database structures
purposes

After this lecture, you should...

• Understand what a document-oriented database (aka document store) is
• Understand what a query is
• Understand how to get the result of a query using callbacks
• Understand what a schema is

The higher-level idea here is “document-oriented database”, but more concretely, we’ll be talking about MongoDB.
lecture overview

- three ways to store data
  (object-oriented, document-oriented, relational)
- some history
- the MongoDB API
- two ways to structure data in MongoDB
  (embedded vs. relational)
- mongoose
3 ways to store data
Showtimes for The Maze Runner

<table>
<thead>
<tr>
<th>All times</th>
<th>Morning</th>
<th>Afternoon</th>
<th>Evening</th>
<th>Night</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Showcase Cinema de Lux Legacy Placed - Haverhill</strong></td>
<td>4:40</td>
<td>7:40</td>
<td>10:40</td>
<td>1:40</td>
</tr>
<tr>
<td><strong>Regal Fenway Stadium 13 &amp; RPX - Map</strong></td>
<td>4:05</td>
<td>7:05</td>
<td>10:05</td>
<td></td>
</tr>
<tr>
<td><strong>Embassy Cinema - Map</strong></td>
<td>4:05</td>
<td>7:15</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**The Maze Runner**

*PG-13* 1h 53m - Adventure/Sci-Fi

Rating: 7.7/10 - Rotten Tomatoes

When Thomas wakes up trapped in a massive maze with a group of other boys, he has no memory of the outside world. He then begins to discover strange messages that hint at a mysterious organization known as W.C.K.D. Only by piecing together fragments of his past will he discover the maze and Thomas hope to uncover... More
In-memory!
Note that a string is technically a primitive but gets converted to an object whenever you things like “foo”.length (see https://javascriptweblog.wordpress.com/2010/09/27/the-secret-life-of-javascript-primitives/ ).
### Showings

<table>
<thead>
<tr>
<th>id</th>
<th>theater</th>
<th>screen</th>
<th>movie</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>7:00pm</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

### Theaters

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>location</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>&quot;Legacy Place&quot;</td>
<td>&quot;Dedham&quot;</td>
</tr>
</tbody>
</table>

### Movies

<table>
<thead>
<tr>
<th>id</th>
<th>title</th>
<th>rating</th>
<th>genre</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>&quot;Fury&quot;</td>
<td>&quot;R&quot;</td>
<td>&quot;action&quot;</td>
</tr>
</tbody>
</table>
Also known as a “hierarchical” model. The relational model, in comparison, would be a “flat” model.

The “document” can be encoded as JSON, for instance.
<table>
<thead>
<tr>
<th>what structures?</th>
<th>how do you query?</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>objects, classes (incl. arrays)</td>
</tr>
<tr>
<td>document</td>
<td>documents, collections</td>
</tr>
<tr>
<td>relational</td>
<td>rows (&quot;tuples&quot;), tables (&quot;relations&quot;)</td>
</tr>
</tbody>
</table>

Relational also: no order or dupes, but...
<table>
<thead>
<tr>
<th></th>
<th>good</th>
<th>bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>easily make arbitrary structures; referencing abstracted</td>
<td>not persistent! need a “root”; no declarative queries</td>
</tr>
<tr>
<td>document</td>
<td>heterogeneous collections</td>
<td>queries over multiple collections are awkward</td>
</tr>
<tr>
<td>relational</td>
<td>query language; transactions; fault tolerance</td>
<td>data “splattered” over many tables; no subtypes; exposed references</td>
</tr>
</tbody>
</table>

Relational also: no order or dups, but...
In relational database terms, there are no “joins” in a document database.
some history
<table>
<thead>
<tr>
<th>Year</th>
<th>Event/Invention</th>
<th>Inventor/Creator</th>
</tr>
</thead>
<tbody>
<tr>
<td>1960</td>
<td>LISP</td>
<td>John McCarthy</td>
</tr>
<tr>
<td>1970</td>
<td>Codd's relational model</td>
<td>Ted Codd</td>
</tr>
<tr>
<td>1972</td>
<td>Parnas: On the criteria</td>
<td>David Parnas</td>
</tr>
<tr>
<td>1970</td>
<td>Simula 67</td>
<td>Ole-Johan Dahl</td>
</tr>
<tr>
<td>1976</td>
<td>WWW</td>
<td>Tim Berners-Lee</td>
</tr>
<tr>
<td>1958</td>
<td>LISP</td>
<td>John McCarthy</td>
</tr>
<tr>
<td>1970</td>
<td>Codd's relational model</td>
<td>Ted Codd</td>
</tr>
</tbody>
</table>

Parnas: On the criteria (1972)
(“we must abandon the assumption that a module is one or more subroutines, and instead allow subroutines and programs to be assembled collections of code from various modules”)

Codd's relational model (1970)
LISP (1958)
MongoDB was 2009
oracle: the relational database leader

Stock value versus S&P 500
Current market capitalization: about $175B

Founded 1977

NoSQL is anything non-relational
Tried to find something more recent.
the MongoDB API
three essential notions

“connection”
› database runs as a server; client accesses via connection
› basis of authentication (e.g. SSL)
› connection has a thread, preserves order of operations

“collection”
› group of related documents, not necessarily homogeneous
› target of query and update
› indexes associated with collections

“document”
› unit of storage in a collection
› queries can look at document structure (cf. key-value store)
› all operations are atomic at document level
mongoDB operations

db = client.connect("mongodb://localhost:27017/moviedb")
› connect to database server and return connection

movies = db.collection("movies")
› get collection; create if necessary

movies.insert({title: "Fury", theater: 123, time: "7pm"})
› insert a document into a collection

cursor = movies.find({title: "Fury"})
› find documents matching query, and return cursor

cursor = movies.update({........})
› replace a document (the whole document!)

note: no mutation of individual document fields; everything within a document is replaced at once!
queries

movies.find({})
› find all movies

movies.find({title: "Fury"})
› find movies with title “Fury”

movies.find({title: {$in: ["Fury", "Lucy"]}})
› find movies with a title in the given array

movies.find({title: "Fury", time: "7pm"})
› find movies with title “Fury” showing at 7pm

movies.find({$or: [{title: "Fury"}, {time: "7pm"}]})
› find movies with title “Fury” or showing at 7pm

other features:
$and
$gt, $lt
nested match
array matches
projections & sorts

movies.find({time: "7pm"}, {title: 1, theater: 1})
  › find movies at 7pm, but return only title and theater

movies.find({time: "7pm"}, {time: 0})
  › find movies at 7pm, but drop time from results

movies.find({title: "Fury"}).sort({time: 1})
  › find movies with title "Fury", sort by time

performance issues?
why bother to project?
what impact will sort have?

sort will force all results to be retrieved, even if we iterate over only a few of them
some real queries (another dataset)

```json
{
    "address": {
        "building": "1007",
        "coord": [ -73.856877, 40.848447 ],
        "street": "Morris Park Ave",
        "zipcode": "10462"
    },
    "borough": "Bronx",
    "cuisine": "Bakery",
    "grades": [
        { "date": { "$date": 1393804800000 }, "grade": "A", "score": 2 },
        { "date": { "$date": 1378857600000 }, "grade": "A", "score": 6 },
        { "date": { "$date": 1358985600000 }, "grade": "A", "score": 10 },
        { "date": { "$date": 1322806400000 }, "grade": "A", "score": 9 },
        { "date": { "$date": 1299715200000 }, "grade": "B", "score": 14 }
    ],
    "name": "Morris Park Bake Shop",
    "restaurant_id": "30075445"
}
```

Show in terminal

db.restaurants.find({"borough": "Bronx"}, {name: 1, borough: 1, "grades.grade": 1 });
db.restaurants.find({"borough": "Bronx", "address.zipcode": "10462"}, {name: 1, borough: 1, "grades.grade": 1 });
db.restaurants.find({"grades.score": 0}, {name: 1, "grades.score": 1,})
db.restaurants.find({"grades.score": { $gt: 50 }}, {name: 1, "grades.score": 1,})

Do some sorting, too.

If you happen to have the MongoDB shell installed you can get this sample dataset too. But don’t worry about it for now.
the node.js driver API

an asynchronous API
› all non-instantaneous ops take callback instead of returning

examples
› not `movies.insert({title: "Fury"...}); rest();`
› but `movies.insert({...}, function (err, result)
{rest();})`

sequencing non trivial
› can be “callback hell”
movies.find({title: "Fury"})
  .toArray(function (err, movies_array)
  {
    if (err !== null) {
      // Handle the error.
    } else {
      // Do something with movies_array.
    }
  });
actual use of `find()`, e.g. in web server

```javascript
router.get('/movies', function(req, res) {
    movies.find({title: "Fury"})
        .toArray(function (err, movies_array) {
            if (err !== null) {
                console.log("Got error", err);
                res.send(500);
            } else {
                res.setHeader("Content-Type", "application/json");
                res.send(JSON.stringify(movies_array, null, 2));
            }
        });
});
```
two ways to structure data in mongo
Back to our object-oriented heap.
Give the class 5 minutes to imagine how you would structure this data.
One document in the collection Movies

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

• 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

• A schema defines the structure of your data
  • This structure impacts how you write your queries