Node.js

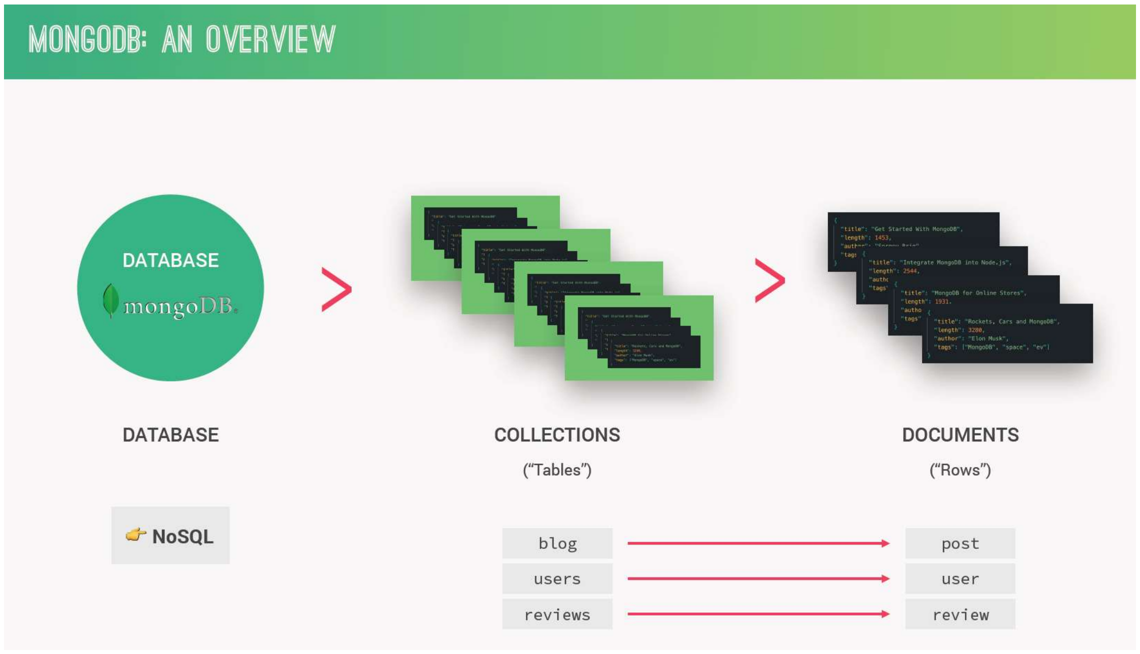
SECTION 07 – Introduction to MongoDB

EC-137:

Sinha, A

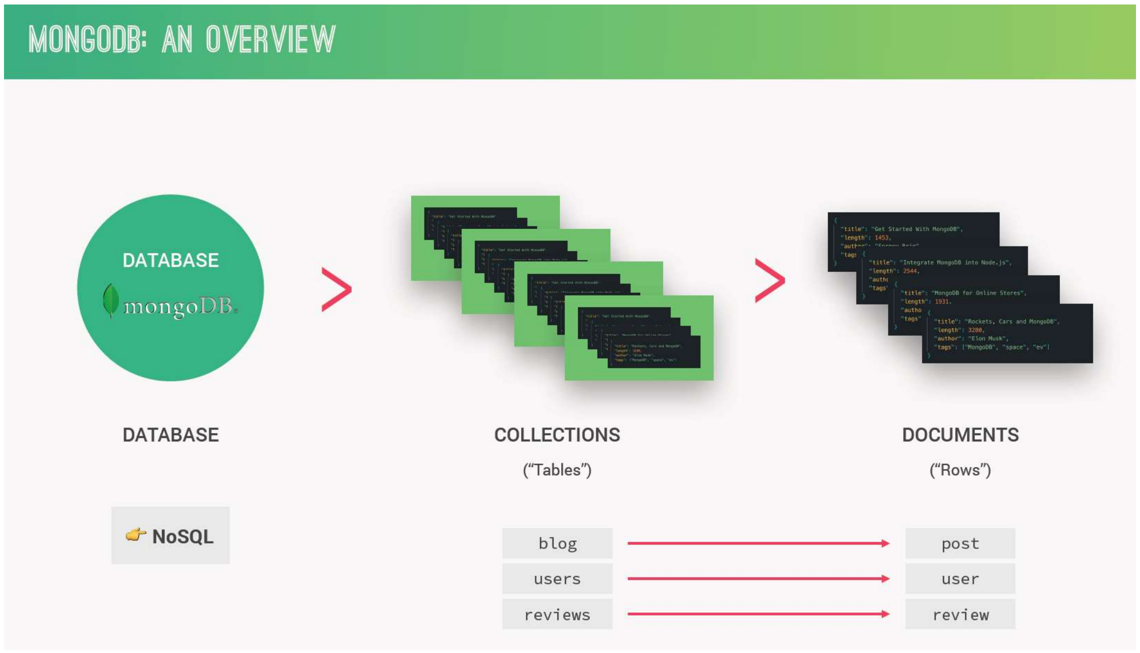
```
//-----------------------------------------------------------
//                        LECTURE 70              ☆ ☆
//        What is MongoDB?
//-----------------------------------------------------------
```

Let's learn what MongoDB actually is, how it works, and a quick overview of how it compares to more traditional databases. And let's begin with a simple overview.



So ***MongoDB is obviously a database, and it's a so-called NoSQL database***. Now some people also say No S Q L, but I'm just gonna keep saying "*no sequel*", all right? Now, the other type of database, which is sort of more traditional, is the relational database, which NoSQL is often compared to.

Anyway, in Mongo, which we can also say instead of MongoDB, *each database can contain one or more collections*. So if you actually are coming from one of these more traditional relational database systems, you can think of a collection as a table of data.



```
{
  "title": "Get Started With MongoDB",
  "length": 1453,
```

```
{
  "title": "Integrate MongoDB into Node.js",
  "length": 2544,
```

```
{
  "title": "MongoDB for Online Stores",
  "length": 1931,
```

```
{
  "title": "Rockets, Cars and MongoDB",
  "length": 3280,
  "author": "Elon Musk",
  "tags": ["MongoDB", "space", "ev"]
}
```

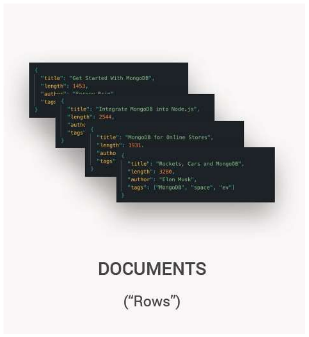
Then, *each collection can contain one or more data structures called documents*, and again, in a relational database, a document would be a row in a table.

So each document contains the data about one single entity, for example, one blog post or one user or one review, or anything else, really. You get the point, right?

Now the collection is like the parent structure that contains all these entities. For example, a blog collection for all posts, a users collection or a reviews collection.

blog
users
reviews
post
user
review

And you can also see here that the *document has a data format that looks a lot like JSON*, which will make our work a lot easier when we start dealing with these documents.



```
{
  "title": "Get Started With MongoDB",
  "length": 1453,
```

```
{
  "title": "Integrate MongoDB into Node.js",
  "length": 2544,
```

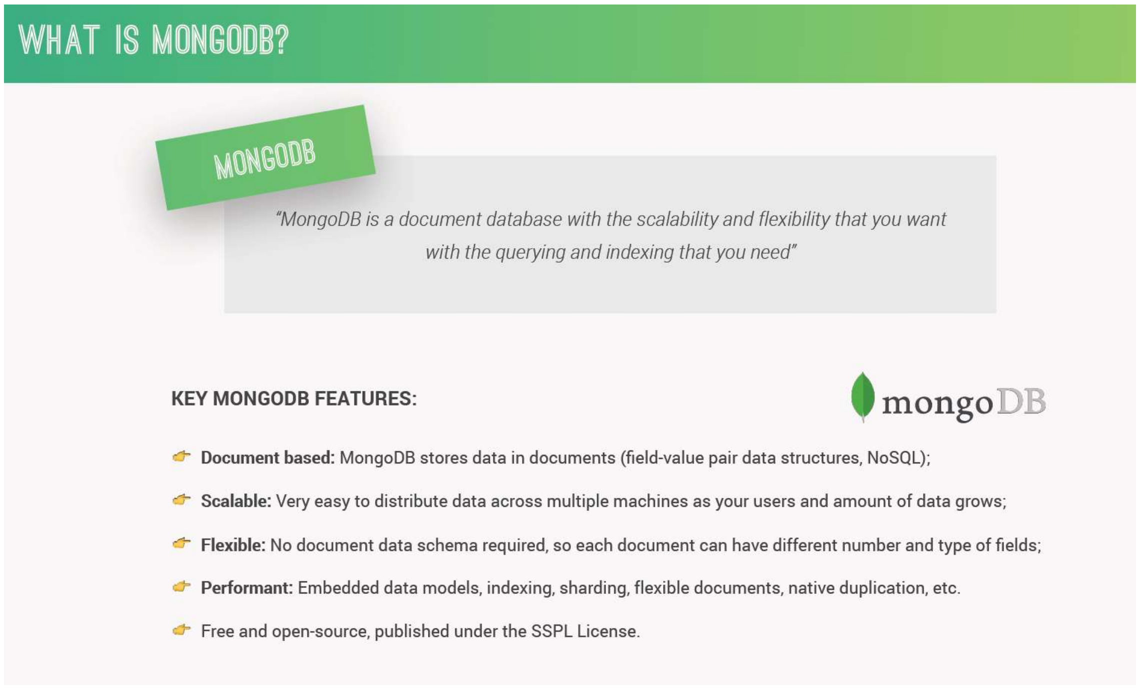
```
{
  "title": "MongoDB for Online Stores",
  "length": 1931,
```

```
{
  "title": "Rockets, Cars and MongoDB",
  "length": 3280,
  "author": "Elon Musk",
  "tags": ["MongoDB", "space", "ev"]
}
```

And of course we will talk a lot about this later, but for now on, let's learn about Mongo's main features.

So, according to MongoDB's website,

MongoDB is a document database with the scalability and flexibility that you want, and with the querying and indexing that you need.



WHAT IS MONGODB?

MONGODB

"MongoDB is a document database with the scalability and flexibility that you want with the querying and indexing that you need"

KEY MONGODB FEATURES:

- **Document based:** MongoDB stores data in documents (field-value pair data structures, NoSQL);
- **Scalable:** Very easy to distribute data across multiple machines as your users and amount of data grows;
- **Flexible:** No document data schema required, so each document can have different number and type of fields;
- **Performant:** Embedded data models, indexing, sharding, flexible documents, native duplication, etc.
- Free and open-source, published under the SSPL License.

Now, that sounds a bit over the top, so let's try to understand what this actually means. So, as we saw before,

1. **Document Based**

MongoDB is a document-based database, so it stores data in documents which are field-value paired data structures like JSON. So again, it stores data in these document instead of rows in a table like in traditional relational databases. It's therefore a NoSQL database and not a relational one.

2. **Scalable**

Also, *MongoDB has built-in scalability, making it very easy to distribute data across multiple machines as your apps get more and more users and starts generating a ton of data.* So whatever you do, MongoDB will make it very easy for you to grow.

3. **Flexible**

Next up, another big feature of MongoDB is its great flexibility. So *there is no need to define a document data schema before filling it with data, meaning that each document can have a different number and type of fields.* And we can also change these fields all the time.

And all this is really in line with some real-world business situations, and therefore can become pretty useful.

4. **Performant**

MongoDB is also a very performant database system. Thanks to *features like embedded data models, indexing, sharding, the flexible documents* that we already talked about, native duplication and so much more. And you don't need to know all of this, of course, but it's sure nice to know that MongoDB is highly performant if we need it to be.

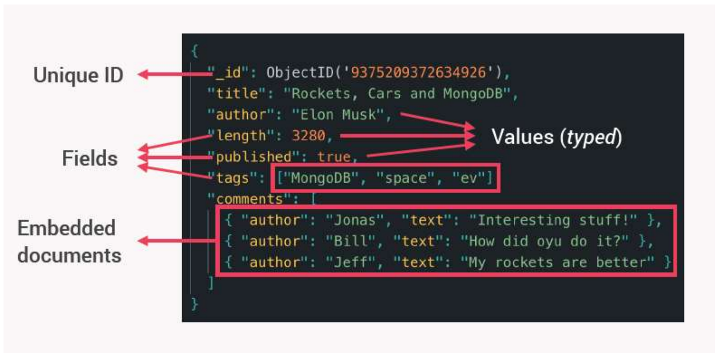
5. **Free and Open Sources, published under the SSPL Licence**

Finally, I just wanted to add that MongoDB is a free and open-source database, published under the SSPL license. So in *summary, we can say that MongoDB is a great database system to build many types of modern, scalable and flexible web applications.*

MongoDB is probably the most used database with NodeJS, and so it's a perfect fit for us to use in this course.

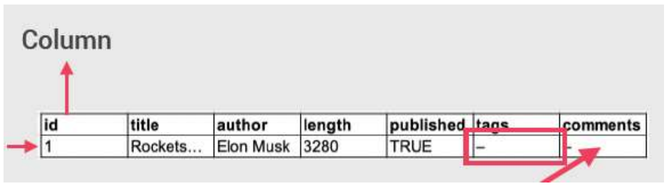
Okay, now let's talk a bit deeper about these documents, and returning to our blog posts example from the beginning, this could be a very simple representation of a single post document, right?

DOCUMENT STRUCTURE



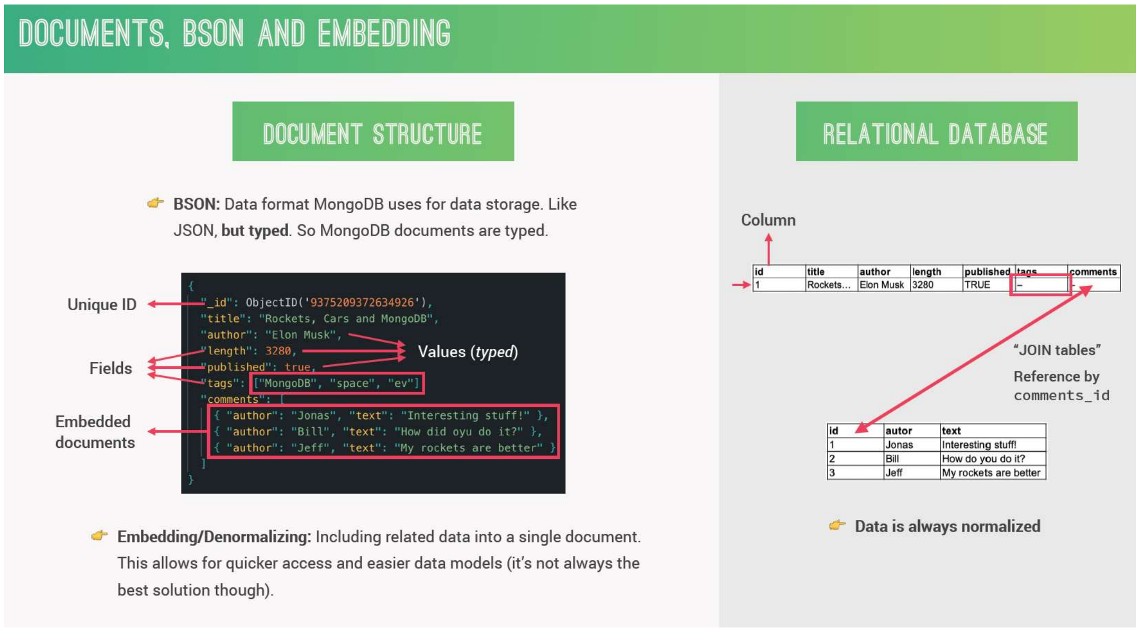
```
{
  "_id": ObjectID('9375209372634926'),
  "title": "Rockets, Cars and MongoDB",
  "author": "Elon Musk",
  "length": 3280,
  "published": true,
  "tags": ["MongoDB", "space", "ev"]
  "comments": [
    { "author": "Jonas", "text": "Interesting stuff!" },
    { "author": "Bill", "text": "How did oyu do it?" },
    { "author": "Jeff", "text": "My rockets are better" }
  ]
}
```

RELATIONAL DATABASE



id	title	author	length	published	tags	comments
1	Rockets...	Elon Musk	3280	TRUE	–	-

And now just for the sake of comparison, here is how that exact same data could look like as a row in a relational database like MySQL, or even in an Excel spreadsheet, if you're more used to that.



DOCUMENTS, BSON AND EMBEDDING

DOCUMENT STRUCTURE

👉 **BSON:** Data format MongoDB uses for data storage. Like JSON, **but typed**. So MongoDB documents are typed.

```
{
  "_id": ObjectID('9375209372634926'),
  "title": "Rockets, Cars and MongoDB",
  "author": "Elon Musk",
  "length": 3280,
  "published": true,
  "tags": ["MongoDB", "space", "ev"]
  "comments": [
    { "author": "Jonas", "text": "Interesting stuff!" },
    { "author": "Bill", "text": "How did oyu do it?" },
    { "author": "Jeff", "text": "My rockets are better" }
  ]
}
```

👉 **Embedding/Denormalizing:** Including related data into a single document. This allows for quicker access and easier data models (it's not always the best solution though).

RELATIONAL DATABASE

id	title	author	length	published	tags	comments
1	Rockets...	Elon Musk	3280	TRUE	–	-

id	autor	text
1	Jonas	Interesting stuff!
2	Bill	How do you do it?
3	Jeff	My rockets are better

👉 **Data is always normalized**

So as I mentioned a bit earlier, **MongoDB uses a data format similar to JSON for data storage called BSON (Binary JavaScript Object Notation)**

BSON is a serialization format encoding format for JSON mainly used for storing and accessing the documents, whereas JSON is a human-readable standard file format mainly used for transmission of data in the form of key-value attribute pairs. ... BSON, in fact, in some cases, uses more space than JSON

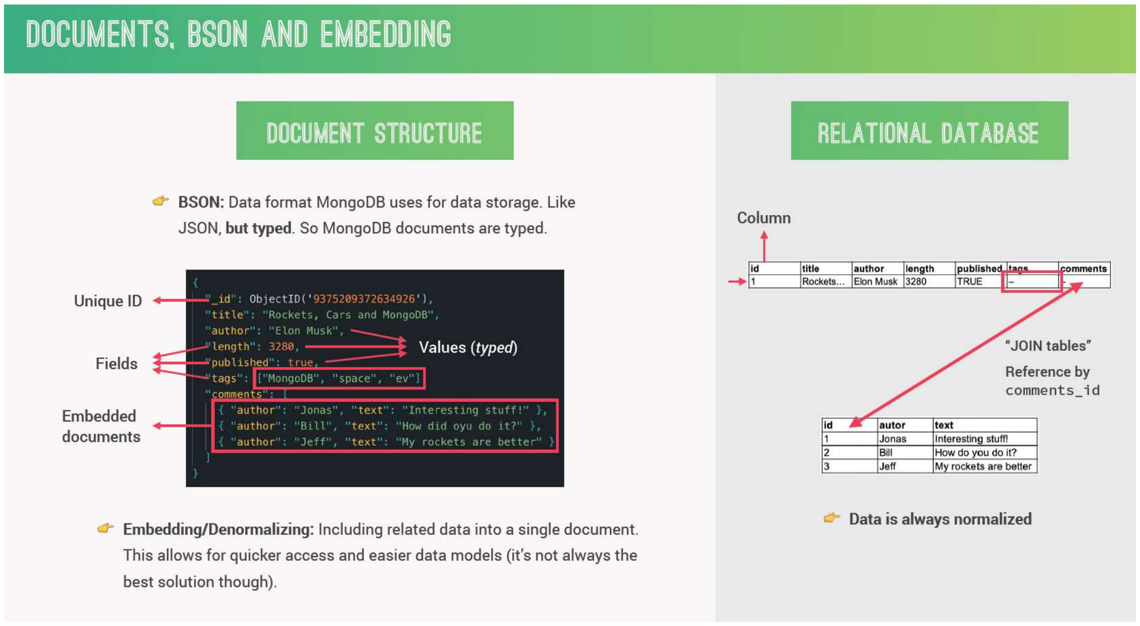
It looks basically the same as JSON, but it's typed, meaning that all values will have a data type such as string, Boolean, date and teacher, double object or more.

We will learn all about this later in practice. So what this means is that all MongoDB documents will actually be typed, which is different from JSON, all right?

Now just like JSON, these BSON documents will also have fields and data is stored in key value pairs. On the other hand, in a relational database, each field is called a column.

So here again you can see how these databases arrange data in table structures while ***our JSON data is so much more flexible***. Take for example the tags field, where we actually have an array, so we have basically multiple values for one field, right?

So MongoDB, space and ev in this case. But in relational databases, that's not really allowed. We cannot have multiple values in one field, and so we would actually have to find workarounds for this which could then involve more work and more overall complication.



DOCUMENTS, BSON AND EMBEDDING

DOCUMENT STRUCTURE

👉 **BSON:** Data format MongoDB uses for data storage. Like JSON, **but typed**. So MongoDB documents are typed.

```
{
  "_id": ObjectID('9375209372634926'),
  "title": "Rockets, Cars and MongoDB",
  "author": "Elon Musk",
  "length": 3280,
  "published": true,
  "tags": ["MongoDB", "space", "ev"]
  "comments": [
    { "author": "Jonas", "text": "Interesting stuff!" },
    { "author": "Bill", "text": "How did oyu do it?" },
    { "author": "Jeff", "text": "My rockets are better" }
  ]
}
```

👉 **Embedding/Denormalizing:** Including related data into a single document. This allows for quicker access and easier data models (it's not always the best solution though).

RELATIONAL DATABASE

id	title	author	length	published	tags	comments
1	Rockets…	Elon Musk	3280	TRUE	–	-

id	autor	text
1	Jonas	Interesting stuff!
2	Bill	How do you do it?
3	Jeff	My rockets are better

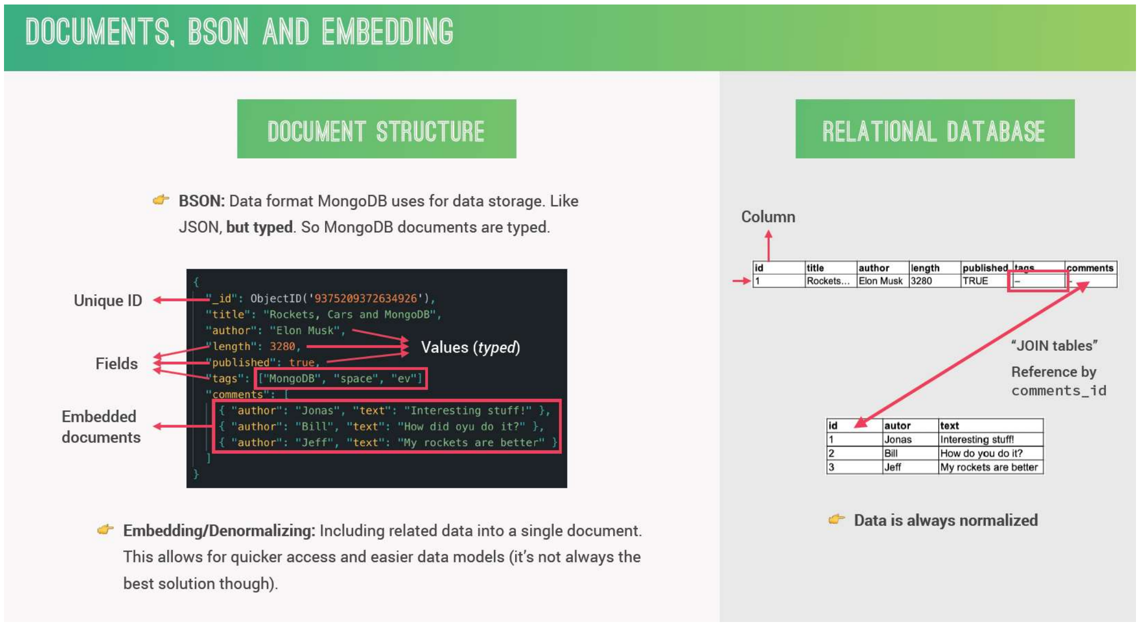
👉 **Data is always normalized**

Now ***another extremely important feature in MongoDB is the concept of embedded documents***, which is, again, something not present in relational databases.

So in our comments field here, we have an array which contains three objects. One for each document, and each of them could actually be its own document, right? So just imagine we had a comments collection which contained a bunch of comment documents. Each of them could actually look exactly like this, so with an author and with the comment text, but instead of doing that, we include these comments right into the blog post document, so in other words, we embed the comment documents right into the post document. So this process of embedding, or de-normalizing as we can also call it, is basically to include, so, to embed, some related data all into one single document. In this example, the comments are related to the post, and so they are included in the same document.

And *this makes a database more performant in some situations because this way, it can be easier to read all the data that we need all at once*.

And this is something that we're gonna talk about a lot when learning about data modeling, but for now, I hope that this still makes sense to you.



DOCUMENTS, BSON AND EMBEDDING

DOCUMENT STRUCTURE

👉 **BSON:** Data format MongoDB uses for data storage. Like JSON, **but typed**. So MongoDB documents are typed.

```
{
  "_id": ObjectID('9375209372634926'),
  "title": "Rockets, Cars and MongoDB",
  "author": "Elon Musk",
  "length": 3280,
  "published": true,
  "tags": ["MongoDB", "space", "ev"]
  "comments": [
    { "author": "Jonas", "text": "Interesting stuff!" },
    { "author": "Bill", "text": "How did oyu do it?" },
    { "author": "Jeff", "text": "My rockets are better" }
  ]
}
```

👉 **Embedding/Denormalizing:** Including related data into a single document. This allows for quicker access and easier data models (it's not always the best solution though).

RELATIONAL DATABASE

id	title	author	length	published	tags	comments
1	Rockets…	Elon Musk	3280	TRUE	–	-

id	autor	text
1	Jonas	Interesting stuff!
2	Bill	How do you do it?
3	Jeff	My rockets are better

👉 **Data is always normalized**

Now, the opposite of embedding or de-normalizing, is normalizing, and that's how the data is always modeled in relational databases. So in that case, it's not possible to embed data, and so the solution is to create a whole new table for the comments and then join the tables by referencing to the ID field of the comments table. Now we're not gonna use relational databases in this course, but I believe it's still important to know the differences if you wanna become a good back-end developer. Anyway, and now just to finish, *two more things about BSON documents*.

1. First, **the maximum size for each document is currently 16 MB, but this might increase in the future.**

2. And second, **each document contains a unique ID, which acts as a primary key of that document. It's automatically generated with the object ID data type each time there is a new document**, and so we don't have to worry about it. All right, and that should be a brief enough overview to get us started,

and to actually use MongoDB from the next lecture on. So, let's move on now.

JSON vs BSON Comparison Table

Below is the topmost Comparison between JSON vs BSON :

The basis Of Comparison	JSON	BSON
Type	Standard file format	Binary file format
Speed	Comparatively less fast	Faster
Space	Consumes comparatively less space.	More space is consumed.
Usage	Transmission of data.	Storage of data.
Encoding and Decoding technique	No such technique.	Faster encoding and decoding technique.
Characteristics	Key-value pair only used for transmission of data.	Lightweight, fast and traversable.
Structure	Language independent format used for asynchronous server browser communication.	Binary JSON which consist of a list of ordered elements containing a field name, type, and value. Field name types are typically a string.
Traversal	JSON doesn't skip rather skims through all the content.	BSON, on the other hand, just indexes on the relevant content and skips all the content which does not have to be in use.
Parse	JSON formats need not be parsed as they are in a human-readable format already.	BSON, on the other hand, needs to be parsed as they are easy for machines to parse and generate.
Creation type	Broadly JSON consists of an object and an array where the object is a collection of key-value pairs, and the array is an ordered list of values.	The binary encoding technique consists of additional information such as lengths of strings and the object subtypes. Moreover, BinData and Date data types are the data types that are not supported in JSON.

```
//----------------------------------------------------------
//                        LECTURE 71                ⭐⭐
// Installing MongoDB on MacOS
//----------------------------------------------------------
```

Please REFER LECTURE VIDEO

```
//----------------------------------------------------------
//                        LECTURE 72                ⭐⭐
// Installing MongoDB on Windows
//----------------------------------------------------------
```

Please REFER LECTURE VIDEO

```
//---------------------------------------------------------------
//                          LECTURE 73                ⭐⭐
//  Creating a Local Database
//---------------------------------------------------------------
```

Basic Operations/Commands for MongoDB

1. ***Create new database or switch to an existing database:***

```
use natours-test
```

2. ***Inserting Many Document in the database:***

```
db.tours.insertMany([{  },{  },{  }])
```

3. ***Inserting One Document in the database:***

```
db.tours.insertOne({ name: "The Forest Hiker", price: 297, rating: 4.7})
```

4. ***Find data in the current database:***

```
db.tours.find()
```

5. ***Show all databases:***

```
show dbs
```

6. ***See all the collections in the current database:***

```
show collections
```

7. ***exit the shell:***

```
quit()
```

```
//------------------------------------------------------------
//                          LECTURE 74                ☆☆
// CRUD: Creating Documents
//------------------------------------------------------------
```

Step 1: *Switching to database/ Creates a database if not already there*

```
> use natours-test

switched to db natours-test
```

Step 2: *Inserting Many Documents into a collection. In our case, "tours" collections*

```
> db.tours.insertMany([{name: "The Sea Explorer", price: 497, rating: 4.8}, {name: "The Desert Rover", price: 997, rating: 4.9, difficulty: "easy"}])

{
        "acknowledged" : true,
        "insertedIds" : [
                ObjectId("618a740e1fefeaf415d35e94"),
                ObjectId("618a740e1fefeaf415d35e95")
        ]
}
```

Step 3: *Find data in the current database collections, In our case, "tours" collections*

```
> db.tours.find()

{ "_id" : ObjectId("618947e3e22121bdcd9f6ca0"), "name" : "The Forest Hiker", "price" : 297, "rating" : 4.7 }

{ "_id" : ObjectId("618a6f52379e744549fc8981"), "name" : "The Snow Adventurer", "price" : 597, "rating" : 4.8 }

{ "_id" : ObjectId("618a740e1fefeaf415d35e94"), "name" : "The Sea Explorer", "price" : 497, "rating" : 4.8 }

{ "_id" : ObjectId("618a740e1fefeaf415d35e95"), "name" : "The Desert Rover", "price" : 997, "rating" : 4.9, "difficulty" : "easy" }
```

```
//----------------------------------------------------------
//                         LECTURE 75                ☆☆
//   CRUD: Querying (Reading) Documents
//----------------------------------------------------------
```

1. *Searching all the documents in a collection, without any searching criterion*

```
> db.tours.find()
```

```
{ "_id" : ObjectId("618947e3e22121bdcd9f6ca0"), "name" : "The Forest Hiker", "price" : 297,
"rating" : 4.7 }

{ "_id" : ObjectId("618a6f52379e744549fc8981"), "name" : "The Snow Adventurer", "price" : 597,
"rating" : 4.8 }

{ "_id" : ObjectId("618a740e1fefeaf415d35e94"), "name" : "The Sea Explorer", "price" : 497,
"rating" : 4.8 }

{ "_id" : ObjectId("618a740e1fefeaf415d35e95"), "name" : "The Desert Rover", "price" : 997,
"rating" : 4.9, "difficulty" : "easy" }
```

2. *Seaching a single document in a tours collection, with a search criterion, in our case, the name of a document*

```
> db.tours.find({name: "The Forest Hiker"})
```

```
{ "_id" : ObjectId("618947e3e22121bdcd9f6ca0"), "name" : "The Forest Hiker", "price" : 297,
"rating" : 4.7 }
```

3. *Searching all documents in a tours collection, with a search criterion, in our case, the difficulty level*

```
> db.tours.find({"difficulty": "easy"})
```

```
{ "_id" : ObjectId("618a740e1fefeaf415d35e95"), "name" : "The Desert Rover", "price" : 997,
"rating" : 4.9, "difficulty" : "easy" }
```

4. *Searching all documents in a tours collection, with all the tours with prices lower then 500 or equal: Using QUERY OPERATORS in MongoDB*

$lte *: "***$***" is a reserved for using operators in MongoDB,* **lte** *stands for less than or equal to*

```
> db.tours.find({ price: {$lte: 500} })
```

```
{ "_id" : ObjectId("618947e3e22121bdcd9f6ca0"), "name" : "The Forest Hiker", "price" : 297,
"rating" : 4.7 }

{ "_id" : ObjectId("618a740e1fefeaf415d35e94"), "name" : "The Sea Explorer", "price" : 497,
"rating" : 4.8 }
```

5. *Searching all documents in a tours collection, with all the tours with price less than 500 and with a rating of atleast 4.8 : Using two search criterions* **--→ AND QUERY**

$gte *: "***$***" is a reserved for using operators in MongoDB,* **gte** *stands for greater than or equal to*

```
> db.tours.find({ price: {$lt: 500}, rating: {$gte: 4.8} })
```

```
{ "_id" : ObjectId("618a740e1fefeaf415d35e94"), "name" : "The Sea Explorer", "price" : 497,
"rating" : 4.8 }
```

6. *Searching all documents in a tours collection, with all the tours with price less than 500 or with a rating of atleast 4.8 : Using two search criterions* **--→ OR QUERY**

```
> db.tours.find({ $or: [{price: {$lt: 500}}, {rating: {$gte: 4.8}}]
})
```

```
{ "_id" : ObjectId("618947e3e22121bdcd9f6ca0"), "name" : "The Forest Hiker", "price" : 297,
"rating" : 4.7 }

{ "_id" : ObjectId("618a6f52379e744549fc8981"), "name" : "The Snow Adventurer", "price" : 597,
"rating" : 4.8 }

{ "_id" : ObjectId("618a740e1fefeaf415d35e94"), "name" : "The Sea Explorer", "price" : 497,
"rating" : 4.8 }

{ "_id" : ObjectId("618a740e1fefeaf415d35e95"), "name" : "The Desert Rover", "price" : 997,
"rating" : 4.9, "difficulty" : "easy" }
```

7. *Searching all documents in a tours collection, with all the tours with price greater than 500 or with a rating of atleast 4.8 : Using two search criterions* **--→ OR QUERY**

```
> db.tours.find({ $or: [{price: {$gt: 500}}, {rating: {$gte: 4.8}}]
})
```

```
{ "_id" : ObjectId("618a6f52379e744549fc8981"), "name" : "The Snow Adventurer", "price" : 597,
"rating" : 4.8 }

{ "_id" : ObjectId("618a740e1fefeaf415d35e94"), "name" : "The Sea Explorer", "price" : 497,
"rating" : 4.8 }

{ "_id" : ObjectId("618a740e1fefeaf415d35e95"), "name" : "The Desert Rover", "price" : 997,
"rating" : 4.9, "difficulty" : "easy" }
```

8. *Searching all documents in a tours collection, with all the tours with price greater than 500 or with a rating of atleast 4.8 : Using two search criterions* **--→ OR QUERY As well as Passing an OBJECT for PROJECTION**

PROJECTION means that we simply want to select some of the fields in the output. All we have to do is, for example, say name equals to 1 . So, what this means is that we only want the name to be in the output and so that's why we set name to 1. All the others are not gonna appear in this case.

```
> db.tours.find({ $or: [{price: {$gt: 500}}, {rating: {$gte: 4.8}}]
}, {name: 1})
```

```
{ "_id" : ObjectId("618a6f52379e744549fc8981"), "name" : "The Snow Adventurer" }

{ "_id" : ObjectId("618a740e1fefeaf415d35e94"), "name" : "The Sea Explorer" }

{ "_id" : ObjectId("618a740e1fefeaf415d35e95"), "name" : "The Desert Rover" }
```

```
//-----------------------------------------------------------
//                         LECTURE 76                ☆☆
//   CRUD: Updating Documents
//-----------------------------------------------------------
```

1. *Updating One document into a collection. In our case, "tours" collections*

```
> db.tours.updateOne({ name: "The Snow Adventurer"}, { $set: {price: 897} })
```

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

```
> db.tours.find( {name: "The Snow Adventurer"} )
```

```
{ "_id" : ObjectId("618a6f52379e744549fc8981"), "name" : "The Snow Adventurer", "price" : 897,
"rating" : 4.8 }
```

2. *Updating One/Many documents into a collection. In our case, "tours" collections*

Challenge: Find a tour with the price greater than or equal to 500, and rating greater than or equal to 4.8, and ***update it adding a premium field and set that to true***, *and then check all if its updated in the tours collection*

```
> db.tours.find({ price: {$gte: 500}, rating: {$gte: 4.8} })
```

```
{ "_id" : ObjectId("618a6f52379e744549fc8981"), "name" : "The Snow Adventurer", "price" : 897,
"rating" : 4.8 }

{ "_id" : ObjectId("618a740e1fefeaf415d35e95"), "name" : "The Desert Rover", "price" : 997,
"rating" : 4.9, "difficulty" : "easy" }
```

```
> db.tours.updateMany({ price: {$gte: 500}, rating: {$gte: 4.8} }, {
$set: {premium: true}})
```

```
{ "acknowledged" : true, "matchedCount" : 2, "modifiedCount" : 2 }
```

```
> db.tours.find()
```

```
{ "_id" : ObjectId("618947e3e22121bdcd9f6ca0"), "name" : "The Forest Hiker", "price" : 297,
"rating" : 4.7 }

{ "_id" : ObjectId("618a6f52379e744549fc8981"), "name" : "The Snow Adventurer", "price" : 897,
"rating" : 4.8, "premium" : true }

{ "_id" : ObjectId("618a740e1fefeaf415d35e94"), "name" : "The Sea Explorer", "price" : 497,
"rating" : 4.8 }

{ "_id" : ObjectId("618a740e1fefeaf415d35e95"), "name" : "The Desert Rover", "price" : 997,
"rating" : 4.9, "difficulty" : "easy", "premium" : true }
```

3. *Replacing the contents of one/many documents in a certain collection. In our case, “tours” collections*

- db.tours.replaceOne()

- db.tours.replaceMany()

So with this .updateMany or .updateOne we usually only update parts of the document, but we can also completely replace the content of the document.

And for that we use .replaceOne. I'm not going to do that, but for the sake of completeness I wanted to show it to you as well. So db.tours.replaceOne or .replaceMany, okay?

And so in here, just like before, you would pass the search query and then new data that you want to put in this document.

```
//-----------------------------------------------------------
//                          LECTURE 77                ☆☆
//   CRUD: Deleting Documents
//-----------------------------------------------------------
```

Finally, let's learn how to delete documents in MongoDB. So just like before, we have delete one to delete one single document and we have delete many, well to delete multiple documents at the same time.

Alright, and just like before, delete one will only work or the first document matching your query. And delete many will of course work for all the documents matching your query.

DELETING ALL THE TOURS WITH THE RATING LESS THAN 4.8

```
> db.tours.find()

{ "_id" : ObjectId("618947e3e22121bdcd9f6ca0"), "name" : "The Forest Hiker", "price" : 297,
"rating" : 4.7 }

{ "_id" : ObjectId("618a6f52379e744549fc8981"), "name" : "The Snow Adventurer", "price" : 897,
"rating" : 4.8, "premium" : true }

{ "_id" : ObjectId("618a740e1fefeaf415d35e94"), "name" : "The Sea Explorer", "price" : 497,
"rating" : 4.8 }

{ "_id" : ObjectId("618a740e1fefeaf415d35e95"), "name" : "The Desert Rover", "price" : 997,
"rating" : 4.9, "difficulty" : "easy", "premium" : true }
```

```
> db.tours.deleteMany({ rating: {$lt: 4.8}})

{ "acknowledged" : true, "deletedCount" : 1 }
```

```
> db.tours.find()

{ "_id" : ObjectId("618a6f52379e744549fc8981"), "name" : "The Snow Adventurer", "price" : 897,
"rating" : 4.8, "premium" : true }

{ "_id" : ObjectId("618a740e1fefeaf415d35e94"), "name" : "The Sea Explorer", "price" : 497,
"rating" : 4.8 }

{ "_id" : ObjectId("618a740e1fefeaf415d35e95"), "name" : "The Desert Rover", "price" : 997,
"rating" : 4.9, "difficulty" : "easy", "premium" : true }
```

DELETING ALL THE DOCUMENTS IN A COLLECTION : BE VERY CAREFULL WHILE USING THIS IN THE REAL WORLD APPLCATION

```
> db.tours.deleteMany({})
```

```
//---------------------------------------------------------------
//                         LECTURE 78                ☆☆
//   Using COMPASS App for CRUD Operations
//---------------------------------------------------------------
```

COMPASS App is the GUI or Graphical User Interface for doing CRUD Operations in MongoDB.

Its an alternative to using the command line terminal for using MongoDB

Please Follow the VIDEO LECTURE for these.

```
//---------------------------------------------------------------
//                         LECTURE 79                ☆☆
//   Creating HOSTED DATABASE with ATLAS
//---------------------------------------------------------------
```

```
//---------------------------------------------------------------
//                         LECTURE 80                ☆☆
//   Connecting to our HOSTED DATABASE
//---------------------------------------------------------------
```

Node.js

SECTION 08 - Using MongoDB with Mongoose

EC-137:

Sinha, A

```
//-----------------------------------------------------------
//                         LECTURE 82                 ☆☆
//   Connecting our Dtabase with the Express App
//-----------------------------------------------------------
```

PLEASE FOLLOW ALONG WITH THE VIDEO LECTURE INCLUDING THE CODE

● ● **IMPORTANT NOTES**

1. **DO NOT Forget to put in the database name at the end of the string, copied from AtlasDB. Else the connection wont be established**
2. **While using a local database, always keep the server running, else thw connection with the database won't work.**
3. **The 2nd argument for the mongoose.connect, is not that important. Its to deal with some deprecation warnings.** ***While creating my own application in the future, I can use the same*****.**
4. **The mongoose.connect, returns a promise, so, we have to handle that promise using the "then" method.**

Filename: **config.env**

```
NODE_ENV=development
PORT=3000

DATABASE=mongodb+srv://ankur:<PASSWORD>@cluster0.59dgp.mongodb.net/n
atours?retryWrites=true&w=majority

DATABASE_LOCAL=mongodb://localhost:27017/natours

DATABASE_PASSWORD=
```

And, that's it for configuration.

Next up, we need to install a MongoDB driver,

so basically a software that allows our Node code to access and interact with a MongoDB database And, there are a couple of different MongoDB drivers, but we're gonna use the one that I would say is the most popular one, which is called ***Mongoose***, which adds a couple of features
to the more native MongoDB driver.

```
---------------------
Installing Mongoose: npm install mongoose@5
---------------------
```

Filename: **server.js**

```
const mongoose = require("mongoose");
const dotenv = require("dotenv");
const app = require("./app");

dotenv.config({ path: "./config.env" });

const DB = process.env.DATABASE.replace(
  "<PASSWORD>",
  process.env.DATABASE_PASSWORD
);

//--------------------------------
// * Connecting and checking the connection with the HOSTED Datbase
//--------------------------------
/*

mongoose
  .connect(DB, {
    useNewUrlParser: true,
    useCreateIndex: true,
    useFindAndModify: false,
  })
  .then((con) => {
    console.log(con.connections);
    console.log("Database connection successful!");
  });

*/
//--------------------------------
// ⋆⋆ Connecting and checking the connection with the LOCAL
Database
//--------------------------------
/*

mongoose
  .connect(process.env.DATABASE_LOCAL, {
    useNewUrlParser: true,
    useCreateIndex: true,
    useFindAndModify: false,
  })
  .then((con) => {
    console.log(con.connections);
    console.log("Database connection successful!");
  });

  */

mongoose
  .connect(DB, {
    useNewUrlParser: true,
    useCreateIndex: true,
    useFindAndModify: false,
  })
  .then(() => console.log("Database connection successful!"));

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

```
});
```

Now, of course, there could also be problems connecting to the database.
For example, the host might be down or we might have some error in our connection string.

And, in that case, we should catch that error. But, we will leave error handling for a bit later in the course and so for now, I'm not having any catch method here.

Just the then method here, assuming that everything works fine.

```
//-----------------------------------------------------------
//       Modified in 2022 - GLOBAL MODIFICATION       ☆ ☆
// UPDATED server.js File:
// Getting Rid of Deprecation Warnings in the console
//-----------------------------------------------------------
```

IMPORTANT: *THIS MODIFICATION IS* ***NOT REQUIRED WHEN USING Mongoose Version 6 or Higher.*** All the previous versions require this in order to get rid of the deprecation warnings in the console.

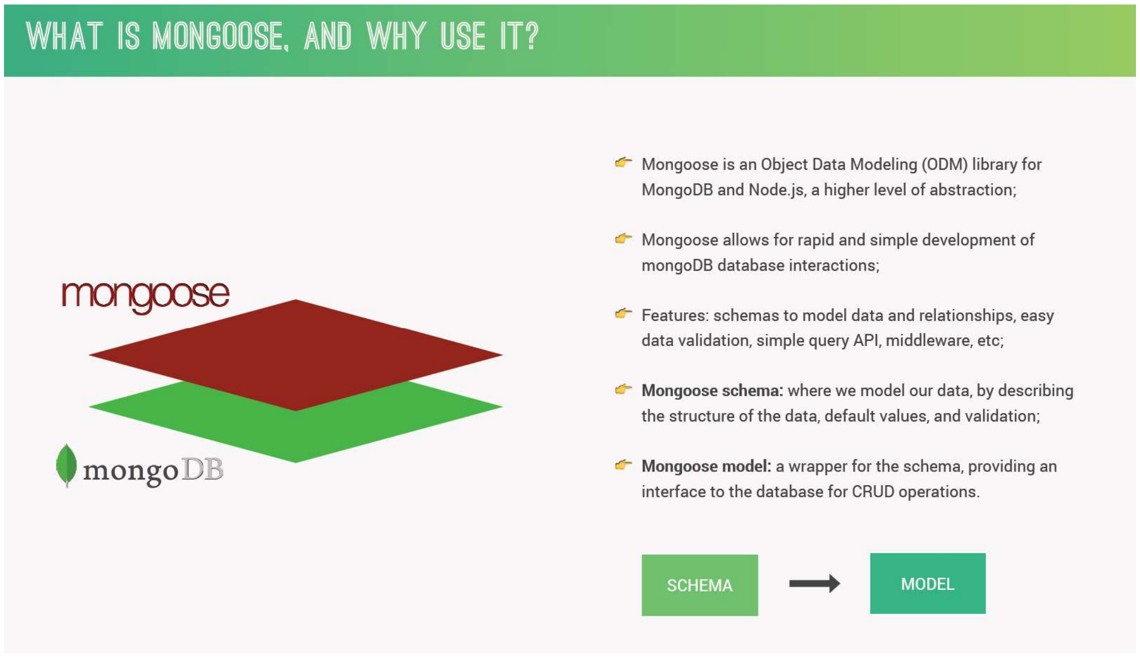
```
const mongoose = require("mongoose");
const dotenv = require("dotenv");

dotenv.config({ path: "./config.env" });

const app = require("./app");

const DB = process.env.DATABASE.replace(
  "<PASSWORD>",
  process.env.DATABASE_PASSWORD
);

// to connect to MongoDB Altas Cloud Database
mongoose
  .connect(DB, {
    useNewUrlParser: true,
    useCreateIndex: true,
    useFindAndModify: false,
    useUnifiedTopology: true,       ← ✸ Added/Modified
this
  })
  .then(() => {
    console.log("Database connection successful!");
  });

// To connect to the local database
// mongoose
//   .connect(process.env.DATABASE_LOCAL, {
//     useNewUrlParser: true,
//     useCreateIndex: true,
//     useFindAndModify: false,
//     useUnifiedTopology: true,
//   })
//   .then(() => {
//     console.log("Database connection successful!");
//   });

const port = process.env.PORT || 3000;

app.listen(port, () => {
  console.log(`App running on port ${port}...`);
});
```

```
//------------------------------------------------------------
//                        LECTURE 83                   ☆ ☆
//   What is Mongoose?
//------------------------------------------------------------
```

So, we connected our application with the database using Mongoose in the last lecture. But wait,

What actually is Mongoose?

Well, Mongoose is an object data modeling library for MongoDB and Node JS, providing a higher level of abstraction.



WHAT IS MONGOOSE, AND WHY USE IT?

- Mongoose is an Object Data Modeling (ODM) library for MongoDB and Node.js, a higher level of abstraction;
- Mongoose allows for rapid and simple development of mongoDB database interactions;
- Features: schemas to model data and relationships, easy data validation, simple query API, middleware, etc;
- **Mongoose schema:** where we model our data, by describing the structure of the data, default values, and validation;
- **Mongoose model:** a wrapper for the schema, providing an interface to the database for CRUD operations.

So, it's a bit like the relationship between Express and Node,

Express is a layer of abstraction over regular Node, while Mongoose is a layer of abstraction over the regular MongoDB driver.

And by the way,

<u>An Object Data Modeling library (ODM) is just a way for us to write JavaScript code that will then interact with a database.</u>

So, we could just use a regular MongoDB driver to access our database, and it would work just fine, but instead we use Mongoose, because it gives us a lot more functionality out of the box, allowing for faster and simpler development of our applications.

So, some of the features Mongoose give us are

1. Schemas to model our data and relationships
2. Easy data validation
3. Simple query API
4. Middleware and much more

A ***Mongoose schema*** *is where we model our data, so where we describe the structure of the data, default values, and validation. We then take that schema and create a model out of it.*

And the ***Model*** *is basically a wrapper around the schema, which allows us to actually interface/interact with the database in order to create, delete, update, and read documents [CRUD operations]. A model is like a blueprint to create documents.*

All right, so this was just a very quick introduction.

Now, let's actually go ahead and create a simple schema and model

```
//-----------------------------------------------------------
//                        LECTURE 84                ⭐⭐
//   Creating a simple tour model
//-----------------------------------------------------------
```

*Mongoose is all about models. A **Model** is like a blueprint to create documents. Its a bit like **classes** in JavaScript which we also use, as a blueprint, in order to create objects out of them.*

1. *We need a model in Mongoose, in order to perform the CRUD Operations.*
2. *In order to create a model, we need a schema.*

Using a schema we model our data, describe the structure of the data, default values, and validation. We then take that schema and create a model out of it.

● NOTE: Mongoose uses the native JavaScript Data Types

● ● NOTE : We are currently using server.js as our file, but we will place these codes, in a different module/file. This one is just for testing purposes.

```
// The Most Basic way of DESCRIBING OUR DATA/SCHEMA

const tourSchema = new mongoose.Schema({
  name: String,
  rating: Number,
  price: Number,
});
```

So again, this is the most basic way of describing a schema, but we can take it one step further by defining something called ***schema type options for each field***, or only for some specific field.

Filename: **server.js**

```
const mongoose = require("mongoose");
const dotenv = require("dotenv");
const app = require("./app");

dotenv.config({ path: "./config.env" });

const DB = process.env.DATABASE.replace(
  "<PASSWORD>",
  process.env.DATABASE_PASSWORD
);

mongoose
  .connect(DB, {
    useNewUrlParser: true,
    useCreateIndex: true,
    useFindAndModify: false,
  })
  .then(() => console.log("Database connection successful!"));

// The Most Basic way of DESCRIBING OUR Data which acts as a
blueprint

// const tourSchema = new mongoose.Schema({
//   name: String,
//   rating: Number,
//   price: Number,
// });

//---------------------------------------
// ⭐ NOTES: Proper way of describing a Schema w/ Schema Type
Options
//---------------------------------------

// default: 4.5:  The default rating value of any tour, using this
schema will be 4.5
```

```
// unique: true: This ensures that the name of the tour should be
UNIQUE
// required: [true, "Error String: A tour must have a price"] : This
is called a VALIDATOR, as it validates our Data, In this case simply
to validate if the name is actually there.

// There are a lot of other validators in Mongoose, and we can even
create our CUSTOM VALIDATORS, But we ll see if we need that further
down the Project.

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

//----------------------------------------
// ⭐⭐⭐ Creating a Model Using the Schema  we just described
//----------------------------------------

// ● In programming a general Convention is that, the Uppercase is
used in Model Names
// ("Tour", tourSchema): (Model_Name, Schema_Name)

// In the next Lecture, It will be used to Create Our Very First
Tour Document

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

```
//---------------------------------------------------------
//                        LECTURE 85                ☆☆
//   Creating documents for Tour Model and testing the Model
//---------------------------------------------------------
```

● ● **NOTE : We are currently using server.js as our file, but we will place these codes, in a different module/file. The testTour document that we will create is just for testing purposes.**

Filename: **server.js**

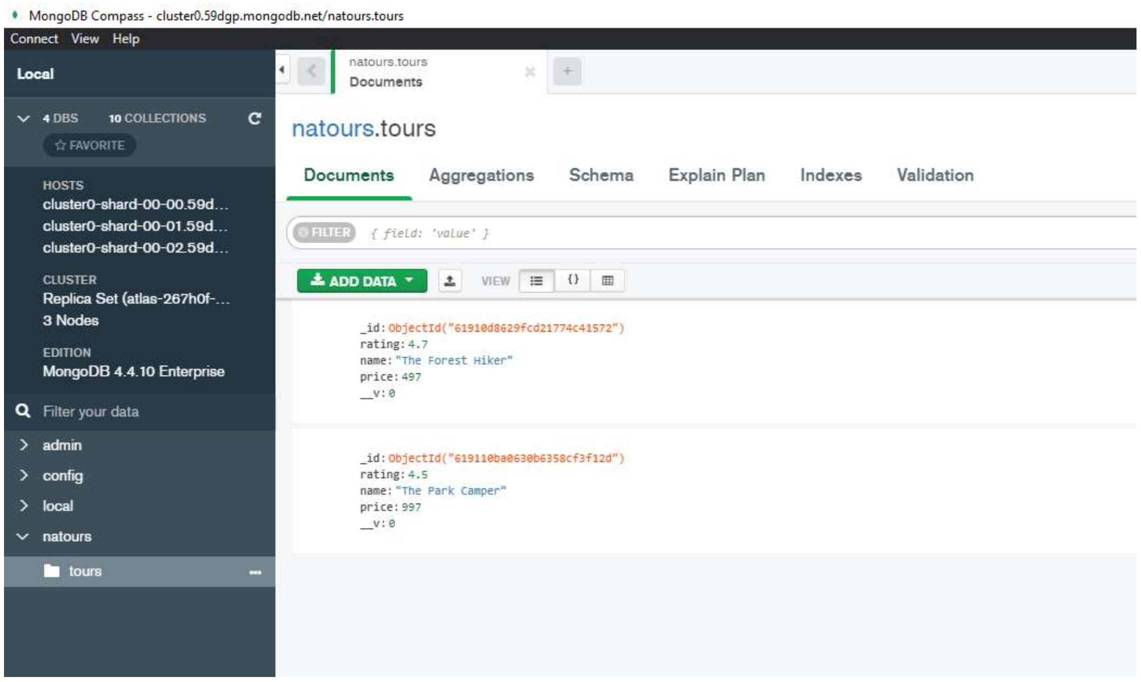
```
const mongoose = require("mongoose");
const dotenv = require("dotenv");
const app = require("./app");

dotenv.config({ path: "./config.env" });

const DB = process.env.DATABASE.replace(
  "<PASSWORD>",
  process.env.DATABASE_PASSWORD
);

mongoose
  .connect(DB, {
    useNewUrlParser: true,
    useCreateIndex: true,
    useFindAndModify: false,
  })
  .then(() => console.log("Database connection successful!"));
```

```
//---------------------------------------
// ⭑ NOTES: Proper way of describing a Schema: w/ Schema Type
Options
//---------------------------------------

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

//---------------------------------------
// ⭑⭑ Creating a Model Using the Schema we just described
//---------------------------------------

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

//--------------------
// ⭑⭑⭑ Creating a new document out of the Tour model or Function
Constructor that we just created :
//--------------------

// This is just like using JavaScript function constructors or
JavaScript Classes, if you are using ES6 -- Basically to create new
Object using a class. The document that we just created is an
INSTANCE of the Tour Model. and thus it has some methods attached to
it which we can use to interact with the database.
```

The testTour document that we will create is just for testing purposes.

```
const testTour = new Tour({
  name: "The Forest Hiker",
  rating: 4.7,
  price: 497,
});

// const testTour = new Tour({
//   name: "The Park Camper",
//   price: 997,
// });

// ● This will save this newly created document, to the tours
collection, in the database. This save will return a Promise, that
we need to consume. We will use async/await later, in order to
consume these Promises. Then we will also handle the error if
something goes wrong.

// MongoDB also adds __v: 0, in our document object: Its not that
important at this point

// Thus we can see the newly created document named "The Forest
Hiker", in our Atlas database, as well as in the MongoDB Compass on
the desktop, that we also configured and connected to our cloud host
previously.

testTour
  .save()
  .then((doc) => {
    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}...`);
});
```



```
_id: ObjectId("61910d8629fcd21774c41572")
rating: 4.7
name: "The Forest Hiker"
price: 497
__v: 0

_id: ObjectId("619110ba0630b6358cf3f12d")
rating: 4.5
name: "The Park Camper"
price: 997
__v: 0
```

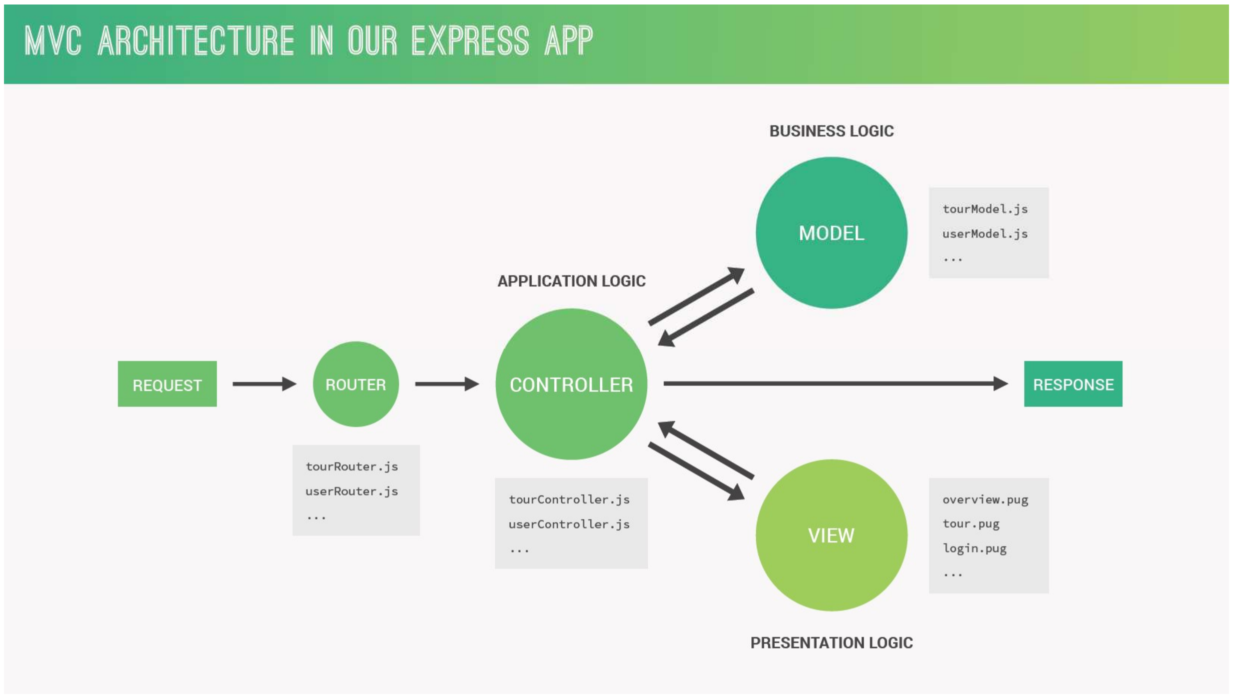
And keep in mind that we had actually deleted the tours collection here in one of the previous videos, right, but Mongoose automatically created this new collection here as soon as we created the first document using the tour model, and so this name here basically comes from that tour model, it simply gives us this plural name **"tours"** , just as we had before.

Perfect, we just made our express application really interact with a MongoDB database for the very first time, right from our code.

```
//-------------------------------------------------------------
//                          LECTURE 86                 ☆☆
//Intro to Backend Architecture: MVC, Types of Logic and More…
//-------------------------------------------------------------
```

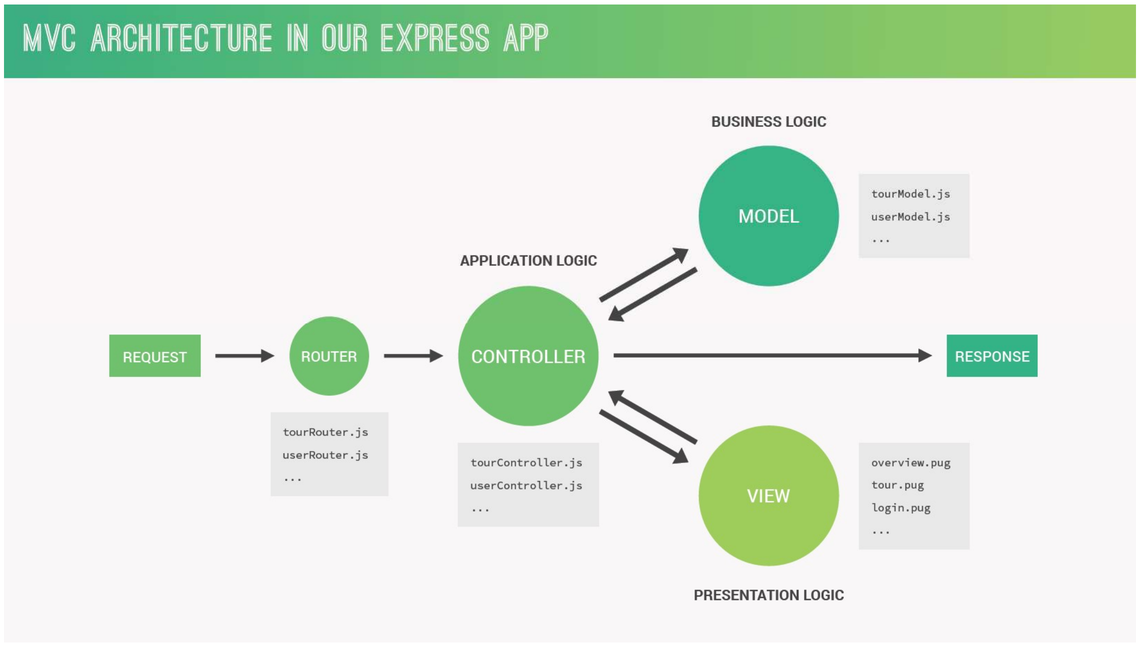
Up until this point, we have just written our code without thinking much about our application architecture. It wasn't really important until now, but now that our app is really starting to grow, we need to start worrying about the way that we design, or code architecture. And this lecture will just be a *brief introduction to back-end architecture*. **Starting with the MVC architecture.**

So, in this project, we're going to use a widely used and well known ***architecture called the model, view, controller or MVC for short.***



And there are different ways of implementing the MVC architecture, some more complex than others, but we're going to implement it in a very straightforward way here.

I just wanted to let you know that if you Google around for MVC, you'll find it implemented in some different ways.



Okay, anyway, in this architecture,

1. *the* ***model layer is concerned with everything about applications data, and the business logic.*** And we're going to learn what business logic means in the next slide.

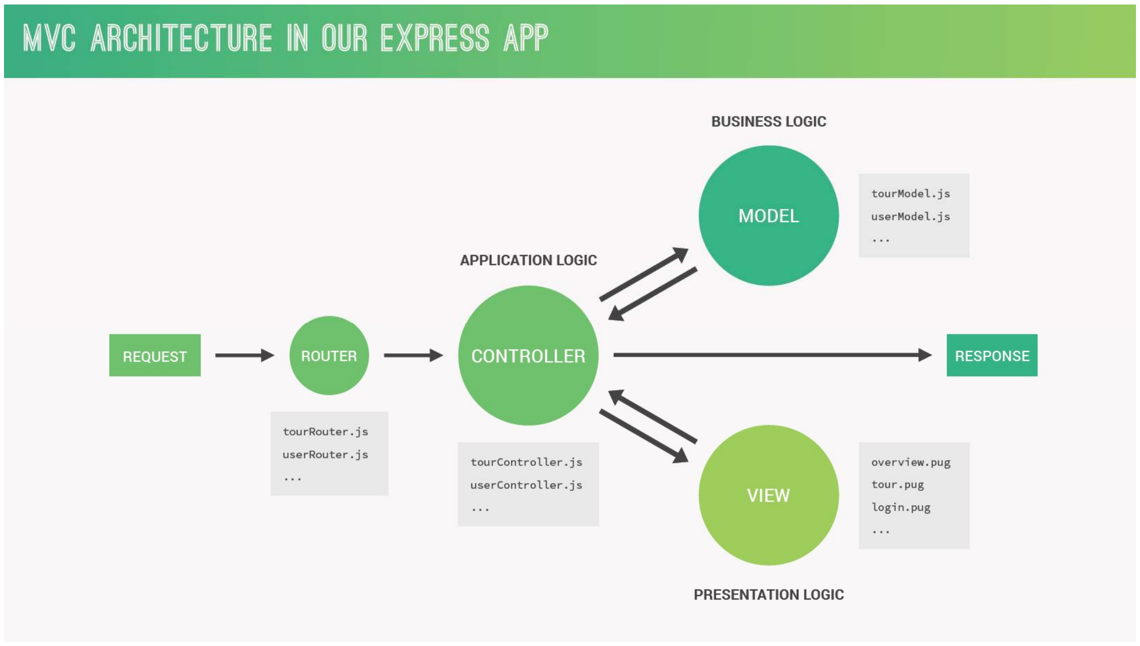
2. Next up, we have the *controller layer and the function of the controllers is to handle the application's request, interact with models, and send back responses to the client*. And all that is called the **application logic.**

3. Finally, the view layer is necessary if we have a graphical interface in our app. Or in other words, if we're building a server-side rendered website, as we talked about before. In this case, *the view layer consists basically of the templates used to generate the view, so the website that we're going to send back to the client*. And that is the **presentation logic.**

For now, we're just building an API though, so we're not really concerned about views just yet. That's for a bit later in the course.

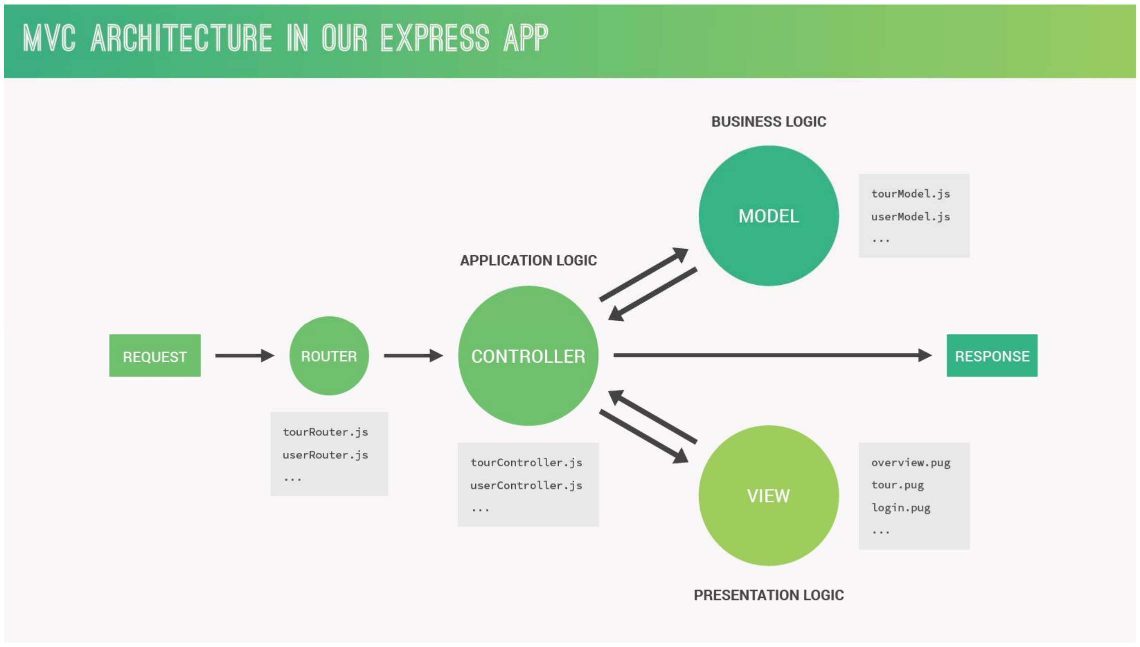
So using a pattern, or an architecture like this allows us to write a more modular application, which is going to be way easier to maintain in scale, as necessary. And we could take it even further, and add more layers of abstraction here. But in this kind of smaller application, the MVC architecture is more than enough for our needs.

Now, all this may sound a bit abstract, so **let's take a look at MVC in the context of our app, and the request-response cycle.**



So as always, it all starts with a request. That request will hit one of our routers, because remember, we have multiple routers. Basically, one for each resource, like tours, users, et cetera. Now the goal of the router is to delegate the request to the correct handler function, which will be in one of the controllers.

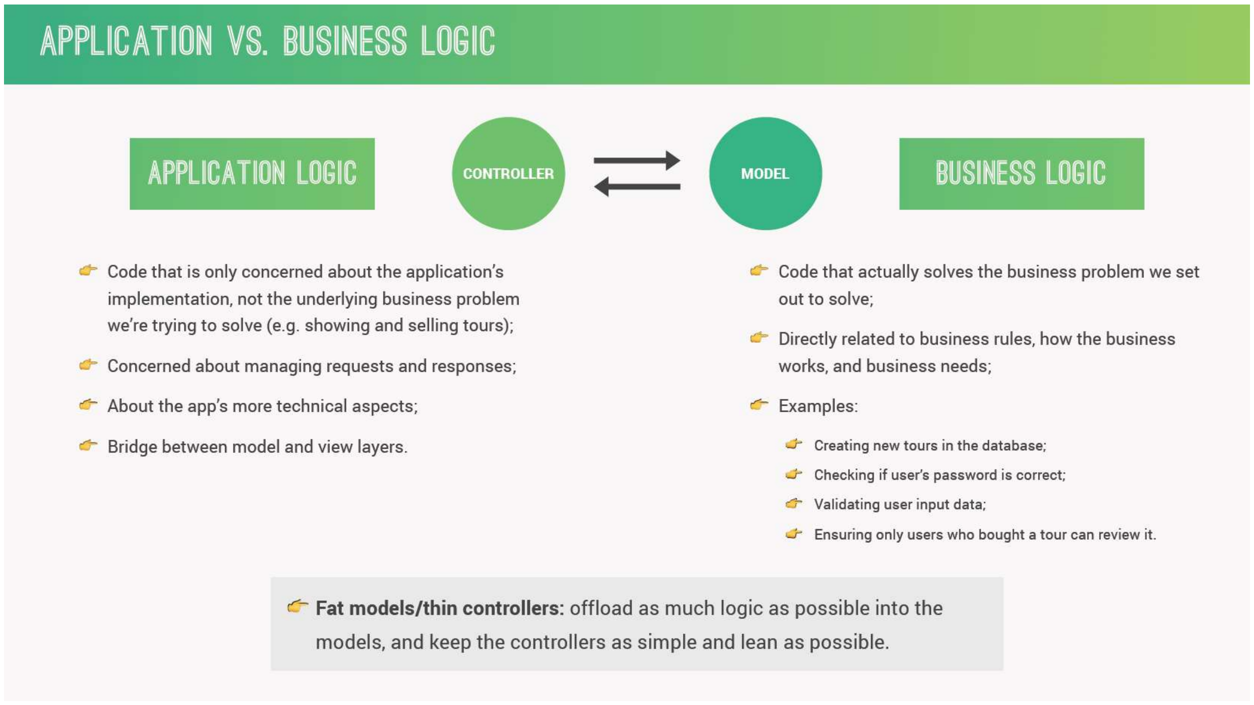
And again, there will be one controller for each of our resources, to keep these different parts of the app nicely separated. Then, depending on the incoming request, the controller might need to interact with one of the models, for example to retrieve a certain document from the database, or to create a new one. Once more, there is one model file for each resource. After getting the data from the model, the controller might then be ready to send back a response to the client, for example, containing that data. Now, in case we want to actually render a website, there is one more step involved. In this case, after getting the data from the model, the controller will then select one of the view templates and inject the data into it. That rendered website will then be sent back as the response.



<u>In the view layer in an Express app there is usually one view template for each page.</u>

Like a tour overview page, a tour detail page, or a login page. In the example of our natours app of course. So, that is a broad overview of the architecture that we're going to implement in this project.

Now to finish, let me just go into a ***bit more detail on model and controller.***



APPLICATION VS. BUSINESS LOGIC

APPLICATION LOGIC

- Code that is only concerned about the application's implementation, not the underlying business problem we're trying to solve (e.g. showing and selling tours);
- Concerned about managing requests and responses;
- About the app's more technical aspects;
- Bridge between model and view layers.

BUSINESS LOGIC

- Code that actually solves the business problem we set out to solve;
- Directly related to business rules, how the business works, and business needs;
- Examples:
  - Creating new tours in the database;
  - Checking if user's password is correct;
  - Validating user input data;
  - Ensuring only users who bought a tour can review it.

Fat models/thin controllers: offload as much logic as possible into the models, and keep the controllers as simple and lean as possible.

*So, **one of the big goals of MVC is to <u>separate Business logic from Application logic.</u>***

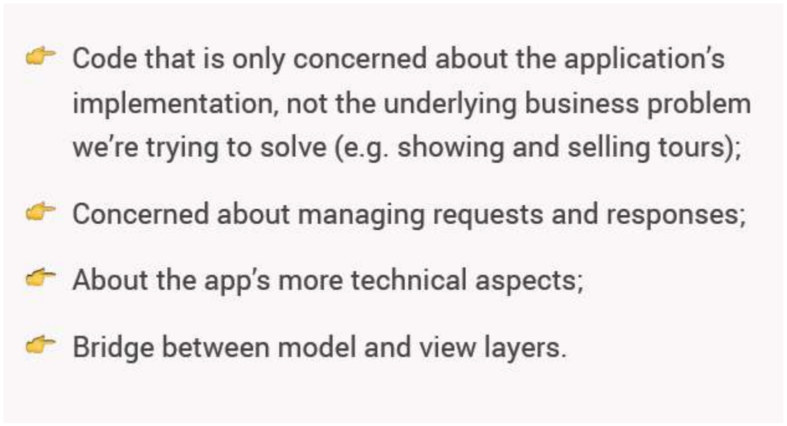
You'll hear about these types of logic all the time when you browse Stack Overflow, or some site like that. But what are these types of logic actually?

Well, the difference is a bit opinionated, but this is my take on it:

APPLICATION LOGIC

1. So, ***application logic*** is all the code that is only concerned about the application's implementation and not the underlying business problem that we're actually trying to solve with the application. Like showing and selling tours, managing stock in a supermarket, or organizing a library, for example. So again, ***application logic is the logic that makes the app actually work.*** For example, a big part of application logic in Express, is all about managing requests and responses. So, in a sense, we can also say that application logic is more about technical stuff.

Also, if we have views in our app, the application logic serves as a bridge between model and view layers So that we never mix business logic with presentation logic. All right?



👉 Code that is only concerned about the application's implementation, not the underlying business problem we're trying to solve (e.g. showing and selling tours);

👉 Concerned about managing requests and responses;

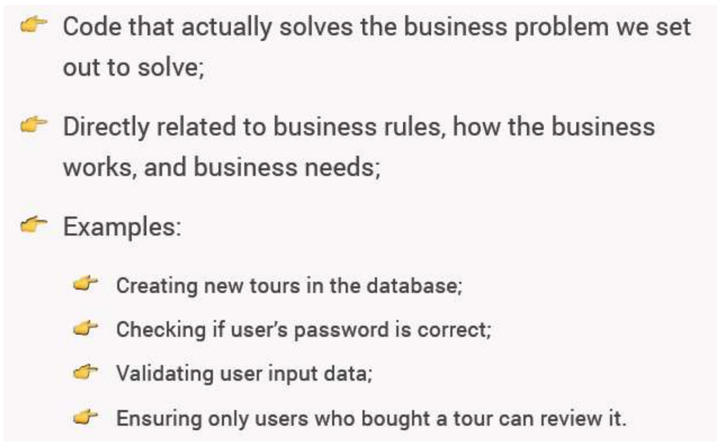
👉 About the app's more technical aspects;

👉 Bridge between model and view layers.

BUSINESS LOGIC

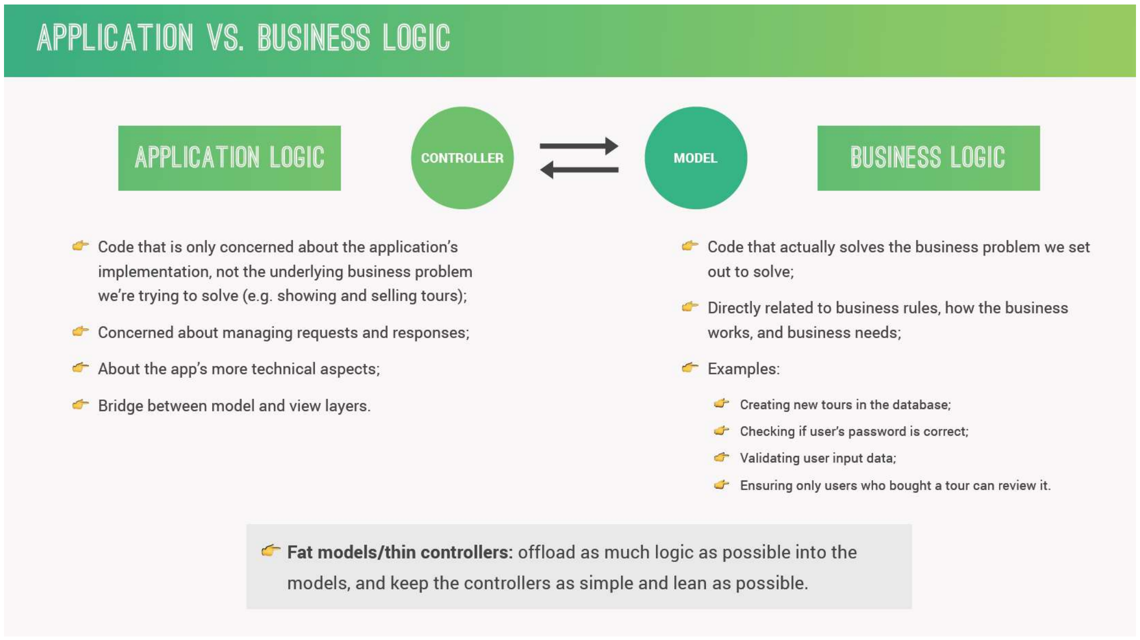
2. Now, about ***business logic***, it's all the code that actually solves the business problem that we set out to solve. Let's say again, that our goal is to show tours to customers and then sell them. And the code that is directly related to the business rules, to how the business works, and the business needs, is business logic.

Now if that still sounds a bit too philosophical, some examples in the context of our natours app are *creating new tours in the app's database, checking if a user's password is correct when he logs in, validating user input data, or ensuring that only users who bought a certain tour can review it*. So all this stuff is concerned with the business itself, and so it's part of the business logic.



- Code that actually solves the business problem we set out to solve;
- Directly related to business rules, how the business works, and business needs;
- Examples:
  - Creating new tours in the database;
  - Checking if user's password is correct;
  - Validating user input data;
  - Ensuring only users who bought a tour can review it.

Now, we need to keep in mind that application logic and business logic are almost impossible to completely separate, and so sometimes they will overlap. But we should do our best efforts to keep the application logic in our controllers and business logic in our models.



APPLICATION VS. BUSINESS LOGIC

CONTROLLER
MODEL

APPLICATION LOGIC

- Code that is only concerned about the application's implementation, not the underlying business problem we're trying to solve (e.g. showing and selling tours);
- Concerned about managing requests and responses;
- About the app's more technical aspects;
- Bridge between model and view layers.

BUSINESS LOGIC

- Code that actually solves the business problem we set out to solve;
- Directly related to business rules, how the business works, and business needs;
- Examples:
  - Creating new tours in the database;
  - Checking if user's password is correct;
  - Validating user input data;
  - Ensuring only users who bought a tour can review it.

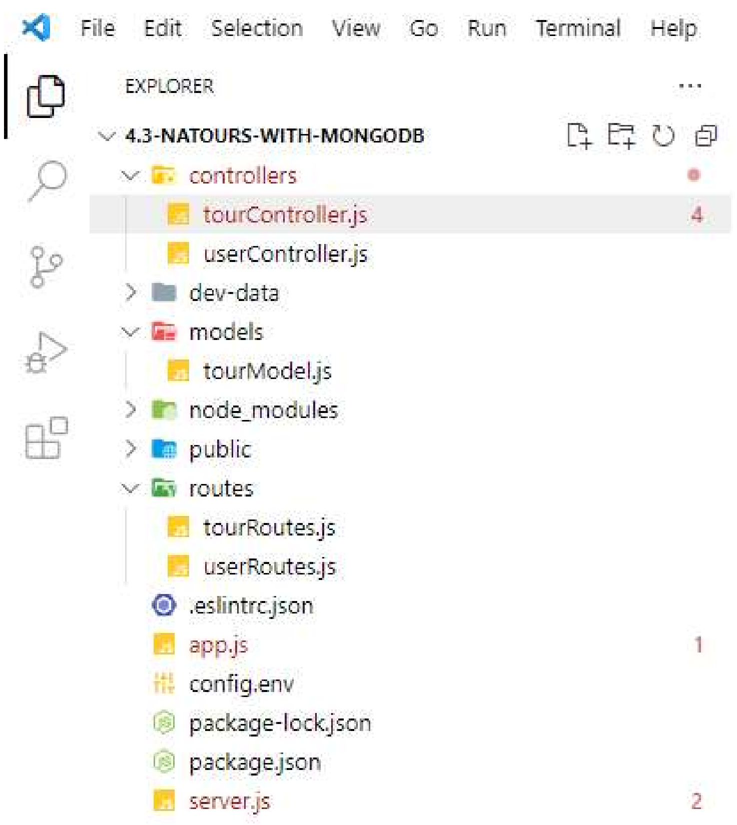
Fat models/thin controllers: offload as much logic as possible into the models, and keep the controllers as simple and lean as possible.

And there is even this philosophy of *fat models, thin controllers, which says we should offload as much logic as possible into the models, to keep the controllers as simple and lean as possible.*

So a **fat model** will have as much business logic as we can offload to it, and a thin controller will have as little logic as possible, so that the controller is really mostly for managing the application's requests and responses.

Okay? So, now keep all this in mind as we move on and progress into building our applications.

```
//------------------------------------------------------------
//                         LECTURE 87                ☆ ☆
//         Refactoring Our Model View Controller ( MVC )
//------------------------------------------------------------
```



```
4.3-NATOURS-WITH-MONGODB
  controllers
    tourController.js        4
    userController.js
  dev-data
  models
    tourModel.js
  node_modules
  public
  routes
    tourRoutes.js
    userRoutes.js
  .eslintrc.json
  app.js                     1
  config.env
  package-lock.json
  package.json
  server.js                  2
```

FILE: 📜

app.js

```
// Module IMPORTS

const express = require("express");
const morgan = require("morgan");

const tourRouter = require("./routes/tourRoutes");
const userRouter = require("./routes/userRoutes");

const app = express();

// 1) MIDDLEWARES

if (process.env.NODE_ENV === "development") {
  app.use(morgan("dev"));
}

app.use(express.json());
app.use(express.static(`${__dirname}/public`));

app.use((req, res, next) => {
  console.log("Hello from the middleware 👋"); // Using next()
  next();
});

app.use((req, res, next) => {
  req.requestTime = new Date().toISOString(); // Manipulating
Request, using next()
  next();
});

// 3) Our ROUTERS WITH Routes
// Middlewares: Mounting the Routers Properly ✔✔

app.use("/api/v1/tours", tourRouter);
app.use("/api/v1/users", userRouter);
```

```
module.exports = app;
```

FILE: server.js

```
const mongoose = require("mongoose");
const dotenv = require("dotenv");
const app = require("./app");

dotenv.config({ path: "./config.env" });

const DB = process.env.DATABASE.replace(
  "<PASSWORD>",
  process.env.DATABASE_PASSWORD
);

mongoose
  .connect(DB, {
    useNewUrlParser: true,
    useCreateIndex: true,
    useFindAndModify: false,
  })
  .then(() => console.log("Database connection successful!"));

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

FOLDER/FILE: 📁
models/tourModel.js

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

//---------------------------------------
//  Proper way of describing a Schema: w/ Schema Type Options

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

//---------------------------------------
//  Creating a Model Using the Schema we just described

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

//---------------------------------------
//  Exporting the model that we created to the tourController.js
file

module.exports = Tour;
```

FOLDER/FILE :

controllers/tourController.js

```
/*

//------------------------------------------
// ---- TESTING PURPOSES ------ PLEASE Find REFACTORED Code
further//
//------------------------------------------

const fs = require("fs");


// Reading the data: Sync : was USED FOR TESTING
const tours = JSON.parse(
  fs.readFileSync(`${__dirname}/../dev-data/data/tours-simple.json`)
);

// PARAM Middleware: Perform Checks before the request hits any of
these handler functions
exports.checkID = (req, res, next, val) => {
  console.log(`Tour id is: ${val}`);
  if (+req.params.id > tours.length) {
    return res.status(404).json({
      status: "fail",
      message: "Invalid ID",
    });
  }
  next();
};

// My Own Middleware: To Check if name and price is present on the
body
exports.checkBody = (req, res, next) => {
  if (!req.body.name || !req.body.price) {
    return res.status(400).json({
      status: "fail",
      message: "Missing name or price",
    });
  }
  next();
};

//----- TOURS Handler

exports.getAllTours = (req, res) => {
  console.log(req.requestTime); // Requesting Time here, then in
response object

  res.status(200).json({
    status: "success",
    requestedAt: req.requestTime,
    results: tours.length,
    data: {
      tours: tours,
    },
  });
};

exports.getTour = (req, res) => {
  console.log(req.params);

  const id = +req.params.id;
  const tour = tours.find((eachEl) => eachEl.id === id);

  res.status(200).json({
    status: "success",
    data: {
      tour: tour,
    },
  });
};
```

```
exports.createTour = (req, res) => {
  // console.log(req.body);

  const newId = tours[tours.length - 1].id + 1;
  const newTour = Object.assign({ id: newId }, req.body);

  tours.push(newTour);

  fs.writeFile(
    `${__dirname}/dev-data/data/tours-simple.json`,
    JSON.stringify(tours),
    (err) => {
      res.status(201).json({
        status: "success",
        data: {
          tour: newTour,
        },
      });
    }
  );
};

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,
  });
};
```

```
*/




//----------------------------------------
// -----> ●● REFACTORED: STARTS FROM HERE------

const Tour = require("../models/tourModel");

// My Own Middleware
exports.checkBody = (req, res, next) => {
  if (!req.body.name || !req.body.price) {
    return res.status(400).json({
      status: "fail",
      message: "Missing name or price",
    });
  }
  next();
};

//----- TOURS Handler

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

  res.status(200).json({
    status: "success",
    requestedAt: req.requestTime,
    // results: tours.length,
    // data: {
    //   tours: tours,
    // },
  });
};

exports.getTour = (req, res) => {
  console.log(req.params);

  const id = +req.params.id;

  // const tour = tours.find((eachEl) => eachEl.id === id);

  // res.status(200).json({
  //   status: "success",
  //   data: {
  //     tour: tour,
  //   },
  // });
};

exports.createTour = (req, res) => {
  res.status(201).json({
    status: "success",
    // data: {
    //   tour: newTour,
    // },
  });
};

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,
  });
};
```

FOLDER/FILE: 📁 routes/tourRoutes.js

```
//------------------------------------------
// ---- TESTING PURPOSES ------ PLEASE Find REFACTORED Code
further//
//------------------------------------------
/*

const express = require("express");
const tourController = require("../controllers/tourController");

const router = express.Router();

// // Named Imports Using Destructuring

// const {
//   getAllTours,
//   createTour,
//   getTour,
//   updateTour,
//   deleteTour,
// } = require("./../controllers/tourController");



router.param("id", tourController.checkID);

//-- TOURS Route

router
  .route("/")
  .get(tourController.getAllTours)
  .post(tourController.checkBody, tourController.createTour);
router
```

```
  .route("/:id")
  .get(tourController.getTour)
  .patch(tourController.updateTour)
  .delete(tourController.deleteTour);

module.exports = router;
*/
//---------------------------------------
// ----->> ●●  STARTS FROM HERE------

const express = require("express");
const tourController = require("../controllers/tourController");

const router = express.Router();

// router.param("id", tourController.checkID);

//-- TOURS Route

router
  .route("/")
  .get(tourController.getAllTours)
  .post(tourController.checkBody, tourController.createTour);
router
  .route("/:id")
  .get(tourController.getTour)
  .patch(tourController.updateTour)
  .delete(tourController.deleteTour);

module.exports = router;
```

```
//-------------------------------------------------------------
//                          LECTURE 88                 ⭐⭐
//   Another Easy Way of Creating Documents: Using Mongoose
//-------------------------------------------------------------
```

But in this lecture I'm going to show you an easier and even better way of doing so, as we implement our create tour handler. So at this point our API basically doesn't do anything anymore. It doesn't work anymore because basically we deleted all the functionality that we had in the last video.

And we did so, so that over the next couple of lectures we can rebuild it using a real database. Okay, so basically finally building our real API.

And we're gonna start by ***implementing the createTour function.*** But just remember the handler function that is called as soon as there is a post request to the tourRoutes.

We are also going to delete the checkBody handler function, which was basically to validate the body, so check if had “name” or “price” property in them.

So our Mongoose model is actually going to take care of that. So this here was nice to see how OUR OWN MIDDLEWARE works, and now we are going to get rid of that.

The Deleted Checkbody Function: Also, deleted from the tourRoutes file

```
// My Own Middleware
exports.checkBody = (req, res, next) => {
  if (!req.body.name || !req.body.price) {
    return res.status(400).json({
      status: "fail",
      message: "Missing name or price",
    });
  }
  next();
};
```

```
.post(tourController.checkBody, tourController.createTour);
```

→ Modified to
`.post(tourController.createTour);`
We are going to create a function to POST Tours using POSTMAN, which is yet another way of creating documents

`.post(tourController.`**createTour**`);` So, this handler function we are going to modify.

We are going to create a new tour based on the data, that comes in the "body", in this case, the body that we specify from the POSTMAN App to Create New Tours

We are also going to create our new tour in an easier way, directly by using Tour.create().

The main difference is that, in this version here, we basically call the method directly on the Tour, while in the 1st version, we called a method on the new document. That is completely different.

So we had the *newTour that we created from the model, and on that newTour, we used the .save method because the document has access to a lot of methods.*

But in the 2nd Version, we called .create method, right on the model itself.

Now, Remember, that the .save method returned a Promise. In the same way, the .create Method also returns a Promise. Here, in the **2nd Version** we are going to use **AYNC/AWAIT method consume the Promise**, and also and hence, to get access to the document.

We will also **HANDLE ERRORS** putting our code into the try/catch block in case we have an error. We will be sending an error message, in this case, if there is a VALIDATION ERROR, like if the "required" field is not there in the POST body. And then the Promise that we created here is rejected and enter the "catch" block.

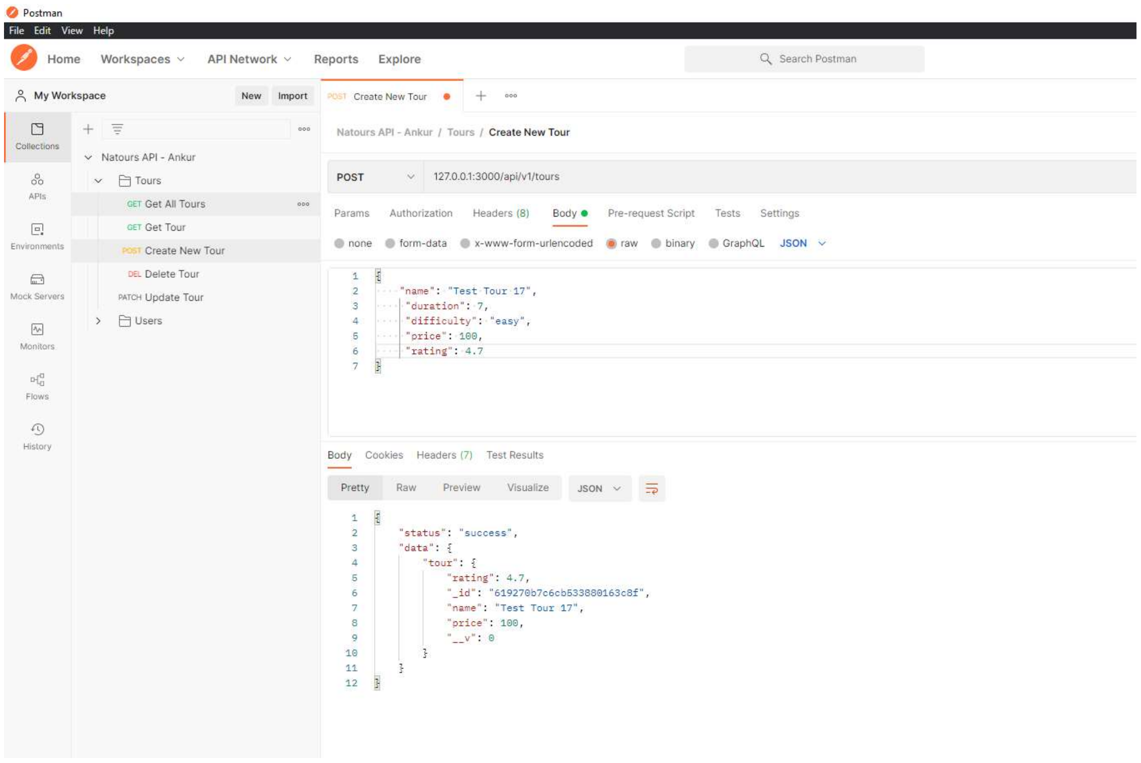
The final piece of the puzzle is to pass some real data that we wantto store in the database, into this, Tour.create({}) method.
That comes from the POST body, and stored in req.body
And That indeed is what we pass into the Tour.create({}) as
Tour.create({req.body})

tourController.js

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

//----- TOURS Handler

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

  res.status(200).json({
    status: "success",
    requestedAt: req.requestTime,
    // results: tours.length,
    // data: {
    //   tours: tours,
    // },
  });
};

exports.getTour = (req, res) => {
  console.log(req.params);

  const id = +req.params.id;

  // const tour = tours.find((eachEl) => eachEl.id === id);

  // res.status(200).json({
  //   status: "success",
  //   data: {
```

```
  //      tour: tour,
  //    },
  // });
};

exports.createTour = async (req, res) => {
  try {
    // const newTour = new Tour({});
    // newTour.save()

    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.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,
  });
};
```

CHECKING the POST Request Tour in Postman App



POST 127.0.0.1:3000/api/v1/tours

```
{
    "name": "Test Tour 17",
     "duration": 7,
     "difficulty": "easy",
     "price": 100,
     "rating": 4.7
}
```

Response Body:

```
{
    "status": "success",
    "data": {
        "tour": {
            "rating": 4.7,
            "_id": "619270b7c6cb533880163c8f",
            "name": "Test Tour 17",
            "price": 100,
            "__v": 0
        }
    }
}
```

And Indeed, we get Created a new document, in our database, added to the tours collection and there is an ID created automatically.

NOTE: We also notice that, there is ✗ **difficulty** and ✗ **duration** In our newly created object, in the database. So its because, its not present in OUR SCHEMA, and hence, its ignored.

So, *basically, everything, thet is NOT PRESENT in OUR SCHEMA is simply **IGNORED**. SEE BELOW*

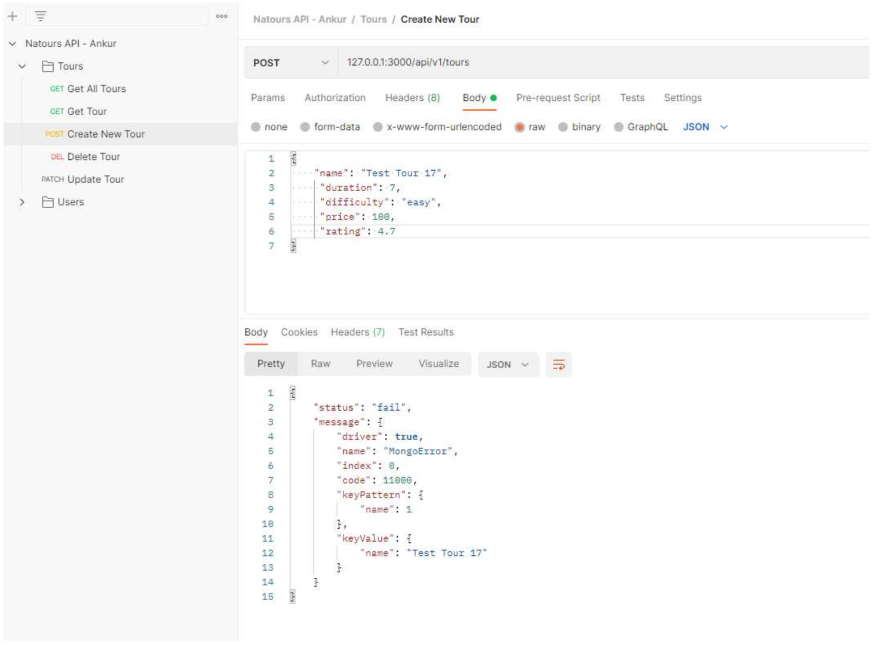
tourModel.js

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

//---------------------------------------
//  Proper way of describing a Schema: w/ Schema Type Options

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

//---------------------------------------
//  Creating a Model Using the Schema we just described

const Tour = mongoose.model("Tour", tourSchema);
```

```
//----------------------------------------
//  Exporting the model that we created to the tourController.js
file

module.exports = Tour;
```

<u>ENCOUNTERING Errors</u>

1. SENDING THE Same POST Request Again, and Analysing the Duplicate data ERROR



Natours API - Ankur / Tours / Create New Tour

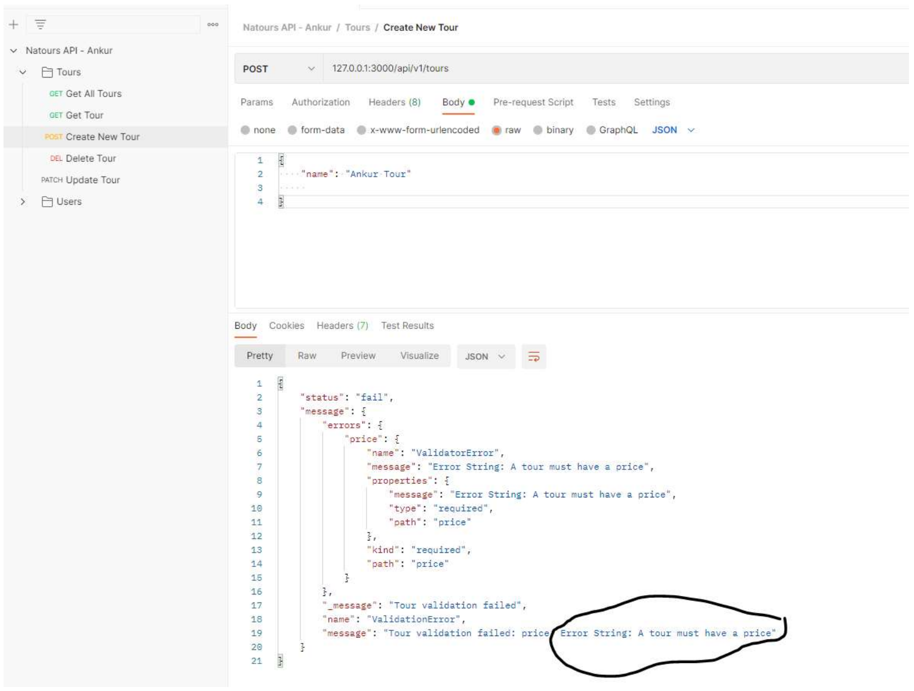
POST 127.0.0.1:3000/api/v1/tours

```
{
    "name": "Test Tour 17",
    "duration": 7,
    "difficulty": "easy",
    "price": 100,
    "rating": 4.7
}
```

Response Body:

```
{
    "status": "fail",
    "message": {
        "driver": true,
        "name": "MongoError",
        "index": 0,
        "code": 11000,
        "keyPattern": {
            "name": 1
        },
        "keyValue": {
            "name": "Test Tour 17"
        }
    }
}
```

2. SENDING Another POST Request Again with an invalid data , and Analysing the **Invalid** data ERROR


Natours API - Ankur
Tours
GET Get All Tours
GET Get Tour
POST Create New Tour
DEL Delete Tour
PATCH Update Tour
Users
Natours API - Ankur / Tours / Create New Tour
Params Authorization Headers (8) Body Pre-request Script Tests Settings
none form-data x-www-form-urlencoded raw binary GraphQL JSON
Body Cookies Headers (7) Test Results
Pretty Raw Preview Visualize JSON

POST 127.0.0.1:3000/api/v1/tours

```
{
    "name": "Ankur Tour"

}
```

```
{
    "status": "fail",
    "message": {
        "errors": {
            "price": {
                "name": "ValidatorError",
                "message": "Error String: A tour must have a price",
                "properties": {
                    "message": "Error String: A tour must have a price",
                    "type": "required",
                    "path": "price"
                },
                "kind": "required",
                "path": "price"
            }
        },
        "_message": "Tour validation failed",
        "name": "ValidationError",
        "message": "Tour validation failed: price: Error String: A tour must have a price"
    }
}
```

WE need to HANDLE all the ERRORS Properly later. There is a section ahead ENTIRELY FOR ERROR HANDLING, where we will create some meaningful errors.

For now we will just set the error to something like Message: "Invaid data sent".

● ● ● IMPORTANT

<u>✕ DON'T USE SUCH KIND OF MESSAGE IN A REAL PRODUCTION APPLICATION</u>

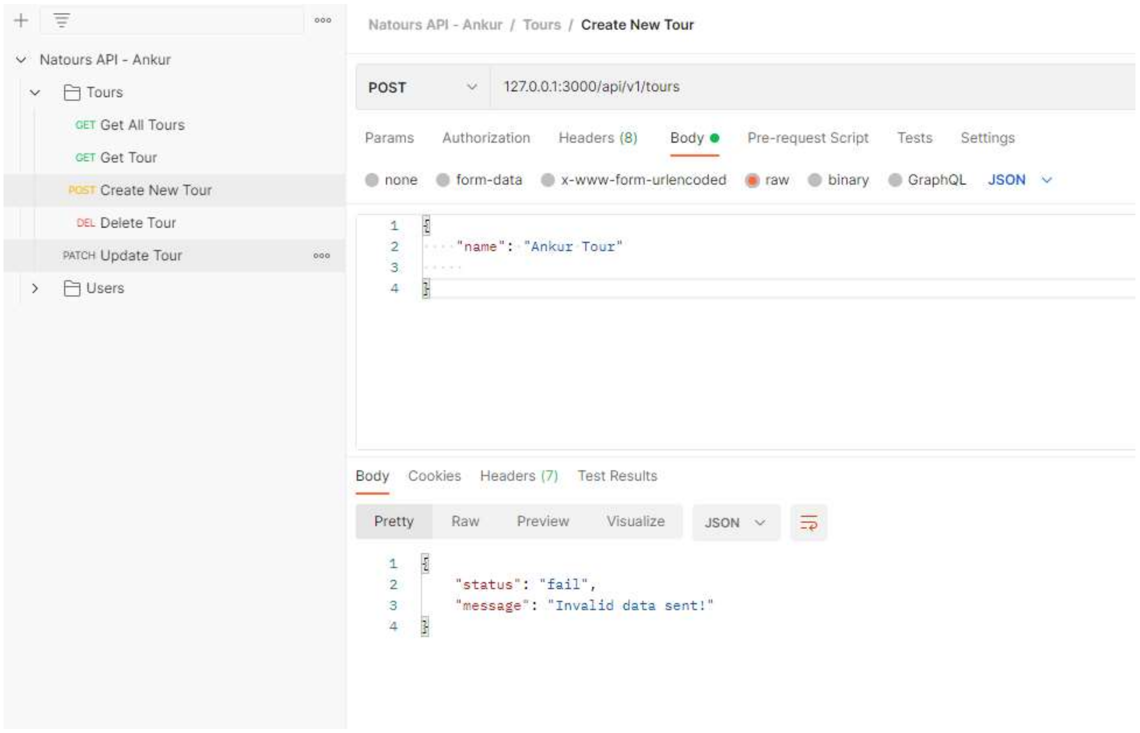
We will see the proper Error Handling later for the CLIENT. For Now, lets keep it like this for development purposes only.

----------------- FILES MODIFIED FOR THIS LECTURE-------------

tourController.js

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

//----- TOURS Handler

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

  res.status(200).json({
    status: "success",
    requestedAt: req.requestTime,
    // results: tours.length,
    // data: {
    //   tours: tours,
    // },
  });
};

exports.getTour = (req, res) => {
  console.log(req.params);

  const id = +req.params.id;

  // const tour = tours.find((eachEl) => eachEl.id === id);

  // res.status(200).json({
  //   status: "success",
  //   data: {
  //     tour: tour,
  //   },
  // });
};

exports.createTour = async (req, res) => {
  try {
    // const newTour = new Tour({});
    // newTour.save()

    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: "Invalid data sent!",
    });
  }
};

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,
  });
};
```

TESTING The Current Error Handling

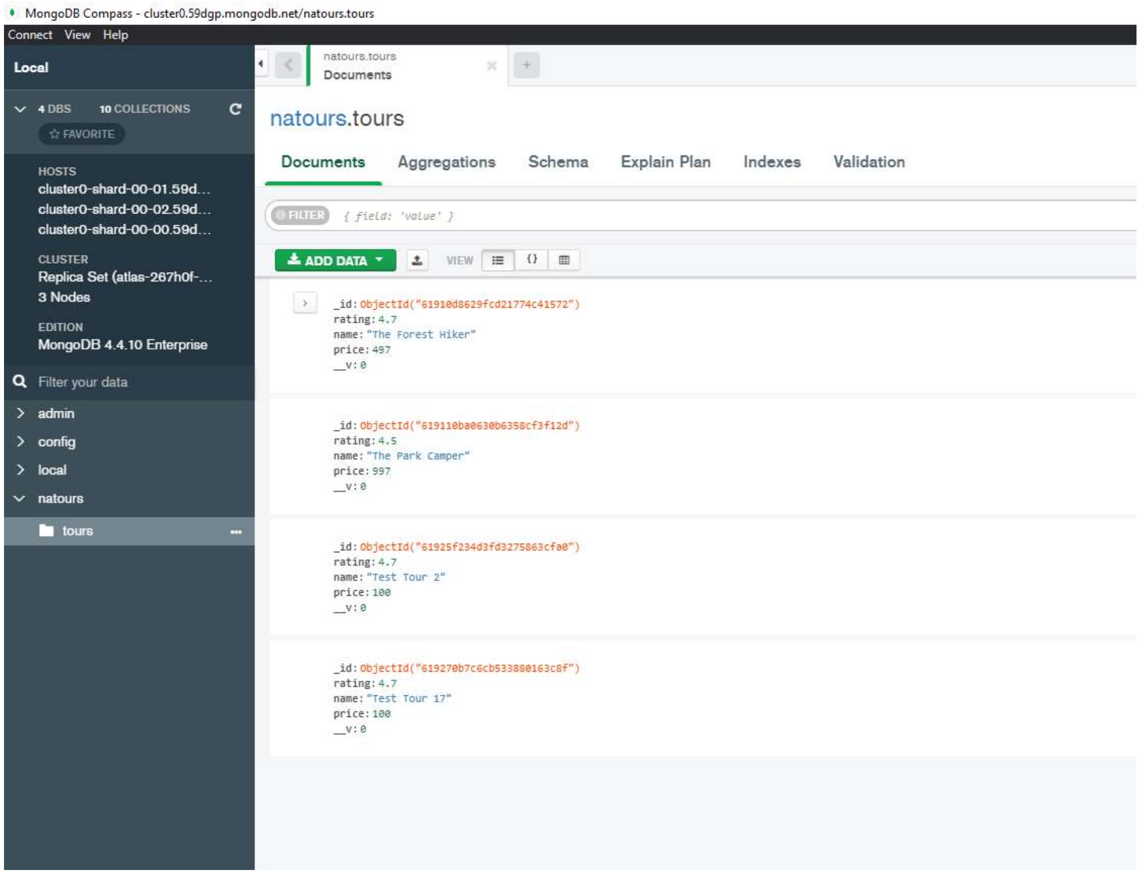

Natours API - Ankur
Tours
GET Get All Tours
GET Get Tour
POST Create New Tour
DEL Delete Tour
PATCH Update Tour
Users
Natours API - Ankur / Tours / Create New Tour
Params Authorization Headers (8) Body Pre-request Script Tests Settings
none form-data x-www-form-urlencoded raw binary GraphQL JSON
Body Cookies Headers (7) Test Results
Pretty Raw Preview Visualize JSON

POST 127.0.0.1:3000/api/v1/tours

```
{
    "name": "Ankur Tour"

}
```

```
{
    "status": "fail",
    "message": "Invalid data sent!"
}
```

TAKING A LOOK AT OUR DATABASE: ***MongoDB COMPASS***



```
_id: ObjectId("61910d8629fcd21774c41572")
rating: 4.7
name: "The Forest Hiker"
price: 497
__v: 0

_id: ObjectId("619110ba0630b6358cf3f12d")
rating: 4.5
name: "The Park Camper"
price: 997
__v: 0

_id: ObjectId("61925f234d3fd3275863cfa0")
rating: 4.7
name: "Test Tour 2"
price: 100
__v: 0

_id: ObjectId("619270b7c6cb533880163c8f")
rating: 4.7
name: "Test Tour 17"
price: 100
__v: 0
```

//--

```
//                          LECTURE 89                ☆☆
//      Reading Documents
//---------------------------------------------------------------
```

Getting all the tours from the database: **tours.find()**

Getting a Single Tour: using **.findById()**

Getting a single tour by the database id

Catch block code for error handling in a single tour with id, while getting it

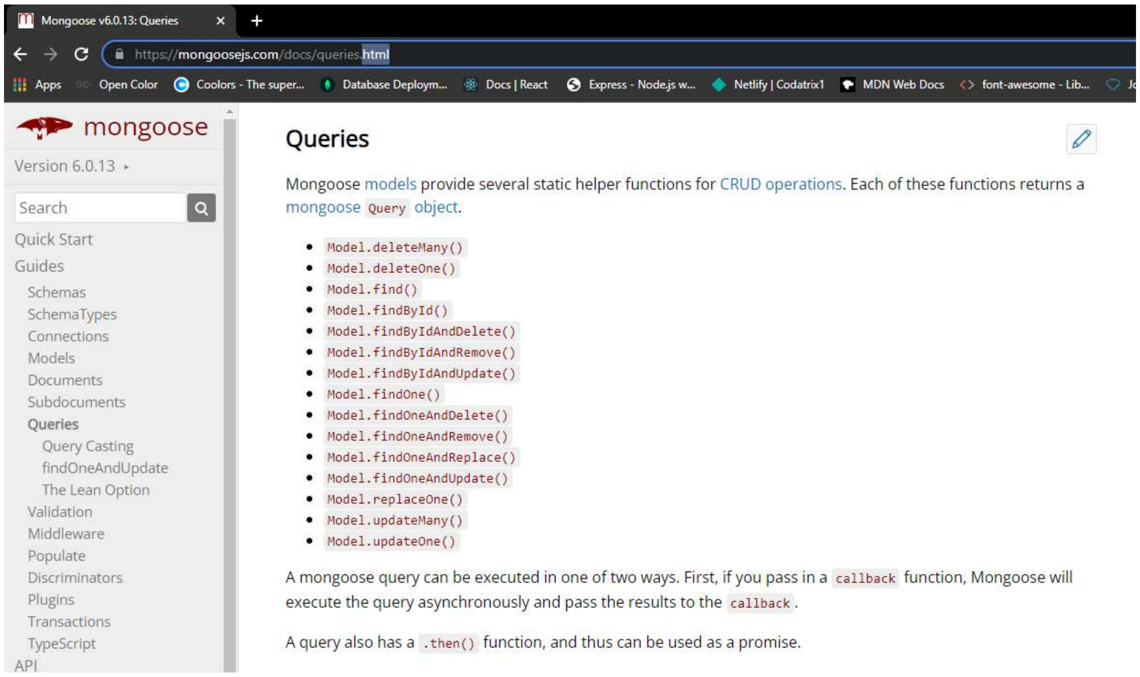
req.id.params : EXPLAINED

All The Functions, uses ASYNC/AWAIT from now on

tourController.js

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

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

    res.status(200).json({
      status: "success",
      requestedAt: req.requestTime,
      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);
    // Tour.findOne({ _id: req.params.id })

    res.status(200).json({
      status: "success",
      data: {
        tour: tour,
      },
    });
  } catch (err) {
    res.status(404).json({
      status: "fail",
      message: err,
    });
  }
};

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: "Invalid data sent!",
    });
  }
};

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,
  });
};
//-----------------------------------------------------------------
```

```
//                              LECTURE 90                    ☆ ☆
//          Updating Documents
//------------------------------------------------------------
```



Mongoose v6.0.13: Queries

https://mongoosejs.com/docs/queries.html

Apps · Open Color · Coolors - The super... · Database Deploym... · Docs | React · Express - Node.js w... · Netlify | Codatrix1 · MDN Web Docs · font-awesome - Lib...

mongoose

Version 6.0.13

Search

Quick Start
Guides
Schemas
SchemaTypes
Connections
Models
Documents
Subdocuments
Queries
Query Casting
findOneAndUpdate
The Lean Option
Validation
Middleware
Populate
Discriminators
Plugins
Transactions
TypeScript
API

Queries

Mongoose models provide several static helper functions for CRUD operations. Each of these functions returns a mongoose `Query` object.

- `Model.deleteMany()`
- `Model.deleteOne()`
- `Model.find()`
- `Model.findById()`
- `Model.findByIdAndDelete()`
- `Model.findByIdAndRemove()`
- `Model.findByIdAndUpdate()`
- `Model.findOne()`
- `Model.findOneAndDelete()`
- `Model.findOneAndRemove()`
- `Model.findOneAndReplace()`
- `Model.findOneAndUpdate()`
- `Model.replaceOne()`
- `Model.updateMany()`
- `Model.updateOne()`

A mongoose query can be executed in one of two ways. First, if you pass in a `callback` function, Mongoose will execute the query asynchronously and pass the results to the `callback`.

A query also has a `.then()` function, and thus can be used as a promise.

- `new`: bool - true to return the modified document rather than the original. defaults to false
- `runValidators`: if true, runs update validators on this command. Update validators validate the update operation against the model's schema.

1. **Updating Documents using async await**
2. **Updating a document by ID: findByIdAndUpdate**
3. **new: true --> We need to return the updated document to the client**
4. **Visit Mongoose Documentation to see all Query Methods**

https://mongoosejs.com/docs/queries.html

tourController.js

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

//----- TOURS Handler

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

    res.status(200).json({
      status: "success",
      requestedAt: req.requestTime,
      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);
    // Tour.findOne({ _id: req.params.id })

    res.status(200).json({
      status: "success",
      data: {
        tour: tour,
      },
    });
  } catch (err) {
    res.status(404).json({
      status: "fail",
      message: err,
    });
  }
};

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: "Invalid data sent!",
    });
  }
};

exports.updateTour = async (req, res) => {
  try {
    const tour = await Tour.findByIdAndUpdate(req.params.id,
req.body, {
      new: true,
      runValidators: true,
    });

    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,
  });
};
```

```
//------------------------------------------------------------------
//                            LECTURE 91                    ☆☆
//      Deleting Documents
//-----------------------------------------------------------------
```

THIS BELOW CODE CONTAINS ALL THE CRUD OPERATIONS

Deleting a document from the tour collection:
Tour.findIdAndDelete()

In a RESTful API we do not send any data to the client, while deleting a document

tourController.js

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

//----- TOURS Handler

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

    res.status(200).json({
      status: "success",
      requestedAt: req.requestTime,
      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);
    // Tour.findOne({ _id: req.params.id })

    res.status(200).json({
      status: "success",
      data: {
        tour: tour,
      },
    });
  } catch (err) {
    res.status(404).json({
      status: "fail",
      message: err,
    });
  }
};

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: "Invalid data sent!",
    });
  }
};

exports.updateTour = async (req, res) => {
  try {
    const tour = await Tour.findByIdAndUpdate(req.params.id,
req.body, {
      new: true,
      runValidators: 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);

    res.status(204).json({
      status: "success",
      data: null,
    });
  } catch (err) {
    res.status(404).json({
      status: "fail",
      message: err,
    });
  }
};


//-----------------------------------------------------------------
```

```
//                          LECTURE 92                 ☆☆
//      Modelling the Tours
//-------------------------------------------------------------
```

So now that you know the basics of Mongoose by implementing all the four CRUD operations, let's now finally Model our tour data a bit better in order to make the tours more complete.

So at this point, our tour documents can only have a name, a rating, and a price. But of course, we need so much more data here, because the ***goal is to have a very data-rich API***.

For now, there is just the ***"required"*** *validator* Later we will add more validators to it

tourModel.js

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

//------------------------------------
//  Proper way of describing a Schema: w/ Schema Type Options

const tourSchema = new mongoose.Schema({
  name: {
    type: String,
    required: [true, "Error String: A tour must have a name"],
    unique: true,
    trim: true,
  },
  duration: {
    type: Number,
    required: [true, "Error String: A tour must have a duration"],
  },
  maxGroupSize: {
    type: Number,
    required: [true, "Error String: A tour must have a group size"],
  },
  difficulty: {
    type: String,
    required: [true, "Error String: A tour must have a difficulty
level"],
  },
  ratingsAverage: {
    type: Number,
    default: 4.5,
  },
  ratingsQuantity: {
    type: Number,
    default: 0,
  },
  price: {
    type: Number,
    required: [true, "Error String: A tour must have a price"],
  },
  priceDiscount: Number,
  summary: {
    type: String,
    trim: true,
    required: [true, "Error String: A tour must have a summary"],
  },
  description: {
    type: String,
    trim: true,
  },
  imageCover: {
    type: String,
    required: [true, "Error String: A tour must have a cover
image"],
  },
  images: [String],
  createdAt: {
    type: Date,
    default: Date.now(),
  },
  startDates: [Date],
});
```

```
//---------------------------------------
//  Creating a Model Using the Schema we just described

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

//---------------------------------------
//  Exporting the model that we created to the tourController.js
file

module.exports = Tour;
```

-------------------- ● ● ALL LECTURE NOTES

1. ***All the schema fields, with a required field is ON <u>THE OVERVIEW PAGE</u>***

required: [true, "Error String: A tour must have a name"]

2. **ratingAverage and ratingsQuantity: Explanation**

Let's talk about the ratings.

So right now we have a field called Rating, but in fact we want to have one field for the ***ratingsAverage***, and one field for the ***ratingsQuantity.*** So basically the amount of ratings that there are.

So later, there will be another resource called reviews, where users will be able to write reviews about tours and give ratings.

And that's gonna be a completely different resource, and a completely different model.

But still, we want to have basically a summary of these ratings and of these reviews here in the tours.

Alright, and the reasons for that you will learn a bit later once we start how and why we model data in ***NoSQL databases***, and ***specifically in MongoDB.***

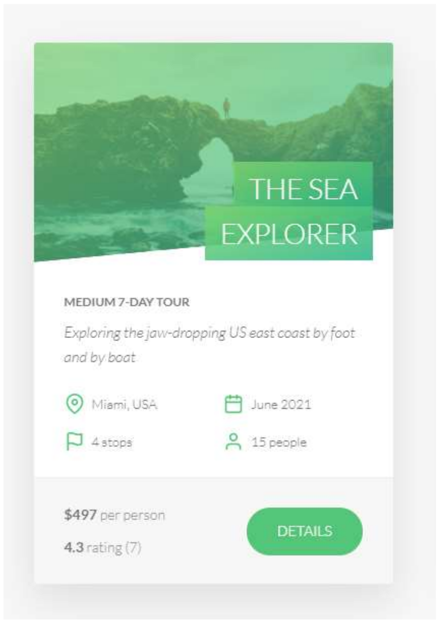
Okay, for now, let's just add the ***ratingsAverage***, and the ***ratingsQuantity.***

3. **trim: true** ---> Only Works for Strings: Removes all the whitespace from the beginning and from the end. Remember in JavaScript ??

4. **Please Check**

www.natours.dev

for all the Schema Expanations



5. **imageCover:**

So the type should be string, because basically this will simply be the name of the image here, which then later, we will be able to read from the file system.

Okay, so really just the name of the image. So basically a reference will be stored in the database. And that's a very common practice.

So we could store the entire image as well in a database, but that's usually not a good idea. ***We simply leave the images somewhere in the file system and then put the name of the image itself in the database as a field.***

So set it to required, true, a tour must have a cover image.

6. **For all the other images**

And now here, we have something new, okay? Because we have multiple images, and I want to save those images as an array. And actually, as an array of strings. And so the way to do it is to simplify specify here an array of strings, just like this.

So I want it still to be of type string, but I want it to be an array. So an array, in which we have a number of strings. And so that's how I can define this. We will use it for something else.

And so we will then actually pass in more schema type options than just a type, and so by then we will use that again, alright?

7. **Creating TimeStamps**

Gives Us the Time in Milliseconds. Its a JavaScript Built-In Function

In Mongoose, its automatically converted to current date, which makes more sense of the data

```
createdAt: {

    type: Date,

    default: Date.now(),

  },
```

8. ***So startDates, and again, we simply define an array, and then say that in there we want dates.***

Alright, so these startDates are basically different dates at which a tour starts.

For example, we can have a tour starting in December this year, and then in February, the next year, and then another one in the summer, and so different dates for the same tour are simply different, let's say, instances of the tour starting on different dates.

Anyway, this one here will not be automatically created by MongoDB, and MongoDB will then automatically try to parse the string that we passed in as the date into a real JavaScript date.

For example, we could pass in something like let's say 2021, March 21st, and again Mongo would then automatically parse this as a date.

---- ● ● CONCLUSION: VERY VERY IMPORTANT Script-----------

Now that our model is finally complete, at least for this section, we will go ahead and write a small script which will automatically import all of the tours from this JSON data.

Okay, *so from this JSON file, actually. So it will basically read the file, get all the tours, and import them into the database.*

Alright, so that's what we're gonna do next, because that will be hugely useful in this section and also in the future.

```
//--------------------------------------------------------------
//                        LECTURE 93              ⭐⭐
//       IMPORTING DEVELOPMENT DATA - from Dev-data Folder
//--------------------------------------------------------------
```

From this JSON file, actually. So it will basically read the file, get all the tours, and import them into the database.

● ● LECTURE NOTES:

Let's now take a small break from building our API and build a fun little script that will import the tour data from our JSON file into the MongoDB database.

And this script is completely independent of the rest of our express application. And so we'll run this completely separately from the comment line just to import everything once. It will run once, only in the beginning.

Usage of : **console.log (process.argv);** Specifying Options: For importing/deleting data through the terminal

The process. argv property is an inbuilt application programming interface of the process module which **is used to get the arguments passed to the node. js process when run in the command line.**

Commands to run from the terminal in order to delete the existing data from the database, i.e. in this case, from the ***tours collection***

1. node dev-data\import-dev-data.js –delete

2. node dev-data\import-dev-data.js --import

```
PS C:\Users\Ankur\Desktop\03-Node-Jonas\Node.js\4.4-natours-with-
MongoDB> node dev-data\import-dev-data.js --import
[
  'C:\\Program Files\\nodejs\\node.exe',
  'C:\\Users\\Ankur\\Desktop\\03-Node-Jonas\\Node.js\\4.4-natours-
with-MongoDB\\dev-data\\import-dev-data.js',
  '--import'
]
(node:6016) [MONGODB DRIVER] Warning: Current Server Discovery and
Monitoring engine is deprecated, and will be removed in a future
version. To use the new Server Discover and Monitoring engine, pass
option { useUnifiedTopology: true } to the MongoClient constructor.
(Use `node --trace-warnings ...` to show where the warning was
created)
Database connection successful!
```

process.exit(); Exits the program, when the operation is complete. Its an agressive way to exit out. But here, we have a small amount of data. So its fine in this case. Please try to find alternatives if possible, while working on a real application.

import-dev-data.js

```
const fs = require("fs");
const mongoose = require("mongoose");
const dotenv = require("dotenv");
const Tour = require("../models/tourModel");

dotenv.config({ path: "./config.env" });

const DB = process.env.DATABASE.replace(
  "<PASSWORD>",
  process.env.DATABASE_PASSWORD
);

mongoose
  .connect(DB, {
    useNewUrlParser: true,
    useCreateIndex: true,
    useFindAndModify: false,
  })
  .then(() => console.log("Database connection successful!"));

// READ JSON FILE

const tours = JSON.parse(
  fs.readFileSync(`${__dirname}/data/tours-simple.json`, "utf-
8")
);

// IMPORT DATA INTO DATABASE

const importData = async () => {
  try {
    await Tour.create(tours);
    console.log("Data successfully loaded!");
  } catch (err) {
    console.log(err);
  }
  process.exit();
};

// DELETE ALL DATA FROM DATABASE: From the tours Collection
```

```
const deleteData = async () => {
  try {
    await Tour.deleteMany();
    console.log("Data successfully deleted!");
  } catch (err) {
    console.log(err);
  }
  process.exit();
};

// console.log(process.argv);

if (process.argv[2] === "--import") {
  importData();
} else if (process.argv[2] === "--delete") {
  deleteData();
}
```

```
//------------------------------------------------------------
//                        LECTURE 94              ☆☆
//            Making the API Better - Filtering
//------------------------------------------------------------
```

Let's now start woking in the API. The features that we will implement in from now on will make the API Easier to use for whoever uses it. ☺

Features:

1. Allow the user to basically filter data using a ***query string****. It will be implemented in GET ALL TOURS*.

   For example

127.0.0.1:3000/api/v1/tours**?duration=5&difficulty=easy**

● ● **NOTE: Do RECTIFY THE ORDER OF variable declaration, in the server.js file according to this LECTURE.**

```
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.DATABASE_PASSWORD
);

mongoose
  .connect(DB, {
    useNewUrlParser: true,
    useCreateIndex: true,
    useFindAndModify: false,
  })
  .then(() => console.log("Database connection successful!"));

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

In Mongoose, there are actually two ways of writing database queries.

1. ***The first one*** *is to just use filter object just like we did in the MongoDB introduction section. Right? Remember that? We will use this method, as its structure looks like the object that we used earlier.* ***Now, the problem with this implementation, is that its actually way too simple.*** That's because, later on, we will have other query parameters. For example, sort, for sorting functionality, or page, for pagination. So what we will have to do is, to basically exclude these special field names from our query string before we actually do the filtering.

2. ***The second way*** *is to use some special Mongoose methods i.e. by chaining some special Mongoose method to basically build the query similar to this one that we have.*

There are tons of methods for the query string like lte, gte, for sorting. Please refer the documentation. They look really weird, but they are really helpful in many situations

IMPORTANT: *We can’t use await directly into our query [req.query]. Its because, its an async function, and will take indefinite time to get back with the result. As we want to chain methods, we need to do it differently. Thus, we need to save our query into a variable, and then chain methods onto it.* ☺

tourController.js : WITH EXPLANATION

```
//----- TOURS Handler

// This data is on the request, and then it is on a field
called query: It would give us an Object nicely formatted with
the data from the query string

exports.getAllTours = async (req, res) => {
  try {
    //-PART 1 ------- BUILD THE QUERY -----
    // Step 1. Create shallow copy of the request object, as
we dont want to maipulate our original object, which infact
creates a new object. Remember JavaScript Reference type
Objects ? // Using destructuring and using spread operator to
create a shallow copy

    const queryObj = { ...req.query };

    // Step 2. Create array of all the tours that we want to
exclude

    const excludedFields = ["page", "sort", "limit",
"fields"];

     // Step 3. Remove all of these fields from the query
object by looping over these fields.//  We are using forEach
as we dont want a new Array // We will implement the excluded
fields in the upcoming lectures, but here we don't want it
right now

    excludedFields.forEach((el) => delete queryObj[el]);

    console.log(req.query, queryObj);

    const query = Tour.find(queryObj);

    // const query = Tour.find({
```

```
    //   duration: 5,
    //   difficulty: "easy",
    // });

    // const query = Tour.find()
    //   .where("duration")
    //   .equals(5)
    //   .where("difficulty")
    //   .equals("easy");

    // -PART 2 --- EXECUTE/SAVE THE QUERY in-order chain more
methods later
    const tours = await query;

    // -PART 3 ------ SEND RESPONSE
    res.status(200).json({
      status: "success",
      requestedAt: req.requestTime,    ← ✸ MODIFIED by Ankur
      results: tours.length,
      data: {
        tours,
      },
    });
  } catch (err) {
    res.status(404).json({
      status: "fail",
      message: err,
    });
  }
};
```

tourController.js: REFACTORED PART: GET ALL TOURS

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

exports.getAllTours = async (req, res) => {
  try {
    // BUILD THE QUERY -----
    const queryObj = { ...req.query };
    const excludedFields = ["page", "sort", "limit",
"fields"];
    excludedFields.forEach((el) => delete queryObj[el]);

    const query = Tour.find(queryObj);

    // EXECUTE THE QUERY
    const tours = await query;

    // SEND RESPONSE
    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);
    // Tour.findOne({ _id: req.params.id })
    res.status(200).json({
      status: "success",
      data: {
        tour,
      },
    });
  } catch (err) {
    res.status(404).json({
      status: "fail",
      message: err,
    });
  }
};

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.updateTour = async (req, res) => {
  try {
    const tour = await Tour.findByIdAndUpdate(req.params.id,
req.body, {
      new: true,
      runValidators: 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);

    res.status(204).json({
      status: "success",
      data: null,
    });
  } catch (err) {
    res.status(404).json({
      status: "fail",
      message: err,
    });
  }
};
```

```
//------------------------------------------------------------
//                        LECTURE 95              ☆☆
//          Making the API Better – Advanced Filtering
//------------------------------------------------------------
```

NOTES and EXPLANATIONS

1. **So this is how we would manually write the filter object for the query that we just specified.**

```
{ difficulty: "easy", duration: { $gte: 5 } }
```

2. **The standard way of writing a query string including these operators in POSTMAN, by adding a third part [gte], to our query string. So the KEY, the OPERATOR, and the VALUE**

127.0.0.1:3000/api/v1/tours**?duration[gte]=5**&difficulty=easy

3. **The query object, looks almost identical to the query that we specified in POSTMAN**

when we   *console.log (req.query)*

{ difficulty: "easy", duration: { $gte: 5 } }   <--- **w/ POSTMAN**

{ difficulty: 'easy', duration: { gte: '5' } }   <---- **in the CONSOLE**

4. **So that's really the only thing that's missing here in front of this operator name.**

Okay, and so the solution for this is to basically replace all the operators like this with their correspondent MongoDB operators, so basically adding this dollar sign here, okay. So let's now implement that,

5. We will implement **REGEX** , in order to replace our query string's **gte** *with* **$gte**
6. **Advanced Filtering : First stringify the the query object, then filter it**

```
     let queryStr = JSON.stringify(queryObj);

    queryStr = queryStr.replace(/\b(gte|gt|lte|lt)\b/g,
(match) => `$${match}`);

    console.log(JSON.parse(queryStr));
```

g → stands for global. We need to use the replacement globally. If we don't use "**g**" in our REGEX, then it would replace ONLY THE FIRST OCCURRENCE. We don't want that.

OUTPUT in the console

```
Database connection successful!

Hello from the middleware 👋

{ difficulty: 'easy', duration: { gte: '5' } }

{ difficulty: 'easy', duration: { '$gte': '5' } }

GET /api/v1/tours?difficulty=easy&duration[gte]=5 404 46.771
ms - 376
```

7. **Now we can filter with all kinds of filters ☺**

Now in the real world, we would then have to write some documentation, basically in order to allow the user to know which kinds of operation they can do on our API, right?

So again, ideally we would completely document our API, specifying which requests can be made using which http methods, and then also what kind of filtering or sorting, or all these features, which of them are available and how they can use it, right?

Now, in our case we of course will not do that but again, don't forget that if you're really implementing an API that is gonna be used by someone else, then you must really do this kind of documentation.

FILE: **tourController.js**

```
exports.getAllTours = async (req, res) => {
  try {
    console.log(req.query);

    // BUILD THE QUERY -----
    // 1) Filtering

    const queryObj = { ...req.query };
    const excludedFields = ["page", "sort", "limit",
"fields"];
    excludedFields.forEach((el) => delete queryObj[el]);

    // 2) Advanced Filtering : First stringify the the query
object, then filter it: USING REGEX

    let queryStr = JSON.stringify(queryObj);
    queryStr = queryStr.replace(/\b(gte|gt|lte|lt)\b/g,
(match) => `$${match}`);

    console.log(JSON.parse(queryStr));

    // { difficulty: "easy", duration: { $gte: 5 }}
    // { difficulty: 'easy', duration: { gte: '5' } }
    // gte, gt, lte, lt

    const query = Tour.find(JSON.parse(queryStr));
    // EXECUTE THE QUERY
    const tours = await query;

    // SEND RESPONSE
    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);
    // Tour.findOne({ _id: req.params.id })

    res.status(200).json({
      status: "success",
      data: {
        tour,
      },
    });
  } catch (err) {
    res.status(404).json({
      status: "fail",
      message: err,
    });
  }
};
```

```
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.updateTour = async (req, res) => {
  try {
    const tour = await Tour.findByIdAndUpdate(req.params.id,
req.body, {
      new: true,
      runValidators: 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);

    res.status(204).json({
      status: "success",
      data: null,
    });
  } catch (err) {
    res.status(404).json({
      status: "fail",
      message: err,
    });
  }
};
```

```
//-------------------------------------------------------------
//                         LECTURE 96               ☆☆
//             Making the API Better - Sorting
//-------------------------------------------------------------
```

NOTES and EXPLANATIONS

Let's now implement sorting. We basically want to enable the users to sort the tours according to search fields for example "price"

IN POSTMAN ---> 127.0.0.1:3000/api/v1/tours**?sort=price**

Sorting in the Ascending Order --\> smaller to greater

127.0.0.1:3000/api/v1/tours**?sort=price**

Sorting in the Descending Order (w/ minus sign) --\> greater to smaller

127.0.0.1:3000/api/v1/tours**?sort=-price**

How would we sort, if two tours have the same price ?

Well, we have to add another sorting criterion which would decide the order. We can add as many criterion as we want to like below, using a comma

127.0.0.1:3000/api/v1/tours**?sort=-price,ratingsAverage**

Default Sorting

So in case that the user does not specify any sort field in the URL query string, we're still gonna add a sort to the query. So query.sort and we will then sort by the created add field, all right? ***And actually in a descending order***, so that the newest ones appear first. So

-createdAt (minus createdAt)

FILE: **tourController.js ← Modified Part Only**

```
exports.getAllTours = async (req, res) => {
  try {
    console.log(req.query);

    // ⭑ BUILD THE QUERY -------------
    // 1A) Filtering
    const queryObj = { ...req.query };
    const excludedFields = ["page", "sort", "limit", "fields"];
    excludedFields.forEach((el) => delete queryObj[el]);

    // 1B) Advanced Filtering
    let queryStr = JSON.stringify(queryObj);
    queryStr = queryStr.replace(/\b(gte|gt|lte|lt)\b/g, (match) =>
`$${match}`);
    console.log(JSON.parse(queryStr));

    let query = Tour.find(JSON.parse(queryStr));

    // 2) Sorting

    if (req.query.sort) {
      const sortBy = req.query.sort.split(",").join(" ");
      console.log(sortBy);

      query = query.sort(sortBy);
      // OUR GOAL: sort("price ratingsAverage")
    } else {
      query = query.sort("-createdAt _id");    ← ☺ BUG FIXED
    }

    // ⭑ EXECUTE THE QUERY-------------
    const tours = await query;

    // ⭑ SEND RESPONSE-----------
    res.status(200).json({
      status: "success",
      results: tours.length,
      data: {
        tours,
      },
    });
  } catch (err) {
    res.status(404).json({
      status: "fail",
      message: err,
    });
  }
};
```

FILE: **tourController.js** **🡐 Complete: REFACTORED**

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

exports.getAllTours = async (req, res) => {
  try {
    console.log(req.query);

    // ⭐ BUILD THE QUERY -------------
    // 1A) Filtering
    const queryObj = { ...req.query };
    const excludedFields = ["page", "sort", "limit", "fields"];
    excludedFields.forEach((el) => delete queryObj[el]);

    // 1B) Advanced Filtering: w/ REGEX
    let queryStr = JSON.stringify(queryObj);
    queryStr = queryStr.replace(/\b(gte|gt|lte|lt)\b/g, (match) =>
`$${match}`);

    let query = Tour.find(JSON.parse(queryStr));

    // 2) Sorting: w/ multiple criterion

    if (req.query.sort) {
      const sortBy = req.query.sort.split(",").join(" ");
      query = query.sort(sortBy);
    } else {
      query = query.sort("-createdAt _id");
    }

    // ⭐ EXECUTE THE QUERY-------------
    const tours = await query;

    // ⭐ SEND RESPONSE-----------
    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);
    // Tour.findOne({ _id: req.params.id })

    res.status(200).json({
      status: "success",
      data: {
        tour,
      },
    });
  } catch (err) {
    res.status(404).json({
      status: "fail",
      message: err,
    });
  }
};

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.updateTour = async (req, res) => {
  try {
    const tour = await Tour.findByIdAndUpdate(req.params.id,
req.body, {
      new: true,
      runValidators: 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);
    res.status(204).json({
      status: "success",
      data: null,
    });
  } catch (err) {
    res.status(404).json({
      status: "fail",
      message: err,
    });
  }
};
```

```
//------------------------------------------------------------
//                       LECTURE 97               ☆☆
//          Making the API Better – Limiting Fields
//------------------------------------------------------------
```

NOTES and EXPLANATIONS

As the next feature in our API, we have ***field limiting,*** so, ***basically, in order to allow clients to choose which fields they want to get back in the response.*** So, for a client, it's always ideal to receive as little data as possible, in order to reduce the bandwidth that is consumed with each request. And that's, of course, especially true when we have really data-heavy datasets, right?

And so it's a very nice feature to allow the API user to only request some of the fields. So, as the ***third feature, we will have field limiting.***

So let's say we only want the name, the duration, the difficulty, and the price, all right. And so, the implementation will actually be very similar to what we did before with sorting.

127.0.0.1:3000/api/v1/tours**?fields=name,duration,difficulty,price**

And here, for example, it expects a string like, name then space, duration and a price for example, okay. And so this way, it will only select these three field names and send back the result only containing that, okay? And actually, this operation of selecting

only certain field names is called **PROJECTING**

```
if (req.query.fields) {

      const fields = x;

      query = query.select("name duration difficulty price")

     }
```

Here for example, we will exclude this field. We will not send this field, which is indeed used by Mongoose internally. But we will not send this in our response to the client. You get the idea. Right? ☺ **We will put a minus sign in front of it in our code.**

"__v": 0

```
if (req.query.fields) {

      const fields = req.query.fields.split(",").join(" ");

      query = query.select(fields);

    } else {

      query = query.select("-__v");

    }
```

In this query, we will have everythig, except the name and duration ☺

127.0.0.1:3000/api/v1/tours**?fields=-name,-duration**

To check the default, which as the code suggests, will exclude "__v" ☺

127.0.0.1:3000/api/v1/tours

All right, great, now there's one last thing that I want to show you which is that **we can also exclude fields right from the schema**. **Alright, and *that can be very useful, for example, when we have sensitive data that should only be used internally. ← ● VERY VERY VERY IMPORTANT***

For example, stuff like ***passwords should never be exposed to the client*** and so therefore, we can exclude some fields right in the schema.

So for example, we might not want the user to see when exactly each tour was created. For example, tour might already be kind of old or something and so, let's say we want to always hide this **createdAt** field, alright, so we can go into our **schema**, which is in the tour model of course, and then at **createdAt**, we simply set the select property here to **false**. ***And so like this, we can basically, permanently hide this from the output.***

FILE: **tourModel.js** **← Modified: REFACTORED**

```
createdAt: {

    type: Date,

    default: Date.now(),

    select: false,

  },
```

FILE: **tourController.js** **🡐 Complete:**
REFACTORED

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

exports.getAllTours = async (req, res) => {
  try {
    console.log(req.query);

    // ⭐ BUILD THE QUERY -------------
    // 1A) Filtering
    const queryObj = { ...req.query };
    const excludedFields = ["sort", "page", "limit", "fields"];
    excludedFields.forEach((el) => delete queryObj[el]);

    // 1B) Advanced Filtering: Using REGEX
    let queryStr = JSON.stringify(queryObj);
    queryStr = queryStr.replace(/\b(gte|gt|lte|lt)\b/g, (match) =>
`$${match}`);

    let query = Tour.find(JSON.parse(queryStr));

    // 2) Sorting: w/ multiple criterion

    if (req.query.sort) {
      const sortBy = req.query.sort.split(",").join(" ");
      query = query.sort(sortBy);
    } else {
      query = query.sort("-createdAt _id");
    }




    // 3) Field limiting                  🡐 ADDED THIS IN THIS LECTURE
    if (req.query.fields) {
      const fields = req.query.fields.split(",").join(" ");
      query = query.select(fields);
    } else {
      query = query.select("-__v");
    }

    // ⭐ EXECUTE THE QUERY-------------
    const tours = await query;

    // ⭐ SEND RESPONSE-----------
    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);
    // Tour.findOne({ _id: req.params.id })

    res.status(200).json({
      status: "success",
      data: {
        tour,
      },
    });
  } catch (err) {
    res.status(404).json({
      status: "fail",
      message: err,
    });
  }
};

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.updateTour = async (req, res) => {
  try {
    const tour = await Tour.findByIdAndUpdate(req.params.id,
req.body, {
      new: true,
      runValidators: 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);

    res.status(204).json({
      status: "success",
      data: null,
    });
  } catch (err) {
    res.status(404).json({
      status: "fail",
      message: err,
    });
  }
};
```

```
//-----------------------------------------------------------------
```

```
//                          LECTURE 98               ☆☆
//             Making the API Better – Pagination
//------------------------------------------------------------
```

NOTES and EXPLANATIONS

Now, another extremely important feature of a good API is to provide pagination. So basically allowing the user to only select a certain page of our results, in case we have a lot of results.

So let's pretend that we have, **for example**, 1000 documents in a certain collection. And we say that on each page we have 100 documents. So that would mean that we'd have 10 pages, right? So 10 times 100 is 1000.

And so based on that, how are we gonna implement pagination, using our query string? Well, we will use the **page** and the **limit** fields

127.0.0.1:3000/api/v1/tours**?page=2&limit=10**

Above means

// **page=2&limit=10**, 1-10 page 1, **11-20 page 2**, 21-30 page 3

query = query.skip(10).limit(10);

NOTE:

For the User, we the skip value is kind of abstract. So we can't hard code it. The solution is that we need to come up with a fixed value for skip, but based on the page requested by the user. For example: If the User requests page 2, then we will skip 10 values, i.e. page 1 has 10 results, if the User requests for page 3, we need to skip 20 results and show only 10 results in page 3... You get the idea right ? ☺

Default Pagination: *We still want pagination even if the user does not define any page or limit*

So, When the page loads

Converting to a number, and setting the default page to 1

Remember JavaScript to convert string to a number? **We add plus sign at the beginning** ☺

```
const page = +req.query.page || 1;
```

Another method to convert string to number : we multiply the string w/ 1

```
const page =  req.query.page * 1 || 1;
```

// Pagination

```
    const page = +req.query.page || 1;

    const limit = +req.query.limit || 100;

    const skip = (page - 1) * limit;


    // page=3&limit=10, 1-10  page 1, 11-20 page 2, 21-30 page 3

    query = query.skip(skip).limit(limit);
```

Code Debugged

Pagination Issue resolved by adding 2 identifiers in sorting: **-createdAt _id**

// 2) Sorting: w/ multiple criterion

```
    if (req.query.sort) {
      const sortBy = req.query.sort.split(",").join(" ");
      query = query.sort(sortBy);
    } else {
      query = query.sort("-createdAt _id");
    }
```

So a *new method there is on the Tour model which is called countDocuments*. Okay? And as the name says this is going to return the number of documents, all right?

Actually it's gonna return a promise but we then await the promise and so it will then come back with the result of the amount of tours. Okay? And so if the number of documents that we skip is greater than or equal to the number of documents that actually exists well then that means that the page does not exist, alright?

That make sense? So if skip is greater than the number of tours, and for now what I'm gonna do here is to throw a new error, ane *that error will be caught by the try catch block in our code.*

However, we will implement much better error handing in the next section. For now, the error will be caught by the try catch block in our code. ⋆☺

FILE: **tourController.js** ← **Complete: REFACTORED**

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

exports.getAllTours = async (req, res) => {
  try {
    console.log(req.query);

    // ☆ BUILD THE QUERY -------------
    // 1A) Filtering
    const queryObj = { ...req.query };
    const excludedFields = ["sort", "page", "limit", "fields"];
    excludedFields.forEach((el) => delete queryObj[el]);

    // 1B) Advanced Filtering: Using REGEX
    let queryStr = JSON.stringify(queryObj);
    queryStr = queryStr.replace(/\b(gte|gt|lte|lt)\b/g, (match) =>
`$${match}`);

    let query = Tour.find(JSON.parse(queryStr));

    // 2) Sorting: w/ multiple criterion

    if (req.query.sort) {
      const sortBy = req.query.sort.split(",").join(" ");
      query = query.sort(sortBy);
    } else {
      query = query.sort("-createdAt _id"); ← MODDED THIS IN THIS
LECTURE
    }

    // 3) Field limiting
    if (req.query.fields) {
      const fields = req.query.fields.split(",").join(" ");
      query = query.select(fields);
    } else {
      query = query.select("-__v");
    }

    // 4) Pagination                        ← ADDED THIS IN THIS
LECTURE
    const page = +req.query.page || 1;    ← default value of page
    const limit = +req.query.limit || 100;
    const skip = (page - 1) * limit;   ← FORMULA for Pagination skip
value

    query = query.skip(skip).limit(limit);

    if (req.query.page) {
      const numTours = await Tour.countDocuments();
      if (skip >= numTours) throw new Error("This page does not
exists");
    }

    // ⋆ EXECUTE THE QUERY-------------
    const tours = await query;

    // Our query would now look like:
query.sort().select().skip().limit() and so on, which whould be in a
chained form, and would be awaited in order to get consumed later.

    // ⋆ SEND RESPONSE-----------
    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);
    // Tour.findOne({ _id: req.params.id })

    res.status(200).json({
      status: "success",
      data: {
        tour,
      },
    });
  } catch (err) {
    res.status(404).json({
      status: "fail",
      message: err,
    });
  }
};

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.updateTour = async (req, res) => {
  try {
    const tour = await Tour.findByIdAndUpdate(req.params.id,
req.body, {
      new: true,
      runValidators: 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);

    res.status(204).json({
      status: "success",
      data: null,
    });
  } catch (err) {
    res.status(404).json({
      status: "fail",
      message: err,
    });
  }
};
```

```
//------------------------------------------------------------
//                         LECTURE 99              ☆☆
//             Making the API Better - Aliasing
//------------------------------------------------------------
```

NOTES and EXPLANATIONS

1.-------------

Another nice small feature that we can add to an API is to provide an alias route to a request that might be very popular, so that might be requested all the time.

And, for example, we might want to provide a route specifically for the f**ive best cheap tours**.

2.**--------------**

So, the five best and cheapest tours. So, that means we have a limit of five and then we're gonna sort by ratings and by price. So, sort and then the average rating, and descending, so minus ratings average. And, in case they have the same average, then we want the cheapest price possible. And so, we're gonna sort also by price.

127.0.0.1:3000/api/v1/tours?**limit=5&sort=-ratingsAverage,price**

3.**--------------**

Now, let's say that this is a request that is done all the time and we want to provide a route that is simple and easy to memorize for the user.

So, let's go to our app and try to implement that.

4.--------------

We're gonna start at their routes, so tourRouter.js

And, ***we then need to create a new route*** .And so, how are we gonna call this route?

Well, let's say top-5-cheap. So, quite a simple name, but it's gonna work. And then, we actually only want to get requests to this route. And so, now it's time to think.

How are we gonna implement this functionality?

And so, ***the solution is gonna be to run a MIDDLEWARE before we actually run this getAllTours handler.*** And so, that middleware function is then gonna manipulate the query object that's coming in.

And so, this is yet another really nice example of using middleware because you really need to get familiar and used to this ***concept of using middleware strategically*** in order to change the request object as we need it.

5.--------------

```
router

  .route("/top-5-cheap")

  .get(tourController.aliasTopTours,
tourController.getAllTours);
```

6.--------------

We need to then, export the newly created middle functionality to the tourController. Don't forget to use **next( )**, as the third parameter. Remember from the Express Middleware lecture? ☺

7. --------------

// Aliasing

```
exports.aliasTopTours = (req, res, next) => {

  req.query.limit = "5";

  req.query.sort = "-ratingsAverage, price";

  req.query.fields = "name,price,ratingsAverage, summary,
difficulty";

  next();

};
```

8. -------------

Basically we are pre-filling the query string, for this exact route, so that the user gets to this page directly via a dedicated route that we just defined

```
exports.aliasTopTours = (req, res, next) => {

  req.query.limit = "5";

  req.query.sort = "-ratingsAverage, price";

  req.query.fields = "name,price,ratingsAverage, summary,
difficulty";

  next();

};
```

<u>CONCLUSION:</u>

Our new route implemented successfully for the user with Aliasing, using the concept of Express Middleware ☺✓✓

127.0.0.1:3000/api/v1/tours/top-5-cheap

So, that worked, that's fantastic. Now, of course, we could do a lot more alias routes here, but that's not really necessary. ***I really just wanted to show you the concept, show you that it can be a nice feature to add to any API and also I wanted to, again, show you the power of middlewares.***

So, we're talking about Mongoose here, but still, it's very important to keep getting familiar with the Express concepts that we talked about before.

Anyway, we're now done with our API features. Up next, we're just gonna quickly refactor all of this to make the entire code a bit more modular and, of course, better. ☺☺

tourRoutes.js

```
const express = require("express");
const tourController = require("../controllers/tourController");

const router = express.Router();

router
  .route("/top-5-cheap")
  .get(tourController.aliasTopTours, tourController.getAllTours);

router
  .route("/")
  .get(tourController.getAllTours)
  .post(tourController.createTour);
router
  .route("/:id")
  .get(tourController.getTour)
  .patch(tourController.updateTour)
  .delete(tourController.deleteTour);

module.exports = router;
```

tourController.js

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

// 5) Aliasing: Basically we are pre-filling the query string, for
this exact route, so that the user gets to this page directly via a
dedicated route that we just defined

exports.aliasTopTours = (req, res, next) => {
  req.query.limit = "5";
  req.query.sort = "-ratingsAverage, price";
  req.query.fields = "name, price, ratingsAverage, summary,
difficulty";
  next();
};

exports.getAllTours = async (req, res) => {
  try {
    console.log(req.query);

    // ⭑ BUILD THE QUERY -------------
    // 1A) Filtering
    const queryObj = { ...req.query };
    const excludedFields = ["sort", "page", "limit", "fields"];
    excludedFields.forEach((el) => delete queryObj[el]);

    // 1B) Advanced Filtering: Using REGEX
    let queryStr = JSON.stringify(queryObj);
    queryStr = queryStr.replace(/\b(gte|gt|lte|lt)\b/g, (match) =>
`$${match}`);

    let query = Tour.find(JSON.parse(queryStr));

    // 2) Sorting: w/ multiple criterion

    if (req.query.sort) {
      const sortBy = req.query.sort.split(",").join(" ");
      query = query.sort(sortBy);
    } else {
      query = query.sort("-createdAt _id");
    }

    // 3) Field limiting
    if (req.query.fields) {
      const fields = req.query.fields.split(",").join(" ");
      query = query.select(fields);
    } else {
      query = query.select("-__v");
    }

    // 4) Pagination
    const page = +req.query.page || 1;
    const limit = +req.query.limit || 100;
    const skip = (page - 1) * limit;

    query = query.skip(skip).limit(limit);

    if (req.query.page) {
      const numTours = await Tour.countDocuments();
      if (skip >= numTours) throw new Error("This page doest not
exist");
    }

    // ⭐ EXECUTE THE QUERY-------------
    const tours = await query;

    // Our query would now look like: query.sort().select().limit()
and so on, which whould be in a chained form, and would be waiated
in order to get consumed

    // ⭐ SEND RESPONSE-----------
    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);
    // Tour.findOne({ _id: req.params.id })

    res.status(200).json({
      status: "success",
      data: {
        tour,
      },
    });
  } catch (err) {
    res.status(404).json({
      status: "fail",
      message: err,
    });
  }
};

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.updateTour = async (req, res) => {
  try {
    const tour = await Tour.findByIdAndUpdate(req.params.id,
req.body, {
      new: true,
      runValidators: 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);

    res.status(204).json({
      status: "success",
      data: null,
    });
  } catch (err) {
    res.status(404).json({
      status: "fail",
      message: err,
    });
  }
};
```

```
//-------------------------------------------------------------
//                        LECTURE 100               ☆☆
//                 Refactoring API Features
//-------------------------------------------------------------
```

NOTES and EXPLANATIONS

PLEASE REFER THE VIDEO FOR BETTER UNDERSTANDING

Now, this is not only to make our code a bit cleaner, it's also to make it more modular and more reusable in the future. So, right now, we have all this code for the features that we built before in this getAllTours.

And, also imagine that we wanted to use these same features for another resource.

For example, for the users or, later, for the reviews. It would be not very practical to, basically, copy the code from here and use it, then, in the other resources, right.

And so, ***what I'm going to do is to, now, create a class in which I'm going to add one method for each of these API features or functionalities,*** as you might call them as well.

I'm actually going to ***export it to its own file****, basically, to* ***create a reusable module that we can, later on, import into other controllers.***

Then, we start with our constructor function and remember that this is the function that gets automatically called as soon as we create a new object out of this class, all right. Now, what do I actually want in these API features? Actually, I'm going to ***parse in two variables*** here, okay? *So the* ***mongoose query*** *and also the* ***queryString*** *that we get from express. So, basically, coming from the route,* all right. So that's what we usually have access to in the req.query, okay.

this : is simply the entire object

So, *in order to chain the various features, one after another*, we need to basically ***return the entire object***, after every method that we put in our class.

And, in fact, I actually wanted to talk to you about this here. Because, ***if you think about it, requesting the next page, which has 0 result, is not really an error.*** The fact that there are no results is enough for the user to realize that, basically, the page that was requested doesn't contain any data. So we do not really need an error in this situation.

And so, I'm just going to go ahead and delete all this code.

```
if (req.query.page) {

      const numTours = await Tour.countDocuments();

      if (skip >= numTours) throw new Error("This page doest not
exist");

    }
```

Give it a save and, again, remember that all of this chaining here only works because after calling each of these methods, we always *return* ***this***. And ***this*** *is the object itself which has access to each of these methods here, making it possible to chain them just like we have it here.*

tourController.js

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

// 5) Aliasing

exports.aliasTopTours = (req, res, next) => {
  req.query.limit = "5";
  req.query.sort = "-ratingsAverage, price";
  req.query.fields = "name, price, ratingsAverage, summary,
difficulty";
  next();
};

// query --> Tour.find()  <-- Mongoose Query
// queryString ---> req.query -> Query coming from Express, basically
coming from the route

class APIFeatures {
  constructor(query, queryString) {
    this.query = query;
    this.queryString = queryString;
  }

  filter() {
    // 1A) Filtering
    const queryObj = { ...this.queryString };
    const excludedFields = ["sort", "page", "limit", "fields"];
    excludedFields.forEach((el) => delete queryObj[el]);

    // 1B) Advanced Filtering: Using REGEX
    let queryStr = JSON.stringify(queryObj);
    queryStr = queryStr.replace(/\b(gte|gt|lte|lt)\b/g, (match) =>
`$${match}`);

    this.query = this.query.find(JSON.parse(queryStr));
    // let query = Tour.find(JSON.parse(queryStr));
    return this;
  }

  sort() {
    // 2) Sorting: w/ multiple criterion
    if (this.queryString.sort) {
      const sortBy = this.queryString.sort.split(",").join(" ");
      this.query = this.query.sort(sortBy);
    } else {
      this.query = this.query.sort("-createdAt _id");
    }

    return this;
  }

  limitFields() {
    // 3) Field limiting
    if (this.queryString.fields) {
      const fields = this.queryString.fields.split(",").join(" ");
      this.query = this.query.select(fields);
    } else {
      this.query = this.query.select("-__v");
    }

    return this;
  }

  paginate() {
    // 4) Pagination
    const page = +this.queryString.page || 1;
    const limit = +this.queryString.limit || 100;
    const skip = (page - 1) * limit;

    this.query = this.query.skip(skip).limit(limit);

    return this;
  }
```

```
}



exports.getAllTours = async (req, res) => {
  try {
    console.log(req.query);

    /*

    --------------------------------------------------
    ● THIS CODE HAS BEEN TOTALLY REFACTORED: SEE ABOVE
    ----------------------------------------------------
--------------------------------------------------------------------
-------

    // ⭑ BUILD THE QUERY -------------
    // 1A) Filtering
    const queryObj = { ...req.query };
    const excludedFields = ["sort", "page", "limit", "fields"];
    excludedFields.forEach((el) => delete queryObj[el]);

    // 1B) Advanced Filtering: Using REGEX
    let queryStr = JSON.stringify(queryObj);
    queryStr = queryStr.replace(/\b(gte|gt|lte|lt)\b/g, (match) =>
`$${match}`);

    let query = Tour.find(JSON.parse(queryStr));

    // 2) Sorting: w/ multiple criterion
    if (req.query.sort) {
      const sortBy = req.query.sort.split(",").join(" ");
      query = query.sort(sortBy);
    } else {
      query = query.sort("-createdAt _id");
    }

    // 3) Field limiting
    if (req.query.fields) {
      const fields = req.query.fields.split(",").join(" ");
      query = query.select(fields);
    } else {
      query = query.select("-__v");
    }

    // 4) Pagination
    const page = +req.query.page || 1;
    const limit = +req.query.limit || 100;
    const skip = (page - 1) * limit;

    query = query.skip(skip).limit(limit);

    if (req.query.page) {
      const numTours = await Tour.countDocuments();
      if (skip >= numTours) throw new Error("This page doest not
exist");
    }

---------------------------------------------------------------------
----------------------
    */

    // ⭑ EXECUTE THE QUERY-------------
    const features = new APIFeatures(Tour.find(), req.query)
      .filter()
      .sort()
      .limitFields()
      .paginate();

    const tours = await features.query;

    // Our query would now look like: query.sort().select().limit()
and so on, which whould be in a chained form, and would be waited in
order to get consumed

    // ⭑ SEND RESPONSE-----------
    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);
    // Tour.findOne({ _id: req.params.id })

    res.status(200).json({
      status: "success",
      data: {
        tour,
      },
    });
  } catch (err) {
    res.status(404).json({
      status: "fail",
      message: err,
    });
  }
};

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.updateTour = async (req, res) => {
  try {
    const tour = await Tour.findByIdAndUpdate(req.params.id,
req.body, {
      new: true,
      runValidators: 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);

    res.status(204).json({
      status: "success",
      data: null,
    });
  } catch (err) {
    res.status(404).json({
      status: "fail",
      message: err,
    });
  }
};
```

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

// 5) Aliasing

exports.aliasTopTours = (req, res, next) => {
  req.query.limit = "5";
  req.query.sort = "-ratingsAverage, price";
  req.query.fields = "name, price, ratingsAverage, summary,
difficulty";
  next();
};



class APIFeatures {
  constructor(query, queryString) {
    this.query = query;
    this.queryString = queryString;
  }

  filter() {
    // 1A) Filtering
    const queryObj = { ...this.queryString };
    const excludedFields = ["sort", "page", "limit", "fields"];
    excludedFields.forEach((el) => delete queryObj[el]);

    // 1B) Advanced Filtering: Using REGEX
    let queryStr = JSON.stringify(queryObj);
    queryStr = queryStr.replace(/\b(gte|gt|lte|lt)\b/g, (match) =>
`$${match}`);

    this.query = this.query.find(JSON.parse(queryStr));
    // let query = Tour.find(JSON.parse(queryStr));

    return this;
  }
  sort() {
    // 2) Sorting: w/ multiple criterion
    if (this.queryString.sort) {
      const sortBy = this.queryString.sort.split(",").join(" ");
      this.query = this.query.sort(sortBy);
    } else {
      this.query = this.query.sort("-createdAt _id");
    }

    return this;
  }

  limitFields() {
    // 3) Field limiting
    if (this.queryString.fields) {
      const fields = this.queryString.fields.split(",").join(" ");
      this.query = this.query.select(fields);
    } else {
      this.query = this.query.select("-__v");
    }

    return this;
  }

  paginate() {
    // 4) Pagination
    const page = +this.queryString.page || 1;
    const limit = +this.queryString.limit || 100;
    const skip = (page - 1) * limit;

    this.query = this.query.skip(skip).limit(limit);

    return this;
  }
}
```

```
exports.getAllTours = async (req, res) => {
  try {
    console.log(req.query);

    // ⭐ EXECUTE THE QUERY-------------
    const features = new APIFeatures(Tour.find(), req.query)
      .filter()
      .sort()
      .limitFields()
      .paginate();

    const tours = await features.query;

    // Our query would now look like: query.sort().select().limit()
and so on, which whould be in a chained form, and would be waited in
order to get consumed

    // ⭐ SEND RESPONSE-----------
    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);
    // Tour.findOne({ _id: req.params.id })
    res.status(200).json({
      status: "success",
      data: {
        tour,
      },
    });
  } catch (err) {
    res.status(404).json({
      status: "fail",
      message: err,
    });
  }
};

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.updateTour = async (req, res) => {
  try {
    const tour = await Tour.findByIdAndUpdate(req.params.id,
req.body, {
      new: true,
      runValidators: 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);

    res.status(204).json({
      status: "success",
      data: null,
    });
  } catch (err) {
    res.status(404).json({
      status: "fail",
      message: err,
    });
  }
};
```

```
//------------------------------------------------------------
//                        LECTURE 101               ⭐⭐
//      Aggregation Pipeline: Matching and Grouping
//------------------------------------------------------------
```

Aggregation pipeline is an extremely powerful and extremely useful **MongoDB framework** for data aggregation.

Now the **aggregation pipeline really is a MongoDB feature.** *But Mongoose, of course, gives us access to it, so that we can use it in the Mongoose driver*

So basically for each of the document that's gonna go through this pipeline, **1 will be added to this numTours counter.**

Group the results for different fields

AvgPrice: 1 for ascending order

```
{
        $sort: {
          avgPrice: 1,
        },
      },
```

Repeating Stages in a Pipeline: Using **$match** twice

```
// AGGREGATION Pipeline ||

exports.getTourStats = async (req, res) => {
  try {
    const stats = await Tour.aggregate([
      {
        $match: { ratingsAverage: { $gte: 4.5 } },
      },

      {
        $group: {
          _id: { $toUpper: "$difficulty" },
          numTours: { $sum: 1 },
          numRatings: { $sum: "$ratingsQuantity" },
          avgRating: { $avg: "$ratingsAverage" },
          avgPrice: { $avg: "$price" },
          minPrice: { $min: "$price" },
          maxPrice: { $max: "$price" },
        },
      },

      {
        $sort: {
          avgPrice: 1,
        },
      },
    ]);

    res.status(200).json({
      status: "success",
      data: {
        stats,
      },
    });
  } catch (err) {
    res.status(404).json({
      status: "fail",
      message: err,
    });
  }
};
```

```
//--------------------------------------------------------------
//                         LECTURE 102                ⭐⭐
//      Aggregation Pipeline: Unwinding and Projecting
//--------------------------------------------------------------
```

So let's imagine that we are really developing this application for the **Natours Company.**

And so let's say that they ask us to implement a function to **calculate the busiest month of a given year. So basically by calculating how many tours start in each of the month of the given year**. And the company really needs this fine tune to prepare accordingly for these tours, like to hire tour guides or to buy the equipment and handle all the stuff like that.

So **this is a real business problem**

$unwind (aggregation)

Definition

$unwind

Deconstructs an array field from the input documents to output a document for *each* element. Each output document is the input document with the value of the array field replaced by the element.

$month (aggregation)

Definition

$month

Returns the month of a date as a number between 1 and 12.

The `$month` expression has the following operator expression syntax:

```
{ $month: <dateExpression> }
```

$project (aggregation)

Definition

$project

Passes along the documents with the requested fields to the next stage in the pipeline. The specified fields can be existing fields from the input documents or newly computed fields.

The `$project` stage has the following prototype form:

```
{ $project: { <specification(s)> } }
```

The `$project` takes a document that can specify the inclusion of fields, the suppression of the `_id` field, the addition of new fields, and the resetting of the values of existing fields. **Alternatively, you may specify the *exclusion* of fields.**

// **Aggregation Pipeline for Calculating Number of Tours Per Month: For the Business Problem: Busiest Months of the year**

```
exports.getMonthlyPlan = async (req, res) => {
  try {
    const year = +req.params.year; // Tours only for 2021
    const plan = await Tour.aggregate([
      {
        $unwind: "$startDates",
      },

      {
        $match: {
          startDates: {
            $gte: new Date(`${year}-01-01`),
            $lte: new Date(`${year}-12-31`),
          },
        },
      },

      {
        $group: {
          _id: { $month: "$startDates" },
          numTourStarts: { $sum: 1 },
          tours: { $push: "$name" },
        },
      },

      {
        $addFields: { month: "$_id" },
      },

      {
        $project: { _id: 0 },
      },

      {
        $sort: {
          numTourStarts: -1,
        },
      },

      {
        $limit: 12,
      },
    ]);

    res.status(200).json({
      status: "success",
      data: {
        plan,
      },
    });
  } catch (err) {
    res.status(404).json({
      status: "fail",
      message: err,
    });
  }
};
```

```
//-------------------------------------------------------------
//                        LECTURE 103               ⭐⭐
//       Virtual Properties
//-------------------------------------------------------------
```

Virtual Properties are basically fields that we can define on our schema

but that will not be persisted. So they will not be saved into the database in order to save us some space there. And most of the time, of course, we want to really save our data to the database, For example a **conversion from miles to kilometers,**

Okay, so let's **now define a virtual property that contains the tour duration in weeks.** And so that's a field basically that **we can very easily convert from the duration that we already have in days,**

Virtual Properties: Converting number of days into Weeks: **w/ regular function, with it OWN "this" keyword: Points to the current document**

Displaying Virtuals in our Schema, for it to be displayed in the Output, as a 2nd argument in our main schema

```
{
    toJSON: { virtuals: true },
    toObject: { virtuals: true },
  }
```

Knowing the Duration in Weeks is A Business LOGIC, and Not Application Logic, Hence we are putting it into Our Model

// Virtual Properties: Converting number of days into Weeks: w/ regular function, with it OWN "this" keyword

```
tourSchema.virtual("durationWeeks").get(function () {
  return this.duration / 7;
});
```

```
//-------------------------------------------------------------
//                         LECTURE 104               ⭐⭐
//       Document Middleware
//-------------------------------------------------------------
```

Mongoose Middleware (also called **pre and post *hooks***) are functions which are passed control during execution of asynchronous functions. Middleware is specified on the schema level and is useful for writing plugins.

Used to run Functions, BEFORE and AFTER, we Run any Event

Types of Middleware

Mongoose has 4 types of middleware: **document** middleware, **model** middleware, **aggregate** middleware, and **query** middleware

DMAQ

REMEMBER: *Will not run while using* **.insertMany()**

Creating Slugs for the **Tour Names** using **Slugify**

```
npm i slugify
```

DOCUMENT MIDDLEWARE: runs before .save() and .create(): **Also called pre and post Hooks**

Warning: "this" points to current document on NEW Document Creation: Will NOT WORK while updating a document

```
tourSchema.pre("save", function (next) {
  this.slug = slugify(this.name, { lower: true });
  next();
});

tourSchema.pre("save", function (next) {
  console.log("Will save document...");
  next();
});

tourSchema.post("save", function (doc, next) {
  console.log(doc);
  next();
});
```

```
//------------------------------------------------------------
//                        LECTURE 105                ⭐⭐
//       Query Middleware
//------------------------------------------------------------
```

QUERY MIDDLEWARE: *It runs before or after **.find()** query*

Defining it in the Schema

```
secretTour: {
      type: Boolean,
      default: false,
    },
```

Creating a Secret Tour: Using the Query Middleware: We want to exclude some secret tours from the regular queries by the customers

REGEX for query middleware for find : ---> **/^find/**

All the strings that start with find

Excluding the Secret Tour from all the queries

Creating a Clock, in order to see, how much time it takes, a query to get executed, **using Query Middleware** for a the **post hook for the find query,**

QUERY MIDDLEWARE Clock : runs before or after .find() query

```
tourSchema.pre(/^find/, function (next) {
  this.find({ secretTour: { $ne: true } });

  this.start = Date.now();
  next();
});

tourSchema.post(/^find/, function (docs, next) {
  console.log(`Query took ${Date.now() - this.start} milliseconds!`);
  // console.log(docs);
  next();
});
```

```
//------------------------------------------------------------
//                        LECTURE 106                ⭐⭐
//      Aggregation Middleware
//------------------------------------------------------------
```

Aggregation middleware allows us to add hooks before or after an aggregation happens

AGGREGATION MIDDLEWARE: excluding the Secret Tour from the tour-stats query while using the aggregation method and adding to our object as a property to the start using JavaScript unshift() array method. Excluding the secret tour from all aggregation methods by adding it to our pipeline itself

```
tourSchema.pre("aggregate", function (next) {
  this.pipeline().unshift({ $match: { secretTour: { $ne: true } } });
  console.log(this.pipeline());
  next();
});
```

Alright, so middleware is some really cool and important stuff that we can add to our models, and there is indeed other cool stuff that we can do with models, for example **implementing instance methods which are methods that will be available on every document after being queried, and that again can be quite handy, and we will do that later in the *authentication section*** as well

```
//-----------------------------------------------------------
//                        LECTURE 107               ⭐⭐
//      Data Validation: Built-In Validators
//-----------------------------------------------------------
```

Well, **VALIDATION** is basically checking if the entered values are in the right format for each field in our document schema, and also that values have actually been entered for all of the required fields. Now, on the other hand, we also have **SANITIZATION,** *which is to ensure that the inputted data is basically clean, so that there is* ***no malicious code being injected into our database,*** *or into the application itself.*

Sanitization will be covered in a very detailed way in the Security Section

So, in that step we remove unwanted characters, or even code, from the input data, **Its a GOLDEN Standard in Backend development. To never ever accept the data which is coming from the input, before being sanitized.**

Validators are avaiable for all the data types: Strings, Numbers, Booleans, Dates or whatever type that I want.

Required is a Validator, but Unique is Technically NOT a Real Validator: Remember that

```
required: [true, "Error String: A tour must have a name"],
unique: true,
```

----> From tourModel.js

```
name: {
      type: String,
      required: [true, "Error String: A tour must have a name"],
      unique: true,
      trim: true,
      maxlength: [
        40,
        "Error String: A tour name must have less than or equal to 40
characters",
      ],
      minlength: [
        10,
        "Error String: A tour name must have greater than or equal to 10
characters",
      ],
    },
```

----> From tourController.js

// PATCH Request: Validator was already set way before in the course, when we created this request

```
exports.updateTour = async (req, res) => {
  try {
    const tour = await Tour.findByIdAndUpdate(req.params.id, req.body, {
      new: true,
      runValidators: true,
    });

    res.status(200).json({
      status: "success",
      data: {
        tour,
      },
    });
  } catch (err) {
    res.status(404).json({
      status: "fail",
      message: err,
    });
  }
};
```

----> From tourModel.js

```
ratingsAverage: {
      type: Number,
      default: 4.5,
      min: [1, "Error String: Rating must be above or equal to 1.0"],
      max: [5, "Error String: Rating must be below or equal to 5.0"],
    },
```

Next up, I want to **restrict this difficulty value here to only three difficulties.** So, easy, medium, and difficult. And if the user puts in something else, then it's not going to work.

enum Validator: *Only for strings*

min, max Validator: *for Numbers, Dates*

There are actually a bunch of other validators. For example, **on strings you have a match validator in order to check if the input matches a given regular expression.**

But, I believe that these ones that I just showed you here are the most important built in validators. Now ***we can also specify our own validators,*** *and so, that is exactly what we will do right in the next video.*

```
difficulty: {
      type: String,
      required: [true, "Error String: A tour must have a difficulty
level"],
      enum: {
        values: ["easy", "medium", "difficult"],
        message:
          "Error String: Difficulty level must be either: easy, medium,
difficult",
      },
    },
//-------------------------------------------------------------
//                         LECTURE 108                ⭐⭐
//      Data Validation: Custom Validators
//-------------------------------------------------------------
```

A Validator is actually really just a simple function which should return either true or false. And if it returns false, then it means there is an error. And on the other hand when we return true, then the validation is correct and the input can be accepted.

Okay, so **let's now build a simple custom validator here. And what I want to validate is if the price discount is actually lower than the price itself.**

That's something that we cannot do using the built-in validators and so we're simply gonna build our own.

```
priceDiscount: {
      type: Number,
      validate: function (val) {
        return val < this.price; // price discount 100 < price 200
      },
    },
```

```
priceDiscount: {
      type: Number,
      validate: {
        // Warning: "this" points to current document on NEW Document
Creation: Will NOT WORK while updating a document
        validator: function (val) {
          return val < this.price;
        },
        message:
          "Error String: Discount value ({VALUE}) must be lower than the
regular price value",
      },
    },
```

({VALUE}) is **equal** to the inputted **"val" in our validator function.** Its really an internal **Mongoose property**, and not a JavaScript Property

But also, **there are a couple of libraries on npm for data validation that we can simply plug in here as custom validators that we do not have to write ourselves.** And **the most popular library is called validator**

Checkout googling " **validator Github** "

We don't need all of them, okay, but there is one very specific, which I want to use, which is **isAlpha.**

So **I want to check if the tour name only contains letters. And so for that I can use this function from the validator library.**

```
npm install validator
```

We will not include this validation in our code, as the Tour name can have space in between them, and this validation would not work. Its just to showcase, how we can use external libraries in our code. ***We well still use this library when we check if the EMAIL is valid, when the user inputs it.***

```
const validator = require("validator");
```

```
name: {
      type: String,
      required: [true, "Error String: A tour must have a name"],
      unique: true,
      trim: true,
      maxlength: [
        40,
        "Error String: A tour name must have less than or equal to 40
characters",
      ],
      minlength: [
        10,
        "Error String: A tour name must have greater than or equal to 10
characters",
      ],
      validate: {
        validator: validator.isAlpha,
        message: "Error String: Tour name must only contain characters",
      },
    },
```

*So if we really wanted to test if the string only contains letters and spaces it would probably be simpler to simply use a **REGEX to test for that kind of pattern**.*

Node.js

SECTION 09 - Error Handling with Express

EC-137: Sinha, A

```
//--------------------------------------------------------------
//                         LECTURE 110                ☆☆
//       Debugging NodeJS with ndb
//--------------------------------------------------------------
```

There are different ways of debugging Node.JS code.

For example, we could use VScode for that. **But, actually, Google very recently released an amazing tool which we can use to debug which is called NDB**

npm install ndb --save-dev

In **package.json** add: Would work for both global and dev dependency installs

```
"debug": "ndb server.js"
```

npm run debug

Okay, but now about the debugging itself. I would say that the fundamental aspect of the debugging is to set break points. So **Break Points are basically points in our code that we can define here in the debugger, where our code will then stop running.**

It will basically freeze in time and we can then take a look at all our variables. Okay, so that will then be extremely useful to find some bugs.

All right, next I also want to take a look at our app variable, so basically the Express application that we export from app.js, remember that.

And here is really a ton of stuff but what I find interesting is to take a look at this **router** here.

Okay, and so **in router we then have stack.** Okay, and so let's open this up a little bit

and so this **stack here is basically the middleware stack that we have in our application.**

```
//------------------------------------------------------------
//                GLOBAL MODIFICATION       ⭐⭐
//              UPDATED server.js File:
// Getting Rid of Deprecation Warnings in the console
//------------------------------------------------------------
```

IMPORTANT: *THIS MODIFICATION IS* ***NOT REQUIRED WHEN USING Mongoose Version 6 or Higher***. All the previous versions require this in order to get rid of the deprecation warnings in the console.

```
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.DATABASE_PASSWORD
);

// to connect to MongoDB Altas Cloud Database
mongoose
  .connect(DB, {
    useNewUrlParser: true,
    useCreateIndex: true,
    useFindAndModify: false,
    useUnifiedTopology: true,       ← ✸ Added/Modified
this
  })
  .then(() => {
    console.log("Database connection successful!");
  });

// To connect to the local database
// mongoose
//   .connect(process.env.DATABASE_LOCAL, {
//     useNewUrlParser: true,
//     useCreateIndex: true,
//     useFindAndModify: false,
//     useUnifiedTopology: true,
//   })
//   .then(() => {
//     console.log("Database connection successful!");
//   });

const port = process.env.PORT || 3000;

app.listen(port, () => {
  console.log(`App running on port ${port}...`);
});
```

```
//-------------------------------------------------------------
//                        LECTURE 111              ☆☆
//       Handling Unhandled Routes
//-------------------------------------------------------------
```

Now Let's create a handler function for all the routes that are not cached by our routers.

All right now how are we gonna implement a route handler for a route that was not cached by any of our other route handlers?

So for doing that remember that all these middleware functions are executed in the order they are in the code. And **so the idea is that if we have a request that makes it into this point here of our code, that is after all the routers, then it means that neither the tourRouter nor the userRouter were able to cache it**, okay.

And so ***if we add a middleware after all other routers***, it will only be reached again if not handled by any of our other routers, okay.

And so in Express, we can use **app.all()**. And **so that's then going to run for all the verbs**, So **all the HTTP methods,**

Location: app.js

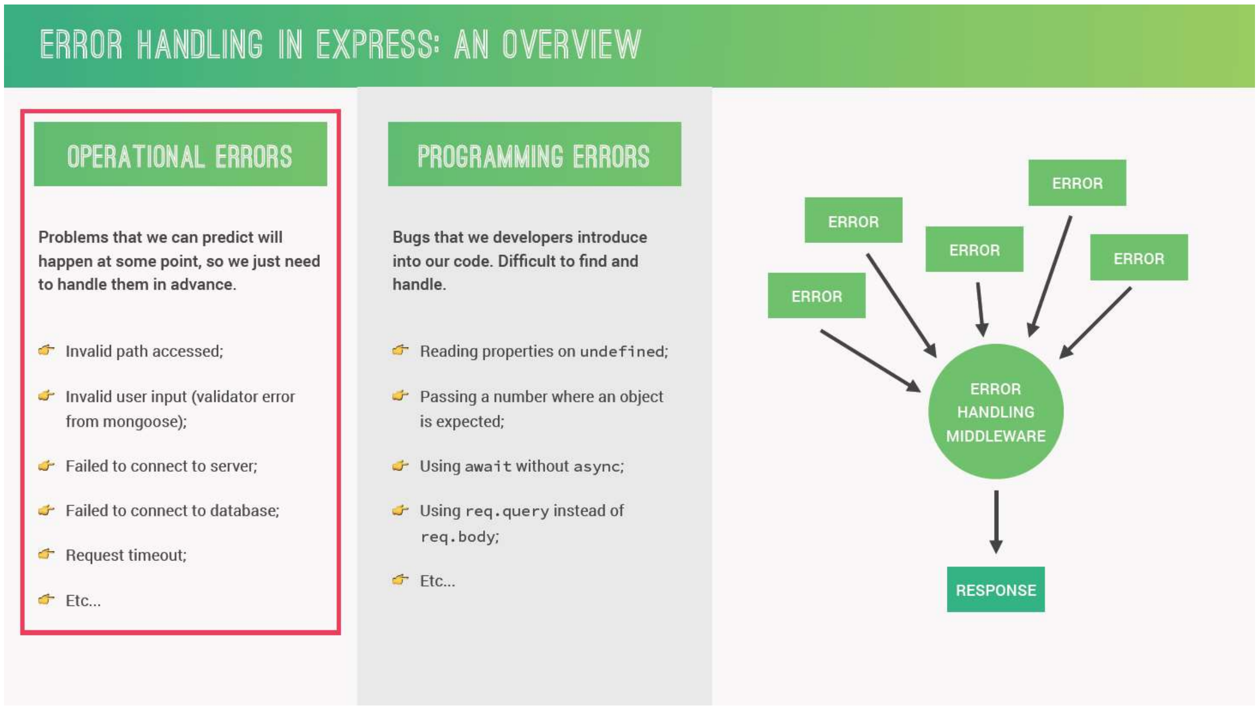
```
app.all("*", (req, res, next) => {
  res.status(404).json({
    status: "fail",
    message: `Can't find ${req.originalUrl} on this server!`,
  });
});
```

Request Url in Postman: **127.0.0.1:3000/api/tours/**

Response:

```
{
    "status": "fail",
    "message": "Can't find /api/tours/ on this server!"
}
```

```
//------------------------------------------------------------
//                        LECTURE 112                  ☆ ☆
//       An Overview of Error Handling
//------------------------------------------------------------
```



ERROR HANDLING IN EXPRESS: AN OVERVIEW

OPERATIONAL ERRORS

Problems that we can predict will happen at some point, so we just need to handle them in advance.

- Invalid path accessed;
- Invalid user input (validator error from mongoose);
- Failed to connect to server;
- Failed to connect to database;
- Request timeout;
- Etc...

PROGRAMMING ERRORS

Bugs that we developers introduce into our code. Difficult to find and handle.

- Reading properties on `undefined`;
- Passing a number where an object is expected;
- Using `await` without `async`;
- Using `req.query` instead of `req.body`;
- Etc...

```
//------------------------------------------------------------
//                          LECTURE 113                 ☆☆
//       Implementing Global Error Handling Middleware
//------------------------------------------------------------
```

Please Refer the Video Lecture for Better Understanding

```
// Module IMPORTS

const express = require("express");
const morgan = require("morgan");

const tourRouter = require("./routes/tourRoutes");
const userRouter = require("./routes/userRoutes");

const app = express();

// 1) MIDDLEWARES

if (process.env.NODE_ENV === "development") {
  app.use(morgan("dev"));
}

app.use(express.json());
app.use(express.static(`${__dirname}/public`));

app.use((req, res, next) => {
  req.requestTime = new Date().toISOString(); // Manipulating Request,
using next()
  next();
});

// 3) Our ROUTERS WITH Routes

// Middlewares: Mounting the Routers Properly ✓✓

app.use("/api/v1/tours", tourRouter);
app.use("/api/v1/users", userRouter);

app.all("*", (req, res, next) => {
  // res.status(404).json({
  //   status: "fail",
  //   message: `Can't find ${req.originalUrl} on this server!`,
  // });

  const err = new Error(`Can't find ${req.originalUrl} on this server!`);
  err.status = "fail";
  err.statusCode = 404;

  next(err);
});

// Global: Error Handling Middleware
app.use((err, req, res, next) => {
  err.statusCode = err.statusCode || 500;
  err.status = err.status || "error";

  res.status(err.statusCode).json({
    status: err.status,
    message: err.message,
  });
});

module.exports = app;
```

GET Request URL in POSTMAN: **127.0.0.1:3000/api/tours/**

Response:

```
{
    "status": "fail",
    "message": "Can't find /api/tours/ on this server!"
}
```

```
//-------------------------------------------------------------
//                         LECTURE 114                ☆☆
//       Better Errors and Refactoring
//-------------------------------------------------------------
```

PLEASE UPDATE the Notes While Revisiting this LECTURE

In our AppError Class, while calling the ***super(message)*****, we put message as the ONLY PARAMETER, message** is actually the **only parameter that the built-in Error Class accepts, which, indeed we are extending**...with our **AppError Class**

This is basically like CALLING ERROR, when we use message in the super.

All the errors that we will create using this class will all be operational errors. *So, errors that we can predict will happen in some point in the future, like for example a user creating a tour without the required fields, right?*

So **that is an operational error**, okay, and so again, from now on, we will always use this **AppError class** here that we're creating right now in order to create all the errors in our application.

So **this.operational**, and set it to **true.**

So all of our errors will get this property set to true, and I'm doing that so that later we can then test for this propertyand only send error messages back to the client for these operational errors that we created using this class.

And *this is useful because some other crazy unexpected errors that might happen in our application, for example a* ***programming error, or some bug in one of the packages*** *that we require into our app, and these errors will then of course not have this .is operational property on them*

What is a stack trace error?

Stack trace error is a generic term frequently associated with long error messages. The stack trace information **identifies where in the program the error occurs** and is helpful to programmers. For users, the long stack track information may not be very useful for troubleshooting web errors.

```
// Global: Error Handling Middleware

app.use((err, req, res, next) => {
  console.log(err.stack);

  err.statusCode = err.statusCode || 500;
  err.status = err.status || "error";

  res.status(err.statusCode).json({
    status: err.status,
    message: err.message,
  });
});
```

GET Request in POSTMAN: `127.0.0.1:3000/api/tours/`

Response:

```
{
    "status": "fail",
    "message": "Can't find /api/tours/ on this server!"
}
```

Log in the Console:

```
[nodemon] starting `node server.js`
App running on port 3000...
Database connection successful!
Error: Can't find /api/tours/ on this server!
```

What does **captureStackTrace** do ?

Creates a `.stack` property on `targetObject`, which when accessed returns a string representing the location in the code at which `Error.captureStackTrace()` was called.

All that matters is that we just add this line of code here, which is **Error.capturestackTrace** so exactly what you get here, and at first we need to specify the current object, which is this, and then the AppError class itself, which is gonna be this.constructor.

Okay, and so this way when a new object is created, and a constructor function is called, then that function call is not gonna appear in the stack trace, and will not pollute it.

Great, just one question that you might have is **'Why didn't I set this.message equal to message**?'

Well, that's just because right here I called the parent class, right, and the **parent class is Error ,** and whatever we pass into it is gonna be the message property. So just as I told you before. And so, basically, in here by doing this parent call we already set the message property to our incoming message.

Location: utils/appError.js

```
class AppError extends Error {
  constructor(message, statusCode) {
    super(message);

    this.statusCode = statusCode;
    this.status = `${statusCode}`.startsWith("4") ? "fail" : "error";
    this.isOperational = true;

    Error.captureStackTrace(this, this.constructor);
  }
}

module.exports = AppError;
```

GET Request in POSTMAN: 127.0.0.1:3000/api/tours/

Response:

```
{    "status": "fail",    "message": "Can't find /api/tours/ on this
server!"}
```

Log in the Console:

```
[nodemon] starting `node server.js`App running on port 3000...Database
connection successful!Error: Can't find /api/tours/ on this server!
```

```
//-------------------------------------------------------------
//                         LECTURE 115               ⭐⭐
//       Catching Errors in Async Functions
//-------------------------------------------------------------
```

PLEASE REFER LECTURE VIDEO

```
//-------------------------------------------------------------
//                         LECTURE 116               ⭐⭐
//       Adding 404 Not Found Error
//-------------------------------------------------------------
```

PLEASE REFER LECTURE VIDEO

```
//-------------------------------------------------------------
//                        LECTURE 117                ⭐⭐
//       Errors: Development Vs Production
//-------------------------------------------------------------
```

In this video, **we're gonna implement some logic in order to send different error messages for the development and production environment.**

So right now, we're sending this same response message here basically to everyone, no matter if we're in development or in production.

But the ***idea is that in production,*** *we want to leak as little information about our errors to the client as possible. So in that case, we only want to send a nice, human-friendly message so that the user knows what's wrong.*

On the other hand, when in ***development,*** *we want to get as much information about the error that occurred as possible, and we want that right in the error message that's coming back.* So we could log that information also to the console, but I think it's way more useful to have that information right in postman, in this case.

So, we already know how to distinguish between the development and the production environment.

LOCATION: **utils/errorController.js**

```
module.exports = (err, req, res, next) => {
  //   console.log(err.stack);

  err.statusCode = err.statusCode || 500;
  err.status = err.status || "error";

  if (process.env.NODE_ENV === "development") {
    res.status(err.statusCode).json({
      status: err.status,
      message: err.message,
      error: err,
      stack: err.stack,
    });
  } else if (process.env.NODE_ENV === "production") {
    res.status(err.statusCode).json({
      status: err.status,
      message: err.message,
    });
  }
};
```

Putting the Error Responses, in their own functions: *Refactoring*

```
const sendErrorDev = (err, res) => {
  res.status(err.statusCode).json({
    status: err.status,
    message: err.message,
    error: err,
    stack: err.stack,
  });
};

const sendErrorProd = (err, res) => {
  res.status(err.statusCode).json({
    status: err.status,
    message: err.message,
  });
};

module.exports = (err, req, res, next) => {
  //   console.log(err.stack);

  err.statusCode = err.statusCode || 500;
  err.status = err.status || "error";

  if (process.env.NODE_ENV === "development") {
    sendErrorDev(err, res);
  } else if (process.env.NODE_ENV === "production") {
    sendErrorProd(err, res);
  }
};
```

Just like we Implemented the Functionality in the ***AppError Class****, in the* ***appError.js*** *file, the* ***operational errors****, we are going to use that functionality into our Production/Operational Environment Error Handling Message.*

```
const sendErrorProd = (err, res) => {
  // Operational, Trusted Error: Send message to client
  if (err.isOperational) {
    res.status(err.statusCode).json({
      status: err.status,
      message: err.message,
    });
    // Programming or other Unknown Error: Don't leak error details to
client
  } else {
    res.status(500).json({
      status: "error",
      message: "Something went wrong!",
    });
  }
};
```

Now **there are real logging libraries on npm**, that we could use here instead of just having this simple console.error, but just logging the error to the console will make it visible in the logs on the hosting platform that you're using.

I think that for now, in this kind of simple application, that is enough. **For example, we're gonna use Heroku by the end of the course to deploy our application.**

Then **when an error like this occurs, it will be logged to the console. In the Heroku platform, we then have access to these logs**. We can keep looking at these logs, and then in there we're gonna find these unexpected errors so that we can then fix them.

Right now, we're actually already building kind of sophisticated and real-world error-handling mechanism here.

FILE: **controllers/errorController.js** **← REFACTORED**

```
const sendErrorDev = (err, res) => {
  res.status(err.statusCode).json({
    status: err.status,
    message: err.message,
    error: err,
    stack: err.stack,
  });
};

const sendErrorProd = (err, res) => {
  // Operational, Trusted Error: Send message to client
  if (err.isOperational) {
    res.status(err.statusCode).json({
      status: err.status,
      message: err.message,
    });

    // Programming or other Unknown Error: Don't leak error details to
client
  } else {
    // 1) Log Error
    console.error("ERROR ❌", err);

    // 2) Send Generic Message
    res.status(500).json({
      status: "error",
      message: "Something went wrong!",
    });
  }
};

module.exports = (err, req, res, next) => {
  //   console.log(err.stack);

  err.statusCode = err.statusCode || 500;
  err.status = err.status || "error";

  if (process.env.NODE_ENV === "development") {
    sendErrorDev(err, res);
  } else if (process.env.NODE_ENV === "production") {
    sendErrorProd(err, res);
  }
};
```

```
//------------------------------------------------------------
//                        LECTURE 118                ⭐⭐
//      Handling Invalid Database IDs
//------------------------------------------------------------
```

There are three types of errors that might be created by Mongoose in which we need to **mark as operational errors** so that we can **send back meaningful error messages to clients in production.**

And let's now start by simulating these three errors, okay?

Type 1: Operational Error

GET Request: 127.0.0.1:3000/api/v1/tours/wwwwwwwww

RESPONSE

```
{
    "status": "error",
    "message": "Cast to ObjectId failed for value \"wwwwwwwww\" (type strin
g) at path \"_id\" for model \"Tour\"",
    "error": {
        "stringValue": "\"wwwwwwwww\"",
        "valueType": "string",
        "kind": "ObjectId",
        "value": "wwwwwwwww",
        "path": "_id",
        "reason": {},
        "name": "CastError",
        "message": "Cast to ObjectId failed for value \"wwwwwwwww\" (type s
tring) at path \"_id\" for model \"Tour\""
    },
    "stack": "CastError: Cast to ObjectId failed for value \"wwwwwwwww\" (t
ype string) at path \"_id\" for model \"Tour\"\n    at model.Query.exec (C:
\\Users\\Ankur\\Desktop\\03-Node-Jonas\\Node.js\\4.4-natours-with-
MongoDB\\node_modules\\mongoose\\lib\\query.js:4498:21)\n    at model.Query
.Query.then (C:\\Users\\Ankur\\Desktop\\03-Node-Jonas\\Node.js\\4.4-
natours-with-MongoDB\\node_modules\\mongoose\\lib\\query.js:4592:15)"
}
```

Type 2: Operational Error: Duplicate Name

POST Request: 127.0.0.1:3000/api/v1/tours

```
{
    "name": "The Forest Hiker",
    "maxGroupSize": 1,
    "difficulty": "easy",
    "duration" : 7,
    "price": 200,
    "summary": "Test Tour",
    "imageCover": "tour-3-cover.jpg",
    "ratungsAverage": 4
  }
```

RESPONSE

```
{
    "status": "error",
    "message": "E11000 duplicate key error collection: natours.tours index: 
name_1 dup key: { name: \"The Forest Hiker\" }",
    "error": {
        "driver": true,
        "name": "MongoError",
        "index": 0,
        "code": 11000,
        "keyPattern": {
            "name": 1
        },
        "keyValue": {
            "name": "The Forest Hiker"
        },
        "statusCode": 500,
        "status": "error"
    },
    "stack": "MongoError: E11000 duplicate key error collection: 
natours.tours index: name_1 dup key: { name: \"The Forest Hiker\" }\n    at 
Function.create (C:\\Users\\Ankur\\Desktop\\03-Node-Jonas\\Node.js\\4.4-
natours-with-MongoDB\\node_modules\\mongodb\\lib\\core\\error.js:59:12)\n    
at toError (C:\\Users\\Ankur\\Desktop\\03-Node-Jonas\\Node.js\\4.4-natours-
with-MongoDB\\node_modules\\mongodb\\lib\\utils.js:130:22)\n    at 
C:\\Users\\Ankur\\Desktop\\03-Node-Jonas\\Node.js\\4.4-natours-with-
MongoDB\\node_modules\\mongodb\\lib\\operations\\common_functions.js:258:39
\n    at handler (C:\\Users\\Ankur\\Desktop\\03-Node-Jonas\\Node.js\\4.4-
natours-with-
MongoDB\\node_modules\\mongodb\\lib\\core\\sdam\\topology.js:961:24)\n    
at C:\\Users\\Ankur\\Desktop\\03-Node-Jonas\\Node.js\\4.4-natours-with-
MongoDB\\node_modules\\mongodb\\lib\\cmap\\connection_pool.js:352:13\n    
at handleOperationResult (C:\\Users\\Ankur\\Desktop\\03-Node-
Jonas\\Node.js\\4.4-natours-with-
MongoDB\\node_modules\\mongodb\\lib\\core\\sdam\\server.js:567:5)\n    at 
MessageStream.messageHandler (C:\\Users\\Ankur\\Desktop\\03-Node-
Jonas\\Node.js\\4.4-natours-with-
MongoDB\\node_modules\\mongodb\\lib\\cmap\\connection.js:308:5)\n    at 
MessageStream.emit (events.js:400:28)\n    at processIncomingData 
(C:\\Users\\Ankur\\Desktop\\03-Node-Jonas\\Node.js\\4.4-natours-with-
MongoDB\\node_modules\\mongodb\\lib\\cmap\\message_stream.js:144:12)\n    
at MessageStream._write (C:\\Users\\Ankur\\Desktop\\03-Node-
Jonas\\Node.js\\4.4-natours-with-
MongoDB\\node_modules\\mongodb\\lib\\cmap\\message_stream.js:42:5)\n    at
writeOrBuffer (internal/streams/writable.js:358:12)\n    at
MessageStream.Writable.write (internal/streams/writable.js:303:10)\n    at
TLSSocket.ondata (internal/streams/readable.js:731:22)\n    at
TLSSocket.emit (events.js:400:28)\n    at addChunk
(internal/streams/readable.js:293:12)\n    at readableAddChunk
(internal/streams/readable.js:267:9)"
}
```

Type 3: Operational Error: Validation Error

PATCH Request: 127.0.0.1:3000/api/v1/tours/619bad0efbb4962194f4d065

```
{
    "ratingsAverage": 6
}
```

RESPONSE

```
{
    "status": "error",
    "message": "Validation failed: ratingsAverage: Error String: Rating
must be below or equal to 5.0",
    "error": {
        "errors": {
            "ratingsAverage": {
                "name": "ValidatorError",
                "message": "Error String: Rating must be below or equal to
5.0",
                "properties": {
                    "message": "Error String: Rating must be below or equal
to 5.0",
                    "type": "max",
                    "max": 5,
                    "path": "ratingsAverage",
                    "value": 6
                },
                "kind": "max",
                "path": "ratingsAverage",
                "value": 6
            }
        },
        "_message": "Validation failed",
        "statusCode": 500,
        "status": "error",
        "name": "ValidationError",
        "message": "Validation failed: ratingsAverage: Error String: Rating
must be below or equal to 5.0"
    },
    "stack": "ValidationError: Validation failed: ratingsAverage: Error
String: Rating must be below or equal to 5.0\n    at _done
(C:\\Users\\Ankur\\Desktop\\03-Node-Jonas\\Node.js\\4.4-natours-with-
MongoDB\\node_modules\\mongoose\\lib\\helpers\\updateValidators.js:236:19)\
n    at C:\\Users\\Ankur\\Desktop\\03-Node-Jonas\\Node.js\\4.4-natours-
with-
MongoDB\\node_modules\\mongoose\\lib\\helpers\\updateValidators.js:212:11\n
at schemaPath.doValidate.updateValidator (C:\\Users\\Ankur\\Desktop\\03-
Node-Jonas\\Node.js\\4.4-natours-with-
MongoDB\\node_modules\\mongoose\\lib\\helpers\\updateValidators.js:170:13)\
n    at C:\\Users\\Ankur\\Desktop\\03-Node-Jonas\\Node.js\\4.4-natours-
with-MongoDB\\node_modules\\mongoose\\lib\\schematype.js:1273:9\n    at
processTicksAndRejections (internal/process/task_queues.js:77:11)"
}
```

<u>Type 1: Operational Error → SOLUTION</u>

GET Request: 127.0.0.1:3000/api/v1/tours/wwwwwwwww

FILE: controllers/errorController.js ← **REFACTORED: BUG FIXED**

```
const AppError = require("../utils/appError");

const handleCastErrorDB = (err) => {
  const message = `Invalid ${err.path}: ${err.value}`;
  return new AppError(message, 400);
};

const sendErrorDev = (err, res) => {
  res.status(err.statusCode).json({
    status: err.status,
    message: err.message,
    error: err,
    stack: err.stack,
  });
};

const sendErrorProd = (err, res) => {
  // Operational, Trusted Error: Send message to client
  if (err.isOperational) {
    res.status(err.statusCode).json({
      status: err.status,
      message: err.message,
    });

    // Programming or other Unknown Error: Don't leak error details to
client
  } else {
    // 1) Log Error
    console.error("ERROR ❌", err);

    // 2) Send Generic Message
    res.status(500).json({
      status: "error",
      message: "Something went wrong!",
    });
  }
};

module.exports = (err, req, res, next) => {
  //   console.log(err.stack);

  err.statusCode = err.statusCode || 500;
  err.status = err.status || "error";
  if (process.env.NODE_ENV === "development") {
    sendErrorDev(err, res);
  } else if (process.env.NODE_ENV === "production") {
    let error = Object.assign(err);                        ← ✹ BUG FIXED

    if (error.name === "CastError") error = handleCastErrorDB(error);

    sendErrorProd(error, res);
  }
};
```

GET Request: 127.0.0.1:3000/api/v1/tours/wwwwwww

RESPONSE Now ☺

```
{
    "status": "fail",
    "message": "Invalid _id: wwwwww"
}
```

RESOLVED:

Find the `let error = { ...err };` line In the **errorController.js** file and change it to `let error = Object.assign(err);` That'll fix your issue!

THIS CODE WILL NOT CLONE YOUR OBJECT!!!

This code does nothing else than giving you back a reference to the err object. That is not what we want. Of course you'll then have access to all not enumerable and not own properties, but you did'nt make a copy.

Read this:
https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Object/assign

A real clone would be: `let error = Object.assign({}, err)` what is exact the same as `let error = { ...err };`

```
//-----------------------------------------------------------
//                        LECTURE 119                ☆☆
//      Handling Duplicate Database Fields
//-----------------------------------------------------------
```

So, *we tried to create a new tour with a name that already exists and the name fields is supposed to be unique and so we get this error.*

Now this **error here doesn't have a name property, okay? And that's because, as I mentioned before, it is actually not an error that is caused by a Mongoose.**

But instead, really, by the underlying MongoDB driver, okay?

And so, what we're gonna do to **identify this error is use this 11,000 code** here.

Type 2: Operational Error: Duplicate Name

POST Request: 127.0.0.1:3000/api/v1/tours

```
{
    "name": "The Forest Hiker",
    "maxGroupSize": 1,
    "difficulty": "easy",
    "duration" : 7,
    "price": 200,
    "summary": "Test Tour",
    "imageCover": "tour-3-cover.jpg",
    "ratungsAverage": 4
  }
```

RESPONSE

```
{
    "status": "error",
    "message": "E11000 duplicate key error collection: natours.tours index:
name_1 dup key: { name: \"The Forest Hiker\" }",
    "error": {
        "driver": true,
        "name": "MongoError",
        "index": 0,
        "code": 11000,
        "keyPattern": {
            "name": 1
        },
        "keyValue": {
            "name": "The Forest Hiker"
        },
        "statusCode": 500,
        "status": "error"
    },
    "stack": "MongoError: E11000 duplicate key error collection:
natours.tours index: name_1 dup key: { name: \"The Forest Hiker\" }\n    at
Function.create (C:\\Users\\Ankur\\Desktop\\03-Node-Jonas\\Node.js\\4.4-
natours-with-MongoDB\\node_modules\\mongodb\\lib\\core\\error.js:59:12)\n
at toError (C:\\Users\\Ankur\\Desktop\\03-Node-Jonas\\Node.js\\4.4-natours-
with-MongoDB\\node_modules\\mongodb\\lib\\utils.js:130:22)\n    at
C:\\Users\\Ankur\\Desktop\\03-Node-Jonas\\Node.js\\4.4-natours-with-
MongoDB\\node_modules\\mongodb\\lib\\operations\\common_functions.js:258:39
\n    at handler (C:\\Users\\Ankur\\Desktop\\03-Node-Jonas\\Node.js\\4.4-
natours-with-
MongoDB\\node_modules\\mongodb\\lib\\core\\sdam\\topology.js:961:24)\n
at C:\\Users\\Ankur\\Desktop\\03-Node-Jonas\\Node.js\\4.4-natours-with-
MongoDB\\node_modules\\mongodb\\lib\\cmap\\connection_pool.js:352:13\n
at handleOperationResult (C:\\Users\\Ankur\\Desktop\\03-Node-
Jonas\\Node.js\\4.4-natours-with-
MongoDB\\node_modules\\mongodb\\lib\\core\\sdam\\server.js:567:5)\n    at
MessageStream.messageHandler (C:\\Users\\Ankur\\Desktop\\03-Node-
Jonas\\Node.js\\4.4-natours-with-
MongoDB\\node_modules\\mongodb\\lib\\cmap\\connection.js:308:5)\n    at
MessageStream.emit (events.js:400:28)\n    at processIncomingData
(C:\\Users\\Ankur\\Desktop\\03-Node-Jonas\\Node.js\\4.4-natours-with-
MongoDB\\node_modules\\mongodb\\lib\\cmap\\message_stream.js:144:12)\n
at MessageStream._write (C:\\Users\\Ankur\\Desktop\\03-Node-
Jonas\\Node.js\\4.4-natours-with-
MongoDB\\node_modules\\mongodb\\lib\\cmap\\message_stream.js:42:5)\n    at
writeOrBuffer (internal/streams/writable.js:358:12)\n    at
MessageStream.Writable.write (internal/streams/writable.js:303:10)\n    at
TLSSocket.ondata (internal/streams/readable.js:731:22)\n    at
TLSSocket.emit (events.js:400:28)\n    at addChunk
(internal/streams/readable.js:293:12)\n    at readableAddChunk
(internal/streams/readable.js:267:9)"
}
```

Type 2: Operational Error: Duplicate Name: SOLUTION

FILE: controllers/errorController.js **🡐 REFACTORED: BUG FIXED**

```
const AppError = require("../utils/appError");

const handleCastErrorDB = (err) => {
  const message = `Invalid ${err.path}: ${err.value}`;
  return new AppError(message, 400);
};

const handleDuplicateFieldDB = (err) => {
  // const value = Object.values(err.keyValue)[0];
  const value = Object.values(err.keyValue).join(",");
  const message = `Duplicate field value(s): ${value}. Use another
value(s).`;
  return new AppError(message, 400);
};

const sendErrorDev = (err, res) => {
  res.status(err.statusCode).json({
    status: err.status,
    message: err.message,
    error: err,
    stack: err.stack,
  });
};

const sendErrorProd = (err, res) => {
  // Operational, Trusted Error: Send message to client
  if (err.isOperational) {
    res.status(err.statusCode).json({
      status: err.status,
      message: err.message,
    });

    // Programming or other Unknown Error: Don't leak error details to
client
  } else {
    // 1) Log Error
    console.error("ERROR ❌", err);

    // 2) Send Generic Message
    res.status(500).json({
      status: "error",
      message: "Something went wrong!",
    });
  }
};

module.exports = (err, req, res, next) => {
  //   console.log(err.stack);

  err.statusCode = err.statusCode || 500;
  err.status = err.status || "error";

  if (process.env.NODE_ENV === "development") {
    sendErrorDev(err, res);
  } else if (process.env.NODE_ENV === "production") {
    let error = Object.assign(err);

    if (error.name === "CastError") error = handleCastErrorDB(error);
    if (error.code === 11000) error = handleDuplicateFieldDB(error);

    sendErrorProd(error, res);
  }
};
```

POST Request: 127.0.0.1:3000/api/v1/tours

```
{
    "name": "The Forest Hiker",
    "maxGroupSize": 1,
    "price": 1000,
    "difficulty": "easy",
    "duration": 12,
    "summary": "Test Tour",
    "imageCover": "tour-3-cover.jpg",
    "ratingsAverage": 4
  }
```

RESPONSE Now ☺

```
{
    "status": "fail",
    "message": "Duplicate field value(s): The Forest Hiker. Use another
value(s)."

}
```

```
//------------------------------------------------------------
//                          LECTURE 120                ☆☆
//      Handling Mongoose Validation Errors
//------------------------------------------------------------
```

```
In Development Mode: npm start
```

POST Request: 127.0.0.1:3000/api/v1/tours/619bad0efbb4962194f4d065

```
{
    "ratingsAverage": 6,
    "difficulty": "whatever",
    "name": "Short"
}
```

RESPONSE:

```
{
    "status": "error",
    "message": "Validation failed: name: Error String: A tour name must hav
e greater than or equal to 10 characters, difficulty: Error String: Difficu
lty level must be either: easy, medium, difficult, ratingsAverage: Error St
ring: Rating must be below or equal to 5.0",
    "error": {
        "errors": {
            "name": {
                "name": "ValidatorError",
                "message": "Error String: A tour name must have greater tha
n or equal to 10 characters",
                "properties": {
                    "message": "Error String: A tour name must have greater
 than or equal to 10 characters",
                    "type": "minlength",
                    "minlength": 10,
                    "path": "name",
                    "value": "Short"
                },
                "kind": "minlength",
                "path": "name",
                "value": "Short"
            },
            "difficulty": {
                "name": "ValidatorError",
                "message": "Error String: Difficulty level must be either:
easy, medium, difficult",
                "properties": {
                    "message": "Error String: Difficulty level must be eith
er: easy, medium, difficult",
                    "type": "enum",
                    "enumValues": [
                        "easy",
                        "medium",
                        "difficult"
                    ],
                    "path": "difficulty",
                    "value": "whatever"
                },
                "kind": "enum",
                "path": "difficulty",
                "value": "whatever"
            },
            "ratingsAverage": {
                "name": "ValidatorError",
                "message": "Error String: Rating must be below or equal to
5.0",
                "properties": {
                    "message": "Error String: Rating must be below or equal
 to 5.0",
                    "type": "max",
                    "max": 5,
                    "path": "ratingsAverage",
                    "value": 6
                },
                "kind": "max",
                "path": "ratingsAverage",
                "value": 6
            }
        },
        "_message": "Validation failed",
        "statusCode": 500,
        "status": "error",
        "name": "ValidationError",
        "message": "Validation failed: name: Error String: A tour name must
 have greater than or equal to 10 characters, difficulty: Error String: Dif
ficulty level must be either: easy, medium, difficult, ratingsAverage: Erro
r String: Rating must be below or equal to 5.0"
    },
    "stack": "ValidationError: Validation failed: name: Error String: A tou
r name must have greater than or equal to 10 characters, difficulty: Error
String: Difficulty level must be either: easy, medium, difficult, ratingsAv
erage: Error String: Rating must be below or equal to 5.0\n    at _done (C:
\\Users\\Ankur\\Desktop\\03-Node-Jonas\\Node.js\\4.4-natours-with-
MongoDB\\node_modules\\mongoose\\lib\\helpers\\updateValidators.js:236:19)\
n    at C:\\Users\\Ankur\\Desktop\\03-Node-Jonas\\Node.js\\4.4-natours-
with-
MongoDB\\node_modules\\mongoose\\lib\\helpers\\updateValidators.js:212:11\n
    at schemaPath.doValidate.updateValidator (C:\\Users\\Ankur\\Desktop\\03
-Node-Jonas\\Node.js\\4.4-natours-with-
MongoDB\\node_modules\\mongoose\\lib\\helpers\\updateValidators.js:170:13)\
n    at C:\\Users\\Ankur\\Desktop\\03-Node-Jonas\\Node.js\\4.4-natours-
with-
MongoDB\\node_modules\\mongoose\\lib\\schematype.js:1273:9\n    at processT
icksAndRejections (internal/process/task_queues.js:77:11)"
}
```

Now **in order to create one big string out of all the strings from all the errors, we basically have to loop over all of these objects, and then extract all the error messages into a new array.**

And in **JavaScript, we use Object.values in order to basically loop over an object**. So the elements of an object.

Type 3: Validation Error: SOLUTION

FILE: controllers/errorController.js

```
const AppError = require("../utils/appError");

const handleCastErrorDB = (err) => {
  const message = `Invalid ${err.path}: ${err.value}`;
  return new AppError(message, 400);
};

const handleDuplicateFieldDB = (err) => {
  // const value = Object.values(err.keyValue)[0];
  const value = Object.values(err.keyValue).join(",");
  const message = `Duplicate field value(s): ${value}. Please use another
value(s).`;
  return new AppError(message, 400);
};

const handleValidationErrorDB = (err) => {
  const errors = Object.values(err.errors).map((el) => el.message);
  const message = `Invalid input data. ${errors.join("|| ")}`;
  return new AppError(message, 400);
};

const sendErrorDev = (err, res) => {
  res.status(err.statusCode).json({
    status: err.status,
    message: err.message,
    error: err,
    stack: err.stack,
  });
};

const sendErrorProd = (err, res) => {
  // Operational, Trusted Error: Send message to client
  if (err.isOperational) {
    res.status(err.statusCode).json({
      status: err.status,
      message: err.message,
    });

    // Programming or other Unknown Error: Don't leak error details to
client
  } else {
    // 1) Log Error
    console.error("ERROR ❌", err);
    // 2) Send Generic Message
    res.status(500).json({
      status: "error",
      message: "Something went wrong!",
    });
  }
};

module.exports = (err, req, res, next) => {
  //   console.log(err.stack);

  err.statusCode = err.statusCode || 500;
  err.status = err.status || "error";

  if (process.env.NODE_ENV === "development") {
    sendErrorDev(err, res);
  } else if (process.env.NODE_ENV === "production") {
    let error = Object.assign(err);

    if (error.name === "CastError") error = handleCastErrorDB(error);
    if (error.code === 11000) error = handleDuplicateFieldDB(error);
    if (error.name === "ValidationError")
      error = handleValidationErrorDB(error);

    sendErrorProd(error, res);
  }
};
```

TESTING In Production Mode: **npm run start:prod**

PATCH Request:
127.0.0.1:3000/api/v1/tours/619bad0efbb4962194f4d065

BODY

```
{
    "ratingsAverage": 6,
    "difficulty": "whatever",
    "name": "Short"
}
```

RESPONSE Now ☺

```
{
    "status": "fail",
    "message": "Invalid input data. Error String: A tour name must have
greater than or equal to 10 characters|| Error String: Difficulty level
must be either: easy, medium, difficult|| Error String: Rating must be
below or equal to 5.0"
}
```

Now we could've made this error, handling error, a lot more complete still.

For example, **we could define different error severity levels like saying, this error is not so important, this error is medium important, and this error is very important or even critical.**

And we could also then email some administrator about critical errors. And really, there's a lot of stuff that we could implement. But again, in a kind of small application like this one, what we have here is already really good, okay? So this is quite a robust strategy already that we have implemented here, and I'm really happy with it, okay?

So all this logic here with the operational errors that we implemented here, so that's already quite sophisticated.

Okay?

Now if we were ever to find another error that we want to mark as operational, then of course all we would have to do is something similar to what we have here, okay? So basically implement another function for that one, and then return our own operational error so that the send error production can then actually send that operational error to the client, right?

Our error controller is actually finished.

But there are still some other errors that we need to handle which are completely outside of Mongo or even of Express. And so we're doing that in the rest of this section.

```
//---------------------------------------------------------------
//                        LECTURE 121                 ☆☆
//       Errors Outside Express: Unhandled Rejections
//---------------------------------------------------------------
```

Config.env w/ PASSWORD for MongoDB

```
NODE_ENV=development
PORT=3000
DATABASE=mongodb+srv://ankur:<PASSWORD>@cluster0.59dgp.mongodb.net/natours?
retryWrites=true&w=majority
DATABASE_LOCAL=mongodb://localhost:27017/natours
DATABASE_PASSWORD=
```

FILE: 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.DATABASE_PASSWORD
);

// to connect to MongoDB Altas Cloud Database
mongoose
  .connect(DB, {
    useNewUrlParser: true,
    useCreateIndex: true,
    useFindAndModify: false,
    useUnifiedTopology: true,
  })
  .then(() => {
    console.log("Database connection successful!");
  });

// To connect to the local database

// mongoose
//   .connect(process.env.DATABASE_LOCAL, {
//     useNewUrlParser: true,
//     useCreateIndex: true,
//     useFindAndModify: false,
//     useUnifiedTopology: true,
//   })
//   .then(() => {
//     console.log("Database connection successful!");
//   });
```

```
const port = process.env.PORT || 3000;

const server = app.listen(port, () => {
  console.log(`App running on port ${port}...`);
});

process.on("unhandledRejection", (err) => {
  console.log(err.name, err.message);
  console.log("UNHANDLED REJECTION! 🔥 Shutting Down ");
  server.close(() => {
    process.exit(1);
  });
});
```

In this video, let's talk about something that we have in node.js called **unhandled rejections** and then learn how we can actually handle them.

So at this point, we have successfully handled errors in our express application by passing operational asynchronous errors down into a global error handling middleware.

This, then sends relevant error messages back to the client depending on the type of error that occurred, right?

However, ***there might also occur errors outside of express and a good example for that in our current application is the mongodb database connection.***

So **imagine that the database is down for some reason or for some reason, we cannot log in. And in that case, there are errors that we have to handle as well. But they didn't occur inside of our express application** and so, of course, our error handler that we implemented will not catch this errors, right? And **so just to test what happens, let's go ahead and change our mongodb password**, okay? Because that way, we're not gonna be able to connect to the database, right?

So an **unhandled promise rejection means that somewhere in our code, there is a promise that got rejected. But that rejection has not been handled anywhere**, all right? **And down here, you also see a deprecation warning which says that in the future unhandled rejections will simply exit the node program that's running**, which may not always be what you want, okay?

So **let's fix this problem and get rid of this unhandled promise rejection.**

So this would work, of course, but I really want to show you ***how to globally handle unhandled rejected promises, because in a bigger application*, it can become a bit more difficult to always keep track of all the promises that might become rejected at some point, okay?**

And so at some point, you might have some unhandled promise rejection somewhere and so let me show you how to deal with that **globally basically.**

And so let's **now learn how to handle unhandled rejections**

And **so remember how in one of the first section of the course, we talked about events and event listeners,** right?

And so now, it's time to actually use that knowledge.

So **each time that there is an unhandled rejection somewhere in our application, the process object will emit an object called unhandled rejection and so we can subscribe to that event just like this.**

So we always have to assume that we as programmers are gonna make errors. And so it's always best to have a central place like this to handle all promise rejections like a last safety net, all right? Now, if we really have like some problem with the database connection, like we have in this example, then our application is not gonna work at all.

And so all we can really do here is to shut down our application, all right?

So to shutdown the application, we use **process.exit** And we actually already used that before in that script where we imported all the data into the database from the JSON file, remember?

So **process.exit** and then in here, **we can actually pass a code.**

And the **code 0** stands for a **success** and **code 1** stands for **uncaught exception.** And so that's the one that's usually used here, all right? So usually, you will always see it like this.

Now, there is just **one problem** with the way we implemented it right now and that is, that the way we implement it here so just **process.exit is a very abrupt way** of ending the program because this will just immediately abort all the requests that are currently still running or pending and so that might not be a good idea, okay?

And **so usually, what we do is to shutdown gracefully where we first close the server and only then, we shut down the application**, okay?

Before we do that, we need to **save the server here basically to a variable**, okay?

And so the result of calling app.listen is a server and to now, on that server, we can then say server.close which will, as the name says, close the server and then after that's done, it will run this callback function that we passed into it and so it's only here, where we then shut down the server, okay?

And so by doing this, **by doing server.close, we give the server, basically time to finish all the request that are still pending or being handled at the time, and only after that, the server is then basically killed**, all right?

IMPORTANT: In the real world scenario, we should always do it like this, okay?

And of course, that's not really ideal that the application crashed, right? Because right now, of course, the app is not running, it's not working at all, right? And so usually, in a production app on a web server, we will usually have some tool in place that restarts the application right after it crashes, or also some of the platforms that host node.js will automatically do that on their own, okay? Because, of course, we don't wanna leave the application hanging like this forever.

And so basically, this is how you handle unhandled rejected promises.

So again, **basically, we are listening to this unhandled rejection event, which then allows us to handle all the errors that occur in asynchronous code which were not previously handled. But now, you might ask, what about the synchronous code?**

Where are we gonna handle that? And the answer to that lies, as you can imagine, in the next video.

```
//-----------------------------------------------------------
//                        LECTURE 122                ☆☆
//              Catching Uncaught Exceptions
//-----------------------------------------------------------
```

Well, **all errors, or let's also call them bugs, that occur in our Synchronous Code but are not handled anywhere are called uncaught exceptions.**

FILE: server.js ← REFACTORED

```
const mongoose = require("mongoose");
const dotenv = require("dotenv");

// ORDER of code Matters Here: Handling the Synchronous Errors if Any

process.on("uncaughtException", (err) => {
  console.log("UNCAUGHT REJECTION! 🔥 Shutting Down ");
  console.log(err.name, err.message);
  process.exit(1);
});

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

const DB = process.env.DATABASE.replace(
  "<PASSWORD>",
  process.env.DATABASE_PASSWORD
);

// to connect to MongoDB Altas Cloud Database
mongoose
  .connect(DB, {
    useNewUrlParser: true,
    useCreateIndex: true,
    useFindAndModify: false,
    useUnifiedTopology: true,
  })
  .then(() => {
    console.log("Database connection successful!");
  });

// To connect to the local database

// mongoose
//   .connect(process.env.DATABASE_LOCAL, {
//     useNewUrlParser: true,
//     useCreateIndex: true,
//     useFindAndModify: false,
//     useUnifiedTopology: true,
//   })
//   .then(() => {
//     console.log("Database connection successful!");
//   });

const port = process.env.PORT || 3000;

const server = app.listen(port, () => {
  console.log(`App running on port ${port}...`);
});

process.on("unhandledRejection", (err) => {
  console.log("UNHANDLED REJECTION! 🔥 Shutting Down ");
  console.log(err.name, err.message);
  server.close(() => {
    process.exit(1);
  });
});
```