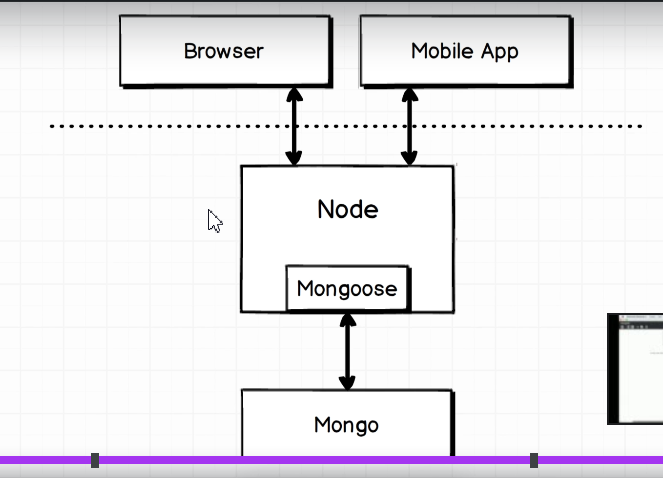
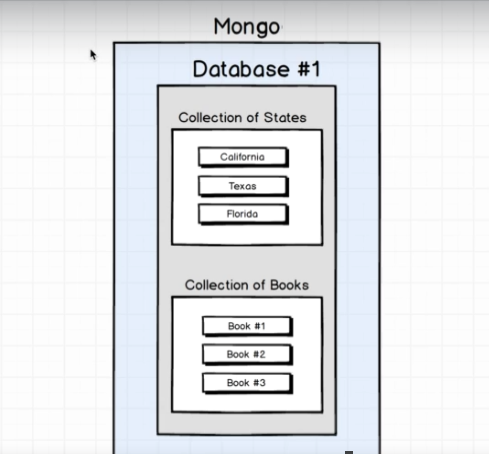
C:\Program Files\MongoDB\Server\5.0\data\

Mongoose 🡪 ORM or ODM





Multiple databases in mongodb – one database for one application

Each databases contains multiple collections

Create , Read, Update, Destroy -> CRUD

User Model has schema , schema contains the type of data the user model handles

When a user model is created which represents the one single object contains mongoose creates a collection in database for us.

const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const UserSchema = new Schema({

  name: String,

});

const User = mongoose.model('user', UserSchema);

const joe = new User({ name: 'Joe' });

    joe.save();

Testing:

beforeEach((done) => {

  mongoose.connection.collections.users.drop(() => {

    done();

  });

});

describe('Creating records', () => {

  it('saves a user', (done) => {

    const joe = new User({ name: 'Joe' });

    joe.save().then(() => {

      assert(!joe.isNew);

      done();

    });

  });

});

Every mongoose has an isNew status which gets false when the object is saved to the database

before((done) => {

  mongoose.connect('mongodb://localhost/users\_test');

  mongoose.connection

    .once('open', () => {

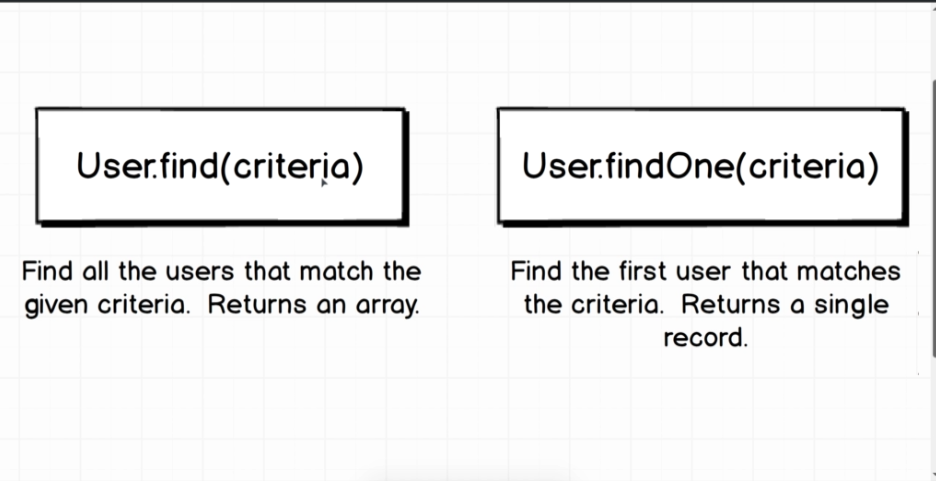
      console.log('Mongoose connection established');

      done();

    })

    .on('error', (err) => console.log(err));

});



 it('finds all users with the name of joe', (done) => {

    User.find({ name: 'Joe' }).then((users) => {

      assert(users[0].\_id.toString() === joe.\_id.toString());

      done();

    });

  });

  it('find a user with a particular id', (done) => {

    User.findOne({ \_id: joe.\_id }).then((user) => {

      assert(user.name === 'Joe');

      done();

    });

  });

Deleting a User

describe('Deleting a user', () => {

  let joe;

  beforeEach((done) => {

    joe = new User({ name: 'Joe' });

    joe.save().then(() => {

      done();

    });

  });

  it('model instance remove', (done) => {

    joe

      .remove()

      .then(() => {

        return User.findOne({ name: 'Joe' });

      })

      .then((user) => {

        assert(user === null);

        done();

      });

  });

  it('class method remove', (done) => {

    User.remove({ name: 'Joe' })

      .then(() => User.findOne({ name: 'Joe' }))

      .then((user) => {

        assert(user === null);

        done();

      });

  });

  it('class method findOneAndRemove', (done) => {

    User.findOneAndRemove({ name: 'Joe' })

      .then(() => User.findOne({ name: 'Joe' }))

      .then((user) => {

        assert(user === null);

        done();

      });

  });

  it('class method findByIdAndRemove', (done) => {

    User.findByIdAndRemove({ \_id: joe.\_id })

      .then(() => User.findOne({ name: 'Joe' }))

      .then((user) => {

        assert(user === null);

        done();

      });

  });

});

Updating the User

describe('Updating records', () => {

  let joe;

  beforeEach((done) => {

    joe = new User({ name: 'Joe' });

    joe.save().then(() => done());

  });

  function assertName(operation, done) {

    operation

      .then(() => {

        return User.find({});

      })

      .then((users) => {

        assert(users.length === 1);

        assert(users[0].name == 'Alex');

        done();

      });

  }

  it('instance type using set and save', (done) => {

    joe.set({ name: 'Alex' });

    assertName(joe.save(), done);

  });

  it('A model instance can Update', (done) => {

    assertName(joe.updateOne({ name: 'Alex' }), done);

  });

  it('Class based update using update', (done) => {

    assertName(User.updateOne({ name: 'Joe' }, { name: 'Alex' }), done);

  });

  it('Class based update using findOne', (done) => {

    assertName(User.findOneAndUpdate({ name: 'Joe' }, { name: 'Alex' }), done);

  });

  it('Class based update using findById and Update', (done) => {

    assertName(User.findByIdAndUpdate(joe.\_id, { name: 'Alex' }), done);

  });

});

Mongo Operators:

Mongo Update Operators:

The following modifiers are available for use in update operations, for example, in [db.collection.updateMany()](https://docs.mongodb.com/manual/reference/method/db.collection.updateMany/#mongodb-method-db.collection.updateMany) and [db.collection.findAndModify()](https://docs.mongodb.com/manual/reference/method/db.collection.findAndModify/#mongodb-method-db.collection.findAndModify).

Allows us to send a instruction to the database which executes it for us, used for bulk operations 🡪 search mongo update operators

  it('A user can have their postcount incremented by 1', () => {

    User.updateMany({ name: 'Joe' }, { $inc: { postCount: 1 } })

      .then(() => {

        return User.findOne({ name: 'Joe' });

      })

      .then((user) => {

        assert(user.postCount === 1);

        done();

      });

  });

VALIDATORS:

const UserSchema = new Schema({

  name: {

    type: String,

    validate: {

      validator: (name) => {

        return name.length > 2;

      },

      message: 'Name must be longer than 2 characters',

    },

    required: [true, 'Name is required'],

  },

  postCount: Number,

});

describe('Validating records', () => {

  it('requires a user name', () => {

    const user = new User({ name: undefined });

    const validationResult = user.validateSync();

    const message = validationResult.errors.name.message;

    assert(message === 'Name is required');

  });

  it("requires a user's name longer than 2 characters", () => {

    const user = new User({ name: 'Jo' });

    const validationResult = user.validateSync();

    const message = validationResult.errors.name.message;

    assert(message === 'Name must be longer than 2 characters');

  });

  it('disallows invalid records from being saved', (done) => {

    const user = new User({ name: 'Al' });

    user.save().catch((validationResult) => {

      const message = validationResult.errors.name.message;

      assert(message === 'Name must be longer than 2 characters');

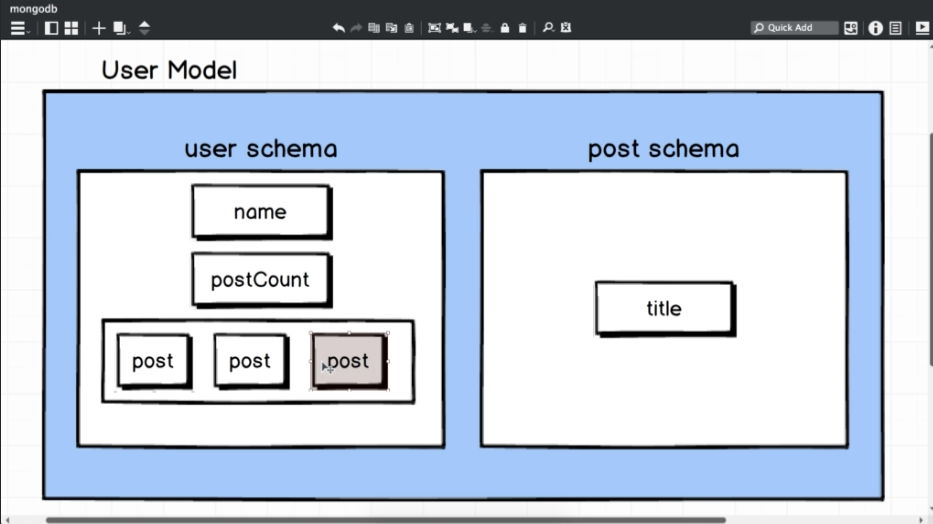
      done();

    });

  });

});

Handling Relational Data:



Here there will be no post model only user model with collections of posts will be created.

The post that we are going to embed inside of it is referred to as a sub document

const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const PostSchema = new Schema({

  title: String,

});

module.exports = PostSchema;

const UserSchema = new Schema({

  name: {

    type: String,

    validate: {

      validator: (name) => {

        return name.length > 2;

      },

      message: 'Name must be longer than 2 characters',

    },

    required: [true, 'Name is required'],

  },

  postCount: Number,

  posts: [PostSchema],

});

const User = mongoose.model('user', UserSchema);

Adding and updating existing subdocuments

describe('Subdocuments', () => {

  it('can create a subdocument', (done) => {

    const joe = new User({

      name: 'Joe',

      posts: [{ title: 'Javascript' }, { title: 'Python' }],

    });

    joe

      .save()

      .then(() => {

        return User.findOne({ name: 'Joe' });

      })

      .then((user) => {

        assert(user.posts.length === 2);

        assert(user.posts[0].title === 'Javascript');

        assert(user.posts[1].title === 'Python');

        done();

      });

  });

  it('can add subdocuments to an existing record', (done) => {

    const joe = new User({

      name: 'Joe',

      posts: [],

    });

    joe

      .save()

      .then(() => {

        return User.findOne({ name: 'Joe' });

      })

      .then((user) => {

        user.posts.push({ title: 'New Posts' });

        return user.save();

      })

      .then(() => {

        return User.findOne({ name: 'Joe' });

      })

      .then((user) => {

        assert(user.posts[0].title === 'New Posts');

        done();

      });

  });

  it('can remove subdouments from an existing record', (done) => {

    const joe = new User({

      name: 'Joe',

      posts: [{ title: 'New Posts' }],

    });

    joe

      .save()

      .then(() => {

        return User.findOne({ name: 'Joe' });

      })

      .then((user) => {

        // removing an alternative mongoose API

        user.posts[0].remove();

        return user.save();

      })

      .then(() => {

        return User.findOne({ name: 'Joe' });

      })

      .then((user) => {

        assert(user.posts.length === 0);

        done();

      });

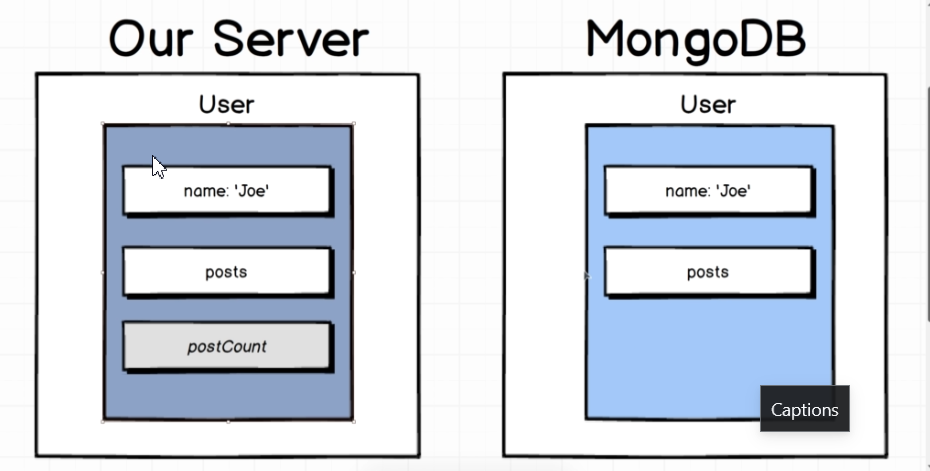
  });

});

Virtual Types:

The field which we have in our schema but are not actually saved in mongoDB database

postCount can be a virtual type



const UserSchema = new Schema({

  name: {

    type: String,

    validate: {

      validator: (name) => {

        return name.length > 2;

      },

      message: 'Name must be longer than 2 characters',

    },

    required: [true, 'Name is required'],

  },

  posts: [PostSchema],

});

UserSchema.virtual('postCount').get(function () {

  return this.posts.length;

}); // function should be specifically used.

const User = mongoose.model('user', UserSchema);

module.exports = User;

Testing Virtual Types:

describe('Virtual Types', () => {

  it('postCount returns number of posts', (done) => {

    const joe = new User({

      name: 'Joe',

      posts: [{ title: 'Javascript' }, { title: 'Python' }],

    });

    joe

      .save()

      .then(() => User.findOne({ name: 'Joe' }))

      .then((user) => {

        assert(user.postCount == 2);

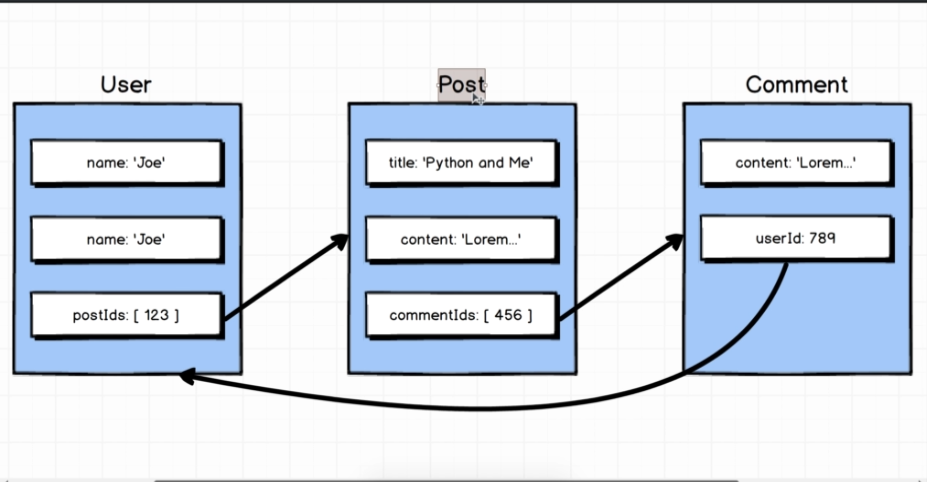
        done();

      });

  });

});

Embedded Documents vs Nested Collections



Association

const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const BlogPostSchema = new Schema({

  title: String,

  content: String,

  comments: [{ type: Schema.Types.ObjectId, ref: 'comment' }],

});

const BlogPost = mongoose.model('blogPost', BlogPostSchema);

module.exports = BlogPost;

const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const CommentSchema = new Schema({

  content: String,

  user: { type: Schema.Types.ObjectId, ref: 'user' },

});

const Comment = mongoose.model('comment', CommentSchema);

module.exports = Comment;

Mongoose handles associations internally, though we add it like a embed object.

beforeEach((done) => {

    joe = new User({ name: 'Joe' });

    blogPost = new BlogPost({ title: 'JS is Great', content: 'lorem epsum' });

    comment = new Comment({ content: 'Congrats on great post' });

    joe.blogPosts.push(blogPost);

    blogPost.comments.push(comment);

    comment.user = joe;

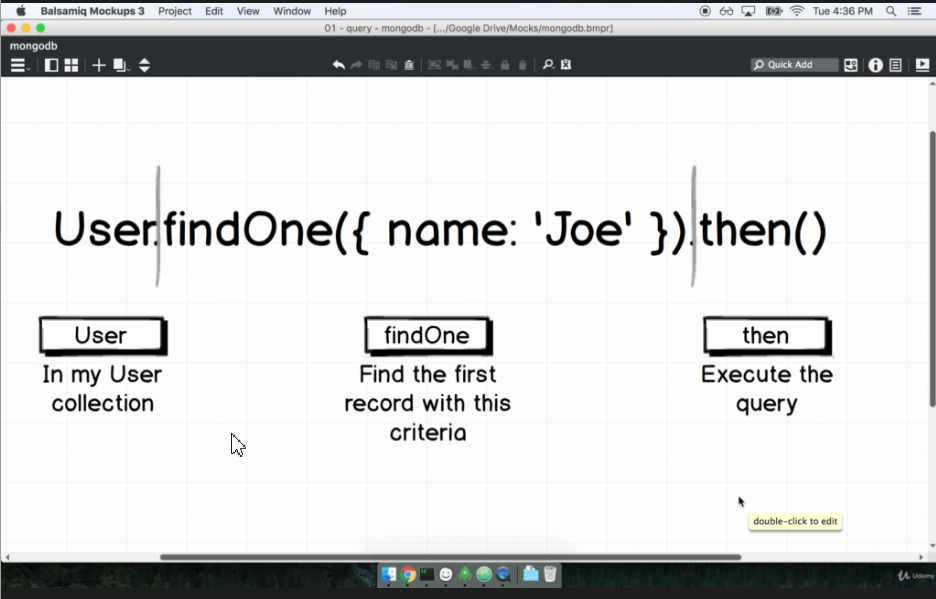
    Promise.all([joe.save(), blogPost.save(), comment.save()]).then(() => {

      done();

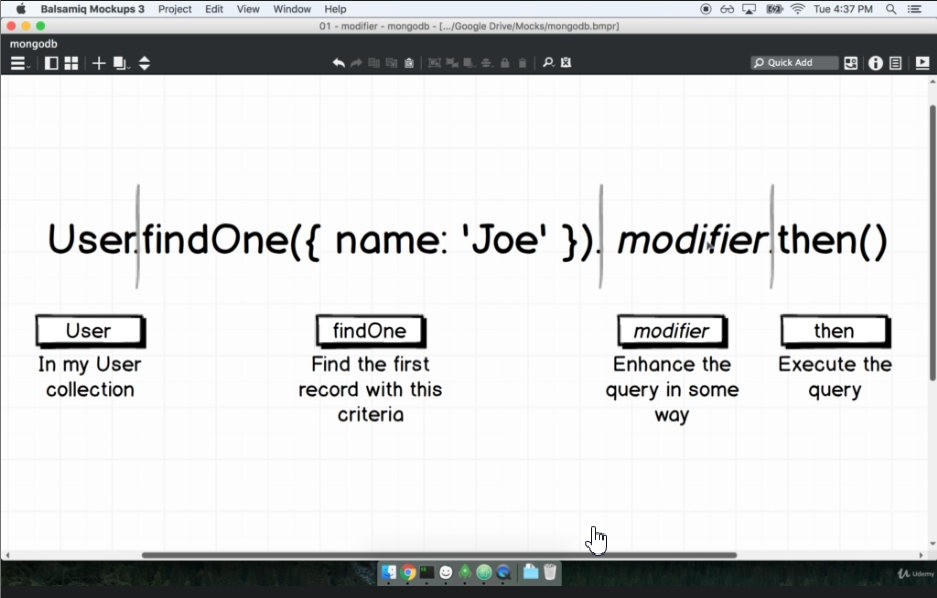
    });

  });

.then 🡪 executes the query



Modifier 🡪 enhances the query in some way



 it('saves a relation between a user and a blogpost', (done) => {

    User.findOne({ name: 'Joe' })

      .populate('blogPosts')

      .then((user) => {

        assert(user.blogPosts[0].title==='JS is Great')

        done();

      });

  });

Loading Deeply nested objects in associations is not available.

 it('saves a full relation graph', (done) => {

    User.findOne({ name: 'Joe' })

      .populate({

        path: 'blogPosts',

        populate: {

          path: 'comments',

          model: 'comment',

          populate: {

            path: 'user',

            model: 'user',

          },

        },

      })

      .then((user) => {

        assert(user.name === 'Joe');

        assert(user.blogPosts[0].title === 'JS is Great');

        assert(

          user.blogPosts[0].comments[0].content === 'Congrats on great post'

        );

        assert(user.blogPosts[0].comments[0].user.name === 'Joe');

        done();

      });

  });

Middleware:

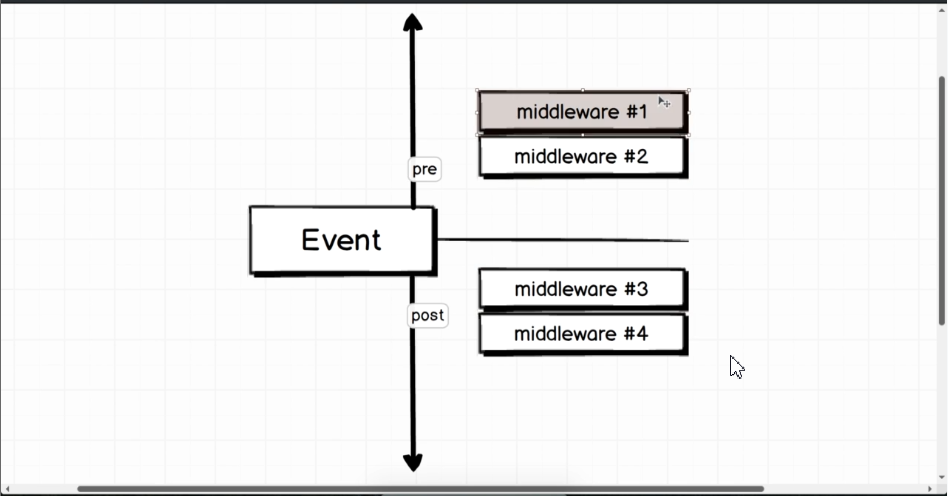
Middleware are post and pre save hooks are used for clean-up purpose

Functions that execute before or after some distinct events in mongoose taken place

One type of event are

1. SAVE
2. VALIDATE
3. REMOVE

All models can have its own sets of middleware



For example , now we are using pre remove to remove all the blogPost data associated with the particular user. For this

1. We should not create cyclic dependency with the help of mongoose
2. We use mongoose query operator to do the removing.

UserSchema.pre('remove', function (next) {

  const BlogPost = mongoose.model('blogPost'); // prevent import of BlogPost which can cause cyclic imports

  BlogPost.remove({ \_id: { $in: this.blogPosts } }).then(() => {

    next(); // for async processing to complete

  }); // this.blogPosts runs in the particular user model.

}); // function should be specifically used instead of arrow function

Testing the middleware

describe('middleware', () => {

  let joe, blogPost;

  beforeEach((done) => {

    joe = new User({ name: 'Joe' });

    blogPost = new BlogPost({ title: 'JS is Great', content: 'lorem epsum' });

    joe.blogPosts.push(blogPost);

    Promise.all([joe.save(), blogPost.save()]).then(() => {

      done();

    });

  });

  it('users clean up dangling blogposts on remove', (done) => {

    joe

      .remove()

      .then(() => {

        return BlogPost.count();

      })

      .then((count) => {

        assert(count === 0);

        done();

      });

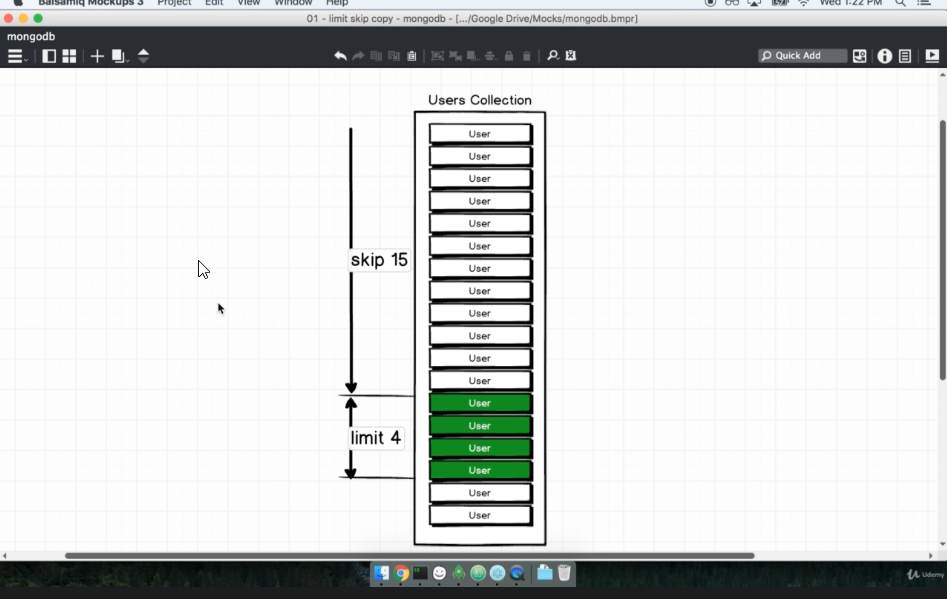
  });

});

Handling Big Collections with Pagination

Two different modifiers used skip and limit

And sort is for sorting



 it('can skip and limit the result set', (done) => {

    User.find({})

      .sort({ name: 'asc' })

      .skip(1)

      .limit(2)

      .then((users) => {

        assert(users.length === 2);

        assert(users[0].name === 'Joe');

        assert(users[1].name === 'Maria');

        done();

      });

  });

  it('can sort the collection result', (done) => {

    User.find({})

      .sort({ name: 'desc' })

      .then((users) => {

        assert(users[0].name == 'Zach');

        done();

      });

  });

In Query

module.exports = (\_id) => {

  Artist.findById(\_id);

};

module.exports = () => {

  const minQuery = Artist.find({})

    .sort({ age: 1 })

    .limit(1)

    .then((artists) => artists[0].age);

  const maxQuery = Artist.find({})

    .sort({ age: -1 })

    .limit(1)

    .then((artists) => artists[0].age);

  return Promise.all([minQuery, maxQuery]).then((result) => {

    return { min: result[0], max: result[1] };

  });

};

Query Operators 🡪 used to build up queries in the mongoDB

Index is a system mongoDB uses to efficiently save and retrieve data, normally mongoDB indexes \_id property to look ups very fast index by using findById

Very frequent look up must be indexed

In cmd:

Mongo

use upstar\_music

db.artists.createIndex({name:’text’})

Mongoose – Update

Model.update 🡪 does not update more than one document until we pass multi to the options

module.exports = (\_ids) => {

  return Artist.update({ \_id: { $in: \_ids } }, { retired: true }, {

    multi:true

  });

};

Count operation 🡪 should be done seperately

module.exports = (criteria, sortProperty, offset = 0, limit = 20) => {

  const query = Artist.find(buildQuery(criteria))

    .sort({ [sortProperty]: 1 })

    .skip(offset)

    .limit(limit);

  return Promise.all([query, Artist.find(buildQuery(criteria)).count()]).then(

    (results) => {

      return {

        all: results[0],

        count: results[1],

        offset: offset,

        limit: limit,

      };

    }

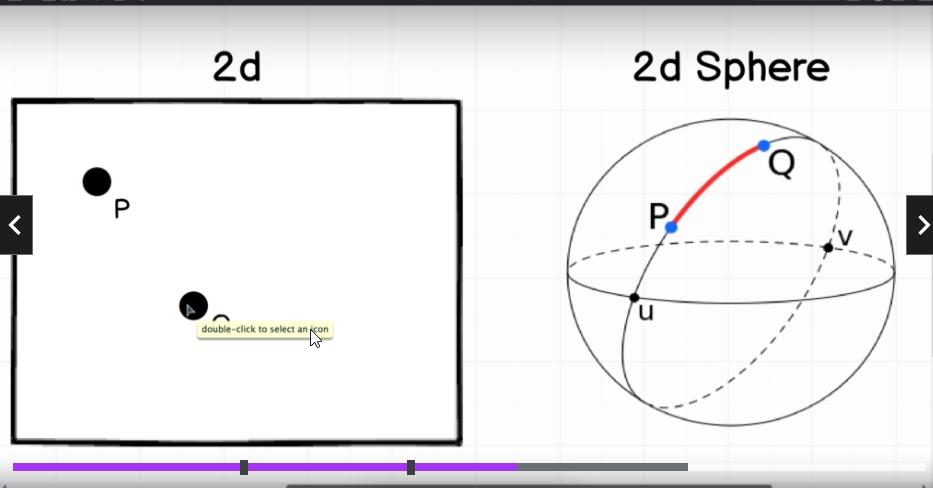
  );

};

Geography in mongoDB

Uses lat and lon

Two systems available



GeoJSON Schema:

geojson.org

const PointSchema = new Schema({

  type: { type: String, default: 'Point' },

  coordinates: { type: [Number], index: '2dsphere' },

});

const DriverSchema = new Schema({

  geometry: PointSchema,

});

Driver.geoNear(

  { type: 'Point', coordinates: [lng, lat] },

  { spherical: true, maxDistance: 20000 }

);

db.Employees.find({}).sort({"Emp salary":-1}).limit(1) //for first highest salary

db.Employees.find({}).sort({"Emp salary":-1}).skip(1).limit(1)