**MongoDB**

**MongoDB Start**

//Console Commands:

//Start Server

* Run
  + cd mongoDB/bin
  + mongod
  + mongod - -port 27018 🡪 can specify the port number instead of default port27017

//Start Client

* Run🡪 Mongo

// Start up a new database by switching to it. NOTE: The db does not exist until you create a collection.

* use lessondb

// Show the current db by running db.

* Db

// Show the available dbs.

* Show dbs

// Show the current collections in the db.

* Show collections

// Insert data into a database or a collection from a binary dump file or human readable json.

* mongorestore – to insert data to DB – to work with mongo dump file
* mongoimport - to import data –human readable json into a collection or DB

// Insert data into the lessondb database with this command. NOTE: This will create the collection automatically,

// ALSO, TAKE NOTE: the contents of the insert are basically a JS object, and include an array.

* db.iWantToGoToThere.insert({"continent": "Africa", "country":"Morocco", "majorcities": ["Casablanca", "Fez", "Marrakech"]})

**Application Schema:**

* Matching the data access patterns of your application

**Normalization**:

* Normalization is the process of organizing data in a database
* Eliminating reduntant(useless) data.
* Ensuring data dependencies make sense i.e data is logically stored.

#### First Normal Form (1NF)

* As per First Normal Form, no two Rows of data must contain repeating group of information i.e each set of column must have a unique value, such that multiple columns cannot be used to fetch the same row. Each table should be organized into rows, and each row should have a primary key that distinguishes it as unique.
* There should not be no repeating columns containing the same kind of data
* All columns should contain a single value
* There should be a primary key to uniquely identify each row
* The Primary key is usually a single column, but sometimes more than one column can be combined to create a single primary key.

#### Second Normal Form (2NF)

* As per the Second Normal Form there must not be any partial dependency of any column on primary key. It means that for a table that has concatenated primary key, each column in the table that is not part of the primary key must depend upon the entire concatenated key for its existence. If any column depends only on one part of the concatenated key, then the table fails Second normal form.
* In example of First Normal Form there are two rows for Adam, to include multiple subjects that he has opted for. While this is searchable, and follows First normal form, it is an inefficient use of space. Also in the above Table in First Normal Form, while the candidate key is {Student, Subject}, Age of Student only depends on Student column, which is incorrect as per Second Normal Form. To achieve second normal form, it would be helpful to split out the subjects into an independent table, and match them up using the student names as foreign keys.

#### Third Normal Form (3NF)

* Third Normal form applies that every non-prime attribute of table must be dependent on primary key, or we can say that, there should not be the case that a non-prime attribute is determined by another non-prime attribute. So this transitive functional dependency should be removed from the table and also the table must be in Second Normal form. For example, consider a table with following fields.

Transactions: ACID – Lack of transaction support

Atomic Operations: single doc completed before anyone sees the doc

One-to-one

One-to-many

Many-to-many

// Find all data in a Collection with db.[COLLECTION\_NAME].find()

// NOTE: the MongoDB \_id was created automatically. This id is specific for each doc in the collection.

* db.iWantToGoToThere.find()

// Adding .pretty() makes the data more readable.

* db.iWantToGoToThere.find().pretty()

// Find specific data by matching a field.

* db.iWantToGoToThere.find({"continent": "Africa"})
* db.iWantToGoToThere.find({"country": "Morocco"})

// Find specific data by matching an \_id.

* By id: db.iWantToGoToThere.find({\_id:[COPY AN OBJECTID FROM THE PREVIOUS FIND RESULTS]});

// A. Use the command line to create a classroom database. A field of name with the person's name, rownumber, os, hobbies with an array of the hobbies the person.

* use classroom
* db.classroom.insert({name: 'Steve', row:3, os:'Mac', hobbies:['Coding', 'Reading', 'Running'] })

// B. Use find commands to get: A list of everyone in row 3, An entry for a single person, all the Mac users in your row and how to find users by an entry in an array.

* db.classroom.find({row:3})
* db.classroom.find({name:'Steve'})
* db.classroom.find({name:'Steve', row:3})
* db.classroom.find({"hobbies": {$in: ["hobby1"]}})

/\* Update, Delete and Drop in MongoDB \*/

* db
* use lessondb

// Show how to update data - using db.[COLLECTION\_NAME].update(). Note that the above will only update the first entry it matches.

* db.iWantToGoToThere.update({"country": "Morocco"}, {$set: {"continent":"Antartica"}})

// To update multiple entries, you need to add {multi:true}

* db.iWantToGoToThere.update({"country": "Morocco"}, {$set: {"continent":"Antartica"}}, {multi:true})

// if the capital doesn't exist - it will create the field

* db.iWantToGoToThere.update({"country": "Morocco"}, {$set: {"capital":"Rabat"}})

// And show the field can now be updated with the same command

* db.iWantToGoToThere.update({"country": "Morocco"}, {$set: {"capital":"RABAT"}})
* db.movieDetails.updateMany({"imdb.votes": {$gt: 10000}, year: {$gte: 2010, $lte: 2013}, "tomato.consensus": null}, $unset{"tomato.consensus": ""}})

// Show how to push to an array with $push

* db.iWantToGoToThere.update({"country": "Morocco"}, {$push: {"majorcities":"Agadir"}})

// Show how to delete an entry with db.[COLLECTION\_NAME].remove()

* db.iWantToGoToThere.remove({"country":"Morocco"})

// Show how to empty a collection with db.[COLLECTION\_NAME].remove()

* db.iWantToGoToThere.remove({})

// Show how to drop a collection with db.[COLLECTION\_NAME].drop()

* db.iWantToGoToThere.drop()

// Show how to drop a database

* db.dropDatabase()

// Add Extreme Basketweaving to your array of hobbies.

* db.classroom.update({name: "Steve"}, {$push: {"hobbies":"Extreme Basketweaving"}})

// They're using a new Operating System now. Change their os field.

* db.classroom.update({name: [name of neighbor]}, {$set: {os:[name of another os]}})

// and wisely decided to move. Remove them from your database.

* db.classroom.remove({name: [name of another neighbor]})

// update the collection with field gavecandy - false.

* db.classroom.update({}, {$set: {gavecandy:false}}, {multi:true})

// Change the value of gavecandy to true for Steve entry.

* db.classroom.update({name:'Steve'}, {$set: {gavecandy:true}})
* addToSET - to add unique values
* truncate – delete all the documents and records

/\*INSERTIN ANIMALS and Introduce Sorting Results by a field \*/

// A) An example of animals you can insert into the zoo db

* use zoo
* db.animals.insert({"name":"Panda", "numlegs":4, "class":"mammal", "weight": 254, "whatIWouldReallyCallIt":"Captain Fuzzy Face"})
* db.animals.insert({"name":"Dog", "numlegs":4, "class":"mammal", "weight": 60, "whatIWouldReallyCallIt":"Captain Fuzzy Face II"})
* db.animals.insert({"name":"Lion", "numlegs":4, "class":"mammal", "weight": 300, "whatIWouldReallyCallIt":"Grumbles"})
* db.animals.insert({"name":"Zebra", "numlegs":4, "class":"mammal", "weight": 500, "whatIWouldReallyCallIt":"Stripes"})
* db.animals.insert({"name":"Chameleon", "numlegs":4, "class":"Reptile", "weight": 5, "whatIWouldReallyCallIt":"Scales"})

// B) Sorting results by field name db.COLLECTION\_NAME.find().sort({FIELD:1}) A value of 1 is for ascending order and -1 is for descending order. The id contains a timestamp, so sorting by id will sort by when they were entered to the database

* db.animals.find().sort({\_id:1})
* db.animals.find().sort({\_id:-1})

// sort by an integer - numlegs:

* db.animals.find().sort({numlegs:1})
* db.animals.find().sort({numlegs:-1})

// sort by a string - class:

* db.animals.find().sort({class:1})
* db.animals.find().sort({class:-1})

/\* Scraper: Server \*/

// Dependencies:

* var request = require('request'); // Snatches html from urls
* var cheerio = require('cheerio'); // Scrapes our html

// Now, make a request call for the "webdev" board on reddit. Notice: the page's html gets saved as the callback's third arg

//Example -1

* request('http://www.foxsports.com/fantasy/football/story/top-200-fantasy-football-player-rankings-draft-strategy-051716', function (error, response, html) {

// Load the html into cheerio and save it to a var. '$' becomes a shorthand for cheerio's selector commands, much like jQuery's '$'.

* var $ = cheerio.load(html);

// an empty array to save the data that we'll scrape

* var result = [];

// With cheerio, find each p-tag with a "title" class (i: iterator. element: the current element)

* $('.premium-image table tbody tr td:first-child').each(function(i, element){

// save the text of the element (this) in a "title" variable

* var title = $(this).text();
* console.log('title', title);

// In the currently selected element, look at its child elements (i.e., its a-tags),then save the values for any "href" attributes that the child elements may have

* var link = $(this). children('a').attr('href');

// save these results in an object that we'll push into the result array we defined earlier

* result.push({

title:title,

link:link

});

});

// log the result once cheerio analyzes each of its selected elements

* console.log(result);

});

//Example – 2

* request('http://screenrant.com/', function (error, response, html) {

// Load the html into cheerio and save it to a var. '$' becomes a shorthand for cheerio's selector commands, much like jQuery's '$'.

* var $ = cheerio.load(html);

// an empty array to save the data that we'll scrape

* var result = [];

// With cheerio, find each h4-tag with the class "headline-link"

* $('h2.title').each(function(i, element) {

// save the text of the h4-tag as "title"

* var title = $(this).text();
* console.log('title', title);

// find the h4 tag's parent a-tag, and save it's href value as "link"

* var link = $(element).parent().attr('href');

//for each h4-tag, make an object with data we scraped and push it to the result array

* result.push({

title:title,

link:link

});

});

// after the program scans each h4.headline-link, log the result

* console.log(result);

});

//Example -3

// run request to grab the html from awwards's clean website section

* request("http://www.awwwards.com/websites/clean/", function (error, response, html) {
* var $ = cheerio.load(html);

// an empty array to save the data that we'll scrape

* var result = [];

// with cheerio, look at each award-winning site, enclosed in "figure" tags with the class name "site"

* $("figure.site").each(function(i, element){

/\* Cheerio's find method will "find" the first matching child element in a parent. We start at the current element, then "find" its first child a-tag. Then, we "find" the lone child img-tag in that a-tag. Then, .attr grabs the imgs src value.So: <figure> -> <a> -> <img src="link"> "link" \*/

* var imgLink = $(element).find('a').find('img').attr("src");

// push the image's url (saved to the imgLink var) into the result array

result.push({"Link": imgLink});

});

// with each link scraped, log the result to the console

* console.log(result);

});

//Example -4

/\* Scraper: Server.js \*/

// Dependencies:

* var request = require('request'); // Snatches html from urls
* var cheerio = require('cheerio'); // Scrapes our html

// first, tell the console what server3.js is doing

console.log("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n" +

"Look at the image of every award winner in \n" +

"one of the pages of awwwards.com. Then,\n" +

"grab the image's source url." +

"\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n")

// run request to grab the html from awwards's clean website section

* request("http://www.awwwards.com/websites/clean/", function (error, response, html) {

// load the html into cheerio

var $ = cheerio.load(html);

// make an empty array for saving our scraped info

* var **result** = [];

// with cheerio, look at each award-winning site,

// enclosed in "figure" tags with the class name "site"

$("figure.site").each(function(i, element){

/\* Cheerio's find method will "find" the first matching child element in a parent.

\* We start at the current element, then "find" its first child a-tag.

\* Then, we "find" the lone child img-tag in that a-tag.

\* Then, .attr grabs the imgs src value.

\* So: <figure> -> <a> -> <img src="link"> -> "link" \*/

* var imgLink = $(element).find('a').find('img').attr("src");

// push the image's url (saved to the imgLink var) into the result array

* result.push({"Link": imgLink});
* });

// with each link scraped, log the result to the console

* console.log(result);
* });

**Count all Documents in a Collection**

To count the number of all documents in the orders collection, use the following operation:

db.orders.count()

This operation is equivalent to the following:

db.orders.find().count()