

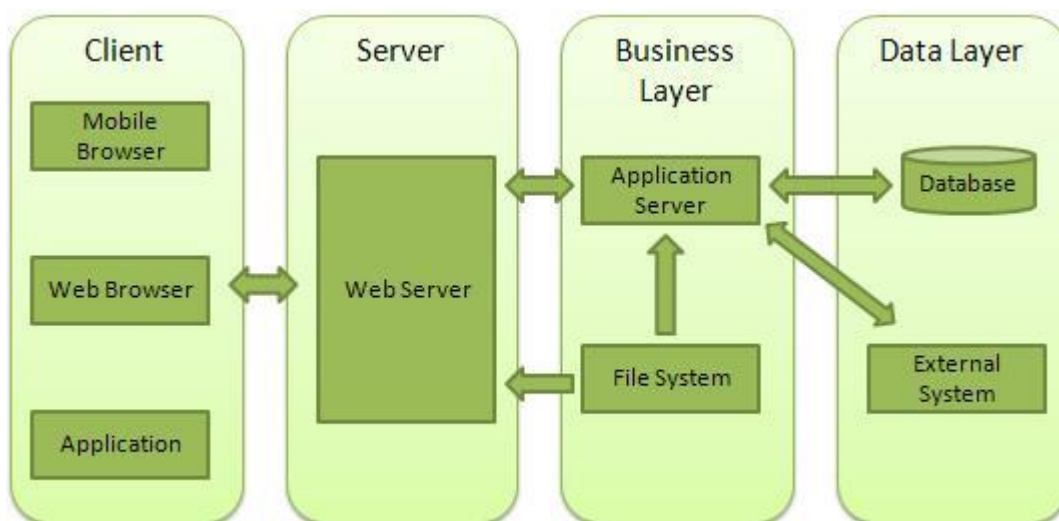
A Web Server is a software application which handles HTTP requests sent by the HTTP client, like web browsers, and returns web pages in response to the clients. Web servers usually deliver html documents along with images, style sheets, and scripts.

Most of the web servers support server-side scripts, using scripting languages or redirecting the task to an application server which retrieves data from a database and performs complex logic and then sends a result to the HTTP client through the Web server.

Apache web server is one of the most commonly used web servers. It is an open source project.

Web Application Architecture

A Web application is usually divided into four layers –



Client – This layer consists of web browsers, mobile browsers or applications which can make HTTP requests to the web server.

Server – This layer has the Web server which can intercept the requests made by the clients and pass them the response.

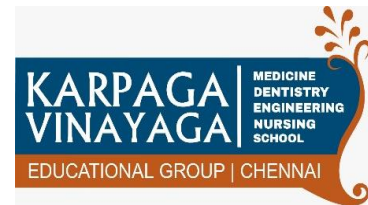
Business – This layer contains the application server which is utilized by the web server to do the required processing. This layer interacts with the data layer via the database or some external programs.



MC4201 –Full Stack Web Development

Dept. Of Computer Applications

UNIT-II



Data – This layer contains the databases or any other source of data.

Examples

Following table describes the most leading web servers available today:

S.N.	Web Server Descriptino
1	Apache HTTP Server This is the most popular web server in the world developed by the Apache Software Foundation. Apache web server is an open source software and can be installed on almost all operating systems including Linux, UNIX, Windows, FreeBSD, Mac OS X and more. About 60% of the web server machines run the Apache Web Server.
2.	Internet Information Services (IIS) The Internet Information Server (IIS) is a high performance Web Server from Microsoft. This web server runs on Windows NT/2000 and 2003 platforms (and may be on upcoming new Windows version also). IIS comes bundled with Windows NT/2000 and 2003; Because IIS is tightly integrated with the operating system so it is relatively easy to administer it.
3.	Lighttpd The lighttpd, pronounced lighty is also a free web server that is distributed with the FreeBSD operating system. This open source web server is fast, secure and consumes much less CPU power. Lighttpd can also run on Windows, Mac OS X, Linux and Solaris operating systems.
4.	Sun Java System Web Server This web server from Sun Microsystems is suited for medium and large web sites. Though the server is free it is not open source. It however, runs on Windows, Linux and UNIX platforms. The Sun Java System web server supports various languages, scripts and technologies required for Web 2.0 such as JSP, Java Servlets, PHP, Perl, Python, and Ruby on Rails, ASP and Coldfusion etc.
5.	Jigsaw Server Jigsaw (W3C's Server) comes from the World Wide Web Consortium. It is open source and free and can run on various platforms like Linux, UNIX, Windows, and Mac OS X Free BSD etc. Jigsaw has been written in Java and can run CGI scripts and PHP programs.

JavaScript in the Desktop with NodeJS

Electron JS

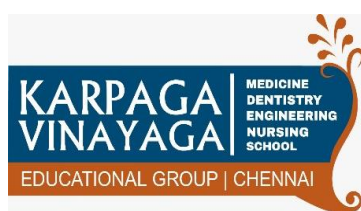
Electron is a popular and open-source JavaScript framework designed as a project by Cheng Zhao and is developed and maintained by [GitHub](#)



MC4201 -Full Stack Web Development

Dept. Of Computer Applications

UNIT-II



. Electron is primarily used to develop desktop GUI applications that include various web technologies. It can also be called a runtime framework that allows you to create desktop-suite applications along with [HTML](#), [CSS](#), and [JavaScript](#). It also combines the rendering feature of Chromium Engine with the [Node.js](#) runtime environment.

To get started with developing using the Electron, you need to have Node and npm (node package manager) installed. If you do not already have these, head over to [Node setup](#) to install node on your local system. Confirm that node and npm are installed by running the following commands in your terminal.

```
node --version
npm --version
```

The above command will generate the following output –

```
v6.9.1
3.10.8
```

Whenever we create a project using npm, we need to provide a **package.json** file, which has all the details about our project. npm makes it easy for us to set up this file. Let us set up our development project.

- Fire up your terminal/cmd, create a new folder named hello-world and open that folder using the cd command.
- Now to create the package.json file using npm, use the following command.

```
npm init
```

- It will ask you for the following information –

```
Press ^C at any time to quit.
name: (hello-world)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author: Ayush Gupta
license: (ISC)
About to write to /home/ayushgp/hello-world/package.json:

{
  "name": "hello-world",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "author": "Ayush Gupta",
  "license": "ISC"
}

Is this ok? (yes) yes
ayushgp@dell:~/hello-world$
```



MC4201 –Full Stack Web Development

Dept. Of Computer Applications

UNIT-II



Just keep pressing Enter, and enter your name at the “author name” field.

Create a new folder and open it using the `cd` command. Now run the following command to install Electron globally.

```
$ npm install -g electron-prebuilt
```

Once it executes, you can check if Electron is installed the right way by running the following command –

```
$ electron --version
```

You should get the output –

```
v1.4.13
```

Now that we have set up Electron, let us move on to creating our first app using it.

Electron takes a main file defined in your *package.json* file and executes it. This main file creates application windows, which contain rendered web pages and interaction with the native GUI (graphical user interface) of your Operating System.

As you start an application using Electron, a **main process** is created. This main process is responsible for interacting with the native GUI of the Operating System. It creates the GUI of your application.

Just starting the main process does not give the users of your application any application window. These are created by the main process in the main file by using the *BrowserWindow* module. Each browser window then runs its own **renderer process**. The renderer process takes an HTML file which references the usual CSS files, JavaScript files, images, etc. and renders it in the window.

The main process can access the native GUI through modules available directly in Electron. The desktop application can access all Node modules like the file system module for handling files, request to make HTTP calls, etc.

Difference between Main and Renderer processes

The main process creates web pages by creating the *BrowserWindow* instances. Each *BrowserWindow* instance runs the web page in its own renderer process. When a *BrowserWindow* instance is destroyed, the corresponding renderer process is also terminated.

The main process manages all web pages and their corresponding renderer processes. Each renderer process is isolated and only cares about the web page running in it.

We have created a **package.json** file for our project. Now we will create our first desktop app using Electron.

Create a new file called *main.js*. Enter the following code in it –



MC4201 –Full Stack Web Development

Dept. Of Computer Applications

UNIT-II



```
const {app, BrowserWindow} = require('electron')
const url = require('url')
const path = require('path')

let win

function createWindow() {
  win = new BrowserWindow({width: 800, height: 600})
  win.loadURL(url.format({
    pathname: path.join(__dirname, 'index.html'),
    protocol: 'file:',
    slashes: true
  }))
}

app.on('ready', createWindow)
```

Create another file, this time an HTML file called *index.html*. Enter the following code in it.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset = "UTF-8">
    <title>Hello World!</title>
  </head>

  <body>
    <h1>Hello World!</h1>
    We are using node <script>document.write(process.versions.node)</script>,
    Chrome <script>document.write(process.versions.chrome)</script>,
    and Electron <script>document.write(process.versions.electron)</script>.
  </body>
</html>
```

Run this app using the following command –

```
$ electron ./main.js
```

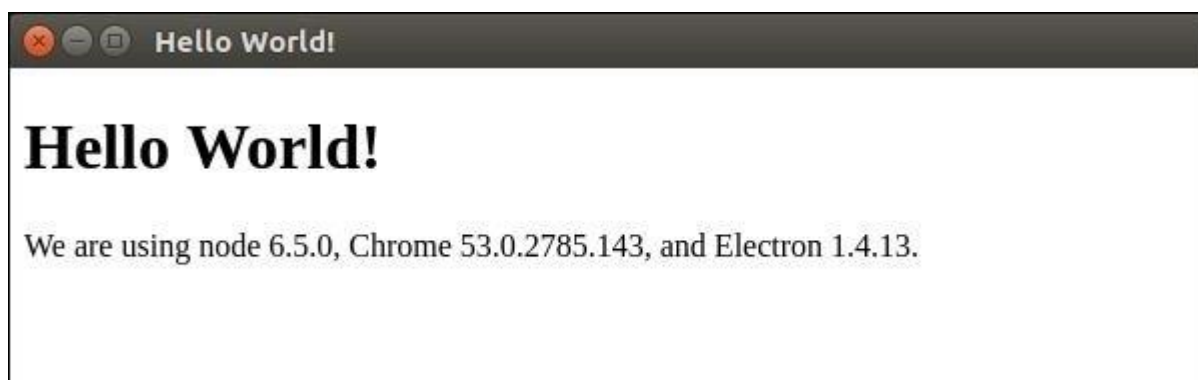
A new window will open up. It will look like the following –



MC4201 –Full Stack Web Development

Dept. Of Computer Applications

UNIT-II



How Does This App Work?

We created a main file and an HTML file. The main file uses two modules – *app* and *BrowserWindow*. The *app* module is used to control your application's event lifecycle while the *BrowserWindow* module is used to create and control browser windows.

We defined a *createWindow* function, where we are creating a new *BrowserWindow* and attaching a URL to this *BrowserWindow*. This is the HTML file that is rendered and shown to us when we run the app.

We have used a native Electron object process in our html file. This object is extended from the Node.js process object and includes all of **its** functionalities while adding many more.

NPM

Node Package Manager (NPM) provides two main functionalities –

Online repositories for node.js packages/modules which are searchable on search.nodejs.org

Command line utility to install Node.js packages, do version management and dependency management of Node.js packages.

NPM comes bundled with Node.js installable after v0.6.3 version. To verify the same, open console, type the following command, and see the result



MC4201 –Full Stack Web Development

Dept. Of Computer Applications

UNIT-II



\$ npm –version

Serving files with the Http Module

// HTTP Module for Creating Server and Serving Static Files Using Node.js

// Static Files: HTML, CSS, JS, Images

// Get Complete Source Code from Pabbly.com

```
var http = require('http');
var fs = require('fs');
var path = require('path');

http.createServer(function(req, res){

    if(req.url === "/"){
        fs.readFile("./public/index.html", "UTF-8", function(err, html){
            res.writeHead(200, {"Content-Type": "text/html"});
            res.end(html);
        });
    }else if(req.url.match("\.css$")){
        var cssPath = path.join(__dirname, 'public', req.url);
        var fileStream = fs.createReadStream(cssPath, "UTF-8");
        res.writeHead(200, {"Content-Type": "text/css"});
        fileStream.pipe(res);
    }else if(req.url.match("\.png$")){
        var imagePath = path.join(__dirname, 'public', req.url);
        var fileStream = fs.createReadStream(imagePath);
        res.writeHead(200, {"Content-Type": "image/png"});
        fileStream.pipe(res);
    }else{
        res.writeHead(404, {"Content-Type": "text/html"});
        res.end("No Page Found");
    }

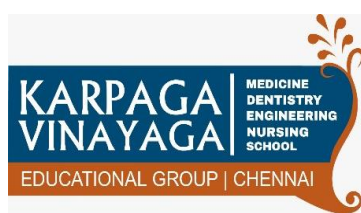
}).listen(3000);
```



MC4201 –Full Stack Web Development

Dept. Of Computer Applications

UNIT-II



Introduction to the Express Framework

ExpressJS is a web application framework that provides you with a simple API to build websites, web apps and back ends. With ExpressJS, you need not worry about low level protocols, processes, etc.

What is Express?

Express provides a minimal interface to build our applications. It provides us the tools that are required to build our app. It is flexible as there are numerous modules available on **npm**, which can be directly plugged into Express.

Express was developed by **TJ Holowaychuk** and is maintained by the [Node.js](https://nodejs.org/) foundation and numerous open source contributors.

Why Express?

Unlike its competitors like Rails and Django, which have an opinionated way of building applications, Express has no "best way" to do something. It is very flexible and pluggable.

Pug

Pug (earlier known as Jade) is a terse language for writing HTML templates. It –

- Produces HTML
- Supports dynamic code
- Supports reusability (DRY)

It is one of the most popular template language used with Express.

MongoDB and Mongoose

MongoDB is an open-source, document database designed for ease of development and scaling. This database is also used to store data.

Mongoose is a client API for **node.js** which makes it easy to access our database from our Express application.

Express JS environment

We will learn how to start developing and using the Express Framework. To start with, you should have the Node and the npm (node package manager) installed. If you don't already have these, go to the [Node setup](#) to install node on your local system. Confirm that node and npm are installed by running the following commands in your terminal.

```
node --version
npm --version
```




MC4201 –Full Stack Web Development

Dept. Of Computer Applications

UNIT-II



You should get an output similar to the following.

```
v5.0.0
3.5.2
```

Now that we have Node and **npm** set up, let us understand what **npm** is and how to use it.

Node Package Manager(npm)

npm is the package manager for node. The npm Registry is a public collection of packages of open-source code for Node.js, front-end web apps, mobile apps, robots, routers, and countless other needs of the JavaScript community. npm allows us to access all these packages and install them locally. You can browse through the list of packages available on npm at [npmJS](https://www.npmjs.com/).

How to use npm?

There are two ways to install a package using npm: globally and locally.

- **Globally** – This method is generally used to install development tools and CLI based packages. To install a package globally, use the following code.

```
npm install -g <package-name>
```

- **Locally** – This method is generally used to install frameworks and libraries. A locally installed package can be used only within the directory it is installed. To install a package locally, use the same command as above without the **-g** flag.

```
npm install <package-name>
```

Whenever we create a project using npm, we need to provide a **package.json** file, which has all the details about our project. npm makes it easy for us to set up this file. Let us set up our development project.

Step 1 – Start your terminal/cmd, create a new folder named hello-world and cd (create directory) into it –

```
ayushgp@dell:~$ mkdir hello-world
ayushgp@dell:~$ cd hello-world/
ayushgp@dell:~/hello-world$
```

Step 2 – Now to create the package.json file using npm, use the following code.

```
npm init
```

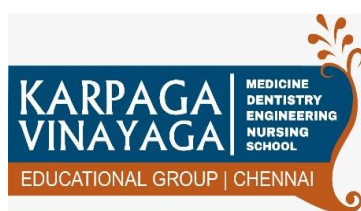
It will ask you for the following information.



MC4201 –Full Stack Web Development

Dept. Of Computer Applications

UNIT-II



```
Press ^C at any time to quit.
name: (hello-world)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author: Ayush Gupta
license: (ISC)
About to write to /home/ayushgp/hello-world/package.json:
{
  "name": "hello-world",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "author": "Ayush Gupta",
  "license": "ISC"
}
Is this ok? (yes) yes
ayushgp@dell:~/hello-world$
```

Just keep pressing enter, and enter your name at the “author name” field.

Step 3 – Now we have our package.json file set up, we will further install Express. To install Express and add it to our package.json file, use the following command –

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

To confirm that Express has installed correctly, run the following code.

```
ls node_modules #(dir node_modules for windows)
```

Tip – The **--save** flag can be replaced by the **-S** flag. This flag ensures that Express is added as a dependency to our **package.json** file. This has an advantage, the next time we need to install all the dependencies of our project we can just run the command *npm install* and it will find the dependencies in this file and install them for us.

This is all we need to start development using the Express framework. To make our development process a lot easier, we will install a tool from npm, nodemon. This tool restarts our server as soon as we make a change in any of our files, otherwise we need to restart the server manually after each file modification. To install nodemon, use the following command –

```
npm install -g nodemon
```

You can now start working on Express.

We have set up the development, now it is time to start developing our first app using Express. Create a new file called **index.js** and type the following in it.

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

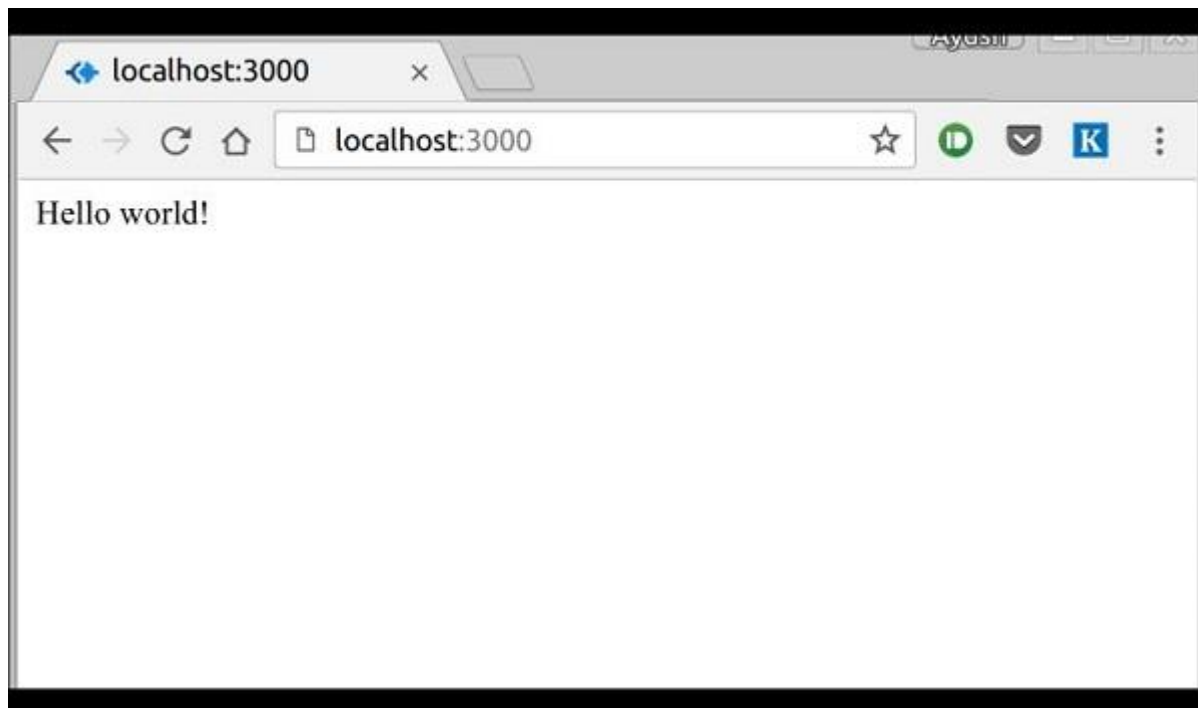
app.get('/', function(req, res){
  res.send("Hello world!");
});
```

```
app.listen(3000);
```

Save the file, go to your terminal and type the following.

```
nodemon index.js
```

This will start the server. To test this app, open your browser and go to **http://localhost:3000** and a message will be displayed as in the following screenshot.



How the App Works?

The first line imports Express in our file, we have access to it through the variable Express. We use it to create an application and assign it to var app.

`app.get(route, callback)`

This function tells what to do when a **get** request at the given route is called. The callback function has 2 parameters, **request(req)** and **response(res)**. The request **object(req)** represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers, etc. Similarly, the response object represents the HTTP response that the Express app sends when it receives an HTTP request.

`res.send()`

This function takes an object as input and it sends this to the requesting client. Here we are sending the string *"Hello World!"*.

`app.listen(port, [host], [backlog], [callback])`



MC4201 –Full Stack Web Development

Dept. Of Computer Applications

UNIT-II



This function binds and listens for connections on the specified host and port. Port is the only required parameter here.

S.No.	Argument & Description
1	port A port number on which the server should accept incoming requests.
2	host Name of the domain. You need to set it when you deploy your apps to the cloud.
3	backlog The maximum number of queued pending connections. The default is 511.
4	callback An asynchronous function that is called when the server starts listening for requests.

Pug is a templating engine for Express. 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!
```



MC4201 –Full Stack Web Development

Dept. Of Computer Applications

UNIT-II



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 seperated**

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.

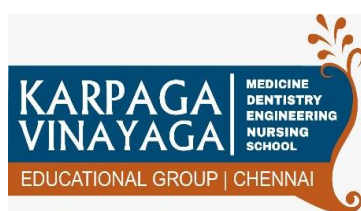
Comments



MC4201 –Full Stack Web Development

Dept. Of Computer Applications

UNIT-II



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);
```

And create a new view file in views directory, called **dynamic.pug**, with the following code –

```
html
head
title=name
body
h1=name
a(href = url) URL
```

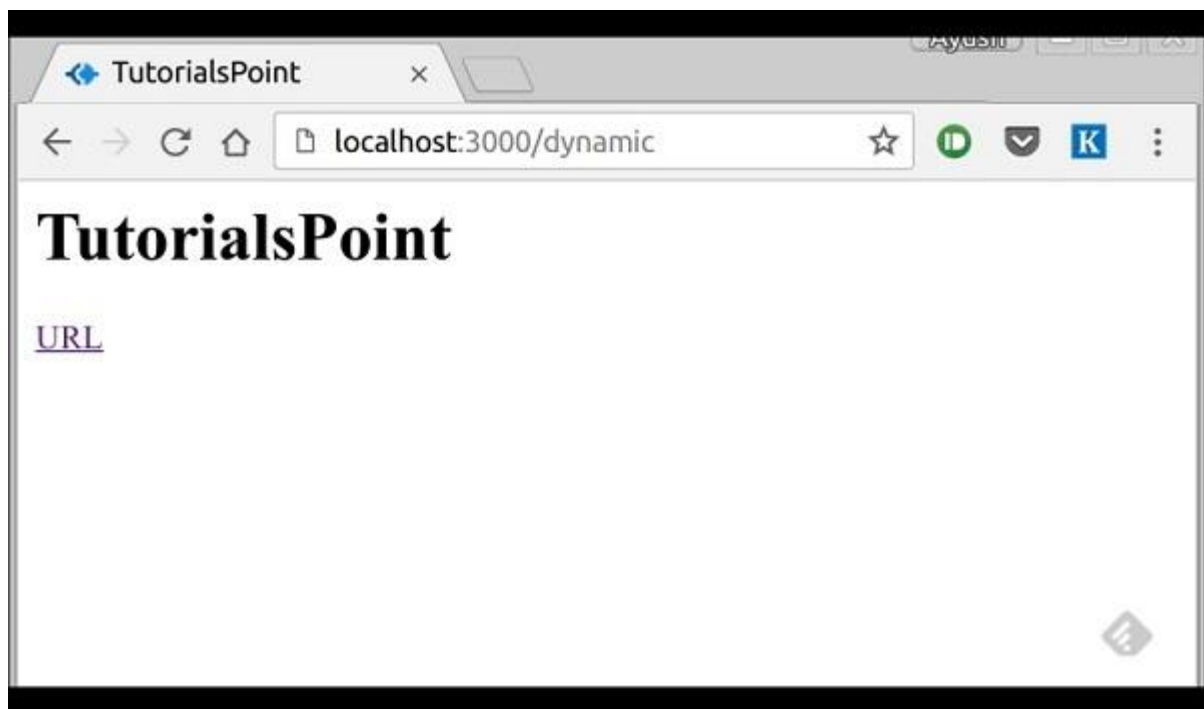
Open localhost:3000/dynamic_view in your browser; You should get the following output –



MC4201 –Full Stack Web Development

Dept. Of Computer Applications

UNIT-II



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
```

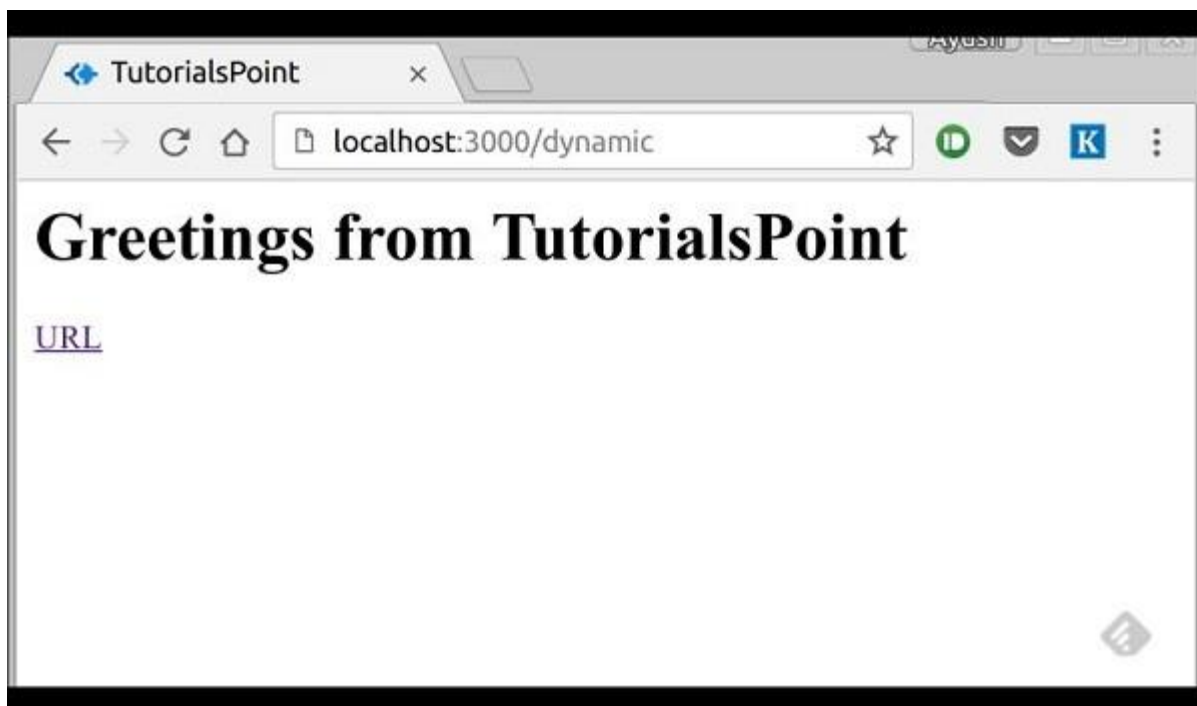
This method of using values is called **interpolation**. The above code will display the following output. –



MC4201 –Full Stack Web Development

Dept. Of Computer Applications

UNIT-II



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 –



MC4201 –Full Stack Web Development

Dept. Of Computer Applications

UNIT-II



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.

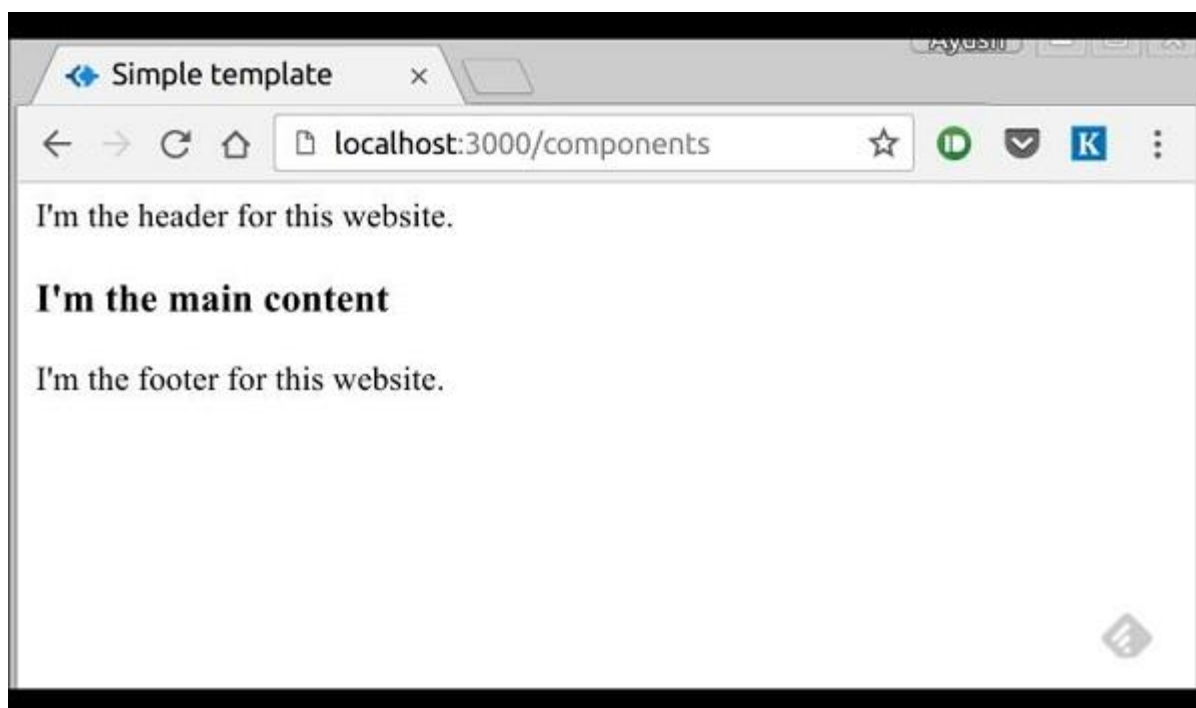
Create a route for this as follows –

```
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](#).

Static files are files that clients download as they are from the server. Create a new directory, **public**. Express, by default does not allow you to serve static files. You need to enable it using the following built-in middleware.

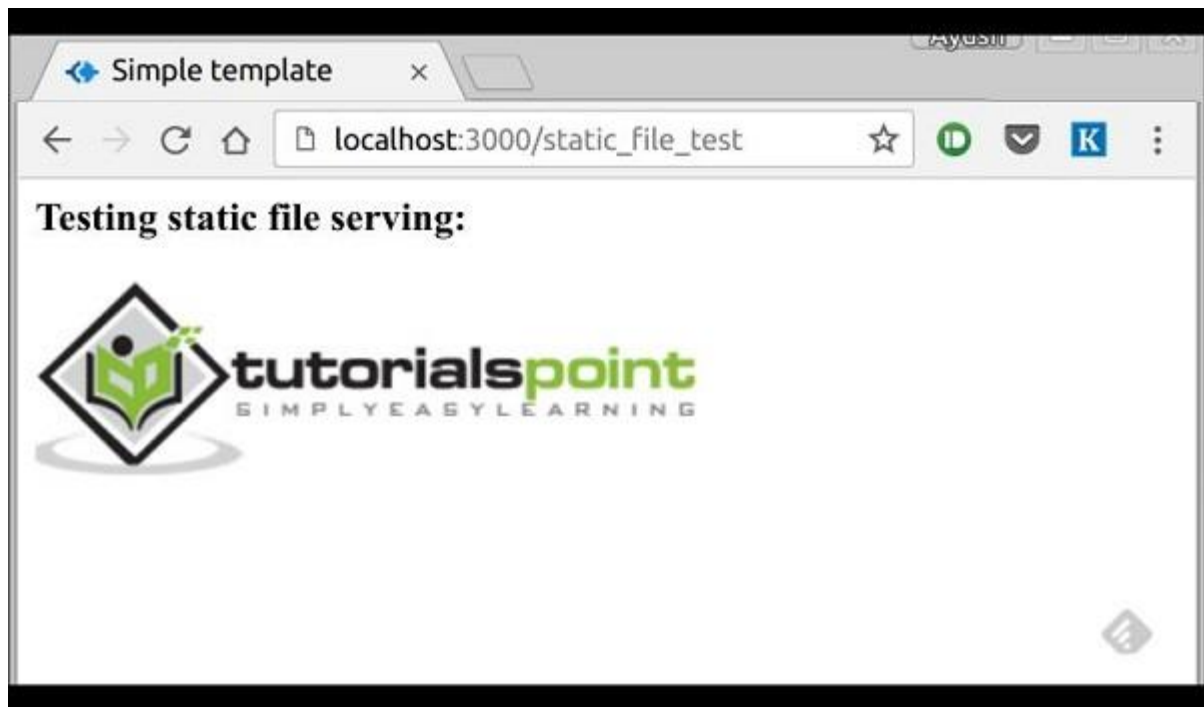
```
app.use(express.static('public'));
```

Note – Express looks up the files relative to the static directory, so the name of the static directory is not part of the URL.

Note that the root route is now set to your public dir, so all static files you load will be considering public as root. To test that this is working fine, add any image file in your new **public** dir and change its name to **"testimage.jpg"**. In your views, create a new view and include this file like –

```
html
head
body
h3 Testing static file serving:
img(src = "/testimage.jpg", alt = "Testing Image
```

You should get the following output –



Multiple Static Directories

We can also set multiple static assets directories using the following program –

```
var express = require('express');  
var app = express();  
  
app.use(express.static('public'));  
app.use(express.static('images'));  
  
app.listen(3000);
```

Virtual Path Prefix

We can also provide a path prefix for serving static files. For example, if you want to provide a path prefix like **'/static'**, you need to include the following code in your **index.js** file –

```
var express = require('express');  
var app = express();  
  
app.use('/static', express.static('public'));  
  
app.listen(3000);
```

Now whenever you need to include a file, for example, a script file called main.js residing in your public directory, use the following script tag –

```
<script src = "/static/main.js" />
```



This technique can come in handy when providing multiple directories as static files. These prefixes can help distinguish between multiple directories.

Using Async Await in Express with Node 14

Since the arrival of node v 7.6.0 `async / await` has been supported in node by default. If you're using Node 8.0+ there is no reason to not use `async / await` in your node applications. If you aren't familiar with `async / await` or aren't sure why you would want to use it over promises, here's a quick rundown:

Less Code

Go from this:

To this:

Better Errors

If you're familiar with promises you know that if a promise is rejected you'll need to handle that error inside a `.catch``, and if you're handling errors for both synchronous and asynchronous code you will likely have to duplicate your error handler.

In the above snippet we can see that there is duplicate code on lines 6 and 8. The catch statement on line 7 will handle any errors that the synchronous function `doSynchronousThings` may throw but it won't



handle any errors thrown by `getUsers` since it is asynchronous. This example may seem palatable since all its doing is printing the error to the console, but if there is any kind of complex error handling logic we want to avoid duplicating it. Async / await lets us do exactly that:

There are many more advantages async / await has over promises and if you're interested I encourage you to read about them [here](#), but for now lets move on to using async / await with express.

Async / Await in Express

Implementing basic async / await functionality in express is quite straightforward. The most important thing to remember to do is to wrap functions you are awaiting in try / catch statements so you do not get silent errors.

Easy, right? Well... yes... but do we really need to write a try / catch statement inside of every route? Surely we can do better.

Wrapping Async Await Routes

Since Async Await is essentially syntactic sugar for promises, and if an `await` statement errors it will return a rejected promise, we can write a helper function that wraps our express routes to handle rejected promises.

This function can be a little tricky to read but is actually quite straightforward, so lets break it down. `asyncMiddleware` is a function



that takes another function and wraps it in a promise. In our use case the function it will take is an express route handler, and since we are passing that handler into `Promise.resolve` it will resolve with whatever value our route handler returns. If, however, one of the `await` statements in our handler gives us a rejected promise, it will go into the `.catch` on line 4 and be passed to `next` which will eventually give the error to our express error middleware to handle. Now all that remains to do is to wrap our routes in our `asyncMiddleware` and we will no longer have to worry about using `try / catch` statements in our routes.

So, our code looks much cleaner, we get the advantages of `async / await`, and we're confident that all of our errors are being caught and handled! An important thing to remember is when using this approach we need to have **all** of our `async` code return promises, as this approach won't work with `async` callbacks that aren't promisified.

Fetching JSON from Express JS

Express.js `express.json()` Function

- Difficulty Level : [Basic](#)
- Last Updated : 07 Jul, 2020

The **`express.json()`** function is a built-in middleware function in Express. It parses incoming requests with JSON payloads and is based on **`body-parser`**.

Syntax:

`express.json([options])`

Parameters: The options parameter have various property like `inflate`, `limit`, `type`, etc.

Return Value: It returns an Object.

Installation of express module:



MC4201 –Full Stack Web Development

Dept. Of Computer Applications

UNIT-II



1. You can visit the link to [Install express module](#). You can install this package by using this command.
`npm install express`
2. After installing the express module, you can check your express version in command prompt using the command.
`npm version express`
3. After that, you can just create a folder and add a file for example, index.js. To run this file you need to run the following command.
`node index.js`

Example 1: Filename: index.js

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

var app = express();

var PORT = 3000;

app.use(express.json());

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

    console.log(req.body.name)

    res.end();

})

app.listen(PORT, function(err) {

    if (err) console.log(err);

    console.log("Server listening on PORT", PORT);

});
```




MC4201 –Full Stack Web Development

Dept. Of Computer Applications


UNIT-II



});

Steps to run the program:

1. The project structure will look like this:

Name	Date modified
 node_modules	06-06-2020 04:55 PM
 index.js	06-06-2020 04:54 PM
 package-lock.json	06-06-2020 04:55 PM

2. Make sure you have installed **express** module using the following command:
npm install express
3. Run index.js file using below command:
node index.js

Output:

Server listening on PORT 3000

4. Now make a POST request to *http://localhost:3000/* with header set to **‘content-type: application/json’** and body {**“name”:”GeeksforGeeks”**}, then you will see the following output on your console:
5. Server listening on PORT 3000
6. GeeksforGeeks

Example 2: Filename: index.js

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

var app = express();

var PORT = 3000;

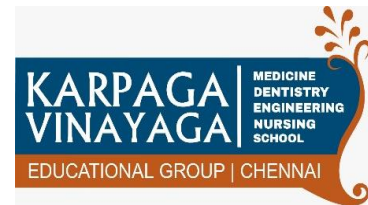
// Without this middleware
```



MC4201 –Full Stack Web Development

Dept. Of Computer Applications

UNIT-II



```
// app.use(express.json());

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

    console.log(req.body.name)

    res.end();

})

app.listen(PORT, function(err) {

    if (err) console.log(err);

    console.log("Server listening on PORT", PORT);

});
```

Run index.js file using below command:

```
node index.js
```

Now make a POST request to *http://localhost:3000/* with header set to **‘content-type: application/json’** and body {**“name”:”GeeksforGeeks”**}, then you will see the following output on your console:
Server listening on PORT 3000

```
TypeError: Cannot read property 'name' of undefined
```

Replicating fetch() with ['node-fetch'](#) package

The node-fetch package does pretty much what you expect: provide you with the fetch() syntax in Node.js. To install, run `npm install node-fetch`, and set up your code like this:

```
const fetch = require('node-fetch');

let url = "https://www.reddit.com/r/popular.json";
```



```
let settings = { method: "Get" };
```

```
fetch(url, settings)
  .then(res => res.json())
  .then((json) => {
    // do something with JSON
  });
```

Here, we've started by importing the package via `require()`, and created a `settings` variable to define our http method as a Get request. From there, we use `fetch(url, settings)` just like we would on the front-end. As usual, we can parse the response `res` as JSON, and then do whatever we need to with it.

Note: from some VERY RUDIMENTARY benchmark testing, it *appears* that node-fetch is the fastest of the three options covered in this article. Here are the times clocked by each (however, this DOES include running the rest of the code from the challenge, not just the fetch/https/request itself):

```
fetch: 0.689 seconds
https: 2.827 seconds
request: 3.65 seconds
```

I'd love for someone else to do a little more testing and verify/disprove this! Feel free to comment below if you're that person. ;)

Using the [http](#)/[https](#) modules provided by Node.js

Node.js comes with a pair of http/https modules, and in this case, the https module provides a built-in method for Get requests. Here's the code we'll be looking at:

```
const https = require('https');
```

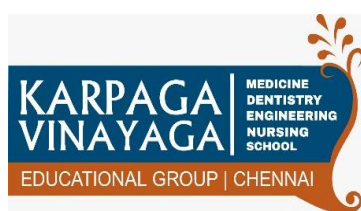
```
let url = "https://www.reddit.com/r/popular.json";
```



MC4201 -Full Stack Web Development

Dept. Of Computer Applications

UNIT-II



```
https.get(url,(res) => {  
  let body = "";  
  
  res.on("data", (chunk) => {  
    body += chunk;  
  });  
  
  res.on("end", () => {  
    try {  
      let json = JSON.parse(body);  
      // do something with JSON  
    } catch (error) {  
      console.error(error.message);  
    };  
  });  
  
}).on("error", (error) => {  
  console.error(error.message);  
});
```

There's a bit more going on here! First, we import the https module with `require()`. We can then call `https.get(url, (res) => { })` to initiate a Get request. Then, inside the body of the callback, we start by creating an empty string `body` that we'll add our the text of our response (again called `res`) to.

From there, we have a few examples of the `.on` syntax, which will listen for a few different events--namely, "data", "end", and "error".

When the response encounters "data", we add each chunk as text to our `body` variable. Once we hit the "end" of the response, we use the `try / catch` syntax to try to parse our `body`'s text as JSON, and return an error if it can't. Lastly, we chain another `.on` call to catch "error" for our initial `https.get()` request.

I find this syntax to be pretty clunky and verbose, although I do like the explicit error handling that is **required** by `https.get()`. However, this module is slower than the `node-fetch` package--see the benchmark results above.



Simplifying syntax with 'request' package

The third strategy I used was the request package, which aims to simplify the (often verbose) syntax of Node.js's http requests. Since this is an external package, start by installing it with `npm install request`.

Here's the code we'll be looking at:

```
const request = require('request');

let url = "https://www.reddit.com/r/popular.json";

let options = {json: true};

request(url, options, (error, res, body) => {
  if (error) {
    return console.log(error)
  };

  if (!error && res.statusCode === 200) {
    // do something with JSON, using the 'body' variable
  };
});
```

Wow, that's really readable! Let's break it down. As with the other examples, we import the package with `require()`, and set our url variable. The request package also has a nifty `options` feature, where you can specify a lot of things--but here, in setting `{ json: true }`, we tell the request to automatically parse the response's body as JSON if there's no error (and we get a 200 status code back). So, to access the JSON we want, just use the `body` variable!

This readability comes at the price of speed, however. Per the benchmark results above, this is the slowest option, most likely because so much is happening under-the-hood. However, the readability is top-notch, and configuring other http requests are just as simple as this Get request example!



MC4201 –Full Stack Web Development

Dept. Of Computer Applications

UNIT-II

