

Unit -3(First Half)

Introduction to Node.js

Node.js is a cross-platform runtime environment and library for running JavaScript applications outside the browser. It is used for creating server-side and networking web applications. It is open source and free to use. It can be downloaded from this link <https://nodejs.org/en/>

Features of Node.js

Following is a list of some important features of Node.js that makes it the first choice of software architects.

1. **Extremely fast:** Node.js is built on Google Chrome's V8 JavaScript Engine, so its library is very fast in code execution.
2. **I/O is Asynchronous and Event Driven:** All APIs of Node.js library are asynchronous i.e. non-blocking. So a Node.js based server never waits for an API to return data. The server moves to the next API after calling it and a notification mechanism of Events of Node.js helps the server to get a response from the previous API call. It is also a reason that it is very fast.
3. **Single threaded:** Node.js follows a single threaded model with event looping.
4. **Highly Scalable:** Node.js is highly scalable because event mechanism helps the server to respond in a non-blocking way.
5. **No buffering:** Node.js cuts down the overall processing time while uploading audio and video files. Node.js applications never buffer any data. These applications simply output the data in chunks.
6. **Open source:** Node.js has an open source community which has produced many excellent modules to add additional capabilities to Node.js applications.
7. **License:** Node.js is released under the MIT license.

Applications of Node.js:

Node.js is used for building different type of applications. Some of the application types are listed below –

1. **Streaming applications** – Node.js can easily handle real-time data streams, where it is required to download resources on-demand without overloading the server or the user's local machine.

Node.js can also provide quick data synchronization between the server and the client, which improves user experience by minimizing delays using the Node.js event loop.

2. **Single page apps** – Node.js is an excellent choice for SPAs because of its capability to efficiently handle asynchronous calls and heavy input/output(I/O) workloads. Data driven SPAs built with Express.js are fast, efficient and robust.
3. **Real-time applications** – Node.js is ideal for building lightweight real-time applications, like messaging apps interfaces, chatbots etc. Node.js has an event- based architecture, as a result has an excellent Web Socket support. It facilitates real-time two-way communication between the server and the client.
4. **APIs** – At the heart of Node.js is JavaScript. Hence, it becomes handling JSON data is easier. You can therefore build REST based APIs with Node.js.

Installation of Node j.s:

Node Package Manager provides two main functionalities:

- It provides online repositories for node.js packages/modules which are searchable of search.npmjs.org
- It also provides command line utility to install Node.js packages, do version management and dependency management of Node.js packages
- The npm comes bundled with Node.js installable in versions after that v0.6.3. You can check the version by opening Node.js command prompt and typing the following command.

Installing Modules using npm

Installing NPM:

To install NPM, it is required to install Node.js as NPM gets installed with Node.js automatically.

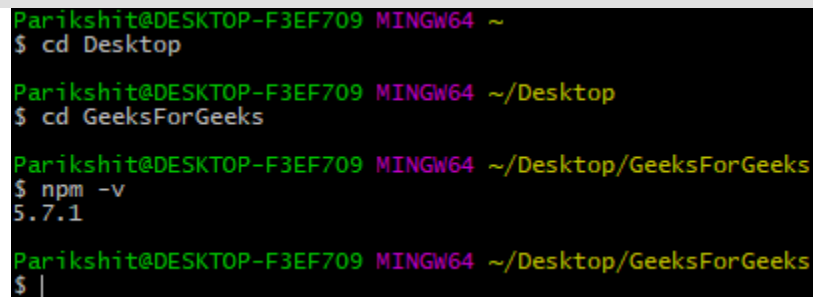
[Install Node.js.](#)

Checking and updating npm version

Version of **npm** installed on system can be checked using following syntax:

Syntax to check npm version:

```
npm -v
```



```
Parikshit@DESKTOP-F3EF709 MINGW64 ~  
$ cd Desktop  
  
Parikshit@DESKTOP-F3EF709 MINGW64 ~/Desktop  
$ cd GeeksForGeeks  
  
Parikshit@DESKTOP-F3EF709 MINGW64 ~/Desktop/GeeksForGeeks  
$ npm -v  
5.7.1  
  
Parikshit@DESKTOP-F3EF709 MINGW64 ~/Desktop/GeeksForGeeks  
$ |
```

Checking npm version

If the installed version is not latest, one can always update it using the given syntax:

Syntax to update npm version:

```
npm update npm@latest -g.
```

As **npm** is a global package, **-g** flag is used to update it **globally**.

Creating a Node Project:

To create a Node project, **npm init** is used in the folder in which user want to create project. The npm command line will ask a number of questions like **name**, **license**, **scripts**, **description**, **author**, **keywords**, **version**, **main file** etc. After npm is done creating the project, a [package.json](#) file will be visible in project folder as a proof that the project has been initialized.

```
Parikshit@DESKTOP-F3EF709 MINGW64 ~/Desktop/GeeksForGeeks
$ 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: (geeksforgeeks)
version: (1.0.0)
description: geeksforgeeks example
entry point: (index.js)
test command:
git repository:
keywords: gfg
license: (ISC) MIT
About to write to C:\Users\Parikshit\Desktop\GeeksForGeeks\package.json:
{
  "name": "geeksforgeeks",
  "version": "1.0.0",
  "description": "geeksforgeeks example",
  "main": "index.js",
  "dependencies": {
    "events": "^2.0.0",
    "node-color": "^1.1.0"
  },
  "devDependencies": {},
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "gfg"
  ],
  "author": "Parikshit Hooda",
  "license": "MIT"
}

Is this ok? (yes)
Parikshit@DESKTOP-F3EF709 MINGW64 ~/Desktop/GeeksForGeeks
```

Installing Packages:

After creating the project, next step is to incorporate the packages and modules to be used in the Node Project. To install packages and modules in the project use the following syntax:

Syntax to install Node Package:

```
npm install package_name
```

Example: Installing the express package into the project. Express is the web development framework used by the Node

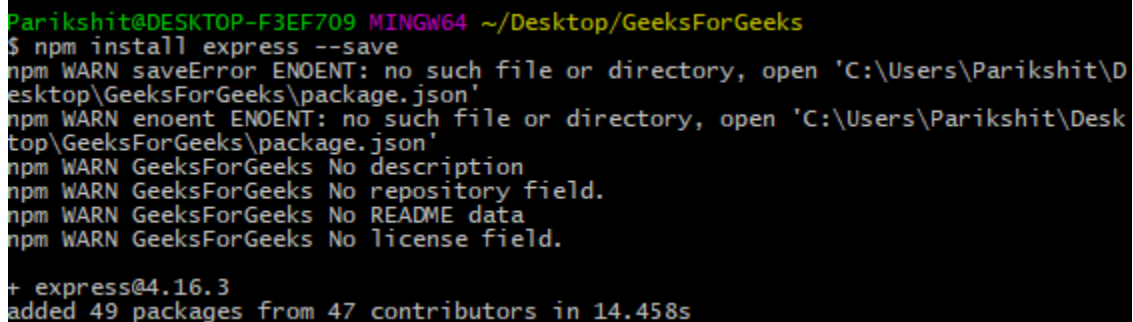
```
npm install express
```

Using a package in Node

To use express in the Node, follow the below syntax:

Syntax to use Installed Packages:

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

A terminal window screenshot showing the command 'npm install express --save' being executed. The output includes several warnings from npm about missing fields in the package.json file (description, repository, README, license) and a final message indicating that express@4.16.3 was installed along with 49 other packages from 47 contributors in 14.458 seconds.

```
Parikshit@DESKTOP-F3EF709 MINGW64 ~/Desktop/GeeksForGeeks
$ npm install express --save
npm WARN saveError ENOENT: no such file or directory, open 'C:\Users\Parikshit\Desktop\GeeksForGeeks\package.json'
npm WARN enoent ENOENT: no such file or directory, open 'C:\Users\Parikshit\Desktop\GeeksForGeeks\package.json'
npm WARN GeeksForGeeks No description
npm WARN GeeksForGeeks No repository field.
npm WARN GeeksForGeeks No README data
npm WARN GeeksForGeeks No license field.

+ express@4.16.3
added 49 packages from 47 contributors in 14.458s
```

Installing express module

Installing a Package Globally

To install a package globally (accessible by all projects in system), add an extra **-g** tag in syntax used to install the package. Installing **nodemon** package globally.

Syntax to Install Packages Globally:

```
npm install nodemon -g
```

```

Parikshit@DESKTOP-F3EF709 MINGW64 ~/Desktop/GeeksForGeeks
$ npm install nodemon -g
C:\Users\Parikshit\AppData\Roaming\npm\nodemon -> C:\Users\Parikshit\AppData\Roaming\npm\node_modules\nodemon\bin\nodemon.js

> nodemon@1.17.2 postinstall C:\Users\Parikshit\AppData\Roaming\npm\node_modules\nodemon
> node -e "console.log('\u001b[32mLove nodemon? You can now support the project via the open collective:\u001b[22m\u001b[39m\n > \u001b[96m\u001b[1mhttps://opencollective.com/nodemon/donate\u001b[0m\n');" || exit 0

Love nodemon? You can now support the project via the open collective:
> https://opencollective.com/nodemon/donate

npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.1.3 (node_modules\nodemon\node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.1.3: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":"x64"})

+ nodemon@1.17.2
added 2 packages from 1 contributor, removed 9 packages and updated 10 packages in 55.69s

```

Installing nodemon package globally

Controlling where the package gets installed:

To install a package and simultaneously save it in [package.json](#) file (in case using Node.js), add **–save** flag. The **–save** flag is default in npm install command so it is equal to **npm install package_name** command.

Example:

```
npm install express --save
```

Usage of Flags:

- **–save:**
flag one can control where the packages are to be installed.
- **–save-prod :**
Using this packages will appear in Dependencies which is also by default.
- **–save-dev :**
Using this packages will get appear in dev Dependencies and will only be used in the development mode.

***Note:** If there is a package.json file with all the packages mentioned as dependencies already, just type npm install in terminal*

Save Directory of Installed Modules

NPM installs the dependencies in local mode (Default) which go to the **node modules** directory present in the folder of Node application. To see all the locally installed modules use **npm ls** command.

Uninstalling Packages:

To uninstall packages using npm, follow the below syntax:

Syntax to uninstall packages:

```
npm uninstall
```

Example: To uninstall the express package

```
Parikshit@DESKTOP-F3EF709 MINGW64 ~/Desktop/GeeksForGeeks
$ npm uninstall express
npm WARN saveError ENOENT: no such file or directory, open 'C:\Users\Parikshit\Desktop\GeeksForGeeks\package.json'
npm WARN enoent ENOENT: no such file or directory, open 'C:\Users\Parikshit\Desktop\GeeksForGeeks\package.json'
npm WARN GeeksForGeeks No description
npm WARN GeeksForGeeks No repository field.
npm WARN GeeksForGeeks No README data
npm WARN GeeksForGeeks No license field.

removed 1 package in 0.624s
```

Uninstalling express

Syntax to uninstall Global Packages:

```
npm uninstall package_name -g
```

Project using Template engine

Templating engines are used to remove the cluttering of our server code with HTML, concatenating strings wildly to existing HTML templates. Pug is a very powerful templating engine which has a variety of features including filters, includes, inheritance, interpolation, etc. There is a lot of ground to cover on this.

To use Pug with Express, we need to install it,

```
npm install --save pug
```

Now that Pug is installed, set it as the templating engine for your app. You **don't** need to 'require' it. Add the following code to your **index.js** file.

```
app.set('view engine', 'pug');
app.set('views', './views');
```

Now create a new directory called views. Inside that create a file called **first_view.pug**, and enter the following data in it.

```
doctype html
html
  head
    title = "Hello Pug"
```

```
body
  p.greetings#people Hello World!
```

To run this page, add the following route to your app –

```
app.get('/first_template', function(req, res){
  res.render('first_view');
});
```

You will get the output as – **Hello World!** Pug converts this very simple looking markup to html. We don't need to keep track of closing our tags, no need to use class and id keywords, rather use '.' and '#' to define them. The above code first gets converted to –

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello Pug</title>
  </head>

  <body>
    <p class = "greetings" id = "people">Hello World!</p>
  </body>
</html>
```

Pug is capable of doing much more than simplifying HTML markup.

Important Features of Pug

Let us now explore a few important features of Pug.

Simple Tags

Tags are nested according to their indentation. Like in the above example, **<title>** was indented within the **<head>** tag, so it was inside it. But the **<body>** tag was on the same indentation, so it was a sibling of the **<head>** tag. We don't need to close tags, as soon as Pug encounters the next tag on same or outer indentation level, it closes the tag for us. To put text inside of a tag, we have 3 methods –

- **Space separated**
 - h1 Welcome to Pug
 - **Piped text**
- div

| To insert multiline text,

| You can use the pipe operator.

- **Block of text**

div.

But that gets tedious if you have a lot of text. You can use "." at the end of tag to denote block of text. To put tags inside this block, simply enter tag in a new line and indent it accordingly.

Comments

Pug uses the same syntax as **JavaScript**(//) for creating comments. These comments are converted to the html comments(<!--comment-->). For example,

//This is a Pug comment This comment gets converted to the following.

<!--This is a Pug comment-->

Attributes

To define attributes, we use a comma separated list of attributes, in parenthesis. Class and ID attributes have special representations. The following line of code covers defining attributes, classes and id for a given html tag.

div.container.column.main#division(width = "100", height = "100") This line of code, gets converted to the following. – <div class = "container column main" id = "division" width = "100" height = "100"></div>

Passing Values to Templates

When we render a Pug template, we can actually pass it a value from our route handler, which we can then use in our template. Create a new route handler with the following.

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

app.get('/dynamic_view', function(req, res){
  res.render('dynamic', {
    name: "TutorialsPoint",
    url: "http://www.tutorialspoint.com"
  });
});

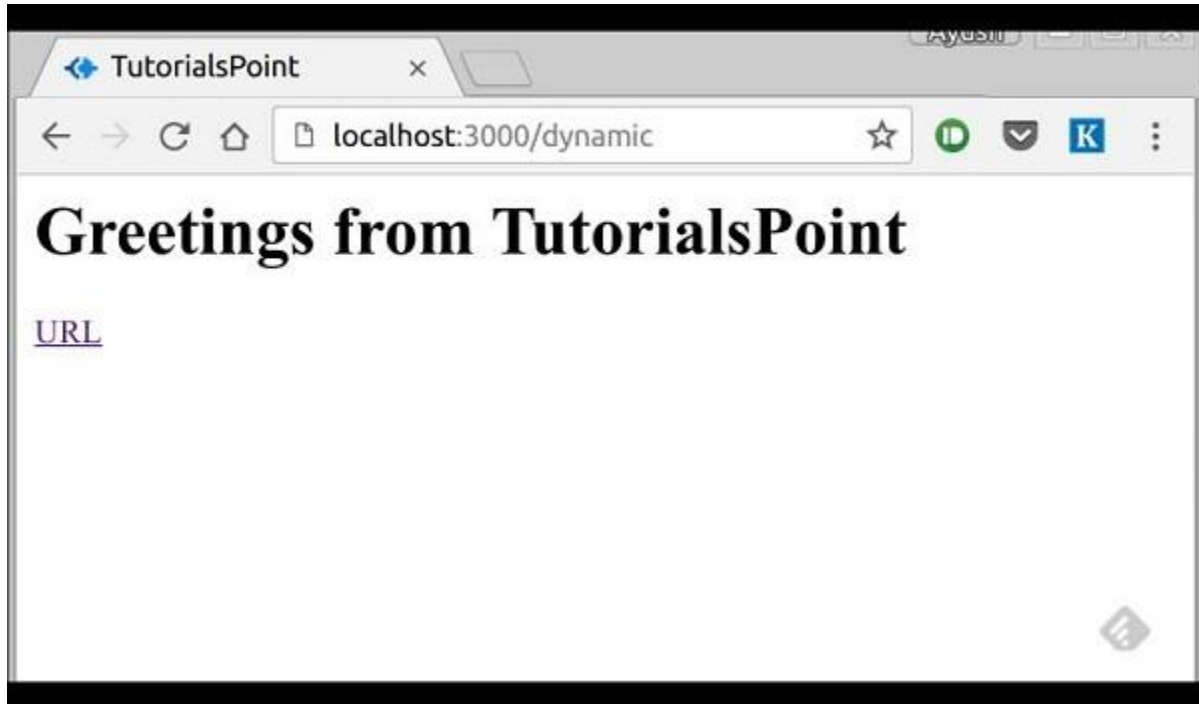
app.listen(3000);
```

We can also use these passed variables within text. To insert passed variables in between text of a tag, we use **#{variableName}** syntax. For example, in the above example, if we wanted to put Greetings from TutorialsPoint, then we could have done the following.

```
html
  head
    title = name
  body
    h1 Greetings from #{name}
```


`a(href = url) URL`

This method of using values is called **interpolation**. The above code will display the following output. –



Conditionals

We can use conditional statements and looping constructs as well.

Consider the following –

If a User is logged in, the page should display "**Hi, User**" and if not, then the "**Login/Sign Up**" link. To achieve this, we can define a simple template like –

```
html
  head
    title Simple template
  body
    if(user)
      h1 Hi, #{user.name}
    else
      a(href = "/sign_up") Sign Up
```

When we render this using our routes, we can pass an object as in the following program –

```
res.render('/dynamic',{
  user: {name: "Ayush", age: "20"}
});
```

You will receive a message – **Hi, Ayush**. But if we don't pass any object or pass one with no user key, then we will get a signup link.

Include and Components

Pug provides a very intuitive way to create components for a web page. For example, if you see a news website, the header with logo and categories is always fixed. Instead of copying that to every view we create, we can use the **include** feature. Following example shows how we can use this feature –

Create 3 views with the following code –

HEADER.PUG

```
div.header.
```

I'm the header for this website.

CONTENT.PUG

```
html
```

```
  head
```

```
    title Simple template
```

```
  body
```

```
    include ./header.pug
```

```
    h3 I'm the main content
```

```
    include ./footer.pug
```

FOOTER.PUG

```
div.footer.
```

I'm the footer for this website.

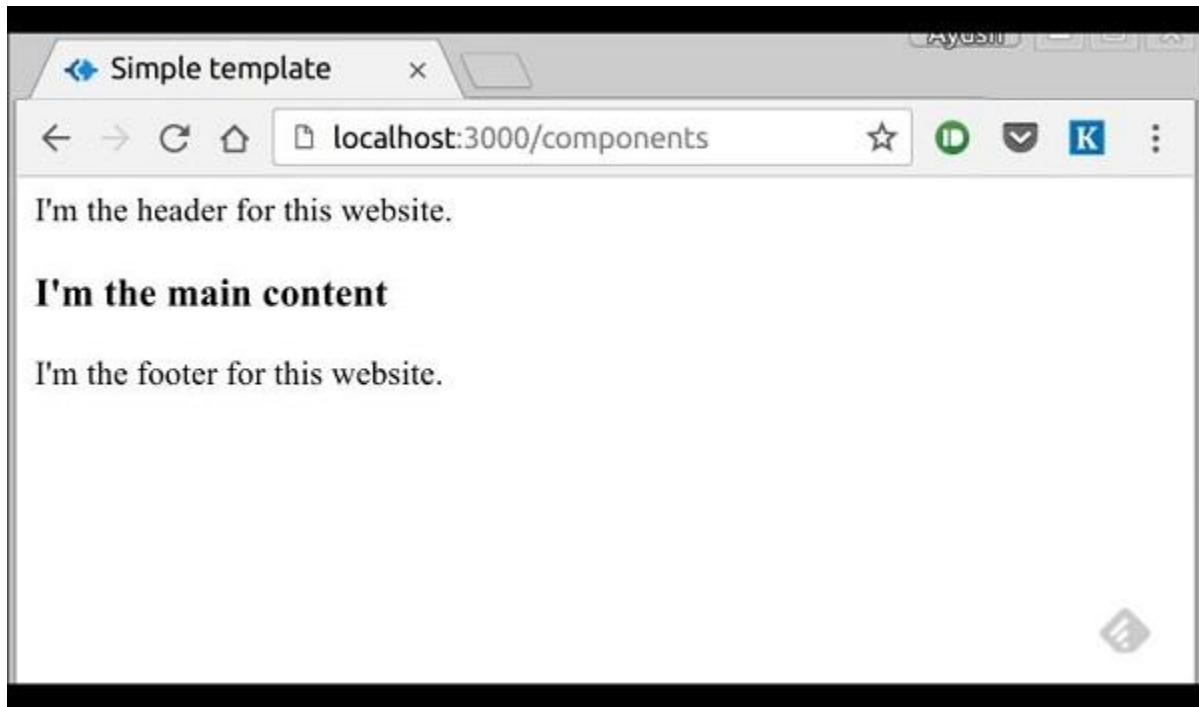
Create a route for this as follows –

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

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

```
res.render('content');  
});  
  
app.listen(3000);
```

Go to localhost:3000/components, you will receive the following output –



include can also be used to include plaintext, css and JavaScript.

There are many more features of Pug. But those are out of the scope for this tutorial. You can further explore Pug at [Pug](#).

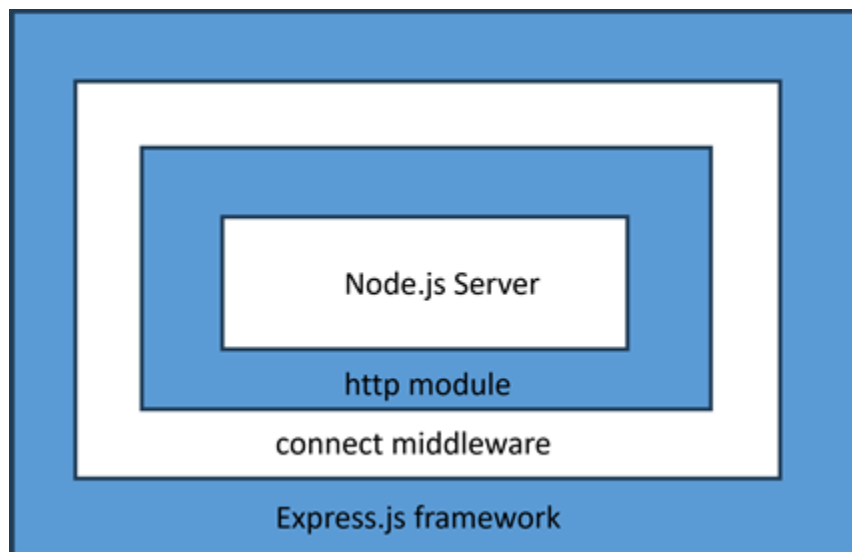
Express Setup in node.js

Express.js is a minimal and flexible web application framework that provides a robust set of features to develop Node.js based web and mobile applications. Express.js is one of the most popular web frameworks in the Node.js ecosystem. Express.js provides all the features of a modern web framework, such as templating, static file handling, connectivity with SQL and NoSQL databases. Node.js has a built-in web server. The `createServer()` method in its `http` module launches an asynchronous `http` server. It is possible to develop a web application with core Node.js features. However, all the low level manipulations of `HTTP` request and responses have to be tediously handled. The web application frameworks take care of these common tasks, allowing the developer to concentrate on the business logic of the application. A web framework such as Express.js is a set of utilities that facilitates rapid, robust and scalable web applications.

Following are some of the core features of Express framework –

- Allows to set up middlewares to respond to `HTTP` Requests.
- Defines a routing table which is used to perform different actions based on `HTTP` Method and URL.
- Allows to dynamically render `HTML` Pages based on passing arguments to templates.

The Express.js is built on top of the `connect` middleware, which in turn is based on `http`, one of the core modules of Node.js API.



Installing Express

The Express.js package is available on npm package repository. Let us install express package locally in an application folder named ExpressApp.

```
D:\expressApp> npm init
D:\expressApp> npm install express --save
```

The above command saves the installation locally in the node_modules directory and creates a directory express inside node_modules.

Hello world Example

Following is a very basic Express app which starts a server and listens on port 5000 for connection. This app responds with Hello World! for requests to the homepage. For every other path, it will respond with a 404 Not Found.

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

app.get('/', function (req, res) {
  res.send('Hello World');
})

var server = app.listen(5000, function () {
  console.log("Express App running at http://127.0.0.1:5000/");
})
```

Save the above code as index.js and run it from the command-line.

```
D:\expressApp> node index.js
Express App running at http://127.0.0.1:5000/
```

Visit <http://localhost:5000/> in a browser window. It displays the Hello World message.



Application object

An object of the top level express class denotes the application object. It is instantiated by the following statement –

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

The Application object handles important tasks such as handling HTTP requests, rendering HTML views, and configuring middleware etc.

The `app.listen()` method creates the Node.js web server at the specified host and port. It encapsulates the `createServer()` method in `http` module of Node.js API.

```
app.listen(port, callback);
```

Basic Routing

The app object handles HTTP requests GET, POST, PUT and DELETE with `app.get()`, `app.post()`, `app.put()` and `app.delete()` method respectively. The HTTP request and HTTP response objects are provided as arguments to these methods by the NodeJS server. The first parameter to these methods is a string that represents the endpoint of the URL. These methods are asynchronous, and invoke a callback by passing the request and response objects.

GET method

In the above example, we have provided a method that handles the GET request when the client visits '/' endpoint.

```
app.get('/', function (req, res) {  
  res.send('Hello World');  
})
```

- [Request Object](#) – The request object represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers, and so on.
- [Response Object](#) – The response object represents the HTTP response that an Express app sends when it gets an HTTP request. The send() method of the response object formulates the server's response to the client.

You can print request and response objects which provide a lot of information related to HTTP request and response including cookies, sessions, URL, etc.

The response object also has a sendFile() method that sends the contents of a given file as the response.

```
res.sendFile(path)
```

Save the following HTML script as index.html in the root folder of the express app.

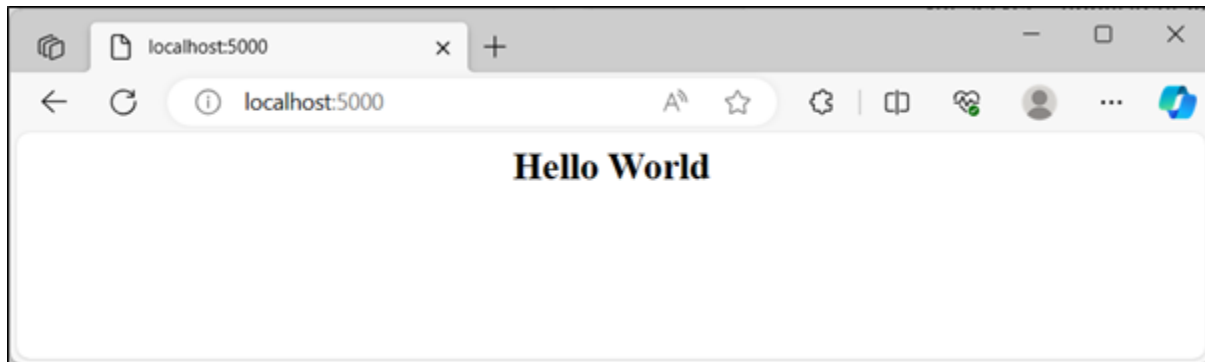
```
<html>  
<body>  
<h2 style="text-align: center;">Hello World</h2>  
</body>  
</html>
```

Change the index.js file to the code below –

```
var express = require('express');  
var app = express();  
var path = require('path');  
  
app.get('/', function (req, res) {  
  res.sendFile(path.join(__dirname, "index.html"));  
})  
  
var server = app.listen(5000, function () {
```

```
console.log("Express App running at http://127.0.0.1:5000/");
})
```

Run the above program and visit <http://localhost:5000/>, the browser shows Hello World message as below:



Let us use `sendFile()` method to display a HTML form in the `index.html` file.

```
<html>
  <body>

    <form action = "/process_get" method = "GET">
      First Name: <input type = "text" name = "first_name"> <br>
      Last Name: <input type = "text" name = "last_name"> <br>
      <input type = "submit" value = "Submit">
    </form>

  </body>
</html>
```

The above form submits the data to `/process_get` endpoint, with GET method. Hence we need to provide a `app.get()` method that gets called when the form is submitted.

```
app.get('/process_get', function (req, res) {
  // Prepare output in JSON format
  response = {
    first_name:req.query.first_name,
    last_name:req.query.last_name
```



```
};  
console.log(response);  
res.end(JSON.stringify(response));  
})
```

The form data is included in the request object. This method retrieves the data from request.query array, and sends it as a response to the client.

The complete code for index.js is as follows –

```
var express = require('express');  
var app = express();  
var path = require('path');  
  
app.use(express.static('public'));  
  
app.get('/', function (req, res) {  
  res.sendFile(path.join(__dirname, "index.html"));  
})  
  
app.get('/process_get', function (req, res) {  
  // Prepare output in JSON format  
  response = {  
    first_name:req.query.first_name,  
    last_name:req.query.last_name  
  };  
  console.log(response);  
  res.end(JSON.stringify(response));  
})  
  
var server = app.listen(5000, function () {  
  console.log("Express App running at http://127.0.0.1:5000/");  
})
```

Visit <http://localhost:5000/>.



First Name:

Last Name:

Now you can enter the First and Last Name and then click submit button to see the result and it should return the following result –

```
{"first_name":"John","last_name":"Paul"}
```

POST method

The HTML form is normally used to submit the data to the server, with its method parameter set to POST, especially when some binary data such as images is to be submitted. So, let us change the method parameter in index.html to POST, and action parameter to "process_POST".

```
<html>
  <body>

    <form action = "/process_POST" method = "POST">
      First Name: <input type = "text" name = "first_name"> <br>
      Last Name: <input type = "text" name = "last_name"> <br>
      <input type = "submit" value = "Submit">
    </form>

  </body>
</html>
```

To handle the POST data, we need to install the body-parser package from npm. Use the following command.

```
npm install body-parser --save
```

This is a node.js middleware for handling JSON, Raw, Text and URL encoded form data.

This package is included in the JavaScript code with the following require statement.

```
var bodyParser = require('body-parser');
```

The `urlencoded()` function creates application/x-www-form-urlencoded parser

```
// Create application/x-www-form-urlencoded parser
var urlencodedParser = bodyParser.urlencoded({ extended: false })
```

Add the following `app.post()` method in the express application code to handle POST data.

```
app.post('/process_post', urlencodedParser, function (req, res) {
  // Prepare output in JSON format
  response = {
    first_name:req.body.first_name,
    last_name:req.body.last_name
  };
  console.log(response);
  res.end(JSON.stringify(response));
})
```

Here is the complete code for `index.js` file

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

var bodyParser = require('body-parser');
// Create application/x-www-form-urlencoded parser
var urlencodedParser = bodyParser.urlencoded({ extended: false })

app.use(express.static('public'));

app.get('/', function (req, res) {
  res.sendFile(path.join(__dirname, "index.html"));
})

app.get('/process_get', function (req, res) {
  // Prepare output in JSON format
  response = {
```

```

    first_name:req.query.first_name,
    last_name:req.query.last_name
  };
  console.log(response);
  res.end(JSON.stringify(response));
})
app.post("/process_post", )
var server = app.listen(5000, function () {
  console.log("Express App running at http://127.0.0.1:5000/");
})

```

Run index.js from command prompt and visit <http://localhost:5000/>.



The image shows a simple web form on a light gray background. It contains two text input fields. The first field is preceded by the label 'First Name:' and the second by 'Last Name:'. Below these fields is a button labeled 'Submit'.

Now you can enter the First and Last Name and then click the submit button to see the following result –

```

{"first_name":"John","last_name":"Paul"}

```

Serving Static Files

Express provides a built-in middleware `express.Static` to serve static files, such as images, CSS, JavaScript, etc.

You simply need to pass the name of the directory where you keep your static assets, to the `express.Static` middleware to start serving the files directly. For example, if you keep your images, CSS, and JavaScript files in a directory named `public`, you can do this –

```

app.use(express.static('public'));

```

We will keep a few images in `public/images` sub-directory as follows –

```

node_modules

```

```
index.js
public/
public/images
public/images/logo.png
```

Let's modify "Hello Word" app to add the functionality to handle static files.

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

app.get('/', function (req, res) {
  res.send('Hello World');
})

var server = app.listen(5000, function () {
  console.log("Express App running at http://127.0.0.1:5000/");
})
```

Save the above code in a file named index.js and run it with the following command.

```
D:\expressApp> node index.js
```

Now open <http://127.0.0.1:5000/images/logo.png> in any browser and see observe following result.

Project using Simple Node Server

the first application to write is a Hello World program. Such a program displays the Hello World message. This illustrates the basic syntax of the language and also serves to test whether the installation of the language compiler has been correctly done or not. In this chapter, we shall write a Hello World application in Node.js.

Console Application

Node.js has a command-line interface. Node.js runtime also allows you to execute JavaScript code outside the browser. Hence, any JavaScript code can be run in a command terminal with Node.js executable.

Save the following single line JavaScript as hello.js file.

```
console.log("Hello World");
```

Open a powershell (or command prompt) terminal in the folder in which hello.js file is present, and enter the following command –

```
PS D:\nodejs> node hello.js
Hello World
```

The Hello World message is displayed in the terminal.

Creating Node.js Application

To create a "Hello, World!" web application using Node.js, you need the following three important components –

- **Import required modules** – We use the require directive to load Node.js modules.
- **Create server** – A server which will listen to client's requests similar to Apache HTTP Server.
- **Read request and return response** – The server created in an earlier step will read the HTTP request made by the client which can be a browser or a console and return the response.

Step 1 - Import Required Module

We use the require directive to load the http module and store the returned HTTP instance into an http variable as follows –

```
var http = require("http");
```

Step 2 - Create Server

We use the created http instance and call http.createServer() method to create a server instance and then we bind it at port 3000 using the listen method associated with the server instance. Pass it a function with parameters request and response.

The createserver() method has the following syntax –

```
http.createServer(requestListener);
```

The requestlistener parameter is a function that executes whenever the server receives a request from the client. This function processes the incoming request and forms a server response.

The requestlistener function takes request HTTP request and response objects from Node.js runtime, and returns a ServerResponse object.

```
listener = function (request, response) {
  // Send the HTTP header
  // HTTP Status: 200 : OK
```

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

// Send the response body as "Hello World"
response.end('<h2 style="text-align: center;">Hello World</h2>');
};
```

The above function adds the status code and content-type headers to the ServerResponse, and Hello World message.

This function is used as a parameter to `createserver()` method. The server is made to listen for the incoming request at a particular port (let us assign 3000 as the port).

Step 3 - Testing Request & Response

Write the sample implementation to always return "Hello World". Save the following script as `hello.js`.

```
http = require('node:http');
listener = function (request, response) {
  // Send the HTTP header
  // HTTP Status: 200 : OK
  // Content Type: text/html
  response.writeHead(200, {'Content-Type': 'text/html'});

  // Send the response body as "Hello World"
  response.end('<h2 style="text-align: center;">Hello World</h2>');
};

server = http.createServer(listener);
server.listen(3000);

// Console will print the message

console.log('Server running at http://127.0.0.1:3000/');
```

In the PowerShell terminal, enter the following command.

```
PS D:\nodejs> node hello.js
```

```
Server running at http://127.0.0.1:3000/
```

The program starts the Node.js server on the localhost, and goes in the listen mode at port 3000. Now open a browser, and enter `http://127.0.0.1:3000/` as the URL. The browser displays the Hello World message as desired.