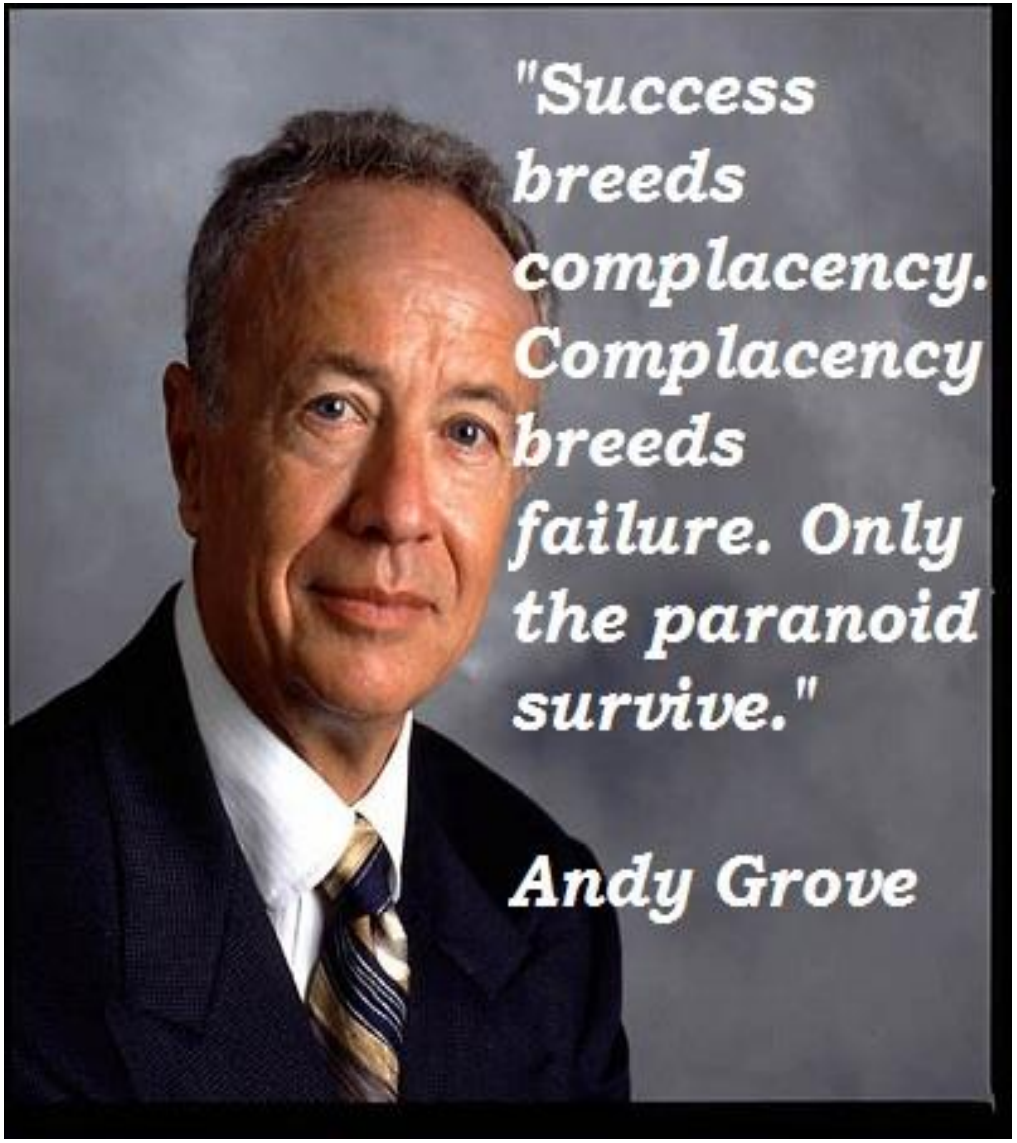
Introduction to MongoDB


"Success breeds complacency. Complacency breeds failure. Only the paranoid survive."
Andy Grove

Why NoSql ?

- Relational databases are not designed to scale
- schema, joins

But people want:

- Scale
- Speed
- Cloud
- New data

C and Latency Tradeoff

- Amazon claims that just an extra one tenth of a second on their response times will cost them 1% in sales.
- Google said they noticed that just a half a second increase in latency caused traffic to drop by a fifth.

What is NoSQL?

- non-relational
- simple API
- schema-free
- open-source
- horizontally scalable (sharding)
- replication support
- eventually consistent /BASE

Different types of NoSQL Databases

- NoSQL database are classified according to their data storage models:
  - Column (Cassandra)
  - Document (MongoDB)
  - Key – value Pair(Dynamo – Amazon)
  - Graph

MongoDB

- Name derived from Hu(**MONGO)**us word
- Document Oriented Database
- Built for High – Performance and scalability
- Document based queries for **Easy Readability**
- Replication and failover for **High Availability**
- Auto Sharding for **Easy Scalability**

Comparison between RDBMS and NoSQL DB

- Example: Class
- Location
- Presenter
  - Presenting at a location
- People
  - Potential attendees in context of a class
- Class
  - Presenter in location with people as actual attendees

Relational Database: Example

- Class schema in a relational database
- Presentation { id, name, location}
- People {id, name}
- Address {id, city, state, zip}

Schema for this class in a relational database model

Presentation

id	name	*location*
1	Chris	SJSU

Address

id	city	state
SJSU	San Jose	CA

People

id	name
10	Simon
11	Chris

Class

id	*person*	*presentation*
20	10	1
20	11	1

Relational database: Example

```
CREATE TABLE Presentation (
        id Integer primary key, name String, location string,
        FOREIGN KEY (location) REFERENCES Address(id));
CREATE TABLE Address (
        id String primary key, city String, state String);
CREATE TABLE People (
        id Integer primary key, name String);
CREATE TABLE Class (
        id Integer, person Integer, presentation Integer,
        PRIMARY KEY (id, person, presentation),
        FOREIGN KEY (person) REFERENCES People(id),
        FOREIGN KEY (presentation) REFERENCES Presentation(id));
```

Relational database: Example

```
select Presentation.name, Presentation.location,
        Address.city, Address.state, People.name
from Presentation, Address, People, Class
where Class.person = People.id
            and Class.presentation = Presentation.id
            and Presentation.location = Address.id;
```

```
| name | location | city | state | name |
+--------+--------+------------+-------+---------+
| Chris | SJSU | San Jose | CA   | Simon|
| Chris | SJSU | San Jose | CA   | Chris  |
```

Relational Database: Recap

1. Schema design

    Primary key (underlined) and foreign key (cursive) constraints

2. Table creation

    DDL

3. Data insertion for each table

    DML

4. Query: join

    DML

5. Data structure creation within application system

    JDBC resultset to e.g. Java objects

NoSQL Database: Use Case Example

```
use course /* database will be created if not present */
db.presentation.insert(
{"id": 1,
  "name": “Simon",
  "location": {"id": "SJSU",
                    "city": "San Jose",
                    "state": "CA"
                  },
"people": [{"id": 10, "name": "Simon"},
                 {"id": 11, "name": "Chris"}
                ]
})
```

NoSQL Database: Use Case Example

- db.presentation.find()
- db.presentation.find({"id": 1})

NoSQL Database: Recap

1. ~~Schema design~~

    Primary key (underlined) and foreign key (cursive) constraints

2. Table creation

    DDL

3. Data insertion for each table

    DML

4. Query: ~~join~~

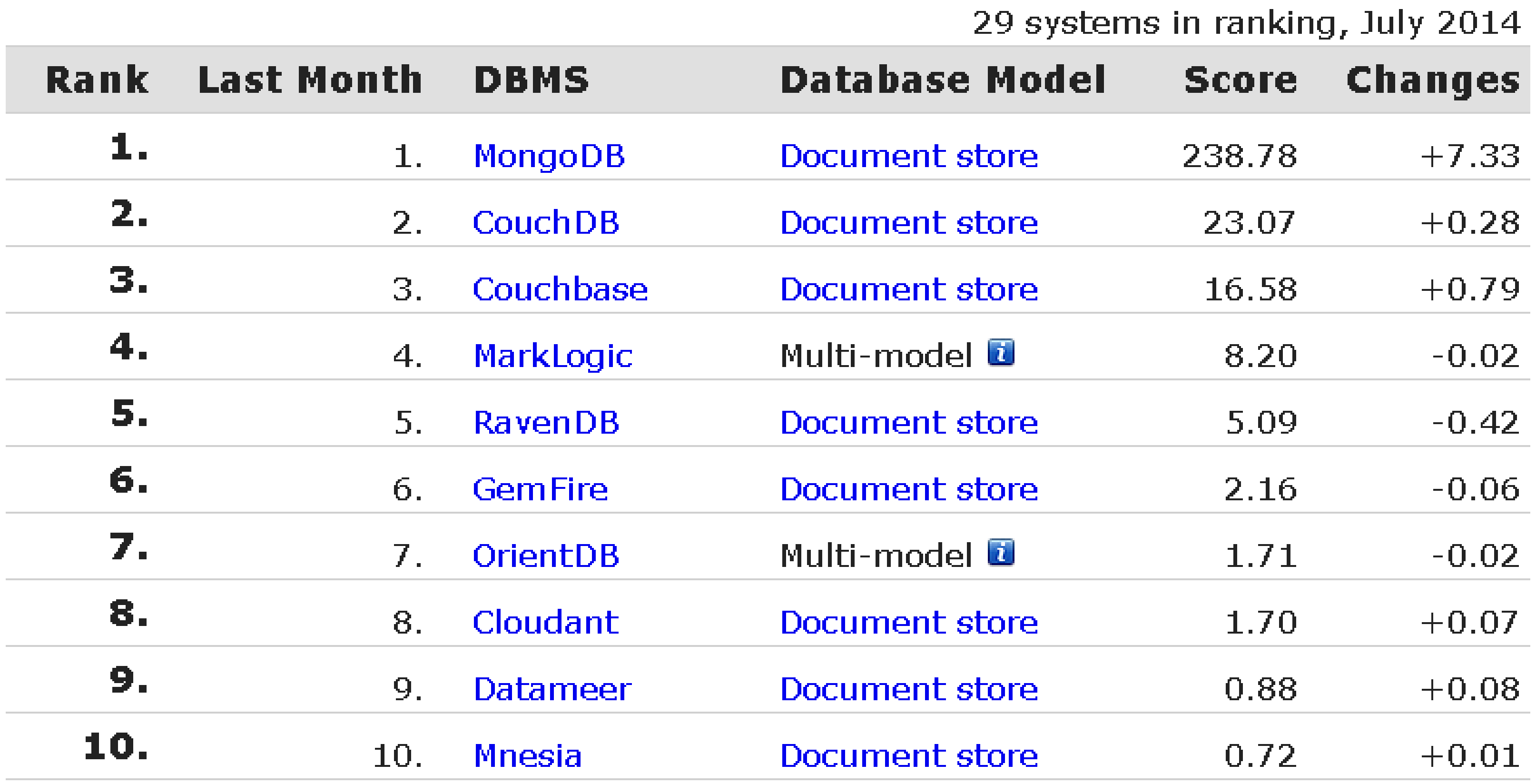
    DML

~~5. Data structure creation within application system~~

~~JDBC resultset to e.g. Java objects~~

NoSQL Database: Major Players

- Too many document NoSQL databases to name a few distinct ones



29 systems in ranking, July 2014

Rank	Last Month	DBMS	Database Model	Score	Changes
1.	1.	MongoDB	Document store	238.78	+7.33
2.	2.	CouchDB	Document store	23.07	+0.28
3.	3.	Couchbase	Document store	16.58	+0.79
4.	4.	MarkLogic	Multi-model	8.20	-0.02
5.	5.	RavenDB	Document store	5.09	-0.42
6.	6.	GemFire	Document store	2.16	-0.06
7.	7.	OrientDB	Multi-model	1.71	-0.02
8.	8.	Cloudant	Document store	1.70	+0.07
9.	9.	Datameer	Document store	0.88	+0.08
10.	10.	Mnesia	Document store	0.72	+0.01

Key Benefit of NoSQL: O(1) Lookup

- Fast lookup
  - No joining required
  - All data about one domain concept in one document
- Direct programming language representation
  - No mapping or ‘ORM’ layer required
- JSON library
  - Direct result representation and manipulation
  - JavaScript: representation in language data types directly
  - E.g., check out MongoDB node.js driver

Key Problem of NoSQL: No Join Operator

- Many NoSQL databases do not implement a join query operator
  - If you need to join data, then you have to do it in the application system layer
- But, wait a moment ...
  - Is it ever necessary to join data in NoSQL databases?
  - Some claim: not necessary due to support of
    - Sub-documents
    - Arrays (lists)
- Let’s look at an example
  - Supplier - Parts

Key Problem of NoSQL: No Join Operator

- Example
  - Supplier - Parts relationship (N:M)
  - Each supplier supplies many parts
  - Each part supplied by many suppliers
- Relational DBMS
  - “Supplier” table
  - “Part” table
  - “Supplies” relationship in table

Key Problem of NoSQL: No Join Operator

Supplier - Part - Supplies

```
| Supplier |      | Part |     | Supplies |
+------------+     +------------+    +------------------------+
| id | name|    | id | name| |supplier_id|part_id |
+----+---------+  +----+-------+  +------------+----------+
| 10 | Supp1 |      | 20 | Part1 |   | 10          |      20     |
| 11 | Supp2 |      | 21 | Part2 |   | 10          | 21       |
                                      | 11       | 20       |
```

Key Problem of NoSQL: No Join Operator

Supplier - Supplies – Part

```
{ "id": 10,
"name": "Supp1",
"supplies": [{"id": 20,"name":"Part1"},
                  {"id": 21, "name":
"Part2"}]}

{ "id": 11,
"name": "Supp2",
"supplies": [{"id": 20,"name":"Part1"}]}
```

Supplier - Supplies – Part

```
{ "id": 10,
"name": "Supp1",
"supplies": [20, 21]}
{ "id": 10,
"name": "Supp1",
"supplies": [20, 21]}

{"id": 20, "name": "Part1"}
{"id": 21, "name": "Part2"}
```

Why use MongoDB?

- MongoDB stores data in Objects
- Uses BSON (Binary JSON)
- No Joins
- No Complex Queries
- Embedded Documents and arrays reduce the need for joins
- No multi-document transactions

Where to use MongoDB ?

- Ideal for Web Applications
- Applications containing semi-structured data and need flexible schema management
- Caching and High Scalability
- Scenarios where **data availability** and **size of data** are priorities over the **transactions** of data

When to not use MongoDB?

- ACID properties are important for storage
- Highly Transactional Applications (Banking domain, Security)
- Problems and applications requiring Joins and complex queries

Key Problem of NoSQL: No Database-Enforced Consistency

- Not enforced
    - Primary key
    - Foreign key
    - Enumeration
    - Cascading delete
    - etc.
- Enforcement can be accomplished
    - When
        - reading or writing
    - In application system code
    - In self-implemented database access layer
    - In separate consistency check process
    - Not at all

How does MongoDB Store data?

- Stores data in form of Documents
- JSON like field – value pair
- Documents analogous to structures in programming languages with key – value pair
- Documents stored in **BSON (Binary JSON)** format
- BSON is JSON with additional type information

NoSQL: Key Insights

- Specialized data models
    - Not universal, but optimized towards special cases
- Specialized query access
    - Not universal, but optimized towards special cases
- Different / absent consistency supervision
    - Relaxed constraints

- Trade-off
    - Gain through specialization
    - Implementation of missing functionality outside of database

CAP Theorem: Theory

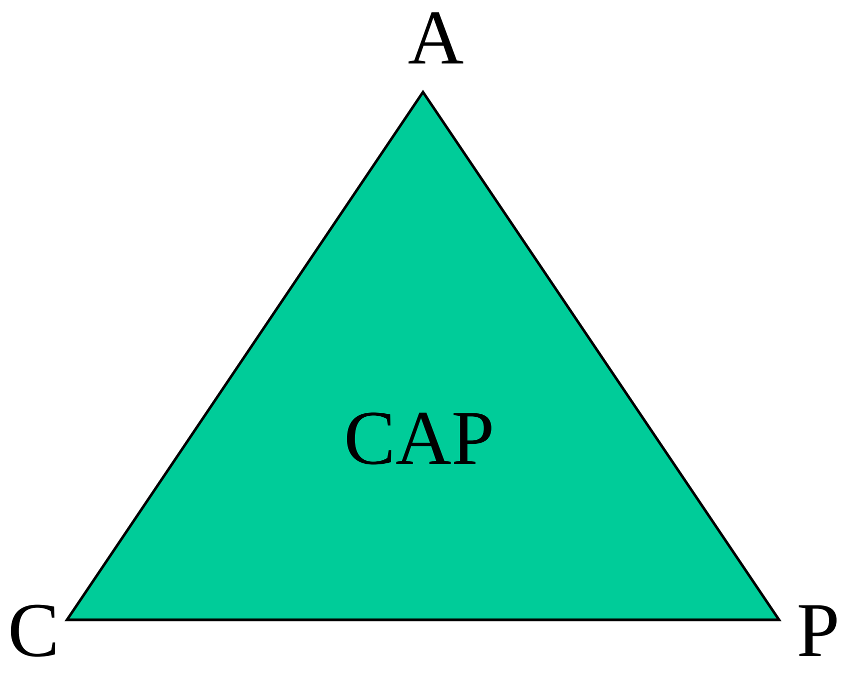
The **CAP theorem** states that it is impossible for a distributed computer system to simultaneously provide all three of the following guarantees:

- *Consistency*
- *Availability*
- *Partition tolerance*

CAP Theorem

- Consistency
- Availability
- Partition Tolerance

• Choose two



Questions?

- Which one would you choose when network partition? (a) C (b) A
- Which of CAP is essential for a distributed system?

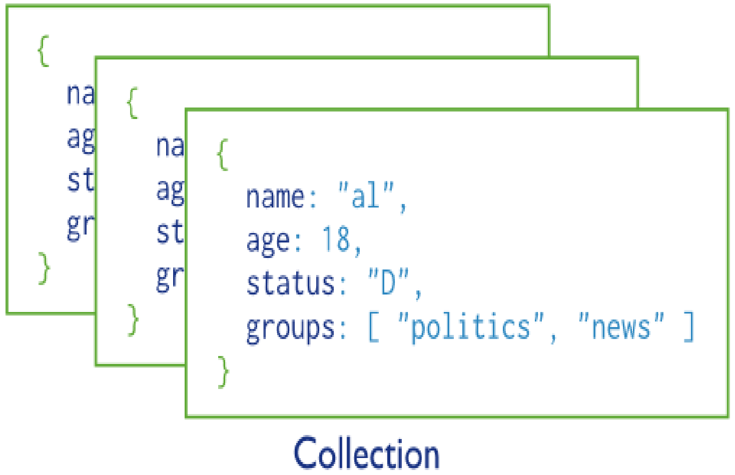
  (a) C (b) A (c) P (d) none of the above
- What is missing in The CAP Theorem in implementing distributed systems?

CAP

- Dynamo does not guarantee C by default
- The event of P forces systems to decide on reducing C or A
- What is the probability of P?
  - Local network
  - Wide area network

Collections in MongoDB

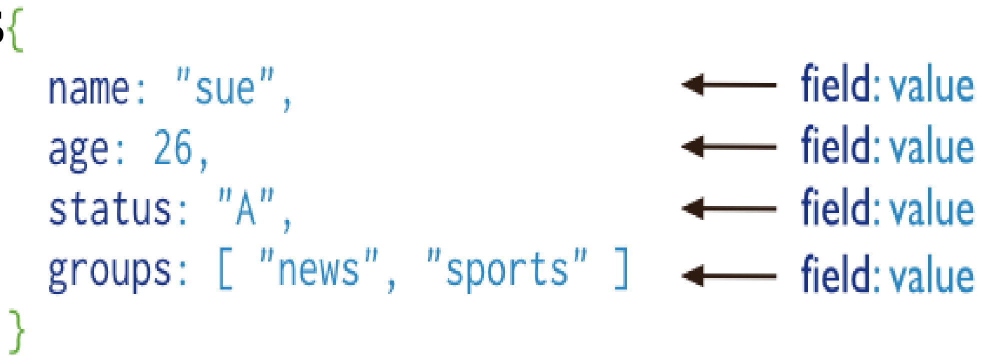
- MongoDB stores all data in Collections
- Collections in MongoDB analogous to tables in relational databases
- It is schema – less and contains a group of related documents
- Created on-the-fly when referenced for the first time



```
{
  name: "al",
  age: 18,
  status: "D",
  groups: [ "politics", "news" ]
}
```

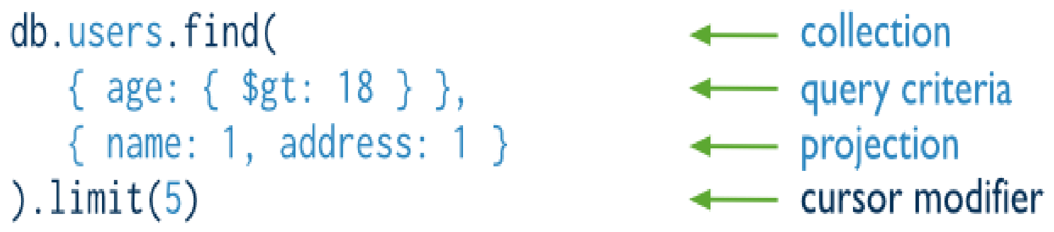
Document in MongoDB

- Stored in Collections
- Analogous to Records/Rows in Relational databases
- Has **_id** field – works like Primary keys in Relational databases
- Sample document containing name, age, status and groups



```
{
   name: "sue",
   age: 26,
   status: "A",
   groups: [ "news", "sports" ]
}
```

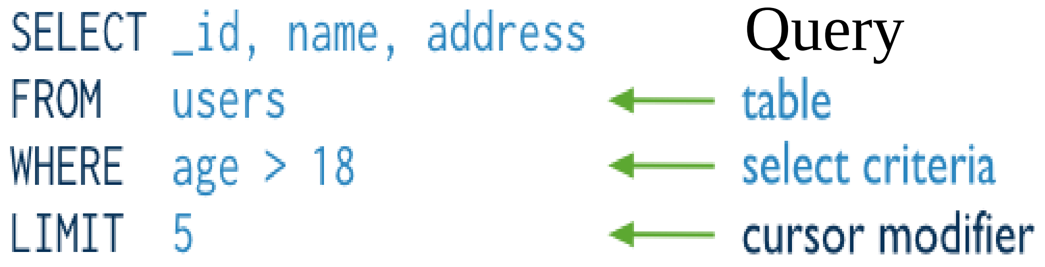
Queries in MongoDB

- MongoDB provides **db.collection.find()** method to retrieve data
- This method accepts both query criteria and
- 

```
db.users.find(
   { age: { $gt: 18 } },
   { name: 1, address: 1 }
).limit(5)
```

Mongo Query

Similar SQL Query



```
SELECT _id, name, address
FROM   users
WHERE  age > 18
LIMIT  5
```

Projections - Queries in MongoDB

- If you include 1 – in projection parameter, it returns the value
- If you include 0 – in projection parameter, it eliminates it from the result

```
db.records.find( { "user_id": { $lt: 42} }, { "_id": 0, "name": 1 , "email": 1 } )
```

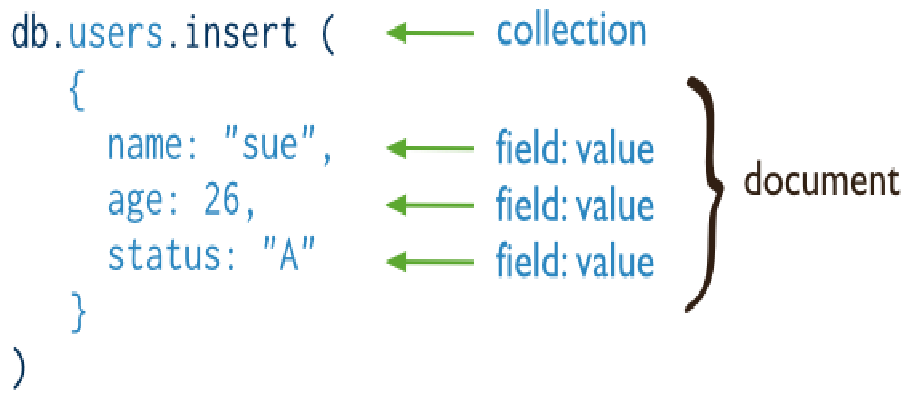
- _id – always included in results. Specify "**_id : 0**" to exclude it from results

```
db.records.find( { "user_id": { $lt: 42 } }, { "history": 0 } )
```

- Excludes history from field from the results, and returns all other fields

Insert Operation

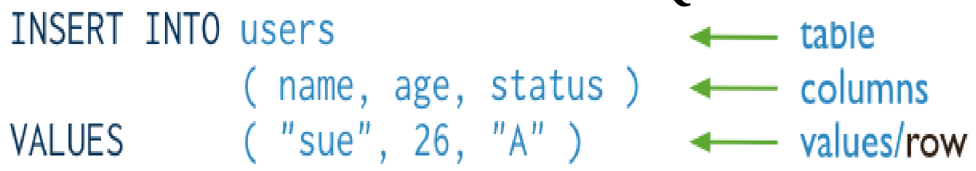
- In MongoDB, **db.collection.insert()** method adds new documents to collections



```
db.users.insert (
   {
     name: "sue",
     age: 26,
     status: "A"
   }
)
```

Mongo Insert

SQL insert



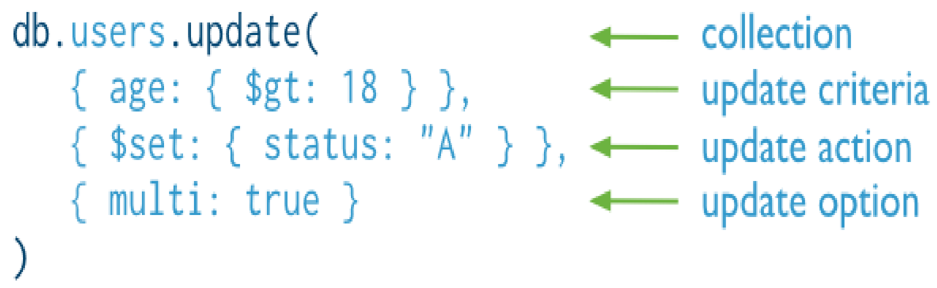
```
INSERT INTO users
                ( name, age, status )
VALUES          ( "sue", 26, "A" )
```

Insert Operation

- If **_id** is not included in the insert query, mongo adds _id internally and computes its value with a unique **ObjectId**
- **ObjectId**:
  - 12 byte BSON type
  - Guarantees uniqueness within that collection
  - Generated based on timestamp, machine ID, process ID and a internal process-local incremental counter

Update Operation

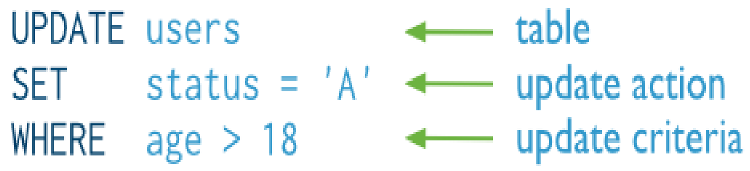
- In MongoDB, **db.collection.update()** method modifies existing documents in a collection



```
db.users.update(
   { age: { $gt: 18 } },
   { $set: { status: "A" } },
   { multi: true }
)
```

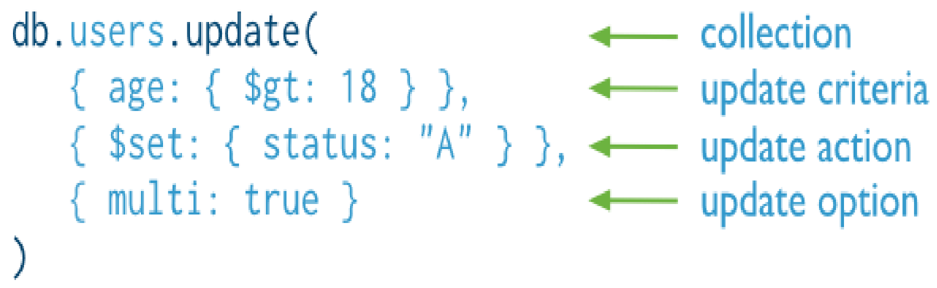
Mongo Update

SQL update



```
UPDATE users
SET    status = 'A'
WHERE  age > 18
```

Update Operation

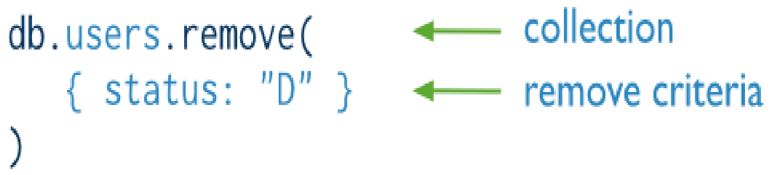


```
db.users.update(
   { age: { $gt: 18 } },
   { $set: { status: "A" } },
   { multi: true }
)
```

- Updates on **users** collection
- Sets “**status**” field to “A”
- With criteria of “**age**” greater than “**18**”
- **multi: true** – updates all the document in a query with the matching criteria

Remove Operation

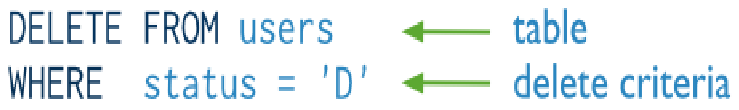
- In MongoDB, **db.collection.remove()** method deletes document from the collection



```
db.users.remove(          ← collection
   { status: "D" }        ← remove criteria
)
```

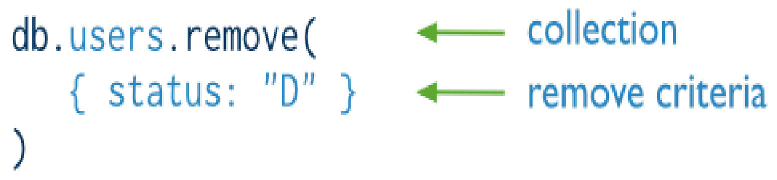
Mongo Delete

SQL Delete



```
DELETE FROM users         ← table
WHERE  status = 'D'       ← delete criteria
```

Remove Operation



```
db.users.remove(
   { status: "D" }
)
```

- Delete operation performed on users collection
- Removes all documents with “**status**” as “**D**”

Additional Operations

- **db.collections.save()**
  - Updates an existing documents if it finds the document with the mentioned values
  - Inserts in the collection, if it does not find a document with the mentioned values

Installing MongoDB

- In Windows:

  https://www.youtube.com/watch?t=1&v=sBdaRlgb4N8

- In Mac/Linux:

  https://www.youtube.com/watch?v=_WJ8m5QHvwc

Using MongoDB with Node.js

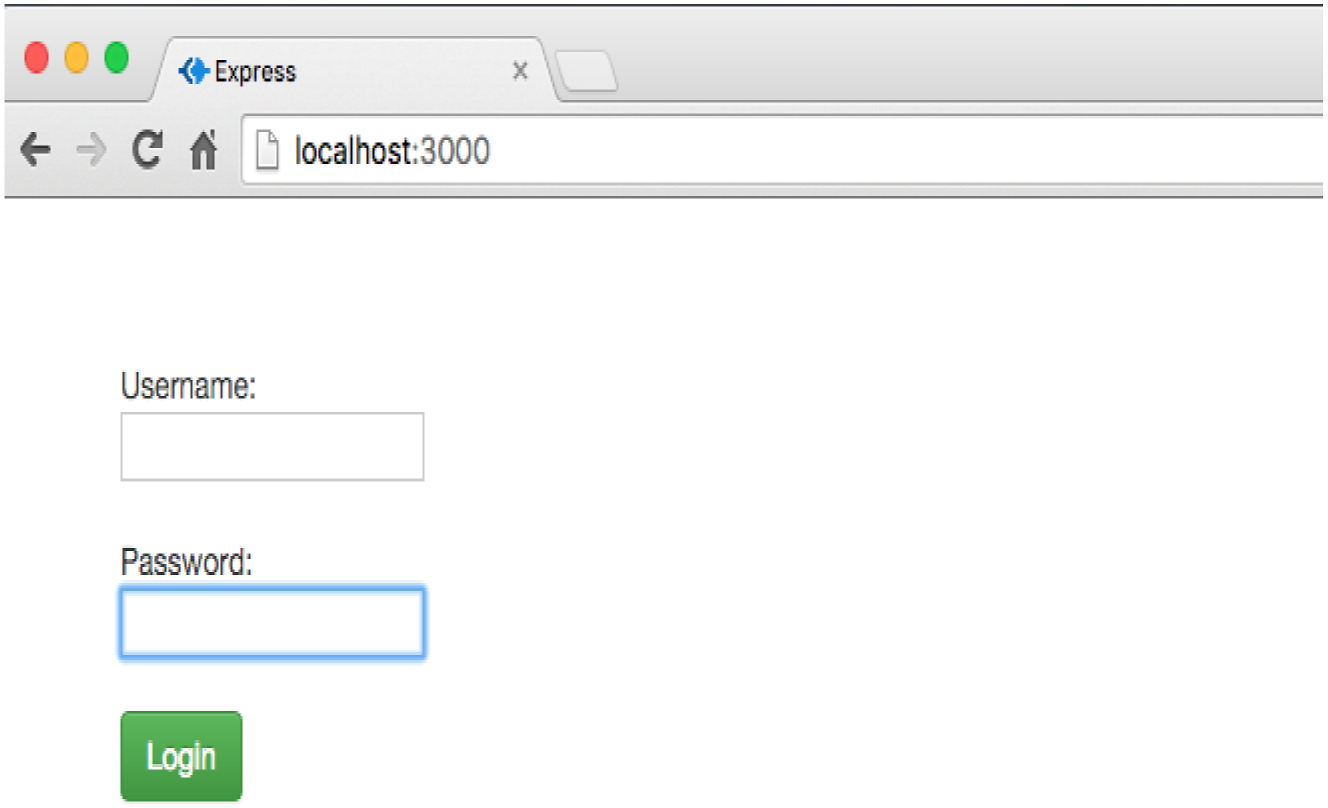
- **Install MongoDB Node.js Module**

```
npm install mongodb
```

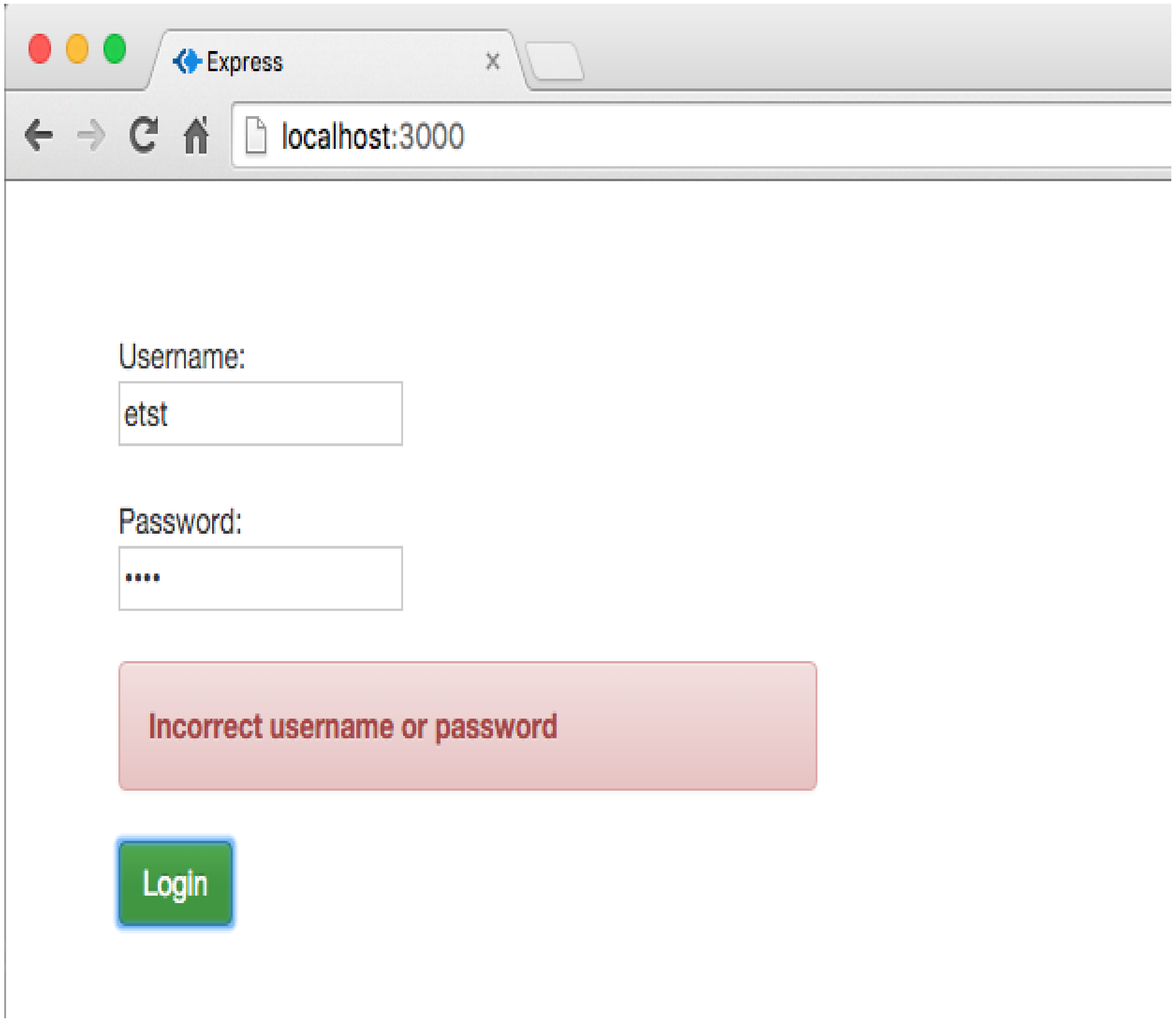
Example

- Login Application
- Access MongoDB to authentic the user
- Use Mongo Store to store sessions in MongoDB

Example – Login Page



Example – Wrong Credentials



Username:

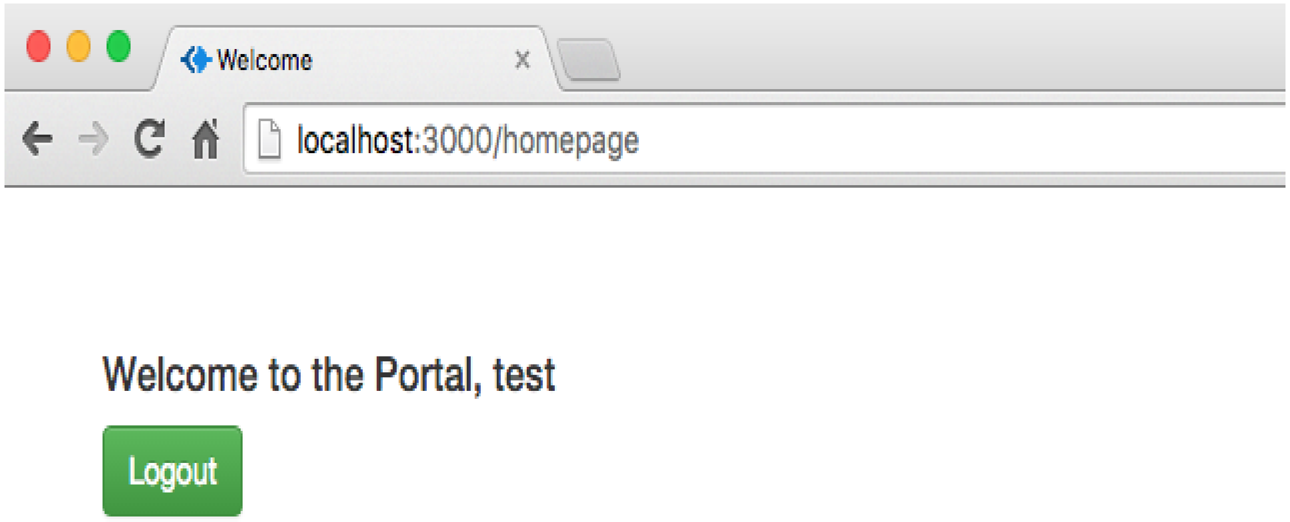
etst

Password:

••••

Incorrect username or password

Example - Homepage



Welcome to the Portal, test

app.js – Configuration file

```
//URL for the sessions collections in mongoDB
var mongoSessionConnectURL = "mongodb://localhost:27017/sessions";
var expressSession = require("express-session");
var mongoStore = require("connect-mongo")(expressSession);
var mongo = require("./routes/mongo");                //Database configuration file
var login = require("./routes/login");                //Login authentication routes file

app.use(expressSession({
        secret: 'cmpe273_teststring',
        resave: false,  //don't save session if unmodified
        saveUninitialized: false,        // don't create session until something
stored
        duration: 30 * 60 * 1000,
        activeDuration: 5 * 60 * 1000,
        store: new mongoStore({
                url: mongoSessionConnectURL
        })
}));
//connect to the mongo collection session and then createServer
mongo.connect(mongoSessionConnectURL, function(){
        console.log('Connected to mongo at: ' + mongoSessionConnectURL);
        http.createServer(app).listen(app.get('port'), function(){
                console.log('Express server listening on port ' + app.get('port'));
        });
});
```

app.js – Configuration file

```
//GET Requests
app.get('/', routes.index);
app.get('/users', user.list);
app.get('/homepage',login.redirectToHomepage);

//POST Requests
app.post('/checklogin', login.checkLogin);
app.post('/logout', login.logout);
```

mongo.js

```
var MongoClient = require('mongodb').MongoClient;
var db;
var connected = false;

/**Connects to the MongoDB Database with the provided URL**/
exports.connect = function(url, callback){
    MongoClient.connect(url, function(err, _db){
      if (err) { throw new Error('Could not connect: '+err); }
      db = _db;
      connected = true;
      console.log(connected +" is connected?");
      callback(db);
    });
};

/**Returns the collection on the selected database**/
exports.collection = function(name){
    if (!connected) {
      throw new Error('Must connect to Mongo before calling "collection"');
    }
    return db.collection(name);

};
```

login.js

```
var mongo = require("./mongo");
var mongoURL = "mongodb://localhost:27017/login";

exports.checkLogin = function(req,res){
		var username = req.param("username");
		var password = req.param("password");
		var json_responses;

		mongo.connect(mongoURL, function(){
		   console.log('Connected to mongo at: ' + mongoURL);
		   var coll = mongo.collection('login');

		   coll.findOne({username: username, password:password}, function(err,
						user){
						if (user) {
							// This way subsequent requests will know the user
							is logged in.
							req.session.username = user.username;
							json_responses = {"statusCode" : 200};
							res.send(json_responses);

						} else {
							json_responses = {"statusCode" : 401};
							res.send(json_responses);
						}
				});
		});
};
```

login.js

```
//Redirects to the homepage
exports.redirectToHomepage = function(req,res)
{
		//Checks before redirecting whether the session is valid
		if(req.session.username)
		{
				//Set these headers to notify the browser not to maintain any cache
for the page being loaded
				res.header('Cache-Control', 'no-cache, private, no-store, must-
revalidate, max-stale=0, post-check=0, pre-check=0');
				res.render("homepage",{username:req.session.username});
		}
		else
		{
				res.redirect('/');
		}
};

//Logout the user - invalidate the session
exports.logout = function(req,res)
{
		req.session.destroy();
		res.redirect('/');
};
```

login.ejs

```
<body ng-app="login" ng-controller="login">
  <div class="row">
            <div class="col-md-4">
               <div class="row">
                       <div class="col-md-12" style="margin: 10px;">
                       Username:<br> <input type="text" name="username”
                                  ng-model="username">
                       </div>
                       <div class="col-md-12" style="margin: 10px;">
                       Password:<br> <input type="password” name="password"
                                             ng-model="password">
                       </div>
                       <div class="col-md-12" style="margin: 10px;">
                         <div class="alert alert-danger" ng-hide="invalid_login">
                                  <strong>Incorrect username or password</strong>
                         </div>
                         <div class="alert alert-danger" ng-hide="unexpected_error">
                                  <strong>Unexpected error, try again</strong>
                         </div>
                       <input type="submit" class="btn btn-success" ng-click="submit();"
                                  value="Login" />
                       </div>
               </div>
      </div>
    </div>
</body>
```

login.ejs – angular Controller

```
//loading the 'login' angularJS module
var login = angular.module('login', []);
//defining the login controller
login.controller('login', function($scope, $http) {
		$scope.invalid_login = true;
		$scope.unexpected_error = true;
		$scope.submit = function() {
			$http({
				method : "POST",
				url : '/checklogin',
				data : {"username" : $scope.username,
					"password" : $scope.password}
			}).success(function(data) {
				//checking the response data for statusCode
				if (data.statusCode == 401) {
					$scope.invalid_login = false;
					$scope.unexpected_error = true;
				}
				else
					window.location.assign("/homepage");
			}).error(function(error) {
				$scope.unexpected_error = false;
				$scope.invalid_login = true;
			});
		};
})
```

homepage.ejs

```
<body>
   <div class="row">
                  <div class="col-md-12">
                              <form action="logout" method="post">
                                          <h4>Welcome to the Portal, <%= username %></h4>
                                          <input type="submit" value="Logout"  class="btn btn-
success"/>
                              </form>
                  </div>
    </div>
</body>
```

Exercise

- Create a Login Application
- Should have option to sign up the user
- Login with the same user
- Show the details of the signed in user
- Use MongoDB to store the data

References

- SQL vs NoSQL - https://www.mongodb.com/nosql-explained
- MongoDB Introduction - http://docs.mongodb.org/manual/core/crud-introduction/
- Installing MongoDB (Mac) - https://www.youtube.com/watch?v=_WJ8m5QHvwc
- Installing MongoDB (Windows) - https://www.youtube.com/watch?t=1&v=sBdaRlgb4N8