}
      });
    },
  },
});

export { mongoose.model("PageHits", pageHitsSchema) };
```

Tip: Don't use arrow functions when defining instance methods, `this` will not work.

Now, you can find all the pages that have the same number of hits are the current page:

```
const aboutPageHits = new PageHits({
  pagePath: '/about',
  hits: 40,
});

aboutPageHits.pagesWithSameHits((err, allPagesWithSameHits) => {
  console.log(allPagesWithSameHits);
})
```

Static methods

You can define static methods on the **Model**.

Write a static method, `pagesWithAtleastHits()`, to find all the pages with at least `n` hits:

```
const pageHitsSchema = new mongoose.Schema({
  pagePath: String,
  hits: {
    type: Number,
    required: true,
  },
  statics: {
    pagesWithAtleastHits() {
      return this.find({
        hits: {gte: n}
      });
    },
  },
});

export { mongoose.model("PageHits", pageHitsSchema) };
```

Tip: Don't use arrow functions when defining static methods, `this` will not work.

Find all the pages with at least `n` hits:

```
const pages = await PageHits.pagesWithAtleastHits(n);
```

Instance vs Static methods

Remember, you use a Schema to create a Model. Then the you can use the Model to create a Document instance. Schema relates to the whole collection whereas a Document instance relates to a particular document in the collection.

Instance Methods	Static Methods
Instance methods work on the Document instance.	Static methods work on the Model.
Use an instance method if you want to do something in relation to a particular document in the collection.	Use a static method if you want to operate on the entire collection.

Query Helpers

You can add query helper functions which let you extend mongoose's chainable query builder API.

You need to use either `find()` or `where()` before you call the query helper method. You can also extend it like you would a normal query chain.

Define a query helper method, `pagesWithNHits()`, to find all pages with `n` page hits:

```
const pageHitsSchema = new mongoose.Schema({
  pagePath: String,
  hits: {
    type: Number,
    required: true,
  },
  query: {
    pagesWithNHits(n) {
      return this.where({
        hits: {eq: n}
      });
    },
  },
});

export { mongoose.model("PageHits", pageHitsSchema) };
```

Use the query helper, `pagesWithNHits()`, to find all pages with `n` page hits.

```
PageHits.find().pagesWithNHits(n).exec((arr, pages) => {
  console.log(pages);
});
```

Virtual properties

Virtual properties are not stored in MongoDB. They only exist in Mongoose.

Virtuals provide `get` to retrieve values for further manipulation and `set` to manipulates values and then store it in the document.

Now `pageHitInfo()` can be used to print information about a current PageHit document:

```
const pageHitsSchema = new mongoose.Schema({
  pagePath: String,
  hits: {
    type: Number,
    required: true,
  },
  virtuals: {
    pageHitInfo: {
      get() {
        return `${this.pagePath} has ${} hit(s).`;
      }
      set(hits) {
        this.hits = hits;
      }
    },
  }
});

export { mongoose.model("PageHits", pageHitsSchema) };
```

Use the setter `pageHitInfo`:

```
const aboutPageHits = new PageHits({
  pagePath: '/about',
  pageHitInfo: 40,
});
```

Use the getter `pageHitInfo`:

``` console.log(aboutPageHits.pageHitInfo); // /about has 40 hit(s).

})
```