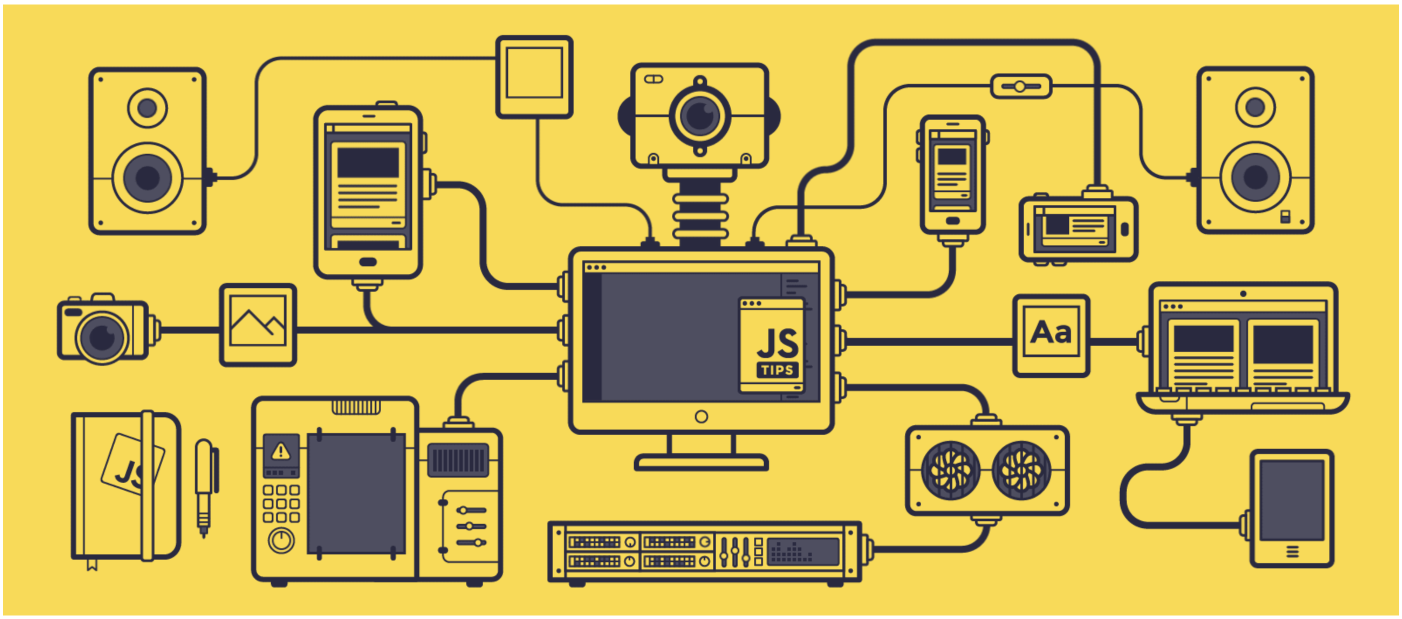
UNIT-5

Back-End Development using Node.Js

5.1 Introduction to web Frameworks

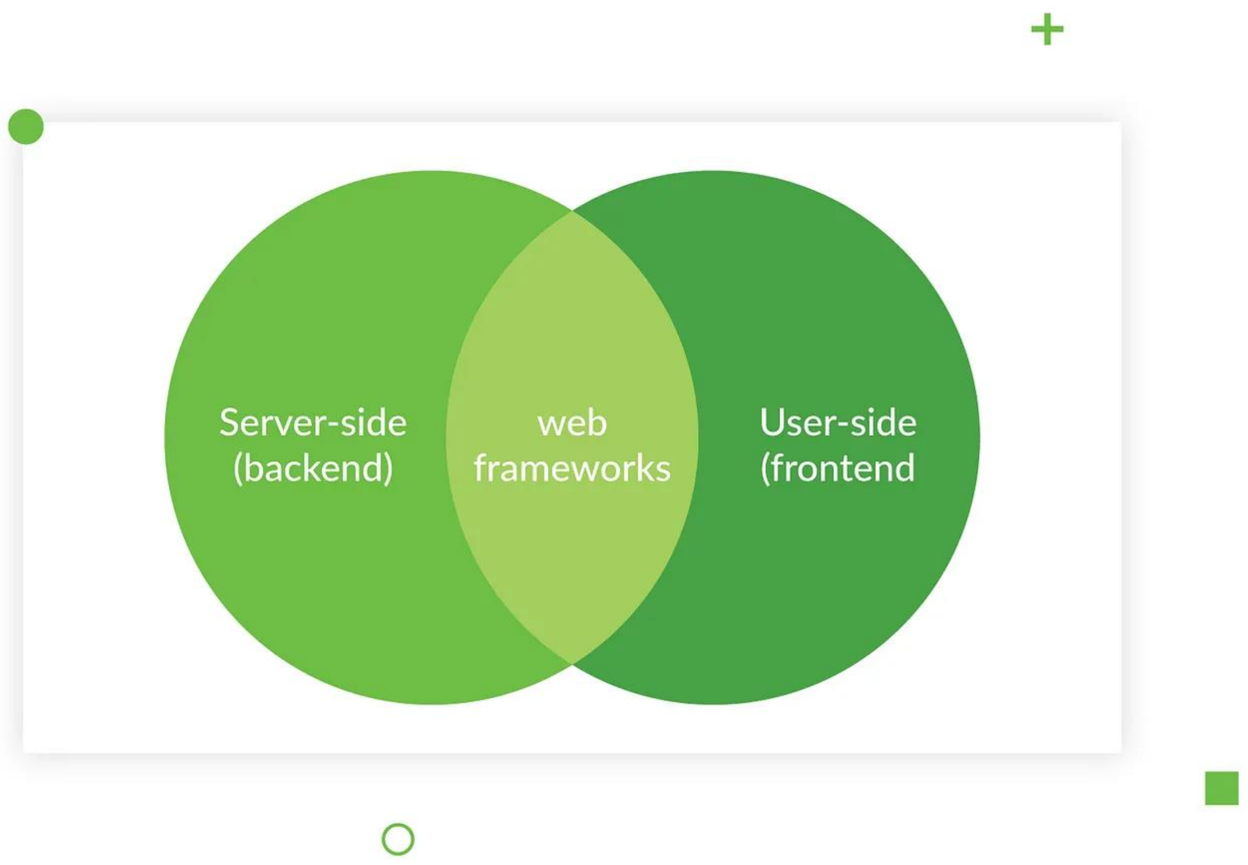
A web framework (WF) or web application framework (WAF) is a software framework that is designed to support the development of web applications including web services, web resources, and web APIs. Web frameworks provide a standard way to build and deploy web applications on the World Wide Web. Web frameworks aim to automate the overhead associated with common activities performed in web development. For example, many web frameworks provide libraries for database access, templating frameworks, and session management, and they often promote code reuse. Although they often target development of dynamic web sites, they are also applicable to static websites.



Types of Web frameworks:

As web standards began to advance, the app logic shifted towards the client- ensuring smarter communication between the user and the web application. With logic on client-side, they (client) can react swiftly to user input. This makes web apps more responsive, easily navigate-able on any device. Thus, we have two functions of frameworks — a) the one to work on the server side, that helps to set up app logic on the server (backend) or b) to work on the client-side (front end). Perform quality assurance (QA) testing: Create and oversee testing schedules to optimize user interface and experience, ensuring optimal display on various browsers and devices.

The front-end frameworks mostly manage the external part of the website, i.e. what the user sees when they open the application. The back-end manages the internal part. Let’s take a deeper look,



1. Server-side Frameworks

The rules and architecture of the server-side frameworks permit you to create simple pages, landings and forms of different kinds. To create a web application with a well-developed interface you need a wider range of functionality. Server-side frameworks handle HTTP requests, database control and management, URL mapping, etc. These frameworks can improve security and form the output data- simplifying the development process. Some of the top server-side frameworks are –

- NET (C#)
- Django (Python)
- Ruby on Rails (Ruby)
- Express (JavaScript/Node.JS)
- Symfony (PHP)

2. Client-side Frameworks

Client-side frameworks don't take care of the business logic like the server-side ones. They function inside the browser. Therefore, you can enhance and implement new user interfaces. A number of animated features can be created with frontend and single page applications. Every client-side framework varies in functionality and use. Here are some client-side frameworks for comparison's sake; all of whom use JavaScript as their programming language-

- Angular
- Ember.JS
- Vue.JS
- React.JS

5.2 Express Framework

5.2.1 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.js is built on top of Node.js, which means that it can take advantage of all the features and functions of Node.js, including its speed and scalability.

What is Express.js Used for?

Here are some of the common use cases for Express.js:

Single-page applications (SPAs):

Express.js can be used to build SPAs, which are web applications that provide a seamless user experience by loading all the necessary resources on a single page.

Mobile applications:

Express.js can be used to build mobile applications that use web technologies like HTML, CSS, and JavaScript.

RESTful APIs:

Express.js is commonly used to build RESTful APIs that can be used by other applications to access and manipulate data.

Server-side rendering:

Express.js can be used for server-side rendering, which is the process of rendering web pages on the server before sending them to the client.

Real-time applications:

Express.js can be used to build real-time applications, which require constant communication between the client and server.

Microservices:

Express.js is also used for building microservices, which are small, independent services that can be combined to build a larger application.

5.2.2 Key Features of Express JS:

Routing:

Express.js provides a simple and flexible routing system that allows developers to map HTTP requests to specific functions.

Middleware:

Express.js provides middleware functions that can be used to perform various operations on incoming requests and outgoing responses. Middleware functions can be used to add functionality to your application, such as authentication, error handling, and more.

Templates:

Express.js provides a variety of template engines that can be used to render HTML pages. It supports popular template engines like EJS, Handlebars, and Pug.

Error Handling:

Express.js provides robust error handling features that help developers handle errors and exceptions in their applications.

Security:

Express.js provides built-in security features like Helmet, which helps protect against common security vulnerabilities like cross-site scripting (XSS) and cross-site request forgery (CSRF).

Easy to Use:

Express.js is easy to use and requires minimal configuration, making it ideal for developers who want to build web applications quickly.

5.2.3 Installing Express

We can install it with npm. Make sure that you have Node.js and npm installed.

Step - 1: Creating a directory for our project and making that our working directory.

```
$ mkdir test
$ cd test
```

Step - 2: Using the npm init command to create a package.json file for our project.

```
$ npm init
```

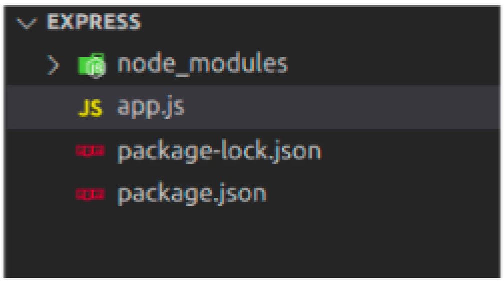
This command describes all the dependencies of our project. The file will be updated when adding further dependencies to the project.

Step - 3: Now in your test(name of your folder) folder type the following command:

```
$ npm install express –save
```

5.3 Create Application using Express

Project Structure: It will look like the following.



```
EXPRESS
  > node_modules
    app.js
    package-lock.json
    package.json
```

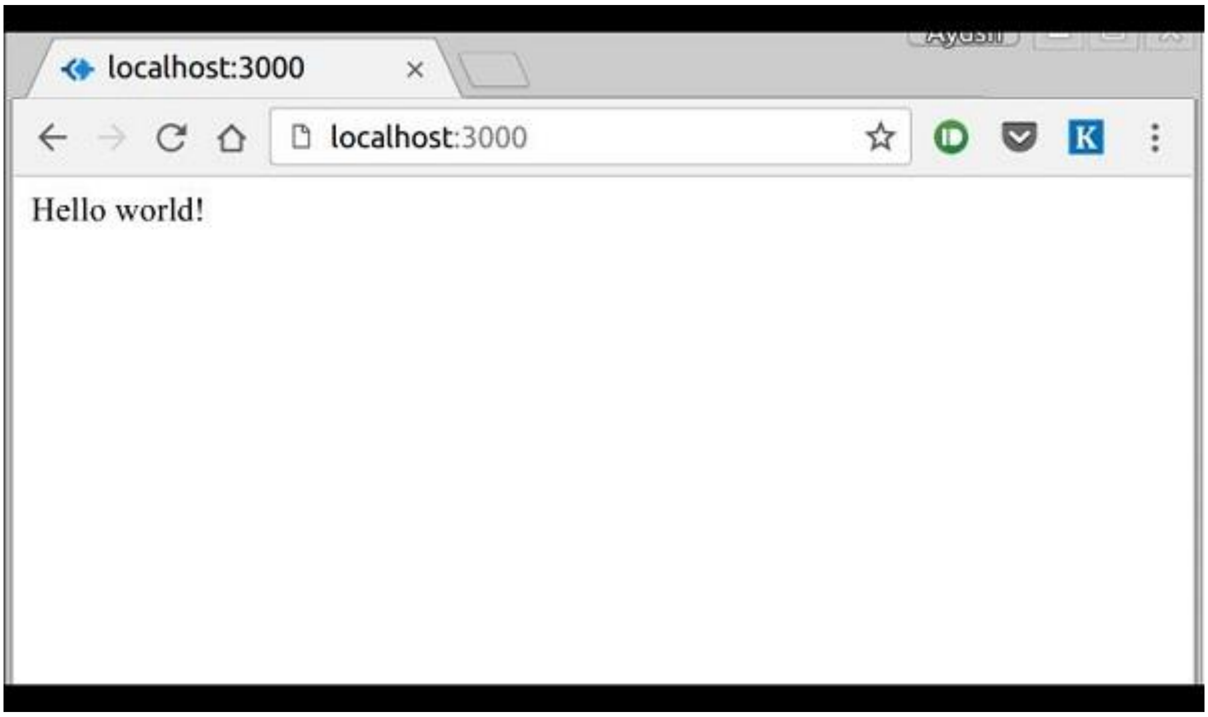
Example: Write the following code in app.js.

```
var express = require("express");
var app = express();
app.get("/", function (req, res) {
 res.send("Welcome to My Project");
});
app.listen(3000);
```

Step to run the application: Start the app by typing following command.

node app.js

Output:



Hello world!

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)

5.4 Routing, Middleware, Templating

5.4.1 Routing:

Web frameworks provide resources such as HTML pages, scripts, images, etc. at different routes.

The following function is used to define routes in an Express application −

app.method(path, handler)

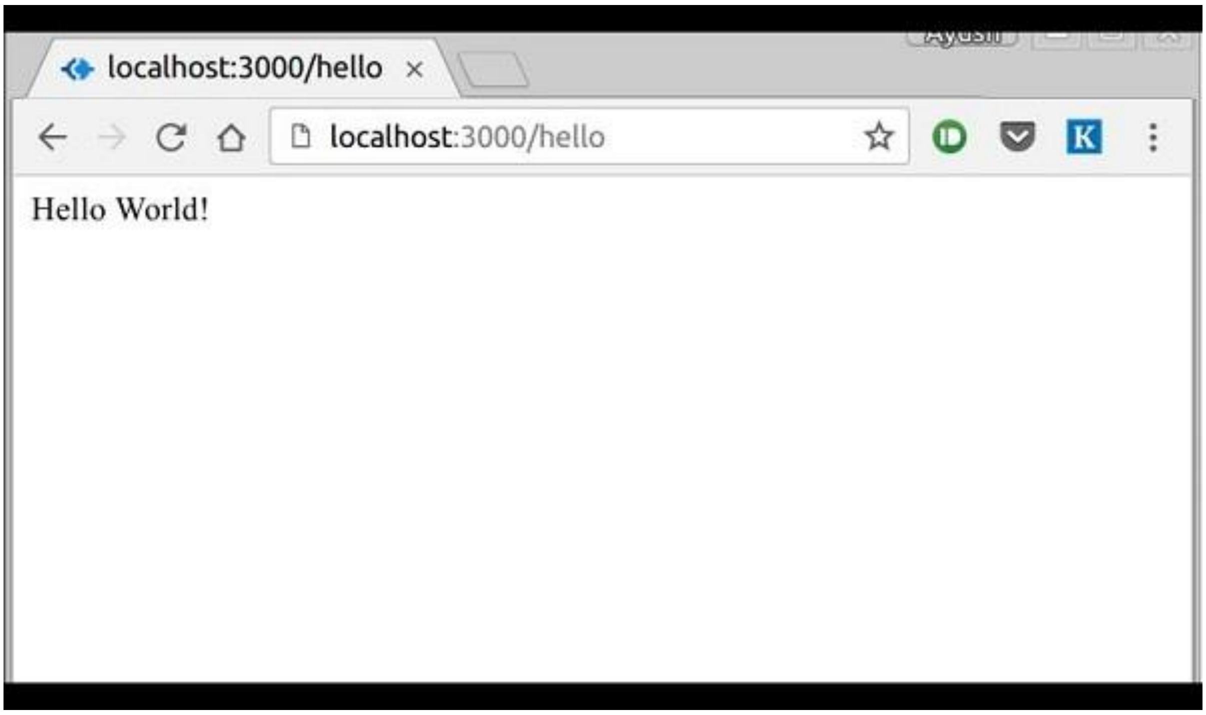
This METHOD can be applied to any one of the HTTP verbs – get, set, put, delete. An alternate method also exists, which executes independent of the request type.

Path is the route at which the request will run.

Handler is a callback function that executes when a matching request type is found on the relevant route. For example,

```
var express = require('express');
var app = express();
app.get('/hello', function(req, res){
 res.send("Hello World!");
});
app.listen(3000);
```

If we run our application and go to localhost:3000/hello, the server receives a get request at route "/hello", our Express app executes the callback function attached to this route and sends "Hello World!" as the response.



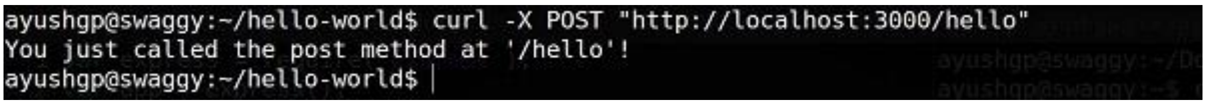
```
Hello World!
```

We can also have multiple different methods at the same route. For example,

```
var express = require('express');
var app = express();
app.get('/hello', function(req, res){
 res.send("Hello World!");
});
app.post('/hello', function(req, res){
 res.send("You just called the post method at '/hello'!\n");
});
app.listen(3000);
```

To test this request, open up your terminal and use cURL to execute the following request −

```
curl -X POST http://localhost:3000/hello
```



```
ayushgp@swaggy:~/hello-world$ curl -X POST "http://localhost:3000/hello"
You just called the post method at '/hello'!
ayushgp@swaggy:~/hello-world$ |
```

A special method, all, is provided by Express to handle all types of http methods at a particular route using the same function. To use this method, try the following.

```
app.all('/test', function(req, res){
   res.send("HTTP method doesn't have any effect on this route!");
});
```

This method is generally used for defining middleware.

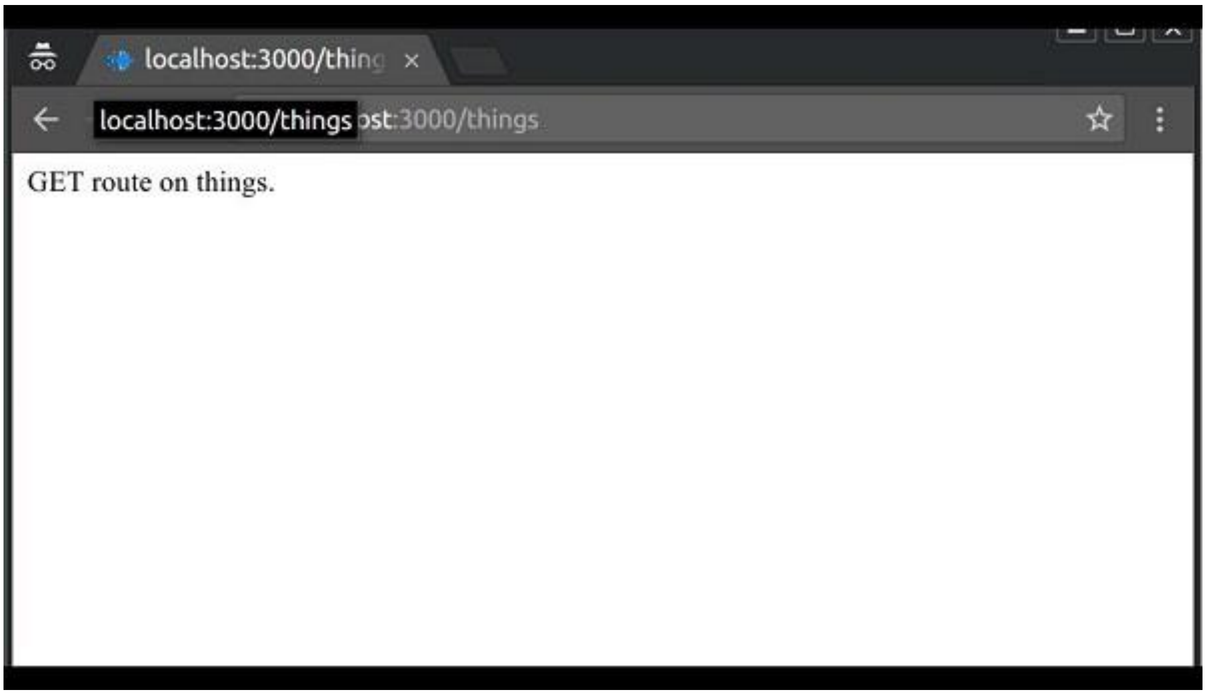
Example: Defining routes like above is very tedious to maintain. To separate the routes from our main index.js file, we will use Express.Router. Create a new file called things.js and type the following in it.

```
var express = require('express');
var router = express.Router();
router.get('/', function(req, res){
   res.send('GET route on things.');
});
router.post('/', function(req, res){
   res.send('POST route on things.');
});
//export this router to use in our index.js
module.exports = router;
```

Now to use this router in our index.js, type in the following before the app.listen function call.

```
var express = require('Express');
var app = express();
var things = require('./things.js');
//both index.js and things.js should be in same directory
app.use('/things', things);
app.listen(3000);
```

The app.use function call on route '/things' attaches the things router with this route. Now whatever requests our app gets at the '/things', will be handled by our things.js router. The '/' route in things.js is actually a subroute of '/things'. Visit localhost:3000/things/ and you will see the following output.



GET route on things.

5.4.2 Middleware:

You can create your own modules, and easily include them in your applications.

Middleware functions are functions that have access to the request object (req), the response object (res), and the next middleware function in the application's request-response cycle. The next middleware function is commonly denoted by a variable named next.

Middleware functions can perform the following tasks:

- Execute any code.
- Make changes to the request and the response objects.
- End the request-response cycle.
- Call the next middleware function in the stack.

The following example is for simple custom middleware

Example:

```
const express = require('express')
const app = express()
const port = 3000

app.get('/', (req, res, next) => {
  console.log("hello");
  setTimeout(next, 5);
},(req, res) => {
  res.send('Hello World!')
})
app.listen(port, () => {
  console.log(`Example app listening on port ${port}`)
})
```

5.4.3 Templating:

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

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

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

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

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

```
doctype html
html
  head
    title = "Hello Pug"
  body
    p.greetings#people Hello World!
```

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

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

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

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

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

Important Features of Pug:

Let us now explore a few important features of Pug.

Simple Tags

Tags are nested according to their indentation. Like in the above example, <title> was indented within the <head> tag, so it was inside it. But the <body> tag was on the same indentation, so it was a sibling of the <head> tag.

We don’t need to close tags, as soon as Pug encounters the next tag on same or outer indentation level, it closes the tag for us.

To put text inside of a tag, we have 3 methods −

- Space 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

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>

Pug.js rendering from string

We start with a very simple example that renders from a string.

Example : simple.js

```
import { render } from 'pug';
const template = 'p #{name} is a #{occupation}';
const data = { 'name': 'John Doe', 'occupation': 'gardener' };
```

```
const output = render(template, data);
console.log(output);
```

The example shows output from a string template.

```
import { render } from 'pug';
```
We load the render function from pug module.
```
const template = 'p #{name} is a #{occupation}';
```

This is our simple string template. The first value is the tag to be rendered. In addition, we add two variables: nameand occupation. To output the variables we use the #{} syntax.

```
const data = { 'name': 'John doe', 'occupation': 'gardener' };
```

This is the data that we pass to the template engine.

```
const output = render(template, data);
```

The render function takes a template string and the context data. It compiles both into the final string output.

OUTPUT:
```
$ node app.js
<p>John Doe is a gardener</p>
```

EXAMPLE: Pug.js compileFile

The compileFile function compiles a Pug template from a file to a function which can be rendered multiple times with different locals.

template.pug

```
p Hello #{name}!
```

This is the template file; it has a .pug extension.

app.js

```
import { compileFile } from 'pug';
const cfn = compileFile('template.pug');
const res = cfn({'name': 'John Doe'});
console.log(res);
const res2 = cfn({'name': 'Roger Roe'});
console.log(res2);
```

We compile the template to a function and call the function with two different local data.

OUTPUT:
```
$ node app.js
<p>Hello John Doe!</p>
<p>Hello Roger Roe!</p>
```

EXMAPLE: Pug.js renderFile

The renderFile function compiles a Pug template from a file and render it with locals to HTML string.

template.pug

```
doctype html
html
  body
    ul
      li Name: #{name}
      li Occupation: #{occupation}
```

In the tempalte, we have a small HTML document with an unordered list. We have two variables.

app.js

```
import { renderFile } from 'pug';
const options = { 'pretty': true }
const locals = { 'name': 'John doe', 'occupation': 'gardener', };
const res = renderFile('template.pug', Object.assign(locals, options));
console.log(res);
```

We merge the locals and the options with Object.assign;

```
const options = { 'pretty': true }
```

In the options map, we set the pretty printing. (Note that this option is deprecated.)

OUTPUT: $ node app.js

```
<!DOCTYPE html>
<html>
<body>
<ul>
<li>Name: John doe</li>
<li>Occupation: gardener</li>
</ul>
</body>
</html>
```

5.5 HTTP Method and RESTful APIs

5.5.1 HTTP Method:

The HTTP method is supplied in the request and specifies the operation that the client has requested. The following table lists the most used HTTP methods −

GET

The GET method requests a representation of the specified resource. Requests using GET should only retrieve data and should have no other effect.

POST

The POST method requests that the server accept the data enclosed in the request as a new object/entity of the resource identified by the URI.

PUT

The PUT method requests that the server accept the data enclosed in the request as a modification to existing object identified by the URI. If it does not exist then the PUT method should create one.

DELETE

The DELETE method requests that the server delete the specified resource.

5.5.2 RESTful APIs:

RESTful URIs and methods provide us with almost all information we need to process a request. The table given below summarizes how the various verbs should be used and how URIs should be named. We will be creating a movies API towards the end; let us now discuss how it will be structured.

Method	**URI**	**Details**	**Function**
GET	/movies	Safe, cachable	Gets the list of all movies and their details
GET	/movies/1234	Safe, cachable	Gets the details of Movie id 1234
POST	/movies	N/A	Creates a new movie with the details provided. Response contains the URI for this newly created resource.

Let us now create this API in Express. We will be using JSON as our transport data format as it is easy to work with in JavaScript and has other benefits. Replace your index.js file with the movies.js file as in the following program.

index.js:

```
var express = require('express');
var bodyParser = require('body-parser');
var multer = require('multer');
var upload = multer();
var app = express();
app.use(cookieParser());
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));
app.use(upload.array());
//Require the Router we defined in movies.js
var movies = require('./movies.js');
//Use the Router on the sub route /movies
app.use('/movies', movies);
app.listen(3000);
```

Now that we have our application set up, let us concentrate on creating the API. Start by setting up the movies.js file. We are not using a database to store the movies but are storing them in memory; so every time the server restarts, the movies added by us will vanish. This can easily be mimicked using a database or a file (using node fs module).

Once you import Express then, create a Router and export it using module.exports –

```
var express = require('express');
var router = express.Router();
var movies = [
   {id: 101, name: "Fight Club", year: 1999, rating: 8.1},
   {id: 102, name: "Inception", year: 2010, rating: 8.7},
   {id: 103, name: "The Dark Knight", year: 2008, rating: 9},
   {id: 104, name: "12 Angry Men", year: 1957, rating: 8.9}
];
//Routes will go here
module.exports = router;
```

GET routes:

Let us define the GET route for getting all the movies −

```
router.get('/', function(req, res){
   res.json(movies);
});
```

To test out if this is working fine, run your app, then open your terminal and enter −

```
curl -i -H "Accept: application/json" -H "Content-Type: application/json" -X GET
localhost:3000/movies
```

The following response will be displayed −

```
[{"id":101,"name":"Fight Club","year":1999,"rating":8.1},
{"id":102,"name":"Inception","year":2010,"rating":8.7},
{"id":103,"name":"The Dark Knight","year":2008,"rating":9},
{"id":104,"name":"12 Angry Men","year":1957,"rating":8.9}]
```

We have a route to get all the movies. Let us now create a route to get a specific movie by its id.

```
router.get('/:id([0-9]{3,})', function(req, res){
   var currMovie = movies.filter(function(movie){
      if(movie.id == req.params.id){
         return true;
      }
   });
   if(currMovie.length == 1){
      res.json(currMovie[0])
   } else {
      res.status(404);//Set status to 404 as movie was not found
      res.json({message: "Not Found"});
   }
});
```

This will get us the movies according to the id that we provided. To check the output, use the following command in your terminal −

```
curl -i -H "Accept: application/json" -H "Content-Type: application/json" -X GET
localhost:3000/movies/101
```

You'll get the following response −

```
{"id":101,"name":"Fight Club","year":1999,"rating":8.1}
```

If you visit an invalid route, it will produce a cannot GET error while if you visit a valid route with an id that doesn't exist, it will produce a 404 error.

We are done with the GET routes, let us now move on to the POST route.

POST routes:

Use the following route to handle the POST data −

```
router.post('/', function(req, res){
  //Check if all fields are provided and are valid:
  if(!req.body.name ||
    !req.body.year.toString().match(/^[0-9]{4}$/g) ||
    !req.body.rating.toString().match(/^[0-9]\.[0-9]$/g)){
    res.status(400);
    res.json({message: "Bad Request"});
  } else {
    var newId = movies[movies.length-1].id+1;
    movies.push({
      id: newId,
      name: req.body.name,
      year: req.body.year,
      rating: req.body.rating
    });
    res.json({message: "New movie created.", location: "/movies/" + newId});
  }
});
```

This will create a new movie and store it in the movies variable. To check this route, enter the following code in your terminal −

```
curl -X POST --data "name = Toy%20story&year = 1995&rating = 8.5" http://localhost:3000/movies
```

The following response will be displayed −

```
{"message":"New movie created.","location":"/movies/105"}
```

To test if this was added to the movies object, Run the get request for /movies/105 again. The following response will be displayed −

```
{"id":105,"name":"Toy story","year":"1995","rating":"8.5"}
```