**Express MongoDB -** [zhuting](https://zhuting.medium.com/?source=post_page-----f7f88e7097da--------------------------------) Mar 5, 2022

We have the following packages that are useful in many web applications:

* cookie-parser: Used to parse the cookie header and populate req.cookies (essentially provides a convenient method for accessing cookie information)
* debug: A tiny node debugging utility modeled after node core’s debugging technique
* morgan: A HTTP request logger middleware for node
* http-errors: Create HTTP errors where needed ( for express error handling)

The script section first defines a “start” script, which is what we are invoking when we call npm start to start the server. From the script definition, you can see this actually starts the JavaScript file ./bin/www with node.

"scripts": {  
 "start": "node ./bin/www",  
 "devstart": "nodemon ./bin/www",  
 "serverstart": "DEBUG=express-locallibrary-tutorial:\* npm run devstart"  
 },

**Using a Database (with Mongoose)**

This section explains how object schema and models are declared, the main field types, and basic validation. It also briefly shows a few of the main ways in which you can access model data.

Express apps can use many different databases, and there are several approaches you can use for performing Create, Read, Update and Delete (CRUD) operations.

There are two common approaches for interacting with a database:

* Using the database native query language (e.g SQL)
* Using an Object Data Model(“ODM”) or an Object Relational Model (“ORM”)  
  An ODM/ORM represents the website’s data as JavaScript objects, which are then mapped to the underlying database. Some ORMs are tied to a specific database, while others provide a database-agnostic backend.

The very best performance can be gained by using SQL. ORMs are often slower because they use translation code to map between objects and database format, which may not use the most efficient database queries.

The benefit of using ORM is that programmers can continue to think in terms of JavaScript objects rather than database semantics.

There are many ODM/ORM solutions available on the NPM package manager site (check out the [odm](https://www.npmjs.com/browse/keyword/odm" \t "_blank) and [orm](https://www.npmjs.com/browse/keyword/orm" \t "_blank) tags for a subset!).

**Designing the models**

We are going to design a model for the library. We need to store information about books (title, summary, author, genre, ISBN) and we might have multiple copies available (with globally unique ids, availability statuses, etc). We might need to store more information about the author than just their name. We want to be able to sort information based on the book title, author, genre, and category.

Diagram

Description automatically generated

Follow the [tutorial](https://www.mongodb.com/docs/manual/tutorial/install-mongodb-on-os-x/) to install MongoDB.

brew tap mongodb/brew  
brew install mongodb-community@5.0

To start mongodb/brew/mongodb-community now and restart at login:

brew services start mongodb/brew/mongodb-community

Or, if you don’t want/need a background service you can just run:

mongod --config /usr/local/etc/mongod.conf

**Defining and creating models**

Models are defined using the schema interface. The Schema allows you to define the fields stored in each document along with their validation requirements and the default values.

Schemas are then “compiled” into models using the mongoose.mode() method.

Each model maps to a collection of documents in the MongoDB database. The documents will contain the fields/schema types defined in the mode Schema .

**Defining schemas and creating a model**

The code below shows how you might define a simple schema.

const mongoose = require('mongoose');  
const Schema = mongoose.Schemaconst SomeModelSchema = new Schema({  
 name: String,  
 binary: Buffer,  
 living: Boolean,  
 updated: { type: Date, default: Date.now() },  
 age: { type: Number, min: 18, max: 65, required: true},  
 mixed: Schema.Types.Mixed,  
 \_id: Schema.Types.ObjectId});const SomeModel = mongoose.model('SomeModel', SomeModelSchema)

Most of [Schema Types](https://mongoosejs.com/docs/schematypes.html) are self-explanatory.

**Validation**

Mongoose provides built-in and custom validatiors, and synchronous and asynchronous validators. It allows you to specify both the acceptable range of values and the error message for validation failure in all cases.

The built-in validators include:

* All SchemaTypes have the built-in required validator
* Numbers have min and max validators
* Strings have: enum, match, maxLength, and minLength

**Virtual properties**

Virtual properties are document properties that you can get and set but that does not get persisted to MongoDB. The getters are useful for formatting or combining fields, while setters are useful for de-composing a single value into multiple values for storage.

**Methods and query helpers**

A schema can also have instance methods, static methods, and query helpers. The instance and static methods are similar, but with the obvious difference that an instance method is associated with a particular record and has access to the current object. Query helpers allow you to extend mongoose’s chainable query builder API.

**Using models**

Once you have created a schema you can use it to create models. The model represents a collection in the database that you can search, while the mode’s instances represent individual documents that you can save and retrieve.

To create a record you can define an instance and then call save()

Creation of records (along with updates, deletes, and queries) are asynchronous operations — you supply a callback that is called when the operation completes.

You can search for records using [query methods](https://mongoosejs.com/docs/queries.html).

All callbacks in Mongoose use the pattern callback(error, result)

You can create references from one document/model instance to another using the ObjectId schema field, or from one document to many using an array of ObjectIds . The field stores the id of the related model. If you need the actual content of the associated document, you can use the populate() method in a query to replace the id with the actual data.

It’s highly recommended to define each model schema in its own file, then export the method to create the model.

**Connect to MongoDB**

const mongoose = require('mongoose');const mongoDB = 'mongodb://127.0.0.1/my\_database';mongoose.connect(mongoDB, { useNewUrlParser: true, useUnifiesTopology: true });const db = mongoose.connection;db.on('error', console.error.bind(console, 'connection error'));

**Define the Library Schema**

We will define a separate module for each model. and start by creating a folder for our models and then create separate files for each of the models.

/models  
 author.js  
 book.js  
 bookInstance.js  
 genre.js

The Author schema defines an author as having String SchemaType for the first and family names, and Date fields for the dates of birth and death.