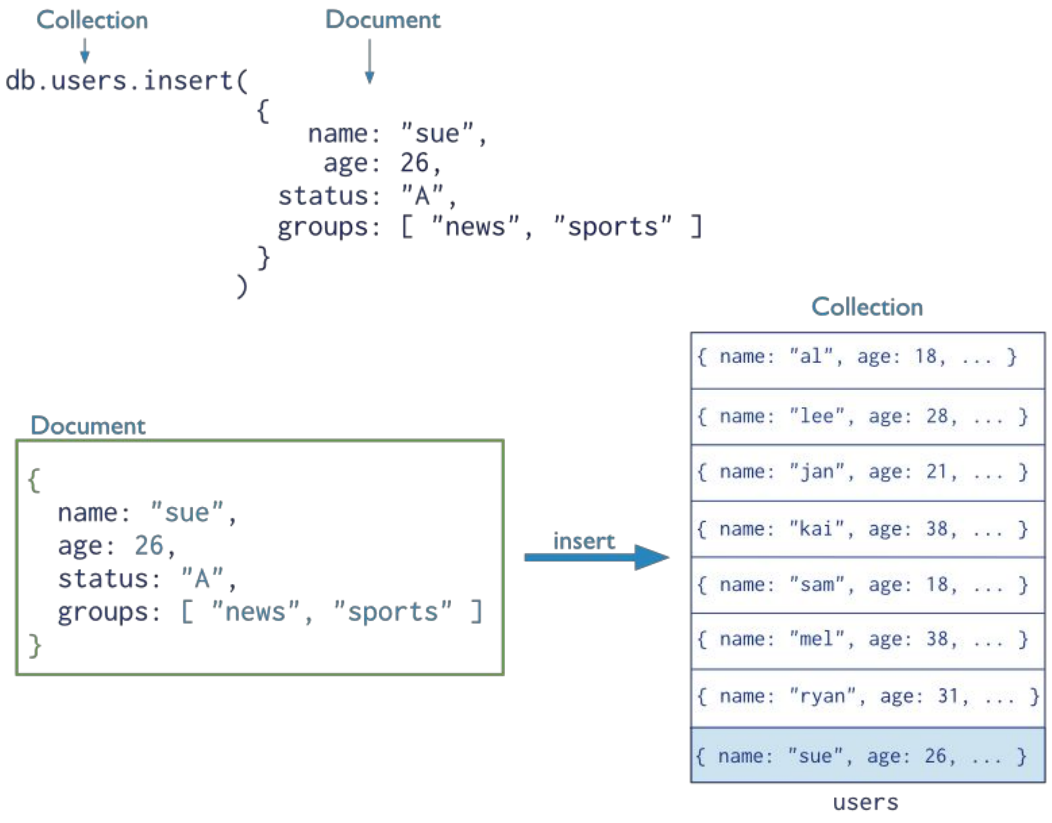
Recitation Notes

1) Introduction to MongoDB & NoSQL (5 minutes)

MongoDB is a high performance document database. It stands in contrast to traditional databases such as MySQL due to its “dynamic schema,” which means that documents in the same collection do not need to have the same set of fields or structure.

- **MySQL (SQL)**: If you have a table of users, must state what field you are storing for each user when creating the table, and then you must use this user structure for all inserts.
- **MongoDB (NoSQL)**: Insert users into a Collection of users--doesn’t matter the format.
- One major way this affects your projects is that you must be careful to preserve consistent representations of your models. Let’s say you have a Collection named ‘Users’ in MongoDB, where you store all users of your application. Once you’ve been working on your app for a while, you decide that you should store users’ email addresses. However, all users who are currently in ‘Users’ don’t have email addresses stored for them. Mongo lets you insert email addresses for users you create in the future, but ‘Users’ will then be in a state where some users have emails and other don’t. This can be dangerous.



```
db.users.insert(
                 {
                    name: "sue",
                     age: 26,
                  status: "A",
                  groups: [ "news", "sports" ]
                 }
               )
```

```
{
  name: "sue",
  age: 26,
  status: "A",
  groups: [ "news", "sports" ]
}
```

```
{ name: "al", age: 18, ... }
{ name: "lee", age: 28, ... }
{ name: "jan", age: 21, ... }
{ name: "kai", age: 38, ... }
{ name: "sam", age: 18, ... }
{ name: "mel", age: 38, ... }
{ name: "ryan", age: 31, ... }
{ name: "sue", age: 26, ... }
```

Fig 1. This is an overview of MongoDB structure.

source: http://docs.mongodb.org/manual/_images/crud-insert-stages.png

We'll be using MongoDB to store a persistent state in our application. With the first stage of the Fritter project, you were unable to store anything persistently--if you ever had to restart the app, all of your data would be gone. This is because we were storing data **in memory**, whereas MongoDB and other databases will store your data **on disk**.

There are a number of Node.js modules built to interface with MongoDB. The most popular one is Mongoose, which is an ORM (object relational mapper). So, Mongoose lets you use server-side JavaScript to query and manipulate data in your database.

Before you can interface between Express & MongoDB, you'll need to make sure MongoDB is fully set up according to the installation instructions here. You can double check that mongo is properly installed by running this command:

```
$ mongo --version
```

Explain that we are just replacing the "models" module of the first Fritter project, and that they can adapt their apps by replacing calls to the in-memory database with calls to MongoDB via Mongoose. Essentially, instead of modifying an in-memory fake "database," we are going to be sending database transactions to MongoDB.

2) Interactive Tutorial (40 minutes)

This recitation walks students through creating an activity tracker application. You can find the repo for the recitation here: https://github.com/kathrynsiegel/6170-mongo-recitation.

Go to the code, and checkout the first branch. This recitation is organized into 6 main parts, each of which has a few exercises. Checking out the next branch will reveal the solutions to the previous part and also the new exercise items.

For part 1, checkout the `part1 branch`.

```
$ git checkout part1
```

Start mongo and the app, and navigate to localhost:3000. You'll see the UI for an activity tracker, but there is no functionality to the form, and no data reported.

Exercise 0 (2 minutes): Walk through the steps to connect Express to MongoDB using Mongoose. Open app.js and point out where mongoose has been imported and connected to MongoDB.

1) Install Mongoose in your project:

   ```
   $ npm install mongoose
   ```

2) Ensure that Mongoose is in your dependencies by looking at the “dependencies” object in package.json.

3) In the main entry point to your Express application (app.js), get the Mongoose module and open a connection to the database:

   ```
   var mongoose = require(‘mongoose’);
   mongoose.connect(‘mongodb://localhost/test’);
   ```

4) Listen for when the connection is successfully established; register a callback for when that happens:

   Note: Remember that we are in the single-threaded world of Javascript. “Expensive” operations, such as connecting and performing operations on a database, should be done asynchronously, so that the main thread is not blocked and is able to accept additional incoming connections.

   ```
   var db = mongoose.connections;
   db.on(‘error’, function() { console.log(‘connection error’); });
   db.once(‘open’, function() {
         // declare schemas and models here
   });
   ```

Exercise 1 (4 minutes): In models/activity.js, declare a schema and make a model for an Activity--i.e. an activity log entry.

Answer:

```
var activitySchema = mongoose.Schema({
      type: String,
      duration: Number,
      intensity: String
});
```

When you are done, to see the answers and go to the next part, do:

```
$ git checkout part2
```

Exercise 2 (4 minutes): Querying MongoDB directly.

Open up a MongoDB shell and run some example commands. Instruct students to go to the MongoDB documentation for more information.

```
$ mongo
 MongoDB shell version: 2.6.4
 connecting to: test
> show dbs
> use test  // switch to db test and create it if it doesn’t exist
> show collections
> db.activities.insert({type: "Jumping Jacks", duration: 30, intensity:
“High”})    // insert into collection activities (create if didn’t exist)
> db.activities.find() // show all items in collection users
> db.activities.update({“_id”: ObjectId("<id from above>")}, {$set:
{"duration": 20}}) // $set is an update operator
> db.activities.find()
> db.activities.remove({“_id” : ObjectId("<id from above>")})
> db.activities.find()
```

Now, we are going to explore sending these commands to MongoDB using Mongoose.

Exercise 3 (10 minutes): Querying the database. Edit the routes/index.js and views/partials/log.ejs files such that the activities stored in the database are sent to the client when the client loads the page.

Step 1: Import the Activity model class

```
var Activity = require(‘../models/activity’);
```

Step 2: Edit the get ‘/’ route so that it loads all of the activities in the activities collection

```
router.get(‘/’, function(req, res, next) {
      Activity.find(function(err, activitiesList) {
            if (err) console.log(err); // handle error
            res.render(‘index’, {activities: activitiesList});
      });
});
```

Step 3: Edit the views/partials/log.ejs to render all of the activities (there should be a for loop already in the source code for you to use).

```
<% for (var i = 0; i < activities.length; i++) { %>
      <tr>
            <td><%= activities[i].type %></td>
            <td><%= activities[i].duration %></td>
            <td><%= activities[i].intensity %></td>
      </tr>
<% } %>
```

Now, run the app, and we see the entry we manually entered into the database shown on the screen.

Exercise 4 (5 minutes): Inserting records into the DB from the app. Modify the routes/index.js file to allow users to create activity records.

```
$ git checkout part3
```

TA note: I have added a lot of client-side JS to minimize live coding. The code lives in public/javascripts/main.js, and performs a post request to '/' using the JQuery post function when the "Submit" button is clicked.

Answer:

```
Activity.create({ type: activity.type, duration: activity.duration,
intensity: activity.intensity}, function(err, record) {
      if (err) console.log(err);
      res.send({success: true, activity: record});
})
```

```
$ git checkout part4
```

The part4 adds the client-side JavaScript (again in public/javascripts/main.js) that inserts into the table on the page. Now, if you submit the form, the client-side JS registers the response and displays the result on the page.

Exercise 5 (5 minutes): Discuss how editing and deleting records follows the same format as inserting records.

Step 1: Use AJAX to send a PUT (for edit) or DELETE (for delete) request to the server.
Step 2: Create the route that receives the request.
Step 3: Make a call to the database via Mongoose (look up Mongoose documentation)
Step 4: Send the data returned by the database query back to the client.

Exercise 6 (5 minutes): Custom instance methods. Define an instance method on activitySchema called getDescription, which should return the concatenation of the activity type and intensity, separated by a dash. Note that if you have an Activity object called "run", you would call this method by stating "run.getDescription."
Answer:

```
activitySchema.methods.getDescription = function(callback) {
      return this.type + “-” + this.intensity;
}
```

```
$ git checkout part5
```

When you run the code in part5, I console.log() out the output of the getDescription method in the routes/index.js file, so you should see it in the server logs.

Exercise 7 (5 minutes): Validators. Write a validator ensuring that the intensity is either “Low”, “Medium”, or “High.” What happens if the validator method returns false? (Answer: an error is returned back to the transaction request in the routes file.)

Answer:

```
activitySchema.path(‘intensity’).validate(function(value) {
      return /Low|Medium|High/i.test(value);
}, ‘Invalid intensity’);
```

```
$ git checkout part6
```

TA note: I’ve added code in the routes/index.js file to console.log() the error and send it back to the client. I’ve also added code that pops up an alert when validation fails. Finally, on the view, I’ve changed the intensity from a dropdown to a text input, so it should be easy to show what happens when an invalid input is submitted.

3) Review of RESTful routes (5 minutes)

- GET - fetch a resource from the server
  - GET /users
  - GET /users/:id
- POST - send resource to the server
  - POST /users
- PUT - modify resource on the server
  - PUT /users/:id
- DELETE - delete resource on the server
  - DELETE /users/:id