

Tema 4

MongoDB y Mongoose

¿Qué es MongoDB?

- Es un sistema de base de datos NoSQL open-source escrito en C++.
- MongoDB es orientado a documentos. Esto quiere decir que en lugar de guardar los datos en registros, guarda los datos en documentos. Estos documentos son almacenados en BSON, que es una representación binaria de JSON.
- Una de las diferencias más importantes con respecto a las bases de datos relacionales, es que no es necesario seguir un esquema. Los documentos de una misma colección, pueden tener esquemas diferentes.

Características principales

- Documentos tipo JSON con esquema dinámico
- Queries con formato de documento
- Indexación de los campos de un documento
- Replicación maestro-esclavo
- Balanceo de carga y escalabilidad horizontal
- Almacenamiento de ficheros mediante “GridFS”

Operaciones - Insertar

- Insertar un documento en una colección:

```
>db.personas.insert({  
  nombre      : 'Mario',  
  apellido    : 'Girón',  
  cursos      : ['Django' , 'Node.js']  
});
```

- Desde la consola de mongo también podemos definir variables y después manipularlas:

```
> var persona = {  
  nombre      : 'Mario',  
  apellido    : 'Girón',  
  cursos      : ['Django' , 'Node.js']  
};
```

```
> db.personas.insert(persona);
```

Operaciones - Consultar

- Con el comando **show collections** podemos visualizar todas las colecciones que tenemos.
- Para lanzar consultas sobre la base de datos empleamos el método find. Si no especificamos ningún parámetro recuperaremos todos los documentos de la colección.

```
>db.personas.find()
```

```
{
  "_id" : ObjectId("54c1093161d622d5fb17c2f7"),
  "nombre" : "Mario",
  "apellido" : "Girón",
  "cursos" : [
    "Django",
    "Node.js"
  ]
}
```

Operaciones - Consultar

- Podemos añadir filtros a nuestras consultas para obtener los documentos que deseamos.
- El primer parámetro del método find es el filtro o criteria. El segundo parámetro es la proyección que nos permite especificar que campos queremos devolver. Si omitimos la proyección devolverá los documentos enteros.

```
>db.micoleccion.find( { _id: 5 } )
```

- Esta consulta devuelve los documentos cuyo campo _id sea igual a 5.
- Podemos añadir varias condiciones de igualdad separándolas por comas.

Operaciones - Consultar

- Existen operadores que nos permiten lanzar consultas más precisas.
- Estos operadores van precedidos del símbolo “\$”.
- Operadores de comparación:

Nombre	Descripción
\$gt	Coincide con los valores que son mayores que el valor especificado en la consulta.
\$gte	Coincide con los valores que son mayores que o igual al valor especificado en la consulta.
\$in	Coincide con cualquiera de los valores que existen en una matriz especificada en la consulta.
\$lt	Coincide con valores menores que el valor especificado en la consulta.
\$lte	Coincide con valores que son de menos de o igual que el valor especificado en la consulta.
\$ne	Coincide con todos los valores que no son iguales al valor especificado en la consulta.
\$nin	Coincide con valores que no existen en una matriz especificada a la consulta.

Operaciones - Consultar

- Existen operadores que nos permiten lanzar consultas más precisas.
- Estos operadores van precedidos del símbolo “\$”.
- Operadores lógicos:

Nombre	Descripción
\$and	Devuelve todos los documentos que coinciden con las condiciones de ambas cláusulas.
\$not	Devuelve los documentos que no coinciden con la expresión de consulta.
\$nor	Devuelve todos los documentos que carezcan de igualar ambas cláusulas.
\$or	Devuelve todos los documentos que coinciden con las condiciones de cualquiera cláusula.

- Existen más operadores : <http://docs.mongodb.org/manual/reference/operator/query/>

Ejemplos de consultas

- La primera consulta devuelve productos cuyo precio sea mayor de 25.
- La segunda devuelve aquellos documentos cuyo campo field tenga un valor dentro del rango (value1,value2).
- La tercera devuelve los documentos cuyo campo _id coincida con alguno de los elementos en el array

```
db.products.find( { precio: { $gt: 25 } } )
```

```
db.collection.find( { field: { $gt: value1, $lt: value2 } } );
```

```
db.bios.find(  
  {  
    _id: { $in: [ 5, ObjectId("507c35dd8fada716c89d0013") ] }  
  }  
)
```

Consultas con Arrays

- Si un campo contiene un array y la consulta tiene varios operadores condicionales, el campo hará matching si un solo elemento o una combinación de elementos del array reúnen las condiciones.

```
{ "_id" : 1 , "puntuación" : [ - 1 , 3 ] }  
{ "_id" : 2 , "puntuación" : [ 1 , 5 ] }  
{ "_id" : 3 , "puntuación" : [ 5 , 5 ] }
```

```
db.jugadores.find ( { puntuacion : { $ gt : 0 , $ lt : 2 } } )
```

```
{ "_id" : 1 , "puntuación" : [ - 1 , 3 ] }  
{ "_id" : 2 , "puntuación" : [ 1 , 5 ] }
```

Consultas con Arrays

- También podemos filtrar dependiendo de los valores que contenga un campo que sea un array.
- La siguiente operación devuelve documentos en la colección de biografías donde el campo array contribuciones contiene el elemento "UNIX" :

```
db.bios.find ( { contribuciones : "UNIX" } )
```

Limitar y Ordenar

- Mediante los métodos limit y sort limitamos y organizamos el número de documentos que aparecen en la respuesta.
- Ejemplo - limitar el número de documentos en la respuesta a los 3 primeros:

```
> db.personas.find().limit(3)
```

- Ejemplo - Ordenar los resultados mediante su campo apellido (1 para descendiente, -1 para ascendente):

```
> db.personas.find().sort({apellido : 1})
```

Operaciones - Actualización

- Para modificar los documentos que ya se encuentran almacenados usaremos el método `.update()`. Presenta la siguiente estructura estructura :

```
db.coleccion.update(  
    filtro,  
    cambio,  
    {  
        upsert: booleano,  
        multi: booleano  
    }  
);
```

- El filtro especifica el/los documentos que queremos modificar igual que si fuera una consulta.
- En el cambio ponemos lo que deseamos modificar teniendo en cuenta que podemos añadir los flags `upsert` y `multi`.
- `upsert true` indica que si no encuentra ningún documento, inserte la modificación como un nuevo documento. `Multi` sirve para modificar varios documentos que hagan matching.

Operaciones - Actualización

- Podemos actualizar el documento entero o determinados campos mediante operadores.

```
db.personas.update(  
  {nombre: 'Mario'},  
  {  
    nombre: 'Mario',  
    apellido: 'Girón',  
    cursos: ['Django', 'Node.js', 'NuevoCurso']  
  }  
);
```

- Operadores de campos “simples”:

\$inc – incrementa en una cantidad numérica el valor del campo deseado.

\$rename – renombrar campos del documento.

\$set – permite especificar los campos que van a ser modificados.

\$unset – eliminar campos del documento.

Operaciones - Actualización

- Operadores de Arrays:

\$pop – elimina el primer o último valor de un array.

\$pull – elimina los valores de un array que cumplan con el filtro indicado.

\$pullAll – elimina los valores especificados de un array.

\$push – agrega un elemento a un array.

\$addToSet – agrega elementos a un array solo si estos no existen ya.

\$each – usado en conjunto con \$addToSet o \$push para indicar varios elementos que deseamos agregar al array.

```
db.personas.update(  
  { nombre: 'Mario' },  
  {  
    $set: { edad : 25 }  
  }  
);
```

Operaciones - Borrado

- Para borrar un registro empleamos el método remove el cual admite filtros como si se tratara de una consulta.
- Si no especificamos ninguna condición borrará todos los documentos de la colección.

```
> db.personas.remove({ nombre: 'Mario' });
```

- Para deshacernos de una colección empleamos drop:

```
> db.personas.drop();
```


Mongoose

- Es un ODM (Object data modeling) de MongoDB para Node.js. Por tanto Mongoose se encarga de convertir los datos de la DB a objetos Javascript que podemos manipular en nuestra aplicación.
- Permite definir nuestros datos mediante esquemas e incluye ciertas funcionalidades que nos facilitarán el trabajo: conversión de tipos, validación, construcción de queries...

```
$ npm install mongoose
```

Definir un esquema

- En Mongoose, todo empieza con un esquema. Cada esquema está relacionado con una colección en MongoDB y define el aspecto que tendrán los documentos de esa colección.

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

var blogSchema = new Schema({
  title: String,
  author: String,
  body: String,
  comments: [{ body: String, date: Date }],
  date: { type: Date, default: Date.now },
  hidden: Boolean,
  meta: {
    votes: Number,
    favs: Number
  }
});
```

Definir un esquema

- Cada clave en el esquema define una propiedad en nuestros documentos que será transformada al tipo indicado en el esquema.
- Los tipos permitidos en un esquema son:
 - String
 - Number
 - Date
 - Buffer
 - Boolean
 - Mixed
 - ObjectId

Modelos

- Para trabajar y manipular instancias que cumplan con un esquema tenemos que emplear modelos.
- Obtenemos un modelo a través del método `model` de mongoose.

```
var Blog = mongoose.model('Blog', blogSchema);  
  
Blog.title = "MiBlog";  
  
console.log(Blog.title);
```

Métodos de instancia

- Además de los métodos predefinidos de un modelo, podemos crear los nuestros. Mediante el campo methods de un esquema agregamos nueva funcionalidad a los modelos.

```
var animalSchema = new Schema({ name: String, type: String });
```

```
// Crear un nuevo método
```

```
animalSchema.methods.findSimilar = function (cb) {  
  return this.model('Animal').find({ type: this.type }, cb);  
}
```

```
var Animal = mongoose.model('Animal', animalSchema);
```

```
var cat = new Animal({ type: 'cat' });
```

```
//Llamada al método
```

```
cat.findSimilar(function (err, cats) {  
  console.log(cats);  
});
```

Métodos estáticos

- Cabe la posibilidad de definir métodos a nivel de modelo, independientes de los valores de cada instancia.
- Para configurar un nuevo método estático, agregamos una función al campo statics del esquema:

```
animalSchema.statics.findByName = function (name, cb) {  
  this.find({ name: new RegExp(name, 'i') }, cb);  
}  
  
var Animal = mongoose.model('Animal', animalSchema);  
Animal.findByName('Barricoco', function (err, animals) {  
  console.log(animals);  
});
```

Propiedades virtuales

- Son propiedades de los documentos que se pueden escribir y leer pero que no se almacenan en la base de datos.
- Los “getters” son muy útiles para dar formato y/o para combinar propiedades:

```
personaSchema.virtual('nombrecompleto').get(function () {  
  return this.nombre + ' ' + this.apellido;  
});
```

- Los “setters” nos permiten almacenar un valor de entrada complejo en diferentes propiedades del documento:

```
personaSchema.virtual('nombrecompleto').set(function (name) {  
  var split = name.split(' ');  
  this.nombre = split[0];  
  this.apellido = split[1];  
});
```

Crear documentos

- Los documentos son instancias de nuestros modelos. Para almacenar una instancia empleamos el método save.

```
var schema = new mongoose.Schema({ color: 'string', size: 'string' });  
var Box = mongoose.model('Box', schema);
```

```
var small = new Box({ size: 'small' });  
small.save(function (err) {  
  if (err) return handleError(err);  
  // Documento creado!  
})
```

```
// Directamente desde el modelo
```

```
Box.create({ size: 'small' }, function (err, small) {  
  if (err) return handleError(err);  
  // Documento creado!  
})
```


Consultar documentos

- Buscar documentos con Mongoose es muy sencillo ya que admite todo los operadores de mongoDB y tiene métodos predefinidos para esta labor.

```
User.find({age: {$gte: 21, $lte: 65}}, callback);
```

```
User.where('age').gte(21).lte(65).exec(callback);
```

- La clase Query también soporta el método where así que podemos anidar sentencias where:

```
User  
  .where('age').gte(21).lte(65)  
  .where('name', /^b/i)
```

Actualizar documentos

- Mongoose ofrece diversas formas de actualizar documentos según nuestras necesidades.
- Si necesitamos obtener el documento:

```
Box.findById(id, function (err, box) {  
  if (err) return handleError(err);  
  
  box.size = 'large';  
  box.save(function (err) {  
    if (err) return handleError(err);  
    res.send(box);  
  });  
});
```

- Esta aproximación implica lanzar una consulta y posteriormente una escritura en la DB con la llamada a save.

Actualizar documentos

- Si no necesitamos obtener el documento podemos usar el método update del modelo:

```
Box.update({ _id: id }, { $set: { size: 'large' }}, callback);
```

- Otra opción un poco más eficiente es llamar al método findAndUpdate.

```
Box.findByIdAndUpdate(id, { $set: { size: 'large' }}, function (err, tank) {  
  if (err) return handleError(err);  
  res.send(tank);  
});
```

Borrar documentos

- Los modelos tienen el método estático `remove` que nos permite borrar los documentos que coincidan con las condiciones:

```
Box.remove({ size: 'large' }, function (err) {  
  if (err) return handleError(err);  
  // Borrado!  
});
```

Modelado de relaciones

- Si queremos que un documento haga referencia a uno o varios documentos de otra colección, podemos apuntar los id dentro de un array. Aunque lo más lógico es emplear la opción `ref` que nos ofrece `mongoose`:

```
var personSchema = Schema({
  _id      : Number,
  name     : String,
  age      : Number,
  posts    : [{ type: Schema.Types.ObjectId, ref: 'Post' }]
});
```

```
var postSchema = Schema({
  _creator : { type: Number, ref: 'Person' },
  title    : String,
});
```

```
var Post = mongoose.model('Post', storySchema);
var Person = mongoose.model('Person', personSchema);
```

Guardar referencias

- guardar una referencia es similar a guardar cualquier otro tipo de propiedad:

```
var john = new Person({ _id: 0, name: 'John', age: 25 });
```

```
var post1 = new Post({  
  title: "MyPost",  
  _creator: john._id    // Asignar el id del creador  
});
```

```
post1.save(function (err) {  
  if (err) return handleError(err);  
  
  // Post almacenado  
  
  john.posts.push(post1);  
  john.save(function (err) {  
    if (err) return handleError(err);  
    //Persona almacenada  
  })  
  
})
```

Population

- Mongoose ofrece un mecanismo para poblar un documento con los documentos a los que hace referencia. En este ejemplo obtenemos el documento de la persona John con todos sus posts en forma de sub-documentos.

```
Person
.findOne({ name: 'John' })
.populate('posts') // only works if we pushed refs to children
.exec(function (err, person) {
  if (err) return handleError(err);
  console.log(person);
})
```