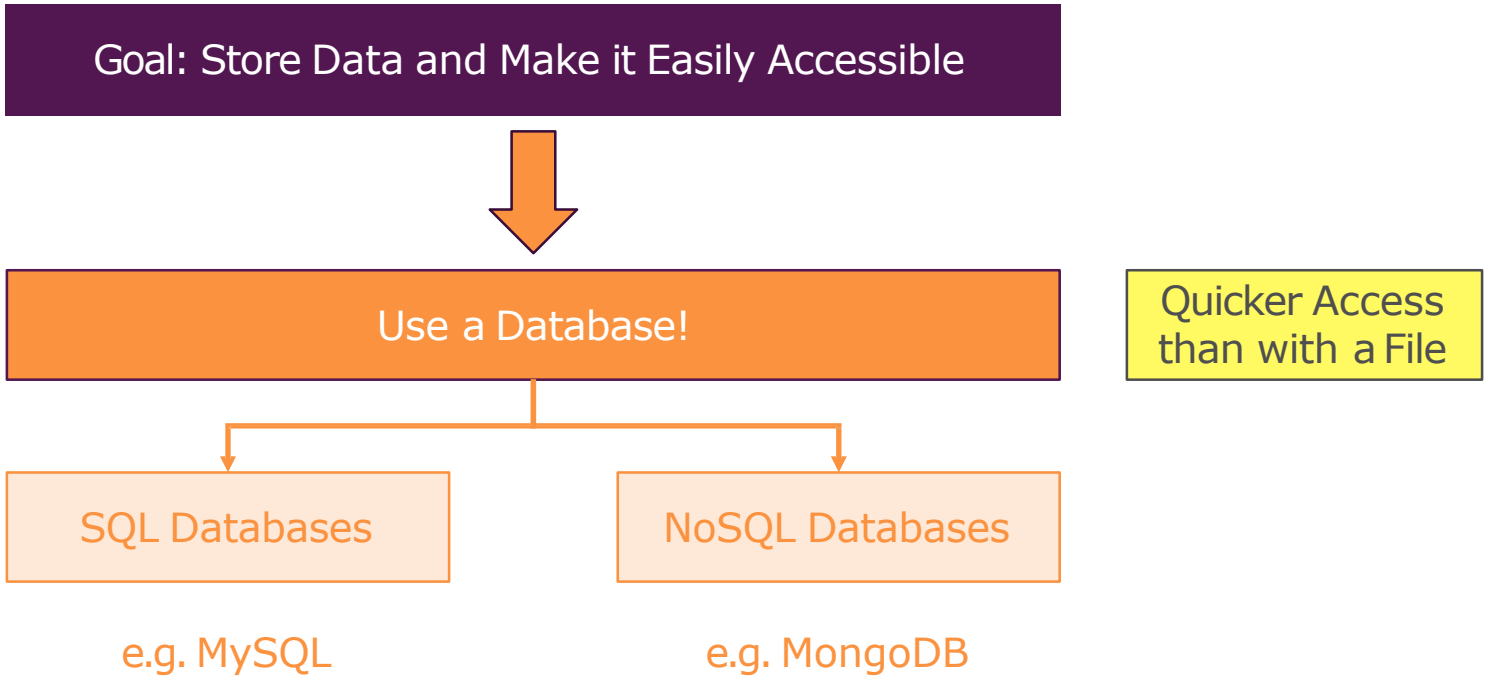
MongoDB – Intro & CRUD

SQL vs NoSQL


Goal: Store Data and Make it Easily Accessible
Use a Database!
Quicker Access than with a File
SQL Databases
NoSQL Databases
e.g. MySQL
e.g. MongoDB

What's SQL?

User

Id	Email	Name
1	josh@miu.edu	Josh Edward
2	emma@miu.edu	Emma Smith
3	…	…

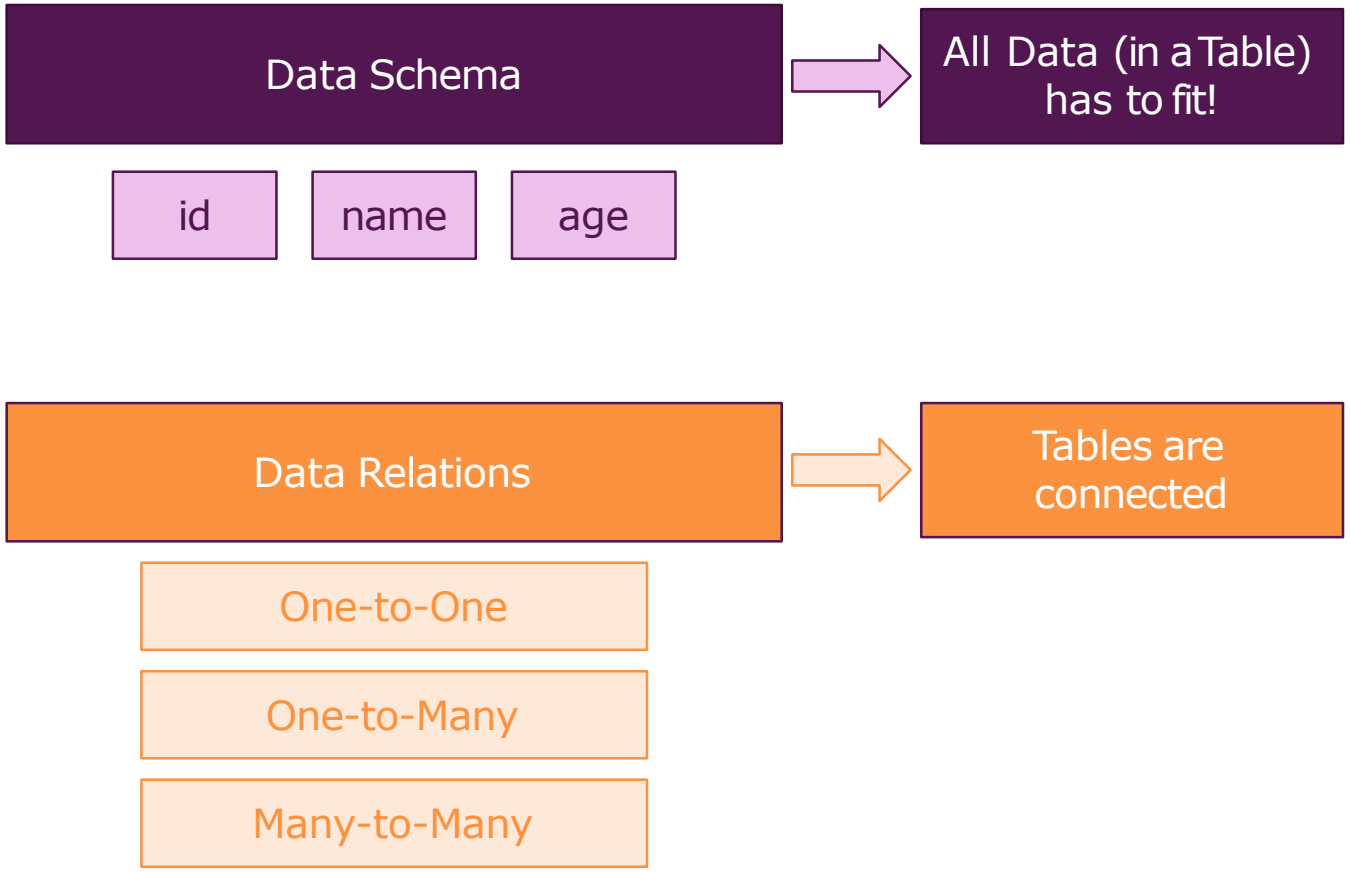
Order

Id	user_id	product_id
1	1	2
2	1	1
3	2	2

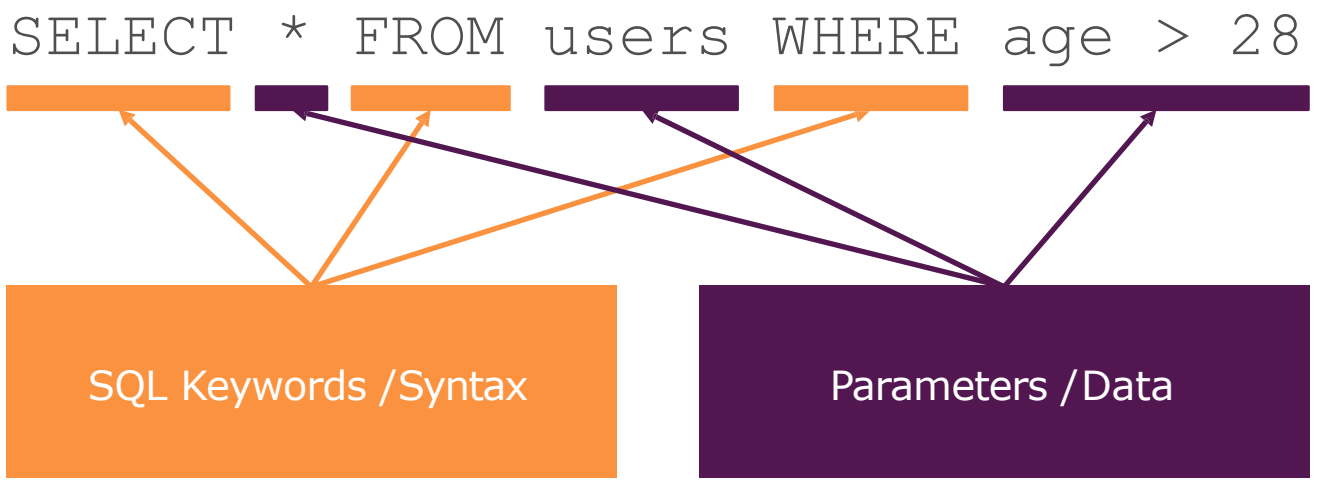
Product

Id	Title	Price	Description
1	Node.js	10	Good
2	Angular	20	Great
3	React.js	20	Great

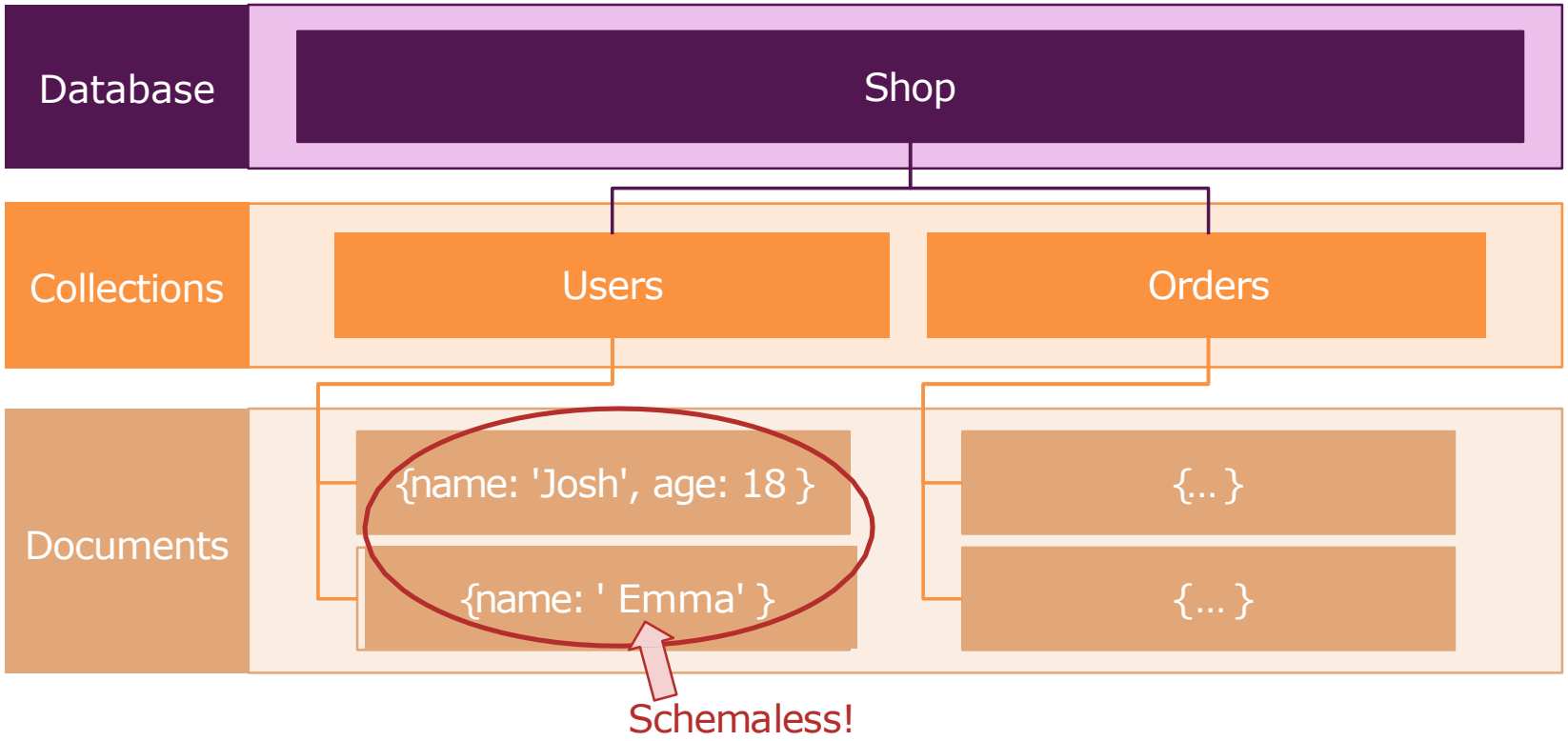
Core SQL Database Characteristics



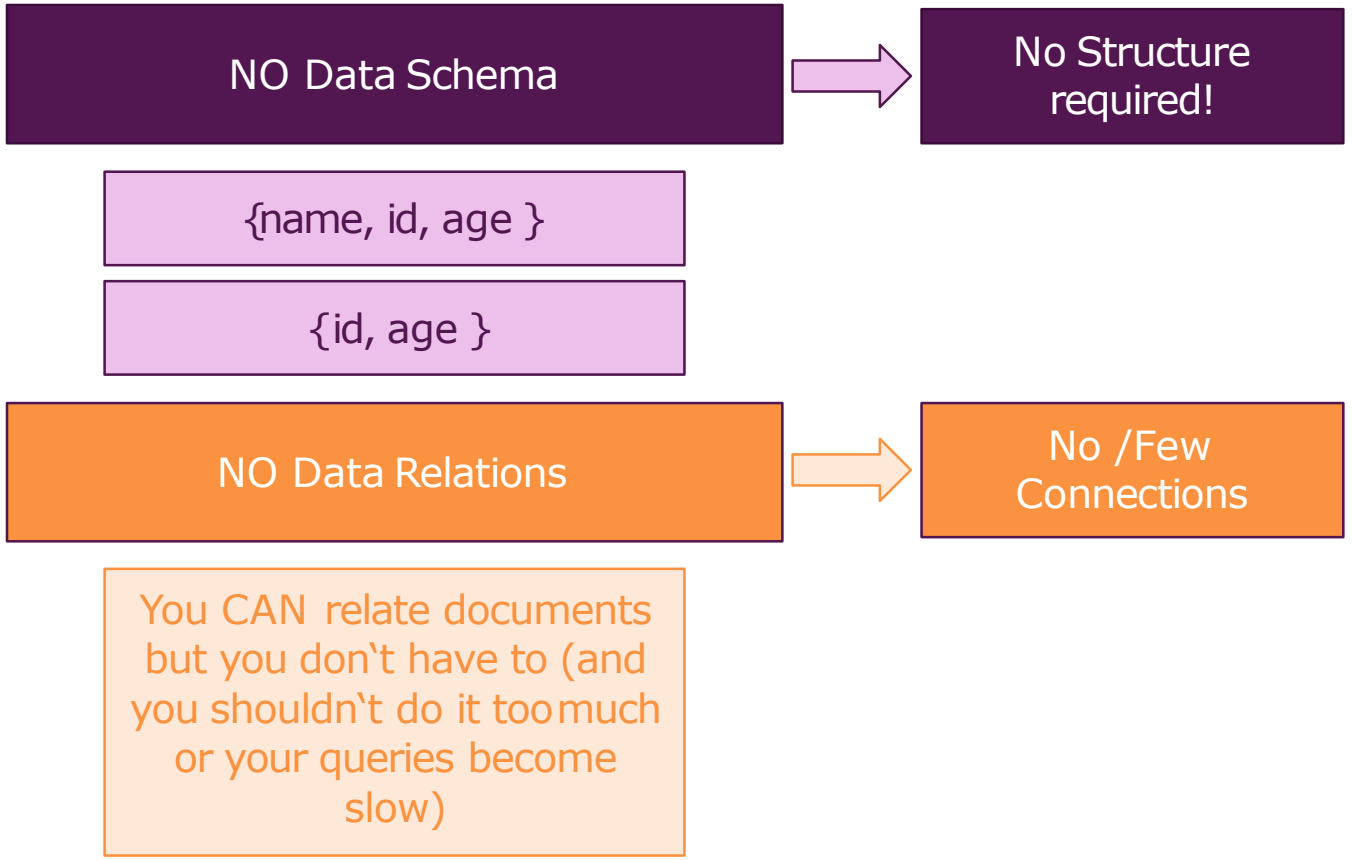
SQL Queries


SELECT * FROM users WHERE age > 28
SQL Keywords /Syntax
Parameters /Data

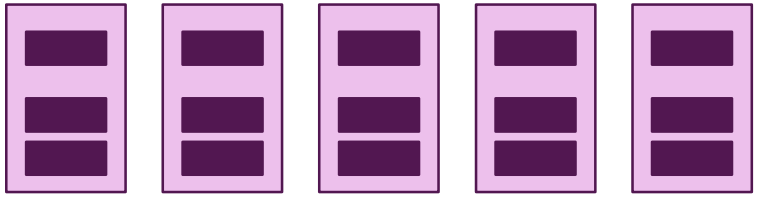
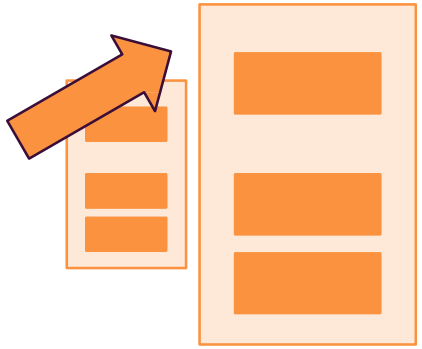
NoSQL


Database
Shop
Collections
Users
Orders
Documents
{name: 'Josh', age: 18 }
{name: ' Emma' }
{… }
{… }
Schemaless!

NoSQL Characteristics



Horizontal vs Vertical Scaling

Horizontal Scaling	Vertical Scaling
	
Add More Servers (and mergeData into one Database)	Improve Server Capacity /Hardware

SQL vs NoSQL

SQL	NoSQL
Data uses Schemas	Schema-less
Relations!	No (or very few) Relations
Data is distributed across multiple tables	Data is typically merged /nested in a few collections
Horizontal scaling is difficult / impossible; Vertical scaling is possible	Both horizontal and vertical scaling is possible
Limitations for lots of (thousands) read & write queries per second	Great performance for mass read & write requests

NoSQL Revolution

- NoSQL (originally referring to "non SQL" or "non relational") databases were created for "Big Data" and Real-Time Web Applications, it provides new data architectures that can handle the ever-growing velocity and volume of data.

Name	Year	Type	Developer
MongoDB	**2008**	**Document**	**10Gen**
CouchDB	2005	Document	Apache
Cassandra	2008	Column Store	Apache
CouchBase	2011	Document	Couchbase
Riak	2009	Key-Value	Basho Technologies
SimpleDB	2007	Document	Amazon
BigTable	2015	Column Store	Google
Azure Cosmos DB	2017	Multi-Model	Microsoft

What is MongoDB?

- MongoDB is an open-source document database that provides high performance, high availability, and automatic scaling.
- Non relational DB, stores BSON documents.
- Schemaless: Two documents don't have the same schema.

BSON

- BSON, short for Binary JSON, is a binary-encoded serialization of JSON-like documents.
- Both JSON and BSON support Rich Documents (embedding documents and arrays within other documents and arrays).
- BSON also contains extensions that allow representation of data types that are not part of the JSON spec. (For example, BSON has a BinData ObjectId, 64 bits Integers and Date type…etc)

BSON characteristics

- Lightweight
  - Keeping spatial overhead to a minimum is important for any data representation format, especially when used over the network.
- Traversable
  - BSON is designed to be traversed easily. This is a vital property in its role as the primary data representation for MongoDB.
- Efficient
  - Encoding data to BSON and decoding from BSON can be performed very quickly in most languages. For example, integers are stored as 32 (or 64) bit integers and they don't need to be parsed to and from text.

Non-Relational

- Scalability and Performance *(embedded data models reduces I/O activity on database system)*
- Depth of Functionality *(Aggregation framework, Text Search, Geospatial Queries)*
- To retains scalability
  - MongoDB **does not support Joins** between two collections *($loopup)*
  - **No relational algebra:** tables/columns/rows *(SQL)*
  - **No Transactions** across multiple collections *(Do it programmatically, documents can be accessed atomically)*

Schema

- By default, a collection does not require its documents to have the same schema, the documents in a single collection do not need to have the same set of fields and the data type for a field can differ across documents within a collection.
- Starting of MongoDB 3.2, you can enforce document validation rules for a collection during update and insert operations

Document Structure

- The value of a field can be any of the BSON data types, including other documents, arrays, and arrays of documents.

```
const doc = {
    _id: new ObjectID('5e44ab7638d4f738f05c57a8'),
    name: { first: "Josh", last: "Edward" },
    birth: new Date('Oct 31, 1979'),
    email: "test@mim.edu",
    phones: ["6414511111", "6414512222"]
}
```

Setup

- Follow the link to install MongoDB
  - https://docs.mongodb.com/manual/tutorial/install-mongodb-on-windows/
- Two ways to start your MongoDB
  - as a Windows Service
    1. From the Services console, locate the MongoDB service.
    2. Right-click on the MongoDB service and click **Stop** (or **Pause**).
  - from the Command Interpreter
    1. Create database directory - C:/data/db
    2. Start your MongoDB database.
       - "C:\Program Files\MongoDB\Server\4.2\bin\mongod.exe" --dbpath="c:\data\db"
    3. Connect to MongoDB
       - "C:\Program Files\MongoDB\Server\4.2\bin\mongo.exe"

Collections

- MongoDB stores documents in collections. (Collections are similar to tables in relational databases)

```
use myDB
```

- If a database/collection does not exist, MongoDB creates the db/collection when you first store data for that collection

```
use myNewDB
db.myNewCollection.insert( { x: 1 } )
```

- *The insert() operation creates both the database myNewDB and the collection myNewCollection if they do not already exist.*

Shell Demo

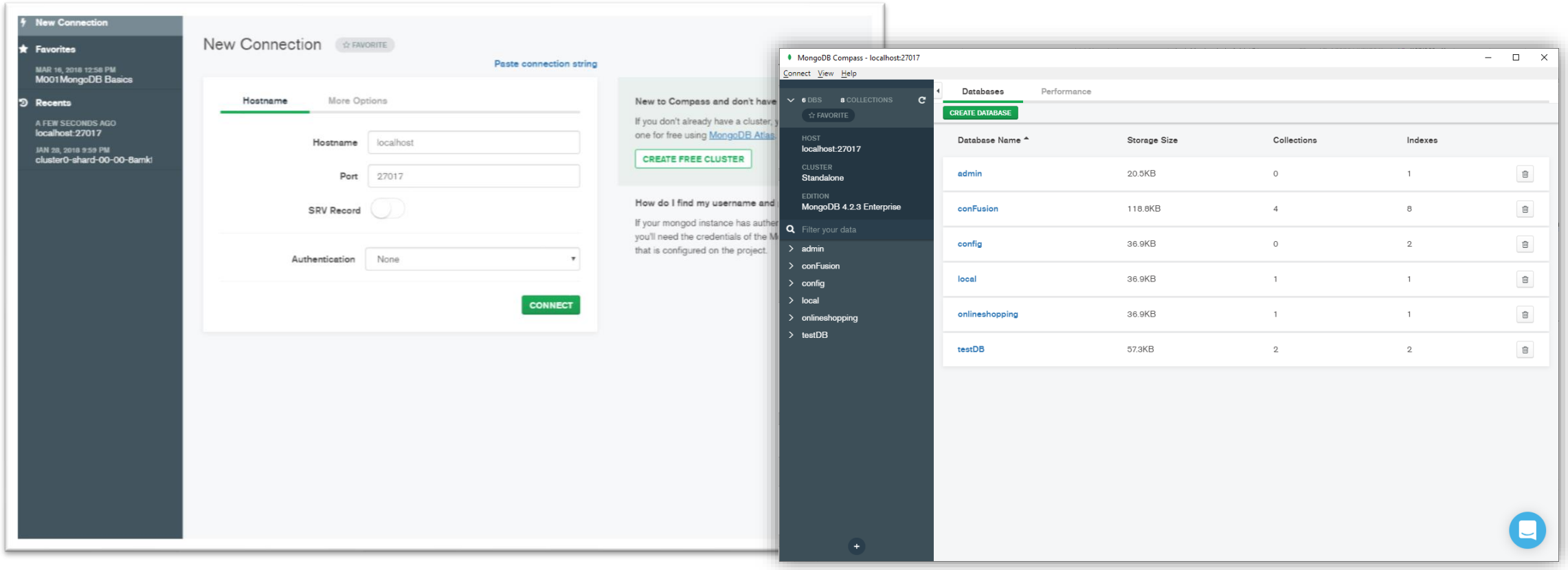
```
show dbs
use testDB // switch or create
show collections
db.testCol.insert({"name": "Josh"})// db var refers to the current database
db.testCol.find() // notice _id
// passing a parameter to find a document that has a property "name" and value "Josh"
db.testCol.find({"name":"Josh"})
// save() = upsert if _id provided
db.testCol.save({"name":"Mike"})
// insert 10 documents - Shell is C++ app that uses V8
for (var i=0; i<10; i++){ db.testCol.insert({"x": i}) }
```

Shell Demo

```
db.testCol.save({a:1, b:2})
db.testCol.save({a:3, b:4, fruit: ["apple", "orange"] })
db.testCol.save({name: "Josh", address: {city: "Fairfield",
                                         zip: 52557,
                                         street: "1000 N 4th street"}
})
// show documents in a nice way, it will only work when you
have nested or larger documents:
db.testCol.find().pretty()
```

MongoDB Compass

- As the GUI for MongoDB, MongoDB Compass allows you to make smarter decisions about document structure, querying, indexing, document validation, and more. Commercial subscriptions include technical support for MongoDB Compass.



General Rules

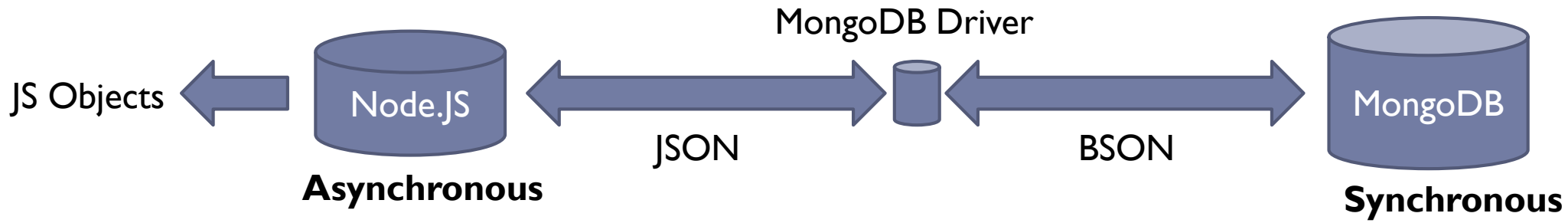
- Field names are strings.
- The field name `_id` is reserved for use as a primary key. It is immutable and always the first field in the document. It may contain values of any BSON data type, other than an array.
- The field names cannot start with the dollar sign ($) character and cannot contain the dot (.) character or `null`. Field names cannot be duplicated.
- The maximum BSON document size is 16 megabytes. *(To store documents larger than the maximum size, MongoDB provides the GridFS API)*

MongoDB Driver

- A library written in JS to handle the communication, open sockets, handle errors and talk with MongoDB Server.

```
npm install mongodb
```

- Note that Mongo Shell is **Synchronous** while Node.JS is **Asynchronous**.



Connect to MongoDB – 3.0+

```
const MongoClient = require('mongodb').MongoClient;

MongoClient.connect('mongodb://localhost:27017', { useUnifiedTopology: true })
    .then(client => {
        console.log('Connected......');
        const db = client.db('testDB');
        db.collection('testCol').find().each(function(err, doc) {
            if (err) throw err;
            // Print the result.
            // Will print a null if there are no documents in the db.
            console.log(doc);
            // Close the DB
            client.close();
        });
    })
    .catch(err => console.log('Error: ', err));
```

In Real Application… Like this?

util/database.js

```
const mongodb = require('mongodb');
const MongoClient = mongodb.MongoClient;

const mongoConnect = (callback) => {
    MongoClient.connect('mongodb://localhost:27017')
        .then(client => {
            console.log('Connected......');
            callback(client);
        })
        .catch(err => console.log(err));
}

module.exports = mongoConnect;
```

models/product.js

```
const mongoConnect = require('../util/database');

class Product {
…
 save() {
   mongoConnect((client) => {
       client.db('onlineshopping').collection('products')
                .insertOne(this)
                .then(result => console.log(result))
                .catch(err => console.log(err));
        });
    }}
```

In Real Application…

util/database.js

```
const mongodb = require('mongodb');
const MongoClient = mongodb.MongoClient;
let _db; //indicate private variable
const mongoConnect = (callback) => {
    MongoClient.connect('mongodb://localhost:27017',
        { useUnifiedTopology: true })
        .then(client => {
            console.log('Connected......');
            _db = client.db('testCol');
            callback();
        })
        .catch(err => console.log(err));
}
const getDb = () => {
    if (_db) {
        return _db;
    }
    throw new Error('No Database Found!');
}
exports.mongoConnect = mongoConnect;
exports.getDb = getDb;
```

app.js

```
const mongoConnect = require('./util/database').m
ongoConnect;
mongoConnect(() => {
    app.listen(3000);
});
```

models/product.js

```
const getDb = require('../util/database').getDb;

class Product {
…
  save() {
    const db = getDb();
    db.collection('products')
      .insertOne(this)
      .then(result => console.log(result))
      .catch(err => console.log(err));
  }
}
```

Using MongoDB CRUD operations in Node/Express

<code examples>

Example - Using findOne()

```
const mongodb = require('mongodb');
const MongoClient = mongodb.MongoClient;

MongoClient.connect('mongodb://localhost:27017', { useUnifiedTopology: true })
    .then(client => {
        console.log('Connected......');

        const db = client.db('onlineshopping');

        db.collection('products').findOne({ 'title': 'First book' }, function (err, doc) {
            if (err) throw err;
            // Print the result.
            // Will print a null if there are no documents in the db.
            console.log(doc);
            // Close the DB
            client.close();
        });
    })
    .catch(err => console.log(err));
```

console.dir vs console.log

`console.log()` only prints out a string, whereas `console.dir()` prints out a navigable object tree

Example - Using find() with query & projection

```
const MongoClient = require('mongodb').MongoClient;
const { ObjectID } = require('mongodb');

MongoClient.connect('mongodb://localhost:27017', { useUnifiedTopology: true }, function (err,
client) {
    if (err) throw err;

    const db = client.db('onlineshopping');

    const query = { _id: new ObjectID('5e44ab7638d4f738f05c57a8') };
    const projection = { title: 1, imageUrl: 1, _id: 0 };

    db.collection('products').find(query).project(projection).toArray(function (err, docArr) {
        if (err) throw err;
        docArr.forEach(function (doc) {
            console.log(doc);
        });
        client.close();
    });
});
```

Note: Projection is a good practice to save bandwidth and retrieve only the data we need.

sort(), limit(), skip()

- Similar to SQL language, MongoDB provides certain methods on the collection object, they work as instructions sent to DB to affect the retrieval of data, all these methods will return a cursor back (chain):

SQL	MongoDB Method
`Order by`	`sort()`
`Limit`	`limit()`
`Skip`	`skip()`

Note: These will set instructions to DB server to process the information before its being sent to client. No processing will ever happen at the client side.

Example - Skip, Limit and Sort

```
const MongoClient = require('mongodb').MongoClient;

MongoClient.connect('mongodb://localhost:27017', { useUnifiedTopology: true }, function (er
r, client) {
    if (err) throw err;

    const db = client.db('onlineshopping');

    const query = {};
    const projection = { title: 1, imageUrl: 1, _id: 0, price: 1 };

    db.collection('products').find(query).project(projection)
        .skip(1).limit(2).sort('price', 1).toArray(function (err, docArr) {
            if (err) throw err;
            docArr.forEach(function (doc) {
                console.log(doc);
            });

            client.close();
        });
});
```

Note: These will be implemented in the DB in a very specific order: **1. sort, 2. skip, 3. limit** no matter how we put them in the code *(remember cursor object is being built and sent to the server only , to run call* `.each()`*)*

Example - Using insertOne()

```
const MongoClient = require('mongodb').MongoClient;

MongoClient.connect('mongodb://localhost:27017', { useUnifiedTopology: true }
    , function (err, client) {
        if (err) throw err;
        const db = client.db('testDB');
        var doc = { 'student': 'John', 'grade': 100 };

        db.collection('testCol').insertOne(doc, (err, docInserted) => {
            if (err) throw err;

            console.log(`Success: ${JSON.stringify(docInserted)}!`);
            return client.close();
        });
    });
```

Example - Using insert() multiple docs

```
const MongoClient = require('mongodb').MongoClient;

MongoClient.connect('mongodb://localhost:27017', { useUnifiedTopology: true },
    function (err, client) {
        if (err) throw err;
        const db = client.db('testDB');
        const docs = [{ 'student': 'Calvin', 'grade': 90 },
        { 'student': 'Susie', 'grade': 95 }];

        db.collection('testCol').insertMany(docs, (err, docInserted) => {
            if (err) throw err;

            console.log(`Success: ${JSON.stringify(docInserted)}!`);
            return client.close();
        });
    });
```

Example - Using update()

```
const MongoClient = require('mongodb').MongoClient;

MongoClient.connect('mongodb://localhost:27017', { useUnifiedTopology: true }
    , function (err, client) {
        if (err) throw err;
        const db = client.db('testDB');
        let query = { student: 'John' };

        db.collection('testCol').findOne(query, (err, doc) => {
            query['_id'] = doc['_id'];
            doc['registration_date'] = new Date(); //add another field

            db.collection('testCol').update(query, doc, function (err, numUpdated) {
                if (err) throw err;
                console.dir("Successfully updated " + numUpdated);
                client.close();
            });

        });
    });
```

Example - Using deleteOne()

```
const MongoClient = require('mongodb').MongoClient;

MongoClient.connect('mongodb://localhost:27017', { useUnifiedTopology: true }
    , function (err, client) {
        if (err) throw err;

        const db = client.db('testDB');
        var query = { 'student': 'Susie' };
        // remove all documents that have 'student' value is 'Susie'
        db.collection('testCol').deleteOne(query, function (err, result) {
            console.log("Result: " + JSON.stringify(result));
            return client.close();
        });
    });
```

Resources

- SQL vs NoSQL: https://academind.com/learn/web-dev/sql-vs-nosql/
- Mongo Shell: https://docs.mongodb.com/manual/mongo/
- MongoDB CRUD Operations: https://docs.mongodb.com/manual/crud/
- Node.js MongoDB Driver API: https://mongodb.github.io/node-mongodb-native/3.5/api/

Homework

- Update online shopping application, change CRUD operations on Product Model to use MongoDB.
  - Admin: save/edit/delete product, view all products
  - Shop: view detail of product, view all products