**What is Mongoose -** [Monib Bormon](https://medium.com/@monibbormon14?source=post_page-----c1bc3031cc08--------------------------------) Mar 4, 2022

Diagram

Description automatically generated

Mongoose is an object modeling tool for MongoDB and Node.js. What this means in practical terms is that you can define your data model in just one place, in your code. It allows defining schemas for our data to fit into, while also abstracting the access to MongoDB. This way we can ensure all saved documents share a structure and contain required properties.

**In Mongoose are Present Bunch of Terminology**

**Schema**

A description of the data structure. A Schema defines the field names and their corresponding time, which helps with validation, defaults, and other options in our models.

**Model**

A model is a primary tool for interacting with MongoDB. It is a fancy constructor for a document. Models are responsible for creating and reading documents from the underlying MongoDB database.

**Validation**

By default, MongoDB does not require the documents in a collection to have the same schema. In other words, the documents do not have to have the same fields and data types. Validation allows you to define the structure of the documents in a collection.

**Create Basic Schema**

**F**irst parameter to the [Schema class constructor](https://mongoosejs.com/docs/api/schema.html#schema_Schema) is a definition object. This object defines what paths a schema has. For example, the below userSchema has a name path and an age path.

**const** userSchema = **new** mongoose.Schema({  
 name: String,  
 age: Number  
});userSchema.path('name'); *// SchemaString { ... }*  
userSchema.path('age'); *// SchemaNumber { ... }*

To create a model in Mongoose, you call the [mongoose.model()](https://mongoosejs.com/docs/api/mongoose.html" \l "mongoose_Mongoose-model" \t "_blank)[function](https://mongoosejs.com/docs/api/mongoose.html#mongoose_Mongoose-model) with a schema as the 2nd parameter. For example, UserModel in the below example will have name and age properties, and will strip out any properties that aren't defined in userSchema.

**const** userSchema = **new** mongoose.Schema({  
 name: String,  
 age: Number  
});**const** UserModel = mongoose.model('User', userSchema);**const** doc = **new** UserModel({  
 name: 'Jean-Luc Picard',  
 age: 59,  
 rank: 'Captain'  
});  
doc.name; *// 'Jean-Luc Picard'*  
doc.age; *// 59// undefined, Mongoose strips out `rank` because it isn't in the schema*  
doc.rank;

Furthermore, Mongoose will cast documents to match the given schema types. This means you can safely pass untrusted data to Mongoose and trust that the data will match your schema.

**const** UserModel = mongoose.model('User', userSchema);**const** doc = **new** UserModel({  
 name: 'Jean-Luc Picard',  
 age: '59' *// Mongoose will convert this to a number*  
});  
doc.age; *// 59*  
**await** doc.save();*// Mongoose will convert '60' from a string to a number, even in an update*  
**await** UserModel.updateOne({}, { $set: { age: '60' } });

**Validation**

In addition to casting values, Mongoose also lets you define validation in your schemas. For example, suppose you want to ensure your users have a name. You can make the name property required in your schema as shown below.

**const** userSchema = **new** mongoose.Schema({  
 *// Make `name` required*  
 name: { type: String, required: true },  
 age: Number  
});  
**const** UserModel = mongoose.model('User', userSchema);**const** doc = **new** UserModel({ age: 30 });**const** err = **await** doc.save().catch(err => err);  
err.message; *// Path `name` is required.*

**Options**

The schema constructor takes 2 parameters: definition and options. You can find a [complete list of schema options on the Mongoose docs](https://mongoosejs.com/docs/guide.html#options).

For example, the [typeKey](https://mongoosejs.com/docs/guide.html" \l "typeKey" \t "_blank)[option](https://mongoosejs.com/docs/guide.html#typeKey) lets you configure what key Mongoose looks for to determine if you're defining a nested path. Suppose you wanted to define a nested key named type:

*// This is \*\*not\*\* how you define a `nested.type` property*  
**const** schema = **new** mongoose.Schema({  
 nested: {  
 type: String  
 }  
});schema.path('nested'); *// SchemaString { ... }*  
schema.path('nested.type'); *// undefined*

There are [several workarounds](https://mongoosejs.com/docs/faq.html#type-key) for this use case. One is to set the typeKey option as shown below.

*// Make Mongoose look for `$type` instead of `type`*  
**const** options = { typeKey: '$type' };  
**const** schema = **new** mongoose.Schema({  
 nested: {  
 type: String  
 },  
 otherProperty: {  
 $type: String  
 }  
}, options);schema.path('nested.type'); *// SchemaString { ... }*  
schema.path('otherProperty'); *// SchemaString { ... }*