Models

Models are fancy constructors compiled from `Schema` definitions. An instance of a model is called a document. Models are responsible for creating and reading documents from the underlying MongoDB database.

Compiling your first model

When you call `mongoose.model()` on a schema, Mongoose *compiles* a model for you.

```
const schema = new mongoose.Schema({ name: String, size: String });
const Tank = mongoose.model('Tank', schema);
```

The first argument is the *singular* name of the collection your model is for. **Mongoose automatically looks for the plural, lowercased version of your model name.** Thus, for the example above, the model Tank is for the **tanks** collection in the database.

Note: The `.model()` function makes a copy of `schema`. Make sure that you've added everything you want to `schema`, including hooks, before calling `.model()`!

Constructing Documents

An instance of a model is called a document. Creating them and saving to the database is easy.

```
const Tank = mongoose.model('Tank', yourSchema);

const small = new Tank({ size: 'small' });
await small.save();

// or

await Tank.create({ size: 'small' });
```

```
// or, for inserting large batches of documents
await Tank.insertMany([{ size: 'small' }]);
```

Note that no tanks will be created/removed until the connection your model uses is open. Every model has an associated connection. When you use `mongoose.model()`, your model will use the default mongoose connection.

```
await mongoose.connect('mongodb://127.0.0.1/gettingstarted');
```

If you create a custom connection, use that connection's `model()` function instead.

```
const connection = mongoose.createConnection('mongodb://127.0.0.1:27017/test');
const Tank = connection.model('Tank', yourSchema);
```

Querying

Finding documents is easy with Mongoose, which supports the rich query syntax of MongoDB. Documents can be retrieved using a `model`'s find, findById, findOne, or where static functions.

```
await Tank.find({ size: 'small' }).where('createdDate').gt(oneYearAgo).exec();
```

See the chapter on queries for more details on how to use the Query api.

Deleting

Models have static `deleteOne()` and `deleteMany()` functions for removing all documents matching the given `filter`.

```
await Tank.deleteOne({ size: 'large' });
```

Updating

Each `model` has its own `update` method for modifying documents in the database without returning them to your application. See the API docs for more detail.

```
// Updated at most one doc, `res.nModified` contains the number
// of docs that MongoDB updated
await Tank.updateOne({ size: 'large' }, { name: 'T-90' });
```

If you want to update a single document in the db and return it to your application, use findOneAndUpdate instead.

Change Streams

Change streams provide a way for you to listen to all inserts and updates going through your MongoDB database. Note that change streams do **not** work unless you're connected to a MongoDB replica set.

```
async function run() {
  // Create a new mongoose model
  const personSchema = new mongoose.Schema({
    name: String
  });
  const Person = mongoose.model('Person', personSchema);

  // Create a change stream. The 'change' event gets emitted when there's a
  // change in the database
  Person.watch().
    on('change', data => console.log(new Date(), data));

  // Insert a doc, will trigger the change stream handler above
  console.log(new Date(), 'Inserting doc');
  await Person.create({ name: 'Axl Rose' });
}
```

The output from the above async function will look like what you see below.

2018-05-11T15:05:35.467Z 'Inserting doc' 2018-05-11T15:05:35.487Z 'Inserted doc' 2018-05-11T15:05:35.491Z { _id: { _data: ... }, operationType: 'insert', fullDocument: { _id: 5af5b13fe526027666c6bf83, name: 'Axl Rose', __v: 0 }, ns: { db: 'test', coll: 'Person' }, documentKey: { _id: 5af5b13fe526027666c6bf83 } }

You can read more about change streams in mongoose in this blog post.

Views

MongoDB Views are essentially read-only collections that contain data computed from other collections using aggregations. In Mongoose, you should define a separate Model for each of your Views. You can also create a View using `createCollection()`.

The following example shows how you can create a new `RedactedUser` View on a `User` Model that hides potentially sensitive information, like name and email.

```
// Make sure to disable `autoCreate` and `autoIndex` for Views,
// because you want to create the collection manually.
const userSchema = new Schema({
  name: String,
  email: String,
  roles: [String]
}, { autoCreate: false, autoIndex: false });
const User = mongoose.model('User', userSchema);
const RedactedUser = mongoose.model('RedactedUser', userSchema);

// First, create the User model's underlying collection...
await User.createCollection();
// Then create the `RedactedUser` model's underlying collection
// as a View.
await RedactedUser.createCollection({
  viewOn: 'users', // Set `viewOn` to the collection name, **not** model name.
  pipeline: [
    {
      $set: {
        name: { $concat: [{ $substr: ['$name', 0, 3] }, '...'] },
        email: { $concat: [{ $substr: ['$email', 0, 3] }, '...'] }
      }
    }
  ]
});

await User.create([
  { name: 'John Smith', email: 'john.smith@gmail.com', roles: ['user'] },
  { name: 'Bill James', email: 'bill@acme.co', roles: ['user', 'admin'] }
]);

// [{ _id: ..., name: 'Bil...', email: 'bil...', roles: ['user', 'admin'] }]
console.log(await RedactedUser.find({ roles: 'admin' }));
```

Note that Mongoose does **not** currently enforce that Views are read-only. If you attempt to `save()` a document from a View, you will get an error from the MongoDB server.

Yet more

The API docs cover many additional methods available like count, mapReduce, aggregate, and more.

Next Up

Now that we've covered `Models`, let's take a look at Documents.