Mongoose 101

Published: Dec 11, 2019

Mongoose is a library that makes MongoDB easier to use. It does two things:

1. It gives structure to MongoDB Collections
2. It gives you helpful methods to use

In this article, you'll learn how to use Mongoose on a basic level.

Prerequisites

I assume you have done the following:

1. You have installed MongoDB on your computer
2. You know how to set up a local MongoDB connection
3. You know how to see the data you have in your database
4. You know what are "collections" in MongoDB

If you don't know any of these, please read "How to set up a local MongoDB connection" before you continue.

I also assume you know how to use MongoDB to create a simple CRUD app. If you don't know how to do this, please read "How to build a CRUD app with Node, Express, and

MongoDB” before you continue.

Connecting to a database

First, you need to download Mongoose.

```
npm install mongoose --save
```

You can connect to a database with the `connect` method. Let’s say we want to connect to a database called `street-fighters`. Here’s the code you need:

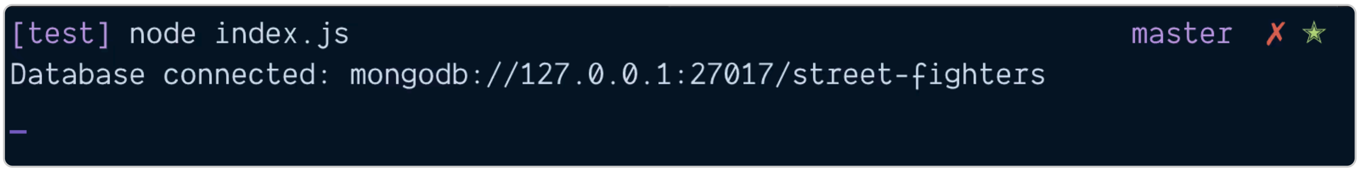
```
const mongoose = require('mongoose')
const url = 'mongodb://127.0.0.1:27017/street-fighters'

mongoose.connect(url, { useNewUrlParser: true })
```

We want to know whether our connection has succeeded or failed. This helps us with debugging.

To check whether the connection has succeeded, we can use the `open` event. To check whether the connection failed, we use the `error` event.

```
const db = mongoose.connection
db.once('open', _ => {
  console.log('Database connected:', url)
})

db.on('error', err => {
  console.error('connection error:', err)
})
```

Try connecting to the database. You should see a log like this:



```
[test] node index.js                                      master  ✗ ✭
Database connected: mongodb://127.0.0.1:27017/street-fighters
_
```

Creating a Model

In Mongoose, you need to **use models to create, read, update, or delete items** from a MongoDB collection.

To create a Model, **you need to create a Schema**. A Schema lets you** define the structure of an entry** in the collection. This entry is also called a document.

Here's how you create a schema:

```
const mongoose = require('mongoose')
const Schema = mongoose.Schema

const schema = new Schema({
  // ...
})
```

You can use 10 different kinds of values in a Schema. Most of the time, you'll use these six:

- String
- Number
- Boolean
- Array
- Date
- ObjectId

Let's put this into practice.

Say we want to create characters for our Street Fighter database.

In Mongoose, it's a normal practice to **put each model in its own file.** So we will create a `Character.js` file first. This `Character.js` file will be placed in the `models` folder.

```
project/
    |- models/
        |- Character.js
```

In `Character.js`, we create a `characterSchema`.

```
const mongoose = require('mongoose')
const Schema = mongoose.Schema

const characterSchema = new Schema({
  // ...
})
```

Let's say we want to save two things into the database:

1. Name of the character
2. Name of their ultimate move

Both can be represented with Strings.

```
const mongoose = require('mongoose')
const Schema = mongoose.Schema

const characterSchema = new Schema({
  name: String,
  ultimate: String,
})
```

Once we've created `characterSchema`, we can use mongoose's `model` method to create the model.

```
module.exports = mongoose.model('Character', characterSchema)
```

Creating a character

Let's say you have a file called `index.js`. This is where we'll perform Mongoose operations for this tutorial.

```
project/
    |- index.js
    |- models/
        |- Character.js
```

First, you need to load the Character model. You can do this with `require`.

```
const Character = require('./models/Character')
```

Let’s say you want to create a character called Ryu. Ryu has an ultimate move called “Shinku Hadoken”.

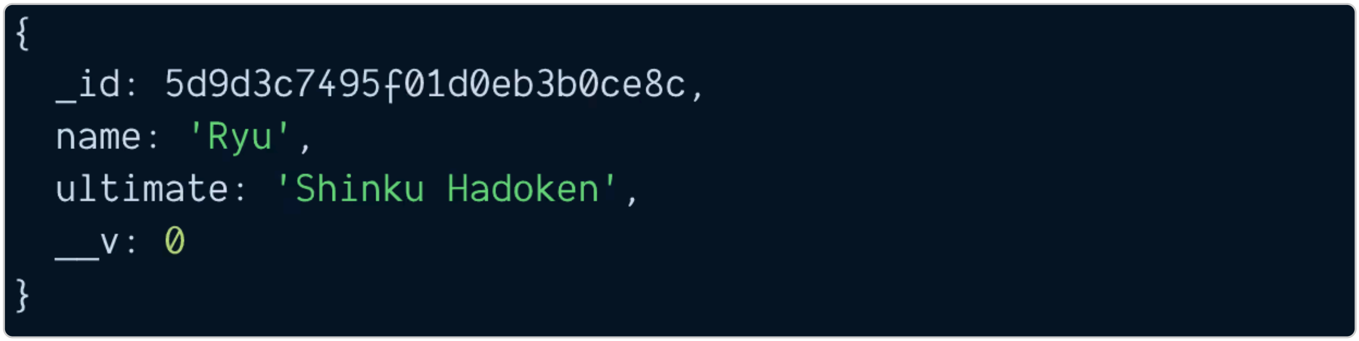
To create Ryu, you use the `new`, followed by your model. In this case, it’s `new Character`.

```
const ryu = new Character({
  name: 'Ryu',
  ultimate: 'Shinku Hadoken',
})
```

`new Character` creates the character in memory. It has not been saved to the database yet. **To save to the database, you can run the `save` method.**

```
ryu.save(function (error, document) {
  if (error) console.error(error)
  console.log(document)
})
```

If you run the code above, you should see this in the console.



```
{
  _id: 5d9d3c7495f01d0eb3b0ce8c,
  name: 'Ryu',
  ultimate: 'Shinku Hadoken',
  __v: 0
}
```

Promises and Async/await

Mongoose supports promises. It lets you write nicer code like this:

```
// This does the same thing as above
function saveCharacter(character) {
  const c = new Character(character)
  return c.save()
}

saveCharacter({
  name: 'Ryu',
  ultimate: 'Shinku Hadoken',
})
  .then(doc => {
    console.log(doc)
  })
  .catch(error => {
    console.error(error)
  })
```

You can also use the `await` keyword if you have an asynchronous function.

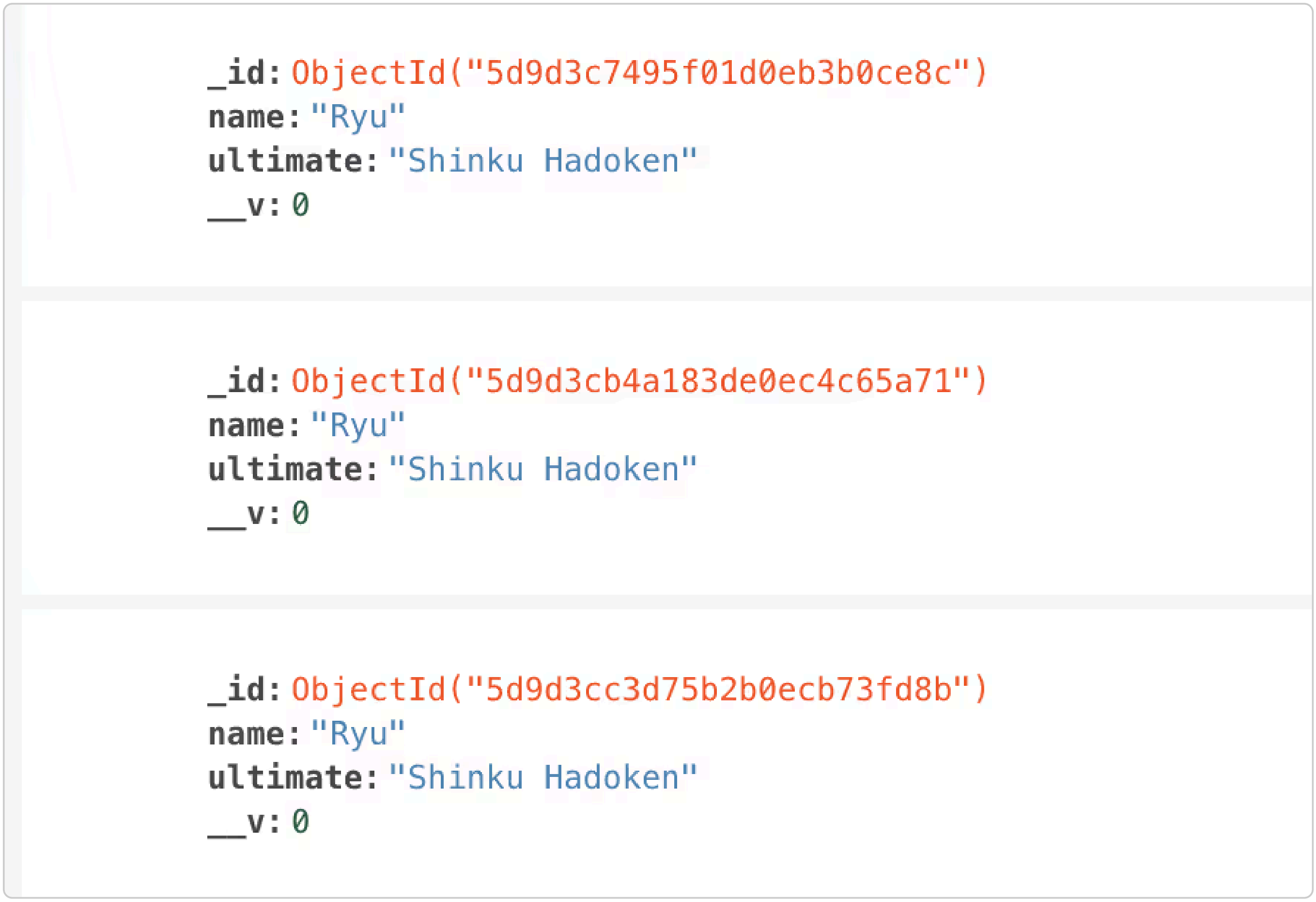
If the Promise or Async/Await code looks foreign to you, I recommend reading "JavaScript async and await" before continuing with this tutorial.

```
async function runCode() {
  const ryu = new Character({
    name: 'Ryu',
    ultimate: 'Shinku Hadoken',
  })

  const doc = await ryu.save()
  console.log(doc)
}

runCode().catch(error => {
  console.error(error)
})
```

Note: I'll use the async/await format for the rest of the tutorial.

Uniqueness

Mongoose adds a new character to the database each you use `new Character` and `save`. If you run the code(s) above three times, you'd expect to see three Ryus in the database.



```
_id: ObjectId("5d9d3c7495f01d0eb3b0ce8c")
name: "Ryu"
ultimate: "Shinku Hadoken"
__v: 0
```

```
_id: ObjectId("5d9d3cb4a183de0ec4c65a71")
name: "Ryu"
ultimate: "Shinku Hadoken"
__v: 0
```

```
_id: ObjectId("5d9d3cc3d75b2b0ecb73fd8b")
name: "Ryu"
ultimate: "Shinku Hadoken"
__v: 0
```

We don't want to have three Ryus in the database. We want to have **ONE Ryu only**. To do this, we can use the **unique** option.

```
const characterSchema = new Schema({
  name: { type: String, unique: true },
  ultimate: String,
})
```

The `unique` option **creates a unique index**. It ensures we cannot have two documents with the same value (for `name` in this case).

For `unique` to work properly, you need to **clear the Characters collection**. To clear the Characters collection, you can use this:

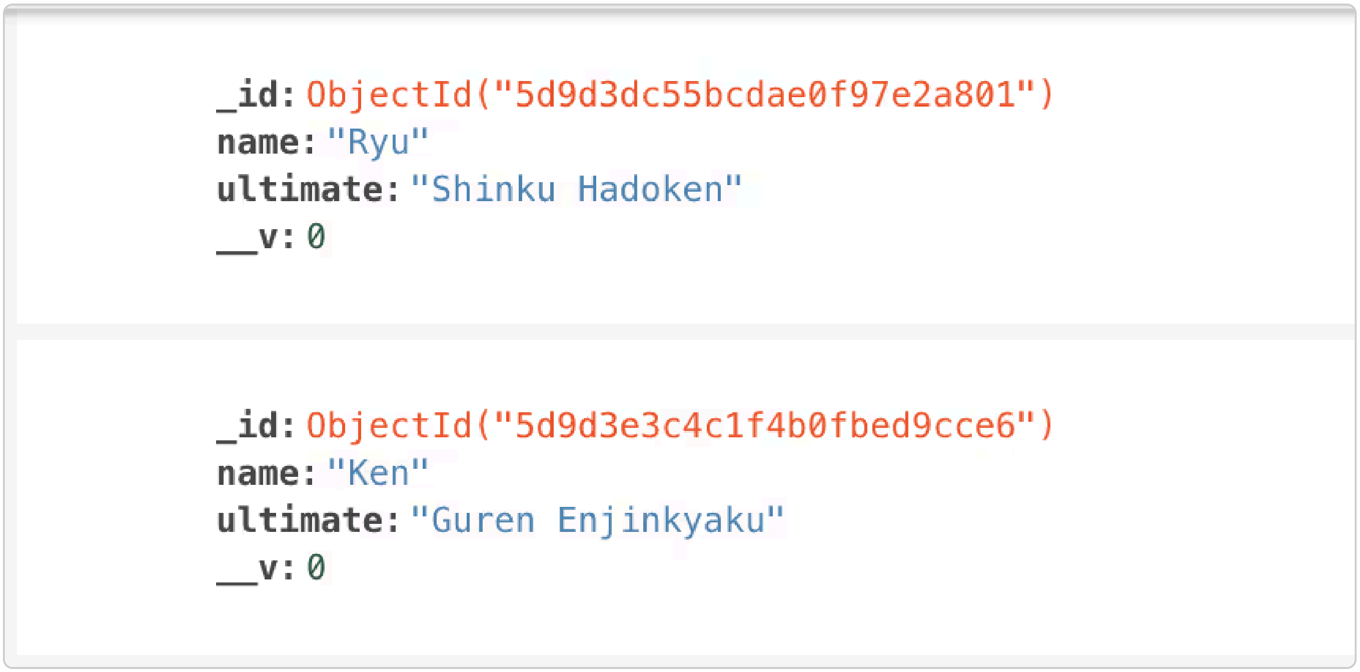
```
await Character.deleteMany({})
```

Try to add two Ryus into the database now. You'll get an `E11000 duplicate key error`. You won't be able to save the second Ryu.

```
MongoError: E11000 duplicate key error collection: street-fighters.characters index: name_1 dup key
: { name: "Ryu" }
    at Function.create (/Users/zellwk/Desktop/test/node_modules/mongodb/lib/core/error.js:44:12)
    at toError (/Users/zellwk/Desktop/test/node_modules/mongodb/lib/utils.js:150:22)
    at /Users/zellwk/Desktop/test/node_modules/mongodb/lib/operations/common_functions.js:266:39
    at handler (/Users/zellwk/Desktop/test/node_modules/mongodb/lib/core/sdam/topology.js:973:24)
    at /Users/zellwk/Desktop/test/node_modules/mongodb/lib/core/sdam/server.js:437:5
    at /Users/zellwk/Desktop/test/node_modules/mongodb/lib/core/connection/pool.js:420:18
    at processTicksAndRejections (internal/process/task_queues.js:75:11) {
  driver: true,
  name: 'MongoError',
  index: 0,
  code: 11000,
  keyPattern: { name: 1 },
  keyValue: { name: 'Ryu' },
  errmsg: 'E11000 duplicate key error collection: street-fighters.characters index: name_1 dup key:
 { name: "Ryu" }',
  [Symbol(mongoErrorContextSymbol)]: {}
}
```

Let's add another character into the database before we continue the rest of the tutorial.

```
const ken = new Character({
  name: 'Ken',
  ultimate: 'Guren Enjinkyaku',
})

await ken.save()
```



```
_id: ObjectId("5d9d3dc55bcdae0f97e2a801")
name: "Ryu"
ultimate: "Shinku Hadoken"
__v: 0
```

```
_id: ObjectId("5d9d3e3c4c1f4b0fbed9cce6")
name: "Ken"
ultimate: "Guren Enjinkyaku"
__v: 0
```

Retrieving a character

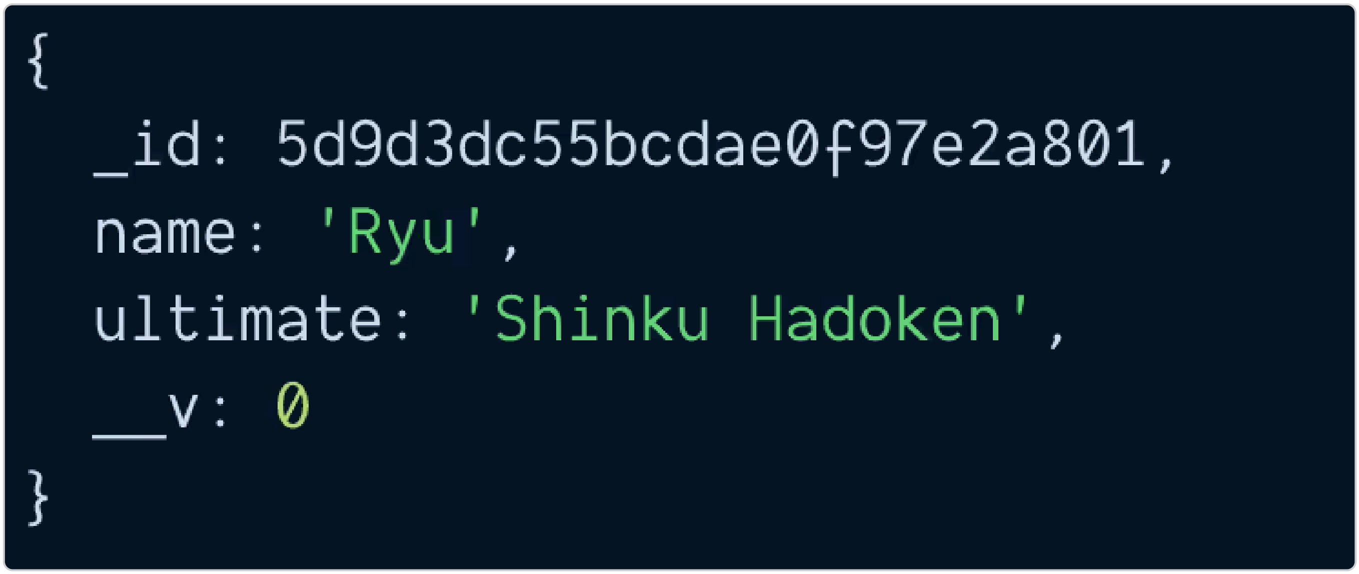
Mongoose gives you two methods to find stuff from MongoDB.

1. `findOne`: Gets one document.
2. `find`: Gets an array of documents

FINDONE

`findOne` **returns the first document** it finds. You can specify any property to search for. Let's search for `Ryu`:

```
const ryu = await Character.findOne({ name: 'Ryu' })
console.log(ryu)
```

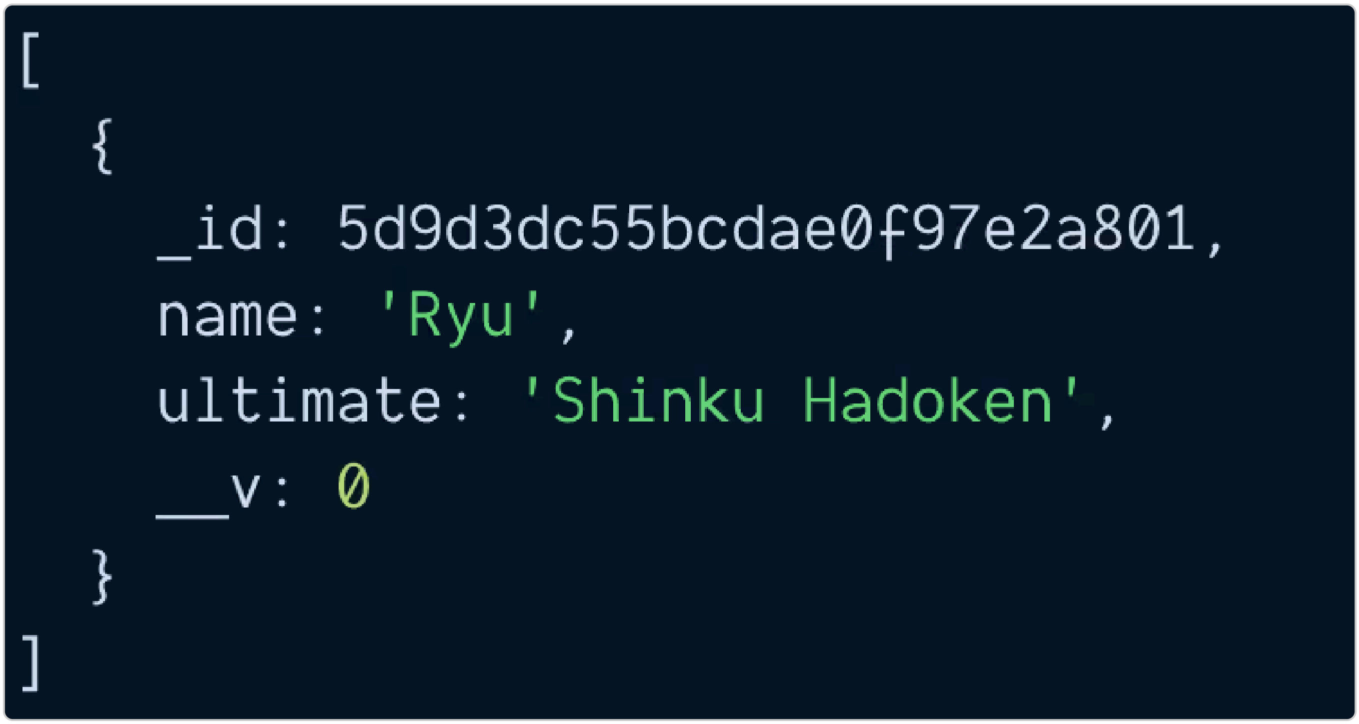


```
{
  _id: 5d9d3dc55bcdae0f97e2a801,
  name: 'Ryu',
  ultimate: 'Shinku Hadoken',
  __v: 0
}
```

FIND

`find` **returns an array** of documents. If you specify a property to search for, it'll return documents that match your query.

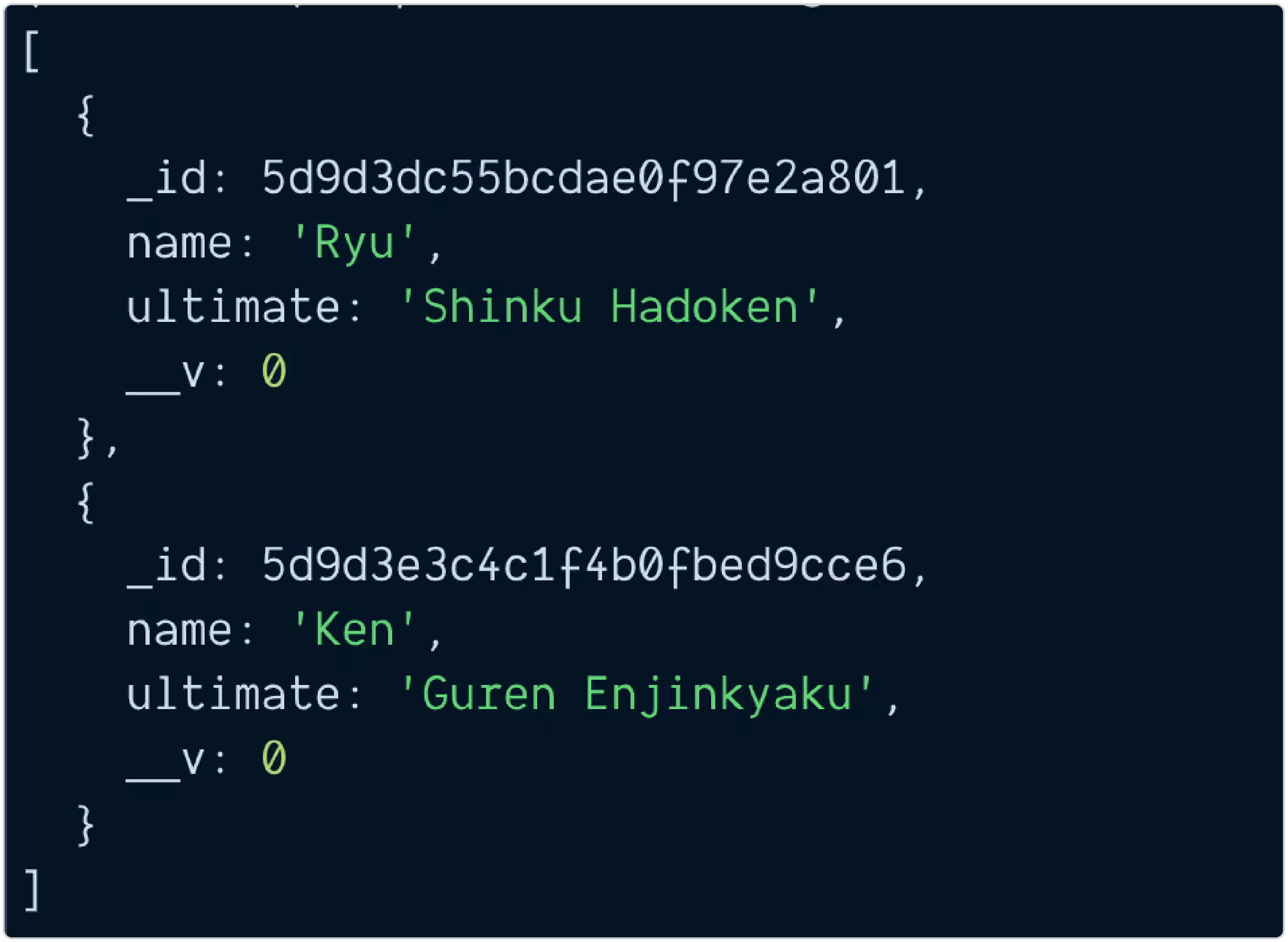
```
const chars = await Character.find({ name: 'Ryu' })
console.log(chars)
```



```
[
  {
    _id: 5d9d3dc55bcdae0f97e2a801,
    name: 'Ryu',
    ultimate: 'Shinku Hadoken',
    __v: 0
  }
]
```

If you did not specify any properties to search for, it'll return an array that contains all documents in the collection.

```
const chars = await Character.find()
console.log(chars)
```



```
[
  {
    _id: 5d9d3dc55bcdae0f97e2a801,
    name: 'Ryu',
    ultimate: 'Shinku Hadoken',
    __v: 0
  },
  {
    _id: 5d9d3e3c4c1f4b0fbed9cce6,
    name: 'Ken',
    ultimate: 'Guren Enjinkyaku',
    __v: 0
  }
]
```

Updating a Character

Let's say Ryu has three special moves:

1. Hadoken
2. Shoryuken
3. Tatsumaki Senpukyaku

We want to add these special moves into the database. First, we need to update our `CharacterSchema`.

```
const characterSchema = new Schema({
  name: { type: String, unique: true },
  specials: Array,
  ultimate: String,
})
```

Then, we use one of these two ways to update a character:

1. Use `findOne`, then use `save`
2. Use `findOneAndUpdate`

findOne and save

First, we use `findOne` to get Ryu.

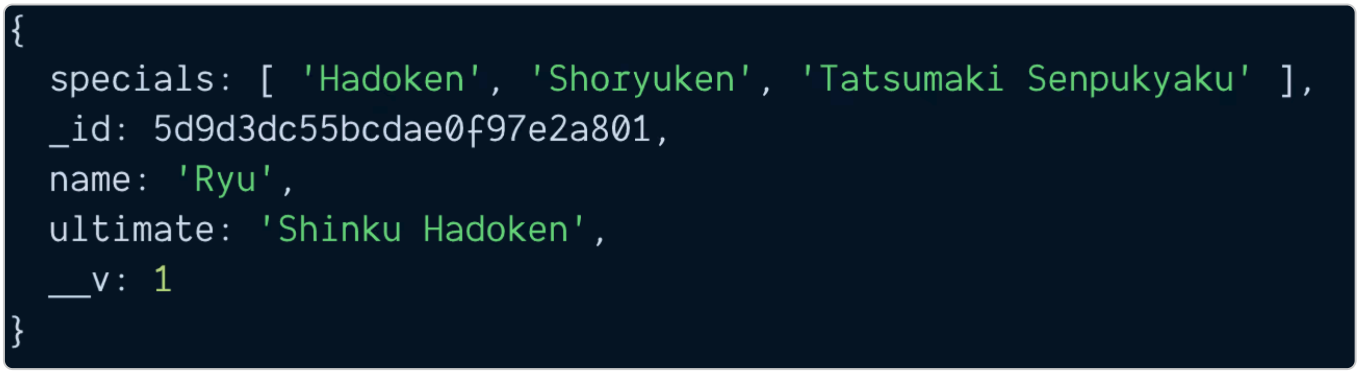
```
const ryu = await Character.findOne({ name: 'Ryu' })
console.log(ryu)
```

Then, we update Ryu to include his special moves.

```
const ryu = await Character.findOne({ name: 'Ryu' })
ryu.specials = ['Hadoken', 'Shoryuken', 'Tatsumaki Senpukyaku']
```

After we modified `ryu`, we run `save`.

```
const ryu = await Character.findOne({ name: 'Ryu' })
ryu.specials = ['Hadoken', 'Shoryuken', 'Tatsumaki Senpukyaku']

const doc = await ryu.save()
console.log(doc)
```



```
{
  specials: [ 'Hadoken', 'Shoryuken', 'Tatsumaki Senpukyaku' ],
  _id: 5d9d3dc55bcdae0f97e2a801,
  name: 'Ryu',
  ultimate: 'Shinku Hadoken',
  __v: 1
}
```

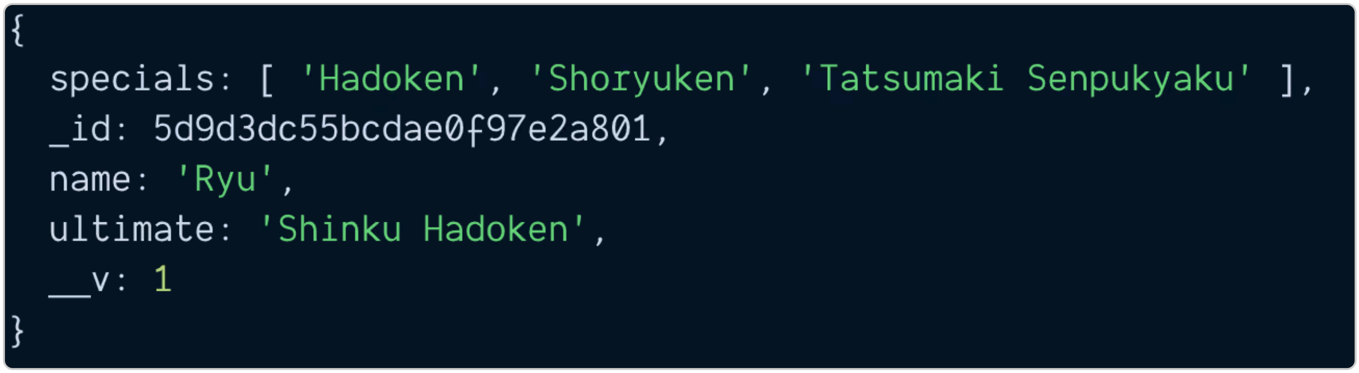
findOneAndUpdate

`findOneAndUpdate` is the same as MongoDB's `findOneAndModify` method.

Here, you search for Ryu and pass the fields you want to update at the same time.

```
// Syntax
await findOneAndUpdate(filter, update)
```

```
// Usage
const doc = await Character.findOneAndUpdate(
  { name: 'Ryu' },
  {
    specials: ['Hadoken', 'Shoryuken', 'Tatsumaki Senpukyaku'],
  },
)

console.log(doc)
```



```
{
  specials: [ 'Hadoken', 'Shoryuken', 'Tatsumaki Senpukyaku' ],
  _id: 5d9d3dc55bcdae0f97e2a801,
  name: 'Ryu',
  ultimate: 'Shinku Hadoken',
  __v: 1
}
```

Difference between findOne + save vs findOneAndUpdate

Two major differences.

First, the **syntax for** `findOne` + `save` **is easier to read** than `findOneAndUpdate`.

Second, `findOneAndUpdate` does not trigger the `save` middleware.

I'll choose `findOne` **+** `save` over `findOneAndUpdate` anytime because of these two differences.

Deleting a character

There are two ways to delete a character:

1. `findOne` + `remove`
2. `findOneAndDelete`

Using findOne + remove

```
const ryu = await Character.findOne({ name: 'Ryu' })
const deleted = await ryu.remove()
```

Using findOneAndDelete

```
const deleted = await Character.findOneAndDelete({ name: 'Ken' })
```

Quick summary

You learned how to use Mongoose to:

1. Connect to a database
2. Create, Read, Update, and Delete documents

< What to do if you're struggling with a Bootcamp

Mongoose 101: Working with subdocuments >

Want to become a better Frontend Developer?

Don't worry about where to start. I'll send you a library of articles frontend developers have found useful!

- 60+ CSS articles

- 90+ JavaScript articles

I'll also send you one article every week to help you improve your FED skills crazy fast!

First Name

Email address

Send it to me

© 2013 - 2024 Zell Liew. All rights reserved · Terms