Node.js

A platform for building fast & scalable network applications.

Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

People matter, results count.

Node.js

(v0.12.0)

Copyright © Capgemini 2017. All Rights Reserved

Document History

Date	Course Version No.	Software Version No.	Developer / SME	Change Record Remarks
14/02/2014	1.0	Node.js v0.12.0	Karthik Muthukrishnan	

Copyright © Capgemini 2017. All Rights Reserved

Course Goals and Non Goals

- **Course Goals**
  - Learning the fundamentals of Node.js.
  - Building Web applications using Express.js
  - Learning the fundamentals of MongoDB and persisting data using Node.js
  - Writing Unit Tests and code coverage using Mocha, Chai, SuperTest & Istanbul

- **Course Non Goals**
  - Covering the entire Node core modules & getting in detail how node engine works.
  - Creating Web Application using MEAN Stack(MongoDB, Express, Angular & Node)
  - Deploying Node applications
  - Creating C++ modules for Node
  - Executing the code in Mac OS, Linux / Unix

Copyright © Capgemini 2017. All Rights Reserved

Pre-requisites

- **HTML, JavaScript, CSS**

Copyright © Capgemini 2017. All Rights Reserved

Intended Audience

- **Web application developers**

Copyright © Capgemini 2017. All Rights Reserved

Day Wise Schedule

- **Day 1**

  Lesson 1: Getting started with Node.js

- **Day 2**

  Lesson 1 (Contd.): Getting started with Node.js

  Lesson 2: Working with Express framework

- **Day 3**

  Lesson 3: Database and Session handling

- **Day 4**

  Lesson 3 (Contd.): Database and Session handling

  Lesson 4: Unit Testing, Logging & Debugging

Copyright © Capgemini 2017. All Rights Reserved

Table of Contents

Lesson 1: Getting started with Node.js

1.1. JavaScript Essentials

- How JavaScript works
- Event loop
- Stack, Heap and Queue

1.2. Node.js Fundamentals

- Introduction to Node.js
- Why Node.js?
- Traditional Programming Limitations
- Creating more call stacks
- Event-Driven Programming
- Node.js Official website
- Downloading and Installing Node.js
- Node.js Globals

1.3. Working with Modules

- Module Introduction
- Modules in Node.js
- Loading a module
- package.json usage
- Creating package.json
- Node Package Manager
- Loading a third party module (installed via NPM)
- Creating and exporting a module

Copyright © Capgemini 2017. All Rights Reserved

Table of Contents

1.4: Working with Buffers

- Buffers in Node
- Creating Buffers in Node
- Writing to Buffer
- Reading from Buffer
- Slicing and copying a buffer

1.5: Event Handling

- Event Handling in Node
- EventEmitter
- EventEmitter Methods
- Creating an EventEmitter

1.6: Working with File System & Streams

- FileSystem module
- File I/O methods
- Stream
- Readable Stream
- Writable Stream

Copyright © Capgemini 2017. All Rights Reserved

Table of Contents

Lesson 2: Working with Express framework

2.1. Web development with Node

- Introduction
- HTTP module in Node.js
- Creating HTTP Server
- HTTP Properties and Methods
- HTTP Events
- Routing

2.2. Templating Engines

- Introduction
- Jade Templating Engine
- Working with Tags in Jade
- Working with id and classes in Jade
- Attributes and Nesting Tags in Jade
- Using if & unless in Jade
- Using for & each in Jade
- Using case & mixins in Jade
- Include and Extend in Jade
- EJS Templating engine

2.3. Working with Express.js

- Introduction
- Introduction to Express.js
- Connect Module
- Express.js Installation

Copyright © Capgemini 2017. All Rights Reserved

Table of Contents

- app.js
- Steps for creating Express.js Application
- application, request, response object properties & methods
- How Express.js works
- Request flow in Express
- Using middleware
- Types of middleware
- Application level middleware
- Router level middleware
- Built-in middleware
- Third party middleware
- Express 4.0 Router
- Express.js Scaffolding

Lesson 3: Database and Session Handling

3.1. MongoDB Basics

- Introduction
- Why MongoDB
- MongoDB Key terminologies
- SQL Terminology vs MongoDB Terminology
- MongoDB vs Relational Databases

3.2. Getting started with MongoDB

- Starting the mongo shell
- Creating and Dropping Database
- Creating and Dropping Collection
- Importing and Exporting Collection

Copyright © Capgemini 2017. All Rights Reserved

Table of Contents

3.3. MongoDB Queries

- Querying MongoDB Documents – find()
- Querying MongoDB Documents – pretty()
- Querying MongoDB Documents – findOne()
- Comparison Operators – gte & lte
- Comparison Operators – in & nin
- Logical Operators – or, not & and
- MongoDB Additional operators – all
- MongoDB Additional operators – exists
- MongoDB Additional operators –regex
- MongoDB Additional operators –where
- MongoDB Additional operators –sort
- MongoDB Additional operators –limit & skip
- Inserting Document(s)
- ObjectId
- Updating Document(s)
- Updating Document - save
- Updating Document – findAndModify
- Auto-Incrementing Sequence Field
- Array Update Operators
- Array Update Operators – Adding array items
- Array Update Operators – Removing array items
- Renaming and Deleting fields and documents
- Aggregation

Copyright © Capgemini 2017. All Rights Reserved

Table of Contents

- Stored JavaScript
- Indexing

3.4. Mongoose Library

- Introduction to Mongoose Library
- Connecting MongoDB using Mongoose Library
- Object Modeling
- RDBMS Schema Vs Mongoose Schema
- Simple Schema
- Complex Schema
- Validations in Model
- Finding documents
- Query API

3.5. Session Handling

- Introduction
- Session Handling in Node.js
- Storing session information in MongoDB

3.6. Cryptography

- Introduction
- Hashing data using bcryptjs

Lesson 4: Unit Testing, Logging & Debugging

4.1. Unit Testing

- Introduction
- Test-Driven Development
- Test-After Development
- Behavior-Driven Development

Copyright © Capgemini 2017. All Rights Reserved

Table of Contents

4.2. Unit Testing in Node

- Unit Testing in Node
- Mocha
- Chai
- SuperTest
- Code coverage using Istanbul

4.3. Logging and Debugging

- Logging using morgan
- Debugging with node-inspector
- Steps to debug

Copyright © Capgemini 2017. All Rights Reserved

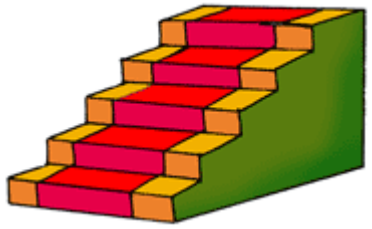
References

- Manning Node.js in Action by Mike Cantelon, Marc Harter, T.J. Holowaychuk, Nathan Rajlich
- O'Reilly Web Development with Node and Express by Ethan Brown
- O'Reilly Learning Node by Shelley Powers
- O'Reilly Learning Node Up and Running by Tom Hughes & Mike Wilson
- O'Reilly Learning Node for Front-End Developers by Garann Means
- Apress Practical Node.js by Azat Mardan
- Apress Node.js for developers by Colin J.Ihrig
- Apress Node.js Recipes by Cory Gackenheimer
- Jumpstart Nodejs by Don Nguyen
- Packt Publishing Node Cookbook by David Mark Clements
- Wrox Professional Node.js by Pedro Teixeria
- Hands-on Node.js by Pedro Teixeria
- Learning Node.js by Marc WandSchneider
- http://nodejs.org/api/
- http://expressjs.com/4x/api.html

Copyright © Capgemini 2017. All Rights Reserved

Next Step Courses (if applicable)

- **Creating Web application using MEAN (MongoDB, ExpressJS, AngularJS, NodeJS) Stack**



Copyright © Capgemini 2017. All Rights Reserved

Supplement Technologies

- **AngularJS**
- **jQuery**
- **Bootstrap**
- **Underscore.js**
- **Backbone.js**

Copyright © Capgemini 2017. All Rights Reserved

Getting started with Node.js

Copyright © Capgemini 2017. All Rights Reserved

How JavaScript works

- **JavaScript is Single Threaded.**
- **A JavaScript engine exists in a single OS process and consumes a single thread.**
- **When the application is running, CPU execution is never performed in parallel, since the JavaScript engine uses this method, it is impossible for users to get the Deadlocks and Race Conditions which actually makes Multi Threaded applications so complex.**

Copyright © Capgemini 2017. All Rights Reserved

Event loop

- **An event loop is a construct that mainly performs two functions in a continuous loop**
  - **Event detection** : In any run of the loop, it has to detect which events just happened.
  - **Event handler triggering** : When an event happens, the event loop must determine the event callback and invoke it.
- **Event loop is just one thread running inside one process, which means that, when an event happens, the event handler can run without interruption**
  - There is at most one event handler running at any given time
  - Any event handler will run to completion without being interrupted
- **This allows the programmer to relax the synchronization requirements and not have to worry about concurrent threads of execution changing the shared memory state.**

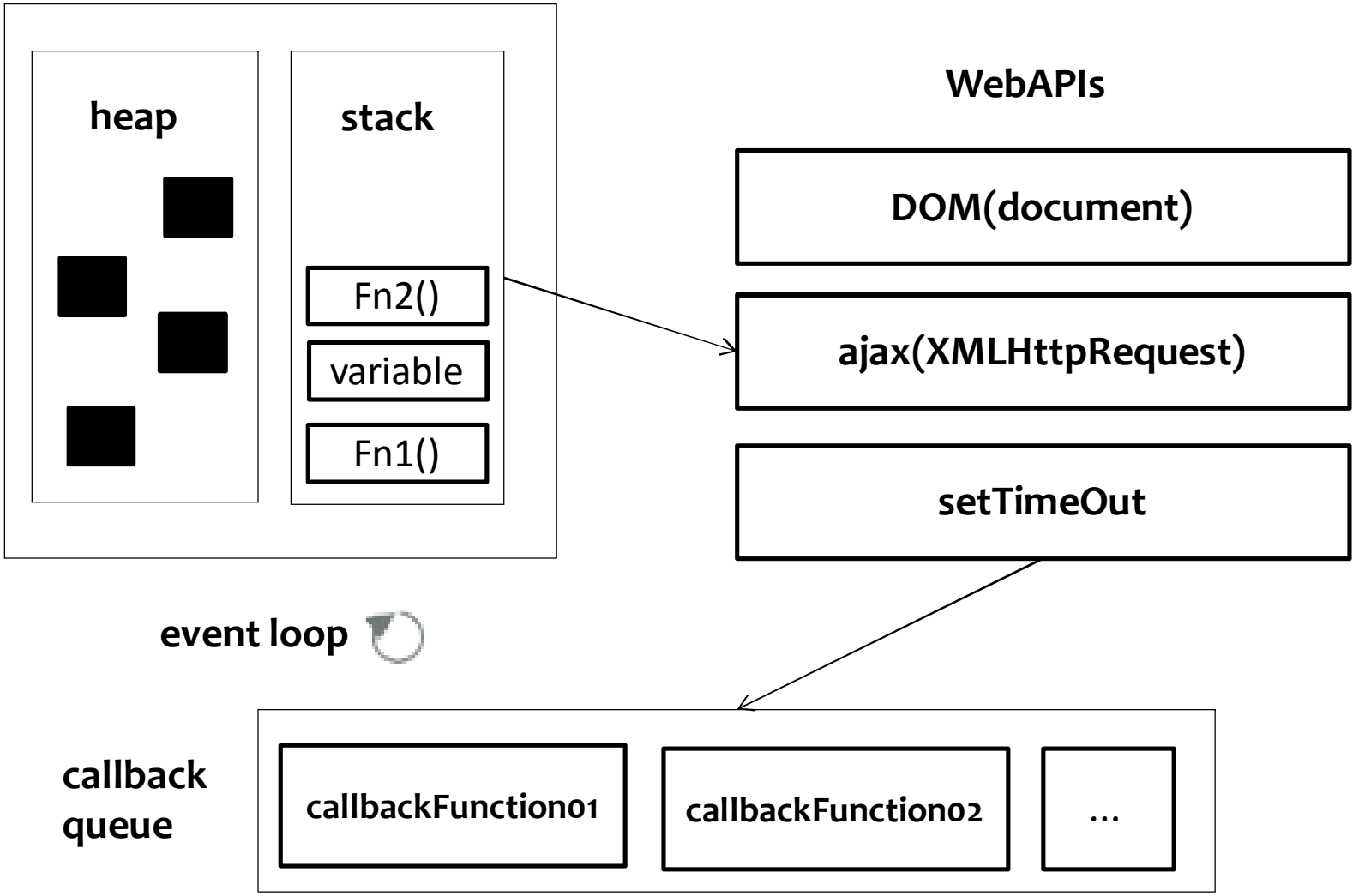
Copyright © Capgemini 2017. All Rights Reserved

Stack, Heap and Queue

- **Different browsers have different JavaScript engines (e.g. Chrome has V8, Firefox has OdinMonkey and IE has Chakra)**
- **JavaScript engine has three important features. They are Stack, Heap and Queue**
- **Each browser will implement these features differently, but all does the same.**
- **Stack**
  - Currently running functions gets added to the stack(frame). Pops out from the once it completes its execution.
- **Heap**
  - Memory allocation happened here. It is a bunch of memory where object's live in a unordered manner.
- **Queue**
  - function calls are queued up which gets added to the stack once it is empty.

Copyright © Capgemini 2017. All Rights Reserved

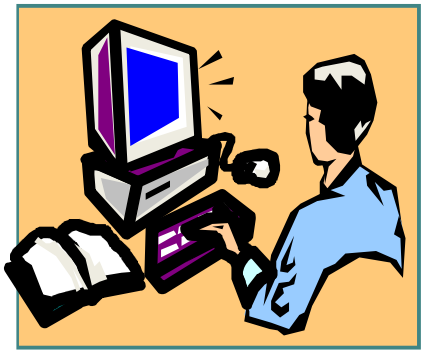
Stack, Heap and Queue



Copyright © Capgemini 2017. All Rights Reserved

Demo

- **EventLoop**



Copyright © Capgemini 2017. All Rights Reserved

Introduction to Node.js

- **In 2009 Ryan Dahl created Node.js or Node, a framework primarily used to create highly scalable servers for web applications. It is written in C++ and JavaScript.**
- **Node.js is a platform built on Chrome's JavaScript runtime(v8 JavaScript Engine) for easily building fast, scalable network applications.**
- **It's a highly scalable system that uses asynchronous, non-blocking I/O model (input/output), rather than threads or separate processes**
- **It is not a framework like jQuery nor a programming language like C# or JAVA . It's a new kind of web server like has a lot in common with other popular web servers, like Microsoft's Internet Information Services (IIS) or Apache**
- **IIS / Apache processes only HTTP requests, leaving application logic to be implemented in a language such as PHP or Java or ASP.NET. Node removes a layer of complexity by combining server and application logic in one place.**

Copyright © Capgemini 2017. All Rights Reserved

Why Node.js?

- **JavaScript everywhere i.e. Server-side and Client-side applications in JavaScript.**
- **Node is very easy to set up and configure.**
- **Vibrant Community**
- **Small core but large community so far we have 60,000 + packages on npm**
- **Real-time/ high concurrency apps (I/O bound)**
- **API tier for single-page apps and rich clients(iOS, Android)**
- **Service orchestration**
- **Top corporate sponsors like Microsoft, Joyent, PayPal etc..**
- **Working with NOSQL(MongoDB) Databases**

Copyright © Capgemini 2017. All Rights Reserved

Traditional Programming Limitations

- **In traditional programming I/O (database, network, file or inter-process communication) is performed in the same way as it does local function calls. i.e. Processing cannot continue until the operation is completed.**
- **When the operation like executing a query against database is being executed, the whole process/thread idles, waiting for the response. This is termed as "Blocking"**
- **Due to this blocking behavior we cannot perform another I/O operation, the call stack becomes frozen waiting for the response.**
- **We can overcome this issue by creating more call stacks or by using event callbacks.**

Copyright © Capgemini 2017. All Rights Reserved

Creating more call stacks

- **To handle more concurrent I/O, we need to have more concurrent call stacks.**
- **Multi-threading is one alternative to this programming model.**
  - Makes use of separate CPU Cores as “Threads”
  - Uses a single process within the Operating System
  - Ran out of Ghz, hardware adds more cores
- **If the application relies heavily on a shared state between threads accessing and modifying shared state increase the complexity of the code and It can be very difficult to configure, understand and debug.**

Copyright © Capgemini 2017. All Rights Reserved

Event-driven Programming

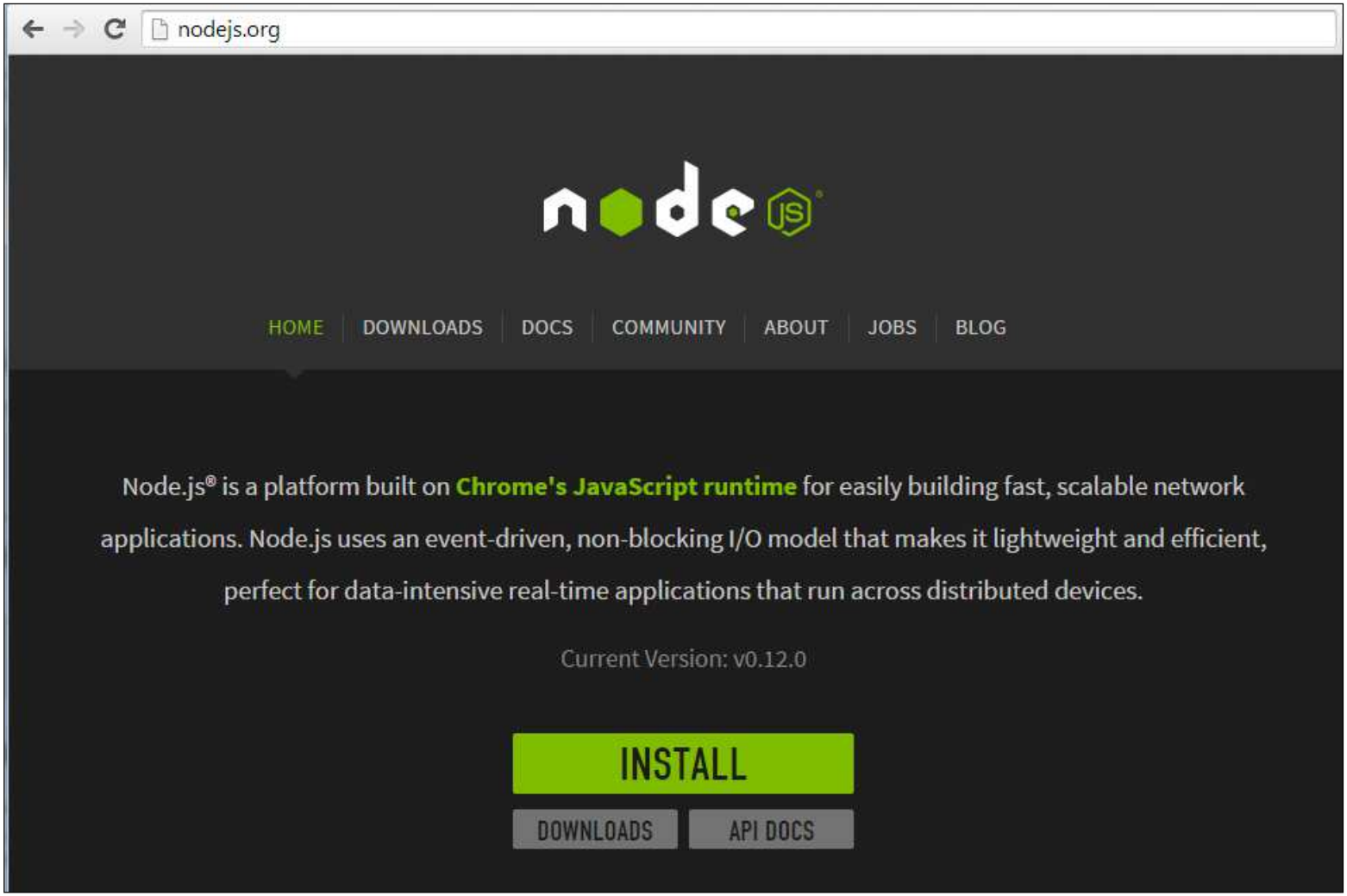
- **Event-driven programming or Asynchronous programming is a programming style where the flow of execution is determined by events.**
- **Events are handled by event handlers or event callbacks**
- **An event callback is a function that is invoked when something significant happens like when the user clicks on a button or when the result of a database query is available.**

```
query_finished = function(result) {
    do_something_with(result);
}
query('SELECT Id,Name FROM employees', query_finished);
```

- **Now instead of simply returning the result, the query will invoke the query_finished function once it is completed.**
- **Node.js supports Event-driven programming and all I/O in Node are non-blocking**

Copyright © Capgemini 2017. All Rights Reserved

nodejs.org – Node.js official Website



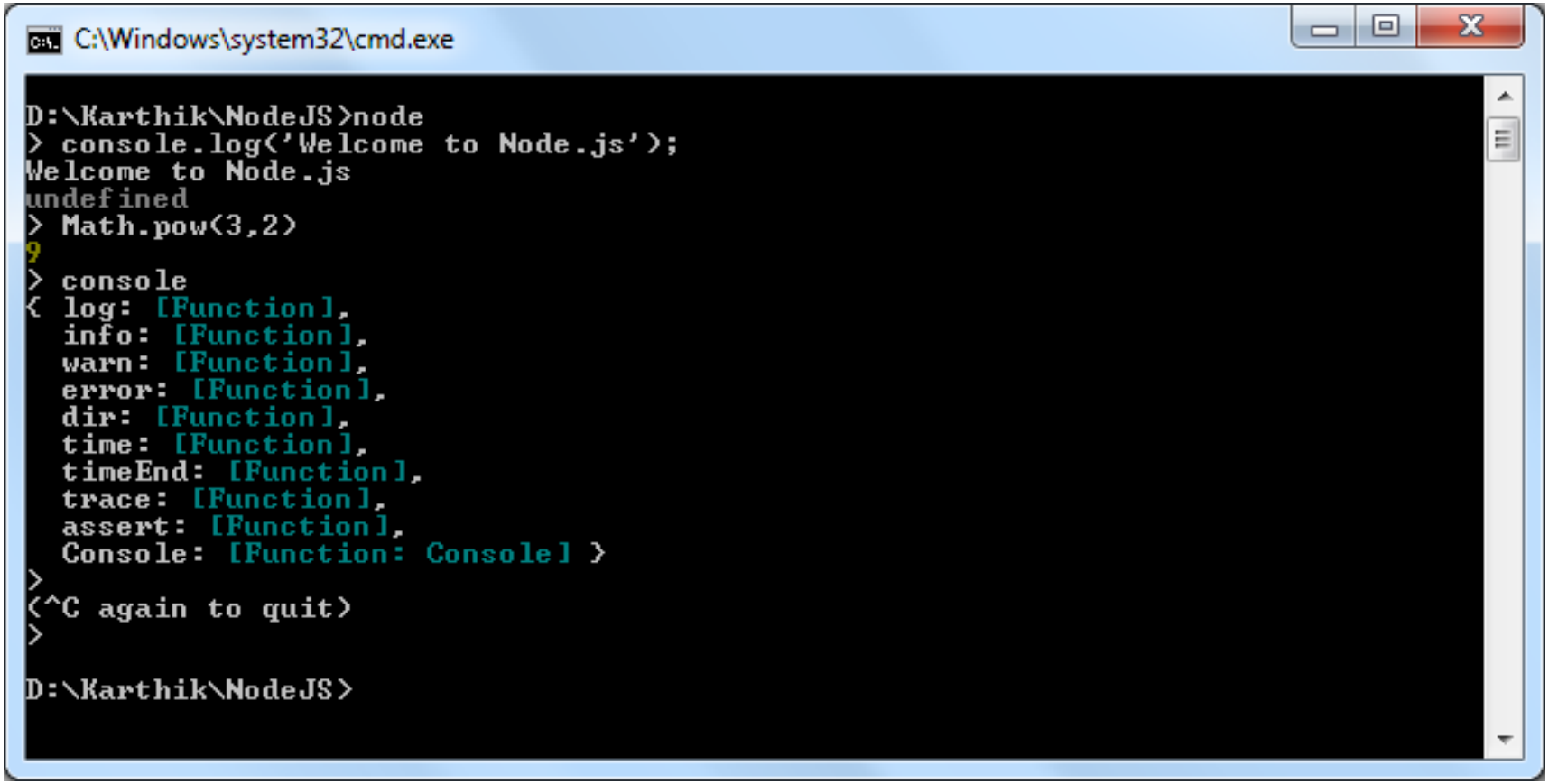
Copyright © Capgemini 2017. All Rights Reserved

Downloading Node.js

Windows Installer node-v0.12.0-x86.msi	Macintosh Installer node-v0.12.0.pkg	Source Code node-v0.12.0.tar.gz
Windows Installer (.msi)	32-bit	64-bit
Windows Binary (.exe)	32-bit	64-bit
Mac OS X Installer (.pkg)	Universal	
Mac OS X Binaries (.tar.gz)	32-bit	64-bit
Linux Binaries (.tar.gz)	32-bit	64-bit
SunOS Binaries (.tar.gz)	32-bit	64-bit
Source Code	node-v0.12.0.tar.gz	

Copyright © Capgemini 2017. All Rights Reserved

Node.js CLI



Copyright © Capgemini 2017. All Rights Reserved

Node.js Globals

- **global**
  - Any variables or members attached to global are available anywhere in our application. GLOBAL is an alias for global
    - global.companyName = 'IGATE'
    - global['companyName'] // We can directly access the members attached to global.
- **process**
  - The process object is a global object which is used to Inquire the current process to know the PID, environment variables, platform, memory usage etc.
    - process.platform
    - process.exit( )
- **console**
  - It provides two primary functions for developers testing web pages and applications
    - console.log('IGATE')

Copyright © Capgemini 2017. All Rights Reserved

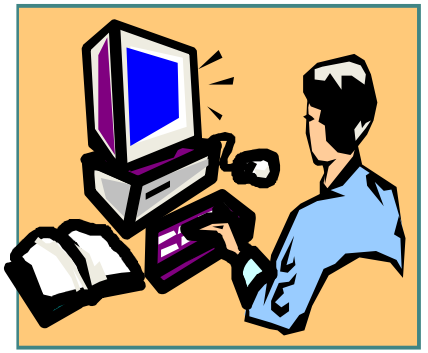
Module Introduction

- **A module is the overall container which is used to structure and organize code.**
- **It supports private data and we can explicitly defined public methods and variables (by just adding/removing the properties in return statement) which lead to increased readability.**
- **JavaScript doesn't have special syntax for package / namespace, using module we can create self-contained decoupled pieces of code.**
- **It avoids collision of the methods/variables with other global APIs.**

Copyright © Capgemini 2017. All Rights Reserved

Demo

- **Module**



Copyright © Capgemini 2017. All Rights Reserved

Modules in Node.js

- **In Node, modules are referenced either by file path or by name.**
- **Node's core modules expose some Node core functions(like global, require, module, process, console) to the programmer, and they are preloaded when a Node process starts.**
- **To use a module of any type, we have to use the require function. The require function returns an object that represents the JavaScript API exposed by the module.**
  - *var module = require('module_name');*

Copyright © Capgemini 2017. All Rights Reserved

Loading a module

- **Modules can be referenced depending on which kind of module it is.**
- **Loading a core module**
  - Node has several modules compiled into its binary distribution. These are called the core modules. It is referred solely by the module name, not by the path and are preferentially loaded even if a third-party module exists with the same name.
    - *var http = require('http');*
- **Loading a file module (User defined module)**
  - We can load non-core modules by providing the absolute path / relative path. Node will automatically adding the .js extension to the module referred.
    - *var myModule = require('d:/karthik/nodejs/module'); // Absolute path for module.js*
    - *var myModule = require('../module'); // Relative path for module.js (one folder up level)*
    - *var myModule = require('./module'); // Relative path for module.js (Exists in current directory)*

Copyright © Capgemini 2017. All Rights Reserved

Loading a module

- **Loading a folder module (User defined module)**
  - We can use the path for a folder to load a module.
    - *var myModule = require('./myModuleDir');*
  - Node will presume the given folder as a package and look for a package definition file inside the folder. Package definition file name should be named as ***pagkage.json***
  - Node will try to parse ***package.json*** and look for and use the ***main*** attribute as a relative path for the entry point.
  - We need to use ***npm init*** command to create ***package.json***.
  - Creating ***Package.json*** using ***npm init*** command
    - D:\Karthik\NodeJs\modules> npm init
    - *{ "name": "Karthik_Modules", "version": "1.0.0", "description": "Karthik Modules for Demo",* ***"main": "index.js",*** *"scripts": { "test": "echo \"Error: no test specified\" && exit 1" }, "author": "Karthik M <karthik.muthukrishnan@igate.com>", "license": "ISC"}*
    - *var myModule = require('d:/Karthik/NodeJS/modules'); // refer index.js placed in modules folder*

Copyright © Capgemini 2017. All Rights Reserved

package.json usage

- **package.json is a configuration file from where the npm can recognize dependencies between packages and installs modules accordingly.**
- **It must be located in project's root directory.**
- **The JSON data in package.json is expected to adhere to a certain schema. The following fields are used to build the schema for package.json file**
  - **name and version** : package.json must be specified at least with a name and version for package. Without these fields, npm cannot process the package.
  - **description and keywords** : description field is used to provide a textual description of package. Keywords field is used to provide an array of keywords to further describe the package. Keywords and a description help people discover the package because they are searched by the npm search command
  - **author** : The primary author of a project is specified in the author field.
  - **main** : Instruct Node to identify its main entry point.

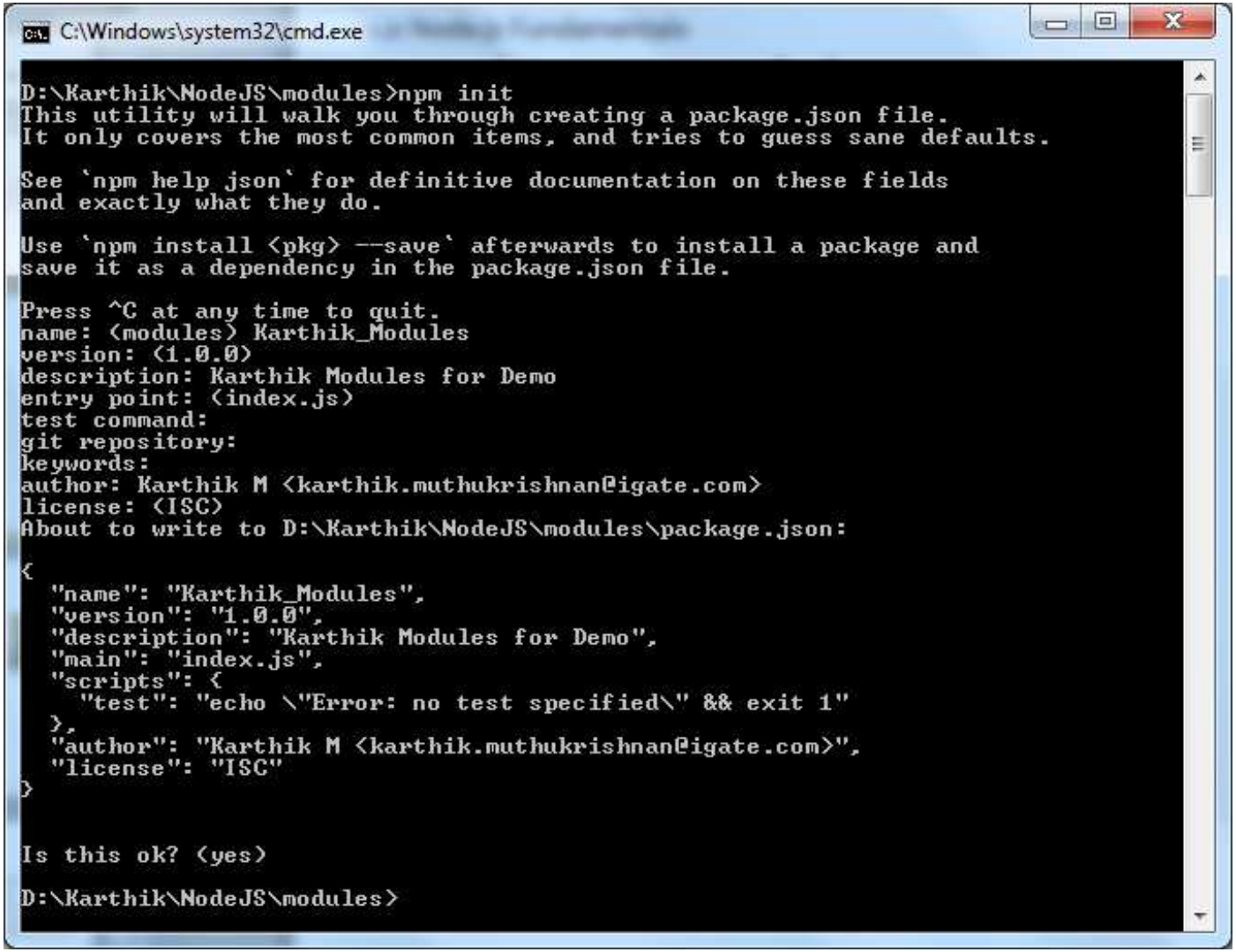
Copyright © Capgemini 2017. All Rights Reserved

package.json usage

- **dependencies** : Package dependencies are specified in the dependencies field.
- **devdependencies** : Many packages have dependencies that are used only for testing and development. These packages should not be included in the dependencies field. Instead, place them in the separate devdependencies field.
- **scripts :** The scripts field, when present, contains a mapping of npm commands to script commands. The script commands, which can be any executable commands, are run in an external shell process. Two of the most common commands are start and test. The start command launches your application, and test runs one or more of your application's test scripts.

Copyright © Capgemini 2017. All Rights Reserved

Creating package.json



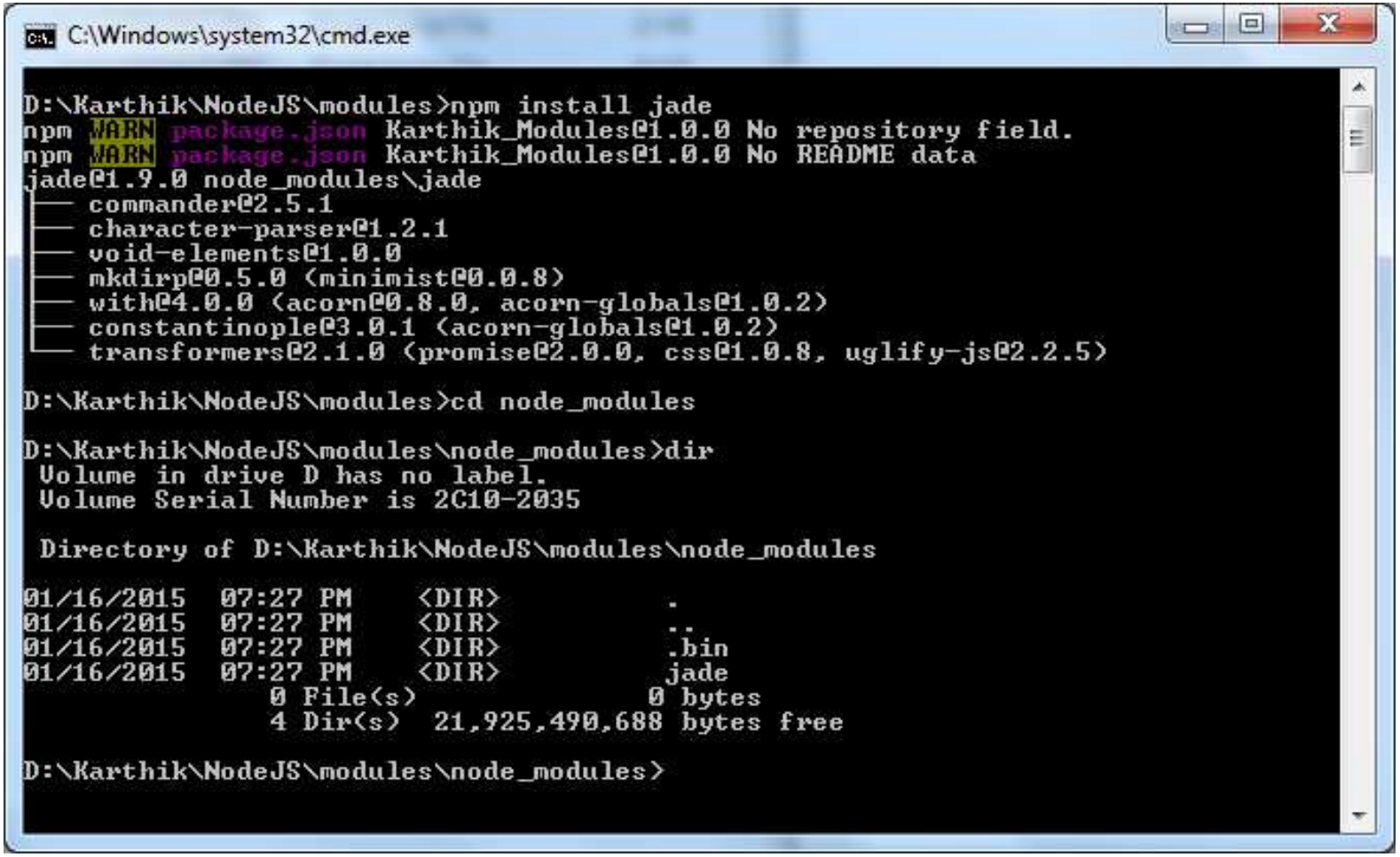
Copyright © Capgemini 2017. All Rights Reserved

Node Package Manager

- **Loading a module(Third party) installed via NPM (Node Package Manager)**
  - Apart from writing our own modules and core modules, we will frequently use the modules written by other people in the Node community and published on the Internet (npmjs.com).
  - We can install those third party modules using the ***Node Package Manager*** which is installed by default with the node installation.
  - To install modules via npm use the **npm install** command.
  - npm installs module packages to the **node_modules** folder.
  - To update an installed package to a newer version use the **npm update** command.
  - If the module name is not relative and is not a core module, Node will try to find it inside the node_modules folder in the current directory.
    - *var jade = require('jade');*
    - *Here jade is not a core module and not available as a user defined module found in relative path, it will look into the node_modules/jade/package.json and refer the file/ folder mentioned in main attribute.*

Copyright © Capgemini 2017. All Rights Reserved

Loading a third party module (installed via NPM)



Copyright © Capgemini 2017. All Rights Reserved

Creating and exporting a module

- **Creating a module that exposes / exports a function called helloWorld**

```
// Save it as myModule.js
exports.helloWorld = function () {
    console.log("Hello World");
}
```

- ***exports* object is a special object created by the Node module system which is returned as the value of the require function when you include that module.**
- **Consuming the function on the exports object created in myModule.js**

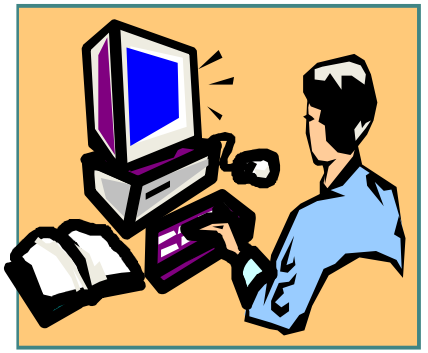
```
// Save it as moduleTest.js
var module = require('./myModule');
module.helloWorld();
```

- **We can replace exports with module.exports**
  - exports = module.exports = { }

Copyright © Capgemini 2017. All Rights Reserved

Demo

- **Modules**



Copyright © Capgemini 2017. All Rights Reserved

Buffers in Node

- **JavaScript doesn't have a byte type. It just has strings.**
- **Node is based on JavaScript with just using string type it is very difficult to perform the operations like communicate with HTTP protocol, working with databases, manipulate images and handle file uploads.**
- **Node includes a binary buffer implementation, which is exposed as a JavaScript API under the Buffer pseudo-class.**
- **Using buffers we can manipulate, encode, and decode binary data in Node. In node each buffer corresponds to some raw memory allocated outside V8.**
- **A buffer acts like an array of integers, but cannot be resized**

Copyright © Capgemini 2017. All Rights Reserved

Creating Buffers in Node

- **new Buffer(n) is used to create a new buffer of 'n' octets. One octet can be used to represent decimal values ranging from 0 to 255.**
- **There are several ways to create new buffers.**
  - **new Buffer(n)** : To create a new buffer of 'n' octets
    - var buffer = new Buffer(10);
  - **new Buffer(arr)** : To create a new buffer, using an array of octets.
    - var buffer = new Buffer([7,1,4,7,0,9]);
  - **new Buffer(str,[encoding])** : To create a new buffer, using string and encoding.
    - var buffer = new Buffer("IGATE","utf-8"); // utf-8 is the default encoding in Node.

```
D:\Karthik\NodeJS>node
> var buffer = new Buffer(10)
undefined
> console.log(buffer);
<Buffer 08 7d ac 00 e0 84 9b 00 0a 00>
> var buffer = new Buffer([7,1,4,7,0,9])
> console.log(buffer);
<Buffer 07 01 04 07 00 09>
> var buffer = new Buffer('IGATE')
> console.log(buffer);
<Buffer 49 47 41 54 45>
```


Copyright © Capgemini 2017. All Rights Reserved

Writing to Buffer

- **Writing to Buffer**
  - buf.write(str, [offset], [length], [encoding]) method is used to write a string to the buffer.
  - buf.write() returns the number of octets written. If there is not enough space in the buffer to fit the entire string, it will write a part of the string.
  - An offset or the index of the buffer to start writing at. Default value is 0.

```
D:\Karthik\NodeJS>node
> var str = new Buffer(26);
undefined
> str.write('IGATE Corporate University');
26
> str.write('IGATE',10,'utf-8');
5
> // Here the second argument indicates an offset and third is encoding format
  // (utf-8) is the default encoding format
```


Copyright © Capgemini 2017. All Rights Reserved

Reading from Buffer

- **Reading from buffers**
  - buf.toString([encoding], [start], [end]) method decodes and returns a string from buffer data.
  - buf.toString() returns method reads the entire buffer and returns as a string.
  - buf.toJSON() method is used to get the JSON-representation of the Buffer instance, which is identical to the output for JSON Arrays.

```
D:\Karthik\NodeJS>node
> var str = new Buffer(7);
undefined
> str.write('IGATE','utf-8');
5
> str.toString('utf-8')
'IGATE??'
> str.toString('utf-8',0,5)
'IGATE'
> str[5] = '*'.charCodeAt();
42
> str[6] = '*'.charCodeAt();
42
> str.toString()
'IGATE**'
> str.toJSON()
[ 73,
  71,
  65,
  84,
  69,
  42,
  42 ]
```


Copyright © Capgemini 2017. All Rights Reserved

Slicing and copying a buffer

- **Slicing a buffer**
  - buffer.slice([start],[end]) : We can slice a buffer and extract a portion from it, to create another smaller buffer by specifying the starting and ending positions.

```
D:\Karthik\NodeJS>node
> var buffer = new Buffer('IGATE ROCKS !!!')
undefined
> var smallBuffer = buffer.slice(6,11)
undefined
> buffer.toString()
'IGATE ROCKS !!!'
> smallBuffer.toString()
'ROCKS'
```

- **Copying a buffer**
  - buffer.copy(targetBuffer, [targetStart], [sourceStart], [sourceEnd]) : It is used to copy the contents of one buffer onto another

```
D:\Karthik\NodeJS>node
> var buffer = new Buffer('IGATE Corporate University')
undefined
> var targetBuffer = new Buffer(5)
undefined
> buffer.copy(targetBuffer,0,0,5)
5
> targetBuffer.toString()
'IGATE'
```


Copyright © Capgemini 2017. All Rights Reserved

Event Handling in Node

- **In node there are two event handling techniques. They are called callbacks and EventEmitter.**
- **Callbacks are for the async equivalent of a function. Any async function in node accepts a callback as it's last parameter**

```
var myCallback = function(data){
  console.log('got data: '+data);
};

var fn = function(callback){
    callback('Data from Callback');
};

fn(myCallback);
```


Copyright © Capgemini 2017. All Rights Reserved

EventEmitter

- **In node.js an event can be described simply as a string with a corresponding callback and it can be emitted.**
- **The *on* or *addListener* method allows us to subscribe the callback to the event.**
- **The emit method "emits" event, which causes the callbacks registered to the event to trigger.**

```
var events = require('events');
var eventEmitter = new events.EventEmitter();
var myCallback = function(data) {
    console.log('Got data: '+data);
};

eventEmitter.on('karthikEvent', myCallback);
var fn = function() {
     eventEmitter.emit('karthikEvent','Data from Emitter');
};
fn();
```


Copyright © Capgemini 2017. All Rights Reserved

EventEmitter Methods

- **All objects which emit events in node are instances of events.EventEmitter which is available inside Event module.**
- **We can access the Event module using require("events")**
- **addListener(event, listener) / on(event, listener)**
  - Adds a listener to the end of the listeners array for the specified event. Where listener is a function which needs to be executed when an event is emitted.
- **once(event, listener)**
  - Adds a one time listener for the event. This listener is invoked only the next time the event is fired, after which it is removed.
- **removeListener(event, listener)**
  - Remove a listener from the listener array for the specified event
- **removeAllListeners([event])**
  - Removes all listeners, or those of the specified event

Copyright © Capgemini 2017. All Rights Reserved

Creating an EventEmitter

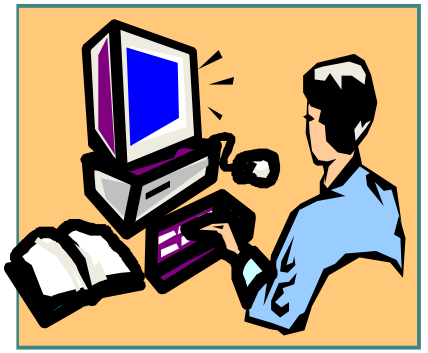
- **We can create Event Emitter pattern by creating a constructor function / pseudo-class and inheriting from the EventEmitter.**

```
var EventEmitter = require('events').EventEmitter,
                   util = require('util');

var Foo = function(){ }

util.inherits(Foo, EventEmitter);

Foo.prototype.someMethod = function() {
    this.emit('customEvent', 'Data from Some Method');
}

var fooObj = new Foo();
fooObj.on('customEvent',function(arg){
          console.log('Custom Event Occurred : '+arg);
});

fooObj.someMethod();
```


Copyright © Capgemini 2017. All Rights Reserved

Demo

- **EventEmitter**



Copyright © Capgemini 2017. All Rights Reserved

File System Module

- By default Node.js installations come with the file system module.
- This module provides a wrapper for the standard file I/O operations.
- We can access the file system module using require("fs")
- All the methods in this module has asynchronous and synchronous forms.
- synchronous methods in this module ends with 'Sync'. For instance *renameSync* is the synchronous method for *rename* asynchronous method.
- The asynchronous form always take a completion callback as its last argument. The arguments passed to the completion callback depend on the method, but the first argument is always reserved for an exception. If the operation was completed successfully, then the first argument will be null or undefined.
- When using the synchronous form any exceptions are immediately thrown. You can use try/catch to handle exceptions or allow them to bubble up.

Copyright © Capgemini 2017. All Rights Reserved

File I/O methods

- **fs.stat(path, callback)**
  - Used to retrieve meta-info on a file or directory.
- **fs.readFile(filename, [options], callback)**
  - Asynchronously reads the entire contents of a file.
- **fs.writeFile(filename, data, [options], callback)**
  - Asynchronously writes data to a file, replacing the file if it already exists. Data can be a string or a buffer.
- **fs.unlink(path, callback)**
  - Asynchronously deletes a file.
- **fs.watchFile(filename, [options], listener)**
  - Watch for changes on filename. The callback listener will be called each time the file is accessed. Second argument is optional by default it is { persistent: true, interval: 5007 }. The listener gets two arguments the current stat object and the previous stat object.

Copyright © Capgemini 2017. All Rights Reserved

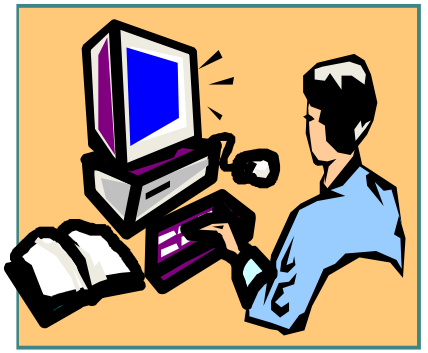
File I/O methods

- **fs.exists(path, callback)**
  - Test whether or not the given path exists by checking with the file system. The callback argument assigned with either true or false based on the existence.
- **fs.rmdir(path, callback)**
  - Asynchronously removes the directory.
- **fs.mkdir(path, [mode], callback)**
  - Asynchronously created the directory.
- **fs.open(path, flags, [mode], callback)**
  - Asynchronously open the file.
- **fs.close(fd, callback)**
  - Asynchronously closes the file.
- **fs.read(fd, buffer, offset, length, position, callback)**
  - Read data from the file specified by fd.

Copyright © Capgemini 2017. All Rights Reserved

Demo

- **FileSystem**



Copyright © Capgemini 2017. All Rights Reserved

Stream

- **A stream is an abstract interface implemented by various objects in Node. They represent inbound (ReadStream) or outbound (WriteStream) flow of data.**
- **Streams are readable, writable, or both (Duplex).**
- **All streams are instances of EventEmitter.**
- **Stream base classes can be loaded using *require('stream')***
- ***ReadStream* is like an outlet of data, once it is created we can wait for the data, pause it, resume it and indicates when it is actually end.**
- ***WriteStream* is an abstraction on where we can send data to. It can be a file or a network connection or even an object that outputs data that was transformed(when zipping a file)**

Copyright © Capgemini 2017. All Rights Reserved

Readable Stream

- ***ReadStream* is like an outlet of data, which is an abstraction for a source of data that you are reading from**
- **A Readable stream will not start emitting data until you indicate that you are ready to receive it.**
- **Readable streams have two "modes": a flowing mode and a non-flowing mode.**
  - In flowing mode, data is read from the underlying system and provided to your program as fast as possible.
  - In non-flowing mode, you must explicitly call stream.read() to get chunks of data out.
- **Readable streams can emit the following events**
  - **'readable'** : This event is fired when a chunk of data can be read from the stream.
  - **'data'** : This event is fired when the data is available. It will switch the stream to flowing mode when it is attached. It is the best way to get the data from stream as soon as possible.

Copyright © Capgemini 2017. All Rights Reserved

Readable Stream

- **'end'** : This event is fired when there will be no more data to read.
- **'close'** : Emitted when the underlying resource (for example, the backing file descriptor) has been closed. Not all streams will emit this.
- **'error'** : Emitted if there was an error receiving data.

- **Readable streams has the following methods**
  - **readable.read([size])** : Pulls data out of the internal buffer and returns it. If there is no data available, then it will return null.
  - **readable.setEncoding(encoding)** : Sets the encoding to use.
  - **readable.resume()** : This method will cause the readable stream to resume emitting data events.
  - **readable.pause()** : This method will cause a stream in flowing-mode to stop emitting data events. Any data that becomes available will remain in the internal buffer.
  - **readable.pipe(destination, [options])** : Pulls all the data out of a readable stream and writes it to the supplied destination, automatically managing the flow.

Copyright © Capgemini 2017. All Rights Reserved

Writable Stream

- **Writable stream interface is an abstraction for a destination that you are writing data to.**
- **Writable streams has the following methods**
  - **writable.write(chunk, [encoding], [callback])** : This method writes some data to the underlying system and calls the supplied callback once the data has been fully handled. Here chunk is a String / Buffer data to write
  - **writable.end([chunk], [encoding], [callback])** : Call this method when no more data will be written to the stream. Here chunk String / Buffer optional data to write
- **Writable streams can emit the following events**
  - **'drain'** : If a writable.write(chunk) call returns false, then the drain event will indicate when it is appropriate to begin writing more data to the stream.
  - **'finish'** : When the end() method has been called, and all data has been flushed to the underlying system, this event is emitted

Copyright © Capgemini 2017. All Rights Reserved

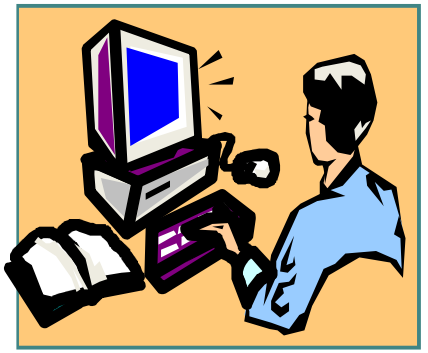
Writable Stream

- **'pipe'** : This is emitted whenever the pipe() method is called on a readable stream, adding this writable to its set of destinations.
- **'unpipe'** : This is emitted whenever the unpipe() method is called on a readable stream, removing this writable from its set of destinations.
- **'error'** : Emitted if there was an error when writing or piping data.

Copyright © Capgemini 2017. All Rights Reserved

Demo

- **Stream**



Copyright © Capgemini 2017. All Rights Reserved

Working with Express framework

Copyright © Capgemini 2017. All Rights Reserved

Introduction

- **Webserver like IIS / Apache serves static files(like html files) so that a browser can view them over the network.**
- **We need to place the files in a proper directory(like wwwroot in IIS), so that we can navigate to it using http protocol. The web server simply knows where the file is on the computer and serves it to the browser.**
- **Node offers a different paradigm than that of a traditional web server i.e. it simply provides the framework to build a web server.**
- **Interestingly building a webserver in node is not a cumbersome process, it can be written in just a few lines, moreover we'll have full control over the application.**

Copyright © Capgemini 2017. All Rights Reserved

HTTP module in Node.js

- **We can easily create an HTTP server in Node.**
- **To use the HTTP server and client one must require('http').**
- **The HTTP interfaces in Node are designed to support many features of the protocol which have been traditionally difficult to use. In particular, large, possibly chunk-encoded, messages.**
- **Node's HTTP API is very low-level. It deals with stream handling and message parsing only. It parses a message into headers and body but it does not parse the actual headers or the body.**
- **HTTP response implements the Writable Stream interface and request implements Readable Stream interface.**

Copyright © Capgemini 2017. All Rights Reserved

Creating HTTP Server

```
/* Loading http module*/
var http = require('http');

/* Returns a new web server object*/
var server = http.createServer(function(req,res){

        /* Sends a response header to the request.*/
        res.writeHead(200,{'content-type':'text/html'});

        /*sends a chunk of the response body*/
        res.write('<h1>Hello IGATE</h1>')

         /* signals server that all the responses has been sent */
        res.end('<b>Response Ended</b>')

});

/* Accepting connections on the specified port and hostname. */
server.listen(3000);

console.log('server listening on localhost:3000');
```


Copyright © Capgemini 2017. All Rights Reserved

HTTP Properties and Methods

- **http.STATUS_CODES**
  - Returns object which is a collection of all the standard HTTP response status codes and its short descriptions
- **http.createServer([requestListener])**
  - Returns a new web server object.
- **http.request(options, [callback])**
  - Returns a new web server object.
- **http.get(options, [callback])**
  - Returns a new web server object.

Copyright © Capgemini 2017. All Rights Reserved

HTTP Methods

- **server.listen(port, [hostname], [backlog], [callback])**
  - Begin accepting connections on the specified port and hostname.
- **server.close([callback])**
  - Stops the server from accepting new connections
- **response.writeHead(statusCode, [reasonPhrase], [headers])**
  - Sends a response header to the request.
- **response.setHeader(name, value)**
  - Sets a single header value for implicit headers.
- **response.write(chunk, [encoding])**
  - This sends a chunk of the response body. This method may be called multiple times to provide successive parts of the body.

Copyright © Capgemini 2017. All Rights Reserved

HTTP Events

- **request**
  - Emitted each time there is a request.
- **close**
  - Emitted when the server closes.
- **connect**
  - Emitted each time a client requests a http CONNECT method
- **finish**
  - Emitted when the response has been sent.

Copyright © Capgemini 2017. All Rights Reserved

Routing

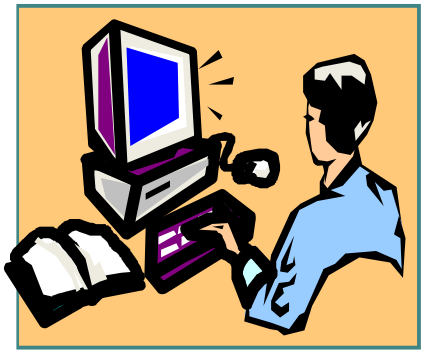
- **Routing refers to the mechanism for serving the client the content it has asked**

```
var http = require('http');
var server = http.createServer(function(req,res){
var path = req.url.replace(/\/?(?:\?.*)?$/, '').toLowerCase();
switch(path) {
        case '' :
                res.writeHead(200, {'Content-Type': 'text/html'});
                res.end('<h1>Home Page</h1>');
                break;
        case '/about' :
                res.writeHead(200, {'Content-Type': 'text/html'});
                res.end('<h1>About us</h1>');
                break;
        default:
                res.writeHead(404, { 'Content-Type': 'text/plain'
});
                res.end('Not Found');
                break;
        }
});
server.listen(3000);
```


Copyright © Capgemini 2017. All Rights Reserved

Demo

- **Creating-HTTP-Server**



Copyright © Capgemini 2017. All Rights Reserved

Introduction

- **A template engine is a library or a framework that uses some rules/languages to interpret data and render views.**
- **Templating solves the problem by allowing you to write in the target language, while at the same time providing the ability to insert dynamic data.**
- **For those of you familiar with the model–view–controller concept, templates belong to the view.**
- **In the Node world, we have many templating engines to choose. Here are some criteria to consider to choose a template engine :**
  - **Performance** : Templating engine to be as fast as possible
  - **Client, server, or both** : Most of the templating engines are available on both the server and client sides which makes our coding get well acquainted with the template engine
  - **Abstraction** : Abstracting the details of HTML away from you
- **Jade, ejs and handlebars are some of the popular templating engines.**

Copyright © Capgemini 2017. All Rights Reserved

Jade templating engine

- **Jade abstracts the details of HTML away from us. It uses whitespace and indentation as part of its language**

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
<title>Jade Demo</title>
<script>
    if (foo) {
        bar(1 + 5)
    }
</script>
<body>
<h1>Jade</h1>
<div id="container">

<p>You are amazing</p>


<p>
  Jade is a terse and
  simple templating
  language with a
  strong focus on
  performance and
  powerful features.
</p>
</body>
</html>
```

JADE

```
doctype html
html(lang="en")
  head
    title= pageTitle
    script.
      if (foo) {
         bar(1 + 5)
      }
  body

    h1 Jade
    #container
      if youAreUsingJade
        p You are amazing
      else
        p Get on it!
      p.
        Jade is a terse and
        simple templating
        language with a
        strong focus on
        performance and
        powerful features.
```


Copyright © Capgemini 2017. All Rights Reserved

Working with Tags in Jade

```
doctype html
html
    head
        title IGATE
    body
        div
            h1 IGATE
            p IGATE Corporate University.
        div
            footer &copy; IGATE 2015
```

```
<!DOCTYPE html>
<html>
  <head>
    <title>IGATE</title>
  </head>
  <body>
    <div>
      <h1>IGATE</h1>
      <p>IGATE Corporate University.</p>
    </div>
    <div>
      <footer>© IGATE 2015</footer>
    </div>
  </body>
</html>
```


Copyright © Capgemini 2017. All Rights Reserved

Working with id and classes in Jade

```
doctype html
html
    head
        title IGATE
    body
        #content
            .block
                input#foo.sty1.sty2
```

```
<!DOCTYPE html>
<html>
  <head>…</head>
  <body>
    <div id="content">
      <div class="block">
        <input id="foo" class="sty1 sty2">
      </div>
    </div>
  </body>
</html>
```


Copyright © Capgemini 2017. All Rights Reserved

Attributes and Nesting Tags in Jade

```
doctype html
html
    head
        title IGATE
    body
        ul#departments
            li
                a(href="#training") Training
            li
                a(href="#HR") HR
```

```
<!DOCTYPE html>
<html>
  <head>
    <title>IGATE</title>
  </head>
  <body>
    <ul id="departments">
      <li>
        <a href="#training">Training</a>
      </li>
      <li>
        <a href="#HR">HR</a>
      </li>
    </ul>
  </body>
</html>
```


Copyright © Capgemini 2017. All Rights Reserved

Passing options to Jade template

```
doctype html
html(lang="en")
    head
        title #{title}
    body
        h1 #{name} welcoming you all to Jade!
        br
        br
        input(type="text", value="#{name}")
        // Single line comment
        //- Invisible comment
```

```
<!DOCTYPE html>
<html lang="en">
  <head>…</head>
  <body>
    <h1>Karthik welcoming you all to Jade!</h1>
    <br>
    <br>
    <input type="text" value="Karthik">
    <!-- Single line comment-->
  </body>
</html>
```

Options : `{"title":"IGATE","name":"Karthik"}`

Copyright © Capgemini 2017. All Rights Reserved

Using if & unless in Jade

```
doctype html
html(lang="en")
    head
        title #{title}
    body
        if name == "Abishek"
            h1 Hello Abishek
        else
            h1 My name is #{name}
        unless condition
            p You have passed false in condition
```

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>IGATE</title>
  </head>
  <body>
    <h1>My name is Karthik</h1>
    <p>You have passed false in condition</p>
  </body>
</html>
```

Options : `{"title":"IGATE","name":"Karthik","condition":false}`

Copyright © Capgemini 2017. All Rights Reserved

Using for & each in Jade

```
doctype html
html(lang="en")
    head
        title #{title}
    body
        ul
            for department in departments
                li= department
            else
                li No departments!
        br
        select
                each department, counter in departments
                    option(value=counter) Department #{department}
```

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>IGATE</title>
  </head>
  <body>
    <ul>
      <li>HR</li>
      <li>Training</li>
      <li>Admin</li>
    </ul>
    <br>
    <select>
      <option value="0">Department HR</option>
      <option value="1">Department Training</option>
      <option value="2">Department Admin</option>
    </select>
  </body>
</html>
```

Options : `{"title":"IGATE","departments":["HR","Training","Admin"]}`

Copyright © Capgemini 2017. All Rights Reserved

Using case & mixins in jade

- **Mixins are functions that take parameters and produce some HTML.**
  - Syntax : *mixin name(param, param2,...)*
  - To use the function *+name(data)*

```
doctype html
html(lang="en")
    head
        title #{title}
    body
        mixin printName(name)
            p Hi #{name}
        case name
            when "Karthik"
                +printName("Karthik")
            when "Ganesh"
                +printName("Ganesh")
            default
                +printName("Guest")
```

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>IGATE</title>
  </head>
  <body>
    <p>Hi Ganesh</p>
  </body>
</html>
```

Options : `{"title":"IGATE","name":"Ganesh"}`

Capgemini CONSULTING.TECHNOLOGY.OUTSOURCING

Copyright © Capgemini 2017. All Rights Reserved

Include & Extend in Jade

- **include & extend are the ways to split logic into a separate file for the purpose of reusing it across multiple files**
- ***include* is a top-to-bottom approach. i.e. The file that includes is processed then the included file is processed.**
- **To include a Jade template use *include /path/filename***
- ***extend* is a bottom-to-top approach (opposite to include).**
- **It works with *extend filename* and *block blockname* statements.**

Copyright © Capgemini 2017. All Rights Reserved

Include & Extend in Jade

layout.jade

```
doctype html
html(lang="en")
    head
        block title
    body
        block content
```

footer.jade

```
footer &copy;CopyRights Reserved 2015 IGATE
```

index.jade

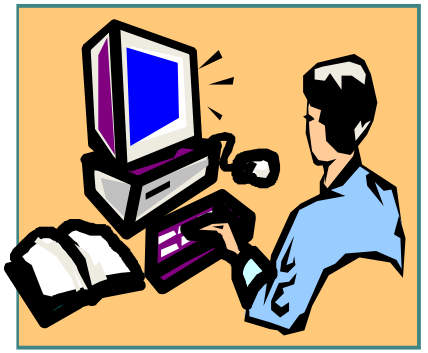
```
extends ./layout.jade
block title
    title IGATE
block content
    h1 IGATE Corporate University
    h3 IGATE training division
    include ./footer.jade
```

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>IGATE</title>
  </head>
  <body>
    <h1>IGATE Corporate University</h1>
    <h3>IGATE training division</h3>
    <footer>©CopyRights Reserved 2015 IGATE</footer>
  </body>
</html>
```


Copyright © Capgemini 2017. All Rights Reserved

Demo

- **Working with Jade Templating Engine**



Copyright © Capgemini 2017. All Rights Reserved

EJS Templating engine

- **EJS simply embeds JavaScript into the templates with a few simple tags to define how the JavaScript needs to be interpreted.**
- **EJS combines data and a template to produce HTML.**
- **JavaScript code gets executed inside<% %> and any code placed inside the tag <%= %> gets added with HTML.**

test.js

```
var ejs = require('ejs'),
    fs = require('fs');

fs.readFile('./test.ejs','utf-8',function (err, data) {
  if (err) throw err;
    var template = ejs.compile(data);
    var html = template({title:'IGATE',name:'Karthik'});
    console.log(html);
});
```

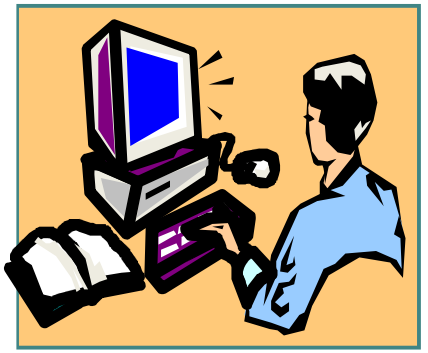
test.ejs

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title><%= title %></title>
    </head>
    <body>
        <h1><%= name %></h1>
    </body>
</html>
```


Copyright © Capgemini 2017. All Rights Reserved

Demo

- **Working with EJS Templating Engine**



Copyright © Capgemini 2017. All Rights Reserved

Introduction

- **If we try to create apps by only using core Node.js modules we will end up by writing the same code repeatedly for similar tasks such as**
  - Parsing of HTTP request bodies
  - Parsing of cookies
  - Managing sessions
  - Organizing routes with a chain of if conditions based on URL paths and HTTP methods of the requests
  - Determining proper response headers based on data types
- **Developers have to do a lot of manual work themselves, such as interpreting HTTP methods and URLs into routes, and parsing input and output data.**
- **Express.js solves these and many other problems using abstraction and code organization.**

Copyright © Capgemini 2017. All Rights Reserved

Introduction toExpress.js

- **Express.js is a web framework based on the core Node.js http module and Connect components**
- **Express.js framework provides a model-view-controller-like structure for your web apps with a clear separation of concerns (views, routes, models)**
- **Express.js systems are highly configurable, which allows developers to pick freely whatever libraries they need for a particular project**
- **Express.js framework leads to flexibility and high customization in the development of web applications.**
- **In Express.js we can define middleware such as error handlers, static files folder, cookies, and other parsers.**
- **Middleware is a way to organize and reuse code, and, essentially, it is nothing more than a function with three parameters: request, response, and next.**

Copyright © Capgemini 2017. All Rights Reserved

Connect module

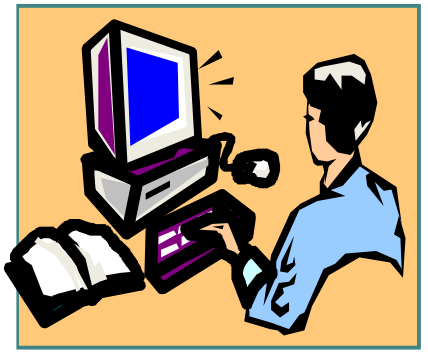
- **Connect is a module built to support interception of requests in a modular approach.**

```
var connect = require('connect');
var app = connect();
var logger = function(req, res, next) {
    console.log(req.method, req.url);
    next();
};
var helloWorld = function(req, res, next) {
    res.setHeader('Content-Type', 'text/plain');
    res.end('Hello World');
};
var byeWorld = function(req, res, next) {
    res.setHeader('Content-Type', 'text/plain');
    res.end('Bye World');
};
app.use(logger);
app.use('/hello',helloWorld);
app.use('/bye',byeWorld);
app.listen(3000);
console.log('Server running at localhost:3000');
```


Copyright © Capgemini 2017. All Rights Reserved

Demo

- **Working with connect module**



Copyright © Capgemini 2017. All Rights Reserved

Express.js Installation

- **The Express.js package comes in two flavors:**
- ***express-generator*: a global NPM package that provides the command-line tool for rapid app creation (scaffolding)**
- ***express*: a local package module in your Node.js app's node_modules folder**

Copyright © Capgemini 2017. All Rights Reserved

app.js

- **App.js is the main file in Express framework. A typical structure of the main Express.js file consists of the following areas**
  - 1. Require dependencies
  - 2. Configure settings
  - 3. Connect to database (optional)
  - 4. Define middleware
  - 5. Define routes
  - 6. Start the server

- **The order here is important, because requests travel from top to bottom in the chain of middleware.**

Copyright © Capgemini 2017. All Rights Reserved

app.js

- **First the dependencies need to be included with require()**

```
var express = require('express');
var http = require(http');
var path = require('path');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');
```

- **Then Express.js object is instantiated (Express.js uses a functional pattern):**

```
var app = express();
```


Copyright © Capgemini 2017. All Rights Reserved

app.js

- **One of the ways to configure Express.js settings is to use app.set(), with the name of the setting and the value.**

```
/* port: a number on which our server should listen to requests */
app.set('port', process.env.PORT || 3000);
/* views: absolute path to the folder with template */
app.set('views', path.join(__dirname, 'views'));
/* view engine: file extension for the template files */
app.set('view engine', 'jade');
```

- **Middleware is the backbone of the Express.js framework and it comes in two flavors.**
  - Defined in external (third-party) modules, such as bodyParser.json from Connect/Express.js body-parser: app.use(bodyParser.json());
  - Defined in the app or its modules, such as app.use(function(req, res, next){...});

Copyright © Capgemini 2017. All Rights Reserved

app.js

- **Routes are processed in the order they are defined. Usually, routes are put after middleware, but some middleware may be placed following the routes. A good example of such middleware, found after a routes, is error handler.**
- **The way routes are defined in Express.js is with helpers app.VERB(url, fn1, fn2, ..., fn), where fnNs are request handlers, url is on a URL pattern in RegExp, and VERB values are as follows:**
  - all: catch every request (all methods)
  - get: catch GET requests
  - post: catch POST requests
  - put: catch PUT requests
  - del: catch DELETE requests

Copyright © Capgemini 2017. All Rights Reserved

app.js

- **Finally to start the server, we need to use createServer method from the core http module. In this method, the system passes the Express.js app object with all the settings and routes**

```
http.createServer(app).listen(app.get('port'), function(){
    console.log('Express server listening on port ' + app.get('port'));
});
```


Copyright © Capgemini 2017. All Rights Reserved

Steps for creating Express.js Application

- **Step – 1 : Create a folder named SampleApp**
- **Step – 2 : Create package.json with the following schema**

```
{
  "name": "SampleApp",
  "version": "1.0.0",
  "description": "Sample Application",
  "main": "app.js",
  "dependencies": {
    "body-parser": "1.10.1",
    "cookie-parser": "1.3.3",
    "express": "4.10.7",
    "jade": "1.8.2"
  },
   "scripts": {
    "start": "node app"
  },
  "author": "Karthik M <karthik.muthukrishnan@igate.com>",
  "license": "ISC"
}
```

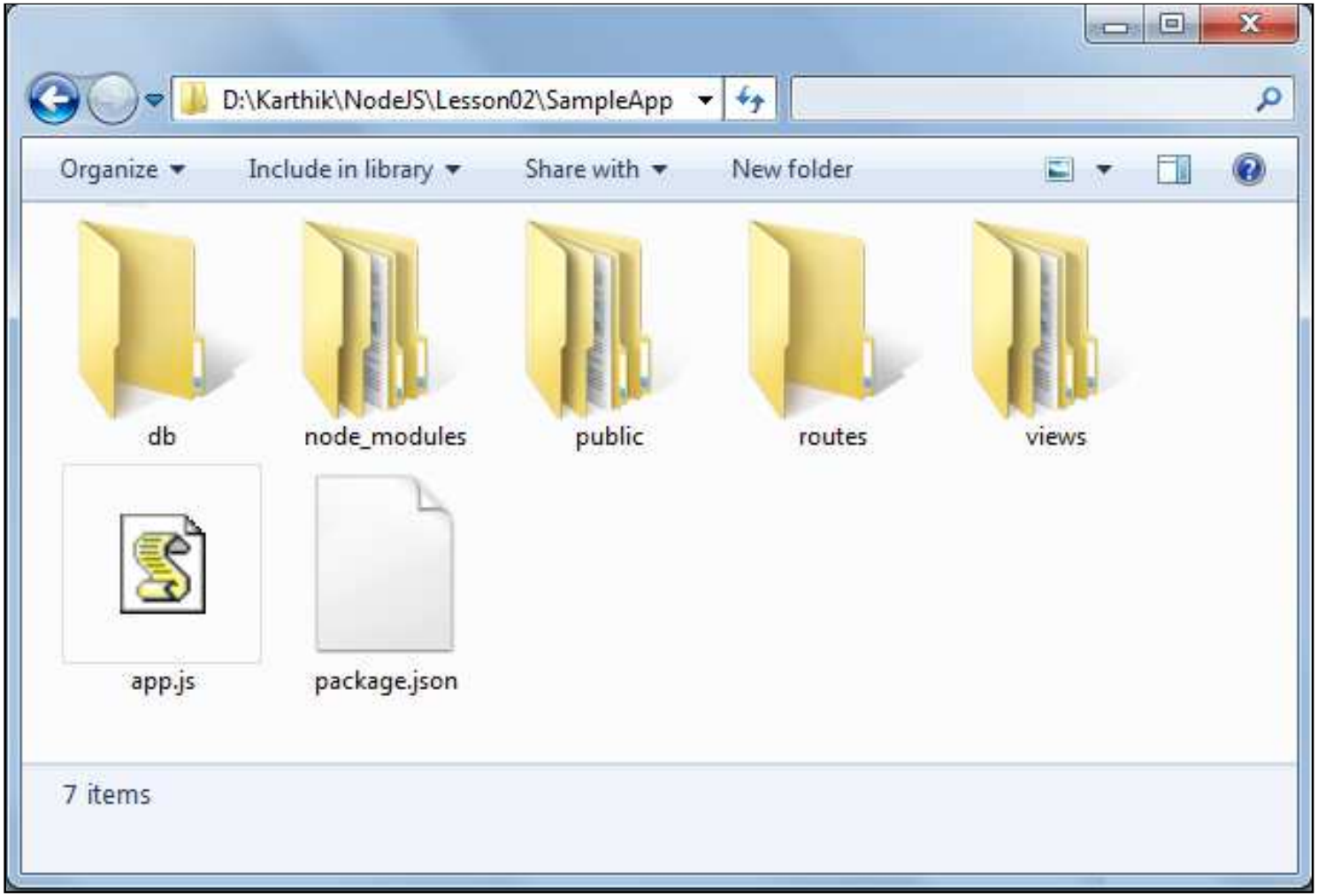

Copyright © Capgemini 2017. All Rights Reserved

Steps for creating Express.js Application

- **Step – 3 : Install the dependencies using npm install command**
  - D:\Karthik\NodeJS\Lesson02\SampleApp>npm install
- **Step – 4 : Create the following folders under SampleApp folder**
  - **public** : All the static (front-end) files like HTML
  - **public/css** : Stylesheet files
  - **public/img :** images
  - **public/js** : Scripts
  - **db** : Seed data and scripts for MongoDB
  - **views** : Jade (or any other template engine) files
  - **views/includes** : Partial / include files
  - **routes :** Node.js modules that contain request handlers
- **Step – 5 : Create the main file named app.js**

Copyright © Capgemini 2017. All Rights Reserved

Steps for creating Express.js Application



Copyright © Capgemini 2017. All Rights Reserved

Steps for creating Express.js Application

- **Step – 6 : Type the following contents in app.js**

```
var express = require('express');
var http = require('http');
var path = require('path');

var app = express();

app.set('port', process.env.PORT || 3000);
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');

app.all('*', function(req, res) {
    res.render('index',{title: 'IGATE Corporate University'});
});

http.createServer(app).listen(app.get('port'),function(){
    console.log('Express.js server listening on port ' +app.get('port'));
});
```


Copyright © Capgemini 2017. All Rights Reserved

Steps for creating Express.js Application

- **Step – 7 : Create index.jade under views folder and type the following contents**

```
doctype html
html
    head
        title= title
    body
        h1= title
        p Welcome to #{title}
```

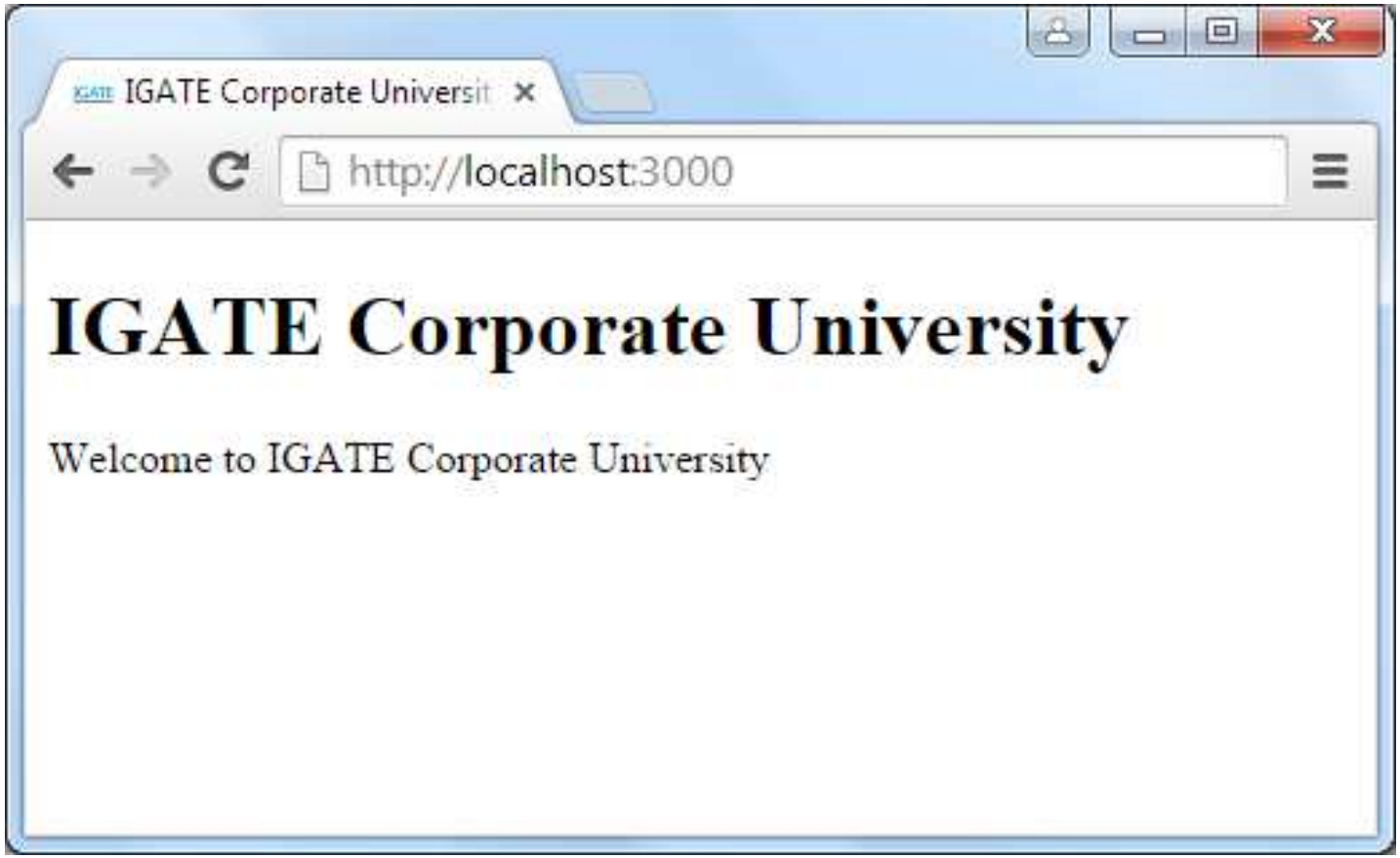
- **Step – 8 : Start the app by typing *npm start* in command prompt.**

```
D:\Karthik\NodeJS\Lesson02\SampleApp>npm start

> SampleApp@1.0.0 start D:\Karthik\NodeJS\Lesson02\SampleApp
> node app

Express.js server listening on port 3000
```


Copyright © Capgemini 2017. All Rights Reserved

Steps for creating Express.js Application

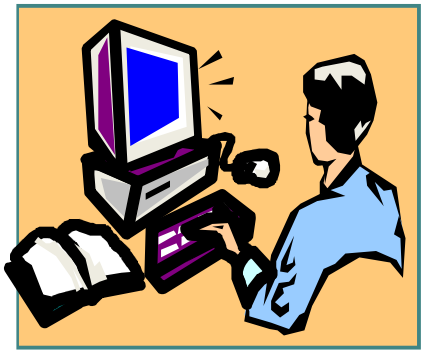
- **Step – 9 : Open browser and type *http://localhost:3000* to view the SampleApp**



Copyright © Capgemini 2017. All Rights Reserved

Demo

- **Creating ExpressJS Sample Application**



Copyright © Capgemini 2017. All Rights Reserved

application object properties & methods

Property/Method	Description
`app.set(name, value)`	Sets app-specific properties
`app.get(name)`	Retrieves value set by `app.set()`
`app.enable(name)`	Enables a setting in the app
`app.disable(name)`	Disables a setting in the app
`app.enabled(name)`	Checks if a setting is enabled
`app.disabled(name)`	Checks if a setting is disabled
`app.configure([env], callback)`	Sets app settings conditionally based on the development environment
`app.use([path], function)`	Loads a middleware in the app
`app.engine(ext, callback)`	Registers a template engine for the app
`app.param([name], callback)`	Adds logic to route parameters
`app.VERB(path, [callback...], callback)`	Defines routes and handlers based on HTTP verbs
`app.all(path, [callback...], callback)`	Defines routes and handlers for all HTTP verbs
`app.locals`	The object to store variables accessible from any view
`app.render(view, [options], callback)`	Renders view from the app
`app.routes`	A list of routes defined in the app
`app.listen()`	Binds and listen for connections

Copyright © Capgemini 2017. All Rights Reserved

application object properties & methods

Copyright © Capgemini 2017. All Rights Reserved

request object properties & methods

Property/Method	Description
`req.params`	Holds the values of named routes parameters
`req.params(name)`	Returns the value of a parameter from named routes or `GET` params or `POST` params
`req.query`	Holds the values of a `GET` form submission
`req.body`	Holds the values of a `POST` form submission
`req.files`	Holds the files uploaded via a form
`req.route`	Provides details about the current matched route
`req.cookies`	Cookie values
`req.signedCookies`	Signed cookie values
`req.get(header)`	Gets the request HTTP header
`req.accepts(types)`	Checks if the client accepts the media types
`req.accepted`	A list of accepted media types by the client
`req.is(type)`	Checks if the incoming request is of the particular media type

Copyright © Capgemini 2017. All Rights Reserved

request object properties & methods

Property/Method	Description
`req.ip`	The IP address of the client
`req.ips`	The IP address of the client, along with that of the proxies it is connected through
`req.stale`	Checks if the request is stale
`req.xhr`	Checks if the request came via an AJAX request
`req.protocol`	The protocol used for making the request
`req.secure`	Checks if it is a secure connection
`req.subdomains`	Subdomains of the host domain name
`req.url`	The request path, along with any query parameters
`req.originalUrl`	Used as a backup for `req.url`
`req.acceptedLanguages`	A list of accepted languages by the client
`req.acceptsLanguage(langauge)`	Checks if the client accepts the language
`req.acceptedCharsets`	A list of accepted charsets by the client
`req.acceptsCharsets(charset)`	Checks if the client accepts the charset
`req.host`	Hostname from the HTTP header

Copyright © Capgemini 2017. All Rights Reserved

response object properties & methods

Property/Method	Description
`res.status(code)`	Sets the HTTP response code
`res.set(field, [value])`	Sets response HTTP headers
`res.get(header)`	Gets the response HTTP header
`res.cookie(name, value, [options])`	Sets cookie on the client
`res.clearCookie(name, [options])`	Deletes cookie on the client
`res.redirect([status], url)`	Redirects the client to a URL, with an optional HTTP status code
`res.location`	The location value of the response HTTP header
`res.charset`	The charset value of the response HTTP header
`res.send([body|status], [body])`	Sends an HTTP response object, with an optional HTTP response code
`res.json([status|body], [body])`	Sends a JSON object for HTTP response, along with an optional HTTP response code

Copyright © Capgemini 2017. All Rights Reserved

response object properties & methods

Property/Method	Description
`res.jsonp([status|body], [body])`	Sends a JSON object for HTTP response with JSONP support, along with an optional HTTP response code
`res.type(type)`	Sets the media type HTTP response header
`res.format(object)`	Sends a response conditionally, based on the request HTTP Accept header
`res.attachment([filename])`	Sets response HTTP header Content-Disposition to attachment
`res.sendfile(path, [options], [callback])`	Sends a file to the client
`res.download(path, [filename], [callback])`	Prompts the client to download a file
`res.links(links)`	Sets the HTTP Links header
`res.locals`	The object to store variables specific to the view rendering a request
`res.render(view, [locals], callback)`	Renders a view

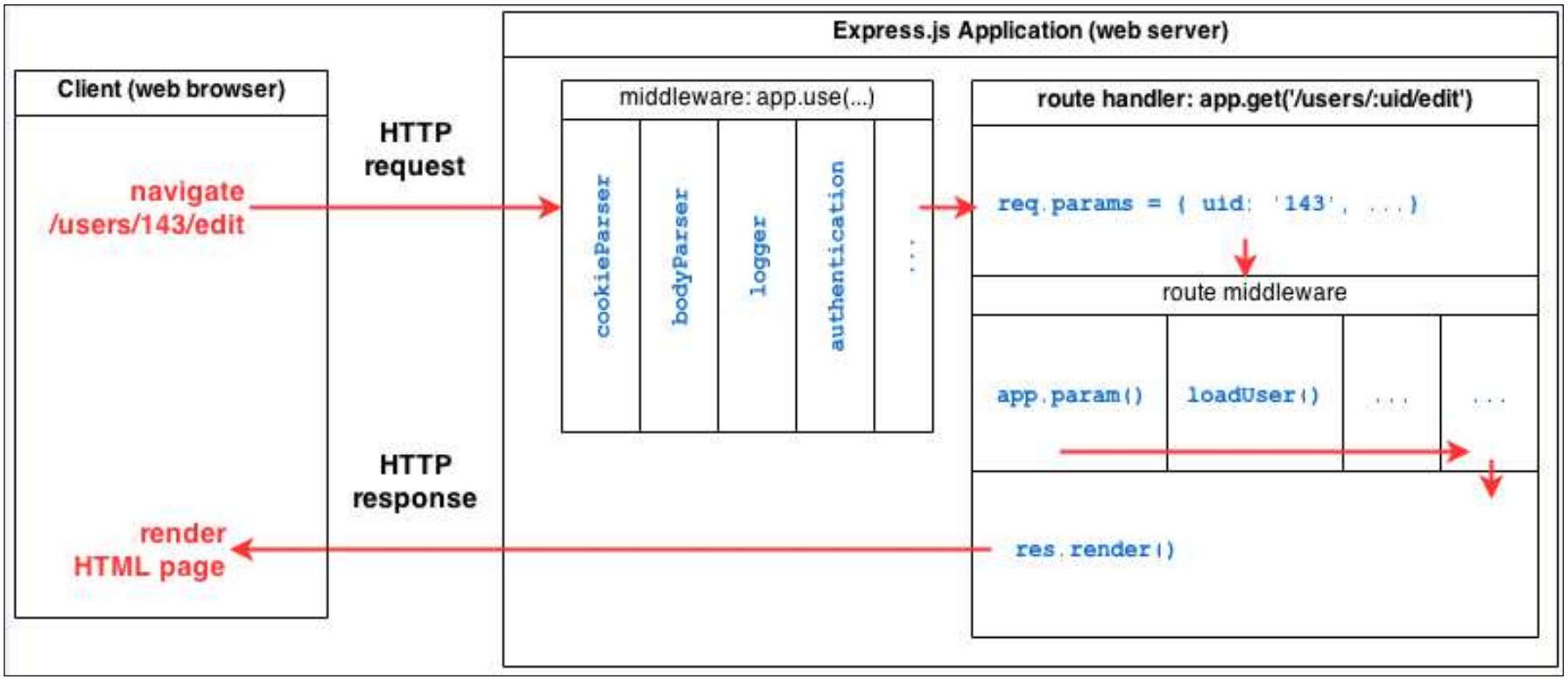
Copyright © Capgemini 2017. All Rights Reserved

How Express.js works

- **Express.js usually has an entry point aka, a main file. Most of the time, this is the file that we start with the node command or export as a module. In the main file we do the following**
  - Include third-party dependencies as well as our own modules, such as controllers, utilities, helpers, and models
  - Configure Express.js app settings such as template engine and its file extensions
  - Connect to databases such as MongoDB, Redis, or MySQL (optional)
  - Define middlewares and routes
  - Start the app and Export the app as a module (optional)
- **When the Express.js app is running, it's listens to requests. Each incoming request is processed according to a defined chain of middleware and routes, starting from top to bottom.**
- **Routes / middleware that are higher in the file have precedence over the lower definitions.**

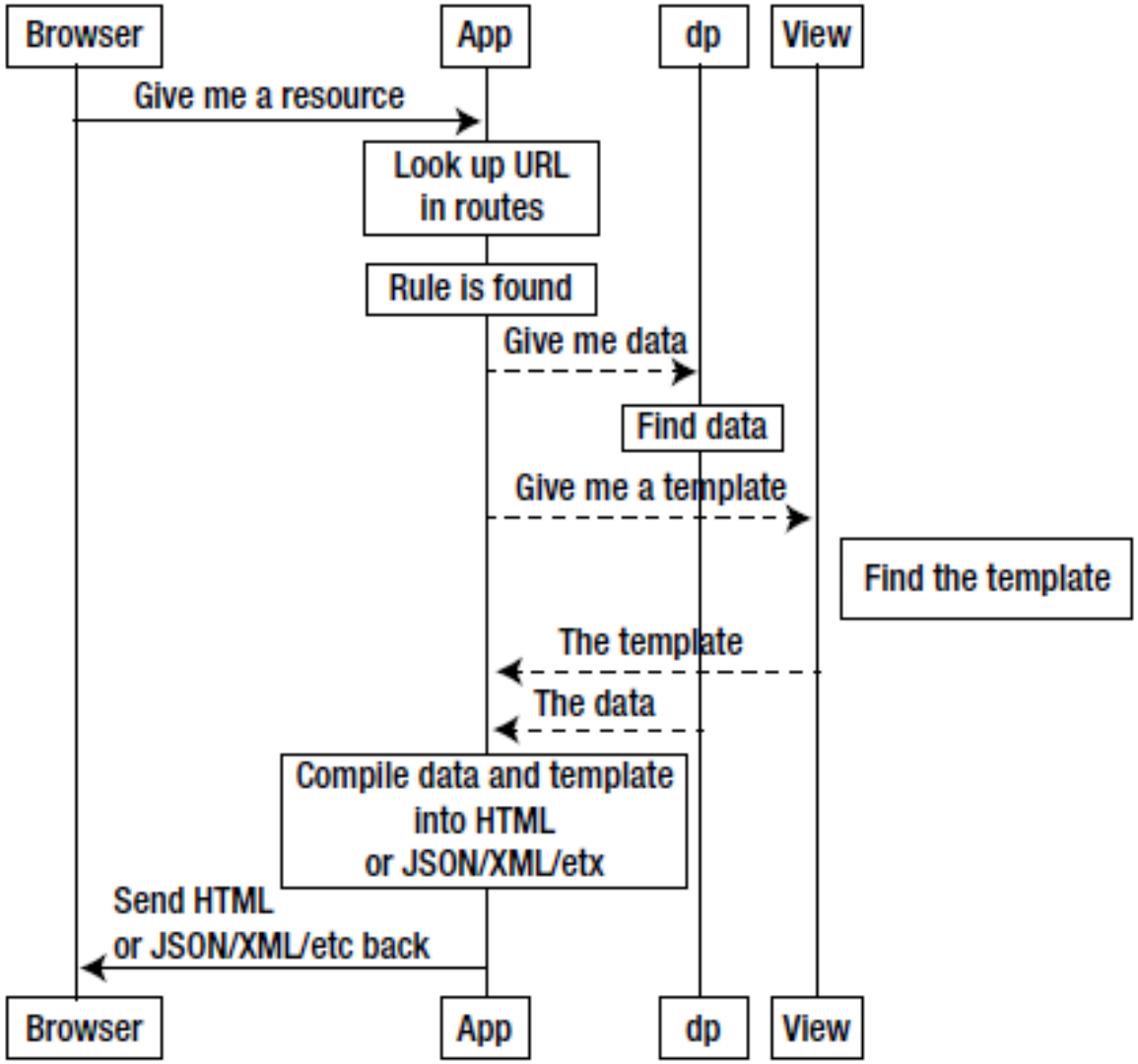
Copyright © Capgemini 2017. All Rights Reserved

How Express.js works



Copyright © Capgemini 2017. All Rights Reserved

Request flow in Express



Copyright © Capgemini 2017. All Rights Reserved

Request flow in Express

- **In Express server the request flow will be :**
  - Route → Route Handler → Template → HTML
- **The route defines the URL schema. It captures the matching request and passed on control to the corresponding route handler**
- **The route handler processes the request and passes the control to a template.**
- **The template constructs the HTML for the response and sends it to the browser.**
- **The route handler need not always pass the control to a template, it can optionally send the response to a request directly.**

Copyright © Capgemini 2017. All Rights Reserved

Using middleware

- **A middleware is a JavaScript function to handle HTTP requests to an Express app.**
- **It can manipulate the request and the response objects or perform an isolated action, or terminate the request flow by sending a response to the client, or pass on the control to the next middleware.**
- **A middleware can:**
  - Execute any code.
  - Make changes to the request and the response objects.
  - End the request-response cycle.
  - Call the next middleware in the stack.
- **If the current middleware does not end the request-response cycle, it must call next() to pass control to the next middleware, otherwise the request will be left hanging.**

Copyright © Capgemini 2017. All Rights Reserved

Types of middleware

- **An Express application can use the following kinds of middleware:**
- **Application-level middleware**
- **Router-level middleware**
- **Built-in middleware**
- **Third-party middleware**

Copyright © Capgemini 2017. All Rights Reserved

Application level middleware

- **Application level middleware are bound to an instance of express, using app.use() and app.VERB().**

```
var app = express();
/*  a middleware with no mount path; gets executed for every request to the app  */
app.use(function (req, res, next) {
        console.log('Time:', Date.now());        next();
});
/* a middleware mounted on /user/:id; will be executed for any type of HTTP request to
/user/:id */
app.use('/user/:id', function (req, res, next) {
        console.log('Request Type:', req.method);      next();
});
/* a route and its handler function (middleware system) which handles GET requests to
/user/:id  */
app.get('/user/:id', function (req, res, next) {
        res.send('USER');
});
```


Copyright © Capgemini 2017. All Rights Reserved

Router level middleware

- **Router level middleware are loaded using router.use() and router.VERB().**

```
var app = express();
var router = express.Router();
/* a middleware with no mount path, gets executed for every request to the router */
router.use(function (req, res, next) {
        console.log('Time:', Date.now());
        next();
});
/* a middleware sub-stack which handles GET requests to /user/:id */
router.get('/user/:id', function (req, res, next) {
        // if user id is 0, skip to the next router
        if (req.params.id == 0) next('route');
        // else pass the control to the next middleware in this stack
        else next(); //
}, function (req, res, next) {
        // render a regular page
        res.render('regular');
});
/* mount the router on the app */
app.use('/', router);
```


Copyright © Capgemini 2017. All Rights Reserved

External middleware

- **The Express core is minimal, yet the team behind it provides various predefined middleware to handle common web development features.**
- **Those middleware vary in size and functionality and extend Express to provide a better framework support.**
- **The popular Express middleware are as follows:**
  - **morgan**: This is an HTTP request logger middleware.
  - **body-parser**: This is a body-parsing middleware that is used to parse the request body, and it supports various request types.
  - **method-override:** This is a middleware that provides HTTP verb support such as PUT or DELETE in places where the client doesn't support it.
  - **cookie-parser:** This is a cookie-parsing middleware that populates the req.cookies object.
  - **express-session**: This is a session middleware used to support persistent sessions.

Copyright © Capgemini 2017. All Rights Reserved

Built-in middleware

- **Except for express.static, all of Express' previously included middleware are now in separate repo.**
- **express.static is based on serve-static, and is responsible for serving the static assets of an Express application. We can have more than one static directory per app.**

```
var options = {
  dotfiles: 'ignore',
  etag: false,
  extensions: ['htm', 'html'],
  index: false,
  maxAge: '1d',
  redirect: false,
  setHeaders: function (res, path, stat) {
    res.set('x-timestamp', Date.now())
  }
};
app.use(express.static('public', options));
```


Copyright © Capgemini 2017. All Rights Reserved

Third-party middleware

- **Express is a routing and middleware web framework with minimal functionality of its own. Functionality to Express apps are added via third-party middleware.**
- **Install the node module for the required functionality and loaded it in your app at the application level or at the router level.**

```
$ npm install cookie-parser
```

```
var express = require('express');
var app = express();
var cookieParser = require('cookie-parser');

// load the cookie parsing middleware
app.use(cookieParser());
```


Copyright © Capgemini 2017. All Rights Reserved

Express 4.0 Router

- **Express 4.0 comes with the new Router.**
- **Router is like a mini express application. It doesn't bring in views or settings, but provides us with the routing APIs like .use, .get, .param, and route.**
- **Creating instance of Router for application frontend routes**

```
var router = express.Router();

// home page route (http://localhost:3000)
router.get('/', function(req, res) {
    res.send('Home page!');
});

// about page route (http://localhost:3000/about)
router.get('/about', function(req, res) {
  res.send('About page!');
});

// apply the routes to our application
app.use('/', router);
```


Copyright © Capgemini 2017. All Rights Reserved

Express 4.0 Router

- **Route middleware in Express is a way to do something before a request is processed.**

```
var router = express.Router();

// route middleware that will happen on every request
router.use(function(req, res, next) {

   // log each request to the console
   console.log(req.method, req.url);

   // continue doing what we were doing and go to the route
   next();
});
// apply the routes to our application
app.use('/', router);
```


Copyright © Capgemini 2017. All Rights Reserved

Express 4.0 Router

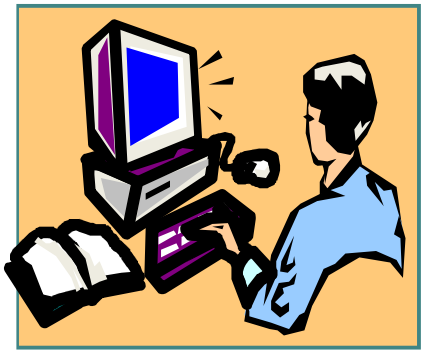
- **Route with parameters & Route Middleware for parameter**

```
var router = express.Router();
// route middleware to validate :name
router.param('name', function(req, res, next, name) {
   // do validation on name here
  console.log('doing name validations on ' + name);

   // once validation is done save the new item in the req
   req.name = name;
   // go to the next thing
   next();
});

// route with parameters (http://localhost:3000/hello/:name)
router.get('/hello/:name', function(req, res) {
   res.send('hello ' + req.params.name + '!');
});

// apply the routes to our application
app.use('/', router);
```


Copyright © Capgemini 2017. All Rights Reserved

Demo

- **ExpressApp-Middleware**



Copyright © Capgemini 2017. All Rights Reserved

Express.js Scaffolding

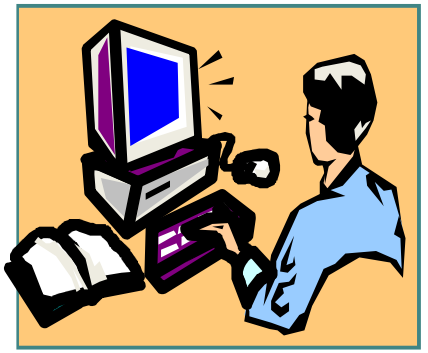
- **To generate a application skeleton for Express.js app, we need to run a terminal command express [options] [dir | appname] the options for which are the following:**
  - -e, --ejs: add EJS engine support (by default, Jade is used)
  - -c <engine>, --css <engine>: add stylesheet <engine> support, such as LESS, Stylus or Compass (by default, plain CSS is used)
  - -f, --force: force app generation on a nonempty directory

```
D:\Karthik\NodeJS\Lesson02\SampleApp>express -e -f

   create : .
   create : ./package.json
   create : ./app.js
   create : ./views
   create : ./views/index.ejs
   create : ./views/error.ejs
   create : ./bin
   create : ./bin/www
   create : ./routes
   create : ./routes/index.js
   create : ./routes/users.js
   create : ./public/images
   create : ./public
   create : ./public/javascripts
   create : ./public/stylesheets
   create : ./public/stylesheets/style.css

   install dependencies:
     $ cd . && npm install

   run the app:
     $ DEBUG=SampleApp ./bin/www
```


Copyright © Capgemini 2017. All Rights Reserved

Demo

- **GuestPhoneBook Application**



Copyright © Capgemini 2017. All Rights Reserved

Database and Session handling

Copyright © Capgemini 2017. All Rights Reserved

Introduction

- **MongoDB is one of the most popular and fastest growing open source NoSQL database.**
- **MongoDB is written in C++. It is fast and scalable.**
- **Most of the MongoDB functionalities can be accessed directly through JavaScript notation and we can make use of all of the standard JavaScript libraries with in it.**
- **It uses JSON for storing and manipulating the data. A JSON database returns query results that can be easily parsed, with little or no transformation, directly by JavaScript and most popular programming languages which reduces the amount of logic need to build into the application layer.**
- **MongoDB represents JSON documents in binary-encoded format called BSON(Binary JSON) behind the scenes. BSON extends the JSON model to provide additional data types and to be efficient for encoding and decoding within different languages.**

Copyright © Capgemini 2017. All Rights Reserved

Why MongoDB?

- **MongoDB is a cross-platform, document oriented database that provides, high performance, high availability and easy scalability**
- **It supports a wide range of Operating Systems (Windows, OSX, Linux)**
- **There are drivers for nearly any language including C/C++, Python, PHP, Ruby, Perl, .NET and Node.js.**
- **Document Oriented Storage i.e. Data is stored in the form of JSON style documents**
- **Index on any attribute**
- **Replication & High Availability**
- **Auto-Sharding**
- **Rich Queries**
- **Migrations and a constantly evolving schemas can be managed easier**

Copyright © Capgemini 2017. All Rights Reserved

MongoDB key terminologies

- **Database**
  - Database is a physical container for collections. Each database gets its own set of files on the file system. A single MongoDB server typically has multiple databases
- **Collection**
  - A *collection* may be considered as a table except there are no aligned columns.
- **Document**
  - Each of the entries or rows inside a collection is called a *document*. Each entry (row) can use varying dynamic schemas in key-value pairs.
  - Collections have dynamic schemas. This means that the documents within a single collection can have any number of different "shapes."
  - Inside a collection of Users there may be one entry with First name & Last name. Then another entry with First, Last, and Middle name, along with e-mail address and date of birth.
  - Documents are basically JSON data blocks stored in memory-mapped files which behave as separate entries in collections.

Copyright © Capgemini 2017. All Rights Reserved

SQL Terminology vs MongoDB Terminology

SQL Terms/Concepts	MongoDB Terms/Concepts
database	database
table	collection
row	document or BSON document
column	field
index	index
table joins	embedded documents and linking
primary key	primary key In MongoDB primary key is automatically set to the_id field.
aggregation (e.g. group by)	aggregation pipeline.

Copyright © Capgemini 2017. All Rights Reserved

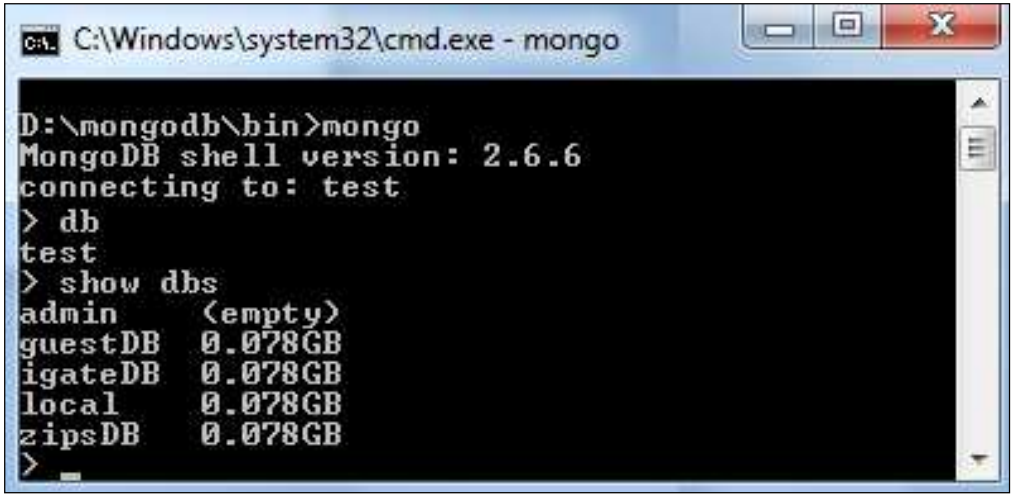
MongoDB vs Relational Databases

- **Relational databases save data in tables and rows.**
- **But in application development our objects are not simply tables and rows. It forces an application developer to write a mapping layer or use an ORM, to translate the object in memory and what is saved in the database. Mapping those to tables and rows can be quite a bit of pain.**
- **In MongoDB, there is no schema to define. There are no tables and no relationships between collections of objects.**
- **Every document you save in Mongo can be as flat and simple, or as complex as your application requires. This makes developer life much easier and your application code much cleaner and simpler.**
- **Further, two documents in the same collection may be different from each other since there is no schema governing the collection.**
- **Structuring a single object is clear and no complex joins**

Copyright © Capgemini 2017. All Rights Reserved

Starting the mongo shell

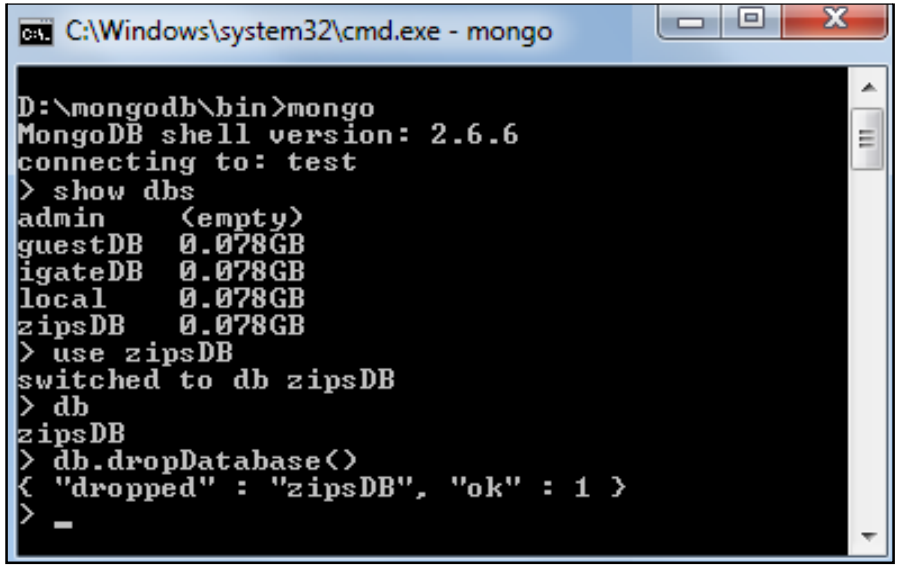
- **To start the mongo shell and connect to your MongoDB instance running on localhost(127.0.0.1) with default port(27017)**
  - **Go to <mongodb installation dir> :** *cd <mongodb installation dir>*
  - **To start mongo goto bin directory and type mongo ex:-** *D:\mongodb\bin>mongo*
- **To check your currently selected database, use the command *db*. Default(test)**
  - *db*
- **To list the databases, use the command show dbs.**
  - *show dbs*



Copyright © Capgemini 2017. All Rights Reserved

Creating and Dropping Database

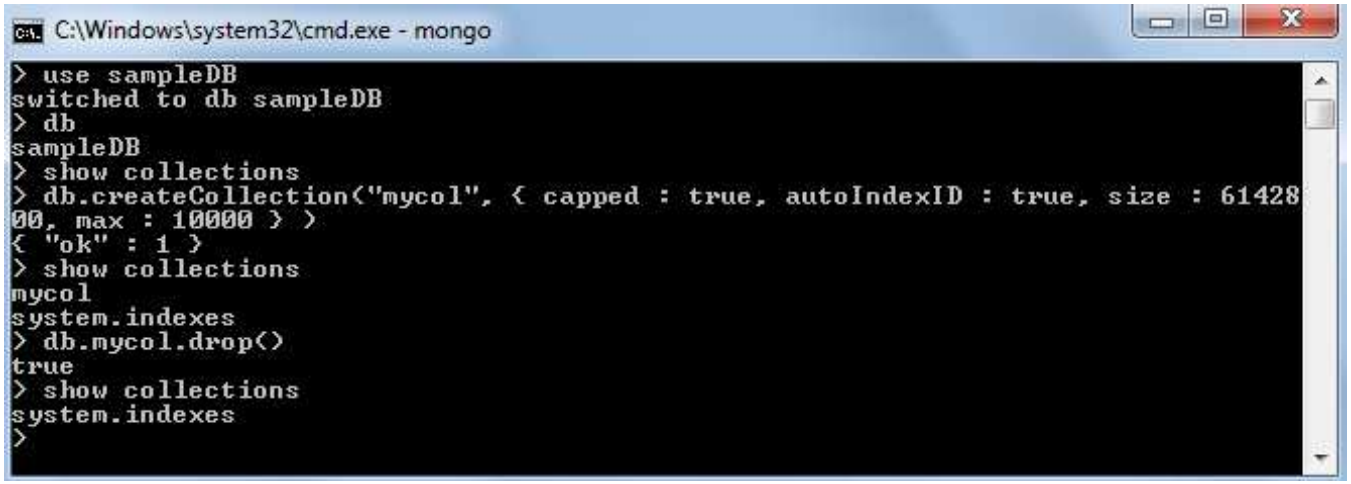
- **MongoDB *use DATABASE_NAME* is used to create database.**
- **If the database doesn't exists it creates a new database, otherwise it will return the existing database which can be used using *db*.**
  - *use DATABASE_NAME*
- **To display database you need to insert at least one document into it.**
- **db.dropDatabase() command is used to drop a existing database.**



Copyright © Capgemini 2017. All Rights Reserved

Creating and Dropping Collection

- **MongoDB *db.createCollection(collectionname, options)* is used to create collection. Options parameter is optional. Following is the list of options**
  - **capped** : Capped collection is a collection fixed size collection that automatically overwrites its oldest entries when it reaches its maximum size.
  - **autoIndexID**: If true, If true, automatically create index on _id field.
  - **size**: Specifies a maximum size in bytes for a capped collection.
  - **max**: Specifies the maximum number of documents allowed in the capped collection.
- **db.<collectionname>.drop() command is used to drop a collection**



Copyright © Capgemini 2017. All Rights Reserved

Importing and Exporting Collection

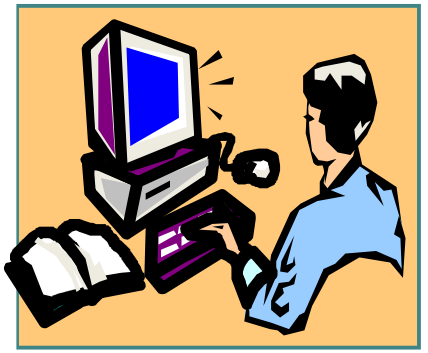
- **To import the collection from the file use *mongoimport* command**
  - *mongoimport --db <DBName> --collection <CollectionName> --file <FileName>*
- **To export the collection from the DB use *mongoexport* command**
  - *mongoexport --db <DBName> --collection <CollectionName> --out <OutputFileName>*

```
D:\mongodb\bin>mongo SampleDB
MongoDB shell version: 2.6.6
connecting to: SampleDB
> db
SampleDB
> show collections
> quit()

D:\mongodb\bin>mongoimport --db SampleDB --collection employees --file d:\Karthik\NodeJS\Lesson03\employees.json
connected to: 127.0.0.1
2015-02-01T14:08:05.130+0530 check 9 75
2015-02-01T14:08:05.135+0530 imported 75 objects

D:\mongodb\bin>mongoexport --db SampleDB --collection employees --out d:\Karthik\NodeJS\Lesson03\employees-backup.json
connected to: 127.0.0.1
exported 75 records

D:\mongodb\bin>mongo SampleDB
MongoDB shell version: 2.6.6
connecting to: SampleDB
> show collections
employees
system.indexes
>
```


Copyright © Capgemini 2017. All Rights Reserved

Demo

- **Importing and Exporting Collections**



Copyright © Capgemini 2017. All Rights Reserved

Querying MongoDB Documents – find()

- **MongoDB's find() method is used to query data from MongoDB collection**
  - ***db.COLLECTION_NAME.find()***
  - **db. *COLLECTION_NAME*.find().pretty()** is used to display the results in a formatted way.
  - **db. *COLLECTION_NAME*.count()** is used to count the number of documents in a collection.

```
D:\mongodb\bin>mongoimport --db SampleDB --collection locations --file d:\Karthik\NodeJS\Lesson03\locations.json
connected to: 127.0.0.1
2015-02-01T14:43:42.868+0530 imported 7 objects

D:\mongodb\bin>mongo SampleDB
MongoDB shell version: 2.6.6
connecting to: SampleDB
> db
SampleDB
> show collections
employees
locations
system.indexes
> db.locations.find()
{ "_id" : 1, "location" : "Bangalore" }
{ "_id" : 2, "location" : "Chennai" }
{ "_id" : 3, "location" : "Gandhinagar" }
{ "_id" : 4, "location" : "Hyderabad" }
{ "_id" : 5, "location" : "Mumbai" }
{ "_id" : 6, "location" : "Noida" }
{ "_id" : 7, "location" : "Pune" }
> db.locations.count()
7
>
```


Copyright © Capgemini 2017. All Rights Reserved

Querying MongoDB Documents – pretty()

- **To print the results in formatted way use pretty()**

```
D:\mongodb\bin>mongo SampleDB
MongoDB shell version: 2.6.6
connecting to: SampleDB
> db
SampleDB
> show collections
employees
locations
system.indexes
> db.employees.find({_id:714709}).pretty()
{
        "_id" : 714709,
        "name" : {
                "first" : "Karthik",
                "last" : "Muthukrishnan"
        },
        "doj" : "2010-04-12T00:00:00.000Z",
        "location" : "Bangalore",
        "isActive" : true,
        "email" : "Karthik.Muthukrishnan@igate.com",
        "qualifications" : [
                "B.Sc(CS)",
                "M.C.A"
        ]
}
> db.employees.find({_id:714709},{name:1,email:1}).pretty()
{
        "_id" : 714709,
        "name" : {
                "first" : "Karthik",
                "last" : "Muthukrishnan"
        },
        "email" : "Karthik.Muthukrishnan@igate.com"
}
> db.employees.find({_id:714709},{name:1,email:1,_id:0}).pretty()
{
        "name" : {
                "first" : "Karthik",
                "last" : "Muthukrishnan"
        },
        "email" : "Karthik.Muthukrishnan@igate.com"
}
>
```


Copyright © Capgemini 2017. All Rights Reserved

Querying MongoDB Documents – findOne()

- **findOne() returns a single document, where as find() returns a cursor.**

```
> db
SampleDB
> show collections
employees
locations
system.indexes
> db.employees.find()[36].doj
2010-04-12T00:00:00.000Z
> new Date(db.employees.find()[36].doj).toLocaleDateString()
Monday, April 12, 2010
> db.employees.findOne()
{
        "_id" : 2072,
        "name" : {
                "first" : "Rohini",
                "last" : "Vijayan"
        },
        "doj" : "1995-07-31T00:00:00.000Z",
        "location" : "Pune",
        "isActive" : true,
        "email" : "rohini.vijayan@igate.com",
        "qualifications" : [
                "B.Com",
                "D.C.A"
        ]
}
> db.employees.findOne().doj
1995-07-31T00:00:00.000Z
> new Date(db.employees.findOne().doj).toLocaleDateString()
Monday, July 31, 1995
>
```


Copyright © Capgemini 2017. All Rights Reserved

Comparison Operators

Name	Description
$gt	Matches values that are greater than the value specified in the query.
$gte	Matches values that are greater than or equal to the value specified in the query.
$in	Matches any of the values that exist in an array specified in the query.
$lt	Matches values that are less than the value specified in the query.
$le	Matches values that are less than or equal to the value specified in the query.
$ne	Matches all values that are not equal to the value specified in the query.
$nin	Matches values that do not exist in an array specified to the query.

Copyright © Capgemini 2017. All Rights Reserved

Comparison Operators – gte & lte

```
> db
SampleDB
> show collections
employees
locations
system.indexes
> db.employees.find({ _id: { $gte: 700000, $lte: 715000 }},{ name : 1 })
{ "_id" : 705062, "name" : { "first" : "Rashmi", "last" : "Keshavamurthy" } }
{ "_id" : 707224, "name" : { "first" : "Latha", "last" : "Subramanian" } }
{ "_id" : 714709, "name" : { "first" : "Karthik", "last" : "Muthukrishnan" } }
>
```


Copyright © Capgemini 2017. All Rights Reserved

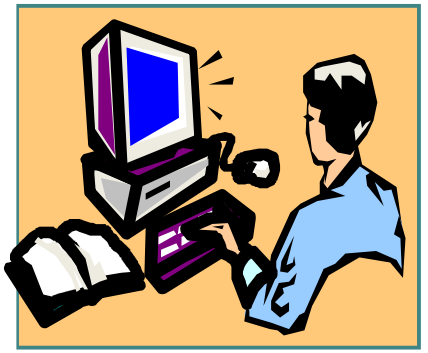
MongoDB Comparison Operators – in & nin

```
> db
SampleDB
> show collections
employees
locations
system.indexes
>
> db.employees.find({ 'name.first' : { $in: ['Vaishali','Veena'] }},{ name : 1, _id : 0 })
{ "name" : { "first" : "Veena", "last" : "Deshpande" } }
{ "name" : { "first" : "Vaishali", "last" : "Kulkarni" } }
{ "name" : { "first" : "Vaishali", "last" : "Kunchur" } }
{ "name" : { "first" : "Vaishali", "last" : "Kasture" } }
{ "name" : { "first" : "Veena", "last" : "Keshavalu" } }
{ "name" : { "first" : "Vaishali", "last" : "Srivastava" } }
>
> db.employees.find({ 'location' : { $nin: ['Bangalore','Mumbai','Pune'] }},{ name : 1, location : 1, _id : 0 })
{ "name" : { "first" : "Karthikeyan", "last" : "Ramanathan" }, "location" : "Chennai" }
{ "name" : { "first" : "Rathnajothi", "last" : "Perumalsamy" }, "location" : "Chennai" }
{ "name" : { "first" : "Hema", "last" : "Gandhi" }, "location" : "Chennai" }
{ "name" : { "first" : "Selvalakshmi", "last" : "Palanichelvam" }, "location" : "Chennai" }
{ "name" : { "first" : "Balachander", "last" : "Meghraj" }, "location" : "Chennai" }
{ "name" : { "first" : "Abishek", "last" : "Radhakrishnan" }, "location" : "Chennai" }
{ "name" : { "first" : "Kalaivani", "last" : "Rajabadhar" }, "location" : "Chennai" }
>
> db.employees.find({ 'qualifications' : { $in: ['M.E(CSE)'] }},{ name : 1, qualifications : 1, _id : 0 }).pretty()
{
        "name" : {
                "first" : "Anil",
                "last" : "Patil"
        },
        "qualifications" : [
                "B.Sc(CS)",
                "M.E(CSE)"
        ]
}
{
        "name" : {
                "first" : "Roshi",
                "last" : "Saxena"
        },
        "qualifications" : [
                "B.Tech(CSE)",
                "M.E(CSE)"
        ]
}
>
```


Copyright © Capgemini 2017. All Rights Reserved

Demo

- **Comparison Operators**



Copyright © Capgemini 2017. All Rights Reserved

Logical Operators

Name	Description
$and	Joins query clauses with a logical AND returns all documents that match the conditions of both clauses.
$nor	Joins query clauses with a logical NOR returns all documents that fail to match both clauses.
$not	Inverts the effect of a query expression and returns documents that do not match the query expression.
$or	Joins query clauses with a logical OR returns all documents that match the conditions of either clause.

Copyright © Capgemini 2017. All Rights Reserved

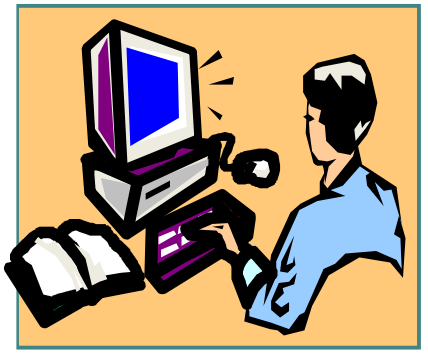
Logical Operators – or, not & and

```
>
> db
SampleDB
>
> db.employees.find({ $or : [{'name.first' : 'Vaishali'},{'name.last' : 'Kulkarni'}] },{ name : 1, _id : 0 })
{ "name" : { "first" : "Vaishali", "last" : "Kulkarni" } }
{ "name" : { "first" : "Vaishali", "last" : "Kunchur" } }
{ "name" : { "first" : "Vaishali", "last" : "Kasture" } }
{ "name" : { "first" : "Shamika", "last" : "Kulkarni" } }
{ "name" : { "first" : "Vaishali", "last" : "Srivastava" } }
{ "name" : { "first" : "Zainab", "last" : "Kulkarni" } }
>
> db.employees.find({ $nor : [ { 'isActive' : true }, { 'location' : 'Bangalore'  } ] },{ name : 1, isActive : 1, _id : 0 })
{ "name" : { "first" : "Anagha", "last" : "Narvekar" }, "isActive" : false }
{ "name" : { "first" : "Shrilata", "last" : "Tavargeri" }, "isActive" : false }
{ "name" : { "first" : "Pravin", "last" : "Surve" }, "isActive" : false }
{ "name" : { "first" : "Ajit", "last" : "Jog" }, "isActive" : false }
{ "name" : { "first" : "Samant", "last" : "Gour" }, "isActive" : false }
{ "name" : { "first" : "Hareshkumar", "last" : "Chandiramani" }, "isActive" : false }
{ "name" : { "first" : "Mandar", "last" : "Ramdas" }, "isActive" : false }
{ "name" : { "first" : "Pushpendra", "last" : "Mishra" }, "isActive" : false }
{ "name" : { "first" : "Bhavna", "last" : "Beri" }, "isActive" : false }
>
> db.employees.find({ $and : [ { location : 'Bangalore' },{'name.last' : 'Kulkarni' } ] },{ name : 1, location : 1, _id : 0 })
{ "name" : { "first" : "Zainab", "last" : "Kulkarni" }, "location" : "Bangalore" }
>
```


Copyright © Capgemini 2017. All Rights Reserved

Demo

- **Logical Operators**



Copyright © Capgemini 2017. All Rights Reserved

MongoDB Additional operators

Name	Description
$all	Matches arrays that contain all elements specified in the query.
$exists	Matches documents that have the specified field.
$regex	Selects documents where values match a specified regular expression.
$where	Matches documents that satisfy a JavaScript expression.
$sort	Reorders the document stream by a specified sort key. Only the order changes; the documents remain unmodified. For each input document, outputs one document.
$limit	Passes the first n documents unmodified to the pipeline where n is the specified limit. For each input document, outputs either one document (for the first n documents) or zero documents (after the first n documents).
$skip	Skips the first n documents where n is the specified skip number and passes the remaining documents unmodified to the pipeline. For each input document, outputs either zero documents (for the first n documents) or one document (if after the first n documents).

Copyright © Capgemini 2017. All Rights Reserved

MongoDB Additional operators - all

```
>
> db
SampleDB
> show collections
employees
locations
system.indexes
>
> db.employees.find({ 'qualifications' : { $all: ['B.Sc(CS)','M.E(CSE)'] }},
                        { name : 1, qualifications : 1, _id : 0 }).pretty()
{
        "name" : {
                "first" : "Anil",
                "last" : "Patil"
        },
        "qualifications" : [
                "B.Sc(CS)",
                "M.E(CSE)"
        ]
}
>
```


Copyright © Capgemini 2017. All Rights Reserved

MongoDB Additional operators – exists & not

```
>
> db
SampleDB
>
> db.employees.find({ qualifications : { $exists : false } },{ name : 1, _id : 0 })
{ "name" : { "first" : "Pramod", "last" : "Patwardhan" } }
{ "name" : { "first" : "Anagha", "last" : "Narvekar" } }
{ "name" : { "first" : "Shrilata", "last" : "Tavargeri" } }
{ "name" : { "first" : "Vinay", "last" : "Gupta" } }
{ "name" : { "first" : "Ajit", "last" : "Jog" } }
{ "name" : { "first" : "Samant", "last" : "Gour" } }
{ "name" : { "first" : "Hareshkumar", "last" : "Chandiramani" } }
{ "name" : { "first" : "Satyen", "last" : "Nande" } }
{ "name" : { "first" : "Mandar", "last" : "Ramdas" } }
{ "name" : { "first" : "Suresh", "last" : "Kumar" } }
{ "name" : { "first" : "Naveen", "last" : "Bandi" } }
{ "name" : { "first" : "Sudhip", "last" : "Rao" } }
{ "name" : { "first" : "Pushpendra", "last" : "Mishra" } }
{ "name" : { "first" : "Bhavna", "last" : "Beri" } }
{ "name" : { "first" : "Bhushan", "last" : "Bhupta" } }
{ "name" : { "first" : "Shefali", "last" : "Pathak" } }
{ "name" : { "first" : "Anjana", "last" : "Pathare" } }
>
> db.employees.find( { qualifications : { $not : { $exists : true } } },{ name : 1, _id : 0 })
{ "name" : { "first" : "Pramod", "last" : "Patwardhan" } }
{ "name" : { "first" : "Anagha", "last" : "Narvekar" } }
{ "name" : { "first" : "Shrilata", "last" : "Tavargeri" } }
{ "name" : { "first" : "Vinay", "last" : "Gupta" } }
{ "name" : { "first" : "Ajit", "last" : "Jog" } }
{ "name" : { "first" : "Samant", "last" : "Gour" } }
{ "name" : { "first" : "Hareshkumar", "last" : "Chandiramani" } }
{ "name" : { "first" : "Satyen", "last" : "Nande" } }
{ "name" : { "first" : "Mandar", "last" : "Ramdas" } }
{ "name" : { "first" : "Suresh", "last" : "Kumar" } }
{ "name" : { "first" : "Naveen", "last" : "Bandi" } }
{ "name" : { "first" : "Sudhip", "last" : "Rao" } }
{ "name" : { "first" : "Pushpendra", "last" : "Mishra" } }
{ "name" : { "first" : "Bhavna", "last" : "Beri" } }
{ "name" : { "first" : "Bhushan", "last" : "Bhupta" } }
{ "name" : { "first" : "Shefali", "last" : "Pathak" } }
{ "name" : { "first" : "Anjana", "last" : "Pathare" } }
>
```


Copyright © Capgemini 2017. All Rights Reserved

MongoDB Additional operators - regex

```
>
> db
SampleDB
> db.employees.find( { 'name.first' : /Ka+/ },{ name : 1, _id : 0 })
{ "name" : { "first" : "Karthikeyan", "last" : "Ramanathan" } }
{ "name" : { "first" : "Karthik", "last" : "Muthukrishnan" } }
{ "name" : { "first" : "Kalaivani", "last" : "Rajabadhar" } }
{ "name" : { "first" : "Kavita", "last" : "Arora" } }
>
> db.employees.find( { 'name.first' : { $regex :  /[K]+/ } },{ name : 1, _id : 0 })
{ "name" : { "first" : "Karthikeyan", "last" : "Ramanathan" } }
{ "name" : { "first" : "Karthik", "last" : "Muthukrishnan" } }
{ "name" : { "first" : "Kalaivani", "last" : "Rajabadhar" } }
{ "name" : { "first" : "Kavita", "last" : "Arora" } }
>
> db.employees.find( { 'name.first' : { $regex :  /[K]+/ } },{ name : 1, _id : 0 }).count()
4
>
```


Copyright © Capgemini 2017. All Rights Reserved

MongoDB Additional operators - where

```
>
> db
SampleDB
>
> db.employees.find( { $where : 'this.name.last === "Kulkarni"' },{ name : 1, _id : 0 })
{ "name" : { "first" : "Vaishali", "last" : "Kulkarni" } }
{ "name" : { "first" : "Shamika", "last" : "Kulkarni" } }
{ "name" : { "first" : "Zainab", "last" : "Kulkarni" } }
>
> db.employees.find( { $where : 'this.location === "Chennai"' },{ name : 1, _id : 0 })
{ "name" : { "first" : "Karthikeyan", "last" : "Ramanathan" } }
{ "name" : { "first" : "Rathnajothi", "last" : "Perumalsamy" } }
{ "name" : { "first" : "Hema", "last" : "Gandhi" } }
{ "name" : { "first" : "Selvalakshmi", "last" : "Palanichelvam" } }
{ "name" : { "first" : "Balachander", "last" : "Meghraj" } }
{ "name" : { "first" : "Abishek", "last" : "Radhakrishnan" } }
{ "name" : { "first" : "Kalaivani", "last" : "Rajabadhar" } }
>
> var getKulkarni = function(){ return this.name.last === 'Kulkarni' }
>
> db.employees.find( { $where : getKulkarni },{ name : 1, _id : 0 })
{ "name" : { "first" : "Vaishali", "last" : "Kulkarni" } }
{ "name" : { "first" : "Shamika", "last" : "Kulkarni" } }
{ "name" : { "first" : "Zainab", "last" : "Kulkarni" } }
>
> db.employees.find( getKulkarni , { name : 1, _id : 0 } )
{ "name" : { "first" : "Vaishali", "last" : "Kulkarni" } }
{ "name" : { "first" : "Shamika", "last" : "Kulkarni" } }
{ "name" : { "first" : "Zainab", "last" : "Kulkarni" } }
>
>
```


Copyright © Capgemini 2017. All Rights Reserved

MongoDB Additional operators - sort

```
>
> db
SampleDB
> db.employees.find( { 'name.first' : { $regex :  /[K]+/ } },
          { name : 1, _id : 0 }).sort( { 'name.first' : 1 })
{ "name" : { "first" : "Kalaivani", "last" : "Rajabadhar" } }
{ "name" : { "first" : "Karthik", "last" : "Muthukrishnan" } }
{ "name" : { "first" : "Karthikeyan", "last" : "Ramanathan" } }
{ "name" : { "first" : "Kavita", "last" : "Arora" } }
>
> db.employees.find( { 'name.first' : { $regex :  /[K]+/ } },
         { name : 1, _id : 0 }).sort( { 'name.first' : -1 })
{ "name" : { "first" : "Kavita", "last" : "Arora" } }
{ "name" : { "first" : "Karthikeyan", "last" : "Ramanathan" } }
{ "name" : { "first" : "Karthik", "last" : "Muthukrishnan" } }
{ "name" : { "first" : "Kalaivani", "last" : "Rajabadhar" } }
>
> db.employees.find( { 'name.first' : { $regex :  /[K]+/ } },
  { name : 1, _id : 0 }).sort({ location:1 , 'name.first':-1 })
{ "name" : { "first" : "Karthik", "last" : "Muthukrishnan" } }
{ "name" : { "first" : "Karthikeyan", "last" : "Ramanathan" } }
{ "name" : { "first" : "Kalaivani", "last" : "Rajabadhar" } }
{ "name" : { "first" : "Kavita", "last" : "Arora" } }
>
```


Copyright © Capgemini 2017. All Rights Reserved

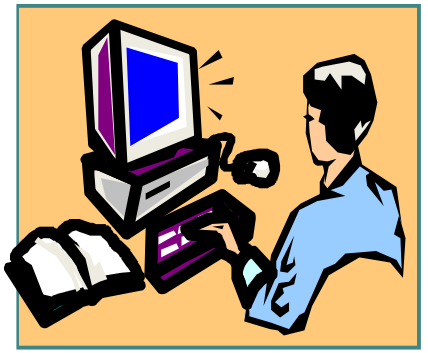
MongoDB Additional operators – limit & skip

```
>
> db
SampleDB
> db.locations.find().sort({ location : -1 })
{ "_id" : 7, "location" : "Pune" }
{ "_id" : 6, "location" : "Noida" }
{ "_id" : 5, "location" : "Mumbai" }
{ "_id" : 4, "location" : "Hyderabad" }
{ "_id" : 3, "location" : "Gandhinagar" }
{ "_id" : 2, "location" : "Chennai" }
{ "_id" : 1, "location" : "Bangalore" }
>
> db.locations.find().sort({ location : -1 }).limit(2)
{ "_id" : 7, "location" : "Pune" }
{ "_id" : 6, "location" : "Noida" }
>
> db.locations.find().sort({ location : 1 }).skip(0 * 2).limit(2)
{ "_id" : 1, "location" : "Bangalore" }
{ "_id" : 2, "location" : "Chennai" }
>
> db.locations.find().sort({ location : 1 }).skip(1 * 2).limit(2)
{ "_id" : 3, "location" : "Gandhinagar" }
{ "_id" : 4, "location" : "Hyderabad" }
>
> db.locations.find().sort({ location : 1 }).skip(2 * 2).limit(2)
{ "_id" : 5, "location" : "Mumbai" }
{ "_id" : 6, "location" : "Noida" }
>
> db.locations.find().sort({ location : 1 }).skip(3 * 2).limit(2)
{ "_id" : 7, "location" : "Pune" }
>
>
```


Copyright © Capgemini 2017. All Rights Reserved

Demo

- **Additional Operators**



Copyright © Capgemini 2017. All Rights Reserved

Inserting Document(s)

- **In MongoDB, to Insert a document or documents into a collection we need to use db.collection.insert()**

```
>
> db
SampleDB
> db.persons.count()
0
> db.persons.insert({name:'Karthik'})
WriteResult({ "nInserted" : 1 })
>
> db.persons.find()
{ "_id" : ObjectId("54ce287fdce2920aa624a655"), "name" : "Karthik" }
>
> db.persons.insert({_id : 1, name:'Abishek'})
WriteResult({ "nInserted" : 1 })
>
> db.persons.find()
{ "_id" : ObjectId("54ce287fdce2920aa624a655"), "name" : "Karthik" }
{ "_id" : 1, "name" : "Abishek" }
>
> db.persons.insert({_id : new ObjectId(), name:'Latha'})
WriteResult({ "nInserted" : 1 })
>
> db.persons.find()
{ "_id" : ObjectId("54ce287fdce2920aa624a655"), "name" : "Karthik" }
{ "_id" : 1, "name" : "Abishek" }
{ "_id" : ObjectId("54ce2968dce2920aa624a656"), "name" : "Latha" }
>
> db.persons.find()[0]._id.getTimestamp()
ISODate("2015-02-01T13:22:07Z")
>
> db.persons.find()[0]._id.getTimestamp().toLocaleDateString()
Sunday, February 01, 2015
>
>
```


Copyright © Capgemini 2017. All Rights Reserved

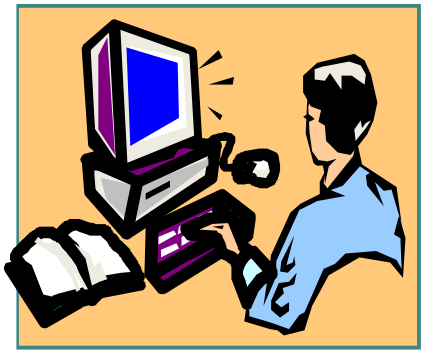
ObjectId

- **In MongoDB, documents stored in a collection require a unique _id field that acts as a primary key.**
- **MongoDB uses ObjectIds as the default value for the _id field, if the _id field is not specified by the user.**
- **ObjectId is a 12-byte BSON type constructed using:**
  - 4-byte value representing the seconds since the Unix time
  - 3-byte machine identifier
  - 2-byte process id
  - 3-byte counter, starting with a random value.
- **In the mongo shell, you can access the creation time of the ObjectId, using the getTimestamp() method. Sorting on an _id field that stores ObjectId values is roughly equivalent to sorting by creation time.**
- **To generate a new ObjectId, use the ObjectId() constructor with no argument**

Copyright © Capgemini 2017. All Rights Reserved

Demo

- **Inserting Documents**



Copyright © Capgemini 2017. All Rights Reserved

Updating Document(s)

- **db.collection.update() modifies an existing document or documents in a collection. The method can modify specific fields of an existing document or documents or replace an existing document entirely, depending on the update parameter.**
- **By default, the update() method updates a single document**

```
db.collection.update(
  <query>,
  <update>,
  {
   upsert: <boolean>,
   multi: <boolean>
  })
```

  - **upsert** : creates a new document when no document matches the query criteria.
  - **multi** : updates multiple documents that meet the query criteria.

Copyright © Capgemini 2017. All Rights Reserved

Updating Document(s)

- **$inc field update operator is used to increment and the $set field update operator is used to replace the value of the field.**

```
> db
SampleDB
> db.employees.find({_id:3861},{name:1,location:1})
{ "_id" : 3861, "name" : { "first" : "Veena", "last" : "Deshpande" }, "location"
 : "Pune" }
> db.employees.update({_id:3861},{ $set: { location:'Bangalore' } })
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>
> db.employees.find({_id:3861},{location:1})
{ "_id" : 3861, "location" : "Bangalore" }
>
>
> db.employees.update({_id:3861},{department:'Training'})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>
> db.employees.find({_id:3861})
{ "_id" : 3861, "department" : "Training" }
>
> db.employees.find({_id:1000})
>
> db.employees.update({_id:1000},{age:25})
WriteResult({ "nMatched" : 0, "nUpserted" : 0, "nModified" : 0 })
> db.employees.update({_id:1000},{age:25}, true)
WriteResult({ "nMatched" : 0, "nUpserted" : 1, "nModified" : 0, "_id" : 1000 })
>
> db.employees.find({_id:1000})
{ "_id" : 1000, "age" : 25 }
>
> db.employees.update({ _id:1000 },{ $inc : { age:3 } })
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>
> db.employees.find({_id:1000})
{ "_id" : 1000, "age" : 28 }
>
```


Copyright © Capgemini 2017. All Rights Reserved

Updating Document - save

- **db.collection.save() updates an existing document (when _id is present already) or inserts a new document (with new ObjectId)**

```
>
> var karthik = db.employees.findOne({_id:714709})
>
> karthik
{ "_id" : 714709, "department" : "Training" }
>
> karthik.location = 'Bangalore'
Bangalore
>
> karthik
{ "_id" : 714709, "department" : "Training", "location" : "Bangalore" }
>
> db.employees.findOne({_id:714709})
{ "_id" : 714709, "department" : "Training" }
>
> db.employees.save(karthik)
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>
> db.employees.findOne({_id:714709})
{ "_id" : 714709, "department" : "Training", "location" : "Bangalore" }
>
> var logith = {}
>
> logith.name = 'Logith Karthik'
Logith Karthik
>
> logith.location = 'Bangalore'
Bangalore
>
> logith
{ "name" : "Logith Karthik", "location" : "Bangalore" }
>
> db.employees.findOne({name:'Logith Karthik'})
null
> db.employees.save(logith)
WriteResult({ "nInserted" : 1 })
>
> db.employees.findOne({name:'Logith Karthik'})
{
        "_id" : ObjectId("54ce32b5dce2920aa624a657"),
        "name" : "Logith Karthik",
        "location" : "Bangalore"
}
```


Copyright © Capgemini 2017. All Rights Reserved

Updating Document - findAndModify

- **db.collection.findAndModify(<document>) modifies and returns a single document. By default, the returned document does not include the modifications made on the update. To return the document with the modifications made on the update, use the new option.**
- **The findAndModify() method has the following form**

```
db.collection.findAndModify({
   query: <document>,
   sort: <document>,
   remove: <boolean>,
   update: <document>,
   new: <boolean>,
   fields: <document>,
   upsert: <boolean>
});
```


Copyright © Capgemini 2017. All Rights Reserved

Updating Document - findAndModify

```
>
> db
SampleDB
> db.employees.findAndModify({
... query : { name : 'Logith Karthik' },
... update : { $set : { age : 5 } },
... new : true });
{
        "_id" : ObjectId("54ce32b5dce2920aa624a657"),
        "name" : "Logith Karthik",
        "location" : "Bangalore",
        "age" : 5
}
>
> db.employees.findAndModify({
... query : { name : 'Logith Karthik' },
... update : { $set : { gender : 'Male' } },
... new : false });
{
        "_id" : ObjectId("54ce32b5dce2920aa624a657"),
        "name" : "Logith Karthik",
        "location" : "Bangalore",
        "age" : 5
}
> db.employees.findOne({name : 'Logith Karthik'})
{
        "_id" : ObjectId("54ce32b5dce2920aa624a657"),
        "name" : "Logith Karthik",
        "location" : "Bangalore",
        "age" : 5,
        "gender" : "Male"
}
>
```


Copyright © Capgemini 2017. All Rights Reserved

Auto-Incrementing Sequence Field

- **In MongoDB, by default we cannot use an auto-increment pattern for the _id field, or any field, because it does not scale for databases with large numbers of documents. Typically the default value ObjectId is more ideal for the _id.**
- **But still we can create an auto-Incrementing sequence field by creating a collection on our own to maintain the sequence.**

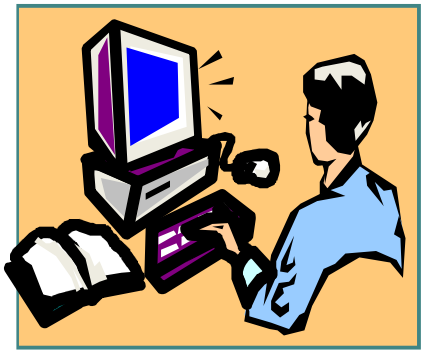
```
function generateSequence(name) {
  var ret = db.counters.findAndModify(
        {
          query: { _id: name },
          update: { $inc: { seq: 1 } },
          new: true,
          upsert:true
        }
  );
  return ret.seq;
}
```

- **By calling generateSequence('test') will create a document under counters collection and maintains the sequence for the same.**

Copyright © Capgemini 2017. All Rights Reserved

Demo

- **Creating Auto Increment Field**



Copyright © Capgemini 2017. All Rights Reserved

Array Update Operators

Name	Description
$	Acts as a placeholder to update the first element that matches the query condition in an update.
$addToSet	Adds elements to an array only if they do not already exist in the set.
$pop	Removes the first or last item of an array.
$pullAll	Removes all matching values from an array.
$pull	Removes all array elements that match a specified query.
$pushAll	*Deprecated.* Adds several items to an array.
$push	Adds an item to an array.

Copyright © Capgemini 2017. All Rights Reserved

Array Update Operators – Adding array items

```
> db
SampleDB
> db.persons.findOne({name:'Karthik'})
{ "_id" : ObjectId("54ce287fdce2920aa624a655"), "name" : "Karthik" }
>
> db.persons.update({name:'Karthik'},{$set:{hobbies:['Programming'] }},true)
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>
> db.persons.update({name:'Karthik'},{$push: { hobbies:'Music' } })
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>
> db.persons.update({name:'Karthik'},{$pushAll:{hobbies:['Cricket','Chess']}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>
> db.persons.findOne({name:'Karthik'})
{
        "_id" : ObjectId("54ce287fdce2920aa624a655"),
        "name" : "Karthik",
        "hobbies" : [
                "Programming",
                "Music",
                "Cricket",
                "Chess"
        ]
}
> db.persons.update({name:'Karthik'},{$addToSet:{hobbies:'Cricket'}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 0 })
>
> db.persons.update({name:'Karthik'},
... { $addToSet: { hobbies: { $each : ['Music','Chess','Tennis'] } } } )
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>
> db.persons.findOne({name:'Karthik'})
{
        "_id" : ObjectId("54ce287fdce2920aa624a655"),
        "name" : "Karthik",
        "hobbies" : [
                "Programming",
                "Music",
                "Cricket",
                "Chess",
                "Tennis"
        ]
}
```


Copyright © Capgemini 2017. All Rights Reserved

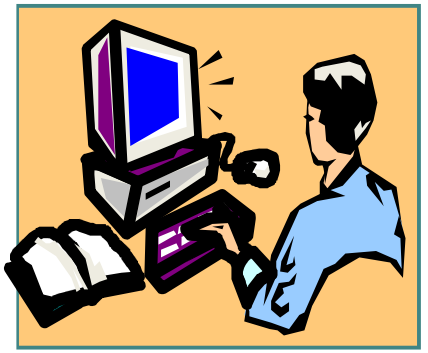
Array Update Operators – Removing array items

```
> db.persons.findOne({name:'Karthik'})
{
        "_id" : ObjectId("54ce287fdce2920aa624a655"),
        "name" : "Karthik",
        "hobbies" : [
                "Programming",
                "Music",
                "Cricket",
                "Chess",
                "Tennis"
        ]
}
> db.persons.update({name:'Karthik'},{$pull: { hobbies:'Music' } })
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>
> db.persons.update({name:'Karthik'},{$pullAll:{hobbies:['Cricket','Chess']}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>
> db.persons.findOne({name:'Karthik'})
{
        "_id" : ObjectId("54ce287fdce2920aa624a655"),
        "name" : "Karthik",
        "hobbies" : [
                "Programming",
                "Tennis"
        ]
}
> db.persons.update({name:'Karthik'},{ $pop: { hobbies : -1 }})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>
> db.persons.findOne({name:'Karthik'})
{
        "_id" : ObjectId("54ce287fdce2920aa624a655"),
        "name" : "Karthik",
        "hobbies" : [
                "Tennis"
        ]
}
```


Copyright © Capgemini 2017. All Rights Reserved

Demo

- **Working with Array items**



Copyright © Capgemini 2017. All Rights Reserved

Renaming and Deleting fields and documents

- **The $unset field update operator is used to delete a particular field.**
- **The $rename field update operator updates the name of a field**
- **db.collection.remove() is used to remove documents from a collection**

```
>
> db.persons.findOne({name:'Karthik'})
{
        "_id" : ObjectId("54ce287fdce2920aa624a655"),
        "name" : "Karthik",
        "hobbies" : [
                "Tennis"
        ]
}
> db.persons.update({name:'Karthik'},{ $rename : {'hobbies' : 'intrests'}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>
> db.persons.findOne({name:'Karthik'})
{
        "_id" : ObjectId("54ce287fdce2920aa624a655"),
        "name" : "Karthik",
        "intrests" : [
                "Tennis"
        ]
}
> db.persons.update({name:'Karthik'},{ $unset : {'intrests' : ''}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>
> db.persons.findOne({name:'Karthik'})
{ "_id" : ObjectId("54ce287fdce2920aa624a655"), "name" : "Karthik" }
>
> db.persons.find({name:'Karthik'}).count()
1
> db.persons.remove({name:'Karthik'})
WriteResult({ "nRemoved" : 1 })
>
> db.persons.find({name:'Karthik'}).count()
0
```


Copyright © Capgemini 2017. All Rights Reserved

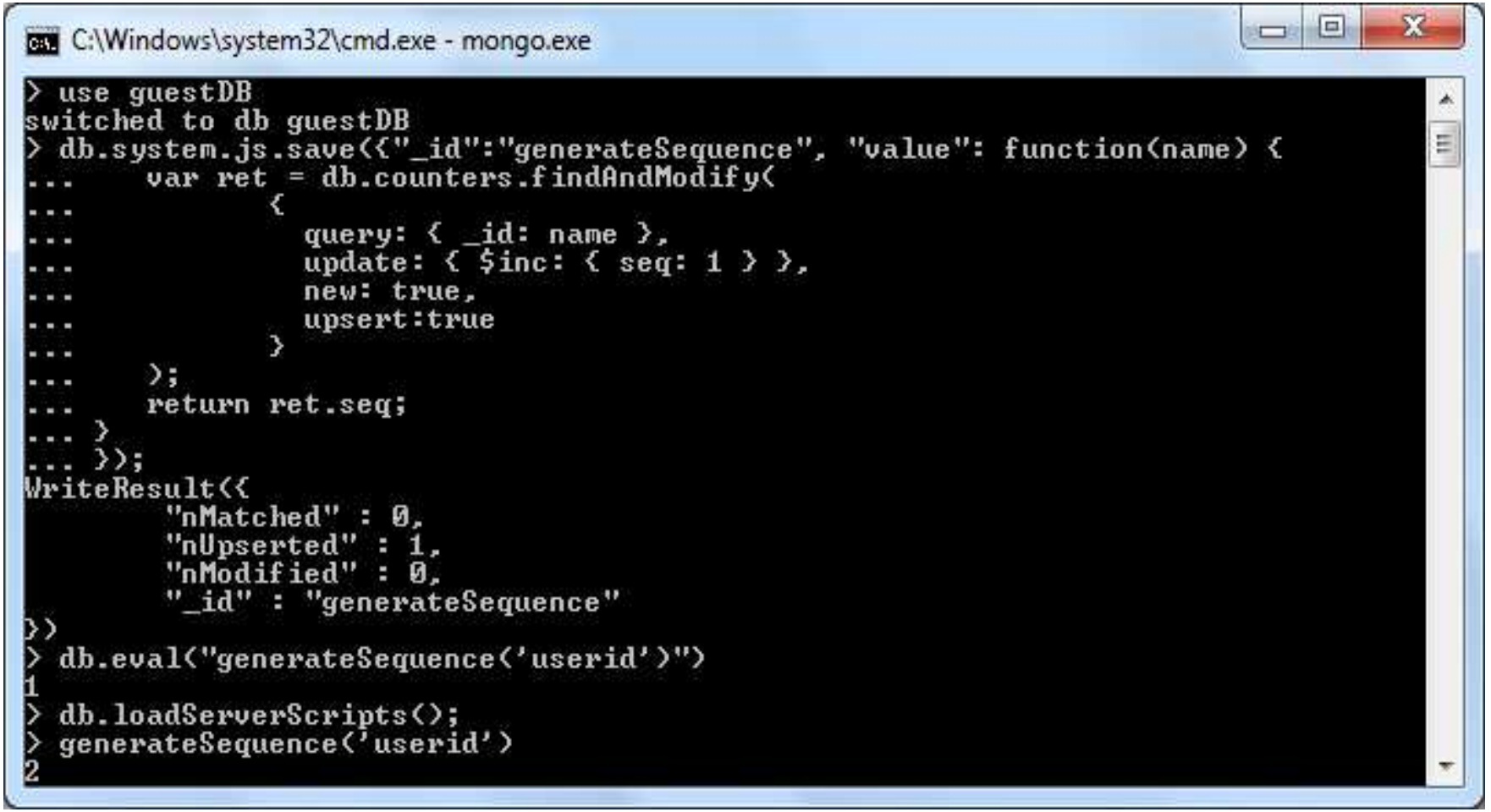
Aggregation

- **db.collection.aggregate() calculates aggregate values for the data in a collection.**

```
>
> db.employees.aggregate(
... {
...     $group: { _id : "$location", employeeCount : { $sum : 1 }}
... },
... {
...     $sort: { employeeCount : -1 }
... }
... );
{ "_id" : "Pune", "employeeCount" : 24 }
{ "_id" : "Mumbai", "employeeCount" : 23 }
{ "_id" : "Bangalore", "employeeCount" : 21 }
{ "_id" : "Chennai", "employeeCount" : 7 }
>



>
> db.employees.aggregate(
... {
...     $match : { location : "Bangalore" }
... },
... {
...     $group : { _id : "$location", employeeCount : { $sum : 1 } }
... }
... );
{ "_id" : "Bangalore", "employeeCount" : 21 }
>
```


Copyright © Capgemini 2017. All Rights Reserved

Stored JavaScript

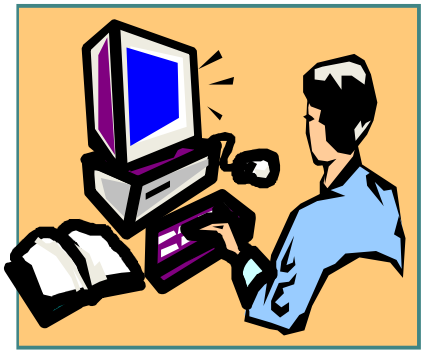


- **Do not store application logic in the database. There are performance limitations to running JavaScript inside of MongoDB.**

Copyright © Capgemini 2017. All Rights Reserved

Demo

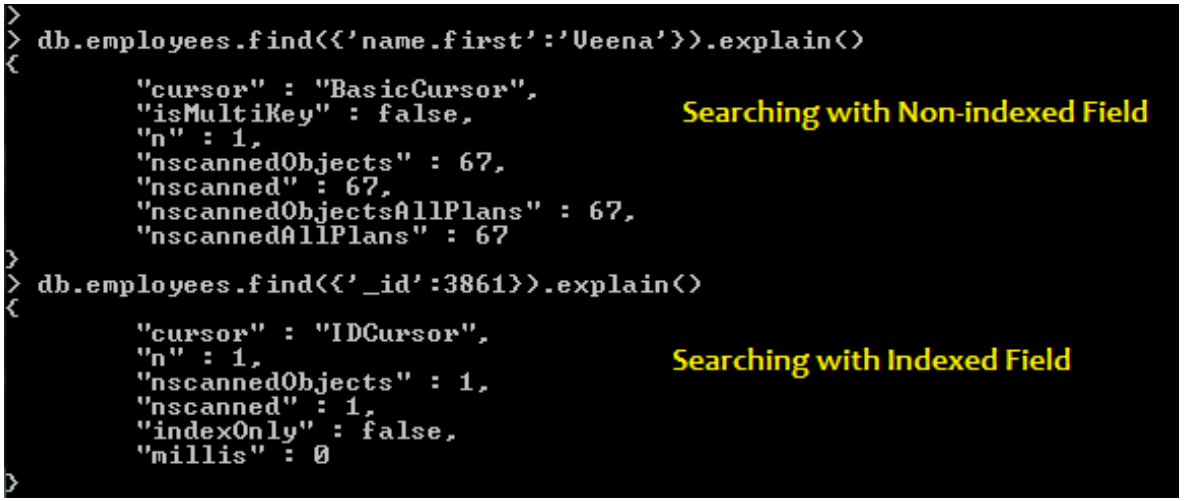
- **Stored JavaScript**



Copyright © Capgemini 2017. All Rights Reserved

Indexing

- **Indexing is an important part of database management.**
- **In MongoDB, if we do a query with non-indexed field, it uses "BasicCursor". BasicCursor indicates a full collection scan where as "BtreeCursor" indicates that the query used is an index field.**
- **explain() method returns a document that describes the process used to return the query results. MongoDB stores its indexes in system.indexes collection. We can view the indexes of DB using *db.system.indexes.find()***



Copyright © Capgemini 2017. All Rights Reserved

Indexing

- **To create an index on the specified field if the index does not already exist.**
  - db.<collection>.ensureIndex({<field>:1(asc)/-1(desc)})
- **To create a unique index so that the collection will not accept insertion of documents where the index key or keys match an existing value in the index.**
  - db.<collection>.ensureIndex({<field>:1(asc)/-1(desc)},{unique: true})
- **To creates a unique index on a field that may have duplicates**
  - db.<collection>.ensureIndex({<field>:1(asc)/-1(desc)},{unique: true, dropDups:true})
- **To create an Index on a Multiple Fields**
  - db.collection.ensureIndex({<field1>:1(asc)/-1(desc), <field2>:1(asc)/-1(desc) })
- **To create index which only references the documents with specified field**
  - db.employees.ensureIndex({nonexistfield:1},{sparse:true});
- **To drop index**
  - db.employees.dropIndex('<indexname>')

Copyright © Capgemini 2017. All Rights Reserved

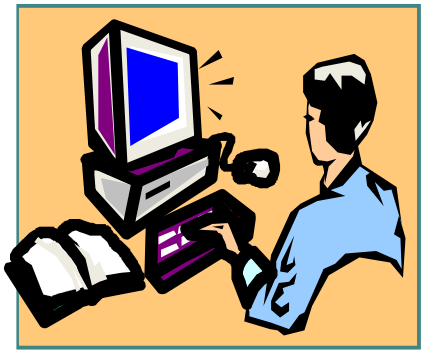
Indexing

```
>
> db.employees.find({'name.first':'Veena'})
{ "_id" : 3861, "name" : { "first" : "Veena", "last" : "Deshpande" }, "doj" : "1
998-05-07T00:00:00.000Z", "location" : "Pune", "isActive" : true, "email" : "vee
na.deshpande@igate.com", "qualifications" : [ "M.Sc(CSE)", "M.E(Electronics)" ]
}
> db.employees.ensureIndex({'name.first':1},{unique: true, dropDups:true})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
> db.employees.find({'name.first':'Veena'}).explain()
{
        "cursor" : "BtreeCursor name.first_1",
        "isMultiKey" : false,
        "n" : 1,
        "nscannedObjects" : 1,
        "nscanned" : 1,
        "nscannedObjectsAllPlans" : 1,
        "server" : "BLRWFL2913:27017",
        "filterSet" : false
}
> db.system.indexes.find()
{ "v" : 1, "key" : { "_id" : 1 }, "name" : "_id_", "ns" : "SampleDB.employees" }
{ "v" : 1, "key" : { "_id" : 1 }, "name" : "_id_", "ns" : "SampleDB.locations" }
{ "v" : 1, "key" : { "_id" : 1 }, "name" : "_id_", "ns" : "SampleDB.persons" }
{ "v" : 1, "key" : { "_id" : 1 }, "name" : "_id_", "ns" : "SampleDB.guests" }
{ "v" : 1, "unique" : true, "key" : { "name.first" : 1 }, "name" : "name.first_1
", "ns" : "SampleDB.employees", "dropDups" : true }
> db.employees.dropIndex('name.first_1')
{ "nIndexesWas" : 2, "ok" : 1 }
```


Copyright © Capgemini 2017. All Rights Reserved

Demo

- **Working with Indexing**



Copyright © Capgemini 2017. All Rights Reserved

Introduction to Mongoose Library

- There are multiple ways to connect to MongoDB from Node.js.
- The native driver's npm package is simply mongodb. It provides a fundamental connectivity and data manipulation APIs
- The most popular ways to connect MongoDB is by using the Node.js native driver and the Mongoose.js ODM .
- Mongoose is an object data modeling (ODM) library that provides a modeling environment for data which enforces the structure as needed. It is the officially supported ODM for Node.js.
- Using Mongoose we can define schemas for MongoDB collections, which are then enforced at the ODM layer.
- Mongoose makes it even easier to use MongoDB with Node.js.

Copyright © Capgemini 2017. All Rights Reserved

Connecting MongoDB using Mongoose Library

- **Connecting to a MongoDB instance with Mongoose is straight-forward, requiring only the resource URL of the database.**
- **mongoose.connect() is used to connect the MongoDB**
- **To establish connection in app.js**

```
var mongoose = require('mongoose');
mongoose.connect(<uri>, function (err, res) {
        if (err) {
           console.log ('ERROR connecting to MongoDB : ' + err);
        }
        else {
         console.log ('Connected to: MongoDB');
        }
});
```


Copyright © Capgemini 2017. All Rights Reserved

Object Modeling

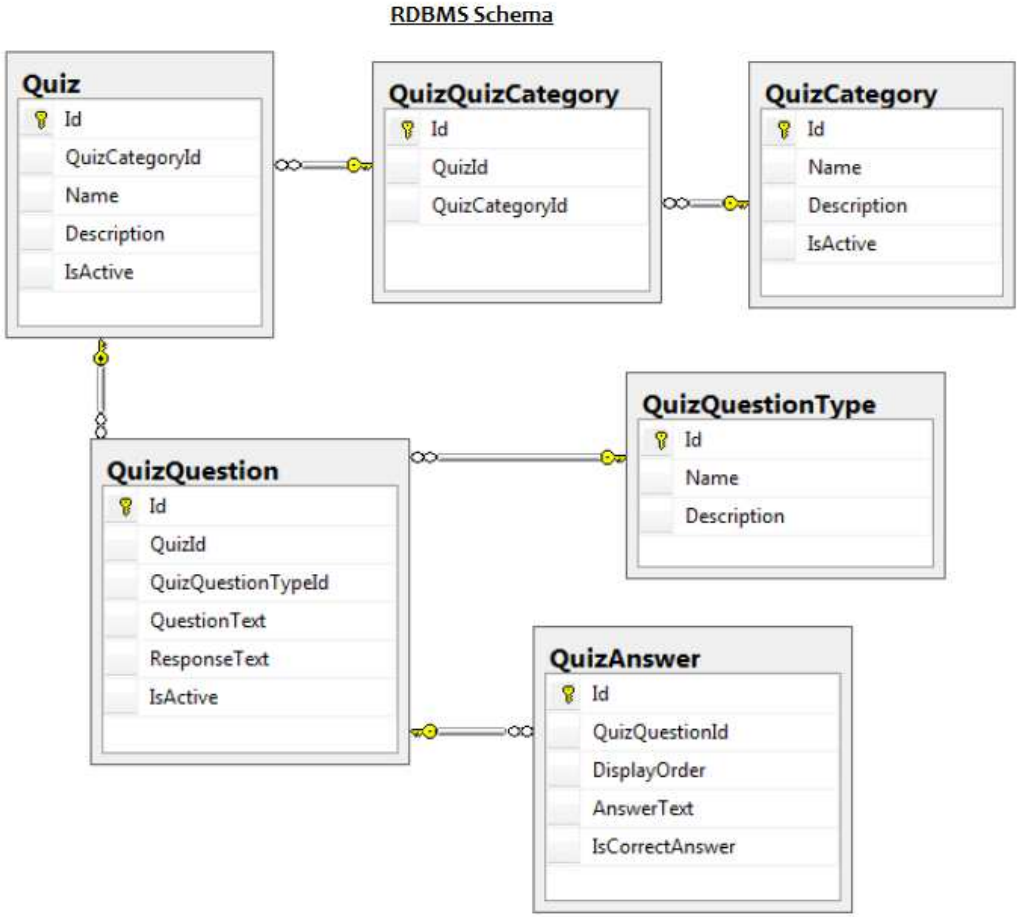
- **MongoDB has a flexible schema that allows for variability between different documents in the same collection. But validations were not enforced.**
- **Using Mongoose we can perform this when new objects are getting created.**
- **Models are defined by passing a Schema instance to mongoose.model()**

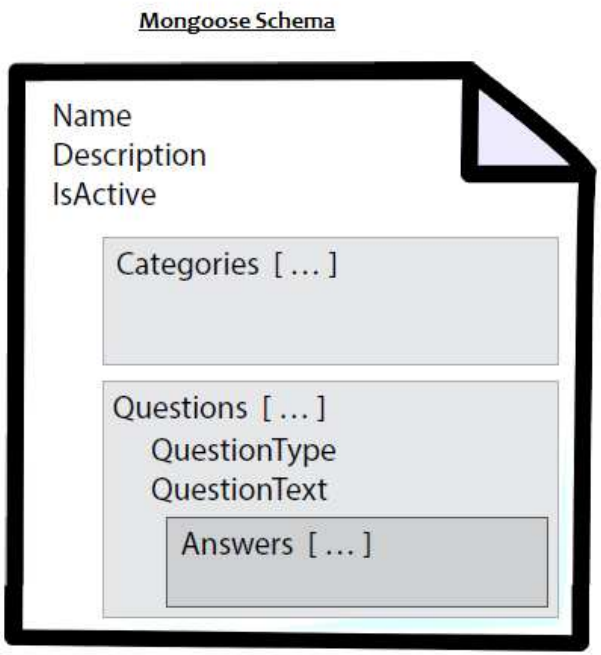
```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;
var userSchema = new mongoose.Schema({
        name: {type: String, trim: true},
        age: { type: Number, min: 0 },
        company: { type: String, default: 'IGATE' }
});
```

- **To instantiate a new user, create a Mongoose model from the schema in the users collection(MongoDB) so that we can populated user's details.**
  - var User = mongoose.model('users', userSchema);

Copyright © Capgemini 2017. All Rights Reserved

RDBMS Schema vs Mongoose Schema





Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING
Copyright © Capgemini 2017. All Rights Reserved

Simple Schema

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var customerSchema = new Schema({
        name:       String,
        address:    String,
        city:       String,
        state:      String,
        country:    String,
        zipCode:    Number,
        createdOn:  Date,
        isActive:   Boolean
});


var simpleSchema = new Schema({ fieldName: SchemaType });
```


Copyright © Capgemini 2017. All Rights Reserved

Complex Schema

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

// child address schema...
var addressSchema = new Schema({
    type: String,
    street: String,
    city: String,
    state: String,
    country: String,
    postalCode: Number
});

// parent customer schema...
var customerSchema = new Schema({
    name: {
        first: String,
        last: String
    },
    address: [ addressSchema ],
    createdOn: { type: Date, default: Date.now },
    isActive: { type: Boolean, default: true},
});
```


Copyright © Capgemini 2017. All Rights Reserved

Validations in Model

- **Validation is defined in the Schema, it's an internal piece of middleware**
- **Validation occurs when a document attempts to be saved, after defaults have been applied.**
- **The path and value that triggered the error can be accessed in the ValidationError object.**
- **For Strings *enum* and for Numbers *min* and *max* are built in validators.**
  - var Person = new Schema({ gender : { type: String, enum: ['M', 'F', 'T'] } })
  - var Person = new Schema({ age : { type: Number, min: 1, max : 100 } })
- **We can also validate a value against a regular expression**
  - var schema = new Schema({ name: { type: String, validate: /[a-zA-Z]+/ } });
- **Validation is asynchronously recursive. i.e. when Model#save is called, embedded documents validation is executed. If an error occurs Model#save callback receives it.**

Copyright © Capgemini 2017. All Rights Reserved

Finding documents

- **Documents can be retrieved through find, findOne and findById. These methods are executed on Models.**
- **Model.find**
  - Model.find(query, fields, options, callback) // fields and options can be omitted
  - Model.find({ '_id': 5 }, 'name age', function (err, docs) { // docs is an array });
- **Model.findOne : receives a single document as second parameter**
  - Model.findOne({ age: 5}, function (err, doc){ // doc is a Document });
- **Model.findById : Same as findOne, but receives a value to search a document by their _id key**
  - Model.findById(obj._id, function (err, doc){ // doc is a Document });

Copyright © Capgemini 2017. All Rights Reserved

Query API

- **A Query is what is returned when calling any Model methods.**
- **Query objects provide a chaining api to specify search terms, cursor options, hints and other behavior.**

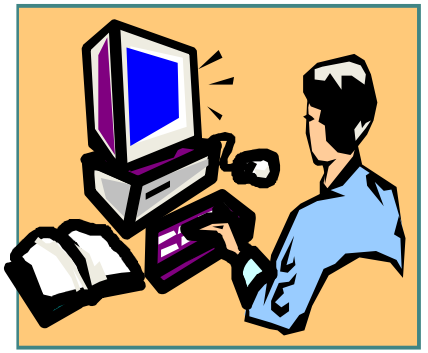
```
var query = Model.find({});
query.where('field', 5);
query.limit(5);
query.skip(100);

query.exec(function (err, docs) {
  // called when the `query.complete` or `query.error` are called internally
});
```

- **Each of these methods returns a Query. If you don't pass a callback to these methods, the Query can be continued to be modified (such as adding options, fields, etc), before it's executed.**

Copyright © Capgemini 2017. All Rights Reserved

Demo

- **Performing CRUD with MongoDB using Mongoose Library**



Copyright © Capgemini 2017. All Rights Reserved

Introduction

- **HTTP is a stateless protocol, server cannot differentiate between different connections of different users.**
- **One of the most common requirements for any web application are sessions.**
- **Using sessions to keep track of users as they browse through the site, becomes an important aspect for web application.**
- **Typically, the process of managing the state of a web-based client is through the use of session IDs. Session IDs are used by the application to uniquely identify a client browser, while background (server-side) processes are used to associate the session ID with a level of access.**
- **There are three methods available to allocate and receive session ID information.**
  - URL Based Session ID's
  - Hidden Post Fields
  - Cookies

Copyright © Capgemini 2017. All Rights Reserved

Session Handling in Node.js

- **Express with Node application makes it very easy to get sessions up and running**
- **Since sessions use cookies to keep track users we need both the cookie-parser and express-session modules.**
- **cookieParser need to be used before the session. This order is required for sessions to work. Create package.json using *npm init* command as given below**

```
{
  "name": "SessionDemo",
  "version": "1.0.0",
  "description": "Session Handling Demo",
  "main": "app.js",
  "dependencies": {
    "cookie-parser": "1.3.3",
    "express": "4.11.2",
    "express-session": "1.10.2",
    "nodemon": "1.2.1"
  },
   "scripts": {
    "start": "nodemon app.js"
  },
  "author": "Karthik.M < karthik.muthukrishnan@igate.com >",
  "license": "ISC"
}
```


Copyright © Capgemini 2017. All Rights Reserved

Session Handling in Node.js

```
var express = require('express');
var cookieParser = require('cookie-parser');
var session = require('express-session');

var app = express();

app.use(cookieParser());
app.use(session({secret:'IGATEKarthik', resave: true,  saveUninitialized: true}));

app.get('/', function(req, res) {
    console.log('home : '+req.sessionID);
    if(req.session.lastPage) {
        res.write('Last page was: ' + req.session.lastPage + '.\n');
    }
    req.session.lastPage = 'Home';
    res.end('Home Page');
});

app.get('/about', function(req, res) {
    console.log('about : '+req.sessionID);
    if(req.session.lastPage) {
        res.write('Last page was: ' + req.session.lastPage + '.\n');
    }
    req.session.lastPage = 'AboutUs';
    res.end('About us');
});

app.listen(process.env.PORT || 3000);
console.log('Server listening on the port 3000');
```


Copyright © Capgemini 2017. All Rights Reserved

Storing session information in MongoDB

- **Anytime we restart our app, all of the user sessions are lost, which gives a problem with our updating process.**
- **There are several ways to solve this problem, but one of the best way is to use an external store for the session data.**
- **More specifically, we can use MongoDB to store your session data. This way the session data is completely separate from our running application.**
- **To set up MongoDB with Express sessions, we need an extra module called *connect-mongo*.**

```
{
  "name": "SessionDemo",
  "version": "1.0.0",
  "dependencies": {
    "connect-mongo": "0.5.3",
    "cookie-parser": "1.3.3",
    "express": "4.11.2",
    "express-session": "1.10.2",
    "nodemon": "1.2.1"
  }
}
```


Copyright © Capgemini 2017. All Rights Reserved

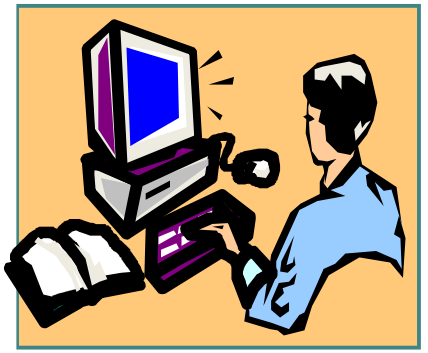
Storing session information in MongoDB

```
var express = require('express');
var cookieParser = require('cookie-parser');
var session = require('express-session');
var MongoStore = require('connect-mongo')(session);

var app = express();

app.use(cookieParser());
app.use(session({
    store : new MongoStore({"db": "SessionDB","host": "localhost","port": "27017","collection": "sessions"}),
    cookie: { maxAge: 50 * 1000 }, /* 5 min */
    secret:'IGATEKarthik',
    resave: true,
    saveUninitialized: true
}));

app.get('/', function(req, res) {
    console.log('home : '+req.sessionID);
    if(req.session.lastPage) {
        res.write('Last page was: ' + req.session.lastPage + '.\n');
    }
    req.session.lastPage = 'Home';
    res.end('Home Page');
});

app.get('/logout', function(req, res) {
    req.session.destroy();
    res.redirect("/");
});

app.listen(process.env.PORT || 3000);
console.log('Server listening on the port 3000');
```


Copyright © Capgemini 2017. All Rights Reserved

Demo

- **Session Handling Demo**



Copyright © Capgemini 2017. All Rights Reserved

Introduction

- **Cryptography is a method of storing and transmitting data in a particular form so that only those for whom it is intended can read and process it.**
- **As a best practice, any private information (like password) stored on the server should be encrypted.**
- **Hashing is the greatest way for protecting passwords and considered to be pretty safe for ensuring the integrity of data or password.**
- **Bcrypt hashing algorithm is a good option for secured password hashing. It is a one way hash i.e. It cannot be decrypted.**
- ***bcryptjs* is the popular node module for hashing private information.**

Copyright © Capgemini 2017. All Rights Reserved

Hashing data using bcryptjs

```
var bcrypt = require('bcryptjs');

/*Synchronous Approach */
var hash = bcrypt.hashSync("Karthik",bcrypt.genSaltSync(10));
bcrypt.compareSync("Karthik", hash);  // returns true
bcrypt.compareSync("karthik", hash)); // returns false

 /*Asynchronous Approach*/
 bcrypt.hash("Karthik", bcrypt.genSaltSync(10), function(err, hash) {
        bcrypt.compare("Karthik", hash, function(err, res) {
              //res returns true
        });
 });
```

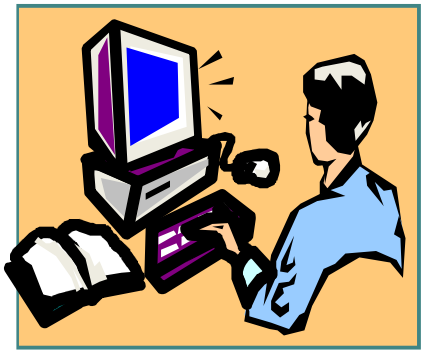
hashSync (data, salt) : where data [required] is the data to be encrypted and salt [required] is the salt to be used in encryption.

genSaltSync (rounds) : where rounds - [OPTIONAL] - the number of rounds to process the data for. (default - 10)

Copyright © Capgemini 2017. All Rights Reserved

Demo

- **Hashing**



Copyright © Capgemini 2017. All Rights Reserved

Unit Testing, Logging & Debugging

Copyright © Capgemini 2017. All Rights Reserved

Introduction

- **Unit Testing is nothing but breaking down the logic of application into small chunks or 'units' and verifying that each unit works as we expect.**
- **Unit Testing the code provides the following benefits :**
  - Reduce code-to-bug turn around cycle
  - Isolate code and demonstrate that the pieces function correctly
  - Provides contract that the code needs to satisfy in order to pass
  - Improves interface design
- **Unit testing can be done in 2 ways**
  - Test-Driven Development
  - Test-After Development

Copyright © Capgemini 2017. All Rights Reserved

Test-Driven Development

- **In Test Driven Development (TDD) automated unit tests are written before the code is actually written. Running these tests give you fast confirmation of whether your code behaves as it should.**
- **TDD can be summarized as a set of the following actions:**
  1. **Writing a test :** In order to write the test, the programmer must fully comprehend the requirements. At first, the test will fail because it is written prior to the feature
  2. **Run all of the tests and make sure that the newest test doesn't pass :** This insures that the test suite is in order and that the new test is not passing by accident, making it irrelevant.
  3. **Write the minimal code that will make the test pass :** The code written at this stage will not be 100% final, it needs to be improved at later stages. No need to write the perfect code at this stage, just write code that will pass the test.

Copyright © Capgemini 2017. All Rights Reserved

Test-Driven Development

4. **Make sure that all of the previous tests still pass:** If all tests succeeds, developer can be sure that the code meets all of the test specifications and requirements and move on to the next stage.
5. **Refactor code :** In this stage code needs to be cleaned up and improved. By running the test cases again, the programmer can be sure that the refactoring / restructuring has not damaged the code in any way.
6. **Repeat the cycle with a new test :** Now the cycle is repeated with another new test.

- **Using TDD approach, we can catch the bugs in the earl stage of development**
- **It works best with reusable code**

Copyright © Capgemini 2017. All Rights Reserved

Test-After Development

- **In Test-After Development, we need to write application code first and then write unit test which tests the application code.**
- **TAD approach helps us to find existing bugs**
- **It drives the design**
- **It is a part of the coding process**
- **Enables Continual progress and refactoring**
- **It defines how the application is supposed to behave.**
- **Ensures sustainable code**

Copyright © Capgemini 2017. All Rights Reserved

Behavior-Driven Development

- **Test-Driven Development (TDD) focus on testing where as Behavior-Driven Development (BDD) mainly concentrates on specification.**
- **In BDD we write test specifications, not the tests which we used to do in TDD i.e. no need focus anymore on the structure of the source code, focus lies on the behavior of the source code**
- **In BDD we need to follow just one rule. Test specifications which we write should follow *Given-When-Then* steps**
- **It is very easy to write / read and understand the test specifications, because BDD follows a common vocabulary instead of a test-centered vocabulary (like test suite, test case, test …)**
- **BDD is a specialized version of TDD that specifies what needs to be unit-tested from the perspective of business requirements.**

Copyright © Capgemini 2017. All Rights Reserved

Behavior-Driven Development

Story: As a registered user, need to login in to the system to access home page. If username or password are invalid, they will stay in login page and the system will shows an error message.

Scenario: Valid Login
Given The user is in login page
And the user enters a valid username
And the user enters a valid password
When the user logs in
Then the user is redirected to Home page

Scenario: Enter an Invalid password
Given The user is in login page
And the user enters a valid username
And the user enters a invalid password
When the user logs in
Then the user is redirected to the Login Page
And the System shows the following message : "Invalid username or password"

Copyright © Capgemini 2017. All Rights Reserved

Unit Testing in Node

- In Node there are numerous tools / testing frameworks are available to achieve the objective of having well tested quality code.
- Mocha, Chai and SuperTest integrates very well together in a natural fashion
- Mocha is a testing framework which follows Behavior-driven Development (BDD) interface makes our tests readable and make us very clear what is being tested.
- Mocha allows to use any assertion library like (chai, should.js, expect.js)
- Chai is a BDD / TDD assertion library for node and the browser that can be delightfully paired with any testing framework like Mocha.
- SuperTest is a HTTP testing library which has a bunch of convenience methods for doing assertions on headers, statuses and response bodies. If the server is not already listening for connections then it is bound to an ephemeral port, so there is no need to keep track of ports.

Copyright © Capgemini 2017. All Rights Reserved

Mocha

- **Mocha is a mature and powerful testing framework for Node.js.**
- **It is more robust and widely used. NodeUnit, Jasmine and Vows are the other alternatives.**
- **Mocha supports both BDD interface's like (describe, it, before) and traditional TDD interfaces (suite, test, setup).**
- **We need explicitly tell Mocha to use the TDD interface instead of the default BDD by specifying *mocha -u tdd <testfile>***
  - **describe** *analogous to* **suite**
  - **it** *analogous to* **test**
  - **before** *analogous to* **setup**
  - **after** *analogous to* **teardown**
  - **beforeEach** *analogus to* **suiteSetup**
  - **afterEach** *analogus to* **suiteTeardown**
- **The mocha package can be installed via**
  - *npm install mocha --save-dev*

Copyright © Capgemini 2017. All Rights Reserved

Chai

- **Chai is a BDD / TDD assertion library for node and the browser that can be delightfully paired with any javascript testing framework.**
- **Chai's chain-capable BDD styles (Expect / Should) API's provide an expressive language & readable style, while the TDD assert style (Assert) API provides a more classical feel.**
- **The chai package is available on npm and it can be installed via**
  - *npm install chai --save-dev*

Should

```
chai.should();

foo.should.be.a('string');
foo.should.equal('bar');
foo.should.have.length(3);
tea.should.have.property('flavors')
  .with.length(3);
```

Expect

```
var expect = chai.expect;

expect(foo).to.be.a('string');
expect(foo).to.equal('bar');
expect(foo).to.have.length(3);
expect(tea).to.have.property('flavors')
  .with.length(3);
```

Assert

```
var assert = chai.assert;

assert.typeOf(foo, 'string');
assert.equal(foo, 'bar');
assert.lengthOf(foo, 3)
assert.property(tea, 'flavors');
assert.lengthOf(tea.flavors, 3);
```


Copyright © Capgemini 2017. All Rights Reserved

SuperTest

- **SuperTest is another assertion library but it differs from other assertion libraries by providing developers with an abstraction layer that makes HTTP assertions.**
- **Instead of testing objects, SuperTest helps us to create assertion expressions that test HTTP endpoints.**
- **SuperTest will bound to an ephemeral port if the server is not already listening for connections, so that there is no need to keep track of ports.**
- **The SuperTest package is available on npm and it can be installed via**
  - *npm install supertest --save-dev*
- **Mocha, Chai and SuperTest is a prowerful combination to test both your models and your controllers. All those three packages will be available under property named devDepencies in the package.json file.**

Copyright © Capgemini 2017. All Rights Reserved

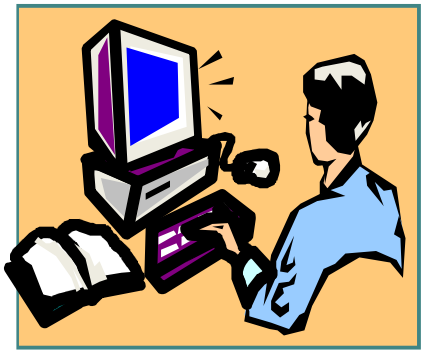
Code Coverage using Istanbul

- **Code coverage tools analyze an application's source code and test suite and then identify code that's missing tests.**
- **Developing maintainable apps requires a good test coverage. Code coverage provides a way to measure the quality of your application.**
- **Istanbul is a JavaScript code coverage tool that provides instrumentation and reporting. It works well with Mocha, Chai & SuperTest.**
- **Istanbul generates html reporter for the code coverage**
- **Istanbul package is available on npm and it can be installed via**
  - npm install istanbul --save-dev

Copyright © Capgemini 2017. All Rights Reserved

Demo

- **Unit testing using Mocha, Chai, Supertest & Istanbul**



Copyright © Capgemini 2017. All Rights Reserved

Logging using morgan

- **Morgan is used for logging request details.**
- **It is used as HTTP request logger middleware(third party) for node.js**
- **By default it shows the logs on the console, it also can be customized to write the logs in a files too.**

```
var logger = require('morgan');
var fs = require('fs');
/* Writing request Logs into file named access.log in append mode */
app.use(logger('dev', {
    stream: fs.createWriteStream('/access.log', {flags: 'a'})
}));
/* Output stream for writing log lines in console to process.stdout. */
app.use(logger('dev'));
```

- **There are various pre-defined formats available for logging like combined, common, dev, short & tiny.**
  - **'dev'** is a combination of :method :url :status :response-time ms - :res[content-length]

Copyright © Capgemini 2017. All Rights Reserved

Debugging with node-inspector

- **Node Inspector is a debugger interface for Node.js applications.**
- **Node Inspector works almost exactly as the Chrome Developer Tools**
- **Node Inspector supports almost all of the debugging features of DevTools, including:**
  - Navigate source files
  - Set breakpoints (and specify trigger conditions)
  - Step over, step in, step out, resume (continue)
  - Inspect scopes, variables, object properties
  - Hover mouse over an expression in source code to display its value in a tooltip
  - Edit variables and object properties
  - Continue to location
  - Break on exceptions
  - Disable/enable all breakpoints

Copyright © Capgemini 2017. All Rights Reserved

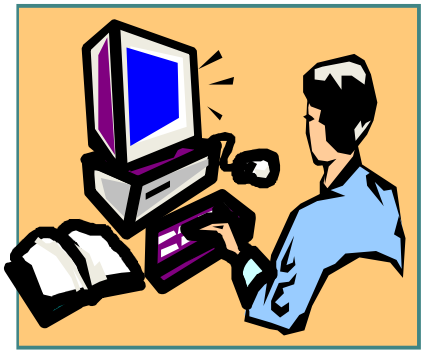
Steps to debug

- **Install node-inspector as a global module**
  - npm install –g node-inspector
- **There are two steps needed to get you up and debugging**
- **Step – 1: Start the Node Inspector Server**
  - ***node-inspector***
- **Step – 2: Enable debug mode in your Node process**
  - To start Node with a debug flag ***node --debug program.js***
  - To pause script on the first line ***node –debug-brk program.js***

Copyright © Capgemini 2017. All Rights Reserved

Demo

- **Logging and Debugging**



Copyright © Capgemini 2017. All Rights Reserved

Node.JS Best practices

Copyright © Capgemini 2017. All Rights Reserved

Lesson Objectives

- At the end of this module you will be able to:
  - Understand the best practices need to be implemented while developing node applications.

Copyright © Capgemini 2017. All Rights Reserved

Using development tools

- Automating restarts
  - Node requires a restart when changes made to the source code by pressing [CTRL+C] to stop the application and then restart it by pressing the [UP] arrow and [Enter]
  - Above mentioned process can be automated and make the development process easier by using tools like nodemon, node-supervisor and forever. These modules watch for file changes and restart the server automatically.
- Automatic browser refresh
  - Refreshing the browser when the static code changes slows down the app development while creating web applications.
  - Instead of manually triggering the page refresh in the browser, it can be automated using tools such as livereload.

Copyright © Capgemini 2017. All Rights Reserved

Avoid blocking the event loop

- Node.js runs on a single thread, everything that will block the event loop will block everything.
  - For instance a web server with a thousand connected clients and if it happens to block the event loop, then every client need to wait.
- Monitor the event loop delay to detect anomalies using StrongOps or by using open-source modules such as blocked.

Copyright © Capgemini 2017. All Rights Reserved

Avoid creating big monolithic applications

- Developers should stop including everything into a single file, instead of breaking things into their own modules and publishing to NPM.
- Node encourages to write tiny modules and publish them to NPM.
- Strive for modularity in Node applications and keeping things as simple as possible.

Copyright © Capgemini 2017. All Rights Reserved

Avoiding poor logging

- Developers should use something better than console.log to implement logging in Node application.
- Production-ready logging module such as bunyan performs the following logging features
  - No need to use util.inspect for large, complex objects
  - Built-in serializers for things like errors, request and response objects
  - Support multiple sources for controlling where the logs go
  - Automatic inclusion of hostname, process id, application name
  - Supports multiple levels of logging (debug, info, error, fatal etc)
  - Advanced functionality such as log file rotation, etc.
- bunyan gives lot of useful information about the current process, it also pipe the logs into a stream (or multiple streams)

Copyright © Capgemini 2017. All Rights Reserved

Testing Node Applications

- Node applications should not considered as completed without writing tests.
- There are many tools available to write test cases for node applications.
  - Testing frameworks: mocha, jasmine, qunit and many other
  - Assertion modules: chai, should.js
  - Mocking Libraries : Modules for mocks, spies, stubs or fake timers such as sinon
  - Code coverage tools: istanbul, blanket

Copyright © Capgemini 2017. All Rights Reserved

Using static analysis tools

- Instead of spotting problems in production it's better to catch them right away in development by using static analysis tools.
- Static analysis tools like ESLint solves the following issues.
  - Possible errors, for example: disallow assignment in conditional expressions, disallow the use of debugger.
  - Enforcing best practices, for example: disallow declaring the same variable more then once, disallow use of arguments.callee.
  - Finding potential security issues, such as the use of eval() or unsafe regular expressions.
  - Detecting possible performance problems.
  - Enforcing a consistent style guide.
- JSLint or JSHint are some of the other static analysis tool

Copyright © Capgemini 2017. All Rights Reserved

Debugging Node.js Applications

- One of the challenge while creating Node.js apps is to debug code.
- ***node-inspector*** has been specifically written for debugging Node.js applications and is popular in the web community for debugging Node applications. It performs the following operations
  - Navigate in your source files
  - Set breakpoints (and specify trigger conditions)
  - Step over, step in, step out, resume (continue)
  - Inspect scopes, variables, object properties
  - Hover your mouse over an expression in your source to display its value in a tooltip
  - Edit variables and object properties
  - Continue to location
  - Break on exceptions and disable/enable all breakpoints

Copyright © Capgemini 2017. All Rights Reserved

Best IDEs for Node.js Development

- Some of the Best IDEs for Node.js Development are :
  - Intellij IDEA
  - Cloud 9
  - Komodo
  - WebStorm
  - Sublime Text
  - Atom
  - Brackets
  - Eclipse
  - Nide
  - Koding
  - WebMatrix
  - Visual Studio

Copyright © Capgemini 2017. All Rights Reserved

Node.js Error handling

- Errors can be divided into two main parts: operational errors and programmer errors.
- Operational errors : Operational errors can happen in well-written applications as well, because they are not bugs, but problems with the system / a remote service, like:
  - Request timeout
  - System is out of memory
  - Failed to connect to a remote service
- Programmer errors : Programmer errors are bugs which can be avoided
  - Calling an async function without a callback
  - Reading property of undefined

Copyright © Capgemini 2017. All Rights Reserved

Handling operational errors

- Try to solve the error : if a file is missing, it need to be created first
- Retry the operation, when dealing with network communication
- Notify with appropriate error message while handling invalid user inputs
- Crash the process, when the error condition is unlikely to change on its own, like the application cannot read its configuration file

Copyright © Capgemini 2017. All Rights Reserved

Handling operational and programmer errors

- While Handling Operational errors :
  - Try to solve the error : if a file is missing, it need to be created first
  - Retry the operation, when dealing with network communication
  - Notify with appropriate error message while handling invalid user inputs
  - Crash the process, when the error condition is unlikely to change on its own, like the application cannot read its configuration file
- While Handling Programmer errors :
  - Crash immediately - as these errors are bugs. A process control system should restart the application

Copyright © Capgemini 2017. All Rights Reserved

Node.js Event handling techniques

- Node.js is a single threaded application but it support concurrency via concept of event and callbacks.
- As every API of Node.js are asynchronous and being a single thread, it uses async function calls to maintain the concurrency.
- Callback is an asynchronous equivalent for a function. A callback function is called at the completion of a given task. All APIs of Node are written is such a way that they supports callbacks.
- In Node.js an event can be described simply as a string with a corresponding callback and it can be emitted.

Copyright © Capgemini 2017. All Rights Reserved

Handling errors for callback

- Modules should expose an error-first callback interface

```
module.exports = function (data, callback) {
  // do some stuff here
  var result = doWork(data);

  // the first parameter is the error which is null here,
  //but if an error occurs, then a new Error should be passed
  return callback(null, result);
}
```


Copyright © Capgemini 2017. All Rights Reserved

Handling errors for Event Emitter

- The callback pattern works well for simple operations that have a start and an end state.
- The event emitter pattern allows to decouple the producers and consumers of events using a standard interface and works well with state changes.
- All events are treated equally, with one exception: errors. In the special case of the event name being "error", the error is instead thrown into the event loop, generating an uncaught exception.
- Uncaught exception can be catched by listening to the uncaughtException that the global process object emits:
  - process.on('uncaughtException', ...) //catch EventEmittor errors

Copyright © Capgemini 2017. All Rights Reserved

Managing asynchronous operations using async.js

- Async is a utility module which provides straight-forward, powerful functions for working with asynchronous JavaScript.
- Async provides around 70 functions that include the usual 'functional' suspects (map, reduce, filter, each…) as well as some common patterns for asynchronous control flow (parallel, series, waterfall…).
- All the functions follow the Node.js convention of providing a single callback as the last argument of your asynchronous function -- a callback which expects an Error as its first argument -- and calling the callback once.
- https://caolan.github.io/async/docs.html provides a complete documentation for all the methods

Copyright © Capgemini 2017. All Rights Reserved

Express.JS Best practices

- Common best practices for creating node and express applications.
  - Always begin a node project using npm init.
  - Always install dependencies with a --save or --save-dev. This will ensure that if project moved to a different platform, by simply running npm install to install all dependencies.
  - Use lowercase file names and camelCase variables.
  - Don't push node_modules to your repositories.
  - Use a config file to store variables
  - Group and isolate routes to their own file.

Copyright © Capgemini 2017. All Rights Reserved

Express.JS Project Structure - Website

```
test-project/
   node_modules/
   config/
      db.js              //Database connection and configuration
      credentials.js     //Passwords/API keys for external services used by your app
      config.js          //Other environment variables
   models/               //For mongoose schemas
      users.js
   routes/               //All routes for different entities in different files
      users.js
    views/
      index.jade
      404.jade
      ...
   public/              //All static content being served
      images/
      css/
      javascript/
   app.js
   routes.js            //Require all routes in this and then require this file in app.js
   package.json
```


Copyright © Capgemini 2017. All Rights Reserved

Express.JS Project Structure - RESTful APIs

```
test-project/
   node_modules/
   config/
      db.js                  //Database connection and configuration
      credentials.js         //Passwords/API keys for external services used by your app
   models/                   //For mongoose schemas
      users.js
   routes/                   //All routes for different entities in different files
      users.js
    app.js
   routes.js                 //Require all routes in this and then require this file in app.js
   package.json
```


Copyright © Capgemini 2017. All Rights Reserved

Things to do in node environment / setup

- Following things will improve app's performance.
  - Set NODE_ENV to “production”
  - Ensure your app automatically restarts
  - Run your app in a cluster
  - Cache request results
  - Use a reverse proxy

Copyright © Capgemini 2017. All Rights Reserved

Summary

- Node.js highly encourages a non-blocking style of programming.
- Node modules should expose an error-first callback interface
- Async.js package is very useful for managing asynchronous operations in Node.
- Nodemon is a utility that will monitor for any changes in source and automatically restart server.
- Node Inspector is a debugger interface for Node.js applications

Copyright © Capgemini 2017. All Rights Reserved