CT449:
Phát triển ứng dụng web

Bùi Võ Quốc Bảo
(bvqbao@cit.ctu.edu.vn)

Cần Thơ, 2022

Credit

- The slides are inspired by the CS193X course created by Victoria Kirst
- https://www.bezkoder.com/

MongoDB

Database definitions

A **database** (**DB**) is an organized collection of data

- By this definition, the JSON file can be considered a database

A **database management system** (**DBMS**) is software that handles the storage, retrieval, and updating of data

- Examples: MongoDB, MySQL, PostgreSQL, etc
- Usually when people say "**database**", they mean data that is managed through a DBMS

MongoDB

MongoDB: A popular open-source DBMS

- A *document-oriented* database as opposed to a *relational* database

Relational database:

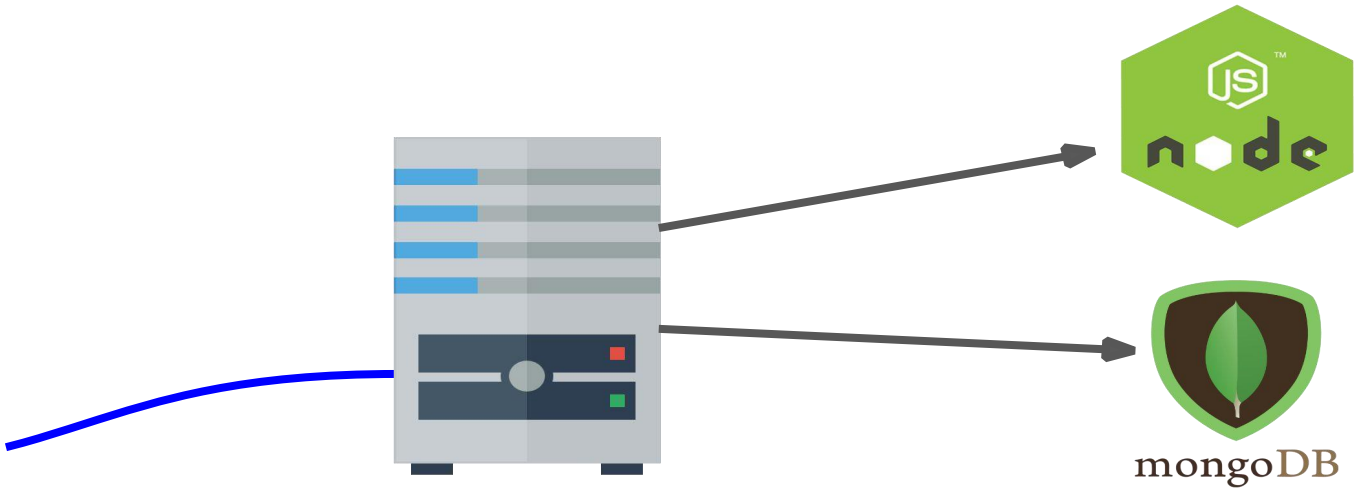
Name	School	Employer	Occupation
Lori	null	Self	Entrepreneur
Malia	Harvard	null	null

Relational databases have fixed schemas; document-oriented databases have flexible schemas

Document-oriented DB:

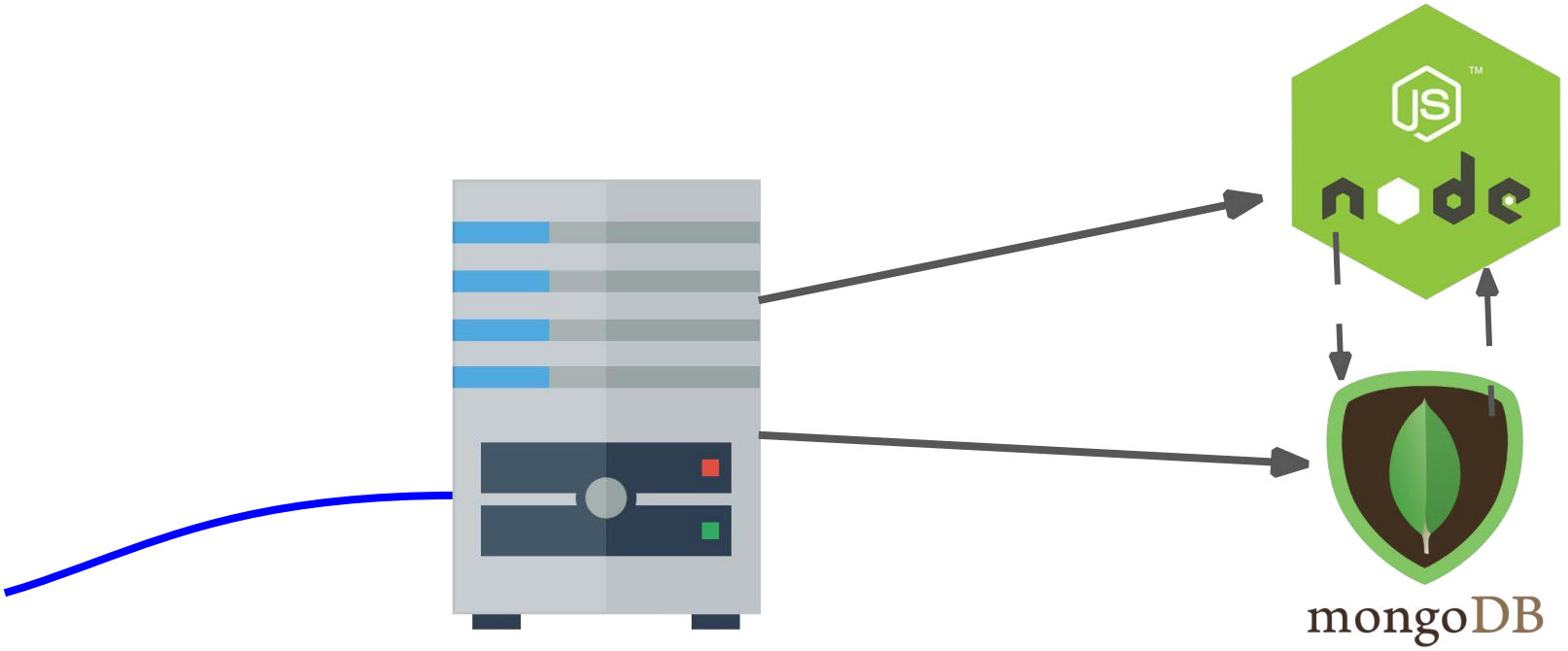
```
{
  name: "Lori",
  employer: "Self",
  occupation: "Entrepreneur"
}
{
  name: "Malia",
  school: "Harvard"
}
```

MongoDB



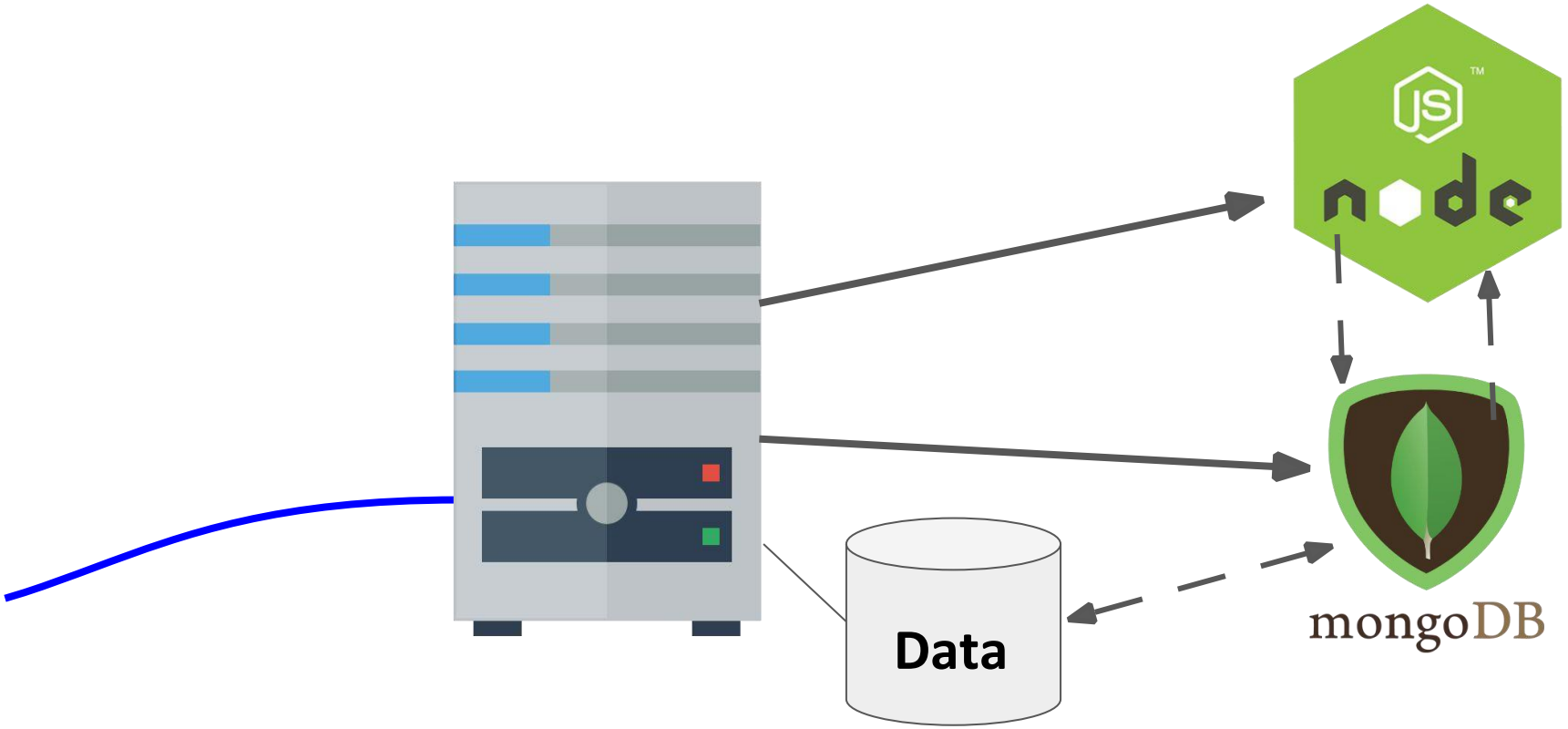
MongoDB is another **software program** running on the computer, alongside our NodeJS server program.
It is also known as the **MongoDB server**.

MongoDB



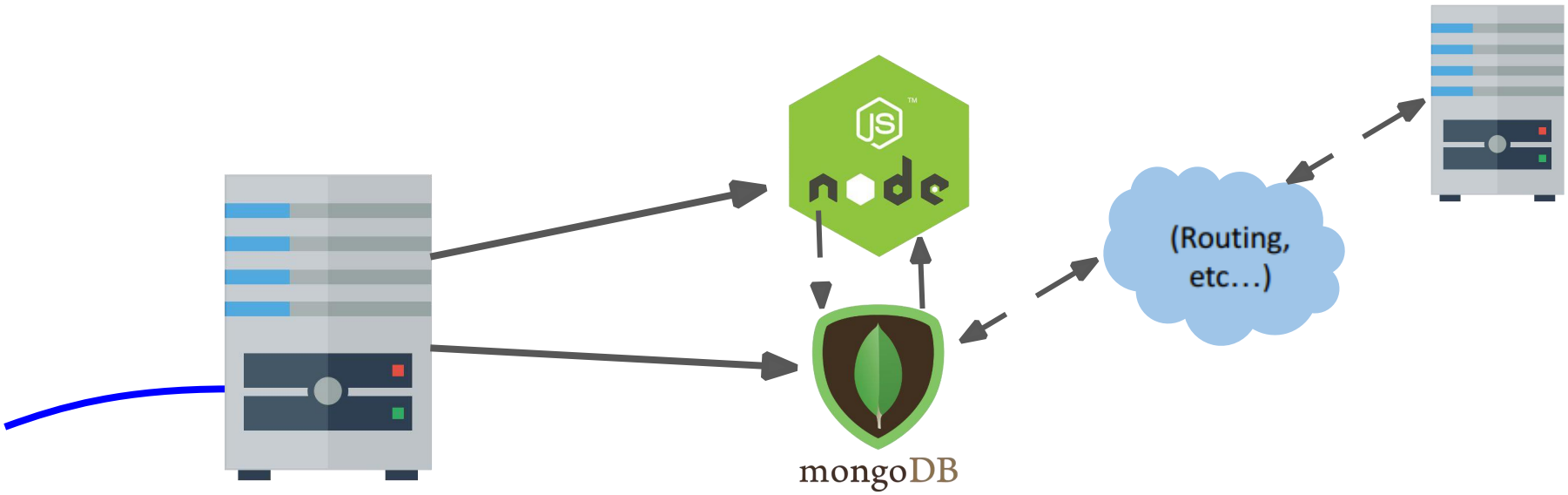
There are MongoDB libraries we can use in NodeJS to communicate with the MongoDB Server, which reads and writes data in the database it manages.

MongoDB



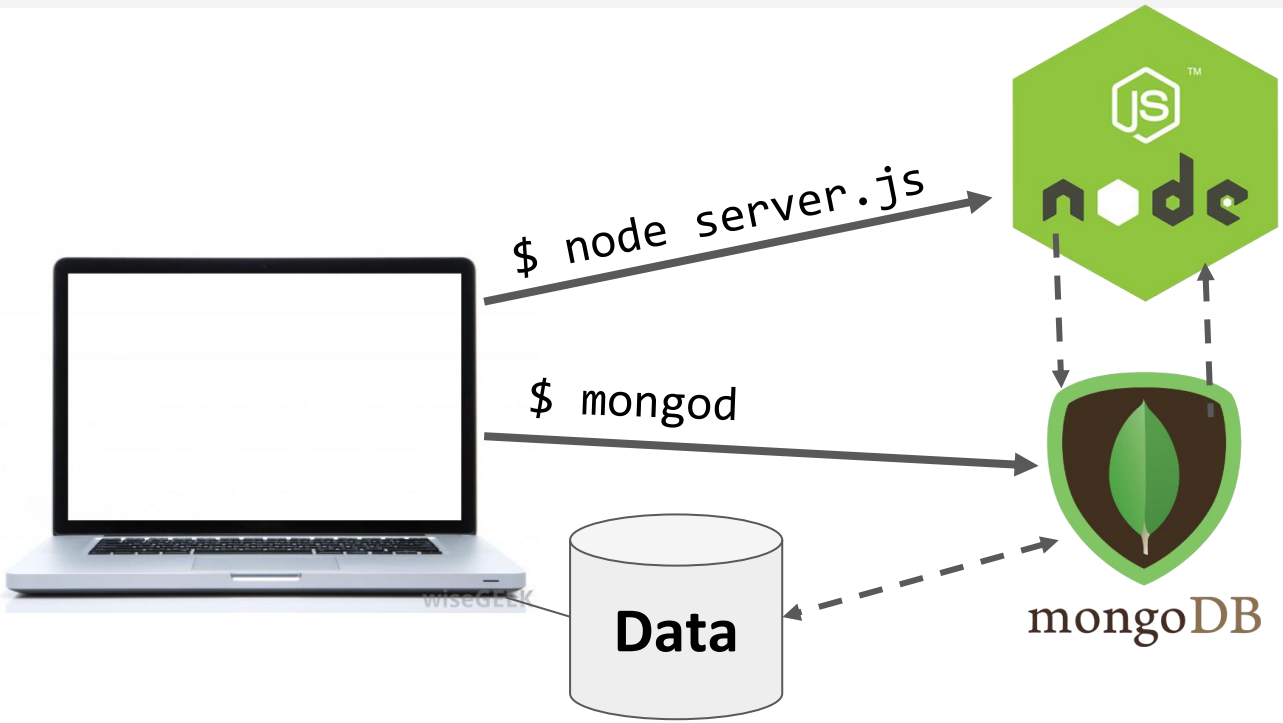
The database the MongoDB Server manages might be local to the server computer...

MongoDB



Or it could be stored on other server computer(s) ("cloud storage").

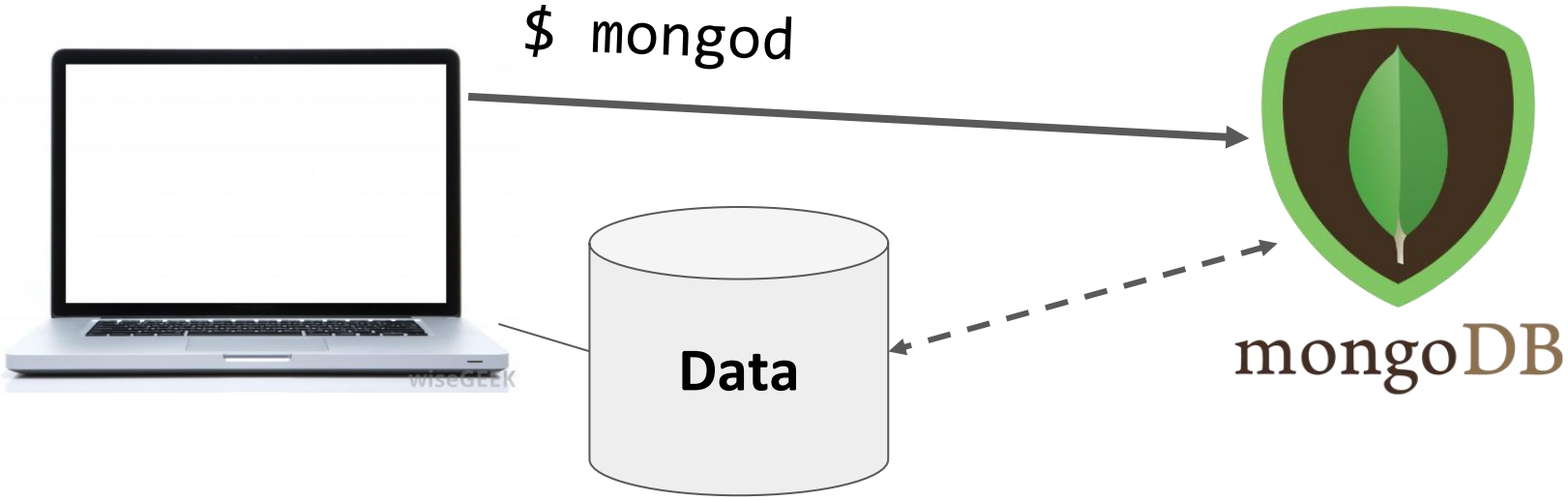
System overview



For development, we will have 2 processes running:

- node will run the main server program on port 3000
- **mongod will run the database server on a port 27017**

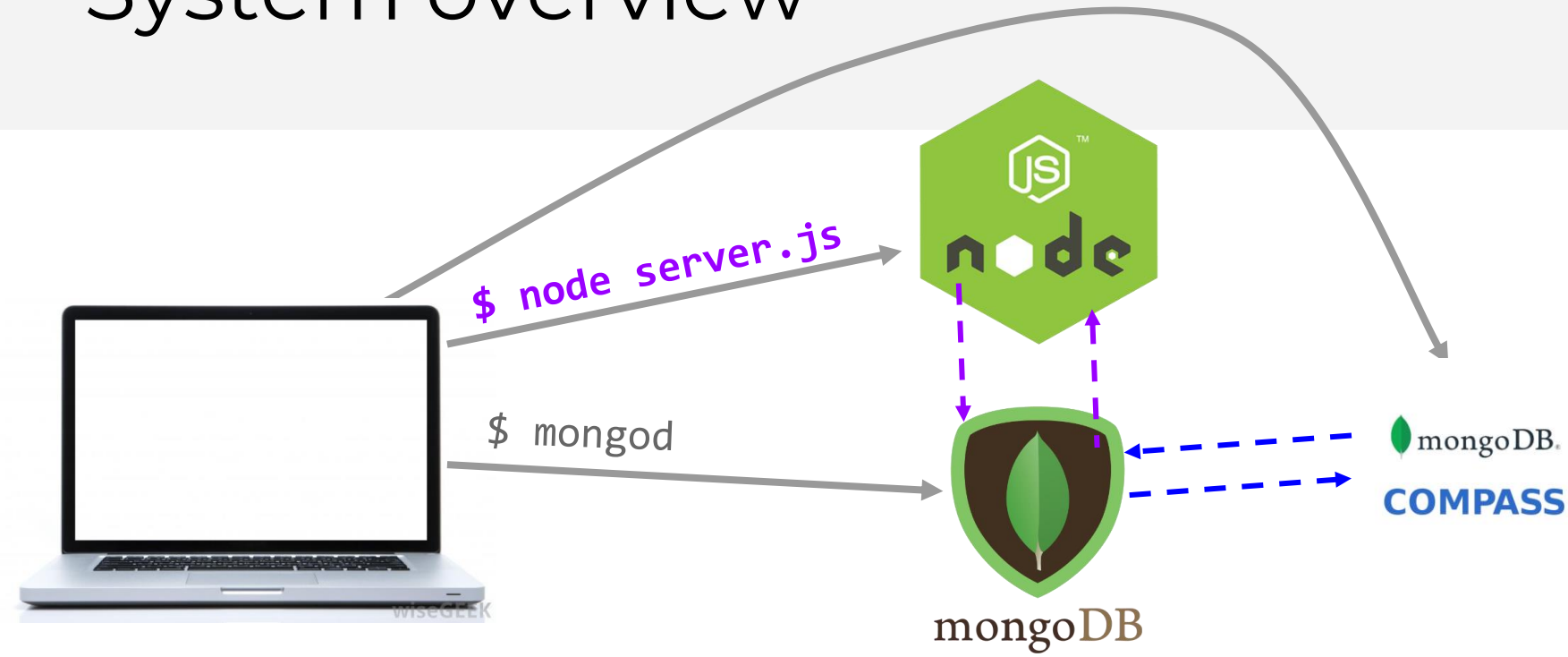
System overview



The mongod server will be bound to port 27017 by default

- The mongod process will be listening for messages to manipulate the database: insert, find, delete, etc

System overview



Three ways of communicating to the MongoDB server:

- NodeJS libraries
- MongoDB shell
- MongoDBCompass

MongoDB concepts

Database:

- A container of MongoDB **collections**

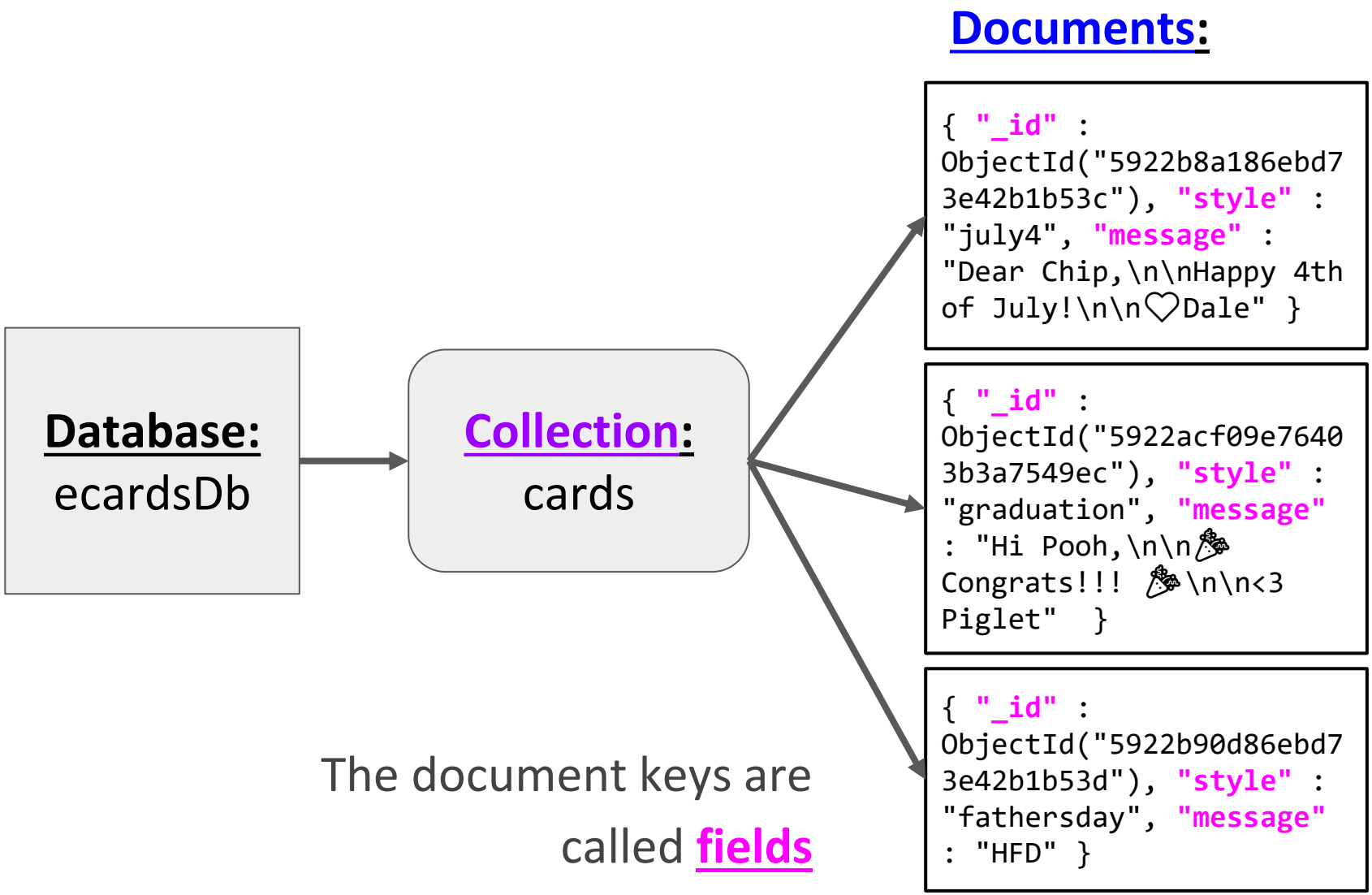
Collection:

- A group of MongoDB **documents**
- (**Table** in a relational database)

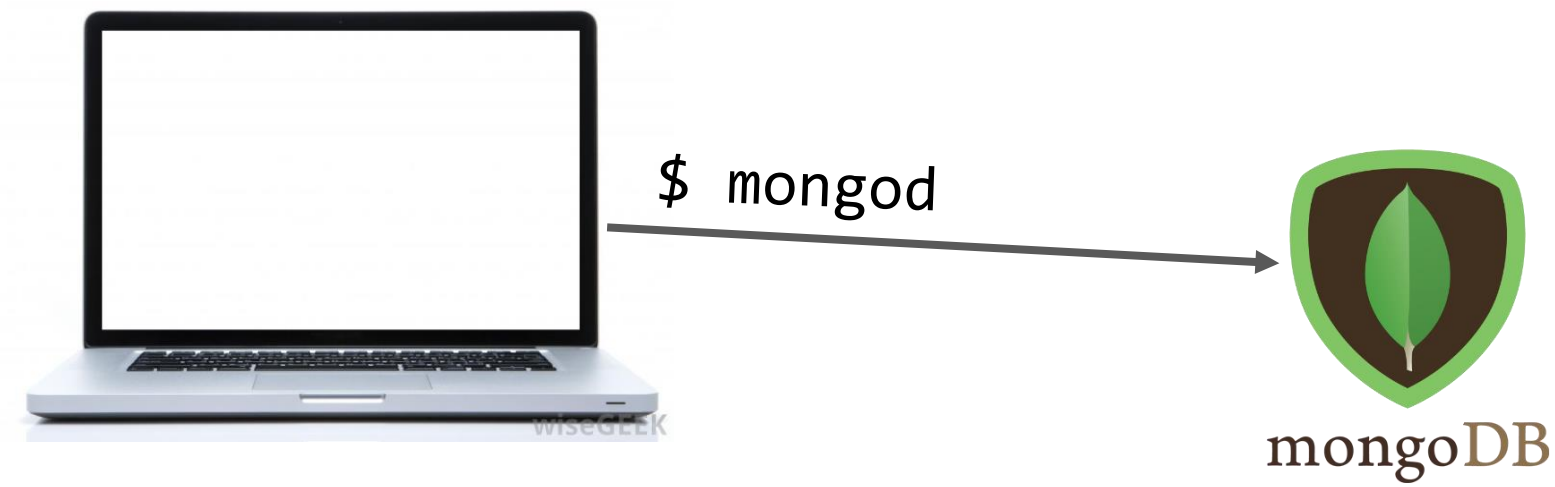
Document:

- A JSON-like object that represents one instance of a collection (**Row** in a relational database)
- Also used more generally to refer to any set of key-value pairs

MongoDB example



mongod: Database process



When you install MongoDB, it will come with the `mongod` command-line program. This launches the MongoDB database management process and binds it to port 27017:

```
$ mongo
```

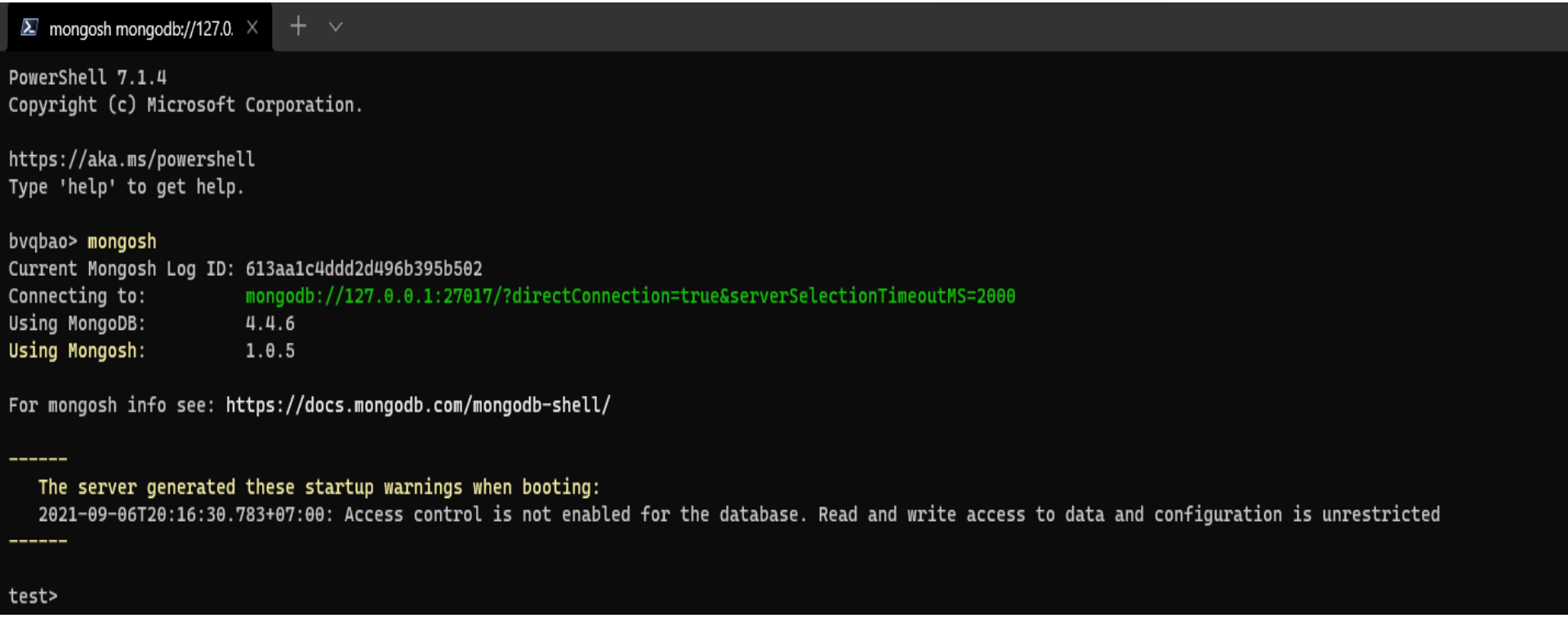
MongoDB Database Tools

A collection of command-line utilities for working with a MongoDB deployment

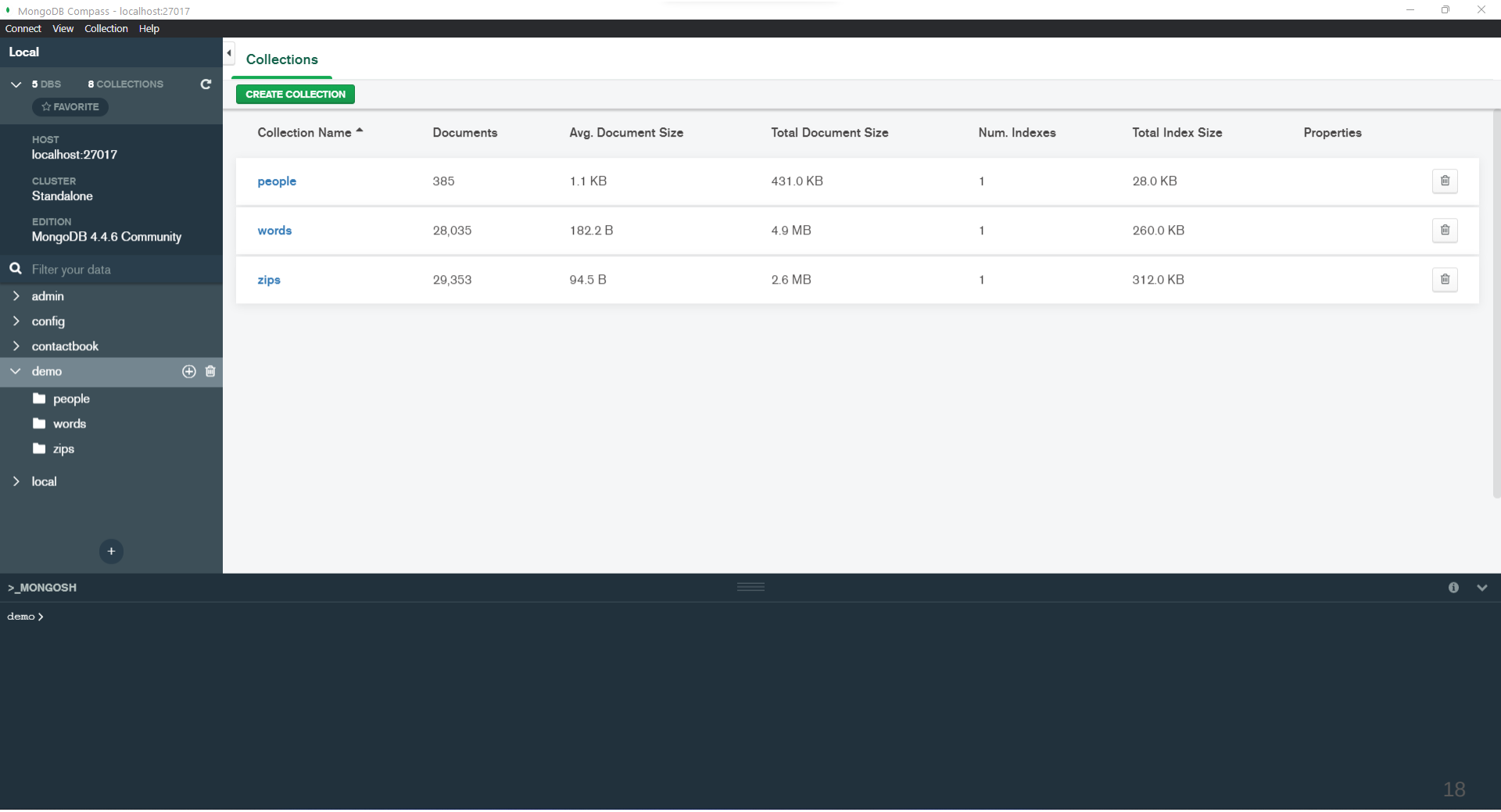
Binary Import / Export	
`mongodump`	Creates a binary export of the contents of a `mongod` database.
`mongorestore`	Restores data from a `mongodump` database dump into a `mongod` or `mongos`
`bsondump`	Converts BSON dump files into JSON.
Data Import / Export	
`mongoimport`	Imports content from an Extended JSON, CSV, or TSV export file.
`mongoexport`	Produces a JSON or CSV export of data stored in a `mongod` instance.
Diagnostic Tools	
`mongostat`	Provides a quick overview of the status of a currently running `mongod` or `mongos` instance.
`mongotop`	Provides an overview of the time a `mongod` instance spends reading and writing data.
GridFS Tools	
`mongofiles`	Supports manipulating files stored in your MongoDB instance in GridFS objects.

MongoDB Shell

Command-line interface to MongoDB



MongoDB Compass



mongo shell commands

`> show dbs`

- Displays the databases on the MongoDB server

`> use` ***databaseName***

- Switches current database (db) to ***databaseName***
- The ***databaseName*** does not have to exist already
  - It will be created the first time you write data to it

`> show collections`

- Displays the collections for the current database

mongo shell commands

> db.***collection***

- Variable referring to the ***collection*** collection

> db.***collection***.find(***query, projection***)

- Prints the results of ***collection*** matching the query
- The ***query*** is a MongoDB Document (i.e. a JSON object)
  - To get everything in the ***collection*** use `db.collection.find()`
  - To get everything in the collection that matches x=foo, `db.collection.find({x: 'foo'})`

mongo shell commands

> db.***collection***.findOne(***query, projection***)

- Prints the first result of ***collection*** matching the query

> db.***collection***.insertOne(***document***)

- Adds ***document*** to the ***collection***
- ***document*** can have any structure

```
> db.test.insertOne({ name: 'dan' })
> db.test.find()
{ "_id" : ObjectId("5922c0463fa5b27818795950"), "name" : "dan" }
```

MongoDB will automatically add a unique **_id** to every document in a collection

mongo shell commands

> db.***collection***.deleteOne(***query***)

- Deletes the first result of ***collection*** matching the query

> db.***collection***.deleteMany(***query***)

- Delete multiple documents from ***collection***.
- To delete all documents, db.***collection***.deleteMany({})

> db.***collection***.drop()

- Removes the collection from the database

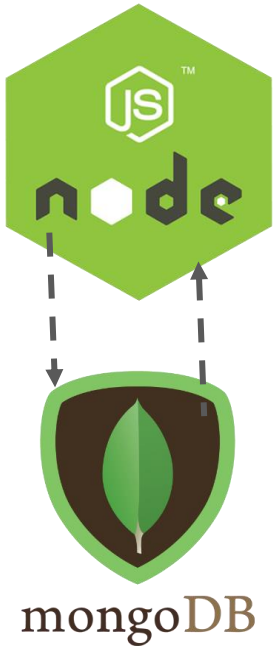
Access control/Authentication

Access control/authentication is not enabled by default for MongoDB. It should not be exposed to a public network

To enable access control:
https://docs.mongodb.com/manual/tutorial/enable-authentication/

NodeJS and MongoDB

NodeJS Libraries



To read and write to the MongoDB database from Node we'll be using the 'mongodb' library

- MongoDB NodeJS Driver
  - Provides CLI-like experience with NodeJS
  - `$ npm install mongodb`
- Object modeling tools
  - Provides schema-based solution to model application data, e.g., **prisma, mongoose**

We will focus on MongoDB NodeJS Driver in this course!

MongoDB NodeJS Driver (References)

https://www.mongodb.com/docs/drivers/node/current/

mongodb objects

The mongodb Node library provides objects to manipulate the connection, database, collections, and documents:

- `require('mongodb').MongoClient:` create connection to MongoDB
- `Db`: Database; can get collections using this object
- `Collection`: Can get/insert/delete documents from this collection via calls like `insertOne`, `find`, etc
- Documents are not special classes; they are just JavaScript objects

Getting a Db object

You can get a reference to the database object by using the `MongoClient.connect(`***url, callback***`)` function:

- ***url*** is the connection string for the MongoDB server
- ***callback*** is the function invoked when connected
  - ***client*** parameter: the connected client

```
const DATABASE_NAME = "englishDict";
const MONGO_URI = `mongodb://localhost:27017/${DATABASE_NAME}`;

let db = null;
MongoClient.connect(MONGO_URI, function(_error, client) {
    db = client.db;
});
```

Important: In MongoDB, a collection is not created until it gets content!

Connection string

```
const DATABASE_NAME = "englishDict";
const MONGO_URI = `mongodb://localhost:27017/${DATABASE_NAME}`;
```

- The URI is to a MongoDB server, which is why it begins with mongodb:// and not http://
- The MongoDB server is running on our local machine, which is why we use `localhost`
- The end of the connection string specifies the database name we want to use
  - If a database of that name doesn't already exist, it will be created the first time we write to it

MongoDB Connection string format

Important: In MongoDB, a database is not created until it gets content!

Callbacks and Promises

Every asynchronous MongoDB method has two versions:

- Callback
- Promise (+ async/await)

The Promise + async/await version is:

```
let client = null;
let db = null;
async function main() {
  client = await MongoClient.connect(MONGO_URI);
  db = client.db();
}
main();
```

Using a collection

```
async function main() {
  client = await MongoClient.connect(MONGO_URI);
  db = client.db();
  collection = db.collection('words');
}
main();
```

`const coll = db.collection(`***collectionName***`);`

- Obtains the collection object named ***collectionName*** and stores it in `coll`
- You do not have to create the collection before using it
  - It will be created the first time we write to it
- This function is **synchronous**

collection.insertOne (Callback)

`collection.insertOne(`***`doc, callback`***`);`

- Adds one item to the collection
- ***doc*** is a JavaScript object representing the key-value pairs to add to the collection
- The ***callback*** fires when it has finished inserting
  - The first parameter is an error object
  - The second parameter is a result object, where `result.`**`insertedId`** will contain the id of the object that was created

Callback version

```
function insertWord(word, definition) {
  const doc = {
    word: word,
    definition: definition
  };
  collection.insertOne(doc, function (err, result) {
    console.log(`Document id: ${result.insertedId}`);
  });
}
```

collection.insertOne (Promise)

```
const result = await collection.insertOne(doc);
```

- Adds one item to the collection
- ***doc*** is a JavaScript object representing the key-value pairs to add to the collection
- Returns a `Promise` that resolves to a `result` object when the insertion has completed
  - `result.`**`insertedId`** will contain the id of the object that was created

Promise version

```
async function insertWordAsync(word, definition) {
  const doc = {
    word: word,
    definition: definition
  };
  const result = await collection.insertOne(doc);
  console.log(`Document id: ${result.insertedId}`);
}
```

We will be using the Promise + async/await versions of all the MongoDB asynchronous functions, as it will help us avoid callback hell

collection.findOne

```
const doc = await coll.findOne(query, options);
```

- Finds the first item in the collection that matches the query
- ***query*** is a JS object representing which fields to match on. ***query*** can contains query operators
- Returns a `Promise` that resolves to a `document` object when `findOne` has completed
  - `doc` will be the JS object, so you can access a field via `doc.`***fieldName***, e.g. `doc._id`
  - If nothing is found, `doc` will be `null`

collection.findOne

```
async function printWord(word) {
  const query = {
    word: word
  };
  const response = await collection.findOne(query);
  console.log(
    `Word: ${response.word},
     definition: ${response.definition}`);
}
```

collection.find

`const cursor = await coll.find(query, options);`

- Returns a `Cursor` to pointing to the first entry of a set of documents matching the query
- You can use `hasNext` and `next` to iterate through the list:

```
async function printAllWordsCursor() {
  const cursor = await collection.find();
  while (await cursor.hasNext()) {
    const result = await cursor.next();
    console.log(`Word: ${result.word}, definition: ${result.definition}`);
  }
}
```

(This is an example of something that is **a lot** easier to do with `async/await`)

FindCursor methods

Methods

- [asyncIterator]
- addCursorFlag
- addListener
- addQueryModifier
- allowDiskUse
- batchSize
- bufferedCount
- clone
- close
- collation
- comment
- count
- emit
- eventNames
- explain
- filter
- forEach
- getMaxListeners
- hasNext
- hint
- limit
- listenerCount
- listeners
- map
- max
- maxAwaitTimeMS
- maxTimeMS
- min
- next
- off
- on
- once
- prependListener
- prependOnceListener
- project
- rawListeners
- readBufferedDocuments
- removeAllListeners
- removeListener
- returnKey
- rewind
- setMaxListeners
- showRecordId
- skip
- sort
- stream
- toArray
- tryNext
- withReadConcern
- withReadPreference
- getEventListeners
- listenerCount
- on
- once
- setMaxListeners

collection.find

```
const cursor = await coll.find(query, options);
const list = await cursor.toArray();
```

- Cursor also has a `toArray()` function that converts the results to an array

```
async function printAllWords() {
  const results = await collection.find().toArray();
  for (const result of results) {
    console.log(`Word: ${result.word}, definition: ${result.definition}`);
  }
}
```

Sort and projection with FindOptions

```
// Query for a movie that has the title 'The Room'
const query = { title: "The Room" };
const options = {
      // Sort matched documents
      //    in descending order by rating
      sort: { "imdb.rating": -1 },
      // Include only the `title` and `imdb` fields
      //    in the returned document
      projection: { _id: 0, title: 1, imdb: 1 },
};
const movie = await movies.findOne(query, options);
//const movie = await movies.find(query, options)
//                          .tryNext();
```

Sort and projection with FindCursor

```
// Query for a movie that has the title 'The Room'
const query = { title: "The Room" };
// Sort matched documents
//    in descending order by rating
const sort = { "imdb.rating": -1 };
// Include only the `title` and `imdb` fields
//    in the returned document
const projection = { _id: 0, title: 1, imdb: 1 };

const movie = await movies.find(query)
                          .project(projection)
                          .sort(sort)
                          .tryNext();
```

collection.replaceOne

```
await collection.replaceOne(
            query, newEntry, options);
```

- Replaces a single document matching ***query*** with ***newEntry***
- ***Option:*** `upsert: <boolean>.` When true, replaceOne() inserts the document from the replacement parameter if no document matches the ***query***.

```
async function replaceWord(word, definition) {
    const query = { word: word };
    const newEntry = { word: word, definition: definition };
    const result = await collection.replaceOne(query, newEntry);
    return result.modifiedCount;
}
```

collection.updateOne/Many

```
await collection.updateOne/Many(
                query, updateDoc, options);
```

- Update a single document/many documents matching ***query*** with ***updateDoc***
- ***UpdateDoc*** contains update operators

Option: `upsert: <boolean>.` When true, `updateOne()` inserts the document from the replacement parameter if no document matches the ***query***.

```
async function updateWord(word, definition) {
    const query = { word: word };
    const updateDoc = {
        $set: {
            definition: definition
        }
    };
    const result = await collection.updateOne(query, updateDoc);
    return result.modifiedCount;
}
```

collection.findOneAndUpdate

```
const result = await
      collection.findOneAndUpdate(
            query, updateDoc, options);
```

- Find a document and update it in one atomic operation
- Return the original (default) or updated document (*options.returnDocument = "after"*)
- *result.value*: the returned document

https://www.mongodb.com/docs/manual/reference/method/db.collection.findOneAndUpdate/

Upsert: insert or update

MongoDB also supports "upsert", which is

- Update the entry if it already exists
- Insert the entry if it doesn't already exist

Available in *replaceOne, updateOne, updateMany* methods

```
const options = { upsert: true };
await collection.updateOne/Many(
            query, updateDoc, options);
```

Upsert: insert or update

```
async function upsertWord(word, definition) {
    const query = { word: word };
    const updateDoc = {
        $set: {
            definition: definition,
        },
    };
    const options = { upsert: true };
    const result = await collection.updateOne(query, updateDoc, options);
    return result.modifiedCount;
}
```

`$set` operator: replaces the value of a field with a specified one

Update arrays in a document

```
// pizza collection
[{
   name: "Steve Lobsters",
   address: "731 Yexington Avenue",
   items: [
     { type: "pizza", size: "large",
             toppings: ["pineapple, ham"], },
     { type: "beverage", name: "Diet Pepsi", size: "16oz", },
   ],
 },
 // To be continued
```

Update arrays in a document

```
// pizza collection (continue)
 {
   name: "Popeye",
   address: "1 Sweethaven",
   items: [
     { type: "pizza", size: "large",
             toppings: ["garlic, spinach"], },
     { type: "calzone", toppings: ["ham"], },
   ],
 },
];
```

Update arrays in a document

```
const itemsToBeAdded = [
     { type: "beverage", name: "Water", size: "17oz", },
     { type: "pizza", size: "large", toppings: ["pepperoni"], }
,];

const query = { name: "Steve Lobsters" };
const addOneItem = {
      $push: { "items": itemsToBeAdded[0] }
};
const result = await pizza.updateOne(query, addOneItem);
```

`$push` operator: append a specified value to an array

Update arrays in a document

```
const itemsToBeAdded = [
     { type: "beverage", name: "Water", size: "17oz", },
     { type: "pizza", size: "large", toppings: ["pepperoni"], }
,];

const query = { name: "Steve Lobsters" };
const addManyItems = {
      $push: { "items": { $each: itemsToBeAdded } }
};
const result = await pizza.updateOne(query, addManyItems);
```

$push operator: append a specified value to an array
$each modifier: modify $push operator to append multiple items for array updates

Update arrays in a document

Positional operator: $

- Update the first array element of each document that matches your query
- Do not use the $ operator in an upsert call because the driver treats $ as a field name in the insert document

```
const query = { name: "Steve Lobsters", "items.type": "pizza" };
const updateDocument = {
      $set: { "items.$.size": "extra large" }
};
const result = await pizza.updateOne(query, updateDocument);
```

Update arrays in a document

Positional operator: $

- Update the first array element of each document that matches your query
- Do not use the $ operator in an upsert call because the driver treats $ as a field name in the insert document

```
{
  name: "Steve Lobsters",
  address: "731 Yexington Avenue",
  items: [
    { type: "pizza", size: "extra large", ... },
    ...
  ]
}
```

Update arrays in a document

All positional operator: $[]

- Update all of the array elements of each document that matches the query

```
const query = { "name": "Popeye" };
const updateDocument = {
   $push: { "items.$[].toppings": "fresh mozzarella" }
};
const result = await pizza.updateOne(query, updateDocument);
```

Update arrays in a document

All positional operator: $[]

- Update all of the array elements of each document that matches the query

```
{
  name:"Popeye",
  address: "1 Sweethaven",
  items: [
    { type: "pizza", ... ,
      toppings: ["garlic", "spinach", "fresh mozzarella"], },
    { type: "calzone", ... ,
      toppings: ["ham", "fresh mozzarella"], },
  ]
}
```

Update arrays in a document

Filtered positional operator: $[<identifier>]

- Update all embedded array elements of each document that matches the query
- To identify which array elements to match, use *params.arrayFilters*

Update arrays in a document

Filtered positional operator: $[<identifier>]

```
const query = { name: "Steve Lobsters" };
const updateDocument = {
     $push: { "items.$[orderItem].toppings": "garlic" }
};
const options = {
    arrayFilters: [{
        "orderItem.type": "pizza",
        "orderItem.size": "large",
    }]
};
const result = await pizza.updateOne(
                        query, updateDocument, options);
```

Update arrays in a document

Filtered positional operator: $[<identifier>]

```
{
  name: "Steve Lobsters",
  address: "731 Yexington Avenue",
  items: [
    { type: "pizza", size: "large",
      toppings: ["pineapple", "ham", "garlic"], ...},
    ...
  ]
}
```

Check out the documentation for more array operators

collection.deleteOne/Many

```
const result = await
      collection.deleteOne/Many(query);
```

- Deletes the first item/all items matching ***query***
- `result.deletedCount` gives the number of docs deleted
- Use `collection.deleteMany({})` to delete everything

collection.deleteOne/Many

```
async function deleteWord(word) {
  const query = {
    word: word
  };
  const response = await collection.deleteOne(query);
  console.log(`Number deleted: ${response.deletedCount}`);
}
```

```
async function deleteAllWords() {
    const response = await collection.deleteMany({});
    console.log(`Number deleted ${response.deletedCount}`);
}
```

collection.findOneAndDelete

```
const result = await
      collection.findOneAndDelete(
            query, options);
```

- Find a document and delete it in one atomic operation
- Return the deleted document
- *result.value*: the returned/deleted document

Advanced queries

MongoDB has a very powerful querying syntax that we did not cover in these examples

For more complex queries, check out:

- Querying
  - Query selectors and projection operators

```
db.collection('inventory').find({ qty: { $lt: 30 } });
```

- Updating
  - Update operators

```
db.collection('products').updateOne(
  { sku: 'abc123' },
  { $inc: { quantity: -2, 'metrics.orders': 1 }})
```

Advanced queries

Search text

- $text query operator: performs a logical OR on each term separated by a space in the search string
- *The search field needs to be indexed*
- *Only one* text index can be created per collection but a text index can include several fields. Every text search queries all the fields specified in that index for matches

```
db.movies.createIndex({ title: "text" });
db.movies.createIndex({
      title: "text", fullplot: "text"
});
```

Advanced queries

```
// Search the phrase "star trek"
const query = { $text: {
      $search: "\"star trek\""
} };

// Return only the `title` of each matched document
const projection = { _id: 0, title: 1, };

// Find documents based on query and projection
const cursor = movies.find(query)
                     .project(projection);
```

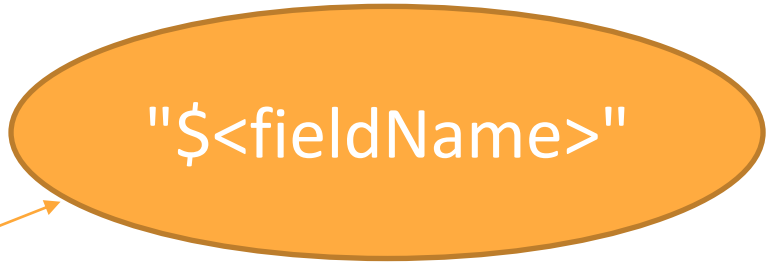
Advanced queries

Aggregation: produces reduced and summarized results in MongoDB

```
// restaurants collection
[   { stars: 3, categories: ["Bakery", "Sandwiches"],
              name: "Rising Sun Bakery" },
    { stars: 4, categories: ["Bakery", "Cafe", "Bar"],
              name: "Cafe au Late" },
    { stars: 5, categories: ["Coffee", "Bakery"],
              name: "Liz's Coffee Bar" },
    { stars: 3, categories: ["Steak", "Seafood"],
              name: "Oak Steakhouse" },
    { stars: 4, categories: ["Bakery", "Dessert"],
              name: "Petit Cookie" },]
```

Advanced queries

Aggregation: produces reduced and summarized results in MongoDB




```
// Count Bakery by stars
const pipeline = [
    { $match: { categories: "Bakery" } },
    { $group: { _id: "$stars", count: { $sum: 1 } } }
];
const aggCursor = restaurants.aggregate(pipeline);
for await (const doc of aggCursor) {
    console.log(doc);
}
```

```
// Output
{ _id: 4, count: 2 }
{ _id: 3, count: 1 }
{ _id: 5, count: 1 }
```

Transactions

In MongoDB, a write operation is **atomic** on the level of a **single** document

Multi-document ACID transactions are also supported from MongoDB version 4.0: https://www.mongodb.com/docs/drivers/node/current/fundamentals/transactions/

Relationships

An example

Assume that we have 2 entities: ***Identifier*** and ***Customer***

- One Customer has only one Identifier
- One Identifier belongs to only one Customer

```
// Identifier
{
      _id: "123xyz",
      cardCode: "BKD2019",
}
// Customer
{
      _id: "cus123",
      name: "bezkoder",
      age: 29,
      gender: "male"
}
```

Normalized data models

Relationships are described using **references**

```
// Identifier
{
        _id: "123xyz",
        cardCode: "BKD2019",
        customer_id: "cus123",
                // reference to customer document
}
// Customer
{
        _id: "cus123",
        name: "bezkoder",
        age: 29,
        gender: "male"
}
```

Embedded data models

```
// Customer
{
      _id: "cus123",
      name: "bezkoder",
      age: 29,
      gender: "male",
      identifier: {
          _id: "123xyz",
          cardCode: "BKD2019",
      }
}
```

Also known as
denormalized models

Normalization vs Denormalization

It depends on the **types of relationship** between collections, on data access patterns, and on data cohesion

- Types of relationship
  - One-to-one -> Denormalization
  - One-to-few -> Denormalization
  - One-to-many -> Normalization, array of references
  - One-to-squillions -> Normalization, parent-reference
  - Many-to-many -> Normalization

https://www.bezkoder.com/mongoose-one-to-many-relationship/#Model_One-to-Many_Relationships_in_MongoDB

Normalization vs Denormalization

It depends on the types of relationship between collections, on **data access patterns**, and on data cohesion

- Data access patterns
  - Mostly read? Denormalization
  - The data gets updated a lot? Normalization

Normalization vs Denormalization

It depends on the types of relationship between collections, on data access patterns, and on **data cohesion**

- Data cohesion: how much the collections are related
  - They intrinsically belong together? Denormalization
  - We frequently need to query both of collections on their own? Normalization

In general, when using a NoSQL database like MongoDB, **favor embedding** unless there is a compelling reason not to

Normalization vs Denormalization

We can use the **subset pattern** to address some potential problems with embedded data models:

- *Large documents*: split the collection into two collections. One collection contains the subset of data which is accessed the most frequently

- *The embedded field is unbounded*: only embed the data required by the application (i.e., a product collection containing the product's ten most recent reviews)

Join two collections

Use $lookup operator (aggregation)

```
// orders collection
[
   { "_id" : 1, "item" : "almonds", "price" : 12,
                  "quantity" : 2 },
   { "_id" : 2, "item" : "pecans", "price" : 20,
                  "quantity" : 1 },
]
```

Join two collections

Use $lookup operator (aggregation)

```
// inventory collection
[
   { "_id" : 1, "sku" : "almonds",
      "description": "product 1", "instock" : 120 },
   { "_id" : 2, "sku" : "bread",
      "description": "product 2", "instock" : 80 },
   { "_id" : 3, "sku" : "cashews",
      "description": "product 3", "instock" : 60 },
   { "_id" : 4, "sku" : "pecans",
      "description": "product 4", "instock" : 70 },
]
```

Join two collections

Use $lookup operator (aggregation)

```
// Left outer join of orders and inventory
const aggCursor = orders.aggregate([
    {
       $lookup: {
          from: "inventory",
          localField: "item",
          foreignField: "sku",
          as: "inventoryDocs"
       }
    }
]);

for await (const doc of aggCursor) {
    console.log(doc);
}
```

Join two collections

Use $lookup operator (aggregation)

```
// Output
{ _id: 1, item: 'almonds', price: 12, quantity: 2,
 inventoryDocs: [ { _id: 1, sku: 'almonds',
      description: 'product 1', instock: 120 } ] }

{ _id: 2, item: 'pecans', price: 20, quantity: 1,
 inventoryDocs: [ { _id: 4, sku: 'pecans',
      description: 'product 4', instock: 70 } ] }
```

Join two collections

$unwind operator (aggregation): deconstructs an array field from the input documents to output a document for each element

```
// Left outer join of orders and inventory
const aggCursor = orders.aggregate([
    {
       $lookup: {
              ...
              as: " inventoryDocs"
        }
    },
    { $unwind: "$ inventoryDocs" }
]);
```

Join two collections

`$unwind` operator (aggregation): deconstructs an array field from the input documents to output a document for each element

```
// Output
{ _id: 1, item: 'almonds', price: 12, quantity: 2,
 inventoryDoc: { _id: 1, sku: 'almonds',
      description: 'product 1', instock: 120 } }

{ _id: 2, item: 'pecans', price: 20, quantity: 1,
 inventoryDoc: { _id: 4, sku: 'pecans',
      description: 'product 4', instock: 70 } }
```

Check out the documentation for more $lookup examples

Questions?