

Mongoose CRUD (Create, Read, Update, Delete)

03 DECEMBER 2015 on Mongoose (/tag/mongoose/), Database (/tag/database/), HTTP/API (/tag/http-api/), Level 4 (/tag/level-4/), Lesson (/tag/lesson/)

We've covered the basics of making queries in the MongoDB Shell (<https://coursework.vschoool.io/mongodb-overview/>), but since we'll mostly be writing our queries in our server's JavaScript code using Mongoose, it makes sense to understand how Mongoose does queries. We'll be covering basic CRUD (Create, Read, Update, Delete) operations.

With Mongoose, you can perform these operations wherever you want to in your code. Usually when we talk about CRUD, we're talking about it in tandem with the 4 most common HTTP methods, `GET`, `POST`, `PUT`, and `DELETE`, which map to the CRUD operations like so:

HTTP Method	CRUD Operation
GET	Read
POST	Create
PUT	Update
DELETE	Delete

Read

This is how we get items from MongoDB. Mongoose gives us 3 basic ways to get stuff from the database (`.find()`, `.findOne()`, and `.findById()`), and one advanced way (`.where()`).

```
.find([query], [callback])
```

Finds all documents in the database that match the query. If no query is given, it returns everything.

```
const Person = require("../models/person");

// .find() finds all instances in the database that match the query you pass
// It returns an array, even if there is only one item in the array.

// No query passed in means "find everything"
Person.find((err, people) => {
  // Note that this error doesn't mean nothing was found,
  // it means the database had an error while searching, hence the error
  if (err) return res.status(500).send(err)
  // send the list of all people
  return res.status(200).send(people);
});

// If query IS passed into .find(), filters by the query parameters
Person.find({name: "John James", age: 36}, (err, people) =>{
  if (err) return res.status(500).send(err)

  // send the list of all people in database with name of "John James"
  // Very possible this will be an array with just one Person object
  return res.status(200).send(people);
});
```

```
.findOne([query], [fieldsToReturn], [callback])
```

Finds one object from the database. If your query matches more than one item in the database, it still only returns the first one it finds.

```
Kitten.findOne(
  // query
  {color: "white", name: "Dr. Miffles", age: 1},

  // Only return an object with the "name" and "owner" fields. "_id"
  // is included by default, so you'll need to remove it if you don't
  {name: true, owner: true},

  // callback function
  (err, kitten) => {
    if (err) return res.status(200).send(err)
    return res.status(200).send(kitten)
  }
);
```

With `.findOne()`:

- If you don't provide a `query`, it will just return the first Kitten in the database.
- If you don't provide a `fieldsToReturn`, it will return the entire object.
- `fieldsToReturn` can also be in the form of a string with spaces between the field names, e.g. `"name age owner"`, instead of an object.

`.findById(id, [fieldsToReturn], [callback])`

Finds a single object in the database by the provided `id`.

```
// Common RESTful way to get the Id is from the url params in req.params
Kitten.findById(req.params.kittenId, (err, kitten) => {
  if (err) return res.status(500).send(err)
  return res.status(200).send(kitten)
});
```

`.where(selector)`

This one is powerful, but a bit more confusing. It allows us to do more complex queries to the database. So far we've done things like "find a Kitten whose name is Dr. Miffles" **AND** whose color is white **AND** who is 1 year old.

But what if we wanted to query for all Kittens in the Database who were between the ages of 1 and 4? This is when `.where()` comes in.

Calling `.where()` on a Mongoose Model (e.g. `Kitten`) actually returns a Mongoose "Query" object. In order to actually execute the query, we have to call the `.exec()` method and pass in our usual callback. Example:

```
Kitten.where("age").gte(1).lte(4).exec((err, kittens) => {  
  // Do stuff  
});
```

What's nice about these queries is that it reads almost like English: "Find all Kittens where the age is greater than or equal to 1, and less than or equal to 4, then execute the query."

Create

This is how we can create new items in the database. This will commonly be from an HTTP `POST` request, although you can do this anywhere you want.

```
const Todo = require("../models/todo");

// Assuming this is from a POST request and the body of the
// request contained the JSON of the new "todo" item to be saved
const newTodoObj = new Todo(req.body);
newTodoObj.save(err => {
  if (err) return res.status(500).send(err);
  return res.status(200).send(newTodoObj);
});
```

That's about it! Create the new object just like you would if you were creating a JavaScript object from an object function constructor

(<http://www.javascriptkit.com/javatutors/oopjs2.shtml>), then call the `.save()` method on that object.

One common operation apps will make is called `findOrCreate`, where it will look for an existing instance in the database and return it if it exists, and create it if it doesn't exist. Mongoose doesn't have a built-in method for doing that, but there is a third-party package called `mongoose-findorcreate` (<https://github.com/drudge/mongoose-findorcreate>) that you can install as a plugin to your model to include that as a static method on all models it is plugged in to.

Update

This is just a combination of "read" and "create", but instead of creating a new one with `const newTodoObj = new Todo(...)`, we query the database and send a change to be made using `findByIdAndUpdate`. Make sure to read the comments below explaining each of the parts:

```
const Todo = require("../models/todo");

// This would likely be inside of a PUT request, since we're updating
// Find the existing resource by ID
Todo.findByIdAndUpdate(
  // the id of the item to find
  req.params.todoId,

  // the change to be made. Mongoose will smartly combine your existing
  // document with this change, which allows for partial updates to
  req.body,

  // an option that asks mongoose to return the updated version
  // of the document instead of the pre-updated one.
  {new: true},

  // the callback function
  (err, todo) => {
    // Handle any possible database errors
    if (err) return res.status(500).send(err);
    return res.send(todo);
  }
)
```

There are a few different ways you could structure your `PUT` request. Above, we opted for the use of the Mongoose shortcut method `findByIdAndUpdate` (there's also a

`findOneAndUpdate`). Check out the documentation here:

`findOneAndUpdate()`

(http://mongoosejs.com/docs/api.html#model_Model.findOneAndUpdate), `findByIdAndUpdate()`

(http://mongoosejs.com/docs/api.html#model_Model.findByIdAndUpdate).

Another option would be to use a `findOne` or `findById`, make the changes to the properties manually, then use `.save` to save the change. The benefit of doing it this way is that you have more

control over the changes being made, but at the expense of having to make two trips to the database (one to retrieve the document, another to save it). Using `findByIdAndUpdate` combines these two trips into one, but also makes it a little harder to make granular modifications. It also bypasses any model "hooks", like a "pre-save" hook. But if that isn't a concern to you, `findByIdAndUpdate` and `findOneAndUpdate` are great shortcut methods to use.

Delete

Similar to the "Update" section above, you can go about deleting a document from the database by first finding it, then running the `.remove()` method on the found document. Also similar to the updating section above, Mongoose v4.0 introduced some helper methods – `.findOneAndRemove()` and `.findByIdAndRemove()` – which is what we'll show in the example below.

```
// The "todo" in this callback function represents the document that was found
// It allows you to pass a reference back to the client in case they need it
Todo.findByIdAndRemove(req.params.todoId, (err, todo) => {
  // As always, handle any potential errors:
  if (err) return res.status(500).send(err);
  // We'll create a simple object to send back with a message and the id of the deleted document
  // You can really do this however you want, though.
  const response = {
    message: "Todo successfully deleted",
    id: todo._id
  };
  return res.status(200).send(response);
});
```

Conclusion

You should definitely spend some time researching different CRUD operations using Mongoose. There are a lot of nuances you will need to learn to become an expert at using it, and the only way you'll be able to learn those nuances is by spending time researching and using Mongoose in your own projects.

Mongoose has lots of really fine-grained controls it allows, such as defining static methods for your models (so that every object created is able to run a specific method, similar to adding a method to a JavaScript "class", A.K.A. function constructor), adding pre/post hooks into your schema (so that you can run certain code before and/or after an item is created from the schema, deleted, updated, etc.), adding options like there's no tomorrow, etc. Check out some of the following Mongoose documentation to become more familiar with the intricacies it has.

- Mongoose API Docs (<http://mongoosejs.com/docs/api.html>)
- Mongoose Schema Guide (<http://mongoosejs.com/docs/guide.html>)
- Mongoose Models Guide (<http://mongoosejs.com/docs/models.html>)
- Mongoose Documents Guide (<http://mongoosejs.com/docs/documents.html>)
- Mongoose Queries Guide (<http://mongoosejs.com/docs/queries.html>)


Good luck, and go create something awesome!

[Bob Ziroll \(/author/bob/\)](#)

[\(/author/bob/\)](#)

Share this post

Read more posts (/author/bob/) by this author.

 (<https://twitter.com/text=Mongoose%20CRUD>)

(/mongoose-
schemas/)

(/express-params-
and-query/)

READ THIS NEXT

YOU MIGHT ENJOY

Mongoose Schema Basics

Understanding
Schemas A "Schema"
can be a tough thing to
understand at first, but
in reality it's...

Express req.params and req.query

When you're first
learning about the
Request/Response
cycle, there's a fairly
confusing topic that
you'll run into
frequently:...