MongoDB and Mongoose

MongoDB is a database that stores data records (documents) for use by an application. Mongo is a non-relational, "NoSQL" database. This means Mongo stores all data associated within one record, instead of storing it across many preset tables as in a SQL database. Some benefits of this storage model are:

* Scalability: by default, non-relational databases are split (or "sharded") across many systems instead of only one. This makes it easier to improve performance at a lower cost.
* Flexibility: new datasets and properties can be added to a document without the need to make a new table for that data.
* Replication: copies of the database run in parallel so if one goes down, one of the copies becomes the new primary data source.

While there are many non-relational databases, Mongo's use of JSON as its document storage structure makes it a logical choice when learning backend JavaScript. Accessing documents and their properties is like accessing objects in JavaScript.

Mongoose.js is an npm module for Node.js that allows you to write objects for Mongo as you would in JavaScript. This can make is easier to construct documents for storage in Mongo.

MongoDB and Mongoose - Create a Model

First of all we need a Schema. Each schema maps to a MongoDB collection. It defines the shape of the documents within that collection.

Schemas are building block for Models. They can be nested to create complex models, but in this case we’ll keep things simple.

A model allows you to create instances of your objects, called documents.

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

**var** Schema = mongoose.Schema;

**var** blogSchema = **new** Schema({

title: String,

author: String,

body: String,

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

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

hidden: Boolean,

meta: {

votes: Number,

favs: Number

}

});

Each key in our code blogSchema defines a property in our documents which will be cast to its associated [SchemaType](https://mongoosejs.com/docs/api.html#schematype_SchemaType). For example, we've defined a property title which will be cast to the [String](https://mongoosejs.com/docs/api.html#schema-string-js) SchemaType and property date which will be cast to a Date SchemaType. Keys may also be assigned nested objects containing further key/type definitions like the meta property above.

The permitted SchemaTypes are:

* String
* Number
* Date
* Buffer
* Boolean
* Mixed
* ObjectId
* Array
* Decimal128
* Map

**INSTANCE 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.html#document-js). We may also define our own custom document instance methods too.

*// define a schema*

**var** animalSchema = **new** Schema({ name: String, type: String });

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

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

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

};

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

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

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

dog.findSimilarTypes(**function**(err, dogs) {

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

});

## MongoDB and Mongoose - Create Many Records with model.create()(MongoDB2)

Sometimes you need to create many instances of your models, e.g. when seeding a database with initial data. Model.create() takes an array of objects like [{name: 'John', ...}, {...}, ...] as the first argument, and saves them all in the db. Create many people with Model.create(), using the function argument arrayOfPeople.

## MongoDB and Mongoose - Use model.find() to Search Your Database(MongoDB2)

Find all the people having a given name, using Model.find() -> [Person]

In its simplest usage, Model.find() accepts a query document (a JSON object ) as the first argument, then a callback. It returns an array of matches. It supports an extremely wide range of search options. Check it in the docs. Use the function argument personName as search key.

## MongoDB and Mongoose - Use model.findOne() to Return a Single Matching Document from Your Database(MongoDB3)

Model.findOne() behaves like .find(), but it returns only one document (not an array), even if there are multiple items. It is especially useful when searching by properties that you have declared as unique. Find just one person which has a certain food in her favorites, using Model.findOne() -> Person. Use the function argument food as search key.

MongoDB and Mongoose - Use model.findById() to Search Your Database By \_id**(MongoDB3)**

When saving a document, mongodb automatically adds the field \_id, and set it to a unique alphanumeric key. Searching by \_id is an extremely frequent operation, so moongose provides a dedicated method for it. Find the (only!!) person having a given \_id, using Model.findById() -> Person. Use the function argument personId as search key.

MongoDB and Mongoose - Perform Classic Updates by Running Find, Edit, then Save(MongoDB4)

In the good old days this was what you needed to do if you wanted to edit a document and be able to use it somehow e.g. sending it back in a server response. Mongoose has a dedicated updating method : Model.update(). It is binded to the low-level mongo driver. It can bulk edit many documents matching certain criteria, but it doesn’t send back the updated document, only a ‘status’ message. Furthermore it makes model validations difficult, because it just directly calls the mongo driver.

Find a person by \_id ( use any of the above methods ) with the parameter personId as search key. Add “hamburger” to the list of her favoriteFoods (you can use Array.push()). Then - inside the find callback - save() the updated Person.

[\*] Hint: This may be tricky if in your Schema you declared favoriteFoods as an Array, without specifying the type (i.e. [String]). In that casefavoriteFoods defaults to Mixed type, and you have to manually mark it as edited using document.markModified('edited-field'). (http://mongoosejs.com/docs/schematypes.html - #Mixed )

## MongoDB and Mongoose - Perform New Updates on a Document Using model.findOneAndUpdate()(mOngoDB4)

Recent versions of mongoose have methods to simplify documents updating. Some more advanced features (i.e. pre/post hooks, validation) behave differently with this approach, so the Classic method is still useful in many situations. findByIdAndUpdate() can be used when searching by Id.

Find a person by Name and set her age to 20. Use the function parameter personName as search key.

Hint: We want you to return the updated document. To do that you need to pass the options document { new: true } as the 3rd argument to findOneAndUpdate(). By default these methods return the unmodified object.

## MongoDB and Mongoose - Delete One Document Using model.findByIdAndRemove(MongoDB5)

Delete one person by her \_id. You should use one of the methods findByIdAndRemove() or findOneAndRemove(). They are like the previous update methods. They pass the removed document to the cb. As usual, use the function argument personId as search key.