Install

NPM install

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

Connection, Schema and Model

Load Environment Variable *config.js*

```
exports.DATABASE_URL = process.env.DATABASE_URL ||
                       global.DATABASE_URL ||
                      'mongodb://localhost/bookapp';
```

Connect to Mongo

```
const {DATABASE_URL} = require('./config');
const mongoose = require('mongoose');
mongoose.Promise = global.Promise;
mongoose.connect(DATABASE_URL);
```

Schemas and Models

Each schema maps to a MongoDB collection and defines the shape of the documents within that collection. Models are constructors which create documents which can be save and retrieved from a database connection.

Define the a basic Schema *models/book.js*

```
const bookSchema = mongoose.Schema({
  title:  String,
  published: Date,
  inPrint: Boolean,
  price: Number,
  author: [ {name: String} ],
  comments: [ {
    body: String,
    date: Date
  }],
  isbn: {
    type: String,
    unique: true
  }
});
```

Create a model *models/book.js*

```
const Book = mongoose.model('Book', bookSchema);
```

Documents

Documents are instance of Models

Create a document

```
const myBook = new Book({
  name: 'Naked Lunch',
  comments: 'published in 1959'
});
```

Model CRUD operations

CRUD operations using the Model

Model: Create()

Model.create(doc(s))

```
Book.create({
  title: 'Catch-22',
  author: { 'Joseph Heller' },
  published: '10 November 1961',
  isbn: '0684833395',
  inPrint: true,
  price: 11.99
}).then(/*...*/)
```

See Also: insertMany()

Model: Find()

Model.find(query, [projection], [opts])

```
Book.find({votes: 42}).then(/*...*/)
```

Model.findById(id, [projection], [opts])

```
Book.findById('584016af5149cd70c9').then(/*...*/)
```

Model.findOne(query, [projection], [opts])

```
Book.findOne({title: 'Catch-22'}, 'title votes')
```

See Also: findById(), findOne(),

Model: Update

Model.update(query, doc, [opts])

Updates one doc w/o returning, use "multi" option to update multiple

```
Book.update( {name: 'Catch-22'},
  {name: 'Catch-99'},
  {multi: true}).then(/*...*/)
```

Model.updateOne(query, doc, [opts]

Updates only one doc regardless of the "multi" option.

```
Book.updateOne( {name: 'Catch-xx'},
  {name: 'Catch-22'}).then(/*...*/)
```

Model.updateMany(query, doc, [opts])

Updates all docs regardless of the "multi" option.

```
Book.updateMany( {name: 'Catch-xx'},
  {name: 'Catch-22'}).then(/*...*/)
```

Model.findByIdAndUpdate(id, [mod], [opts])

- Finds a matching document
- If it exists, update it else create it (see upsert)
- Return the inserted or updated document (see new)

```
Book.findByIdAndUpdate('584347d2af5143db9cd95c02',
  {comment: 'satirical novel'}),
  {upsert: true, new: true}).then(/*...*/)
```

Model.findOneAndUpdate(query, [mod], [opts])

- Use "new" option to return the modified doc
- Use "upsert" option to create the doc if it doesn't already exist

```
Book.findOneAndUpdate( {name: 'Catch-22'},
  {comment: 'published 1961'}),
  {upsert: true, new: true}).then(/*...*/)
```

Model: Delete and Remove

Model.remove(query)

- Sends a remove command directly to MongoDB
- Removes the first document that matches the condition
- No Mongoose docs are involved, no middleware (hooks) are executed
- To remove all documents set the "justOne" option to false

```
Book.remove({name: 'Catch-22'}, {justOne: false})
```

Model.deleteMany(query)

- Deletes all docs regardless of the "justOne" option.

```
Book.deleteMany({name: 'Life After Life'}).then()
```

Model.deleteOne(query)

- Deletes at most one doc regardless of the "justOne" option.

```
Book.deleteOne({name: 'Catch-22'}).then(/*...*/)
```

Model.findByIdAndRemove(id, [opts])

```
Book.findByIdAndRemove('516...0c9').then(/*...*/)
```

Model.findOneAndRemove(id, [opts])

```
Book.findOneAndRemove({name: 'Catch-22'}).then(/**/)
```

Document: Save and Remove

doc.save([opts], [opts.safe])

- Creates this document using save

```
const myBook = new Book({
  name: 'Naked Lunch',
  comments: 'published in 1959'
});
myBook.save()
```

doc.save([opts], [opts.safe])

- Updates this document using save

```
Book.findOne({name: 'Naked Lunch'})
  .then( doc => {
    doc.name = 'The Naked Lunch'
    return doc.save()
}).then(/*...*/)
```

doc.remove([function])

- Removes this document

```
Book.findOne({name: 'The Iliad and Odyssey'})
  .then( doc => doc.remove() );
```

Custom Methods

Mongoose provides 3 ways of defining custom functionality.

- **Statics**: model-level functionality similar to Model.create()
- **Methods**: document-level functionality similar to doc.save()
- **Virtuals:** property-level functionality with getters/setters

Statics

Add static method to bookSchema *models/book.js*

```
bookSchema.statics.findByAuthor = function(name) {
  return this.find( {author.lastName: name} );
};
```

Usage

```
Book.findByAuthor('Heller')
  .then( res => console.log(res) );
```

Methods

Add instance method to bookSchema *models/book.js*

```
bookSchema.methods.apiRepr = function() {
  return {
    title: this.title,
    date: this.date.toLocaleDateString()
  }
}
```

Usage

```
res.json( book.apiRepr() );
```

Virtual

Add virtual get/set to bookSchema *models/book.js*

```
bookSchema.virtual('fullName')
  .get( function() {
    const auth = this.author;
    return `${auth.firstName} ${auth.lastName}`;
  })
  .set(function( fullName ) {
    const [first, last] = fullName.split(' ');
    this.author.firstName = first;
    this.author.lastName = last;
  });
```

Validators

Mongoose provides several ways to validate input

- All types have the required validator
- Numbers have min and max validators.
- Strings have enum, match, maxlength and minlength validators
- All types also have custom validate function

Required - Require title

```
title: {
  type: String,
  required: true
},
```

Required - Make price required if inPrint field is true

```
inPrint: Boolean,
price: {
  type: Number,
  required: [function() {return this.inPrint},
    'If inPrint is true then price is required'
]},
```

Match - Validate ISBN using RegEx. Define a custom error message.

```
isbn: {
  type: String,
  match: [
    /^\d{9}(\d|X)$/,
    'ISBN ({VALUE}) must be match ISBN-10 format'
]},
```

Validate - Create a custom validator function with error message

```
language: {
  type: String,
  validate: {
    validator: function(input) {
      return languages.indexOf(input) !== -1
    },
    message: 'Language {VALUE} is not a valid!'
}},
```

Validate - Also works with promises to make async fetch calls

```
...
  validate: function(val) {
    return fetch(url).then( result => !!result)
  }
```

Population

Population automatically replaces the specified paths in the document with document(s) from other collection(s).

Example schema with cross-collection lookups

```
const personSchema = mongoose.Schema({
  _id     : Number,
  name    : String,
  age     : Number,
  stories : [{
    type: mongoose.Schema.Types.ObjectId,
    ref: 'Story' }]
});

const storySchema = mongoose.Schema({
  _creator : { type: Number, ref: 'Person' },
  title    : String,
  fans     : [{ type: Number, ref: 'Person' }]
});

const Story  = mongoose.model('Story', storySchema);
const Person = mongoose.model('Person',
personSchema);
```

Using Populate

```
Story.findOne({ title: 'Once upon a timex.' })
  .populate('_creator')
  .then( story => {
    console.log(`Creator: ${story._creator.name}`);
  }).catch(/*...*/);
```

Queries

Build a query using chaining syntax, instead of a JSON object.

```
Person
  .find({name: 'Catch-22'})
  .where('inPrint').equals(true)
  .where('price').lt(20)
  .limit(10)
  .sort('-price')
  .select('title author price')
  .then( ( books ) => { console.log(books) })
  .catch( ( err ) => { console.log(err) })
```