Node JS

Lesson 02 : Module Introduction

Lesson Objectives

Working with Modules
Modules in Node.js
Loading a module
package.json usage
Creating package.json
Node Package Manager
Loading a third party module (installed via NPM)
Creating and exporting a module
Working with Buffers
Buffers in Node
Http

Module Introduction

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

Module 1
Module 2
Module 3

Modules in Node.js

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

Loading a module

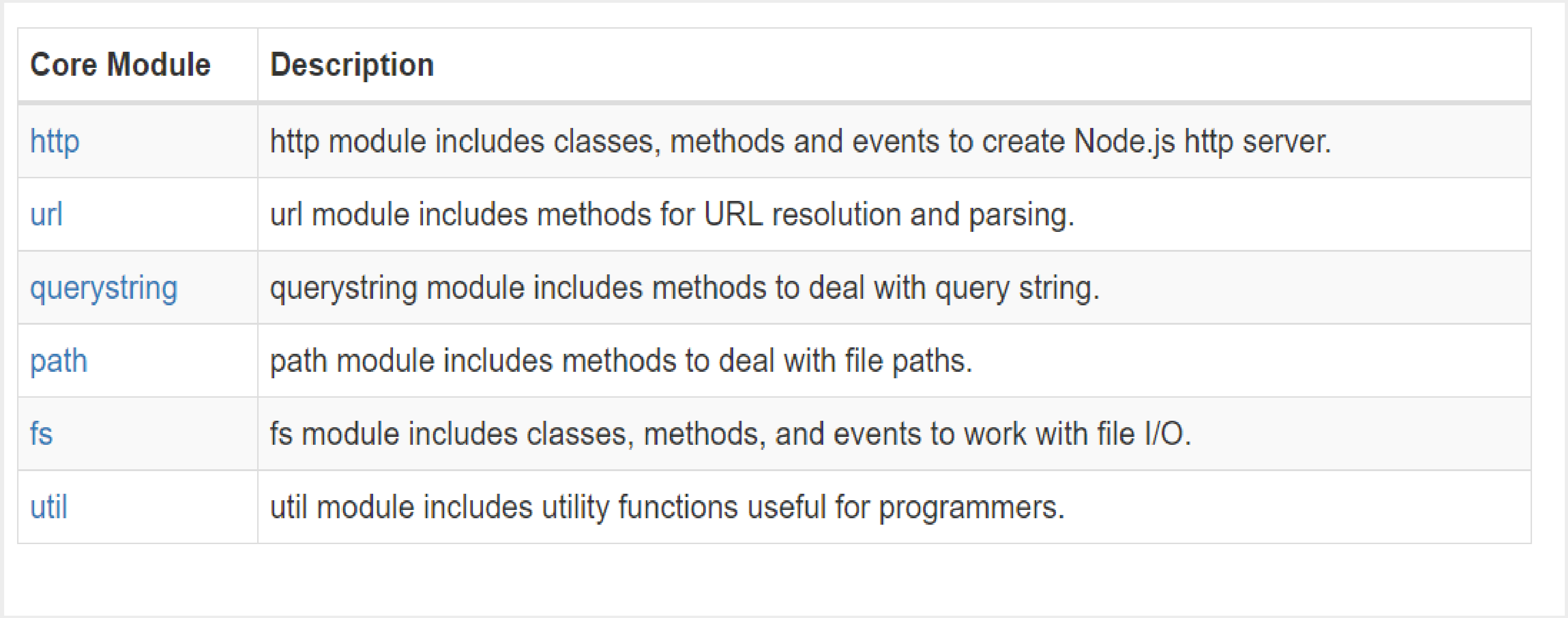
- Node.js includes three types of modules:
- Core Modules
- Local Modules
- Third Party Modules
- **Loading a core module**

- Node has several modules compiled into its binary distribution. These are called the core modules. It is referred solely by the module name, not by the path and are preferentially loaded even if a third-party module exists with the same name.
- var module = require('module_name');
  - *var http = require('http');*

2.1: Working with modules

Loading a module

- Node.js is a light weight framework. 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.



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.

Loading a module

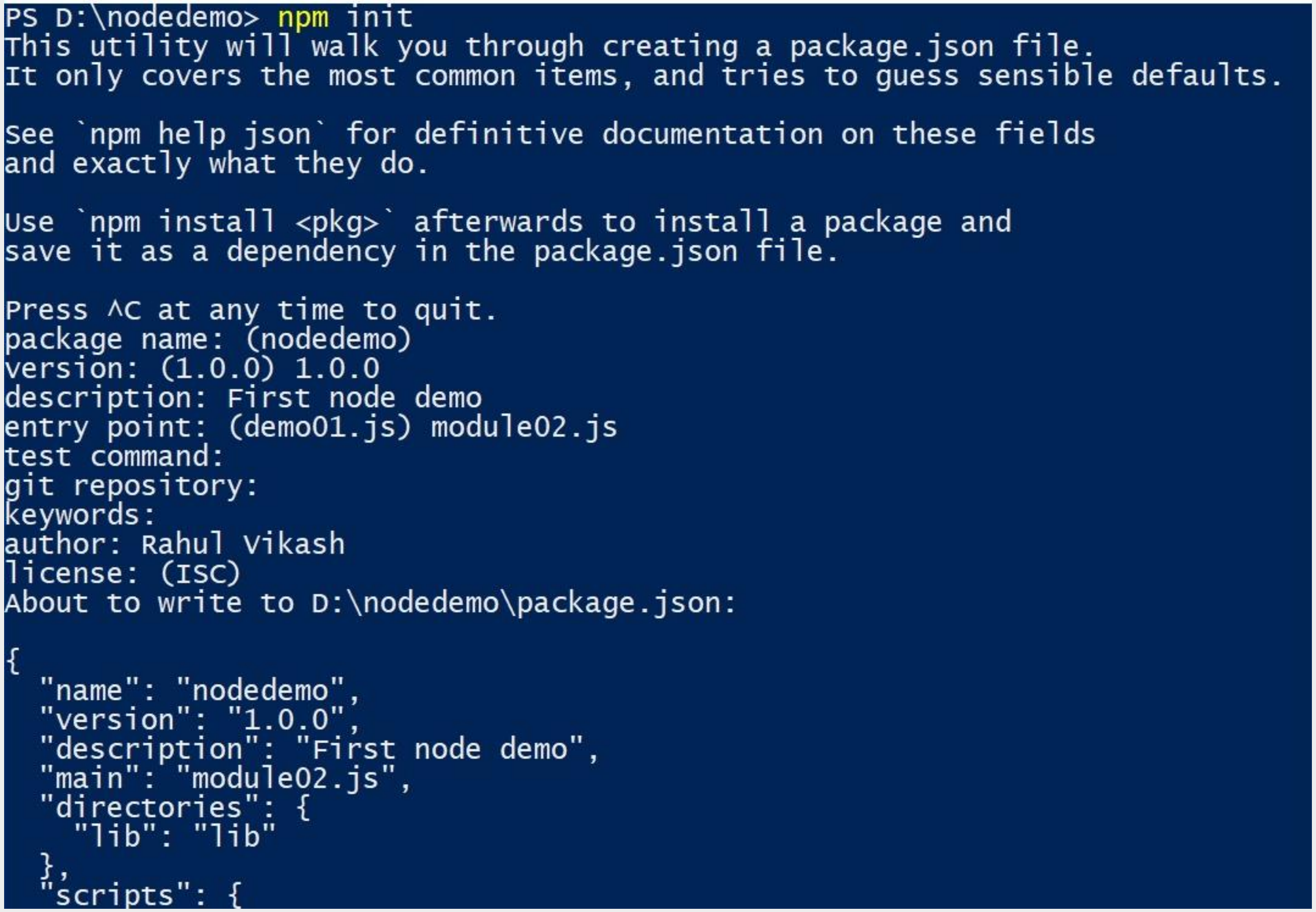
- **Loading a file module (User defined module)**
- We can load non-core modules by providing the absolute path / relative path. Node will automatically adding the .js extension to the module referred.
  - *var myModule = require('../module'); // Relative path for module.js (one folder up level)*
  - *var myModule = require('./module'); // Relative path for module.js (Exists in current directory)*
- **Third-Party Modules**
  - Express
  - Pug/Jade
  - Grunt
  - Gulp
  - Yo
  - Karma
  - Q

Loading a module

➢**Loading a folder module** (User defined module)

- We can use the path for a folder to load a module.
  - *var myModule = require('./myModuleDir');*
- Node will presume the given folder as a package and look for a package definition file inside the folder. Package definition file name should be named as ***pagkage.json***
- Node will try to parse ***package.json*** and look for and use the ***main*** attribute as a relative path for the entry point.
- We need to use ***npm init*** command to create ***package.json***.
- Creating ***Package.json*** using ***npm init*** command

Creating package.json



```
PS D:\nodedemo> npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (nodedemo)
version: (1.0.0) 1.0.0
description: First node demo
entry point: (demo01.js) module02.js
test command:
git repository:
keywords:
author: Rahul Vikash
license: (ISC)
About to write to D:\nodedemo\package.json:

{
  "name": "nodedemo",
  "version": "1.0.0",
  "description": "First node demo",
  "main": "module02.js",
  "directories": {
    "lib": "lib"
  },
  "scripts": {
```

package.json usage

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

package.json usage

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

Node Package Manager

➢ **Loading a module(Third party) installed via NPM (Node Package Manager)**

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

Creating and exporting a module

➢Creating a module that exposes / exports a hello

```
 //module1.js
var hello={
            getData :function(){
                        console.log("In GetData");},
            getLogin :function(){
                        console.log("In GetLogin");}};
module.exports=hello;
```

➢*exports* object is a special object created by the Node module system which is returned as the value of the require function when you include that module.

➢Consuming the function on the exports

```
// Save it as app.js
var obj=require("./module1.js");

obj.getData();
```

➢We can replace exports with module.exports

- exports = module.exports = { }

Demo

Modules

- Module1
- Module2
- module2.1
- Module 3
- Module 4
- Module 4

Buffers in Node

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

Creating Buffers in Node

➢new Buffer(n) is used to create a new buffer of 'n' octets. One octet can be used to represent decimal values ranging from 0 to 255.

➢There are several ways to create new buffers.

- **new Buffer(n)** : To create a new buffer of 'n' octets
  - var buffer = new Buffer(10);
- **new Buffer(arr)** : To create a new buffer, using an array of octets.
  - var buffer = new Buffer([7,1,4,7,0,9]);
- **new Buffer(str,[encoding])** : To create a new buffer, using string and encoding.
  - var buffer = new Buffer("IGATE","utf-8"); // utf-8 is the default encoding in Node.

Writing to Buffer

➢Writing to Buffer

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

Reading from Buffer

➢Reading from buffers

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

Slicing and copying a buffer

➢Slicing a buffer

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

➢Copying a buffer

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

Demo

Buffer

Event Handling in Node

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

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

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

fn(myCallback);
```

EventEmitter

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

```
const eve=require('events');
const emitter=new eve.EventEmitter();

//Register a listner
emitter.on('message',function(msg){
            console.log('In Event listner',msg);
});

//Raise an event
emitter.emit('message',{id:1001,name:"Rahul"});
```

EventEmitter Methods

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

Creating an EventEmitter

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

```
    const event=require('events');
const util=require('util');
var Person= function(name){
                this.name=name;
}
util.inherits(Person,event.EventEmitter);
var person1=new Person("Vikash");
var person2=new Person("Rahul");
var person3=new Person("Ajay");

var people=[person1,person2,person3];

people.forEach(function(person){
                person.on('dep',function(msg){
                                console.log(person.name+' department is '+msg)
                });});
person1.emit('dep','JAVA');
```

Demo

EventModule

- Event01
- Event02
- event03

File System Module

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

File I/O methods

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

File I/O methods

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

Demo

FileModule

- Asynfile
- Syncfile
- directory

Stream

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

Readable Stream

➢*ReadStream* is like an outlet of data, which is an abstraction for a source of data that you are reading from

➢A Readable stream will not start emitting data until you indicate that you are ready to receive it.

➢Readable streams have two "modes": a flowing mode and a non-flowing mode.

- In flowing mode, data is read from the underlying system and provided to your program as fast as possible.
- In non-flowing mode, you must explicitly call stream.read() to get chunks of data out.

➢Readable streams can emit the following events

- '**readable**' : This event is fired when a chunk of data can be read from the stream.
- '**data**' : This event is fired when the data is available. It will switch the stream to flowing mode when it is attached. It is the best way to get the data from stream as soon as possible.

Readable Stream

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

➢Readable streams has the following methods

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

Writable Stream

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

Writable Stream

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

Demo

Stream

Http

- A Web Server is a software application which handles HTTP requests sent by the HTTP client, like web browsers, and returns web pages in response to the clients. Web servers usually deliver html documents along with images, style sheets, and scripts.
- Node.js has a built-in module called HTTP, which allows Node.js to transfer data over the Hyper Text Transfer Protocol (HTTP).
- To include the HTTP module, use the require() method:
  - **var http = require('http');**

Http-Creating server

- The HTTP module can create an HTTP server that listens to server ports and gives a response back to the client.

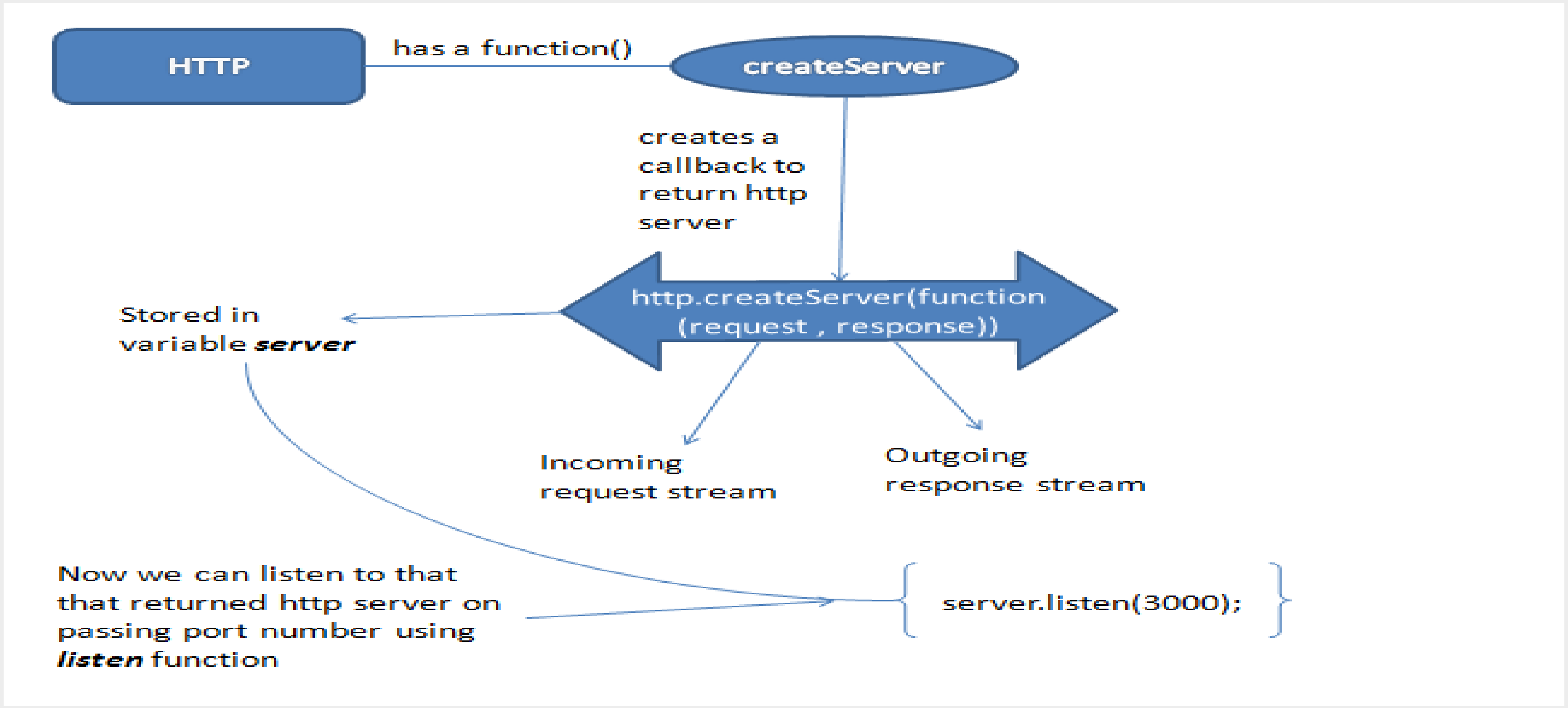
- Use the createServer() method to create an HTTP server:

```
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
```

Http-Creating server

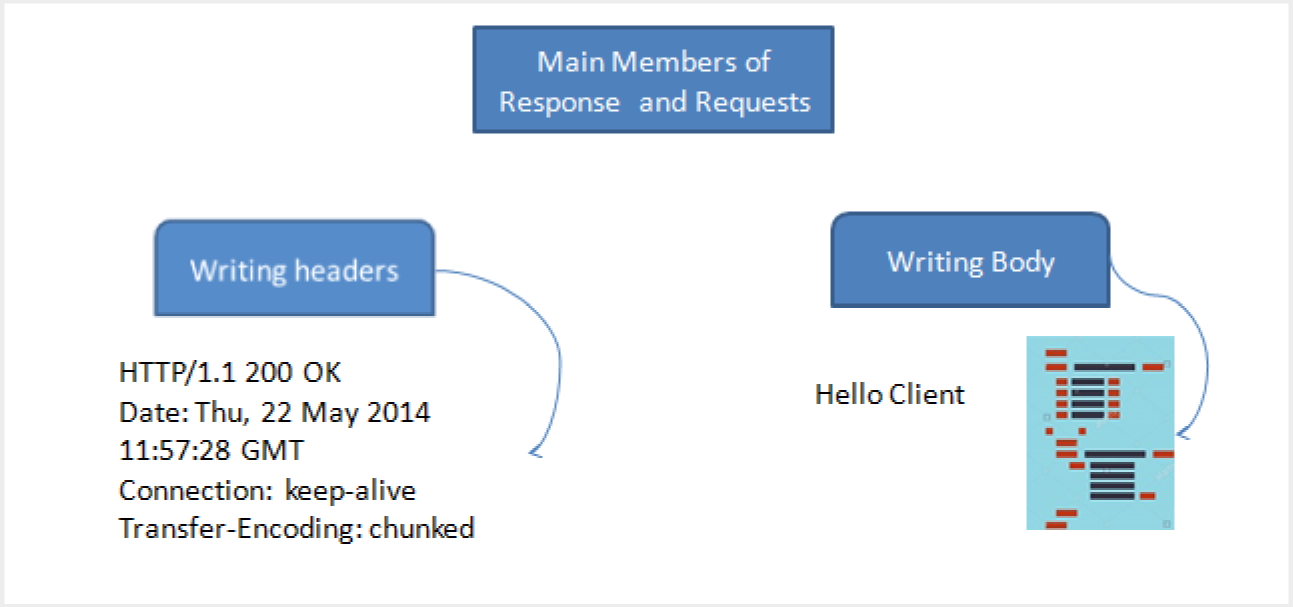
- createServer , which takes a callback and returns an HTTP server. On each client request, the callback is passed in two arguments—the incoming request stream and an outgoing server response stream.


HTTP
has a function()
createServer
creates a callback to return http server
http.createServer(function (request , response))
Stored in variable *server*
Incoming request stream
Outgoing response stream
Now we can listen to that that returned http server on passing port number using *listen* function
server.listen(3000);

Http

```
response.writeHead(200, { 'Content-Type':
'text/html' });
response.setHeader("Content-Type", "text/html");
```

➢This function takes the status code along with optional headers that will be added on to any headers you might have already queued using response.setHeader.



```
HTTP/1.1 200 OK
Date: Thu, 22 May 2014
11:57:28 GMT
Connection: keep-alive
Transfer-Encoding: chunked
```

Http

```
var http = require("http");
var server = http.createServer(function(request,
response) {
response.writeHead(200, {"Content-Type":
"text/html"});
response.write("<!DOCTYPE "html">");
response.write("<html>");
response.write("<head>");
response.write("<title>Hello World Page</title>");
response.write("</head>");
response.write("<body>");
response.write("Hello World!");
response.write("</body>");
response.write("</html>");
response.end();
});

server.listen(80);
console.log("Server is listening");
```

Demo

- ➢ http
- ➢ Router
  - Request
  - response

Lab

Lab 1.2
Lab 1.3