UNIT-5: Express JS

View – Engine [PUG]

A **template engine** enables you to use static template files in your application. At runtime, the template engine replaces variables in a template file with actual values, and transforms the template into an HTML file sent to the client. This approach makes it easier to design an HTML page.
Some popular template engines that work with Express are **Pug**, **Mustache**, **and EJS etc**

Pug is a very powerful templating engine which has a variety of features including **filters, includes, inheritance, interpolation.**

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

```
npm install 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 **.js** file.

```
app.set('view engine', 'pug');
app.set('views','__dirname); (* In case same folder)
```

PUG, is a JavaScript library that was previously known as **JADE**. It is an easy-to-code template engine used to code **HTML** in a more readable fashion. One upside to *PUG* is that it equips developers to code reusable *HTML* documents by pulling data dynamically from the API.

PUG syntax is sensitive to indentation/whitespace . Extension of the pug file is **".pug"**

- **Tags**

Tags in Pug are used to define elements in the HTML output. They are similar to HTML tags but are written in a concise and indentation-based format, making Pug code more readable and less verbose compared to traditional HTML.

```
doctype html
html
 head
   title My Pug Page
 body
   h1 Welcome to Pug
   p This is a paragraph.

// In this example, html, head, body, h1, and p
are all tags used to structure the HTML document.
```

- ## Tag Attributes

Tags can have attributes just like in HTML. Attributes are specified within parentheses and can include dynamic values using template interpolation. Simply write the attributes inside the parenthesis and separate multiple

attributes with either a space ' ' or a comma ,

Bear in mind that the actual value of the attribute is a Javascript expression

Attributes can span over multiple lines as well

```
a(href="/about", title="About Us") About
img(src="/images/logo.png", alt="Logo")

// Here, href, title, src, and alt are attributes of a and img tags, respectively.
```

- ## Plain text

Plain text in Pug can be retrieved via a variety of methods, each with its own syntax and advantages.

```
// Unescaped Text using '|'
    p
        | The pipe always goes at the beginning of its own line,
        | not counting indentation.

// Block Expansion using '.'
  p.
    This is a long paragraph of plain text.
    It spans multiple lines without indentation.
```

- ## Code

Pug allows you to write inline JavaScript code in your templates. There are three types of code: Unbuffered, Buffered, and Unescaped Buffered.

Unbuffered Code vs. Buffered Code

- **Unbuffered Code** starts with a hyphen - and does not directly output anything. It can be used in the *PUG* code later to make changes.
- **Buffered Code** starts with =. If there is an expression, it will evaluate it and output the result.

```
o        // Unbuffered Code
o        -var number = 4
o        // Buffered Code
o        h4= "3 times number is: " + number*3 //Output is 12 in h4 tag
```

- **Comments**
  Buffered comments look the same as single-line JavaScript comments. They act sort of like markup tags, producing *HTML* comments in the rendered page.

  Like tags, buffered comments must appear on their own line.

- **Buffered Comments vs. Unbuffered Comments vs. Multiline Comments (*After Running code → on web page → right click → open view page source code)**

  - **Buffered Comments** are added using a double forward-slash(//). They appear in the rendered *HTML* file.
  - **Unbuffered Comments** are added using the double forward-slash followed by a hyphen(//-). They do not appear in the rendered *HTML* file.
  - **Multiline Comments** are added using a double forward-slash // followed by an indented block.

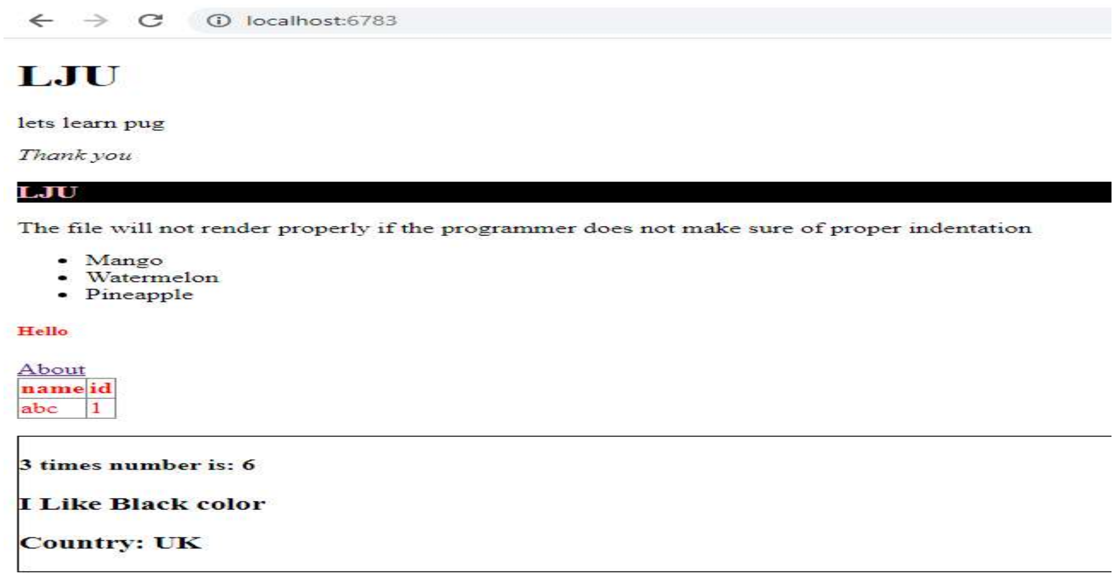
Example-1

Simple example to understand the concept of pug (**public/one.pug**)

```
doctype html
html(lang="en")
head
  title PUG Tutorial
 body
    //1
    h1 LJU
    //2
    p lets learn pug
    //3
    p
      i Thank you
    //4
    h5(style="text-transform:uppercase;color:pink") Lju
    //5
    p
      |The file will not
      |render properly if the
      |programmer does not make
      |sure of proper indentation
      //6
      ul
         li Mango
         li Watermelon
  li Pineapple

// 7 single attribute
h5(style="color:red") Hello

// 8 Attributes spanning over multiple lines
a(
  href='/about'
  target="_blank"
) About

// 9 Multiple attributes separated by a comma
table(border='1px solid',style='border-collapse:collapse;color:red')
  tr
    th name
    th id
  tr
    td abc
    td 1

// Unbuffered Code
-var number = 2
-var color = 'Black'
-var list = ["India", "USA", "UK"]

// 10 Buffered Code
h4= "3 times number is: " + number*3
h3= "I Like " + color + " color"
h3= "Country: " + list[2]
```

src/pug.js

```
var expr = require("express");
var app = expr();
app.set("view engine","pug");
var p = require("path");
const staticp = p.join(__dirname,"../public");
app.get("/",(req,res)=>{
   res.render(staticp+"/one");
});
app.get("/about",(req,res)=>{
   res.write("<h1>Welcome to about page</h1>");
   res.send();
});
app.listen(6783);
```



localhost:6783

LJU

lets learn pug

Thank you

LJU

The file will not render properly if the programmer does not make sure of proper indentation

- Mango
- Watermelon
- Pineapple

Hello

About

name	id
abc	1

3 times number is: 6

I Like Black color

Country: UK

Example-2

Create one pug file which contains a text field and a email field. By submitting the form, on next page called “/data” submitted data will be displayed.

/src/pug.js

```
var expr = require("express");
var app = expr();
app.set("view engine","pug");
var p = require("path");
const staticp = p.join(__dirname,"../public");
app.get("/",(req,res)=>{
   res.render(staticp+"/one");
});
app.get("/data",(req,res)=>{
   res.set("content-type","text/html")
   res.write("<h2>Welcome " + req.query.name + "</h2>");
   res.write("<h3>Your Email Id is :  " + req.query.email + "</h3>");
   res.send();
});
app.listen(6785);
```

/public/one.pug

```
html(lang="en")
   head
      title Pug Form
   body
         h2 Form
         form(action="/data",method="get")
```

```
        div
          label Enter Your Name
          input(type="text", name="name")
        div
          label Enter Your Email
          input(type="email", name="email")
        div
          input(type="submit",value="Submit")
```

Example-3

Write express JS script to pass data like message, name and id from express application to pug template in h1, h2 and h3 tags respectively and display data in browser. (Both file in same folder)

Pug.js

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

app.set('view engine', 'pug');
app.get('/', (req, res) => {
  res.render(__dirname+'/two', {message: 'Hello from Express!',name: 'lju',id: 2  });
});
app.listen(6002)
```

two.pug (same folder)

```
html
  head
    title My Express App
  body
    h1 #{message}
    h2 #{name}
    h3(style="color:red") #{id}
```

Example:4

Write express js script to load student form using pug file which contains following fields
Name(text)
Email(email)
Course(radio : CE, IT, CSE)
Once form submitted then data must be displayed on '/data' page using pug file. Means data should be submitted from express application to PUG file.

form.pug(public folder)

```
html
  head
    title Pug Form
  body
    h2 Student Form
    form(action="/data",method="get")
      div
        label Enter Your Name
        input(type="text", name="name")
      div
        label Enter Your Email
        input(type="email", name="email")
      div
        label Course
          input(type="radio", name="course", value="IT",id="IT")
          |IT
          input(type="radio", name="course", value="CE",id="CE")
          |CE
          input(type="radio", name="course", value="CSE",id="CSE")
          |CSE
      div
        input(type="submit",value="Submit")
```

form_output.pug (public folder)

```
html
 head
  title FORM Data
 body
  h1 Welcome!
  table(border='1px solid',style='border-collapse:collapse;color:blue', cellpadding=10)
    tr
      th Name
      th Email
      th Course
    tr
      td #{name}
```

```
        td #{email}
        td #{course}
```

pug_form.js

```
var expr = require("express");
var app = expr();
app.set("view engine","pug");
var p = require("path");
const staticp = p.join(__dirname,"../public");
app.get("/",(req,res)=>{
    res.render(staticp+"/form");
});
app.get("/data",(req,res)=>{
    res.render(staticp+"/form_output",{name:req.query.name, email:req.query.email,
course:req.query.course});
});

app.listen(6785);
```

File Upload

Multer is a node.js middleware for handling **multipart/form-data**, which is primarily used for uploading files. Multer adds a file or files object to the request object. The file or files object contains the data of files uploaded via the form.

Installation
Npm install multer

What is Multipart Data?

The ENCTYPE attribute of <form> tag specifies the method of encoding for the form data. It is one of the two ways of encoding the HTML form. It is specifically used when file uploading is required in HTML form. It sends the form data to server in multiple parts because of large size of file.

- In general, when a “form” is submitted, browsers use “application-x-www-form-urlencoded” content-type.
- This type contains only a list of keys and values and therefore are not capable of uploading files.
- Whereas, when you configure your form to use “multipart/form-data” content-type, browsers will create a “multipart” message where each part will contain a field of the form. A multipart message will consist of text input and file input. This way using multipart/form-data you can upload files.

Multer adds a body object and a file or files object to the request object. The body object contains the values of the text fields of the form, the file or files object contains the files uploaded via the form.

Multer will not process any form which is not multipart (*multipart/form-data). *Must define in form*

****File information:** Each file contains the following information:

Key	Description
fieldname	Field name specified in the form
originalname	Name of the file on the user’s computer
encoding	Encoding type of the file
mimetype	Mime type of the file (Multipurpose Internet Mail Extensions)
size	Size of the file in bytes
destination	The folder to which the file has been saved
filename	The name of the file within the destination
path	The full path to the uploaded file

DiskStorage() method

The disk storage engine gives you full control on storing files to disk.

There are two options available, destination and filename. They are both functions that determine where the file should be stored.

- **Destination:** It is used to determine within which folder the uploaded files should be stored. This can also be given as a string (e.g. '/tmp/uploads'). If no destination is given, the operating system's default directory for temporary files is used.
- **Filename:** It is used to determine what the file should be named inside the folder. If no filename is given, each file will be given a random name that doesn't include any file extension.

```
var storage = multer.diskStorage({
   destination:"single",
   filename: function (req, file, cb) {
                  cb(null, file.originalname)
         }
})
```

The 2 arguments in cb are

- **null** - as we don't want to show any error.
- **file.originalname** - here, we have used the same name of the file as they were uploaded. You can use any name of your choice.

Optional objects of Multer(opts)

Multer accepts an options object, the most basic of which is the dest property, which tells multer where to upload the files. In case you omit the options object, the files will be kept in memory and never written to disk.

By default, Multer will rename the files so as to avoid naming conflicts. The renaming function can be customized according to your needs.

The following are the options that can be passed to Multer.

Key	Description
dest or storage	Where to store the files
fileFilter	Function to control which files are accepted
limits	Limits of the uploaded data
preservePath	Keep the full path of files instead of just the base name

If you want more control over your uploads, you'll want to use the storage option instead of dest.

- **limits** - You can also put a limit on the size of the file that is being uploaded with the help of using limits.
  The following code will go inside the **multer()** .

```
// inside multer({}), file upto only 1MB can be uploaded
const upload = multer({
      storage: storage,
      limits : {fileSize : 1000000}
});
```

  Here, fileSize is in bytes. (1000000 bytes = 1MB)

- **fileFilter -** Set this to a function to control which files should be uploaded and which should be skipped

```
function fileFilter (req, file, cb) {
  // Allowed ext
   const filetypes = /jpeg|jpg|png|gif/;

 // Check ext
  const extname =
filetypes.test(path.extname(file.originalname).toLowerCase())
;
 // Check mime
 const mimetype = filetypes.test(file.mimetype);

 if(mimetype && extname){
    return cb(null,true);
 } else {
    cb('Error: Images Only!');
 }}
```

- **To upload Single file**

  .single(fieldname): Accept a single file with the name fieldname. The single file will be stored in req.file.

- **To upload Multiple file**

  .array(fieldname[, maxCount]) : Accept an array of files, all with the name fieldname. Optionally error out if more than maxCount files are uploaded. The array of files will be stored in req.files.

Example1: Write an express js script that accepts single file to be uploaded using the multer middleware and saves the file to the specific directory called "single".

[Both file in same folder]

In mul.html file :

```
<html>
   <form action="/uploadfile" method="post" enctype="multipart/form-data">
      <input type="file" name="mypic" accept=".jpg,.jpeg,.png"/>
      <input type="submit" value="Upload"/>
   </form>
</html>
```

In mul.js file:

```
// require the installed packages
const express = require('express')
const app = express();
const multer = require('multer');
// SET STORAGE
var store = multer.diskStorage({
   destination:"single",
   filename: (req, file, cb) =>{
     cb(null, file.originalname)
   }
})
var upload = multer({ storage: store }).single('mypic')

//CREATE EXPRESS APP
app.use(express.static("./",{index:"mul.html"}))

app.post('/uploadfile', upload, (req, res) => {
  if (req.file) {
     res.send("<h1>File <span style='color:red'>"+ req.file.originalname + "</span> has been uploaded in <span style='color:red'>" + req.file.destination + " </span>folder")
   }
})
app.listen(6788)
```

Example 2 : Write an express js script that accepts multiple files max number 5 to be uploaded using the multer middleware and saves the files to the specific directory called “multiple”. With specific file name.

[Both file in same folder]

In Mul2.html

```
<html>
   <form action="/uploadfile" method="post" enctype="multipart/form-data">
      <input type="file" name="myfile" accept=".doc,.docx" multiple/>
      <input type="submit" value="Upload"/>
   </form>
</html>
```

In mul2.js

```
// require the installed packages
const express = require('express')
const multer = require('multer');
const app = express();
// SET STORAGE
var store = multer.diskStorage({
  destination:"multiple",
  filename: function (req, file, cb) {
    cb(null, file.fieldname+"-"+Date.now()+".docx") // fieldname will change  with file name
“myfile-the number of milliseconds since January 1, 1970”
  }
})
var upload = multer({ storage: store }).array('myfile',5)

app.use(express.static("./",{index:"mul2.html"}))

app.post('/uploadfile', upload, (req,res) => {
    if (req.files) {
      res.set("content-type","text/html")
      for(i of req.files){
        res.write("<h2>File <span style='color:red'>" + JSON.stringify(i.originalname) + "</span>
has been uploaded </h2>")
      }
  res.send()
  }})
app.listen(6788);
```

RESTful APIs

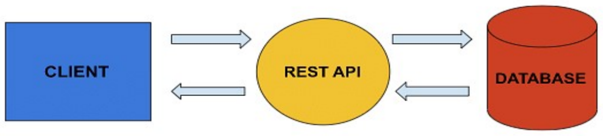
RESTful APIs

REST (**Representational State Transfer)** API is the standard way to send and receive data for web services.

Let's understand the meaning of each word in the REST acronym.

- **REpresentational** means formats (such as XML, JSON, YAML, HTML, etc)
- **State** means data
- **Transfer** means carrying data between consumer and provider using the HTTP protocol

A client sends a req which first goes to the rest API and then to the database to get or put the data after that, it will again go to the rest API and then to the client. Using an API is just like using a website in a browser, but instead of clicking on buttons, we write code to req data from the server. It's incredibly adaptable and can handle multiple types of requests.

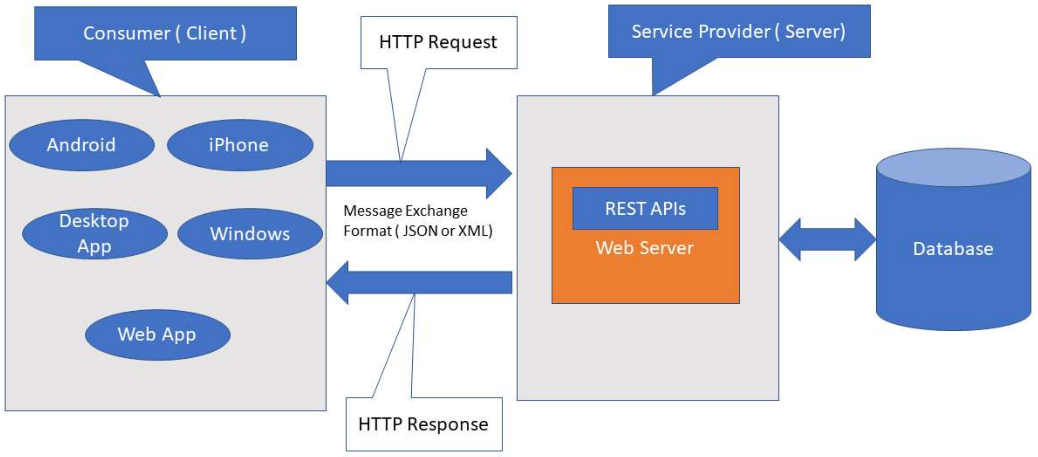


REST (Representational state transfer) is a popular architecture that is used to create web services. Express is a back-end web application framework of node js, and with the help of express, we can create an API very easily.

API (Application Programming Interface) is a code that allows two software programs to communicate with each other.

REST API is a software that allows two apps to communicate with one another over the internet and through numerous devices.

REST Architecture



- **Request and Response:** Request is the input to a web service, and the response is the output from a web service.
- **Message Exchange Format:** It is the format of the request and response. There are two popular message exchange formats: XML and JSON.
- **Service Provider or Server:** The service provider is one that hosts the web service.
- **Service Consumer or Client:** A service consumer is one who is using a web service.

In simple words A RESTful API is an architectural style for an application program interface (API) that uses HTTP requests to access and use data.

What Is express.Router() For

Express Routers are a way to organize your Express application such that your primary app.js file does not become bloated.
To create an instance of an Express Router, we call the **.Router()** method on the top-level Express import. Then to use that router, we mount it at a certain path using **app.use()** and pass in the router as the second argument. This router willnow be used for all paths that begin with that path segment. To create a router to handle all requests beginning with **/hello**, the code would look like this

```
const helloRouter = express.Router();
app.use('/hello', helloRouter);
```

We will be using JSON as our transport data format as it is easy to work. we are using Router from Express, and we are exporting it using **module.exports**. So, our app will work fine.

In below example **movies.js** is serve as **provider** and **index.js** is serve as **consumer**

// [waiter in hotel is API(application programming interface)
// ,we are client, server is chef]

In Index.js (same folder)

```
const expr=require("express")
const app=expr()
const m1=require("./data")
app.use("/p",m1)
app.listen(3015)
```

In data.js

```
const expr=require("express")
const router=expr.Router()

const mvi = [{id:101,name:"DON-1",Year:2013,rating:9.5},
{id:102,name:"DON-2",Year:2016,rating:9.5},
{id:103,name:"DON-3",Year:2103,rating:100}]
```

```
module.exports=router
router.get("/",(req,res)=>
{
   res.json(mvi)
})
```

//for specific id

```
router.get("/:id([0-9]{3,})",(req,res)=>
{
   var currmovie=mvi.filter((m)=>
   {
      if(m.id==req.params.id)
      {
         return true
      }
   })
 if(currmovie.length==1)
 {
   res.json(currmovie[0])
 }
 else{
   res.json("not found")
 }
})
```

**Run node index.js in terminal and observe output on :
localhost:3015/p for all movie list and
Localhost:3015/p/103 for specific movie

Mail send-node mailer

Nodemailer is a module for Node.js applications to allow easy email sending.

Install nodemailer using the following command:

```
npm install nodemailer
```

var trans = nodemailer.createTransport (options[, defaults])

- This line declares a variable trans and assigns it the result of a function call to createTransport() method of the nodemailer object.
- This indicates that there is likely an imported or required module named nm which provides this functionality.
- The purpose of this line is to create an email transport object for sending emails.

Where,

- options – is an object that defines connection data (see below for details)
- defaults – is an object that is going to be merged into every message object.

This allows you to specify shared options, for example to set the same from address for every message.

General options

- **service:** "gmail": This specifies the email service provider being used, which in this case is Gmail.

- **host: "smtp.gmail.com":** This specifies the SMTP (Simple Mail Transfer Protocol) server host for Gmail, which is smtp.gmail.com.

- **port: 587:** This specifies the port number through which the connection to the SMTP server will be made. Port 587 is commonly used for secure SMTP (SMTPS).

- **secure**: true: This indicates that the connection to the SMTP server should be secure. This typically means using TLS (Transport Layer Security) to encrypt the communication.

- **auth: { ... }:** This object contains authentication credentials required to log in to the SMTP server and send emails.
  - **user: "prof.Priyen@gmail.com":** This specifies the username or email address used for authentication.
  - **pass: "cgpawoohgfoddhwwe":** This specifies the App password for the email account being used for authentication. This App password is provided in plain text, which is not recommended for security reasons. It's safer to use environment variables or other secure methods to store sensitive information like passwords.

- **.sendMail(mailOptions, callback)**

The .sendMail() method in nodemailer is used to send an email using the specified transport configuration. Here's a detailed explanation of the method and its parameters.

Parameters

1. **mailOptions:** An object containing the details of the email to be sent. The mailOptions object can include the following properties:
   - **from:** The sender's email address. This can be a simple email address or an address with a name. (from: 'your-email@gmail.com' // or 'Your Name <your-email@gmail.com>')
   - **to:** The recipient's email address. This can be a single email address or a comma-separated list of addresses. (to: 'recipient1@example.com, recipient2@example.com')
   - **cc: (Optional**) Carbon copy recipients. This can be a single email address or a comma-separated list of addresses. (cc: 'cc-recipient@example.com')
   - **bcc: (Optional)** Blind carbon copy recipients. This can be a single email address or a comma-separated list of addresses. (bcc: 'bcc-recipient@example.com')
   - **subject**: The subject line of the email.(subject: 'Subject of your email')
   - **text:** The plain text body of the email.(text: 'Hello, this is the plain text body of the email')
   - **html: (Optional)** The HTML body of the email.(html: '<p>Hello, this is the <b>HTML</b> body of the email</p>')

2. **Callback Function:** A function to handle the response once the email has been sent or an error has occurred. It takes two arguments:

   - **error:** An error object if an error occurred, or null if no error occurred.
   - **info:** An object containing information about the sent message, such as the message ID and response.

Simple example to send mail using nodemailer.

Example

mailer.js

```
var nm = require("nodemailer");

var trans = nm.createTransport({
   host : "smtp.gmail.com",
   port:465,
   auth:{
      user : "sender@gmail.com",
      pass : "2stepverificationkey"
   }
});
var mailoption = {
   from:"sender@gmail.com",
   to:"receiver1@gmail.com,receiver2@gmail.com ",
   subject : "Hello",
   // text : 'Test mail',
   html:'Testing node mailer, <h1>Effect of h1</h1>'

};
trans.sendMail(mailoption,(err,info)=>{
   if(err){
      console.error(err);
   }
   console.log(info);});
```

Output: Run node mailer.js in terminal and observe

Example

Perform the following tasks as asked.

- **Create a HTML file for feedback form and this file should be loaded on home('/') page.**
- **Fields are :**
  name, email, dropdown for the feedback ,text area for the additional comments, and submit button.
- **Once feedback submitted, message "Thank you for your feedback." Will be displayed on page '/feedback' and also send mail to the entered email id with the submitted feedback data. (Data can be submitted using get/post method)**

mail_form.js

```
var expr = require("express");
const app = expr();
var nm = require("nodemailer");

app.get('/',(req,res)=>{
   res.sendFile(__dirname + '/mail.html')
})

app.get('/feedback',(req,res)=>{
   var trans = nm.createTransport({
      host : "smtp.gmail.com",
      port: 465,
      auth:{
         user : "sender@gmail.com",
         pass : "2stepverificationkey"
      }
   });
   var mailoption = {
      from:"noreply@gmail.com<noreply>",
      to:req.query.mail,
      bcc:"bcc@gmail.com",
      subject : "Feedback response",
      html: "<h1>Hello "+req.query.uname+"!</h1><h2> Thank you for submitting the feedback.<br>
Your feedback: <span style=color:red>"+req.query.feedback+"</span></h2><h3> Additional
comments: "+req.query.comments+"</h3>"
```

```
    };
    trans.sendMail(mailoption,(err,info)=>{
        if(err){
            console.error(err);
        }
        console.log(info);
    });
    res.send("Thank you for your feedback.")
})
app.listen(5001)
//gmail >> manage account >> security >> 2 step verification >> app passwords >> select "others" >>
add any name and copy password written in yellow
```

mail.html

```
<html>
    <head><title>Feedback Form</title></head>
    <body>
        <form action="/feedback" method="get">
            Name: <input type="text" name="uname"/>
            Email: <input type="text" name="mail"/>
            Feedback: <select name="feedback">
                <option value="bad">Bad</option>
                <option value="good">Good</option>
                <option value="verygood">Very Good</option>
                <option value="excellent"SS>Excellent</option>
            </select>
            Comments: <textarea name="comments" rows="10" cols="15"
placeholder="Comments"></textarea>
            <input type="submit" value="Submit"/>
        </form>
    </body>
</html>
```