香港中文大學
The Chinese University of Hong Kong

CSCI2720 - Building Web Applications

Lecture 17: MongoDB

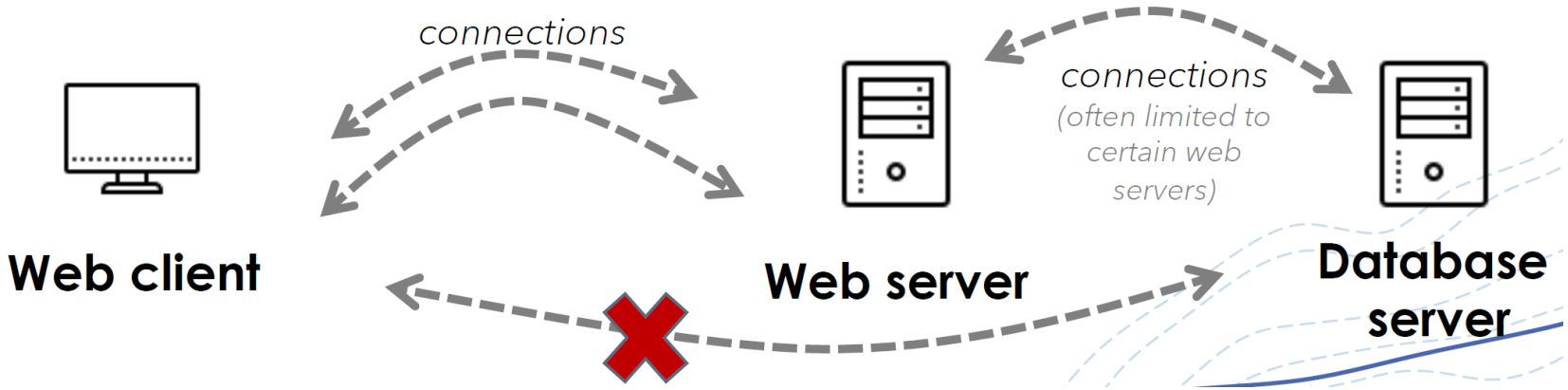
Dr Colin Tsang

Outline

- Database systems
- MongoDB
- Mongoose
- 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

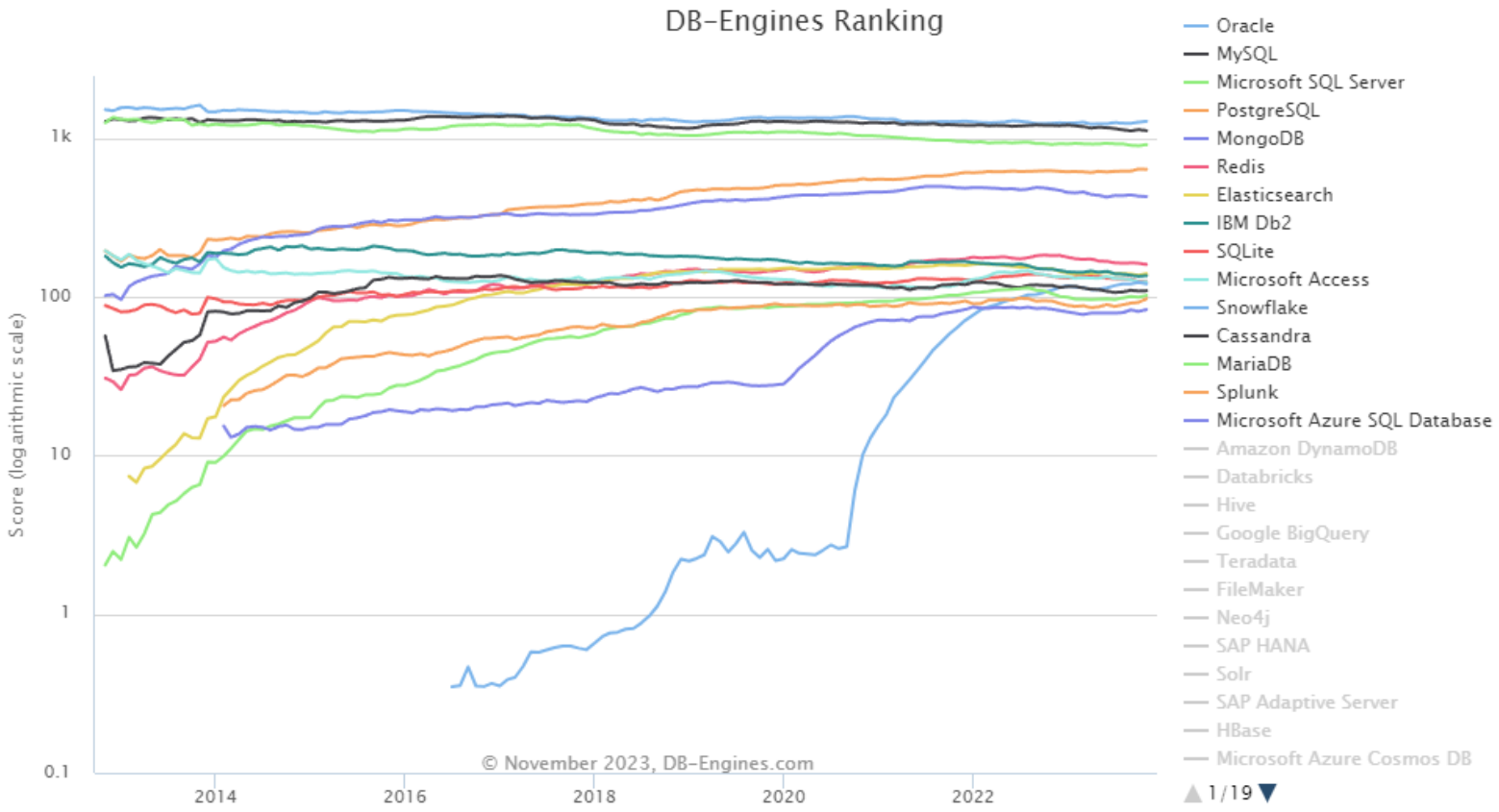


Database systems

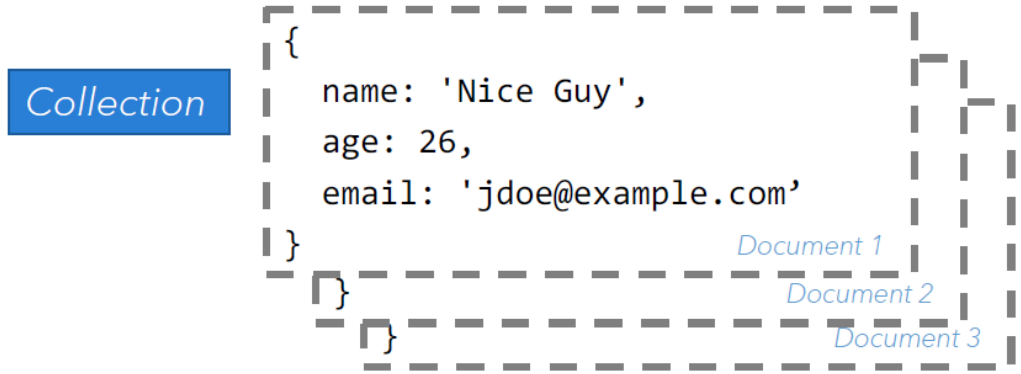
- **MySQL**: an open-source relational database management system (RDBMS)
  - Since 1995
  - Support relations
    - i.e., with table, rows/columns
- **MongoDB**: an open-source NoSQL database management system (DBMS)
  - Since 2009
  - NoSQL – lack of support of the concept of *relations*
    - i.e., no table, rows/columns
  - Supporting dynamic schemas

Database systems

- Trend popularity: https://db-engines.com/en/ranking_trend



MongoDB



- 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 database
- Why not MySQL in this course?
  - See: https://www.simform.com/blog/mongodb-vs-mysql-databases/

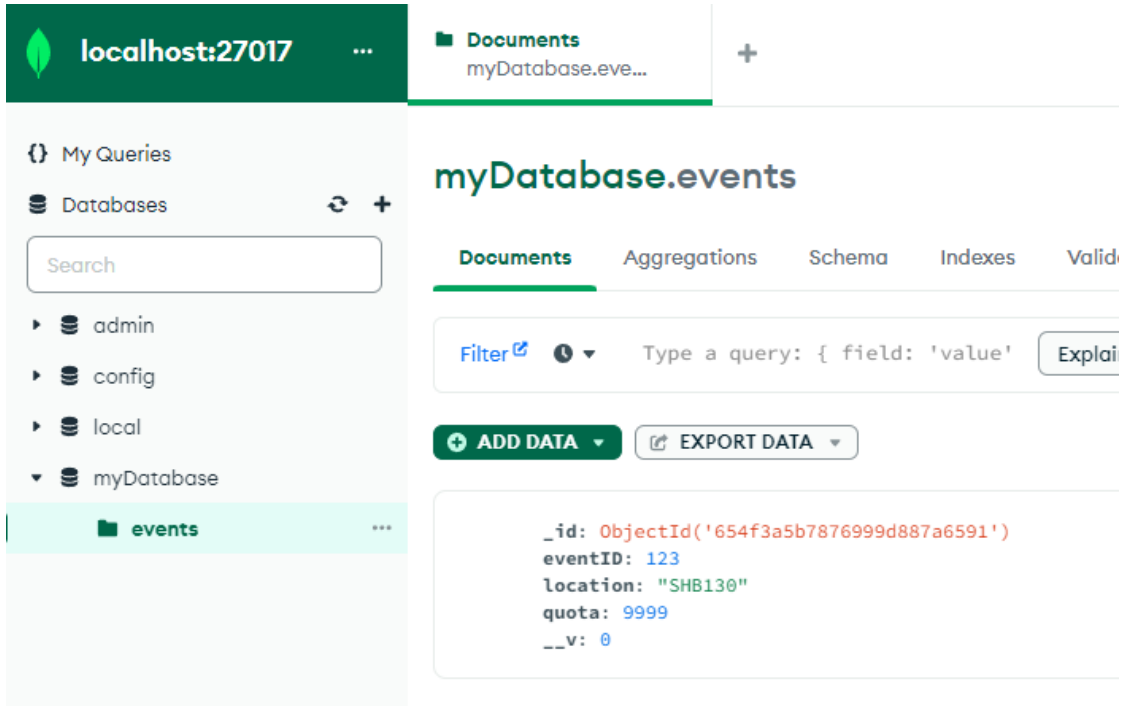
Document

- 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 filed can be any of the BSON types (*string*, *double*, *integer*, etc), including other *documents*, *arrays*, and *arrays of documents*.

- See: https://www.mongodb.com/docs/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 version VS enterprise version
  - *MongoDB Atlas*: a cloud service where both free-tier and paying option are available
  - See our **Lab08** materials for more details.
- Directly access by
  - *Mongo Shell*: command line interface
  - *Mongo Compass*: graphical user interface, recommended in **Lab08**
  - *Web interface*: for cloud only

MongoDB with Node.js

- In a web app, the backend (e.g., our Node.js) is responsible to access the database server.
  - The client is usually disallowed to interact with the database.
- Two ways to access MongoDB from Node.js
  - Use MongoDB directly with *Mongo Shell commands* in Node.js, using the official MongoDB Node.js driver
    - Difficult to use…….
    - See: https://www.mongodb.com/docs/drivers/node/current/
  - Use *Mongoose* module
    - See **Lab08**

What is Mongoose

- An **Object Data Modelling (ODM)** library on top of MongoDB
- Support *schemas* to describe documents
- Automatic generation of *data model* from *schema*
- Simplify interaction with MongoDB from Node.js

Mongoose: Connecting to MongoDB

```
// mongoose needs to be installed with npm
const mongoose = require('mongoose');
// put your database link here
mongoose.connect('mongodb://127.0.0.1:27017/myDatabase');

// 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...");

  // your code here if the connection is open

})
```

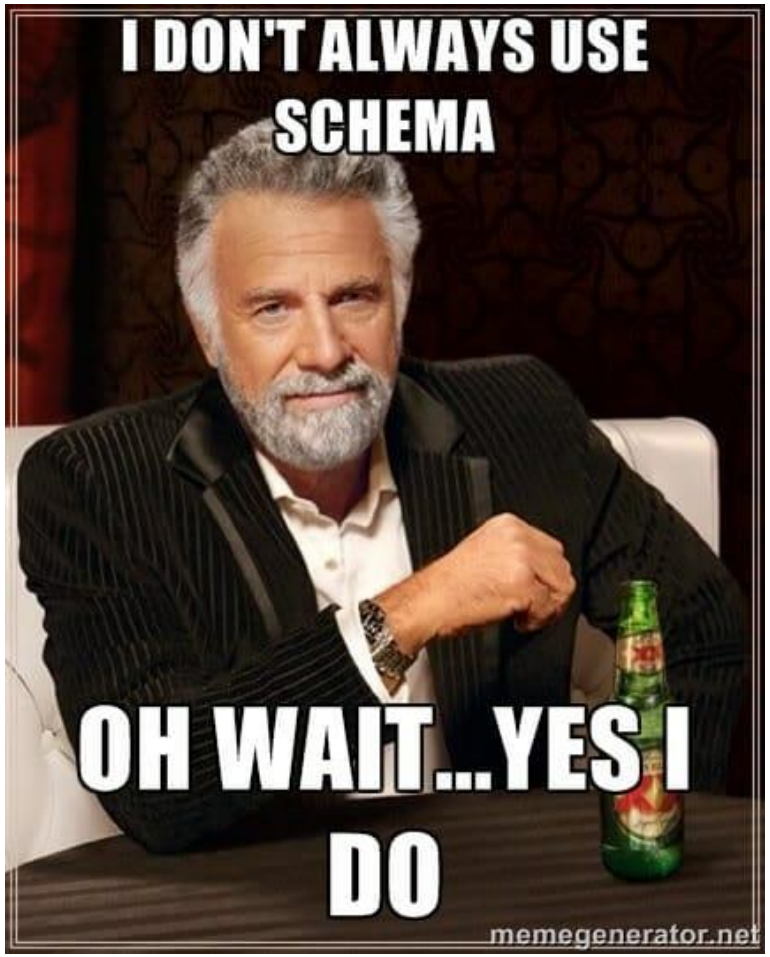
Mongoose: Connecting to MongoDB

- **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
dbUri = 'mongodb://127.0.0.1:27017/myDatabase'
const options = {
    user: 'myDatabaseUserName',
    pass: 'myDatabasePassword',
    }
mongoose.connect(dbUri, options);
```

Schema

- Schema describes a document in a collection
  - Type check and automatic type conversion
    - **type** affects how property values are casted
  - Check if a value is unique
  - Check if a required value is omitted
    - Properties like **required** and **unique** are enforced



Defining Schema

```
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 }], // an array field, can store multiple comments here
    date: { type: Date, default: Date.now }, // assign type and default value
    hidden: Boolean,
    meta: {
    votes: Number,
    favs: Number
    } // nested object field, only a single set of meta here
});
```

Schema types

- **SchemaType**:
  - **String**
  - **Number**
  - **Date**
  - **Buffer**
  - **Boolean**
  - **Mixed**
  - **ObjectID**
  - **Array**
- See: https://mongoosejs.com/docs/schematypes.html

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's name
  - E.g., the collection name should be users for the following model

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

CRUD in Mongoose

- CRUD: Create, Read, Update, and Delete
- You will come across these operations quite often in databases on the web
- CRUD operations are supported by Mongoose query functions
  - All queries are executed asynchronously
- **Important:** do you still remember *callbacks* and *promise*?
  - Revisit Lecture 07 if you forgot about it
  - In Mongoose *version 5.0* (released in May 2023), it dropped support for *callbacks*…..
  - The code in most online materials *cannot be run in the current version*
    - Including: our lecture notes last year, ChatGPT, many online tutorials, etc.
  - Mongoose only support *promise* now

CRUD – Create (with callbacks)

model

Server

```
const newUser = new User({
    name: 'John',
    email: 'john@example.com',
    password: '123'
});
```

Database

```
newUser.save(function(err, savedUser) {
  if (err) {
    console.error("failed to create new user", err);
  } else {
      console.log("sucessfully created a new user", savedUser);

  }
});
```

CRUD – Create

- We have our **UserSchema** and a model named **User** from the previous slides.
- We can create a document with *promise*:

Server

```
//Creating a new user
let newUser = new User({
name: "John",
email: "john@example.com",
password: "123456",
});
```

Send to database using Promise

```
//Saving this new user to database
newUser
.save()
.then(() => {
    console.log("sucessfully created a new user");
})
.catch((error) => {
    console.log("failed to create new user");
});
```

CRUD - Read

```
// Read all data
User.find({})
.then((data) => {
    console.log(data);
})
.catch((err) => {
    console.log("failed to read");
});

// Search for john
User.find({ name: {$eq: "John"} })
.then((data) => console.log("document with the name John:", data))
.catch((error) => console.log(error));
```

Query operators

- Standard MongoDB operators are supported in Mongoose:
  - **$eq** equal
  - **$gt** greater than
  - **$gte** greater than or equal to
  - **$in** in an arrray
  - **$lt** less than
  - **$lte** less than or equal to
  - **$ne** not equal
  - **$nin** not in an array

- More on: https://www.mongodb.com/docs/manual/reference/operator/query/

CRUD – Update

```
// update the password if name = john
User.findOneAndUpdate(
    { name: {$eq: "John"} },
    { password:"John123"},
    { new: true},
)
.then((data) => {console.log('the updated document is:', data)})
.catch((error) => console.log(error));
```

CRUD - Delete

```
// delete the document if name = john
Event.findOneAndDelete(
    { name: {$eq: "John"} },
)
.then((data) => {console.log('the deleted data is:', data)})
.catch((error) => console.log(error));
```

CRUD

- Many other built-in methods in Mongoose Model to perform CRUD:
  - https://mongoosejs.com/docs/api/model.html
    - **Model.createCollection()**
    - **Model.findByID()**
    - **Model.findByIDAndUpdate()**
    - **Model.findOneAndReplace()**
    - **Model.replaceOne()**
    - **Model.updateMany()**
    - **Model.deleteOne()**
    - **Model.deleteMany()**

Documents in another collection

Person schema	*name* (String)	age (Number)	Stories (array of Story)

Story schema	_creator (Person)	title (String)	fans (array of Person)

```
const personSchema = Schema({
    name : String,
    age : Number,
    // An array of ObjectId references that refer to documents in the Story collection
    stories : [{ type: Schema.Types.ObjectId, ref: 'Story' }]
});

const storySchema = Schema({
    // A reference to a document in the Person collection using its ObjectId
    _creator : { type: Schema.Types.ObjectId, ref: 'Person' },
    title : String,
    // An array of ObjectId references that refer to documents in the Person collection
    fans : [{ type: Schema.Types.ObjectId, ref: 'Person' }]
});

const Story = mongoose.model('Story', storySchema);
const Person = mongoose.model('Person', personSchema);
```

- More on: https://mongoosejs.com/docs/populate.html

Other databases

- Only brief ideas of MongoDB is given here as a taster of DBMS
- Many cloud solutions (provided by big companies) also have their own database services to allow easy connection and collaboration with cloud apps
  - Amazon DynamoDB
  - Google Cloud Firestore
- Database efficiency could be heavily affecting app performances

Further reading

- MongoDB manual:
  - https://www.mongodb.com/docs/manual/
- Mongoose Quick Start:
  - https://mongoosejs.com/docs/
- Mongoose documentations:
  - https://mongoosejs.com/docs/api/mongoose.html