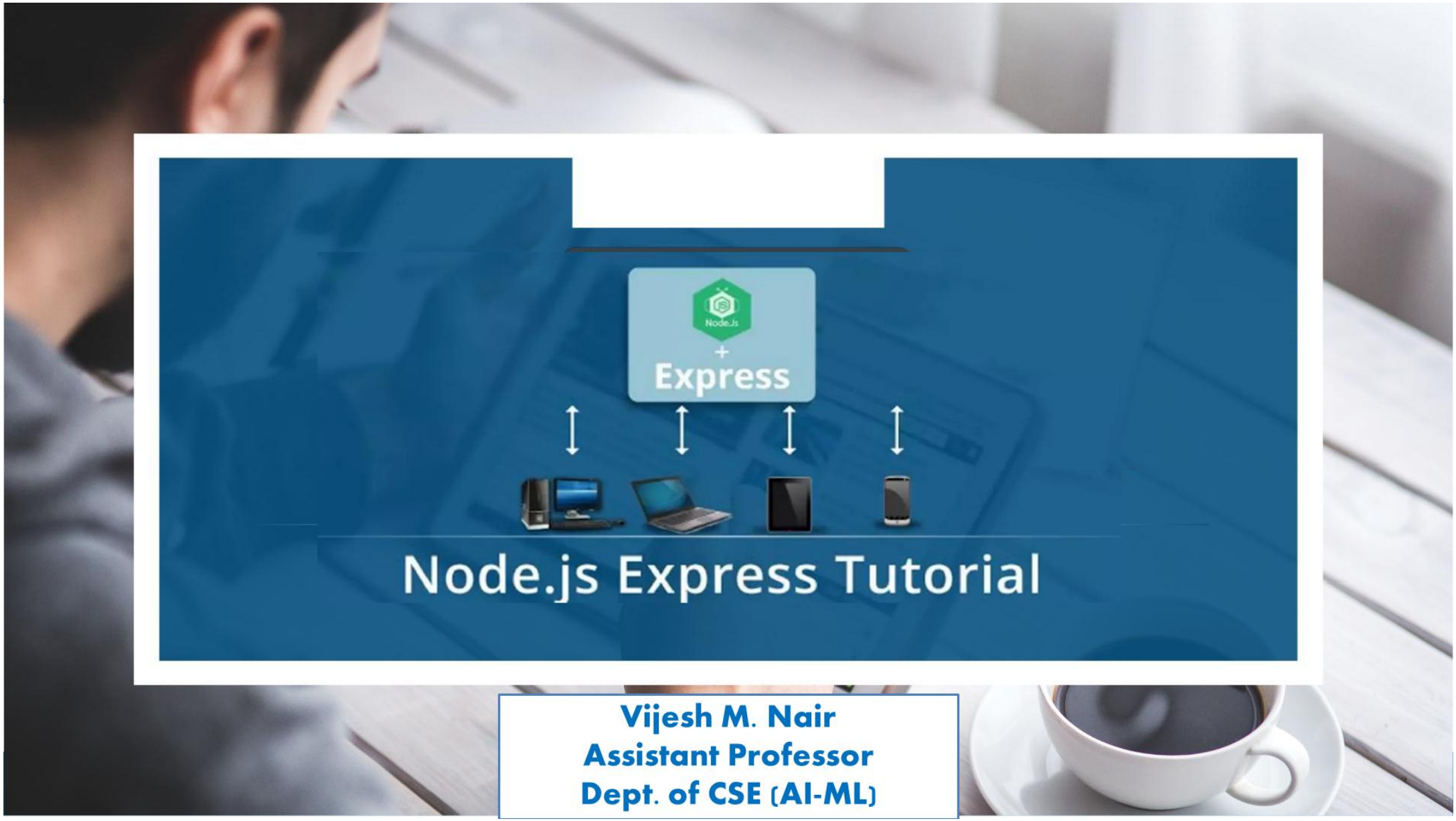
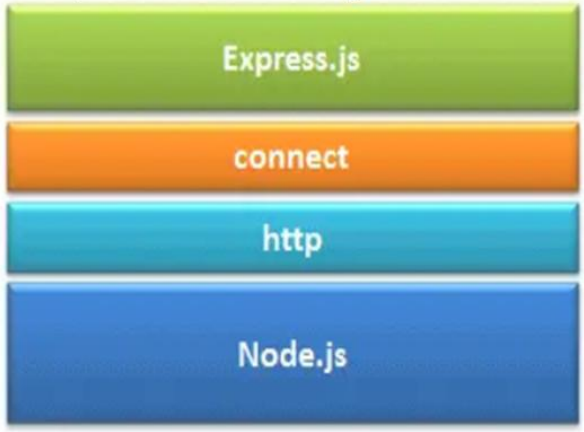


Vijesh M. Nair
Assistant Professor
Dept. of CSE (AI-ML)

What is Express ?

What is Express.js?

- Express.js is a web application framework that is built on top of Node.js.
- It provides a minimal interface with all the tools required to build a web application.
- It helps in easy management of the flow of data between server and routes in the server-side applications.
- It was developed by TJ Holowaychuk and was released in the market on 22nd of May, 2010.
- Formerly it was managed by IBM but currently, it is placed under the stewardship of the Node.js Foundation incubator.
- Express.js is based on the Node.js middleware module called connect which in turn uses http module. So, any middleware which is based on connect will also work with Express.js.



Features of Express.js

Features of Express.js

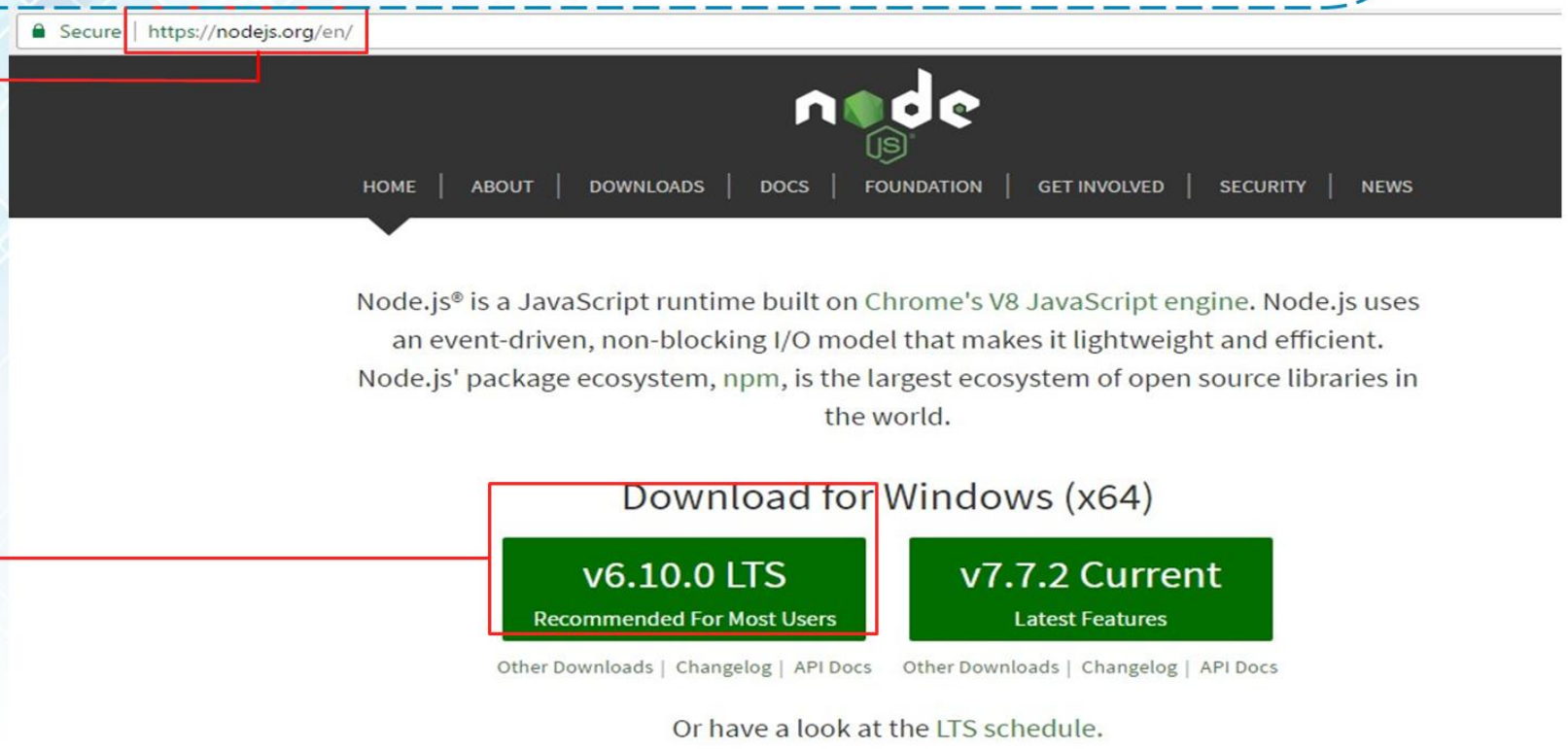
- Express is majorly responsible for handling the backend part in the MEAN stack.
- Mean Stack is the open-source JavaScript software stack that is heavily used for building dynamic websites and web applications in the market.
- Here, MEAN stands for MongoDB, Express.js, AngularJS, and Node.js.
- Express quickens the development pace of a web application.
- It also helps in creating mobile and web application of single-page, multi-page, and hybrid types.
- Express can work with various templating engines such as Pug, Mustache, and EJS.
- Express follows the Model-View-Controller (MVC) architecture.
- It makes the integration process with databases such as MongoDB, Redis, MySQL effortless.
- Express also defines an error-handling middleware.
- It helps in simplifying the configuration and customization steps for the application.

Express.js Installation

Express.js Installation

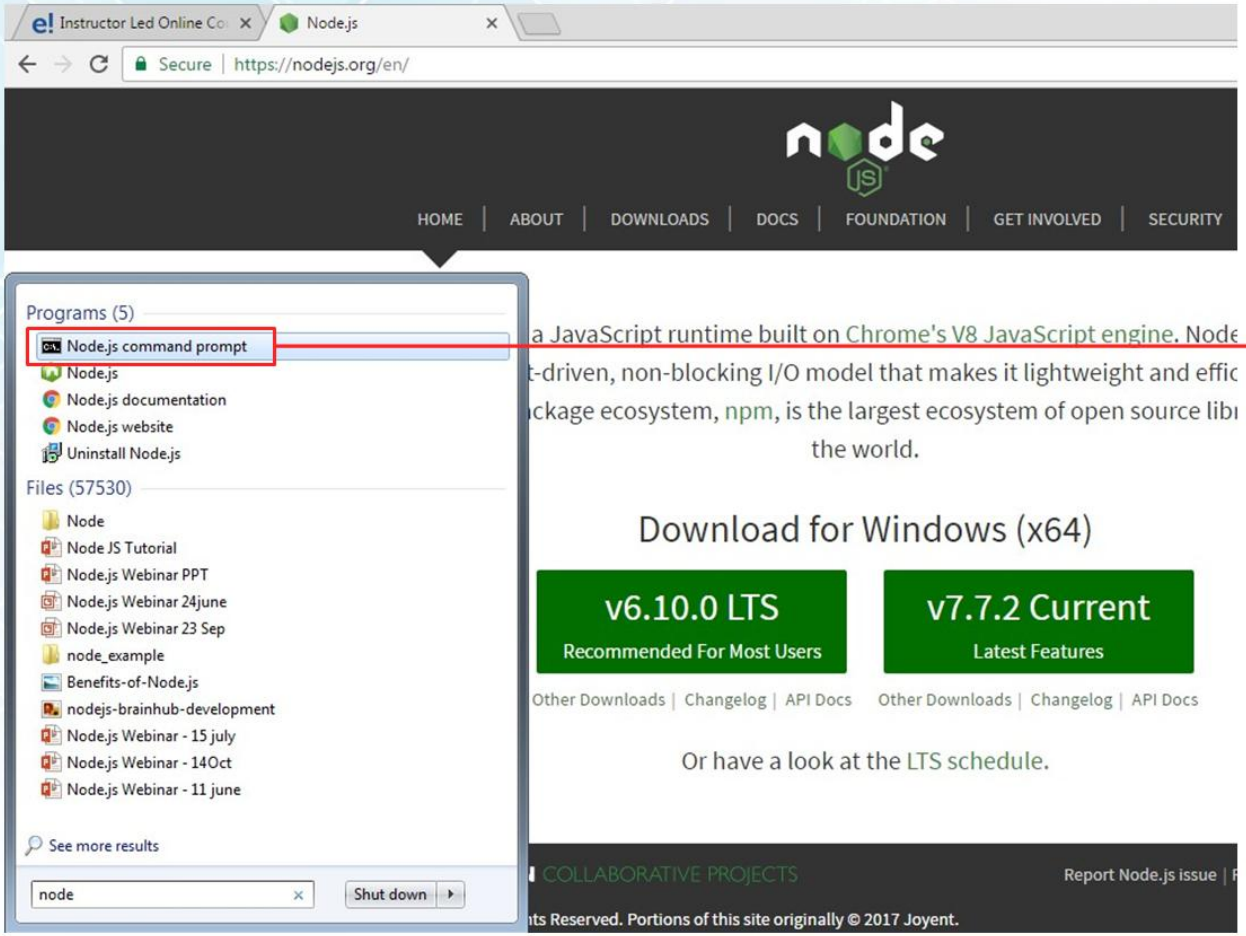
➢In order to install Express.js in your system, first, you need to make sure that you have Node.js & npm already installed.

1 Go to *https://nodejs.org/en/*



2 Download and Install

Node.js Installation



3 Open node.js command prompt

- node –version

Check the version of Node.js installed

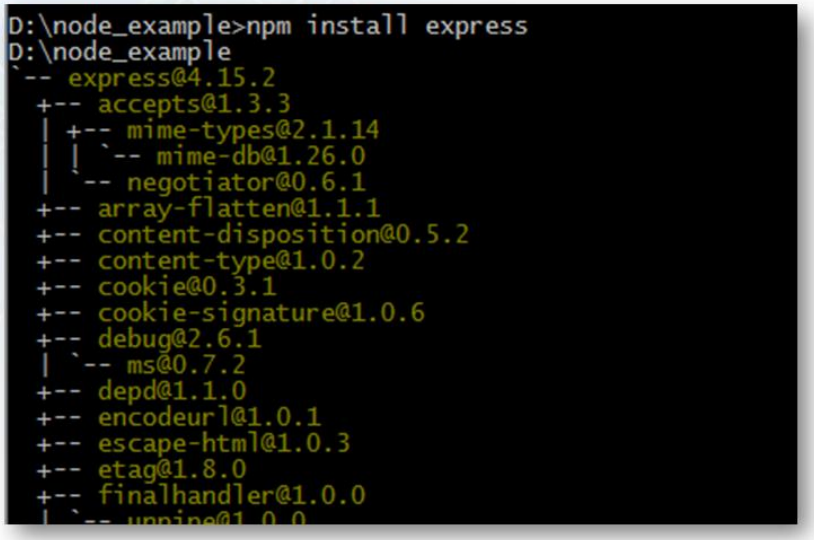
NPM

- NPM stands for Node Package Manager
- Provides online repositories for node.js packages/modules
- Provides command line utility to install Node.js packages along with version management and dependency management.

```
D:\node_example>npm version
{ npm: '3.10.10',
  ares: '1.10.1-DEV',
  http_parser: '2.7.0',
  icu: '57.1',
  modules: '48',
  node: '6.9.5',
  openssl: '1.0.2k',
  uv: '1.9.1',
  v8: '5.1.281.89',
  zlib: '1.2.8' }

D:\node_example>
```

NPM

npm install	Install all the modules as specified in package.json
npm install <**Module** Name>	Install Module using npm
npm install <**Module** Name> **-g**	Install dependency globally



Express.js Installation

- **Step 1:** Creating a directory for our project and make that our working directory.
  - ✓ $ mkdir Express
  - ✓ $ cd Express
- **Step 2:** Using 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 Installing Express
- **Step 3:** Now, you can install the express.js package in your system. To install it globally, you can use the below command:
  - ✓ $ npm install -g express
- **Step 4:** Or, if you want to install it locally into your project folder, you need to execute the below command:
  - ✓ $ npm install express --save

Express.js Installation

Install other important modules along with express

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

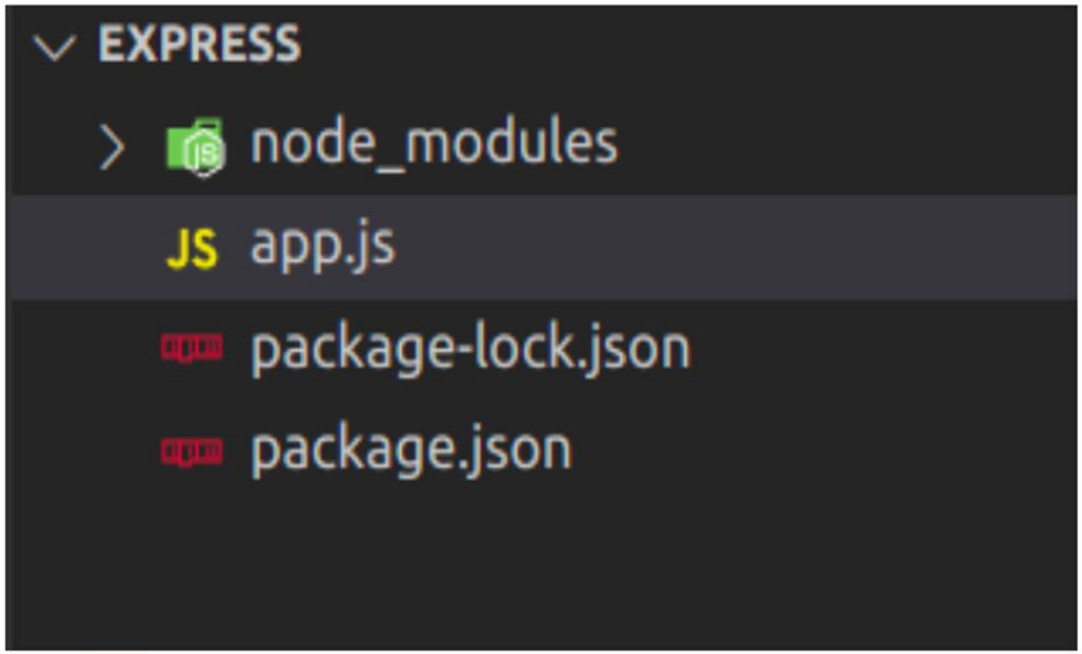
cookie-parser: It is used to parse Cookie header and populate req.cookies with an object keyed by the cookie names.

multer: This is a node.js middleware for handling multipart/form-data.

- **Step 5:**
  - ✓ npm install body-parser --save
  - ✓ npm install cookie-parser --save
  - ✓ npm install multer --save

Express.js Installation

Project Structure: It will look like the following.



Write the Source code in app.js.

Routes

Routing and HTTP Methods

- Routing is the process of determining a specific behavior of the application.
- It basically defines how an application should respond to a client request to a particular route, path or URI along with a particular HTTP Request method.
- Each route can contain more than one handler functions, which will be executed when the specific route is browsed by the user.
- Structure of Route definition ----- app.METHOD(PATH, HANDLER)
  - ✓ app is an instance of express where you can use any variable.
  - ✓ METHOD is an HTTP request method such as get, post, put, delete
  - ✓ PATH is the route to the server for a specific webpage
  - ✓ HANDLER is the callback function that is executed when the matching route is found.
- There are basically four main HTTP methods that can be supplied in the request which helps in specifying the operation requested by the user.

Routing and HTTP Methods

Method	Description

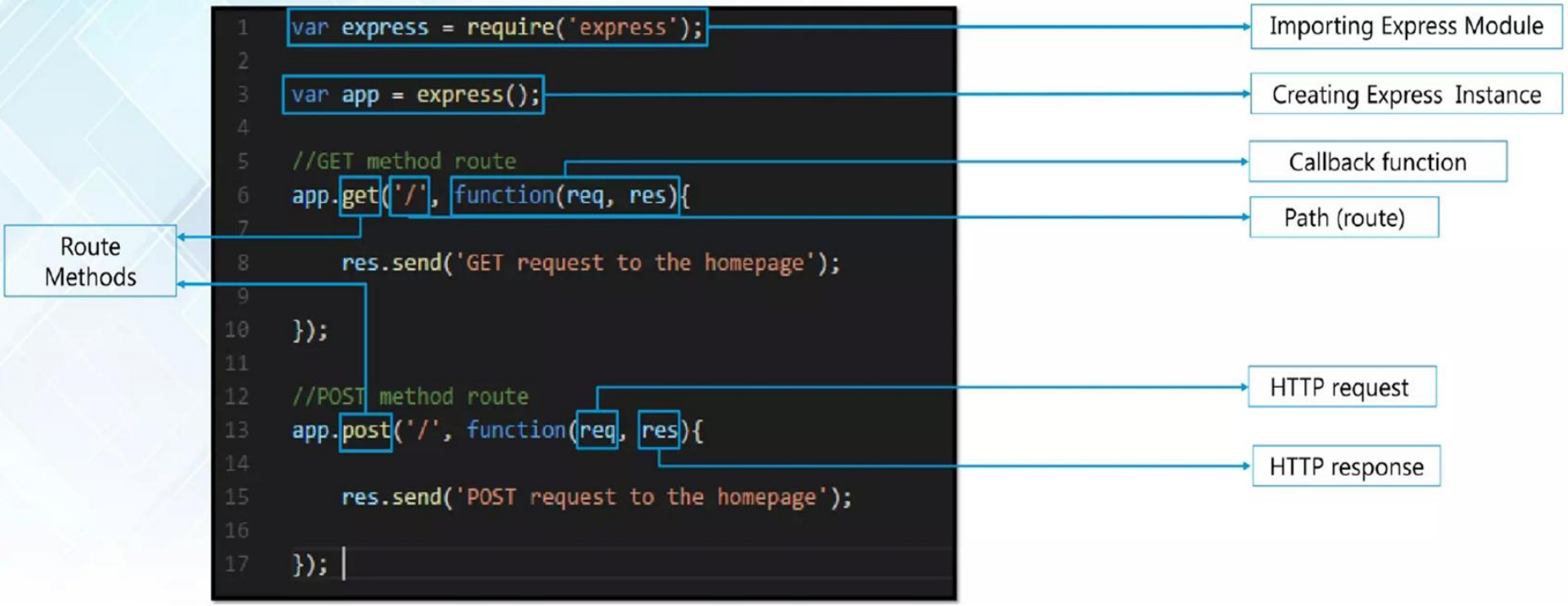
1. GET	The HTTP GET method helps in requesting for the representation of a specific resource by the client. The requests having GET just retrieves data and without causing any effect.
	`app.get('/', (req, res) => {`
	`res.send('Welcome to Node.js Express Tutorial!!');`
	`});`

2. POST	The HTTP POST method helps in requesting the server to accept the data that is enclosed within the request as a new object of the resource as identified by the URI. `app.post('/api/books', (req, res)=> { //Method Body });`
3. PUT	The HTTP PUT method helps in requesting the server to accept the data that is enclosed within the request as an alteration to the existing object which is identified by the provided URI. `app.put('/api/books/:id', (req, res) => { //method body });`
4. DELETE	The HTTP DELETE method helps in requesting the server to delete a specific resource from the destination. `app.delete('/api/books/:id', (req, res) => { //method body });`

Routes

Routing refers to the definition of application end points (URIs) and how they respond to client requests

A ***route method*** is derived from one of the HTTP methods, and is attached to an instance of the express class



Routes

app.all(), special routing method (not derived from any HTTP method)

app.all() is used for loading middleware functions at a path for all request methods

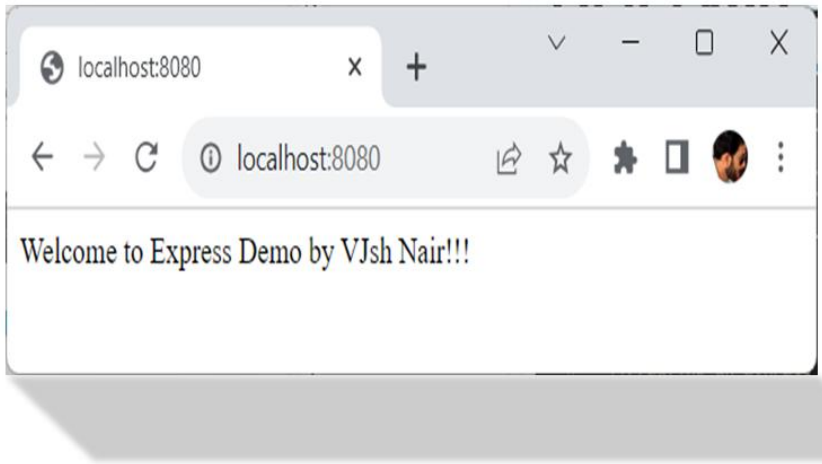
```
app.all('/secret', function (req, res, next) {
  console.log('Accessing the secret section ...')
  next() // pass control to the next handler
})
```

First Express.js Program

D: > node_example > Express > JS app.js > ...

```
1   //Importing express module
2   const express = require('express')
3
4   //Creating an express module object
5   const app = express()
6
7   //Creating Callback function and sending response
8   app.get('/', (req, res) => res.send('Welcome to Express Demo by VJsh Nair!!!'))
9
10  //Establish the server connection
11  //PORT ENVIRONMENT VARIABLE
12  const port = 8080;
13  app.listen(port, () => console.log(`Listening on port ${port}..`));
```

Output:



First Express.js Program - Explanation

- Import the express module.
- Next, step is to create the express module object so that you can use it in your application.
- Once done, you need to create a call back function which will be invoked when the user will try to browse the root of the application that i.e. http://localhost:8080.
- This callback function will be then responding with the String that you have passed and display on the webpage.
- In the callback function, the 'res' parameter is provided by the 'request' module to send the data back to the web page.
- Finally, we need to assign the port for the server. In this example, we are creating an environment variable to assign the port at 8080.
- Finally, you need to make use of the listen to function in order to make the server application listen to client requests on the assigned port.

Route Handler

Provide multiple callback functions which behave like middleware to handle a request

Exception -> Callbacks might invoke next('route') to bypass the remaining route callbacks

```
app.get('/example/a', function (req, res) {
  res.send('Hello from A!')
})
```

A single callback function can handle a route

```
app.get('/example/b', function (req, res, next) {
  console.log('the response will be sent by the next function ...')
  next()
}, function (req, res) {
  res.send('Hello from B!')
})
```

Next to bypass the remaining route callbacks

Used to impose pre-conditions on a route, then pass control to subsequent routes
(if no reason to proceed with the current route)

Route Handler

Route handlers can be in the form of a function, an array of functions, or combinations of both

Creating Functions

```
1  //An array of callback functions can handle a route. For example:
2
3  var cb0 = function (req, res, next) {
4    console.log('CB0')
5    next()
6  }
7
8  var cb1 = function (req, res, next) {
9    console.log('CB1')
10   next()
11 }
12
13 var cb2 = function (req, res) {
14   res.send('Hello from C!')
15 }
16
17 app.get('/example/c', [cb0, cb1, cb2])
```

```
1  //A combination of independent functions and arrays of functions
2
3  var cb0 = function (req, res, next) {
4    console.log('CB0')
5    next()
6  }
7
8  var cb1 = function (req, res, next) {
9    console.log('CB1')
10   next()
11 }
12
13 app.get('/example/d', [cb0, cb1], function (req, res, next) {
14   console.log('the response will be sent by the next function ...')
15   next()
16 }, function (req, res) {
17   res.send('Hello from D!')
18 })
```

Router handlers in form of functions & array of functions

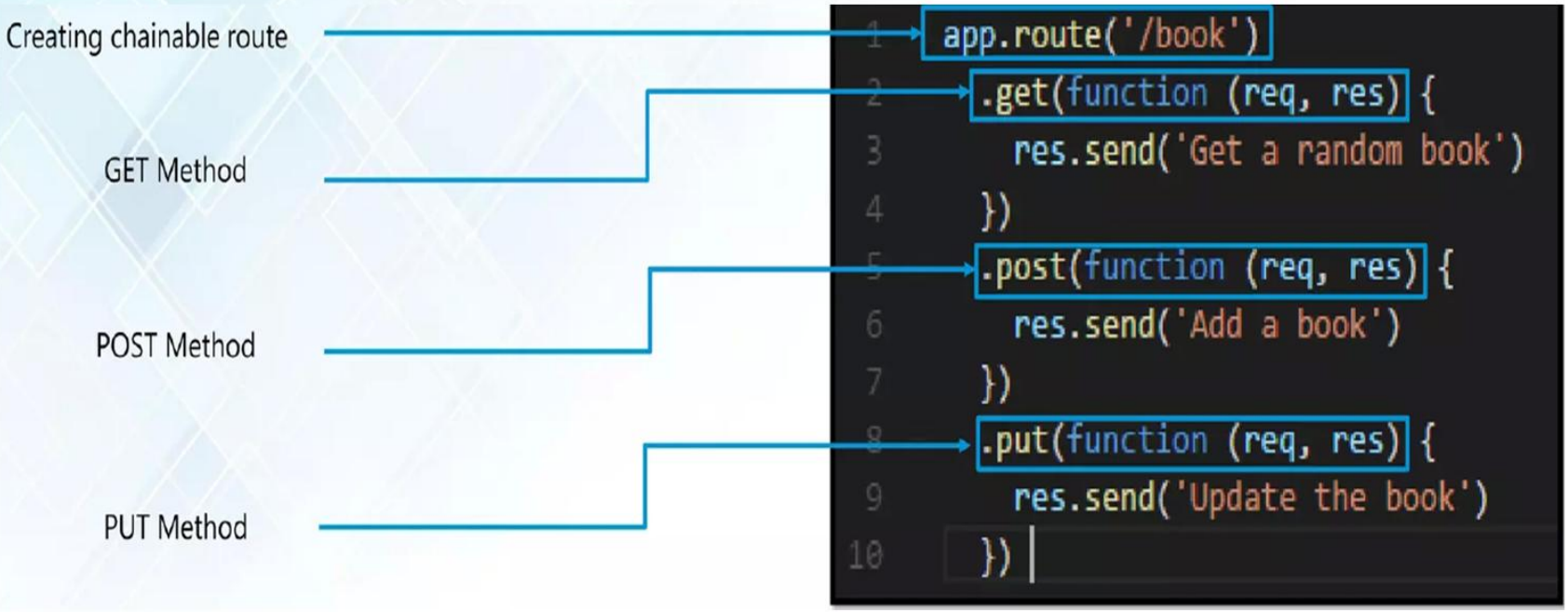
Response Methods

- Methods on the response object (res) for sending a response to the client
- Terminate the request-response cycle
- If none of these methods are called, the client request will be left pending

Method	Description
res.download()	Prompt a file to be downloaded.
res.end()	End the response process.
res.json()	Send a JSON response.
res.jsonp()	Send a JSON response with JSONP support.
res.redirect()	Redirect a request.
res.render()	Render a view template.
res.send()	Send a response of various types.
res.sendFile()	Send a file as an octet stream.
res.sendStatus()	Set the response status code and send its string representation as the response body.

Routes (app.route)

- Create chainable route handlers for a route path by using *app.route()*
- Path is specified at a single location
- Creating modular routes is helpful, as it reduces redundancy and typos



Express router

express Router

- Defining routes like previous example is very tedious to maintain. To separate the routes from our main **app.js** file, we will use **Express.Router**.
- **express.Router()** function is used to create a new router object.
- This function is used when you want to create a new router object in your program to handle requests.
- Multiple requests can be easily differentiated with the help of the **Router()** function in Express.js.

Syntax:

```
express.Router( [options] )
```

Optional Parameters:

- **case-sensitive:** This enables case sensitivity.
- **mergeParams:** It preserves the req.params values from the parent router.
- **strict:** This enables strict routing.

Return Value: This function returns the New Router Object.

express Router

```
var express = require('express')
var router = express.Router()

// middleware that is specific to this router
router.use(function timeLog (req, res, next) {
  console.log('Time: ', Date.now())
  next()
})
// define the home page route
router.get('/', function (req, res) {
  res.send('Birds home page')
})
// define the about route
router.get('/about', function (req, res) {
  res.send('About birds')
})

module.exports = router
```

Creates a router as a module

Loads a middleware function

Defines Home Route

Define About Route

- Router class to create modular, mountable route handlers
- A Router instance is a complete middleware and routing system

```
//Then, load the router module in the app:

var birds = require('./birds')
// ...
app.use('/birds', birds)
```

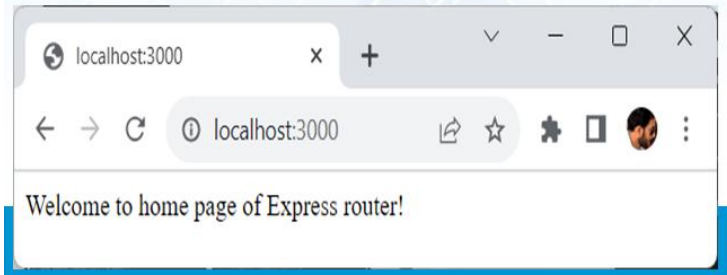
express Router - Example

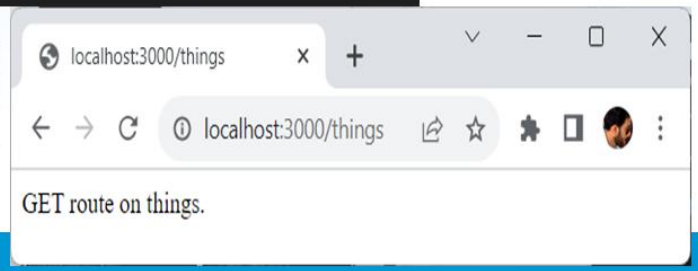
D: > node_example > Express > EX_2 > JS app.js > ...

```
1   var express = require('Express');
2   var app = express();
3   const port = 3000;
4
5   //Creating Callback function and sending response
6   app.get('/', (req, res) => {
7       console.log("GET Request Received from Parent directory");
8       res.send('Welcome to home page of Express router!')})
9
10  var things = require('./things.js');
11
12  //both index.js and things.js should be in same directory
13  app.use('/things', things);
14
15  app.listen(port,console.log("Server listening on PORT", port));
```

D: > node_example > Express > EX_2 > JS things.js > ...

```
1   var express = require('express');
2   var router = express.Router();
3
4   router.get('/', function(req, res){
5       console.log("GET Request Received from things directory");
6       res.send('GET route on things.');
7   });
8   router.post('/', function(req, res){
9       res.send('POST route on things.');
10  });
11
12  //export this router to use in our index.js
13  module.exports = router;
```





Output

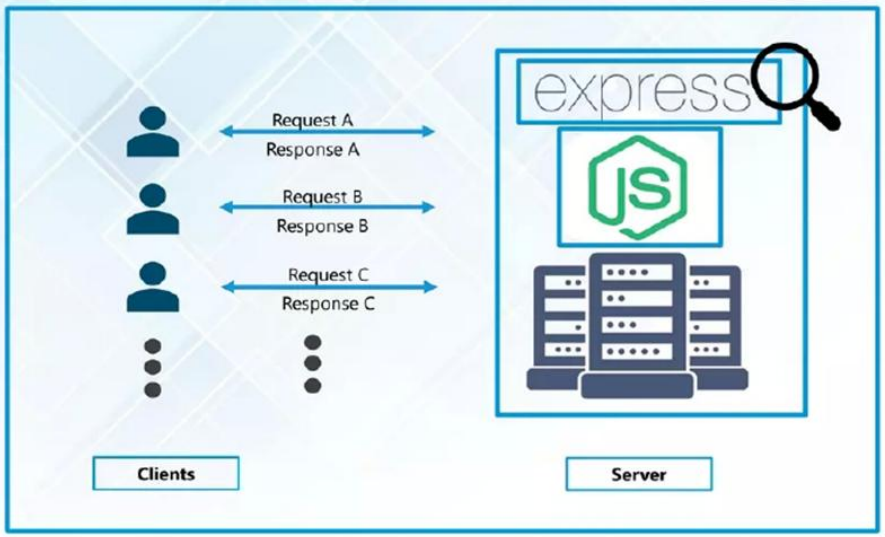
```
D:\node_example\Express\EX_2>node app.js
Server listening on PORT 3000
GET Request Received from Parent directory
GET Request Received from things directory
```

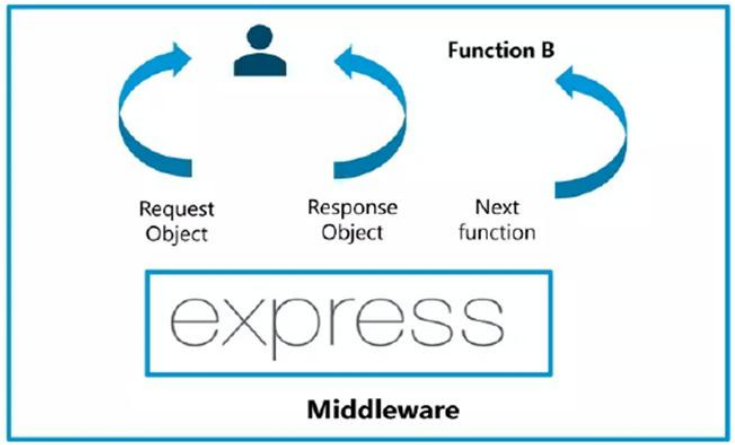
Middleware

Middleware

Middleware functions are functions that have access to the *request object* (req), the *response object* (res), and the *next function* in the application's request-response cycle.

next function is invoked to executes the middleware succeeding the current middleware



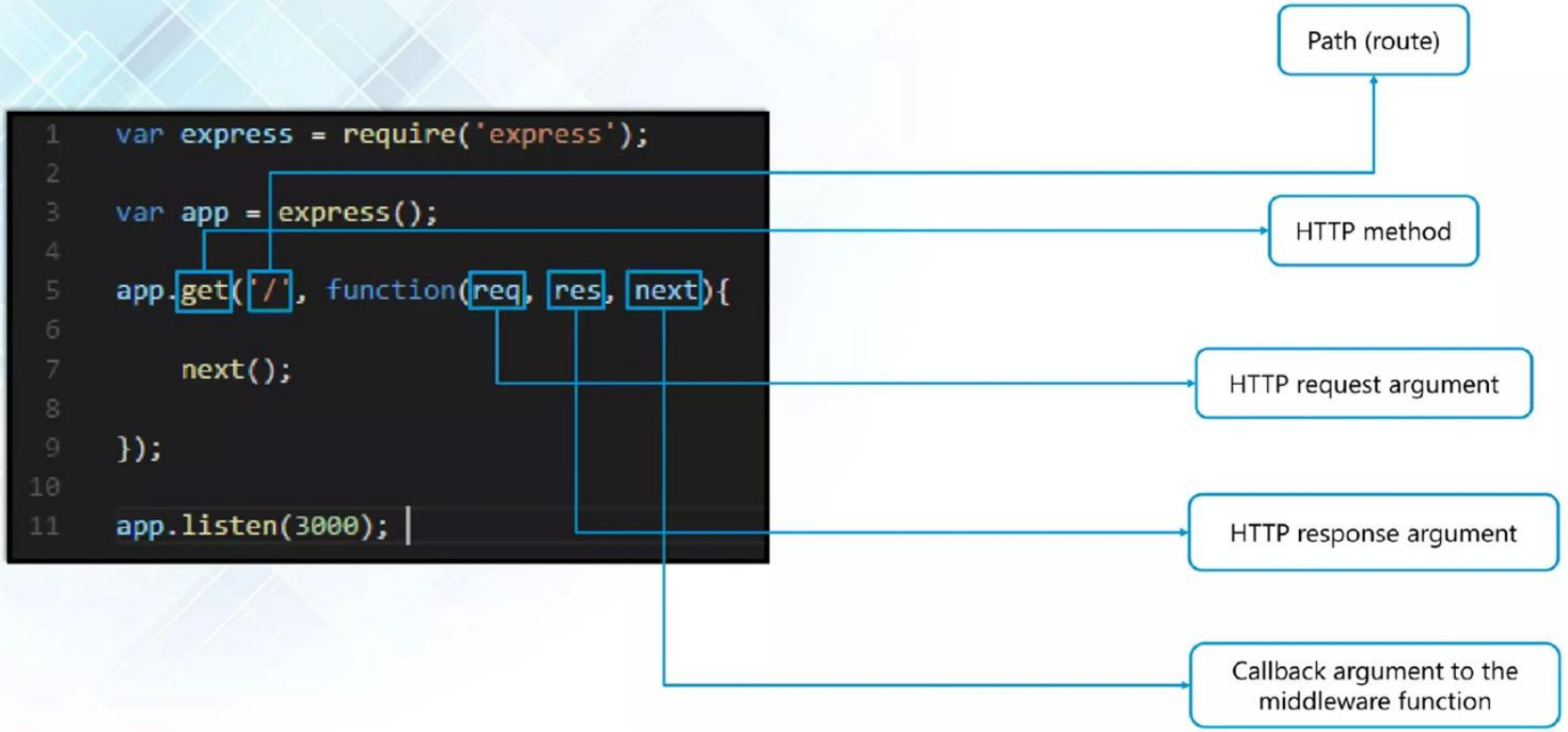


Middleware

Middleware functions performs the following tasks:

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

Middleware


1 var express = require('express');
2
3 var app = express();
4
5 app.get('/', function(req, res, next){
6
7 next();
8
9 });
10
11 app.listen(3000);
Path (route)
HTTP method
HTTP request argument
HTTP response argument
Callback argument to the middleware function

Middleware

```
1   var express = require('express')
2   var app = express()
3
4   var requestTime = function (req, res, next) {
5     req.requestTime = Date.now()
6     next()
7   }
8
9   app.use(requestTime)
10
11  app.get('/', function (req, res) {
12    var responseText = 'Hello World!<br>'
13    responseText += '<small>Requested at: ' + req.requestTime + '</small>'
14    res.send(responseText)
15  })
16
17  app.listen(3000);
```

Uses the *requestTime* middleware function

Application-Level Middleware

Bind application-level middleware to an instance (*app.use()* & *app.METHOD()*)

```
1   var app = express()
2
3   app.use(function (req, res, next) {
4     console.log('Time:', Date.now())
5     next()
6   })
```

Router-Level Middleware

- Router-level middleware binds to an instance of express.Router()
- Works in the same way as application-level middleware

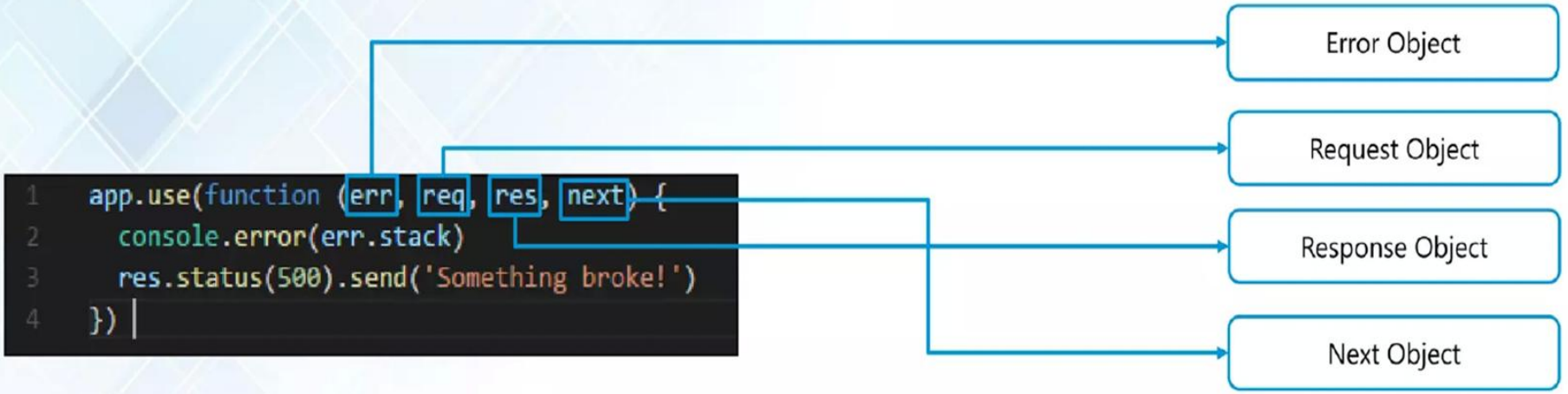
```
1  var app = express()
2  var router = express.Router()
3
4  // a middleware function with no mount path. This code is executed for every request to the router
5  router.use(function (req, res, next) {
6    console.log('Time:', Date.now())
7    next()
8  })
9
10 // a middleware sub-stack shows request info for any type of HTTP request to the /user/:id path
11 router.use('/user/:id', function (req, res, next) {
12   console.log('Request URL:', req.originalUrl)
13   next()
14 }, function (req, res, next) {
15   console.log('Request Type:', req.method)
16   next()
17 })
```

Router middleware i.e. executed for every router request

Router middleware i.e. executed for given path

Error-Handling Middleware

- Error-handling middleware always takes ***four*** arguments: *(err, req, res, next)*
- next object will be interpreted as regular middleware and will fail to handle errors
- Define error-handling middleware functions in the same way as other middleware functions



Built-in Middleware

- Except express.static, all of the middleware functions that were previously bundled with Express are now in separate modules
- *express.static* function is responsible for serving static assets such as HTML files, images, etc.

Independent Module

These middleware and libraries are officially supported by the Connect/Express team:

Modules	Description
body-parser	Parse incoming request bodies in a middleware before your handler
compression	Node.js compression middleware
connect-timeout	Times out a request in the Express application framework
cookie-parser	Parse Cookie header and populate req.cookies with an object keyed by the cookie names
cookie-session	Simple cookie-based session middleware
csurf	Node.js CSRF protection middleware
errorhandler	Error handler middleware
express-session	Create a session middleware
method-override	Create a new middleware function to override the req.method property with a new value
morgan	Create a new morgan logger middleware function
response-time	Creates a middleware that records the response time for requests in HTTP servers
serve-favicon	Middleware for serving a favicon
serve-index	Serves pages that contain directory listings for a given path
serve-static	Middleware function to serve files from within a given root directory
vhost	Middleware function to hand off request to handle, for the incoming host

Template Engines with Express

Template Engines with Express

- Template engine enables you to use static template files in your application
- Easier to design an HTML page
- Popular template engines: Pug, Mustache, and EJS
- Express application generator uses Jade as its default

```
app.get('/', function (req, res) {
  res.render('index', { title: 'Hey', message: 'Hello there!' })
})
```

Rending Template inside the application

```
html
  head
    title= title
  body
    h1= message
```

Declaring HTML template using Jade

Template Engines with Express

To render template files, set property in app.js :

- *views*, the directory where the template files are located

```
app.set('views', './views')
```

- *view engine*, the template engine to use

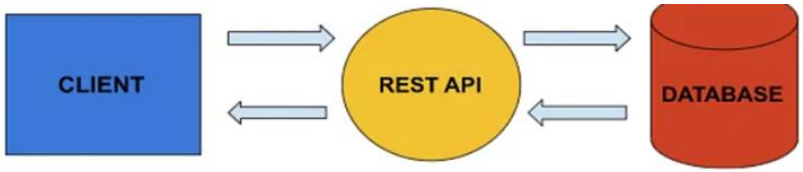
```
app.set('view engine', 'pug')
```

REST API

REST API

- An API (Application Programming Interface), as the name suggests, is an interface that defines the interaction between different software components.
- API (Application Programming Interface is a code that allows two software programs to communicate with each other.
- Web APIs define what requests can be made to a component (for example, an endpoint to get a list of books), how to make them (for example, a GET request), and their expected responses.
- REST (Representational State Transfer) is a standard architecture for building and communicating with web services.
- It is an industry-standard way for web services to send and receive data. They use HTTP request methods to facilitate the request-response cycle and typically transfer data using JSON, and more rarely - HTML, XML and other formats.

REST API



- Restful API is very popular and commonly used to create APIs for web-based applications. Express is a back-end web application framework of node js, and with the help of express, we can create an API very easily.
- 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.
- Express JS is one of the most popular HTTP server libraries for Node.js, which by default isn't as friendly for API development.
- Using Express, we simplify API development by abstracting away the boilerplate needed to set up a server, which makes development faster, more readable and simpler.
- We can spin up a prototype API in seconds and a couple of lines of code.

HTTP Request Types

There are a few types of HTTP methods that we need to grasp before building a REST API.

These are the methods that correspond to the CRUD tasks:

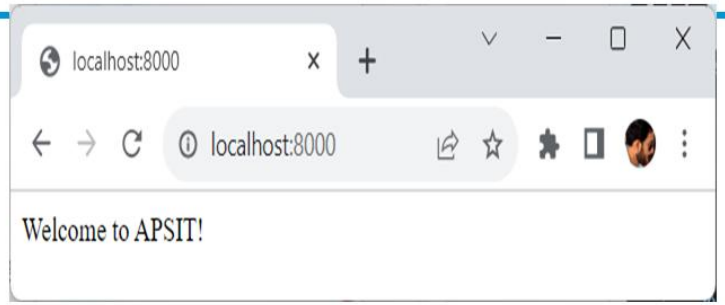
- `POST`: Used to submit data, typically used to *create* new entities or edit already existing entities.
- `GET`: Used to request data from the server, typically used to *read* data.
- `PUT`: Used to completely replace the resource with the submitted resource, typically used to *update* data.
- `DELETE`: Used to *delete* an entity from the server.

Note: Notice that you can use either `POST` or `PUT` to edit stored data. You're free to choose whether you even want to use `PUT` since it can be omitted fully. Though, stay consistent with the HTTP verbs you use. If you're using `POST` to both create and update, then don't use the `PUT` method at all.

REST API - Example

D: > node_example > Express > EX_2 > JS rest.js > ...

```
1   var express = require('express');
2   var app = express();
3   app.get('/', function (req, res) {
4      console.log("Got a GET request for the homepage");
5      res.send('Welcome to APSIT!');
6   })
7   app.post('/', function (req, res) {
8      console.log("Got a POST request for the homepage");
9      res.send('I am Impossible! ');
10  })
11  app.delete('/del_student', function (req, res) {
12     console.log("Got a DELETE request for /del_student");
13     res.send('I am Deleted!');
14  })
15  app.get('/enrolled_student', function (req, res) {
16     console.log("Got a GET request for /enrolled_student");
17     res.send('I am an enrolled student of APSIT.');
18  })
19  // This responds a GET request for abcd, abxcd, ab123cd, and so on
20  app.get('/ab*cd', function(req, res) {
21     console.log("Got a GET request for /ab*cd");
22     res.send('Pattern Matched.');
23  })
24  var server = app.listen(8000, function () {
25  var host = server.address().address
26    var port = server.address().port
27  console.log("Example app listening at http://%s:%s", host, port)
28  })
```

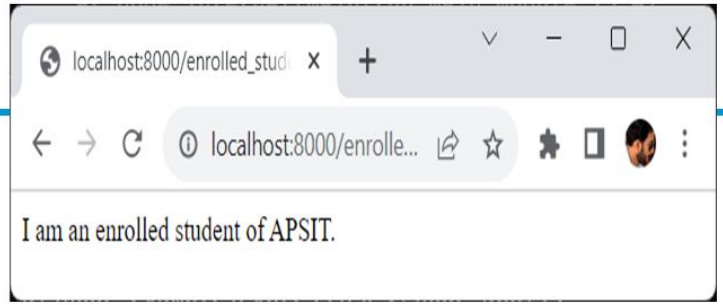



```
D:\node_example\Express\EX_2>node rest.js
Example app listening at http://:::8000
Got a GET request for the homepage
```

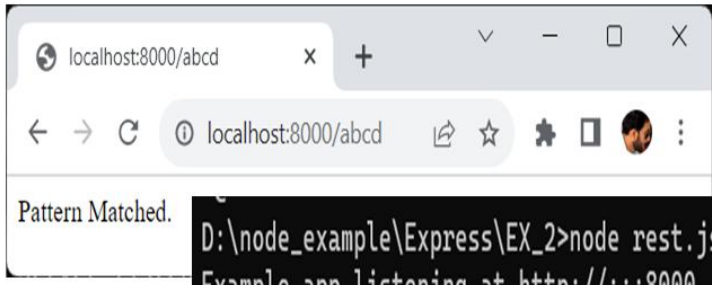
REST API - Example

D: > node_example > Express > EX_2 > JS rest.js > ...

```
1   var express = require('express');
2   var app = express();
3   app.get('/', function (req, res) {
4      console.log("Got a GET request for the homepage");
5      res.send('Welcome to APSIT!');
6   })
7   app.post('/', function (req, res) {
8      console.log("Got a POST request for the homepage");
9      res.send('I am Impossible! ');
10  })
11  app.delete('/del_student', function (req, res) {
12     console.log("Got a DELETE request for /del_student");
13     res.send('I am Deleted!');
14  })
15  app.get('/enrolled_student', function (req, res) {
16     console.log("Got a GET request for /enrolled_student");
17     res.send('I am an enrolled student of APSIT.');
18  })
19  // This responds a GET request for abcd, abxcd, ab123cd, and so on
20  app.get('/ab*cd', function(req, res) {
21     console.log("Got a GET request for /ab*cd");
22     res.send('Pattern Matched.');
23  })
24  var server = app.listen(8000, function () {
25  var host = server.address().address
26    var port = server.address().port
27  console.log("Example app listening at http://%s:%s", host, port)
28  })
```




```
D:\node_example\Express\EX_2>node rest.js
Example app listening at http://:::8000
Got a GET request for the homepage
Got a GET request for /enrolled_student
```

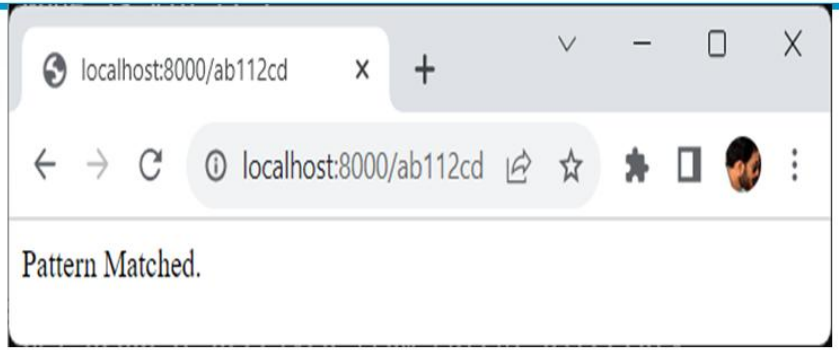



```
D:\node_example\Express\EX_2>node rest.js
Example app listening at http://:::8000
Got a GET request for the homepage
Got a GET request for /enrolled_student
Got a GET request for /ab*cd
```

REST API - Example

D: > node_example > Express > EX_2 > JS rest.js > ...

```
1   var express = require('express');
2   var app = express();
3   app.get('/', function (req, res) {
4      console.log("Got a GET request for the homepage");
5      res.send('Welcome to APSIT!');
6   })
7   app.post('/', function (req, res) {
8      console.log("Got a POST request for the homepage");
9      res.send('I am Impossible! ');
10  })
11  app.delete('/del_student', function (req, res) {
12     console.log("Got a DELETE request for /del_student");
13     res.send('I am Deleted!');
14  })
15  app.get('/enrolled_student', function (req, res) {
16     console.log("Got a GET request for /enrolled_student");
17     res.send('I am an enrolled student of APSIT.');
18  })
19  // This responds a GET request for abcd, abxcd, ab123cd, and so on
20  app.get('/ab*cd', function(req, res) {
21     console.log("Got a GET request for /ab*cd");
22     res.send('Pattern Matched.');
23  })
24  var server = app.listen(8000, function () {
25  var host = server.address().address
26    var port = server.address().port
27  console.log("Example app listening at http://%s:%s", host, port)
28  })
```




```
D:\node_example\Express\EX_2>node rest.js
Example app listening at http://:::8000
Got a GET request for the homepage
Got a GET request for /enrolled_student
Got a GET request for /ab*cd
Got a GET request for /ab*cd
```

Express Generator

Express Generator

- **Express Generator** is a Node.js Framework like ExpressJS which is used to create express Applications easily and quickly. It acts as a tool for generating express applications.
- **Features of Express-Generator:**
  - It generates express Applications in one go using only one command.
  - The generated site has a modular structure that we can modify according to our needs for our web application.
  - The generated file structure is easy to understand.
  - We can also configure options while creating our site like which type of view we want to use (For example, ejs, pug, and handlebars).

Express Generator - Installation

Note: You should have installed Node and Express before using Express-generator on your machine.

```
npm install express-generator -g
```

For Creating a Simple Express.js Web Application, Open command prompt/Terminal in your local fileSystem and execute the below command.

Syntax:

```
express <Your-ExpressJsApplication-Name>
```

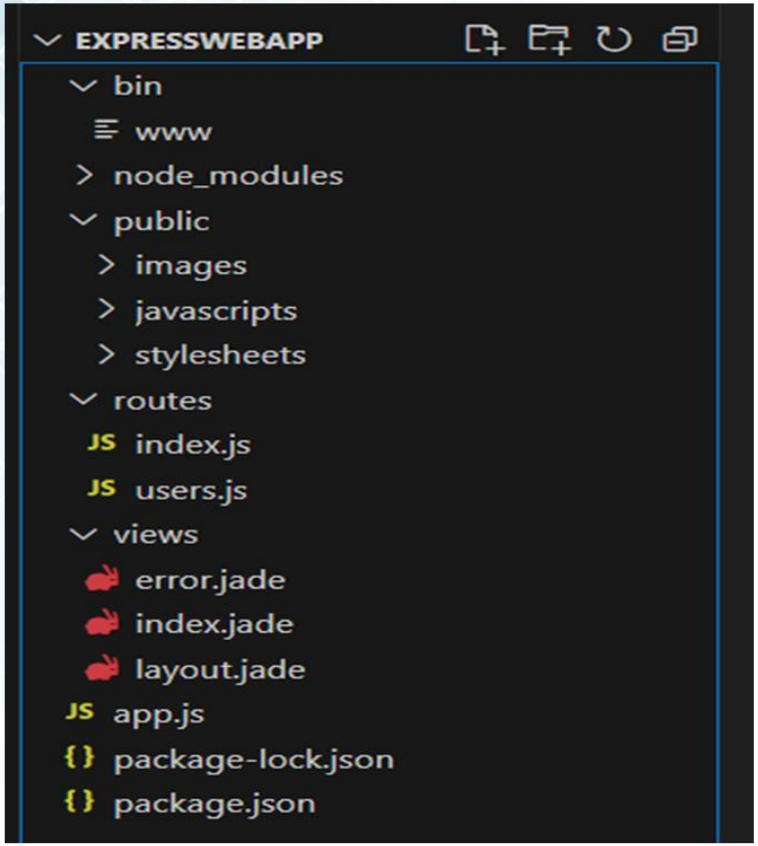
Example:

```
express ExpressWebApp
```

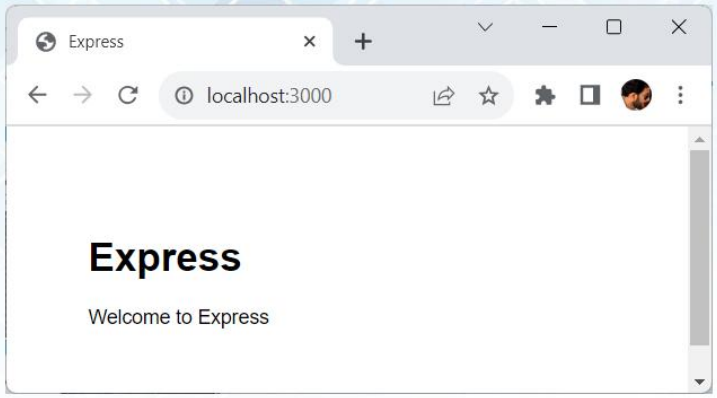
Express Generator - Installation

After creating the express-generator the structure looks like given below:

Express-generator Structure:



Example



app.js > ...

```
1   var createError = require('http-errors');
2   var express = require('express');
3   var path = require('path');
4   var cookieParser = require('cookie-parser');
5   var logger = require('morgan');
6
7   var indexRouter = require('./routes/index');
8   var usersRouter = require('./routes/users');
9
10  var app = express();
11
12  // view engine setup
13  app.set('views', path.join(__dirname, 'views'));
14  app.set('view engine', 'jade');
15
16  app.use(logger('dev'));
17  app.use(express.json());
18  app.use(express.urlencoded({ extended: false }));
19  app.use(cookieParser());
20  app.use(express.static(path.join(__dirname, 'public')));
21
22  app.use('/', indexRouter);
23  app.use('/users', usersRouter);
24
25  // catch 404 and forward to error handler
26  app.use(function(req, res, next) {
27    next(createError(404));
28  });
29
30  // error handler
31  app.use(function(err, req, res, next) {
32    // set locals, only providing error in development
33    res.locals.message = err.message;
34    res.locals.error = req.app.get('env') === 'development' ? err : {};
35
36    // render the error page
37    res.status(err.status || 500);
38    res.render('error');
39  });
40
41  module.exports = app;
```

Authentication and Authorization in Express

Vijesh Nair

Authentication

- Authentication is the process of verifying the identity of a user or entity. It ensures that the user claiming to be a particular person or entity is indeed who they say they are.
- In web applications, authentication is typically performed when a user tries to access protected resources or perform privileged actions.
- There are several commonly used authentication mechanisms:
  - ✓ **Username and password:** This is the most common method where users provide a username (or email) and a password to prove their identity.
  - ✓ **Token-based authentication:** Tokens, such as JSON Web Tokens (JWT), are generated and issued to users upon successful authentication. These tokens are then included in subsequent requests to authenticate and authorize the user.
  - ✓ **Single sign-on (SSO):** SSO allows users to authenticate once and gain access to multiple applications or services without providing their credentials repeatedly.

Authorization

- Authorization determines the actions or resources a user is allowed to access within an application after successful authentication.
- Once a user's identity is established, the authorization process enforces access control rules and permissions to ensure that the user can only perform their authorized actions.
- Authorization is typically based on roles, permissions, or policies assigned to users. Some common authorization mechanisms include:
  - ✓ **Role-based access control (RBAC):** Users are assigned specific roles (e.g., admin, moderator, user), and each role has a set of predefined permissions that determine what actions or resources they can access.
  - ✓ **Attribute-based access control (ABAC):** Access control decisions are made based on various attributes associated with users, resources, and environmental conditions.
  - ✓ **Policy-based access control:** Access control policies are defined in a centralized manner, specifying the conditions and rules for granting or denying access to resources.

JWT Token

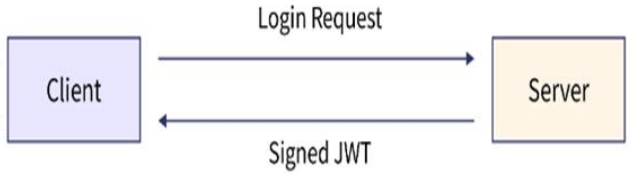
- JSON Web Token (JWT) is a compact, URL-safe means of representing claims between two parties.
- It is a self-contained token that contains JSON-encoded information, allowing for secure transmission of data between parties in a compact and verifiable format.
- JWTs are commonly used for authentication and authorization purposes in web applications.
- JWTs consist of three parts: a header, a payload, and a signature, each encoded and separated by dots.
- **Header:**
  - ✓ The header contains information about the type of token (JWT) and the algorithm used to sign it. It is typically encoded using Base64Url encoding.
- **Payload:**
  - ✓ The payload, also known as the claims, contains the actual data being transmitted.
  - ✓ It includes information about the user or any other relevant metadata.
  - ✓ The payload can contain registered claims (predefined standard claims) or custom claims (specific to the application). Like the header, the payload is also Base64Url encoded.

JWT Token

- **Signature:**
  - ✓ The signature is used to verify the integrity of the token and ensure that it has not been tampered with.
  - ✓ The signature is created by combining the encoded header, the encoded payload, and a secret key known only to the server.
  - ✓ The resulting string is hashed using a specified algorithm (such as HMACSHA256 or RSA) to produce the signature.

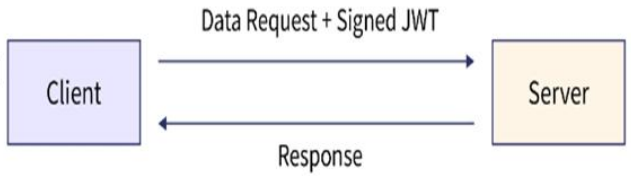
- **Structure of a JWT:-**

1. The header, which is a string that has been Base64-encoded, is the initial part of the JWT. It provides the hashing algorithm that was used to produce the token's sign and type. The header would appear something like this if it were decoded:




```
{ "alg": "HS256", "typ": "JWT"}
```

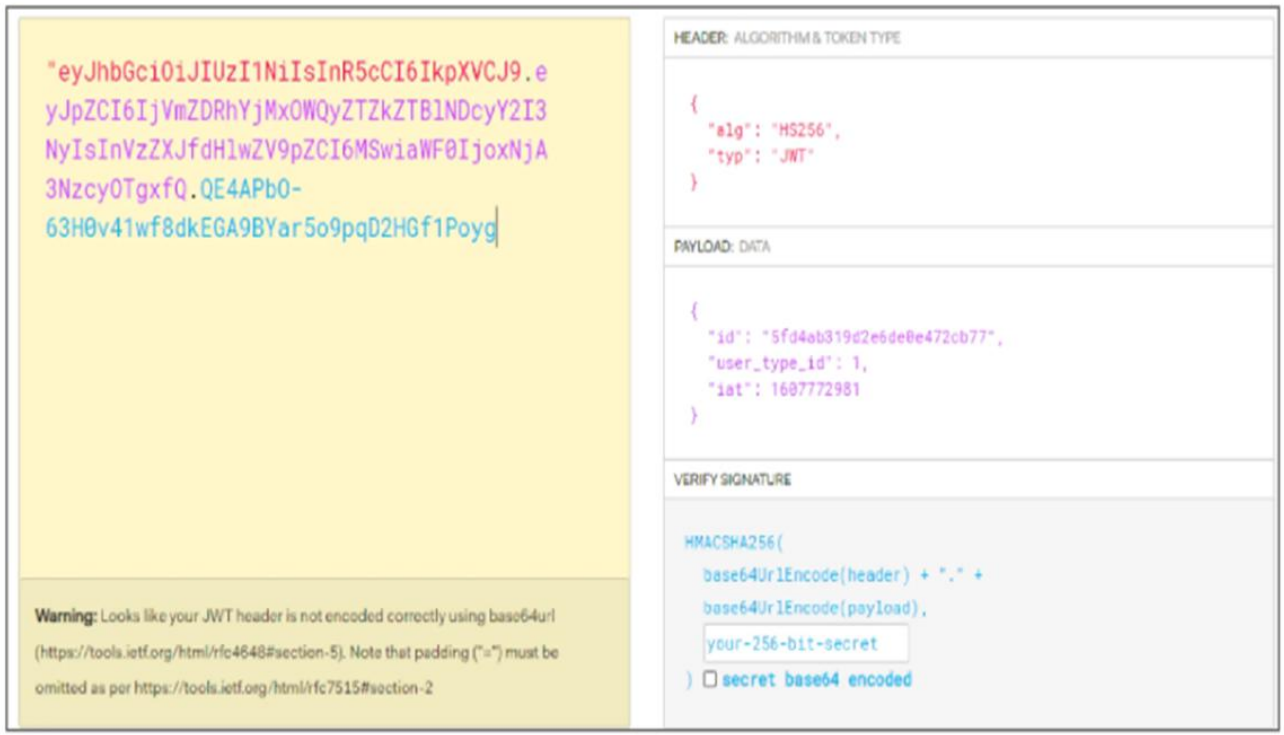
1. The payload, or the second portion, comprises the JSON object with user data, such as (id or id_type), which was returned to the user. This is merely Base64-encoded, so anyone can easily decode it.




```
{"sub": "1234567890", "name": "John Doe", "userId": "1516239022", "user_type_id":1 }
```

JWT Token

The token's signature is made in the final part using the procedure stated in the header section.



How to apply Authentication?

- Two user types (admin & user) will exist. All users must first log in and register, and only administrators and common users are authorized.
- Regular users will be able to view regular events, while administrators will be able to access special events.
- Suppose there are four routes and REST APIs to get this journey started:

  1- POST login route (everyone has access)

  2- POST register route (everyone has access)

  3- GET events array (only regular user has access)

  4- GET special events array (only admin user has access)

STEP-1

- Start by creating a blank Node.js project using the default settings in your terminal.

```
$ npm init -y
```

- The Express framework, JWT, bcryptjs, and mongoose should now be installed:

```
$ npm install --save express JSON web token bcryptjs
```

- Then Express will be used as the router to establish the fundamental framework for various endpoint types, such as registration and login. Additionally, we'll make a routers folder (routers/index.js).

- Next, let's create a file named (middleware/auth.js) that will serve as our authentication service and a file called (controllers/user) that will serve as our controller for user functions.

STEP-1

- Let's now build our server, use these modules, and set them up in the Express app (server.js):

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

app.use(express.json());

// Import Routes
const authRoute = require('./routes/index');

// Route Middlewares
app.use('/api', authRoute);

const port = 3000;
app.listen(port, function(){console.log("Server running on localhost:" + port);});
```

STEP-2

- Go to the routers/index.js folder to set the express Router import userController configuration now.

```
const router = require('express').Router();
const userController = require('../controllers/user');

// Register a new User
router.post('/register', userController.register);

// Login
router.post('/login', userController.login);

module.exports = router;
```

STEP-3

To add userController functions, navigate to the routers' controllers/user subdirectory now.

- Connect to DB

```
mongoose.connect(db, function(err){
    if(err){
        console.error('Error! ' + err)
    } else {
      console.log('Connected to mongodb')
    }
});
```

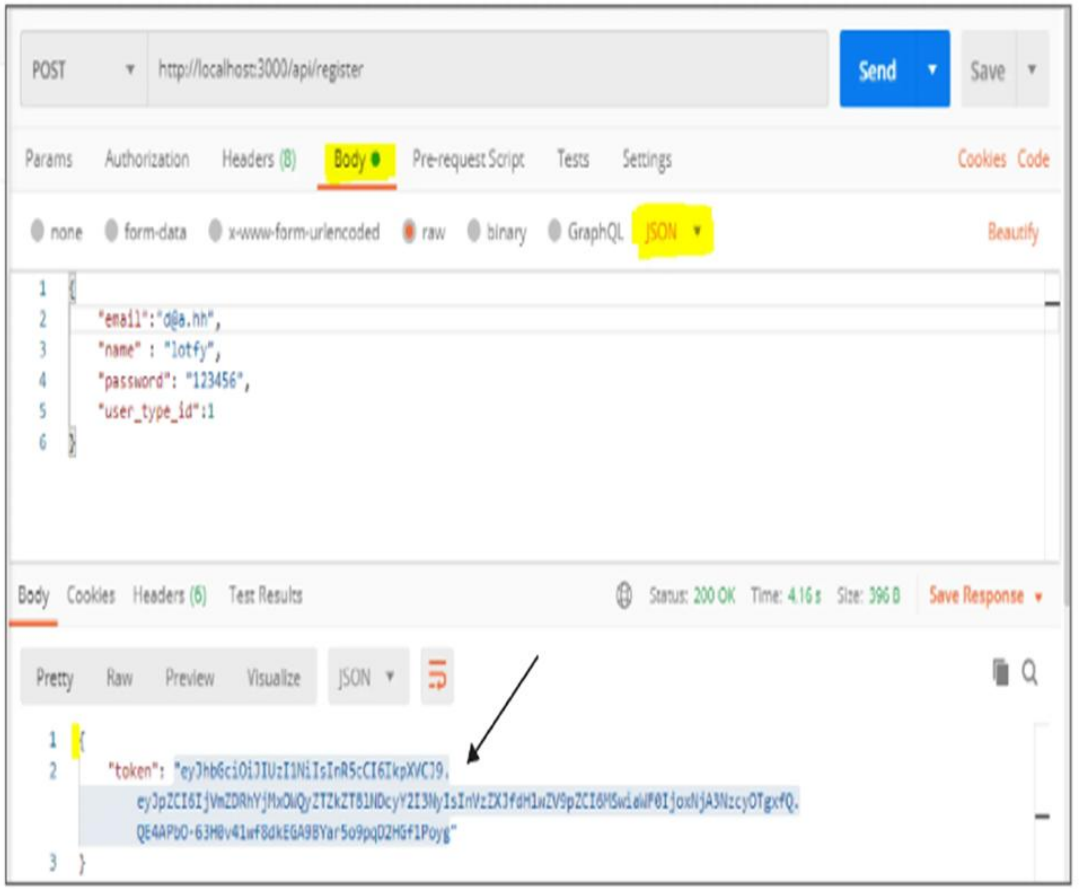
STEP-3

- Register function creation
  - Using the bcrypt module, hash a password
  - Construct a user object.
  - User saving in the database
  - then create the payload Create an access token (for information on payload, see the section on JWT structure).

```
exports.register = async (req, res) => {
    //Hash password
    const salt = await bcrypt.genSalt(10);
    const hasPassword = await bcrypt.hash(req.body.password, salt);

    // Create an user object
    let user = new User({
        email: req.body.email,
        name: req.body.name,
        password: hasPassword,
        user_type_id: req.body.user_type_id
    })

    // Save User in the database
    user.save((err, registeredUser) => {
        if (err) {
            console.log(err)
        } else {
            // create payload then Generate an access token
            let payload = { id: registeredUser._id, user_type_id: req.body.user_type_id || 0 };
            const token = jwt.sign(payload, config.TOKEN_SECRET);
            res.status(200).send({ token })
        }
    })
}
```

STEP-3

- Let's submit a POST request after the authentication service is operational to check whether or not registration is successful.
- To do this, I'll be utilizing Postman's rest-client. Use Insomnia or any other rest client of your choosing to complete this.
  - ✓ Let's provide the following JSON` to the link endpoint: "email": "d@a.hh", "name": "lotfy", "password": "123456", and "user_type_id": 1"
  - ✓ The response that you should receive is the access token: The "Token" is "eyJhbGciOiJIUz..."



STEP-4

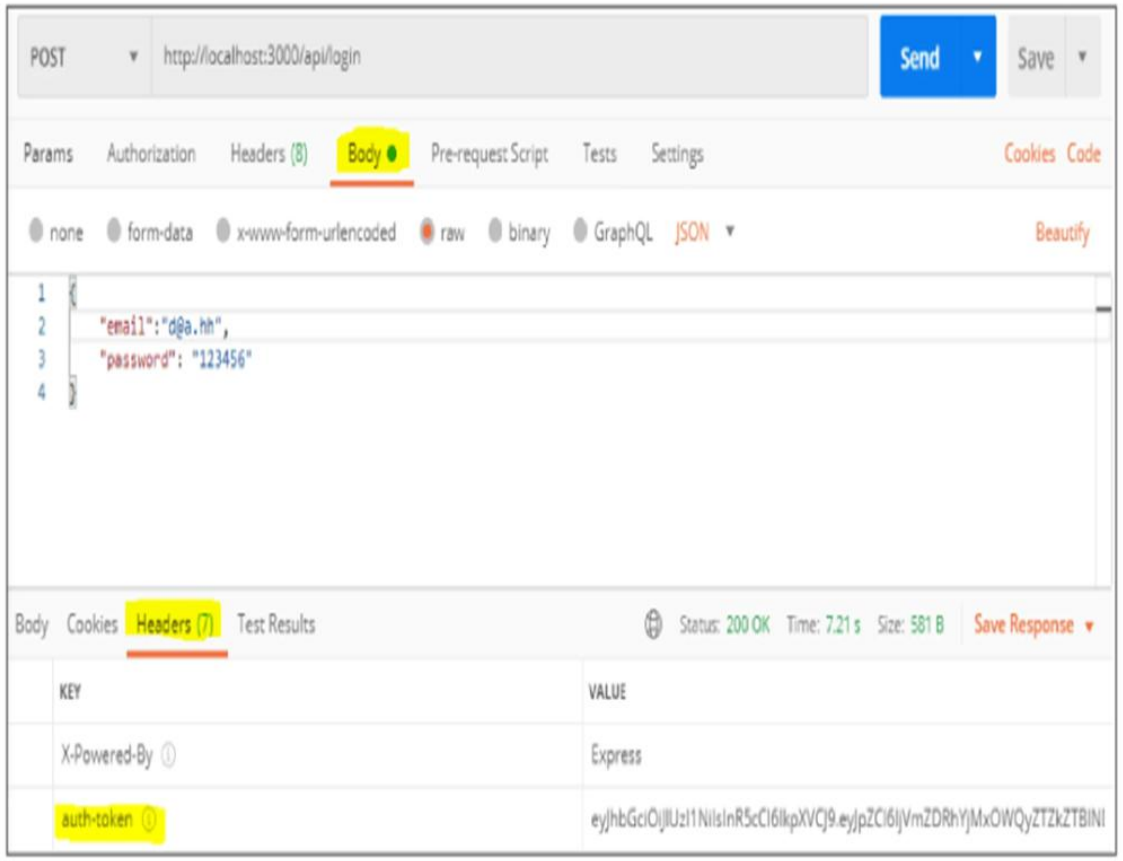
- Now that we have registered a new user, we must log in using their credentials after receiving a token as a response.
- create a login function
  - ✓ make login comparison functions credentials and a hashed password.
  - ✓ Construct the payload Create a return token and an access token in headers.

```
exports.login = async (req, res) => {

    User.findOne({ email: req. body.email }, async (err, user) => {
        if (err) {
            console.log(err)
        } else {
            if (user) {
                const valid pass = await bcrypt.compare(req.body.password, user.password);
                if (!validPass) return res.status(401).send("Mobile/Email or Password is wrong");

                // Create and assign token
                let payload = { id: user._id, user_type_id: user.user_type_id };
                const token = jwt.sign(payload, config.TOKEN_SECRET);

                res.status(200).header("auth-token", token).send({ "token": token });
            }
            else {
                res.status(401).send('Invalid mobile')
            }

        }
    })
}
```

STEP-3

- Let's submit a POST request to test whether not login is functional.

- ✓ Let's post a request with the following JSON to the link endpoint: "Password": "123456", "email": "d@a.hh".
- ✓ You should receive a 200 code and the followin access token in the response's header: "auth-token": "eyJhbGciOiJIUz."



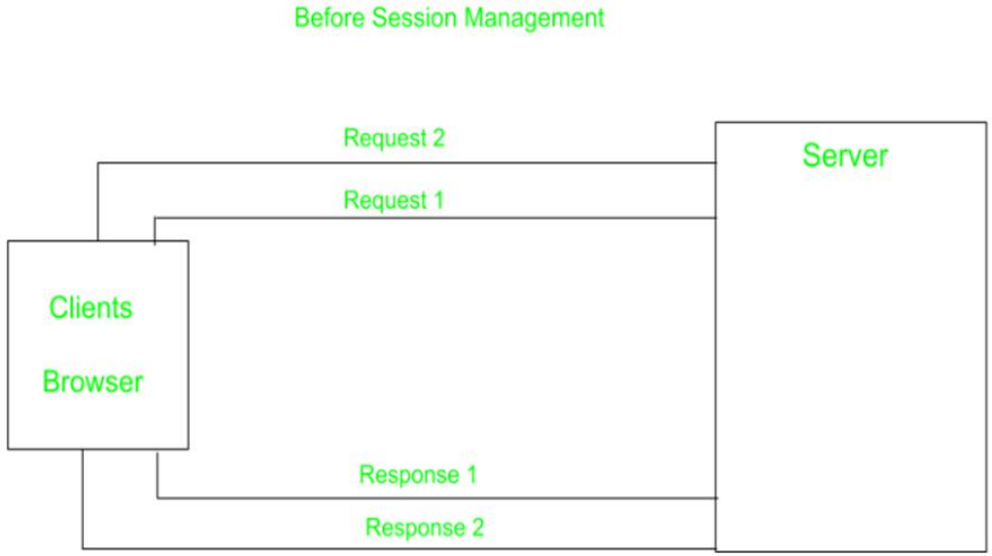
Sessions in Express

Session Management in Express

- HTTP is a stateless protocol, the client and server forget about one another after each request and response cycle.
- The stateless HTTP protocol is used by a website to transport data from a client to a server.
- Sessions enable the HTTP protocol to go from being stateless to stateful.
- A session can be thought of as the period between logging in and logging out.
- As the HTTP protocol is stateless, cookies let us monitor the status of the application using small files kept on the user's machine.
- Cookies stored by the browser can hold a maximum of 4kB (4096 bytes) in size.
- Hence, a session can be thought of as the period between logging in and logging out.
- Cookies are used to maintain this session. There are various cookie varieties. Session cookies are the type of cookies used for managing sessions.

Session Management in Express

- Session Management is a technique that keeps track of users' states (data).
- If the client sends a request to the server then it is followed by HTTP protocol (stateless).
- Stateless protocol means the server forgets about the client and treats every request as a new request.
- The state is managed by the session management techniques.

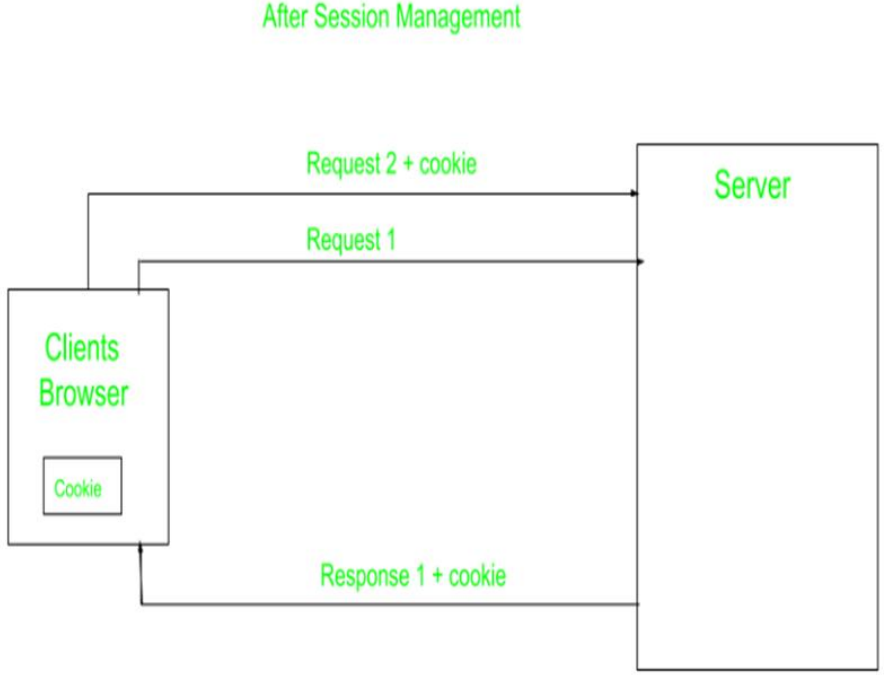


How do Sessions Work?

- The server will construct a session and save it on the server side after receiving a login request from the client.
- When the server answers to the client, it transmits a cookie.
- The unique id for the session that was previously stored on the server will be found in this cookie and will be kept on the client.
- Every time a request is made to the server, this cookie will be transmitted.
- To preserve a one-to-one match between a session and a cookie, we use this session ID to look for the session maintained in the database or the session store.
- As a result, HTTP protocol connections will become stateful.

The Flow of Cookie-based Session Management

1. New login request is sent by the browser to the server.
2. The server will then determine if any cookies have been sent by the browser.
3. There isn't going to be a cookie value inside the server database for this request because it is a new one.
4. As a result, the server will send a cookie to your browser and store its ID there.
5. The browser will then create the cookie for the domain of that server.
6. Your browser should deliver the cookie in the HTTP header with each request for that server's website.
7. The ID given by the browser will then be checked by Server. Then, the server will use the session indicated by the cookie if that is the case.



The Difference Between Session and Cookie

Cookie	Session
Cookies are client-side files stored locally on a computer that contain user data.	User data stored in the server side is called sessions.
Cookies expire when the user-defined lifespan expires.	The session ends when the user closes the browser or logs out of the software.
It has a limited capacity for information storage.	It has a practically infinite capacity for data storage.
We don't need to run a function to start cookies because they are stored locally on the machine.	The session start() function must be used to start the session.
Cookies are not secured.	When compared to cookies, sessions are more secure.
Cookies save information to a text file.	Session saves data in encrypted form.

Express Session options

```
const oneDay = 1000 * 60 * 60 * 24;
app.use(sessions({
    secret: "thisismysecrctekeyfhrgfgrfrty84fwir767",
    saveUninitialized:true,
    cookie: { maxAge: oneDay },
    resave: false
}));
```

- `secret` - a random unique string key used to authenticate a session. It is stored in an environment variable and can't be exposed to the public. The key is usually long and randomly generated in a production environment.
- `saveUninitialized` - this allows any `uninitialized` session to be sent to the store. When a session is created but not modified, it is referred to as `uninitialized`.
- `cookie: { maxAge: oneDay }` - this sets the cookie expiry time. The browser will delete the cookie after the set duration elapses. The cookie will not be attached to any of the requests in the future. In this case, we've set the `maxAge` to a single day as computed by the following arithmetic.
- `resave` - takes a Boolean value. It enables the session to be stored back to the session store, even if the session was never modified during the request. This can result in a race situation in case a client makes two parallel requests to the server. Thus modification made on the session of the first request may be overwritten when the second request ends. The default value is `true`. However, this may change at some point. `false` is a better alternative.

Session Store

- Every express session store needs to implement certain methods and be an EventEmitter.
- This module will always call the required methods on the store.
- If applicable, this module will call for recommended methods for the store.
- Optional methods are those that this module does not use at all but that aid in the user presentation of consistent stores.

- **store.all(callback) :** optional

  All express sessions in the store can be obtained as an array using this optional function. The callback should be used to refer to as callback(error, sessions).

- **store.destroy(sid, callback) :** required

  With a session ID, this necessary function is used to remove/destroy a session from the storage (sid). Once the session is terminated, the callback should be invoked as a callback(error).

- **store.clear(callback) :** optional

  All express sessions in the store can be deleted using this optional technique. Once the store has been cleaned, the callback should be invoked as callback(error).

- **store.length(callback) :** optional

  The number of all sessions in the shop can be obtained using this callback. The callback should be invoked as callback(error, len).

- **store.get(sid, callback) :** required

  When a session ID is provided, this necessary method is used to retrieve a session from the store (sid). The callback should be invoked as a callback(error, session).

- **store.set(sid, session, callback) :** required

  With an express session ID (sid) and session (session) object, this necessary function is used to upsert a session into the store. Once the session has been set up in the store, the callback should be invoked as callback(error).

Installation

Step 1: Initialize the project using the following command in terminal

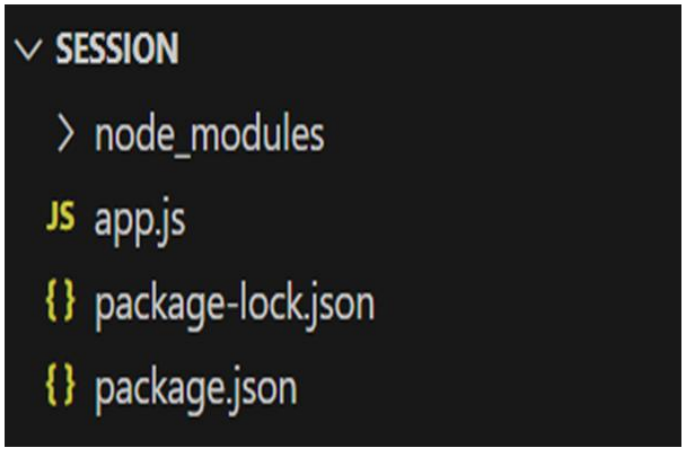
```
npm init
```

Step 2: Install the following required modules using the terminal.

```
npm install express express-session cookie-parser
```

Step 3: Create an **app.js** file as given below.

Project Structure Image:

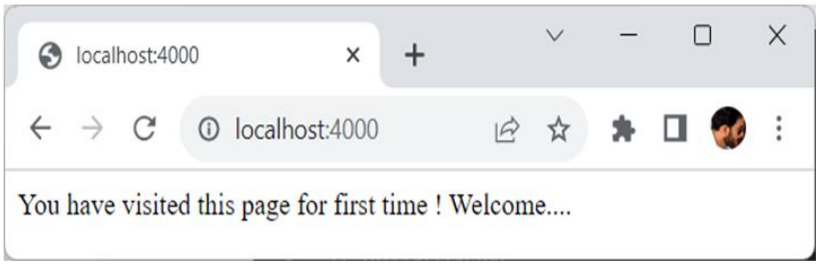


Example

```
JS app.js > ...
1   const express = require("express");
2   const session = require("express-session");
3   const cookieParser = require("cookie-parser");
4   const PORT = 4000;
5
6   const app = express();
7
8   // Initialization
9   app.use(cookieParser());
10
11  app.use(session({
12    secret: "VJNair",
13    saveUninitialized: true,
14    resave: true
15  }));
16
17  app.get('/', (req, res) => {
18    if (req.session.view) {
19
20      // The next time when user visits, // he is recognized by the cookie
21      // and variable gets updated.
22      req.session.view++;
23      res.send("You visited this page for "
24        + req.session.view + " times");
25    }
26    else {
27
28      // If user visits the site for first time
29      req.session.view = 1;
30      res.send("You have visited this page"
31      + " for first time ! Welcome....");
32    }
33  })
34
35  // Host
36  app.listen(PORT, () =>
37    console.log(`Server running at ${PORT}`));
```

Output:

```
D:\node_example\Express\Session>node app.js
Server running at 4000
```



The number of times you visit the **same page**, the number of times counter will increase.

Integrating with React

Overview

- The MERN (MongoDB, Express, React, NodeJS) stack, which uses Javascript for both the backend and frontend and a document-oriented or non-relational database (MongoDB), meaning it's structured like JSON rather than a big Excel sheet like SQL databases, is very popular for creating full stack applications.
- To create a MERN stack application, it is necessary to link our backend i.e., express with our frontend reacts, and create an express react linkage.
- In the past few years, the combination of Express.js and React.js has proven to be a powerful tool in the software developer's tool belt.
- With these two frameworks, front-end engineers can quickly create React apps on the front-end, quick communicate with a back-end through their own API.

What is Frontend?

- In web development, the term "frontend" refers to the client-side portion of a web application or website that users interact with directly.
- The front end is responsible for the visual and interactive components of the website, including the layout, design, and user interface elements.
- In other words, the frontend is part of the web application that users see and interact with within their web browser.
- It includes HTML, CSS, and JavaScript, which are used to create and style the webpage and add interactive elements such as buttons, forms, and animations.

What is Backend?

- The backend is responsible for handling tasks such as:
  - ✓ Authentication and authorization: verifying user credentials and determining access levels.
  - ✓ Data processing and storage: processing and storing data received from the frontend and serving it to the frontend when requested.
  - ✓ Server-side scripting: running server-side scripts to perform dynamic tasks such as generating web pages and interacting with databases.
  - ✓ API development: creating and managing APIs that allow frontend and backend applications to communicate with each other.

Prerequisites

- Node.js must be installed on your computer. The usage of Node Version Manager (nvm) is the simplest method for installing Node.js. You can install, update, and switch between several Node.js versions with this bash script.
- The following command can be used to install the most recent version of Node.js after nvm has been installed:

```
nvm install node
```

By using the following command, you may determine the Node.js version:

```
node -v
```

Installing Node.js will also install npm (Node Package Manager) on your system. You can check the version of npm by running the following command:

```
npm -v
```

Project Setup

1. Building the Frontend React Application

- The backend code will be placed in the server folder and the frontend code in the client folder.
- To build the front-end application, we'll utilize create-react-app.

```
D:\node_example\Express>mkdir react+xpress

D:\node_example\Express>cd "react+xpress"

D:\node_example\Express\react+xpress>npx create-react-app client
```

```
Success! Created client at D:\node_example\Express\react+xpress\client
Inside that directory, you can run several commands:

  npm start
    Starts the development server.

  npm run build
    Bundles the app into static files for production.

  npm test
    Starts the test runner.

  npm run eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can't go back!

We suggest that you begin by typing:

  cd client
  npm start

Happy hacking!

D:\node_example\Express\react+xpress>
```

Project Setup

2. Building the Backend Node.js server

Installing the dependencies

Open a new terminal and start a Node.js project in the server directory you just made.

```
mkdir server
cd server
npm init -y
```

Note: We are using npm init -y to initialize the project without any prompts. If you want to initialize the project with prompts, you can use npm init.

Installing the application's necessary dependencies is the next step.

```
npm install express cors
```

We will set up nodemon as a development dependency for our needs.

Project Setup

2. Building the Backend Node.js server

Install server dependencies

To start the server, we will lastly add a start script to the package.json file.

```
"scripts": {
    "start": "node server.js",
    "dev": "nodemon server.js"
}
```

At this stage, the dev and start scripts have been added to the package.json file, and all of the dependencies have been installed. Let's start the server creation process right away.

```
D:\node_example\Express>cd "react+xpress"

D:\node_example\Express\react+xpress>mkdir server

D:\node_example\Express\react+xpress>cd server

D:\node_example\Express\react+xpress\server>npm init -y
Wrote to D:\node_example\Express\react+xpress\server\package.json:

{
  "name": "server",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

1

```
D:\node_example\Express\react+xpress\server>npm install express cors

added 60 packages, and audited 61 packages in 4s

8 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

2

```
D:\node_example\Express\react+xpress\server>npm install nodemon --save-dev

added 33 packages, and audited 94 packages in 2s

11 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

3

```
D:\node_example\Express\react+xpress\server>npm install express --save

up to date, audited 94 packages in 1s

11 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

4

Project Setup

2. Building the Backend Node.js server

Creating the server - Start by adding a new server.js file to the server folder.

1. Import express and cors

```
const express = require('express');
const cors = require('cors');
```

Here, we're importing the express and cors modules using the built-in Node.js function need(), which allows you to include other modules in your application.

2. Create an Express application

```
const app = express();

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

Create an Express application by using the top-level express() function, which is exported by the express module.

To support cross-origin requests, we use cors. The Cors middleware is added to the Express application using app.use(). Express's built-in middleware method express.json() is used to parse incoming requests with JSON payloads.

2. Building the Backend Node.js server

Creating the server - Start by adding a new server.js file to the server folder.

3. Create a GET route

```
app.get('/message', (req, res) => {
    res.json({ message: "Hello from server!" });
});
```

Now, we'll build a JSON endpoint with the message Hello from server! that returns messages. To construct a GET route, we are using app.get(). It needs two justifications:

- The endpoint's path, which in this case is simply /message.
- The callback, which may be a single middleware function or an assortment/series of several.

We are using res.json() to send a JSON response because we are returning a JSON object.

4. Start the server

```
app.listen(8000, () => {
  console.log(`Server is running on port 8000.`);
});
```

We will use app.listen(), which accepts two arguments, to launch the server.

- The port that the server will run on. It is 8000 in our situation.
- When the server starts, a callback function is invoked, and in our example, it just logs the server's running message.

2. Building the Backend Node.js server

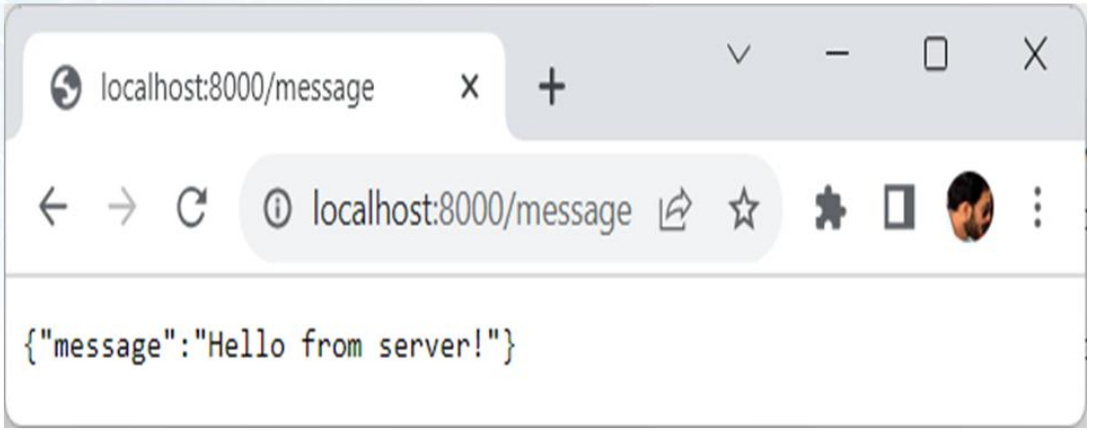
Creating the server - The finished server.js file will appear:

D: > node_example > Express > react+xpress > server > JS server.js > ...

```
1   const express = require("express");
2   const cors = require("cors");
3   const app = express();
4
5   app.use(cors());
6   app.use(express.json());
7
8   app.get("/message", (req, res) => {
9     res.json({ message: "Hello from server!" });
10  });
11
12  app.listen(8000, () => {
13    console.log(`Server is running on port 8000.`);
14  });
```

2. Building the Backend Node.js server

```
D:\node_example\Express\react+xpress\server>npm run dev

> server@1.0.0 dev
> nodemon server.js

[nodemon] 3.0.1
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node server.js`
Server is running on port 8000.
```



Project Setup

3. Connect React with Express.js / Node.js

In our express react amalgamation, replace the existing code by opening the App.js file in the src folder and adding the following.

D: > node_example > Express > react+xpress > client > src > JS App.js > ...

```
1   import React, { useState, useEffect } from "react";
2   import "./App.css";
3
4   function App() {
5     const [message, setMessage] = useState("");
6
7     useEffect(() => {
8       fetch("http://localhost:8000/message")
9         .then((res) => res.json())
10        .then((data) => setMessage(data.message));
11    }, []);
12
13    return (
14      <div className="App">
15        <h1>{message}</h1>
16      </div>
17    );
18  }
19
20  export default App
```

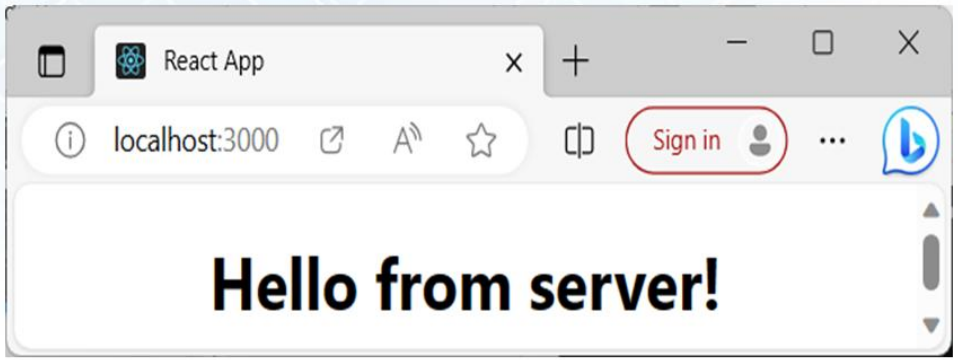
```
D:\node_example\Express\react+xpress\server>npm run dev

> server@1.0.0 dev
> nodemon server.js

[nodemon] 3.0.1
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node server.js`
Server is running on port 8000.
```

1

```
D:\node_example\Express\react+xpress\client>npm start

> client@0.1.0 start
> react-scripts start

(node:6056) [DEP_WEBPACK_DEV_SERVER_ON_AFTER_SETUP_MIDDLEWARE] Deprecatio
ware' option is deprecated. Please use the 'setupMiddlewares' option.
(Use `node --trace-deprecation ...` to show where the warning was create
(node:6056) [DEP_WEBPACK_DEV_SERVER_ON_BEFORE_SETUP_MIDDLEWARE] Deprecati
leware' option is deprecated. Please use the 'setupMiddlewares' option.
Starting the development server...
Compiled successfully!

You can now view client in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://192.168.1.37:3000

Note that the development build is not optimized.
To create a production build, use npm run build.

webpack compiled successfully
```

2

Output:



The final react app connected with the backend


Thank You!