accenture

Strategy | Consulting | Digital | Technology | Operations

High performance. Delivered.

Serverside development with NodeJS

Learning Objectives

At the end of this module, you will be able to understand:

- NodeJS Concepts
- NodeJS Frameworks
- NPM Module
- Events - Event emitters, event requests, event listening
- Streams - Reading, writing, piping, solving backpressure
- NodeJS Promises (Bluebird Promises Library)
- ExpressJS Building Blocks and Middleware (Express 4.x.x)
- User Parameters, Parsers and Routes
- Connecting to Database(MySQL/MongoDB)

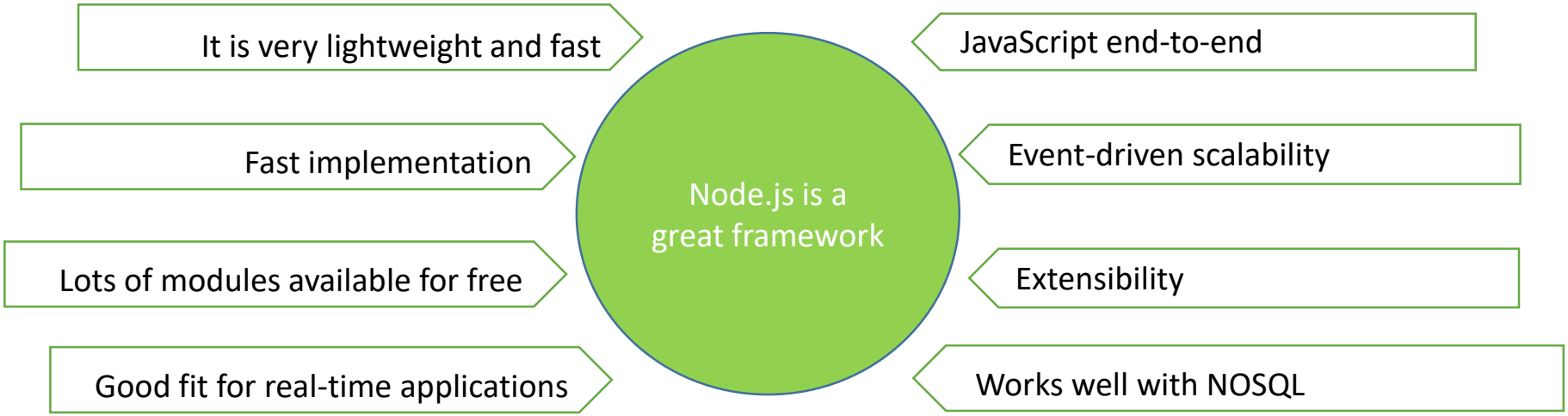
Node.JS Introduction

- NodeJS is an open source, cross-platform runtime environment used to build scalable network applications using JavaScript on the server side.
- Node.js was originally written in 2009 by Ryan Dahl.
- It can be used to build applications like:
  - Chat Servers
  - File uploaders
  - Ad Servers
  - Real time Data Apps

Node.JS Introduction

- NodeJS is an open source, cross-platform JavaScript runtime environment for executing JavaScript code server-side
- It is primarily used to build fast and scalable network applications as Web Servers
- Node.js was originally written in 2009 by Ryan Dahl
- Its written in C,C++ and JavaScript
- Built on Google Chrome's V8 JavaScript Engine
- It has an event-driven architecture capable of asynchronous I/O
- It runs single threaded event based loop, so all executions become non-blocking

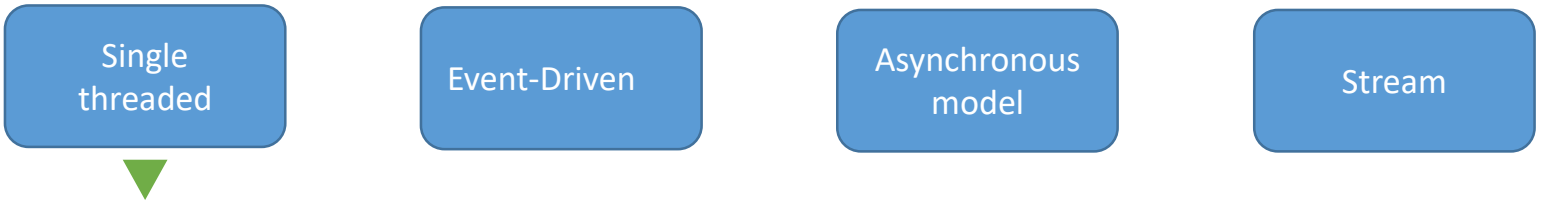
Reason to use NodeJS



NodeJS rigid principles

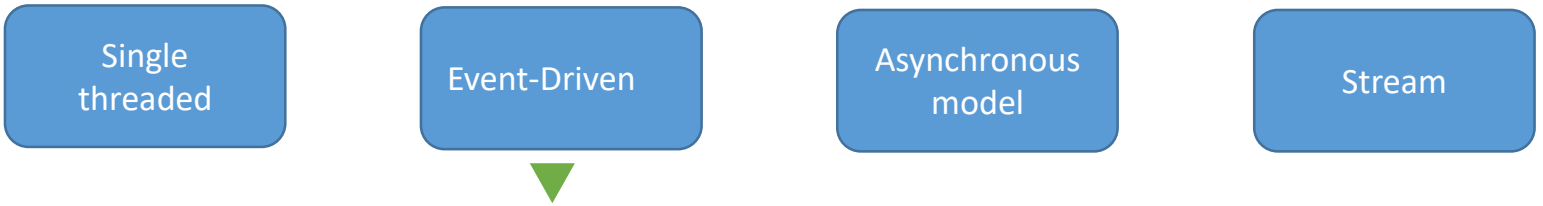
- A Node program/process runs on a single thread, ordering execution through an event loop
- Web applications are I/O intensive, so the focus should be on making I/O fast
- Program flow is always directed through asynchronous callbacks
- Expensive CPU operations should be split off into separate parallel processes, emitting events as results arrive
- Complex programs should be assembled from simpler programs
- Design choices aim to optimize throughput and scalability in Web applications with many input/output operations, as well as for real-time Web applications

Features



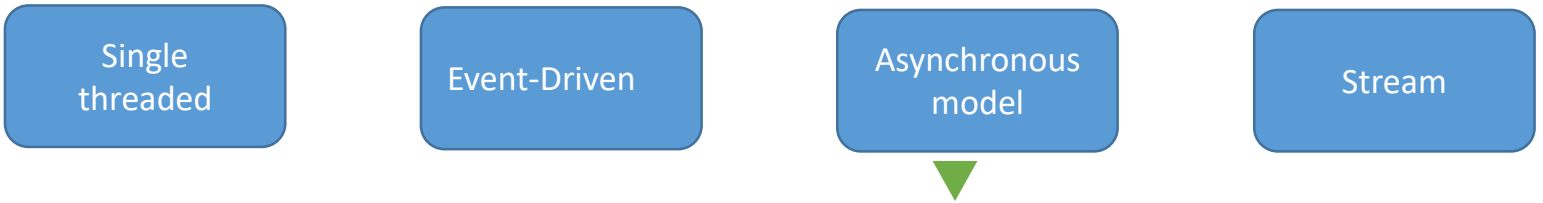
- Node.js operates on a single thread, using non-blocking I/O calls, allowing it to support tens of thousands of concurrent connections without incurring the cost of thread context switching.
- The design of sharing a single thread among all the requests that use the observer pattern is intended for building highly concurrent applications, where any function performing I/O must use a callback.
- In order to accommodate the single-threaded event loop, Node.js utilizes the libuv library that, in turn, uses a fixed-sized thread pool that is responsible for some of the non-blocking asynchronous I/O operations.
- Execution of parallel tasks in Node.js is handled by a thread pool. The main thread call functions post tasks to the shared task queue that threads in the thread pool pull and execute

Features



- Event driven programming is a programming paradigm in which the flow of the program is determined by events like messages from other programs or threads.
- In an event-driven application, there is a main loop that listens for events, and then triggers a callback function when one of those events is detected.
- Node uses observer pattern.
- Node thread keeps an event loop and whenever any task get completed, it fires the corresponding event which signals the event listener function to get executed
- Node.js uses an event loop for scalability, instead of processes or threads.
- callbacks are defined, and the server automatically enters the event loop at the end of the callback definition. Node.js exits the event loop when there are no further callbacks to be performed.

Features



- Node.js works asynchronously by using the event loop and callback functions, to handle multiple requests coming in parallel
- All APIs of Node.js are asynchronous.
- This feature means that if a Node receives a request for some Input/Output operation, it will execute that operation in the background and continue with the processing of other requests.
- It will not wait for the response from the previous requests.
- Node.js has an event-driven architecture capable of asynchronous I/O

Features

Single threaded

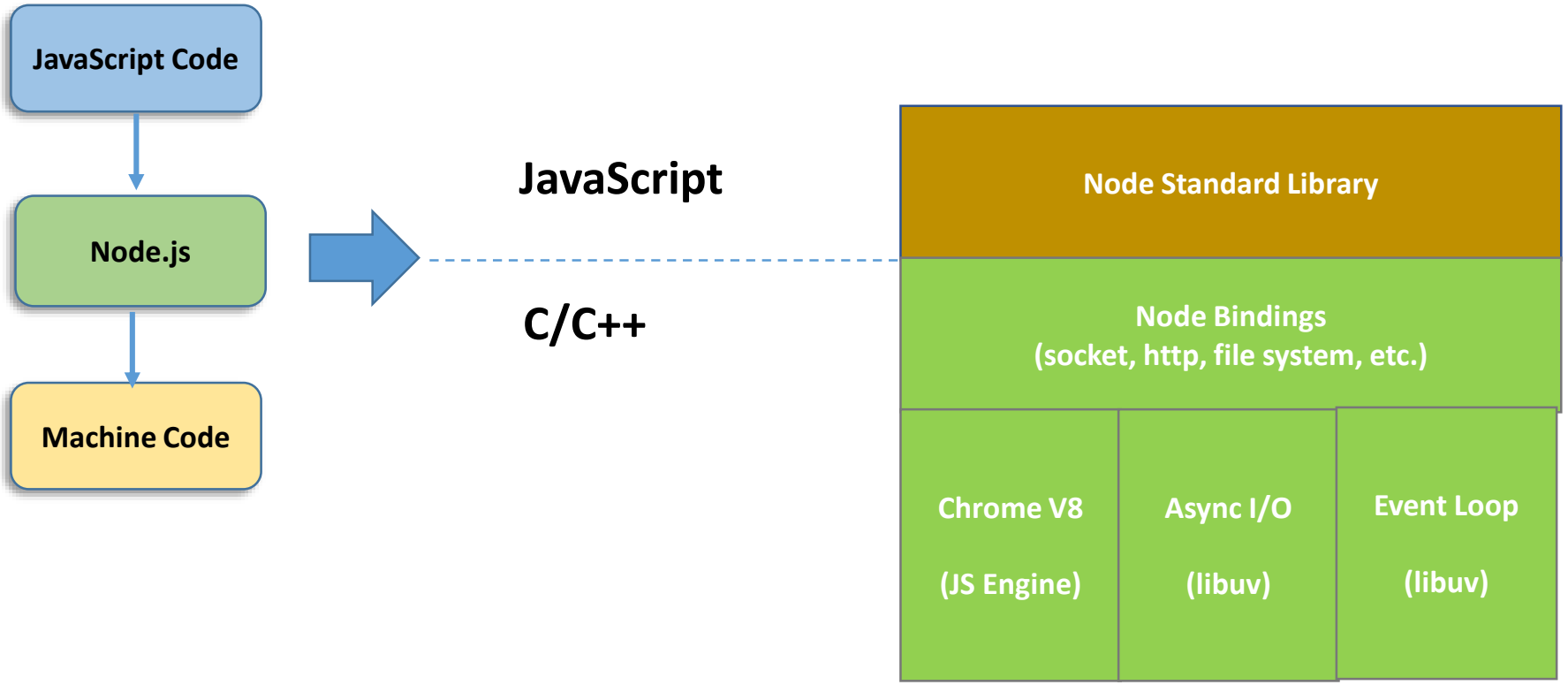
Event-Driven

Asynchronous model

Stream

- Streams are objects that let to read data from a source or write data to a destination in continuous fashion
- NodeJS application never buffers any data and simply output the data in chunks.
- It reduces the complete processing time and benefits the developers when they are working on real-time audio or video encoding. E.g., file uploading in real time, file encoding while uploading, building proxies between data layers.

Architecture



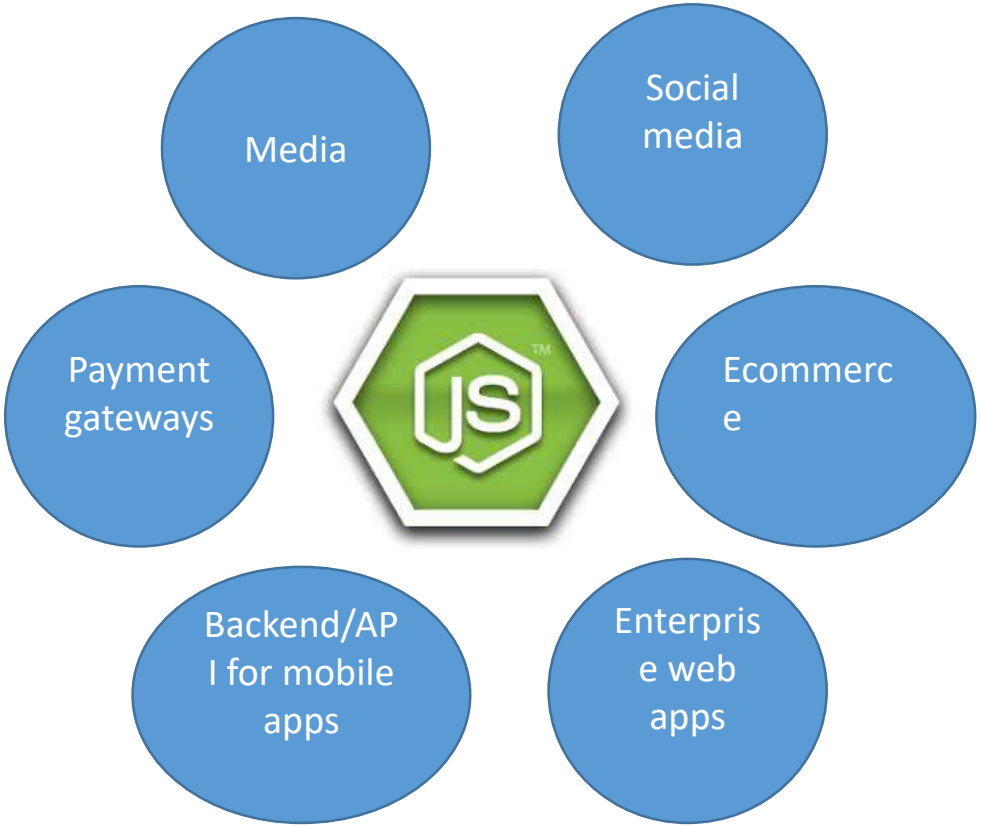
libuv

- It was primarily developed for use by Node.js
- libuv as a high performance evented I/O library which offers the same API on Windows and Unix.
- libuv enforces an asynchronous, event-driven style of programming
- Its core job is to provide an event loop and callback based notifications of I/O and other activities
- Offers core utilities like timers, non-blocking networking support, asynchronous file system access, child processes etc.
- Projects that use libuv are Luvit, Julia, pyuv, and others.

JavaScript V8 Engine

- The V8 JavaScript Engine is an open source high-performance JavaScript engine developed by the Chromium Project for the Google Chrome web browser.
- It is written in C++.
- It compiles and executes JavaScript source code, handles memory allocation for objects, and garbage collects objects it no longer needs
- It has also been used in Couchbase, MongoDB and Node.js that are used server-side.

Potential application areas of Node.js



Companies using NodeJS

Walmart

ebay

PayPal

DOW JONES

intuit

NETFLIX

LinkedIn

The New York Times

Microsoft

UBER

YAHOO!

Kingfisher

Pros and Cons

Pros

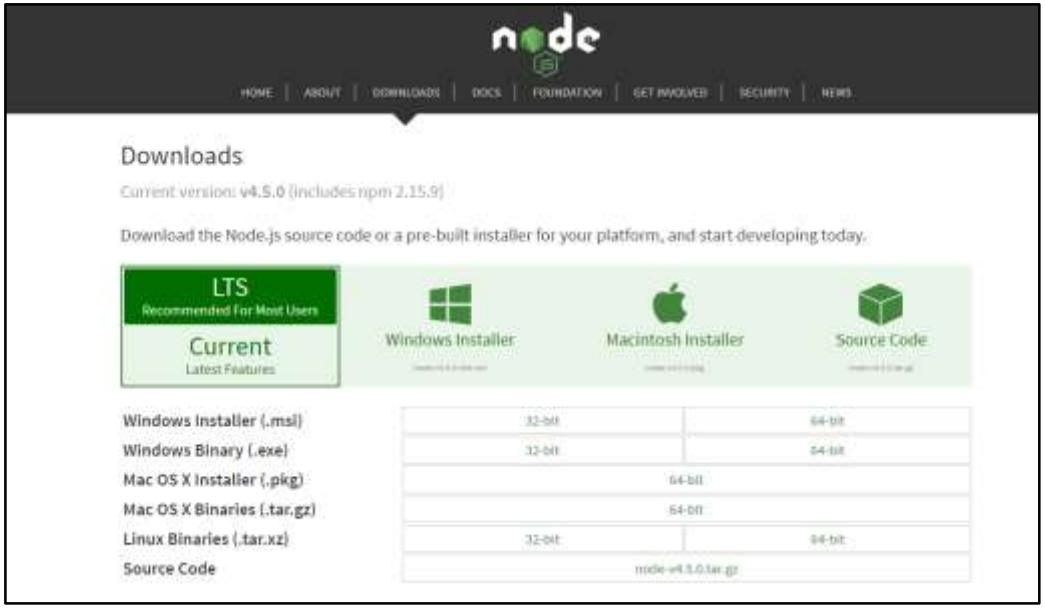
- Asynchronous event driven IO helps concurrent request handling.
- It's great to have common language on both client and server.
- NPM, the Node packaged modules has already become huge, and still

Cons

- Any CPU intensive computation will block Node.js responsiveness, so a threaded platform is a better approach.
- Every time using a callback end up with tons of nested callbacks.
- Using Node.js with a

Installation

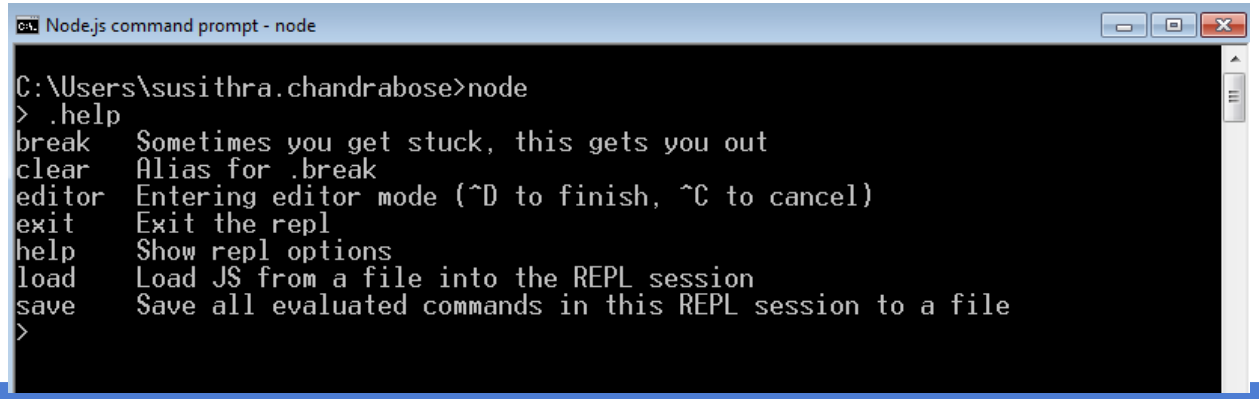
- Stand-alone installers available at http://nodejs.org/download/



- Verify installation and node installation using
  - node –version and npm --version

REPL

- Node.js provides you with a REPL (read-evaluate-print-loop)
- Test arbitrary JavaScript and experiment and explore solutions to the problem.
- At each step, the REPL prints the outcome of the last statement executed.
- The REPL does not execute your input code until all brackets have been balanced.



REPL Commands

- .break - When in the process of inputting a multi-line expression, entering the .break command (or pressing the <ctrl>-C key combination) will abort further input or processing of that expression.
- .clear - Resets the REPL context to an empty object and clears any multi-line expression currently being input.
- .exit - Close the I/O stream, causing the REPL to exit.
- .help - Show this list of special commands.
- .save - Save the current REPL session to a file: > .save ./file/to/save.js
- .load - Load a file into the current REPL session. > .load ./file/to/load.js
- .editor - Enter editor mode (<ctrl>-D to finish, <ctrl>-C to cancel)

First Demo

first_node.js

```
var date=new Date().getHours();
console.log(date +':00 is a good time to learn NodeJS');
```

Command Line

```
$ node first_example.js
14:00 is a good time to learn NodeJS
```

Blocking Vs. Non-Blocking Code

- Blocking Code

```
Read file from Filesystem, set equal to "contents"
Print contents
Do something else
```

- Non-Blocking Code

```
Read file from Filesystem
  whenever you're complete, print the contents
Do Something else
```

This is a "Callback"

Blocking Vs. Non-Blocking Code

- Blocking Code

```
var contents = fs.readFileSync('/etc/hosts');
console.log(contents);
console.log('Doing something else');
```

Stop process until complete

- Non-Blocking Code

```
fs.readFile('/etc/hosts', function(err, contents) {
  console.log(contents);
});
console.log('Doing something else');
```

NodeJS Modules

Modules

- Modules are node libraries that helps us to use code of one file as part of another file.
- Module then acts as an API injected in to the bootstrap file.
- A module encapsulates related code into a single unit of code.
- When creating a module, this can be interpreted as moving all related functions into a file.
- Node modules run in their own scope so that they do not conflict with other modules. Node relatedly provides global access to help facilitate module interoperability.
- Node.Js has a simple module loading system. In node.Js, files and modules are in one-to-one correspondence (each file is treated as a separate module).

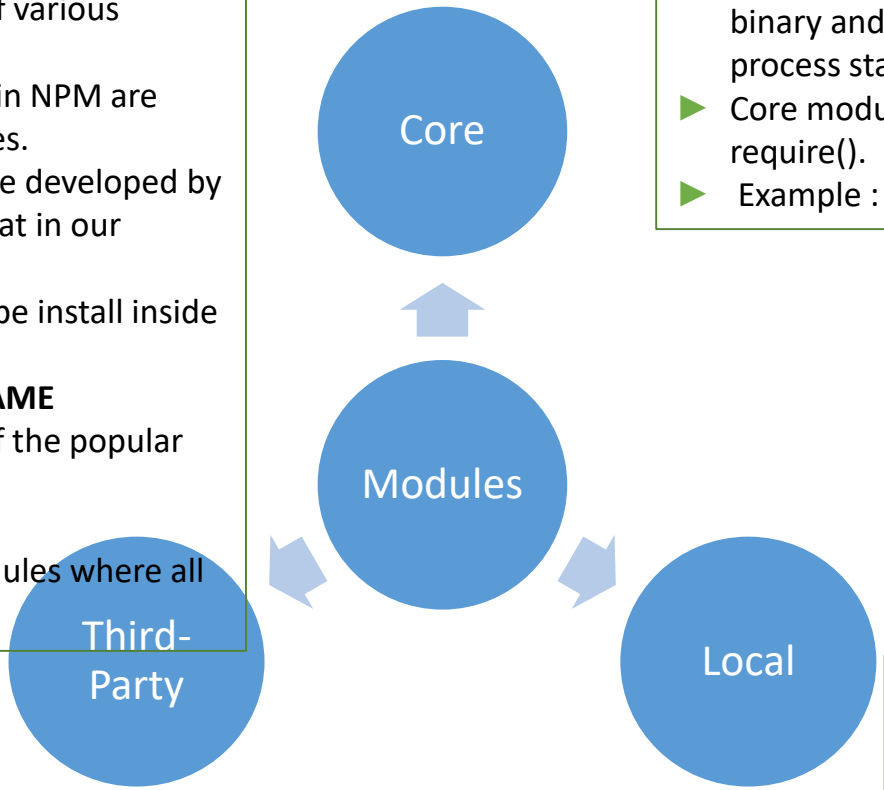
require is an importer of code from other locations

exports bundles the code in an object and allows it to be reused

Modules Types

- Include bare minimum functionalities of Node.js.
- The core modules are defined in node's "lib" folder.
- Node has several modules compiled into the binary and load automatically when Node.js process starts. .
- Core modules are loaded by passing name to require().
- Example : file system – require('fs');

- Packages are collection of various modules.
- Packages that are stored in NPM are called third party packages.
- These type of modules are developed by others and we can use that in our project.
- Third party modules can be install inside the project folder.
  npm install PACKAGE_NAME
- **Example :**To install one of the popular packages gulp
  npm install gulp
- Creates folder node_modules where all packages are installed



- Modules created locally in Node.js application.
- Include different functionalities of application in separate files and folders.
- It can also be packaged and distribute it via NPM, so that Node.js community can use it.

Core Modules

Core

Core Module	Description
http	http module includes classes, methods and events to create Node.js http server.
url	url module includes methods for URL resolution and parsing.
querystring	querystring module includes methods to deal with query string.
path	path module includes methods to deal with file paths.
fs	fs module includes classes, methods, and events to work with file I/O.
util	util module includes utility functions useful for programmers.
buffer	The buffers module provides a way of handling streams of binary data.
OS	The os module provides a number of operating system-related utility methods
cluster	The cluster module provides a way of creating child processes that runs simultaneously and share the same server port.

Popular npm modules

Third-Party

express	• Express.js, a Sinatra-inspired web development framework for Node.js.
hapi	• A very modular and simple to use configuration-centric framework for building web and services applications
socket.io	• Server-side component that enables real-time bidirectional event-based communication.
pug (formerly Jade)	• One of the popular templating engines, inspired by HAML, a default in Express.js.
mongo	• MongoDB wrappers to provide the API for MongoDB object databases in Node.js
bluebird	• A full featured Promise library. Allows to “promisify” other node modules.

accenture

Local Modules

Local

- Export functionalities of one file

```
exports.myText = 'This text was exported from myModule.';
```

- Import other files functionality

```
var myModule = require('./my-module.js');

console.log('text from module:',
myModule.myText);
```

Modules :Local Modules

- Export functionalities

```
exports.add = function(a,b)
return a+b;
};

exports.subtract = function(a,b) {
  return a-b;
};
```

or

```
module.exports = {
  add: function(a,b) {
    return a+b;
  },

  subtract: function(a,b) {
    return a-b;
  }
};
```

Local Modules: Demo

Local

1. Declare a literal in a module file and use this literal in another module.
2. Declare a variable in a module and use this object in another module.
3. Declare a function in a module that add two numbers and returns sum of these numbers. Call this function in another module.
4. Crate a module with a class Person. Person has first and list name as fields and a function that returns full name. Use this class to initialize fields and display the full name in another module.
5. Create a module that exports an object and use the object's properties in another modules.
6. Create a module that exports many functions as object's properties and call them in another modules.

Local Modules: Module from Different folder

Local

Syntax: require(‘path/module.js’)
i.e. require(‘./util/log.js’)

NodeJS allows to import modules using the folder name. This folder is considered as package.
i.e. Require (‘./util’)

In such as case we need to create package.js file that specify the module information.

i.e.

```
{
        “name”: “log” ,
        “main”: “./export-multi-functions.js”
}
```

Reading from File

data1.jso
n

```
{
    "name": "Modern Web Academy"
}
```

Reading JSON from file in two ways

```
var fs = require('fs');
var data = require('./data1.json');

console.log(data.name);

fs.readFile('./data1.json', 'utf-8', function(err, data) {

    data = JSON.parse(data);
    console.log(data.name);
});
```

File System module

- Node.js includes fs module to access physical file system. The fs module is responsible for all the asynchronous or synchronous file I/O operations.
- The asynchronous form always takes 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.
- When using the synchronous form any exceptions are immediately thrown. Exceptions may be handled using try/catch, or they may be allowed to bubble up.

Writing to a file

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

var myString = '{"name": "Modern Web Academy"}';

fs.writeFile('myFile.json', myString);

var actualObject = {
    name: 'Modern Web Academy'
};

fs.writeFile('myProcessedFile.json', JSON.stringify(actualObject));
```

Read directories

- Reading directories is similar to reading files
- Uses fs object – FileSystem
- Uses readdir method

read-directory-demo.js

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

fs.readdir('exampleDir', function(err,
data) {
    console.log(data);
});
```

Displaying list of files as array

Terminal

```
C:\Users\james.hicks\Documents\AOWP\nodeJS\NodeJS_Demos\Demo6>node read-directory-demo.js
[ 'Downloads', 'Photos', 'Videos' ]
```

Demo

Example to illustrate the use of readline module

```
const readline = require('readline');

const prompt = readline.createInterface({
    input: process.stdin,
    output: process.stdout
});

prompt.question('What do you think of Node.js? ', (answer) => {
    console.log('Thank you for your valuable feedback:'+ answer);
prompt.close();
});
```

Importing read line module to read input from console

interface for reading data from a stream one line at a time

A statement or query to write to output, prepended to the prompt and a callback function that is invoked with the user's input in response to the query.

readline.Interface is closed because the interface waits for data to be received on the input stream.

Executing the node code

```
$ node readline_demo.js
What do you think of Node.js? good
Thank you for your valuable feedback: good
```

fs Modules: Methods

fs.readFile(fileName [,options], callback)
Reads existing file.

fs.writeFile(filename, data[, options], callback)
Writes to the file. If file exists then overwrite the content otherwise creates new file.

fs.open(path, flags, [, mode], callback)
Opens file for reading or writing.

fs.rename(oldPath, newPath, callback)
Renames an existing file.

fs.appendFile(file, data, [, options], callback)
Appends new content to the existing file.

fs Modules: Methods

fs.exists(path, callback)
Determines whether the specified file exists or not.

fs.mkdir(path, callback)
Creates a directory.

fs.rmdir(path, callback)
Renames an existing directory.

fs.readdir(path, callback)
Reads the content of the specified directory.

Path Module

- The path module provides utilities for working with file and directory paths..
- The key motivation for using the 'path' module is to remove inconsistencies in handling file system paths.
- Almost all these methods perform only string transformations

Example to illustrate use of path function

```
var path= require('path');
 var result ="";
 result+= path.normalize('/foo/bar//baz/asdf/quux/..');
 result+="\n" + path.join('/foo', 'bar', 'baz/asdf', 'quux', '..');
 result+="\n"+path.resolve('foo/bar', '/tmp/file/', '..', 'a/../subfile');
 console.log(result);
```

Displaying the result

```
$ node path_demo.js
\foo\bar\baz\asdf
\foo\bar\baz\asdf
C:\tmp\subfile
```

Path Module

path. sep:
Will return the separater character.

path. delimiter:
Will return the delimiter character.

path.extName(path):
Will return the extension of the path.

path. isAbsolute(path):
Will return true if path is absolute.

path. relative(fromPath, toPath)
Will return relative path for toPath from fromPath.

Buffer module

- JavaScript language had no mechanism for reading or manipulating streams of binary data.
- When dealing with TCP streams or the file system, it's necessary to handle octet streams.
- The Buffer class was introduced as part of the Node.js API to make it possible to interact with octet streams in the context of things like TCP streams and file system operations.
- A buffer is a region of a physical memory storage used to temporarily store data while it is being moved from one place to another.
- In node, each buffer corresponds to some raw memory allocated outside V8.
- A buffer acts like an array of integers, but cannot be resized.

Example to illustrate creating a buffer object and use of methods

```
buf = new Buffer(10);
buf.write("Accenture", 0, "ascii");
console.log(buf.toString('base64'));
buf = buf.slice(0,5);
console.log(buf.toString('utf8'));
```

Output

```
$ node buffer.js
QWNjZW50dXJlAA==
Accen
```

Buffer module

Creating Buffer:

var buf = new Buffer(size);

Var buf= Buffer.alloc(size);

Var buf = Buffer.from(string/array/buffer);

Example: var buffer1 = Buffer.from(“Sample buffer”);

Buffer module

Writing to buffer:

buf.write(string[, offset][, length][, encoding]);

buf. writeInt32LE(value);

buf. writeDoubleBE(value);

Buffer module

Reading from buffer:

buf.toString([encoding][, start][, end]) ;

buf. readFloatBE();

buf. readUInt16BE();

util module

- The util module contains a number of useful functions that are used for general purpose.
- Utility Library help convert and validate format of value.

Example to illustrate the use of util module

```
var util = require('util');
console.log(util.format('%s:%s', 'Name', 'Accenture', 'Solutions'));
console.log(util.format('{%j:%j}', 'Name', 'Accenture'));
console.log(util.isArray([]));
console.log(util.isArray(new Array));
console.log(util.isArray({}));
console.log(util.isDate(new Date()));
```

Output

```
$ node util.js
Name:Accenture
Solutions
{"Name":"Accenture"}
true
true
false
true
```

util module: methods

util.isArray(object) : Checks weather object is an array. Returns true / false.
util.isBoolean(object) : Returns true / false.
util.isBuffer(object) : Returns true / false.
util.isDate(object) : Returns true / false.
util.isError(object) : Returns true / false.
util.isFunction(object) : Returns true / false.
util.isNull(object) : Returns true / false.
util.isNullOrUndefined(object) : Returns true / false.
util.isNumber(object) : Returns true / false.
util.isObject(object) : Returns true / false.

OS module

- The os module provides a number of operating system-related utility method .
- It also provides information about the computer's operating system

Example to illustrate the use of OS module

```
var os = require('os');
console.log(os.hostname());
console.log(os.type());
console.log(os.platform());
console.log(os.arch());
console.log(os.release());
console.log(os.cpus().length);
console.log('percentage Memory consumed '+(100*(1 -
os.freemem()/ os.totalmem())));
```

Outpu t

```
$ node os.js
BDC8-LX-1WQSLC2
Windows_NT
win32
x64
10.0.14393
4
percentage Memory
consumed
63.01390008233431
```

Cluster module

- A single instance of Node.js runs in a single thread.
- To take advantage of multi-core systems, the user will sometimes want to launch a cluster of Node.js processes to handle the load.
- The cluster module allows easy creation of child processes that all share server ports

Creating a child process based on number of logical core of machine

```
var cluster = require('cluster');
if (cluster.isMaster)
   {
    var numCPUs = require('os').cpus().length;
    for (var i = 0; i < numCPUs; i++) {
   cluster.fork();
  }
 Object.keys(cluster.workers).forEach(function(id) {
      console.log(cluster.workers[id].process.pid);
   });
 }
```

Displaying the child's ID

```
$ node Cluster.js
100644
78400
107528
107612
```

cluster Modules: Demo

Create a module to use cluster module and create the worker process to perform the separate tasks of printing console. Finish the process once the task is completed.

Functions

The function is a named procedure that usually execute logically related statements to achieve a specific task an may return a value:

- Function definition
- Function declaration
- Function call

```
function func_name () {
        statements;
}

func_name();
```

Functions

```
function func_name (parameter_list) {
        statements;
        return return_value;
}
var var_name =  func_name(parameters);
```

Default value:

Parameter list : para1 = value1, para2=value2

Optional Parameters:

Parameter list : para1, para2?

Check: if(para2 != undefined)

Functions

Rest parameters

Rest parameters are used when number of parameters are not known.

Place ellipsis … before the parameter

i.e.

...restOfName

```
function func_name (para1, para2, …restPara)  {
        statements;
        // Access restPara[0]
        return return_value;
}
```

Example

Create a module to find simple interest using a function. Function has three parameters amount, years and ROI. If ROI is not specified its should be 6%.

Functions

Anonymous Function: is a function without name.

Var ref = function () { }

Call to the function: ref();

Recursive Function: When a function calls itself.

The function is called inside same function definition.

Example: factorial function

Lambda Functions

Fat arrow / Lambda function is a concise mechanism to represent anonymous function.

There are 3 parts to a Lambda function:

- Parameters: A function may optionally have parameters
- The fat arrow notation/lambda notation (=>): It is also called as the goes to operator
- Statements: represent the function's instruction set

Syntax:

```
( param1, parma2,…param n )=>statement;
```

Lambda Functions

- Some characteristics of Lambda function.
- Parentheses () are not necessary if one and only one parameter is there.
- Curly braces {} are not required if one and only one statement is there in the body of the function.
- return statement not required if one and only one statement is there.
- Empty parentheses () are needed when there is no parameter

- var sum = (x, y) => x + y;
- console.log(sum(5,7));

Example

Modify the previous example to define the function using lambda notation and call the function.

Global Objects

Node.js has a number of built-in global identifiers. These objects are available in all modules. Some of these objects are true globals and can be accessed from anywhere, other exist at module level in every module.

Identifiers	Description
global	The global namespace. Setting a property to this namespace makes it globally visible within the running process.
__filename	Contains the absolute path of the currently executing file
__dirname	Contains the path to the root directory of the currently executing script.
module	A reference to the current module. exports is used for defining what a module exports and makes available through require().
exports	A reference to the module.exports that is shorter to type
require()	The require() function is a built-in function, and used to include other modules that exist in separate files, a string specifying the module to load. It accepts a single argument
process	A process object is a global object, which provides interaction with the current Node process and can be accessed from anywhere

Global Object: Demo

Create a module to use global objects such as __filename, __dirname, setTimeout, setInterval, console etc.

%

Event Handling

Node.js Events

Node.js is perfect for event-driven applications.

One of the reasons for Node.js' high speed is the fact that it is coded around events.

Node.js core API is built around an idiomatic asynchronous event-driven architecture in which certain kinds of objects called "emitters" periodically emit named events that cause Function objects i.e listeners to be called.

For instance: a net.Server object emits an event each time a peer connects to it.

Event Loop

The event loop makes Node.js such a valuable framework, allowing for thousands and tens of thousands of simultaneous connections and responsive reactions to IO-based events

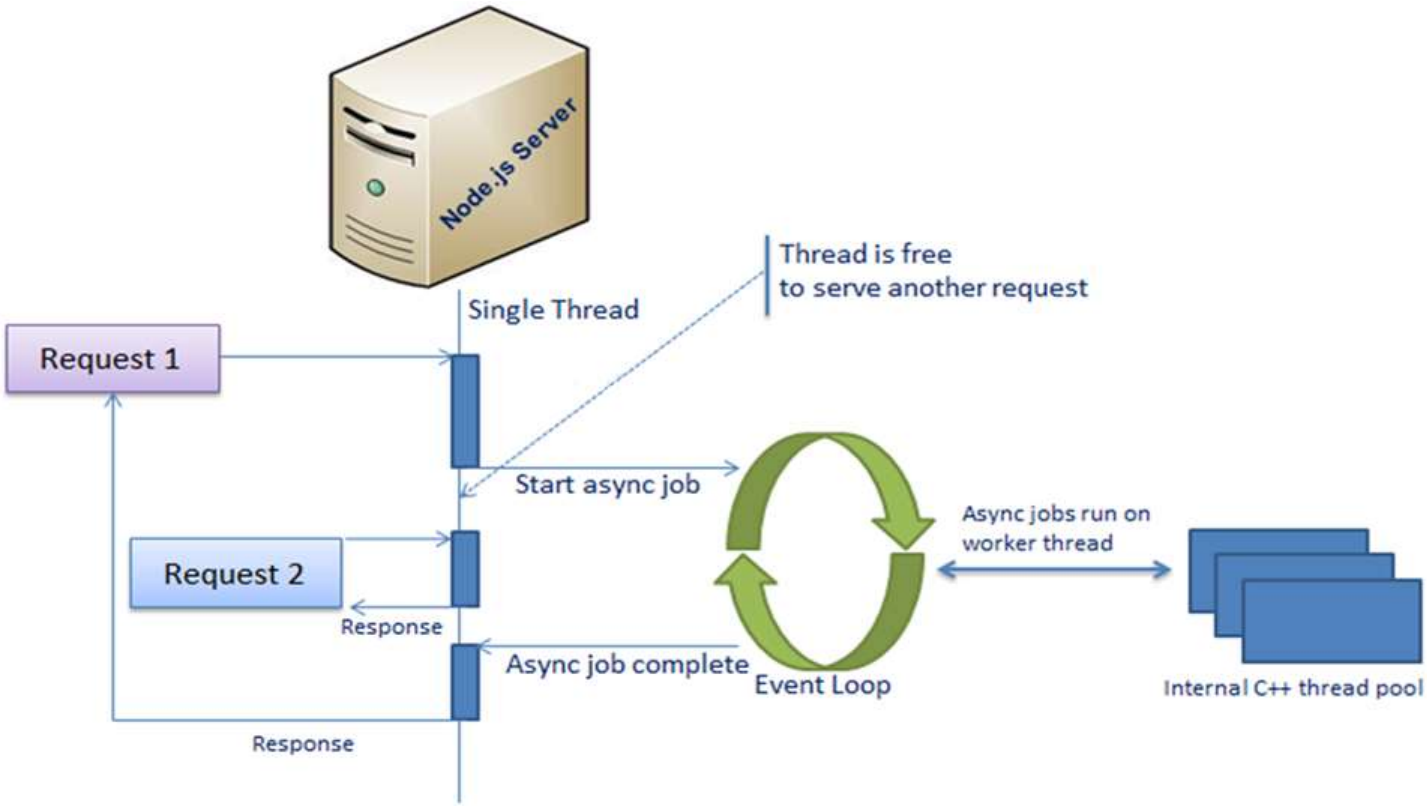
In most programming languages and frameworks, this is accomplished by the use of multiple threads or some concurrent execution mechanism, but Node.js uses an event loop for scalability, instead of processes or threads

Instead, code is executed within an iterative event loop, which cycles through a series of phases and notifies the application for the completion of these steps.

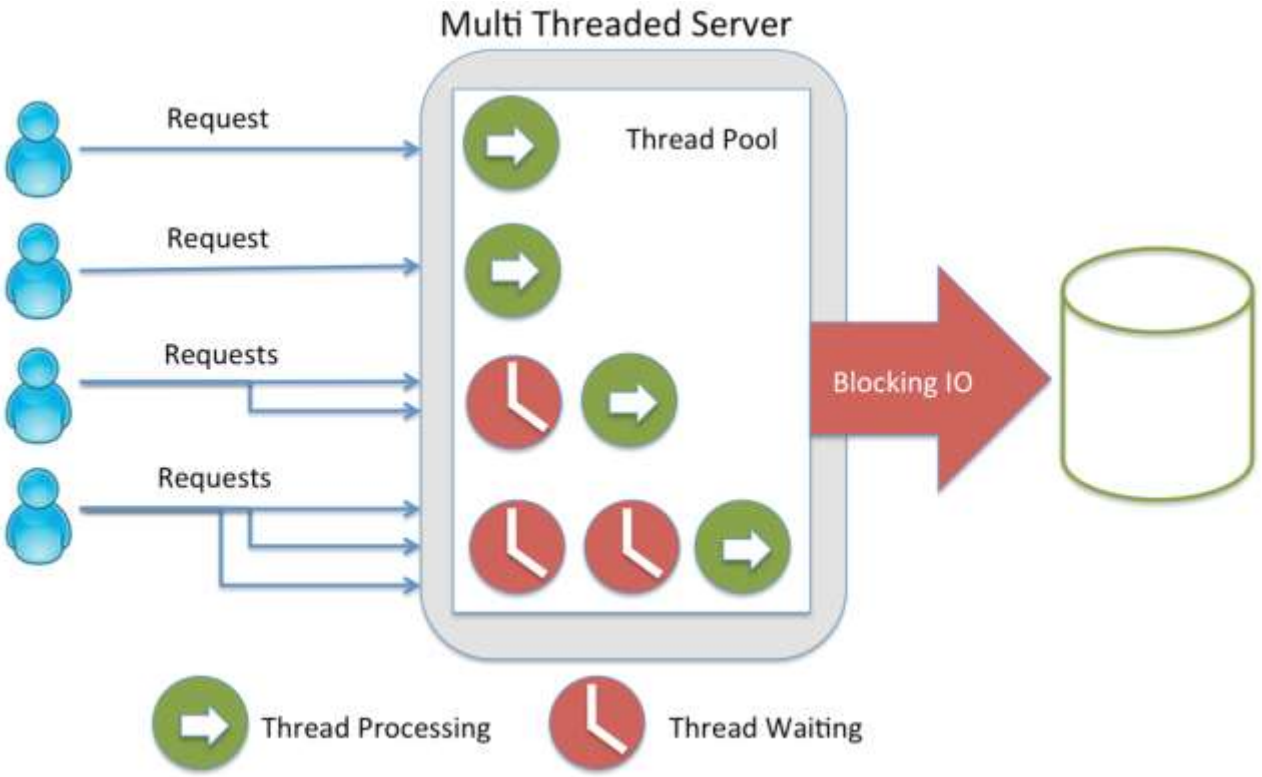
Node.js's event loop does not need to be called explicitly ,instead callbacks are defined, and the server automatically enters the event loop at the end of the callback definition. Node.js exits the event loop when there are no further callbacks to be performed.

Event-loops are the core of event-driven programming, almost all the UI programs use event-loops to track the user event, for example: Clicks, Ajax Requests etc

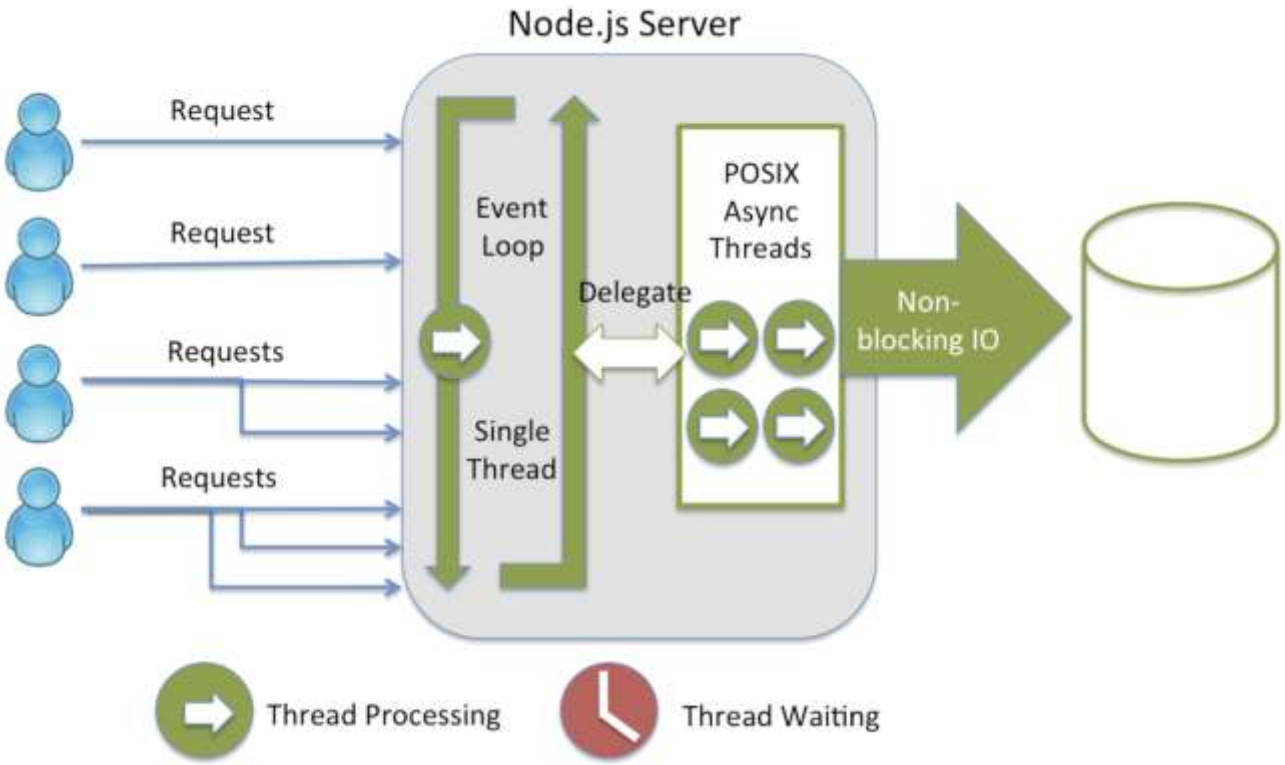
Event Loop



Traditional servers



Node servers



Threads VS Event-driven

Threads	Asynchronous Event-driven
Lock application / request with listener-workers threads	only one thread, which repeatedly fetches an event
Using incoming-request model	Using queue and then processes it
multithreaded server might block the request which might involve multiple events	manually saves state and then goes on to process the next event
Using context switching	no context switches
Using multithreading environments where listener and workers threads are used frequently to take an incoming-request lock	Using asynchronous I/O facilities (callbacks) environments

EventEmitter class

- The EventEmitter is a module that facilitates communication between objects in Node. EventEmitter is at the core of Node asynchronous event-driven architecture.
- Many of Node's built-in modules inherit from EventEmitter.
- EventEmitter class lies in the events module.
- Objects of this type emit named events that cause previously registered listeners to be called.
- Emitter object basically has two main features:
  - ✓ Emitting name events.
  - ✓ Registering and unregistering listener functions.

EventEmitter methods

Methods	Description
addListener(eventName, listener)	Adds a listener at the end of the listeners array for the specified eventName.
on(eventName, listener)	Adds a listener at the end of the listeners array for the specified event named 'eventName'
once(eventName, listener)	Adds a one time listener function for the event named eventName
emit(eventName[, ...args])	Synchronously calls each of the listeners registered for the event named eventName, in the order they were registered, passing the supplied arguments to each.
listenerCount(eventName)	Returns the number of listeners listening to the event named eventName.
removeListener(eventName, listener)	Removes a listener from the listener array for the specified eventName

EventEmitter: Demo

Event is an action occurrence detected by the program to handle

```
const EventEmitter = require('events');
class MyEmitter extends EventEmitter {}
const myEmitter = new MyEmitter();

 myEmitter.on("change", function () {
 console.log("change event has occured");
 });
myEmitter.emit('change');
```

event occurred and displaying the changes happened to

```
$ node event.js
change event has occured
```

Event Emitter: Demo

Create a node module to create two named events and attach event handlers / listeners. Emit the events to execute the listeners to perform separate tasks.

Create a node module to attach multiple listener functions to same event. Display the listener counts of the event.

Streams

Node.js Streams

- Streams are objects that let read data from a source or write data to a destination in continuous fashion .
- A stream is an abstract interface for working with streaming data in Node.js.
- The stream module provides a base API that makes it easy to build objects that implement the stream interface.
- All streams are instances of EventEmitter
- There are four fundamental stream types within Node.js:

Readable streams

Stream which is used for read operation.

Example: fs.createReadStream()

Writeable streams

Stream which is used for write operation

Example: fs.createWriteStream()

Duplex streams

Stream which can be used for both read and write operation Example : net.Socket

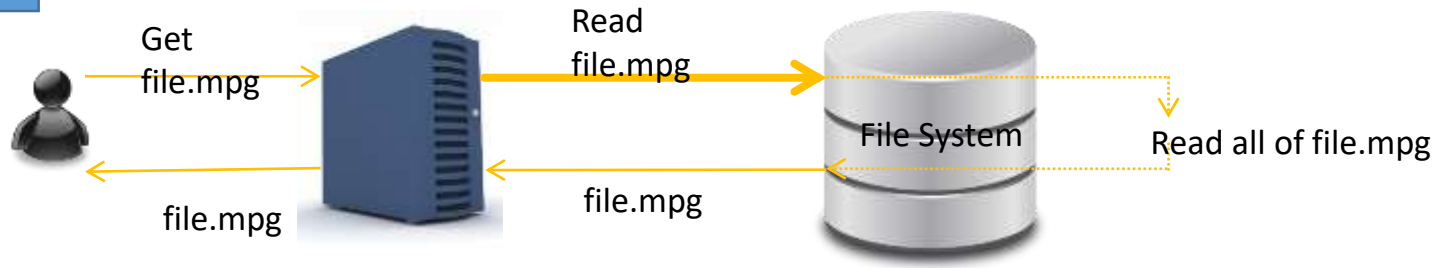
Transform streams

Duplex streams that can modify or transform the data as it is written and read like Encryption and compression streams

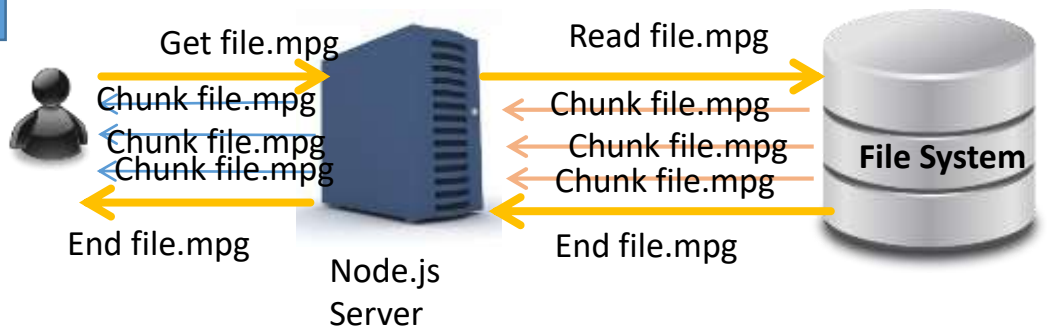
Example: zlib.createDeflate()).

Buffered VS Streams

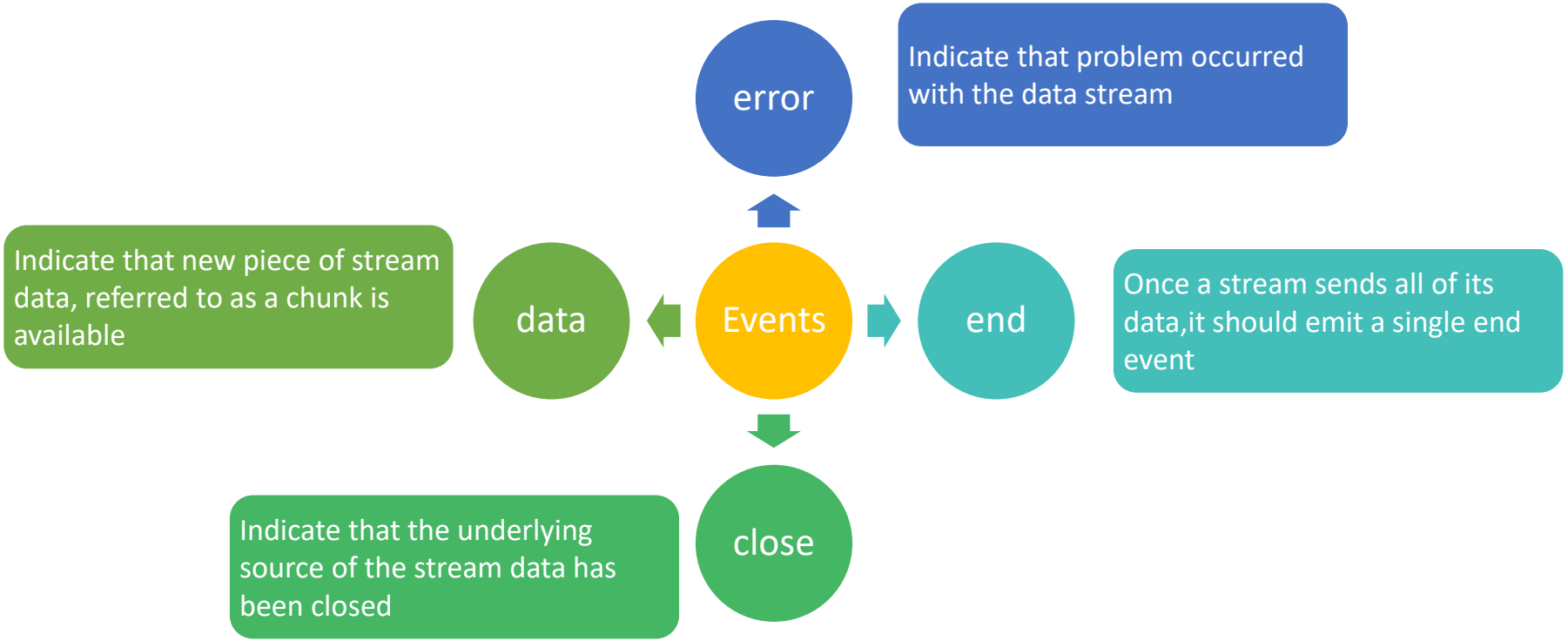
Buffered web response



Streaming web response



Streams Events



Readable Streams

- Readable streams are an abstraction for a source from which data is consumed.
- All Readable streams implement the interface defined by the stream.Readable class
- Important event of readable stream is ‘readable’
- This event is raised whenever there is new data to be read from a stream.
- Once inside the event handler, call the read function on the stream to read data from the stream.
- If this is the end of the stream, the read function returns null
- Examples of Readable streams include:
  - ✓ HTTP responses, on the client
  - ✓ HTTP requests, on the server
  - ✓ fs read streams
  - ✓ zlib streams
  - ✓ crypto streams
  - ✓ TCP sockets
  - ✓ child process stdout and stderr
  - ✓ process.stdin

Demo

Reading a file using streams

```
var fs = require("fs");
var data = '';
var readerStream =fs.createReadStream('input.txt');
readerStream.setEncoding('UTF8');
readerStream.on('data', function(chunk) {
   console.log('got chunk of', chunk.length, 'bytes');
   data += chunk;
});
readerStream.on('end',function(){
   console.log(data);
});
readerStream.on('error', function(err){
   console.log(err.stack);
});

console.log("Program Ended");
```

Creating an readable stream

Method sets the character encoding for data read

Storing chunks of data sent from readable stream

end is emitted when there is no more data to be consumed from the stream

Error occurs when stream unable to generate data

Writable Streams

- Writable streams are an abstraction for a destination to which data is written.
- All Writable streams implement the interface defined by the stream.Writable class.
- To write to a stream, call ‘write’ to write some data.
- Call end function to finish writing (end of stream).
- Data can also be written to the end function
- Examples of Writable streams include:
  - ✓ HTTP requests, on the client
  - ✓ HTTP responses, on the server
  - ✓ fs write streams
  - ✓ zlib streams
  - ✓ crypto streams
  - ✓ TCP sockets
  - ✓ child process stdin
  - ✓ process.stdout, process.stderr

Demo

Writing content to a file using streams

```
var fs = require('fs');
var data = " Writing data with streams" ;
var writableStream = fs.createWriteStream('outputFile.txt');
writableStream.write(data,'UTF8');
writableStream.end();

writableStream.on('finish', function() {
console.log("file has been written!!!");
});

writableStream.on('error', function(err){
console.log(err.stack);
});

console.log("Program Ended");
```

Creating a writable stream which creates a file in current directory

Finish event is emitted after the stream.end()

error event is emitted if an error occurred while writing

Writable Stream: Demo

Create a node module to write to a text file. Write big text data in chunks using many number of iteration in the program.

Read the this file using the read file demo.

Readable Stream: Demo

Create a node module to read a big text file and display on the console/ terminal.

Piping the Streams

- Piping is a mechanism where we provide the output of one stream as the input to another stream.
- It is normally used to get data from one stream and to pass the output of that stream to another stream.
- There is no limit on piping operations.
- All the streams support a pipe operation that can be done using the pipe member function.
- We can pipe from the readable stream to a writeable stream like pipe command line operator in unix

Demo

Copying the file by piping the streams

```
var fs = require('fs');
var file = fs.createReadStream("readme.md");
var newFile = fs.createWriteStream("readme_copy.md");

file.pipe(newFile);
newfile.on('close', function () {
      console.log('file has been copied');
});
```

Pipes all of the data from the readable into newFile

Note :It is possible to attach multiple Writable streams to a single Readable stream

Piping the Streams

Chaining the pipe:

```
readableStream
  .pipe(readWriteStream1)
  .pipe(readWriteStream2)
  .pipe(finalWriteStream);
```

Is equivalent to:

```
readableStream.pipe(readWriteStream1)
readWriteStream1.pipe(readWriteStream2)
readWriteStream2.pipe (finalWriteStream )
```

Event based alternative of pipe

Copying the file by read-write and event concept

```
readable.pipe(writable)
readable.on('data', (chunk) => {
  writable.write(chunk);
});
readable.on('end', () => {
  writable.end();
});
```

Event based alternative of pipe.

Duplex Streams

- Duplex streams are streams that implement both the Readable and Writable interfaces.
- Examples of Duplex streams include:
  - ✓ TCP sockets
  - ✓ zlib streams
  - ✓ crypto streams

Demo –Create a TCP server that can be connected to via Telnet

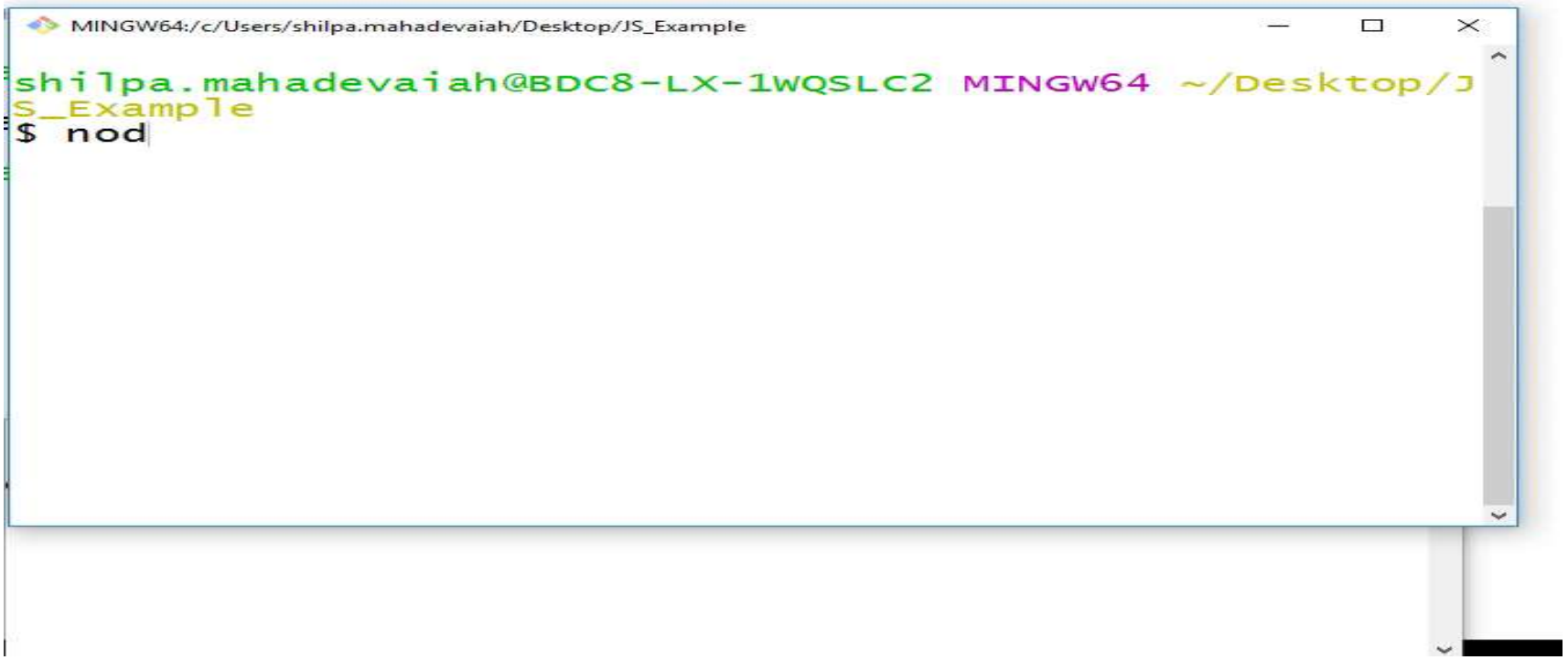
```
var stream = require("stream");
var net = require("net");
net.createServer(function(socket) {
      console.log('client connected');
      socket.write("Go ahead and type something!");
      socket.on("readable", function() {
      process.stdout.write(this.read())
   });
 })

.listen(8081 , function(){
  console.log('server bound');
});
```

Net module provides a way of creating TCP servers and TCP clients.

Creates a new TCP server.

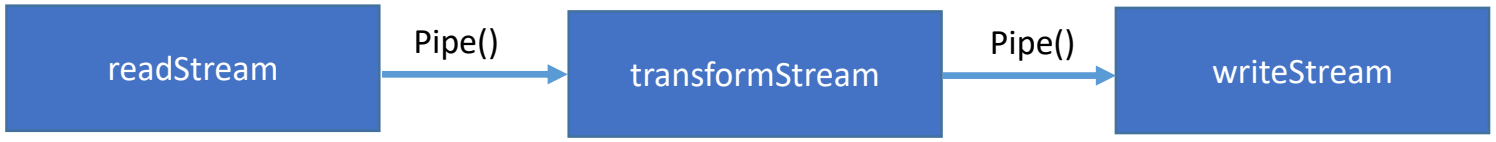
Server listening for connection on 8081 port

Demo –Create a TCP server that can be connected to via Telnet



Transform Streams

- Node.js transform streams are streams which read input, process the data manipulating it, and then output new data.
- They can be composed into this pipeline where the data flows from a readable stream into one or more transform streams and ends up in a writable stream.
- Examples of Transform streams include:
  - ✓ zlib - for gzip compressing and uncompressing
  - ✓ crypto - for encrypting, decrypting, and calculating message digests



Demo – Compressing stream with gzip

```
var fs = require('fs');
var zlib = require('zlib');

var gzip = zlib.createGzip();
var rstream = fs.createReadStream('myfile.txt');
var wstream = fs.createWriteStream('myfile.txt.gz');

rstream
  .pipe(gzip)
  .pipe(wstream)
  .on('finish', function () {
    console.log('done compressing');
  });
```

zlib module provides compression functionality implemented using Gzip

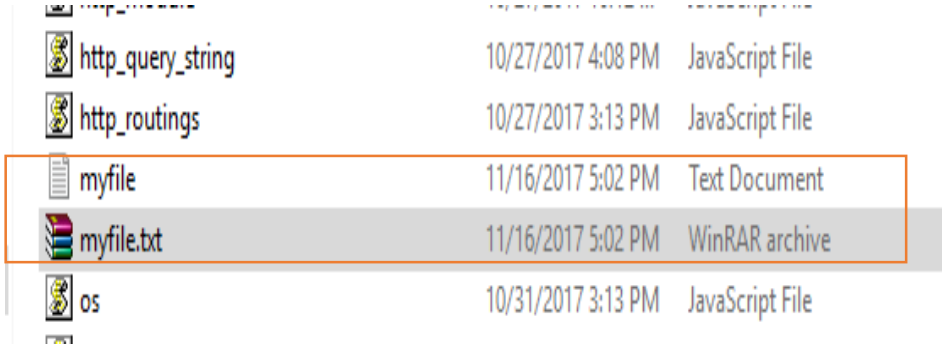
Creates and returns a new Gzip object

reads from myfile.txt

compresses

writes to myfile.txt.gz

```
$ node transform_stream.js
done compressing
```



Http core module

Creating a Web Server using Node

Node.js provides an **http** module which can be used to create an HTTP client of a server.

Hence community can build full-fetched application servers

Node.js was created specifically to make scalable server-side and networking application

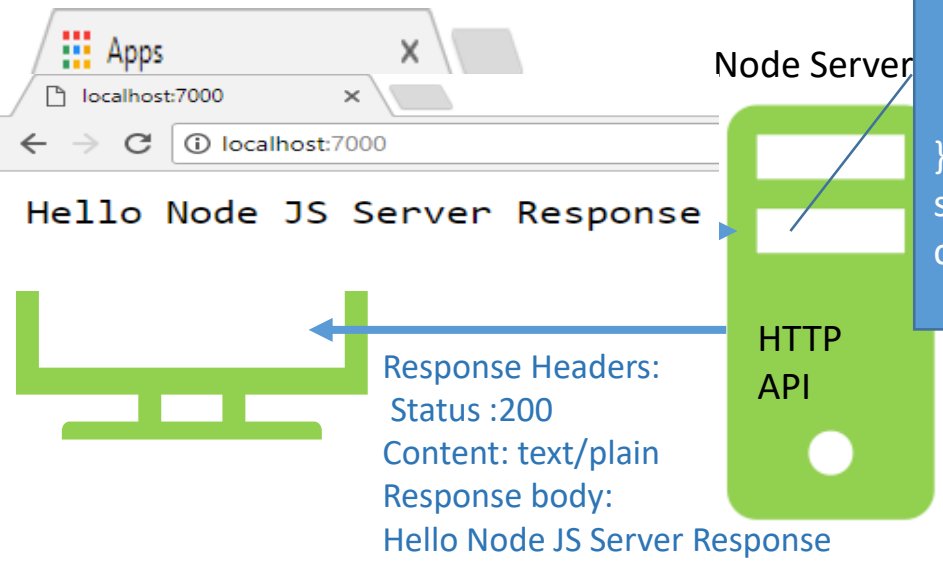
HTTP module

Ships with well-tested functionality to handle network connections effectively

HTTP is a stateless data transfer protocol built upon a request/response model

Clients make requests to servers, which then return a response

Node HTTP Server handle Client Requests




```
var http = require('http');
var server=http.createServer(
    function (request, response) {
     response.writeHead(200, {'Content-type': 'text/plain'});
   response.write("Hello Node JS Server Response");
   response.end();
});
server.listen(7000);
console.log('Server listening at port 7000');
```

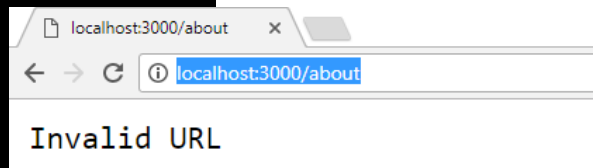
```
$ node web_server.js
  Server listening at port 7000
```

http methods

Method	Description
server = http.createServer([requestListener]);	Returns a new web server object. The requestListener is a function which is automatically added to the 'request' event.
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.write(chunk, [encoding]);	This sends a chunk of the response body. If this method is called and response.writeHead() has not been called, it will switch to implicit header mode and flush the implicit headers.
response.writeHead(statusCode[, statusMessage][, headers])	Sends a response header to the request.
response.end([data], [encoding]);	This method signals to the server that all of the response headers and body have been sent; that server should consider this message complete. The method, response.end(), MUST be called on each response.

Node JS HTTP Server Routings

```
var http = require('http');
var server = http.createServer( function(request,response){
                console.log(request.url);
        if(request.url =='/')
        {       response.write('<h1>Hello from Server</h1>');
            response.end();
        }
        else if(request.url=="/contact")
        {   response.write('<h1>Hello from Server- Contact Page
             response.end();
        }
        else
        {  response.write("Invalid URL");
            response.end();
        }
    });
server .listen(3000);
console.log("Server listening on 3000");
```

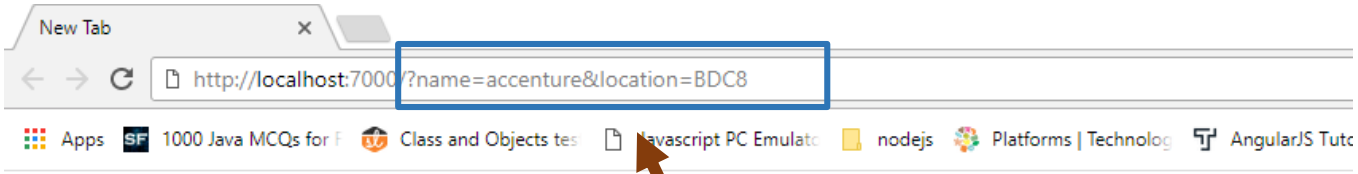


URL module

- The url module is built-in in node and provides utilities for URL resolution and parsing.
- A URL string is a structured string containing multiple meaningful components. When parsed, a URL object is returned containing properties for each of these components.

```
┌──────────────────────────────────────────────────────────────────────────────────────┐
│                                          href                                        │
├──────────┬─┬─────────────────────┬────────────────────┬───────────────────────────┬────┤
│ protocol │ │        auth         │        host        │           path            │hash│
│          │ │                     ├──────────────┬─────┼─────────┬─────────────────┤    │
│          │ │                     │   hostname   │port │pathname │     search      │    │
│          │ │                     │              │     │         ├─┬───────────────┤    │
│          │ │                     │              │     │         │ │    query      │    │
" https:   //    user   :   pass   @ sub.host.com : 8080   /p/a/t/h  ?  query=string   #hash "
│          │ │         │           │   hostname   │port │         │                 │    │
│          │ │         │           ├──────────────┴─────┤         │                 │    │
│ protocol │ │username │ password  │        host        │         │                 │    │
├──────────┴─┴─────────┴───────────┼────────────────────┤         │                 │    │
│   origin   │                     │       origin       │pathname │     search      │hash│
├────────────┴─────────────────────┴────────────────────┴─────────┴─────────────────┴────┤
│                                          href                                        │
└──────────────────────────────────────────────────────────────────────────────────────┘
```

Read the Query String




```
var http = require('http');
var url = require('url');

http.createServer(function(request, response){
  response.writeHead(200, {'Content-type':'text/plan
  response.write('Hello Node JS Server Response');
  response.end( );

  queryString= url.parse(request.url,true).query;
  console.log(queryString);
}).listen(7000);
```

URL module easily split the query string into readable parts

```
$ node http_query_string.js
{ name: 'accenture', location: 'BDC8' }
```

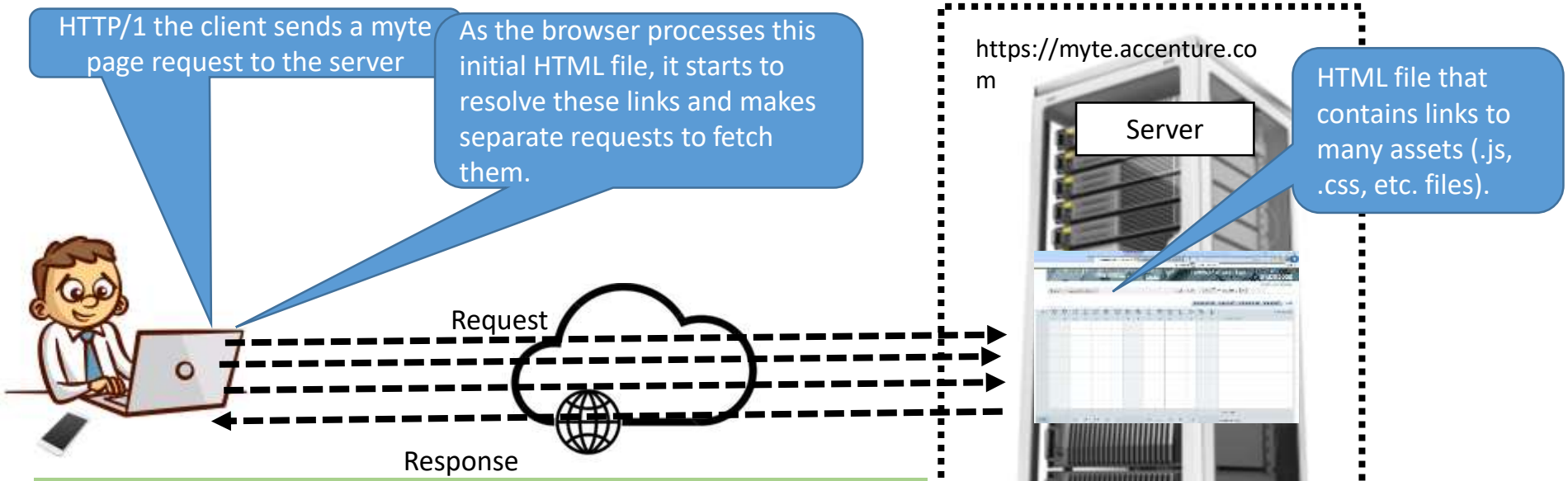
Method takes a URL string, parses it, and returns a URL object.

The query property is returns the query string without the leading ASCII question mark (?) in object format

HTTP/2 - Experimental

- HTTP/2 is the latest evolution of the Hypertext Transfer Protocol (HTTP), used to manage the communication between web servers and browsers.
- HTTP/2 was developed by the IETF's HTTP Working Group, which maintains the HTTP protocol.
- HTTP/2 is a major revision of the HTTP network protocol used by the World Wide Web.
- It was derived from the earlier experimental SPDY protocol, originally developed by Google.
- HTTP/2 is the first new version of HTTP since HTTP 1.1
- HTTP/2 has been redeveloped with this in mind, allowing the browser to multiplex requests. This means that instead of limiting the number of parallel connections, multiple requests can be sent at one time
- The standardization effort was supported by Chrome, Opera, Firefox, Internet Explorer 11, Safari, Amazon Silk, and Edge browsers

Server Push



This is how HTTP/1 works and problem with the current approach is that the user has to wait while the browser parses responses, discovers links and fetches assets. This delays rendering and increases load times. There are workarounds like inlining some assets, but it also makes the initial response bigger and slower.

Server Push

This is where HTTP/2 Server Push capabilities come into the picture as the server can send assets to the browser before it has even asked for them.

served via HTTP/2

the assets were sent immediately together with the initial request.

Request

Response

HTML file that contains links to many assets (.js, .css, etc. files).

HTTP/2 Server Push with Node.js and speed up your client's load time.

Benefits of HTTP/2

All HTTP/1.1 requests have to have headers which are typically duplicate the same info, while H2 forces all HTTP headers to be sent in a compressed format.

Header Compression

Multiplexing

Allows browsers to include multiple requests in a single HTTP connection which in turn enables browsers to request all the assets in parallel.

Servers can push web assets (CSS, JS, images) before a browser knows it needs them which speeds up page load times by reducing number of requests

HTTP/2 Server Push

Stream priority

Allows browsers to specify priority of assets. For example, browser can request HTML first to render it before any styles or JavaScript.
Resources can have dependency levels allowing the server to prioritize which requests to fulfill first

HTTP/2 is a binary protocol making it a lot more efficient when transferring data

Binary

Pipelining

In HTTP/1.1, the server must send responses in the same order the requests were received. HTTP/2 is asynchronous so smaller or faster responses can be handled sooner

Demo (1 of 2)

http2 server

```
const http2 = require('http2’);

const server = http2.createServer();

server.on('stream', (stream, requestHeaders) => {
      stream.respond({ ':status': 200, 'content-type': 'text/plain' });
      stream.end('Hello from node!!!');
});
server.listen(4000);
```

Exposing node to use http2

```
$ node  http2_server.js
(node:121256) ExperimentalWarning: The http2 module is an experimental API.
server is listening
```

Demo (2 of 2)

http2 client

```
const http2 = require('http2');
const client = http2.connect('http://localhost:4000');
const req = client.request({ ':method': 'GET', ':path': '/' });
var message=' ';
req.on('response', (responseHeaders) => {
        console.log(‘Got response..');
});
req.on('data', (chunk) => {
        message=message+chunk;
});
req.on('end', () => {
        console.log(message);
        client.destroy();
});
```

Exposing node to use http2

```
$node   http2_client.js
(node:4480) ExperimentalWarning: The http2 module is an experimental API.
Got response..
 Hello from node!!!
```

NodeJs Frameworks

NodeJS Framework

Node.js frameworks provides higher level of functionality ,a level of abstraction to simplify and speed up construction work.

They extend its core functionality and have built latest features.

They are lightweight and flexible modules to full-stack and highly opinionated frameworks.

Web Framework like http, ftp, APIs allow to use predefined structure and components.

It helps us to build web API that allow to get and save data a backend

Good-to-go frameworks are available as npm packages would be a better option in the real world.

Helps to streamline development of fast websites, rich APIs, and real-time apps

Express

Express - A Minimalist Web Framework

- Express is a popular unopinionated web framework, written in JavaScript and hosted within the node.js runtime environment
- The Express module acts as the webserver in the Node.js-to-AngularJS stack as it runs in Node.js, it is easy to configure, implement, and control.
- Express provides a robust set of features for web and mobile applications
- Express was first released in 2009 by T. J. Holowaychuk
- It's open source, with more than 100 contributors, and is actively developed and supported.
- Express provides a myriad of HTTP utility methods and middleware for creating a robust API quickly and easily

Features of Express

Route management

- Express makes it easy to define routes (URL endpoints) that tie directly to the Node.js script functionality on the server.

Error handling

- Express provides built-in error handling for “document not found” and other errors.

Easy integration

- An Express server can easily be implemented behind an existing reverse proxy system, such as Nginx or Varnish. This allows you to easily integrate it into your existing secured system.

Cookies

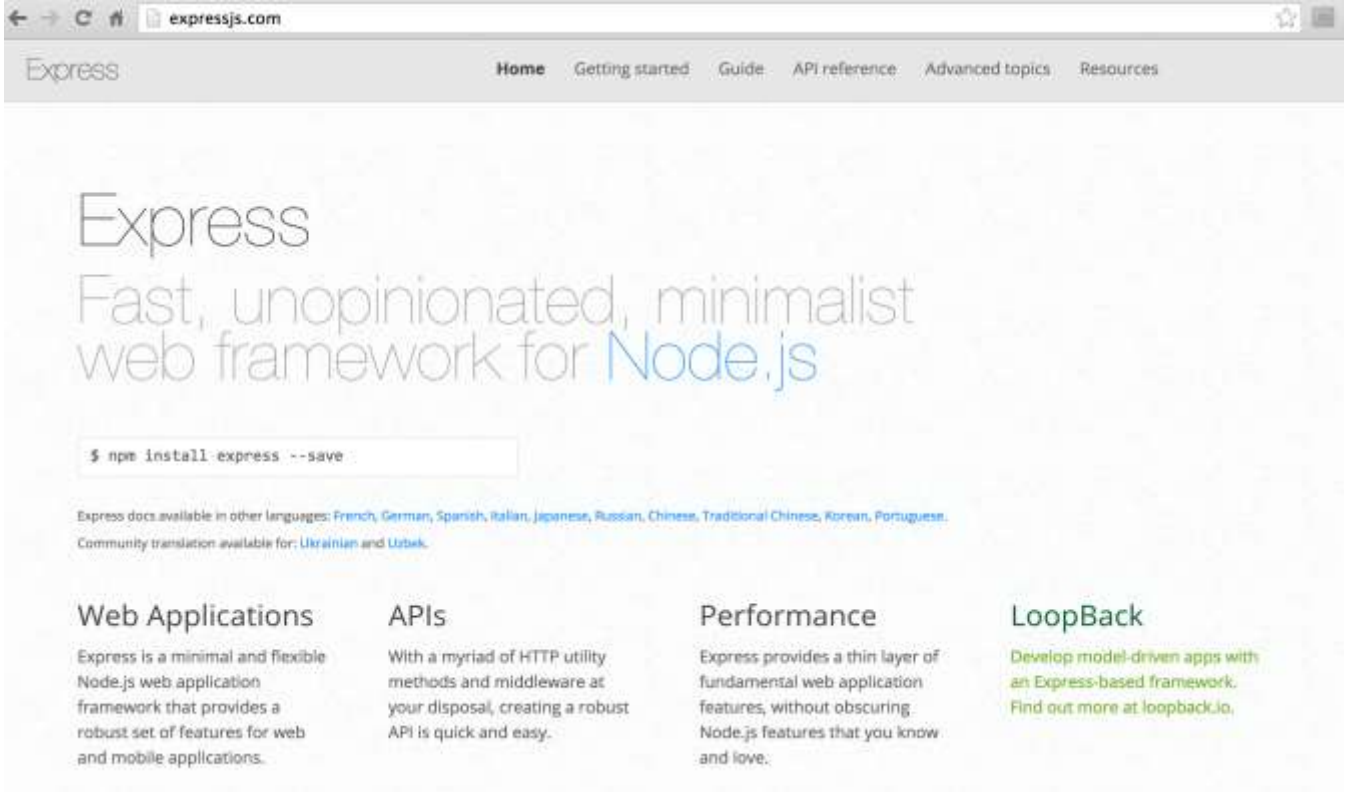
- Express provides easy cookie management

Session and cache management

- Express also enables session management and cache management.

Companies using Express in production

Express official website



Express : Installation

NPM to install ExpressJS

npm install express

For any specific version

npm install express@4.5
npm install express@1.2.3
npm install express@latest

express

Express Hello World code

Expres
s app
code

```
const express = require('express')
const app = express()

app.get('/', (req, res) => res.send('Hello World!'))
app.listen(3000, () => console.log('Example app listening on port 3000!'))
```

Calls function express to initialize object app

This code will be call the function specified when a GET for the URI / is invoked

Run
server

```
$ node express-app.js
Example app listening on port 3000!
```

Sets up the HTTP server for listening port 3000

Test
the
app

Express application methods

Methods	Description
app.get(path, callback [, callback ...])	Routes HTTP GET requests to the specified path with the specified callback functions.
app.post(path, callback [, callback ...])	Routes HTTP POST requests to the specified path with the specified callback functions
app.put(path, callback [, callback ...])	Routes HTTP PUT requests to the specified path with the specified callback functions.
app.delete(path, callback [, callback ...])	Routes HTTP DELETE requests to the specified path with the specified callback functions
app.listen(port, [hostname], [backlog], [callback])	Binds and listens for connections on the specified host and port. This method is identical to Node's http.Server.listen().
app.use([path,] callback [, callback...])	Mounts the specified middleware function or functions at the specified path: the middleware function is executed when the base of the requested path matches path.

Request

- The req object represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers, and so on.
- Request holds the data coming in from client to server
- By convention, the object is always referred to as req but its actual name is determined by the parameters to the callback function.

Properties	Description
req.method	Contains a string corresponding to the HTTP method of the request: GET, POST, PUT, and so on.
req.params	This property is an object containing properties mapped to the named route “parameters”
req.body	Contains key-value pairs of data submitted in the request body. By default, it is undefined, and is populated when you use body-parsing middleware such as body-parser and multer
req.query	This property is an object containing a property for each query string parameter in the route

Response

- The res object represents the HTTP response that an Express app sends when it gets an HTTP request
- Response holds the data sent from server to client.
- By convention, the object is always referred to as res but its actual name is determined by the parameters to the callback function

Response Methods

Methods	Description
res.end([data] [, encoding])	Ends the response process. This method actually comes from Node core, specifically the response.end() method of http.ServerResponse.
res.json([body])	Sends a JSON response. This method sends a response (with the correct content-type) that is the parameter converted to a JSON string using JSON.stringify().
res.redirect([status,] path)	Redirects to the URL derived from the specified path, with specified status, a positive integer that corresponds to an HTTP status code . If not specified, status defaults to “302 “Found”.
res.send([body])	Sends the HTTP response. The body parameter can be a Buffer object, a String, an object, or an Array.
res.sendFile(path [,options] [, fn])	Transfers the file at the given path. Sets the Content-Type response HTTP header field based on the filename’s extension
res.status(code)	Sets the HTTP status for the response

Calling Node's HTTP functions

We can respond from Express using Node's write and end functions

app.js

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

app.get('/', function(request, response) {
  response.write('Hello world');
  response.end();
});

app.listen(3000);
```

using Node API

same thing

response.send('Hello world')

using Express API

Routing

- Routing refers to determining how an application responds to a client request to a particular endpoint, which is a URI (or path) and a specific HTTP request method (GET, POST, and so on).
- Each route can have one or more handler functions, which are executed when the route is matched.
- Route definition takes the following structure:

```
appObj.methodName (path, callback )
```

- Where:
  - app is an instance of express.
  - METHOD is an HTTP request method, in lowercase.
  - PATH is a path on the server.
  - HANDLER is the function executed when the route is matched.

Demo (1of 2)

app.js

```
var express=require('express');
var app=express();
app.get('/',function(request,response){
        response.sendFile(__dirname + '/index.html')
})

app.get('/contactNumbers',function(request,response){
        var phone_list=[908823213,0804523411,6768456222,7621234555];
        response.send(phone_list);
})

app.get('/location',function(request,response){
        var loc={name:"accenture",
                 location:"BDC"};
        response.send(loc);
})
```

serve an index.html file that is stored in current folder

Sending an array to browser

Sending an object to browser and alternatively can use json() metho

Demo (2 of 2)

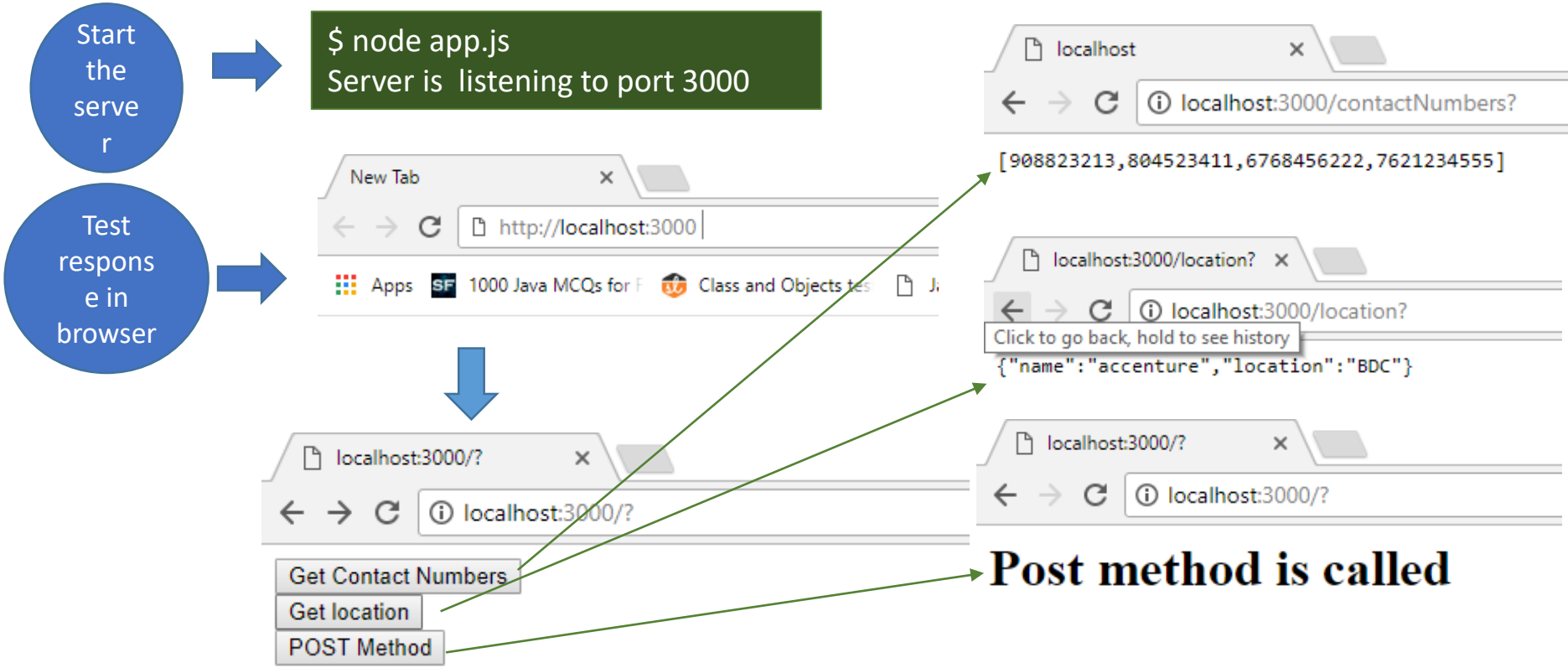
app.js

```
app.post('/',function(request,response) {
        response.send('<h1>Post method is called</h1>');
 })
app.listen(3000,function() {
        console.log('Server is  listening to port 3000');
 })
```

Index
. html

```
<!DOCTYPE html>
<html>
  <body>
     <form>
          <button type="submit" formaction="/contactNumbers" method="get"> Get Contact Numbers
          </button> <br>
          <button type="submit" formaction="/location" method="get"> Get location </button> <br>
    </form>
     <form action="/"  method="post">
          <button type="submit">POST Method</button> <br>
    </form>
</body>   </html>
```

Run & test



Express server responding with array

app.js

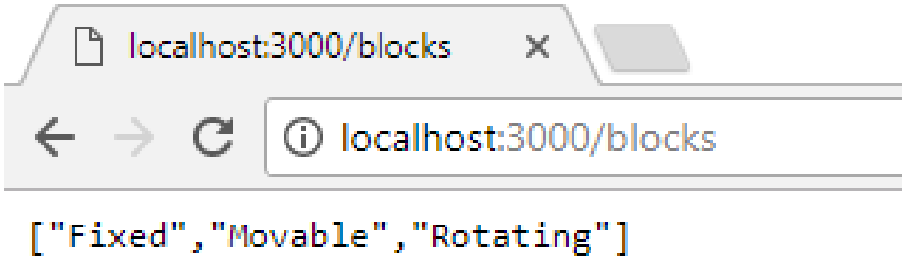
```
var express = require('express');
var app = express();

app.get('/blocks', function(request, response) {
var blocks = ['Fixed', 'Movable', 'Rotating'];
response.json(blocks);
});

app.listen(3000, () => console.log(' server listening on port 3000!')
));
```

To improve efficiency, limit the number of results returned

Always returns all the Blocks

Start the express server

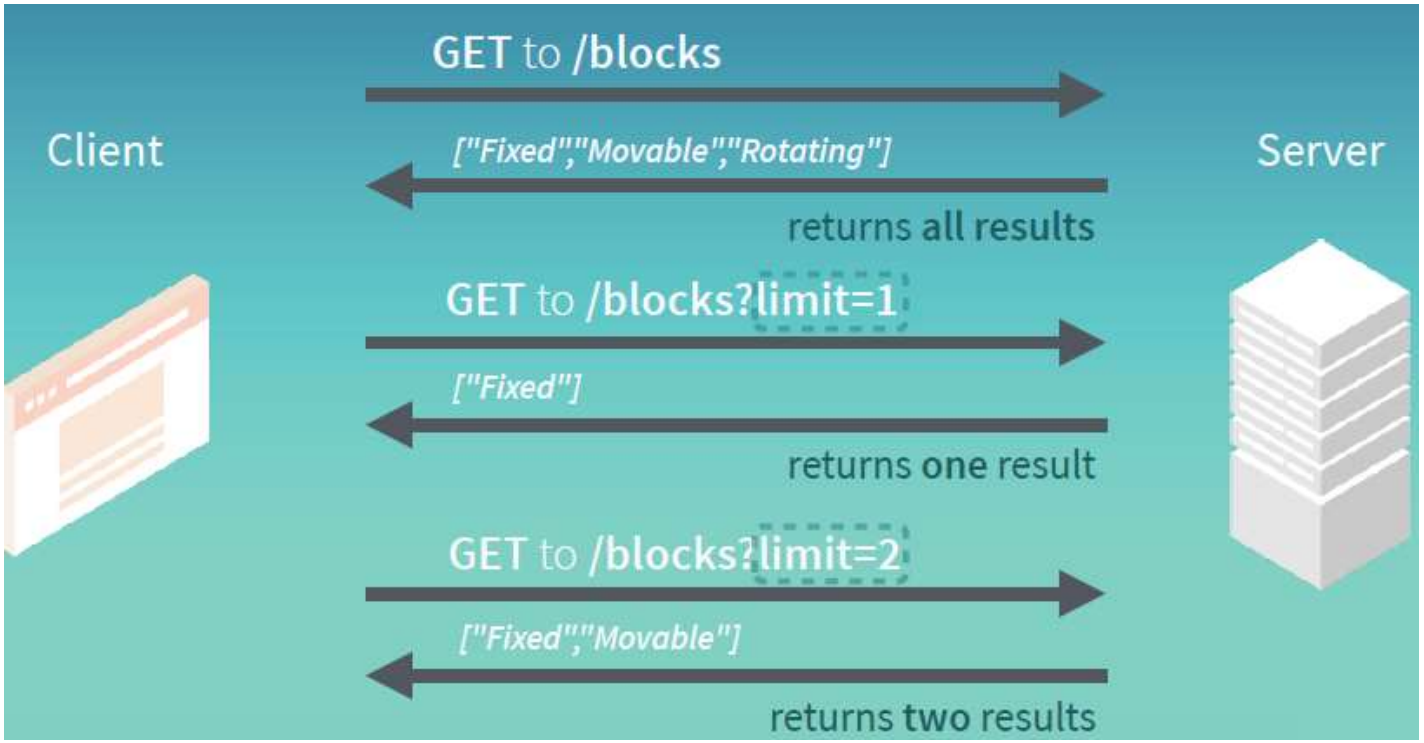
```
$node app.js
Server listening on port 3000!
```



Reading query string and user parameter

Limiting the number of Blocks returned

Query strings are a great way to limit the number of results returned from an endpoint



Reading query string (1 of 2)

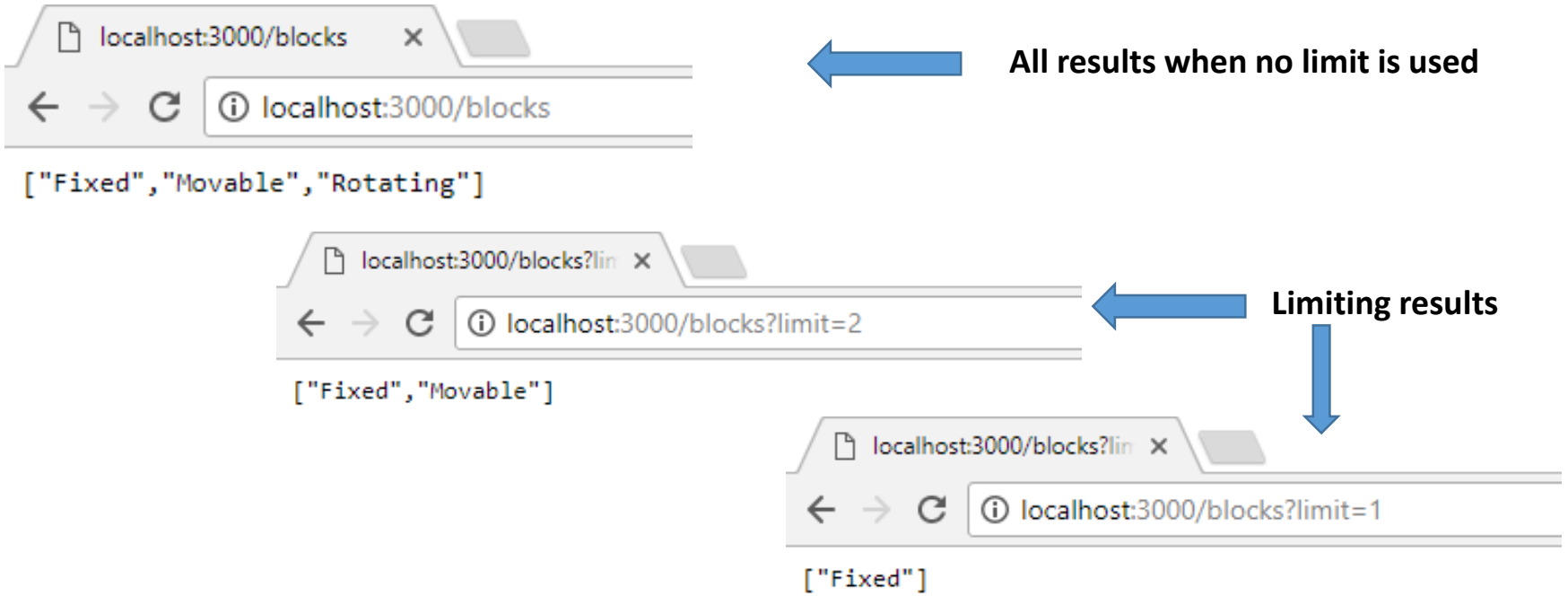
Query string parameters are added to request.query

```
var express = require('express’);
var app = express();

app.get('/blocks', function(request, response) {
        var blocks = ['Fixed', 'Movable', 'Rotating'];
        if(request.query.limit >= 0)    {
            response.json(blocks.slice(0, request.query.limit));
        } else    {
            response.json(blocks);
        }
});
app.listen(3000, () => {
        console.log("Listening on 3000");
} );
```

request.query to access query string named limit

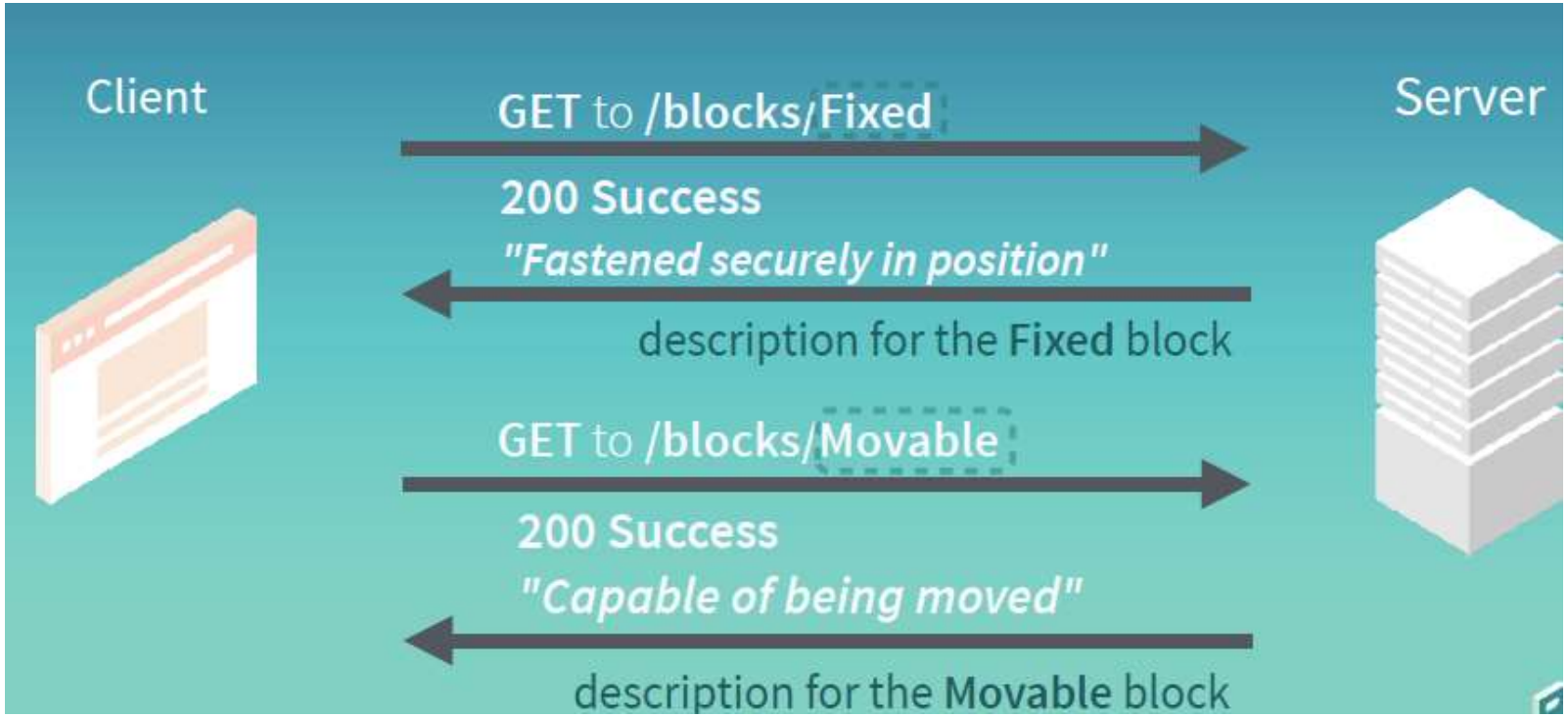
The slice function returns a portion of an Array

Reading query string (2 of 2)



Returning description for a specific Block

We can use meaningful URLs to return the description for specific types of Blocks



Creating Dynamic Routes using user parameters

```
var express = require('express’);
var app = express();
var blocks = {
        'Fixed':'Fastened securely in a position',
        'Movable': 'Capable of moving around’,
        'Rotating': 'Moving in a circle around its center'
        };
app.get('/blocks/:name', function(request, response){
          var description = blocks[request.params.name];
          if(!description)    {
                response.status(404).json('No description found for ' + request.params.name);
           } else     {
                response.send(description);
          }
 });
app.listen(3000 ,() => { console.log("Listening on 3000") } );
```

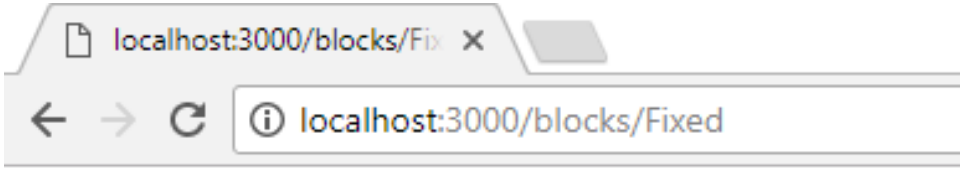
Creates name property on the request.params object

Look up the Block’s description

Sets the 404 Not Found status code if description not found

Defaults to 200 Success status code

Reading user parameters



Fastened securely in a position

Handling valid route

"No description found for Swiping"

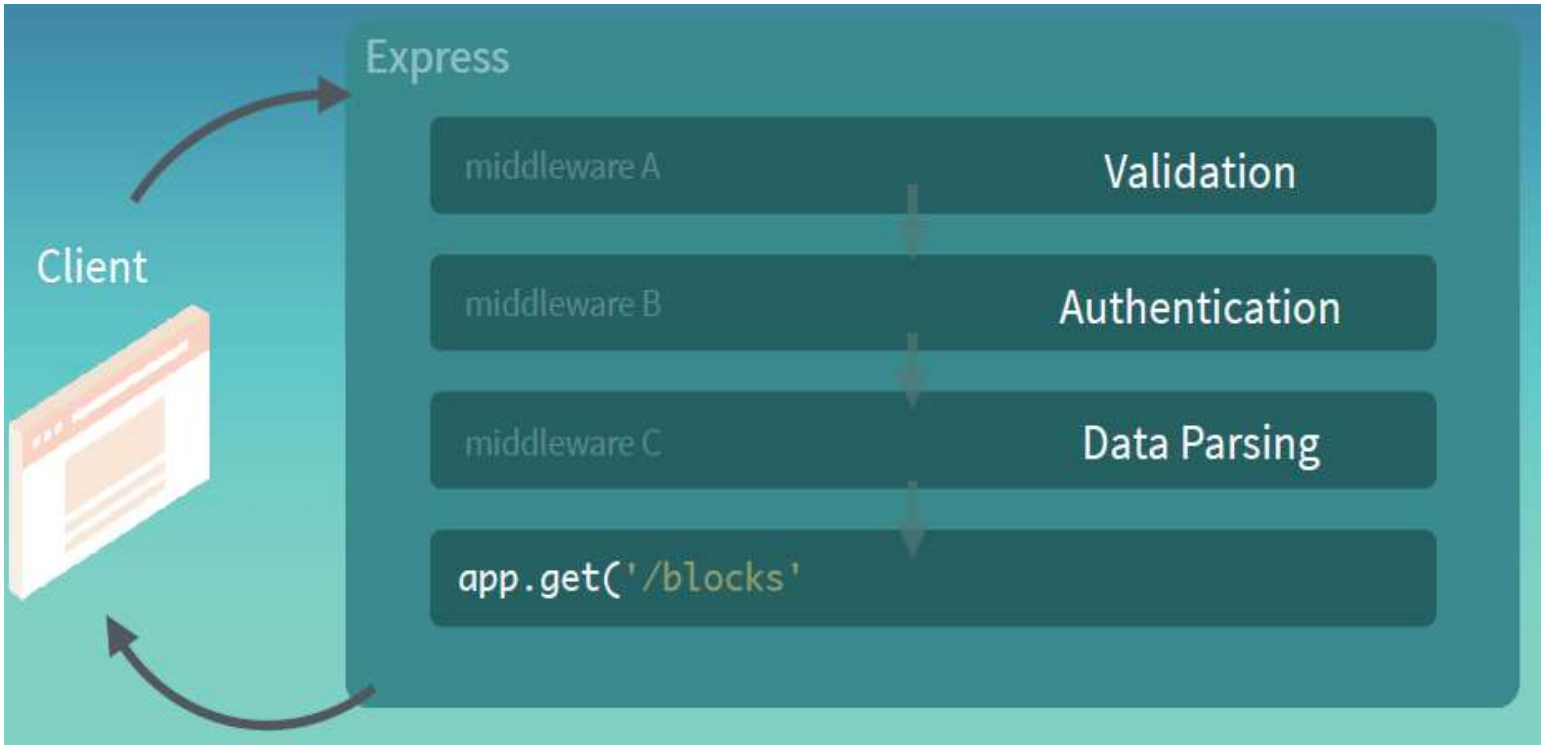
Handling Invalid route

Middleware in Express

- Middleware functions are functions that have access to the request object (req), the response object (res), and the next function in the application's request-response cycle.
- The next function is a function in the Express router which, when invoked, executes the middleware succeeding the current middleware.
- Express is a routing and middleware web framework that has minimal functionality of its own: An Express application is essentially a series of middleware function calls.
- Middleware functions can perform the following tasks:
  - 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 function does not end the request-response cycle, it must call next() to pass control to the next middleware function. Otherwise, the request will be left hanging.

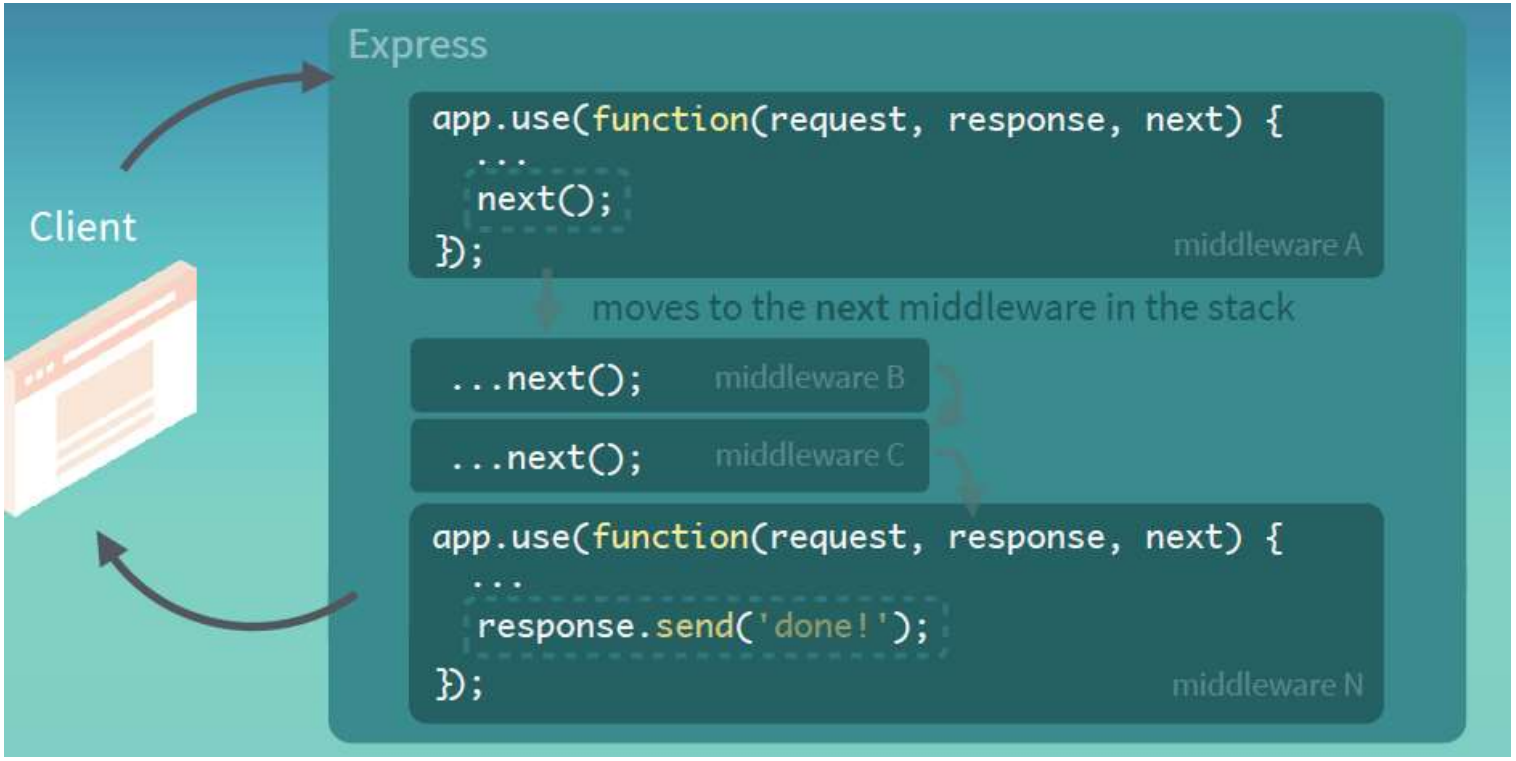
Understanding Middleware

Functions executed sequentially that access request and response



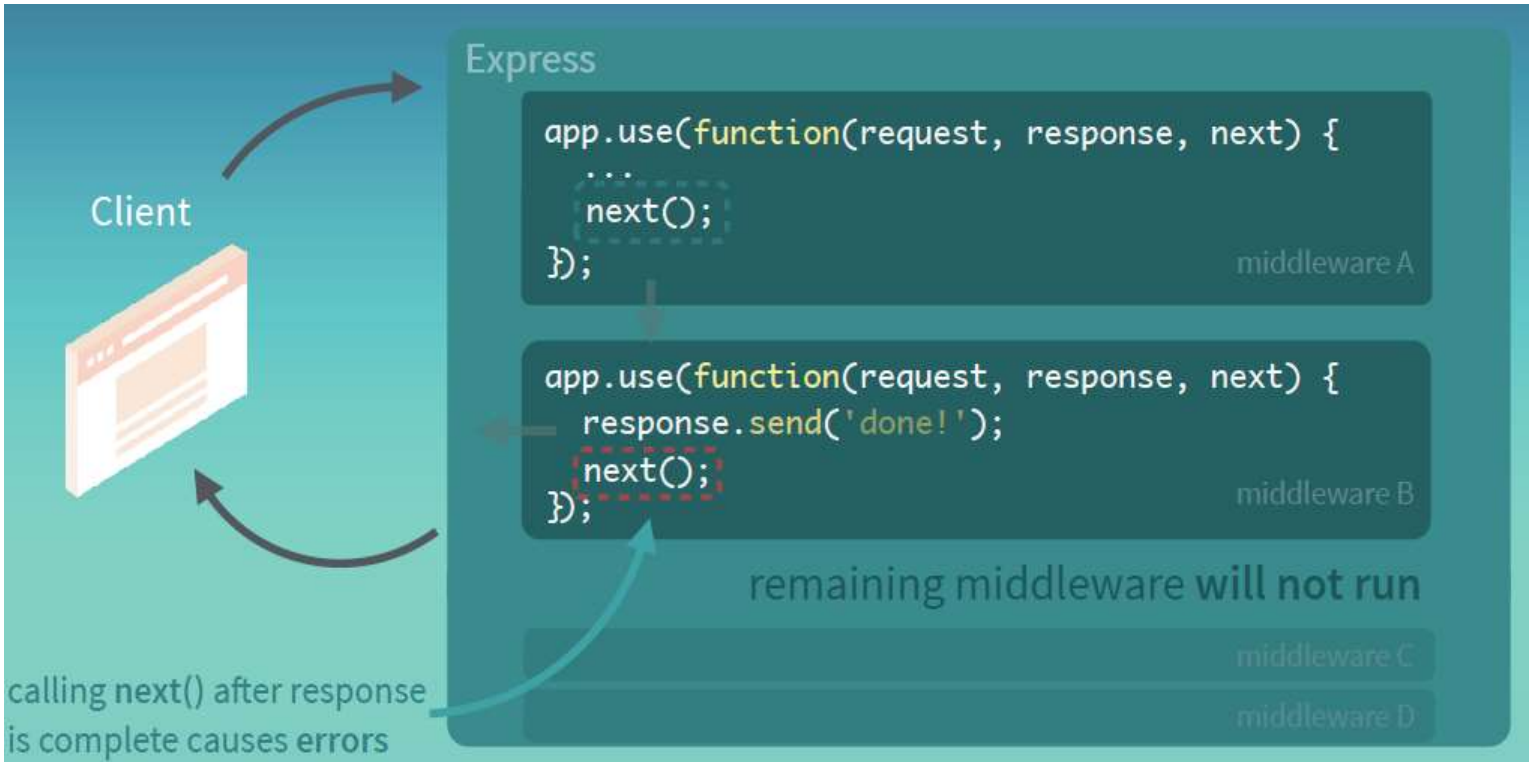
Executing Middleware functions

When next is called, processing moves to the next middleware



Returning from Middleware functions

The flow stops once the response is sent back to the client



Demo - Middleware function requestTime

app.js

```
var express = require('express')
var app = express()

var requestTime = function (req, res, next) {
        req.requestTime = new Date()
        next()
}

app.use(requestTime)

app.get('/', function (req, res) {
        var responseText = 'Hello World!<br>'
        responseText += '<small>Requested at: ' + req.requestTime + '</small>'
        res.send(responseText)
})
app.listen(3000);
```

Middleware function

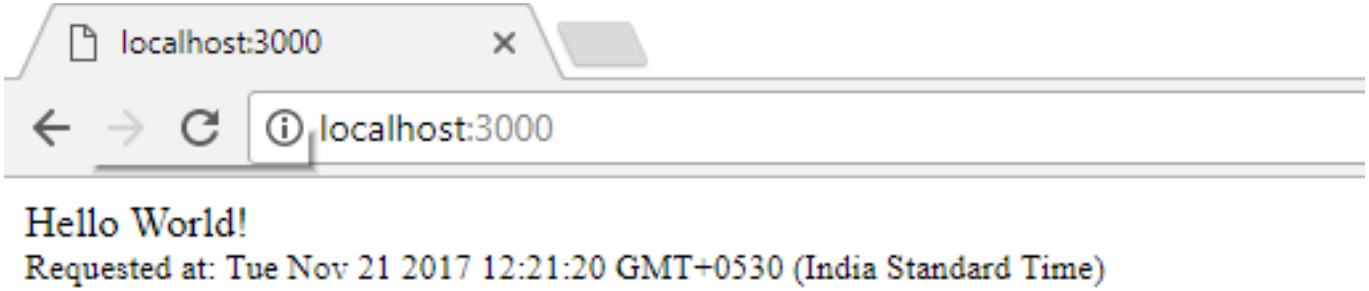
app.use is used to load the middleware function

Demo – start server & test

Start server

```
$ node middleware_function.js
```

Test the app



Serving Static Files

- To serve static files such as images, CSS files, and JavaScript files, use the express.static built-in middleware function in Express.
- Pass the name of the directory that contains the static assets to the express.static middleware function to start serving the files directly.
  - app.use(express.static('public'))
- To use multiple static assets directories, call the express.static middleware function multiple times:
  - app.use(express.static('public'))
  - app.use(express.static('files'))

Demo – code

- Express-static-Demo
  - express-static-demo.js
  - Index.html
  - Node_modules

```
var express = require('express');
var app = express();
app.use(express.static(__dirname));

app.listen(3000);
```

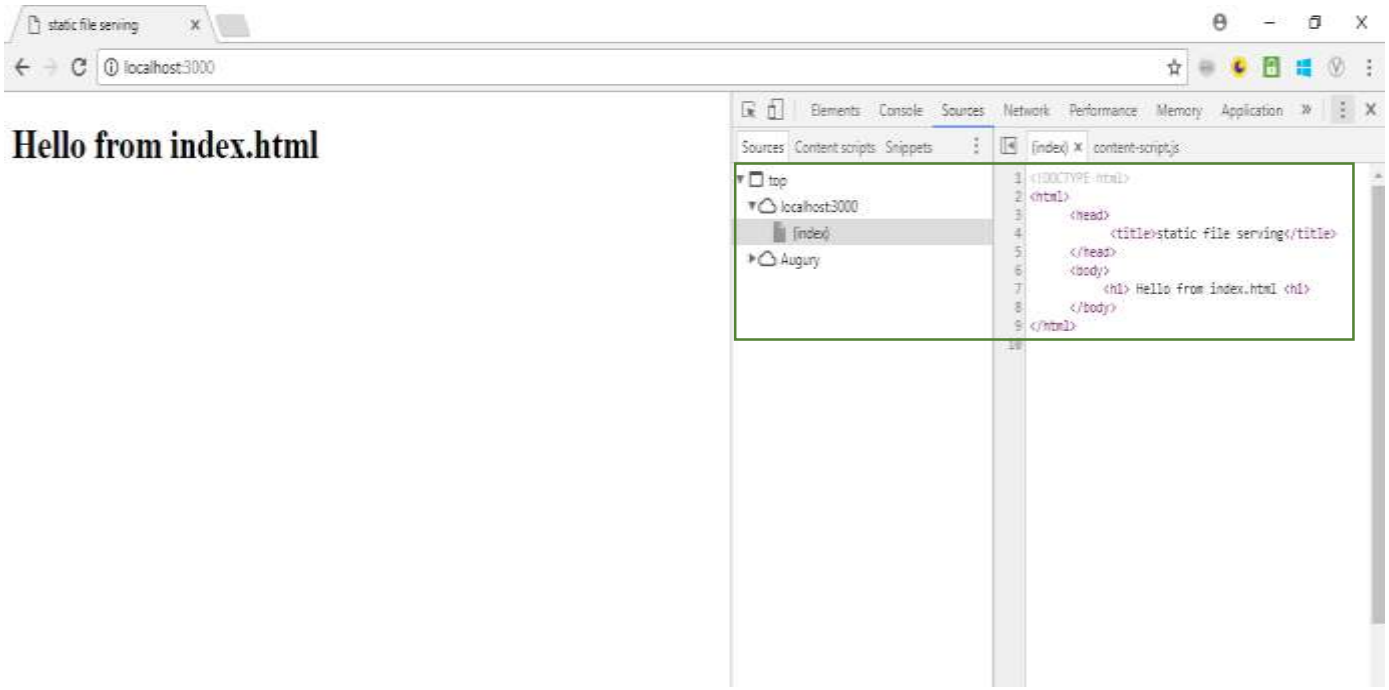
```
<!DOCTYPE html>
<html>
    <head>
        <title>static file serving</title>
    </head>
    <body>
        <h1> Hello from index.html <h1>
    </body>
</html>
```

Demo – run server & test

Run server

```
$ node express-static-demo.js
```

Test the app



Serving static assets

The static middleware serves everything under the specified folder

- Express-static-assets
  - app.js
  - public
    - Index.html
    - Accenture.png
  - Node_modules

```
var express = require('express');
var app = express();
app.use(express.static('public'));

app.listen(3000);
```

```
<!DOCTYPE html>
 <html>
  <head>
    <title> Express Middleware </title>
</head>
<body>
<h1>Hello World !</h1>
<p><img src='Accenture.png'></p>
</body>
</html>
```

Server to handling data sent by POST request


```
var express = require('express')
var app = express();
var bodyParser = require('body-parser');
var urlencodedParser = bodyParser.urlencoded ({ extended: false })

app.use(function(req, res, next) {
      res.header("Access-Control-Allow-Origin",
      res.header("Access-Control-Allow-Headers", "Origin
      Content-Type, Accept");
      next();
});

app.post('/quotes', urlencodedParser,function(request, response) {
       console.log('Name :'+request.body.name);
       console.log('quote :'+request.body.quote);
       response.send('Thank you for submitting quote');
});

app.listen(3000,()=> { console.log("Server listening at 3000");  })


// Content-Type : application/x-www-form-urlencoded
```

Middleware to parse HTTP request body

body object containing the parsed data is populated on the request object after the middleware

Data persistence using mongodb

NoSQL

- A **NoSQL** originally referring to "non SQL" or "non relational“
- A mechanism for storage and retrieval of data that is modeled in means other than the tabular relations used in relational databases
- NoSQL databases especially target large sets of distributed data.
- A very flexible and schema-less data model.

Types of NoSQL Databases

Key -Value

redis

riak

Graph Store

Neo4j the graph database

HyperGraphDB

Document based

mongoDB

CouchDB relax

Wide-Column

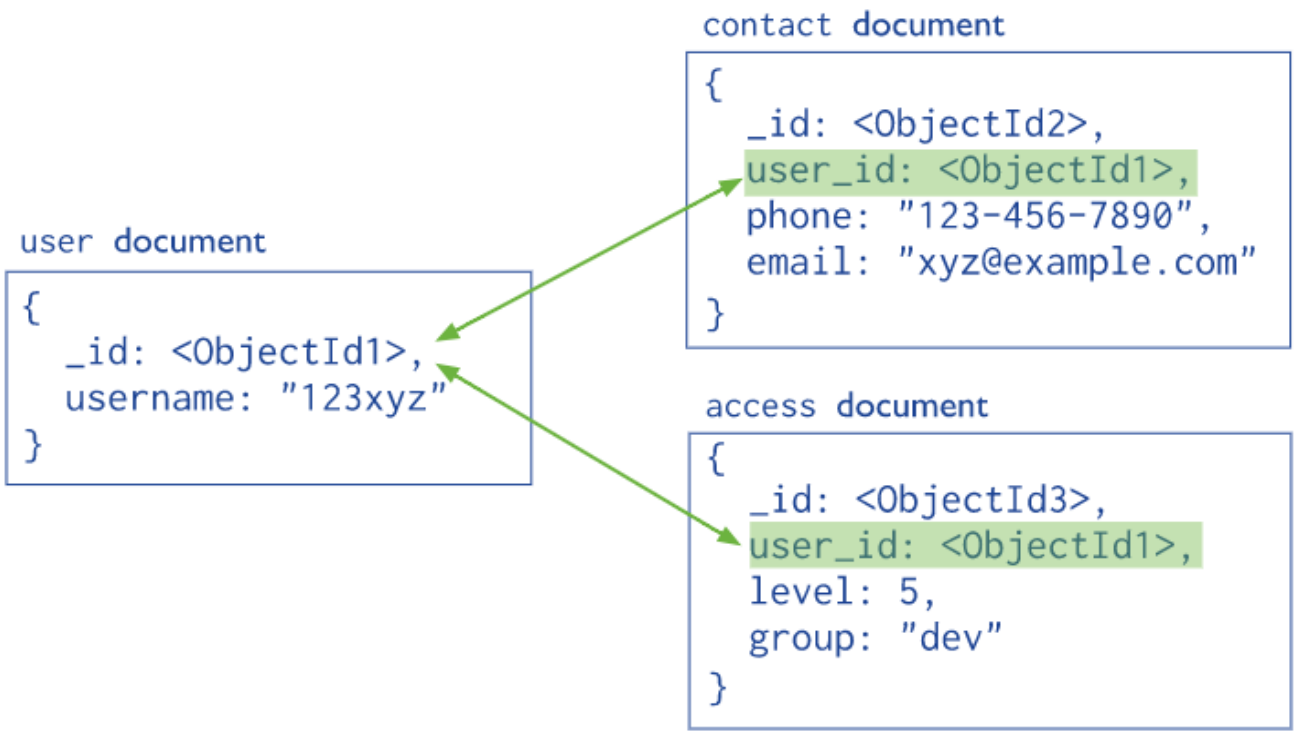
HBASE

Cassandra

Document database

- A most popular ways of storing data is a document data model.
- A document database is designed to store semi-structured data as documents, typically in JSON.
- A document is a set of key-value pairs.
- JSON document support makes it easier for Developers to serialize and load objects containing relevant properties and data

Document database



Introduction to MongoDB

- Popular free and open-source cross-platform document-oriented database
- MongoDB is developed by MongoDB Inc
- First version was release on February 11 2009.
- Written in C++, C and JavaScript
- Provides high performance, high availability, and automatic scaling.
- Full index support

Features of MongoDB

MongoDB stores data in flexible, JSON-like documents, meaning fields can vary from document to document and data structure can be changed over time

MongoDB works on concept of collection and document.

MongoDB is a distributed database at its core, so high availability, horizontal scaling, and geographic distribution are built in and easy to use.

Data are stored in BSON format.

Companies using MongoDB

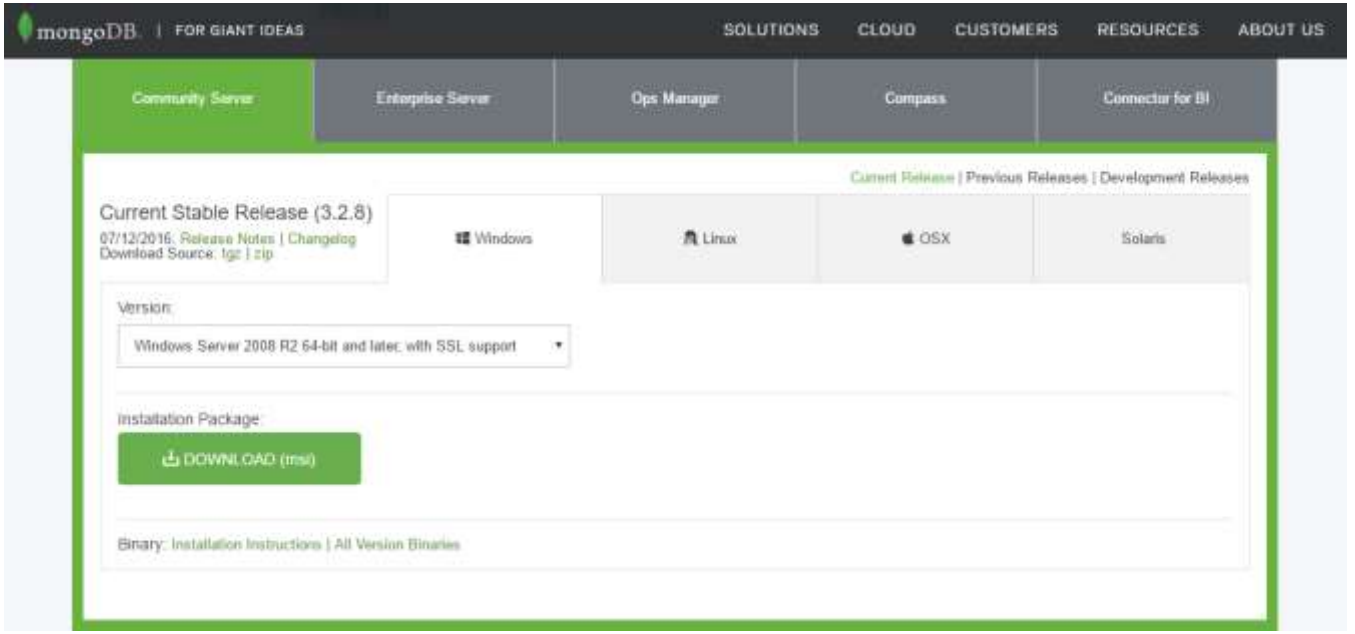
Document structure

- MongoDB documents are similar to JSON objects.
- BSON is a binary representation of JSON documents
- Data structure composed of field and value pairs.

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

Download

https://www.mongodb.com/download-center#community



MongoDB: Setup MongoDB Environment

MongoDB requires a data directory. Default is \data\db

- Create the data Directory

Terminal

```
C:\>md data\db
```

- Start MongoDB

Terminal

```
C:\>"C:\Program Files\MongoDB\Server\3.2\bin\mongod.exe"
```

The mongo Shell

- The mongo shell is an interactive JavaScript interface to MongoDB.
- It is used to query and update data as well as perform administrative operations.
- It is a component of the MongoDB distributions
- Start the mongo shell at command prompt

```
C:\Program Files\MongoDB\Server\3.2\bin\mongo.exe
```

- Type quit() or use the <Ctrl-C> shortcut.

Shell commands

```
C:\Program Files\MongoDB\Server\3.2\bin\mongo.exe
MongoDB shell version: 3.2.8
connecting to: test
MongoDB Enterprise > show databases
MyDb        0.000GB
exampleDb   0.000GB
local       0.000GB
mongochat   0.000GB
sampledb    0.000GB
MongoDB Enterprise > use local
switched to db local
MongoDB Enterprise > show collections
mytables
startup_log
MongoDB Enterprise >
```

Databases in MongoDB

- A single MongoDB server typically has multiple databases.
- Default database of MongoDB is 'db', which is stored within data folder.
- MongoDB can create databases on the fly.
- It is not required to create a database before start working with it.

Databases command

"show dbs" / "show databases" command provides with a list of all the databases.

```
D:\mongodb\bin>mongo
MongoDB shell version: 1.8.1
connecting to: test
> show dbs
admin    (empty)
comedy   0.03125GB
local    (empty)
student  0.03125GB
test     0.03125GB
> _
```

Run 'db' command to show to the current database object or connection.

```
D:\mongodb\bin>mongo
MongoDB shell version: 1.8.1
connecting to: test
> db
test
> _
```

To connect to a particular database, run use command

```
> db
test
> use student
switched to db student
>
```

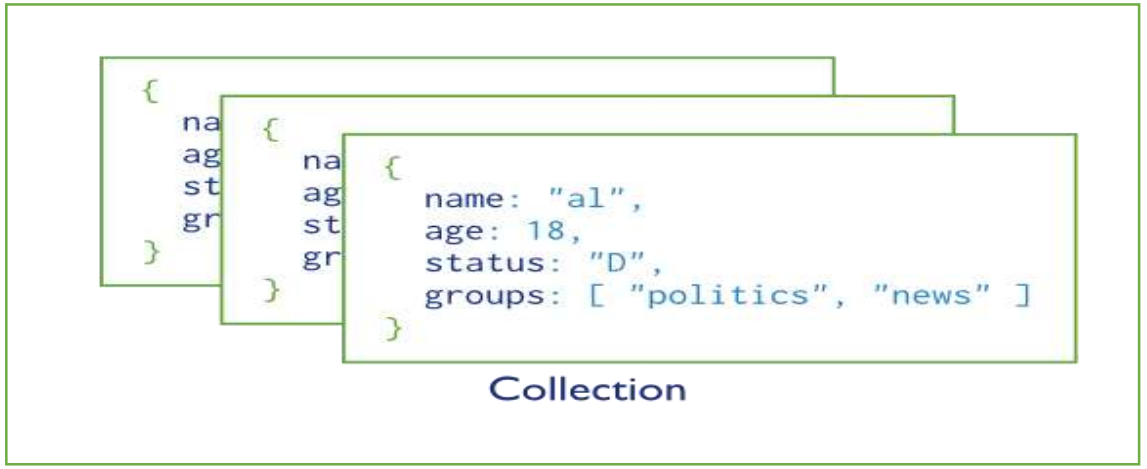
Document

- *The document* is the unit of storing data in a MongoDB database.
- *Document* use JSON style for storing data.
- Often, the term "object" is used to refer a document.
- Documents are analogous to the records of an RDBMS.
- Insert, update, and delete operations can be performed on a collection.

```
var mydoc = {
        _id: ObjectId("5099803df3f4948bd2f98391"),
        name: { first: "Alan", last: "Turing" },
        birth: new Date('Jun 23, 1912'),
        death: new Date('Jun 07, 1954'),
        contribs: [ "Turing machine", "Turing test",
"Turingery" ],
        views : NumberLong(1250000)
      }
```

Collections

- MongoDB databases hold collections of documents.
- A *collection* is analogous to a table of an RDBMS.
- A *collection* may store documents those who are not same in structure.
- MongoDB stores BSON documents, i.e. data records, in collections
- A collection exists within a single database



Creation of Collection

- If a collection does not exist, MongoDB creates the collection when first store data for that collection.
- **Implicit creation :**

```
db.myNewCollection2.insertOne( { x: 1 } )
db.myNewCollection3.createIndex( { y: 1 } )
```

- **Explicit Creation :**

```
db.createCollection("name")
```

Demo

C:\Program Files\MongoDB\Server\3.2\bin\mongo.exe

```
MongoDB Enterprise > use mydatabase
switched to db mydatabase
MongoDB Enterprise > db
mydatabase
MongoDB Enterprise > db.createCollection("Employee");
{ "ok" : 1 }
MongoDB Enterprise > show collections
Employee
MongoDB Enterprise > db.Department.insert({"dept_id":"10","name":"LKM"});
WriteResult({ "nInserted" : 1 })
MongoDB Enterprise > show collections
Department
Employee
MongoDB Enterprise >
```

RDBMS VS MongoDB

RDBMS	MongoDB
Database	Database
Table, View	Collection
Row/Record	Document
Column	Field
Index	Index
Join	Embedded Document
Primary Key	_id field is always the primary key
Aggregation	aggregation pipeline

Drop Collection

```
db.collection_name.drop()
```

- Removes a collection or view from the database
- Returns:
  - true when successfully drops a collection.
  - false when collection to drop does not exist.

Insert Document

- Create or insert operations add new documents to a collection.
- If the collection does not currently exist, insert operations will create the collection.
- Methods to insert documents into a collection:

db.collection.insertOne ({ key:value })	Inserts a single document into a collection.
db.collection.insertMany()	db.collection.insertMany() inserts multiple documents into a collection.
db.collection.insert()	db.collection.insert() inserts a single document or multiple documents into a collection.

_id Field

1. The **_id field** is the default **field** for Bson ObjectId's.
2. If the document does not specify an _id field, then mongod will add the _id field and assign a unique ObjectId for the document before inserting.
3. Most drivers create an ObjectId and insert the _id field, but the mongod will create and populate the _id if the driver or application does not.
4. If the document contains an _id field, the _id value must be unique within the collection to avoid duplicate key error.

Insertion structure

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

} document

Examples

Without Specifying an _id Field

```
db.products.insertOne(  {   item: "card”  , qty:15   }  );
```

Specifying an _id Field

```
db.products.insertOne(  {   _id:10,   item: "box",   qty:20
  }  );
```

Several Document

```
db.products.insertMany(
      [
        {item:"card", qty:15},
        {item:"envelope", qty:20},
        {item:"stamps", qty:30}
]  );
```

Examples

Perform an Unordered Insert

```
db.products.insert(
 [
    { _id:20, item:"lamp", qty:50, type:"desk"},
    { _id:21, item:"lamp", qty:20, type:"floor"},
    { _id:22, item:"book",  qty:100 }
  ],
   {ordered:false}
  );
```

Finding the Document

- Selects documents in a collection or view and returns a cursor to the selected documents.
- Methods to read documents from a collection:

```
db.collection.find(query, projection)
```

Find method structure

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

← collection
← query criteria
← projection
← cursor modifier

Demo

Find All

```
db.products.find()
```

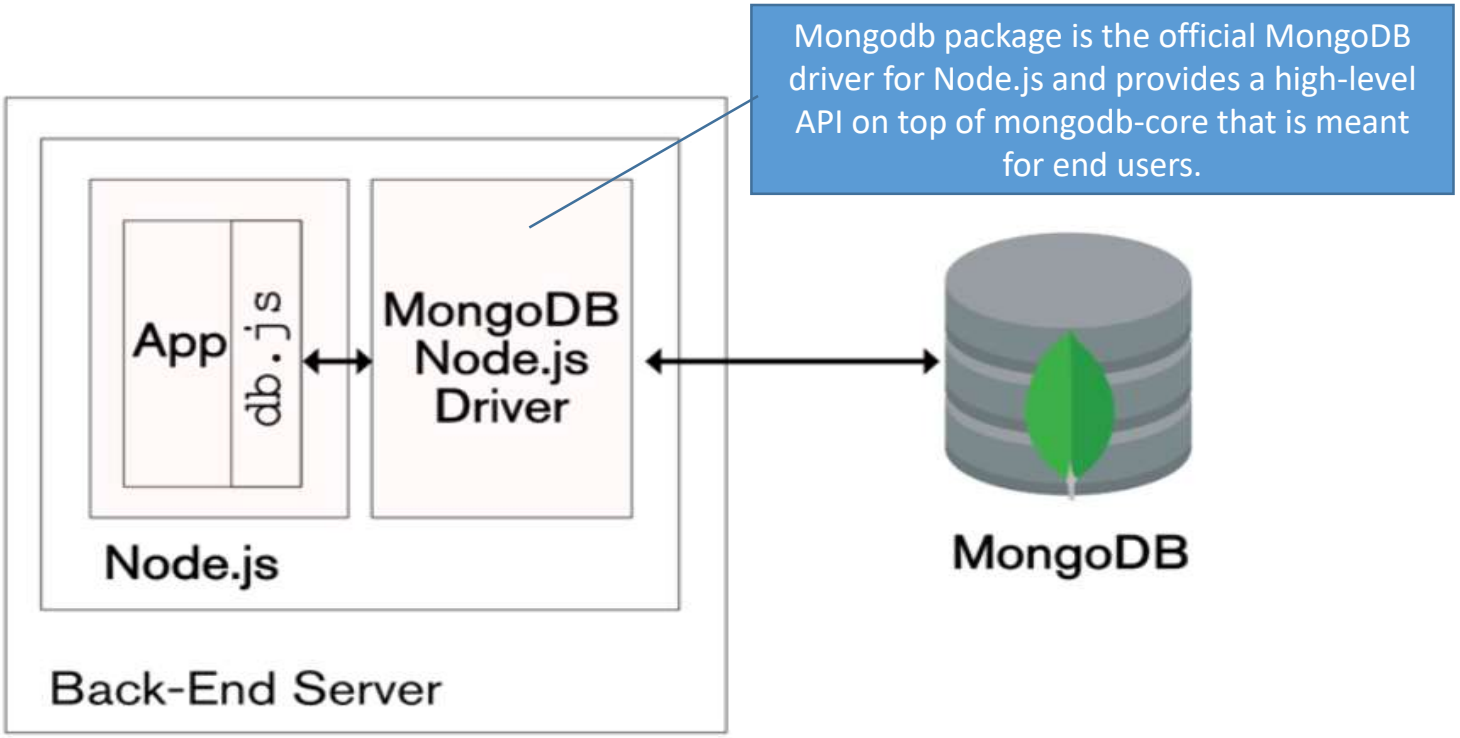
selection

```
db.products.find( {   qty:  {  $gt:15
   } }  )
```

Projection

```
db.products.find(
     {   qty:  {  $gt:15  }  },
          {item:1,qty:1   }
)
```

Architecture



Installing MongoDB driver for node

- The official mongodb driver for node

```
$npm install mongodb
```

- Once installed, the MongoDB must be started
  - Go to installation folder and run **mongod --dbpath=/data**
- When run, the MongoDB can be used from Node.js

Steps: Node connecting with Mongodb

Import mongodb package using node's **require** method

Create a mongodb **client**

Specify a running mongod instance using an **uri**

Connect client to running DB 'test'

Successful connection will print console log.

```
var mongodb =require('mongo
```

```
var mongoclient = mongodb.Mongocli
```

```
var url = 'mongodb://localhost:27017/test'
```

```
mongoclient.Connect(url, function(err, client) { });
```

Node connection with mongodb native driver

```
var mongodb = require('mongodb');
var mongoClient = mongodb.MongoClient;
const url = 'mongodb://localhost:27017’;
const dbName = 'myproject’;

mongoClient.connect(url, function (err, client) {
    if (err) {
      console.log('Unable to connect to the mongoDB server. Error: ', err);
    } else {
      console.log(‘Connected successfully to server’);
      const db = client.db(dbName);
      client.close();
   }
} );
```

import the MongoDB native drivers

"MongoClient" interface in order to connect to a MongoDB server

Connection URL. This is where MongoDB server is running.

Connect method to connect to the Server

Inserting a single Document

```
mongoClient.connect(url,function(err,client){
    if(err){
        console.log("error connecting to database :"+err);
      } else {
        console.log("Connected to database");
        const db = client.db(dbName);
        db.collection("users").insertOne({
            name : "Mac",
            age : 25,
            location : "Pune"
          });
    client.close();
  }
});
```

inserts a single document into a collection

Inserting a multiple Document

```
mongoClient.connect(url,function(err,client){
    if(err){
        console.log("error connecting to database :"+err);
      } else {
            const db = client.db(dbName);
            var user1 = {name:"Anitha",age:"30",location:"Pune"};
            var user2 = {name:"Amith",age:"45",location:"Noida"};
            var user3 = {name:"Arun",age:"26",location:"Bengaluru"};
     db.collection("users").insert([user1,user2,user3],function(err,result) {
          if(err){
            console.log("Error occured "+err);
          }else{
            console.log("3 documents inserted");
           }
      });
     client.close();
  }  });
```

A document or array of documents to insert into the collection.

Updating single document

```
mongoClient.connect(url,function(err,client){
    if(err){
        console.log("error connecting to database :"+err);
      }else{
        console.log("Connected to database");
        const db = client.db(dbName);
        db.collection("users").updateOne(
                {name:“Amith"},
                {$set:{name:“Amith Kumar"}
            });
      client.close();
    }
});
```

Updates a single document within the collection based on the filter..

Updating Multiple document

```
mongoClient.connect(url,function(err,client){
    if(err){
        console.log("error connecting to database :"+err);
      }else{
        const db = client.db(dbName);
        db.collection("users").updateMany({name:“Arun"},{$set:{location:“Chennai"}},
                                function(err,number){
                                    if(err){
                                        console.log("Error occured...");
                                    }else if(number.result.n){
                                        console.log(number.result.n+ " documents updated");
                                    }else{
                                        console.log("No document found with give criteria");
                                    }    });
      client.close();
    }  });
```

Updates multiple documents within the collection based on the filter

Deleting a document

```
mongoClient.connect(url,function(err,client){
    if(err){
        console.log("error connecting to database :"+err);
      } else {
        const db = client.db(dbName);
        db.collection("users").deleteOne({name:“Arun"});
      client.close();
    }
 });
```

Removes a single document from a collection.

Finding Documents

```
mongoClient.connect(url,function(err,client){
    if(err){
        console.log("error connecting to database :"+err);
      }else{
            const db = client.db(dbName);
             db.collection("users").find({}).sort({name:1}).toArray(function(err, result) {
                if (err) {
                    console.log(err);
                  } else {
                console.log(result);
              }
          });
     client.close();
  }  });
```

Find all user and sorting the documents according to the name in ascending order and returning as an array