**Mongoose Notes**

Next install Mongoose from the command line using npm:

npm install mongoose --save

Now say we like fuzzy kittens and want to record every kitten we ever meet in MongoDB. The first thing we need to do is include mongoose in our project and open a connection to the test database on our locally running instance of MongoDB.

*// getting-started.js*

**const** mongoose = require('mongoose');

**main**().**catch**(err => console.**log**(err));

**async** **function** **main**() {

**await** mongoose.**connect**('mongodb://127.0.0.1:27017/test');

*// use `await mongoose.connect('mongodb://user:password@127.0.0.1:27017/test');` if your database has auth enabled*

}

For brevity, let's assume that all following code is within the main() function.

With Mongoose, everything is derived from a [Schema](https://mongoosejs.com/docs/guide.html). Let's get a reference to it and define our kittens.

**const** kittySchema = **new** mongoose.**Schema**({

name: **String**

});

So far so good. We've got a schema with one property, name, which will be a String. The next step is compiling our schema into a [Model](https://mongoosejs.com/docs/models.html).

**const** **Kitten** = mongoose.**model**('Kitten', kittySchema);

A model is a class with which we construct documents. In this case, each document will be a kitten with properties and behaviors as declared in our schema. Let's create a kitten document representing the little guy we just met on the sidewalk outside:

**const** silence = **new** **Kitten**({ name: 'Silence' });

console.**log**(silence.name); *// 'Silence'*

Kittens can meow, so let's take a look at how to add "speak" functionality to our documents:

*// NOTE: methods must be added to the schema before compiling it with mongoose.model()*

kittySchema.methods.speak = **function** **speak**() {

**const** greeting = this.name

? 'Meow name is ' + this.name

: 'I don\'t have a name';

console.**log**(greeting);

};

**const** **Kitten** = mongoose.**model**('Kitten', kittySchema);

Functions added to the methods property of a schema get compiled into the Model prototype and exposed on each document instance:

**const** fluffy = **new** **Kitten**({ name: 'fluffy' });

fluffy.**speak**(); *// "Meow name is fluffy"*

We have talking kittens! But we still haven't saved anything to MongoDB. Each document can be saved to the database by calling its [save](https://mongoosejs.com/docs/api/model.html#model_Model-save) method. The first argument to the callback will be an error if any occurred.

**await** fluffy.**save**();

fluffy.**speak**();

Say time goes by and we want to display all the kittens we've seen. We can access all of the kitten documents through our Kitten [model](https://mongoosejs.com/docs/models.html).

**const** kittens = **await** **Kitten**.**find**();

console.**log**(kittens);

We just logged all of the kittens in our db to the console. If we want to filter our kittens by name, Mongoose supports MongoDBs rich [querying](https://mongoosejs.com/docs/queries.html) syntax.

**await** **Kitten**.**find**({ name: /^fluff/ });

This performs a search for all documents with a name property that begins with "fluff" and returns the result as an array of kittens to the callback.

**Schemas**

[**Defining your schema**](https://mongoosejs.com/docs/guide.html#definition)

Everything in Mongoose starts with a Schema. Each schema maps to a MongoDB collection and defines the shape of the documents within that collection.

**import** mongoose **from** 'mongoose';

**const** {**Schema**} = mongoose;

**const** blogSchema = **new** **Schema**({

title: **String**, *// String is shorthand for {type: String}*

author: **String**,

body: **String**,

comments: [{ body: **String**, date: **Date** }],

date: { type: **Date**, default: **Date**.now },

hidden: **Boolean**,

meta: {

votes: **Number**,

favs: **Number**

}

});

If you want to add additional keys later, use the [Schema#add](https://mongoosejs.com/docs/api/schema.html" \l "schema_Schema-add) method.

Schemas not only define the structure of your document and casting of properties, they also define document [**instance methods**](https://mongoosejs.com/docs/guide.html#methods)**,**[**static Model methods**](https://mongoosejs.com/docs/guide.html#statics)**,**[**compound indexes**](https://mongoosejs.com/docs/guide.html#indexes)**, and document lifecycle hooks called**[**middleware**](https://mongoosejs.com/docs/middleware.html)**.**

[**Creating a model**](https://mongoosejs.com/docs/guide.html#models)

To use our schema definition, we need to convert our blogSchema into a [Model](https://mongoosejs.com/docs/models.html) we can work with. To do so, we pass it into mongoose.model(modelName, schema):

**const** **Blog** = mongoose.**model**('Blog', blogSchema);

*// ready to go!*

[**Ids**](https://mongoosejs.com/docs/guide.html#_id)

By default, Mongoose adds an \_id property to your schemas.

**const** schema = **new** **Schema**();

schema.**path**('\_id'); *// ObjectId { ... }*

When you create a new document with the automatically added \_id property, Mongoose creates a new [\_id of type ObjectId](https://masteringjs.io/tutorials/mongoose/objectid) to your document.

**const** **Model** = mongoose.**model**('Test', schema);

**const** doc = **new** **Model**();

doc.\_id **instanceof** mongoose.Types.ObjectId; *// true*

You can also overwrite Mongoose's default \_id with your own \_id. Just be careful: Mongoose will refuse to save a document that doesn't have an \_id, so you're responsible for setting \_id if you define your own \_id path.

**const** schema = **new** **Schema**({ \_id: **Number** });

**const** **Model** = mongoose.**model**('Test', schema);

**const** doc = **new** **Model**();

**await** doc.**save**(); *// Throws "document must have an \_id before saving"*

doc.\_id = 1;

**await** doc.**save**(); *// works*

[**Instance methods**](https://mongoosejs.com/docs/guide.html#methods)

Instances of Models are [documents](https://mongoosejs.com/docs/documents.html). Documents have many of their own [built-in instance methods](https://mongoosejs.com/docs/api/document.html). We may also define our own custom document instance methods.

*// define a schema*

**const** animalSchema = **new** **Schema**({ name: **String**, type: **String** },

{

*// Assign a function to the "methods" object of our animalSchema through schema options.*

*// By following this approach, there is no need to create a separate TS type to define the type of the instance functions.*

methods: {

**findSimilarTypes**(cb) {

**return** mongoose.**model**('Animal').**find**({ type: this.type }, cb);

}

}

});

*// Or, assign a function to the "methods" object of our animalSchema*

animalSchema.methods.findSimilarTypes = **function**(cb) {

**return** mongoose.**model**('Animal').**find**({ type: this.type }, cb);

};

Now all of our animal instances have a findSimilarTypes method available to them.

**const** **Animal** = mongoose.**model**('Animal', animalSchema);

**const** dog = **new** **Animal**({ type: 'dog' });

dog.**findSimilarTypes**((err, dogs) => {

console.**log**(dogs); *// woof*

});

* Overwriting a default mongoose document method may lead to unpredictable results. See [this](https://mongoosejs.com/docs/api/schema.html#schema_Schema-reserved) for more details.
* The example above uses the Schema.methods object directly to save an instance method. You can also use the Schema.method() helper as described [here](https://mongoosejs.com/docs/api/schema.html#schema_Schema-method).
* Do **not** declare methods using ES6 arrow functions (=>). Arrow functions [explicitly prevent binding this](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions#No_binding_of_this), so your method will **not** have access to the document and the above examples will not work.