MongoDB Reference

Documentation: http://www.mongodb.org/

MongoDB is a database that stores structured data in a JSON-like format. It can be accessed using Node.js via the Mongoose database driver.

Why MongoDB?

In the Callback Piazza assignment, you stored questions/responses in `localStorage`, accessing/modifying data in a persistent JavaScript object.

Unfortunately, `localStorage` is not suitable for most needs. The data is stored on the client-side, which brings multiple drawbacks:

- Clients are free to modify the data in any way they want, potentially corrupting it.
- The data is local to each user. You can't aggregate data across users. For instance, if this were Facebook, you wouldn't be able to see any of your friends' posts. Their data would be stored in their `localStorage`, which you don't have access to.
- Because it's stored on the client's machine, `localStorage` is limited to 10MB of data, which is literally nothing for most modern applications.

To solve these problems, we store all application-specific data on a central computer that's owned by us, which we call a server. The server handles HTTP requests from clients, accessing its application-specific data and forming responses that are sent back. Just like any other computers, servers can store their data on disk, giving them the capacity of any modern storage device. Additionally, since the server is centralized, it holds the data for all clients, allowing it to aggregate information across users and perform meaningful operations.

As a means of storing data, we use a database—a repository that stores structured information in a way that be accessed easily and efficiently. MongoDB is one such database.

Starting MongoDB

MongoDB stores data in memory and on your filesystem. It runs as a process that listens on a TCP port (usually 27017) on your computer. A server can interface with Mongo by sending requests to Mongo's TCP port.

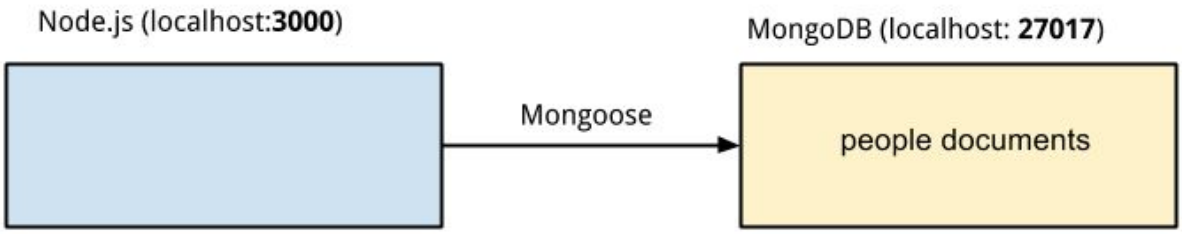
To start running the Mongo process, execute the following command in your terminal:

```
mongod --dbpath /path/to/store/data
```

You can choose any filesystem path to store data in. In our assignments, we'll usually include a data directory in which to store your MongoDB data. Change "/path/to/store/data" to a path that refers to that data directory.

Interfacing with Mongo

Mongo stores data in documents, which you can consider JavaScript objects. These objects, however, can't contain functions.



Mongoose <http://mongoosejs.com/docs/index.html> is a Node.js module that lets us create/read/update/delete documents in Mongo. To begin, we first need to tell Mongoose the format of the documents we're storing. We can specify this format with what's known as a schema, an outline of what fields a document contains and the types of those fields. For instance, let's say we're modeling a person. We could have a model like this:

```
var personSchema = mongoose.Schema({
  age: Number,
  name: String,
  country: String
});
```

For each property, the schema contains its name and type. A person has an age, which is a number, and a name and a country, which are both strings.

Once we have a schema, we need to give it to Mongoose to get back a model that we can use to interface with MongoDB.

```
var Person = mongoose.model('Person', personSchema);
```

We pass in a name and the schema we just created. In return, Mongoose gives us back a model object, which we can now use to access MongoDB.

Creating a document

With our model object, we can create a person document as follows:

```
// create a person document representing Vivek
var vivek = new Person({
      age: 21,
      name: 'Vivek Nair',
      country: 'United States of America'
});

// save the person to Mongo
vivek.save(function(error) {
      if (error) {
            throw error;
      }
      // save has completed
});
```

Notice that creating a new Person document was only the first step. We must execute `vivek.save()` to store the document in Mongo. Additionally, after the `save()` function finishes (i.e. within the callback), an `'_id'` attribute will be added to `vivek` that uniquely identifies the person document.

```
vivek._id // outputs '54556b04e549d3a72ad80b7d'
```

Reading a document

To read a Mongo document, we can use a generated object ID to search for the associated document.

```
var id = '54556b04e549d3a72ad80b7d';
Person.findById(id, function(error,  person) {
      if (error) {
            throw error;
      }

      // perform operations on the retrieved person document
      console.log(person);
});
```

To retrieve all Mongo documents of a given type, use the `find()` method:

```
Person.find(function(error,  people) {
      if (error) {
            throw error;
```

```
        }

        // people is an array of person documents
        console.log(people);
});
```

Updating a document
To update a Mongo document, we first read the document and then freely modify its object properties.

```
var id = '54556b04e549d3a72ad80b7d';
Person.findById(id, function(error,  person) {
        if (error) {
                throw error;
        }

        // update properties on the person object
        person.age = 25;

        // write these changes to the database
        person.save(function(error) {
                if (error) {
                        throw error;
                }
        });
});
```

Note that simply modifying properties is not sufficient to update the document; you must call `save()` on your Mongoose object to store the document back into MongoDB.

Deleting a document
To delete a document, we call `findByIdAndRemove()`, with the first parameter being the ID of the document to remove:

```
var id = '54556b04e549d3a72ad80b7d';
Person.findByIdAndRemove(id, function(error) {
        if (error) {
                throw error;
        }

        // successfully deleted the person document
});
```

In each of these cases, we pass in a callback. The callback's first parameter is an error object; if the operation succeeded (i.e. no errors occurred), the first parameter will be `null`.

Multiple models
Note that you can define multiple schemas and corresponding models. Each model represents a different document type in MongoDB.