structuring a database app
today’s lecture

• Database structures, continued
  • Two ways to structure data in MongoDB
  • Mongoose
• An example app with Express, MongoDB, and Mongoose
• Designing your HTTP interface: REST style
purposes

After today’s lecture, you should understand:
• What a schema is
• How the choice of a schema affects how you write database queries
• The model-view-controller separation in a typical database web app
• The REST convention for HTTP interfaces
MongoDB recap

- Document-oriented databases store collections of hierarchical documents
  - E.g. MongoDB stores collections of arbitrary JSON objects
- Queries let you retrieve a subset of data in the database, based on specific conditions
Was shown in the terminal during the last lecture:

```javascript
db.restaurants.find({borough: "Bronx"}, {name: 1, borough: 1, "grades.grade": 1 });
```

```javascript
db.restaurants.find({borough: "Bronx", "address.zipcode": "10462"}, {name: 1, borough: 1, "grades.grade": 1 });
```

```javascript
db.restaurants.find({"grades.score": 0}, {name: 1, "grades.score": 1});
```

```javascript
db.restaurants.find({"grades.score": {$gt: 50}}, {name: 1, "grades.score": 1});
```
queries

restaurants.find({})
› find all restaurants

.find({borough: "Bronx"})
› find restaurants with borough “Bronx”

.find({borough: "Bronx"}, {name: 1})
› find Bronx restaurants, returning only their names

.find({borough: {$in: ["Bronx", "Queens"]}})
› find restaurants with a borough in the given array

.find({"grades.score": {$gt: 50}})
› find restaurants with a score in the nested “grades” array higher than 50
The Maze Runner

When Thomas wakes up trapped in a massive maze with a group of other boys, he has no memory of the outside world other than strange dreams about a mysterious organization known as W.C.K.D. Only by piecing together fragments of his past will he discover the maze and Thomas hopes to uncover ... More
Back to our object-oriented heap.

Give the class 5 minutes to imagine how you would structure this data in MongoDB.
Remember that MongoDB stores everything in terms of collections of JavaScript objects

The term “Map” was probably used in this diagram because keys will be identifiers of the respective value objects.
two ways of structuring data in MongoDB
embedded

```
{
  title: "Fury",
  time: "7:00pm",
  theater: {
    name: "West Newton Cinema",
    location: "Newton"
  }
}
```

one document in the collection Movies
We’re still storing this in a document store, but we’re here modeling the data as you would in a relational database, as two collections of tuples.
Consistency: no transactions in Mongo; only atomic for single documents.

We will get back to how to query each one of these.
mongoose
what is mongoose?

an “object-document mapper”
› like an “object-relational mapper” for relational databases
› but documents closer to objects than tuples

what mongoose provides
› schema declarations (good for validation)
› model classes (can extend with methods; insert with save)
› helpful API, especially populate method
Note the reference from movie to theater. Note that MongoDB or Mongoose do not actually ensure referential integrity (see [http://stackoverflow.com/questions/13761115/express-js-and-mongoose-model-relationships-in-model-or-router](http://stackoverflow.com/questions/13761115/express-js-and-mongoose-model-relationships-in-model-or-router)). But the knowledge of references is used for instance in the populate function, which we'll see in the next slide.

Note the save()!
Note the two nested calls here. The outer one is just a regular find.
(Hmm, I think the “theater” object might actually have an _id field as well this example.)

Note that we’re calling find on the Mongoose model constructor here, as opposed to on a plain MongoDB collection. So we get Mongoose model objects instead of plain JSON back from find and populate.
Consistency: no transactions in Mongo; only atomic for single documents.
an example app with Express, MongoDB, and Mongoose
mongod --config /usr/local/etc/mongod.conf
cd "/Users/ebakke/ZRoot/ta6.170/6170-lectures/L09-movies-example"
npm start

http://localhost:3000/movies
https://github.mit.edu/6170-fa15/6170-lectures/tree/master/L09-movies-example
the standard Express boilerplate

- Running "express someapp --ejs" in an empty directory creates a bunch of files to start with:

```
create : someapp
create : someapp/package.json
create : someapp/app.js
create : someapp/public
create : someapp/public/javascripts
create : someapp/public/images
create : someapp/public/stylesheet
create : someapp/public/stylesheet/style.css
create : someapp/routes
create : someapp/routes/index.js
create : someapp/routes/users.js
create : someapp/views
create : someapp/views/index.ejs
create : someapp/views/error.ejs
create : someapp/bin
create : someapp/bin/www
```

install dependencies:
$ cd someapp && npm install

run the app:
$ DEBUG=someapp:* npm start
“Utility” modules typically have no dependencies on any major modules in an application; they are typically pure logic without state or side effects.
Model-View-Controller separation in movies example

- **Model**: Mongoose schema in `movie-data-mongoose.js`
- **View**: EJS templates, static CSS
- **Controller**: (route handlers in `routes/movies-mongoose.js`)

(Note: The movies example is a bit controller-heavy; could be improved by moving some logic from the controller to the model.)

It would often be OK to have the view depend on the model as well, but this example does not need this.
package.json (library dependencies)

{
    "name": "movies",
    "description": "toy movie app to demonstrate mongo",
    "version": "0.0.1",
    "private": true,
    "scripts": {
        "start": "node ./bin/www"
    },
    "dependencies": {
        "express": "4.9.4",
        "ejs": "~0.8.5",
        "mongodb": "*",
        "mongoose": "*",
        "body-parser": "~1.0.0",
        "debug": "*"
    }
}

bodyParser: this stuff is added for you by the express generator; it populates the req.body property in route handlers

redirect handles the top level “/“ URL. We could just take it out an the app would work fine, except if you just try to go to localhost:3000 rather than localhost:3000/movies. See the “/movies” argument when registering movieRoutes.

app.set configures the view directory and view engine; will be used in route handlers

static serves up anything in the public directory as-is (just a CSS file in our app)
Setting up all the dependencies of the controller. Most importantly, the controller depends on the model and on the express routing API.

There is really quite a lot of stuff in this controller that would be nice to keep in a separate file, notably the setting up of the database connection and the population with sample data (not shown here).

We’ll get back to the controller after we look at the model.
This is a minimal Mongoose-based model, using the “relational” schema we discussed before.
GET /find (movies-mongoose.js controller)

// search page: render form
router.get('/find', function(req, res) {
    res.render('movies/find');
});

This is a
This is what an EJS template looks like. This one is just plain HTML, with some “include” directives.
This find/populate logic might be better to move to the model through an instance method, which we will introduce latter.
This EJS template takes a few parameters.
This find/populate logic might be better to move to the model through an instance method, which we will introduce latter.

Note the need to look up the id of the theater, and the continuation involved.
If you find yourself doing a lot of logic on Mongoose model objects (aka Mongoose documents) from the controller (route handlers), a good way to move this out of the controller and into the model is to add an instance method to the Mongoose Schema. This might have been a good idea in some parts of the Movies example, e.g. to factor out the database operations from renderQuery.

The example shown here is from the Mongoose documentation site.

```javascript
// define a schema
var animalSchema = new Schema({ name: String, type: String });

// assign a function to the "methods" object of our animalSchema
animalSchema.methods.findSimilarTypes = function (cb) {
    return this.model('Animal').find({ type: this.type }, cb);
}

var Animal = mongoose.model('Animal', animalSchema);
var dog = new Animal({ type: 'dog' });

dog.findSimilarTypes(function (err, dogs) {
    console.log(dogs); // woof
});

http://mongoosejs.com/docs/guide.html#methods
```
designing your HTTP interface: REST style
browser as editor

Screenshot of Berners Lee's original browser on NeXT platform
a web for shared document editing

Tim Berners Lee’s web
distributed document editing
like one big Wiki

HTTP methods
GET: get a document from a server
POST: send a document to a server
PUT: update a document at a server
DELETE: delete a document from a server

request components
query string: for searching
body: for new version of page
mosaic (1993)

Note the form for entering the search string. Not in original HTML.
**what happened?**

<table>
<thead>
<tr>
<th>early browsers</th>
<th>Mosaic</th>
</tr>
</thead>
<tbody>
<tr>
<td>just displayed pages</td>
<td>added forms</td>
</tr>
<tr>
<td>only needed GET</td>
<td></td>
</tr>
</tbody>
</table>

**to pass form data**

GET: in query string

POST: in body

---

No fundamental difference between GET and POST in this view; just use whichever is convenient.
Here’s a page from my favorite tea company, showing the listing page for green teas. Their website is old fashioned though.
so now rather than DELETE pages/3

we see

POST /deletePage
GET /deletePage?id=3

or
GET /foo.asp?action=delete&page=3

Note that the semantics of the HTTP verbs is completely lost here.
Four basic functions of persistent storage; term CRUD probably coined by James Martin in 1970s. SQL offers the four basic functions, so does TBL’s web. We’ll see that REST comes full circle here, returning the web to this structure.
Note that now the verb has semantics! Determines what action is being performed. More on this in a few slides.
kinds of path

collection paths
http://tea.com/teas
http://tea.com/teas/123/reviews
http://tea.com/teas/green

instance paths
http://tea.com/teas/123
http://tea.com/teas/123/reviews/4

Note 3rd collection path. Paths are rather a restricted query language!
<table>
<thead>
<tr>
<th>actions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>show reviews of tea #123</strong></td>
</tr>
<tr>
<td>GET <a href="http://tea.com/teas/123/reviews">http://tea.com/teas/123/reviews</a></td>
</tr>
<tr>
<td><strong>create a review of tea #123</strong></td>
</tr>
<tr>
<td>POST <a href="http://tea.com/teas/123/reviews">http://tea.com/teas/123/reviews</a></td>
</tr>
<tr>
<td><strong>update review #4 of tea #123</strong></td>
</tr>
<tr>
<td>PUT <a href="http://tea.com/teas/123/reviews/4">http://tea.com/teas/123/reviews/4</a></td>
</tr>
<tr>
<td><strong>delete review #4 of tea #123</strong></td>
</tr>
<tr>
<td>DELETE <a href="http://tea.com/teas/123/reviews/4">http://tea.com/teas/123/reviews/4</a></td>
</tr>
</tbody>
</table>

Ask them to do this as an exercise. Explain why PUT is idempotent; takes id of resource, unlike POST.
do pretty URIs matter?

one view
URIs are just code points
can be generated by the app
user can ignore them

but
users see URIs and type them
for web services, may be no context for action

Some web frameworks, such as Chlipala’s UrWeb, treat URIs as code addresses that are just generated automatically and not regarded as user-visible names. That seems suitable if the different parts of the web are tightly integrated, but if we’re looking at a web service in which different operations can be called in a context independent way, it seems important to make the URIs names like functions in an API.
**conclusion**

- A *schema* defines the structure of your data
- Schemas involving multiple MongoDB collections are harder to query
- Typical database web apps are divided into:
  - Model: Database schema and access logic
  - View: Page templates and static content
  - Controller: Route handlers for HTTP requests
- **HTTP interfaces should follow REST conventions**
  - URIs refer to resources or collections of resources
  - HTTP verbs are used properly