



# MongoDB, Mongoose and Cloud Storage

Frank Walsh, Diarmuid O'Connor

# Agenda

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



# Databases in Enterprise Apps

---

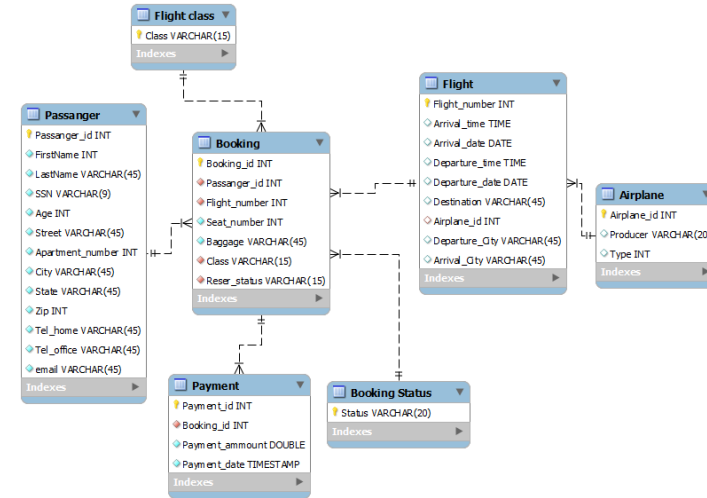
- Most data driven enterprise applications need a database
  - Persistence: storage of data
  - Concurrency: many applications sharing the data at once.
  - Integration: multiple systems using the same DB
- Enterprise Application DBs require backups, fail over, maintenance, capacity provisioning.
  -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



# Structured & Unstructured Data

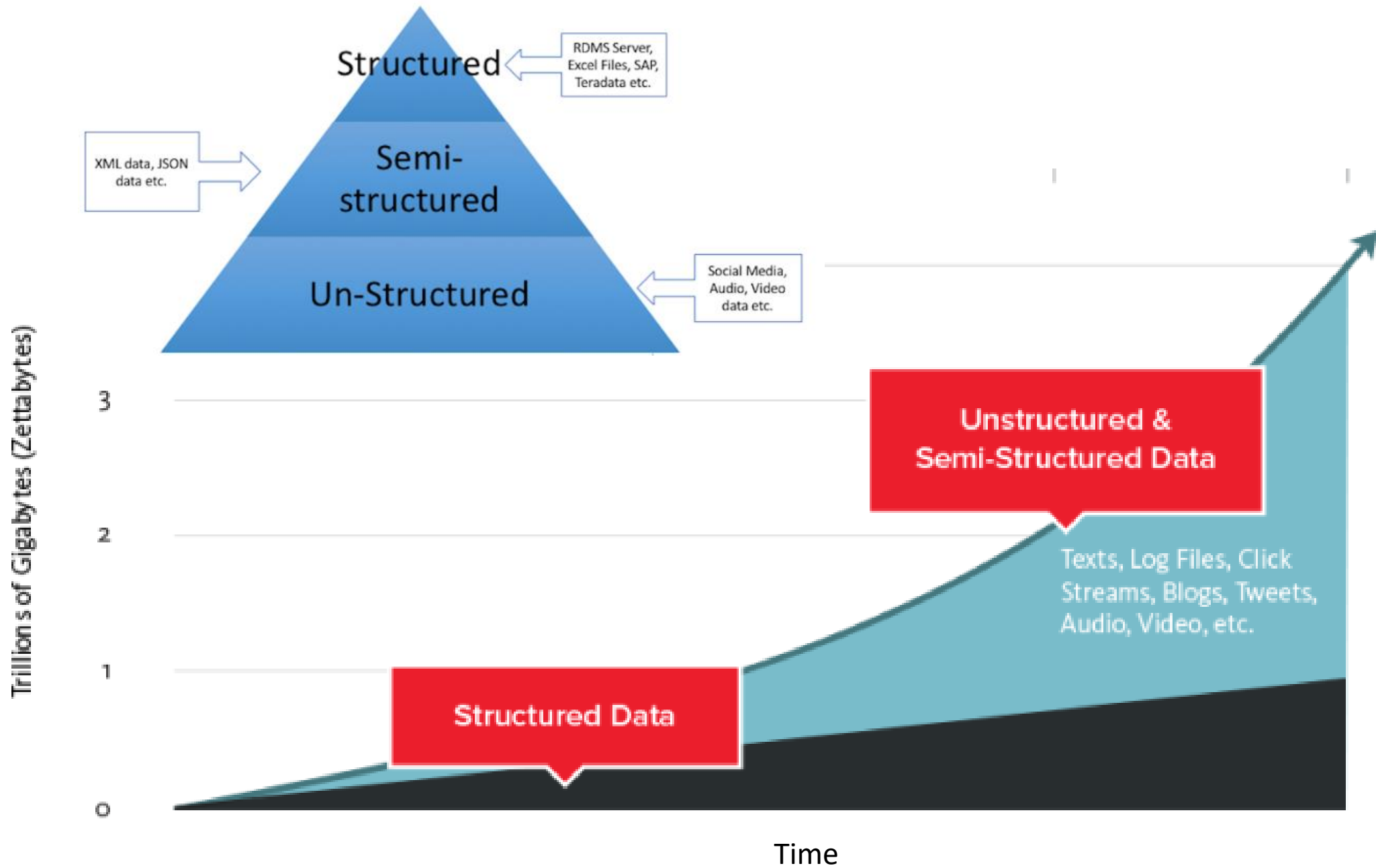
- Structured data:
  - Organise data into structured tables and rows
  - Relations have to be simple, they cannot contain any structure such as a nested record or a list
- Unstructured Data
  - Much more varied
  - No pre-defined Structure
  - E.g. text files, images, audio
- Semi-structured data:
  - JSON/XML
  - Doesn't obey tabular structure of Relational DB
  - Sometime "self-describing" using tags

## Relational Database



## JSON

```
{
  "employees": [
    {
      "division": "Engineering",
      "name": "Michael"
    },
    {
      "division": "HR",
      "name": "Laura"
    },
    {
      "division": "Marketing",
      "name": "Elise"
    }
  ],
  "location": {
    "city": "Mountain View",
    "country": "US",
    "state": "California",
    "street": "1600 Amphitheatre Parkway"
  },
  "name": "Google"
}
```



# Databases in the Cloud

- For some apps, a traditional relational database (structured data) 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?
  - 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.






[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" ]  
}
```

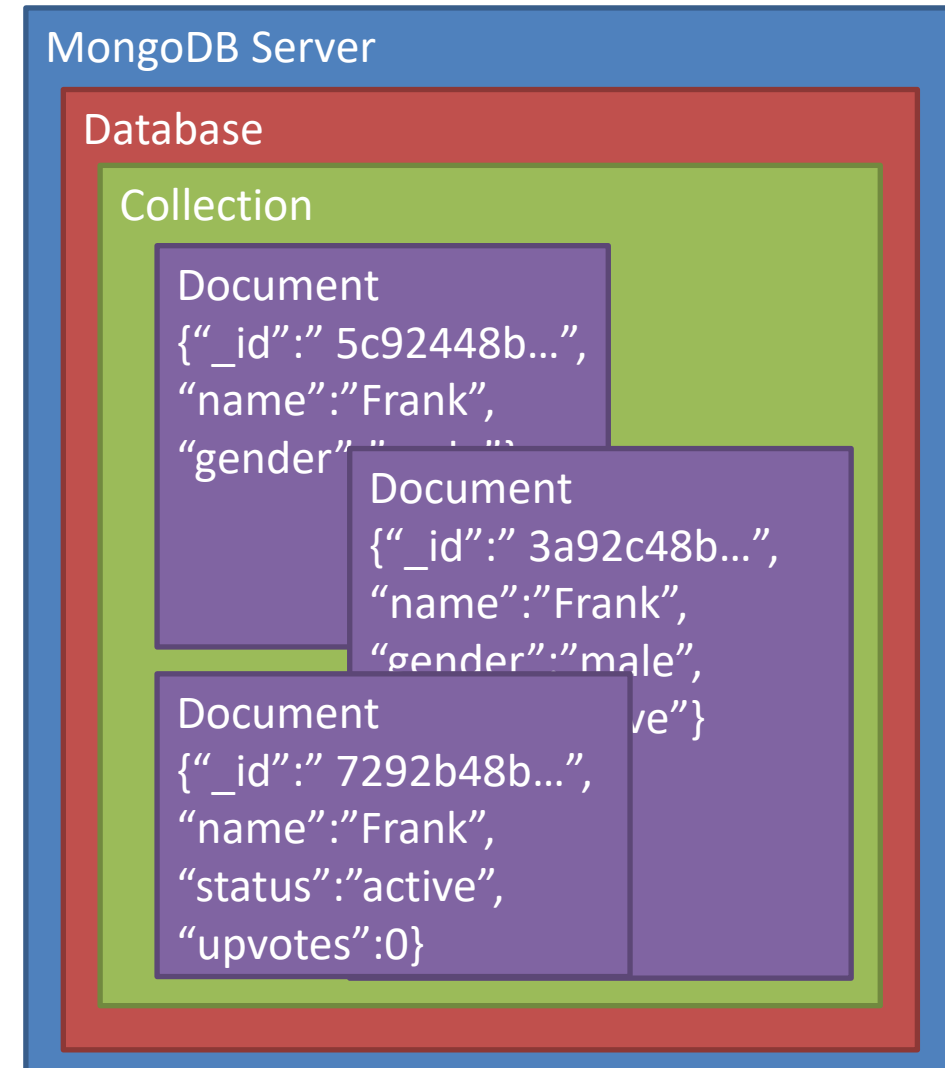


← field: value  
← field: value  
← field: value  
← field: value



# 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.

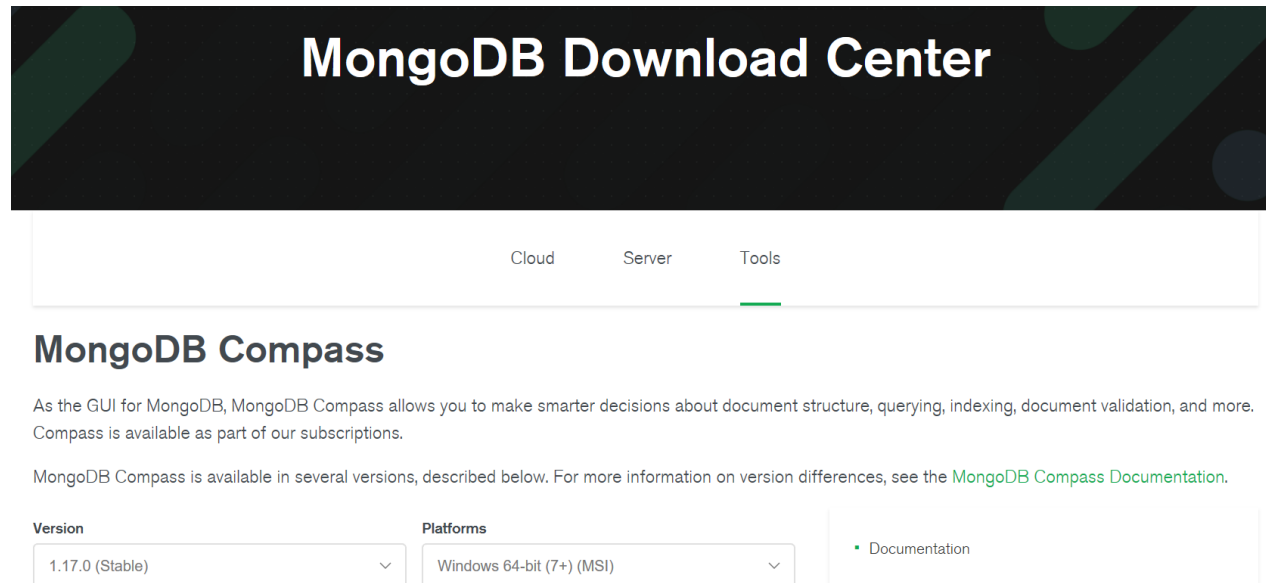
[Install on Windows](#)  
Install MongoDB Community Edition on Windows systems and optionally start MongoDB as a Windows service.

- 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 Hibernate
- Perhaps an 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

```
// Connect to database
mongoose.connect("mongodb://localhost:27017/contacts_db");
const db = mongoose.connection;

db.on('error', (err) => {
  console.log(`database connection error: ${err}`);
});
db.on('disconnected', () => {
  console.log('database disconnected');
});
db.once('open', () => {
  console.log(`database connected to ${db.name} on ${db.host}`);
});
```



# Mongoose Schemas and Models

- Mongoose supports models
  - Used for creating and reading documents from the underlying MongoDB database
- Mongoose models are “compiled” using a **mongoose.Schema**
  - Each of the properties must have a type
    - Number, String, Boolean, array, object

```
constructor() {  
  super();  
  const contactSchema = new mongoose.Schema({  
    email: { type: String, unique: true, index: true },  
    password: String,  
    phone: String,  
    type: String,  
    dob: String,  
    userName: String,  
    name: String,  
  });  
  this.model = mongoose.model('Contacts', contactSchema);  
}
```

# Mongoose Schemas – Arrays & Subdocuments

```
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)

```
import mongoose from 'mongoose';

const Schema = mongoose.Schema;

const validateEmail = email => {
  const re = /^^\w+([\.-]?\w+)*@\w+([\.-]?\w+)*(\.\w{2,3})+$/;
  return re.test(email)
}

const UserSchema = new Schema({
  username: { type: String, required: true },
  password: { type: String, required: true },
  email: { type: String, validate: [validateEmail, "Please fill a valid email address"] }
});
```

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)

HOWEVER WE ARE USING JOI FOR THIS....  
(Make it independent of data layer)

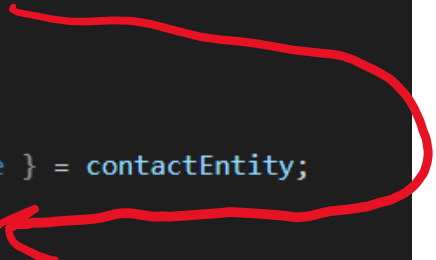
# Data Manipulation Mongoose

- 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, promises, or **async await**.

# Create with Mongoose

- AccountRepository.js

```
export default class extends AccountRepository {  
  
  constructor() {  
    super();  
    const contactSchema = new mongoose.Schema({  
      email: { type: String, unique: true, index: true },  
      password: String,  
      phone: String,  
      type: String,  
      dob: String,  
      userName: String,  
      name: String,  
    });  
    this.model = mongoose.model('Contacts', contactSchema);  
  }  
  
  async persist(contactEntity) {  
    const { email, password, phone, type, dob, userName, name } = contactEntity;  
    const mongooseAccount = new this.model(contactEntity);  
    await mongooseAccount.save();  
    return new Account(mongooseAccount.id, mongooseAccount.name, mongooseAccount.userName,  
  }  
}
```



# Update with Mongoose

```
async merge(accountEntity) {  
  const { id, firstName, lastName, email, password } = accountEntity;  
  const mongooseAccount = this.model.findByIdAndUpdate(id, { firstName, lastName, email, password });  
  return new Account(mongooseAccount.id, mongooseAccount.firstName, mongooseAccount.lastName, mongooseAccount.password);  
}
```

# 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.

```
async getByEmail(email) {  
  const mongooseAccount = await this.model.findOne({email: email});  
  return new Account(mongooseAccount.id, mongooseAccount.firstName, mongooseAccount.lastName);  
}
```



# 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