Queries

Mongoose models provide several static helper functions for CRUD operations. Each of these functions returns a mongoose `Query` object.

- `Model.deleteMany()`
- `Model.deleteOne()`
- `Model.find()`
- `Model.findById()`
- `Model.findByIdAndDelete()`
- `Model.findByIdAndRemove()`
- `Model.findByIdAndUpdate()`
- `Model.findOne()`
- `Model.findOneAndDelete()`
- `Model.findOneAndReplace()`
- `Model.findOneAndUpdate()`
- `Model.replaceOne()`
- `Model.updateMany()`
- `Model.updateOne()`

A mongoose query can be executed in one of two ways. First, if you pass in a `callback` function, Mongoose will execute the query asynchronously and pass the results to the `callback`.

A query also has a `.then()` function, and thus can be used as a promise.

Executing

When executing a query, you specify your query as a JSON document. The JSON document's syntax is the same as the MongoDB shell.

```
const Person = mongoose.model('Person', yourSchema);

// find each person with a last name matching 'Ghost', selecting the `name` and `occupation` fie
const person = await Person.findOne({ 'name.last': 'Ghost' }, 'name occupation');
```

```
// Prints "Space Ghost is a talk show host".
console.log('%s %s is a %s.', person.name.first, person.name.last, person.occupation);
```

What `person` is depends on the operation: For `findOne()` it is a potentially-null single document, `find()` a list of documents, `count()` the number of documents, `update()` the number of documents affected, etc. The API docs for Models provide more details.

Now let's look at what happens when no `await` is used:

```
// find each person with a last name matching 'Ghost'
const query = Person.findOne({ 'name.last': 'Ghost' });

// selecting the `name` and `occupation` fields
query.select('name occupation');

// execute the query at a later time
const person = await query.exec();
// Prints "Space Ghost is a talk show host."
console.log('%s %s is a %s.', person.name.first, person.name.last, person.occupation);
```

In the above code, the `query` variable is of type Query. A `Query` enables you to build up a query using chaining syntax, rather than specifying a JSON object. The below 2 examples are equivalent.

```
// With a JSON doc
await Person.
  find({
    occupation: /host/,
    'name.last': 'Ghost',
    age: { $gt: 17, $lt: 66 },
    likes: { $in: ['vaporizing', 'talking'] }
  }).
  limit(10).
  sort({ occupation: -1 }).
  select({ name: 1, occupation: 1 }).
  exec();

// Using query builder
await Person.
  find({ occupation: /host/ }).
  where('name.last').equals('Ghost').
  where('age').gt(17).lt(66).
  where('likes').in(['vaporizing', 'talking']).
  limit(10).
  sort('-occupation').
  select('name occupation').
  exec();
```

A full list of Query helper functions can be found in the API docs.

Queries are Not Promises

Mongoose queries are **not** promises. Queries are thenables, meaning they have a `.then()` method for async/await as a convenience. However, unlike promises, calling a query's `.then()` executes the query, so calling `then()` multiple times will throw an error.

```
const q = MyModel.updateMany({}, { isDeleted: true });

await q.then(() => console.log('Update 2'));
// Throws "Query was already executed: Test.updateMany({}, { isDeleted: true })"
await q.then(() => console.log('Update 3'));
```

References to other documents

There are no joins in MongoDB but sometimes we still want references to documents in other collections. This is where population comes in. Read more about how to include documents from other collections in your query results here.

Streaming

You can stream query results from MongoDB. You need to call the Query#cursor() function to return an instance of QueryCursor.

```
const cursor = Person.find({ occupation: /host/ }).cursor();

for (let doc = await cursor.next(); doc != null; doc = await cursor.next()) {
  console.log(doc); // Prints documents one at a time
}
```

Iterating through a Mongoose query using async iterators also creates a cursor.

```
for await (const doc of Person.find()) {
  console.log(doc); // Prints documents one at a time
}
```

Cursors are subject to cursor timeouts. By default, MongoDB will close your cursor after 10 minutes and subsequent `next()` calls will result in a `MongoServerError: cursor id 123 not found` error. To override this, set the `noCursorTimeout` option on your cursor.

```
// MongoDB won't automatically close this cursor after 10 minutes.
const cursor = Person.find().cursor().addCursorFlag('noCursorTimeout', true);
```

However, cursors can still time out because of session idle timeouts. So even a cursor with `noCursorTimeout` set will still time out after 30 minutes of inactivity. You can read more about working around session idle timeouts in the MongoDB documentation.

Versus Aggregation

Aggregation can do many of the same things that queries can. For example, below is how you can use `aggregate()` to find docs where `name.last = 'Ghost'`:

```
const docs = await Person.aggregate([{ $match: { 'name.last': 'Ghost' } }]);
```

However, just because you can use `aggregate()` doesn't mean you should. In general, you should use queries where possible, and only use `aggregate()` when you absolutely need to.

Unlike query results, Mongoose does **not** `hydrate()` aggregation results. Aggregation results are always POJOs, not Mongoose documents.

```
const docs = await Person.aggregate([{ $match: { 'name.last': 'Ghost' } }]);

docs[0] instanceof mongoose.Document; // false
```

Also, unlike query filters, Mongoose also doesn't cast aggregation pipelines. That means you're responsible for ensuring the values you pass in to an aggregation pipeline have the correct type.

```
const doc = await Person.findOne();

const idString = doc._id.toString();

// Finds the `Person`, because Mongoose casts `idString` to an ObjectId
const queryRes = await Person.findOne({ _id: idString });

// Does **not** find the `Person`, because Mongoose doesn't cast aggregation
// pipelines.
const aggRes = await Person.aggregate([{ $match: { _id: idString } }]);
```

Sorting

Sorting is how you can ensure your query results come back in the desired order.

```
const personSchema = new mongoose.Schema({
  age: Number
});

const Person = mongoose.model('Person', personSchema);
for (let i = 0; i < 10; i++) {
  await Person.create({ age: i });
}

await Person.find().sort({ age: -1 }); // returns age starting from 10 as the first entry
await Person.find().sort({ age: 1 }); // returns age starting from 0 as the first entry
```

When sorting with mutiple fields, the order of the sort keys determines what key MongoDB server sorts by first.

```
const personSchema = new mongoose.Schema({
  age: Number,
  name: String,
  weight: Number
});

const Person = mongoose.model('Person', personSchema);
const iterations = 5;
for (let i = 0; i < iterations; i++) {
  await Person.create({
    age: Math.abs(2 - i),
    name: 'Test' + i,
    weight: Math.floor(Math.random() * 100) + 1
  });
}

await Person.find().sort({ age: 1, weight: -1 }); // returns age starting from 0, but while keep
```

You can view the output of a single run of this block below. As you can see, age is sorted from 0 to 2 but when age is equal, sorts by weight.

```
[
  {
    _id: new ObjectId('63a335a6b9b6a7bfc186cb37'),
    age: 0,
    name: 'Test2',
    weight: 67,
    __v: 0
  },
  {
    _id: new ObjectId('63a335a6b9b6a7bfc186cb35'),
    age: 1,
    name: 'Test1',
    weight: 99,
    __v: 0
  },
  {
    _id: new ObjectId('63a335a6b9b6a7bfc186cb39'),
    age: 1,
    name: 'Test3',
    weight: 73,
    __v: 0
  },
  {
    _id: new ObjectId('63a335a6b9b6a7bfc186cb33'),
    age: 2,
    name: 'Test0',
    weight: 65,
    __v: 0
```

```
  },
  {
    _id: new ObjectId('63a335a6b9b6a7bfc186cb3b'),
    age: 2,
    name: 'Test4',
    weight: 62,
    __v: 0
  }
];
```

Next Up

Now that we've covered `Queries`, let's take a look at Validation.