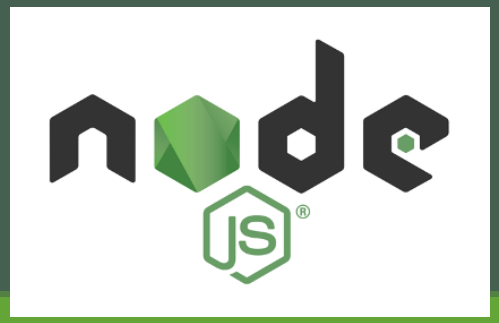

mongoose

elegant mongodb object modeling for node.js

Contenido

- ODMs
- Instalación mongoose

OBJECT DATA MAPPING

- Data Mapper es un patrón de diseño utilizado para separar la capa de acceso a la base de datos de la capa de dominio que contiene una representación de los datos
- La capa de separación puede estar compuestas de uno más *mappers* (o Data Access Objects), que llevan a cabo la transferencia de los datos
- La implementación del patrón en aplicaciones con bases de datos SQL se denomina ORM.
- Con bases de datos NoSQL se conoce como ODM.

ODM vs ORM

- Data Mapper es un patrón de diseño utilizado para poder independizar la capa de acceso a la base de datos de la capa de dominio que contiene una representación de los datos normalmente en forma de objetos
- La capa de separación puede estar compuestas de uno más *mappers* (o Data Access Objects), que llevan a cabo la transferencia de los datos
- La implementación del patrón en aplicaciones con bases de datos SQL se denomina ORM.
- Con bases de datos NoSQL se conoce como ODM.

MONGOOSE

- Es un framework javascript que permite el acceso a una base de datos MongoDB usando un modelo de objetos
- Una **db** contiene muchas **colecciones** y una colección muchos **documentos**. No existe ningun requerimiento que obligue a que dos documentos de una misma colección tengan la misma estructura. Aunque normalmente se establece que dos documentos almacenados en la misma colección tendrán una estructura parecida
- En Mongoose se definen los **esquemas de documentos**; Cada esquema puede contener una lista de campos y sus restricciones:
  - Tipo de dato y restricciones específicas tales como: valor mínimo, valor requerido, único,...
- Un **modelo** representa una conexión a una colección de la bd que usa el esquema anterior.

MONGOOSE: Instalación y conexión a BD

- para instalar Mongoose usamos:

```
$ npm install mongoose
```

- para conectar a MongoDB desde Mongoose se usa el método **connect()** al que se le pasan como argumentos la URL de la bd y los parámetros de conexión: username, password, server host, database name, etc.
- Ej.

```
const mongoose = require("mongoose");
mongoose.connect("mongodb://username:password@server_host/database_name");
```

- Una vez que se ha establecido la conexión, se puede invocar al objeto **connection** que emite el evento **open**. La conexión puede terminar con el método **close()**

```
const connection = mongoose.connection;
connection.once("open", function( ) {console.log("Connection established");
connection.close( );
});
```

MONGOOSE: Primera aplicación

- Ejemplo de Código de la página oficial: http://mongoosejs.com.

```
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost:27017/test');

var Cat = mongoose.model('Cat', { name: String });

var kitty = new Cat({ name: 'Zilda' });
kitty.save(err => {
if (err) // ...
console.log('meow');
});
```

```
const mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/test');
const Cat = mongoose.model('Cat', { name: String });
const kitty = new Cat({ name: 'Zildjian' });
kitty.save().then(() => console.log('meow'));
```

MONGOOSE: SCHEMAS

- Los esquemas se utilizan para definir la estructura y los atributos de un documento. Cada esquema mapea a una colección de la bd y define la "forma" del documento dentro de la colección.
- Para crear un esquema determinado se genera una instancia del objeto Schema
- Ej: Esquema de dos colecciones: estudiantes y proyectos

```
//db/schema.js
var Schema = mongoose.Schema
var StudentSchema = new Schema({
  name: String,
  age: Number
});
var ProjectSchema = new Schema({
  title: String,
  unit: String
});
```

MONGOOSE: SUBDOCUMENTOS

- En el caso de querer almacenar información referente a una relación entre objetos, se almacenaría como un subdocumento.
- Existen dos tipos diferentes de subdocumentos:
  - arrays de subdocumentos
  - subdocumento anidado simple (a partir de mongoose 4.2.0)
- Ej:

```
//db/schema.js

var Schema = mongoose.Schema
var projectSchema = new Schema({
  title: String,
  unit: String
});
var studentSchema = new Schema({
  name: String,
  age: Number
  projects:  [projectSchema]
});
```

MONGOOSE: MODELS

- Los Modelos son los constructors compilados a partir de las definiciones de esquemas
- Una instancia del modelo se llama documento.
- Los modelos son los responsables de crear y leer documentos en y desde la bd.
- Ej:

```
//db/schema.js
...
let Student = mongoose.model("Student", StudentSchema);
let Project = mongoose.model("Project", ProjectSchema)
```

El primer argumento es el nombre en **singular** de la colección. Mongoose automaticamente busca el nombre en plural. Por tanto, en el ejemplo las colecciones referidas en la bd serían: students y projects

MONGOOSE: INSERT DATA

- Insertar datos en MongoDB es un proceso de dos pasos que utiliza los modelos Mongoose:
  - El primer paso es instanciar un objeto usando el constructor de un modelo. En el ejemplo anterior el constructor es Student( ). Una vez creado se puede manipular como cualquier objeto JavaScript.
  - Para insertar los datos se utiliza el método **save( )** del modelo, que recibe una función callback con el parámetro para tratar el **error**.

```
//db/schema.js

…
let ana = new Student({
                name: 'ana'
})
let project1 = new Project({
                title: 'Mongoose'
})
ana.projects.push(project1);
ana.save((err,student)=>{
        if(err) { console.log(err)}
        else {console.log(student+' se guardó en la bd!!!')}
})
```

MONGOOSE: PROMESAS

- Todos los métodos de Mongoose están preparados para trabajar con callbacks o con promesas. Ej:

```
//db/schema.js

…
let ana = new Student({
                name: 'ana'
})
let project1 = new Project({
                title: 'Mongoose'
})
ana.projects.push(project1);
anna.save().then(student => {  console.log(student)})
                .catch(err => {console.log(err)});
```

MONGOOSE: QUERY DATA

- Mongoose provee de una serie de consultas predefinidas:
  - Model.deleteMany()
  - Model.deleteOne()
  - Model.find()
  - Model.findById()
  - Model.findByIdAndDelete()
  - Model.findByIdAndRemove()
  - Model.findByIdAndUpdate()
  - Model.findOne()
  - Model.findOneAndDelete()
  - Model.findOneAndRemove()
  - Model.findOneAndUpdate()
  - Model.replaceOne()
  - Model.updateMany()
  - Model.updateOne()

- Todos los métodos pueden usarse pasando un callbacks con los parámetros de error y resultado o como promesa resolviendo el cumplimiento de la promesa a través del método .then()

MONGOOSE: QUERY DATA

- Mongoose provee de una serie de consultas predefinidas:
  - Model.find({}, callback) -- Busca todas las ocurrencias
  - Model.findById(someId, callback) -- Busca un único modelo por id
  - Model.findOne({someKey: someValue}, callback) -- Busca un único modelo por clave:valor
  - Model.remove({someKey: someValue}, callback) -- Elimina todos los modelos que cumplen la condición
- Todos los métodos pueden usarse pasando un callbacks con los parámetros de error y resultado o como promesa resolviendo el cumplimiento de la promesa a través del método .then()

- Ej.: Implementando los métodos dentro de un controlador: controllers/studentsController.js

```
const Schema = require("../db/schema.js");
const Student = Schema.Student;
const Project = Schema.Project;
let studentsController = {
  index(){
    Student.find({}, (err, students) => {
      console.log(students);
    });
  }
};
studentsController.index();
```

Ejercicio 22

- Implementa el resto de métodos descritos, sobre el ejemplo: show, update y delete.
- Implementa un método que devuelva el número de estudiantes o proyectos
- Implementa un método para borrar todos los proyectos de un estudiante

Ejercicio 23

- Crea la capa de persistencia del ejemplo de movies (ejercicio 21).

MONGOOSE: VIRTUALS

- Son propiedades de documentos que pueden insertarse y recuperarse de un esquema pero que no se persisten a MongoDB.
- Los **getters** son útiles para formatear o combinar campos.
- Los **setters** son útiles para descomponer valores simples en valores múltiples de cara al almacenamiento.
- Ej:

```
// define a schema
var personSchema = new Schema({ name: { first: String, last: String } });
// compile our model
var Person = mongoose.model('Person', personSchema);
// create a document
var axl = new Person({ name: { first: 'Axl', last: 'Rose' } });

// to handle the full_name we do a getter function:
personSchema.virtual('fullName').get(function () {
        return this.name.first + ' ' + this.name.last; });

// now mongoose will call the getter function each time an access to fullname
is set
console.log(axl.fullName);
```

MONGOOSE: VIRTUALS

- Ej getter y setter en una misma función sobre un mismo campo virtual:

```
personSchema.virtual('fullName')
      .get(function() {
            return this.name.first + ' ' + this.name.last; })
      .set(function(v){
            this.name.first = v.substr(0, v.indexOf(' '));
            this.name.last = v.substr(v.indexOf(' ') + 1); });

axl.fullName = 'William Rose';
// Now `axl.name.first` is "William"
```

MONGOOSE: VIRTUALS

- **Alias:** Virtual cuyos getter y setter obtienen y devuelven otra propiedad.
- Es útil para ahorrar ancho de banda, se pueden convertir nombres cortos de propiedades en nombres largos para facilitar la lectura del Código
- **Ej:**

```
var personSchema = new Schema({
    n: { type: String,
         alias: 'name'
        }
});

var person = new Person({ name: 'Val' });
console.log(person); // { n: 'Val' }
console.log(person.toObject({ virtuals: true })); // { n: 'Val', name: 'Val' }
console.log(person.name); // "Val"
person.name = 'Not Val';
console.log(person); // { n: 'Not Val' }
```

MONGOOSE: VALIDATORS

- Mogoose ofrece validadores a nivel de campo de un documento, que se ejecutan en el momento de guardar el documento.
- Si ocurre un error de validación, la operación de guardado se interrumpe y el error pasa al callback.

- Existen validadores predefinidos y opciones para crear validaciones personalizadas:

- Validadores predefinidos:
- Todos los tipos de datos: **Required**
- Numbers: **min** y **max**
- Strings: **enum**, **match**, **minlength** y **maxlength**

- Ej:

```
var UserSchema = new Schema({
  ...
  username: {
    type: String,
    *unique: true,
    required: true
  }
});
```

**unique no es un validador como tal. Es un helper para la construcción de índices. Es decir, indicando unique obligamos a construir un índice sobre el campo*

MONGOOSE: VALIDATORS

- **required:** valida la existencia del campo. Evita que se guarde un documento sin este campo.
- **unique:** Valida que el valor a guardar sea único en la colección.
- **match:** Valida que el valor a guardar tenga un determinado patrón
- **enum:** Obliga a que el valor sea uno de los definidos en una colección de strings.
- Ej.:

```
var UserSchema = new Schema({
  username: {
    type: String,
    unique: true,
    required: [true, 'nombre obligatorio']
  },
  email: {
    type: String,
    match: /.+\@.+\..+/
  },
  role: {
    type: String,
    enum: ['Admin', 'Owner', 'User']
  },
});
```

MONGOOSE: VALIDATORS

- Podemos definir **validaciones personalizadas** mediante la propiedad **validate.** A la propiedad se le pasa un array con una función validadora:

```
var UserSchema = new Schema({
  ...
  password: {
    type: String,
    validate: {
      function(password) {
        return password.length >= 6;
      },
      'Password should be longer'
    }
  },
});
```

MONGOOSE: MIDDLEWARE

- También conocidos como *hooks pre y post*, son funciones que toman el control durante la ejecución de funciones asíncronas.
- Existen dos tipos: pre middleware y post middleware.
- Se define a nivel de esquema y puede modificar la consulta o el documento mismo al ser ejecutado.
- Mongoose presenta 4 tipos de middleware:
  - Documento. En las funciones middleware de documentso: **this** se refiere al documento
    - incorpora middleware en las siguientes funciones: validate, save, remove, init (síncrono)
  - Query
    - count, find(findOne, findOneAndRemove,findOneAndUpdate), remove update (updateOne,updateMany)
  - Aggregate
  - Model
    - insertMany

MONGOOSE: PRE middleware

- El Pre middleware se ejecuta antes de que suceda la operación
- Por ejemplo, un middleware pre-save sera ejecutado antes de salvar el documento. Esta funcionalidad es útil para validaciones más complejas, asignación de valores por defecto o para eliminar documentos dependientes (eliminando un usuario eliminamos sus posts)
- Las funciones Pre middleware se ejecutan una detrás de otra cuando cada función invoca el método **next()**.

```
var schema = new Schema(..);
schema.pre('save', function(next) {
 // do stuff
next();
});
```

- En Mongoose 5 se puede usar una función que retorne una promesa. En concreto se puede usar async/await

```
schema.pre('save', async function() {

   await doStuff();
   await doMoreStuff();
   });
```

MONGOOSE: POST middleware

- El Post middleware se ejecuta después de la operación
- El post-save middleware se ejecutará después de grabar en la bd.
- Las funciones Post middleware si llevan dos parámetros el segundo será el método **next()** que llamará al siguiente post middleware
- Ej:

```
schema.post('save', function(doc) {
 console.log('%s has been saved', doc._id); });
```

Ejercicio 24

- Sobre el ejercicio 22 (students y projects) crear un middleware PRE que convierta el campo **name** a mayúsculas antes de guardarlo en la bd
- Generar una función de log con un Post middleware que recoja los registros borrados de la aplicación

MONGOOSE: DBRef

- En MongoDB podemos definir referencias a otras colecciones incluyendo el _id de la colección referenciada y la propiedad DBRef.
- En los schemas de Mongoose la relación se establece de manera análoga utilizando una propiedad que contenga un array de propiedades: {type: Schema.ObjectId, ref:'Nombre_colección' }
- Ej:

```
var mongoose = require('mongoose')
, Schema = mongoose.Schema

var PersonSchema = new Schema({
name : String
, age : Number
, stories : [{ type: Schema.ObjectId, ref: 'Story' }]
});

var StorySchema = new Schema({
_creator : { type: Schema.ObjectId, ref: 'Person' }
, title : String
, fans : [{ type: Schema.ObjectId, ref: 'Person' }]
});

var Story = mongoose.model('Story', StorySchema);
var Person = mongoose.model('Person', PersonSchema);
```

MONGOOSE: DBRef

- Guardando documentos en la bd:

```
let aaron = new Person({ name: 'Aaron', age: 100 });

aaron.save(function (err) {
    if (err) //do something

    let story1 = new Story({
    title: "A man who cooked Nintendo"
    , _creator: aaron._id
    });

    story1.save(function (err) {
        if (err) console.err
    });
})
```

- En Mongoose 5, podemos manejar async/await en lugar de callbacks para manejar la creación de registros.

MONGOOSE: Populate

- Para recuperar documentos referenciados utilizamos el método populate para incorporar el contenido del documento hijo al documento padre.
- En este caso se utiliza la función de callback .exec() para ejecutar la función.

```
Story
    .findOne({ title: /Nintendo/i })
    .populate('_creator') // <--
    .exec(function (err, story) {
        if (err) //lo que sea
        console.log('The creator is %s', story._creator.name);
        // prints "The creator is Aaron"
})
```

- Para recuperar sólo ciertos campos utilizar un array de propiedades a mostrar dentro de la función populate:

```
.populate('_creator', ['name']) // <-- only return the Persons name
```

Ejercicio 25

- Rehacer la función create del ejercicio 22 para incorporar la creación de proyectos dentro de la colección proyectos y después incorporarlos por referencia a la colección students. Para ello tienes que seguir los siguientes pasos:
  - Rehacer el studentSchema incorporando el _id del projectSchema por referencia en una propiedad projects.
  - Convertir la función créate en async y rehacer la lógica de la función para guardar proyectos en la colección projects y students que referencien los projects.
  - Devolver el documento students con el contenido del documento projects.

MONGOOSE: DISCRIMINATORS

- Es un mecanismo de herencia de esquemas
- Se utilizan para almacenar documentos similares en la misma colección pero con diferentes restricciones de esquema
- Se define un discriminador sobre el esquema base o padre:

```
const options = {discriminatorKey: 'itemtype'};
```

- En la instanciación del objeto Schema se pasa el array de opciones:

```
const baseSchema = new Schema({
title: { type: String, required: true },
date_added: { type: Date, required: true },
redo: { type: Boolean, required: false },
},options)
```

- Se crea el modelo para el esquema base:

```
const Base = mongoose.model('item',baseSchema);
```

MONGOOSE: DISCRIMINATORS

- Para generar los esquemas hijos debemos definir el campo discriminatorio mediante la función:
  ModeloBase.discriminator(schemaName,schema,discriminatorPropValue)
- Ejemplo:

```
const Book = Base.discriminator('Book', new
mongoose.Schema({
author: { type: String, required: true },
}),
);
const Movie = Base.discriminator('Movie', new
mongoose.Schema({
director: { type: String, required: true },
}),
);
const Tvshow = Base.discriminator('Tvshow', new
mongoose.Schema({
season: { type: Number, required: true },
}),
);
```

MONGOOSE: DISCRIMINATORS

- Para generar los esquemas hijos debemos definir el campo discriminatorio mediante la función: **ModeloBase.discriminator(schemaName,schema,discriminatorPropValue)**
- Ejemplo:

```
const Book = Base.discriminator('Book', new mongoose.Schema({
author: { type: String, required:true },}), 'book');
const Movie = Base.discriminator('Movie', new mongoose.Schema({
director: { type: String, required: true },}), 'movie');
const Tvshow = Base.discriminator('Tvshow', new mongoose.Schema({
season: { type: Number, required: true },}), 'tvshow');
```

- Ej.: Generamos los modelos hijos:

```
BookMod = mongoose.model('Book');
MovieMod = mongoose.model('Movie');
TvShowMod = mongoose.model('Tvshow');
```

MONGOOSE: DISCRIMINATORS

- Para persistir los datos usamos la función créate sobre cada modelo hijo:

```
const bookdocs = [{title: 'The castle', author: 'F.Kafka', date_added: Date.now()}];

const mooviedocs = [{title: 'Matrix', director: 'Wachowsky Bros.', date_added: Date.now()}];

const tvdocs = [{title: 'Games of Thrones', season: 1, date_added: Date.now()}];

BookMod.create(bookdocs);
MovieMod.create(mooviedocs);
TvShowMod.create(tvdocs);
```

- Podemos obtener el listado completo mediante el modelo base:

```
const docs = await Base.find();
```

Ejercicio 26

- En nuestro ejercicio de estudiantes y proyectos queremos tener una única colección de personas en las que tengamos estudiantes con sus proyectos y añadiremos también documentos de profesores (teachers).
- Los estudiantes y profesores compartirán el campo name.
- Los estudiantes mantendrán su relación de proyectos.
- Los profesores tendrán un campo subject para almacenar la asignatura que imparten
- El campo discriminatorio será el **role**
- Modificar el método create para que almacene profesores o estudiantes según venga informado el campo role en la petición.