COMP-308 Winter 2018

Lesson 4 Review

- **Express**
  - Fast, minimalist **web framework for Node.js**
  - Runs on top of Connect
  - Adds **HTML templates**, extends response object, etc.
- **Installation and configuration**
  - Use **npm** to install it
  - **package.json** file keeps track of dependencies
  - Load express module in memory using **require** method
  - Mount middleware functions
  - Tell server to listen for requests
- **Application object**
  - Contains methods to configure your express application
  - **set**, **get**, **engine**, **use**, VERB – **get** and **post**, **route**, **param** methods
- **Request object**
  - Contains methods and properties that provide info about current request
  - **query, params, body, path, host, ip, cookies** properties
  - **param** method
- **Response object**
  - Handles and responds requests

Lesson 4 Review

- **Response object methods**
  - **status**, **set**, **cookie**, **redirect**, **send**, **json**, **render**
- External **middleware**
  - morgan
  - body-parser
  - method-override
  - compression
  - express.static
  - cookie-parser
  - session
- MVC Pattern
  - **Models**
  - **Controllers**
  - **Views**
- **Express application structure**
  - **Horizontal** for simple apps
    - **app**, **config**, and **public** folders
  - **Vertical** for apps with multiple features
    - Nests several horizontal structures, one for each feature
  - File-naming conventions
    - *feature.server.controller.js*
    - *feature.client.controller.js*
- **Developing Express applications**
  - Setup the structure
  - Create appropriate modules to configure and initialize express.
  - Implement controllers, views, routes, models.

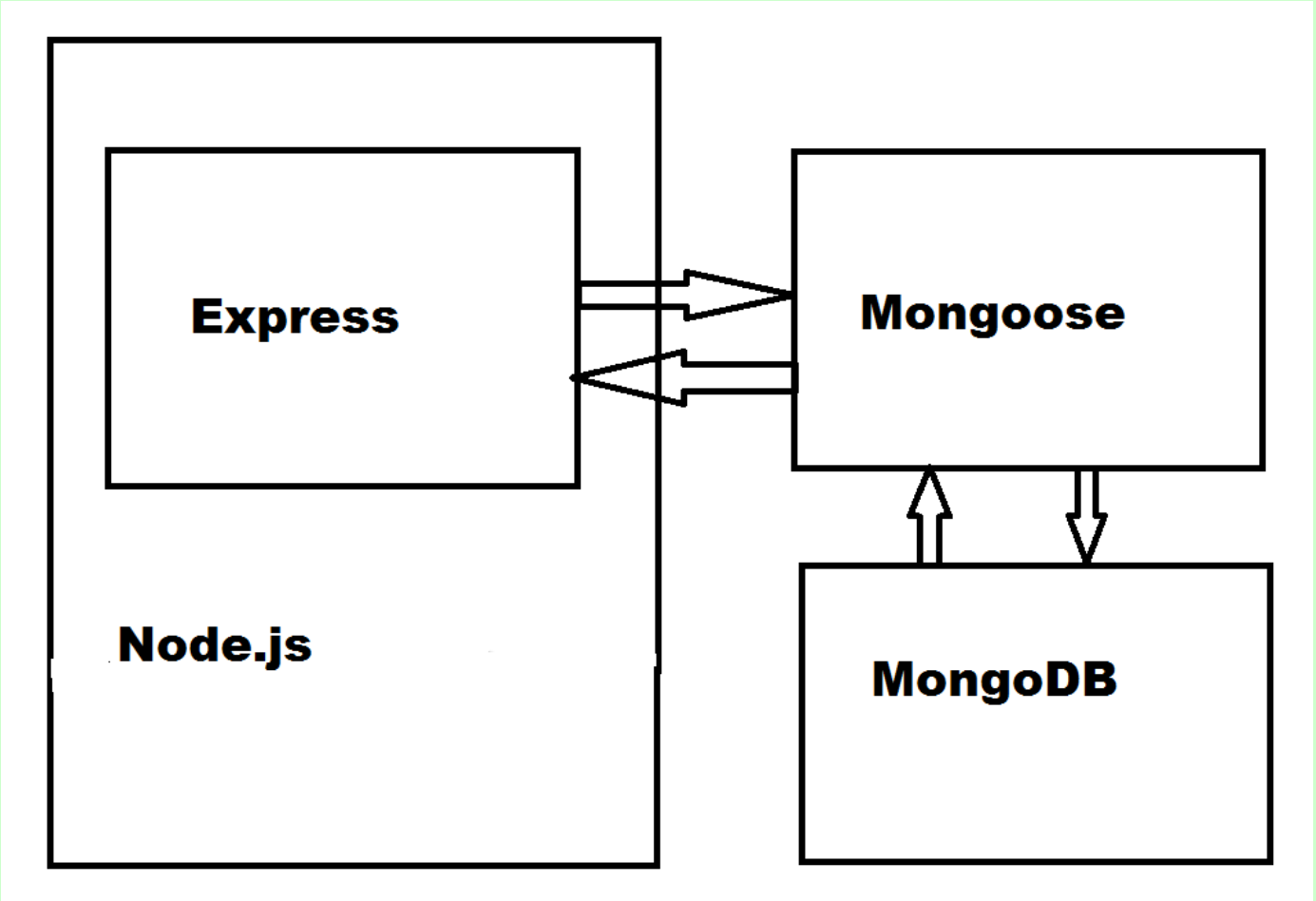
Introduction to MongoDB

Objectives:

- Define NoSQL.
- Examine MongoDB's document model, query language, and deployment architecture.
- Work with MongoDB shell.
- Connect to MongoDB database using Mongoose.

Adding MongoDB to Express app

- Express application architecture including a MongoDB database:



Introduction to NoSQL

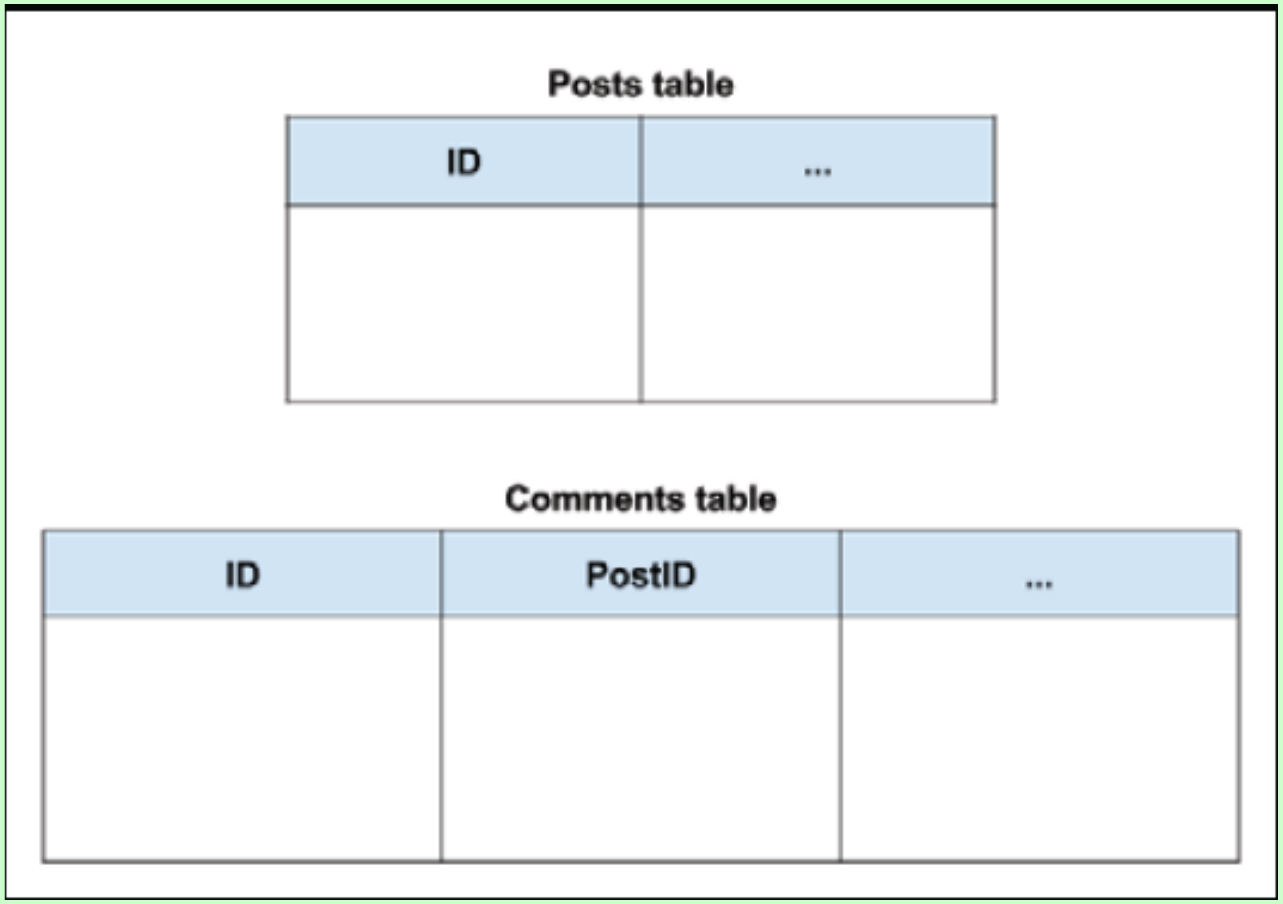
- The growth of Web raised the need for **larger**, **more scalable** storage solutions.
  - a variety of **key-value** storage solutions were designed for **better availability**, **simple querying**, and **horizontal scaling**.
- This new kind of data store became more and **more robust**, offering many of the features of the relational databases.
- Different storage design patterns emerged, including **key-value** storage, **column storage**, **object storage**, and the most popular one, **document storage**.

Relational vs Document-oriented DB

- In a common relational database, **data is stored in different tables**, often connected using a **primary to foreign key relation**.
- A program will later reconstruct the model using various SQL statements to arrange the data in some kind of hierarchical object representation.
- **Document-oriented databases** handle data differently.
  - Instead of using tables, they **store hierarchical documents in standard formats**, such as **JSON** and **XML**.

Relational DB example

- blog post model – data stored in different tables:



Document-oriented DB example

- In a document-based database, the blog post will be **stored completely as a single document** that can later be queried.
- For instance, **in a database that stores documents in a JSON format**, the blog post document would probably look like the following code snippet:

```
{
    "title": "First Blog Post",
    "comments": [
    ]
}
```

- This model will allow **faster read operations** since your application won't have to rebuild the objects with every read.

Intro to MongoDB

- Created in 2007 by Dwight Merriman and Eliot Horowitz
- Derived from the word **humongous**, MongoDB was able to **support complex data storage**, while maintaining the **high-performance** approach of other NoSQL stores.
- **eBay** and The **New York Times** began to use MongoDB data storage in their production environment.

MongoDB Key features

- JSON-like storage format named BSON (**BinaryJSON) -** a **document** consists of a **list of elements**, each with a **string typed field** name and a **typed field value**.
  - These documents **support all of the JSON specific data types** along with other data types, such as the **`Date`** type.
  - use of the **_id field as primary key -** The **_id** field value will usually be **a unique identifier type**, named **ObjectId**, that is either generated by the application driver or by the mongod service.
  - If driver fails to provide a **_id** field with a unique **`ObjectId`**, the **`mongod`** service will **add it automatically**.

BSON example

- A BSON representation of the blog post object from the previous example would look like the following code snippet:

```
{
    "_id": ObjectId("52d02240e4b01d67d71ad577"),
    "title": "First Blog Post",
    "comments": [
    ...
    ]
}
```

MongoDB ad hoc queries

- MongoDB is able to run queries by indexing BSON documents and using a **unique query language**.
- The following SQL statement example:

```
SELECT * FROM Posts WHERE Title LIKE '%mongo%';
```

- Can be replicated in MongoDB as follows:

```
db.posts.find({ title:/mongo/ });
```

- MongoDB is **almost as query-able** as your traditional relational database.

MongoDB indexing

- An index **maps document fields** and can tell the engine which documents are compatible with the query statement.
- For instance, we want to retrieve all the posts that have more than 10 comments and our document is defined as follows:

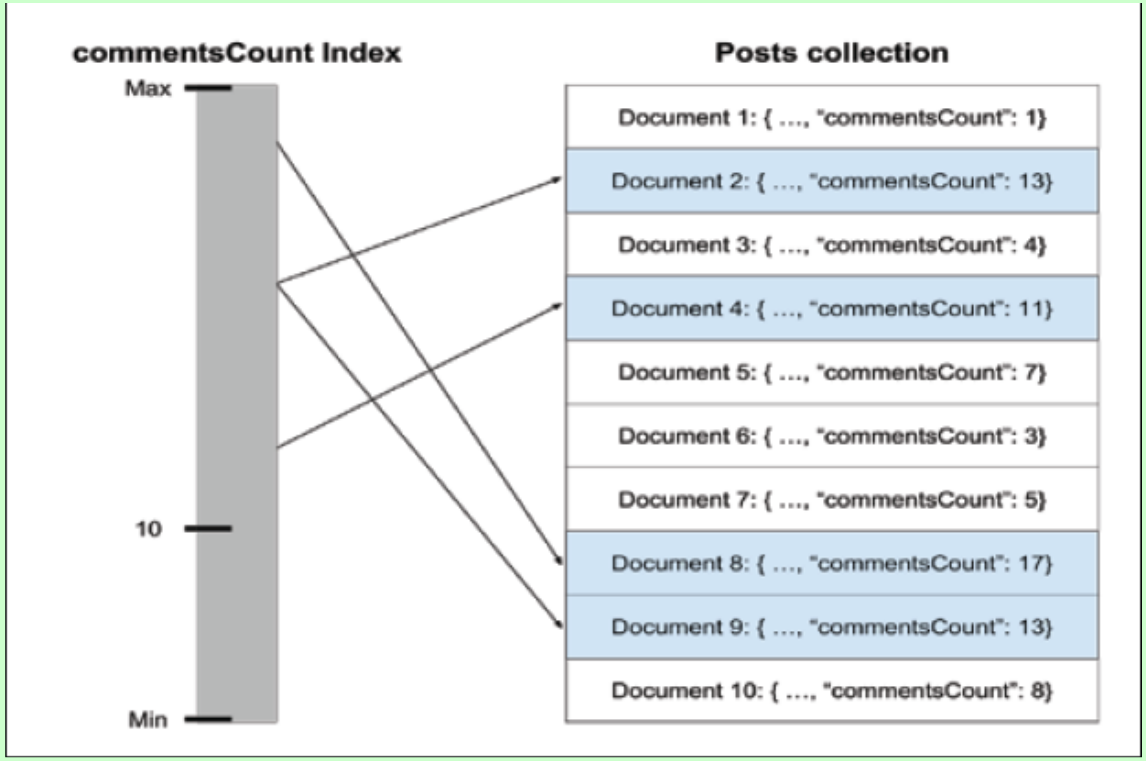
```
{
    "_id": ObjectId("52d02240e4b01d67d71ad577"),
    "title": "First Blog Post",
    "comments": [
    ],
    "commentsCount": 12
}
```

- The MongoDB query that requests for **documents with more than 10 comments** would be as follows:

```
db.posts.find({ commentsCount: { $gt: 10 } });
```

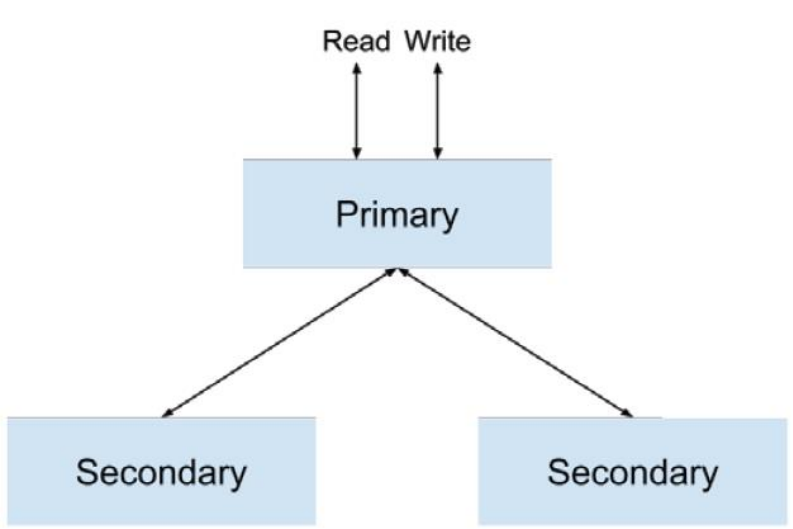
MongoDB indexing

- If a *commentCount* index was defined, then MongoDB would only have to check which documents have *commentCount* larger than 10, before retrieving these documents.
- The following diagram illustrates how a commentCount index would work:



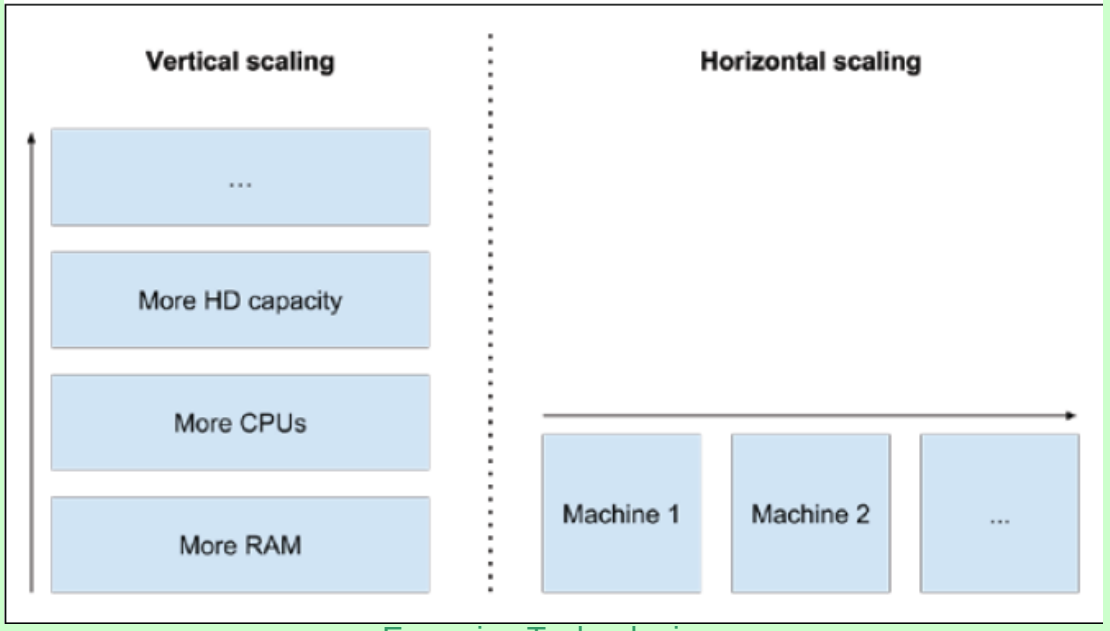
MongoDB replica set

- Replication of databases helps **protect your data to recover from hardware failure** and increase read capacity.
- A replica set is a set of MongoDB services that host the same dataset.
- One service is used as the **primary** and the other services are called **secondaries**.
- All of the set instances support ***read*** operations, but only the **primary instance is in charge of *write* operations**



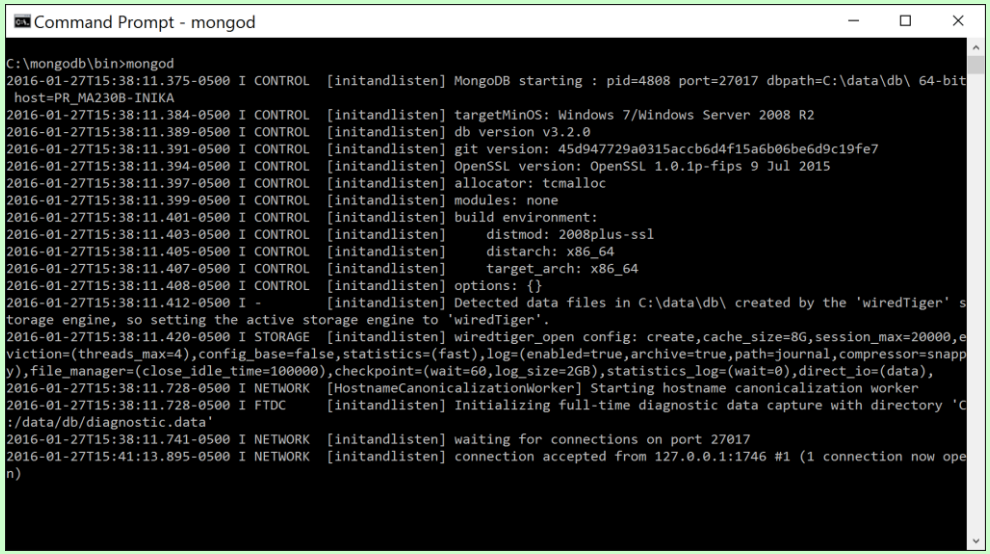
MongoDB sharding

- Scaling is a common problem with a growing web application.
- The various approaches to solve this issue can be divided into two groups:
  - **vertical** scaling - increasing single machine resources
  - **horizontal** scaling - with multiple machines
  - **MongoDB supports horizontal scaling**, which it refers to as **sharding**

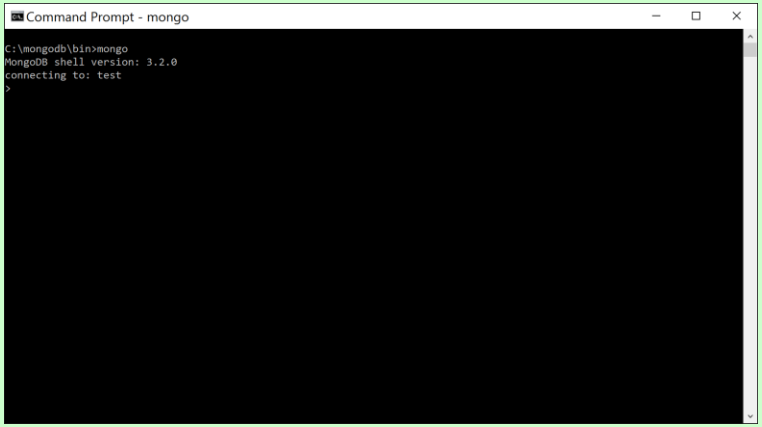


MongoDB shell

- To interact with MongoDB, you'll use the MongoDB shell. First run **mongod**:



- Then run **mongo**:



Creating and showing databases

- MongoDB shell will automatically connect to the default *test* database.
- To switch to another database called mean by executing the following command:

use mean

- Or, to run the shell executable with the database name as an argument, as follows:

mongo mean

- To list all the other databases in the current MongoDB server, just execute the following command:

show dbs

MongoDB collections

- A MongoDB *collection* is a **list of MongoDB documents** and is the **equivalent of a relational database table**.
- A collection is created when the first document is being inserted.
- Unlike a table, a collection doesn't enforce any type of schema and can host different structured documents.
- Insert and retrieve example:

  db.posts.insert({"title":"First Post**", "user": "**bob**"})**

  db.posts.find()
- Show all available collections:

  show collections
- To drop a collection: db.posts.drop()

MongoDB CRUD operations

- **Create a new document** – use **insert**(), **update**(), or **save**() methods.
- **Read** a document – use **find**() method.
- **Update** a document - use **update**() or **save**() methods.
- **Deleting** documents – use **remove**() method.

Creating a document using insert()

- The most common way to create a new document is to use the **insert()** method.
- The insert method takes a single argument that represents the new document.
- To insert a new post, just issue the following command in the MongoDB shell:

```
db.posts.insert({"title":"Second Post", "user": "alice"})
```

Creating a document using update()

- The **update()** method is usually used to **update an existing document**.
- You can also use it to **create a new document**, if no document matches the query criteria, using the following **upsert** flag:

```
db.posts.update({
    "user": "alice"
    }, {
    "title": "Second Post",
    "user": "alice"
    }, {
    upsert: true
    })
```

Creating a document using save()

- Another way of **creating a new document** is by calling the **save()** method, passing it a document that either doesn't have an _id field or has an _id field that doesn't exist in the collection:

```
db.posts.save({"title":"Second Post", "user": "alice"})
```

- This will have the same effect as the update() method and will create a new document instead of updating an existing one.

Finding all the collection documents

- To **retrieve all the documents** in the posts collection, you should either pass an empty query to the **find()** method or not pass any arguments at all.
- The following query will retrieve all the documents in the posts collection:

```
db.posts.find()
```

- Furthermore, performing the same operation can also be done using the following query:

```
db.posts.find({})
```

Retrieve a specific document

- Use an **equality condition query** that will grab all the documents, which comply with that condition.
- For instance, to retrieve all the posts created by alice, you will need to issue the following command in the shell:

db.posts.find({ "user": "alice" })

- This will retrieve all the documents that have the *user* property equal to alice.

Build more complex queries

- Using **query operators**, you can look for different sorts of conditions.
- For example, to retrieve all the posts that were created by either alice or bob, you can use the following **$in** operator:

```
db.posts.find({ "user": { $in: ["alice", "bob"] } })
```

- Like in SQL, you can use AND/OR operators to build multiple condition query statements.
- To perform an AND query, you simply add the properties you'd like to check to the query object.
- For instance, take look at the following query:

```
db.posts.find({ "user": "alice", "commentsCount": { $gt: 10 } })
```

Build more complex queries

- An OR query is a bit more complex because it involves the **$or** operator.
- To understand it better, take a look at another version of the previous example:

```
db.posts.find( { $or: [{ "user": "alice" }, { "user": "bob" }] })
```

- Like the query operators example, this query will also **return all the posts created by either bob or alice**.

Updating existing documents

- The **update()** method takes three arguments to update existing documents:
  - the **selection criteria** that indicate which documents to update
  - the second argument is the **update statement**
  - The third argument is the **options object**.

```
db.posts.update({
"user": "alice"
}, {
$set: {
"title": "Second Post"
}
}, {
multi: true
})
```

Using Mongoose

- Node.js Object Document Mapper (ODM) module that adds MongoDB support to your Express application
- To install it add in the dependencies section of package.json file:

```
"mongoose": "~4.3.7“
```

- Then run **npm install**
- To connect to MongoDB, you will need to use the MongoDB connection URI in *config/env/development.js*:

```
module.exports = {
    db: 'mongodb://localhost/mean-book',
    sessionSecret: 'developmentSessionSecret'
};
```

Using Mongoose

- Now in your *config* folder, create a new file named *mongoose.js* that contains the following code:

```
var config = require('./config'),
mongoose = require('mongoose');
module.exports = function() {
var db = mongoose.connect(config.db);
return db;
};
```

Using Mongoose

- Change server.js file, and change it to look like the following code snippet:

```
process.env.NODE_ENV = process.env.NODE_ENV ||
'development';
var mongoose = require('./config/mongoose'),
express = require('./config/express');
var db = mongoose();
var app = express();
app.listen(3000);
module.exports = app;
console.log('Server running at http://localhost:3000/');
```

- Run **node server**

Using a Schema

- Mongoose uses a **Schema** object **to define the document list of properties**, each with its own **type** and **constraints**, **to enforce the document structure**.
- After specifying a schema, you will go on to **define a Model constructor** that you'll use **to create instances of MongoDB documents**.
- Let's **define a user schema** and **model**, and how to use a model instance **to create, retrieve, and update user documents**.

Creating a Schema

- Create a new file named *user.server.model.js* In *app/models* folder. Paste the following lines of code:

```
var mongoose = require('mongoose'),
Schema = mongoose.Schema;
var UserSchema = new Schema({
firstName: String,
lastName: String,
email: String,
username: String,
password: String
});
// use schema to define the User model
mongoose.model('User', UserSchema);
```

Registering the User model

- Include the user.server.model.js file in your Mongoose configuration file in order to register the User model.
  - Change your config/mongoose.js file to look like the following code snippet:

```
var config = require('./config'),
mongoose = require('mongoose');
module.exports = function() {
    var db = mongoose.connect(config.db);
    require('../app/models/user.server.model');
    return db;
};
```

Creating new users using save()

- Under the *app/controllers* folder, create a new file named *users.server.controller.js* and paste the following lines of code:

```
var User = require('mongoose').model('User');
exports.create = function(req, res, next) {
    var user = new User(req.body);
    user.save(function(err) {
        if (err) {
            return next(err);
        } else {
            res.json(user);
        }
    });
};
```

Specifying user-related routes

- Begin by creating a file named *users.server.routes.js* inside the *app/routes* folder.
- In this newly created file, paste the following lines of code:

```
var users =
require('../../app/controllers/users.server.controller');
module.exports = function(app) {
    app.route('/users').post(users.create);
};
```

Modifying express.js and testing the app

- Change your *config/express.js* file to include look like the following line:

```
require('../app/routes/users.server.routes.js')(app);
```

- Run **node server**
- To create a new user, perform an HTTP POST request to the base users route, and make sure the request body includes the following JSON:

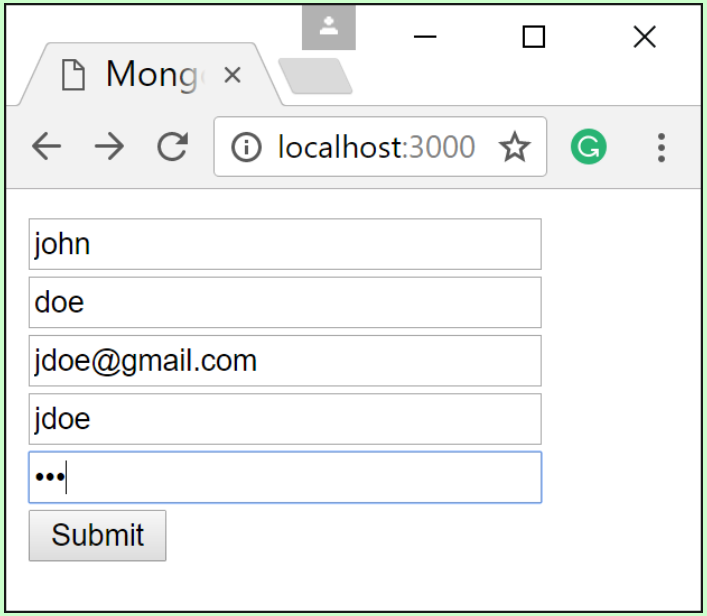
```
{
"firstName": "First",
"lastName": "Last",
"email": "user@example.com",
"username": "username",
"password": "password"
}
```

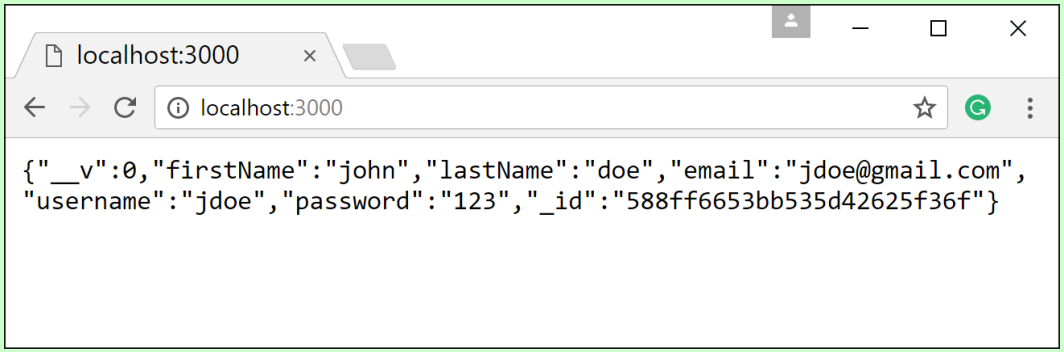
MongooseTest1 example

- Run *mongod*
- Run *mongo*
- Run MongooseTest1 example
- Goto localhost:3000
- Enter the data
- Click submit
- Goto to mongo monitor:

  use mean-book

  db.posts.find()





References

- Textbook
- http://expressjs.com/
- http://www.tutorialspoint.com/mongodb/mongodb_quick_guide.htm
- http://mongoosejs.com/docs/
- http://www.tutorialspoint.com//nodejs/nodejs_express_framework.htm