MongoDB and Cloud Storage

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)

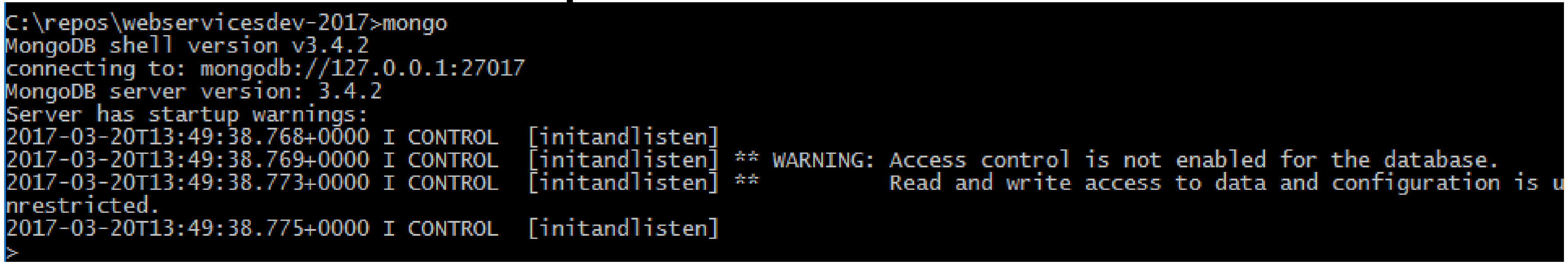
- For complete MongoDB installation instructions, see the manual.
- Starting MongoDB:

```
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.

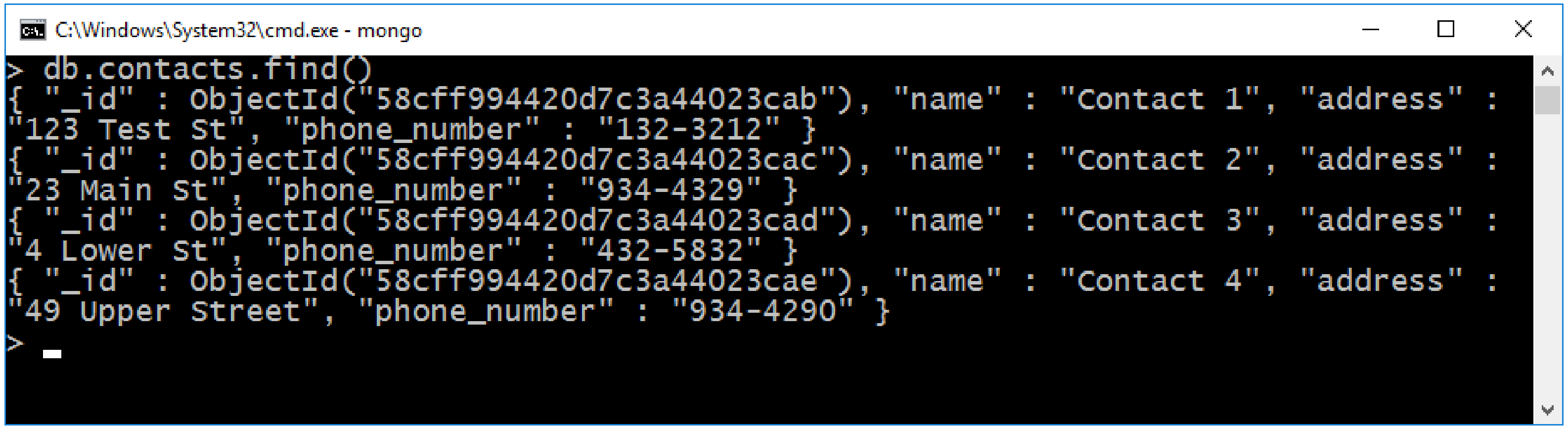


```
C:\repos\webservicesdev-2017>mongo
MongoDB shell version v3.4.2
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 3.4.2
Server has startup warnings:
2017-03-20T13:49:38.768+0000 I CONTROL  [initandlisten]
2017-03-20T13:49:38.769+0000 I CONTROL  [initandlisten] ** WARNING: Access control 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 usingMySQL console
- Example: list of contacts

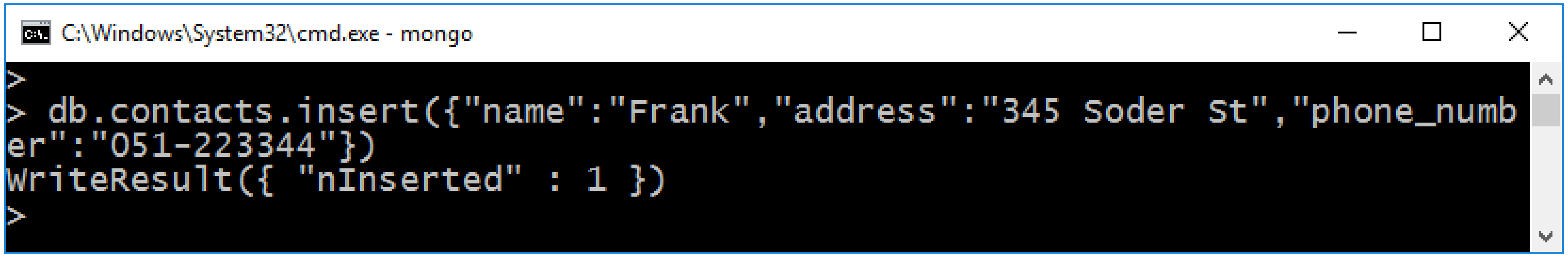


```
> db.contacts.find()
{ "_id" : ObjectId("58cff994420d7c3a44023cab"), "name" : "Contact 1", "address" :
"123 Test St", "phone_number" : "132-3212" }
{ "_id" : ObjectId("58cff994420d7c3a44023cac"), "name" : "Contact 2", "address" :
"23 Main St", "phone_number" : "934-4329" }
{ "_id" : ObjectId("58cff994420d7c3a44023cad"), "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() - collectionAPI method (coorresponds to collection URL in last lecture…)
- The Result Set is a list of JavaScript objects, representing matched documents

MongoDB: Inserts

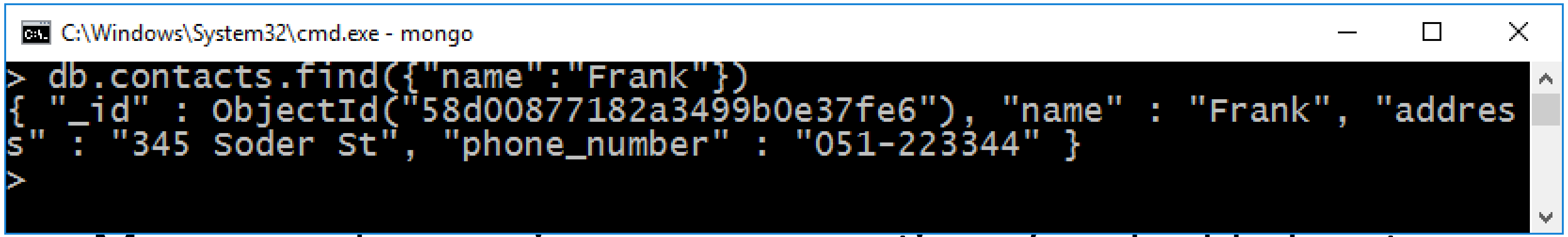
- Collections do not need to be created explicitly
  - just insert a document
- MongoDB automatically assigns a 12 byte unique identifier to any document
  - the **_id** property
  - Stored internally as binary



```
>
> db.contacts.insert({"name":"Frank","address":"345 Soder St","phone_numb
er":"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

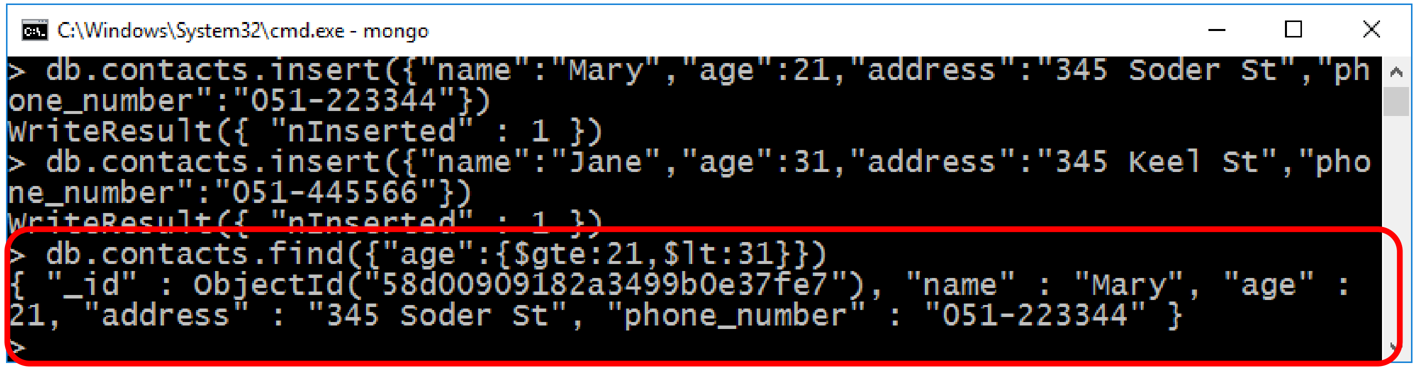


```
> db.contacts.find({"name":"Frank"})
{ "_id" : ObjectId("58d00877182a3499b0e37fe6"), "name" : "Frank", "addres
s" : "345 Soder St", "phone_number" : "051-223344" }
>
```

- 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

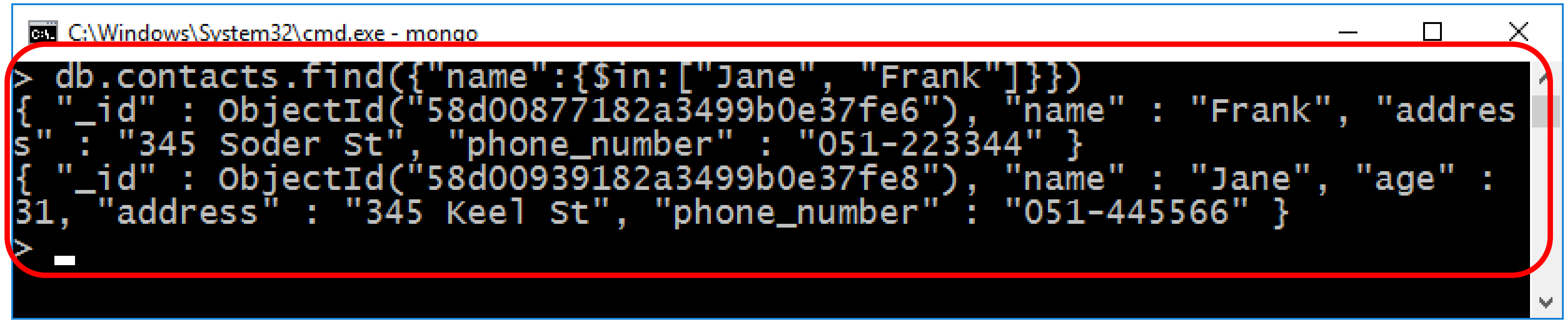
- Common meta-properties used to query data are:
  - **$gt, $gte, $lt, $lte** meaning>, >=, <,<=



```
> db.contacts.insert({"name":"Mary","age":21,"address":"345 Soder St","ph
one_number":"051-223344"})
WriteResult({ "nInserted" : 1 })
> db.contacts.insert({"name":"Jane","age":31,"address":"345 Keel St","pho
ne_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**

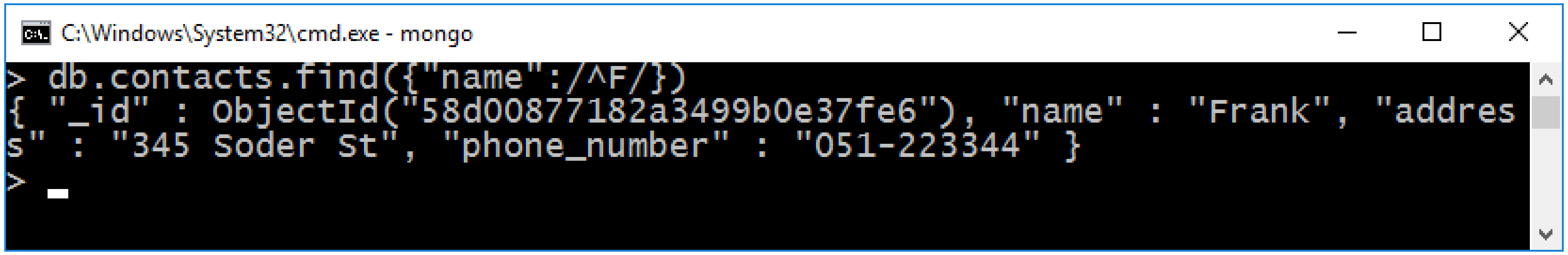
-



```
> db.contacts.find({"name":{$in:["Jane", "Frank"]}})
{ "_id" : ObjectId("58d00877182a3499b0e37fe6"), "name" : "Frank", "addres
s" : "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 }



C:\Windows\System32\cmd.exe - mongo

```
> db.contacts.find({"name":/^F/})
{ "_id" : ObjectId("58d00877182a3499b0e37fe6"), "name" : "Frank", "addres
s" : "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
```

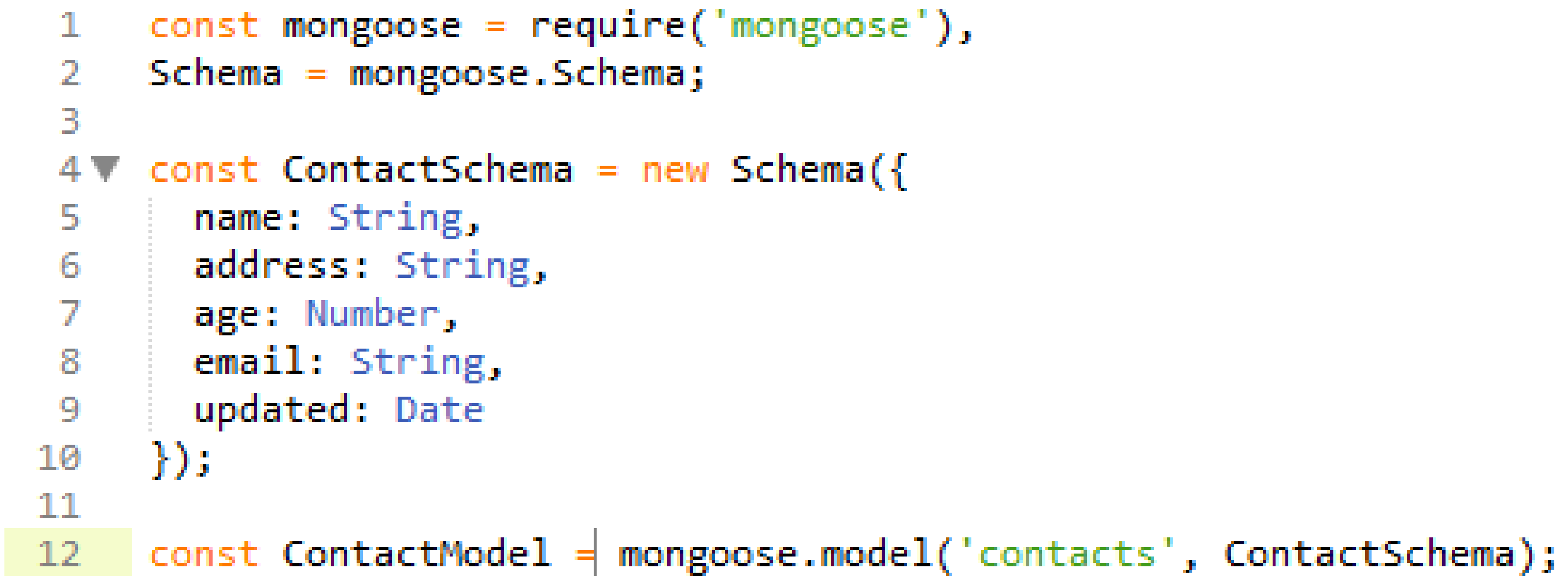
- In node
  - Load the module

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

- Connect to the database
  - mongoose.connect(mongoDbPath);

Mongoose Schemas and Models

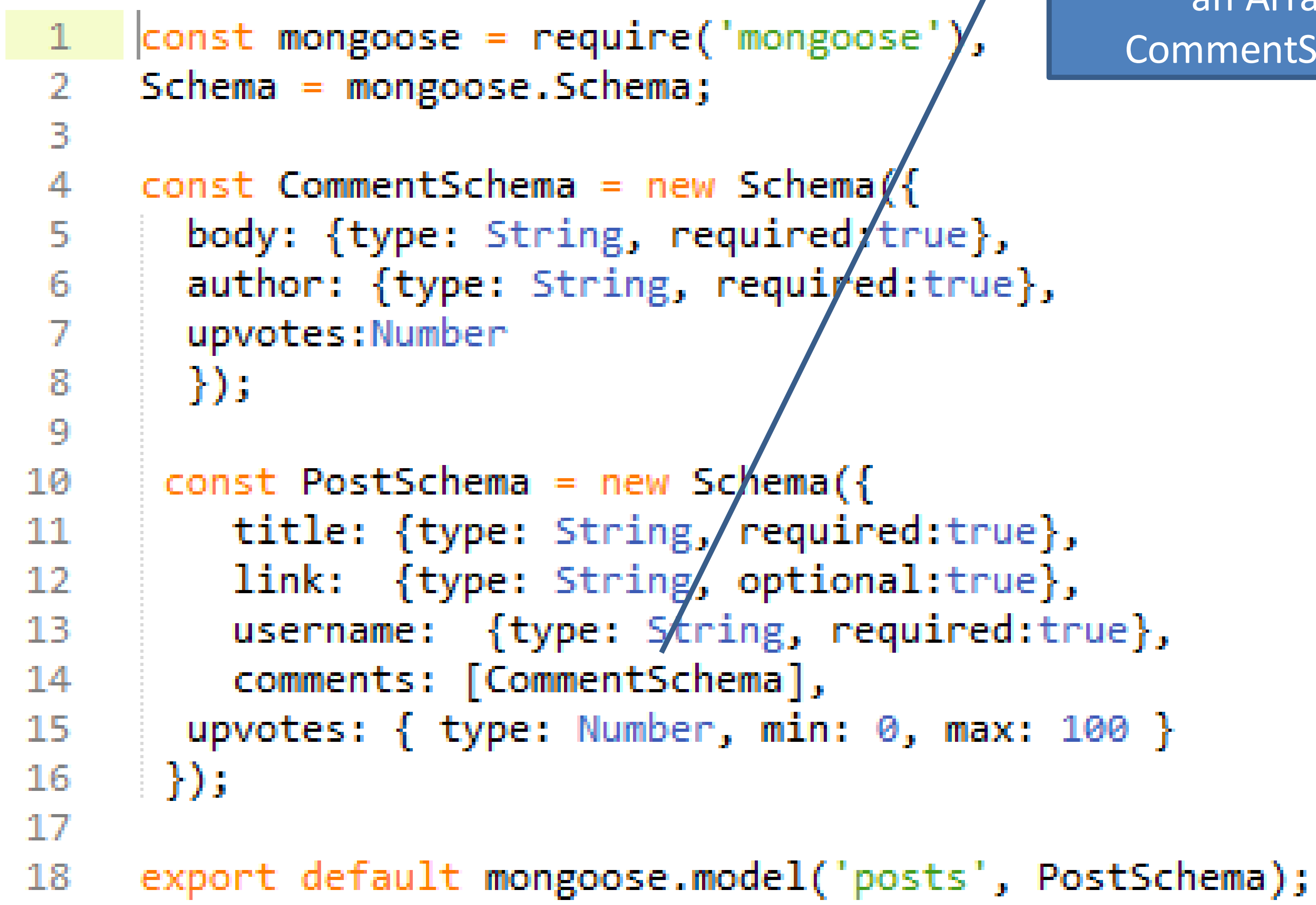
- Mongoose supports models
  - i.e. fixed types of documents
  - Needs a mongoose.Schema
  - Each of the properties must have a type
    - Number, String, Boolean, array, object



```javascript
const mongoose = require('mongoose'),
Schema = mongoose.Schema;

const ContactSchema = new Schema({
  name: String,
  address: String,
  age: Number,
  email: String,
  updated: Date
});

const ContactModel = mongoose.model('contacts', ContactSchema);
```

Mongoose Schemas - Arrays

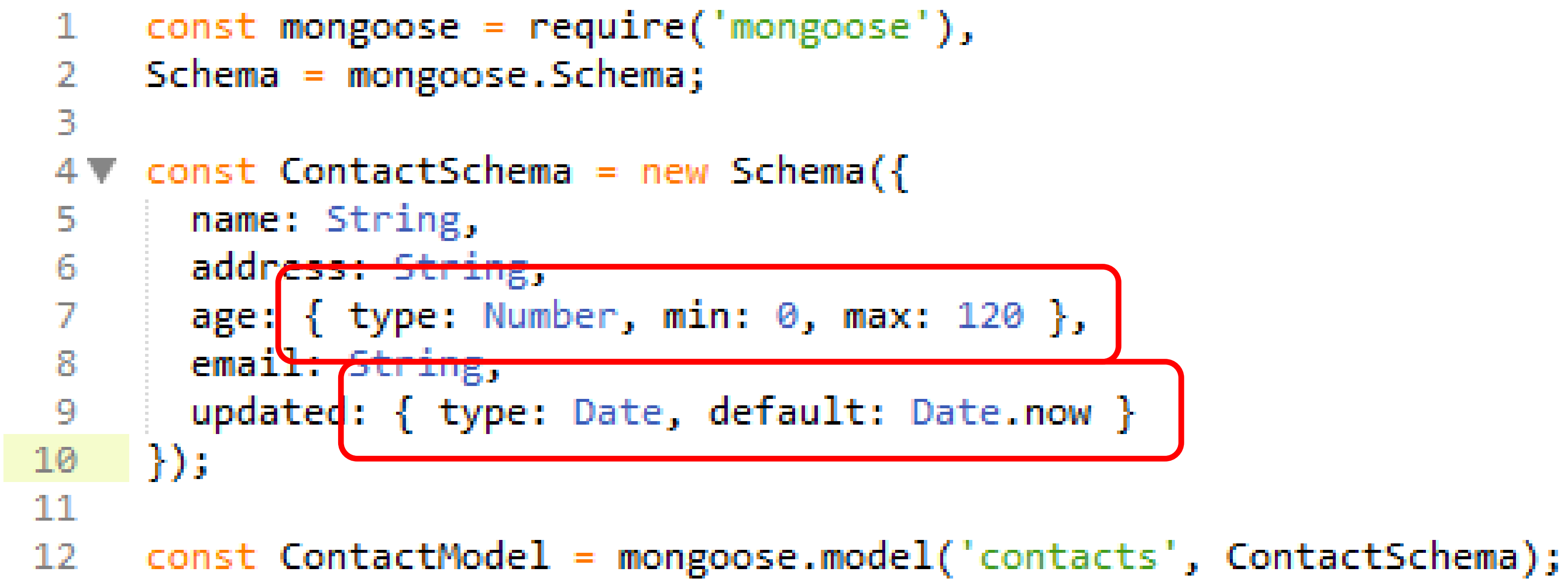
Comments property is an Array of CommentSchemas



```
const mongoose = require('mongoose'),
Schema = mongoose.Schema;

const CommentSchema = new Schema({
  body: {type: String, required:true},
  author: {type: String, required:true},
  upvotes:Number
  });

 const PostSchema = new Schema({
    title: {type: String, required:true},
    link:  {type: String, optional:true},
    username:  {type: String, required:true},
    comments: [CommentSchema],
  upvotes: { type: Number, min: 0, max: 100 }
 });

export default mongoose.model('posts', PostSchema);
```

Mongoose Schema - Validation

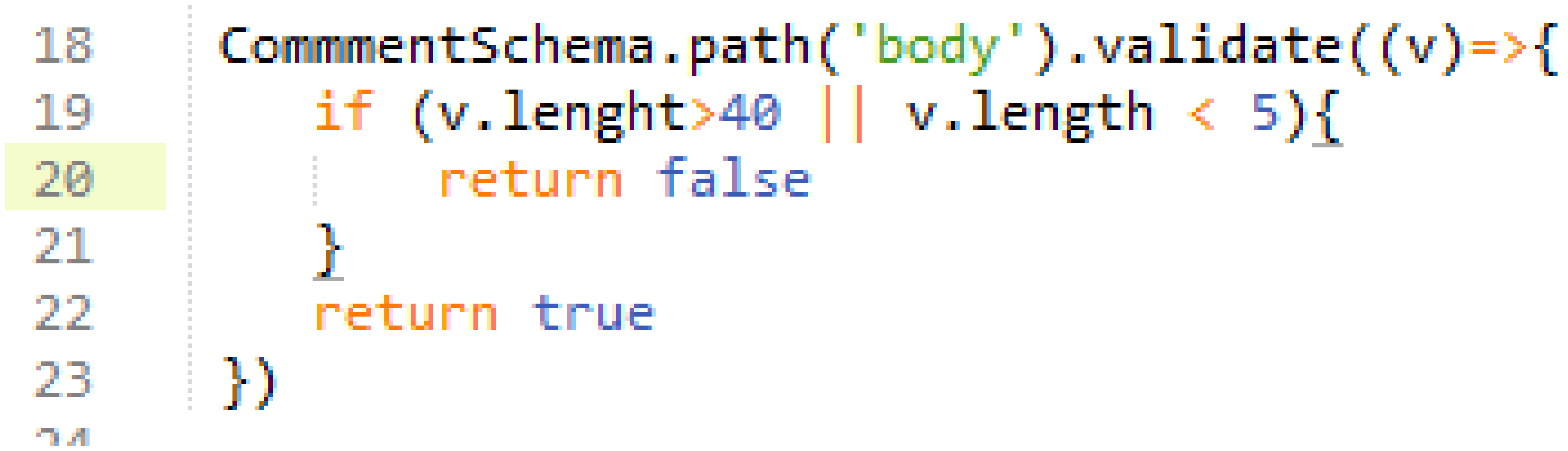
- Can define validation constraints on properties :



```
const mongoose = require('mongoose'),
Schema = mongoose.Schema;

const ContactSchema = new Schema({
  name: String,
  address: String,
  age: { type: Number, min: 0, max: 120 },
  email: String,
  updated: { type: Date, default: Date.now }
});

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)

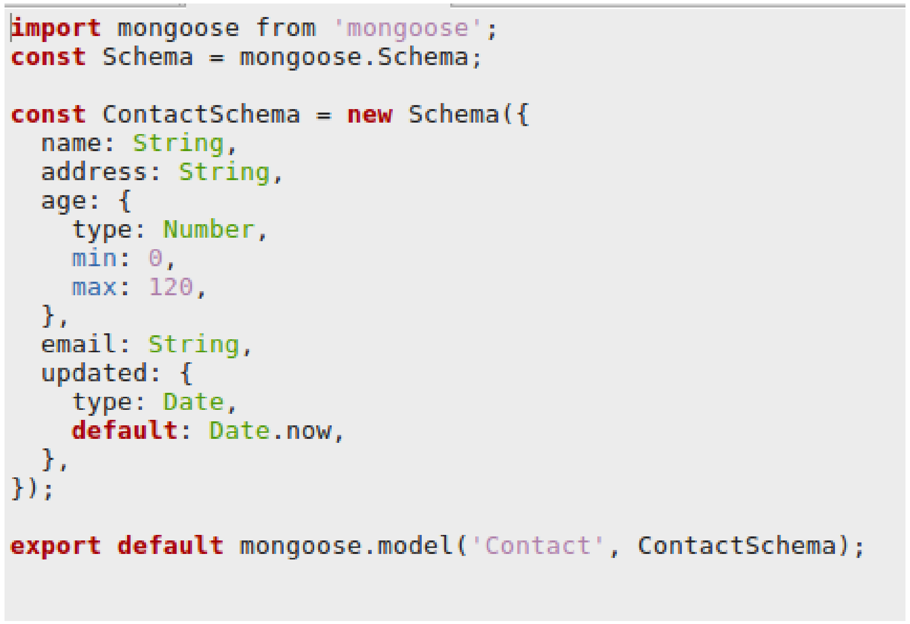


```
CommmentSchema.path('body').validate((v)=>{
    if (v.lenght>40 || v.length < 5){
        return false
    }
    return true
})
```

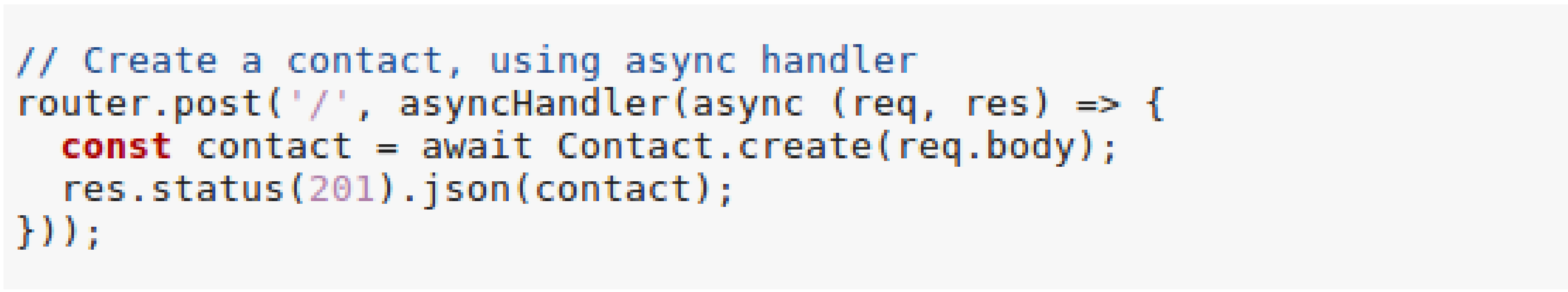
Data Manipulation Mongoose

- 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 Contact with Mongoose

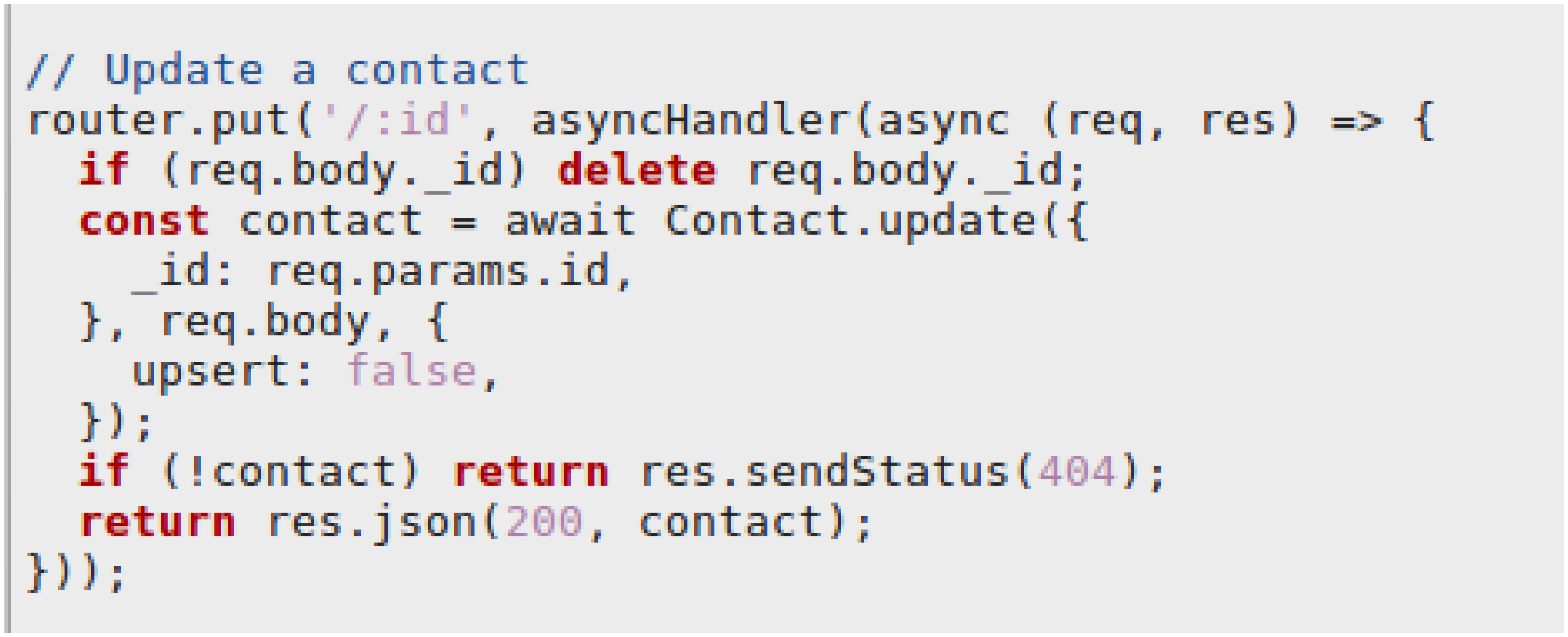


```
import mongoose from 'mongoose';
const Schema = mongoose.Schema;

const ContactSchema = new Schema({
  name: String,
  address: String,
  age: {
    type: Number,
    min: 0,
    max: 120,
  },
  email: String,
  updated: {
    type: Date,
    default: Date.now,
  },
});

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



```
// Create a contact, using async handler
router.post('/', asyncHandler(async (req, res) => {
  const contact = await Contact.create(req.body);
  res.status(201).json(contact);
}));
```

Update Contact with Mongoose



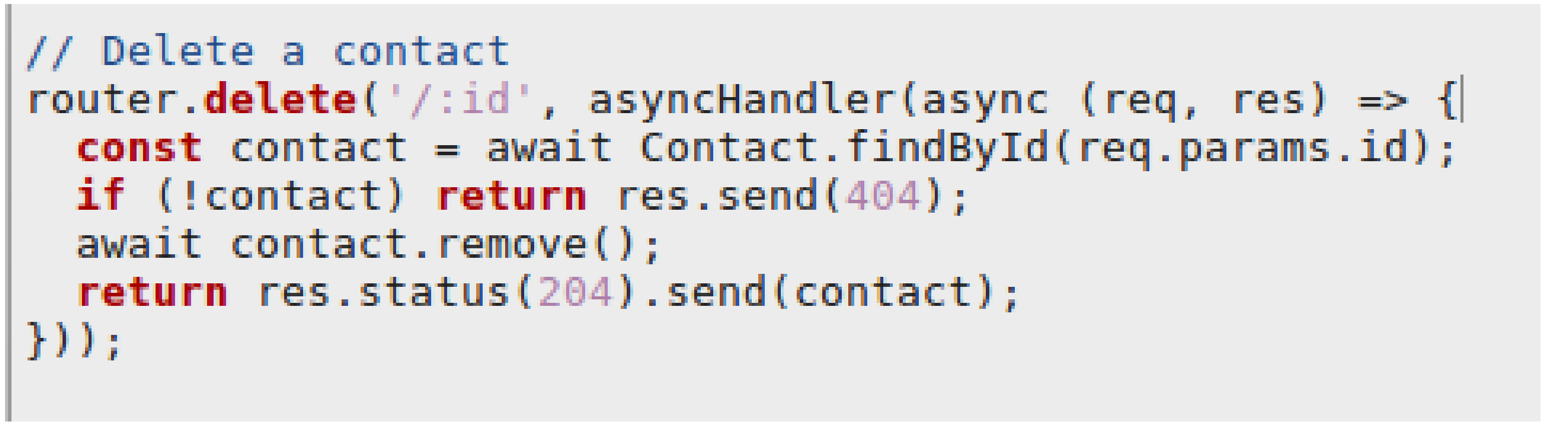
```
// Update a contact
router.put('/:id', asyncHandler(async (req, res) => {
  if (req.body._id) delete req.body._id;
  const contact = await Contact.update({
    _id: req.params.id,
  }, req.body, {
    upsert: false,
  });
  if (!contact) return res.sendStatus(404);
  return res.json(200, contact);
}));
```

Mongoose Queries

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

Mongoose Queries

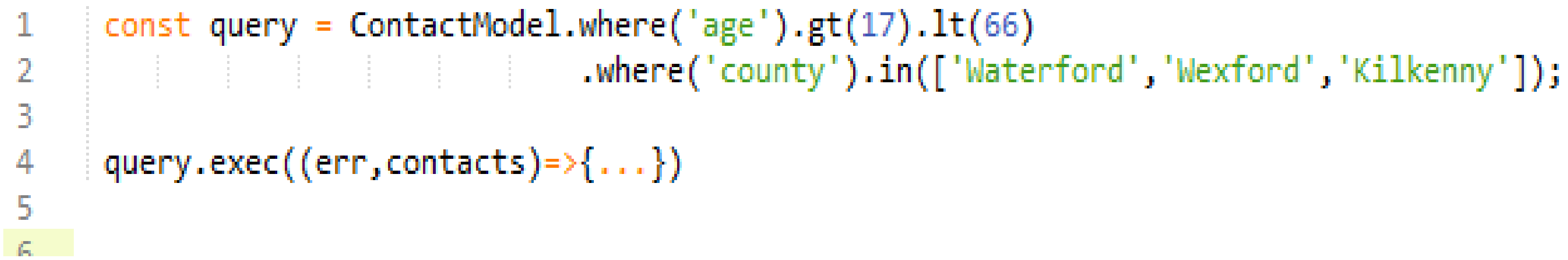
- Mongoose supports many queries:
  - For equality/non-equality
  - Selection of some properties
  - Sorting
  - Limit & skip
- All queries are executed over the object returned by Model.find*()
  - Model.findOne() returns a single document, the first match
  - Model.find() returns all
  - Model.findById()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



```
// Delete a contact
router.delete('/:id', asyncHandler(async (req, res) => {
  const contact = await Contact.findById(req.params.id);
  if (!contact) return res.send(404);
  await contact.remove();
  return res.status(204).send(contact);
}));
```

Mongoose Queries

- Can build complex queries and execute them later

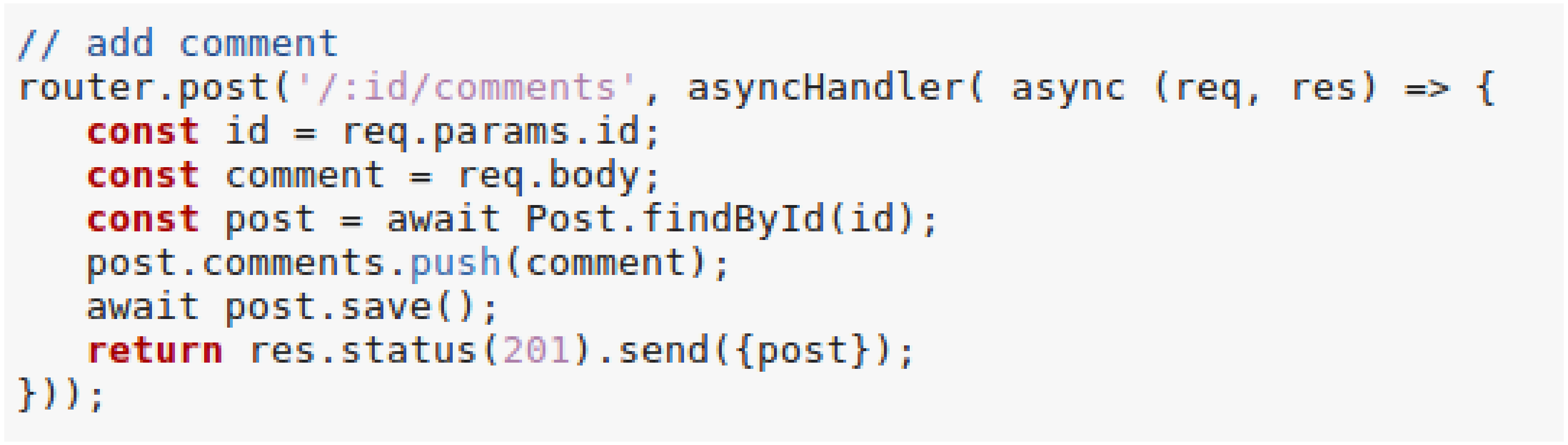


```
const query = ContactModel.where('age').gt(17).lt(66)
                           .where('county').in(['Waterford','Wexford','Kilkenny']);

query.exec((err,contacts)=>{...})
```

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

Mongoose Sub-Docs

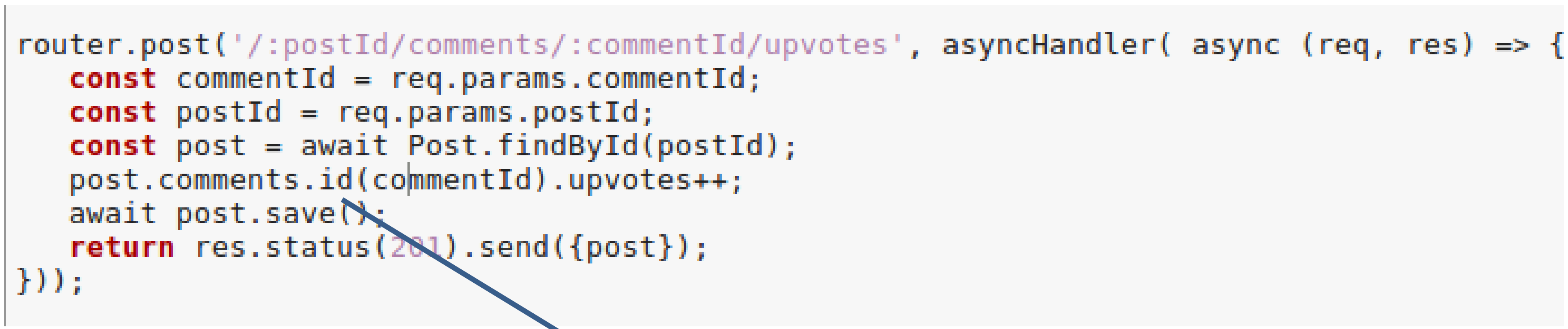
- Ex: Hacker News – Adding a comment to a post.



```
// 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

- Updating a Sub-Document(e.g. incrementing the upvotes for a comment)

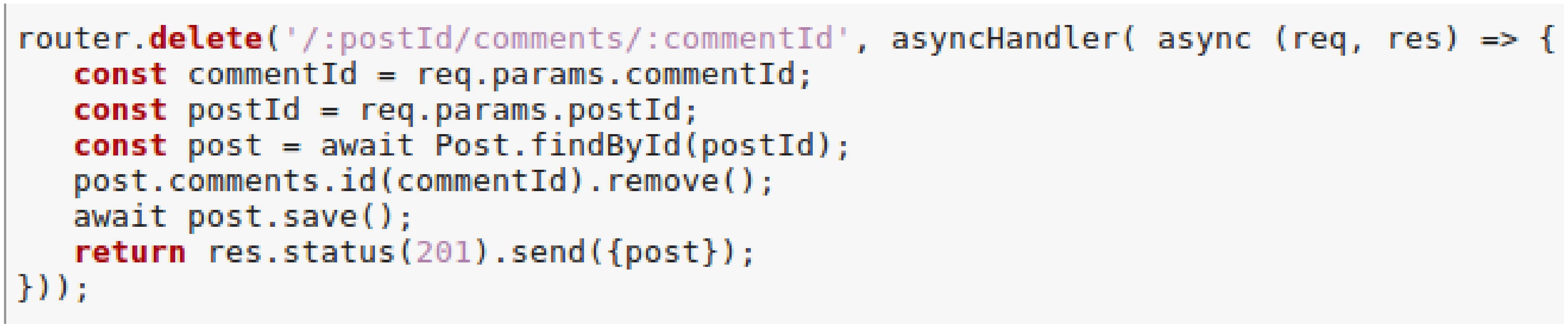


```
router.post('/:postId/comments/:commentId/upvotes', asyncHandler( async (req, res) => {
   const commentId = req.params.commentId;
   const postId = req.params.postId;
   const post = await Post.findById(postId);
   post.comments.id(commentId).upvotes++;
   await post.save();
   return res.status(201).send({post});
}));
```

Each subdocument is assigned it's own _id from MongoDB. This is a special method to access sub documents

Mongo Sub docs

- Removing a sub document

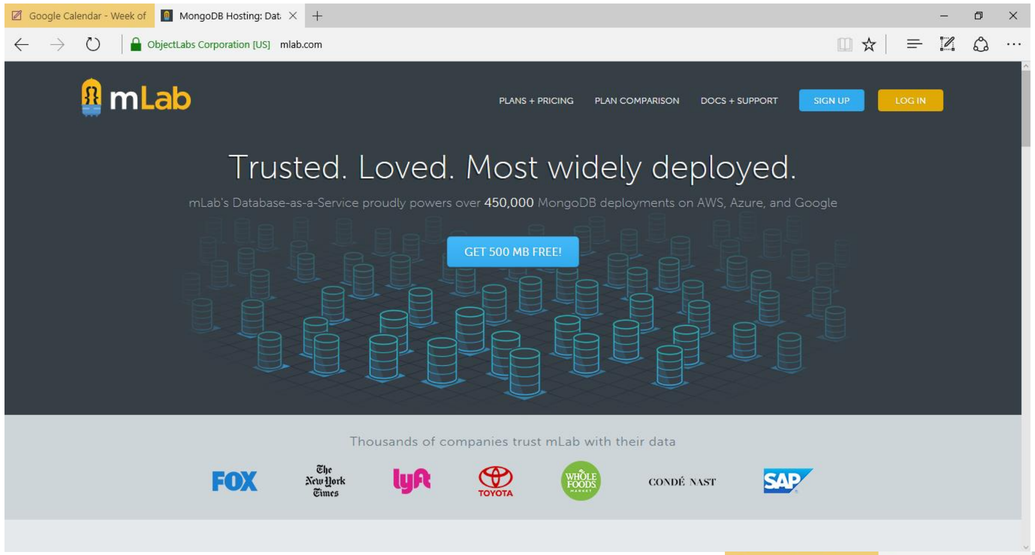


```
router.delete('/:postId/comments/:commentId', asyncHandler( async (req, res) => {
   const commentId = req.params.commentId;
   const postId = req.params.postId;
   const post = await Post.findById(postId);
   post.comments.id(commentId).remove();
   await post.save();
   return res.status(201).send({post});
}));
```

MongoDB as a Service

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

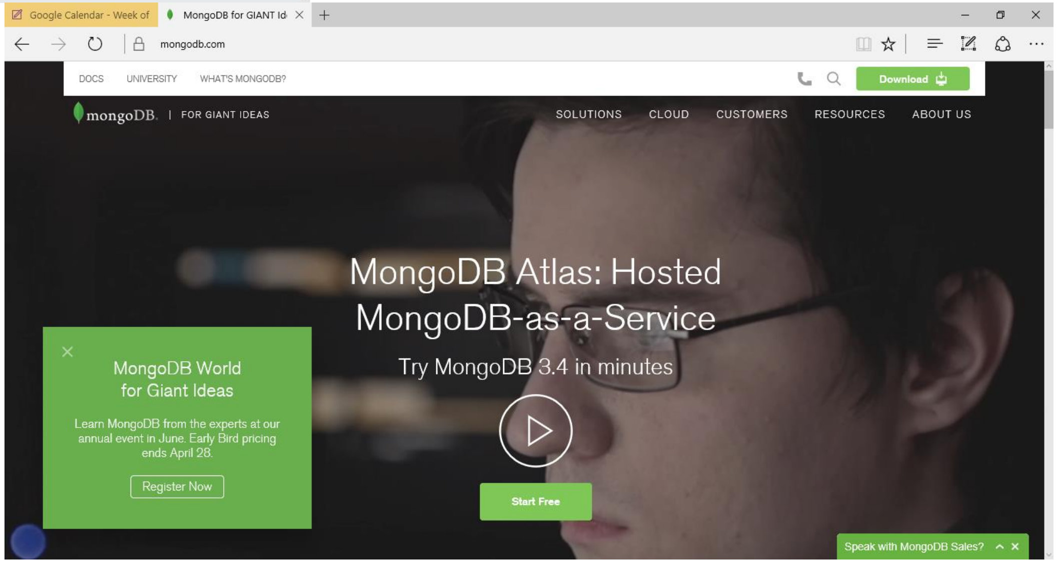
MongoDB as a Service



Trusted. Loved. Most widely deployed.

mLab's Database-as-a-Service proudly powers over **450,000** MongoDB deployments on AWS, Azure, and Google

Thousands of companies trust mLab with their data



MongoDB Atlas: Hosted MongoDB-as-a-Service

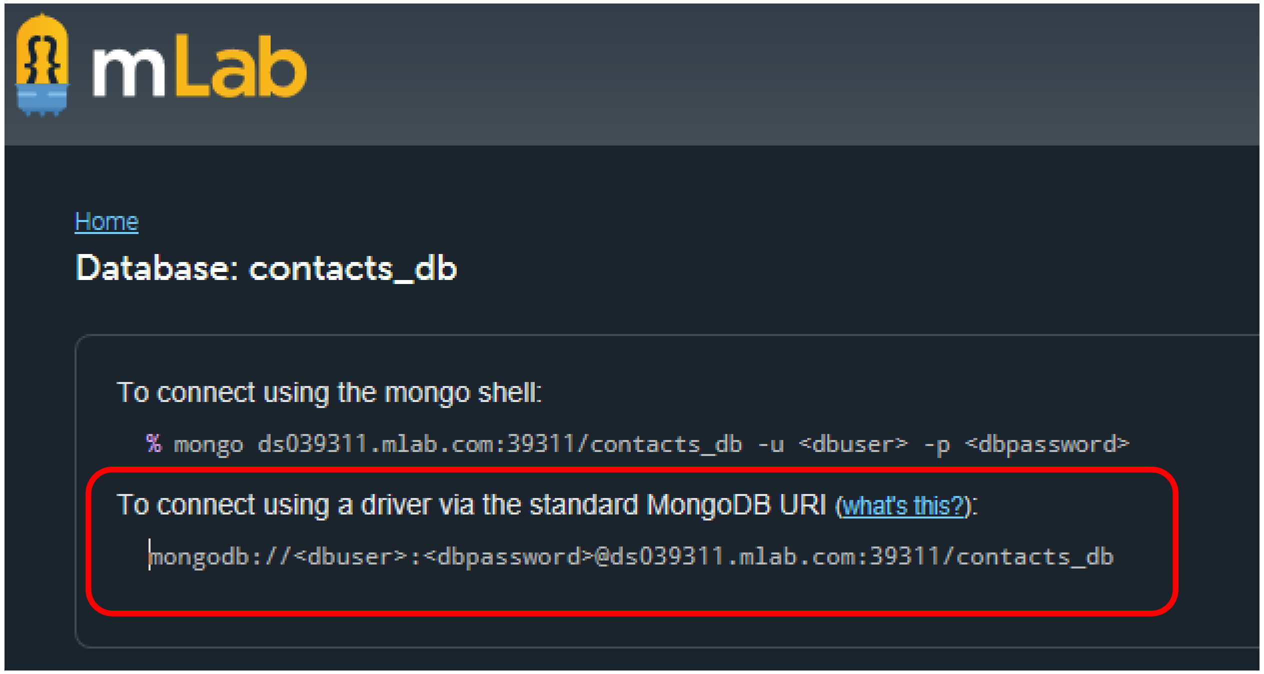
Try MongoDB 3.4 in minutes

MongoDB World for Giant Ideas

Learn MongoDB from the experts at our annual event in June. Early Bird pricing ends April 28.

MongoDB as a Service

- Most providers allow free access teir
- 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



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
```