

Unidad 9. DESARROLLO AVANZADO DE BACKEND

Unidad 9: MongoDB Avanzado y Optimización de Consultas

Objetivos de la clase

- Comprender el funcionamiento de una indexación en MongoDB
- Analizar documentos para definir sus índices
- Conocer la operación “populate” de mongoose
- Utilización de middleware en mongoose

Objetivos de la clase

- Comprender el concepto de aggregation
- Realizar caso práctico para una aggregation
- Comprender el concepto de paginación y utilización de mongoose-paginatev2

Glosario

DBaaS: Database as a Service. Sirve para poder contar con un alojamiento de nuestra base de datos desde un servidor en la nube, para evitar hacer almacenamientos físicos propios.

Mongo Atlas: DBaaS pensada para MongoDB para setear Clusters y alojar múltiples bases de datos.

Cliente CLI: Cliente que conecta a la base de datos desde una terminal de comandos.

Cliente UI: Cliente que conecta a la base de datos a partir de un programa de escritorio, como MongoDB Compass

Cliente Web: Cliente que conecta a la base de datos a partir del navegador, como la gestión a partir de Atlas

Cliente App: Client que conecta dentro de nuestro sistema de backend. Como nuestro programa de nodejs con mongoose.

Mongoose: ODM utilizado para gestionar schemas definidos para mantener un control en la flexibilidad de MongoDB. Además de otras operaciones importantes para el manejo de sus datos.

Glosario

Indexación: Técnica utilizada para colocarse en una propiedad de un documento, permite realizar búsquedas más rápidas cuando se involucra a dicha propiedad

find().explain("executionStats"): No devuelve el resultado de la búsqueda, sino que tiene por objetivo devolver las estadísticas de la operación.

executionStats.executionTimeMillis: Tiempo en milisegundos que demoró hacer la operación

population: Operación que permite transformar la referencia de un documento en su documento correspondiente en la colección indicada.

middleware: Operación intermedia que ocurre entre la petición a la base de datos y la entrega del documento o los documentos correspondientes.

pre: Middleware utilizado para realizar una operación "antes" de devolver el resultado de la operación principal.

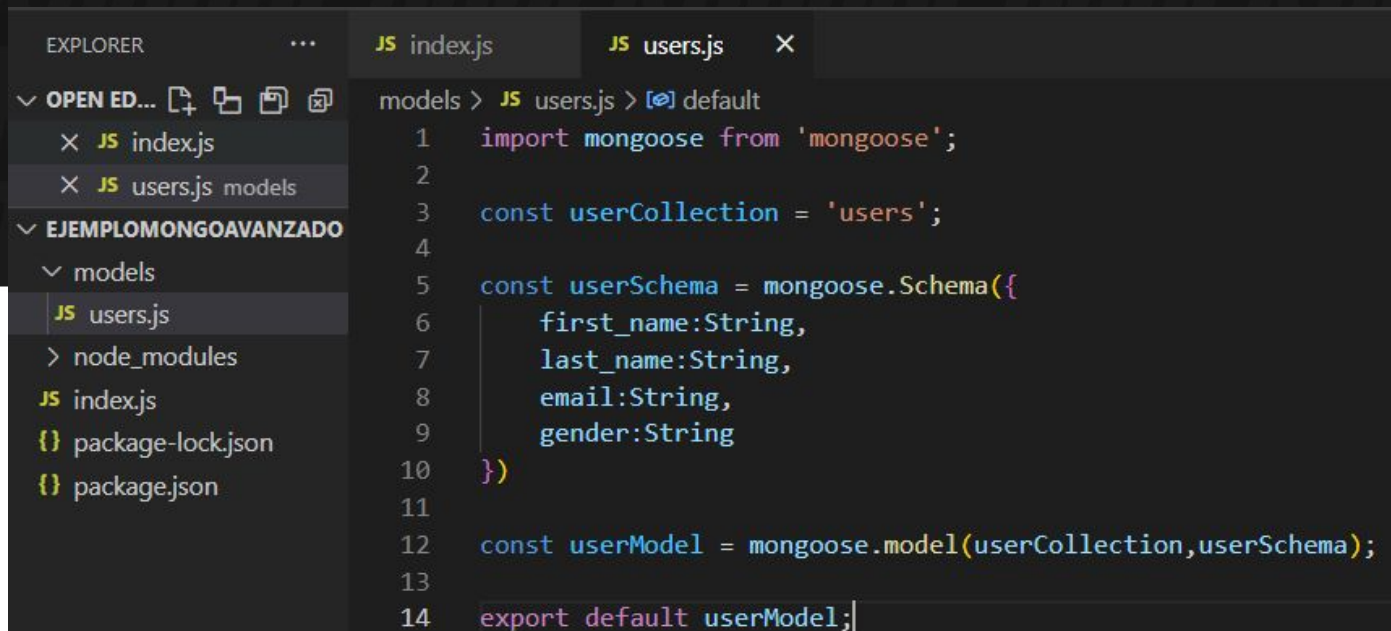


Ejemplo en vivo

El profesor tendrá importado en su base de datos 5000 usuarios en una colección “users”, sobre éste, se realizarán ciertas búsquedas con y sin filtros.

Se analizará el flujo y utilización de un índice para comparar resultados.

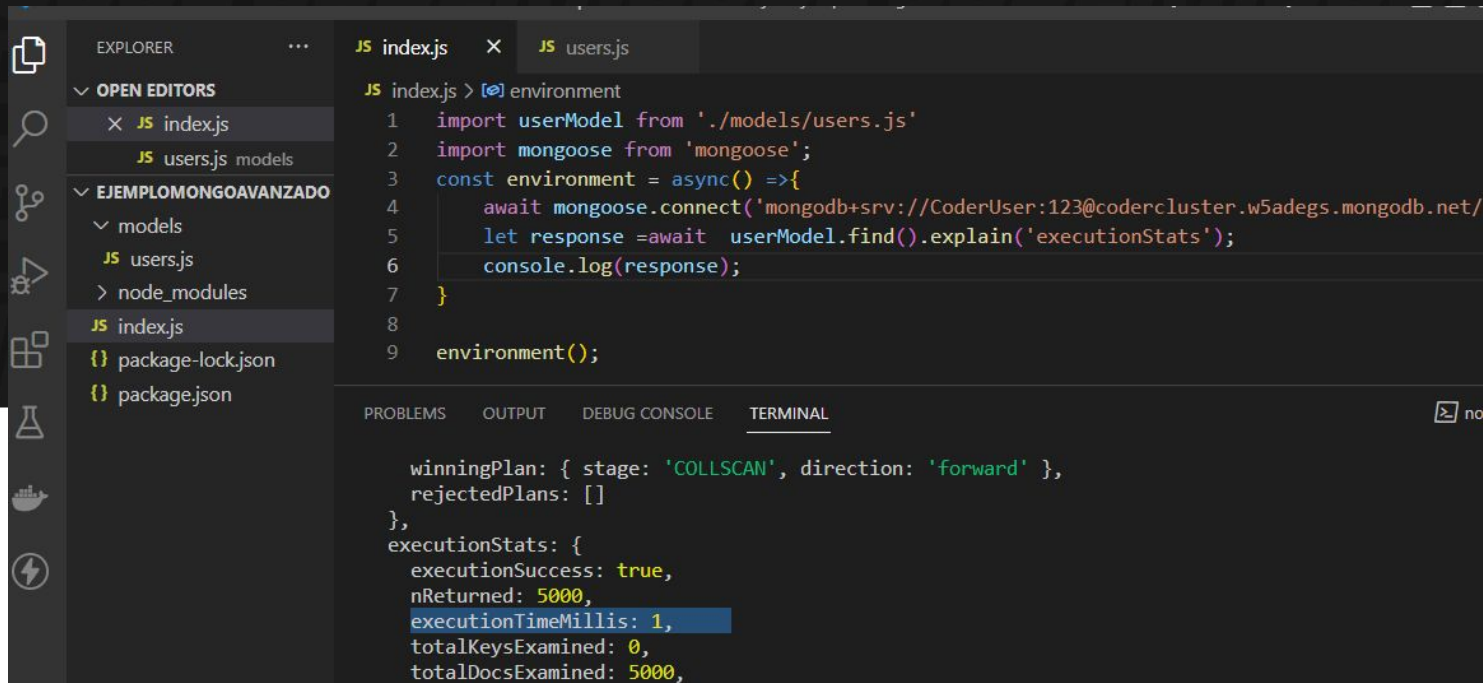
Ejemplo: Desarrollando el schema users con mongoose para empatar con la base de datos



The screenshot shows the Visual Studio Code interface with the Explorer panel on the left and the Code Editor on the right. The Explorer panel shows a project structure with a folder named 'EJEMPLOMONGOAVANZADO' containing a 'models' folder. Inside 'models', the file 'users.js' is selected. The Code Editor shows the content of 'users.js', which is a JavaScript file using Mongoose to define a user schema and model. The code is as follows:

```
models > JS users.js > [default]
1  import mongoose from 'mongoose';
2
3  const userCollection = 'users';
4
5  const userSchema = mongoose.Schema({
6    first_name:String,
7    last_name:String,
8    email:String,
9    gender:String
10  });
11
12  const userModel = mongoose.model(userCollection,userSchema);
13
14  export default userModel;
```

Ejemplo: Archivo index para realizar la búsqueda



The screenshot shows the Visual Studio Code editor with the Explorer sidebar on the left. The Explorer sidebar shows the project structure: **OPEN EDITORS** (index.js, users.js), **EJEMPLOMONGOAVANZADO** (models, users.js, node_modules, index.js, package-lock.json, package.json). The main editor area shows the `index.js` file with the following code:

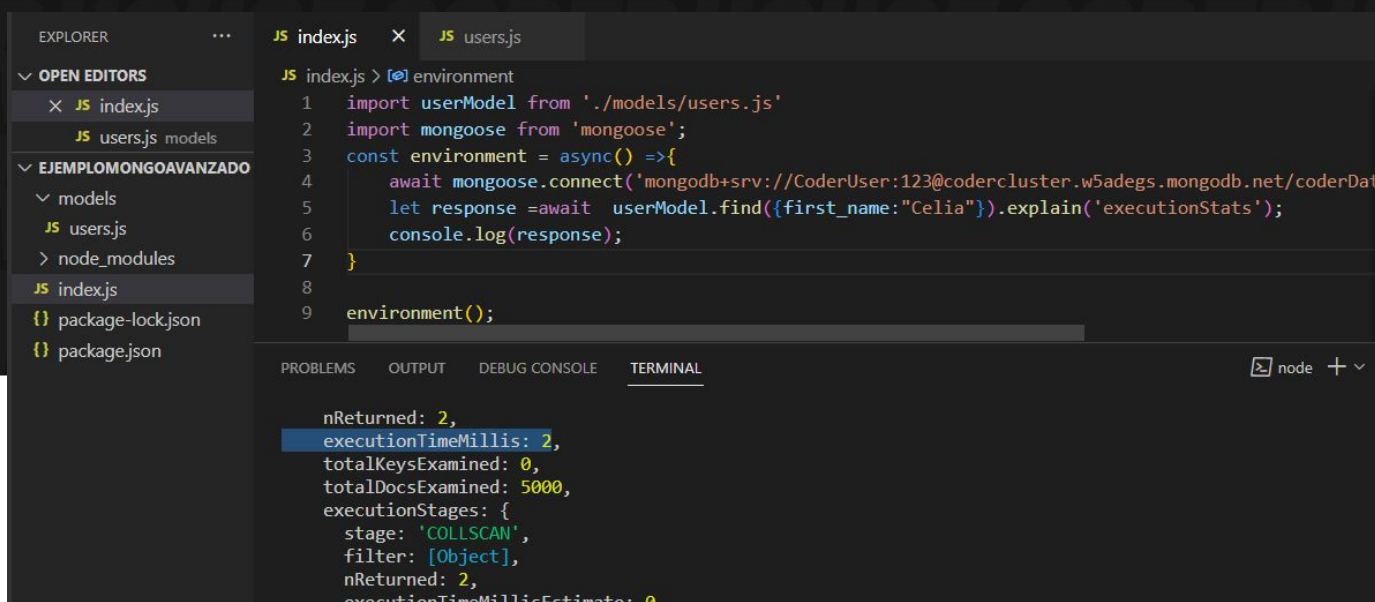
```
JS index.js > [?] environment
1 import userModel from './models/users.js'
2 import mongoose from 'mongoose';
3 const environment = async() =>{
4     await mongoose.connect('mongodb+srv://CoderUser:123@codercluster.w5adegs.mongodb.net/');
5     let response =await userModel.find().explain('executionStats');
6     console.log(response);
7 }
8
9 environment();
```

The bottom panel shows the **TERMINAL** output:

```
winningPlan: { stage: 'COLLSCAN', direction: 'forward' },
rejectedPlans: []
},
executionStats: {
  executionSuccess: true,
  nReturned: 5000,
  executionTimeMillis: 1,
  totalKeysExamined: 0,
  totalDocsExamined: 5000,
```

Tiempo de ejecución de la búsqueda para 5 mil usuarios: 1 milisegundo

Ejemplo: Realizando búsqueda con filtro por nombre



The screenshot shows a VS Code editor with two files open: `index.js` and `users.js`. The `index.js` file contains a function `environment` that connects to a MongoDB database and performs a search for users with the first name 'Celia'. The terminal output shows the execution results, including the number of documents returned (2) and the execution time (2 milliseconds).

```
JS index.js > environment
1 import userModel from './models/users.js'
2 import mongoose from 'mongoose';
3 const environment = async() =>{
4   await mongoose.connect('mongodb+srv://CoderUser:123@codercluster.w5adegs.mongodb.net/coderDat
5   let response =await userModel.find({first_name:"Celia"}).explain('executionStats');
6   console.log(response);
7 }
8
9 environment();
```

```
nReturned: 2,
executionTimeMillis: 2,
totalKeysExamined: 0,
totalDocsExamined: 5000,
executionStages: {
  stage: 'COLLSCAN',
  filter: [Object],
  nReturned: 2,
  executionTimeMillisEstimate: 0
```

Tiempo de ejecución de la búsqueda para 5 mil usuarios buscando por nombre:
2 milisegundos

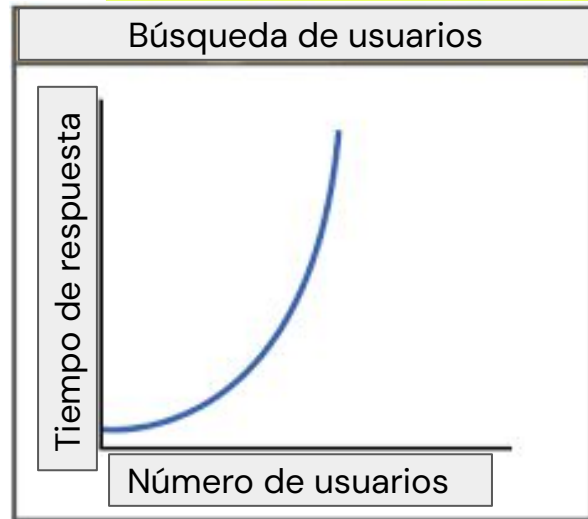
Ejemplo: Analizando resultados

Una diferencia muy pequeña... ¿o no?

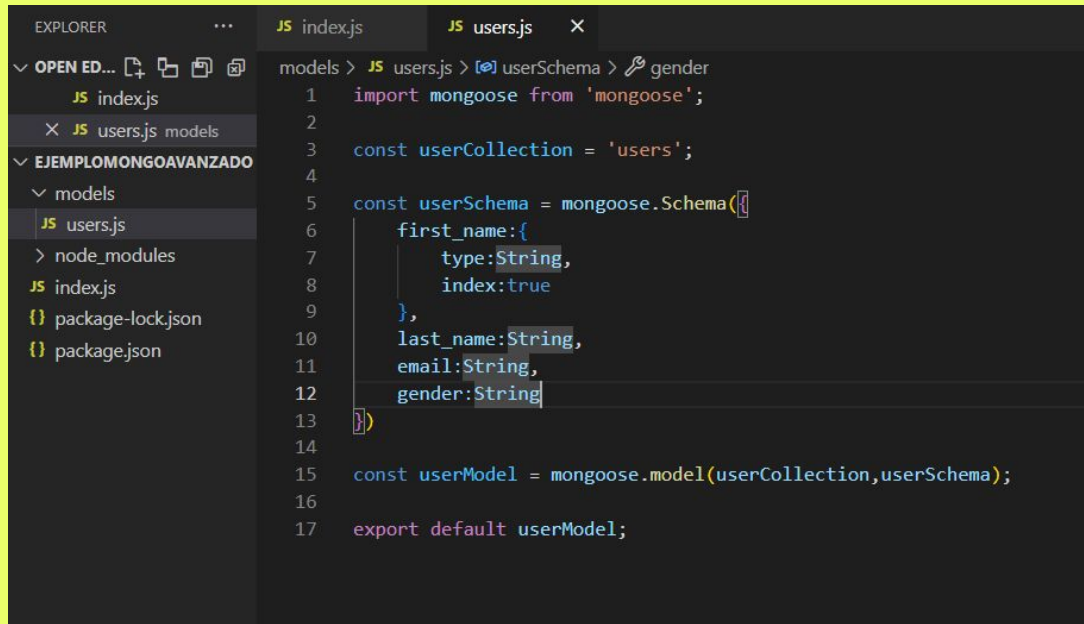
El tiempo de búsqueda general, contra el tiempo de búsqueda por filtro es de un milisegundo aproximadamente, ¿es esto realmente una problemática a considerar?

El ejemplo hasta el momento muestra 5 mil documentos. ¿Qué pasará cuando tengamos 20, 50 o 100 mil documentos? Los números pueden dispararse considerablemente en cuanto a su tiempo de respuesta.

Es por ello que debemos anticiparnos a los crecimientos exponenciales que puede haber a lo largo de la trayectoria de la aplicación.



Ejemplo: Agregando "index" al campo a buscar

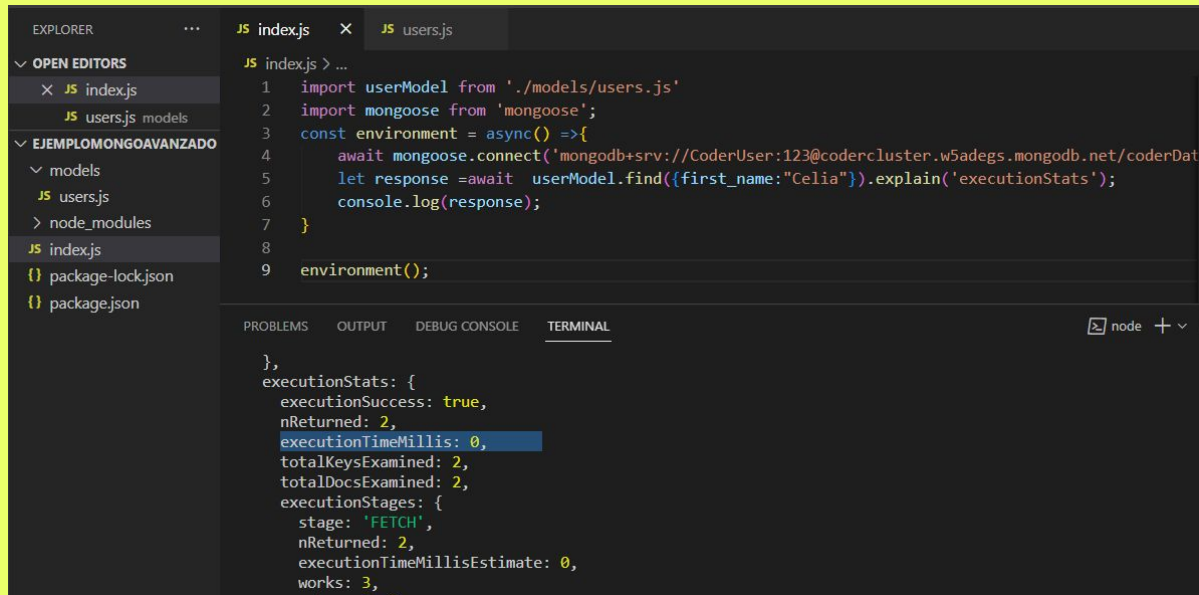


The screenshot shows a code editor with a file explorer on the left and a code editor on the right. The file explorer shows a project structure with files like `index.js`, `users.js`, `package-lock.json`, and `package.json`. The code editor shows the `users.js` file with the following code:

```
1 import mongoose from 'mongoose';
2
3 const userCollection = 'users';
4
5 const userSchema = mongoose.Schema({
6   first_name: {
7     type: String,
8     index: true
9   },
10  last_name: String,
11  email: String,
12  gender: String
13 });
14
15 const userModel = mongoose.model(userCollection, userSchema);
16
17 export default userModel;
```

Agregamos indexación al campo que estamos buscando dentro del documento

Ejemplo: Realizando búsqueda indexada



The screenshot shows a VS Code editor with two files open: `index.js` and `users.js`. The `index.js` file contains a MongoDB query that finds a user by first name and explains the execution stats. The terminal output shows the execution stats for the query, indicating that the search was very fast.

```
JS index.js > ...
1 import userModel from './models/users.js'
2 import mongoose from 'mongoose';
3 const environment = async() =>{
4   await mongoose.connect('mongodb+srv://CoderUser:123@codercluster.w5adegs.mongodb.net/coderDat
5   let response =await userModel.find({first_name:"Celia"}).explain('executionStats');
6   console.log(response);
7 }
8
9 environment();
```

```
{
  executionStats: {
    executionSuccess: true,
    nReturned: 2,
    executionTimeMillis: 0,
    totalKeysExamined: 2,
    totalDocsExamined: 2,
    executionStages: {
      stage: 'FETCH',
      nReturned: 2,
      executionTimeMillisEstimate: 0,
      works: 3,
      ...
    }
  }
}
```

Tiempo de ejecución de la búsqueda para 5 mil usuarios buscando por nombre: 0 milisegundos (¡Tan bajo que no fue perceptible!)

Ejemplo: Conclusión

Una clara diferencia de performance

Es evidente que hay una reducción en el tiempo de respuesta, y que esta mejora de performance será mucho más notoria conforme el número de documentos crezca.

Entonces, podemos crear indexación de los campos que consideremos que pueden ocasionar problemas de lentitud dentro de nuestras búsquedas

Parte de tu trabajo está en analizar los servicios que utiliza tu aplicativo, para poder desarrollar una estrategia de indexación adecuada a las búsquedas que realicemos en la base de datos.

¡Importante!

Un índice no debe ser utilizado en todos los campos, sólo deben ser utilizados en los campos que sepamos tienen repercusión en nuestras búsquedas.

Colocar un índice en cada campo de cada documento, significa alentar procesos de escritura en cada insert, así también como generar un almacenamiento adicional e innecesario en la base de datos.



Actividad colaborativa

Análisis de indexación

¿Qué índices colocar en cada documento?
¡Revisaremos diferentes casos para practicar lo visto hasta ahora!

Duración: **15-20 minutos**

Análisis de indexación

Consigna:

Se hará revisión de un conjunto de schemas. Analizar e identificar las propiedades de un documento y definir en cuáles deberíamos utilizar una indexación. La decisión debe ser tomada según el contexto de cada Schema.

Justificar en cada caso por qué se tomó dicho índice como una posible opción. Recuerda que las respuestas de esta actividad pueden llegar a ser subjetivas, según la justificación que brindemos para cada caso.

PD: Recuerda que puede haber más de un índice por documento.



ACTIVIDAD COLABORATIVA

//Contexto: Schema de estudiante de un curso en específico (grade refiere a la calificación de dicho curso)

```
{  
  first_name:String,  
  last_name:String,  
  email:String,  
  telephone:String,  
  age:Number,  
  grade:Number,  
  gender:String,  
  address:String  
}
```

1

//Contexto: Schema de usuario de una aplicación de entregas y envíos

```
{  
  first_name:String,  
  last_name:String,  
  email:String,  
  telephone:String,  
  age:Number,  
  gender:String,  
  address:String,  
  postal_code:String,  
}
```

3

//Contexto: Schema de ticket de compra generado desde un ecommerce

```
{  
  buyer_id : ObjectId,  
  total_ammount:Number,  
  products:Array,  
  destination_address:String,  
  destination_postal_code:String,  
  comments:String,  
}
```

2

//Contexto: Schema de un libro correspondiente a un negocio de librería.

```
{  
  title:String,  
  description:String,  
  prize:Number,  
  reviews:Array,  
  rating:Number,  
  images:Array,  
  author:ObjectId,  
  num_of_pages:Number  
}
```

4

Populations en Mongoose

Algunas cosas a considerar antes de comenzar con su uso

- ✓ populate es un método propio de mongoose, por lo que tenemos que instalarlo.
- ✓ Hay que tener siempre claro el nombre de la propiedad dentro del objeto, así también como la referencia de la colección, para poder hacer un populate efectivo.
- ✓ Recuerda no guardar directamente el valor a referenciar en el `_id`, sino asignarle otro nombre (se profundizará en el ejemplo)



1. Tener un entorno asíncrono

Por cuestiones de tiempo y complejidad, no levantaremos un servidor esta vez, sino que definiremos un entorno asíncrono, para poder ejecutar las operaciones de Mongoose, sin tener que llamar a un endpoint.

Además, aprovecharemos para realizar la operación "connect" de mongoose y conectar con la base de datos a utilizar (Recomendable usar el link de conexión de Atlas)

```
JS index.js  X
JS index.js > ...
1
2  import mongoose from 'mongoose';
3  const environment = async() =>{
4    await mongoose.connect('mongodb+srv://Co
5
6  }
7
8  environment();
9
10
11
12
13 |
```



2. Definir el modelo student

```
EXPLORER
  OPEN ED...
  JS studentjs models
  JS usersjs models
  EJEMPLO...
  models
    JS coursesjs
    JS studentjs
    JS usersjs
  node_modules
  JS indexjs
  package-lockjson
  packagejson

models > JS studentjs > ...
1  import mongoose from 'mongoose';
2
3  const studentCollection = 'students';
4
5  const studentSchema = mongoose.Schema({
6    first_name: String,
7    last_name: String,
8    email: String,
9    gender: String,
10   courses: {
11     type: [
12       {
13         course: {
14           type: mongoose.Schema.Types.ObjectId,
15           ref: "courses"
16         }
17       }
18     ],
19     default: []
20   }
21 });
22
23 const studentModel = mongoose.model(studentCollection, studentSchema);
24
25 export default studentModel;
```

Definiremos un modelo como los vistos en clases anteriores, sin embargo, notamos una estructura considerablemente más compleja en el campo **courses**.

La estructura del campo **courses** indica que **cada elemento que se ingrese al arreglo** debe tener un campo "course", el cual será un id que hará referencia a la colección **courses**.

Este "ref" es el que utilizamos para saber que haremos un populate a la colección indicada.



3. Creando el primer documento en la colección students (corroborar creación en tu base de datos)

```
JS index.js > [❌] environment
1  import mongoose from 'mongoose';
2  import studentModel from './models/student.js'
3  const environment = async () => {
4    await mongoose.connect('mongodb+srv://CoderUser:1
5    //Creamos nuestro estudiante de prueba en la base
6    await studentModel.create({
7      first_name: "Hilda",
8      last_name: "Coruño",
9      email: "correoHilda@correo.com",
10     gender: "Female",
11   })
12 }
13
14 environment();
```



QUERY RESULTS: 1-1 OF 1

```
{
  _id: ObjectId("62d0ccde16e1b4b228ad5128"),
  first_name: "Hilda",
  last_name: "Coruño",
  email: "correoHilda@correo.com",
  gender: "Female",
  > courses: Array
  __v: 0
}
```



4. Definir el modelo course

```
models > JS courses.js > ...
1  import mongoose from 'mongoose';
2
3  const courseCollection = 'courses';
4
5  const courseSchema = mongoose.Schema({
6    title:String,
7    description:String,
8    difficulty:Number,
9    topics:{
10     type:Array,
11     default:[]
12   },
13   professor:String,
14   students:{
15     type:Array,
16     default:[]
17   }
18 });
19
20 const courseModel = mongoose.model(courseCollection, courseSchema);
21
22 export default courseModel;
23
24
25
```

course hará referencia a los diferentes cursos a los que puede pertenecer un estudiante.

Éste no contará con referencia y su uso para este ejemplo se reducirá a ser un elemento que pueda agregarse al arreglo "courses" de los documentos de tipo estudiante.



5. Creando el primer curso en la base

```
JS index.js > [?] environment
• 1 import mongoose from 'mongoose';
  2 import studentModel from './models/student.js';
  3 import courseModel from './models/courses.js';
  4 const environment = async () => {
  5     await mongoose.connect('mongodb+srv://CoderUser:123@codercluster.w5adegs.mongodb.net/coderDatabase
  6
  7     //Creamos nuestro curso de prueba en la base
  8     await courseModel.create({
  9         title: "Curso de Backend",
 10         description: "Es un curso que permite desarrollar servidores bien bonitos",
 11         difficulty: 5,
 12         topics: ["Javascript", "Servidores", "Motores de plantillas", "Middlewares", "Base de datos"],
 13         professor: "Mauricio"
 14     })
 15
 16 }
 17
 18 environment();
```


Hasta este punto del proceso tenemos los siguientes elementos en la base:

Colección "users"

QUERY RESULTS: 1-1 OF 1

```
_id: ObjectId("62d0ccde16e1b4b228ad5128")
first_name: "Hilda"
last_name: "Coruño"
email: "correoHilda@correo.com"
gender: "Female"
> courses: Array
__v: 0
```

Colección "courses"

QUERY RESULTS: 1-1 OF 1

```
>
_id: ObjectId("62d0ce0a25973977f1e75451")
title: "Curso de Backend"
description: "Es un curso que permite desarrollar servidores bien bonitos"
difficulty: 5
> topics: Array
professor: "Mauricio"
> students: Array
__v: 0
```

El objetivo será colocar el `_id` del curso dentro del arreglo de cursos del estudiante

Colección "users"

QUERY RESULTS: 1-1 OF 1

```
_id: ObjectId("62d0ccde16e1b4b228ad5128")
first_name: "Hilda"
last_name: "Coruño"
email: "correoHilda@correo.com"
gender: "Female"
> courses: Array
  __v: 0
```

Colección "courses"

QUERY RESULTS: 1-1 OF 1

```
> _id: ObjectId("62d0ce0a25973977f1e75451")
  title: "Curso de Backend"
  description: "Es un curso que permite desarrollar servidores bien bonitos"
  difficulty: 5
  > topics: Array
  professor: "Mauricio"
  > students: Array
  __v: 0
```



6. Agregando el id del curso al arreglo de cursos del estudiante

JS index.js × JS student.js

JS index.js > [🔗] environment

```
1 import mongoose from 'mongoose';
2 import studentModel from './models/student.js';
3 import courseModel from './models/courses.js';
4 const environment = async () => {
5   await mongoose.connect('mongodb+srv://CoderUser:123@codercluster.w5adegs.mongodb.net/coderhouse');
6
7   let student = await studentModel.find({_id:"62d0ccde16e1b4b228ad5128"})
8   /**
9    * Nota que el curso lo agregamos en un campo "course" y no directamente al _id.
10   * El nombre de la propiedad debe coincidir con la indicada en el schema.
11   */
12   student.courses.push({course:"62d0ce0a25973977f1e75451"});
13   let result = await studentModel.updateOne({_id:"62d0ccde16e1b4b228ad5128"},student);
14 }
15
16 environment();
```



Y ese _id... ¿de dónde salió?

Si regresamos a nuestra base de datos, notaremos que ya se encuentra agregado el curso... ¿con dos ids?

El id del campo "course" es donde agregamos el curso, **éste es el que utilizaremos para nuestra population.**

Por otra parte, el _id adicional lo autogeneró Mongo al agregar el nuevo documento. Éste no lo utilizaremos.

QUERY RESULTS: 1-1 OF 1

```
_id: ObjectId("62d0ccde16e1b4b228ad5128")
first_name: "Hilda"
last_name: "Coruño"
email: "correoHilda@correo.com"
gender: "Female"
✓ courses: Array
  ✓ 0: Object
    course: ObjectId("62d0ce0a25973977f1e75451")
    _id: ObjectId("62d0d1968d4f33cd8797127f")
  __v: 0
```



APROXIMACIÓN AL PROCESO

```
let student = await studentModel.find({_id:"62d0ccde16e1b4b228ad5128"})  
console.log(JSON.stringify(student,null,'\t'));
```

**Y obtenemos al estudiante con
los cursos sin populate**

**7. Volvemos a pedir al
estudiante en la base de
datos.**

```
[  
  {  
    "_id": "62d0ccde16e1b4b228ad5128",  
    "first_name": "Hilda",  
    "last_name": "Coruño",  
    "email": "correoHilda@correo.com",  
    "gender": "Female",  
    "courses": [  
      {  
        "course": "62d0ce0a25973977f1e75451",  
        "_id": "62d0d1968d4f33cd8797127f"  
      }  
    ],  
    "__v": 0  
  }  
]
```



8. Añadiendo a la query un populate

```
JS index.js  X  JS student.js

JS index.js > [?] environment
1  import mongoose from 'mongoose';
2  import studentModel from './models/student.js';
3  import courseModel from './models/courses.js';
4  const environment = async () => {
5      await mongoose.connect('mongodb+srv://CoderUser:123@codercluster.w5adegs.mongodb.net/coderDatabase?retryWrites=true&appName=coderhouse');
6      /**
7       * La razón por la cual se coloca courses.course, es debido a que los elementos que queremos poblar están
8       * dentro de un array, entonces: courses es el arreglo y course el curso en cuestión, si no utilizamos esta
9       * sintaxis, no podremos encontrar el elemento interno.
10     */
11     let student = await studentModel.find({_id:"62d0ccde16e1b4b228ad5128"}).populate('courses.course');
12     console.log(JSON.stringify(student,null,'\t'));
13 }
14
15 environment();
16
```

¡Voilà! Tenemos ahora el documento con el estudiante, además de tener su curso agregado dentro del arreglo **y completo, no sólo el id** population en su máximo esplendor.

```
{
  "_id": "62d0ccde16e1b4b228ad5128",
  "first_name": "Hilda",
  "last_name": "Coruño",
  "email": "correoHilda@correo.com",
  "gender": "Female",
  "courses": [
    {
      "course": {
        "_id": "62d0ce0a25973977f1e75451",
        "title": "Curso de Backend",
        "description": "Es un curso que permite desarrollar servidores bien bonitos",
        "difficulty": 5,
        "topics": [
          "Javascript",
          "Servidores",
          "Motores de plantillas",
          "Middlewares",
          "Base de datos"
        ],
        "professor": "Mauricio",
        "students": [],
        "__v": 0
      },
      "_id": "62d0d1968d4f33cd8797127f"
    }
  ],
  "__v": 0
}
```

¡Importante!

Una population es un puente entre dos documentos, como una relación unidireccional.

¡Jamás hagas una population bidireccional! Esto ocasionará que uno llame a otro, otro a uno, uno a otro, otro a uno... etc.

Por ejemplo, ¿qué pasaría si nuestro documento de cursos tuviera referencia del estudiante en su arreglo "students"?

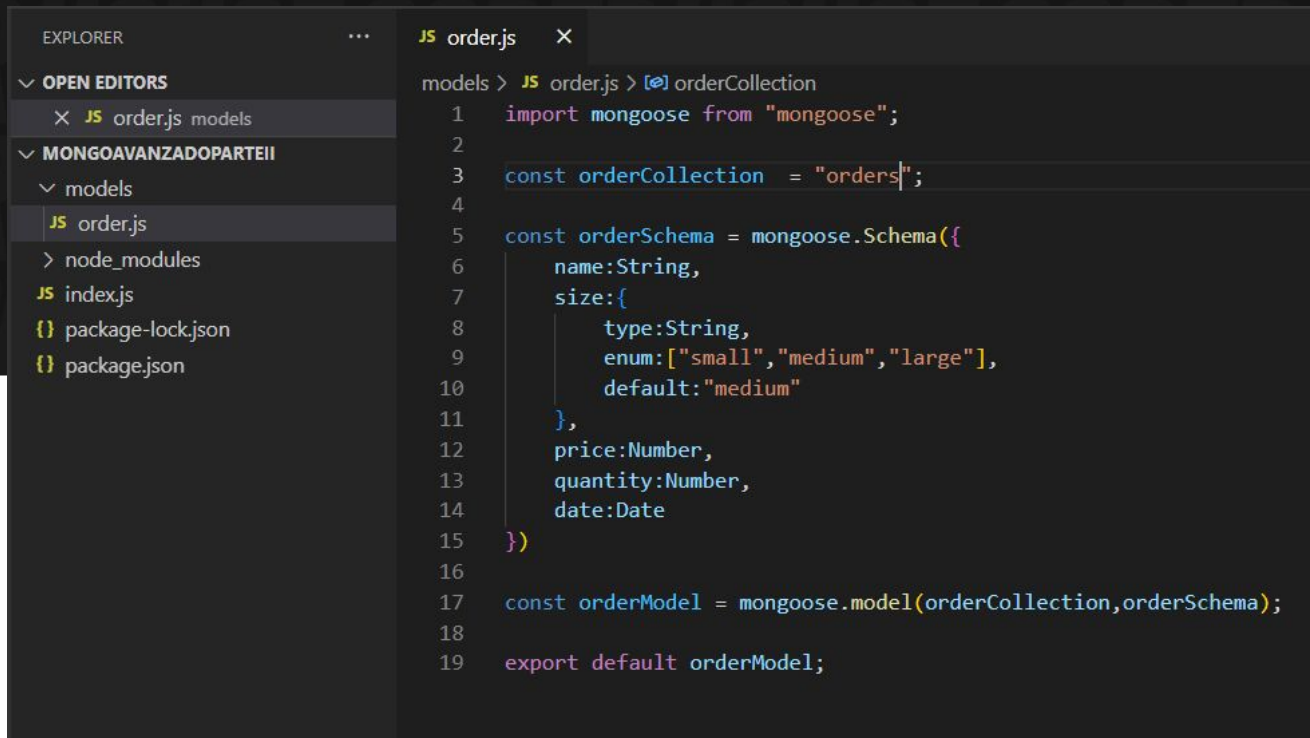


Ejemplo en vivo

Se desea gestionar una base de datos para una pizzería. Dado un conjunto de órdenes:

- ✓ Definir las ventas totales de los diferentes sabores para las pizzas medianas.

Ejemplo: Desarrollando el schema para órdenes de pizza.



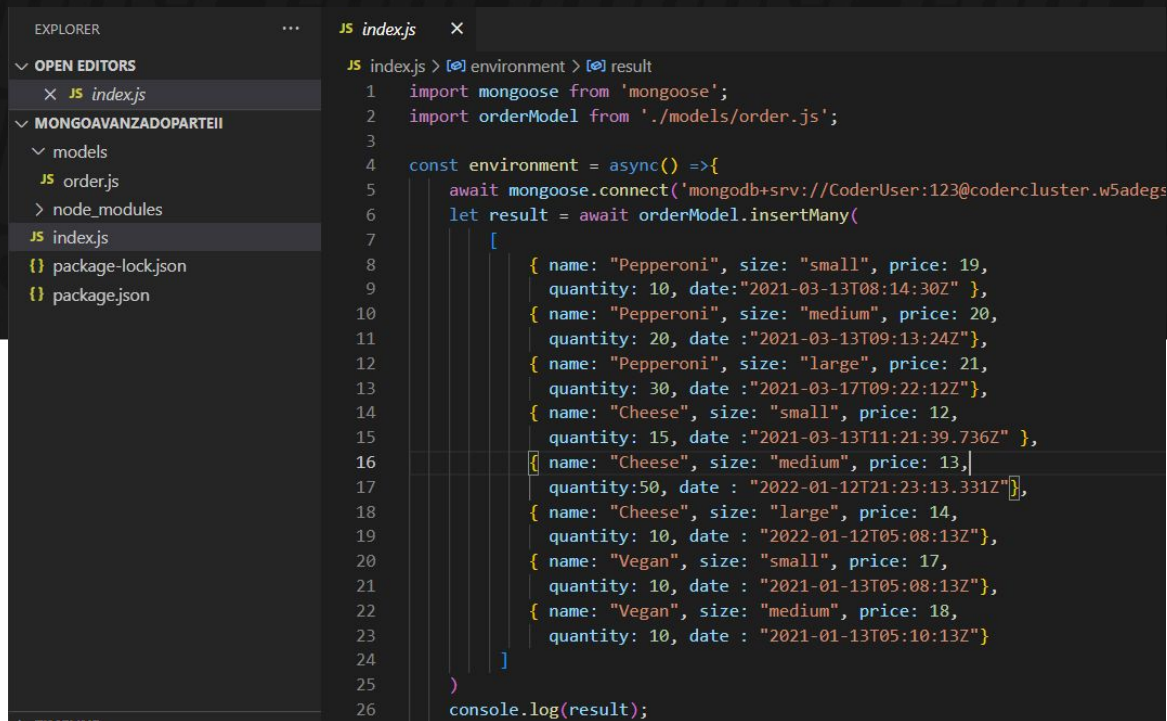
The screenshot shows the VS Code interface with the Explorer on the left and the Editor on the right. The Explorer shows the project structure with the following files and folders:

- EXPLORED
- OPEN EDITORS
 - JS order.js models
- MONGOAVANZADOPARTEII
 - models
 - JS order.js
 - node_modules
 - JS index.js
 - package-lock.json
 - package.json

The Editor shows the code in `models > JS order.js` with the following content:

```
1 import mongoose from "mongoose";
2
3 const orderCollection = "orders";
4
5 const orderSchema = mongoose.Schema({
6   name:String,
7   size:{
8     type:String,
9     enum:["small","medium","large"],
10    default:"medium"
11  },
12  price:Number,
13  quantity:Number,
14  date:Date
15 });
16
17 const orderModel = mongoose.model(orderCollection,orderSchema);
18
19 export default orderModel;
```

Ejemplo: Cargando datos de prueba en archivo index.js

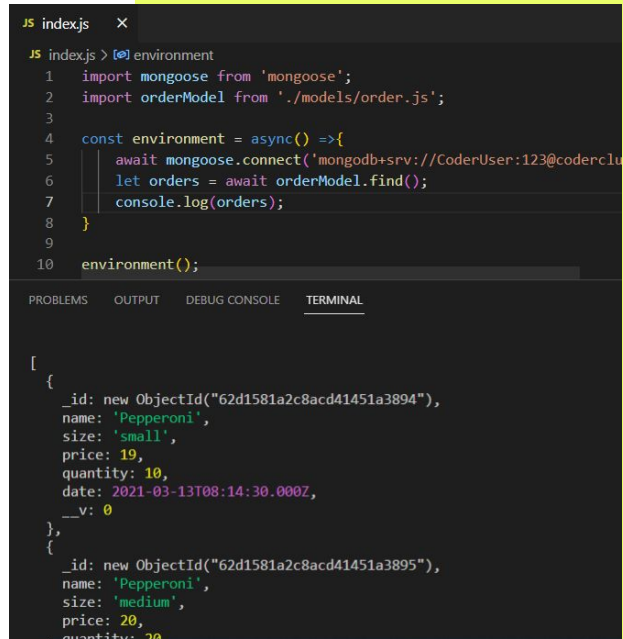


```
JS index.js x
JS index.js > [environment] > [result]
1 import mongoose from 'mongoose';
2 import orderModel from './models/order.js';
3
4 const environment = async() =>{
5   await mongoose.connect('mongodb+srv://CoderUser:123@codercluster.w5adegs
6   let result = await orderModel.insertMany(
7     [
8       { name: "Pepperoni", size: "small", price: 19,
9         quantity: 10, date:"2021-03-13T08:14:30Z" },
10      { name: "Pepperoni", size: "medium", price: 20,
11        quantity: 20, date : "2021-03-13T09:13:24Z"},
12      { name: "Pepperoni", size: "large", price: 21,
13        quantity: 30, date : "2021-03-17T09:22:12Z"},
14      { name: "Cheese", size: "small", price: 12,
15        quantity: 15, date : "2021-03-13T11:21:39.736Z" },
16      { name: "Cheese", size: "medium", price: 13,
17        quantity:50, date : "2022-01-12T21:23:13.331Z"},
18      { name: "Cheese", size: "large", price: 14,
19        quantity: 10, date : "2022-01-12T05:08:13Z"},
20      { name: "Vegan", size: "small", price: 17,
21        quantity: 10, date : "2021-01-13T05:08:13Z"},
22      { name: "Vegan", size: "medium", price: 18,
23        quantity: 10, date : "2021-01-13T05:10:13Z"}
24     ]
25   )
26   console.log(result);
```

Ejemplo: Verificando inserción

Corroborando que todos los datos estén cargados

Realizamos un find para corroborar que nos devuelva las órdenes de la base de datos. Una vez que nos devuelva los resultados, estamos listos para poder comenzar a realizar las aggregations y resolver lo que se nos solicita.



```
JS index.js X
JS index.js > environment
1 import mongoose from 'mongoose';
2 import orderModel from './models/order.js';
3
4 const environment = async() =>{
5   await mongoose.connect('mongodb+srv://CoderUser:123@coderclo');
6   let orders = await orderModel.find();
7   console.log(orders);
8 }
9
10 environment();

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

[
  {
    _id: new ObjectId("62d1581a2c8acd41451a3894"),
    name: 'Pepperoni',
    size: 'small',
    price: 19,
    quantity: 10,
    date: 2021-03-13T08:14:30.000Z,
    __v: 0
  },
  {
    _id: new ObjectId("62d1581a2c8acd41451a3895"),
    name: 'Pepperoni',
    size: 'medium',
    price: 20,
    quantity: 20
  }
]
```

Ejemplo: Análisis de la primera petición

Primera petición: Definir las ventas de los diferentes sabores de las pizzas medianas.

El equipo de ventas corrobora que hay bajas en el número de peticiones de pizzas medianas y necesita confirmar el monto general que ha habido en las órdenes del tamaño “mediano” (ésto debido a que fue el tamaño protagónico de su última campaña de marketing).

Ahora toca analizar los sabores y corroborar cuáles están brindando una mayor rentabilidad, y cuáles deberían salir o sustituirse por un nuevo sabor.

¿Qué debería hacer nuestra aggregation?

- ✓ Primero, una stage para filtrar las pizzas por su tamaño, ya que sólo nos interesa la campaña de pizzas medianas.
- ✓ Segundo, agrupar las pizzas por sabor para corroborar cuántos ejemplares se vendieron de dichos sabores.

Ejemplo: Aplicando nuestra primera aggregation

```
2 import mongoose from 'mongoose';
3
4 const environment = async() =>{
5   await mongoose.connect('mongodb+srv://CoderUser:123@coderccluster.w5adegs.mongodb.net/coderDatabase?retryWrites=true');
6   /**
7    * Nota cómo la operación "Aggregate" recibe un Array. Este Array se debe a que podemos colocar las stages que
8    * consideremos para resolver la petición que se nos ha hecho.
9    */
10  let orders = await orderModel.aggregate([
11    {
12      //Stage 1: Filtrar las órdenes para obtener sólo aquellas que tengan el tamaño mediano.
13      //Recordamos que match nos permitirá aplicar un filtro como cualquiera de los que hemos hecho.
14      $match: {size:"medium"}
15    },
16    {
17      //Stage 2: Agrupar por sabores y acumular el número de ejemplares de cada sabor.
18      //.$group necesitará crear un nuevo _id, éste corresponderá al "name" (sabor) de la pizza.
19      /**
20       * NOTA IMPORTANTE: Observa cómo utilizamos "$name", esta sintaxis significa que tomemos el valor "name" del
21       * documento en el cual se encuentre. así, podemos acceder a cualquier valor del documento, como "$quantity"
22       */
23      $group: {_id:"$name",totalQuantity:{sum:"$quantity"}}
24    }
25  ])
26  console.log(orders);
27 }
```

Ejemplo: Analizando el resultado de nuestra operación

Análisis de resultados

Una vez finalizada nuestra primera aggregation, el resultado es:

Notamos cómo es posible realizar operaciones más complejas que sólo una búsqueda con el uso de aggregations.

Una vez obteniendo los resultados, el equipo de Marketing determinará cuál es la mejor decisión según su contexto.

```
[  
  { _id: 'Pepperoni', totalQuantity: 20 },  
  { _id: 'Cheese', totalQuantity: 50 },  
  { _id: 'Vegan', totalQuantity: 10 }  
]
```

¡Pero espera!

Nuestra Marketing lead desea unos cambios de último momento.

Nuestra líder de campaña necesita nuevos cambios en la forma de entregar nuestra información:

- Primero, desea que los resultados se entreguen de mayor a menor por cantidad de ventas.
- Segundo, desea que los resultados se almacenen en una nueva colección "reports" con el fin de poder consultar el reporte para análisis futuros.

¡Al cliente lo que pida! Aquí va la magia de las aggregations



La maravilla de un aggregation

La ventaja de utilizar aggregations, es que una vez que tenemos definida la estructura de nuestras primeras operaciones, si más adelante nos piden hacer algún cambio o que haga algunas operaciones extras, simplemente hay que meterlo al arreglo de stages que ya tenemos desarrollado

En este caso, la aggregation tendrá 4 stages adicionales, según lo solicitado:

- ✓ \$sort: Para poder ordenar los campos que tenemos actualmente.
- ✓ \$group (Sí, otra vez): Para poder agrupar ahora todos nuestros resultados en un único campo.
- ✓ \$project: Para poder crear un documento nuevo a partir del arreglo de resultados de nuestra aggregation (además de asignarle un _id nuevo).
- ✓ \$merge: Para poder escribir este resultado en la colección nueva de "orders"

Ejemplo: Agregando las nuevas stages (Parte I)

```
23     $group: { _id: "$name", totalQuantity: { $sum: "$quantity" } }
24   },
25   {
26     //¡Nueva stage! Stage 3: Ordenar los documentos ya agrupados de mayor a menor
27     $sort: { totalQuantity: -1 }
28   },
29   [
30     /**
31      * ¡Nueva stage! Stage 4: Guardaremos todos los documentos de la agregación, en un nuevo documento dentro
32      * de un arreglo con el nombre "orders". ¡De otra manera, los resultados se guardarán sueltos en la
33      * colección! $push indica que se guardarán en un arreglo, y $$ROOT toma todo el documento para insertar
34      * (sin $$ROOT, tendríamos que especificar atributo por atributo qué queremos agregar, algo tardado)
35      */
36     $group: { _id:1, orders: { $push: "$$ROOT" } }
37   ],
38   {
39     /**
40      * ¡Nueva stage! Stage 5: Una vez que agrupamos todos los elementos en un único documento, utilizaremos
41      * $project para generar un nuevo ObjectId, así podremos guardarlo sin haber coincidencias.
42      * Al utilizar un $project, si colocamos _id:0 esto significa que genere un ObjectId propio
43      */
44     $project:{
45       "_id":0,
46       orders:"$orders",
47     }
48   },
```

Ejemplo: Agregando las nuevas stages (Parte II)

```
36     $group: { _id:1, orders: { $push: "$$ROOT" } }
37   },
38   {
39     /**
40     * ¡Nueva stage! Stage 5: Una vez que agrupamos todos los elementos en un único documento, utilizaremos
41     * $project para generar un nuevo ObjectId, así podremos guardarlo sin haber coincidencias.
42     * Al utilizar un $project, si colocamos _id:0 ésto significa que genere un ObjectId propio
43     */
44     $project:{
45       "_id":0,
46       orders:"$orders",
47     }
48   },
49   {
50     /**
51     * Stage final SIEMPRE: Agregar los elementos a la colección "reports". Si se deseara agregar un nuevo paso,
52     * recuerda que igualmente esta stage siempre deberá ser la última
53     */
54     $merge: {
55       into: 'reports'
56     }
57   }
58 ]
59 }
60 environment();
```

¡Reporte generado!

```
{ field: value }  
  
_id: ObjectId("62d42d1c0bfb34c6c241f860")  
orders: Array  
  0: Object  
    _id: "Cheese"  
    totalQuantity: 50  
  1: Object  
    _id: "Pepperoni"  
    totalQuantity: 20  
  2: Object  
    _id: "Vegan"  
    totalQuantity: 10
```

Gracias a los nuevos pasos de nuestra aggregation, ahora no sólo podemos obtener la información de los pedidos procesados y los distintos sabores de cada conjunto de órdenes de pizza medianas. Sino que acabamos de armar un sistema para poder generar reportes, para análisis de datos de otros sectores.

¡Acabamos de resolver un problema del día a día en el mundo laboral!



Para pensar

¿Qué cambios deberíamos hacer si me piden dinamizar el tamaño de las pizzas? Es decir, poder obtener reportes para cualquier tamaño que se nos solicite.



Agrupación de estudiantes

Duración: 15–20 min



ACTIVIDAD EN CLASE

Agrupación de estudiantes

Realizar las siguientes consultas en una colección de estudiantes.

Los estudiantes deben contar con los datos:

- ✓ first_name : Nombre
- ✓ last_name : Apellido
- ✓ email: correo electrónico
- ✓ gender: género
- ✓ grade: calificación
- ✓ group : grupo

(El profesor puede proporcionarte algunos datos de prueba)



ACTIVIDAD EN CLASE

Agrupación de estudiantes

Una vez generados tus datos de prueba:

1. Obtener a los estudiantes agrupados por calificación del mejor al peor
2. Obtener a los estudiantes agrupados por grupo.
3. Obtener el promedio de los estudiantes del grupo 1B
4. Obtener el promedio de los estudiantes del grupo 1A
5. Obtener el promedio general de los estudiantes.
6. Obtener el promedio de calificación de los hombres
7. Obtener el promedio de calificación de las mujeres.



Para pensar

¿Qué diferencia habría entre un mongoose-paginate y un skip, offset, limit en una consulta find?

¿Qué debo elegir?



Hands on lab

En esta instancia de la clase **crearemos una paginación elemental con los estudiantes del ejercicio pasado**, a partir de un ejercicio práctico

¿De qué manera?

El profesor te demostrará cómo hacerlo y tú lo puedes ir replicando en tu computadora. Si surgen dudas las puedes compartir para resolverlas en conjunto de la mano de los tutores.

Tiempo estimado: **35-40 minutos**

Sistema de paginación de estudiantes

¿Cómo lo hacemos? **Se creará una vista simple con Handlebars donde se podrán mostrar los estudiantes**

- ✓ Los estudiantes serán mostrados en la vista `"/students"`
- ✓ Debe existir un enlace `"Anterior"` para regresar a los estudiantes anteriores, **siempre que haya una página anterior**
- ✓ Debe existir un enlace `"Siguiente"` para continuar con la paginación de estudiantes, **siempre que haya una página siguiente**
- ✓ Debe indicarse la página actual.
- ✓ Todo debe vivir en un servidor de express escuchando en el puerto 8080.