

Dynamic Schemas at the Application Level [MEAN Stack]

May 18, 2017

A few notes before we get started:

1. This post assumes a working knowledge of the full MEAN stack. I don't suggest getting into this if you haven't built a full stack app on your own.
2. This is just a proof of concept. I've never seen this technique used before, nor do I have any professional experience as of writing this (I just finished school). Apply it at your own risk.
 - That said, if you have seen something like this before, point me in that direction!
3. The example code for this post is on GitHub [here](#). I encourage you to look over it as you read.

One of the awesome things about [MongoDB](#) is its "schema-less" design; each document is self-describing. This offers two key advantages:

1. The schema can change as requirements change without costly database-wide operations.
2. We can group documents by the way we naturally think about them instead of limiting ourselves to one schema (consider the housing example below):

```
> db.residences.find()
{ "_id" : ObjectId("591d7692cf0efffc5bb9b896"), "type" : "house", "beds" : 3,
  { "_id" : ObjectId("591d76bbcf0efffc5bb9b897"), "type" : "apartment", "rent"
  { "_id" : ObjectId("591d772ccf0efffc5bb9b898"), "type" : "dormitory", "cost"
```

Having this flexibility at the database layer is great for a backend, or for a full stack app where entities follow basic design patterns (think IS-A relationships). But what if different users need drastically different models? Can we exploit Mongo's flexibility in our application code?

Dynamic schemas in the view

Changing a view based on a user's role isn't a novel concept. Filtering the data returned to a user is commonplace too; this is the purpose of [SQL views](#). It's easy to do on the frontend,

especially with Angular:

```
<!-- view-example/index.html -->

...

<table ng-show="userType == 'student'" class="table">
  <thead>
    <th>Title</th>
    <th>Edition</th>
    <th>Author</th>
  </thead>
  <tbody>
    <tr ng-repeat="b in books">
      <td></td>
      <td></td>
      <td></td>
    </tr>
  </tbody>
</table>

<table ng-show="userType == 'admin'" class="table">
  <thead>
    <th>ISBN</th>
    <th>Internal ID</th>
  </thead>
  <tbody>
    <tr ng-repeat="b in books">
      <td></td>
      <td></td>
    </tr>
  </tbody>
</table>

...
```

This doesn't change the data itself though, or how we manipulate the data. Let's take this idea to the controller.

Dynamic schemas in the controller

Again, definitely look at the GitHub repo to see how this all works in context.

For this example, we'll use vehicles – specifically, cars and heavy trucks. Even though different vehicles are described in different ways, it might make sense to put them in the same Mongo collection (for a small business, perhaps).

DB layer

Here's our car schema:

```
// controller-example/db/car.js

const mongoose = require("mongoose");
const Schema = mongoose.Schema;

const CarSchema = new Schema({
  vehicleType: { type: String, default: "car" },
  make: String,
  model: String,
  color: String,
  numSeats: Number
});

const Car = mongoose.model("car", CarSchema, "vehicles");
module.exports = Car;
```

And here's our truck schema:

```
// controller-example/db/truck.js

const mongoose = require("mongoose");
const Schema = mongoose.Schema;

const TruckSchema = new Schema({
  vehicleType: { type: String, default: "truck" },
  make: String,
  sleeperCab: Boolean,
  numAxles: Number,
  horsepower: Number
});

const Truck = mongoose.model("truck", TruckSchema, "vehicles");
module.exports = Truck;
```

We're mapping both schemas to the same Mongo collection with the `collection` parameter (the third one) of `mongoose.model()` (docs are [here](#)).

App layer (backend)

Here's the interesting bit. We can take advantage of Node's `require()` function to load the particular schema we need exactly when and where we need it. We don't need to specify a class for our controller up front like we would with other frameworks (ex. Rails or Django):

```
// controller-example/backend.js

app.post('/', function(req, res) {
  var vehicle = req.body;

  if (vehicle.vType == "car") {
    const Car = require('./db/car');
    Car.create(vehicle, function(err, car) {
      res.send(car);
    });
  }

  if (vehicle.vType == "truck") {
    const Truck = require('./db/truck');
    Truck.create(vehicle, function(err, truck) {
      res.send(truck);
    });
  }
});
```

App layer (frontend)

The frontend code for this is simple:

```
// controller-example/public/frontend.js

var Car = $resource(url);
$scope.car = { vType: "car" };
var Truck = $resource(url);
$scope.truck = { vType: "truck" };

$scope.createVehicle = function(vehicle) {
  var newVehicle;
  if (vehicle.vType == "car") {
```

```
    newVehicle = new Car(vehicle);
  }
  if (vehicle.vType == "truck") {
    newVehicle = new Truck(vehicle);
  }
  newVehicle.$save(function(savedVehicle) {
    var vStr = JSON.stringify(savedVehicle, null, 1);
    alert('created ' + savedVehicle.vType + '!\n' + vStr);
    $scope.vehiclesStrArr.push(vStr);
  });
};
```

Because of the way we structured the backend, we can send both car and truck requests to the same route! That being said, it would probably be better practice to have two separate routes for maintainability.

Generalizing for >2 schemas

(Because no new technique is complete without some hand-wavy scaling discussion, right?)

**DEV Community**
@ThePracticalDev

Chapter 1: Databases with cool-sounding names

12:27 AM - Nov 22, 2016

4,405 3,659 people are talking about this

The above example uses just two schemas. But what if we need more? Maybe each user (who knows how many) structures their data slightly differently. This is precisely what Mongo was designed for, and it was the inspiration for this little experiment.

We can calculate a schema path on the fly, then pass it to `require()`:

```
const schemaPath = './db/schema-' + req.body.username;  
const theSchema = require(schemaPath);
```


This way, we can send some kind of schema identifier along with any API requests and load the correct one based on a particular value – in this case, a username.

Closing thoughts

Like I said earlier, this is just a proof of concept. I have no idea if or how it's been used before, or if it would be a useful or sensible thing to do in the long term. I really want to hear from more experienced developers on this; drop a comment below or contact me via [Twitter](#)!

0 Comments

hudsonburgess

 Pham Tuan ▾ Recommend Share

Sort by Best ▾



Start the discussion...

Be the first to comment.

ALSO ON HUDSONBURGESS

A Basic Task Management System

1 comment • a year ago

Madona — Great Post!! Very well written. I also feel like sharing ProofHub here for task management. It empowers teams to work ...

On Objectivity

2 comments • a year ago

Daniel Henry — This is a good post. Differing on definitions makes discussion of certain topics very difficult on the internet, which ...

 Subscribe Add Disqus to your siteAdd DisqusAdd Disqus' Privacy PolicyPrivacy PolicyPrivacy**Hudson Burgess**

Hudson Burgess
hudsonburgess7@gmail.com

[hab9sr](#)[hudsonburgess7](#)

Things I think about when I'm not talking.
Which is most of the time.