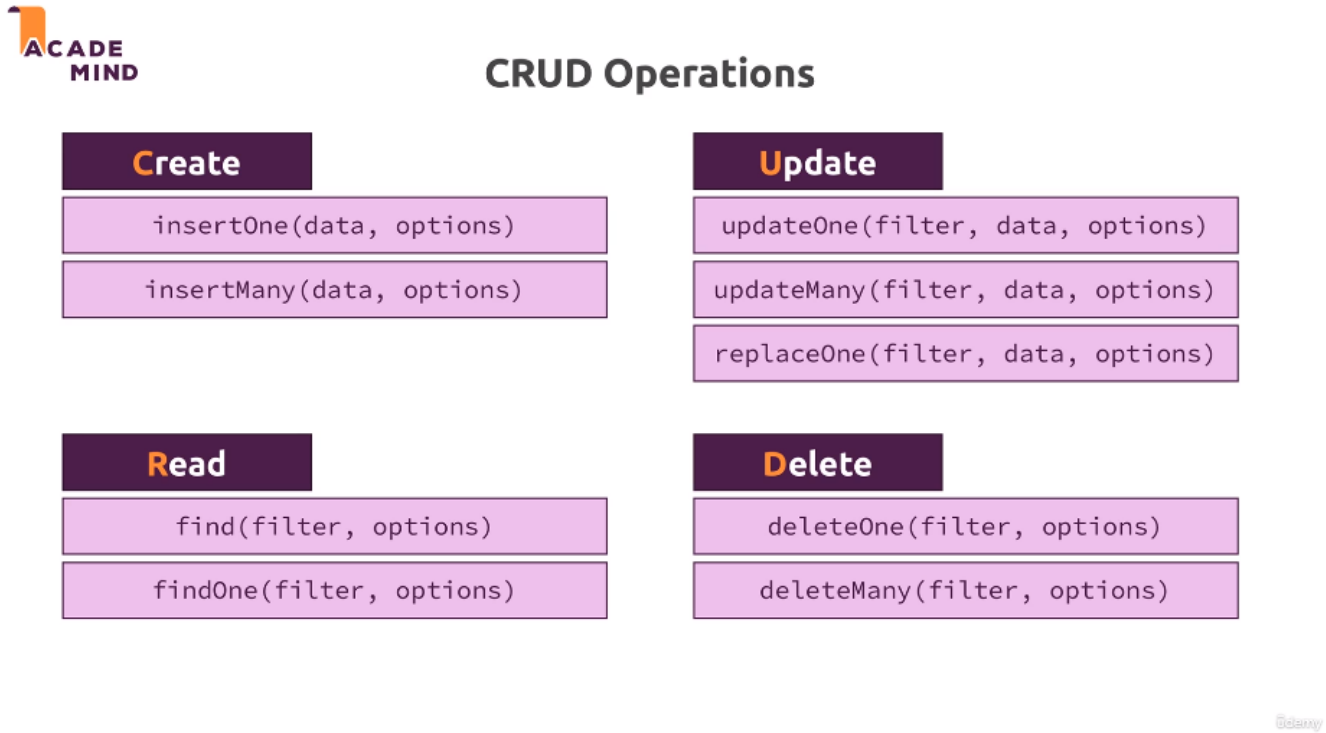
- Use a database

```
use <db_name>
```

- Insert documents in mongoDB

```

```

Note -> HERE WE ARE RUNNING ALL THE COMMANDS IN MONGO SHELL NOT USING MONGOOSE



NOTE -> $keyword are MongoDB's reserved keyword

- $set -> used in insert & update operations, tells the MongoDB what to set new or change in the existing document.

```
> db.flightData.updateMany({}, {$set: {marker: "toDelete"}})
{ "acknowledged" : true, "matchedCount" : 2, "modifiedCount" : 2 }
> db.flightData.find().pretty()
{
        "_id" : ObjectId("5b97827de62da95ae64206a8"),
        "departureAirport" : "MUC",
        "arrivalAirport" : "SFO",
        "aircraft" : "Airbus A380",
        "distance" : 12000,
        "intercontinental" : true,
        "marker" : "toDelete"
}
{
        "_id" : "txl-lhr-1",
        "departureAirport" : "TXL",
        "arrivalAirport" : "LHR",
        "marker" : "toDelete"
}
```

- $gt -> Stands for greater than, mostly used for finding documents with certain greater than values.

```
> db.flightData.findOne({distance: {$gt: 900}})
{
        "_id" : ObjectId("5b97882ce62da95ae64206ab"),
        "departureAirport" : "MUC",
        "arrivalAirport" : "SFO",
        "aircraft" : "Airbus A380",
        "distance" : 12000,
        "intercontinental" : true
}
```

Query Selectors

Comparison

For comparison of different BSON type values, see the specified BSON comparison order.

Name	Description
`$eq`	Matches values that are equal to a specified value.
`$gt`	Matches values that are greater than a specified value.
`$gte`	Matches values that are greater than or equal to a specified value.
`$in`	Matches any of the values specified in an array.
`$lt`	Matches values that are less than a specified value.
`$lte`	Matches values that are less than or equal to a specified value.
`$ne`	Matches all values that are not equal to a specified value.
`$nin`	Matches none of the values specified in an array.

Difference between update, updateOne & updateMany

- **updateOne** and **updateMany** needs the **$set** keyword to work and update documents and if not given they **throw an error**, while **update()** will work without it as well, just that it will instead of updating the value of certain key or keys we passed it will replace the entire document with the second argument object we passed to it only leaving the previous key intact (just like replaceOne).

```
> db.flightData.updateOne({_id: ObjectId("5b97882ce62da95ae64206ab")}, {delayed: false})
2018-09-11T11:26:26.614+0200 E QUERY    [js] Error: the update operation document must contain atomic
operators :
DBCollection.prototype.updateOne@src/mongo/shell/crud_api.js:542:1
@(shell):1:1
> db.flightData.updateMany({_id: ObjectId("5b97882ce62da95ae64206ab")}, {delayed: false})
2018-09-11T11:26:33.027+0200 E QUERY    [js] Error: the update operation document must contain atomic
operators :
DBCollection.prototype.updateMany@src/mongo/shell/crud_api.js:625:1
@(shell):1:1
> db.flightData.update({_id: ObjectId("5b97882ce62da95ae64206ab")}, {delayed: false})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.flightData.find().pretty()
{ "_id" : ObjectId("5b97882ce62da95ae64206ab"), "delayed" : false }
{
        "_id" : ObjectId("5b97882ce62da95ae64206ac"),
        "departureAirport" : "LHR",
        "arrivalAirport" : "TXL",
        "aircraft" : "Airbus A320",
        "distance" : 950,
        "intercontinental" : false
}
```

- replaceOne -> it will replace the matching document with the second object passed to it basically replacing all the keys just keeping the previous _id intact.

```
> db.flightData.replaceOne({_id: ObjectId("5b97882ce62da95ae64206ab")}, {
...     "departureAirport": "MUC",
...     "arrivalAirport": "SFO",
...     "aircraft": "Airbus A380",
...     "distance": 12000,
...     "intercontinental": true
...   })
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.flightData.find().pretty()
{
        "_id" : ObjectId("5b97882ce62da95ae64206ab"),
        "departureAirport" : "MUC",
        "arrivalAirport" : "SFO",
        "aircraft" : "Airbus A380",
        "distance" : 12000,
        "intercontinental" : true
}
```

*NOTE -> In shell when we use find() we get a Cursor which doesn't give us complete data but a subpart of it as total data might be large. We can use **it** to get next sub-parts. Remember this doesn't happen for findOne, update, insert, delete.*

NOTE -> .pretty() is also a method that just exists in cursor.

- PROJECTION -> It allows us to retrieve on specific mentioned keys from a collection of documents instead of all the keys of the documents in that collection.

```
> db.passengers.find({},{name: 1}).pretty()
{
        "_id" : ObjectId("5b978c66e62da95ae64206ad"),
        "name" : "Max Schwarzmueller"
}
{ "_id" : ObjectId("5b978c66e62da95ae64206ae"), "name" : "Manu Lorenz" }
{ "_id" : ObjectId("5b978c66e62da95ae64206af"), "name" : "Chris Hayton" }
{ "_id" : ObjectId("5b978c66e62da95ae64206b0"), "name" : "Sandeep Kumar" }
{ "_id" : ObjectId("5b978c66e62da95ae64206b1"), "name" : "Maria Jones" }
{ "_id" : ObjectId("5b978c66e62da95ae64206b2"), "name" : "Alexandra Maier" }
{ "_id" : ObjectId("5b978c66e62da95ae64206b3"), "name" : "Dr. Phil Evans" }
{ "_id" : ObjectId("5b978c66e62da95ae64206b4"), "name" : "Sandra Brugge" }
{ "_id" : ObjectId("5b978c66e62da95ae64206b5"), "name" : "Elisabeth Mayr" }
{ "_id" : ObjectId("5b978c66e62da95ae64206b6"), "name" : "Frank Cube" }
{ "_id" : ObjectId("5b978c66e62da95ae64206b7"), "name" : "Karandeep Alun" }
{ "_id" : ObjectId("5b978c66e62da95ae64206b8"), "name" : "Michaela Drayer" }
{ "_id" : ObjectId("5b978c66e62da95ae64206b9"), "name" : "Bernd Hoftstadt" }
{ "_id" : ObjectId("5b978c66e62da95ae64206ba"), "name" : "Scott Tolib" }
{ "_id" : ObjectId("5b978c66e62da95ae64206bb"), "name" : "Freddy Melver" }
{ "_id" : ObjectId("5b978c66e62da95ae64206bc"), "name" : "Alexis Bohed" }
{ "_id" : ObjectId("5b978c66e62da95ae64206bd"), "name" : "Melanie Palace" }
{ "_id" : ObjectId("5b978c66e62da95ae64206be"), "name" : "Armin Glutch" }
{ "_id" : ObjectId("5b978c66e62da95ae64206bf"), "name" : "Klaus Arber" }
{ "_id" : ObjectId("5b978c66e62da95ae64206c0"), "name" : "Albert Twostone" }
Type "it" for more
```

_id is included by default. 1 for include and 0 for not include.

- Going to nested objects in a document

```
> db.flightData.find({"status.description": "on-time"}).pretty()
{
        "_id" : ObjectId("5b97882ce62da95ae64206ab"),
        "departureAirport" : "MUC",
        "arrivalAirport" : "SFO",
        "aircraft" : "Airbus A380",
        "distance" : 12000,
        "intercontinental" : true,
        "status" : {
                "description" : "on-time",
                "lastUpdated" : "1 hour ago",
                "details" : {
                        "responsible" : "Max Schwarzmueller"
                }
        }
}
```

ACADEMIND

Example #2: Patient Data

Patient

```
{
  "firstName": "Max",
  "lastName": "Schwarzmueller",
  "age": 29,
  "history": [
    { "disease": "cold", "treatment": … },
    { … }
  ]
}
```

ACADEMIND

Tasks

1	Insert 3 patient records with at least 1 history entry per patient
2	Update patient data of 1 patient with new age, name and history entry
3	Find all patients who are older than 30 (or a value of your choice)
4	Delete all patients who got a cold as a disease

ACADEMIND

Module Summary

Databases, Collections, Documents	CRUD Operations
▪ A Database holds multiple Collections where each Collection can then hold multiple Documents ▪ Databases and Collections are created "lazily" (i.e. when a Document is inserted) ▪ A Document can't directly be inserted into a Database, you need to use a Collection!	▪ CRUD = Create, Read, Update, Delete ▪ MongoDB offers multiple CRUD operations for single-document and bulk actions (e.g. insertOne(), insertMany(), ...) ▪ Some methods require an argument (e.g. insertOne()), others don't (e.g. find()) ▪ find() returns a cursor, NOT a list of documents! ▪ Use filters to find specific documents
Document Structure	**Retrieving Data**
▪ Each document needs a unique ID (and gets one by default) ▪ You may have embedded documents and array fields	▪ Use filters and operators (e.g. $gt) to limit the number of documents you retrieve ▪ Use projection to limit the set of fields you retrieve

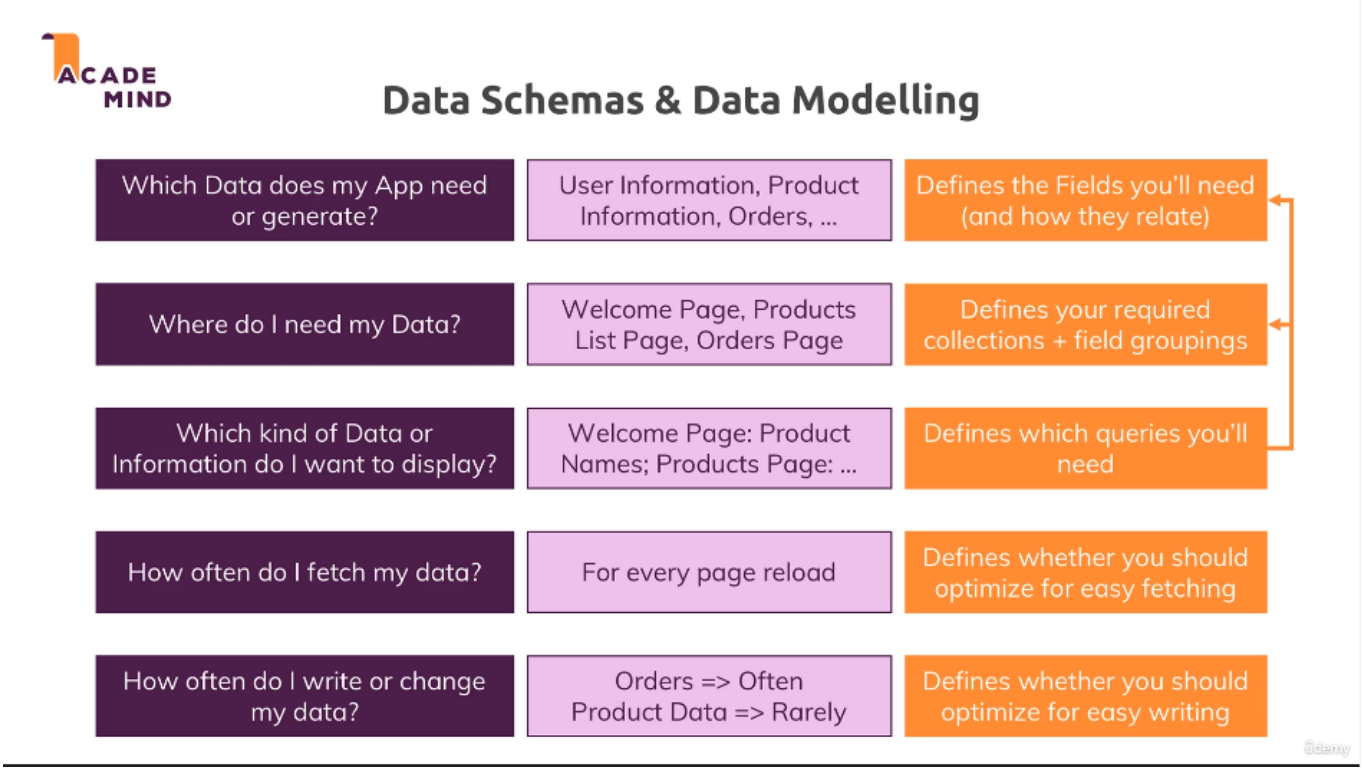
- Dropping Database or Collections

Resetting Your Database

Important: We will regularly start with a clean database server (i.e. all data was purged) in this course.

To get rid of your data, you can simply load the database you want to get rid of (`use databaseName`) and then execute `db.dropDatabase()`.

Similarly, you could get rid of a single collection in a database via `db.myCollection.drop()`.



Joining in MongoDB using $lookup

NOTE -> just like .populate() in mongoose

Query ->

```
> db.books.aggregate([{$lookup: {from: "authors", localField: "authors", foreignField: "_id", as: "cre
ators"}}]).pretty()
```

Result ->

```
{
        "_id" : ObjectId("5b98d9984d01c52e1637a9a5"),
        "name" : "My favorite Book",
        "authors" : [
                ObjectId("5b98d9e44d01c52e1637a9a6"),
                ObjectId("5b98d9e44d01c52e1637a9a7")
        ],
        "creators" : [
                {
                        "_id" : ObjectId("5b98d9e44d01c52e1637a9a6"),
                        "name" : "Max Schwarz",
                        "age" : 29,
                        "address" : {
                                "street" : "Main"
                        }
                },
                {
                        "_id" : ObjectId("5b98d9e44d01c52e1637a9a7"),
                        "name" : "Manuel Lor",
                        "age" : 30,
                        "address" : {
                                "stree" : "Tree"
                        }
                }
        ]
} _
```

Notice that in above example creates was stored using references and got populated.

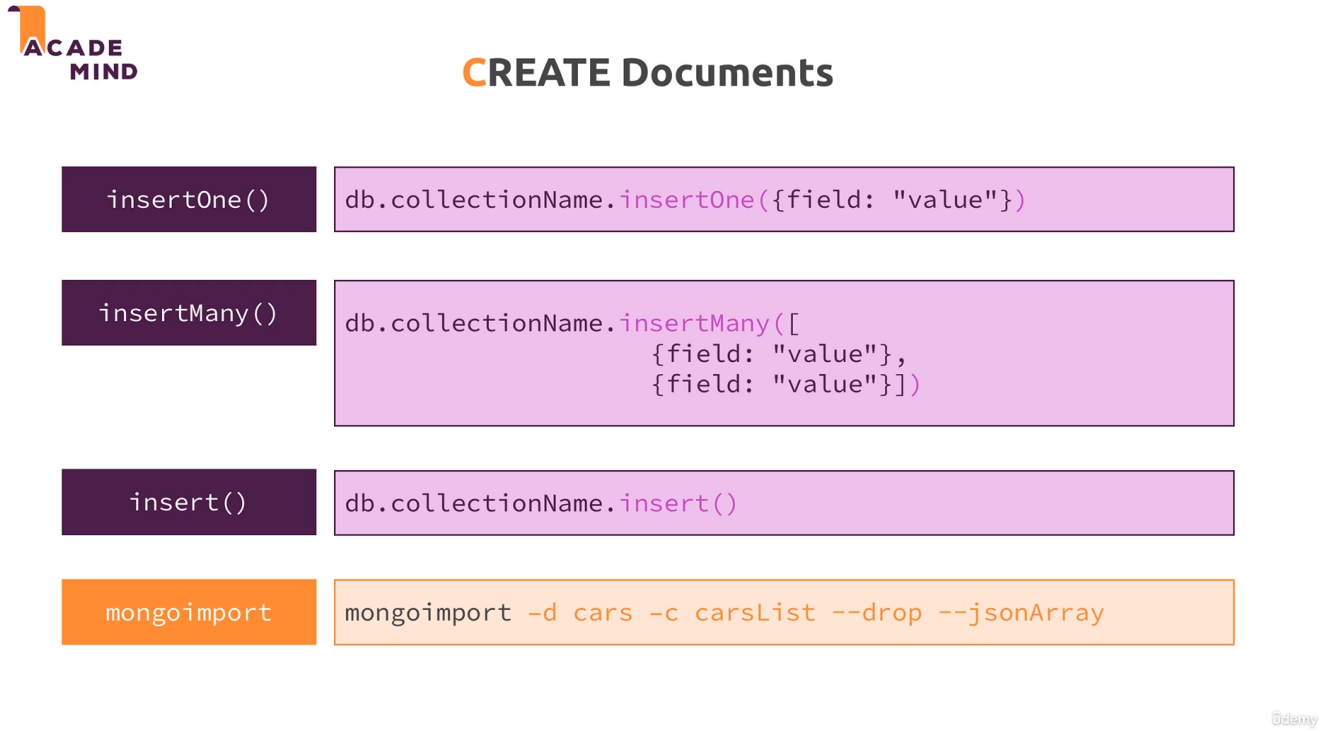
```
db.posts.aggregate([
        {
                $lookup: {
                        from: "users",
                        localField: "author",
                        foreignField: "_id",
                        as: "populatedAuthor"
                }
        }
]).pretty();
```

```
// document
{
        _id: <>,
        title: "",
        text: "",
        creator: {},
```

```
        comments: [
                {
                        text: "",
                        author: <>
                }
        ]
}

db.createCollection('posts', {
        validator: {
                $jsonSchema: {
                    bsonType: 'object',
                        required: ['title', 'text', 'creator', 'comments'],
                    properties: {
                            title: {
                                    bsonType: 'string',
                                    description: 'must be a string and is
required'
                            },
                            text: {
                                bsonType: 'string',
                                description: 'must be a string and is
required'
                        },
                        creator: {
                                bsonType: 'objectId',
                                description: 'must be an objectid and is
required'
                        },
                        comments: {
                                bsonType: 'array',
                                description: 'must be an array and is
required',
                                items: {
                                        bsonType: 'object',
                                            required: ['text', 'author'],
                                            properties: {
                                                text: {
                                                bsonType: 'string',
                                                description: 'must be a
string and is required'
                                                    },
                                                        author: {
                                                bsonType: 'objectId',
                                                description: 'mustbe an
objectid and is required'
```

```
                                                    }
                                                }
                                            }
                                        }
                                    }
                                }
                            }
    });
```

Creating new documents in MongoDB database



- **insertOne ->**

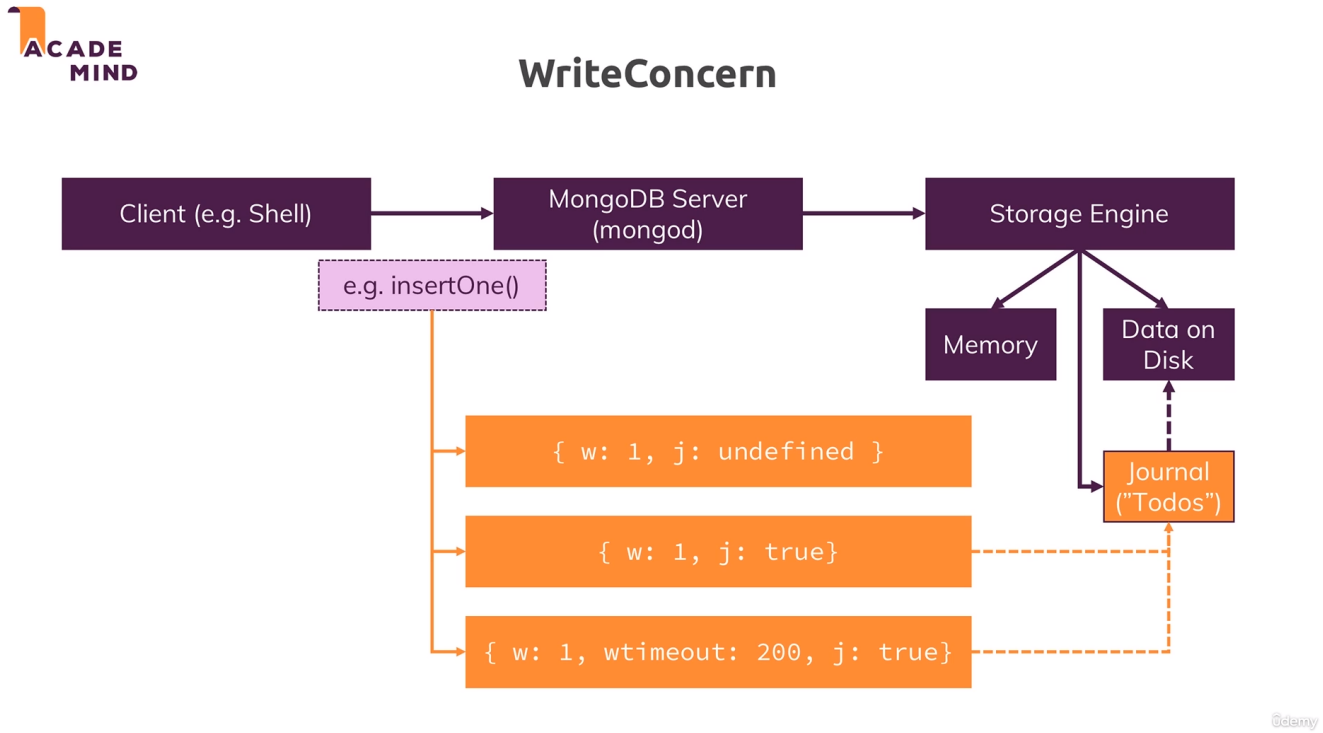
```
> db.persons.insertOne({name: "Manuel", age: 31, hobbies: ["Cars", "Cooking"]})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("5b9f4a0f44ade05febc85b60")
}
```

- **insertMany ->**

```
> db.persons.insertMany([{name: "Anna", age: 29, hobbies: ["Sports", "Yoga"]}])
{
        "acknowledged" : true,
        "insertedIds" : [
                ObjectId("5b9f4a5544ade05febc85b61")
        ]
}
> db.persons.insertMany([{name: "Maria", age: 31}, {name: "Chris", age: 25}])
{
        "acknowledged" : true,
        "insertedIds" : [
                ObjectId("5b9f4a7f44ade05febc85b62"),
                ObjectId("5b9f4a7f44ade05febc85b63")
        ]
}
```

NOTE -> Use insertOne or insertMany instead of insert as they are the latest version, returns the _id of inserted documents and are less error prone.

Write Concern



we know that storage engine in mongodb stores the document at first at memory instead of disc as it might be actively needed and retrive & write in memory is quite fast than disc.

so we can pass a second object to insert commands

```
/*
w -> if 1 - write to disc then return response
```

```
      else return a response immiediatly even though it hasn't been
          written on disc

         (also something about number of instances)

  j -> if true - write this task in journal and then only return a response
           else don't write task in journal

  wtimeout -> within how much time mongodb server to report a success
  */

  { w: 1, j: undefined }
  { w: 1, j: true }
  { w: 1, wtimeout: 200, j: true }
```

Journal basically holds all the tasks that server needs to-do. Quite helpful incase while executing query if DB server goes down it will have a record of what it was doing and has to do now.

```
> db.persons.insertOne({name: "Michaela", age: 51}, {writeConcern: {w: 1, j: true}})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("5b9f4fb044ade05febc85b6a")
}
```

*NOTE -> In insertMany if an error occurred while inserting multiple documents, the documents that were before the document on which error occurred in insertMany array won't be removed from collection in DB and no document are the error throwing document will be inserted. This is the default behaviour.

However if we want to continue inserting the remaining documents we can pass on a second attribute to insertMany like this ->

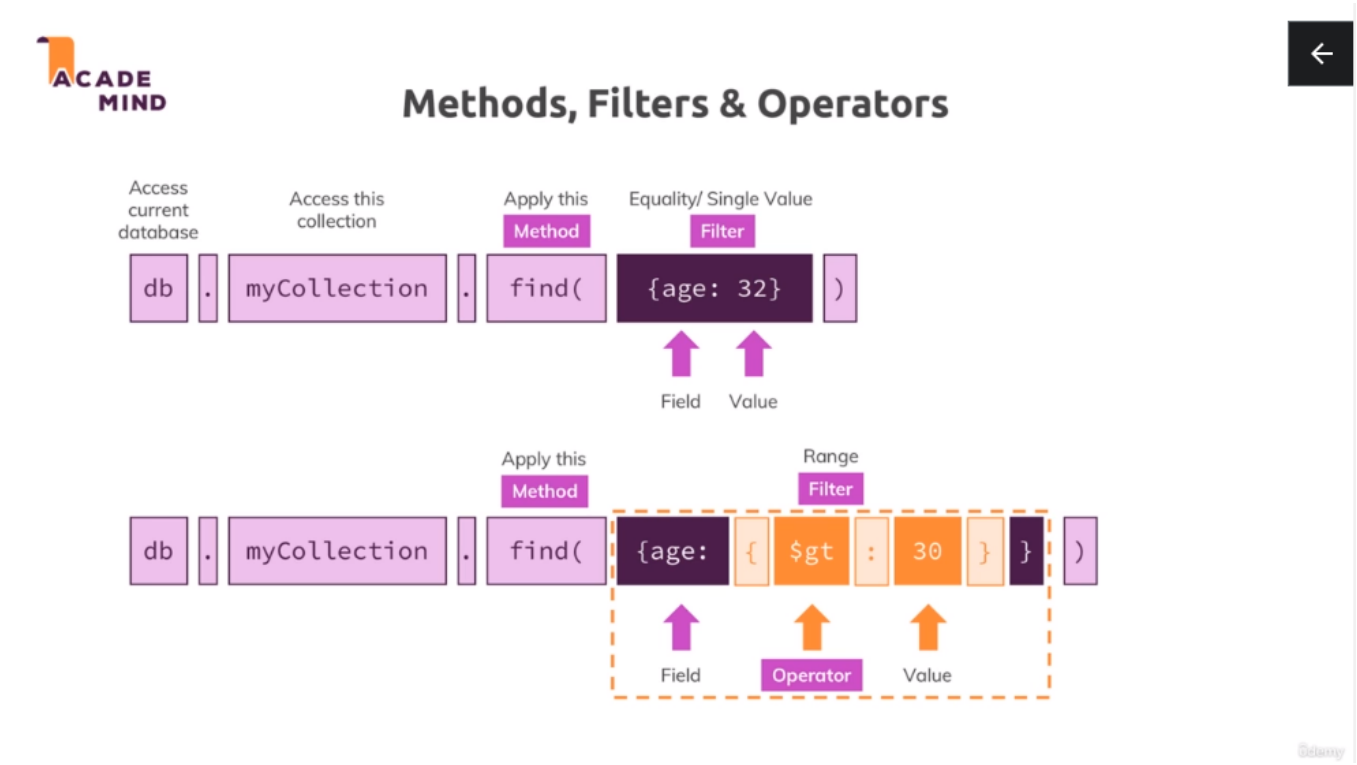
-

```
db.collection_name.insertMany([
        {num: 1},
        {num: 2},
        {num: 3},
        {num: 4},
        {num: 5},
],
        {
                ordered: false
```

```
        }
);
```

Example ->

```
> db.companies.insertMany([{name: "Fresh Beverages Inc", stock: 88, _id: 1}, {name: "Cool Burgers Inc"
, stock: 12, _id: 4}], {ordered: false})
```



- To find all documents in which genres array contain "Drama", there can be other elements in it as well.

```
> db.movies.find({genres: "Drama"}).pretty()
```

- To find all documents in which genres array only consist of "Drama", there cannot be any other elements in it.

```
> db.movies.find({genres: ["Drama"]}).pretty()
```

$or

- Takes an array of conditions and returns all documents which satisfy at least 1 condition (basically or of these conditions).

```
> db.movies.find({$or: [{"rating.average": {$lt: 5}}, {"rating.average": {$gt: 9.3}}]}).pretty()
```

This example returns all the movies which have rating (less than 5) or (greater than 9.3).

$nor

- Just like or but wrap the entire array of condition in ! (not) operator and then returns the documents.

$and

- Returns the documents which satisfy all the conditions. Takes an array of condition.

```
> db.movies.find({$and: [{"rating.average": {$gt: 9}}, {genres: "Drama"}]}).count()
3
> db.movies.find({"rating.average": {$gt: 9}, genres: "Drama"}).count()
3
> db.movies.find({genres: "Drama", genres: "Horror"}).count()
23
> db.movies.find({$and: [{genres: "Drama"}, {genres: "Horror"}]}).count()
17
```

In the above example photo -

- First command gives us count of results with rating > 9 and genres contains "Drama".
- Second command does the same but without $and operator. By default mongoDB works like and operator for the condition.

So why use $and ?
reason -

- In Third command the second condition will overwrite the first condition as they both uses same genres field. So we will get all the movies with genres "Horror".
- In Forth command this issue is solved by using $and operator. This is why we need $ and operator

Read $not form this

Logical

Name	Description
$and	Joins query clauses with a logical **AND** returns all documents that match the conditions of both clauses.
$not	Inverts the effect of a query expression and returns documents that do *not* match the query expression.
$nor	Joins query clauses with a logical **NOR** returns all documents that fail to match both clauses.
$or	Joins query clauses with a logical **OR** returns all documents that match the conditions of either clause.

$exists

- used to apply condition like if a field exists or not
- remember it will return the document even if that the field contains NULL, as it is existing.

```
> db.users.find({age: {$exists: true}}).pretty()
{
        "_id" : ObjectId("5b9f652dc62d13aa2e81c29e"),
        "name" : "Manuel",
        "hobbies" : [
                {
                        "title" : "Cooking",
                        "frequency" : 5
                },
                {
                        "title" : "Cars",
                        "frequency" : 2
                }
        ],
        "phone" : "012177972",
        "age" : 30
}
>
```

To solve the NULL issue we can use $ne along side $exists to put it not equal to NULL.

```
> db.users.find({age: {$exists: true, $ne: null}}).pretty()
{
        "_id" : ObjectId("5b9f652dc62d13aa2e81c29e"),
        "name" : "Manuel",
        "hobbies" : [
                {
                        "title" : "Cooking",
                        "frequency" : 5
                },
                {
                        "title" : "Cars",
                        "frequency" : 2
                }
        ],
        "phone" : "012177972",
        "age" : 30
}
```

$type

- checks if a field is of a particular type.

```
> db.users.find({phone: {$type: "number"}}).pretty()
{
        "_id" : ObjectId("5b9f652dc62d13aa2e81c29d"),
        "name" : "Max",
        "hobbies" : [
                {
                        "title" : "Sports",
                        "frequency" : 3
                },
                {
                        "title" : "Cooking",
                        "frequency" : 6
                }
        ],
        "phone" : 131782734
}
```

Returns all documents where phone field contains datatype of number.

$expr

This is used to get documents based on the evaluation of an expression

```
> db.sales.find({$expr: {$gt: ["volume", "target"]}}).pretty()
{
        "_id" : ObjectId("5b9f6815c62d13aa2e81c2a0"),
        "volume" : 100,
        "target" : 120
}
{
        "_id" : ObjectId("5b9f6815c62d13aa2e81c2a1"),
        "volume" : 89,
        "target" : 80
}
{
        "_id" : ObjectId("5b9f6815c62d13aa2e81c2a2"),
        "volume" : 200,
        "target" : 177
}
> db.sales.find({$expr: {$gt: ["$volume", "$target"]}}).pretty()
{
        "_id" : ObjectId("5b9f6815c62d13aa2e81c2a1"),
        "volume" : 89,
        "target" : 80
}
{
        "_id" : ObjectId("5b9f6815c62d13aa2e81c2a2"),
        "volume" : 200,
        "target" : 177
}
```

$size

```
> db.users.find({hobbies: {$size: 3}}).pretty()
{
        "_id" : ObjectId("5b9f707ead7ae74ebe5bc6c7"),
        "name" : "Chris",
        "hobbies" : [
                "Sports",
                "Cooking",
                "Hiking"
        ]
}
```

$all

- Used to find documents whose array data member contains certain values (it can contain more values along side our mentioned values as well)

In the below example we are trying to find all document who contains "action" & "thriller" their are 2, But there is an issue - it return only 1 document because here in this command order of elements matters

This matches -> ["action", "thriller"]
This doesn't -> ["thriller", "action"]

```
> db.moviestarts.find({genre: ["action", "thriller"]}).pretty()
{
        "_id" : ObjectId("5b9f6d1651dfa62353d12f04"),
        "title" : "Teach me if you can",
        "meta" : {
                "rating" : 8.5,
                "aired" : 2014,
                "runtime" : 90
        },
        "visitors" : 590378,
        "expectedVisitors" : 500000,
        "genre" : [
                "action",
                "thriller"
        ]
}
```

Using $all ->

```
> db.moviestarts.find({genre: {$all: ["action", "thriller"]}}).pretty()
```

*Both matches -> ["action", "thriller"], ["thriller", "action"]

$elemMatch

Used to match elements in array in MongoDB document, perticularly useful when it's a nested object or document

```
> db.users.find({$and: [{"hobbies.title": "Sports"}, {"hobbies.frequency": {$gte: 3}}]}).count()
```

This will return all documents where and object in hobbies has a title of sports and any object in hobbies has frequency >= 3. No need for them to be a same object

But if we want them to be a same object the we can use elemMatch ->

```
> db.users.find({hobbies: {$elemMatch: {title: "Sports", frequency: {$gte: 3}}}}).pretty()
{
        "_id" : ObjectId("5b9f652dc62d13aa2e81c29d"),
        "name" : "Max",
        "hobbies" : [
                {
                        "title" : "Sports",
                        "frequency" : 3
                },
                {
                        "title" : "Cooking",
                        "frequency" : 6
                }
        ],
        "phone" : 131782734
}
```

Cursor methods

Remember .find() returns a cursor instead of array

- **.sort()**
  Takes an object with consists of fields by whom to sort by and then order. (1 -> asc, -1 -> desc)

there can be multiple fields in sort function and if their is a match then it moves to second field to sort by it for those matching documents.

NOTE -> even in one sort function different fields can have different sorting order (asc or desc)

```
> db.movies.find().sort({"rating.average": 1, runtime: -1})
```

- **.skip()**
  Takes a number as argument and skips that many documents from start

```
> db.movies.find().sort({"rating.average": 1, runtime: -1}).sort({"rating.average": 1}).skip(10).prett
y()
```

- **.limit()**
  Takes in a number as an argument. Limits how much documents the cursor fetches in a single turn.

```
> db.movies.find().sort({"rating.average": 1, runtime: -1}).skip(100).limit(10).count()
```

Projections

These allow us to model the output data and structure it as we want to. It also works on embedded documents

```
> db.movies.find({}, {name: 1, genres: 1, runtime: 1, "rating.average": 1, "schedule.time": 1, _id: 0}
).pretty()
```

Original Document is quite long and complex.

Projected document ->


```
{
        "name" : "The Last Ship",
        "genres" : [
                "Drama",
                "Action",
                "Thriller"
        ],
        "runtime" : 60,
        "schedule" : {
                "time" : "21:00"
        },
        "rating" : {
                "average" : 7.8
        }
}
```


Update

$min -> Takes in a number argument and assign the min(newVal, currVal) value to the specified field.

$max -> Takes in a number argument and assign the max(newVal, currVal) value to the specified field.

In below example currVal is 35 for the age in document

```
> db.users.updateOne({name: "Chris"}, {$max: {age: 32}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 0 }
> db.users.updateOne({name: "Chris"}, {$max: {age: 38}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

$inc -> increment specified field by given value

$mul -> multiplies specified field by given value

$unset -> remove a field from documents

```
> db.users.updateMany({isSporty: true}, {$unset: {phone: ""}})
{ "acknowledged" : true, "matchedCount" : 3, "modifiedCount" : 3 }
```

$rename -> renames a field from the document

```
> db.users.updateMany({}, {$rename: {age: "totalAge"}})
{ "acknowledged" : true, "matchedCount" : 4, "modifiedCount" : 3 }
```

$upsert -> if no matching document is found add a new document with what information is available.

```
> db.users.updateOne({name: "Maria"}, {$set: {age: 29, hobbies: [{title: "Good food", frequency: 3}],
isSporty: true}})
{ "acknowledged" : true, "matchedCount" : 0, "modifiedCount" : 0 }
> db.users.updateOne({name: "Maria"}, {$set: {age: 29, hobbies: [{title: "Good food", frequency: 3}],
isSporty: true}}, {upsert: true})
{
        "acknowledged" : true,
        "matchedCount" : 0,
        "modifiedCount" : 0,
        "upsertedId" : ObjectId("5b9fb59d51dfa62353d131f4")
}
```

Playing with array element update

```
> db.users.updateMany({hobbies: {$elemMatch: {title: "Sports", frequency: {$gte: 3}}}}, {$set: {"hobbi
es.$.highFrequency": true}})
```

Here $ is the matched element placeholder.

Here $[] is placeholder for all array elements and just one.

```
> db.users.updateMany({"hobbies.frequency": {$gt: 2}}, {$set: {"hobbies.$[el].goodFrequency": true}},
{arrayFilters: [{"el.frequency": {$gt: 2}}]})
{ "acknowledged" : true, "matchedCount" : 5, "modifiedCount" : 5 }
```

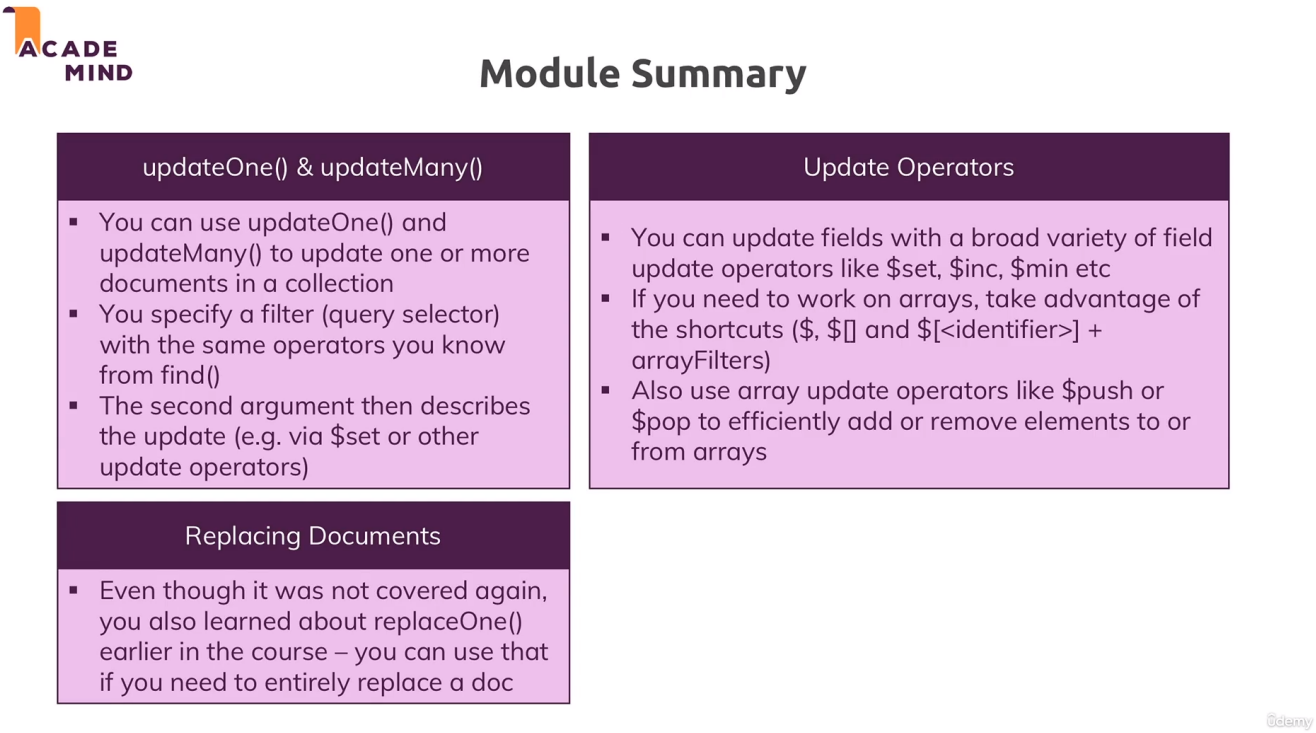
```
db.collection_name.updateMany(
        {filter_for_documents},
        {
                $set:
```

```
                        {"array_name.$[notation_for_single_element].
<some_key>": newValue}
        },
        {
                "notation_for_single_element.
<some_other_key_for_filtering>":
                        value_or_expression
        }
)
```

Pushing a new element into array via update

```
// For inserting single element

db.collection_name.updateOne(
        {filter_for_documents},
        {
                $push: {
                        name_of_array_in_document: new_element
                }
        }
)

// For inserting multiple Elements

db.collection_name.updateOne(
        {filter_for_documents},
        {
                $push: {
                        name_of_array_in_document: {
                                $each: [ new_element_1, new_element_2, ...]
                        }
                }
        }
)
```

```
> db.users.updateOne({name: "Maria"}, {$push: {hobbies: {title: "Sports", frequency: 2}}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```



Indexes

When we use indexing for a particular field we store all the values of that field from all documents in a separate place in sorted order where each value also has a pointer to it's original document in memory for retrieval if required.

Index can be based on just one field -> simple indexing
Or on multiple fields together -> compound indexing

Indexing is efficient if queries only fetch a fraction of documents from a collection.
It is slow if we fetch a large amount of data.

```
// Single field indexing
db.collection_name.createIndex({field_name: 1});

// Compound indexing
db.collection_name.createIndex({field_name_1: 1, field_name_2: 1});

// 1 for asc order, -1 for desc order
```

Covered Queries

When we only need to return the fields that are covered by the indexes (no need to return _id as well) the after we find the index we don't find the document in memory so required data is directly returned from the indexed fields.

Multikey indexes

If we index an array in a document the all the values of that array are pulled out and inserted in index storage individually.

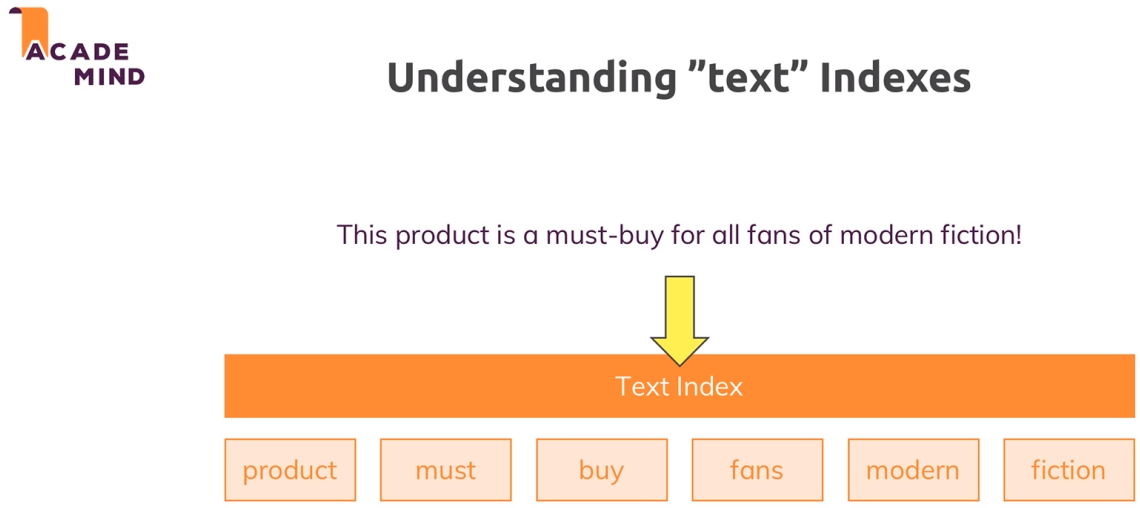
For example if we have 1000 documents each containing the array of say hobbies of size 5 for each then we will have 5000 entries in our hobbies index. Each entry will point to it's original document

we and combine normal field and multi-key index in compound indexing. But not 2 or more multi-key indexes it will give us an error.

```
// Example Document
{
        name: String,
        hobbies: Array,
        addresses: Array
}
```

Will work -> db.users.createIndex({ name: 1, hobbies: 1});
Will not work -> db.users.createIndex({ hobbies: 1, addresses: 1});

Text index



In this type of index the indexing is done on a text field which contains many words like a sentence, when the index is created on it using the "text" attribute then all the words of the particular field are separated and stopwords are removed the n each word is inserted in index storage where they point to their original storage.

This makes searching all the documents containing that word in that particular field quite easy and efficient then using regex.

```
> db.products.find().pretty()
{
        "_id" : ObjectId("5b9cbab3b96c9724db7f7b9c"),
        "title" : "A Book",
        "description" : "This is an awesome book about a young artist!"
}
{
        "_id" : ObjectId("5b9cbab3b96c9724db7f7b9d"),
        "title" : "Red T-Shirt",
        "description" : "This T-Shirt is red and it's pretty awesome!"
}
> db.products.createIndex({description: 1})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
> db.products.dropIndex({description: 1})
{ "nIndexesWas" : 2, "ok" : 1 }
```

I the above image the second command that creates the index on description field will create the index based on whole string as a whole and not individual words so this is the wrong way for this example.

```
> db.products.createIndex({description: "text"})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
```

Correct way

Remember that text indexing is quite expensive. So use it just for one field.

Here we didn't specified that it has to find text from description field

```
> db.products.find({$text: {$search: "awesome"}}).pretty()
{
        "_id" : ObjectId("5b9cbab3b96c9724db7f7b9c"),
        "title" : "A Book",
        "description" : "This is an awesome book about a young artist!"
}
{
        "_id" : ObjectId("5b9cbab3b96c9724db7f7b9d"),
        "title" : "Red T-Shirt",
        "description" : "This T-Shirt is red and it's pretty awesome!"
}
```

```
> db.products.find({$text: {$search: "\"red book\""}}).pretty()
> db.products.find({$text: {$search: "\"awesome book\""}}).pretty()
{
        "_id" : ObjectId("5b9cbab3b96c9724db7f7b9c"),
        "title" : "A Book",
        "description" : "This is an awesome book about a young artist!"
}
```

MongoDB also gives the searches a score.

```
> db.products.find({$text: {$search: "awesome t-shirt"}}).pretty()
{
        "_id" : ObjectId("5b9cbab3b96c9724db7f7b9c"),
        "title" : "A Book",
        "description" : "This is an awesome book about a young artist!"
}
{
        "_id" : ObjectId("5b9cbab3b96c9724db7f7b9d"),
        "title" : "Red T-Shirt",
        "description" : "This T-Shirt is red and it's pretty awesome!"
}
> db.products.find({$text: {$search: "awesome t-shirt"}}, {score: {$meta: "textScore"}}).pretty()
{
        "_id" : ObjectId("5b9cbab3b96c9724db7f7b9d"),
        "title" : "Red T-Shirt",
        "description" : "This T-Shirt is red and it's pretty awesome!",
        "score" : 1.7999999999999998
}
{
        "_id" : ObjectId("5b9cbab3b96c9724db7f7b9c"),
        "title" : "A Book",
        "description" : "This is an awesome book about a young artist!",
        "score" : 0.625
}
```

Creating text index for 2 fields combined

```
> db.products.getIndexes()
[
        {
                "v" : 2,
                "key" : {
                        "_id" : 1
                },
                "name" : "_id_",
                "ns" : "test.products"
        }
]
> db.products.createIndex({title: "text", description: "text"})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
```

We can even assign different languages to indexes and different weight to multiple fields for text indexing.

```
> db.products.createIndex({title: "text", description: "text"}, {default_language: "english", weights:
 {title: 1, description: 10}})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
```

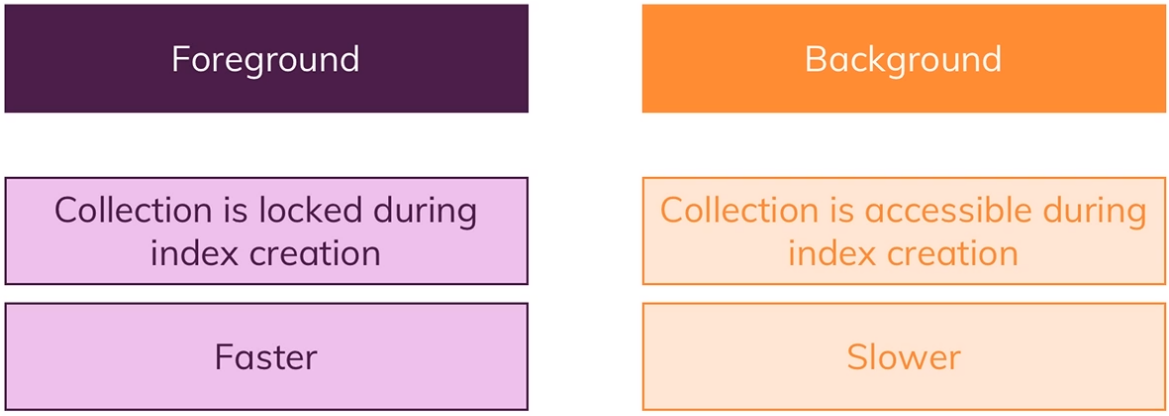
Partial Indexes

Indexes on a field that are made only on fulfilment of a condition on that field.

```
db.restaurants.createIndex(
   { cuisine: 1, name: 1 },
   { partialFilterExpression: { rating: { $gt: 5 } } }
)
```

Building Indexes

Building Indexes



By default index are created as a foreground process but if we want we can make it a background process.

```
> db.ratings.createIndex({age: 1}, {background: true})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
```

Module Summary

What and Why?

- Indexes allow you to retrieve data more efficiently (if used correctly) because your queries only have to look at a subset of all documents
- You can use single-field, compound, multi-key (array) and text indexes
- Indexes don't come for free, they will slow down your writes

Queries & Sorting

- Indexes can be used for both queries and efficient sorting
- Compound indexes can be used as a whole or in a "left-to-right" (prefix) manner (e.g. only consider the "name" of the "name-age" compound index)

Query Diagnosis & Planning

- Use explain() to understand how MongoDB will execute your queries
- This allows you to optimize both your queries and indexes

Index Options

- You can also create TTL, unique or partial indexes
- For text indexes, weights and a default_language can be assigned

Geospacial Data

Query ->

```
> use awesomeplaces
switched to db awesomeplaces
> db.places.insertOne({name: "Califorina Academy of Sciences", location: {type: "Point", coordinates:
[-122.4724356, 37.7672544]}})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("5b9ce80359380a287a267865")
}
```

```
db.collection_name.insertOne({

        field_name_1: value_1,
        field_name_2: value_2,
        ...,
        our_geo_field_name: {
                type: select_from_predifined_values_in_geojson,
                coordinates: [
                        longitude_value,
                        latitude_value
                ]
        }

})
```

"our_geo_field_name" can be named anything like location etc. however it's structure is fixed like specified above.

NOTE -> For working with GeoJSON data we need a geospatial index on the field that contains geojson data in each document in the collections

Here "location" was the field name, it can be any name that our geojson field has.

```
> db.places.createIndex({location: "2dsphere"})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
```

$near ->

```
> db.places.find({location: {$near: {$geometry: {type: "Point", coordinates: [-122.471114, 37.771104]}
}}}).pretty()
{
        "_id" : ObjectId("5b9ce80359380a287a267865"),
        "name" : "Califorina Academy of Sciences",
        "location" : {
                "type" : "Point",
                "coordinates" : [
                        -122.4724356,
                        37.7672544
                ]
        }
}
```

$maxDistance & $minDistance ->

```
> db.places.find({location: {$near: {$geometry: {type: "Point", coordinates: [-122.471114, 37.771104]}
, $maxDistance: 500, $minDistance: 10}}}).pretty()
{
        "_id" : ObjectId("5b9ce80359380a287a267865"),
        "name" : "Califorina Academy of Sciences",
        "location" : {
                "type" : "Point",
                "coordinates" : [
                        -122.4724356,
                        37.7672544
                ]
        }
}
```

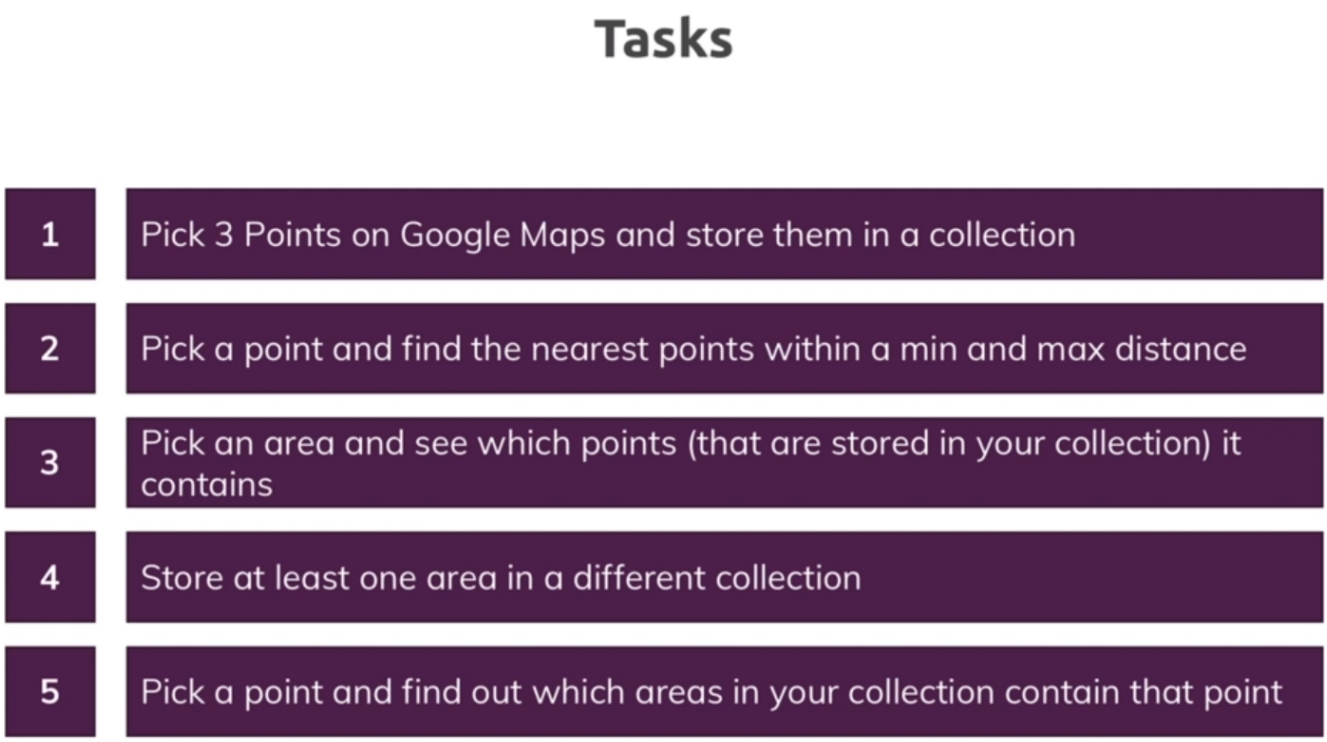
$geoWithin ->

```
> db.places.find({location: {$geoWithin: {$geometry: {type: "Polygon", coordinates: [[p1, p2, p3, p4,
p1]]}}}})
```

$geoIntersects ->

```
> db.areas.find({area: {$geoIntersects: {$geometry: {type: "Point", coordinates: [-122.48446, 37.77776
]}}}}).pretty()
```

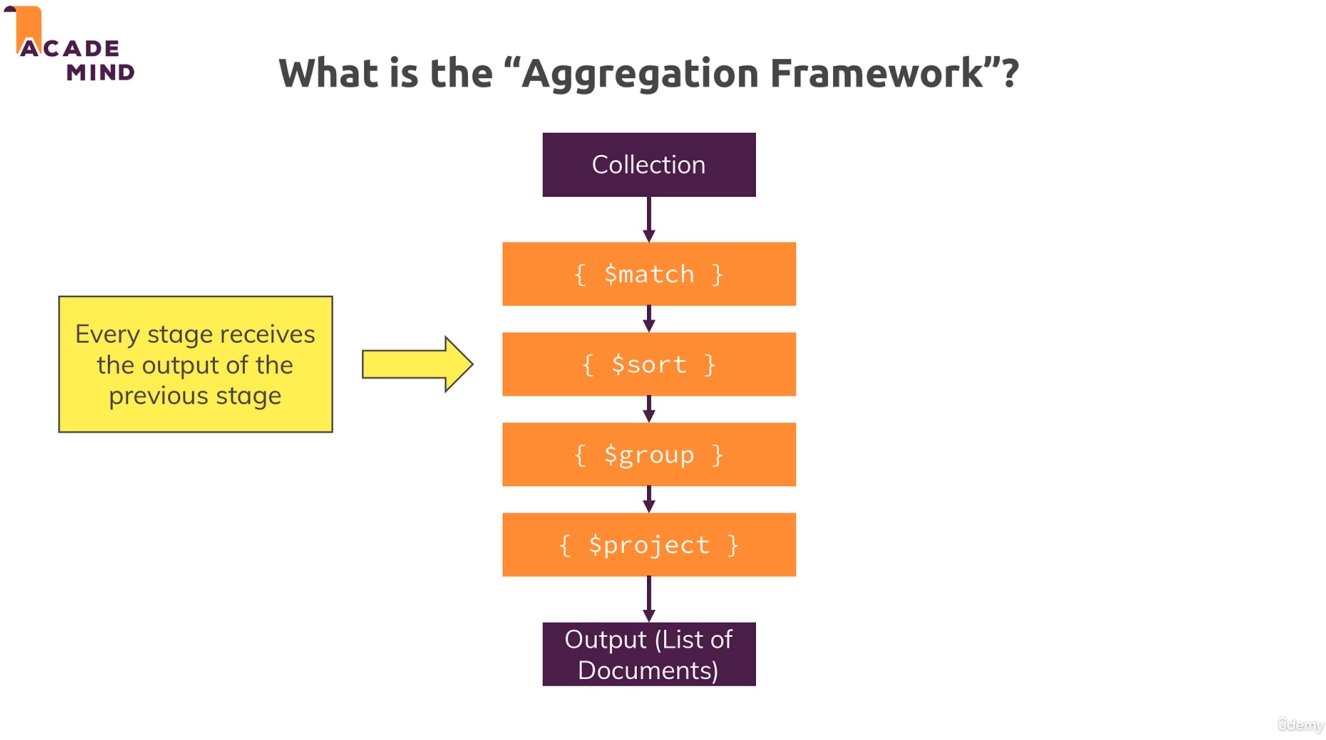
For Practices ->




```
// Answer - 1 -

db.places.insertOne({
        "name": "bcm",
        "location": {
                "type": "Point",
                "coordinates": [ 75.83376732971888, 30.88934486739099 ]
        }
});
// similarly of gtbh, gne


// Answer - 2 -

db.places.find({
        location: {
                $near: {
                        $geometry: {
                                type: "Point",
                                coordinates: [ 75.83584885928511,
30.8883195853984]
```

```
                        },
                        $maxDistance: 3000,
                        $minDistance: 100
                }
        }
}).pretty();

// Rest of the part done in mind
```

Aggregation Framework



Query 1 ->

```
db.persons.aggregate([
        { $match: { gender: 'female'} },
        { $group: { _id: { state: "$location.state" }, totalPersons: { $sum:
1 } } }
]).pretty();
```

Output ->

Query 2 ->

```
db.persons.aggregate([
        { $match: { gender: 'female'} },
        { $group: { _id: { state: "$location.state" }, totalPersons: { $sum:
1 } } },
        { $sort: { totalPersons: -1 } }
]).pretty();
```

Output ->

```
[
  { _id: { state: 'midtjylland' }, totalPersons: 33 },
  { _id: { state: 'nordjylland' }, totalPersons: 27 },
  { _id: { state: 'new south wales' }, totalPersons: 24 },
  { _id: { state: 'syddanmark' }, totalPersons: 24 },
  { _id: { state: 'australian capital territory' }, totalPersons: 24 },
  { _id: { state: 'south australia' }, totalPersons: 22 },
  { _id: { state: 'hovedstaden' }, totalPersons: 21 },
  { _id: { state: 'danmark' }, totalPersons: 21 },
  { _id: { state: 'queensland' }, totalPersons: 20 },
  { _id: { state: 'overijssel' }, totalPersons: 20 },
  { _id: { state: 'sjælland' }, totalPersons: 19 },
  { _id: { state: 'nova scotia' }, totalPersons: 17 },
  { _id: { state: 'canterbury' }, totalPersons: 16 },
  { _id: { state: 'yukon' }, totalPersons: 16 },
  { _id: { state: 'northwest territories' }, totalPersons: 16 },
  { _id: { state: 'gelderland' }, totalPersons: 16 },
  { _id: { state: 'bayern' }, totalPersons: 15 },
  { _id: { state: 'northern territory' }, totalPersons: 15 },
  { _id: { state: 'tasmania' }, totalPersons: 15 },
  { _id: { state: 'victoria' }, totalPersons: 14 }
]
Type "it" for more
```

```
// Question - create aggregate function pipeline for below features
// older than 50
// group by gender, num of each gender, avg age
// order by total person in that gender

db.persons.aggregate([

        { $match: { "dob.age": { $gt: 50 } } },
        { $group: { _id: { gender: "$gender" }, totalPeople: { $sum: 1 },
avgAge: { $avg: "$dob.age"}} },
        { $sort: { totalPeople: 1 } }
]).pretty();
```

Output ->

$project

It is the forth and final stage of aggregation framework and is quite similar to projection in find().

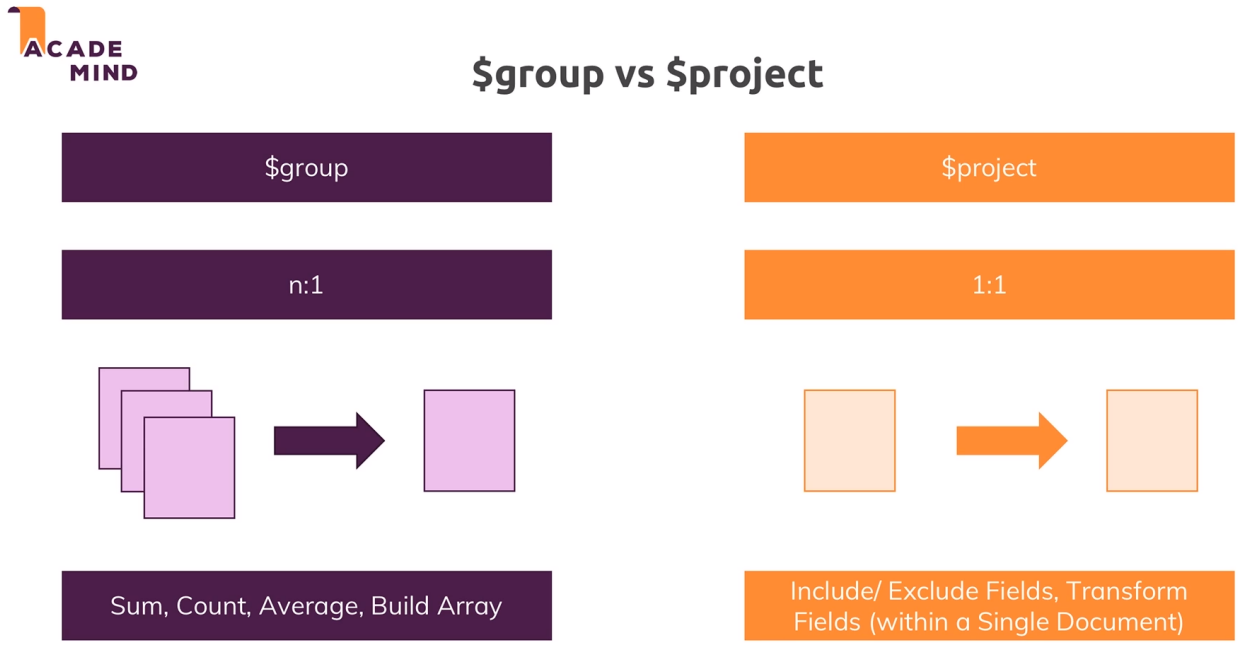
```
> db.persons.aggregate([
...     { $project: { _id: 0, gender: 1, fullName: { $concat: ["Hello", "World"] } } }
... ]).pretty();
{ "gender" : "male", "fullName" : "HelloWorld" }
{ "gender" : "female", "fullName" : "HelloWorld" }
{ "gender" : "male", "fullName" : "HelloWorld" }
{ "gender" : "male", "fullName" : "HelloWorld" }
{ "gender" : "female", "fullName" : "HelloWorld" }
{ "gender" : "male", "fullName" : "HelloWorld" }
{ "gender" : "male", "fullName" : "HelloWorld" }
{ "gender" : "female", "fullName" : "HelloWorld" }
{ "gender" : "female", "fullName" : "HelloWorld" }
{ "gender" : "female", "fullName" : "HelloWorld" }
{ "gender" : "female", "fullName" : "HelloWorld" }
{ "gender" : "female", "fullName" : "HelloWorld" }
{ "gender" : "female", "fullName" : "HelloWorld" }
{ "gender" : "female", "fullName" : "HelloWorld" }
{ "gender" : "male", "fullName" : "HelloWorld" }
{ "gender" : "male", "fullName" : "HelloWorld" }
{ "gender" : "female", "fullName" : "HelloWorld" }
{ "gender" : "female", "fullName" : "HelloWorld" }
{ "gender" : "female", "fullName" : "HelloWorld" }
{ "gender" : "male", "fullName" : "HelloWorld" }
Type "it" for more
```

```
$project: {
  _id: 0,
  gender: 1,
  fullName: {
    $concat: [
      { $toUpper: { $substrCP: ['$name.first', 0, 1] } },
      {
        $substrCP: [
          '$name.first',
          1,
          { $subtract: [{ $strLenCP: '$name.first' }, 1] }
        ]
      },
      ' ',
      { $toUpper: '$name.last' }
    ]
  }
}
```

Note -> we can have multiple aggregation stages in any order.

```
db.persons.aggregate([
        {
                $project: {
                        _id: 0,
                        name: 1,
                        email: 1,
                        birthdate: { $convert: { input: "$dob.date", to:
'date' } },
                        age: "$dob.age",
                        location: {
                                type: "Point",
                                coordinates: [
                                        {
                                                $convert: {
                                                        input:
"$location.coordinates.longitude",
                                                        to: 'double',
```

```
                                                onError: 0.0,
                                                onNull: 0.0
                                        }
                                },
                                {
                                        $convert: {
                                                input:
"$location.coordinates.latitude",
                                                to: 'double',
                                                onError: 0.0,
                                                onNull: 0.0
                                        }
                                }

                        ]
                }
        }
        }
]).pretty()
```




```
db.friends.aggregate([
        { $unwind: "$hobbies"},
        { $group: { _id: { age: "$age" }, allHobbies: { $addToSet:
"$hobbies" } } }
```

```
]).pretty()
```

$filter ->


```
db.friends.aggregate([
        { $unwind: "$examScores" },
        { $project: { _id: 1, name: 1, hobbies: 1, age: 1, examScores:
"$examScores.score" } },
        { $sort: { examScores: -1 } },
        { $group: { _id: { id: "$_id", name: "$name" }, maxScore: { $max:
"$examScores"} } },
        { $sort: { maxScore: -1 } }
]).pretty()
```

$bucket

```
db.persons.aggregate([
    {
      $bucket: {
        groupBy: '$dob.age',
        boundaries: [18, 30, 40, 50, 60, 120],
        output: {
            numPersons: { $sum: 1 },
            averageAge: { $avg: "$dob.age" }
        }
      }
    }
  ]).pretty();
```

$bucketAuto

```
db.persons.aggregate([
    {
      $bucketAuto: {
        groupBy: '$dob.age',
        buckets: 5,
        output: {
          numPersons: { $sum: 1 },
          averageAge: { $avg: '$dob.age' }
        }
      }
    }
  ]).pretty();
```

$skip & $limit

To get 10 oldest peoples in collection

```
db.persons.aggregate([
    { $project: { _id: 0, name: 1, birthdate: { $toDate: "$dob.date" } } },
    { $sort: { birthdate: 1 } },
    { $limit: 10 }
 ]).pretty();
```

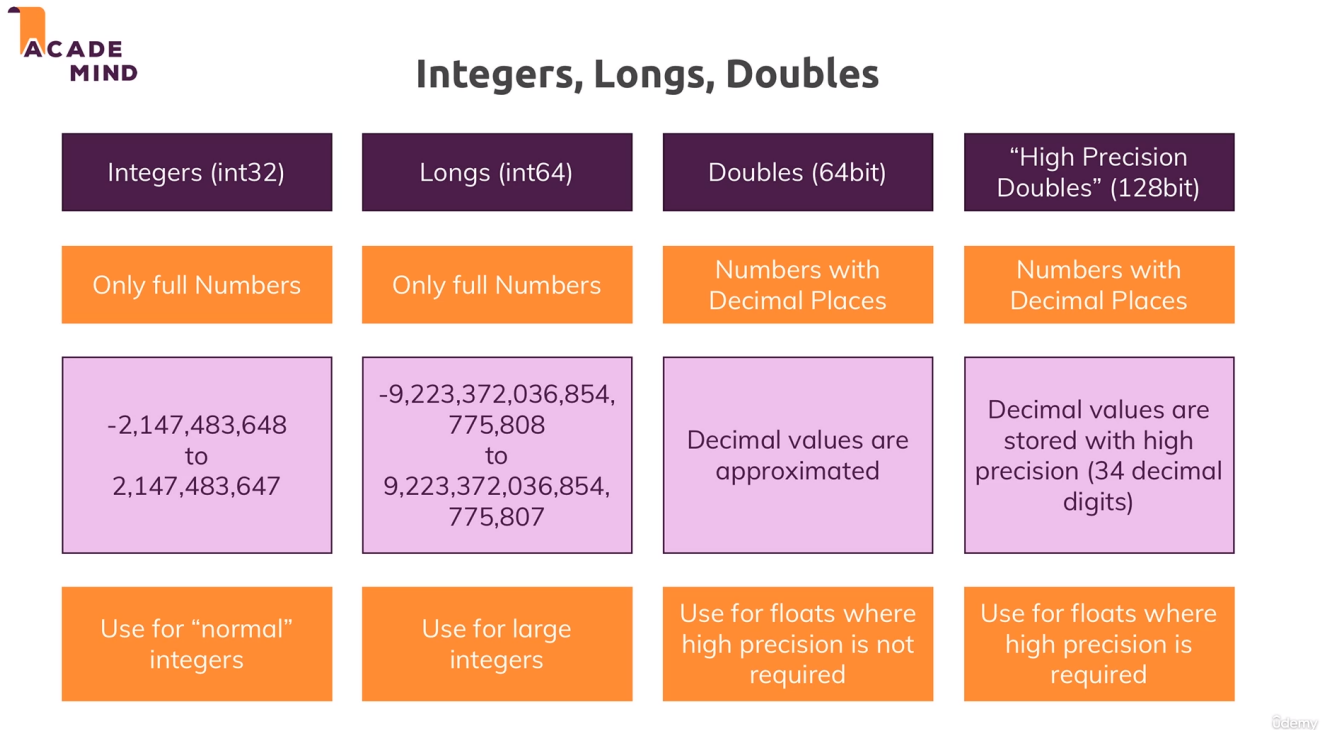
To get next 10

```
db.persons.aggregate([
    { $project: { _id: 0, name: { $concat: ["$name.first", " ", "$name.last"] }, birthdate: { $toDa
    { $sort: { birthdate: 1 } },
    { $skip: 10 },
    { $limit: 10 }
 ]).pretty();
```

$out

It funnels out the result of current aggregation pipeline to a new collection

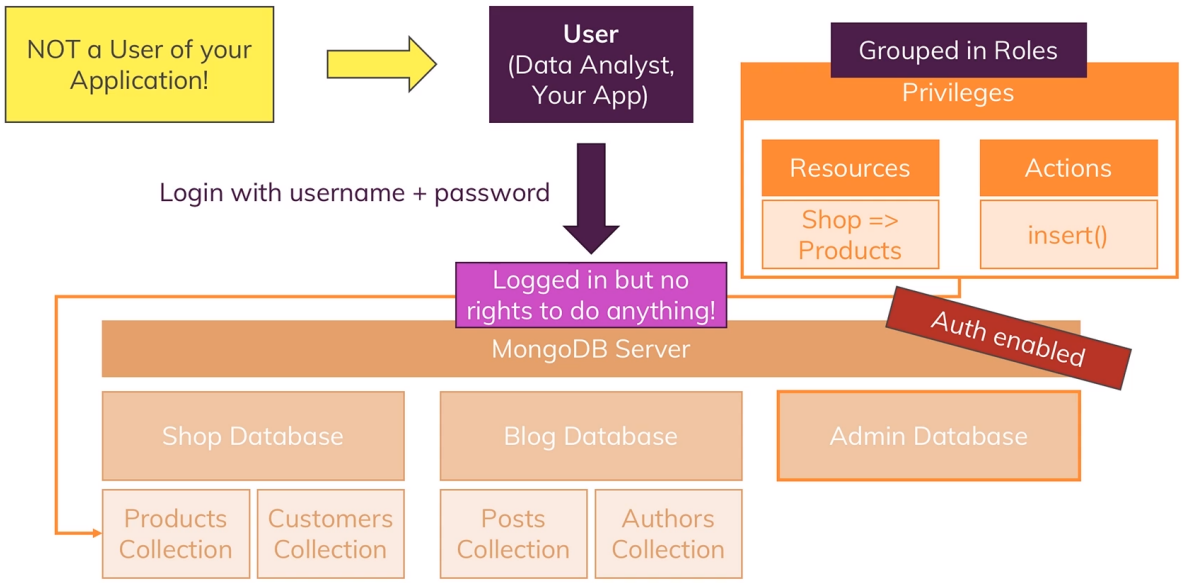
```
...
{ $out: collection_name }
...
```

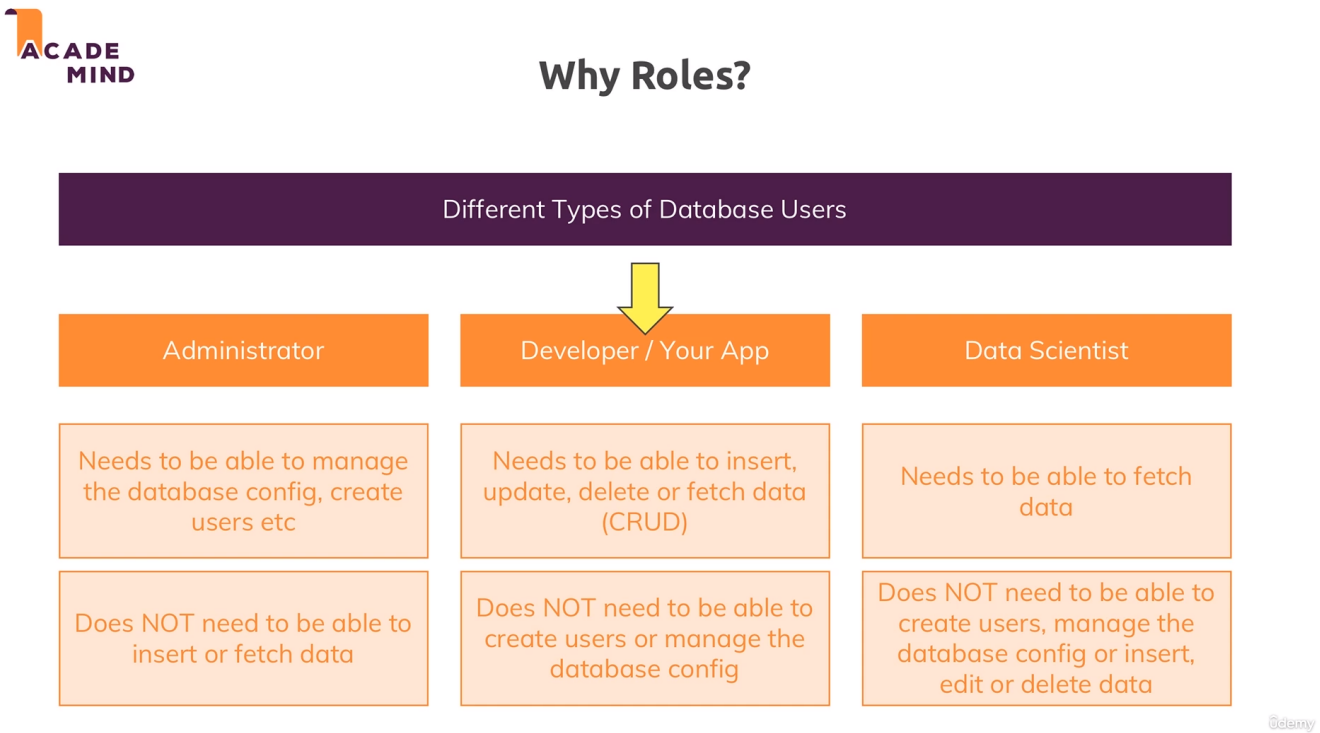


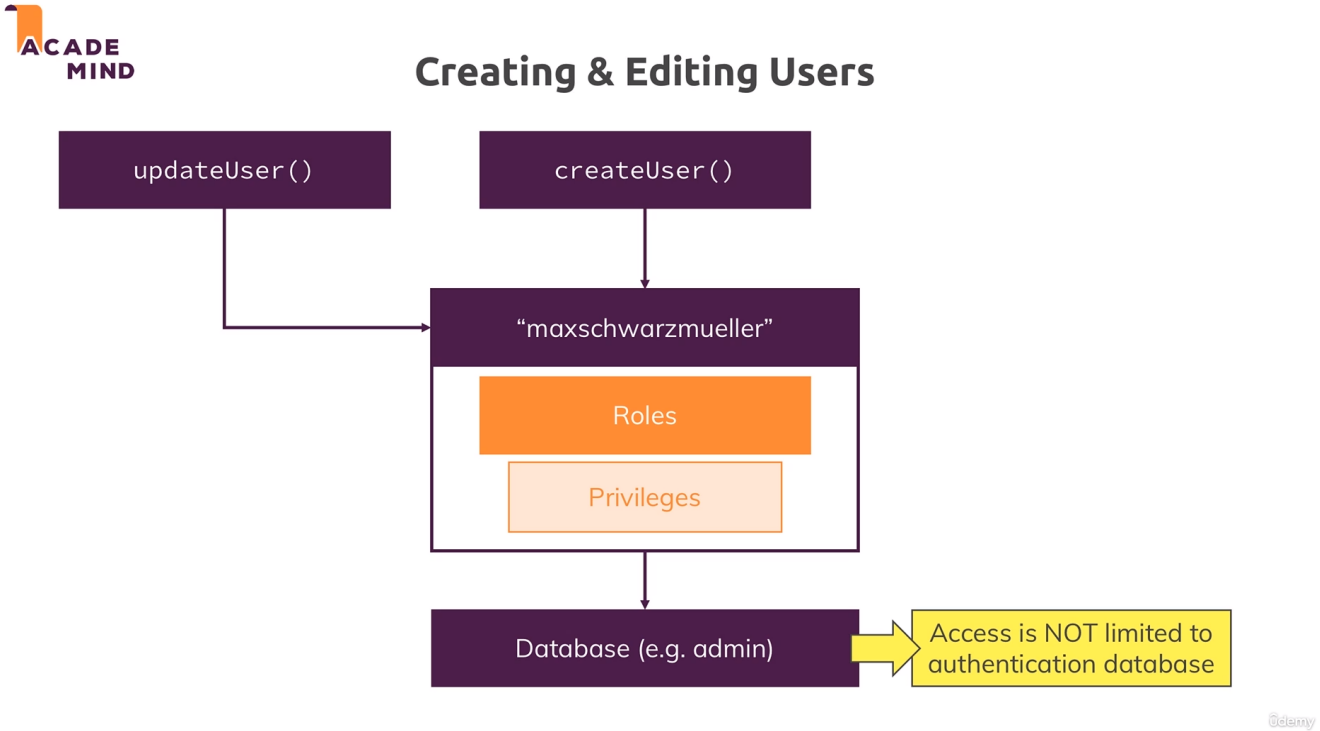
Security Checklist

- Authentication & Authorization
- Transport Encryption
- Encryption at Rest
- Auditing
- Server & Network Config and Setup
- Backups & Software Updates

Role Based Access Control



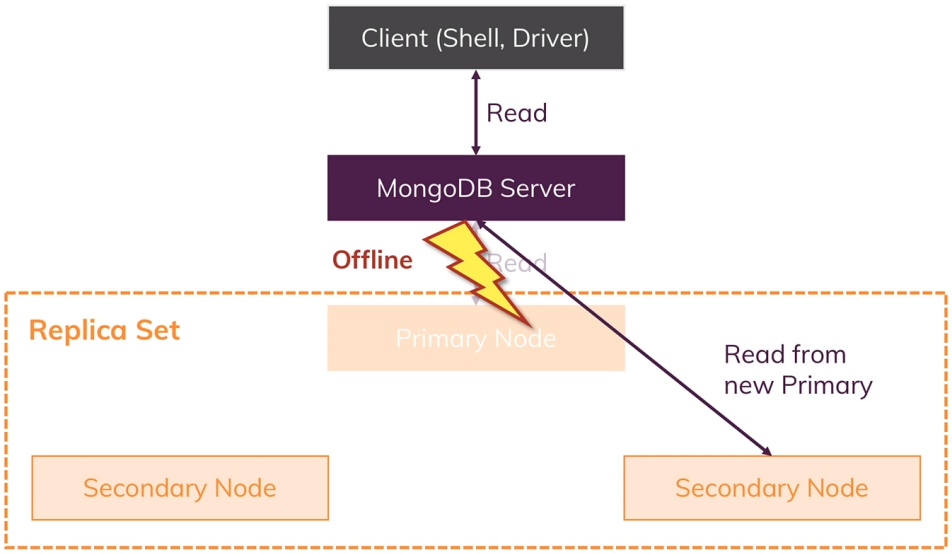

ACADEMIND
Why Roles?
Different Types of Database Users
Administrator
Developer / Your App
Data Scientist
Needs to be able to manage the database config, create users etc
Needs to be able to insert, update, delete or fetch data (CRUD)
Needs to be able to fetch data
Does NOT need to be able to insert or fetch data
Does NOT need to be able to create users or manage the database config
Does NOT need to be able to create users, manage the database config or insert, edit or delete data


ACADEMIND
Creating & Editing Users
updateUser()
createUser()
"maxschwarzmueller"
Roles
Privileges
Database (e.g. admin)
Access is NOT limited to authentication database

Replica Set

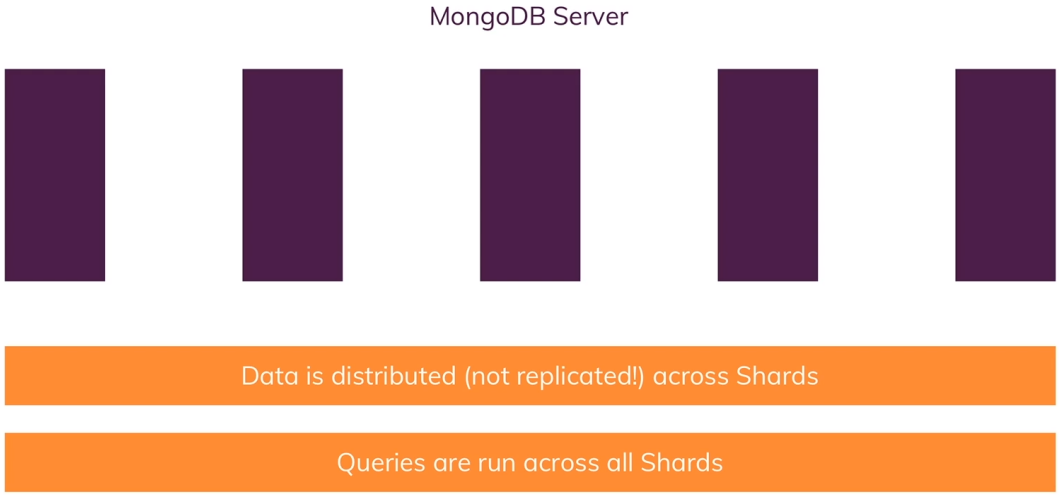
ACADE MIND

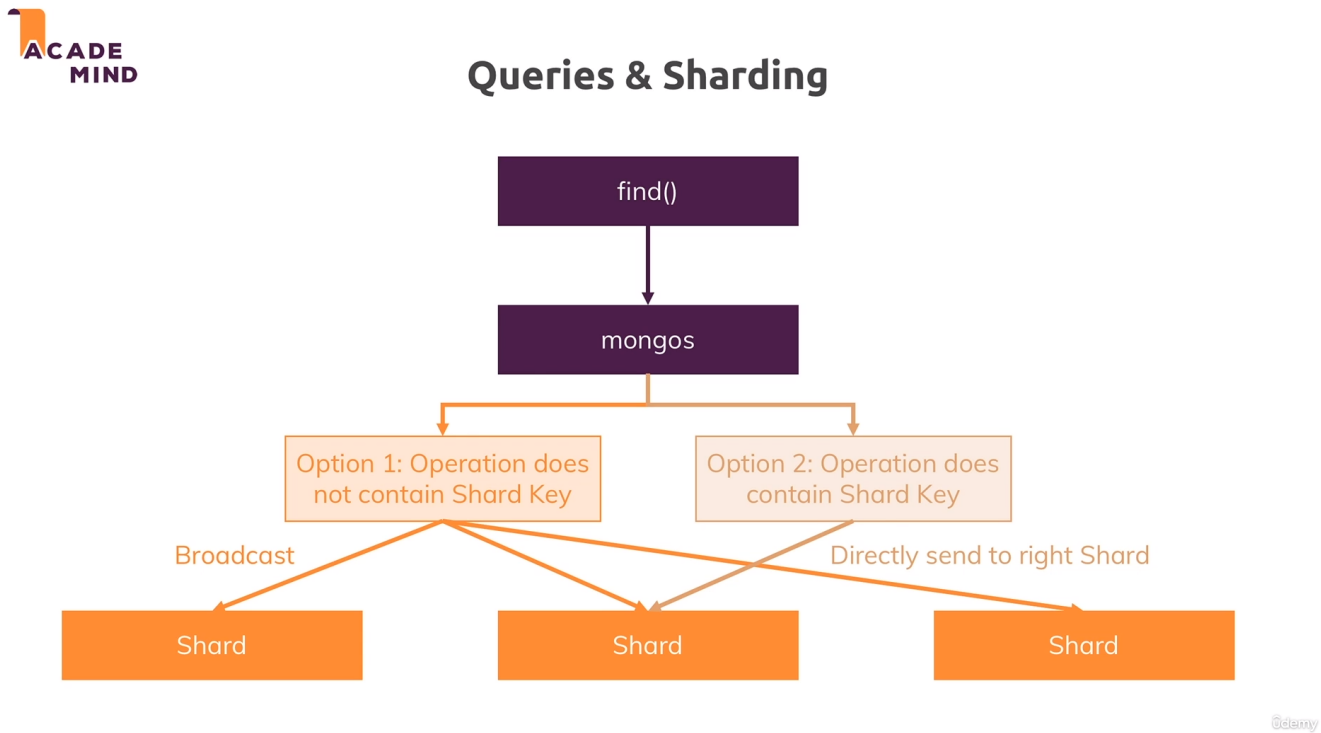
Replica Sets Reads

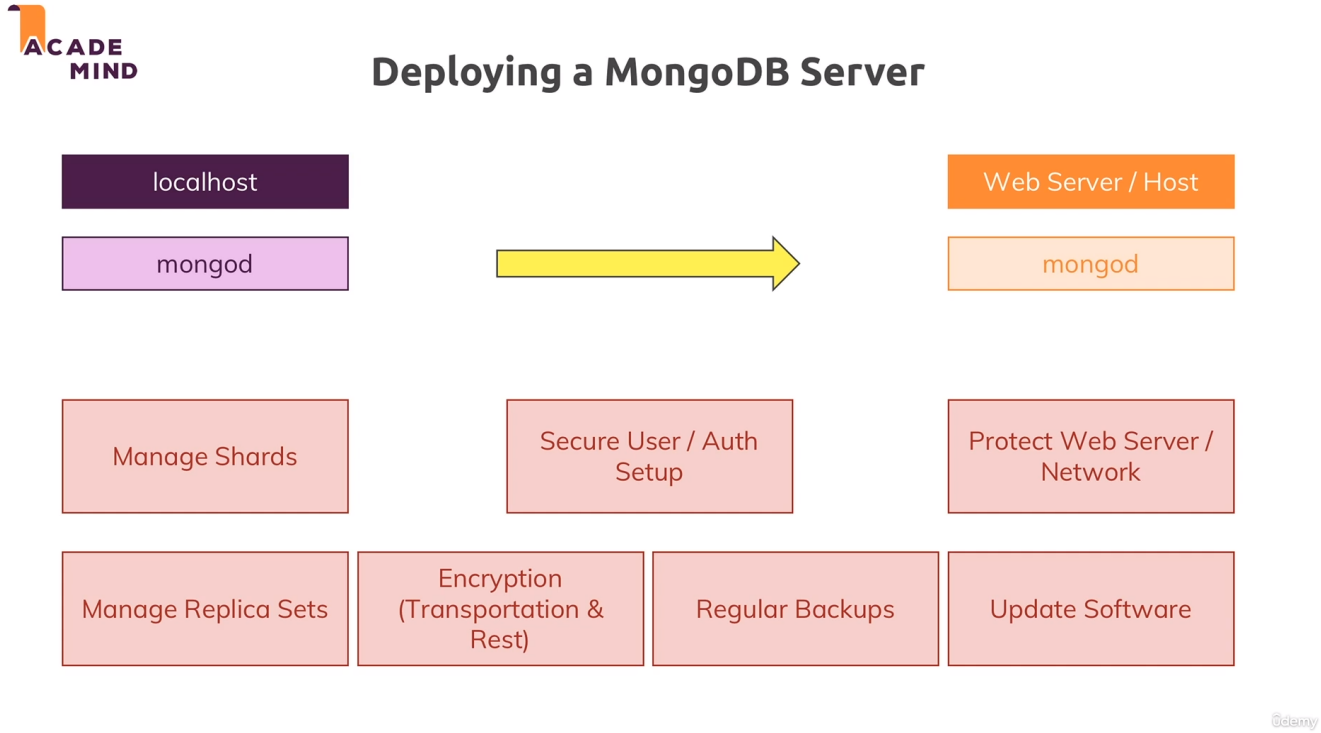


ACADE MIND

Sharding (Horizontal Scaling)



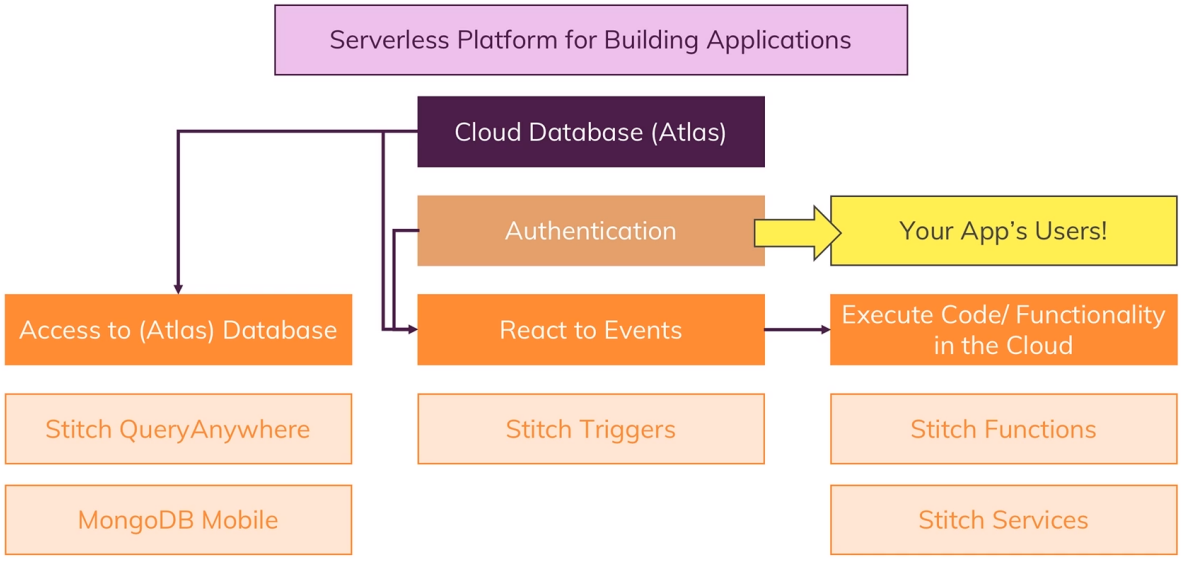




MongoDB Stitch / MongoDB Realms

ACADEMIND

What is Stitch?



ACADEMIND

Serverless?