INSTITUTE FOR ADVANCED COMPUTING AND SOFTWARE DEVELOPMENT,AKURDI

MEAN STACK TECHNOLOGIES,VOL 1

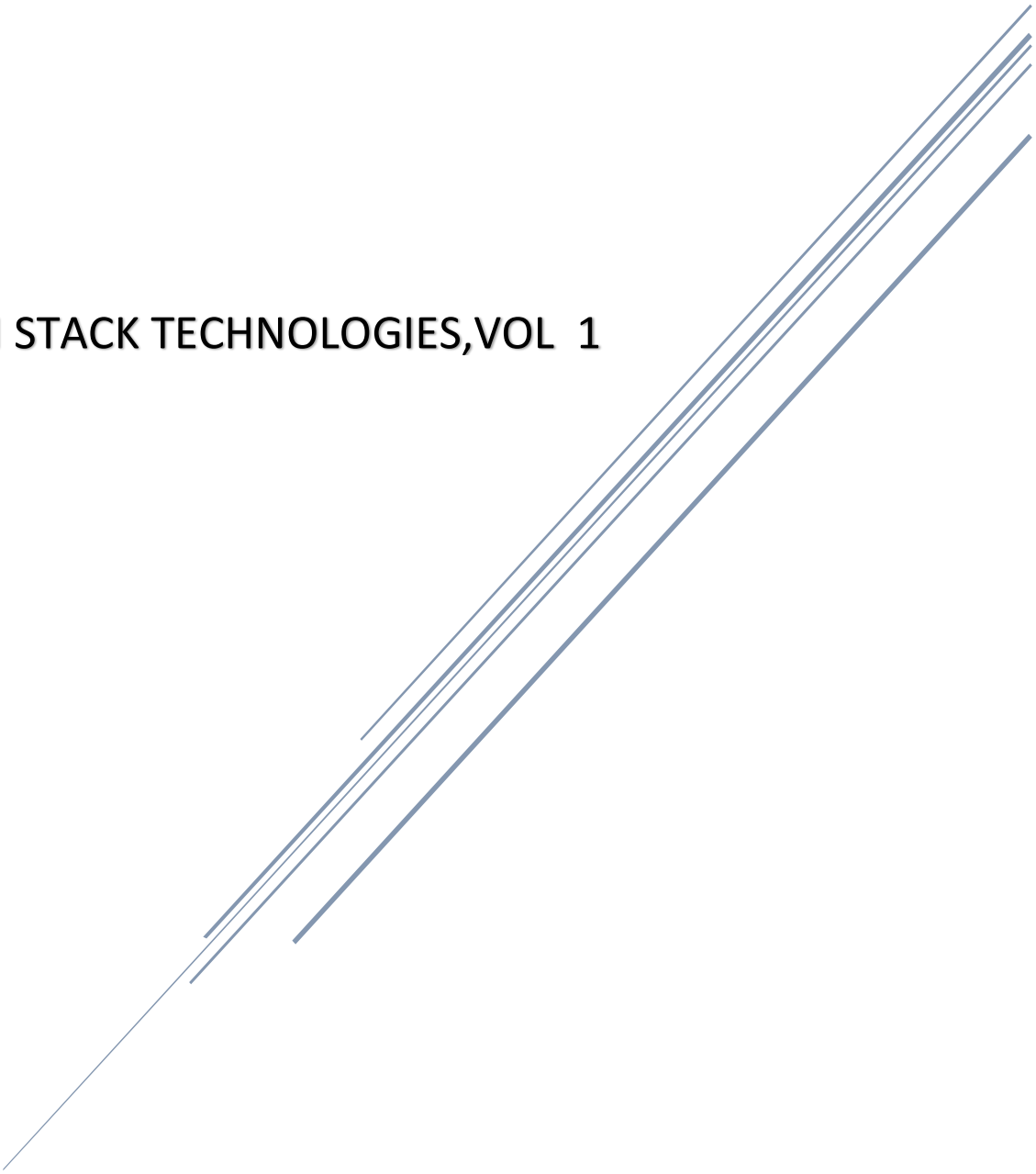

Table of Contents

NODEJS INTRODUCTION

Node.js is an open-source and cross-platform JavaScript runtime environment. It is a popular tool for almost any kind of project!

Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser. This allows Node.js to be very performant.

A Node.js app is run in a single process, without creating a new thread for every request. Node.js provides a set of asynchronous I/O primitives in its standard library that prevent JavaScript code from blocking and generally, libraries in Node.js are written using non-blocking paradigms, making blocking behavior the exception rather than the norm.

When Node.js performs an I/O operation, like reading from the network, accessing a database or the filesystem, instead of blocking the thread and wasting CPU cycles waiting, Node.js will resume the operations when the response comes back.

This allows Node.js to handle thousands of concurrent connections with a single server without introducing the burden of managing thread concurrency, which could be a significant source of bugs.

Node.js has a unique advantage because millions of frontend developers that write JavaScript for the browser are now able to write the server-side code in addition to the client-side code without the need to learn a completely different language.

In Node.js the new ECMAScript standards can be used without problems, as you don't have to wait for all your users to update their browsers - you are in charge of deciding which ECMAScript version to use by changing the Node.js version, and you can also enable specific experimental features by running Node.js with flags.

Node.js Frameworks and Tools

Node.js is a low-level platform. In order to make things easy and exciting for developers, thousands of libraries were built upon Node.js by the community.

Many of those established over time as popular options. Here is a non-comprehensive list of the ones worth learning:

AdonisJs: A full-stack framework highly focused on developer ergonomics, stability, and confidence. Adonis is one of the fastest Node.js web frameworks.
Express: It provides one of the most simple yet powerful ways to create a web server. Its minimalist approach, unopinionated, focused on the core features of a server, is key to its success.
Fastify: A web framework highly focused on providing the best developer experience with the least overhead and a powerful plugin architecture. Fastify is one of the fastest Node.js web frameworks.
Gatsby: A React-based, GraphQL powered, static site generator with a very rich ecosystem of plugins and starters.
hapi: A rich framework for building applications and services that enables developers to focus on writing reusable application logic instead of spending time building infrastructure.
koa: It is built by the same team behind Express, aims to be even simpler and smaller, building on top of years of knowledge. The new project born out of the need to create incompatible changes without disrupting the existing community.

Loopback.io: Makes it easy to build modern applications that require complex integrations.
Meteor: An incredibly powerful full-stack framework, powering you with an isomorphic approach to building apps with JavaScript, sharing code on the client and the server. Once an off-the-shelf tool that provided everything, now integrates with frontend libs React, Vue, and Angular. Can be used to create mobile apps as well.
Micro: It provides a very lightweight server to create asynchronous HTTP microservices.
NestJS: A TypeScript based progressive Node.js framework for building enterprise-grade efficient, reliable and scalable server-side applications.
Next.js: A framework to render server-side rendered React applications.
Nx: A toolkit for full-stack monorepo development using NestJS, Express, React, Angular, and more! Nx helps scale your development from one team building one application to many teams collaborating on multiple applications!
Sapper: Sapper is a framework for building web applications of all sizes, with a beautiful development experience and flexible filesystem-based routing. Offers SSR and more!
Socket.io: A real-time communication engine to build network applications.
Strapi: Strapi is a flexible, open-source Headless CMS that gives developers the freedom to choose their favorite tools and frameworks while also allowing editors to easily manage and distribute their content. By making the admin panel and API extensible through a plugin system, Strapi enables the world's largest companies to accelerate content delivery while building beautiful digital experiences.

How much JavaScript do you need to know to use Node.js?
As a beginner, it's hard to get to a point where you are confident enough in your programming abilities.

While learning to code, you might also be confused at where does JavaScript end, and where Node.js begins, and vice versa.

I would recommend you to have a good grasp of the main JavaScript concepts before diving into Node.js:

Lexical Structure
Expressions
Types
Variables
Functions
this
Arrow Functions
Loops
Scopes
Arrays
Template Literals
Semicolons
Strict Mode
ECMAScript 6, 2016, 2017
With those concepts in mind, you are well on your road to become a proficient JavaScript developer, in both the browser and in Node.js.

The following concepts are also key to understand asynchronous programming, which is one fundamental part of Node.js:

Asynchronous programming and callbacks
Timers
Promises
Async and Await
Closures
The Event Loop

How to install Node.js
Node.js can be installed in different ways. This post highlights the most common and convenient ones.
Official packages for all the major platforms are available at https://nodejs.org/en/download/.

Run Node.js scripts from the command line
The usual way to run a Node.js program is to run the node globally available command (once you install Node.js) and pass the name of the file you want to execute.

If your main Node.js application file is app.js, you can call it by typing:
node app.js
While running the command, make sure you are in the same directory which contains the app.js file.

Expose functionality from a Node.js file using exports
Node.js has a built-in module system.

A Node.js file can import functionality exposed by other Node.js files.

When you want to import something you use

const library = require('./library')
to import the functionality exposed in the library.js file that resides in the current file folder.

In this file, functionality must be exposed before it can be imported by other files.

Any other object or variable defined in the file by default is private and not exposed to the outer world.

This is what the module.exports API offered by the module system allows us to do.

When you assign an object or a function as a new exports property, that is the thing that's being exposed, and as such, it can be imported in other parts of your app, or in other apps as well.

You can do so in 2 ways.

The first is to assign an object to module.exports, which is an object provided out of the box by the module system, and this will make your file export just that object:

```
const car = {
  brand: 'Ford',
  model: 'Fiesta'
}

module.exports = car

//..in the other file

const car = require('./car')
```

The second way is to add the exported object as a property of exports. This way allows you to export multiple objects, functions or data:

```
const car = {
  brand: 'Ford',
  model: 'Fiesta'
}

exports.car = car
```

or directly

```
exports.car = {
  brand: 'Ford',
  model: 'Fiesta'
}
```

And in the other file, you'll use it by referencing a property of your import:

```
const items = require('./items')
items.car
```

or

```
const car = require('./items').car
```

What's the difference between module.exports and exports?

The first exposes the object it points to. The latter exposes the properties of the object it points to.

Introduction to npm

npm is the standard package manager for Node.js.

In January 2017 over 350000 packages were reported being listed in the npm registry, making it the biggest single language code repository on Earth, and you can be sure there is a package for (almost!) everything.

It started as a way to download and manage dependencies of Node.js packages, but it has since become a tool used also in frontend JavaScript.

There are many things that npm does.

Yarn is an alternative to npm. Make sure you check it out as well.

Downloads
npm manages downloads of dependencies of your project.

Installing all dependencies
If a project has a package.json file, by running

npm install
it will install everything the project needs, in the node_modules folder, creating it if it's not existing already.

Installing a single package
You can also install a specific package by running

npm install <package-name>
Often you'll see more flags added to this command:

--save installs and adds the entry to the package.json file dependencies
--save-dev installs and adds the entry to the package.json file devDependencies
The difference is mainly that devDependencies are usually development tools, like a testing library, while dependencies are bundled with the app in production.

Updating packages
Updating is also made easy, by running

npm update
npm will check all packages for a newer version that satisfies your versioning constraints.

You can specify a single package to update as well:

npm update <package-name>

Where does npm install the packages?
When you install a package using npm you can perform 2 types of installation:

a local install
a global install
By default, when you type an npm install command, like:

npm install lodash
the package is installed in the current file tree, under the node_modules subfolder.

As this happens, npm also adds the lodash entry in the dependencies property of the package.json file present in the current folder.

A global installation is performed using the -g flag:

npm install -g lodash
When this happens, npm won't install the package under the local folder, but instead, it will use a global location.

Where, exactly?

The npm root -g command will tell you where that exact location is on your machine.

On macOS or Linux this location could be /usr/local/lib/node_modules. On Windows it could be C:\Users\YOU\AppData\Roaming\npm\node_modules

If you use nvm to manage Node.js versions, however, that location would differ.

I for example use nvm and my packages location was shown as /Users/joe/.nvm/versions/node/v8.9.0/lib/node_modules.

When you install using npm a package into your node_modules folder, or also globally, how do you use it in your Node.js code?

Say you install lodash, the popular JavaScript utility library, using

npm install lodash
This is going to install the package in the local node_modules folder.

To use it in your code, you just need to import it into your program using require:

```
const _ = require('lodash')
```

To see the latest version of all installed npm packages, including their dependencies:

npm list

What is Node js Module?

Module in Node.js is a simple or complex functionality organized in single or multiple JavaScript files which can be reused throughout the Node.js application.

Each module in Node.js has its own context, so it cannot interfere with other modules or pollute global scope.

Also, each module can be placed in a separate .js file under a separate folder.

There are different Node.js Module Types.

Core Modules

Local Modules

Core Modules

The core modules are built in modules .These core modules are compiled into its binary distribution and load automatically when Node.js process starts.

The following table lists some of the important core modules in Node.js.

Name	Description
http	http module includes classes, methods and events to create Node.js http server.
Path	includes methods to deal with file paths.
Querystring	querystring module includes methods to deal with query string
url	url module includes methods for URL resolution and parsing
Fs	fs module includes classes, methods, and events to work with file I/O.

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 NPM, so that Node.js community can use it.

In Node.js, module should be placed in a separate JavaScript file.

At the end, we have assigned this object to module.exports.

The module.exports in the above example exposes a log object as a module.

Loading the Modules

Loading Core/Local Modules

In order to use Node.js core or NPM modules, you first need to import it using require() function as shown below.

```
var module = require('module_name');
```

As per above syntax, specify the module name in the require() function. The require() function will return an object, function, property or any other JavaScript type, depending on what the specified module returns.

Sample Program app.js

```
const readline = require('readline').createInterface({
   input: process.stdin,
   output: process.stdout
  })

  readline.question(`What's your name?`, name => {
   console.log(`Hi ${name}!`)
   readline.close()
  })

  /*How to make a Node.js CLI program interactive?

Node.js since version 7 provides the readline module to perform exactly this: get input from a
readable stream such as the process.
stdin stream, which during the execution of a Node.js program is the terminal input, one line at a
time.
This piece of code asks the username, and once the text is entered and the user presses enter, we
send a greeting.

The question() method shows the first parameter (a question) and waits for the user input. It calls
the callback function once enter is pressed.

In this callback function, we close the readline interface.*/
```

Demo2 for creating a module Arithmetic.js

```
const readline = require('readline').createInterface({
   input: process.stdin,
   output: process.stdout
  })
var arithmetic={
   add: function(num1,num2)
   {
      return num1+num2;
   },
   factorial:function(num)
   {
     var  fact  =1;
     for(var i=1;i<num;i++)
     {
        fact=fact*i;
     }
     return fact;
   }
}
module.exports=arithmetic;
readline.question("Enter number:", num => {
   console.log("THe factorial is"+ arithmetic.factorial(num));
   readline.close()
  })
```

Loading arithmetic module in app.js

```
const arithmetic=require('./arithmetic');

var res=arithmetic.add(10,20);
console.log("The addition is "+res);
var fact_res=arithmetic.factorial(4);
console.log(fact_res);
```

'fs' module

The fs module provides an API for interacting with the file system in a manner closely modeled around standard POSIX functions.

All file system operations have synchronous and asynchronous forms.

The asynchronous form always takes a completion callback as its last argument.

In busy processes, use the asynchronous versions of these calls. The synchronous versions will block the entire process until they complete, halting all connections.

Read File Sync and Async

Use fs.readFile() method to read the physical file asynchronously.

fs.readFile(fileName [,options], callback)

Parameter Description:

filename: Full path and name of the file as a string.

options: The options parameter can be an object or string which can include encoding and flag. The default encoding is utf8 and default flag is "r".

callback: A function with two parameters err and fd. This will get called when readFile operation completes.

Use fs.readFileSync() method to read file synchronously as shown below.

Write a file Async

Use fs.writeFile() method to write data to a file. If file already exists then it overwrites the existing content otherwise it creates a new file and writes data into it.

fs.writeFile(filename, data[, options], callback)

Parameter Description:

filename: Full path and name of the file as a string.

Data: The content to be written in a file.

options: The options parameter can be an object or string which can include encoding, mode and flag. The default encoding is utf8 and default flag is "r".

callback: A function with two parameters err and fd. This will get called when write operation completes

Create one text file SAMPLE.TXT with some content into it.

Demo for reading file synchronously in app.js

```
var fs=require('fs');
data=fs.readFileSync("SAMPLE.txt");
console.log(data.toString());
console.log("Reading data completed");
```

Demo for reading file Asynchronously in app.js

```
var fs=require("fs");
fs.readFile("SAMPLE.txt",function(error,data){
   if(error){
              console.log("error occured");
          }else{
      console.log(data.toString());
      console.log("Reading Data completed here");
          }
```

```
});
console.log("Reading Data completed here");
fs.stat("SAMPLE.txt",function(error,stats){
    if(error){
                console.log("error in stat");
            }
            else{
             console.log(stats);
            }
});
console.log("Readign program stat ends here");
```

Demo for writing data into text file Asynchronously in app.js program

```
fs = require('fs')

fs.writeFile('Testwrite.txt','abc', function (err,data) {
  if (err) {
    return console.log(err);
  }
  console.log('writing done');
});
```

'http' Module

To use the HTTP server and client one must require('http').

The HTTP interfaces in Node.js are designed to support many features of the protocol which have been traditionally difficult to use.

http.createserver() returns a new instance of http.Server.

server.listen() method creates a listener on the specified port or path.

When an HTTP request hits the server, node calls the request handler function with a few handy objects for dealing with the transaction, request and response.

Demo App.js for client server program using nodejs.

```
var http=require('http');
processrequest=function(req,resp){
  resp.write("Hello World!");
  console.log("request is received")
  resp.end("Welcome to Node server!!!");

};
serv=http.createServer(processrequest);
serv.listen(3000);
console.log("Your computer is a server running at port 3000");
```

Run the above program and make a request from browser to http://localhost:3000

'url' Module

url is a module that helps us to deal with url.

The URL module splits up a web address into readable parts.

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

Demo app.js for using url module for reading the different url paths requested and accordingly sending back different responses

```
http=require("http");
url=require("url");
processdata=function(req,resp){
d=url.parse(req.url);
console.log(d);
switch(d.pathname){
case "/":
  resp.writeHead(200,{'Content-Type':'text/html'});
  resp.end("<h1>You are on Home page</h1>");
  break;
case "/about":
  resp.writeHead(200,{'Content-Type':'text/html'});
  resp.end("<h1>a You are on About us Page</h1>");
  break;
default:
  resp.writeHead(200,{'Content-Type':'text/html'});
  resp.end("<h1>page not found</h1>");
  break;
```

```
}
}
http.createServer(processdata).listen(3000);
console.log("server is running at 3000");
```

Run the above program and make a request from browser to http://localhost:3000 and see the response. Also observe the response for http://localhost:3000/about and http://localhost:3000/contactus

'querystring' module

The querystring module provides utilities for parsing and formatting URL query strings.
It can be accessed using:
const querystring = require('querystring');
The querystring.parse() method parses a URL query string (str) into a collection of key and value pairs.

Demo App.js for using querystring module for fetching username and password for validating the user and sending back response accordingly.
Step1: create a module loginmodule.js

```
var loginUser= {

   validate:function(name,password) {
      if(name=="ashwini" && password=="123")
      {
         return true;
      }
      return false;

   }
}
module.exports=loginUser;
```

Step2: create app.js

```
fs=require('fs')
http=require("http");
url=require("url");
query=require("querystring");
loginUser=require("./loginmodule1");


processdata=function(req,resp){
reqUrl=url.parse(req.url);
console.log(reqUrl);
switch(reqUrl.pathname){
case "/":
  resp.writeHead(200,{'Content-Type':'text/html'})
  fs.readFile("Welcome.html",function(error,data){
      if(error){
```

```
                                        console.log("error ocureed");
                                    }
                                    else{
                                    resp.end(data);
                                    }
  });
  break;
case "/validate":
  resp.writeHead(200,{'Content-Type':'text/html'})
  data=query.parse(reqUrl.query);
  if(loginUser.validate(data.username,data.password)==true)
  {
   resp.end("Login successfull. Welcome to your page");
  }
  else
  {
     resp.end("Invalid credentials!!!");
  }
  break;
default:
  resp.writeHead(200,{'Content-Type':'text/html'})
  resp.end("<h1>page not found</h1>");
  break;


}
}
http.createServer(processdata).listen(3000);
console.log("server is running at 3000");
```

Step3: create Welcome.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Login</title>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.0/css/bootstrap.min.css">
  <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>
  <script
src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.16.0/umd/popper.min.js"></script>
  <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.0/js/bootstrap.min.js"></script>
</head>
<body>

<div class="container">
  <h2>Form control: input</h2>
  <p>The form below contains two input elements; one of type text and one of type
password:</p>
  <form action="/validate">
```

```
    <div class="form-group">
      <label for="usr">Name:</label>
      <input type="text" class="form-control" id="usr" name="username">
    </div>
    <div class="form-group">
      <label for="pwd">Password:</label>
      <input type="password" class="form-control" id="pwd" name="password">
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
  </form>
</div>

</body>
</html>
```

Run app.js and see the output in browser for this url http://localhost:3000

ExpressJS

Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications.

You can assume express as a layer built on the top of the Node.js that helps manage a server and routes.

It can be used to design single-page, multi-page and hybrid web applications.

It allows to setup middlewares to respond to HTTP Requests.

It defines a routing table which is used to perform different actions based on HTTP method and URL.

Why to use express?

Express.js simplifies development and makes it easier to write secure, modular and fast applications.

You can do all that in plain old Node.js, but some bugs can (and will) surface, including security concerns (eg. not escaping a string properly)

Ultra fast I/O

Asynchronous and single threaded

MVC like structure

Robust API makes routing easy

Due to middleware functionalities request processing is more efficient.

Installation of express

You have to install the express framework globally to create web application using Node terminal.

Use the following command to install express framework globally.

npm install -g express

Use the following command to install express module:

npm install express --save

req and res Objects

Express.js Request and Response objects are the parameters of the callback function which is used in Express applications.

The express.js request object represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers, and so on.

The Response object (res) specifies the HTTP response which is sent by an Express app when it gets an HTTP request.

res.send([body])

This method is used to send HTTP response.

GET and POST requests

GET and POST both are two common HTTP requests.

GET requests are used to send only limited amount of data because data is sent into header while POST requests are used to send large amount of data because data is sent in the body.

GET requests send the data in url therefore it is not safe way sending data, whereas POST requests send the data enclosed in body part so it is secure way of sending data.

Express.js facilitates you to handle GET and POST requests using the instance of express.

express.get() for GET request and express.post() for POST requests.

Demo app.js for using express framework

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


app.get('/', function (req, res) {
  res.send('<h1>Welcome to Express World</h1>');
})

var server = app.listen(3000, function () {
 console.log("Example app listening at 3000");
})
```

Demo app.js for routing using express

```
express=require("express");
app=express();


app.get("/",function(req,resp){
    resp.send("<h1>Hello world!! you requested "+ req.path+"</h1>");
});
app.get("/about",function(req,resp){
    resp.send("<h1>about us !!! you requested "+ req.path+"</h1>");
});

app.listen(3000,function(){
   console.log("server listening on port 3000");
});
```

Creating Middlewares using App.use()

app.use is a way to register middleware or chain of middlewares (or multiple middlewares) before executing any end route logic or intermediary route logic depending upon order of middleware registration sequence.

Middleware: forms chain of functions/middleware-functions with 3 parameters req, res, and next. next is callback which refer to next middleware-function in chain and in case of last middleware-function of chain next points to first-middleware-function of next registered middlerare-chain.

Demo app.js for observing the middleware chain

```
express=require("express");
app=express();
app.use(function(req,resp,next){
                                console.log("url:"+req.url)
                                next();
});
app.use(function(req,resp,next){
                                console.log("Method:"+req.method)
                                next();
});

app.get("/",function(req,resp){
```

```
    resp.send("<h1>Hello world!!</h1>");
});
app.get("/about",function(req,resp){
    resp.send("<h1>about us</h1>");
});

app.listen(3000,function(){console.log("server listening at 3000");});
```

Routing

Routing is made from the word route. It is used to determine the specific behavior of an application. It specifies how an application responds to a client request to a particular route, URI or path and a specific HTTP request method (GET, POST, etc.).

It can handle different types of HTTP requests.

Body-parser module

body-parser extract the entire body portion of an incoming request stream and exposes it on req.body.

The middleware was a part of Express.js earlier but now you have to install it separately.

This body-parser module parses the JSON, buffer, string and URL encoded data submitted using HTTP POST request. Install body-parser using NPM as shown below.

npm install body-parser –save

Demo for observing body-parser module and routing of get as well as post requests using expressjs

Step1- create registration.html

```
<!DOCTYPE html>
<html>
<body>
<form action="/register" method="post">
<table>
<tr><td>Enter First Name:</td><td><input type="text" name="firstname"/><td></tr>
<tr><td>Enter Last Name:</td><td><input type="text" name="lastname"/><td></tr>
<tr><td>Enter Password:</td><td><input type="password" name="password"/></td></tr>
<tr><td>Sex:</td><td>
<input type="radio" name="gender" value="male"> Male
<input type="radio" name="gender" value="female">Female
</td></tr>
<tr><td>About You :</td><td>
<textarea rows="5" cols="40" name="aboutyou" placeholder="Write about yourself">
</textarea>
</td></tr>
<tr><td colspan="2"><input type="submit" value="register"/></td></tr>
</table>
</form>
</body>
</html>
```

Step2- Create app.js

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

```
bodyparser=require("body-parser");
app.use(bodyparser.urlencoded({extended:false}));

app.use(function(req,resp,next){
                                console.log("url:"+req.url)
                                next();
});
app.use(function(req,resp,next){
                                console.log("Method:"+req.method)
                                next();
});

app.get('/',function(req,res){
   res.sendFile(__dirname+'/registration.html')
})

app.post('/register', function (req, res) {
res.send('<p>Firstname: ' + req.body.firstname + '</p>'+
'<p>Lasttname: ' + req.body.firstname + '</p>'+
'<p>Gender: ' + req.body.gender + '</p>'+
'<p>about you : ' + req.body.aboutyou + '</p>');
})

var server = app.listen(3000, function () {
  console.log("Example app listening at 3000");
})
```

Demo for login of user

Step1- Create Login.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Login</title>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.0/css/bootstrap.min.css">
  <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>
  <script
src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.16.0/umd/popper.min.js"></script>
  <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.0/js/bootstrap.min.js"></script>
</head>
<body>

<div class="container">
  <h2>Login</h2>

  <form action="/validate" method="POST">
    <div class="form-group">
      <label for="usr">Name:</label>
      <input type="text" class="form-control" id="usr" name="username">
```

```
  </div>
  <div class="form-group">
    <label for="pwd">Password:</label>
    <input type="password" class="form-control" id="pwd" name="password">
  </div>
  <button type="submit" class="btn btn-primary">Submit</button>
 </form>
</div>

</body>
</html>
```

Step2- Create loginmodule.js

```
var loginUser= {

   validate:function(name,password) {
      if(name=="ashwini" && password=="123")
      {
         return true;
      }
      return false;

   }
}

module.exports=loginUser;
```

Step3- Create app.js

```
loginUser=require('./loginmodule');

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

bodyParser=require('body-parser');
app.use(bodyParser.urlencoded({extended:false}));
app.use(bodyParser.json());


app.get('/',function(req,res){
   res.sendFile(__dirname+'/Login.html');
})

app.post("/validate",function(req,res){
username=req.body.username;
password=req.body.password;
console.log("username="+username+" password="+password);
if(loginUser.validate(username,password)==true)
{
   res.send("<form action=\"/continue\" method=\"post\">"+
   "<h1> Hi "+username+", welcome  to online shop</h1>"+
   "<input type=\"submit\" value=\"submit\"/>");
```

```
}
else{
   res.sendFile(__dirname+'/Login.html');
}
})


app.post("/continue",function(req,res)
{
   res.send("Hi "+req.body.username +" you are on continue page");
})

server=app.listen(3000,function(){
   console.log("server listening at port 3000");
})
```

State Management using Cookies

Cookies are simple, small files/data that are sent to client with a server request and stored on the client side. Every time the user loads the website back, this cookie is sent with the request. This helps us keep track of the user's actions.

The following are the numerous uses of the HTTP Cookies
Session management
Personalization(Recommendation systems)
Session tracking
To use cookies with Express, we need the cookie-parser middleware. To install it, use the following code –

npm install --save cookie-parser

cookie-parser is a middleware which parses cookies attached to the client request object. cookie-parser parses Cookie header and populates req.cookies with an object keyed by the cookie names

Demo for cookie management using cookie parser.
Step1- create Login.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Home</title>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.0/css/bootstrap.min.css">
  <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>
  <script
src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.16.0/umd/popper.min.js"></script>
  <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.0/js/bootstrap.min.js"></script>
</head>
<body>

<div class="container">
  <h2>Login</h2>
```

```
 <form action="/validate" method="POST">
  <div class="form-group">
   <label for="usr">Name:</label>
   <input type="text" class="form-control" id="usr" name="username">
  </div>
  <div class="form-group">
   <label for="pwd">Password:</label>
   <input type="password" class="form-control" id="pwd" name="password">
  </div>
  <button type="submit" class="btn btn-primary">Submit</button>
 </form>
</div>

</body>
</html>
```

Step2- create loginmodule.js

```
var loginUser= {

   validate:function(name,password) {
      if(name=="ashwini" && password=="123")
      {
         return true;
      }
      return false;

   }
}
module.exports=loginUser;
```

Step3- app.js

```
loginUser=require('./loginmodule');

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

cookieparser=require('cookie-parser');

bodyParser=require('body-parser');
app.use(bodyParser.urlencoded({extended:false}));
app.use(bodyParser.json());
app.use(cookieparser());


app.get('/',function(req,res){

   res.sendFile(__dirname+'/Login.html');
})

app.post("/validate",function(req,res){
```

```
username=req.body.username;
password=req.body.password;
console.log("username="+username+" password="+password);
if(loginUser.validate(username,password)==true)
{
   let userinfo={"uname":username};
   res.cookie("userinfo",userinfo,{maxAge:4000});
   res.send("<form action=\"/continue\" method=\"post\">"+
   "<h1> Hi "+username+", welcome  to online shop</h1>"+
   "<input type=\"submit\" value=\"submit\"/>");
}
else{
   res.sendFile(__dirname+'/Login.html');
}
})


app.post("/continue",function(req,res)
{
   if(req.cookies.userinfo=="undefined")
   {
   console.log("The cookie is "+req.cookies.userinfo["uname"].toString());
   res.send("Hi "+req.cookies.userinfo["uname"] +" you are on continue page");
   }
   else
   {
      res.send("<h1>session expired. login again</h1>");
   }
})

server=app.listen(3000,function(){
   console.log("server listening at port 3000");
})
```

Introduction to Angular

Angular is an application design framework and development platform for creating efficient and sophisticated single-page apps.

To install Angular on your local system, you need the following:

Node.js

npm package manager

Angular, the Angular CLI, and Angular applications depend on npm packages for many features and functions. To download and install npm packages, you need an npm package manager.

Install the Angular CLI

You use the Angular CLI to create projects, generate application and library code, and perform a variety of ongoing development tasks such as testing, bundling, and deployment.

To install the Angular CLI, open a terminal window and run the following command:

npm install -g @angular/cli

Create a workspace and initial application

You develop apps in the context of an Angular workspace.

To create a new workspace and initial starter app:

Run the CLI command ng new and provide the name my-app, as shown here:

ng new my-app

The ng new command prompts you for information about features to include in the initial app.

Accept the defaults by pressing the Enter or Return key.

The Angular CLI installs the necessary Angular npm packages and other dependencies. This can take a few minutes.

The CLI creates a new workspace and a simple Welcome app, ready to run

Run the application

The Angular CLI includes a server, so that you can build and serve your app locally.

Navigate to the workspace folder, such as my-app.

Run the following command:

cd my-app

ng serve --open

The ng serve command launches the server, watches your files, and rebuilds the app as you make changes to those files.

The --open (or just -o) option automatically opens your browser to http://localhost:4200/.

If your installation and setup was successful, you should see a default page in browser

Angular Components Overview

Components are the main building block for Angular applications. Each component consists of:
An HTML template that declares what renders on the page
A Typescript class that defines behavior
A CSS selector that defines how the component is used in a template
Optionally, CSS styles applied to the template

Creating a component using the Angular CLI

To create a component using the Angular CLI:

From a terminal window, navigate to the directory containing your application.
Run the ***ng generate component <component-name>*** command, where <component-name> is the name of your new component.

By default, this command creates the following:
A folder named after the component
A component file, <component-name>.component.ts
A template file, <component-name>.component.html
A CSS file, <component-name>.component.css
A testing specification file, <component-name>.component.spec.ts
Where <component-name> is the name of your component.

Creating a component manually

Although the Angular CLI is the easiest way to create an Angular component, you can also create a component manually. This section describes how to create the core component file within an existing Angular project.

To create a new component manually:

Navigate to your Angular project directory.

Create a new file, <component-name>.component.ts.

At the top of the file, add the following import statement.

```
import { Component } from '@angular/core';
```

After the import statement, add a @Component decorator.

```
@Component({
})
```

Choose a CSS selector for the component.

```
@Component({
  selector: 'app-component-overview',
})
```

For more information on choosing a selector, see Specifying a component's selector.

Define the HTML template that the component uses to display information. In most cases, this template is a separate HTML file.

```
@Component({
  selector: 'app-component-overview',
  templateUrl: './component-overview.component.html',
})
```

For more information on defining a component's template, see Defining a component's template.

Select the styles for the component's template. In most cases, you define the styles for you component's template in a separate file.

```
@Component({
  selector: 'app-component-overview',
  templateUrl: './component-overview.component.html',
  styleUrls: ['./component-overview.component.css']
})
```

Add a class statement that includes the code for the component.

```
export class ComponentOverviewComponent {

}
```

Specifying a component's CSS selector

Every component requires a CSS selector. A selector instructs Angular to instantiate this component wherever it finds the corresponding tag in template HTML. For example, consider a component, hello-world.component.ts that defines its selector as app-hello-world. This selector instructs angular to instantiate this component any time the tag, <app-hellow-world> in a template.

To specify a component's selector, add a selector statement to the @Component decorator.

```
@Component({
  selector: 'app-component-overview',
})
```

Defining a component's template

A template is a block of HTML that tells Angular how to render the component in your application. You can define a template for your component in one of two ways: by referencing an external file, or directly within the component.

To define a template as an external file, add a templateUrl property to the @Component decorator.

```
@Component({
  selector: 'app-component-overview',
  templateUrl: './component-overview.component.html',
})
```

To define a template within the component, add a template property to the @Component decorator that contains the HTML you want to use.

```
@Component({
 selector: 'app-component-overview',
 template: '<h1>Hello World!</h1>',
})
```
If your want your template to span multiple lines, you can use backticks (`). For example:

```
@Component({
 selector: 'app-component-overview',
 template: `<h1>Hello World!</h1>
        <p>This template definition spans
         multiple lines.</p>`
})
```
An Angular component requires a template defined using template or templateUrl. You cannot have both statements in a component.

Declaring a component's styles
You can declare component styles uses for its template in one of two ways: by referencing an external file, or directly within the component.

To declare the styles for a component in a separate file, add a stylesUrls property to the @Component decorator.

```
@Component({
 selector: 'app-component-overview',
 templateUrl: './component-overview.component.html',
 styleUrls: ['./component-overview.component.css']
})
```
To select the styles within the component, add a styles property to the @Component decorator that contains the styles you want to use.

```
@Component({
 selector: 'app-component-overview',
 template: '<h1>Hello World!</h1>',
 styles: ['h1 { font-weight: normal; }']
})
```
The styles property takes an array of strings that contain the CSS rule declarations.

Templates

Each Angular template in your app is a section of HTML that you can include as a part of the page that the browser displays. An Angular HTML template renders a view, or user interface, in the browser, just like regular HTML, but with a lot more functionality.

When you generate an Angular app with the Angular CLI, the app.component.html file is the default template containing placeholder HTML.
With special Angular syntax in your templates, you can extend the HTML vocabulary of your apps. For example, Angular helps you get and set DOM (Document Object Model) values dynamically with features such as built-in template functions, variables, event listening, and data binding.

Interpolation and template expressions

Interpolation allows you to incorporate calculated strings into the text between HTML element tags and within attribute assignments. Template expressions are what you use to calculate those strings.

Interpolation {{...}}

Interpolation refers to embedding expressions into marked up text. By default, interpolation uses as its delimiter the double curly braces, {{ and }}.