MongoDB®

Complete Course

COURSE TOPICS

MONDODB

BASICS

- Introduction to MongoDB
- NoSQL Vs SQL
- JSON Vs BSON
- Managing DB & Collections
- Advanced CRUD Operations
- Comparison Operators
- Cursors in MongoDB
- Logical Operators
- $expr & Elements Operator
- Projection & Relationship
- Embedded Documents

ADVANCED

- Introduction to Indexes
- Creating and Managing Index
- Understanding the Aggregation Framework
  - Introduction to Aggregation
  - Basic Aggregation Operations
  - Combining Aggregation Stages
  - Aggregation Operators and Expressions
- Pipeline Stages
  - ($match, $project, $group, $sort,
    $limit, $unwind, $filter, $skip etc)

PROJECTS

Project 1:

Working with MongoDB Node.js Driver (How to perform CRUD operations In real life project)

Project 2:

Working with Mongoose & Node.js

COURSE TOPICS

MONDODB

MONGODB ATLAS

MONGODB COMPASS

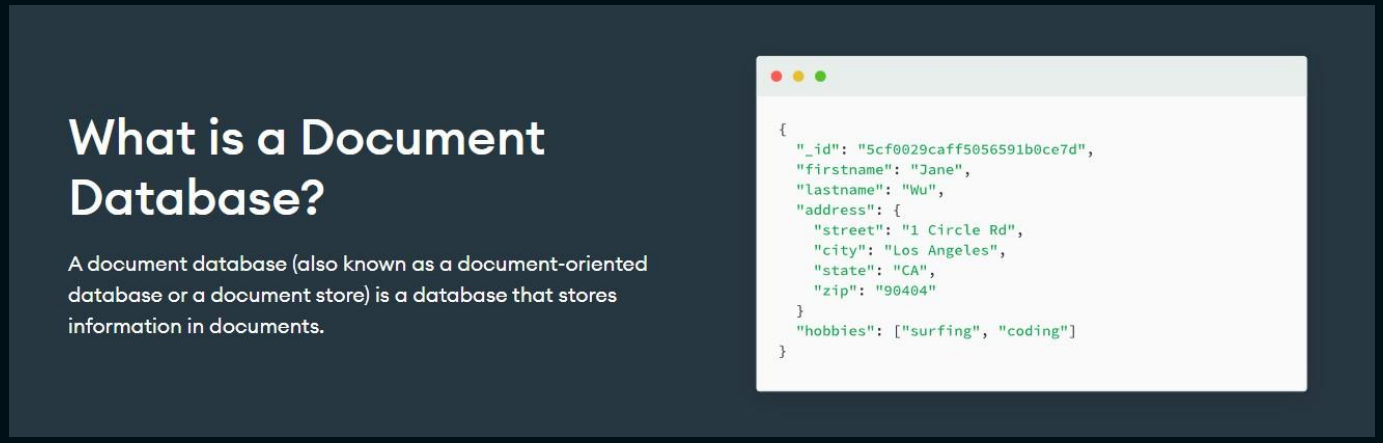
Introduction to MongoDB

What is MongoDB?

NoSQL vs SQL

What is MongoDB?

- MongoDB is an open-source, document-oriented NoSQL database management system.



- Designed for flexibility, scalability, and performance in handling unstructured or semi-structured data

More About MongoDB

10gen

mongoDB

- It was created by a company called 10gen, which is now known as MongoDB, Inc. The company was founded by Eliot Horowitz and Dwight Merriman in 2007. The first version of MongoDB was released in 2009.

Clusters in MongoDB

- In MongoDB, a **cluster** refers to a group of interconnected servers (nodes) that work together to store and manage data.

More About MongoDB

HUMONGOUS

HU MONGO US

SQL

- SQL databases are relational databases.
- They use structured tables to store data in rows and columns.
- Suitable for applications with well-defined schemas and fixed data structures.
- E-commerce Platform, HR Managementetc
- Examples: MySQL, PostgreSQL, Oracle.

MongoDB(NOSQL)

- NoSQL databases are non-relational databases.
- They provide flexibility in data storage, allowing varied data types and structures.
- Ideal for applications with dynamic or evolving data models.
- CMS, Social Media Platforms, Gamingetc
- Examples: MongoDB, Cassandra, Redis.

SQL

Columns

Table: Students

Rows

student_id	first_name	last_name	age	grade
1	Vinod	Thapa	16	11
2	Thapa	Technical	17	12

Table: Subjects

subject_id	subject_name
1	Mathematics
2	Computer

Table: Grades

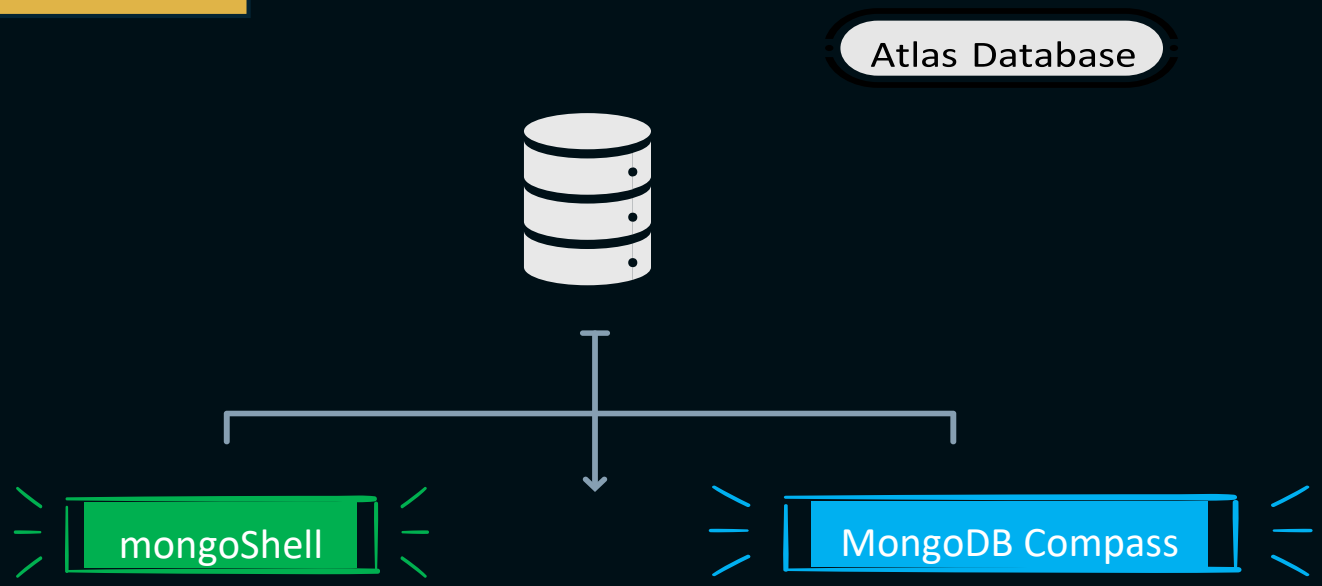
student_id	subject_id	marks
1	1	100
2	2	99

NOSQL (MONGODB)

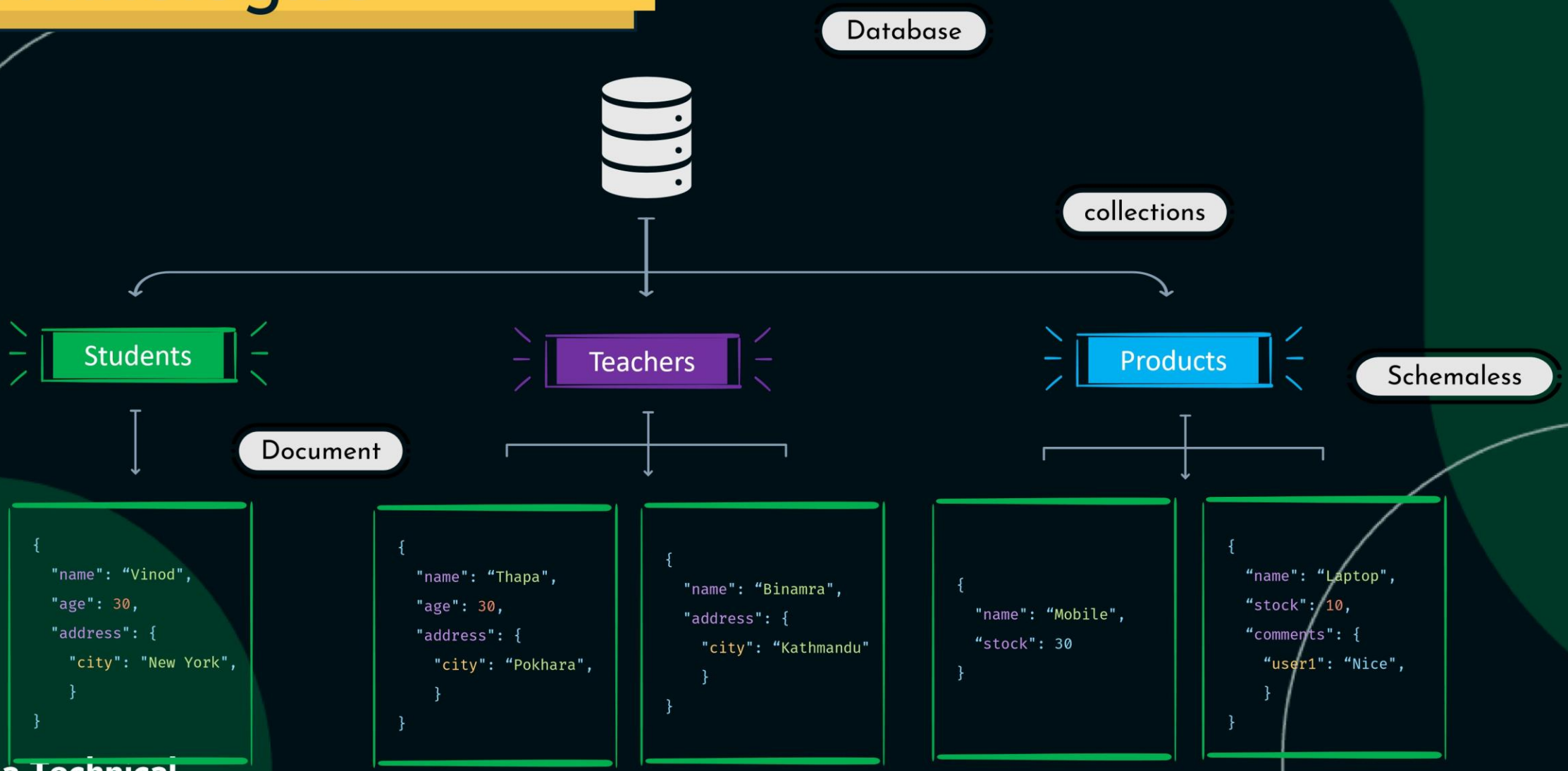
Collection: Students
2- documents

```
[
  {
    "_id": "1",
    "first_name": “Vinod",
    "last_name": “Thapa",
    "age": 16,
    "grade": 11,
    "subjects": [
      { "subject_name": "Mathematics", "marks": 100 },
      { "subject_name": "Computer", "marks": 100 }
    ]
  },
  {
    "_id": "2",
    "first_name": "Thapa",
    "last_name": "Technical",
    "age": 17,
    "grade": 12,
    “extra": ‘sport captain’,
    "subjects": [
      { "subject_name": "Mathematics", "marks": 100 },
      { "subject_name": "Computer", "marks": 100 }
    ]
  }
]
```

MongoDB Data



MongoDB Terminologies



Thapa Technical

Key Features of MongoDB

Flexible Schema Design

- MongoDB allows dynamic, schema-less data structures.
- Easily accommodate changing data requirements.

Scalability and Performance

- Horizontal scaling supports large datasets and high traffic.
- Optimized read and write operations for fast performance.

Document-Oriented Storage

- Data is stored in flexible, JSON-like BSON documents.
- Self-contained units with rich data types and nested arrays.

Dynamic Queries

- Rich query language with support for complex queries.
- Utilize indexes to speed up query execution.

Aggregation Framework

- Perform advanced data transformations and analysis.
- Process data using multiple pipeline stages.

Open Source and Community

- MongoDB is open-source with a vibrant community.
- Regular updates, improvements, and support.

How MongoDB Works?

Frontend

HTML, CSS, JS

React.js, Next.js

Backend

Node.js

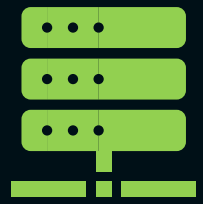
Express.js

Next.js

Python

Database

MongoDB Server



Read & Write Data files

WiredTiger & MMAPV1 (×)

Storage Engine

JSON Vs BSON

- In MongoDB, we write in JSON format only but behind the scene data is stored in BSON (Binary JSON) format, a binary representation of JSON.
- By utilizing BSON, MongoDB can achieve higher read and write speeds, reduced storage requirements, and improved data manipulation capabilities, making it well-suited for handling large and complex datasets while maintaining performance efficiency.

JSON vs BSON

JSON

Easy to Read & Write

```
{
    "name": "Thapa",
    "age": 29,
    "isStudent": false,
    "scores": [92, 108],
    "address": {
      "city": "Pokhara"
    }
}
```

BSON

Not Easy to Read

```
\x1e\x00\x00\x00
\x02
name\x00
\x04\x00\x00\x00Thapa\x00
\x10
age\x00
\x1e\x00\x00\x00\x00\x00\x00\x00
\x08
isStudent\x00
\x00
\x04
scores\x00
\x05\x00\x00\x00\x03\x00\x00\x00
\x10
\x00\x00\x00\x00\x00\x00\x00\x00
\x10
\x00\x00\x00\x00\x00\x00\x00\x00
\x10
\x00\x00\x00\x00\x00\x00\x00\x00
```

Installing MongoDB

- https://www.mongodb.com/try/download/community
- https://www.mongodb.com/try/download/shell
- https://www.mongodb.com/try/download/database-tools

BSON in MongoDB

Binary JSON Format: BSON, Binary JSON, is used in MongoDB for data storage and transmission.

Efficient Storage: Designed for efficient data storage and transmission in MongoDB.

Diverse Data Types: Supports a wider range of data types, including Binary, Date, and Regular Expression.

Compact & Fast: BSON's binary format is more compact, leading to smaller storage and faster processing.

Native to MongoDB: MongoDB stores data in BSON format, ensuring seamless integration.

Performance Boost: Faster serialization improves data access and manipulation speed.

Managing Databases in MongoDB

Creating / Deleting Databases

Creating / Deleting Collections

Managing Databases and Collections

- `show dbs;`
- `use <database-name>;`
- `db.dropDatabase();`

- `show collections;`
- `db.createCollection('<collection-name>');`
- `db.<collection-name>.drop();`

Insert Operation in MongoDB

Inserting Documents in MongoDB

When to use Quotes and when not to?

Ordered and Unordered Inserts

Case Sensitivity in MongoDB

Inserting Documents in MongoDB

```
db.<collection-name>.insertOne({
    field1: value1,
    field2: value2,
    ...
});
```

```
db.<collection-name>.insertMany([
    { field1: value1, field2: value2, ... },
    { field1: value1, field2: value2, ... },
    // ...
]);
```

MONGOD INSERTONE

InsertOne

```
db.Students.insertOne(
  {
    name: “Binamra",
    age: 20,
  }
);
```

Before

```
[
  {
    name: "Vinod",
    age: 29,
  }
]
```

After

```
[
  {
    name: "Vinod",
    age: 29,
  },
  {
    name: “Binamra",
    age: 20,
  }
]
```

MONGOD INSERTMANY

Before

```
[
  {
    name: "Vinod",
    age: 29,
  }
]
```

InsertMany

```
db.Students.insertMany([
  {
    name: "Binamra",
    age: 20,
  },
  {
    name: "Thapa",
    age: 21,
  },
]);
```

After

```
[
  {
    name: "Vinod",
    age: 29,
  },
  {
    name: “Binamra",
    age: 20,
  },
  {
    name: "Thapa",
    age: 21,
  }
]
```

When to use Quotes and when not to?

- **Special Characters**

  If a field name contains special characters or spaces, or starts with a numeric digit, using quotes is necessary.

- **Reserved Words**

  If a field name is a reserved keyword in MongoDB, use quotes to distinguish it from the reserved keyword.

Ordered and Unordered Inserts

When executing bulk write operations, "ordered" and "unordered" determine the batch behavior.

- **Ordered Inserts**

  Default behavior is ordered, where MongoDB stops on the first error.

  ```
  db.<collection-name>.insertMany([ doc1, doc2, ... ]);
  ```

- **Unordered Inserts**

  When executing bulk write operations with unordered flag, MongoDB continues processing after encountering an error.

  ```
  db.<collection-name>.insertMany([ doc1, doc2, ... ],{ ordered: false });
  ```

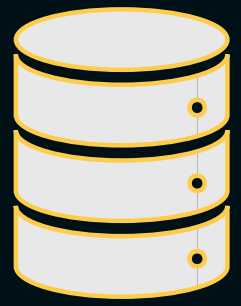
$Ordered Inserts

Documents Before the wrong one will be inserted & after the one with error will not.

{Correct Doc. 1}

{Wrong Doc. 2}

{Correct Doc. 3}



{Correct Doc. 1}

$Ordered = false

Documents before the one with an error will be inserted, and the documents after the one with an error will also be inserted. Only the document with the error will not be inserted.

Case Sensitivity in MongoDB

- Collection names are case-sensitive.
- Field names within documents are also casesensitive.

```
db.Product.insertOne({ name: 'thapa', age: 30 });
db.product.insertOne({ name: 'thapa', age: 30 });
```

Read Operations in MongoDB

Inserting Documents in MongoDB

Ordered and Unordered Inserts

Case Sensitivity in MongoDB

Comparison Operators

Logical Operators

Cursors in MongoDB

Finding Documents in MongoDB

- **find()**

  db.collection_name.find({ key: value })

- **findOne()**

  db.collection_name.findOne({ key: value })

Importing JSON in MongoDB

- `mongoimport jsonfile.json –d database_name –c collection_name`
- `mongoimport products.json –d shop –c products`
- `mongoimport products.json –d shop –c products ––jsonArray`

- Here, `––jsonArray` accepts the import of data expressed with multiple MongoDB documents within a single JSON array.
- Limited to imports of 16 MB or smaller.

Comparison Operators

$eq	$ne	$gt	$gte
$lt	$lte	$in	$nin

- `db.products.find({ 'price': { $eq: 699 } });`
- `db.category.find({ price: { $in: [249, 129, 39]} });`

Introduction to Cursors

Cursors in MongoDB are used to efficiently retrieve large result sets from queries, providing control over the data retrieval process.

MongoDB retrieves query results in batches using cursors.

Cursors are a pointer to the result set on the server.

Cursors are used to iterate through query results.

Automatic Batching

MongoDB retrieves query results in batches, not all at once.

Default batch size is usually 101 documents.

This improves memory efficiency and network usage.

Cursor Methods

count() limit() skip() sort()

- `db.products.find({ price: { $gt: 250 } }).count();`
- `db.products.find({ price: { $gt: 250 } }).limit(5);`
- `db.products.find({ price: { $gt: 250 } }).limit(5).skip(2);`
- `db.products.find({ price: { $gt: 1250 } }'.limit(3).sort({ price: 1 });`
  - (1) for ascending and (-1) for descending

Cursor Methods (Caveats)

Performance Implications

- `skip()` can be inefficient for large offsets.
- Using `sort()` on large result sets may impact performance.

Use with Caution

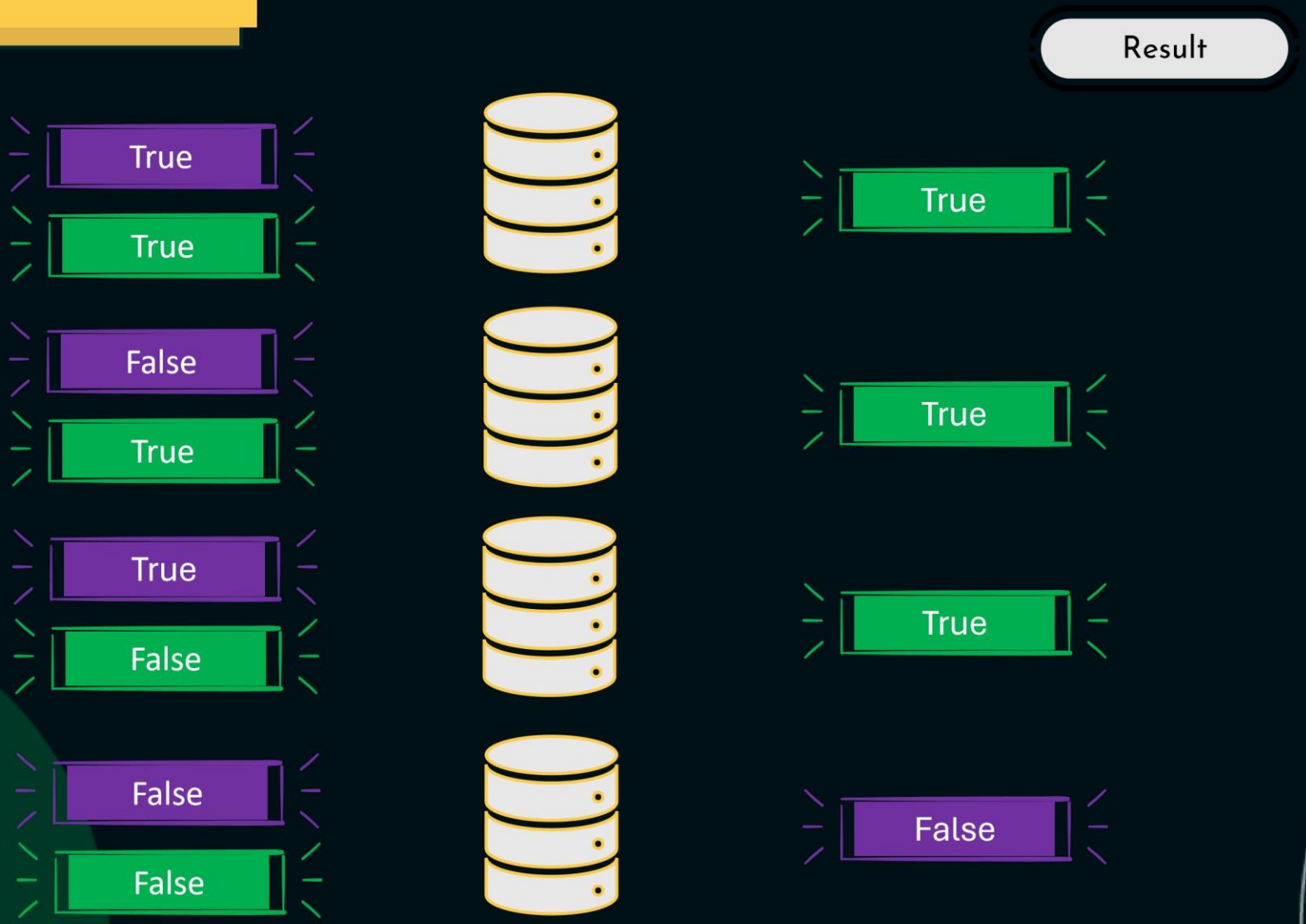
- Be cautious when using `limit()` and `skip()` on large collections.
- Consider using indexing to optimize query performance.

Logical Operators

$and $or $not $nor

- `{ $and: [ { condition1 }, { condition2 }, ... ] }`
- `{ field: {$not: { operator: value } } }`

$or Operators



Complex Expressions

- The $expr operator allows using aggregation expressions within a query.
- Useful when you need to compare fields from the same document in a more complex manner.
- Syntax
  - `{ $expr: { operator: [field, value] } }`
- Example
  - `db.products.find({ $expr: { $gt: ['$price', 1340] } });`

Elements Operator

$exists

$type

$size

- { field: { $exists:<boolean>} }
- { field: { $type: "<bson-data-type>" } }
- { field: { $size: <arraylength> } }

Projection

- `db.collection.find({}, { field1: 1, field2: 1})`
- To include specific fields, use projection with a value of 1 for the fields you want.
- To exclude fields, use projection with a value of 0 for the fields you want to exclude.
- You cannot include and exclude fields simultaneously in the same query projection.

Embedded Documents

- Query documents inside embedded documents using dot notation.
- db.collection.find({ “parent.child”: value })

$all vs $elemMatch

- The $all operator selects the documents where the value of a field is an array that contains all the specified elements.
  - { <field>: { $all: [<value1> , <value2> ...] }}

- The $elemMatch operator matches documents that contain an array field with at least one element that matches all the specified query criteria.
  - { <field>: { $elemMatch: { <query1>, <query2>, ... } }}

Update Operations in MongoDB

updateOne() and updateMany()

Removing and renaming fields

Adding, removing items from array

Updating embedded documents

updateOne() and updateMany()

- db.collectionName.updateOne(
  { filter },
  { $set: { existingField: newValue, newField: "new value", // ... }}
  );

- db.collectionName.updateMany(
  { filter },
  { $set: { existingField: newValue, // ... }, }
  );

Removing and Renaming Fields

- `db.collectionName.updateOne( { filter }, { $unset: { fieldName: 1 } } );`

- ```
  db.collectionName.updateOne(
        { filter },
        { $rename: { oldFieldName: "newFieldName" } }
  );
  ```
  ```

Updating arrays and Embedded Documents

```
db.collectionName.updateOne(
  { filter },
  { $push: { arrayField: "new element" } }
);
```

```
db.collectionName.updateOne(
  { filter },
  { $pop: { arrayField: value } }
);
```

```
db.collectionName.updateOne(
  { filter },
  { $set: { 'arrayField.$.text": "Updated text" }}
);
```

Delete Operations in MongoDB

- `db.collectionName.deleteOne({ filter });`

- `db.sales.deleteMany({ price: 55 });`

MONGOD DELETEONE

Before

```
[
  {
    name: "Vinod",
    age: 29,
  },
  {
    name: “Binamra",
    age: 20,
  }
]
```

DeleteOne

```
db.Students.deleteOne(
  {
    name: “Binamra"
  }
);
```

After

```
[
  {
    name: "Vinod",
    age: 29,
  }
]
```

Indexes in MongoDB

What are Indexes?

Benefits of Indexes

Managing Indexes

Unique, Text Index

When not to use Indexes?

What are Indexes?

Indexes are specialized data structures that optimize data retrieval speed in MongoDB.

- Indexes store a fraction of data in a more searchable format.
- They enable MongoDB to locate data faster during queries.
- Indexes are separate from collections and multiple indexes can exist per collection.

Benefits of Indexes

Faster Querying: Indexes drastically accelerate data retrieval, particularly for large collections.

Efficient Sorting: Indexes facilitate rapid sorting based on specific fields.

Improved Aggregation: Aggregation operations become more efficient with optimized indexes.

Indexing on Multiple Fields: Complex queries can be executed efficiently by utilizing multiple fields in indexes.

explain()

- Use `explain()` method to understand query execution in detail.

- `db.products.find({ name: 'Air Fryer' }).explain();`

- `db.products.find({ name: 'Air Fryer' }).explain("executionStats");`
  - Use it to measure the time taken to execute a query.

Managing Indexes

- `db.products.createIndex({ field: 1 });`
  - (1) for storing indexes in ascending order.
  - (-1) for storing indexes in descending order.
- `db.collection.getIndexes();`
  - _id is a default index.
- `db.collection.dropIndex({ field: 1 });`
- `db.collection.dropIndex("index_name");`

Unique and Text Indexes

- `db.collection.createIndex({ field: 1 },{ unique: true });`

- `db.collection.createIndex({ field: "text" });`
- `db.collection.find({ $text: { $search: "keyword" }});`
  - Searching using index is faster than `$regex` searching.
  - `db.products.find({ field: { $regex: "air" } })`

When not to use Indexes?

Indexes on Rarely Used Fields

- Indexing fields that are seldom used in queries can consume unnecessary space and resources.

Balancing Act

- Indexing requires disk space and memory. Overindexing can lead to resource strain and impact overall performance.

Indexing Small Collections

- In smaller collections, the cost of index maintenance might outweigh the benefits gained from querying.

Aggregation in MongoDB

What is Aggregation?

What is Aggregation?

- Definition: Aggregation is the process of performing transformations on documents and combining them to produce computed results.
- Pipeline Stages: Aggregations consist of multiple pipeline stages, each performing a specific operation on the input data.
- Benefits
  - Aggregating Data: Complex calculations and operations are possible.
  - Advanced Transformations: Data can be combined, reshaped, and computed for insights.
  - Efficient Processing: Aggregation handles large datasets efficiently.

$match

- The $match stage is similar to the query used as the first argument in .find(). It filters documents based on specified conditions.

- Syntax

```
{ $match: { <query> } }
```

- Example

```
db.products.aggregate([
        { $match: { company: "64c23350e32f4a51b19b9235" } }
]);
```

$group

- The $group stage groups documents by specified fields and performs aggregate operations on grouped data
- ```
  {
    $group:
      {
        _id: <expression>, // Group key
        <field1>: { <accumulator1> : <expression1> },
        ...
      }
  }
  ```
- ```
  db.products.aggregate([
        { $group: { _id: { comp: "$company" },totalProducts: { $sum: 1 } }}
  ]);
  ```
- This groups products by company and calculates the total number of products for each company.

$group (continued)

- The $group stage can calculate various aggregate values within grouped data.

- db.products.aggregate([

```
        { $group: {
                _id: { comp: "$company" },
                totalPrice: { $sum: "$price" },
                totalProducts: { $sum: 1 },
                averagePrice: { $avg: "$price"}
        } }
]);
```

$sort

- `{ $sort: { <field>: <order> }}`

- ```
  db.products.aggregate([
        { $sort: { totalProducts: 1 } }
  ]);
  ```
  ```

$project

- The $project stage reshapes documents, includes or excludes fields, and performs operations on fields.

- `{ $project: { <field1>: <expression1>, ... } }`

- ```
  db.products.aggregate([
        { $project: { name: 1, discountedPrice: { $subtract: ["$price", 5] } } }
  ]);
  ```

- Projects the name field and calculates a discountedPrice field by subtracting 5 from the price.

- $sum, $subtract, $multiply, $avg, etc. are types of expression operator.
  ```

$push

- The $push stage adds elements to an array field within documents.

- `{ $push: <expression> }`

- ```
  db.products.aggregate([
        { $group: { _id: { company: "$company" }, products:{ $push: "$name" } } }
  ]);
  ```
  ```

$unwind

- The $unwind stage deconstructs an array field and produces multiple documents.

- `{ $unwind: <array> }`

- ```
  db.products.aggregate([
          { $unwind: "$colors" },
          { $group: { _id: { company: "$company" }, products: { $push: "$colors" }}}
  ]);
  ```

- Deconstructs the colors array field, groups products by company, and creates an array of colors for each company.
  ```

$addToSet

- The $addToSet stage adds elements to an array field while preventing duplicates.

- ```
  db.products.aggregate([
          { $unwind: "$colors" },
          { $group: {
                  _id: { company: "$company" },
                  products: { $addToSet: "$colors" }
          } }
  ])
  ```

- Groups products by company and creates an array of unique colors for each company.
  ```

$size

- The $size stage calculates the length of an array field.

- `{ $size: <array> }`

- ```
  db.products.aggregate([
          { $project: { name: 1, numberOfColors: { $size: "$colors" } } }
  ]);
  ```

- Projects the name field and calculates the number of colors in the colors array.
  ```

$limit and $skip

- The $limit and $skip stages are useful for pagination, limiting, and skipping results.

- `{ $limit: <positive integer> }`

- ```
  db.products.aggregate([
          { $skip: 10 },
          { $limit: 10 }
  ]);
  ```
  ```

$filter

- The $filter stage filters elements of an array based on specified conditions.

```
{
    $project: {
        <field>: {
            $filter: {
                input: '$<array>',
                as: '<variable>'
                cond: <expression>
            }
        }
    }
}
```

$addFields

- The $addFields stage adds new fields to documents in a cleaner way compared to $project.

- `{ $addFields: { <field1>: <expression1>, ... }}`

- ```
  db.products.aggregate([
        { $addFields: { discountedPrice: { $subtract: ["$price", 5] } }}
  ]);
  ```
  ```

Introduction to MongoDB Atlas

- MongoDB Atlas is MongoDB's fully managed cloud database service.
- It offers an easy way to deploy, manage, and scale MongoDB databases in the cloud.
- Atlas eliminates the need for manual setup and maintenance, allowing developers to focus on their applications.
- It provides automated scaling options to accommodate growing workloads.
- Atlas supports global clusters, enabling databases to be deployed across multiple regions for better data availability and reduced latency.

MongoDB Atlas Setup

Working with MongoDB Compass

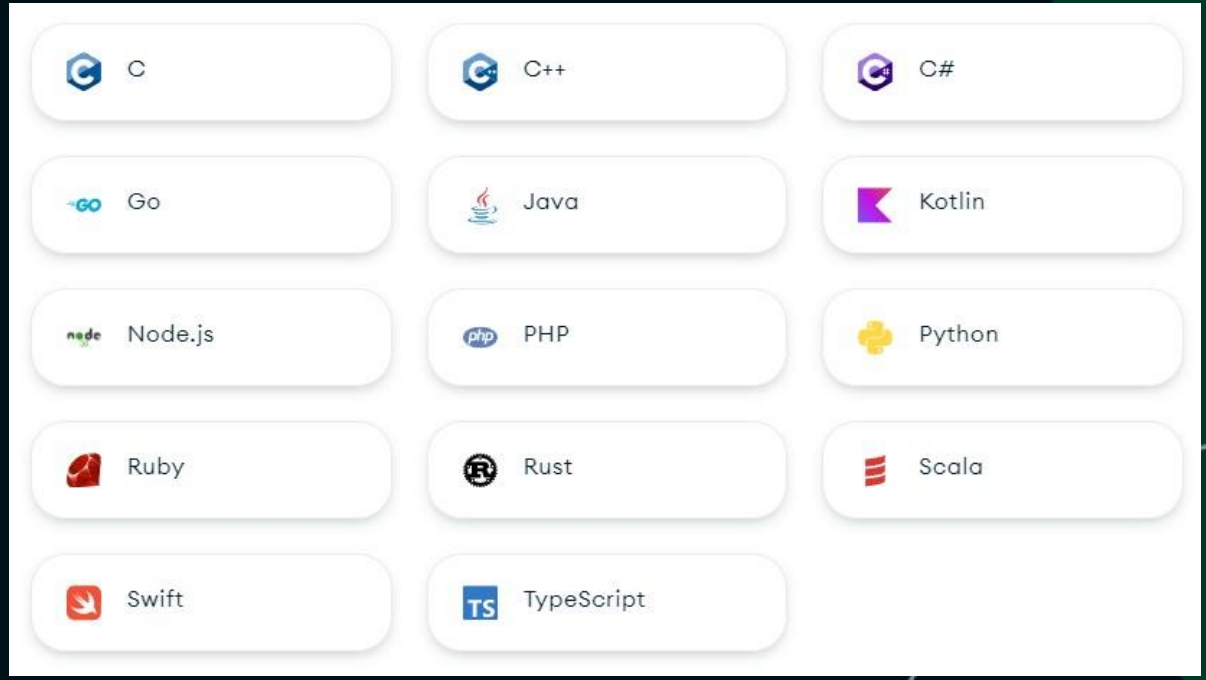
Working with MongoDB Drivers

Introduction to MongoDB Drivers

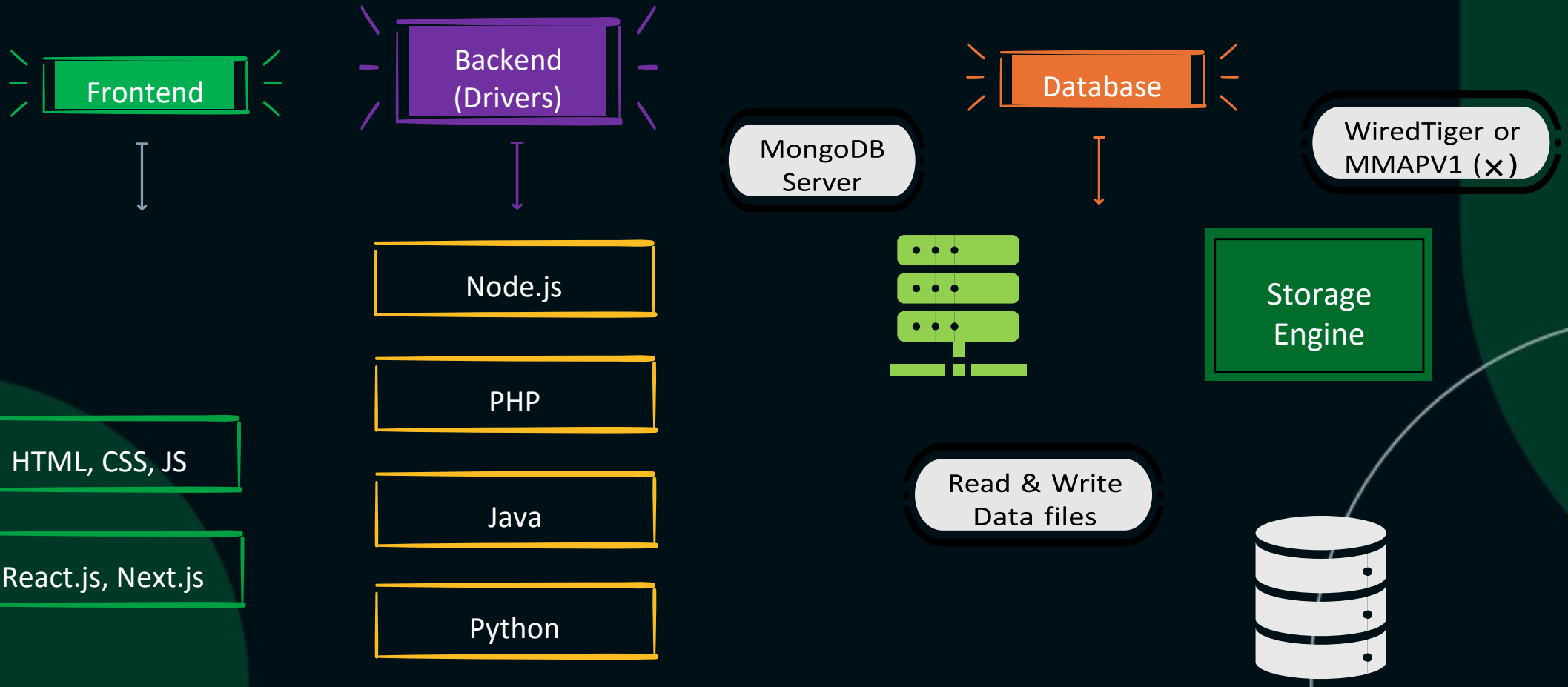
Working with Node.js MongoDB Drivers

Introduction to MongoDB Drivers

- Software libraries that allow applications to interact with MongoDB databases.
- MongoDB offers official and community-supported drivers for various programming languages.
- Drivers provide APIs tailored to specific programming languages.
- https://www.mongodb.com/docs/drivers/

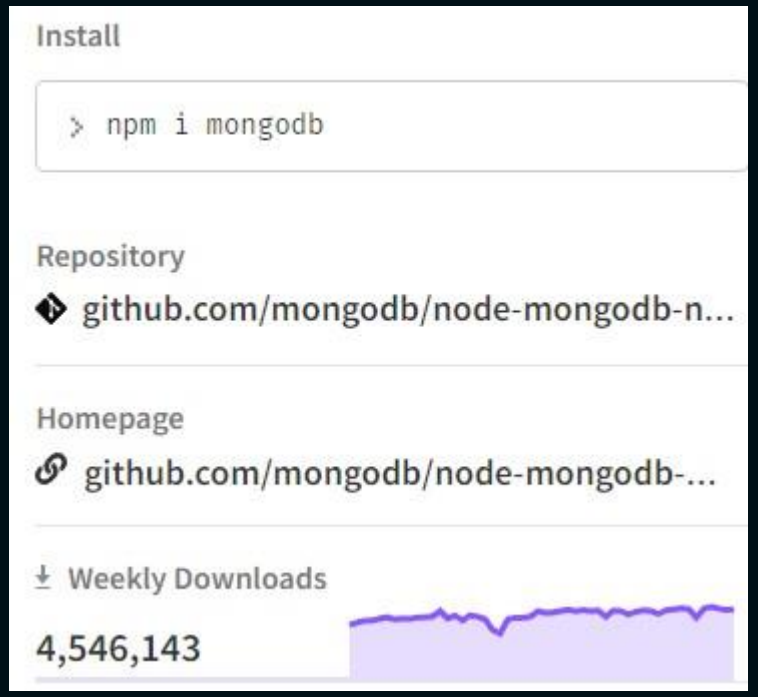


How MongoDB Works?



Getting Started with Node.js MongoDB Drive

- Download and install Node.js from official Node.js website.
- Create a node.js project using npminit –y
- Install mongodb driver using npm install mongodb
- https://www.npmjs.com/package/mongodb
- Create a connection with MongoDB database and start working with it.



Getting Started with Node.js MongoDB Driver

- Connect to MongoDB Server: Use the **MongoClient** class and a valid URI to establish a connection to the MongoDB server.
- Select a Database: Access a specific database using the **client.db(databaseName)** method.
- Access a Collection: Retrieve a collection reference using the **db.collection(collectionName)** method.
- Perform Operations: Perform CRUD operations like querying, inserting, updating, and deleting documents within the collection.
- Close Connection: Safely close the connection using the **client.close()** method when done

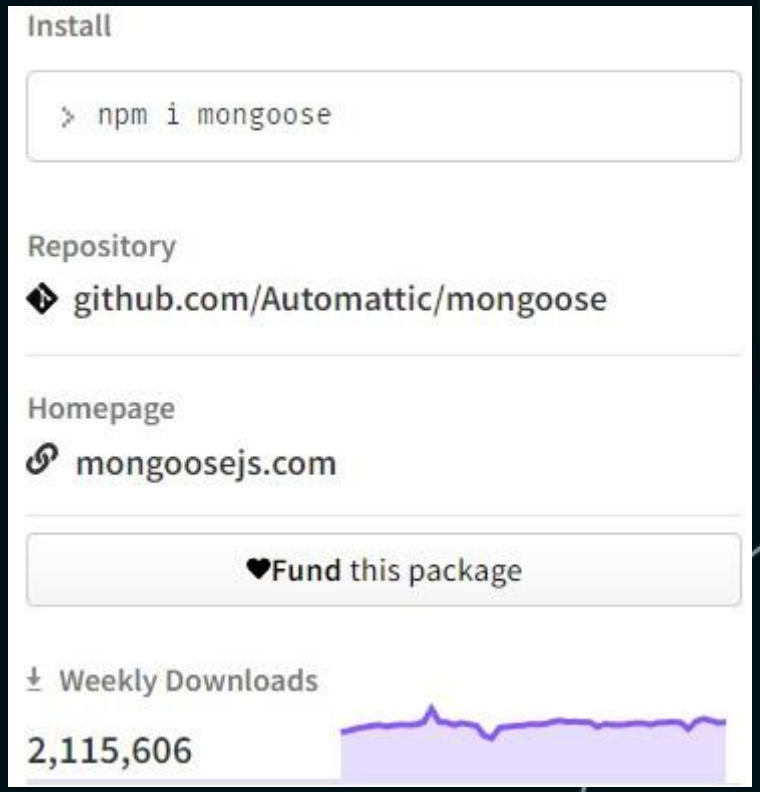
Working with Mongoose

Introduction to MongoDB Drivers

Working with Node.js MongoDB Drivers

Getting Started with Node.js MongoDB Drive

- It's an Object Data Modeling (ODM) library for MongoDB and Node.js.
- It makes MongoDB interaction more straightforward and organized.
- It provides a structured, schemabased data modeling approach.



Why Mongoose instead of official driver?

Structured Schemas

- Mongoose lets you define your data's structure using schemas which makes it easier to understand your database structure and work with it.

Validation

- Mongoose provides built-in validation to ensure validity before saving it to database

Relationships

- MongoDB doesn’t provide relations itself. So, Mongoose helps to replicate relations in MongoDB and helps us to relate schemas with each other easily (one-to-one, one-to-many, etc.)

Middleware

- Mongoose offers running custom functions before or after certain operations which can be useful in many cases.

Complex Queries

- MongoDB helps to write complex queries, aggregations, etc. with simpler syntax to help us to work on projects easily