**PART A**

(Part A: TO BE REFFERED BY STUDENTS)

**Experiment No. 10**

**A.1 AIM:**

Develop a web application that involves modules operation, express js and database operations using NodeJS.

**A.2 Pre requisite:**

Basic Knowledge of JavaScript, Basics of DBMS, and HTTP methods.

**A.3 Outcome:**

After successful completion of this experiment students will be able to:

1. Implement effective use of NodeJS
2. Implement the web application that involves database operations using NodeJS.

**A.4 Theory:**

Node.js is an extremely popular open-source, cross-platform web, and network app development resource. Thousands of developers around the world use it daily to develop I/O-intensive web applications such as video streaming sites, single-page applications, online chat applications, and other web apps. Built on Google Chrome's JavaScript-based runtime environment, Node.js brings many advantages to the table, making it a better choice than other server-side platforms like Java or PHP.

What Can Node.js Do?

1. Node.js can generate dynamic page content.
2. Node.js can create, open, read, write, delete, and close files on the server.
3. Node.js can collect form data.
4. Node.js can add, delete, modify data in your database

What is a Node.js File?

1. Node.js files contain tasks that will be executed on certain events.
2. A typical event is someone trying to access a port on the server.
3. Node.js files must be initiated on the server before having any effect
4. Node.js files have extension ".js"

Why Connect Node.js With MySQL?

1. MySQL is an open-source database, so you don’t have to pay anything to use it.
2. MySQL is very powerful. It can handle much of the functionality of the most expensive and powerful database packages.
3. MySQL supports diverse operating systems with many languages like PHP, JAVA, etc.
4. MySQL performs quickly—even with large data sets

## Download Node.js

The official Node.js website has installation instructions for Node.js: [https://nodejs.org](https://nodejs.org/)

**What is Express?**

ExpressJS is one of the most popular HTTP server libraries for Node.js, which by default isn't as friendly for API development. Using Express, we simplify API development by abstracting away the boilerplate needed to set up a server, which makes development faster, more readable and simpler. You can spin up a prototype API in seconds and a couple of lines of code.

Although it's primary use was to simplify things with sensible defaults, it's highly customizable using functions called "middleware".

**HTTP Request Types:**

There are a few types of HTTP methods that we need to grasp before building a REST API. These are the methods that correspond to the CRUD tasks:

* **POST:** Used to submit data, typically used to create new entities or edit already existing entities.
* **GET:** Used to request data from the server, typically used to read data.
* **PUT:** Used to completely replace the resource with the submitted resource, typically used to update data.
* **DELETE:** Used to delete an entity from the server.

## Setting Up the Project

First, let's initialize a new Node.js project:

$ npm init

Fill the requested information to your requirements - you don't have to fill all of the fields, but they're an easy way to set up identifiable data for a project. Fields like the name are much more relevant for publishing applications to the Node Package Manager, amongst other fields.

Alternatively, you can use the default settings by adding the -y flag to the call:

$ npm init -y

Either way - you'll end up with a project with a package.json file. This is a json file that contains all the relevant metadata on your project and will look something along these lines by default:

{

"name": "app",

"version": "1.0.0",

"main": "index.js",

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1"

},

"author": "",

"license": "ISC",

"keywords": [],

"description": ""

}

The "entry"/"main" of your application is the file to run to start the project correctly - usually your main script, and index.js by default.

Additionally, the version of your application and "scripts" are here! You can supply any number of custom commands in the "scripts" section, with a command associated to an alias. Here, the test alias is a wrapper for an echo statement.

You'd run the test of the application via:

$ npm test

> app@1.0.0 test /Users/david/Desktop/app

> echo "Error: no test specified" && exit 1

Error: no test specified

#### Oftentimes, there's a start alias that masks one or more processes that should be run when we want to start an application. In the basic form - we just run the index page with node:

{

"name": "app",

"version": "1.0.0",

"main": "index.js",

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1",

"start": "node index.js"

},

"author": "",

"license": "ISC",

"keywords": [],

"description": ""

}

You can put any number of commands besides node index.js as the start script and when you run npm start - they'll all run:

$ test npm start

> app@1.0.0 start /Users/david/Desktop/app

> node index.js

Note: Since we only have one command in the starting script, it's functionally equivalent to just calling $ node index.js in the command line to start the application.

Now that you're familiar with the project, let's install Express!

$ npm install --save express

A new file is created in the directory, alongside a node\_modules directory. The package-lock.json file keeps track of your dependencies and contains their versions and names:

{

"name": "app",

"version": "1.0.0",

"lockfileVersion": 1,

"requires": true,

"dependencies": {

"accepts": {

"version": "1.3.7",

"resolved": "https://registry.npmjs.org/accepts/-/accepts-1.3.7.tgz",

"integrity": "sha512-Il80Qs2WjYlJIBNzNkK6KYqlVMTbZLXgHx2oT0pU/fjRHyEp+PEfEPY0R3WCwAGVOtauxh1hOxNgIf5bv7dQpA==",

"requires": {

"mime-types": "~2.1.24",

"negotiator": "0.6.2"

}

},

...

The node\_modules directory actually hosts the code of the dependencies, and can get quite large very quickly. Just by installing Express, we've already got a hefty number of modules installed and tracked in the package-lock.json file.

These modules are, factually, small so it's not an issue by any means. By using the package-lock.json file, another client can know which dependencies to download and which versions to use to correctly be able to start up your application.

First, let's create a file called hello-world.js:

$ nano hello-world.js

Then, let's import the Express framework within it:

const express = require('express');

Next, we'll want to instantiate the Express app:

const app = express();

And set our port:

const port = 3000;

The port will be used a bit later, when we tell the app to listen to requests. These three lines are boilerplate - but the great thing is, that's all the boilerplate there is!

Now, we can create a simple GET endpoint right beneath the boilerplate. When a user hits the endpoint with a GET request, the message "Hello World, from express" will be returned (and rendered in the browser or displayed on the console).

We'd like to set it to be on the home page, so the URL for the endpoint is /:

app.get('/', (req, res) => {

res.send('Hello World, from express');

});

At this point, let's start our clients:

app.listen(port, () => console.log(`Hello world app listening on port ${port}!`))

Let's run the application and visit the only endpoint we have via our browser:

$ node hello-world.js

**Hello world app listening on port 3000!**

# **HTTP Module**

Once you have downloaded and installed Node.js on your computer, let's try to display "Hello World" in a web browser. Create a Node.js file named "myfirst.js", and add the following code:

myfirst.js

var http = require('http');  
  
http.createServer(function (req, res) {  
  res.writeHead(200, {'Content-Type': 'text/html'});  
  res.end('Hello World!');  
}).listen(8080);

## Read the Query String

The function passed into the http.createServer() has a req argument that represents the request from the client, as an object (http.IncomingMessage object).

This object has a property called "url" which holds the part of the url that comes after the domain name:

demo\_http\_url.js

var http = require('http');  
http.createServer(function (**req**, res) {  
  res.writeHead(200, {'Content-Type': 'text/html'});  
  res.write(**req.url**);  
  res.end();  
}).listen(8080);

# **File System Module**

### **Example**

Create a new file using the **appendFile()** method:

var fs = require('fs');  
  
fs.appendFile('mynewfile1.txt', 'Hello content!', function (err) {  
  if (err) throw err;  
  console.log('Saved!');  
});

### **Example**

Create a new, empty file using the **open()** method:

var fs = require('fs');  
  
fs.open('mynewfile2.txt', 'w', function (err, file) {  
  if (err) throw err;  
  console.log('Saved!');  
});

### **Example**

Create a new file using the **writeFile()** method:

var fs = require('fs');  
  
fs.writeFile('mynewfile3.txt', 'Hello content!', function (err) {  
  if (err) throw err;  
  console.log('Saved!');  
});

### **Example**

**Delete** "mynewfile2.txt":

var fs = require('fs');  
  
fs.unlink('mynewfile2.txt', function (err) {  
  if (err) throw err;  
  console.log('File deleted!');  
});

### **Example**

**Rename** "mynewfile1.txt" to "myrenamedfile.txt":

var fs = require('fs');  
  
fs.rename('mynewfile1.txt', 'myrenamedfile.txt', function (err) {  
  if (err) throw err;  
  console.log('File Renamed!');  
});

# **URL Module**

To include the URL module, use the require() method:

var url = require('url');

Parse an address with the url.parse() method, and it will return a URL object with each part of the address as properties:

### **Example** [**Get your own Node.js Server**](https://www.w3schools.com/spaces/)

Split a web address into readable parts:

var url = require('url');  
var adr = 'http://localhost:8080/default.htm?year=2017&month=february';  
var q = url.parse(adr, true);  
  
console.log(q.host); //returns 'localhost:8080'  
console.log(q.pathname); //returns '/default.htm'  
console.log(q.search); //returns '?year=2017&month=february'  
  
var qdata = q.query; //returns an object: { year: 2017, month: 'february' }  
console.log(qdata.month); //returns 'february'

## Start with NodeJS and MySQL

## MySQL Database

To be able to experiment with the code examples, you should have MySQL installed on your computer. You can download a free MySQL database at <https://www.mysql.com/downloads/>.

## Install MySQL Driver

Once you have MySQL up and running on your computer, you can access it by using Node.js. To access a MySQL database with Node.js, you need a MySQL driver. This tutorial will use the "mysql" module, downloaded from NPM. To download and install the "mysql" module, open the Command Terminal and execute the following:

C:\Users\Your Name>npm install mysql

Now you have downloaded and installed a mysql database driver. Node.js can use this module to manipulate the MySQL database:

var mysql = require('mysql');

## Create Connection

## Start by creating a connection to the database. Use the username and password from your MySQL database.

demo\_db\_connection.js

var mysql = require('mysql');  
  
var con = mysql.createConnection({  
  host: "localhost",  
  user: "yourusername",  
  password: "yourpassword"  
});  
  
con.connect(function(err) {  
  if (err) throw err;  
  console.log("Connected!");  
});

## Creating a Database

## To create a database in MySQL, use the "CREATE DATABASE" statement:

## Example: Create a database named "mydb":

var mysql = require('mysql');  
  
var con = mysql.createConnection({  
  host: "localhost",  
  user: "yourusername",  
  password: "yourpassword"  
});  
  
con.connect(function(err) {  
  if (err) throw err;  
  console.log("Connected!");  
  con.query("**CREATE DATABASE mydb**", function (err, result) {  
    if (err) throw err;  
    console.log("Database created");  
  });  
});

## Creating a Table

## To create a table in MySQL, use the "CREATE TABLE" statement. Make sure you define the name of the database when you create the connection:

### **Example:** Create a table named "customers":

var mysql = require('mysql');  
  
var con = mysql.createConnection({  
  host: "localhost",  
  user: "yourusername",  
  password: "yourpassword",  
**database: "mydb"**  
});  
  
con.connect(function(err) {  
  if (err) throw err;  
  console.log("Connected!");  
  var sql = "**CREATE TABLE customers (name VARCHAR(255), address VARCHAR(255)**)";  
  con.query(sql, function (err, result) {  
    if (err) throw err;  
    console.log("Table created");  
  });  
});

## Insert Into Table

## To fill a table in MySQL, use the "INSERT INTO" statement.

### **Example:** Insert a record in the "customers" table:

var mysql = require('mysql');  
  
var con = mysql.createConnection({  
  host: "localhost",  
  user: "yourusername",  
  password: "yourpassword",  
  database: "mydb"  
});  
  
con.connect(function(err) {  
  if (err) throw err;  
  console.log("Connected!");  
**var sql = "INSERT INTO customers (name, address) VALUES ('Company Inc', 'Highway 37')";**  con.query(sql, function (err, result) {  
    if (err) throw err;  
    console.log("1 record inserted");  
  });  
});

## Selecting From a Table

## To select data from a table in MySQL, use the "SELECT" statement.

### **Example:** Select all records from the "customers" table, and display the result object:

var mysql = require('mysql');  
  
var con = mysql.createConnection({  
  host: "localhost",  
  user: "yourusername",  
  password: "yourpassword",  
  database: "mydb"  
});  
  
con.connect(function(err) {  
  if (err) throw err;  
  con.query("**SELECT \* FROM customers**", function (err, result, fields) {  
    if (err) throw err;  
    console.log(result);  
  });  
});

## Delete Record

## You can delete records from an existing table by using the "DELETE FROM" statement:

### **Example:** Delete any record with the address "Mountain 21":

var mysql = require('mysql');  
  
var con = mysql.createConnection({  
  host: "localhost",  
  user: "yourusername",  
  password: "yourpassword",  
  database: "mydb"  
});  
  
con.connect(function(err) {  
  if (err) throw err;  
**var sql = "DELETE FROM customers WHERE address = 'Mountain 21'";**  con.query(**sql,** function (err, result) {  
    if (err) throw err;  
    console.log("Number of records deleted: " + result.affectedRows);  
  });  
});

## Update Table

## You can update existing records in a table by using the "UPDATE" statement:

### **Example:** Overwrite the address column from "Valley 345" to "Canyon 123":

var mysql = require('mysql');  
  
var con = mysql.createConnection({  
  host: "localhost",  
  user: "yourusername",  
  password: "yourpassword",  
  database: "mydb"  
});  
  
con.connect(function(err) {  
  if (err) throw err;  
**var sql = "UPDATE customers SET address = 'Canyon 123' WHERE address = 'Valley 345'";**  con.query(**sql,** function (err, result) {  
    if (err) throw err;  
    console.log(result.affectedRows + " record(s) updated");  
  });  
});

**A.5 Procedure/Task:**

* Create a web application to use build in modules operations using Node JS
* Develop a web application that involves database operations using NodeJS

Prepare the document. Save and close the file and name it as **EXP10\_Name of Student**

**PART B**

(PART B: TO BE COMPLETED BY STUDENTS)

(Students must submit the soft copy as per following segments within two hours of the practical. The soft copy must be uploaded on the Blackboard or emailed to the concerned lab in charge faculties at the end of the practical in case the there is no Black board access available)

|  |  |
| --- | --- |
| Roll No. : | Name: |
| Class : | Batch : |
| Date of Experiment : | Date/Time of Submission : |
| Grade : |  |

**B.1 Code:**

*(Paste your Code here)*

**B.2 Output**

*(Take screen shots of the output at run time and paste it here)*

**B.3 Conclusion:**

*(Students must write the conclusion as per the attainment of individual outcome listed above)*

**B.3 Observations and Learning:**

*(Students must write their observations and learnings as per the attainment of individual outcome listed above)*