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.findOneAndRemove()`
- `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
- Queries are Not Promises
- References to other documents
- Streaming
- Versus Aggregation

Executing

When executing a query with a `callback` function, 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` fielc
Person.findOne({ 'name.last': 'Ghost' }, 'name occupation', function (err, person) {
  if (err) return handleError(err);
  // Prints "Space Ghost is a talk show host".
  console.log('%s %s is a %s.', person.name.first, person.name.last,
    person.occupation);
});
```

Mongoose executed the query and passed the results to `callback`. All callbacks in Mongoose use the pattern: `callback(error, result)`. If an error occurs executing the query, the `error` parameter will contain an error document, and `result` will be null. If the query is successful, the `error` parameter will be null, and the `result` will be populated with the results of the query.

Anywhere a callback is passed to a query in Mongoose, the callback follows the pattern `callback(error, results)`. What `results` 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 detail on what is passed to the callbacks.

Now let's look at what happens when no `callback` is passed:

```
// 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
query.exec(function (err, person) {
  if (err) return handleError(err);
  // 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
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(callback);

// Using query builder
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(callback);
```

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

Queries are Not Promises

Mongoose queries are **not** promises. They have a `.then()` function for co and async/await as a convenience. However, unlike promises, calling a query's `.then()` can execute the query multiple times.

For example, the below code will execute 3 `updateMany()` calls, one because of the callback, and two because `.then()` is called twice.

```
const q = MyModel.updateMany({}, { isDeleted: true }, function() {
  console.log('Update 1');
});

q.then(() => console.log('Update 2'));
q.then(() => console.log('Update 3'));
```

Don't mix using callbacks and promises with queries, or you may end up with duplicate operations. That's because passing a callback to a query function immediately executes the query, and calling `then()` executes the query again.

Mixing promises and callbacks can lead to duplicate entries in arrays. For example, the below code inserts 2 entries into the `tags` array, **not** just 1.

```
const BlogPost = mongoose.model('BlogPost', new Schema({
  title: String,
  tags: [String]
}));

// Because there's both `await` **and** a callback, this `updateOne()` executes twice
// and thus pushes the same string into `tags` twice.
const update = { $push: { tags: ['javascript'] } };
await BlogPost.updateOne({ title: 'Introduction to Promises' }, update, (err, res) => {
  console.log(res);
});
```

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 } }])
```

Next Up

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