



# MongoDB and Mongoose

Frank Walsh, Diarmuid O'Connor

# Agenda

- Cloud Databases
- MongoDB
  - Querying
  - Integrating with Node.js
  - The Contacts API implementation

# Databases in Enterprise Apps

- Most data driven enterprise applications need a database
- In traditional enterprise applications, this requires
  - Backups
  - Fail over
  - Maintenance
  - Capacity provisioning
- Usually handled by a Database Administrator.

# Databases in the Cloud

- For some apps, a traditional database may not be the best fit
  - Does the app require transactional integrity
  - Do you need db schema definition
  - Do you know your scaling requirements, particularly if it's a web app
- 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
- Latency is predictable and constant
- No need to define schemas 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**
- Use in-memory storage aggressively

# 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
  - but closer to traditional SQL databases than others
- Uses JSON natively - perfect fit for Node.js
- Query language with many SQL features
  - Uses JSON too, and has an easy learning curve
- Commercial support - 10gen.com product
  - cloud hosting providers - e.g. mongoLab.com
- Community support - popular choice

# Mongo Terminology

- Each database contains a set of "Collections"
- Collections are analogous to SQL tables
- 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

# Getting Started (locally)

- For complete MongoDB installation instructions, see [the manual](#).
- Starting MongoDB:  
`mongod`
- This starts the process.
- Can add other parameters, for instance location of data.

# Mongo Shell

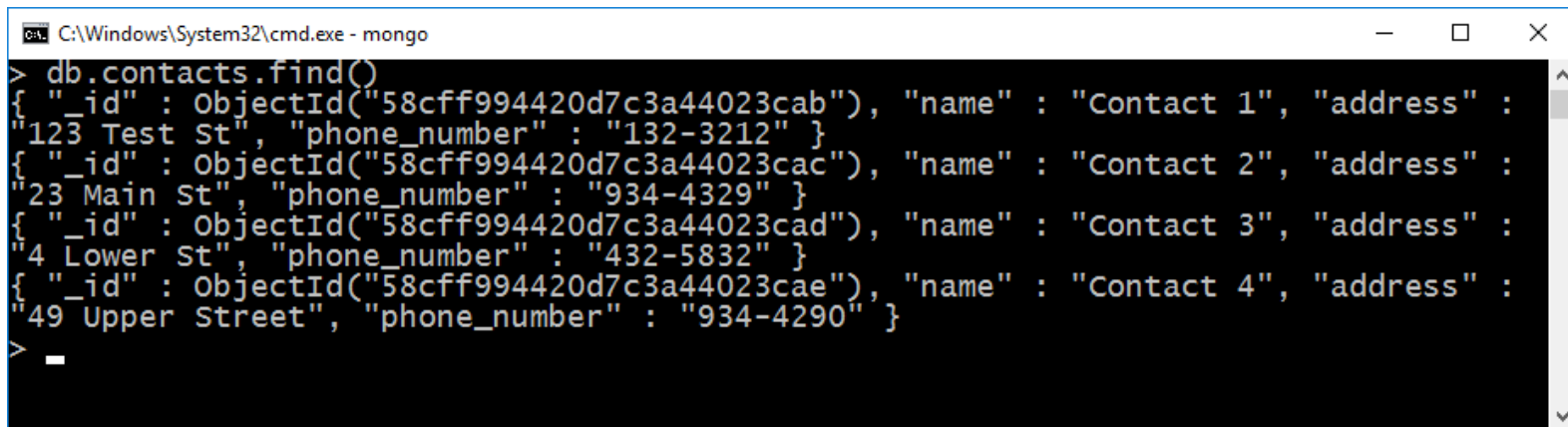
- Interactive JavaScript interface to MongoDB.
- Query/update data and perform administrative operations.

```
C:\repos\webervicesdev-2017>mongo
MongoDB shell version v3.4.2
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 3.4.2
Server has startup warnings:
2017-03-20T13:49:38.768+0000 I CONTROL [initandlisten]
2017-03-20T13:49:38.769+0000 I CONTROL [initandlisten] ** WARNING: Access control is not enabled for the database.
2017-03-20T13:49:38.773+0000 I CONTROL [initandlisten] ** Read and write access to data and configuration is u
nrestricted.
2017-03-20T13:49:38.775+0000 I CONTROL [initandlisten]
>
```

- By default, Mongo shell will attempt to connect to the MongoDB instance running on the localhost interface on port 27017.

# The MongoDB Query Language

- MongoDB provides a JavaScript API and JSON-based query language
- Use the MongoDB shell to execute queries
  - similar to using MySQL console
- Example: list of contacts

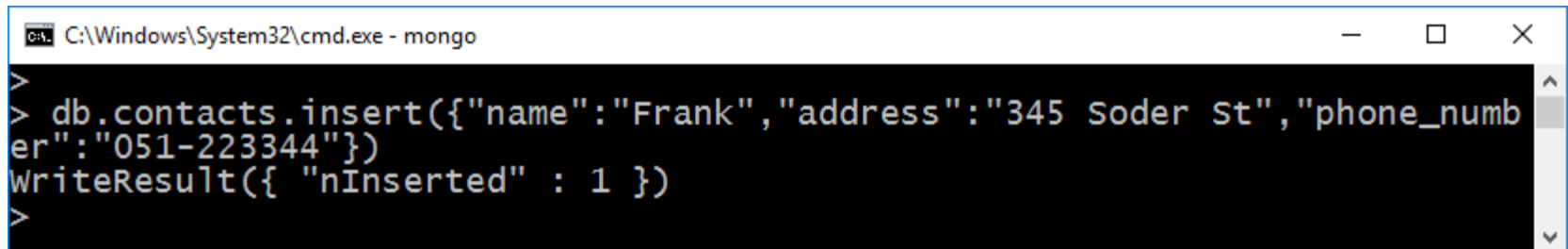


```
C:\Windows\System32\cmd.exe - mongo
> db.contacts.find()
{"_id" : ObjectId("58cff994420d7c3a44023cab"), "name" : "Contact 1", "address" :
123 Test St", "phone_number" : "132-3212" }
{"_id" : ObjectId("58cff994420d7c3a44023cac"), "name" : "Contact 2", "address" :
23 Main St", "phone_number" : "934-4329" }
{"_id" : ObjectId("58cff994420d7c3a44023cad"), "name" : "Contact 3", "address" :
4 Lower St", "phone_number" : "432-5832" }
{"_id" : ObjectId("58cff994420d7c3a44023cae"), "name" : "Contact 4", "address" :
49 Upper Street", "phone_number" : "934-4290" }
>
```

- db - current database
- contacts - the contacts collection
- .find() - collection API method (corresponds to collection URL in last lecture...)
- The Result Set is a list of JavaScript objects, representing matched documents

# MongoDB: Inserts

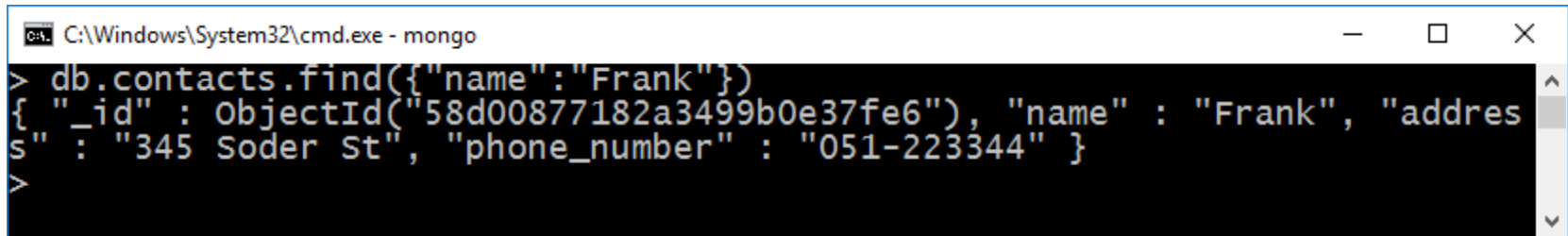
- Collections do not need to be created explicitly
  - just insert a document
- MongoDB automatically assigns a 12 byte unique identifier to any document
  - the **\_id** property
  - Stored internally as binary

A screenshot of a Windows command prompt window titled "C:\Windows\System32\cmd.exe - mongo". The window has standard Windows window controls (minimize, maximize, close) in the top right corner. The command prompt shows a MongoDB shell session with the following text:

```
> db.contacts.insert({"name":"Frank","address":"345 Soder St","phone_number":"051-223344"})
WriteResult({ "nInserted" : 1 })
>
```

# MongoDB:Queries

- Documents are retrieved by specifying a set of conditions to match against
- simplest case : query-by-example
- provide a subset of properties that must match

A screenshot of a Windows command prompt window titled "C:\Windows\System32\cmd.exe - mongo". The prompt shows a MongoDB query: `> db.contacts.find({"name":"Frank"})`. The result is a JSON document: `{ "_id" : ObjectId("58d00877182a3499b0e37fe6"), "name" : "Frank", "address" : "345 Soder St", "phone_number" : "051-223344" }`. The window has standard Windows window controls (minimize, maximize, close) in the top right corner.

```
C:\Windows\System32\cmd.exe - mongo
> db.contacts.find({"name":"Frank"})
{ "_id" : ObjectId("58d00877182a3499b0e37fe6"), "name" : "Frank", "address" : "345 Soder St", "phone_number" : "051-223344" }
```

- More complex queries use a convention of embedded meta-properties to specify conditions these are signified with a \$ prefix.
  - Example: `{name:{$exists:true}}`  
returns documents that have a name property

# MongoDB: Queries

- Common meta-properties used to query data are:
  - \$gt, \$gte, \$lt, \$lte** meaning  $>$ ,  $\geq$ ,  $<$ ,  $\leq$

```
C:\Windows\System32\cmd.exe - mongo
> db.contacts.insert({"name":"Mary","age":21,"address":"345 Soder St","phone_number":"051-223344"})
WriteResult({ "nInserted" : 1 })
> db.contacts.insert({"name":"Jane","age":31,"address":"345 Keel St","phone_number":"051-445566"})
WriteResult({ "nInserted" : 1 })
> db.contacts.find({"age":{"$gte":21,$lt:31}})
{"_id" : ObjectId("58d00909182a3499b0e37fe7"), "name" : "Mary", "age" : 21, "address" : "345 Soder St", "phone_number" : "051-223344" }
```

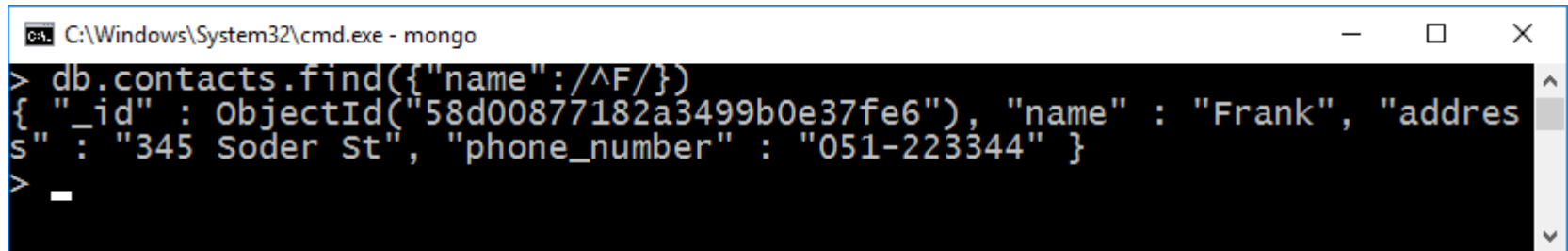
- \$or, \$in, \$nin**

```
C:\Windows\System32\cmd.exe - mongo
> db.contacts.find({"name":{"$in":["Jane","Frank"]}})
{"_id" : ObjectId("58d00877182a3499b0e37fe6"), "name" : "Frank", "address" : "345 Soder St", "phone_number" : "051-223344" }
{"_id" : ObjectId("58d00939182a3499b0e37fe8"), "name" : "Jane", "age" : 31, "address" : "345 Keel St", "phone_number" : "051-445566" }
```



# MongoDB: Queries

- **regular expressions** {word: /th^/i }

A screenshot of a Windows command prompt window titled "C:\Windows\System32\cmd.exe - mongo". The prompt shows a MongoDB query: > db.contacts.find({"name":/^F/}) followed by a JSON document: {"\_id": ObjectId("58d00877182a3499b0e37fe6"), "name": "Frank", "address": "345 Soder St", "phone\_number": "051-223344" }. The prompt is followed by a cursor and a hyphen.

```
C:\Windows\System32\cmd.exe - mongo
> db.contacts.find({"name":/^F/})
{"_id": ObjectId("58d00877182a3499b0e37fe6"), "name": "Frank", "address": "345 Soder St", "phone_number": "051-223344" }
> _
```

- db.contacts.find().limit(5)
  - limits the number of documents in the result set.
- db.contacts.find().skip( 5 )
  - Set the Starting Point of the Result Set

# MongoDB:Updates

- Documents are updated by providing:
  - a query to select the relevant subset of documents,
  - an update specification, which is either:
    - a complete replacement document
    - meta-properties that modify specific document properties
- example:  
**\$set** changes specific properties  
Example:complete replacement:  
> db.city.insert( {name:'dublin'} )  
> db.city.update( {name:'dublin'}, {name:'Dublin',county:'Dublin'} )
- Example:modify specific properties:  
> db.city.insert( {name:'Cork',county:'cork'} )  
> db.city.update( {name:'Cork'}, {\$set:{county:'Cork'}} )
- See <http://www.mongodb.org/display/DOCS/Updating> for more

# MongoDB:Update Properties

- Common meta-properties used with the update command are:
  - **\$set** - sets specified properties, but leaves others alone  
\$set:{name:'New Name'}
- **\$unset** - deletes specified properties  
\$unset:{name:1}
- **\$inc** - increments a numeric property  
inc:{ upvotes: 2 }  
adds 2 to the counter property, or if it does not exist, sets it to 2
- **\$push, \$pop** - add to or remove values from, an array
  - \$push: { comments: {who:..., msg:...} }
  - \$pop: {comments: -1 }

# MongoDB:Upserts

- The MongoDB update command can optionally insert a document if it is not found. This is known as an 'upsert'
- This is useful when starting counters as it avoids corrupting the count when two independent updates try to initialize the counter.

```
db.counters.update( {name:'foo'}, {$inc:{value:1}},  
true)
```

- The first update will create the counter:  
{name:'foo', value:1}
- The second update will increment the counter:  
{name:'foo', value:2}

**MONGOOSE**

# 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

# Installing Mongoose

- Run the following from the CMD/Terminal
  - \$ npm install -save mongoose
- In node
  - Load the module
    - import mongoose from 'mongoose';
- Connect to the database
  - mongoose.connect(mongoDbPath);

# 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

```
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}],
```

reviews property is an  
Array of  
MovieReviewSchema

# Mongoose Schema - Validation

- Can define validation constraints on properties :

```
1  const mongoose = require('mongoose'),
2  Schema = mongoose.Schema;
3
4  ▼ const ContactSchema = new Schema({
5    name: String,
6    address: String,
7    age: { type: Number, min: 0, max: 120 },
8    email: String,
9    updated: { type: Date, default: Date.now }
10  });
11
12  const ContactModel = mongoose.model('contacts', ContactSchema);
```

# Mongoose Custom Validation

- Developers can define custom validation on their properties (e.g. validate length of comment 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*" callback 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: Movies – Adding a review to a favourite movie.

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

# 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

# MongoDB as a Service

## MongoDB Atlas


Cloud-hosted MongoDB service on AWS, Azure and Google Cloud. Deploy, operate, and scale a MongoDB database in just a few clicks

[Start Free](#)



# MongoDB as a Service

- Most providers allow free access to their
- Provide user credentials wrapped in a URL
- All you need to do is update your config with the relevant URL
- Careful to ignore credentials when pushing to github/public repo



[Home](#)  
Database: contacts\_db

To connect using the mongo shell:

```
% mongo ds039311.mlab.com:39311/contacts_db -u <dbuser> -p <dbpassword>
```

To connect using a driver via the standard MongoDB URI ([what's this?](#)):

```
|mongodb://<dbuser>:<dbpassword>@ds039311.mlab.com:39311/contacts_db
```