

PUG

- ✓ **PUG**, 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.
- ✓ This 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 Node.js

Installation

Pug is available via npm:

```
npm install pug
```

PUG syntax is sensitive to indentation/whitespace. Extension of the pug file is “.pug”

Pug file example.

1.pug

```
html
  head
    title My Page
  body
    h1 Heading
    p My paragraph here.
```

This will get translated to the following HTML content on demand.

```
<html>
  <head>
    <title>My Page</title>
  </head>
  <body>
    <h1>Heading</h1>
    <p>My paragraph here.</p>
  </body>
</html>
```

Tags

Tags are what make *PUG* clean, readable, and easy to code. Instead of having closing tags, *PUG* relies on whitespace/indentation that is later rendered accordingly to an *HTML* doc.

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.

```

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

```

Attributes

- You can 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

Comments

Buffered Comments vs. Unbuffered Comments vs. Multiline Comments

- **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.

| Display content

- The `|` tells Pug to treat this as **raw, unwrapped text**, not a tag.

div.c1#i1

is Pug shorthand syntax for creating a `<div>` element with: `class="c1"` and `id="i1"`

Equivalent HTML:

```
<div class="c1" id="i1"></div>
```

Example-1

Simple example to understand the concept of pug

one1.pug

```
html
head
  title PUG Tutorial
body
  //1
  h1 LJU
  //2
  h1(style="color:red;font-size:35px;") lets learn pug
  //3
  p(style="font-size:25px;")
    i italic para
      b(style="color:blue") bold italic blue para
    b bold para

  //4
  p
    | The file will not
    | render properly if the
    | programmer does not make
    | sure of proper indentation
  //5
  ul
    li Mango
    li Watermelon
    li Pineapple

  // 6
  ol(type="A")
    li Janyary
    li February
    li March

  // 7 Attributes spanning over multiple lines
  a(
    href='/about'
```

```
target="_blank"
style="color:purple;font-size:20px"
) About
```

// 8 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
  tr
    td xyz
    td 2
```

// Unbuffered Code

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

// 9 Buffered Code

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

// This is a Buffered Comment

// -This is an Unbuffered Comment. It does not appear in the rendered HTML file.

```
//
  This is
  a multiline
  comment
```

backend/pug.js

```
var expr = require("express");
var app = expr();
app.set("view engine", "pug");

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

```
res.render(__dirname + "/one1");
});
app.get("/about", (req, res) => {
  res.write("<h1>Welcome to about page</h1>");
  res.send();
});
app.listen(6783);
```

LJU

lets learn pug


italic para ***bold italic blue para*** **bold para**

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

- Mango
- Watermelon
- Pineapple

A. Janyary
B. February
C. March

About



name	id
abc	1
xyz	2

3 times number is: 6

I Like Black color

Country: UK



Example

What will be the possible output of below code?

```
p(style="font-size:25px;")
  i Good morning
//
  p b LJU Students
p b hello students
p
  b
    i test

-var a="red"
h4 My favorite color is
  i(style="color:red")= a

|Display content
This
```

Output:

Good morning

b hello students

test

My favorite color is *red*

Display content

HTML file of above pug file to understand the structure

```
<p style="font-size:25px;"><i>Good morning</i></p>
<!--p b LJU Students-->
<p>b hello students</p>
<p> <b> <i>test </i></b></p>
<h4>My favourite color is <i style="color:red">red</i></h4>
Display content
<this> </this>
```

Example-2

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

Pug_form.js

```
var expr = require("express");
var app = expr();
app.set("view engine", "pug");
app.get("/", (req, res) => {
  res.render(__dirname + "/form");
});
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);
```

form.pug

```
html(lang="en")
  head
    title Pug Form
  body
    .form
      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.

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+'/index_u', {message: 'Hello from Express!',name: 'lju',id: 2 });
});
app.listen(6002)
```

index_u.pug

```
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

```
html
  head
    title Pug Form
  body
    h2 Student Form
    form(action="/data",method="post")
      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")
```

pug_form.js

```
var express = require("express");
var app = express();
app.set("view engine", "pug");
app.use(express.urlencoded())
app.get("/",(req,res)=>{
```

```
res.render(__dirname+"/form");
});
app.post("/data",(req,res)=>{
  res.render(staticp+"/form_output",{name:req.body.name, email:req.body.email,
course:req.body.course});
});

app.listen(6785);
```

form_output.pug

```
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}
```

Multer

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
```

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
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

Storage(DiskStorage)

- ✓ 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** 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** 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)
}
})

```

multer(opts)

Multer accepts an options object, the most basic of which is the **dest/storage** 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.**

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

Key	Description
dest / 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.
 - **.single(fieldname)**
 - Accept a single file with the name fieldname. The single file will be stored in **req.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**.

Create an HTML Form

Create an HTML form that allows users to select and upload a file to your server. The form will be submitted using the POST method and will have an encoding type of "multipart/form-data" to handle file upload:

To create the HTML form, you will use the **< form >** element with the following attributes:

- ✓ **action:** This attribute specifies the URL or route where the form data will be sent when the form is submitted

- ✓ **method**: This attribute specifies the HTTP method to be used when submitting the form. For file uploads, you should use the POST method, as it allows larger amounts of data to be sent. So, set the method attribute to "POST".
- ✓ **enctype**: This attribute specifies the content type used to submit the form data. For file uploads, you need to set the enctype attribute to "multipart/form-data". This encoding type is necessary for browsers to properly handle file uploads.

Inside the form, you will add an **<input>** element of type "file". This element allows users to select a file from their local machine. Give the input element a **name** attribute so that it can be identified when the form is submitted.

Finally, you can add any additional form fields or submit buttons as needed. When the form is submitted, the selected file(s) will be sent to the specified route for handling.

Set Up Multer Middleware

Multer is a middleware for handling multipart/form-data requests, specifically designed for file uploads in Node.js.

- ✓ First, import the **multer** module using **require('multer')**. This ensures that you have access to the Multer functionality.
- ✓ Next, define the storage configuration for uploaded files using **multer.diskStorage()**. This configuration determines where the uploaded files will be stored on the server. It takes an object with two functions: **destination** and **filename**.
- ✓ The destination function specifies the directory where the uploaded files will be saved.
- ✓ The filename function determines the name of the uploaded file.
- ✓ We append the original name of the file using **file.originalname** to maintain some context about the uploaded file. You can modify this function to generate filenames based on your specific needs.
- ✓ After setting up the storage configuration, you create an instance of Multer by calling **multer({ storage })**, passing in the **storage** configuration object. This creates the Multer middleware that you can use in your Express application to handle file uploads.

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**".

m1.js

```
// require the installed packages
const express = require('express')
const multer = require('multer');
//CREATE EXPRESS APP
```

```
const app = express();
app.use(express.static(__dirname,{index: 'm1.html'}));
```

// SET STORAGE

```
var store = multer.diskStorage({
  destination:"single",
  filename: function (req, file, cb) {
    cb(null, file.originalname)
  }
})
var upload = multer({ storage: store })
app.post('/uploadfile', upload.single('mypic'), (req, res) => {
  const file = req.file
  if (file) {
    res.send("<h1>File <span style='color:red'>"+ file.originalname + "</span> has been
uploaded in <span style='color:red'>" + file.destination + " </span>folder")
  }
})
app.listen(6788);
```

m1.html

```
<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>
```

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"**.

multiple.js

```
// require the installed packages
const express = require('express')
const multer = require('multer');
//CREATE EXPRESS APP
const app = express();

app.use(express.static(__dirname,{index: 'multiple.html'}));

// SET STORAGE
var store = multer.diskStorage({
  destination:"multiple",
  filename: function (req, file, cb) {
    cb(null, file.fieldname+"-"+Date.now()+".docx")
  }
})
var upload = multer({ storage: store })
app.post('/uploadfile', upload.array('mypic',5), (req,res) => {
  const files = req.files
  if (files) {
    res.set("content-type","text/html")
    for(i of files){
      res.write("<h2>File <span style='color:red'>" + JSON.stringify(i.originalname) +
"</span> has been uploaded </h2>")
    }
    res.send()
  }
})
app.listen(6788);
```

multiple.html

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

```
</form>
</html>
```

In this example, we use **Date.now()** to generate a unique timestamp for each uploaded file. It returns the number of milliseconds since January 1, 1970

limits

- ✓ Set File Size Limits
- ✓ When setting file size limits, we need to specify the maximum file size that can be uploaded. The limits option of Multer can be used to set the maximum file size in bytes.

In the example, we set the maximum file size to 1MB (1,000,000 bytes). You can change this to any value that fits your requirements.

Here's an example of how to set file size limits:

```
const upload = multer({
  storage,
  limits: {
    fileSize: 1000*1000 // 1MB (decimal)
  }
});
```

Or

```
const upload = multer({
  storage,
  limits: {
    fileSize: 1024*1024 // 1MB (binary)
  }
});
```

In this example, we set the maximum file size to 1MB.

The megabyte is one of several multipliers used to represent larger numbers of bytes. For example, a **kilobyte (KB)** is equal to **1,000 bytes (decimal)** or **1,024 bytes (binary)**. As such, a **megabyte** is equal to **1,000 KB (decimal)** or **1,024 KB (binary)**.

Example

Write express js script to upload file with size limit of 1 MB to a specific directory named "Data" on the server. And in this folder file must be stored in format of "lju-file.pdf" where "lju" is the field name.

mult1.js

```
// require the installed packages
const express = require('express')
const multer = require('multer');
//CREATE EXPRESS APP
const app = express();

app.use(express.static(__dirname,{index:'mult1.html'}))

// SET STORAGE
var storage = multer.diskStorage({
  destination:"Data",
  filename: function (req, file, cb) {
    console.log(file)
    cb(null, file.fieldname+"-"+ "File.pdf")
  }
})
var upload = multer({ storage: storage,limits:{fileSize:1024*1024} })
app.post('/uploadfile', upload.single('lju'), (req,res) => {
  const file = req.file
  res.send("<h1 style='color:red'>" + file.originalname+" has been Uploaded</h1>")
})
app.listen(6788);
```

mult1.html

```
<html>
  <form action="/uploadfile" method="post" enctype="multipart/form-data">
    <input type="file" name="lju" accept=".pdf"/>
    <input type="submit" value="Upload"/>
  </form>
</html>
```

Node mailer

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

Install nodemailer using the following command:

```
npm install nodemailer
```

```
let transporter = nodemailer.createTransport(options[, defaults])
```

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

- **port** – is the port to connect to (defaults to 587 if *secure* is *false* or 465 if *true*)
- **host** – is the hostname or IP address to connect to (defaults to *'localhost'*)
- **auth** – defines authentication data

Authentication

If authentication data is not present, the connection is considered authenticated from the start. Otherwise you would need to provide the authentication options object.

auth is the authentication object

- **user** is the username
- **pass** is the password for the user if normal login is used

To start sending emails with Nodemailer, all you need to is:

- Create a transporter object
- Configure the mailoptions object
- Deliver a message with `sendMail()`

The following are the possible fields of an email message:

- ✓ **from** - The email address of the sender. All email addresses can be plain *'sender@server.com'* or formatted *"Sender Name" sender@server.com'*, see Address object for details
- ✓ **to** - Comma separated list or an array of recipients email addresses that will appear on the To: field
- ✓ **cc** - Comma separated list or an array of recipients email addresses that will appear on the Cc: field

- ✓ **bcc** - Comma separated list or an array of recipients email addresses that will appear on the Bcc: field
- ✓ **subject** - The subject of the email
- ✓ **text** - The plaintext version of the message
- ✓ **html** - The HTML version of the message
- ✓ **attachments** - An array of attachment objects. Attachments can be used for embedding images as well.

Simple example to send mail using nodemailer.

mailer.js

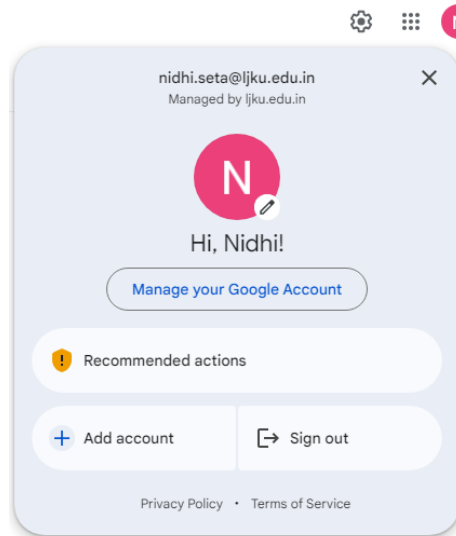
```
var nm = require("nodemailer");

var trans = nm.createTransport({
  host: "smtp.gmail.com",
  port: 465,
  auth: {
    user: "sender@gmail.com",
    pass: "App Password"
  }
});

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);
});
```

gmail >>
manage your google account >>



security >>
2 step verification(If set) >>

- Home
- Personal info
- Data & privacy
- Security**
- People & sharing
- Payments & subscriptions
- About

How you sign in to Google

Make sure you can always access your Google Account by keeping this info

2-Step Verification	On since Aug 20, 2022	>
Passkeys and security keys	Start using passkeys	>
Password	Last changed Jun 13	>
Skip password when possible	On	>

app passwords >>

← 2-Step Verification

Second steps

Make sure you can access your Google Account by keeping this information up to date and adding more sign-in options

Passkeys and security keys	Add a security key	>
Google prompt	2 devices	>
Authenticator	Add authenticator app	>
Phone number	085110 80337	>
Backup codes	Get backup codes	>

App passwords

App Passwords aren't recommended and are unnecessary in most cases. To help keep your account secure, use "Sign in with Google" to connect apps to your Google Account.

App passwords	2 App passwords	>
---------------	-----------------	---

Enter App name and create password >>

← App passwords

App passwords help you sign into your Google Account on older apps and services that don't support modern security standards.

App passwords are less secure than using up-to-date apps and services that use modern security standards. Before you create an app password, you should check to see if your app needs this in order to sign in.

[Learn more](#)

Your app passwords

temp	Created on 1:47 AM	🗑️
test	Created on 1:45 AM	🗑️

To create a new app specific password, type a name for it below...

App name

Nodemailer

Create

RESTful API

- ✓ [REST](#), which stands for **REpresentational State Transfer**, is a software development architecture that defines a set of rules for communication between a client and a server. Let's break this down a little more:
 - ✓ A **REST client** is a code or app used to communicate with REST servers.
 - ✓ A **server** contains resources that the client wants to access or change.
 - ✓ A **resource** is any information that the API can return.
 - ✓ A REST API, also known as a RESTful API, is an API that conforms to the REST architecture. These APIs use the [HTTP](#) protocol to access and manipulate data on the server.
 - ✓ An popular architectural style of how to structure and name APIs and the endpoints is called **REST(Representational State Tansfer)**. An API is always needed to create mobile applications, single page applications, use AJAX calls and provide data to clients.
 - ✓ 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.
 - ✓ Use the express.Router class to create modular, mountable route handlers. A Router instance is a complete middleware and routing system; for this reason, it is often referred to as a “mini-app”.
 - ✓ The following example creates a router as a module, loads a middleware function in it, defines some routes, **and mounts the router module on a path in the main app.**

Example

Write an expressJS code in which RESTapi is created for json object named data which contains name,id,branch,city and contact properties.

- ✓ On “/api” page it should display all the content.
- ✓ Upon passing id on the browser url it should display the content having that id. (i.e. on localhost:7899/api/101)
- ✓ Upon passing branch on the browser url it should display the content having that branch. (i.e. on localhost:7899/api/cse)

api.js

```
const expr = require("express");
const router = expr.Router();
const data=[
  { id:101,name:"ABC",branch:"CSE",contact:9876543210,city:"Ahmedabad" },
  { id:102,name:"BCD",branch:"CE",contact:9876543210,city:"Ahmedabad" },
  { id:103,name:"XYZ",branch:"CSE",contact:9876543210,city:"Rajkot" },
```

```

    { id:104,name:"PQR",branch:"IT",contact:9876543210,city:"Ahmedabad" },
    { id:105,name:"ABC",branch:"CSE",contact:9876543210,city:"Surat" },
    { id:106,name:"ABC",branch:"IT",contact:9876543210,city:"Rajkot" }
  ]

  router.get("/",(req,res)=>{
    res.set("content-type","text/html")
    for(i of data){
      res.write("<h3>ID: " + JSON.stringify(i.id) + ", Name: " + i.name + ", Branch: " + i.branch
+ ", Contact: " + i.contact + ", City: " + i.city + "</h3>");
    }
    res.send();
  })

  router.get("/:id",(req,res)=>{
    var current_data = data.filter((i1)=> i1.id == req.params.id)
    if(current_data.length > 0){
      res.send(current_data);
    }else{
      res.send("Not Found")
    }
  })

  router.get("/branch/:branch",(req,res)=>{

    var cd = data.filter((b)=>b.branch.toLowerCase() == req.params.branch.toLowerCase())
    if(current_data.length > 0){
      res.send(current_data);
    }else{
      res.send("Not Found")
    }
  })
  module.exports = router;

```

main.js

```

const expr = require("express");
const app = expr();
const api = require("./api")
app.use("/api",api);

```

```
app.listen(7899);
```

To run:

node main.js

Check output in browser:

<http://localhost:7899/api>

ID: 101, Name: ABC, Branch: CSE, Contact: 9876543210, City: Ahmedabad

ID: 102, Name: BCD, Branch: CE, Contact: 9876543210, City: Ahmedabad

ID: 103, Name: XYZ, Branch: CSE, Contact: 9876543210, City: Rajkot

ID: 104, Name: PQR, Branch: IT, Contact: 9876543210, City: Ahmedabad

ID: 105, Name: ABC, Branch: CSE, Contact: 9876543210, City: Surat

ID: 106, Name: ABC, Branch: IT, Contact: 9876543210, City: Rajkot

<http://localhost:7899/api/101>

[{"id":101,"name":"ABC","branch":"CSE","contact":9876543210,"city":"Ahmedabad"}]

<http://localhost:7899/api/branch/it>

[{"id":104,"name":"PQR","branch":"IT","contact":9876543210,"city":"Ahmedabad"}, {"id":106,"name":"ABC","branch":"IT","contact":9876543210,"city":"Rajkot"}]