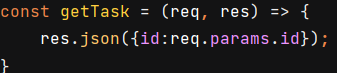
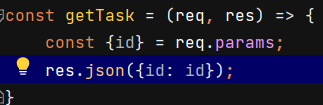
03 Task manager

1. Nó có nhiều cách để gọi và trả ra chuỗi json:

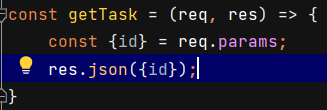
C1



C2

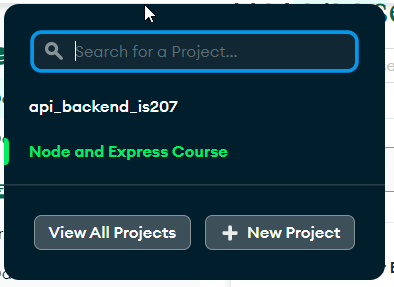


C3

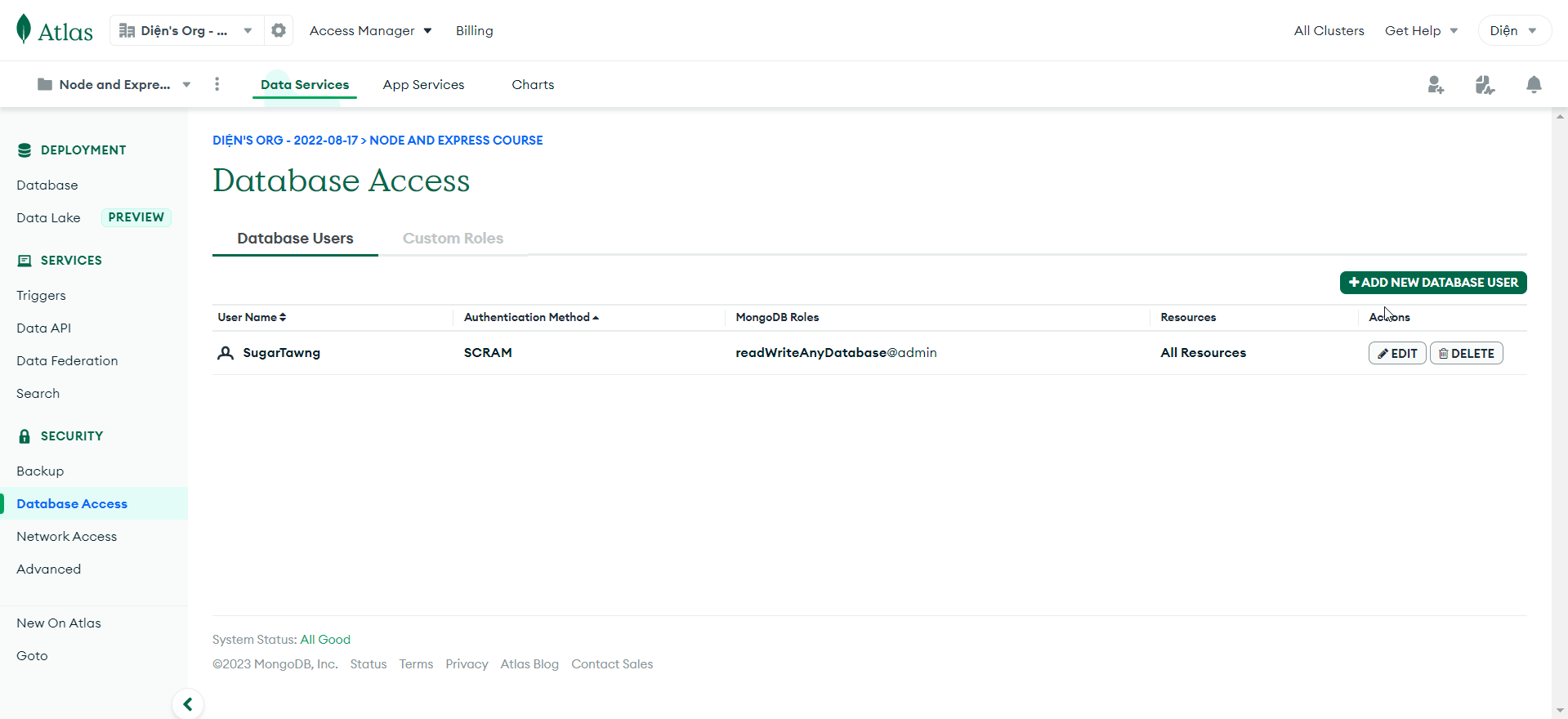


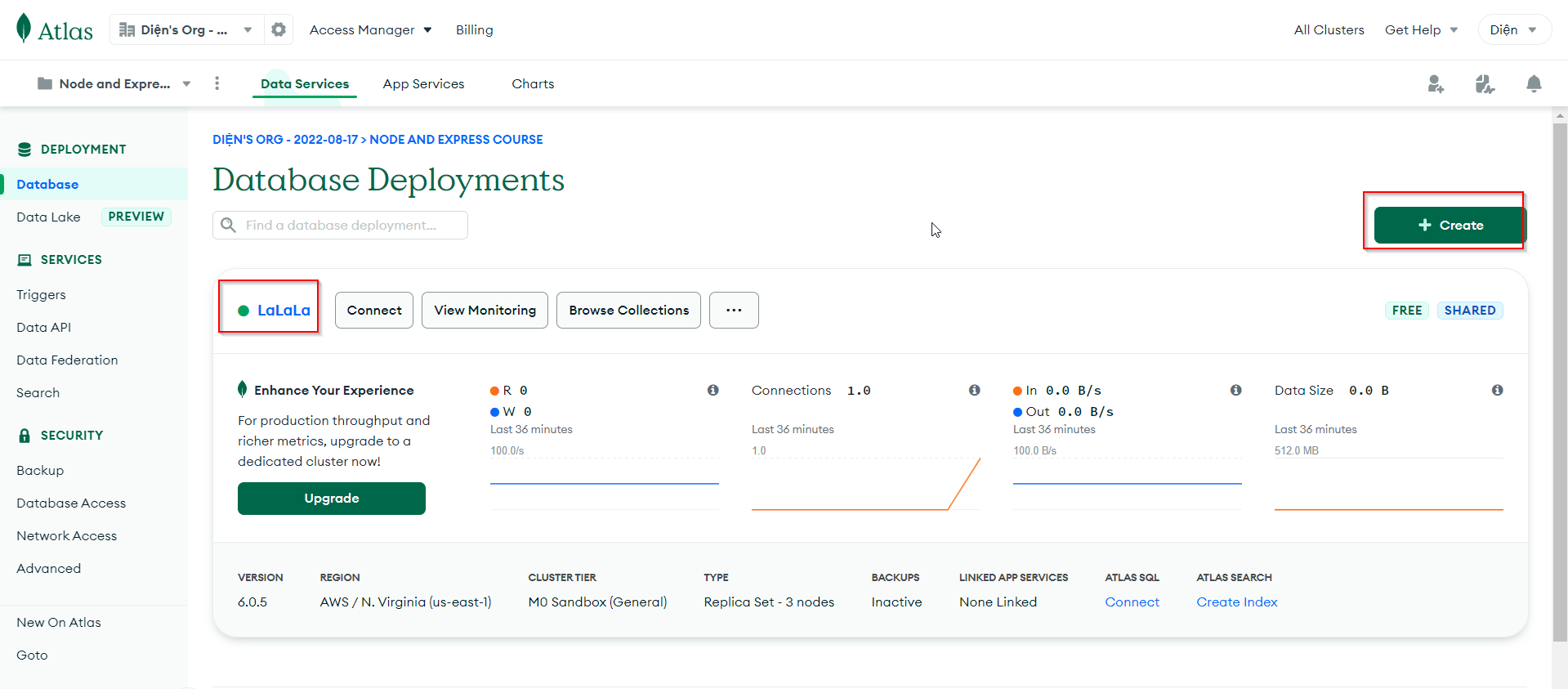
1. NoSql database

Có thể tạo ra nhiều project

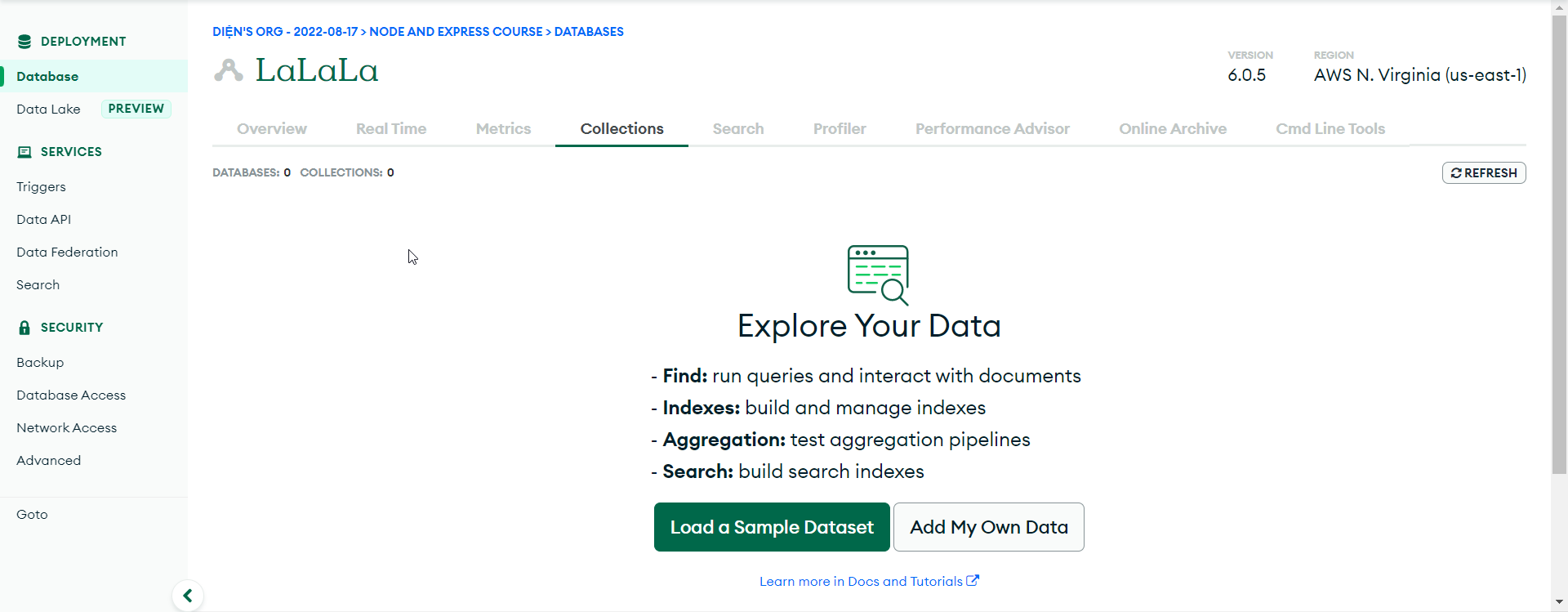


Trong một project có thể add nhiều user và có nhiều cụm (cluster)

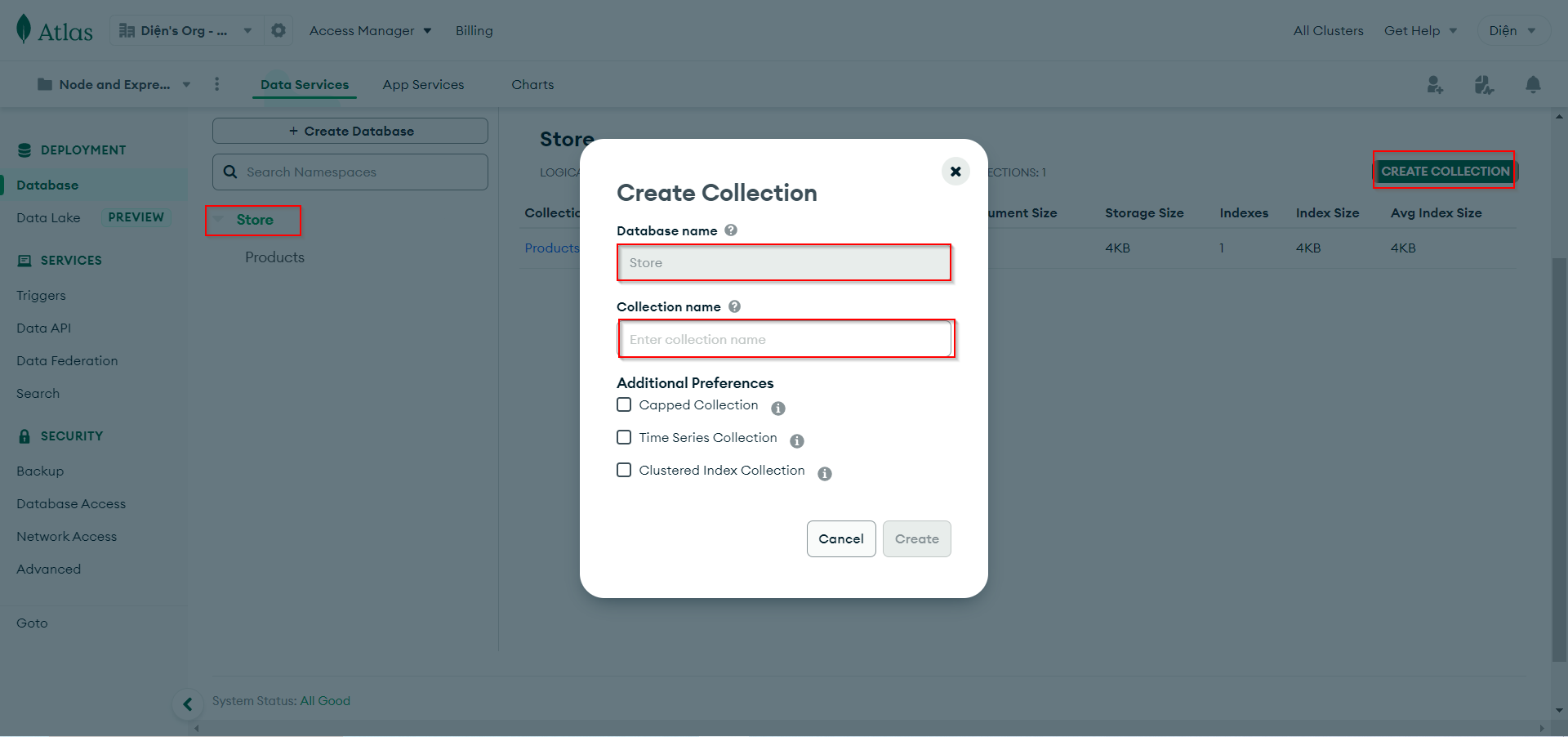




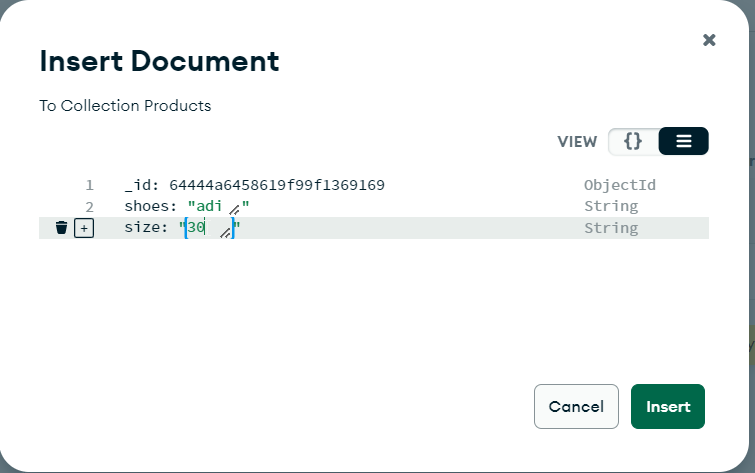
Trong mỗi cụm có thể tạo ra nhiều database trong đó

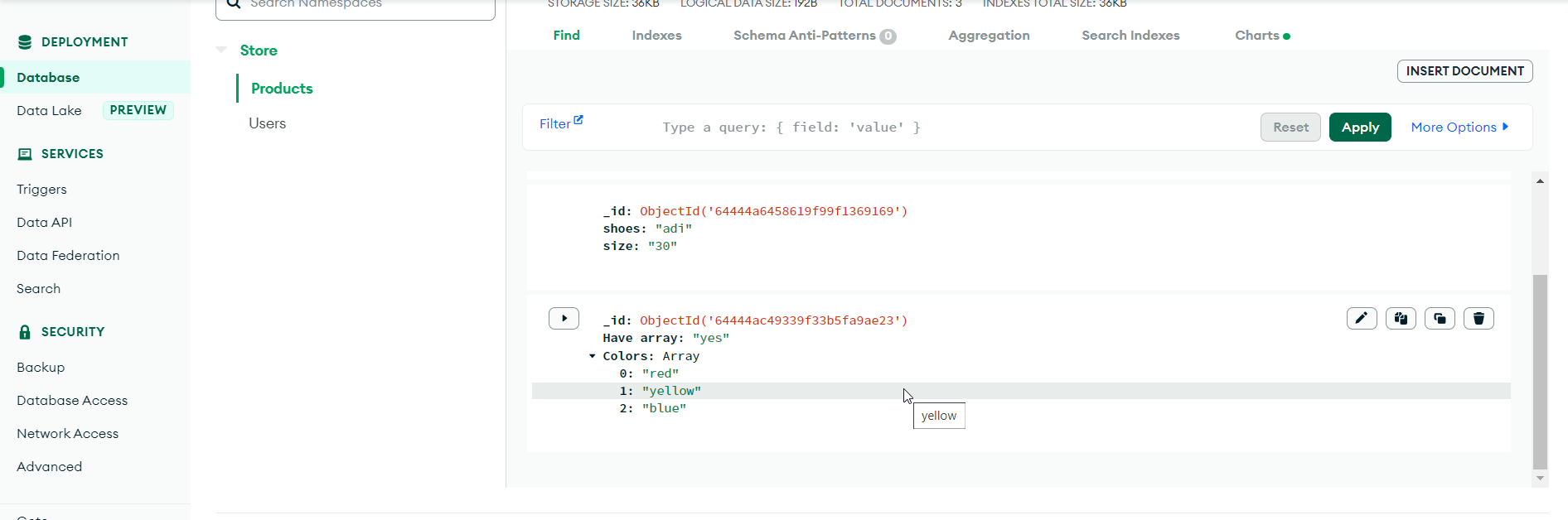


Trong mỗi database ta có thể tạo ra được các collections cho nó.



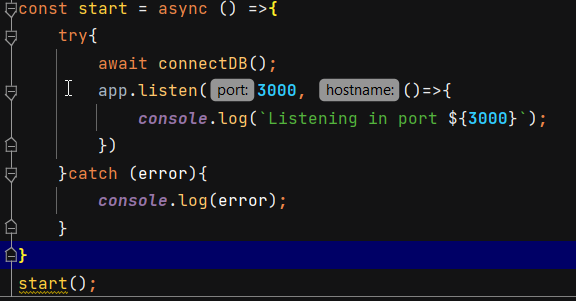
Và trong mỗi collection ta tạo được các object với key và value. Và mỗi key-value có thể được xác định các kiểu dữ liệu bên cạnh nó.



Vì là NoSQL nên cấu trúc của chúng có thể không giống nhau một chút nào hết. Trong mỗi object của nó, nó có các nút bên hông, nên ta có thể thấy rõ là nó có hỗ trợ các phương thức crud trong mongodb. 

Connect to DB:

Để cho code thống nhất thì mình thực hiện đồng bộ hóa luôn phần kết nối DB với listen server. Nếu connect đến DB thành công thì thực hiện listen server, nếu không thành công thì không cần phải listen server mà quăng ra lỗi luôn.



Nhúng thư viện vào một file khác thì ta sẽ có hai cách: (cần phải coi lại)

Gán nó cho một biến, từ biến đó nó có thể gọi cho các thành phần module mà nó export

Nếu không gán vào một biến thì nó chỉ được đưa vào file đó. Không được gọi các thành phần từ module export

Xuất schema dưới dạng một mô hình mongoDB được đặt tên là `User’. Mình đặt tên ở đây là User thì collection trong ứng với nó trong mongoDB cũng có tên là User.



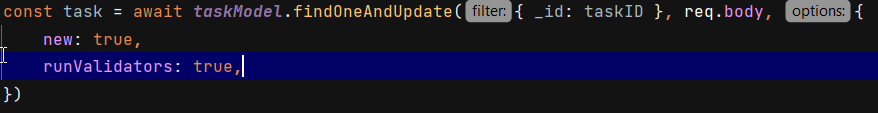
Cú pháp const { id: taskID } = req.params được sử dụng trong Javascript để giải nén (destructuring) thuộc tính id từ đối tượng req.params và gán giá trị này vào biến taskID.



Filter một object sử dụng thông tin của một filed {key, value}



Khi update, nó sử dụng các hàm update, trong đó có setting các option cần thiết cho nó. Nó khác với các phương thức khác bởi vì muốn nó tuân thủ theo các validator thì phải active option cần thiết của nó.

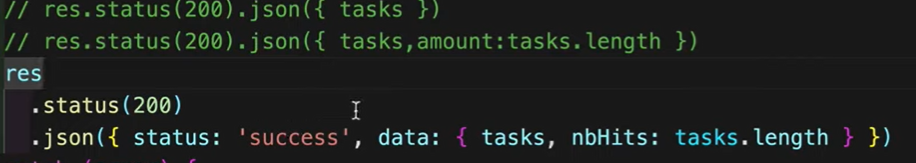


Theo quy chuẩn mà thằng ytb nói. Hoặc cũng tùy theo lựa chọn của dev.

Khi mà bạn sử dụng put thì bạn sẽ thay một nguồn mới cho cái có trước nó (Ghi đè).

Khi mà bạn sử dụng patch thì bạn chỉ thay đổi một phần từ nguồn cũ đó.

Theo quy chuẩn của các lập trình viên trong công ty thì mình có các định dạng để trả về khi thực hiện xong một request để front-end có thể handle nó. Thường thì nó sẽ tuân theo một định dạng cụ thể. Các ví dụ:



Trong delete, có thể chúng ta setting phương thức trả về là trả về object mà nó đã bị xóa, cũng có thể setting trạng thái của request hoặc các phương án setting khác. Bởi vì không có xử lý nào liên quan đến object bị xóa nữa.

Wrapper function là gì?

Wrapper function là một hàm chứa 1 hàm khác trong người nó. Nó có rất nhiều trường hợp phải dùng, một trong những trường hợp đó là thêm một số xử lý vào một hàm cũ mà không cần phải thay đổi hàm cũ đó. Một trường hợp khác là nó đóng vai trò như một lớp abstraction. Ví dụ như mình có 10 hàm để gọi, thay vì mình gọi lần lượt 10 hàm ra thì mình sẽ tạo một Wrapper function giúp mình gọi 10 hàm đó. Giờ mình sẽ tập trung vào cái việc mà tạo ra nhiều cái wrapper function để thêm các xử lý mới mà không làm thay đổi hàm cũ.

Đầu tiên ta tạo ra hàm Add(x,y) và gọi nó. Sau khi tạo ra hàm đó rồi thì tôi lại muốn thêm nó một số tính năng mới. Ví dụ ở đây ta muốn prinf out một số cái nữa. Ta sẽ tạo ra một wrapper function nó nhận tham số đầu vào là một function khác. Trong wrapper function đó, nó trả ra một function khác hoàn toàn (tên bất kỳ), trong function bất kỳ này nó nhận các thông số đầu vào là một hàm khác. tương ứng với hàm mà được nó bao (nếu muốn thêm thêm đối số cũng không sao nhưng KHÔNG được sai thứ tự). Trong hàm bất kỳ này, ta có thể thêm bất kỳ câu lệnh thực thi gì mà ta muốn và sau đó gọi lại hàm cũ để thực thi những tính năng đã xây dựng trong đó. Đó là tất cả những gì mà hàm wrapper xử lý. Bây giờ ta sẽ gán hàm cũ để gọi hàm wrapper và truyền vào hàm wrapper tham số tương ứng của hàm cũ

Sau cuối cùng thì ta log ra hàm Add(), mà hàm Add() này được gán bằng hàm Logger(), hàm Logger này trả về một hàm mới, gàm mới này có chứa hàm cũ và sẽ trả ra các sử lý hiện có ở hàm cũ đó.

* Vậy cuối cùng, nó đã trả ra hàm mới, có xử lý chứa hàm cũ, vậy ta đã thêm các xử lý cần thiết mà không cần sửa lại hàm sẵn có =))).

Và cuối cùng, bạn có thể unwrap nó bằng cách bỏ gán đi.

Return 1 function từ một function khác

Một hàm có thể được thực thi rồi trả về gì đó. Thông thường thì là một giá trị (true, false, String, Int, v.v). Mình có thể tạo ra một hàm, cuối cùng nó trả ra một một giá trị cho đối tượng gọi.

Ví dụ: Mình có thể khai báo một hàm để trả ra chuỗi “One”. Sử dụng hàm trả về đó để gán cho một biến. Nếu mà mình gán mà không gọi thì giá trị nó sẽ là một function, nếu mà mình gán mà gọi thì giá trị của nó sẽ là giá trị trả ra của hàm. Bạn cũng có thể là gọi hàm trực tiếp ngay bên trong console.log() để hiện ra những gì nó trả về.

Trong js thì người ta xem function như là một kiểu dữ liệu. Khi gán một biến cho một hàm thì giá trị của biến đó là hàm đã được gán. Khi mà lưu trữ một hàm trong một biến, ta có thể sử dụng biến đó để gọi thực thi hàm bằng cách thêm operator () ở phía trước biến.

Phía trước tôi có nói là function ở js nó được xem xét là một data type, vì vậy một function có thể return về 1 datatype => một function có thể return về function. Mình có thể viết dưới cú pháp là một function trả về một function. Nó là data type nên có thể hiểu đây là datatype nó muốn return về.

Với cú pháp:

Function abc(){

Return function(){

}

}

Thì khi gọi function ở ngoài với cú pháp abc() nó sẽ trả về hàm bên trong của nó, nếu muốn hàm bên trong nó thực thi luôn thì gọi abc()(), hoặc gán hàm gọi abc() cho một biến, bây giờ biến đó là hàm function(), chỉ cần gọi biến đó thì hàm function sẽ được thực thi.

Vì như nói ở đằng trước là khi gọi hàm bao ở ngoài cùng bằng cú pháp abc(),thì nó sẽ trả ra ta hàm function() bên trong nó, khi muốn thực thi hàm bên trong nó, ta phải gọi abc()() để thực thi bên trong hoặc gán hàm thực thi `abc()` cho một biến, và sau đó gọi cho biến đó và toán tử gọi hàm để thực thi hàm. Với cú pháp:

Function abc(){

Return function(){

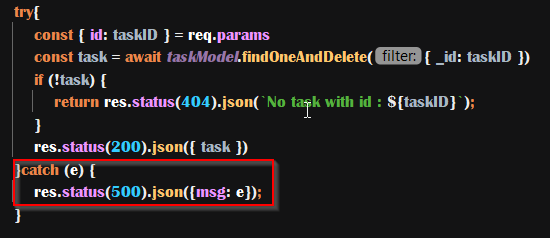
Return datatype;

}

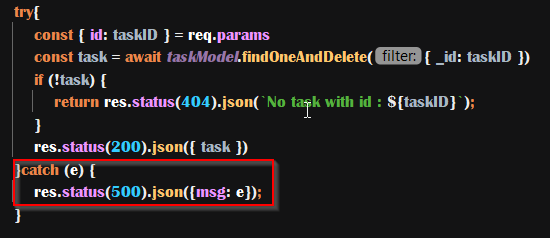
}

Khi gọi hàm abc(), ta sẽ được trả về hàm bên trong của nó, muốn nó trả return datatype của hàm bên trong thì phải thực thi 2 lần hàm ở bên ngoài abc()(). Hoặc cũng có cách khác là sẽ gán hàm thực thi abc() cho một biến rồi thực thi biến đó bằng cách thêm toán tử () để return datatype.

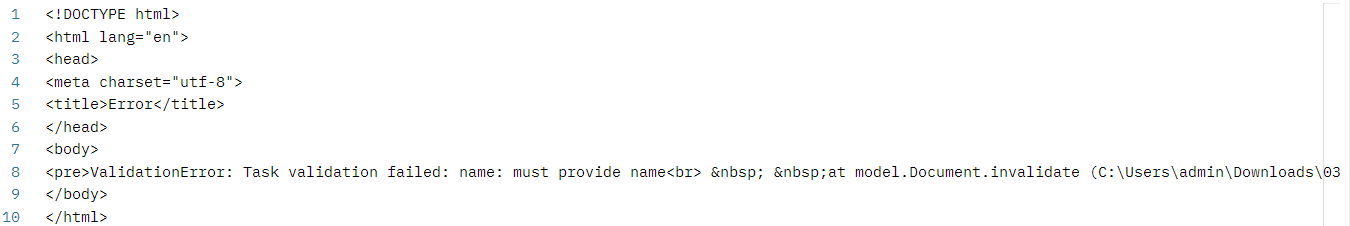
Hàm asyncWrapper sử dụng kiến thức của return function và wrapfunction để thực hiện bao xử lý try catch để bắt lỗi trong các hàm đó. Nếu có lỗi thì sẽ bắt lỗi và gọi hàm next để gọi đến middleware tiếp theo để xử lý lỗi. Nếu middleware đó chưa được tạo ra thì nó sẽ lấy hàm errorHandler mặc định được built sẵn trong nó. Nếu mình sử dụng thư viện default thì không thể custom để trả ra lỗi theo ý của mình theo statusCode và chuỗi json như dưới đây mà nó trả ra một đùm.



Hàm wrapper



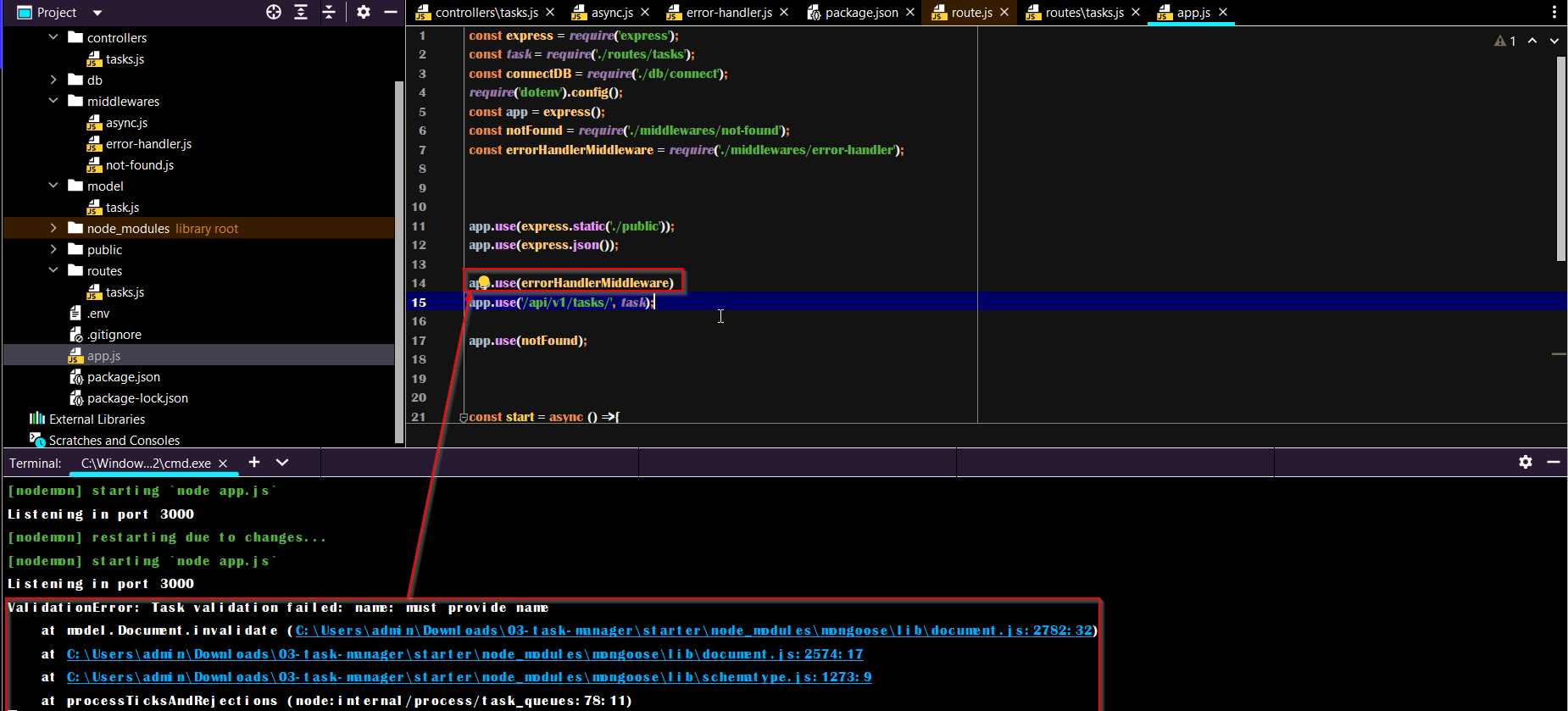
Custom Handle error trả về



Default error handler nếu chúng ta gọi đến phương thức next mặc định.

Chúng ta cũng có thể bỏ hàm customErrorHandler vào hàm wrapperFunction nhưng mà mình muốn tách biệt hàm wrapperFunction và hàm ErrorHandler function thì mình cũng có thể tách bạch nó bằng cách viết một ErrorHandler function để xử lý.

*Chú ý:* Hàm next ở đây nó chỉ nói là pass đến middleware tiếp theo, chứ nó không nhận biết và pass đến middleware phía trước được, bởi vì nó xử lý theo dòng chảy.

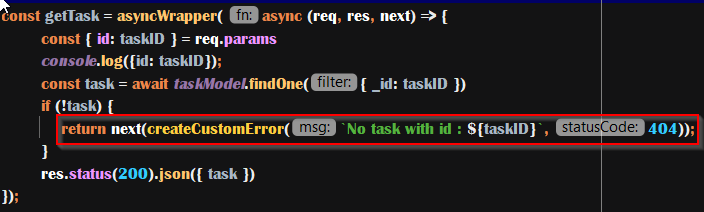


Không thể pass next middleware (để xử lý lỗi, bởi vì errorHandlerMiddleware được đặt trên url đang được xử lý)

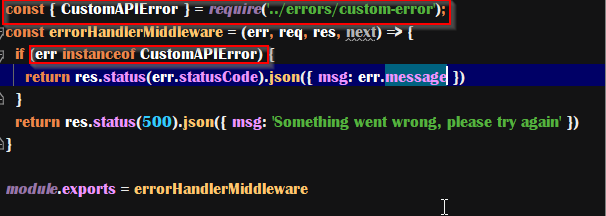
Để xử lý thêm lỗi 404, thì ta có thể setting thêm 1 error custom class extend Error class, khi mà không tìm ra object thì ta có thể khởi tạo 1 instance và sau đó thảy error đó vào middleware error handler để xử lý.



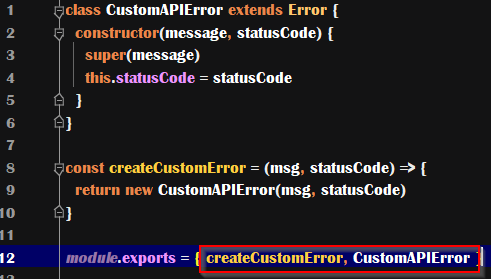
Nhúng instance của customError vào file.



Khi không tìm thấy object, Tạo ra errorCustomclass và thảy vào next middleware là ErrorHandlerclass để xử lý.



Nhúng vào file, class của customError để nhận biết được instance của nó, sau đó thì xử lý các lỗi cho phù hợp.



Export class và instance của nó.

Deployment: Mình sẽ không deploy cái này, bởi vì nó chưa setting security, setting authentication, v.v

Ta có thể thấy là giá trị port được set cứng hardcode. Những giá trị như 3000, 6000, 7000 là được set cứng là một phương án rất phù hợp để chạy ở local. Nhưng khi ta deploy nó ở nhiều môi trường khác nhau. Cái nền tảng mà project đó được host lên muốn được setting một cách độc lập với để nó có ý nghĩa với nền tảng đó bởi vì trên nền tảng đó nó có rất nhiều application được host lên đó. Để là điều đó có khả năng thực hiện thì phải khai báo một biến port để quản lý port của nó thay vì hard code. Và ta phải setting nó để có thể sử dụng và quản lý thông qua process của nó bằng process.env



Khi setting như thế này thì biến port có thể được operator xử dụng cho quá trình deploy, nếu biến port không được setting bởi operator thì nó sẽ sử dụng giá trị mặc định của nó là 3000.

Ví dụ, khi được deploy lên một nền tảng nào đó, nền tảng đó có thể setting port cho nó bằng cách:

Ví dụ

