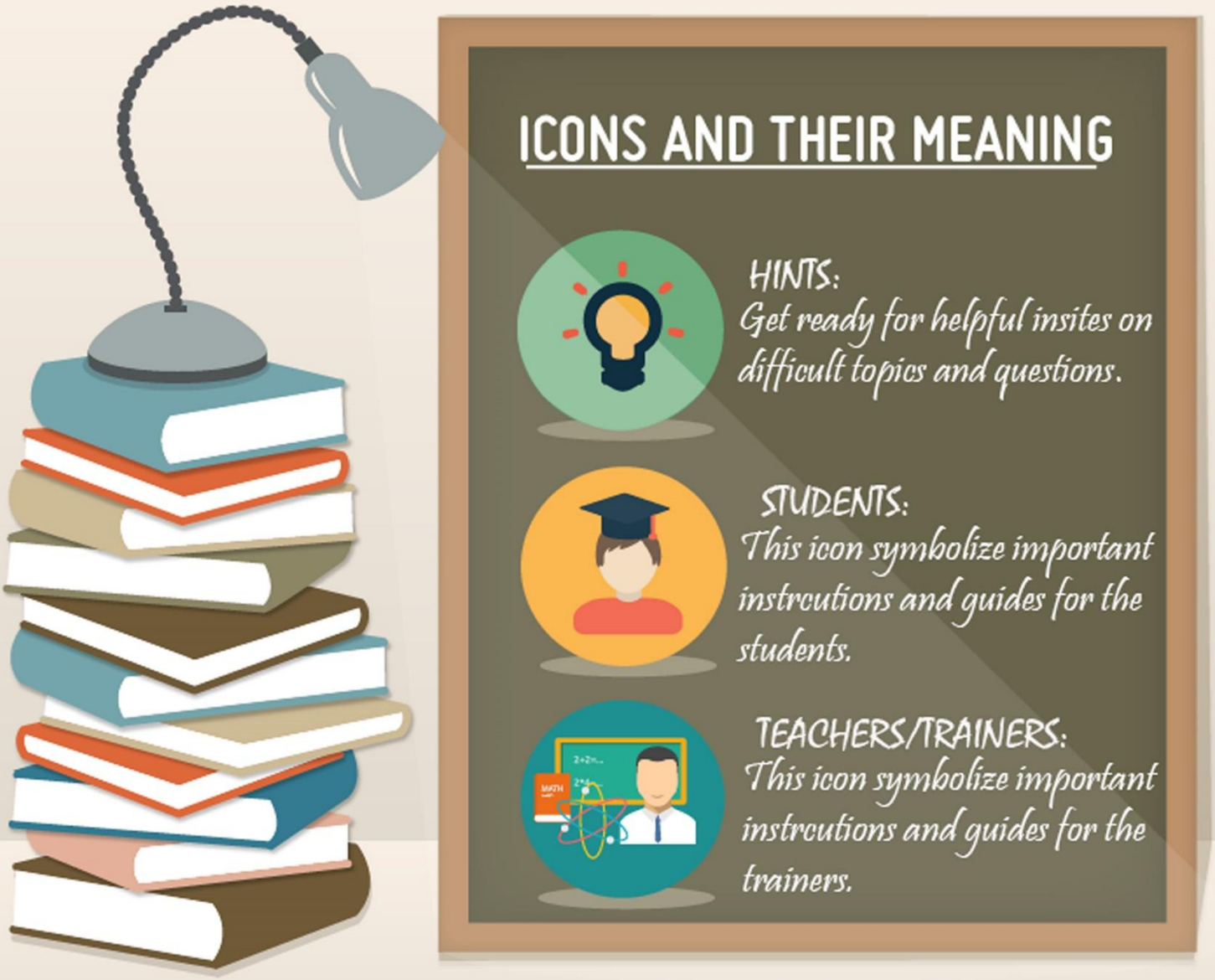
Full Stack Module-VI

Manual V8.3

ANUDIP FOUNDATION


ICONS AND THEIR MEANING
HINTS:
Get ready for helpful insites on difficult topics and questions.
STUDENTS:
This icon symbolize important instrcutions and guides for the students.
TEACHERS/TRAINERS:
This icon symbolize important instrcutions and guides for the trainers.

Total Duration: __ Hours

Lesson 01: Getting Started with Node.js (120 minutes)

Objective: After completing this lesson you will be able to learn about : • Introduction • Purpose • Installation	**Materials Required:** • Computer With Windows XP and above • Stable Internet connection
Self- Learning Duration: 120 minutes	**Practical Duration:** nil
Total Duration: 120 minutes	

Introduction

NodeJS is an open-source, cross-platform runtime environment for developing server-side web applications. NodeJS also has an event-driven architecture capable of asynchronous I/O.

NodeJS uses an event-driven, non-blocking I/O model that makes it lightweight and efficient.

Purpose of Node.Js

Node.js is a platform built on Chrome's JavaScript runtime for easily building fast and scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.

It also provides a rich library of various JavaScript modules which simplifies the development of web applications using Node.js to a great extent.

Features of Node.Js

These are the most important features of Node.JS:

- Extremely fast
- No buffering
- Highly scalable but single threaded
- Event driven and asynchronous

Where to use and not to use Node.Js

Areas where Node.JS can be used:

- I/O bound Applications
- Data Streaming Applications
- Data Intensive Real-time Applications (DIRT)
- JSON APIs based Applications
- Single Page Applications

I/O bound applications

A program is I/O bound if it would go faster if the I/O subsystem was faster. Which exact I/O system is meant can vary; you typically associate it with disk, but of course networking or communication in general is common too. A program that looks through a huge file for some data might become I/O bound, since the bottleneck is then the reading of the data from disk (actually, this example is perhaps kind of old-fashioned these days with hundreds of MB/s coming in from SSDs).

In simpler terms, I/O Bound means the rate at which a process progresses is limited by the speed of the I/O subsystem. A task that processes data from disk, for example, counting the number of lines in a file is likely to be I/O bound.

Data Streaming Applications

Data streaming applications or services are applications that send data as it arrives or as it is created, or even data in existing files, directly to a client service or application such as a web browser like Chrome, or the apps on a Roku video player, or mobile apps on an iPhone or Android phone.

One good example of this are for stock broker trading applications. As stocks are traded at a stock exchange, the prices of the stocks change. A streaming application can take those prices and send them in real time to customers or to other application that use the data for further processing - such as software that will watch the prices and create an alert if the price goes higher or lower than a certain point.

Data Intensive Real-time Applications (DIRT)

Data-intensive applications not only deal with huge volumes of data but, very often, also exhibit compute-intensive properties. Data-intensive applications handle datasets on the scale of multiple terabytes and petabytes. Datasets are commonly persisted in several formats and distributed across different locations. Such applications process data in multistep analytical pipelines, including transformation and fusion stages.

The processing requirements scale almost linearly with the data size, and they can be easily processed in parallel. They also need efficient mechanisms for data management, filtering and fusion, and efficient querying and distribution. It is totally an event-driven non-blocking I/O model. Its performance on distributed devices is impeccable for its lightweight and efficient architecture. Notably, Node.js is an open source, a cross-platform environment that runs across OS X, Microsoft Windows, and Linux. Developers use it for server-side and network application development.

JSON APIs based Applications

JSON or JavaScript Object Notation is an encoding scheme that is designed to eliminate the need for an ad-hoc code for each application to communicate with servers that communicate in a defined way. JSON API module exposes an implementation for data stores and data structures, such as entity types, bundles, and fields.

Single Page Applications

A single-page application (SPA) is a website design approach where each new page's content is served not from loading new HTML pages but generated dynamically through JavaScript's ability to manipulate the DOM elements on the existing page itself.

In a more traditional web page architecture, an index.html page might link to other HTML pages on the server that the browser will download and display from scratch. An SPA approach allows the user to continue consuming and interacting with the page while new elements are being updated or fetched, and can result in much faster interactions and content reloading.

Areas where Node.JS must not be used:

- CPU intensive applications

Why Node.JS not used in CPU Intensive Applications

Node is, despite its asynchronous event model, by nature single threaded. When you launch a Node process, you are running a single process with a single thread on a single core. So your code will not be executed in parallel, only I/O operations are parallel because they are executed asynchronous. As such, long running CPU tasks will block the whole server and are usually a bad idea.

Given that you just start a Node process like that, it is possible to have multiple Node processes running in parallel though. That way you could still benefit from your multithreading architecture, although a single Node process does not. You would just need to have some load balancer in front that distributes requests along all your Node processes.

The Installation procedure

Head over to Node's official website and download the latest stable version of node.js for your Operating System.

Node.js provides installers for Windows and Mac OS X. Download the installer, run it and follow the on-screen instructions to setup node.js.

For Linux platforms you need to download the Linux Binary(x86/x64), extract it and add the node.js executable to your system's path variable.

Follow the steps below to install node.js in Linux. The current version of node.js at the time of writing this post is 6.10.3.

```
# Download the latest version of node.js
$ wget https://nodejs.org/dist/v6.10.3/node-v6.10.3-linux-x64.tar.xz

# Extract the tar file
$ tar xzvf node-v6.10.3-linux-x64.tar.xz

# Create a directory to store nodejs binary
$ sudo mkdir -p /usr/local/nodejs

# Move nodejs binary to the new directory
$ sudo mv node-v6.10.3-linux-x64/* /usr/local/nodejs
```

Now you need to add node.js executable in the PATH variable.

Add the following to your ~/.bashrc or ~/.bash_profile file and restart the terminal -

export PATH="$PATH:/usr/local/nodejs/bin"

Verify your installation by typing node in the terminal -

$ node

>

Understanding the use of .bashrc

.bashrc is a shell script that Bash runs whenever it is started interactively. It initializes an interactive shell session. You can put any command in that file that you could type at the command prompt.

You put commands here to set up the shell for use in your particular environment, or to customize things to your preferences. A common thing to put in .bashrc are aliases that you want to always be available.

.bashrc runs on every interactive shell launch. If you say:

$ bash ; bash ; bash

and then hit Ctrl-D three times, .bashrc will run three times. But if you say this instead:

$ bash -c exit ; bash -c exit ; bash -c exit

then .bashrc won't run at all, since -c makes the Bash call non-interactive. The same is true when you run a shell script from a file.

The REPL terminal

Node.js comes with a built-in REPL terminal which works just like your browser's console except that you can't use DOM features with node.js REPL. This REPL stands for READ-EVAL-PRINT-LOOP. It creates a simple programming environment that reads user's input, evaluates it, prints the result, and then repeats the same cycle.

Simply open your terminal, type node and press enter to start the REPL terminal -

$ node

>

You can type any valid Javascript in the REPL and see the output instantly. Let's try evaluating a simple expression –

> 121 + 392

513

You can use Javascript's built-in methods like console.log –

```
> console.log("Hello World!")
Hello World!
undefined
```

You can create variables, write for loops and see the result instantly –

```
> var names = ["Ram", "Shyam", "Alice", "Bob"]
undefined
> for(var i = 0; i < names.length; i++) {
... console.log(names[i]);
... }
Ram
Shyam
Alice
Bob
undefined
>
```

You can create a function and call it from the terminal –

```
> var sum = function(x, y) {
...     return x+y;
... }
undefined
> sum(10,20)
30
```

Once you're done experimenting, press Ctrl + C, or type .exit and then press enter to exit the REPL.

Reviewing the chapter

- NodeJS is an open-source, cross-platform runtime environment for developing server-side web applications.
- Node.js is a platform built on Chrome's JavaScript runtime for easily building fast and scalable network applications.
- Head over to Node's official website and download the latest stable version of node.js for your Operating System.
- Node.js comes with a built-in REPL terminal which works just like your browser's console except that you can't use DOM features with node.js REPL. This REPL stands for READ-EVAL-PRINT-LOOP.

Testing your skills

1. REPL stands for _____

a) READ-EVAL-PRINT-LOOP, b) READ-END-PRINT-LOOP, c) ROTATE-EXIT-PRINT-LOOP, d) None

2. The full form of DIRT is

a) Disc Input Read Track, b) Data In Read Track, c) Data Intensive Real-time Applications, d) None

3. In which of these applications Node.JS cannot be used

a) Data Streaming, b) JSON APIs, c) Single Page, d) CPU Intensive

4. In which of these applications Node.JS can be used

a) Data Streaming, b) JSON APIs, c) Single Page, d) all of the above

5. Which of these are features of Node.JS

a) Slow, b) No buffering, c) Multi Threaded, d) All of the above

Lesson 02: Module Introduction (120 minutes)

Objective: After completing this lesson you will be able to learn about : • Modules in Node.js • Creating and exporting a module	**Materials Required:** • Computer With Windows XP and above • Stable Internet connection
Self- Learning Duration: 60 minutes	**Practical Duration:** 60 minutes
Total Duration: 120 minutes	

Modules in Node.js

When you write Node.js applications, you could actually put all your code into one huge index.js file, no matter how large or complex your application is. The Node.js interpreter doesn't care. But in terms of code organization, you would end up with a hard to understand and hard to debug mess quite quickly. So as a human being, you should care about how to structure your code. This is where modules come in.

The role of modules in Node.js

You can think of Node.js modules as JavaScript libraries - a certain part of your overall codebase (for example, a collection of functions) which you want to keep together, but which you also want to keep separated from the rest of your codebase to keep things cleanly separated.

Just like we keep our socks in one drawer and our shirts in another drawer in our wardrobe - even if we combine both to create an outfit for the day - we can keep different parts of our codebase in different modules and then combine them into a coherent application.

Types of modules in Node.js

There are 3 types of modules in Node.js

1. Core Modules
2. Local Modules
3. Third Party Modules

The Core Modules

The core modules include bare minimum functionalities of Node.js. These core modules are compiled into its binary distribution and load automatically when Node.js process starts. However, you need to import the core module first in order to use it in your application. Have a look at some important core modules in Node.js:

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.

The Local Modules

Local modules are modules created locally in your Node.js application. These modules include different functionalities of your application in separate files and folders. You can also package it and distribute it via Node Pacakge Manager (or NPM), so that Node.js community can use it. For example, if you need to connect to MongoDB and fetch data then you can create a module for it, which can be reused in your application.

Defining Node Package Manager or NPM

NPM is inarguably the world's largest Software Registry, containing over 800,000 code packages. Open-source developers use NPM to share software. In fact, several organizations use NPM to manage private development. NPM is free to download and use.

Third Party Modules

The module.exports or exports is a special object which is included in every JS file in the Node.js application by default. module is a variable that represents current module and exports is an object that will be exposed as a module. So, whatever you assign to module.exports or exports, will be exposed as a module.

Creating and exporting a module in Node.js

A Node.js Module is a .js file with one or more functions.

Focus on the syntax to define a function in Node.js module :

*exports.<function_name> = function (argument_1, argument_2, .. argument_N) { /** function body */ };*

Please note that 'exports' is a keyword which tells Node.js that the function is available outside the module.

Following is an example where we create a Calculator Node.js Module with functions add, subtract and multiply. And use the Calculator module in another Node.js file.

calculator.js

```
// Returns addition of two numbers
exports.add = function (a, b) {
    return a+b;
};

// Returns difference of two numbers
exports.subtract = function (a, b) {
    return a-b;
};

// Returns product of two numbers
exports.multiply = function (a, b) {
    return a*b;
};
```

moduleExample.js

```
var calculator = require('./calculator');

var a=10, b=5;

console.log("Addition : "+calculator.add(a,b));
console.log("Subtraction : "+calculator.subtract(a,b));
console.log("Multiplication : "+calculator.multiply(a,b));
```

The Output:

```
$ node moduleExample.js
Addition : 15
Subtraction : 5
Multiplication : 50
```

Another example to highlight the creation and export of a module

Now let's look at how to create our own module and export it for use elsewhere in our program. Start off by creating a *user.js* file and adding the following:

```
const getName = () => {
  return 'Jim';
};

exports.getName = getName;
```

Now create an *index.js* file in the same folder and add this:

```
const user = require('./user');
console.log(`User: ${user.getName()}`);
```

Run the program using node index.js and you should see the following output to the terminal:

```
User: Jim
```

So what has gone on here? Well, if you look at the *user.js* file, you'll notice that we're defining a *getName* function, then using the exports keyword to make it available for import elsewhere. Then in the *index.js* file, we're importing this function and executing it. Also notice that in the require statement, the module name is prefixed with *./*, as it's a local file. Also note that there's no need to add the file extension.

Exporting Multiple Methods and Values

We can export multiple methods and values in the same way:

```
const getName = () => {
  return 'Jim';
};

const getLocation = () => {
  return 'Munich';
};

const dateOfBirth = '12.01.1982';

exports.getName = getName;
exports.getLocation = getLocation;
exports.dob = dateOfBirth;
```

And in index.js:

```
const user = require('./user');
console.log(
  `${user.getName()} lives in ${user.getLocation()} and was born on ${user.dob}.`
);
```

The code above produces this:

```
Jim lives in Munich and was born on 12.01.1982.
```

Do observe the part on how the name we give the exported *dateOfBirth* variable can be anything we fancy (dob in this case). It doesn't have to be the same as the original variable name.

Practical Session

Instructions: Create the code….using an editor….and save it with HTML extension.

1. Create a module that returns the current date and time:

```
exports.myDateTime = function () {

  return Date();

};
```

2. Create and use a module in node.js files.

Note: After saving the code, click the created HTML file to view the output.

Reviewing the chapter

- You can think of Node.js modules as JavaScript libraries - a certain part of your overall codebase (for example, a collection of functions) which you want to keep together, but which you also want to keep separated from the rest of your codebase to keep things cleanly separated.
- Core, Local, and Third party are the three types of modules in Node.js

Testing your skills

1. Which of these are included within the http module

a) Classes, b) methods, c) events, d) all of the above

2. _____module includes methods to deal with file paths

a) path, b) url, c) util, d) fs

3. URL module includes methods for URL resolution and _____

a) Events, b) query string, c) parsing, d) None of the above

4. Which of the following are part of core modules

a) path, b) http, c) util, d) all of the above

5. _____ tells Node.js that the function is available outside the module.

a) outset, b) exports, c) funcOut, d) findOne

Lesson 03: Database and Session Handling (120 minutes)

Objective: After completing this lesson you will be able to learn about : • Mongoose Library • RDBMS Schema Vs Mongoose Schema	**Materials Required:** • Computer With Windows XP and above • Stable Internet connection
Self- Learning Duration: 120 minutes	**Practical Duration:** nil
Total Duration: 120 minutes	

Mongoose library

Mongoose is an Object Data Modeling (ODM) library for MongoDB and Node.js. It manages relationships between data, provides schema validation, and is used to translate between objects in code and the representation of those objects in MongoDB.

Mongoose provides an incredible amount of functionality around creating and working with schemas. Mongoose currently contains eight SchemaTypes. These are:

1. String
2. Number
3. Date
4. Buffer
5. Boolean
6. Mixed
7. ObjectId
8. Array

Each data type allows you to specify:

- a default value
- a custom validation function
- indicate a field is required
- a get function that allows you to manipulate the data before it is returned as an object
- a set function that allows you to manipulate the data before it is saved to the database
- create indexes to allow data to be fetched faster

Further to these common options, certain data types allow you to further customize how the data is stored and retrieved from the database. For example, a String data type also allows you to specify the following additional options:

- convert it to lowercase
- convert it to uppercase
- trim data prior to saving
- a regular expression that can limit data allowed to be saved during the validation process
- an enum that can define a list of strings that are valid

The *Number* and *Date* properties both support specifying a minimum and maximum value that is allowed for that field.

Most of the eight allowed data types should be quite familiar to you. However, there are several exceptions that may jump out to you, such as *Buffer*, *Mixed*, *ObjectId*, and *Array*.

Buffer

The Buffer data type allows you to save binary data. A common example of binary data would be an image or an encoded file, such as a PDF document.

Mixed

The Mixed data type turns the property into an "anything goes" field. This field resembles how many developers may use MongoDB because there is no defined structure. Be wary of using this data type as it loses many of the great features that Mongoose provides, such as data validation and detecting entity changes to automatically know to update the property when saving.

ObjectId

The ObjectId data type commonly specifies a link to another document in your database. For example, if you had a collection of books and authors, the book document might contain an ObjectId property that refers to the specific author of the document.

Array

The Array data type allows you to store JavaScript-like arrays. With an Array data type, you can perform common JavaScript array operations on them, such as push, pop, shift, slice, etc.

RDBMS Schema Vs Mongoose Schema

While Mongoose Schema depends upon non relational database, the RDBMS schema remains always highly dependent on the relational database.

NoSQL databases like MongoDB have the concept of databases and collections while relational databases like MySQL have databases and tables. In a relational database you define a schema for each table in the database. This means that your tables will have a pre-defined set of columns and a value type that save your datasets in each row.

A NoSQL database like MongoDB is schema-less. Instead of storing rows in a table you create "documents" that are stored in collections. Each document can be thought of as a kind of very efficient JSON file and a collection is, well, a collection of those documents.

But then, when using Mongoose, there is a need to define a schema? What for?

To newcomers to NoSQL the idea of defining a schema may sound somewhat strange. The whole idea is to be able to define random JSON documents and save them.

But have you ever wondered how to pull these random documents out and figure out which properties were saved to document?

The fact may stun you. In almost 90% of the applications, you shall be using model classes along with some standard set of fields that need to be saved to your database. Mongoose models and their schemas allow you to perform validation on the fields you expect or are required.

Now you may ask: If you stop there then you've just got the equivalent of a relational schema. Then, what's the point? Why not just use a relational database?

There are certain kinds of data that defy classification. For example, some types of CRM software that require you to save data about a lead's dependents (for tax purposes maybe) and other data that rarely needs to be accessed on its own but almost always needs to be displayed – and therefore stored – alongside the main model makes perfect sense within a non-relational database.

One thing people overlook in the Mongoose docs is the second, optional options object you can pass to your Mongoose schema. This is an example of a Mongoose model schema that has a defined schema but also allows arbitrary fields to be stored to a document just like when using the native driver. You know, just in case we need to store an array of family member objects alongside a User model but the family member objects themselves are pretty useless on their own. Mongoose makes NoSQL models easy.

Below is an example of the Mongoose schema signature its usage process:

```
// Here's the schema signature:
// Mongoose.schema(schemaObject, options)

// Defining a schema on a Mongoose model
// that allows storage of arbitrary fields
var User = Mongoose.schema({
  username: String,
  password: String
}, {
  strict: false
});
```

Reviewing the chapter

- Mongoose is an Object Data Modeling (ODM) library for MongoDB and Node.js.
- Mongoose provides an incredible amount of functionality around creating and working with schemas.
- While Mongoose Schema depends upon non relational database, the RDBMS schema remains always highly dependent on the relational database.
- Mongoose makes NoSQL models easy.

Testing your skills

1. Mongoose currently contains _____ number of SchemaTypes

a) 6, b) 8, c) 10, d) 12

2. Each data type allows you to specify

a) default value, b) custom validation function, c) Both (a) and (b), d) Only (b)

3. _____ data type allows you to save binary data.

a) Buffer, b) Array, c) Mixed, d) None of the above

4. Which of the following are JavaScript array operations

a) Push, b) Pop, c)Slice, d) All of the above

5. String data type also allows you to specify the following additional options

a) trim data prior to saving, b) convert it to uppercase, c) Both (a) and (b), d) Only (a)

Lesson 04: Unit Testing, Logging & Debugging (120 minutes)

Objective: After completing this lesson you will be able to learn about : • Unit testing in node • Test driven development • Debugging	**Materials Required:** • Computer With Windows XP and above • Stable Internet connection
Self- Learning Duration: 60 minutes	**Practical Duration:** 60 minutes
Total Duration: 120 minutes	

Unit Testing in Node

A good programmer is responsible, meticulous and reliable. Programmers like these make sure their code works. No matter which environment, or which machine. These masters of their craft, always cover code with tests, to ensure their work is valid. Testing a program is necessary to have a better understanding of the outcome.

Every unit test will feature the following parts in its structural setup:

- Test setup
- Calling the tested method
- Asserting

Every single unit test should test one concern only. However, this doesn't mean that you can add one assertion only.

Modules Used for Node.js Unit Testing

For unit testing, we are going to use the following modules:

- test runner: mocha, alternatively tape
- assertion library: chai, alternatively the assert module (for asserting)
- test spies, stubs and mocks: sinon (for test setup).

Spies, stubs and mocks - which one and when?

Before doing some hands-on unit testing, let's take a look at what spies, stubs and mocks are!

Spies

You can use spies to get information on function calls, like how many times they were called, or what arguments were passed to them.

```
it('calls subscribers on publish', function () {
  var callback = sinon.spy()
  PubSub.subscribe('message', callback)

  PubSub.publishSync('message')

  assertTrue(callback.called)
})
// example taken from the sinon documentation site:
http://sinonjs.org/docs/
```

Stubs

Stubs are like spies, but they replace the target function. You can use stubs to control a method's behaviour to force a code path (like throwing errors) or to prevent calls to external resources (like HTTP APIs).

```
it('calls all subscribers, even if there are exceptions', function
(){
  var message = 'an example message'
  var error = 'an example error message'
  var stub = sinon.stub().throws()
  var spy1 = sinon.spy()
  var spy2 = sinon.spy()

  PubSub.subscribe(message, stub)
  PubSub.subscribe(message, spy1)
  PubSub.subscribe(message, spy2)

  PubSub.publishSync(message, undefined)

  assert(spy1.called)
  assert(spy2.called)
  assert(stub.calledBefore(spy1))
})
// example taken from the sinon documentation site:
http://sinonjs.org/docs/
```

Mocks

A mock is a fake method with a pre-programmed behavior and expectations.

```
it('calls all subscribers when exceptions happen', function () {
  var myAPI = {
    method: function () {}
  }

  var spy = sinon.spy()
  var mock = sinon.mock(myAPI)
  mock.expects("method").once().throws()

  PubSub.subscribe("message", myAPI.method)
  PubSub.subscribe("message", spy)
  PubSub.publishSync("message", undefined)

  mock.verify()
  assert(spy.calledOnce)
// example taken from the sinon documentation site:
http://sinonjs.org/docs/
})
```

As you can see, for mocks you have to define the expectations upfront.

Imagine, that you'd like to test the following module:

```
const fs = require('fs')
const request = require('request')

function saveWebpage (url, filePath) {
  return getWebpage(url, filePath)
    .then(writeFile)
}

function getWebpage (url) {
  return new Promise (function (resolve, reject) {
    request.get(url, function (err, response, body) {
      if (err) {
        return reject(err)
      }

      resolve(body)
    })
  })
}

function writeFile (fileContent) {
  let filePath = 'page'
  return new Promise (function (resolve, reject) {
    fs.writeFile(filePath, fileContent, function (err) {
      if (err) {
        return reject(err)
      }

      resolve(filePath)
    })
  })
}

module.exports = {
  saveWebpage
}
```

This module does one thing: it saves a web page (based on the given URL) to a file on the local machine. To test this module we have to stub out both the fs module as well as the request module.

Before actually starting to write the unit tests for this module, at RisingStack, we usually add a test-setup.spec.js file to do basics test setup, like creating sinon sandboxes. This saves you from writing sinon.sandbox.create() and sinon.sandbox.restore() after each tests.

```
// test-setup.spec.js
const sinon = require('sinon')
const chai = require('chai')

beforeEach(function () {
  this.sandbox = sinon.sandbox.create()
})

afterEach(function () {
  this.sandbox.restore()
})
```

Also, please note, that we always put test files next to the implementation, hence the .spec.js name. In our package.json you can find these lines:

```
{
  "test-unit": "NODE_ENV=test mocha '/**/*.spec.js'",
}
```

Once we have these setups, it is time to write the tests itself!

```
const fs = require('fs')
const request = require('request')

const expect = require('chai').expect

const webpage = require('./webpage')

describe('The webpage module', function () {
  it('saves the content', function * () {
    const url = 'google.com'
    const content = '<h1>title</h1>'
    const writeFileStub = this.sandbox.stub(fs, 'writeFile',
function (filePath, fileContent, cb) {
      cb(null)
    })

    const requestStub = this.sandbox.stub(request, 'get', function
(url, cb) {
      cb(null, null, content)
    })

    const result = yield webpage.saveWebpage(url)

    expect(writeFileStub).to.be.calledWith()
    expect(requestStub).to.be.calledWith(url)
    expect(result).to.eql('page')
  })
})
```

Test Driven Development

Test-Driven-Development (TDD) is an increasingly popular, and practical, development methodology in today's software industry, and it is easy to apply in Node.js.

The process is: define a test that expects the output we want from our library, API, or whatever it is we're testing to produce; ensure that the test fails – because we have not yet implemented any functionality; then write the implementation code required to make that test pass.

Modern languages and testing frameworks make this easy to achieve, and we've evolved to the point in technology where we can write concise, easily maintainable tests before even thinking about writing implementation code.

Debugging

Debugging is necessary to figure out issues (if any) in the Node.js application being developed.

The debug module

With this module, you can enable third-party modules to log to the standard output, *stdout*.

To use the debug module, you have to set the DEBUG environment variable when starting your applications. You can also use the * character to wildcard names. The following line will print all the express related logs to the standard output:

DEBUG=express node app.js*

The output will look like this:

```
express:application set "x-powered-by" to true +0ms
express:application set "etag" to 'weak' +3ms
express:application set "etag fn" to [Function: wetag] +2ms
express:application set "env" to 'development' +1ms
express:application set "query parser" to 'extended' +0ms
express:application set "query parser fn" to [Function: parseExtendedQueryString] +0ms
express:application set "subdomain offset" to 2 +0ms
express:application set "trust proxy" to false +0ms
express:application set "trust proxy fn" to [Function: trustNone] +1ms
express:application booting in development mode +0ms
express:application set "view" to [Function: View] +0ms
express:application set "views" to '/Users/gergelyke/Development/risingstack/trace/heapdump-experiment/views' +1ms
express:application set "jsonp callback name" to 'callback' +0ms
express:router use / query +1ms
express:router:layer new / +0ms
express:router use / expressInit +1ms
express:router:layer new / +0ms
express:router:route new / +0ms
express:router:layer new / +0ms
express:router:route get / +1ms
express:router:layer new / +0ms
express:router dispatching GET / +11s
express:router query  : / +2ms
express:router expressInit  : / +1ms
express:router dispatching GET /favicon.ico +249ms
express:router query  : /favicon.ico +1ms
express:router expressInit  : /favicon.ico +0ms
finalhandler default 404 +1ms
```

Node.js includes a full-featured out-of-process debugging utility accessible via a simple TCP-based protocol and built-in debugging client. To start the built-in debugger you have to start your application this way:

node debug app.js

Once you have done that, you will see something like this:

```
gergelyke ~/Development/risingstack/node-hero-node-debug $ node debug ap
p.js
< Debugger listening on port 5858
debug> . ok
break in app.js:5
  3 }
  4
> 5 var res = add('apple', 4)
  6 console.log(res)
  7 });
debug>
```

Basic Usage of the Node Debugger

To navigate this interface, you can use the following commands:

c => continue with code execution

n => execute this line and go to next line

s => step into this function

o => finish function execution and step out

repl => allows code to be evaluated remotely

You can add breakpoints to your applications by inserting the debugger statement into your codebase:

```
function add (a, b) {
  debugger
  return a + b
}

var res = add('apple', 4)
```

Practical Session

Instructions: Create the code….using an editor….and save it with HTML extension.

1. Use the createServer() method to create an HTTP server

var http = require('http');

//create a server object:

http.createServer(function (req, res) {

res.write('Hello World!'); //write a response to the client

res.end(); //end the response

}).listen(8080); //the server object listens on port 8080

2. Use the createServer() method to create another HTTP server

Note: After saving the code, click the created HTML file to view the output.

Reviewing the chapter

In this chapter, we talked about the procedures used in unit testing for nodejs applications. We also focused on debugging procedures.

Testing your skills

1. Which of the following are parts of unit testing

a) Test setup, b) asserting, c) Both (a) and (b), d) none of the above

2. _____ is a fake method with a pre-programmed behavior and expectations

a) mock, b) stubs, c) spies, d) None of the above

3. TDD stands for

a) Test downward development, b) Test driven development, c) Test direct document, d) none

4. To finish a function in node debugger, we use

a) o, b) s, c) n, d) repl

5. They replace the target function. Who are they?

a) mock, b) stubs, c) spies, d) None of the above