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
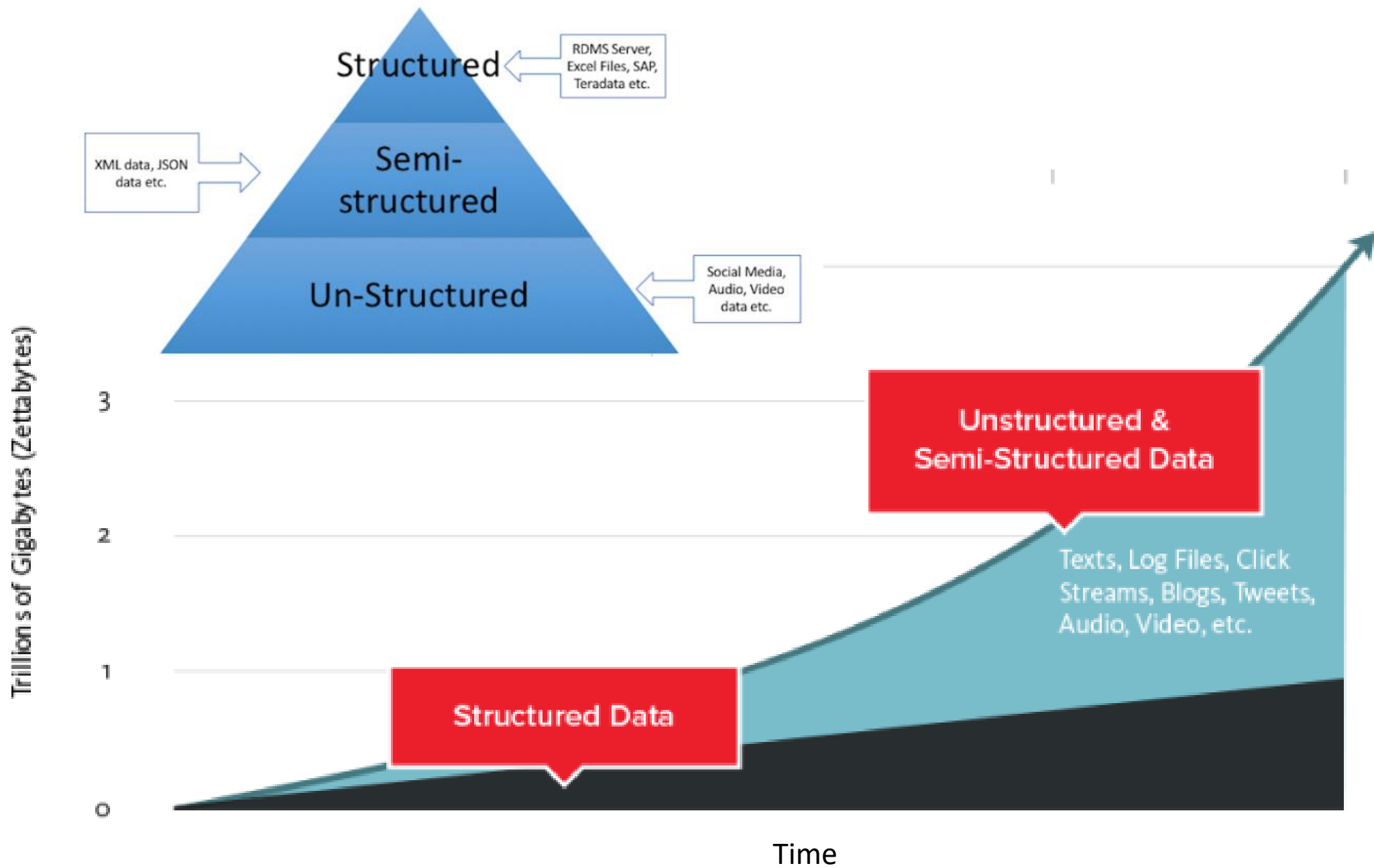 - Lists, Queues, Stacks, Graphs, Hashing

Relational Database
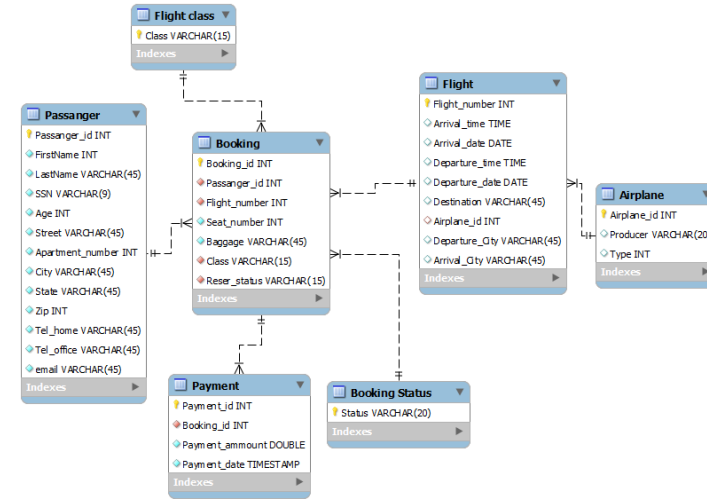
In Memory Data Structures
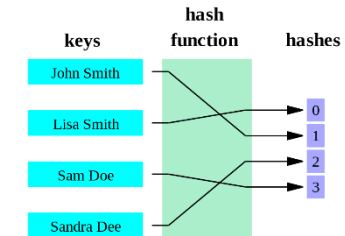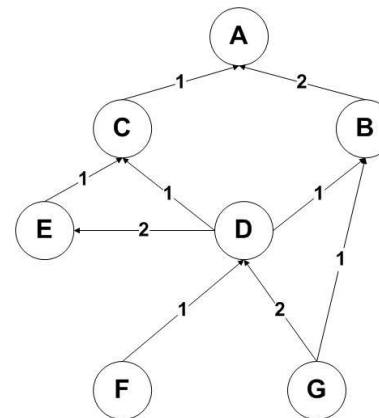
Databases in the Cloud

- 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 predicable 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 you DB
  - Designed for scale
  - Can be outsourced so you don't have to deal with infrastructure requirements.
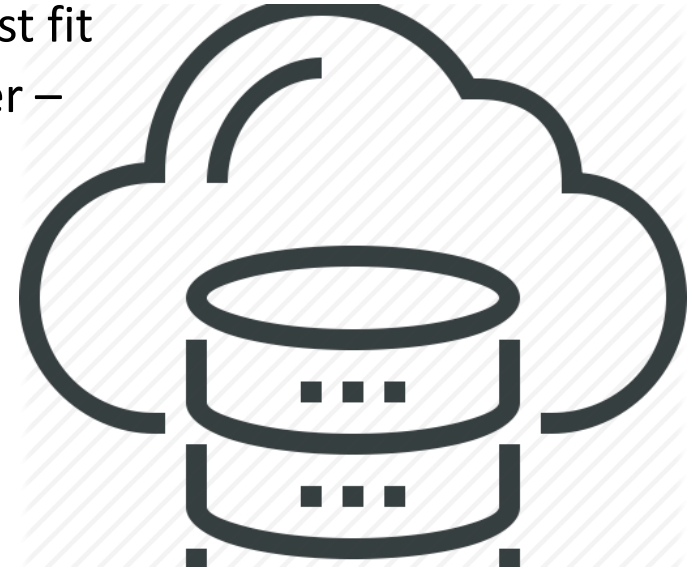
Cloud DB Advantages

- 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

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",              ←——— field: value
  age: 26,                  ←——— field: value
  status: "A",              ←——— field: value
  groups: [ "news", "sports" ]  ←——— 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...)
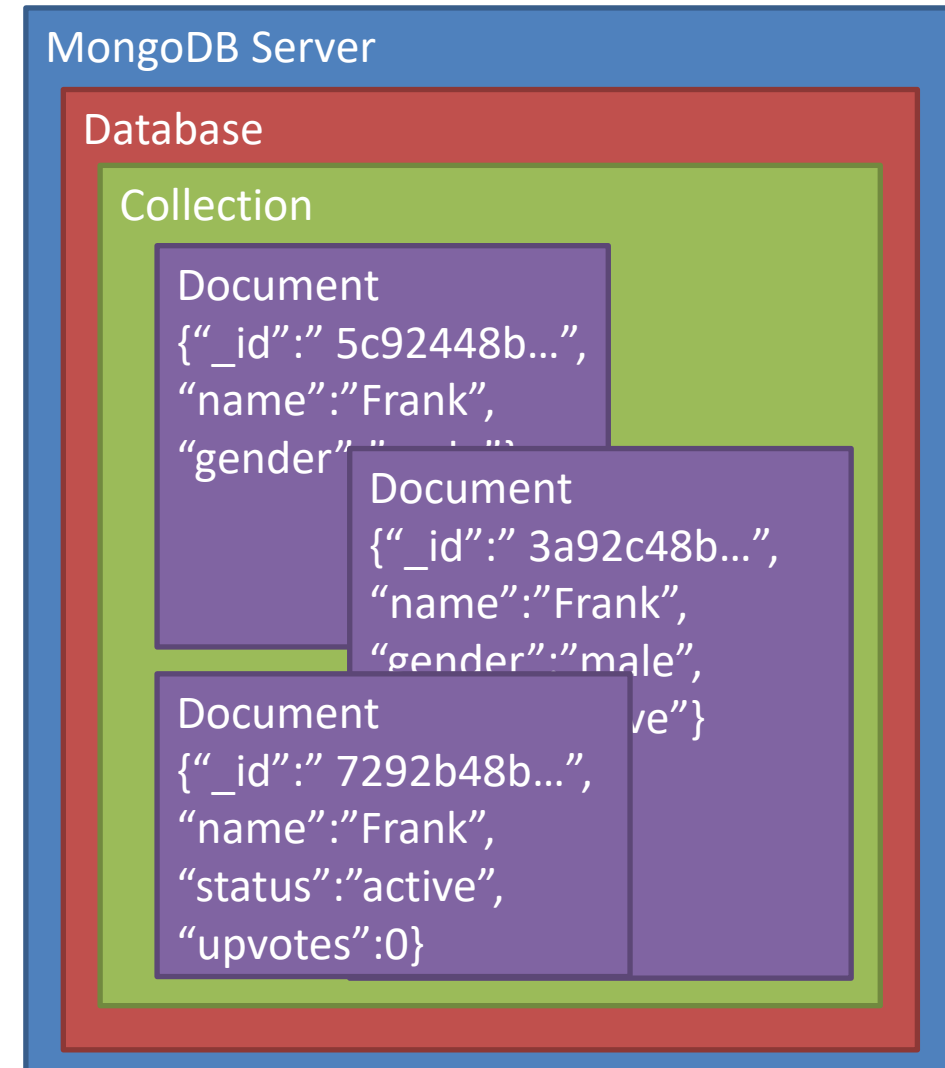- The documents can all be different
  - means you have rapid development
  - adding a property is easy - just starting using in your code
- Makes deployment easier and faster
  - roll-back and roll-forward are safe - unused properties are just ignored
- Collections can be indexed and queries
- Operations on individual documents are atomic

MongoDB Server

Database

Collection

Document
{"_id":" 5c92448b…",
"name":"Frank",
"gender"...}

Document
{"_id":" 3a92c48b…",
"name":"Frank",
"gender":"male",
...ve"}

Document
{"_id":" 7292b48b…",
"name":"Frank",
"status":"active",
"upvotes":0}

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

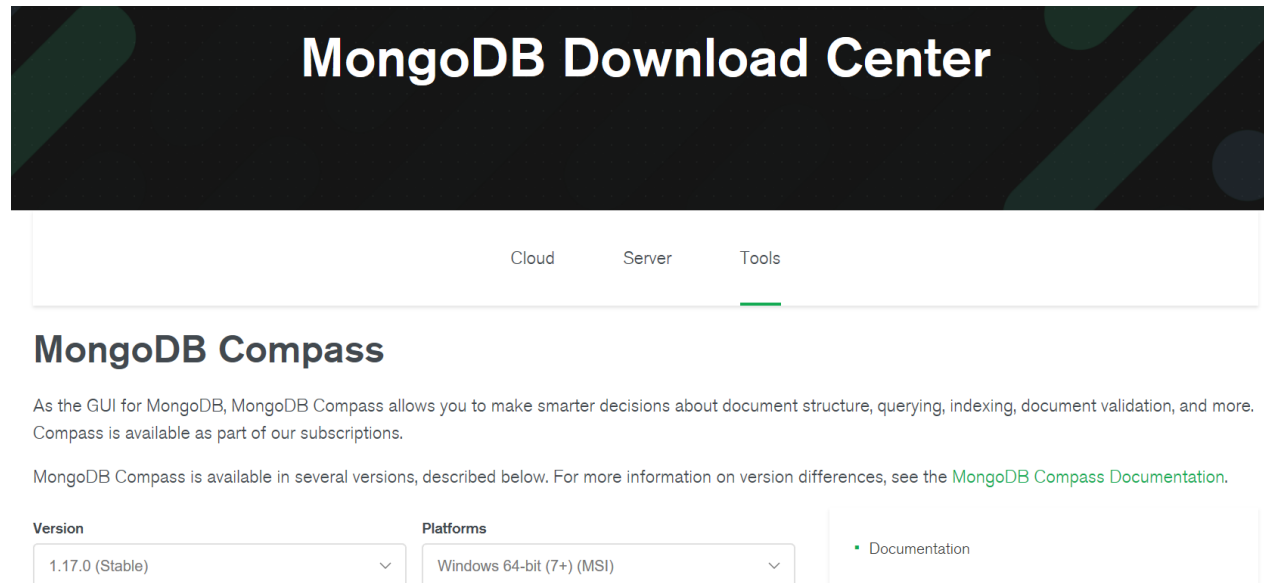   Install MongoDB Community Edition on macOS systems from MongoDB archives.

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](#):

MongoDB Download Center

Cloud   Server   Tools

MongoDB Compass

As the GUI for MongoDB, MongoDB Compass allows you to make smarter decisions about document structure, querying, indexing, document validation, and more. Compass is available as part of our subscriptions.

MongoDB Compass is available in several versions, described below. For more information on version differences, see the MongoDB Compass Documentation.

Version	Platforms	
1.17.0 (Stable)	Windows 64-bit (7+) (MSI)	• Documentation

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 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.mongoDB);
```

Mongoose Schemas and Models

- A Mongoose schema defines the structure of the document
  - Properties, default values, validation etc.
- A Mongoose model is a "wrapper" on the Mongoose schema.
  - provides an interface to the database for creating, querying, updating, deleting records, etc

```javascript
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);
```

Demo

- Contacts API: Manage contact details...

Contacts Schema:
```
name: String
address: String
age: Number
email: String
updated: Date
```
Solution in Code Examples...

Sub Documents

Validators

Queries

Object Refs.

LECTURE 2

Mongoose Schemas – Arrays & sub-documents

Example: Data Model for supports user Posts on a web site:

Title: Check the Dune movie trailer
Link: http://dunethemovie.com
username:fxwalsh
upvotes:20
comments:

body: Wow Looks Great!!!
author: JPrithet
upvotes: 0

body: When is it out?
author: BHope
upvotes: 0

body: Original is better!
author: MPower
upvotes: 0

```javascript
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);
```

Mongoose Schemas - Arrays

```javascript
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 :

```javascript
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);
```

```javascript
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);
```

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

Mongoose Custom Validation

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

```
//Make sure name starts with capital letter,
const nameValidator = (name) => {
    let nameRegEx = /\b([A-ZÀ-ÿ][-,a-z. ']+[ ]*)+/;
    return nameRegEx.test(name);
};

ContactSchema.path('name').validate(nameValidator);
```

Using Regular Expression (regex) to test for a valid name format, for example begins with a capital letter. If you've not come across them before check out https://www.w3schools.com/jsref/jsref_obj_regexp.asp

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

Data Manipulation Mongoose

- Mongoose supports all the CRUD operations:
    - **Create** –> Model.create(docs) or new Model(doc).save()
    - **Read** –> Model.find(conditions)
    - **Update** –> Model.update(conditions, props)
    - **Remove** –> Model.remove(conditions)
- Can operate with "*error first*" callbacks, promises, pr async-await

Create with Mongoose (async-await)

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

const ContactSchema = new Schema({
    name: { type: String, required: true },
    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);
```

```
∨ api \ contacts
  JS contactModel.js
  JS index.js
```

```javascript
import express from 'express';
import Contact from './contactModel';

const router = express.Router();

// add a contact (simple - no validation or body checking)
router.post('/', async (req, res) => {
  const newContact = await new Contact(req.body).save();
  res.status(201).json(newContact);
}
);
```

Create with Mongoose (promise)

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

const ContactSchema = new Schema({
    name: { type: String, required: true },
    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);
```

```
∨ api \ contacts
    JS  contactModel.js
    JS  index.js
```

```js
import express from 'express';
import Contact from './contactModel';

const router = express.Router();

// add a contact (simple - no validation or body checking)
router.post('/', (req, res) => {
    new Contact(req.body).save()
    .then(contact =>res.status(201).json(contact));
});
```

Update with Mongoose

- Async-Await Syntax

```
router.put('/:id', async (req, res, next) => {
  if (req.body._id) delete req.body._id;
  const query = { _id: req.params.id };
  const result = await Contact.update(query, req.body)
  res.status(201).json(result);
});

export default router;
```

- Promise Syntax

```
router.put('/:id', (req, res, next) => {
  if (req.body._id) delete req.body._id;
  const query = { _id: req.params.id };
  Contact.update(query, req.body)
    .then(contact => res.status(201).json(contact));
});
```

Mongoose Queries

- Mongoose supports many queries:
  - For equality/non-equality
  - Selection of some properties
  - Sorting
  - Limit & skip
- Mongoose models provide several static helper functions for CRU operations.
  - Model.findOne() returns a single document, the first match
  - Model.find() returns all
  - Model.findById() queries on the _id field.

- `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()`

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

Object Referencing

- Can use Object Id to reference documents in other collections
- Similar to 'foreign keys' in SQL databases

Using Object ID to reference a document in Products

```
const ContactSchema = new Schema({
    name: { type: String, required: true },
    address: String,
    age: {
        type: Number,
        min: 0,
        max: 120,
        required: true
    },
    email: String,
    updated: {
        type: Date,
        default: Date.now
    },
    productsPurchased: [{type: mongoose.Schema.Types.ObjectId, ref: 'Products'}]
}
);
```

Query Population using Refs

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

```
//Demo of Referencing in Mongoose: Error handling left out to simplify code.
//to run, type npx babel-node ref_example.js at the command line.

const createData = async () => {

    //Create a product
    const product1 = await new Product({
        productName: "PS5",
        productCode: "X123456"
    }).save();

    //Create contact and add product IDs
    const contact1 = await new Contact({
        name: "Frank Walsh",
        age: 22,
        productsPurchased: [product1._id]
    }).save();

    //query db for contact and populate user field
    const contact = await Contact.findById(contact1._id).populate('productsPurchased');
    console.log(JSON.stringify(contact, null, "\t"));
```

output

```
{
    "productsPurchased": [
        {
            "_id": "5fc8d72bdf5a440d7c003be4",
            "productName": "PS5",
            "productCode": "X123456",
            "__v": 0
        }
    ],
    "_id": "5fc8d72bdf5a440d7c003be5",
    "name": "Frank Walsh",
    "age": 22,
    "updated": "2020-12-03T12:16:43.336Z",
    "__v": 0
}
```

SCHEMA METHODS

Example: Using Schema Methods for Simple Authentication

- Restrict access to 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 you 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 Depen

express-session

npm v1.15.6   downloads 3M/m   build passing   coverage 100%

nstallation

a Node.js module available through the npm reg

command:

` express-session

1. User Schema with Static & Instance Methods

```javascript
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.

2. 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
}));
```

3. Use User Route to authenticate

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

/api/users/index.js

```javascript
// 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

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

4. Add Authentication Middleware

authenticate.js

```javascript
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

```javascript
import authenticate from './authenticate';

app.use('/api/movies', authenticate, moviesRouter);
```

Authentication middleware applied on /api/movies route.