Colleen Minor

CS 496: MOBILE/CLOUD SOFTWARE DEVEL

Summer 2016

MongoDB is a document-oriented database. Document-oriented databases store data in a way that is structured such that the system is able to extract metadata. In the case of MongoDB, data is stored in a binary representation called BSON (Binary JSON). Because of this, MongoDB allows documents (a document is what a single object, approximately equivalent to a row in an SQL table, is called) to be retrieved based on their content. For example, say you had a collection called employees that had the String fields “first name”, “last name” and “job,” and the Number fields “started.” To find all of the employees with the job “welder” who started in 1999, you would use the command “db.employees.find({job:”welder”, started: 1999}).” This would return a list of all employees with “welder” listed as their job and 1999 listed as their year started. It also allows for the retrieval of specific fields of documents; if you only wanted a list of the first and last names of welders, you could use “db.employees.find({job:”welder”}, {firstName: 1, lastName 1}) and a list of the names of welders would be retrieved. You can also retrieve docs with the boolean operator $or. To find all employees who’s first name is Tom *or* who’s last name is Jones, you would enter `db.employees.find({$or: [{firstName: “Tom”}, {lastName: “Jones”}] });

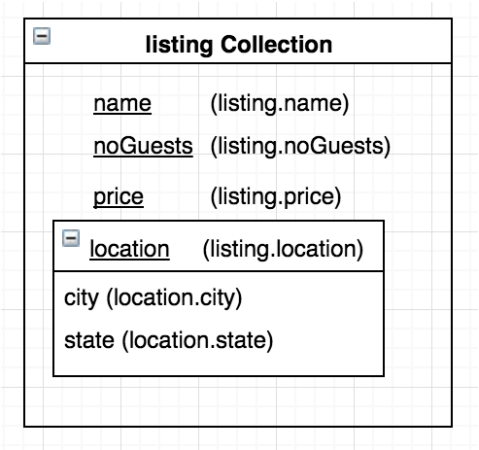
MongoDb also allows you to embed an array of sub-documents into a parent document. Say that employees had a field called “address” and it looked something like, `{address: {street: “1969 SW Cherry”, city: “Newberg”, state: “OR”}}`. To find all employees who are from the state of Oregon, you would use `db.employee.find({“address.state”: “OR”})`. Searching with arrays is similar: say there was a field in the “employees” collection called “skills” that looked like skills: [plumping, cooking, resting]. In order to find only employees that can do plumbing and cooking, you would enter `db.employees.find({skills: [“plumbing”, “cooking”]});`.

For modeling one-to-many relationships, there are two main ways to do it: embedding an array of sub-documents into a parent document, as discussed above, or creating a separate collection and adding document-IDs to an array in the first collection. For my project, I plan to use just one collection and embed array of subdocuments in order to add the “location” field. Remember that a sub-document is a document with a schema of it’s own. I plan to use Mongoose.js with node to connect my project to a Mangodb database on Mlab. Diagrams below:

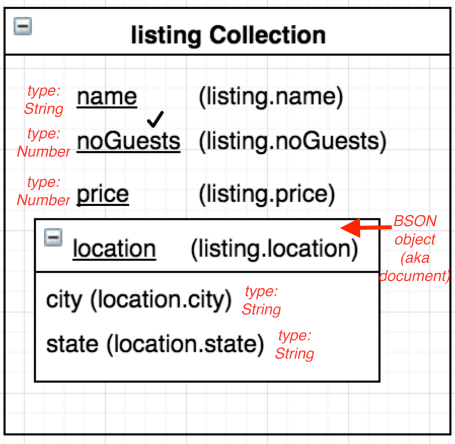
Here is my plan for to create ListingSchema, which will contain the schema locationScema.

```
var locationSchema = db.Schema({
  city:  {type: String, unique: false},
  state: {type: String, unique: false}
});
var ListingSchema = db.Schema({
  location: locationSchema,
  name: {type: String, unique: true},
  noGuests: {type: Number, unique: false}, //This is the number of guests allowed.
  price: {type: Number, unique: false}
});
```

Here is a diagram of the basic layout:



With labels:



The collection is called *listing*, and it's elements are a String called name, a Number called noGuests (the number of guests allowed), a Number called price, and an embedded sub-document called location, which contains a String called city and a String called state. My reason for using this set-up is that it is simple and it prevents confusion. And because of MongoDB's ability to retrieve documents on the basis of content, it is easy to list retrieve all of the listings in a particular city or state, which I might want my API to be able to search by.

A sacrifice that comes with my design is efficiency in the execution of queries. Because I am not using any indexes apart from the default _id index, MongoDB must perform a collection scan, i.e. scan every document in the collection, to select the documents which match my query statements. I justify this sacrifice by the fact that my queries are still less expensive than doing relational database matches, and this option is also simpler.

Another popular non-relational database is CouchDb. CouchDb also uses a document store, but it has some key differences. For one thing, CouchDb organizes its documents via views, and requires that all views be defined up front. This doesn't make a difference in the design I plan to use, because Mongoose also requires that all schemas be designed up front, but if I were using another implementation of MongoDb, MongoDb would allow for new fields to be added to objects in a collection as needed. Another difference is that CouchDB uses HTTP requests to populate or query the database. The HTTP methods POST, GET, PUT and DELETE for the four basic CRUD (Create, Read, Update, Delete) operations on all resources. This would require a different implementation of those operations, if my API were to use those. The closest I could get to my planned implementation would probably be to use the CouchDb client for Nodejs called Cradle, and even then my implementation would look significantly different, and I would need to write a MapReduce function to query the database. A difference that would affect my design would be that CouchDb does not have a built-in way to enforce a "unique" field. Therefor I would have to come up with another way to make sure that multiple listings with the same value for "name" are not saved. That could require creating a pointer document where the taken names are set as the key for each save, and aborting the "new listing" save if that save is unsuccessful (unsuccessful because the key is already taken).

A possible use case for my API is that someone wants to create an app that uses information about what Airbnb places are available, but they discover that Airbnb does not have an official API. So, they would use my API to retrieve the data about available listings. Another use case, in case I design the API with MongoDb's $sort option: A user wants a list of places from most-to-least expensive or vise-versa, but Airbnb does not have that option.