
# UNIVERSIDAD DE GRANADA

## Diseño y Desarrollo de Sistemas de Información

## Trabajo del Tema 4, lenguajes NoSQL

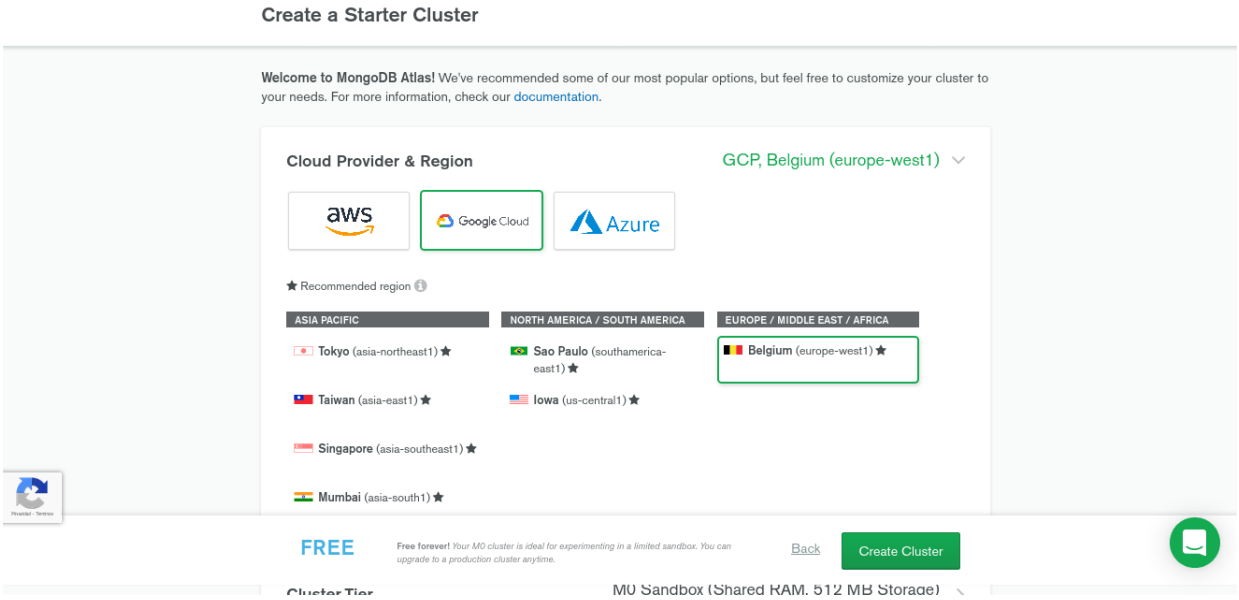
**Javier Expósito Martínez, Inés Nieto Sánchez, Laura Sánchez Sánchez, Leire Requena Garcia, Clara María Romero Lara**

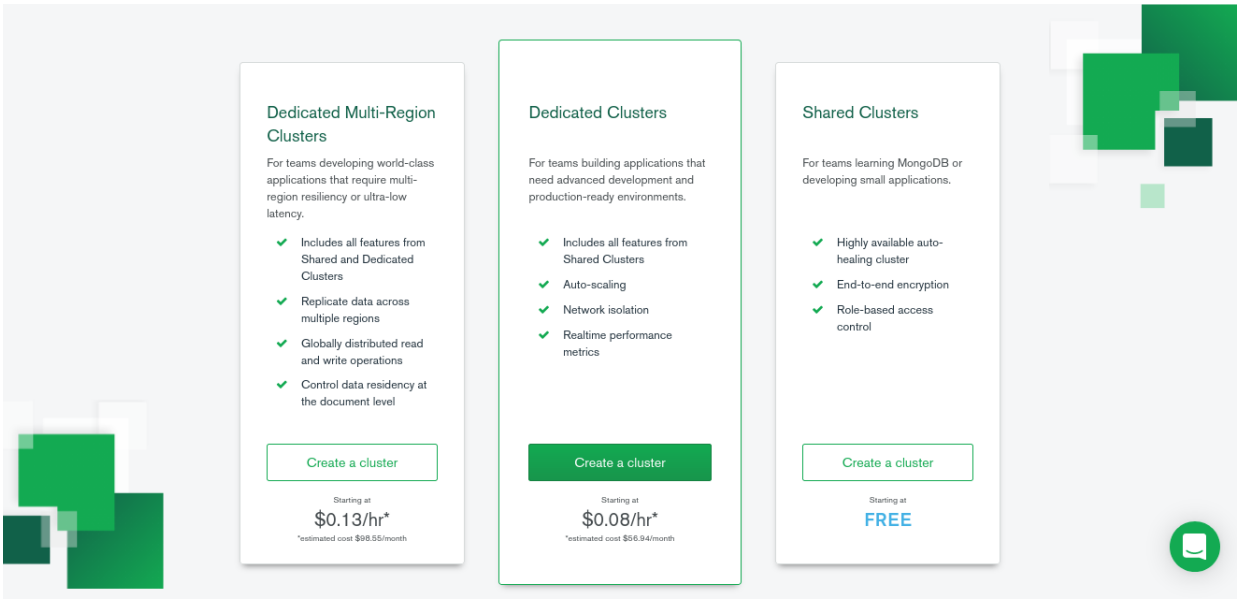
9 de enero de 2021

# Índice general

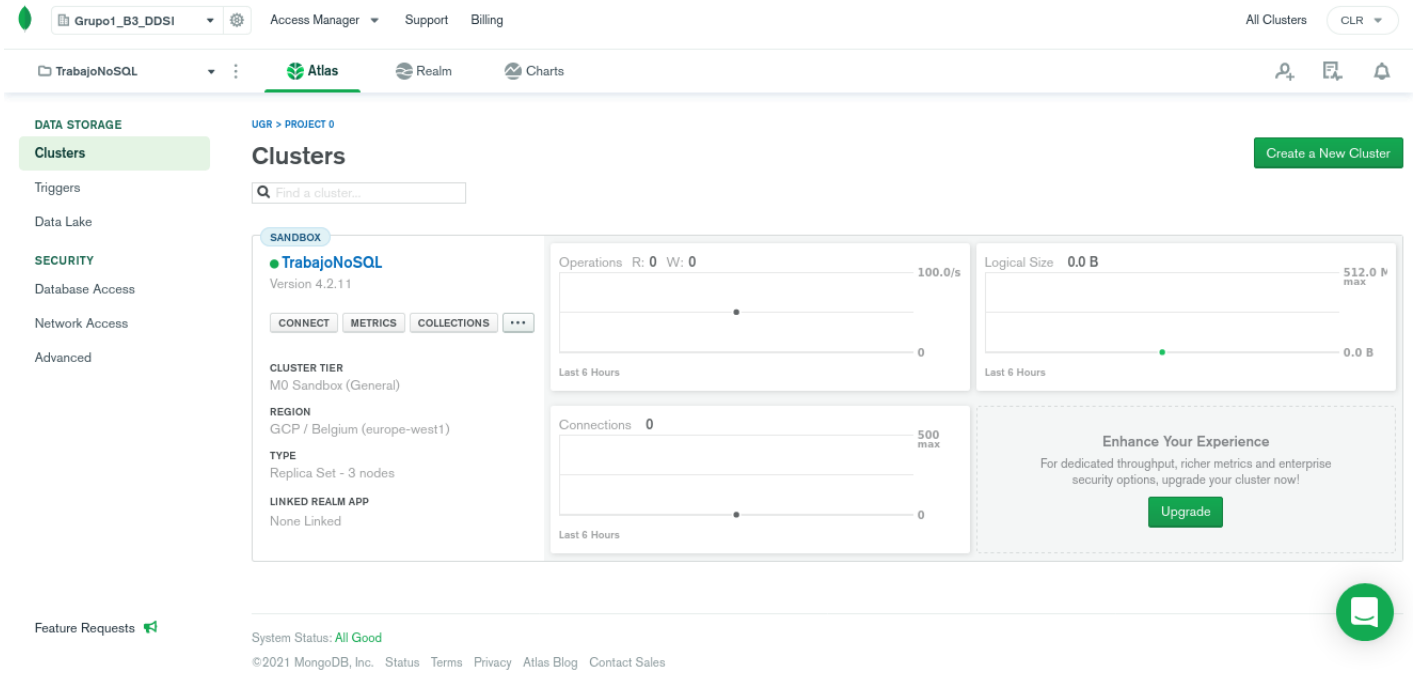
# Apartado 1:
# Proceso de instalación

El lenguaje NoSQL que hemos escogido para el desarrollo de este trabajo es MongoDB, más particularmente su servicio Atlas: un clúster en la nube que, aun con alternativas de pago, nos permite trabajar los contenidos de este trabajo sin limitaciones. Lo primero es registrarnos: tras introducir nuestros datos personales, tendremos que seleccionar algunas opciones para nuestro clúster, entre ellas el servidor que nos va a hostear, la región de este, o elegir un plan de pago según nuestras necesidades.

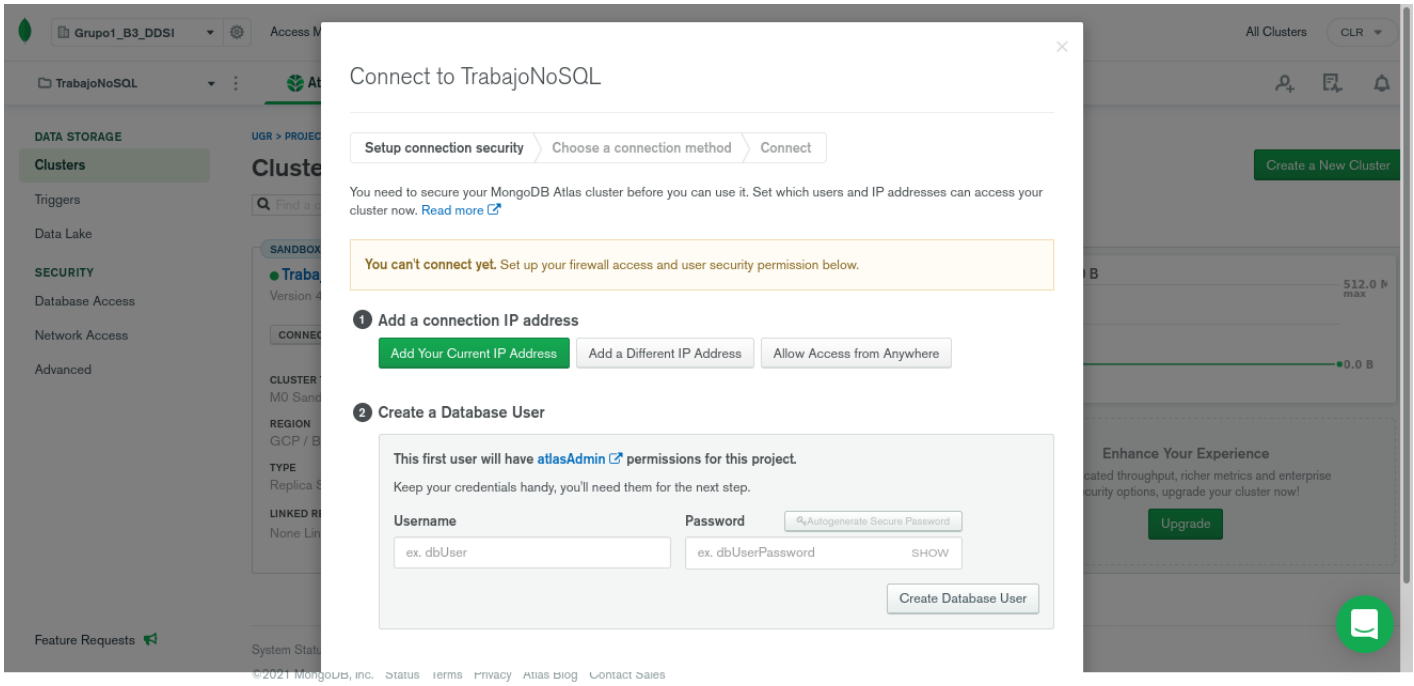


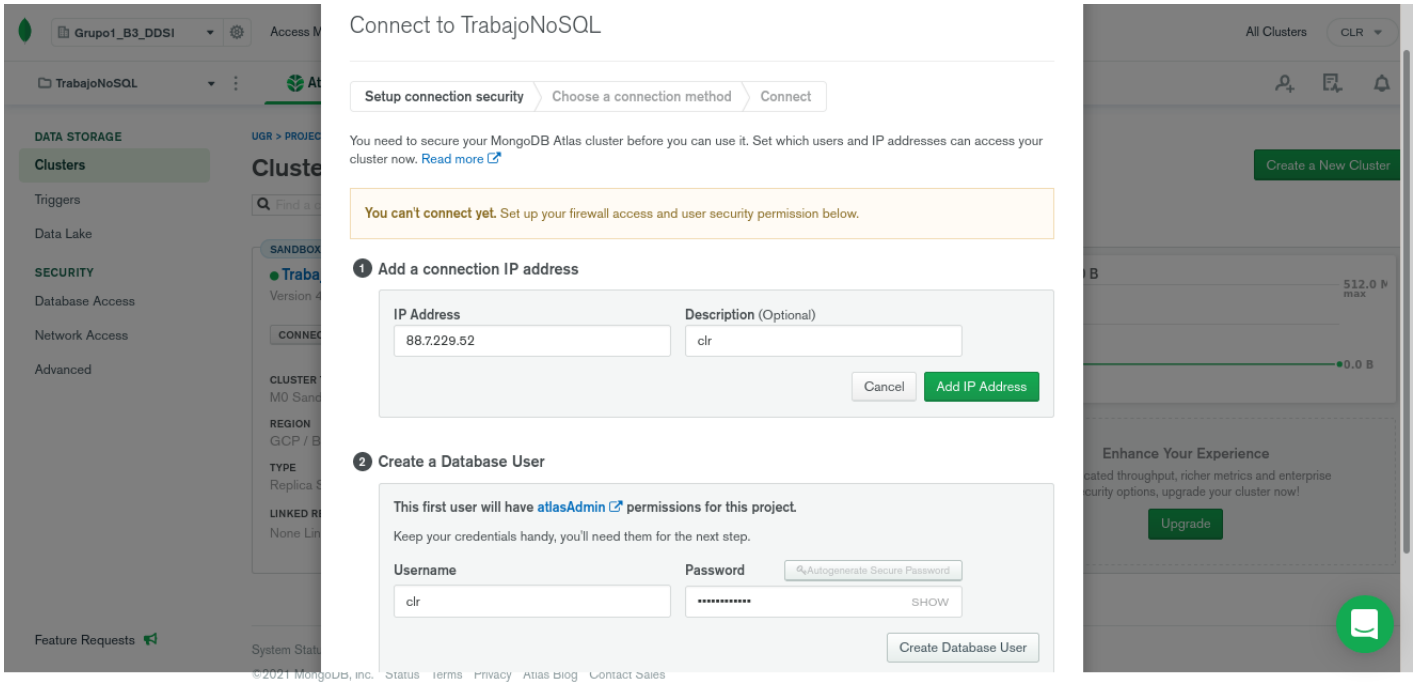
Para este trabajo hemos escogido las opciones gratuítas: clúster compartido, y un servidor de Google Cloud en Bélgica. Una vez confirmemos la configuración, se creará nuestro clúster. Esto tomará algo menos de 5 minutos.

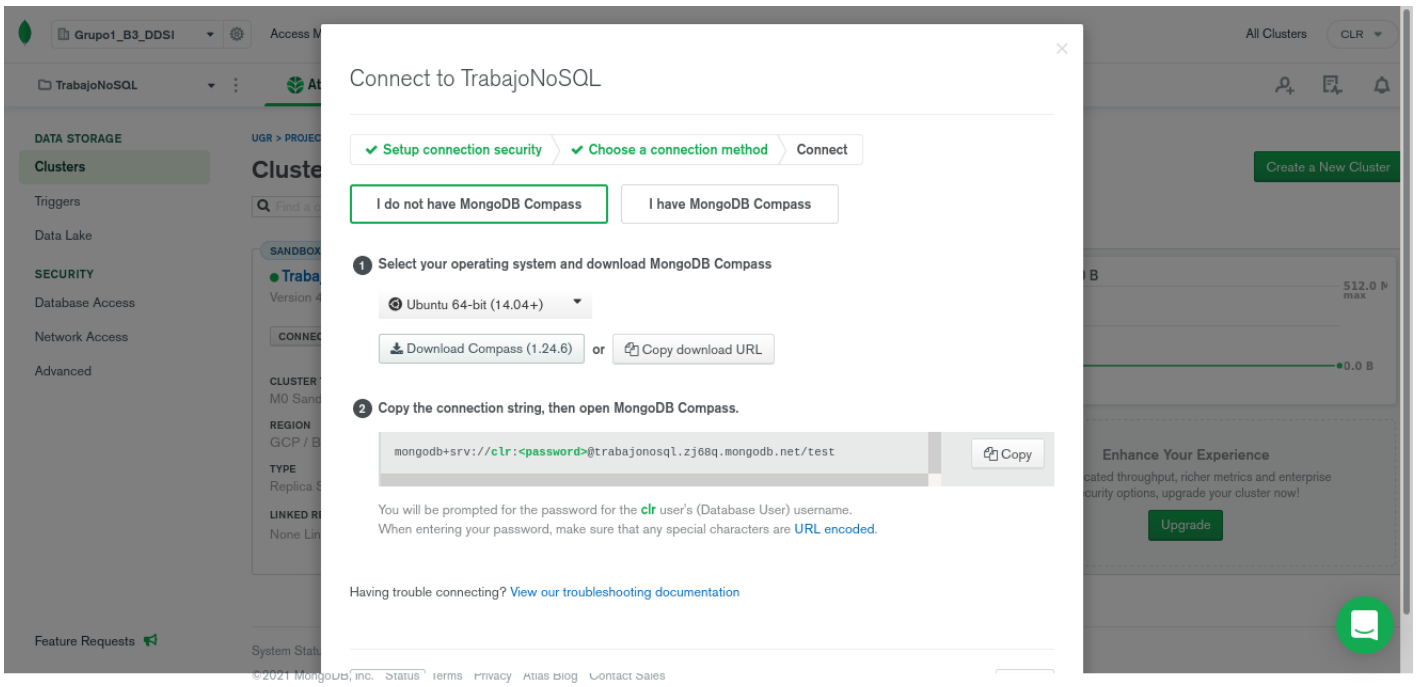



Una vez desplegado el clúster, tenemos que conectarnos. Desde el botón Connect añadiremos nuestra dirección IP. También crearemos un usuario: si somos el primer user, recibiremos por defecto permisos de administrador Atlas.

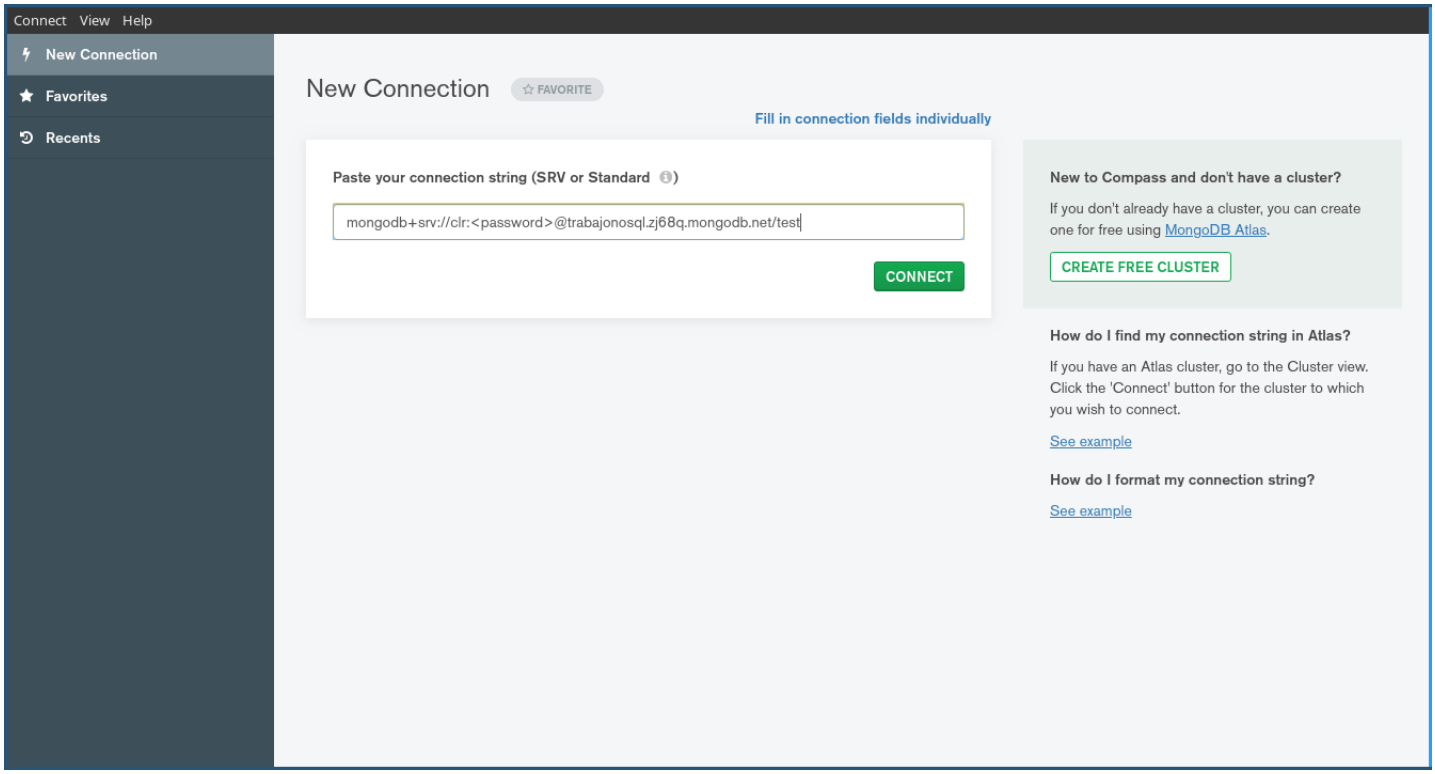


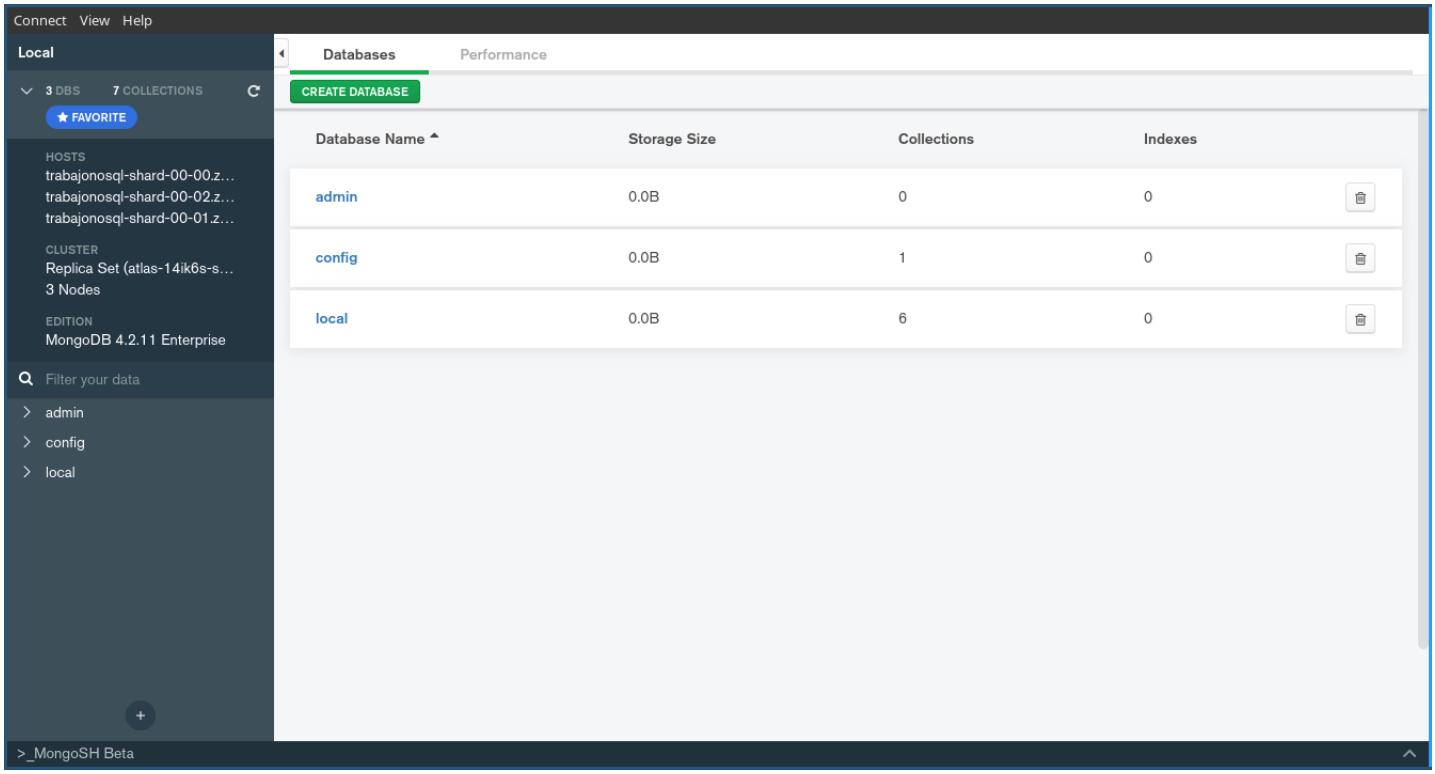
A continuación escogemos como nos vamos a conectar al clúster: las opciones son la shell de mongoDB (y ejecutar desde la terminal), seleccionar un driver para aplicaciones en python o javascript, y su GUI, MongoDB Compass. Para facilitar el desarrollo de este trabajo hemos escogido Compass.



Descargamos Compass y introducimos la conexión que nos ha facilitado Atlas. Ya tenemos nuestra base de datos conectada al clúster de Atlas, y cualquier usuario que registremos desde Atlas a nuestro proyecto podrá conectarse al clúster desde Compass y trabajar en él.



# Apartado 2:
# Descripción de DDL y DML

En SQL, estos comandos componen lo que tradicionalmente conocemos como DDL (Data Definition Language) y DML (Data Manipulation Language). Veremos sus equivalentes en MongoDB, pero antes haremos un breve inciso para hablar sobre las diferencias entre el modelo relacional y el modelo de documentos.

Antes de crear una base de datos basada en el modelo de documentos tenemos que dejar atrás las restricciones del modelo relacional: un documento y sus campos no equivalen a una tabla y sus columnas, y las restricciones de claves son mucho más laxas.

Es realmente importante tener esto en mente al desplegar una base de datos NoSQL, ya que toda la velocidad que ganamos se debe a que no se aplican tantas restricciones. Una enorme parte de las comprobaciones que hagamos para mantener la integridad del sistema se harán desde la aplicación externa, no desde la base de datos. Con estos conceptos claros, veamos como se adapta el DDL y DML:

## DDL

- **CREATE**: No es estrictamente necesario explicitar la creación de una colección (el equivalente a las tablas), sino que al introducir el primer `insert()` se creará la colección.

  En mongoDB, la orden equivalente a un CREATE es `db.collection.createCollection(nombre, opciones)`. Las opciones usadas han sido capped (colección de tamaño estático), size (tamaño de una colección capped) y max (número máximo de documentos en esta).

- **ALTER**: No existe como tal. Recordamos la diferencia con el modelo relacional: no estamos usando tablas. Si queremos alterar la estructura de un documento añadiendo un nuevo campo, lo haremos con update().

  La orden que hemos utilizado para emular un ALTER es update con el operador `$set`, para añadir un campo a todos los documentos: `db.collection.update({condiciones}, {$set : {nuevo campo}},opciones)` . En las condiciones indicamos qué documentos queremos actualizar, así que si lo dejamos vacío serán todos. Las opciones a configurar son upsert (crear un nuevo documento si ninguno coincide con las condiciones), y multi (aplicar los cambios a todos los documentos que coincidan).

- **DROP**: Se mantiene igual respecto a SQL. La orden es `db.Collection.drop()` y elimina la colección indicada.

# DML

- **INSERT**: Como hemos dicho, si es la primera instancia implica la declaración de la colección. Si al nombre de un campo le añadimos _id, este campo será considerado automáticamente clave primaria. Para la sentencia, sería simplemente `db.collection.insert({'campo1':valor1, 'campo2':valor2,...}`
- **SELECT**: Devuelve los documentos que coincidan con los criterios indicados. La orden es `db.collection.find(criter proyeccion)`, donde criterio son los criterios de búsqueda; y proyección los campos que devuelve de los documentos coincidentes.
- **UPDATE**: Para actualizar información de un campo, se mantiene de forma muy similar a SQL, y además cuenta con los añadidos vistos en ALTER o DELETE. Emplea operadores para especificar el tipo de actualización (`$set, $unset...`).

  La sentencia utilizada para actualizar un campo sería `db.collection.update( {condicion}, {$set:{ 'Campo a actualizar': valor nuevo}}, opciones )`.
- **DELETE**: Existen operadores como `$unset` para update(), que eliminan un campo de un documento; o `deleteOne()`, que elimina un documento de la colección. Para eliminar un campo: `db.collection.update( {condicion}, {$unset:{ 'Campo a borrar': ''}}, opciones )`, donde el nuevo valor del campo es simplemente conjunto vacío. Para eliminar un documento: `db.collection.deleteOne( {condicion} )`.

# Apartado 3:
# Creación, inserción y modificación de estructuras

De nuevo, hemos creado una base de datos muy similar a la del Seminario 2. Cuenta con tres colecciones: `Stock(id, cantidad)`, `Clientes(id, nombre)` y `Pedidos(id, Cliente-id, Stock-id, Artículos, Fecha)`. A continuación, veamos como hemos desplegado la base de datos y algunas otras sentencias. Se incluye entre paréntesis el equivalente a SQL.

### Creación de las colecciones y documentos (CREATE + INSERT)

```
1  db.Stock.insert({"_id":1, "cantidad":10})
2  db.Stock.insert({"_id":2, "cantidad":31})
3  db.Stock.insert({"_id":3, "cantidad":6 })
4  db.Stock.insert({"_id":4, "cantidad":10})
5  db.Stock.insert({"_id":5, "cantidad":23})
6  db.Stock.insert({"_id":6, "cantidad":43})
7
8  db.Clientes.insert({"_id":1, "nombre":"Paco"})
9  db.Clientes.insert({"_id":2, "nombre":"Mari"})
10 db.Clientes.insert({"_id":3, "nombre":"Alex"})
11
12 db.Pedidos.insert({"_id":1,"Cliente_id":1,"Stock_id":2,"Articulos":3})
13 db.Pedidos.insert({"_id":2,"Cliente_id":1,"Stock_id":6,"Articulos":5})
14 db.Pedidos.insert({"_id":3,"Cliente_id":9,"Stock_id":9,"Articulos":3})
```

Es en esta tabla Pedido donde empezamos a ver diferencias difíciles de salvar respecto al modelo relacional: no existen restricciones de clave foránea. Esto quiere decir que el sistema no nos va a impedir registrar un pedido con un Cliente-id y Stock-id que no existe, como en la tercera sentencia. Solo se podría solucionar desde la implementación de una aplicación externa.

### Añadir un campo a todos los documentos de una colección (ALTER)

```
1  db.Pedidos.update({}, {\$set : {"fecha":new Date()}},{upsert:false, multi:true})
```

Vamos a profundizar un poco en esta sentencia para destacar la variedad de usos del `update()`: estamos filtrando sin especificar ningún criterio para tomar todos los documentos de la colección, creamos y insertamos simultáneamente el nuevo campo (fecha), y con la flag multi nos aseguramos que esto se aplique a todos los documentos.

### Actualizar información en documento (UPDATE)

```
db.Pedidos.update( { _id: 3 }, {\$set:{ "Cliente_id":2, "Stock_id": 4}} )
```

### Crear colección (CREATE)

```
db.createCollection("Pruebas", { capped : true, size : 5242880, max : 5000 } )
```

Como ya hemos mencionado, la primera llamada desde un método a una colección la crea automáticamente. Si decidimos llamar al método `createCollection()` es probablemente porque queremos algún parámetro específico para ésta, por lo que hemos creado una colección capada e indicado su tamaño y número máximo de documentos.

### Crear documento, eliminar campo del documento, eliminar documento (INSERT, DELETE)

```
db.Pruebas.insert( {"_id": 1, "test": "123"} )

db.Pruebas.update( { _id: 1 }, { \$unset: { test: ""} })

db.Pruebas.deleteOne( {"_id": 1} )
```

### Borrar colección (DROP)

```
db.Pruebas.drop()
```

### Buscar según campo (SELECT)

```
db.Clientes.find({"Nombre":"Alex"})
```

# Apartado 4:
# Descripción del mecanismo de conexión al SGBD

MongoDB cuenta con drivers para una multitud de lenguajes, y Atlas provee lineas de conexión generadas automáticamente para aplicaciones JavaScript y Python. Para conectarnos a nuestro clúster desde una aplicación Python seguiremos los siguientes pasos:

1. Desde una terminal, instalamos con Pip el driver Pymongo:

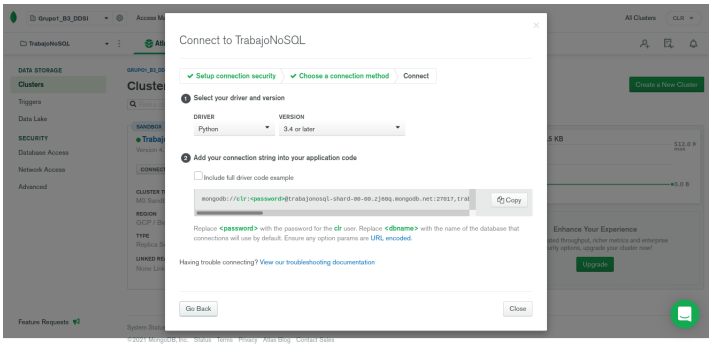
```
python -m pip install pymongo[snappy,gssapi,srv,tls]
```

2. Abrimos la shell de Python, importamos el paquete y comprobamos qué versión tenemos:

```
> python
> import pymongo
> pymongo.version
```

3. Desde Atlas, elegimos en nuestro clúster la opción conectar. Igual que para conectarnos con Compass, introduciremos nuestra IP y usuario.

4. Para el método de conexión escogemos conectar a aplicación. Desde el desplegable, escogemos Python y nuestra versión de Pymongo, la cual hemos consultado en el paso 2.




5. Importamos Pymongo y MongoClient a nuestro programa Python con el siguiente comando

```
from pymongo import MongoClient
```

6. En la línea que nos da Atlas (string de conexión), sustituímos `<password>` por nuestra contraseña, y `<dbname>` por la base de datos por defecto que queramos.

7. Una vez hemos la string de conexión tiene nuestra contraseña y base de datos, la pegamos dentro de la función MongoClient() en nuestro programa Python de la siguiente manera, sustituyendo la línea punteada:

```
client = MongoClient('- - - - - - - - - - - - - - - -')
```

# Apartado 5:
# ¿Sería adecuado para nuestro sistema del torneo de padel?

En este trabajo hemos desarrollado una pequeña base de datos que se aproxima al modelo relacional. Esto nos ha causado ciertos problemas de consistencia e integridad, y solo demuestra que no existe superioridad entre SQL y NoSQL: simplemente existen sistemas que encajan mejor en uno o en el otro.

La conclusión que sacamos tras el desarrollo de este trabajo es que no, no sería el mejor lenguaje para el desarrollo de nuestra práctica. El sistema del Torneo de Pádel cuenta con una gran cantidad de restricciones de claves foráneas que habríamos tenido que comprobar desde la aplicación, ya sea en Java o en otro lenguaje. Es un ejercicio altamente dependiente del sistema relacional difícil de traducir al de documentos.

Los lenguajes NoSQL tienen mucho que ofrecer, son extremadamente rápidos respecto a SQL y altamente flexibles, pero están hechos para otro tipo de aplicaciones que necesitan de esa velocidad y flexibilidad. Nuestro sistema no requiere altas velocidades ni va a sufrir grandes modificaciones en su estructura, además que es más dependiente de la base de datos que de la implementación en una aplicación.

# Apartado 6:
# Repositorio en GitHub

Los archivos fuente están publicados en github junto a un makefile por si se desa compilar el pdf en cualquier ordenador; lee el siguiente QR para acceder: