

# Backend - DB

Backend + Base de Datos

## 1. Clonar el repositorio

```
git clone https://github.com/borjas-prodolliet/clase-19-backend-bd.git
```

```
npm install
```

## 2. Obtener string de conexión

Desde este [link](#)

## DATA STORAGE

## Clusters

Triggers

Data Lake

## SECURITY

Database Access

Network Access

Advanced

BORJASS ORG - 2020-10-24 &gt; NODE WITH MONGO

## Clusters

## SANDBOX

## Cluster0

Version 4.2.10

CONNECT

METRICS

COLLECTIONS

...

## CLUSTER TIER

M0 Sandbox (General)

## REGION

AWS / N. Virginia (us-east-1)

## TYPE

Replica Set - 3 nodes

## LINKED REALM APP

None Linked

## ATLAS SEARCH

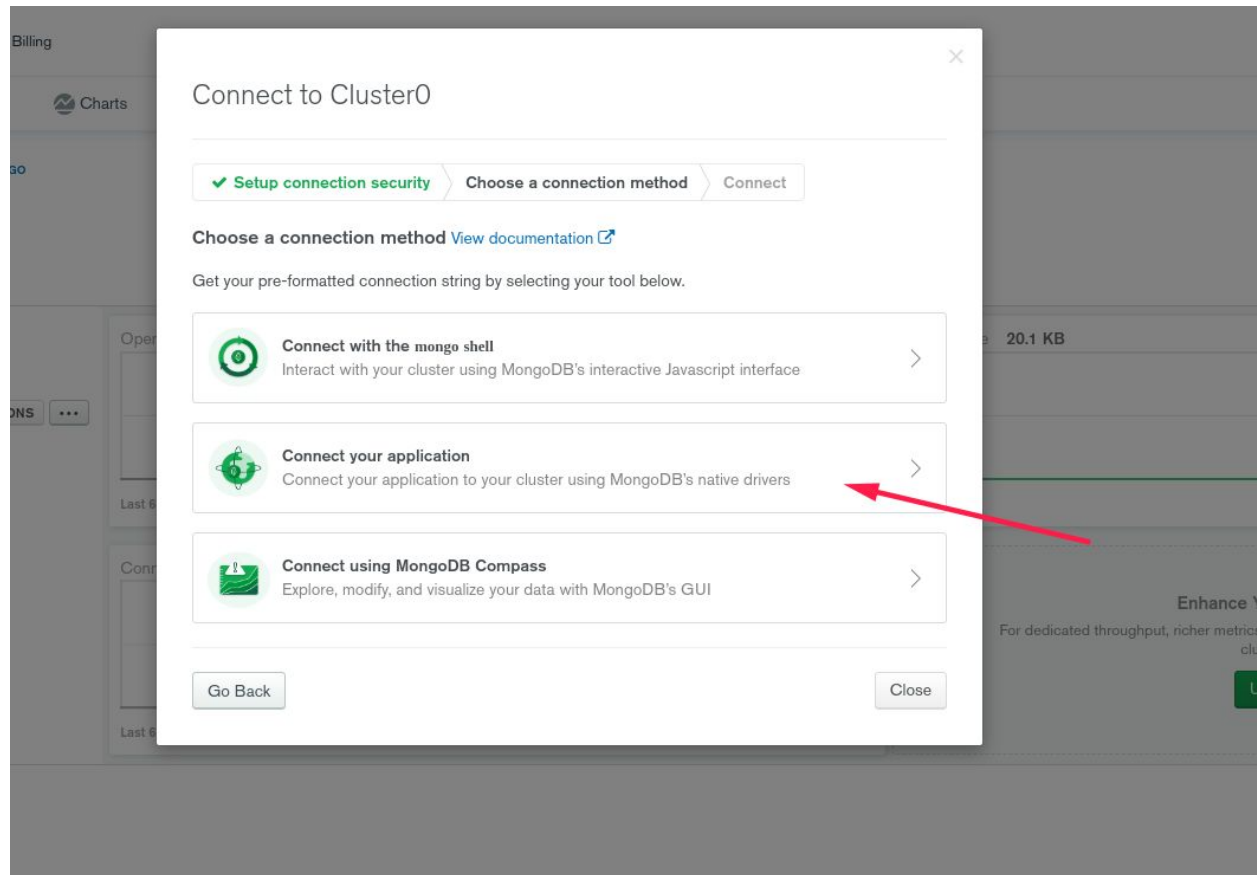
[Create Index](#)

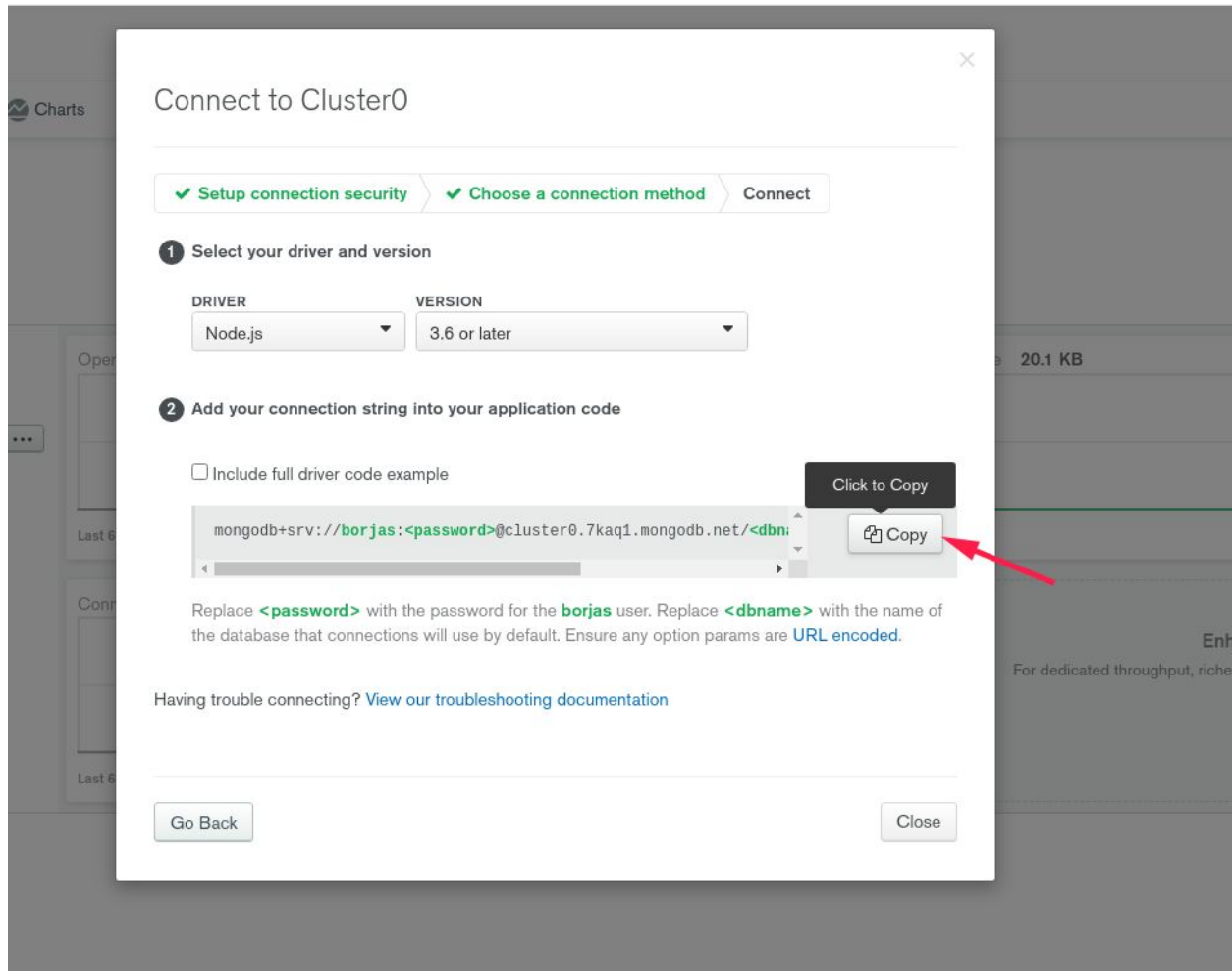
Operations R: 0 W: 0

Last 6 Hours

Connections 0

Last 6 Hours





### 3. Configurar variables de ambiente

Configurar las siguientes variables:

**DATABASE:** Será la url obtenida desde la BD de atlas, por buenas prácticas reemplazamos las partes de la url `<password>` y `<dbname>` por sus equivalentes en mayúsculas.

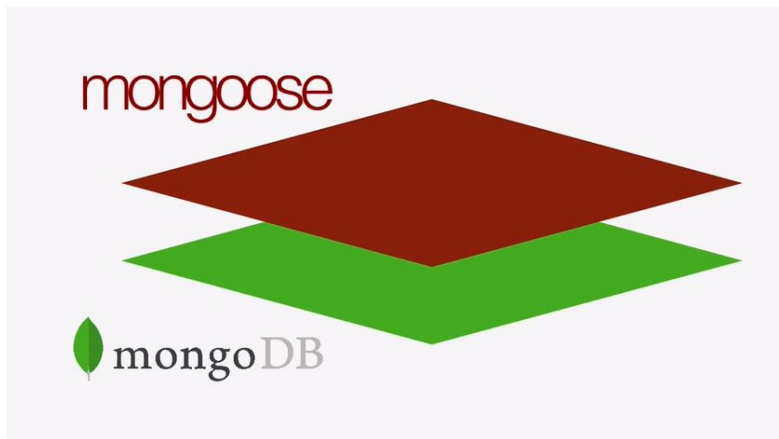
**PASSWORD:** La clave configurada para el usuario de BD.

**DBNAME:** El nombre de la BD, para nuestro caso natours.

## config.env

```
NODE_ENV=development
PORT=3000
DATABASE=mongodb+srv://alejandro:<PASSWORD>@cluster0.7kaq1.mongodb.net/<DBNAME>
?retryWrites=true&w=majority
USERNAME=alejandro
PASSWORD=123456789
DBNAME=natours
```

### 3. Instalar Mongoose



Mongoose es una librería que me ayuda a conectar mi aplicación node con mi base de datos mongo y realizar múltiples operaciones de manera sencilla.

También mongoose permite hacer modelado de datos de objetos para mongo y node, brindando un nivel mayor de abstracción entre la base de datos y la aplicación.



```
npm i mongoose
```

**Mongoose schema:** Donde modelamos nuestros datos, describiendo la estructura de los

datos, valores por defecto, validaciones, etc...

**Mongoose model:** Una interfaz para el esquema, interfaz para hacer operaciones CRUD a la base de datos.

## 4. Configurar server.js para consumir BD

### Server.js

```
const mongoose = require('mongoose'); // Importar la libreria mongoose
const dotenv = require('dotenv');
dotenv.config({path: './config.env'});
const app = require('./app');

const DB = process.env.DATABASE.replace( // Armamos el string de conexión.
  '<PASSWORD>', // Reemplazamos <PASSWORD> y <DBNAME>
  process.env.PASSWORD).replace( // por sus correspondientes variables
  '<DBNAME>', // de ambiente
  process.env.DBNAME);

mongoose.connect(DB, { // Conectamos con la BD
  useNewUrlParser: true,
  useCreateIndex: true,
  useFindAndModify: false
}).then(con => {
  console.log(con.connections);
  console.log('Connection successfully!');
})

const port = process.env.PORT || 3000;
app.listen(port, () => {
  console.log(`App running on port ${port}...`);
});
```

Luego probar si la aplicación se está pudiendo conectar con la base.

```
npm run start
```

```
db:
  Db {
    _events: [Object],
    _eventsCount: 3,
    _maxListeners: undefined,
    s: [Object],
    serverConfig: [Getter],
    bufferMaxEntries: [Getter],
    dbName: [Getter] } } ]
Connection successfully!
```

Conexion exitosa!

## 5. Crear un Esquema y Modelo para Tour

Creamos un esquema y modelamos Tour.

### Server.js

```
const mongoose = require('mongoose');
const dotenv = require('dotenv');
dotenv.config({path: './config.env'});
const app = require('./app');

const DB = process.env.DATABASE.replace(
  '<PASSWORD>',
  process.env.PASSWORD).replace(
  '<DBNAME>',
  process.env.DBNAME);

mongoose.connect(DB, {
  useNewUrlParser: true,
  useCreateIndex: true,
  useFindAndModify: false
}).then(con => {
  console.log(con.connections);
  console.log('Connection successfully!');
})

const tourSchema = new mongoose.Schema({ // Creación del esquema con sus características
```

```

    name: {
      type:String,
      required: [true, 'A tour must have a name'],
      unique: true
    },
    rating: {
      type: Number,
      default:4.5
    },
    price: {
      type: Number,
      required: [true, 'A tour must have a price']
    }
  });
const Tour = mongoose.model('Tour', tourSchema); // Creación del modelo
                                                    // basado en el esquema

const port = process.env.PORT || 3000;
app.listen(port, () => {
  console.log(`App running on port ${port}...`);
});

```

## 6. Guardar nuestro primer documento desde la app

### Server.js

```

const mongoose = require('mongoose');
const dotenv = require('dotenv');
dotenv.config({path: './config.env'});
const app = require('./app');

const DB = process.env.DATABASE.replace(
  '<PASSWORD>',
  process.env.PASSWORD).replace(
  '<DBNAME>',
  process.env.DBNAME);

mongoose.connect(DB, {
  useNewUrlParser: true,

```



```

    useCreateIndex: true,
    useFindAndModify: false
  }).then(con => {
    console.log(con.connections);
    console.log('Connection successfully!');
  })

const tourSchema = new mongoose.Schema({
  name: {
    type:String,
    required: [true, 'A tour must have a name'],
    unique: true
  },
  rating: {
    type: Number,
    default:4.5
  },
  price: {
    type: Number,
    required: [true, 'A tour must have a price']
  }
});

const Tour = mongoose.model('Tour', tourSchema);

const testTour = new Tour ({ // Creamos el nuevo tour que queremos insertar
  name: 'The forest Hiker', // basándonos en el modelo creado
  rating: 4.7,
  price: 497
});

testTour // Guardamos el tour creado, esto devuelve una promesa
  .save() // si esta todo ok imprimimos el documento, si no
  .then(doc => { // vamos a imprimir el error
    console.log(doc);
  })
  .catch(err => {
    console.log('ERROR: ', err)
  });

const port = process.env.PORT || 3000;
app.listen(port, () => {
  console.log(`App running on port ${port}...`);
});

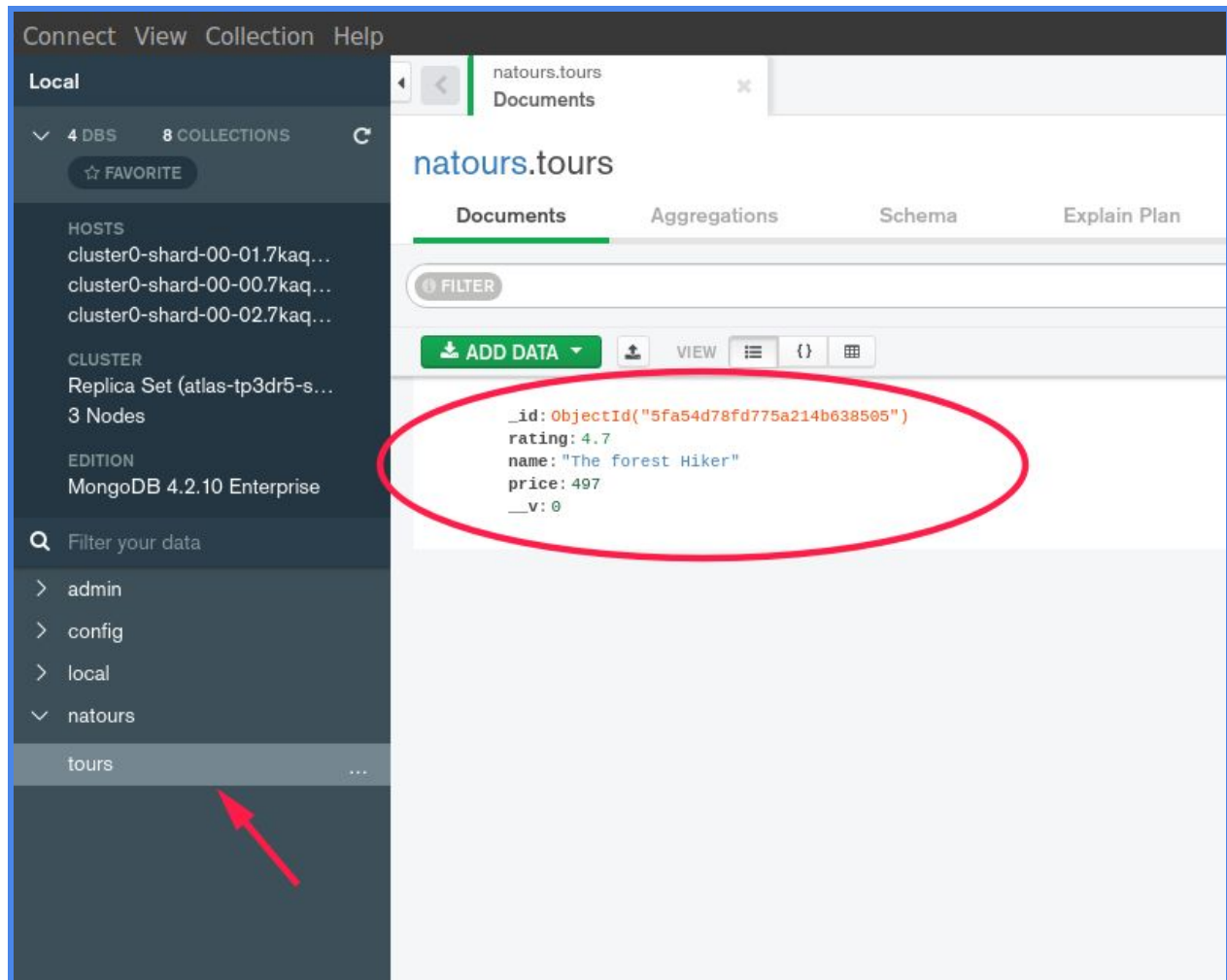
```

## 6.1 Verificamos que el nuevo documento esté en la BD

Verificamos que en la consola se esté imprimiendo el documento que creamos.

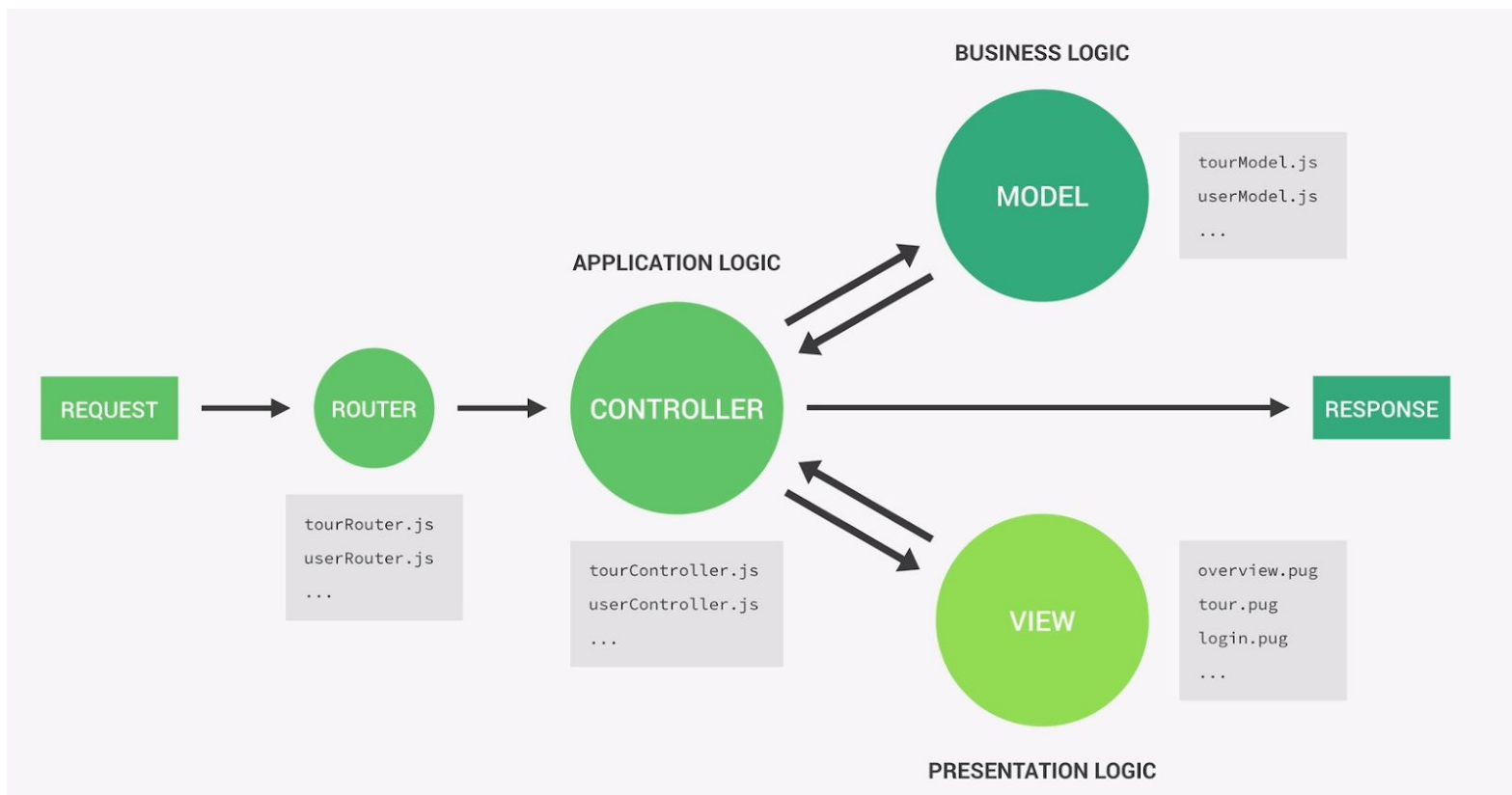
```
    _maxListeners: undefined,  
    s: [Object],  
    serverConfig: [Getter],  
    bufferMaxEntries: [Getter],  
    dbName: [Getter] } } ]  
Connection successfully!  
{ rating: 4.7,  
  _id: 5fa54d78fd775a214b638505,  
  name: 'The forest Hijer',  
  price: 497,  
  __v: 0 }  
[]
```

Luego vamos a MongoDB Compass y verificamos que exista el documento.



## 7. Introducción a la arquitectura de backend MVC

Separando nuestra arquitectura en capas nos permitirá tener una aplicación más modular, la cual será más fácil de mantener y escalar.



El circuito comienza con el **request**, luego el **request** será manejado por alguno de los **routers**.

El objetivo del **router** es delegar el **request** a la función manejadora correcta, la cual estará en alguno de nuestros **controllers**.

Luego dependiendo del **request** el **controller** podría necesitar interactuar con alguno de los **modelos**, por ejemplo para traer un documento de la base de datos o crear uno.

Luego con la respuesta del **modelo**, el **controller** debería estar listo para dar una **respuesta**.

En el caso de que quisiéramos renderizar un sitio web, después de obtener una respuesta del **modelo**, el **controller** va a seleccionar uno de los templates de **vista** y luego va a inyectar los datos en la **vista**. Después de esto va a renderizar la vista y devolverla como **respuesta**.

## Lógica de Aplicación y Lógica de Negocio

Uno de los principales objetivos de la arquitectura MVC es separar la **lógica de negocio** de la **lógica de la aplicación**.



## Lógica de Aplicación

La lógica de aplicación es el **código que solo concierne a la implementación de la aplicación** y no relacionado con el problema de negocio.

La lógica de la aplicación es:

- manejo de request y responses
- sobre los aspectos más técnicos de la app
- es un puente entre el modelo y las vistas

## Logica de Negocio

Es el **código que resuelve los problemas de negocio** como por ejemplo: mostrar o vender tours. Está directamente relacionado con las reglas de negocio, con cómo el negocio funciona y que necesita el negocio.

Ejemplos:

- Crear nuevo tour en la BD
- Verificar si un password para un usuario es correcto
- Validar entrada de datos del usuario
- Permitir solo a los usuarios que compraron tours hacer un review.

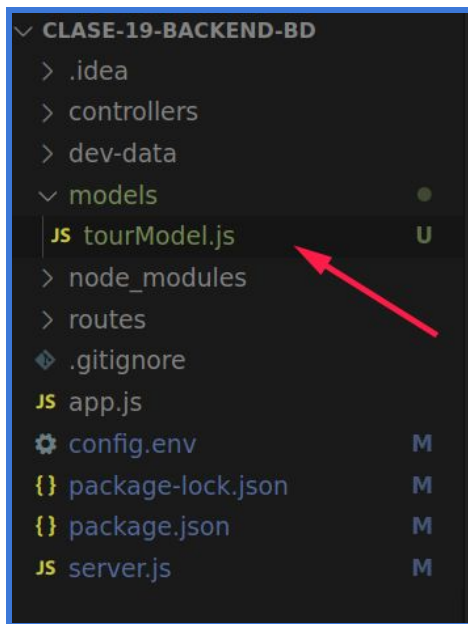
## Conceptos

No siempre es posible separar la lógica de negocio de la lógica de aplicación, pero deberíamos esforzarnos por hacerlo lo más que se pueda.

**Modelos cargados y controladores livianos:** Esta es la filosofía que deberíamos seguir, cargar toda la lógica posible en los modelos y mantener los controladores lo más simple y livianos posibles.

## 8. Refactorizando nuestra app para MVC

Crear carpeta models y dentro de la carpeta crear el archivo **tourModel.js**



Extraemos la parte tourSchema y el model desde el **server.js** y lo ponemos en **tourModel.js**.

### tourModel.js

```
const mongoose = require('mongoose');

const tourSchema = new mongoose.Schema({
  name: {
    type: String,
    required: [true, 'A tour must have a name'],
    unique: true
  },
  rating: {
    type: Number,
```

```

        default:4.5
      },
      price: {
        type: Number,
        required: [true, 'A tour must have a price']
      }
    });

const Tour = mongoose.model('Tour', tourSchema);

module.exports = Tour; // Exportamos el modelo

```

Dejamos en el **server.js** solo la parte de la conexión con la BD.

## server.js

```

const mongoose = require('mongoose');
const dotenv = require('dotenv');
dotenv.config({path: './config.env'});
const app = require('./app');

const DB = process.env.DATABASE.replace(
  '<PASSWORD>',
  process.env.PASSWORD).replace(
  '<DBNAME>',
  process.env.DBNAME);

mongoose.connect(DB, {
  useNewUrlParser: true,
  useCreateIndex: true,
  useFindAndModify: false
}).then(con => {
  console.log(con.connections);
  console.log('Connection successfully!');
});

const port = process.env.PORT || 3000;
app.listen(port, () => {
  console.log(`App running on port ${port}...`);
});

```

Luego procedemos a limpiar nuestros archivos **tourController.js** y **tourRoutes.js** para dejar de utilizar el archivo que estábamos usando para leer los tours y comenzar a utilizar nuestra base de datos mongo.

## tourController.js

```
const Tour = require('../models/tourModel');

exports.createTour = (req, res) => {

  res.status(201).json({
    status: 'success'
  });

};

exports.getAllTours = (req, res) => {
  console.log(req.requestTime);

  res.status(200).json({
    status: 'success',
    requestedAt: req.requestTime,
  });
};

exports.getTour = (req, res) => {

  res.status(200).json({
    status: 'success',
  });
};

exports.updateTour = (req, res) => {
  res.status(200).json({
    status: 'success',
    data: {
      tour: '<Updated tour here...>'
    }
  });
};

exports.deleteTour = (req, res) => {
  res.status(204).json({
```



```
    status: 'success',  
    data: null  
  });  
};
```

## tourRoutes.js

```
const express = require('express');  
const tourController = require('../controllers/tourController');  
  
const router = express.Router();  
  
router  
  .route('/')  
  .get(tourController.getAllTours)  
  .post(tourController.createTour);  
  
router  
  .route('/:id')  
  .get(tourController.getTour)  
  .patch(tourController.updateTour)  
  .delete(tourController.deleteTour);  
  
module.exports = router;
```

## 9. Implementando la función: createTour

Vamos a implementar la función createTour en nuestro **tourController.js**

## tourController.js

```
const Tour = require('../models/tourModel');  
  
exports.createTour = async (req, res) => {  
  
  try {
```

```

    const newTour = await Tour.create(req.body); // Creamos nuestro tour

    res.status(201).json({ // Respondemos el tour creado
      status: 'success',
      data: {
        tour: newTour
      }
    });
  } catch (err) { // Respondemos el error
    res.status(400).json({
      status: 'fail',
      message: err
    })
  }
};

exports.getAllTours = (req, res) => {

  console.log(req.requestTime);

  res.status(200).json({
    status: 'success',
    requestedAt: req.requestTime,
  });
};

exports.getTour = (req, res) => {

  res.status(200).json({
    status: 'success',
  });
};

exports.updateTour = (req, res) => {
  res.status(200).json({
    status: 'success',
    data: {
      tour: '<Updated tour here...>'
    }
  });
};

exports.deleteTour = (req, res) => {

```

```
res.status(204).json({
  status: 'success',
  data: null
});
};
```

## 9.1 Probando la función creatTour

Operacion: **POST**

URL: <http://localhost:3000/api/v1/tours/>

Body:

```
{
  "name": "Test tour 1",
  "price": 100,
  "duration": 10,
  "difficulty": "easy",
  "rating": 4.7
}
```

POST Create New Tour

+

...

Create New Tour

POST

http://localhost:3000/api/v1/tours/

ParamsAuthorizationHeaders (8)BodyPre-request ScriptTestsSettings

none

form-data

x-www-form-urlencoded

raw

binary

GraphQL

JSON

```
1 {
2   "name": "Test tour 1",
3   "price": 100,
4   "duration": 10,
5   "difficulty": "easy",
6   "rating": 4.7
7 }
```

BodyCookiesHeaders (6)Test Results

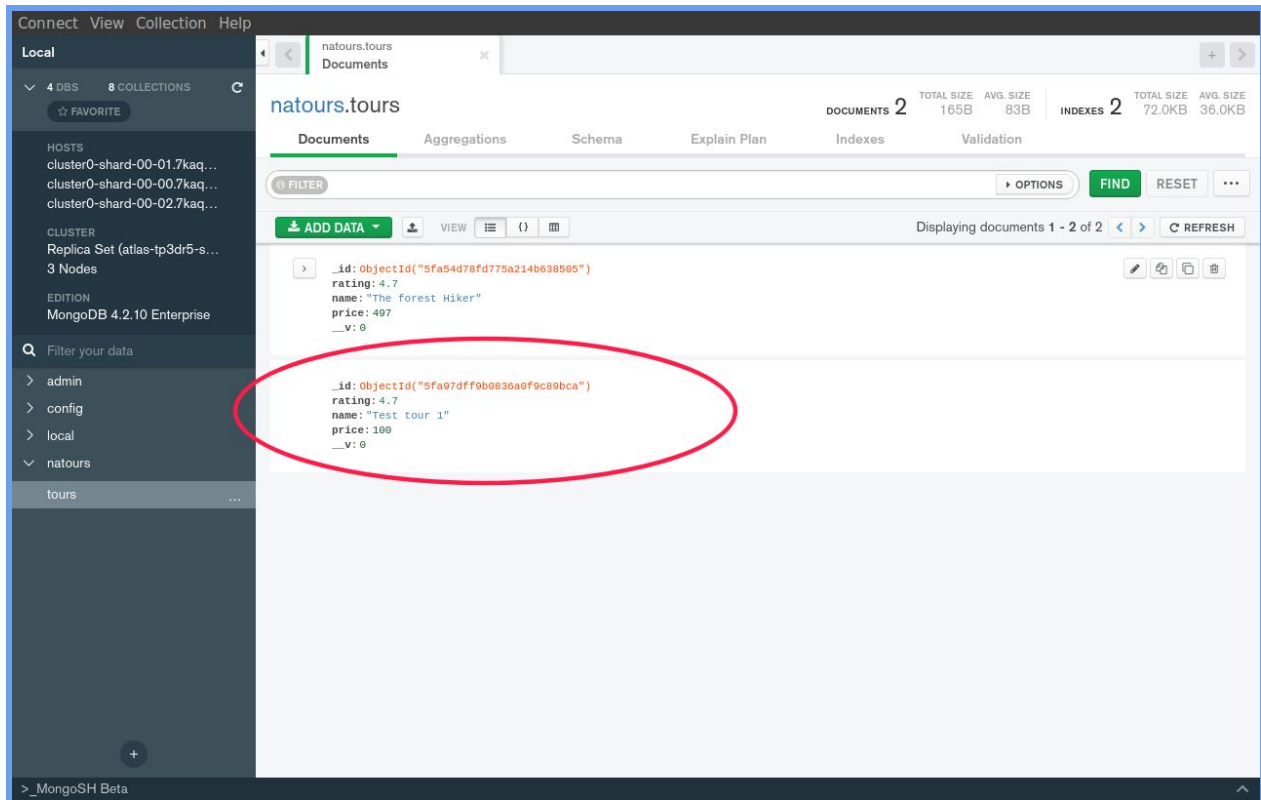
PrettyRawPreviewVisualize

JSON

≡

```
1 {
2   "status": "success",
3   "data": {
4     "tour": {
5       "rating": 4.7,
6       "_id": "5fa97dff9b0836a0f9c89bca",
7       "name": "Test tour 1",
8       "price": 100,
9       "__v": 0
10    }
11  }
12 }
```

## 9.2 Verificar los datos en MongoDB Compass



## 10. Implementando la función: getAllTours

Implementamos la función `getAllTours` en nuestro `tourController.js`.

### `tourController.js`

```
const Tour = require('../models/tourModel');

exports.createTour = async (req, res) => {

  try {
    const newTour = await Tour.create(req.body);
```

```

    res.status(201).json({
      status: 'success',
      data: {
        tour: newTour
      }
    });
  } catch (err) {
    res.status(400).json({
      status: 'fail',
      message: err
    })
  }
}

};

exports.getAllTours = async (req, res) => {
  try {
    const tours = await Tour.find();    // Buscamos los tours

    res.status(200).json({              // Devolvemos los tours
      status: 'success',
      results: tours.length,            // La cantidad de tours
      data: {
        tours
      }
    });
  } catch (err) {
    res.status(404).json({              // Devolvemos el error
      status: 'fail',
      message: err
    })
  }
}

};

exports.getTour = (req, res) => {

  res.status(200).json({
    status: 'success',
  });
};

exports.updateTour = (req, res) => {
  res.status(200).json({
    status: 'success',
  });
};

```

```
    data: {  
      tour: '<Updated tour here...>'  
    }  
  });  
};  
  
exports.deleteTour = (req, res) => {  
  res.status(204).json({  
    status: 'success',  
    data: null  
  });  
};
```

## 10.1 Probando la función getAllTours

Operacion: **GET**

URL: <http://localhost:3000/api/v1/tours/>

GET Get All Tour

×

+

⋮

▶ Get All Tour

GET

▼

http://localhost:3000/api/v1/tours/

ParamsAuthorizationHeaders (6)BodyPre-request ScriptTestsSettings

Query Params

	KEY	VALUE
	Key	Value

BodyCookiesHeaders (6)Test Results

PrettyRawPreviewVisualizeJSON

```
1  {
2    "status": "success",
3    "results": 2,
4    "data": {
5      "tours": [
6        {
7          "rating": 4.7,
8          "_id": "5fa54d78fd775a214b638505",
9          "name": "The forest Hiker",
10         "price": 497,
11         "__v": 0
12       },
13       {
14         "rating": 4.7,
15         "_id": "5fa97dff9b0836a0f9c89bca",
16         "name": "Test tour 1",
17         "price": 100,
18         "__v": 0
19       }
20     ]
21   }
22 }
```



## 11. Implementando la función: getTour

tourController.js

```
const Tour = require('../models/tourModel');

exports.createTour = async (req, res) => {

  try {
    const newTour = await Tour.create(req.body);

    res.status(201).json({
      status: 'success',
      data: {
        tour: newTour
      }
    });
  } catch (err) {
    res.status(400).json({
      status: 'fail',
      message: err
    })
  }
};

exports.getAllTours = async (req, res) => {
  try {
    const tours = await Tour.find();

    res.status(200).json({
      status: 'success',
      results: tours.length,
      data: {
        tours
      }
    });
  } catch (err) {
    res.status(404).json({
      status: 'fail',

```

```

        message: err
      })
    }
  };

exports.getTour = async (req, res) => {

  try {
    const tour = await Tour.findById(req.params.id); // Obtenemos el tour
    res.status(200).json({                             // que coincida con el id
      status: 'success',
      data: {
        tour
      }
    });
  } catch (err) {
    res.status(404).json({
      status: 'fail',
      message: err
    })
  }
};

exports.updateTour = (req, res) => {
  res.status(200).json({
    status: 'success',
    data: {
      tour: '<Updated tour here...>'
    }
  });
};

exports.deleteTour = (req, res) => {
  res.status(204).json({
    status: 'success',
    data: null
  });
};

```

## 11.1 Probando la función getTour

Operation: **GET**

URL: <http://localhost:3000/api/v1/tours/ID>

The screenshot shows a REST client interface with a tab titled "GET Get Tour". The request method is "GET" and the URL is "http://localhost:3000/api/v1/tours/5fa54d78fd775a214b638505". The "Headers" tab is selected, showing 6 headers. Below the headers, the "Body" tab is selected, showing the response in JSON format. The response is a JSON object with a "status" of "success" and a "data" object containing a "tour" object with details like "rating", "\_id", "name", "price", and "\_v".

**GET Get Tour**

GET <http://localhost:3000/api/v1/tours/5fa54d78fd775a214b638505>

Params Authorization Headers (6) Body Pre-request Script Tests Settings

Query Params

KEY	VALUE
Key	Value

Body Cookies Headers (6) Test Results

Pretty Raw Preview Visualize JSON

```
1 {
2   "status": "success",
3   "data": {
4     "tour": {
5       "rating": 4.7,
6       "_id": "5fa54d78fd775a214b638505",
7       "name": "The forest Hiker",
8       "price": 497,
9       "_v": 0
10    }
11  }
12 }
```

## 12. Implementando la función: updateTour

### tourController.js

```
const Tour = require('../models/tourModel');

exports.createTour = async (req, res) => {

  try {
    const newTour = await Tour.create(req.body);

    res.status(201).json({
      status: 'success',
      data: {
        tour: newTour
      }
    });
  } catch (err) {
    res.status(400).json({
      status: 'fail',
      message: err
    })
  }
};

exports.getAllTours = async (req, res) => {
  try {
    const tours = await Tour.find();

    res.status(200).json({
      status: 'success',
      results: tours.length,
      data: {
        tours
      }
    });
  } catch (err) {
    res.status(404).json({
      status: 'fail',
      message: err
    })
  }
};
```

```

    }
  };

exports.getTour = async (req, res) => {

  try {
    const tour = await Tour.findById(req.params.id);
    res.status(200).json({
      status: 'success',
      data: {
        tour
      }
    });
  } catch (err) {
    res.status(404).json({
      status: 'fail',
      message: err
    })
  }
};

exports.updateTour = async (req, res) => {
  try {
    const tour = await Tour.findByIdAndUpdate(req.params.id, req.body, {
      new: true // Buscamos el tour y lo actualizamos
    }); // con los datos que vienen en el body
    res.status(200).json({
      status: 'success',
      data: {
        tour
      }
    });
  } catch (err) {
    res.status(404).json({
      status: 'fail',
      message: err
    })
  }
};

exports.deleteTour = (req, res) => {
  res.status(204).json({
    status: 'success',
    data: null
  });
};

```

```
};
```

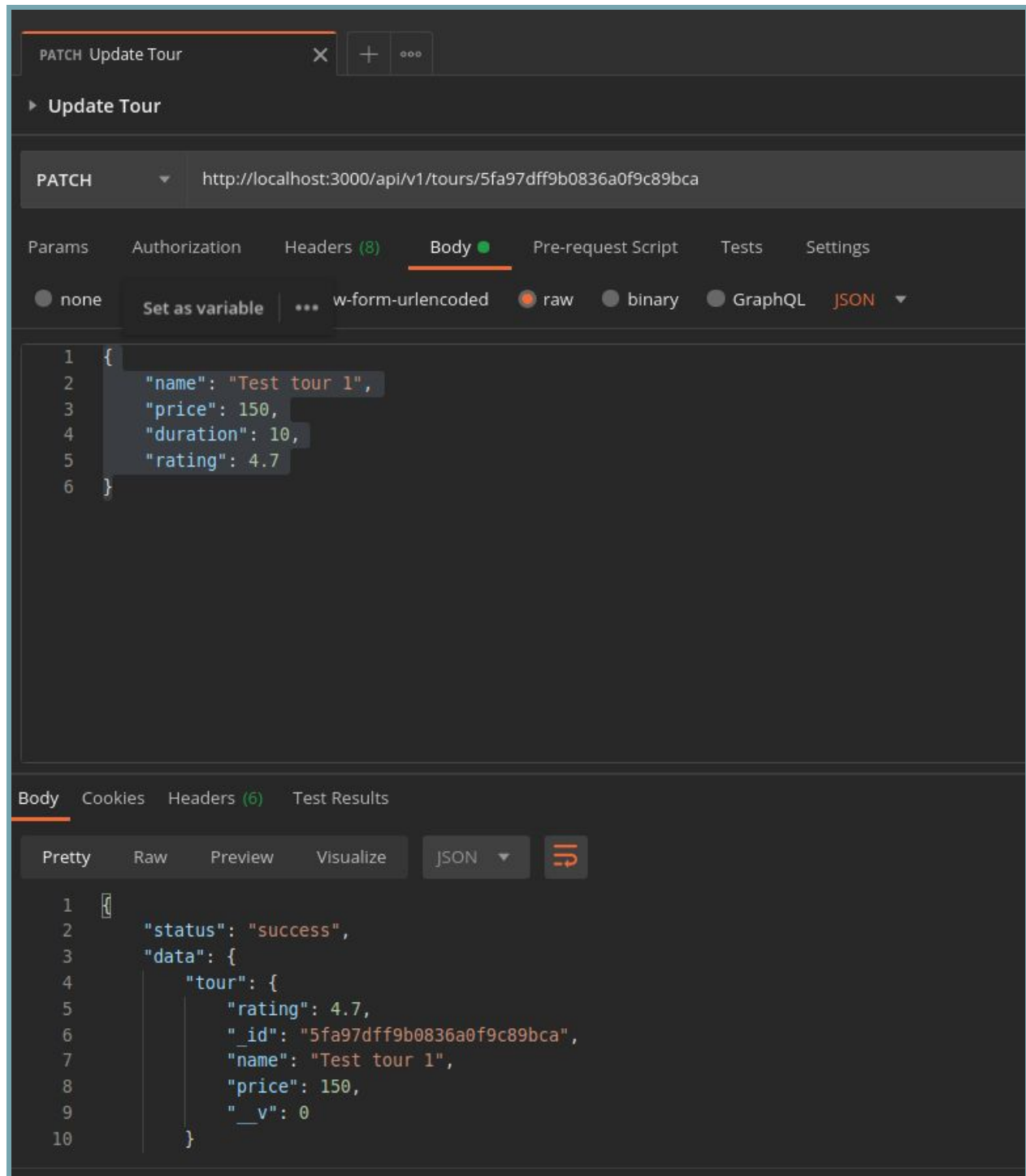
## 12.1 Probando la función updateTour

Operacion: **PATCH**

URL: <http://localhost:3000/api/v1/tours/ID>

Body:

```
{  
  "name": "Test tour 1",  
  "price": 150,  
  "duration": 10,  
  "rating": 4.7  
}
```



### 13. Implementando la función: deleteTour

## tourController.js

```
const Tour = require('../models/tourModel');

exports.createTour = async (req, res) => {

  try {
    const newTour = await Tour.create(req.body);

    res.status(201).json({
      status: 'success',
      data: {
        tour: newTour
      }
    });
  } catch (err) {
    res.status(400).json({
      status: 'fail',
      message: err
    })
  }
};

exports.getAllTours = async (req, res) => {
  try {
    const tours = await Tour.find();

    res.status(200).json({
      status: 'success',
      results: tours.length,
      data: {
        tours
      }
    });
  } catch (err) {
    res.status(404).json({
      status: 'fail',
      message: err
    })
  }
};

exports.getTour = async (req, res) => {
```



```

try {
  const tour = await Tour.findById(req.params.id);
  res.status(200).json({
    status: 'success',
    data: {
      tour
    }
  });
} catch (err) {
  res.status(404).json({
    status: 'fail',
    message: err
  })
}
};

exports.updateTour = async (req, res) => {
  try {
    const tour = await Tour.findByIdAndUpdate(req.params.id, req.body, {
      new: true
    });
    res.status(200).json({
      status: 'success',
      data: {
        tour
      }
    });
  } catch (err) {
    res.status(404).json({
      status: 'fail',
      message: err
    })
  }
};

exports.deleteTour = async (req, res) => {
  try {
    await Tour.findByIdAndDelete(req.params.id); // Borramos el tour coincide
                                                    // con el ID
    res.status(200).json({
      status: 'success',
      data: null
    });
  } catch (err) {
    res.status(404).json({

```

```
    status: 'fail',  
    message: err  
  })  
}  
};
```

## 13.1 Probando la función deleteTour

Operacion: **DELETE**

URL: <http://localhost:3000/api/v1/tours/ID>

The screenshot shows a REST client interface with a tab titled "DEL Delete Tour". The request method is "DELETE" and the URL is "http://localhost:3000/api/v1/tours/5fa97dff9b0836a0f9c89bca". The "Headers" tab is selected, showing 6 headers. The "Body" tab is also visible, showing a JSON response in "Pretty" format: {"status": "success", "data": null}. The response is displayed in a code editor with line numbers 1 through 4.

**DELETE** ▼ http://localhost:3000/api/v1/tours/5fa97dff9b0836a0f9c89bca

Params Authorization Headers (6) Body Pre-request Script Tests Settings


Query Params

KEY	VALUE
Key	Value

Body Cookies Headers (6) Test Results

Pretty Raw Preview Visualize JSON ≡

```
1  {  
2    "status": "success",  
3    "data": null  
4  }
```



MongoDB Com

Connect View Collection Help

Local

4 DBS8 COLLECTIONS

☆ FAVORITE

HOSTS

cluster0-shard-00-01.7kaq...

cluster0-shard-00-00.7kaq...

cluster0-shard-00-02.7kaq...

CLUSTER

Replica Set (atlas-tp3dr5-s...

3 Nodes

EDITION

MongoDB 4.2.10 Enterprise

Filter your data

> admin

> config

> local

> natours

tours...

natours.toursDocuments

natours.tours

DocumentsAggregationsSchemaExplain Plan

FILTER

ADD DATA

VIEW

```
_id: ObjectId("5fa54d78fd775a214b638505")
rating: 4.7
name: "The forest Hiker"
price: 497
_v: 0
```