

# Esta clase va a ser

- grabada  
a

**Clase 17.** PROGRAMACIÓN BACKEND

# Mongo avanzado II

# Temario

16

## Mongo avanzado I

- ✓ Teoría de indexación
- ✓ Manejo de populations en Mongo

17

## Mongo avanzado II

- ✓ [Aggregations](#)
- ✓ [Paginación con 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

**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

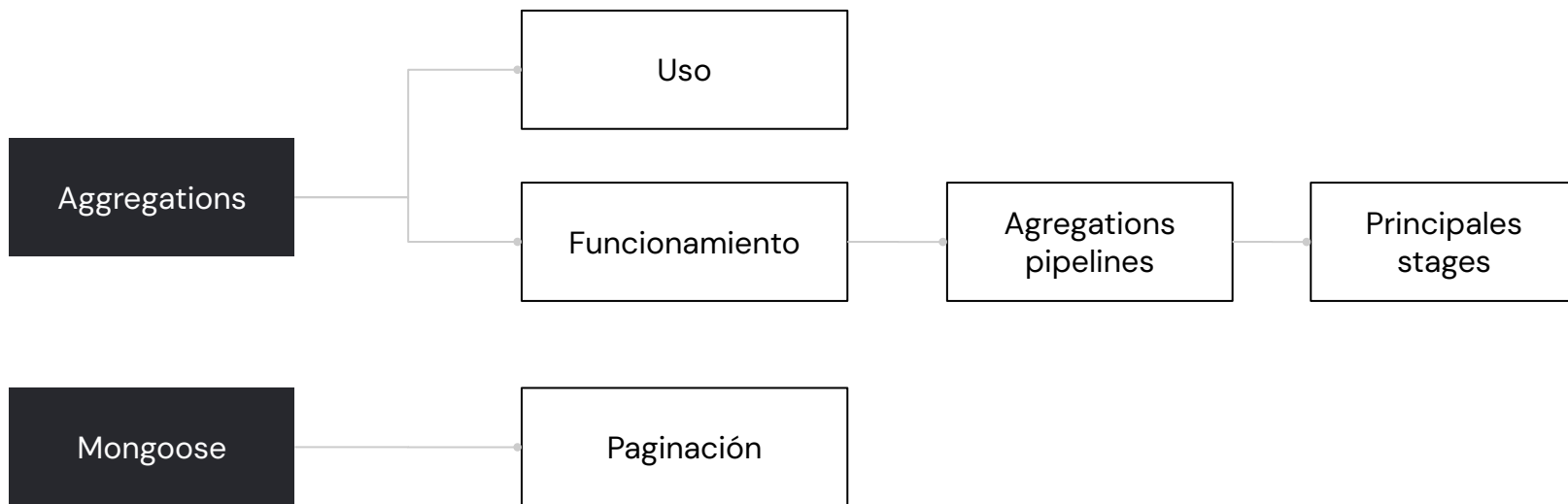
**population:** Operación que permite transformar la referencia de un documento en su documento correspondiente en la colección indicada.

**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.



## MAPA DE CONCEPTOS



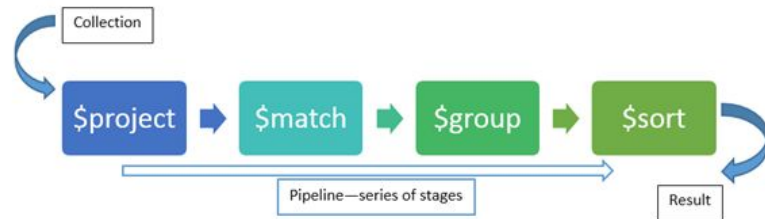
# Aggregations

# Aggregation

Consiste en la realización de múltiples operaciones, generalmente sobre múltiples documentos.

Pueden utilizarse para:

- ✓ Agrupar documentos con base en un criterio específico.
- ✓ Realizar alguna operación sobre dichos documentos, con el fin de obtener un solo resultado.
- ✓ Analizar cambios de información con el paso del tiempo.





# Funcionamiento

Los aggregation pipelines consistirán en un conjunto de pasos (stages), donde cada paso corresponderá a una operación a realizar.

Podemos definir tantas stages como necesitemos con el fin de llegar a los resultados esperados.

Los documentos resultantes de la stage que finalice, se utilizan como “input” de la siguiente stage, y así sucesivamente hasta llegar al final.

Un ejemplo de un pipeline de aggregation puede ser:

1. Primero filtra los documentos que tengan un valor x mayor a 20
2. Luego ordénalos de mayor a menor
3. Luego en un nuevo campo devuelve el valor máximo
4. Luego en un nuevo campo devuelve el valor mínimo
5. Luego en un nuevo campo devuelve la suma total de todos los documentos

# Principales stages disponibles en un aggregation pipeline

`$count` : Cuenta el número de documentos disponibles que se encuentren en la stage actual.

`$group`: Permite agrupar los documentos disponibles en nuevos grupos según un criterio especificado. **cada grupo cuenta con un `_id` nuevo**, además de los valores acumulados.

`$limit`: Limita el número de documentos que saldrán de dicha stage.

`$lookup`: Permite realizar un “left join” (combinación de campos), de una colección de la misma base de datos a los documentos de la stage actual.

Puedes revisar la [lista completa de stages](#)

# Principales stages disponibles en un aggregation pipeline

`$set / $addFields` : Agregan una nueva propiedad a los documentos que se encuentren en dicha stage.

`$skip`: Devuelve sólo los documentos que se encuentren después del offset indicado.

`$sort`: Ordena los documentos en la stage actual.

`$match`: Devuelve sólo los documentos que cumplan con un criterio de búsqueda, podemos colocar filtros comunes aquí

`$merge`: escribe los resultados del pipeline en una colección. Debe ser la última stage del pipeline para poder funcionar.

Puedes revisar la [lista completa de stages](#)

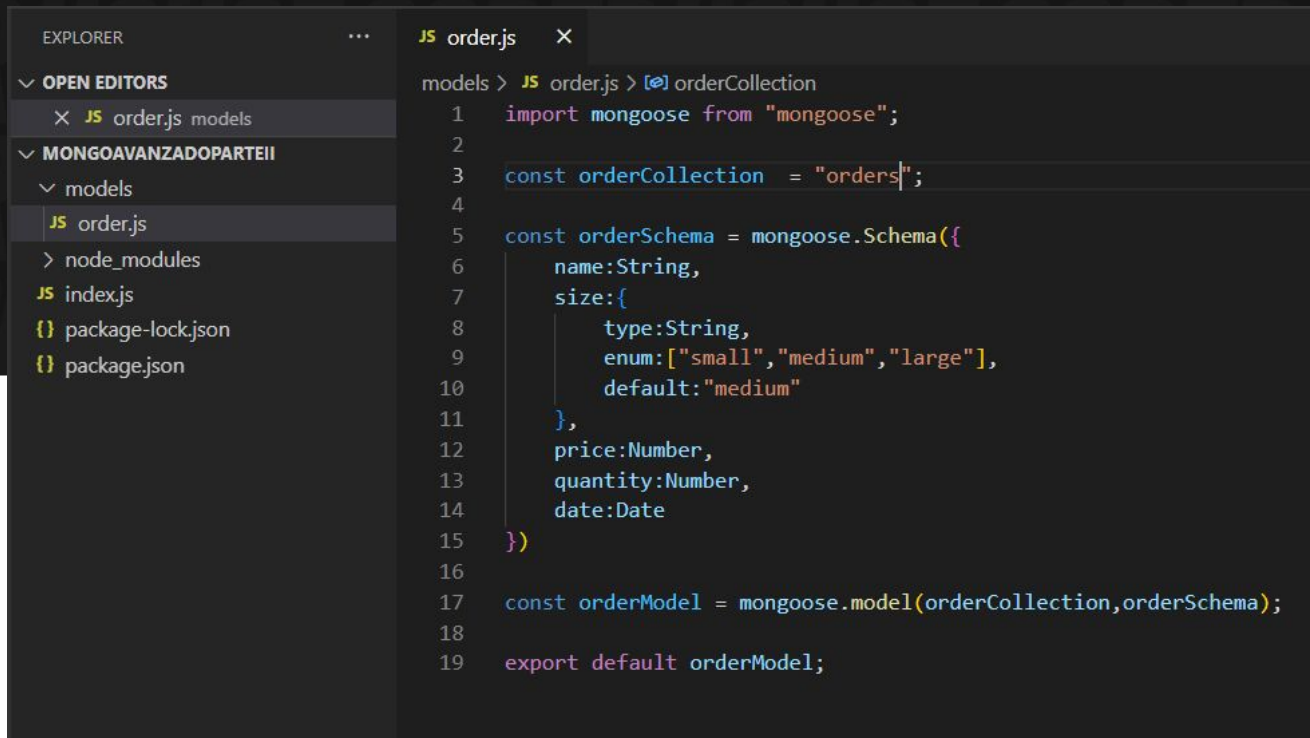


## Ejemplo en vivo

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.



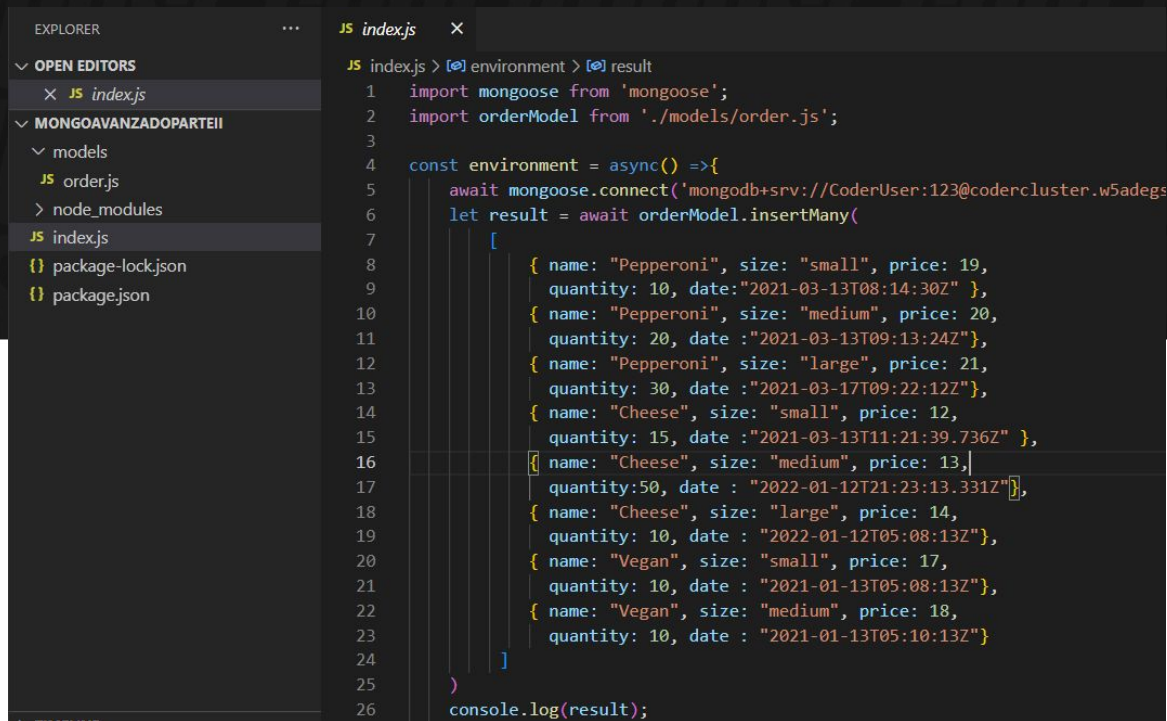
The image shows a VS Code editor window with a dark theme. On the left, the Explorer sidebar is open, showing a file tree with the following structure:

- EXPLORED
- OPEN EDITORS
  - JS order.js models
- MONGOAVANZADOPARTEII
  - models
    - JS order.js (selected)
  - node\_modules
  - JS index.js
  - package-lock.json
  - package.json

The main editor area shows the content of `order.js` in the `models` directory. The code is as follows:

```
models > JS order.js > [1] orderCollection
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



The image shows a VS Code editor interface. On the left, the Explorer sidebar displays the project structure:

- EXPLORED
- OPEN EDITORS
  - JS index.js
- MONGOAVANZADOPARTEI
  - models
    - order.js
  - node\_modules
  - JS index.js
  - package-lock.json
  - package.json

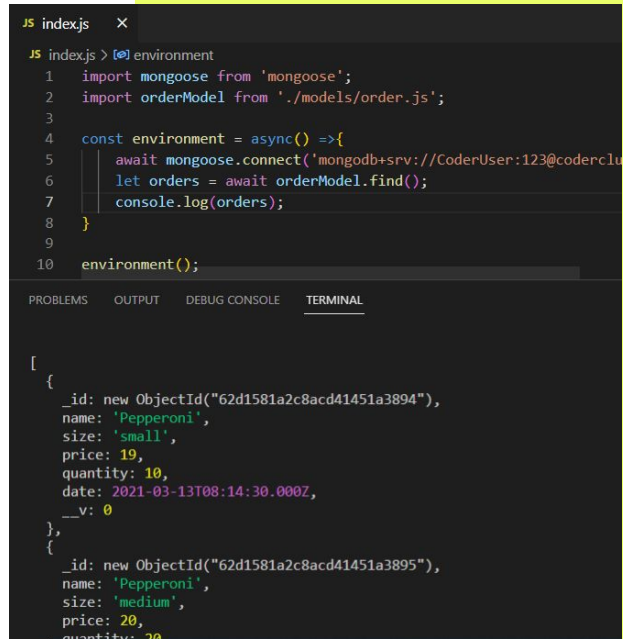
The main editor area shows the content of `index.js`:

```
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   },
49 ]
```

# 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();
```

Ejemplo: Analizando resultado final

# ¡Reporte generado!

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!

```
{ 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
```



## 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.



# Break

¡10 minutos y volvemos!

# Paginación con mongoose

# Paginación

Cuando recién trabajamos con nuestros primeros datos, es maravilloso ver cómo nuestras búsquedas pueden devolvernos todos los datos que necesitamos.

Sin embargo, esta “maravilla” comienza a convertirse en un problema cuando el número de datos que tenemos incrementa considerablemente.

Recordemos que los datos que nosotros obtenemos, al final tenemos que enviarlos a través del internet, para que el cliente la utilice.

¿Qué tan lento será el proceso de enviar 5000 usuarios en una sola vuelta? ¿Y 10 mil?

# Paginación como control de resultados



**¡Pensemos en páginas!**

Aprender a pensar en páginas nos permitirá segmentar los resultados en pequeños trozos de información, brindándonos al final una referencia de **en qué página estamos, cuál es la página anterior y cuál la siguiente**

# Paginación utilizando mongoose-paginate-v2



# mongoose-paginate-v2

mongoose-paginate-v2 es un **plugin** para mongoose que nos permitirá realizar paginaciones eficientes para los modelos que nosotros especifiquemos.

Cuenta con una gran optimización y agregado de funcionalidades frente a su v1

Para poder comenzar a utilizarlo sólo tenemos que instalarlo con npm:

```
npm install mongoose-paginate-v2
```



v2  
mongoose **PAGINATE**

a custom pagination library for mongoose (NodeJS)

<https://github.com/aravindnc/mongoose-paginate-v2>

# Nociones de mongoose-paginate-v2

- ✓ docs: Los documentos devueltos en la página
- ✓ totalDocs: Los documentos totales de la consulta realizada.
- ✓ limit: Límite de resultados por página.
- ✓ page: Página actual en la que nos encontramos
- ✓ totalPages: Páginas totales que pueden ser solicitadas en la búsqueda.
- ✓ hasNextPage: Indica si es posible avanzar a una página siguiente.
- ✓ nextPage: Página siguiente en la búsqueda
- ✓ hasPrevPage: Indica si es posible retroceder a una página anterior.
- ✓ prevPage: Página anterior en la búsqueda.
- ✓ pagingCounter: Número de documento en relación con la página actual.

Para poder utilizarlo, basta con importar el módulo de paginate en el Schema donde lo utilizaremos.

```
models > JS users.js > ...  
1  ✓ import mongoose from 'mongoose';  
2  import mongoosePaginate from 'mongoose-paginate-v2';  
3  const usersCollection = 'users';  
4  
5  const usersSchema = mongoose.Schema({  
6    first_name:String,  
7    last_name:String,  
8    email:String,  
9    gender:String,  
10  });  
11  |  
12  const usersModel = mongoose.model(usersCollection,usersSchema);  
13  
14  export default usersModel;
```

# Entonces, antes de instanciar el modelo, colocamos un “plugin” de paginación a nuestro Schema

```
models > JS users.js > ...  
1  import mongoose from 'mongoose';  
2  import mongoosePaginate from 'mongoose-paginate-v2';  
3  const usersCollection = 'users';  
4  
5  const usersSchema = mongoose.Schema({  
6    first_name:String,  
7    last_name:String,  
8    email:String,  
9    gender:String,  
10  })  
11  usersSchema.plugin(mongoosePaginate);  
12  const userModel = mongoose.model(usersCollection,usersSchema);  
13  
14  export default userModel;
```

El modelo ahora puede usar el método “paginate”. El primer argumento es el filtro, y el segundo son las opciones.

```
1 import userModel from "../models/users.js";
2 import mongoose from 'mongoose';
3
4 const environment = async () => {
5   ⚡await mongoose.connect('mongodb+srv://CoderUser:123@codercluster.w5adegs.mc
6     let users = await userModel.paginate({gender:"Female"},{limit:20,page:1})
7     console.log(users);
8   }
9
10  environment();
11
```

Ahora, en el campo "docs" obtendremos los resultados que hayamos solicitado. ¡Pero hay más!

No solo obtenemos resultados, sino que tenemos toda la información sobre la paginación realizada, y cómo podemos continuarla.

```
{
  docs: [
    {
      _id: new ObjectId("62d45f0fca562e325737f631"),
      first_name: 'Orly',
      last_name: 'Restall',
      email: 'orestall0@multiply.com',
      gender: 'Female'
    },
    {
      id: new ObjectId("62d45f0fca562e325737f632")
    }
  ]
}
```

```
    },
    ],
    totalDocs: 2221,
    limit: 20,
    totalPages: 112,
    page: 1,
    pagingCounter: 1,
    hasPrevPage: false,
    hasNextPage: true,
    prevPage: null,
    nextPage: 2
  }
}
```



## 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 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.



# Segunda pre-entrega de tu Proyecto final

Deberás entregar el proyecto que has venido armando, cambiando persistencia en base de datos, además de agregar algunos endpoints nuevos a tu ecommerce



# Profesionalizando la BD

### Objetivos generales

- ✓ Contarás con Mongo como sistema de persistencia principal
- ✓ Tendrás definidos todos los endpoints para poder trabajar con productos y carritos.

### Objetivos específicos

- ✓ Profesionalizar las consultas de productos con filtros, paginación y ordenamientos
- ✓ Profesionalizar la gestión de carrito para implementar los últimos conceptos vistos.

### Formato

- ✓ Link al repositorio de Github, sin la carpeta de node\_modules

### Sugerencias

- ✓ Permitir comentarios en el archivo
- ✓ La lógica del negocio que ya tienes hecha no debería cambiar, sólo su persistencia.
- ✓ Los nuevos endpoints deben seguir la misma estructura y lógica que hemos seguido.



## ENTREGA DEL PROYECTO FINAL

### Se debe entregar

- ✓ Con base en nuestra implementación actual de productos, modificar el método GET / para que cumpla con los siguientes puntos:
  - Deberá poder recibir por query params un limit (opcional), una page (opcional), un sort (opcional) y un query (opcional)
    - -limit permitirá devolver sólo el número de elementos solicitados al momento de la petición, en caso de no recibir limit, éste será de 10.
    - page permitirá devolver la página que queremos buscar, en caso de no recibir page, ésta será de 1
  - query, el tipo de elemento que quiero buscar (es decir, qué filtro aplicar), en caso de no recibir query, realizar la búsqueda general
  - sort: asc/desc, para realizar ordenamiento ascendente o descendente por precio, en caso de no recibir sort, no realizar ningún ordenamiento



## ENTREGA DEL PROYECTO FINAL

### Se debe entregar

- ✓ El método GET deberá devolver un objeto con el siguiente formato:

```
{  
  status: success/error  
  payload: Resultado de los productos solicitados  
  totalPages: Total de páginas  
  prevPage: Página anterior  
  nextPage: Página siguiente  
  page: Página actual  
  hasPrevPage: Indicador para saber si la página  
  previa existe  
  hasNextPage: Indicador para saber si la página  
  siguiente existe.  
  prevLink: Link directo a la página previa (null si  
  hasPrevPage=false)  
  nextLink: Link directo a la página siguiente (null si  
  hasNextPage=false)  
}
```

- ✓ Se deberá poder buscar productos por categoría o por disponibilidad, y se deberá poder realizar un ordenamiento de estos productos de manera ascendente o descendente por precio.



## ENTREGA DEL PROYECTO FINAL

### Se debe entregar

- ✓ Además, agregar al router de carts los siguientes endpoints:
  - DELETE api/carts/:cid/products/:pid deberá eliminar del carrito el producto seleccionado.
  - PUT api/carts/:cid deberá actualizar el carrito con un arreglo de productos con el formato especificado arriba.
  - PUT api/carts/:cid/products/:pid deberá poder actualizar SÓLO la cantidad de ejemplares del producto por cualquier cantidad pasada desde req.body
  - DELETE api/carts/:cid deberá eliminar todos los productos del carrito
  - Esta vez, para el modelo de Carts, en su propiedad products, el id de cada producto generado dentro del array tiene que hacer referencia al modelo de Products. Modificar la ruta /:cid para que al traer todos los productos, los traiga completos mediante un "populate". De esta manera almacenamos sólo el Id, pero al solicitarlo podemos desglosar los productos asociados.



## ENTREGA DEL PROYECTO FINAL

### Se debe entregar

- ✓ Crear una vista en el router de views  `'/products'` para visualizar todos los productos con su respectiva paginación. Cada producto mostrado puede resolverse de dos formas:
  - Llevar a una nueva vista con el producto seleccionado con su descripción completa, detalles de precio, categoría, etc. Además de un botón para agregar al carrito.
  - Contar con el botón de “agregar al carrito” directamente, sin necesidad de abrir una página adicional con los detalles del producto.
- ✓ Además, agregar una vista en  `'/carts/:cid (cartId)'` para visualizar un carrito específico, donde se deberán listar SOLO los productos que pertenezcan a dicho carrito.

¿Preguntas?



**Opina y valora**  
esta clase

**Muchas gracias.**

# Resumen de la clase hoy

- ✓ Concepto de Aggregation
- ✓ Caso práctico de Aggregation
- ✓ Concepto de paginación
- ✓ Caso práctico de paginación

**#DemocratizandoLaEducación**