香港中文大學
The Chinese University of Hong Kong

MONGODB BASICS

CSCI2720 2022–23 Term 1

Building Web Applications

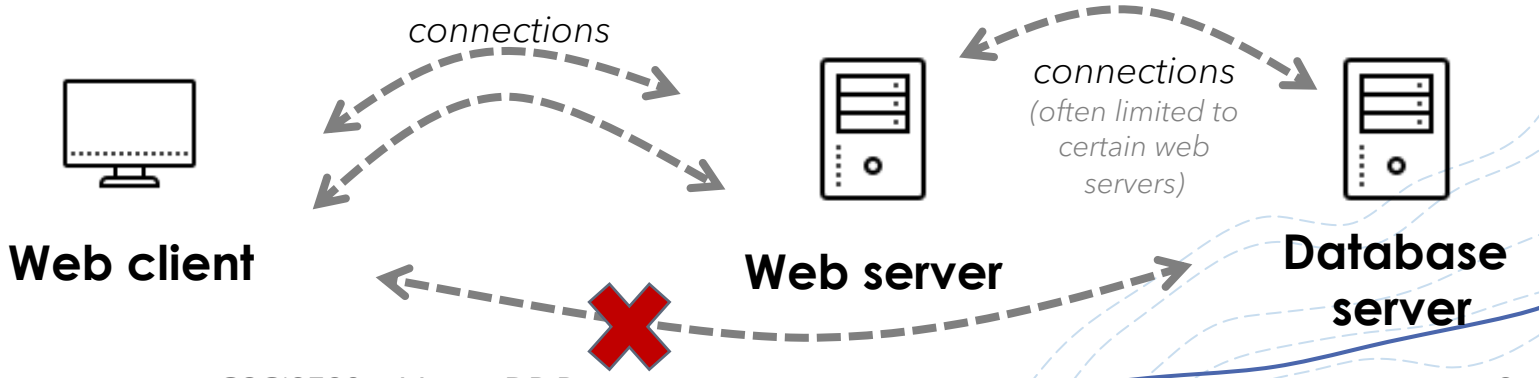
Dr. Chuck-jee Chau *and previous contributors*
chuckjee@cse.cuhk.edu.hk

OUTLINE

- MongoDB and NoSQL
- Using MongoDB
- CRUD Operations in MongoDB
- Other cloud databases
- Mongoose: connecting to MongoDB
- Schema and model
- CRUD in Mongoose
- Documents in another collection

DATABASE SYSTEMS

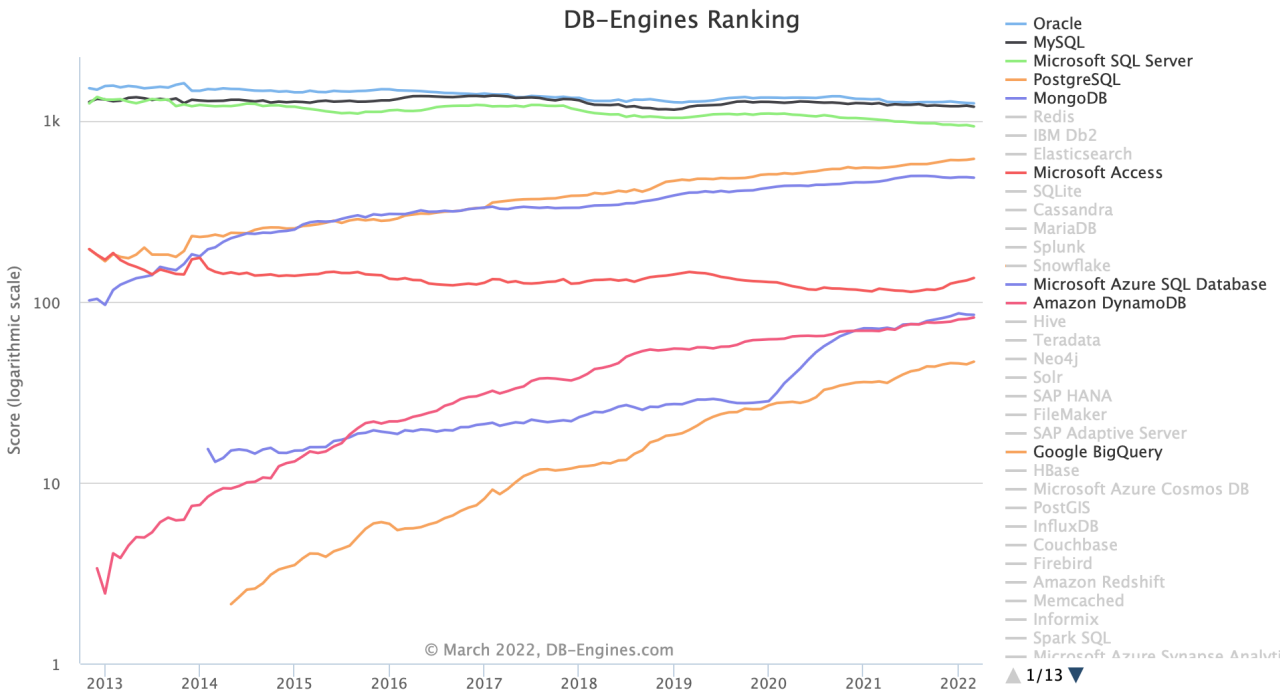
- A database server is often used for carefully organized data, for retrieval by the web server
  - *e.g.*, web user, shop inventory, message board, …
- Relational database: tables of rows and columns
- Non-relational (***NoSQL***) database: flexible documents



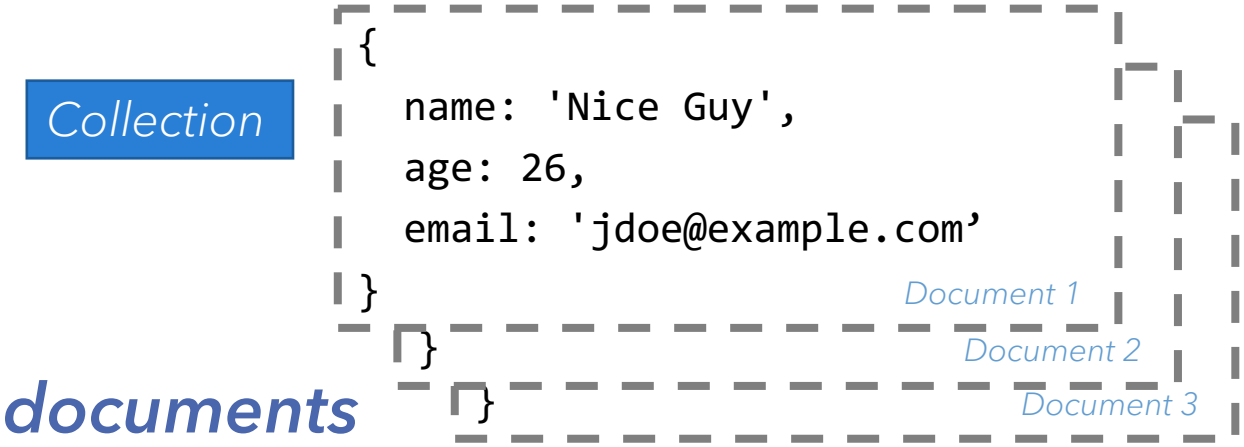
DATABASE SYSTEMS

- ***MongoDB***: An open-source NoSQL database management system (DBMS)
  - Since 2009
  - NoSQL – lack of support of the concept of *relations*
  - Not easy to implement *joins* in query
  - Supporting dynamic schemas

See: https://db-engines.com/en/ranking_trend



DOCUMENT



- MongoDB stores data records as ***documents***
  - Document ≈ JS object ≈ Row in a relational DB
  - A ***collection*** is a group of documents
    - Collection ≈ Array of objects ≈ Table in a relational DB
  - Diversity and scalability
- A MongoDB database holds one or more collections
  - A MongoDB server can hold one or more databases

- *MongoDB vs. MySQL:* https://www.simform.com/mongodb-vs-mysql-databases/

DOCUMENT

- A MongoDB document is a JSON-style data structure composed of field-and-value pairs
  - ***BSON*** (Binary JSON): a binary-encoded serialization of JSON documents
  - The value of a field can be any of the BSON types (*String*, *Double*, *32/64-bit integer*, etc.), including other *documents*, *arrays*, and *arrays of documents*
  - *See: https://docs.mongodb.com/manual/reference/bson-types/*

DOCUMENT

- By default, MongoDB automatically assigns a unique value of type **`ObjectId`** in a compulsory field **`_id`**
- **`_id`** serves as the ***primary key***
  - It is always the first field in the document
  - Its value must be unique
  - It can be a value of any type except array

USING MONGODB

- Two ways to use MongoDB
  - *Local installation*: More controllable, allow testing locally, community (free) vs. enterprise versions
    - Guide: *https://docs.mongodb.com/manual/administration/install-community/*
  - *MongoDB Atlas*: The cloud service where both free-tier and paying options are available
    - Guide: *https://docs.atlas.mongodb.com/getting-started/*
- Directly access to the ***cloud*** or ***local*** server
  - Mongo Shell (command line interface)
  - Mongo Compass (graphical user interface)
  - Web interface (for cloud only)

ACCESSING MONGODB

- In a web application, the backend (e.g. Node.js) is responsible to access the database server
- Two possible ways to access MongoDB from Node.js

1. Use MongoDB directly with ***Mongo Shell commands*** in Node.js, using the official MongoDB Node.js driver
   - *See: https://docs.mongodb.com/drivers/node/*
2. Yet, we will use the ***Mongoose*** module for an extra layer between MongoDB and Node.js to allow more control on data models

WHAT IS MONGOOSE?

- An Object Data Modeling (ODM) library on top of MongoDB
- Supports ***schemas*** to describe documents
  - Type check and automatic type conversion
  - Check if a value is unique
  - Check if a required value is omitted
- Automatic generation of data model from schema
- Simplify interaction with MongoDB from Node.js

CONNECTING TO MONGODB

```
const mongoose = require('mongoose'); // mongoose needs to be installed with npm

// testDb is the database name, for example
const dbUri = "mongodb://localhost/testDb";
mongoose.connect(dbUri);

// mongoose.connection is an instance of the connected DB
const db = mongoose.connection;

// Upon connection failure
db.on('error', console.error.bind(console, 'connection error:'));
// Upon opening the database successfully
db.once('open', function () {
  console.log("Connection is open...");
  /* further actions which depends on db... */
});
```

CONNECTING TO MONGODB

- **`mongoose.connect()`** connects to the database if it exists
  - Otherwise, it creates the database and then connects to it

```
// Additional options including DB username and password
// can be passed to connect() in the 2nd parameter
const options = {
  user: 'myDatabaseUserName',
  pass: 'myDatabasePassword',
  dbName: 'csci2720Db'
}

mongoose.connect(dbUri, options);
```

DEFINING SCHEMA

- Schema describes a document in a collection
  - Properties like **`required`** and **`unique`** are enforced
  - **`type`** affects how property values are casted

```
const mongoose = require('mongoose');
const Schema = mongoose.Schema;

const UserSchema = Schema({
    name: { type: String, required: true },
    email: { type: String, unique: true, required: true },
    password: { type: String, required: true }
});
```

DEFINING SCHEMA

```
// This example illustrates how to describe a complex document
// ( Source: https://mongoosejs.com/docs/guide.html)
const mongoose = require('mongoose’);
const Schema = mongoose.Schema;

const blogSchema = new Schema({
  title:  String,
  author: String,
  body:   String,
  comments: [{ body: String, date: Date }],
  date: { type: Date, default: Date.now },
  hidden: Boolean,
  meta: {
    votes: Number,
    favs:  Number
  }
});
```

SCHEMA TYPES

- Possible types (**`SchemaType`**)
  - `String`
  - `Number`
  - `Date`
  - `Buffer`
  - `Boolean`
  - `Mixed`
  - `ObjectId`
  - `Array`

- Keys may also be assigned nested objects containing further key/type definitions
- *See: https://mongoosejs.com/docs/schematypes.html*

15

FROM SCHEMA TO MODEL

- Models in Mongoose allow easy creation of documents
  - Syntax similar to creating a new JS object
    - Documents are instances of a model
- Name of the corresponding collection should be the small-letter ***plural form*** of the model name
  - "user*s*" in this example

```
// Compiling the Schema into a Model
User = mongoose.model('User', UserSchema);
// 'User' is the model name, pointing to 'users' collection
```

CRUD OPERATIONS

- Common operations for data items
  - ***C***reate
  - ***R***ead / Retrieve
  - ***U***pdate
  - ***D***elete
- You will come across these operations quite often in databases on the web
- CRUD operations are supported by Mongoose query functions
  - Queries are executed ***asynchronously***, and results are passed to ***callbacks***

CALLBACK FUNCTIONS IN MONGOOSE

- All Mongoose callback functions use the pattern

```
callback(error, results)
```

- Unsuccessful ➔ **`error`** is an *error document*, and **`results`** is **`null`**
- Successful ➔ **`error`** is **`null`** and **`results`** depends on the operation, *e.g.*,
  - **`findOne()`** ➔ null or single document
  - **`find()`** ➔ A list of documents
  - **`count()`** ➔ the number of documents
  - **`update()`** ➔ the number of documents affected
  - etc.

CREATION (CRUD)

```
// Create a document and return its instance
User.create({
  name: 'John',
  email: 'john@example.com',
  password: '123'
}, (err, user) => {
  if (err)
    return handleError(err);
  // Here "user" is an instance of the created document
});
```

Note: In practice, we should save a hashed version of the password instead of the original password

RETRIEVAL (CRUD)

```
// Get all instances (from collection 'Users')
User.find( (err, results) => {
  if (results.length > 0) {
    // Iterate through results here ...
  }
});

// Retrieve all users named 'John'
User.find({ name: 'John' }, (err, results) => { ... });

// Retrieve at most one instance
User.findOne({ email: '...' }, (err, result) => {
  // "result" is either null or an instance of matched document
});
```

EXECUTING MODEL METHODS

- Any model method which involves specifying query conditions can be executed in two ways (at least!)
  1. When a callback function is passed, the operation will be executed immediately with the results passed to the callback

```
// Find all users named 'John' but retrieve
// only 'name' and 'email' fields
User.find({ name: 'John' }, 'name email',
  (err, results) => {
    if (err)
      return handleError(err);
    // Process results here ...

});
```

EXECUTING MODEL METHODS

2. When no callback function is passed, an instance of ***Query*** is returned, which can be refined and executed

```
const query = User.find({ name: 'John' });

// selecting only 'name' and 'email' fields
query.select('name email');

query.exec( (err, results) => {
  if (err)
    return handleError(err);

  // Process results here ...
});
```

OTHER WAYS TO QUERY

```
Person                    // Using a JSON doc
  .find({
    occupation: /host/,    // a regex
    'name.last': 'Ghost',
    age: { $gt: 17, $lt: 66 },
    likes: { $in: ['vaporizing', 'talking'] }
  })
  .limit(10)
  .sort({ occupation: -1 })             // Desc order
  .select({ name: 1, occupation: 1 })   // Projection
  .exec(callback);
```

```
Person                    // Using query builder
  .find({ occupation: /host/ })
  .where('name.last').equals('Ghost')
  .where('age').gt(17).lt(66)
  .where('likes').in(['vaporizing', 'talking'])
  .limit(10)
  .sort('-occupation')
  .select('name occupation')
  .exec(callback);
```

QUERY OPERATORS

- Standard MongoDB operators are supported:
  - **`$eq`** equal value
  - **`$gt`** greater than a value
  - **`$gte`** greater than or equal to
  - **`$in`** in an array
  - **`$lt`** less than a value
  - **`$lte`** less than or equal to
  - **`$ne`** not equal to
  - **`$nin`** not in an array
- *See more: https://docs.mongodb.com/manual/reference/operator/query/*

UPDATE (CRUD)

```
// Update by querying
const conditions = { email: 'john@example.com' };

User.findOne( conditions, function( err, user ) {
  if (err)
    return handleError(err);

  if (user != null) {
    user.name = 'John Doe';
    user.save();
  }
}
```

UPDATE (CRUD)

- Further ways to update documents

```
// Locate a user by email, and then change his name
const conditions = { email: 'john@example.com' },
    update = { $set: { name: 'John Doe' }};

User.update(conditions, update, callback);
// Note: Result passed to callback is # of updated documents
```

```
// Change all users whose has the name "John" to "John Doe"
const conditions = { name: 'John' },
    update = { $set: { name: 'John Doe' }},
    options = { multi: true };

User.update(conditions, update, options, callback);
```

See: http://mongoosejs.com/docs/api.html#model_Model.update

OTHER STATIC FUNCTIONS

- More Mongoose functions returning a Query object:
  - `Model.deleteMany()`
  - `Model.deleteOne()`
  - `Model.find()`
  - `Model.findById()`
  - `Model.findByIdAndDelete()`
  - `Model.findByIdAndRemove()`
  - `Model.findByIdAndUpdate()`
  - `Model.findOne()`
  - `Model.findOneAndDelete()`
  - `Model.findOneAndRemove()`
  - `Model.findOneAndReplace()`
  - `Model.findOneAndUpdate()`
  - `Model.replaceOne()`
  - `Model.updateMany()`
  - `Model.updateOne()`
- Very often, there are more than one method to achieve a database action
- *See: https://mongoosejs.com/docs/queries.html*

DELETE (CRUD)

```
// Delete all documents satisfying the condition
User.remove({ email: 'john@example.com' }, callback);

// One can also build a query and then call remove()
// on the query object to remove the matching documents
User.find({ email: 'john@example.com' })
  .remove(callback);
```

DOCUMENTS IN ANOTHER COLLECTION

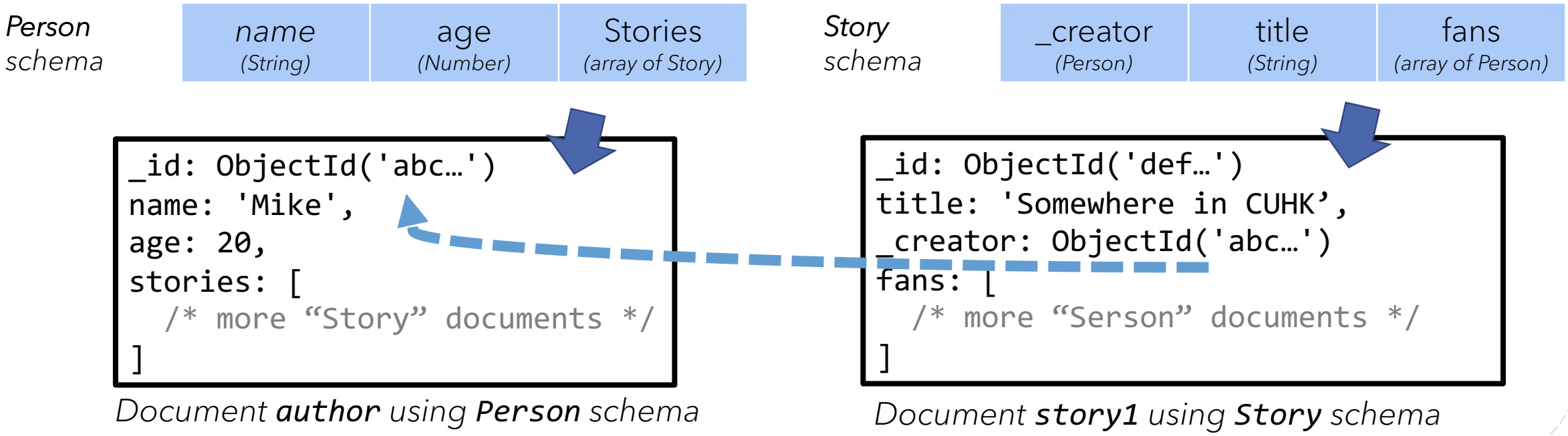
```
// Schemas of documents that refer to other documents (by _id)
// Source: http://mongoosejs.com/docs/populate.html
const mongoose = require('mongoose');
/* connecting to db, initializing, etc. ... */
const Schema = mongoose.Schema;

const personSchema = Schema({
  name    : String,
  age     : Number,
  stories : [{ type: Schema.Types.ObjectId, ref: 'Story' }] // array of ObjectId
});
const storySchema = Schema({
  _creator : { type: Schema.Types.ObjectId, ref: 'Person' },
  title    : String,
  fans     : [{ type: Schema.Types.ObjectId, ref: 'Person' }]
});
const Story  = mongoose.model('Story', storySchema);
const Person = mongoose.model('Person', personSchema);
```

DOCUMENTS IN ANOTHER COLLECTION

```
// ... Cont'd from the previous slide
// Saving refs
const author = new Person(
  { name: 'Mike', age: 20 });

author.save(function (err) {
  if (err) return handleError(err);

  const story1 = new Story({
    title: "In CUHK...",
    _creator: author._id
    // assign the _id from the person
  });
  story1.save(function (err) {
    if (err) return handleError(err);
    // that's it!
  });
});
```

```
// Populate (via refs)

Story
.findOne({ title: 'In CUHK...' })
.populate('_creator')
.exec(function (err, story) {
  if (err) return handleError(err);

  // In the result, the value of _creator is
  // replaced by the document it refers to
  // with the help of populate()
  console.log('The creator is %s',
  story._creator.name);
  // prints "The creator is Mike"
});
```

DOCUMENTS IN ANOTHER COLLECTION



- The `.populate()` method allows retrieving referenced documents
  - e.g., `story1.fans[2].name`, `author.stories[1]._creator`
  - *See: https://mongoosejs.com/docs/populate.html*

OTHER CLOUD DATABASES

- Only brief ideas of MongoDB is given here as a taster of DBMS
- Many cloud solutions also provide their own database services to allow easy connection and collaboration within cloud apps, *e.g.*,
  - Amazon DynamoDB
  - Google Cloud Firestore
  - ➢ *SQL vs. NoSQL*
  - ➢ Not just limited for Node.js development
- Database efficiency could be heavily affecting app performances!
- See: *Cloud-based DBMS's popularity grows at high rates* https://db-engines.com/en/blog_post/82

READ FURTHER…

MongoDB Manual

https://docs.mongodb.com/manual/

Mongo Shell Quick Reference

https://docs.mongodb.com/manual/reference/mongo-shell/

Mongoose Quick Start

http://mongoosejs.com/docs

Mongoose API

https://mongoosejs.com/docs/api.html