



# MongoDB, Mongoose and Cloud Storage

Frank Walsh, Diarmuid O'Connor

# Agenda

- Cloud Databases
- MongoDB
- Mongoose
- Mongo in the cloud



# Databases in Enterprise Apps

---

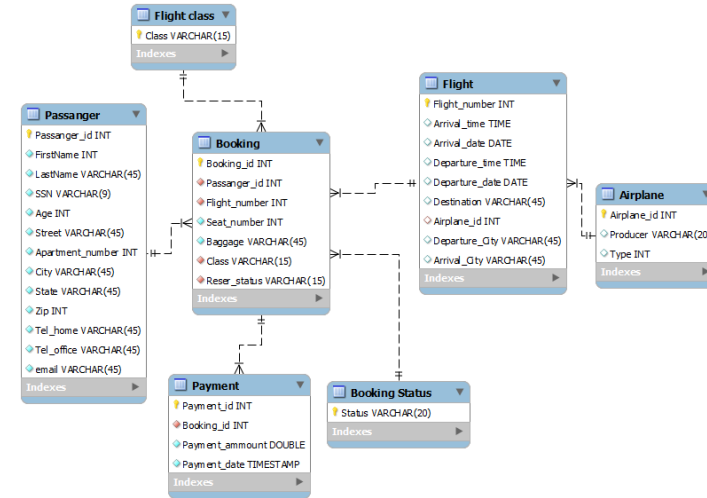
- Most data driven enterprise applications need a database
  - Persistence: storage of data
  - Concurrency: many applications sharing the data at once.
  - Integration: multiple systems using the same DB
- Enterprise Application DBs require backups, fail over, maintenance, capacity provisioning.
  - Traditionally handled by a Database Administrator (the DBA).



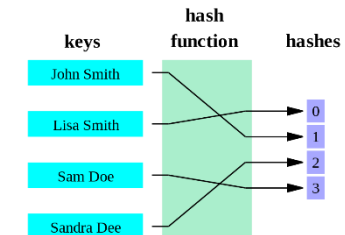
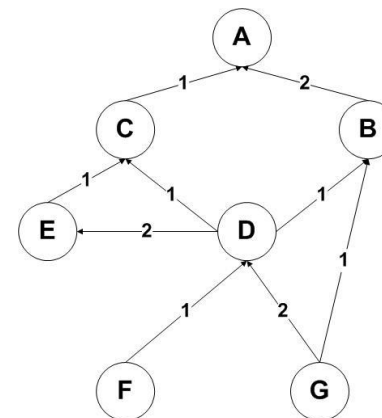
# Structured & Unstructured Data

- Relational Databases:
  - Organise data into structured tables and rows
  - Relations have to be simple, they cannot contain any structure such as a nested record or a list
- In memory data structures
  - Much more varied structure
  - Lists, Queues, Stacks, Graphs, Hashing

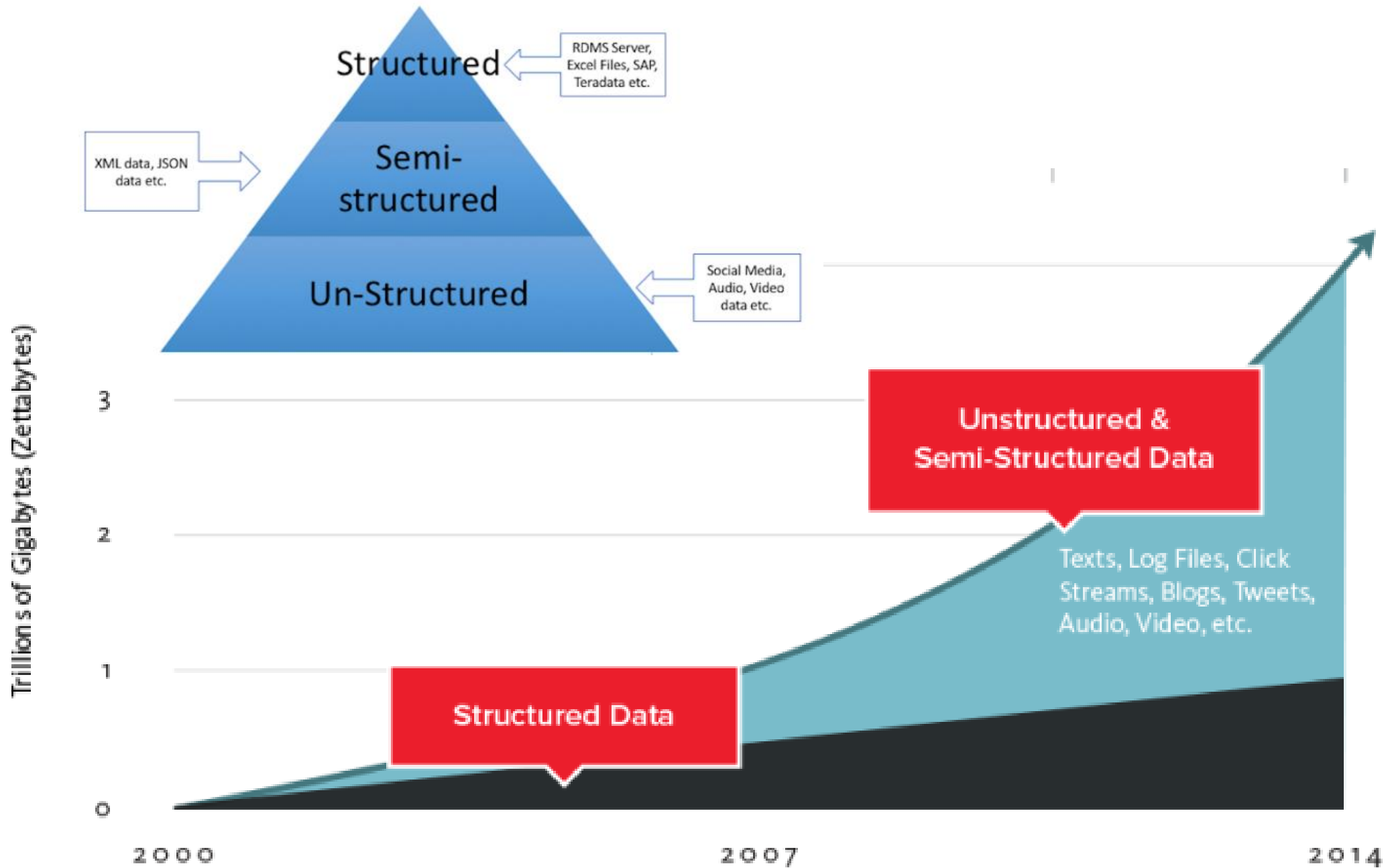
## Relational Database



## In Memory Data Structures



data format



# Databases in the Cloud

- For some apps, a traditional relational database may not be the best fit
  - Organisations are capturing more data and processing it quicker – can be expensive/difficult on traditional DB
  - Traditionally, relational database is designed to run on a single machine in predictable environment
  - May be economic to run large data and computing loads on clusters.
  - Hard to estimate scaling requirements, particularly if it's a web app?
  - Are you going to do Data mining?
- One approach is to use the **Cloud** for your DB
  - Designed for scale
  - Can be outsourced so you don't have to deal with infrastructure requirements.



# Cloud DB Advantages

- Removes Management costs
- Inherently scalable
- No need to define schemas(if NoSQL) etc.
- Lots of Cloud DB offerings out there
  - SQL based
  - NoSQL based
- If organisation policy/standards do not allow outsourcing:
  - Can host yourself, most NoSQL DBs are free.

# Cloud Database Practices

- Drop Consistency
  - this makes distributed systems much easier to build
- Drop SQL and the relational model
  - simpler structures are easier to distribute:
    - key/value pairs
    - **structured documents**
    - **pseudo-tables**
    - tend to be schema-free, accepting data as-is
- Offer HTTP interfaces using XML or **JSON**
  - Web APIs!!!



# Designing Distributed Data

- App data is not homogeneous
  - some kinds of data will be much larger
- consider using different databases for different requirements.
- user details,billing - needs consistency
  - require traditional database
- user data,content - needs partition tolerance
  - replicate to keep safe
- analytics,sessions - needs availability
  - "eventually consistent" is good enough




[This Photo](#) by Unknown Author is licensed under [CC BY-NC-ND](#)

# MONGODB

# Introduction

- Document-oriented database
- A record in MongoDB is a document, which is a data structure composed of field and value pairs.
- MongoDB documents are similar to JSON objects
- Field Values can be other documents, arrays, arrays of other documents.
  - Reduces need for “Joins”
- Community support - popular choice

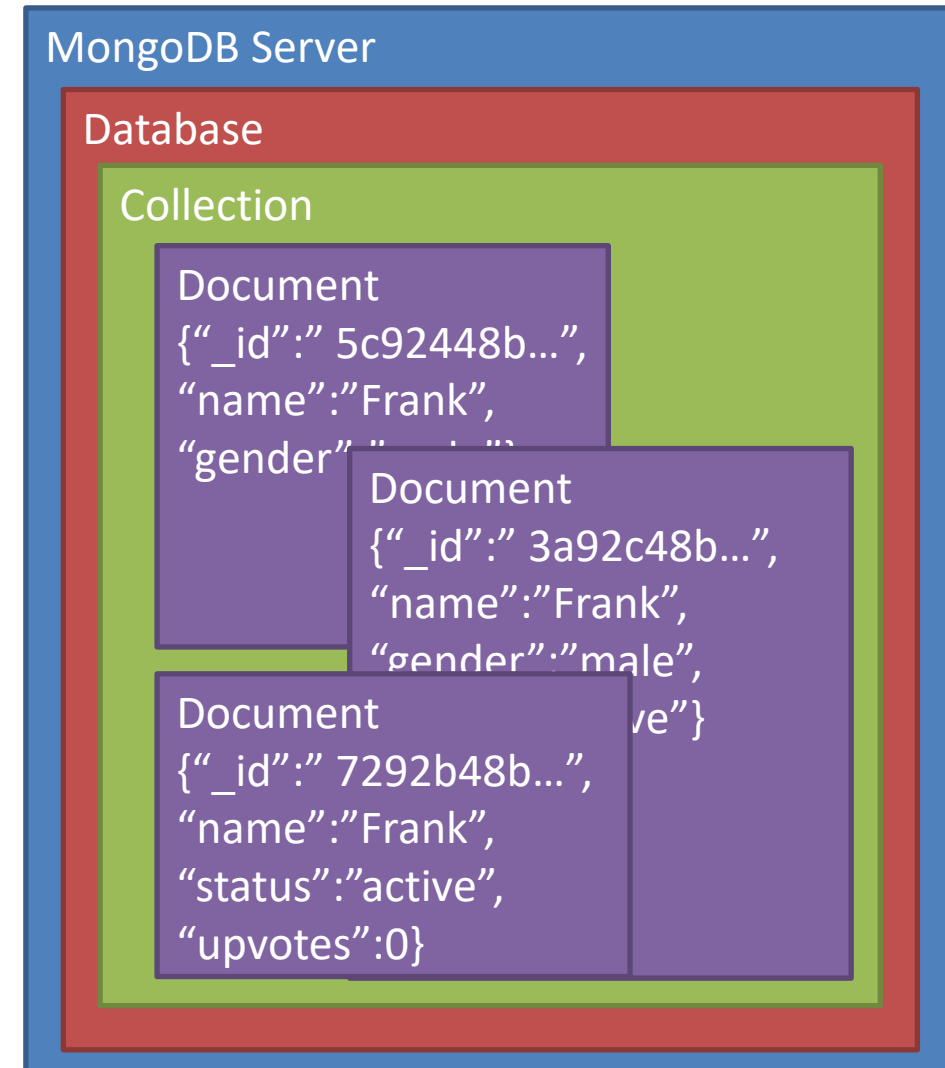
```
{  
  name: "sue",  
  age: 26,  
  status: "A",  
  groups: [ "news", "sports" ]  
}
```



← field: value  
← field: value  
← field: value  
← field: value

# Mongo Terminology

- Each **database** contains a set of "Collections"
- Collections contain a set of JSON documents
  - there is no schema (in the DB...)
- The documents can all be different
  - means you have rapid development
  - adding a property is easy - just starting using in your code
- Makes deployment easier and faster
  - roll-back and roll-forward are safe - unused properties are just ignored
- Collections can be indexed and queries
- Operations on individual documents are atomic




# Mongo Documents

- MongoDB stores data records as BSON documents.
  - BSON is a binary representation of JSON documents.
- Each document stored in a collection requires a unique `_id` field and is reserved for use as a primary key.
- If an inserted document omits the `_id` field, the MongoDB driver automatically generates an ObjectId for the `_id` field.
  - ObjectId values consist of 12 bytes.

```
_id: ObjectId("5c92448b7fbccf28a0c501aa")  
name: "Contact 4"  
address: "49 Upper Street"  
phone_number: "934-4290"
```

# Getting Started (locally)

- Install Mongo community edition for your OS:

[Install MongoDB](#) > Install MongoDB Community Edition 

## Install MongoDB Community Edition

These documents provide instructions to install MongoDB Community Edition.

[Install on Linux](#)  
Install MongoDB Community Edition and required dependencies on Linux.

[Install on macOS](#)  
Install MongoDB Community Edition on macOS systems from MongoDB archives.

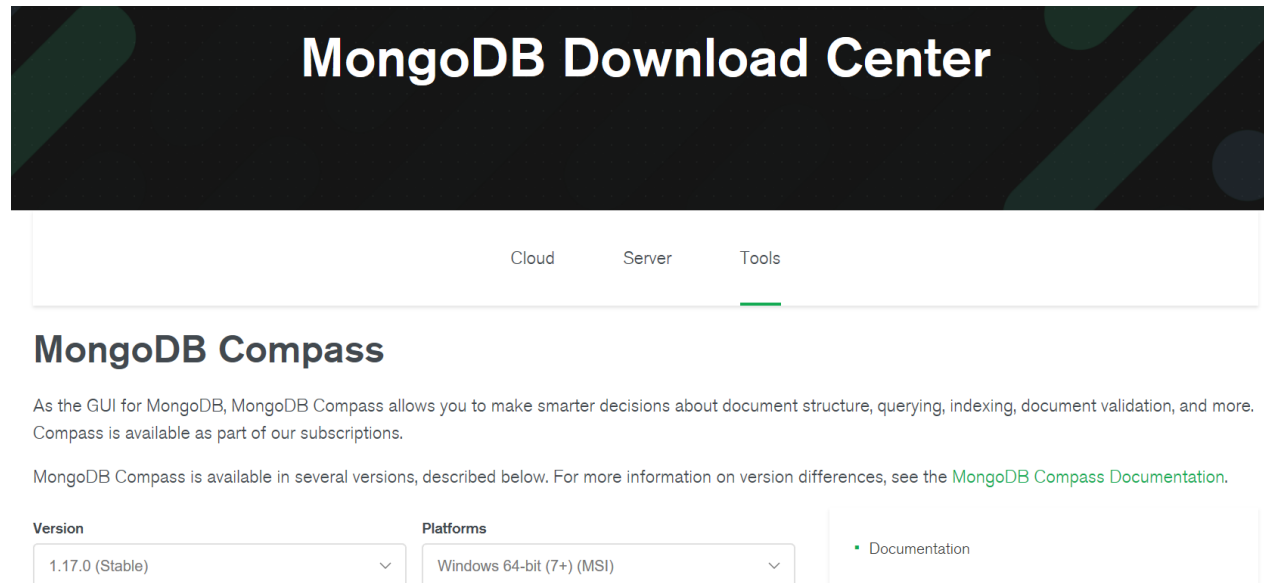
[Install on Windows](#)  
Install MongoDB Community Edition on Windows systems and optionally start MongoDB as a Windows service.

- Specify a directory for your db files and start Mongodb server.

```
mkdir db  
mongod -dbpath db
```

# Getting Started (locally)

- Install Mongo Compass, Graphical User Interface for managing MongoDB.
  - For windows, comes as part of mongodb install
  - Other platforms can get it [here](#):





# MONGOOSE

Mongo with Node.js



## Mongoose Overview

- Mongoose is a object-document model module in Node.js for MongoDB
  - Wraps the functionality of the native MongoDB driver
  - Exposes models to control the records in a doc
  - Supports validation on save
  - Extends the native queries

# mongoose


elegant **mongodb** object modeling for **node.js**

[read the docs](#)

[discover plugins](#)

 Star 18,205

Version 5.4.19

 Fork 2,570

Let's face it, **writing MongoDB validation, casting and business logic boilerplate is a drag**. That's why we wrote Mongoose.

# Mongoose first?

---

- Shortcut to understanding the basics
- Similar to Object Relational Mapping libraries like JPA/Hibernate
- Easier concept if coming from relational DB background.



# Installing & Using Mongoose

1. Run the following from the CMD/Terminal

```
npm install --save mongoose
```

2. Import the module

```
import mongoose from 'mongoose';
```

3. Connect to the database

```
mongoose.connect(process.env.mongoose);
```

# Mongoose Schemas and Models

- Mongoose supports models
  - i.e. fixed types of documents
  - Needs a mongoose.Schema
  - Each of the properties must have a type
    - Number, String, Boolean, array, object

```
import mongoose from 'mongoose';

const Schema = mongoose.Schema;

const UserSchema = new Schema({
  username: { type: String, unique: true, required: true },
  password: { type: String, required: true }
});

export default mongoose.model('User', UserSchema);
```

# Mongoose Schemas – Arrays & sub-documents

```
1 |const mongoose = require('mongoose')
2 |Schema = mongoose.Schema;
3
4 |const CommentSchema = new Schema({
5 |  body: {type: String, required:true},
6 |  author: {type: String, required:true},
7 |  upvotes:Number
8 |});
9
10|const PostSchema = new Schema({
11|  title: {type: String, required:true},
12|  link:  {type: String, optional:true},
13|  username: {type: String, required:true},
14|  comments: [CommentSchema],
15|  upvotes: { type: Number, min: 0, max: 100 }
16|});
17
18|export default mongoose.model('posts', PostSchema);
```

Comments property is  
an Array of  
CommentSchemas

# Mongoose Schemas - Arrays

```
import mongoose from 'mongoose';

const Schema = mongoose.Schema;

const MovieReviewSchema = {
  userName : { type: String},
  review : {type: String}
}

const MovieSchema = new Schema({
  adult: { type: Boolean},
  id: { type: Number, required: true, unique: true },
  poster_path: { type: String},
  overview: { type: String},
  release_date: { type: String},
  reviews : [ MovieReviewSchema],
  original_title: { type: String},
  genre_ids: [{type: Number}],
```

Review property is an  
Array of  
MovieReviewSchema

# Mongoose Schema – Built-in Validation

---

constraints on properties :

```
import mongoose from 'mongoose';
const Schema = mongoose.Schema;

const ContactSchema = new Schema({
  name: {type: String, required:[true, 'Name is a required property']},
  address: String,
  age: {
    type: Number,
    min: 0,
    max: 120,required: true
  },
  email: String,
  updated: {
    type: Date,
    default: Date.now,
  },
});

export default mongoose.model('Contact', ContactSchema);
```

```
import mongoose from 'mongoose';

const Schema = mongoose.Schema;

const UserSchema = new Schema({
  username: { type: String, unique: true, required: true},
  password: {type: String, required: true }
});

export default mongoose.model('User', UserSchema);
```

# Mongoose Custom Validation

- Developers can define custom validation on their properties (e.g. validate email field is correct format)

```
ContactSchema.path('email').validate((email) => {  
  var emailRegex = /^[^\w-\u005C.]+@([^\w-\u005C.]+[^\w-]{2,4})?$/;  
  return emailRegex.test(email);  
}, 'A valid e-mail address is required');
```

Using Regular Expression (regex) to test for a valid email. If you've not come across them before check out [https://www.w3schools.com/jsref/jsref\\_obj\\_regexp.asp](https://www.w3schools.com/jsref/jsref_obj_regexp.asp)



# Mongoose Custom Validation

- Developers can define custom validation on their properties (e.g. validate length of username when trying to save)

```
const UserSchema = new Schema({
  username: { type: String, unique: true, required: true },
  password: { type: String, required: true },
  favourites: [{ type: mongoose.Schema.Types.ObjectId,
    ref: 'Movie' }]
});

UserSchema.path('username').validate(v => (v.length < 5), false, true)
```

# Data Manipulation Mongoose

- Mongoose supports all the CRUD operations:
  - Create → `Model.create()`
  - Read → `Model.find()`
  - Update → `Model.update(condition, props, cb)`
  - Remove → `Model.remove()`
- Can operate with "*error first*" callbacks or promises.

# Create with Mongoose

```
const router = express.Router();

router.post('/', (req, res) => {
  | Movie.create(req.body).then(movie => res.status(201).json(movie))
  |});
```

# Update with Mongoose

```
// Update a user
router.put('/:id', (req, res) => {
  if (req.body._id) delete req.body._id;
  User.update({
    _id: req.params.id,
  }, req.body, {
    upsert: false,
  })
  .then(user => res.json(200, user));
});
```

# Mongoose Queries

- Mongoose provides a more expressive version of the native MongoDB
  - Instead of:  
`{ $or: [{ conditionOne: true }, { conditionTwo: true } ] }`
  - Do: `.where({ conditionOne: true }).or({ conditionTwo: true })`

# Mongoose Queries

- Mongoose supports many queries:
  - For equality/non-equality
  - Selection of some properties
  - Sorting
  - Limit & skip
- All queries are executed over the object returned by Model.find\*()
  - Model.findOne() returns a single document, the first match
  - Model.find() returns all
  - Model.findById() queries on the \_id field.

```
router.post('/:userName/favourites', (req, res, next) => {
  const newFavourite = req.body;
  const userName = req.params.userName;
  if (newFavourite && newFavourite.id) {
    Movie.findOneAndUpdate({id: newFavourite.id}, newFavourite, {new: true, upsert: true}).then(movie => {
      User.findById(userName).then(
        (user) => {
          (user.favourites.indexOf(movie._id) > -1) ? user.favourites.push(movie._id.toString());
          user.save().then(user => res.status(201).send(user))
        }
      );
    }).catch((err) => console.log(err));
  } else {
    res.status(401).send("unable")
  }
}
```

# Mongoose Queries

- Can build complex queries and execute them later

```
1  const query = ContactModel.where('age').gt(17).lt(66)
2    .where('county').in(['Waterford', 'Wexford', 'Kilkenny']);
3
4  query.exec((err, contacts) => {...})
5
6
```

- The above finds all contacts where age >17 and <66 and living in either Waterford, Kilkenny or Wexford

# Mongoose Sub-Docs

- Ex: Hacker News – Adding a comment to a post.

```
// add comment
router.post('/:id/comments', asyncHandler( async (req, res) => {
  const id = req.params.id;
  const comment = req.body;
  const post = await Post.findById(id);
  post.comments.push(comment);
  await post.save();
  return res.status(201).send({post});
}));
```



# Mongoose Sub-Docs

- Ex: Movies – Adding a review to a favourite movie.

```
router.post('/:id/reviews', (req, res) => {  
  const id = parseInt(req.params.id);  
  Movie.findByIdByMovieDBId(id).then(movie => {  
    movie.reviews.push(req.body);  
    movie.save().then(res.status(200).send(movie.reviews)));  
  });  
});
```

# **SCHEMA METHODS**

# Example: Using Schema Methods for Simple Authentication

- Restrict access to Posts API (require authentication):
  - Create users schema with methods for
    - Finding users
    - Checking password
  - Use **express-session** middleware to create and manage user session (using cookies)
  - Create an authentication route to set up “session”
  - Create your own authentication middleware and place it on /api/movies route

# Aside: Sessions

- Requests to Express apps are stand-alone by default
  - no request can be linked to another.
  - By default, no way to know if this request comes from client that already performed a request previously.
- Sessions are a mechanism that makes it possible to “know” who sent the request and to associate requests.
- Using Sessions, every user of your API is assigned a unique session:
  - Allows you to store state.
- The `express-session` module is middleware that provides sessions for Express apps.

## express-session

1.15.6 • Public • Published a year ago

Readme

9 Depend

## express-session

npm

v1.15.6

downloads

3M/m

build

passing

coverage

100%

## Installation

a Node.js module available through the npm registry

command:

```
npm install express-session
```

# User Schema with Static & Instance Methods

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

UserSchema.statics.findByUserName = function(username) {
  return this.findOne({ username: username });
};

UserSchema.methods.comparePassword = function(candidatePassword) {
  const isMatch = this.password === candidatePassword;
  if (!isMatch) {
    throw new Error('Password mismatch');
  }
  return this;
};

export default mongoose.model('User', UserSchema);
```

Static Method: belongs to schema. Independent of any document instance

Instance Method: belongs to a specific document instance.

# express-session middleware

- Session middleware that stores session data on server-side
  - Puts a unique ID on client

```
npm install --save express-session
```

- Add to Express App middleware stack:

```
//session middleware
app.use(session({
  secret: 'ilikecake',
  resave: true,
  saveUninitialized: true
}));
```

# Create User Route to authenticate

- Use **/api/user** to authenticate, passing username and password in HTTP body

/api/users/index.js

```
// authenticate a user, using async handler
router.post('/', asyncHandler(async (req, res) => {
  if (!req.body.username || !req.body.password) {
    res.status(401).send('authentication failed');
  } else {
    const user = await User.findByUserName(req.body.username);
    if (user.comparePassword(req.body.password)) {
      req.session.user = req.body.username;
      req.session.authenticated = true;
      res.status(200).end("authentication success!");
    } else {
      res.status(401).end('authentication failed');
    }
  }
});
```

Using static method to find User document

Using instance method to check password

/index.js

```
app.use('/api/users', usersRouter);
```

# Authentication Middleware

authenticate.js

```
import User from '../api/users/userModel';  
// Authentication and Authorization Middleware  
export default async (req, res, next) => {  
  if (req.session) {  
    let user = await User.findByUserName(req.session.user);  
    if (!user)  
      return res.status(401).end('unauthorised');  
    next();  
  } else {  
    return res.status(401).end('unauthorised');  
  }  
};
```

Checks for user ID in session object.  
If exists, called next middleware function, otherwise end req/res cycle with 401

index.js

```
import authenticate from './authenticate';  
  
app.use('/api/posts', authenticate, postsRouter);
```

Authentication middleware applied on /api/posts route.



# Object Referencing

Using Object ID to  
reference Movie  
document

```
✓ const UserSchema = new Schema({  
  username: { type: String, unique: true, required: true },  
  password: { type: String, required: true },  
  favourites: [{ type: mongoose.Schema.Types.ObjectId,  
                ref: 'Movie' }]  
});
```

# Query Population using Refs

<https://github.com/fxwalsh/ewd-examples-2020.git>

- Allows you to automatically replace the specified paths in the document with document(s) from other collection(s).

```
async function refTest() {
  const user1 = new User({
    username: "user99",
    password: "pass1"
  });
  await user1.save();

  const post1 = new Post({
    title: "A Post",
    user: user1._id
  });

  await post1.save()
  Post.find({})
    .populate('user')
    .exec(function (error, posts) {
      console.log(JSON.stringify(posts, null, "\t"))
    });
}

refTest();
```

output

```
{
  "upvotes": 0,
  "_id": "5c93899a1f4eaa3cf4e4fbc8",
  "title": "A Post",
  "user": {
    "_id": "5c9389981f4eaa3cf4e4fbc7",
    "username": "user99",
    "password": "pass1",
    "__v": 0
  },
  "comments": [],
  "__v": 0
}
```

# **MONGODB AS A SERVICE**

# MongoDB as a Service

- Best practice for initial development is to host MongoDB process on your development machine
- In production environments, Mongo will be hosted:
  - on it's own instance or
  - provisioned as a service



This Photo by Unknown Author is licensed under [CC BY-SA](#)