AMRITA VISHWA VIDYAPEETHAM

15CSE337
Cloud Computing and Services

Express.JS

Dr Ganesh Neelakanta Iyer
Associate Professor, Dept of Computer Science and Engg
Amrita Vishwa Vidyapeetham, Coimbatore

Recap

- What is Node.js?
- Advantages?

https://expressjs.com/

Express.JS

- Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile 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

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

QUICK PREVIEW OF MEANSTACK

MEAN STACK

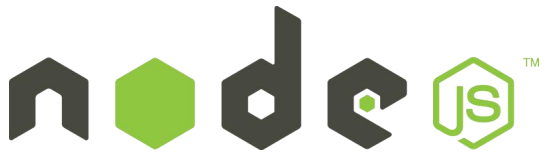
- The Friendly & Fun Fullstack JavaScript framework

MongoDB is the leading NoSQL database, empowering businesses to be more agile and scalable

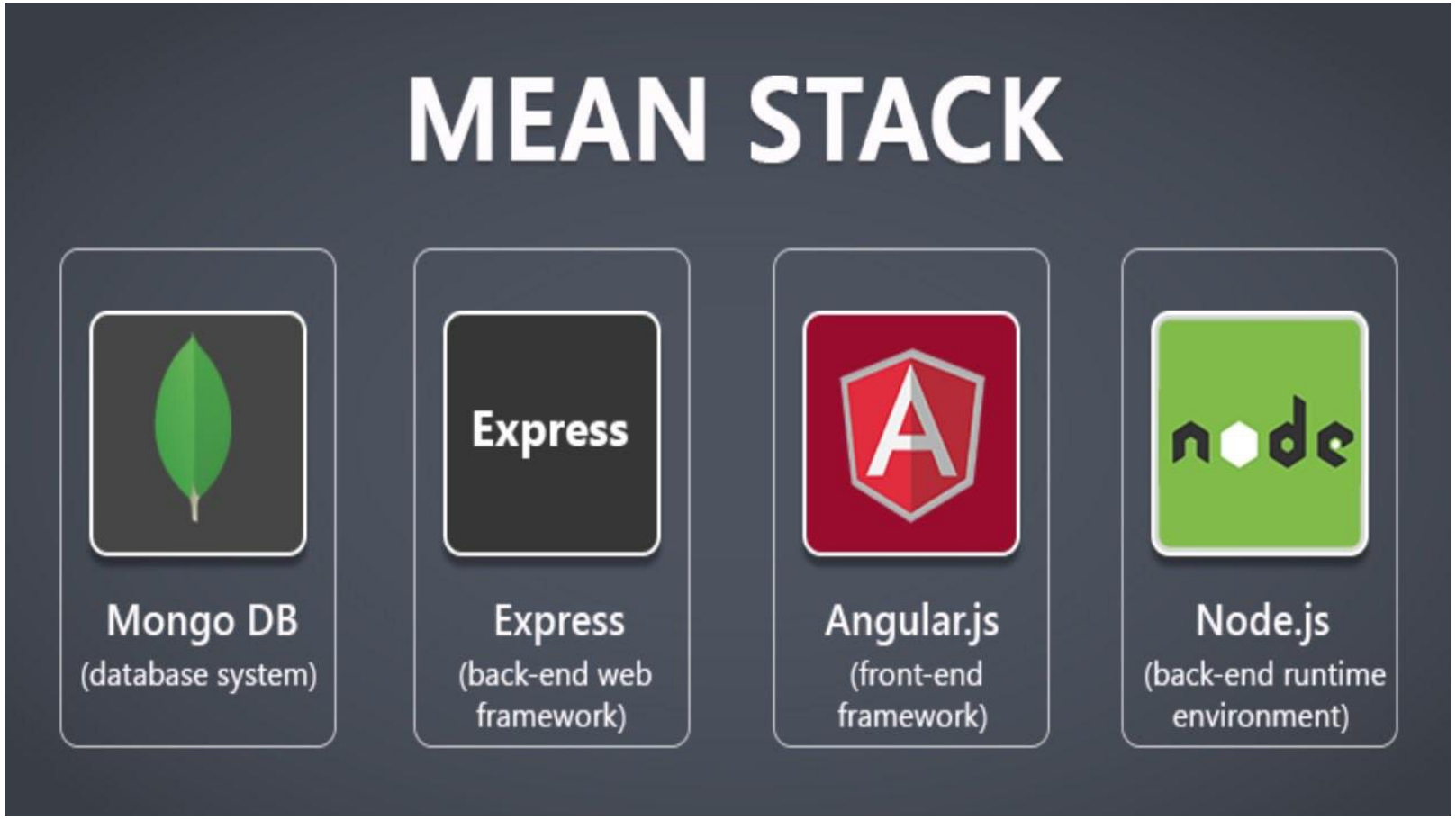
Express is a minimal and flexible node.js web application framework, providing a robust set of features for building single and multi-page, and hybrid web applications

AngularJS lets you extend HTML vocabulary for your application. The resulting environment is extraordinarily expressive, readable, and quick to develop.

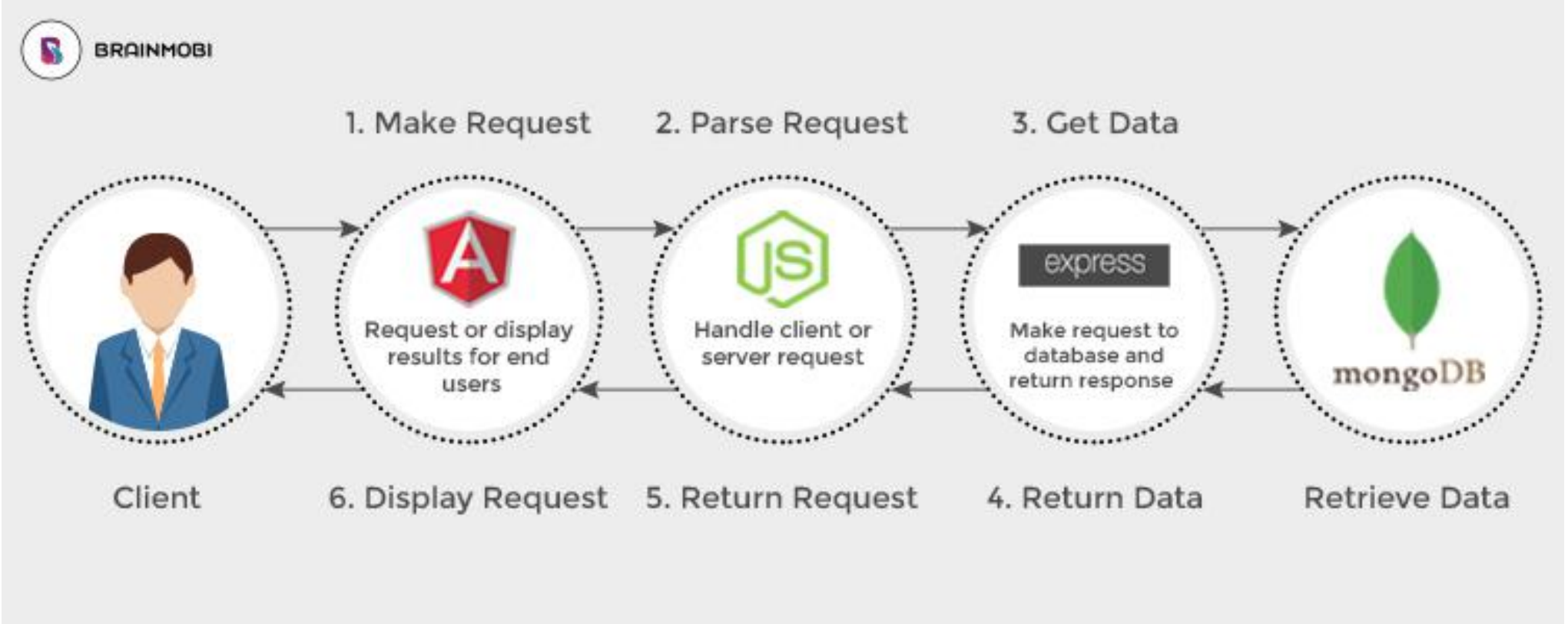
Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications



MEAN STACK


MEAN STACK
Mongo DB
(database system)
Express
Express
(back-end web framework)
Angular.js
(front-end framework)
node
Node.js
(back-end runtime environment)

MEAN STACK – HOW IT WORKS



BACK TO EXPRESS HELLO WORLD

Getting Started

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

- Now to create the package.json file using npm, use the following code. – `npm init`
- Just keep pressing enter, and enter your name at the “author name” field.

Getting Started

- 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)`

Getting Started

- 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`

Hello World

- 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.

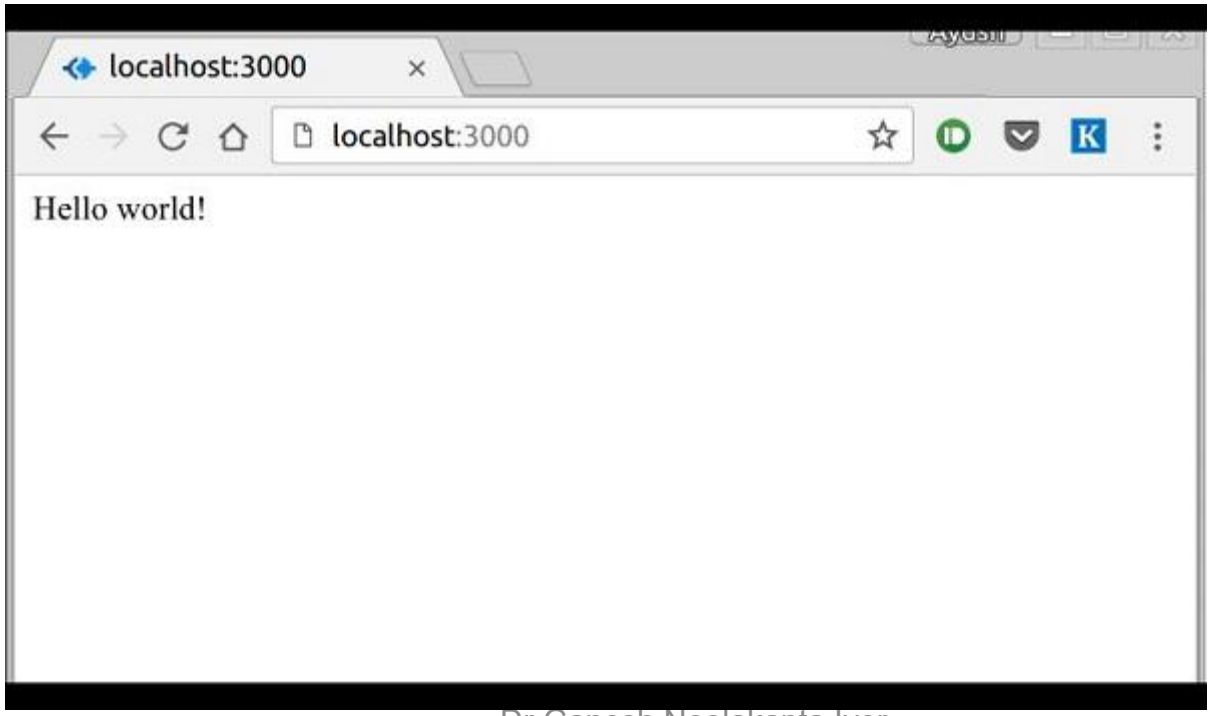
```
var express = require('express');
var app = express();

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
```

Hello World

- 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 arameters, ***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

How the App works?

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]])

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

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.

ExpressJS Routing

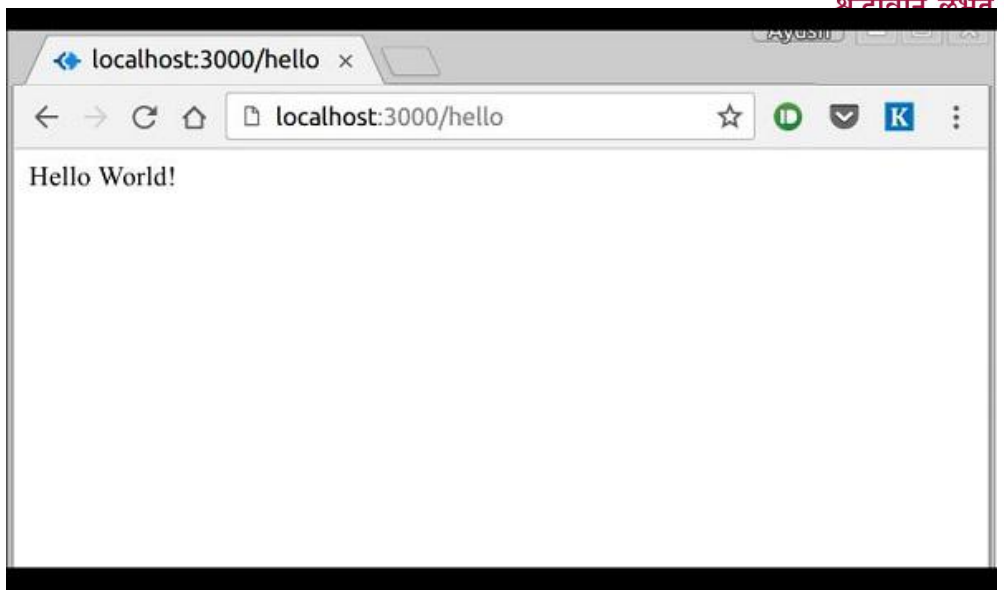
- ***Routing*** refers to the definition of application end points (URIs) and how they respond to client request

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

ExpressJS Routing

```
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

ExpressJS Routing

- We can also have multiple different methods at the same route. For example,
- To test this request, open up your terminal and use cURL to execute the following request −
- curl -X POST "http://localhost:3000/hello"
- 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

```
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);
```

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

Express Routers

- 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;
```

Express Routers

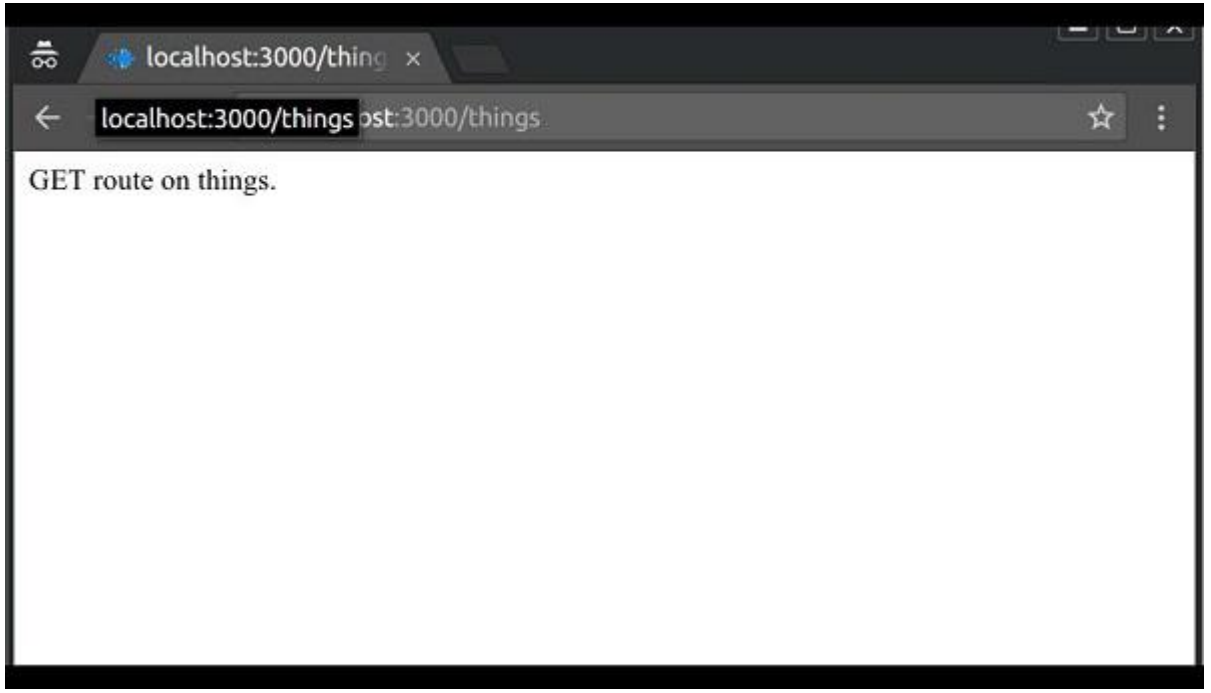
- Now to use this router in our **index.js**, type in the following before the **app.listen** function call.

```
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.

Express Routers

- Routers are very helpful in separating concerns and keep relevant portions of our code together. They help in building maintainable code. You should define your routes relating to an entity in a single file and include it using the above method in your **index.js** file.



ExpressJS - URL Building

- We can now define routes, but those are static or fixed. To use the dynamic routes, we SHOULD provide different types of routes. Using dynamic routes allows us to pass parameters and process based on them.
- Here is an example of a dynamic route −

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

app.get('/:id', function(req, res){
   res.send('The id you specified is ' + req.params.id);
});
app.listen(3000);
```

ExpressJS - URL Building

- To test this go to **http://localhost:3000/123**. The following response will be displayed.

ExpressJS Pattern Matched Routes

- You can also use **regex** to restrict URL parameter matching
- Let us assume you need the **id** to be a 5-digit long number. You can use the following route definition

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

app.get('/things/:id([0-9]{5})', function(req, res){
   res.send('id: ' + req.params.id);
});

app.listen(3000);
```

ExpressJS Pattern Matched Routes

- Note that this will **only** match the requests that have a 5-digit long **id**. You can use more complex regexes to match/validate your routes. If none of your routes match the request, you'll get a ***"Cannot GET <your-request-route>"*** message as response. This message be replaced by a 404 not found page using this simple route

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

//Other routes here
app.get('*', function(req, res){
   res.send('Sorry, this is an invalid URL.');
});
app.listen(3000);
```

ExpressJS Middleware

- 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
- These functions are used to modify **req**and **res** objects for tasks like parsing request bodies, adding response headers, etc

Here is a simple example of a middleware function in action

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

//Simple request time logger
app.use(function(req, res, next){
   console.log("A new request received at " + Date.now());

   //This function call is very important. It tells that more processing is
   //required for the current request and is in the next middleware
   function/route handler.
   next();
});

app.listen(3000);
```

- The above middleware is called for every request on the server. So after every request, we will get the following message in the console

```
A new request received at 1467267512545
```

To restrict it to a specific route (and all its subroutes), provide that route as the first argument of ***app.use()***. For Example

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

//Middleware function to log request protocol
app.use('/things', function(req, res, next){
   console.log("A request for things received at " + Date.now());
   next();
});

// Route handler that sends the response
app.get('/things', function(req, res){
   res.send('Things');
});

app.listen(3000);
```

- Now whenever you request any subroute of '/things', only then it will log the time

Order of Middleware Calls

- One of the most important things about middleware in Express is the order in which they are written/included in your file
- the order in which they are executed, given that the route matches also needs to be considered

Order of Middleware Calls

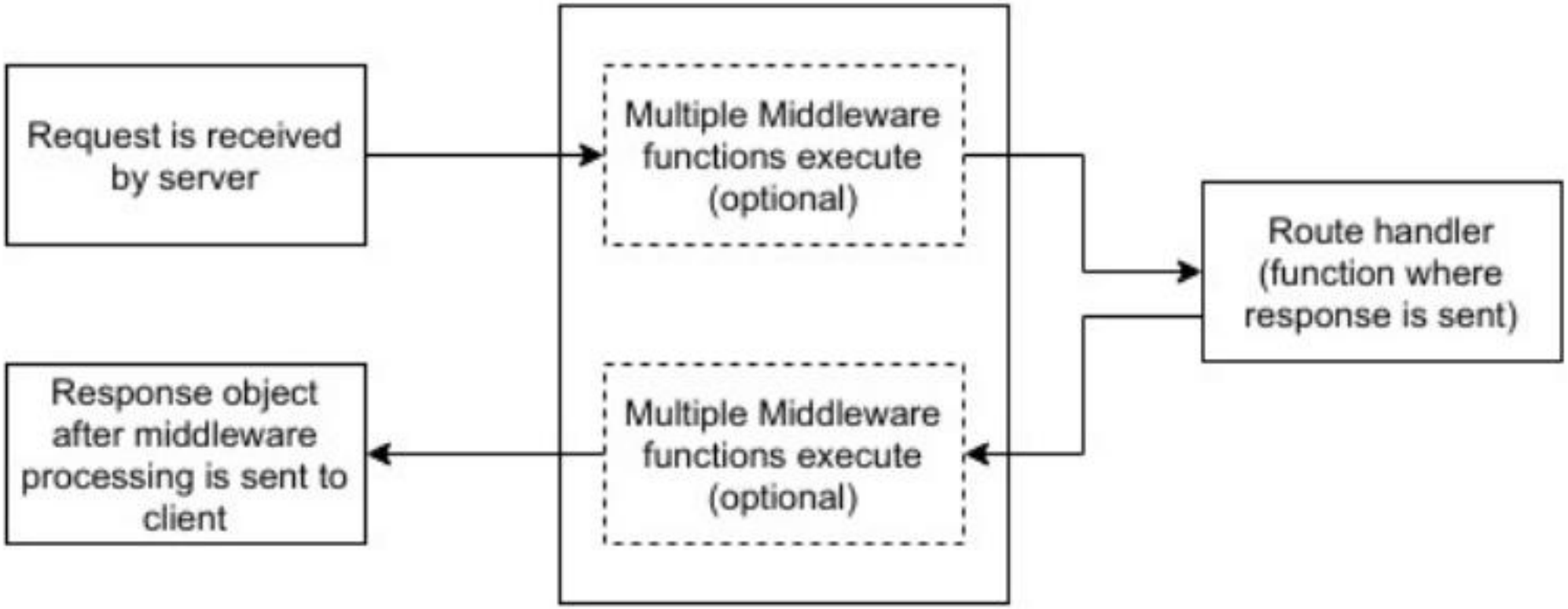
- For example, here the first function executes first, then the route handler and then the end function. This example summarizes how to use middleware before and after route handler; also how a route handler can be used as a middleware itself.

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

//First middleware before response is sent
app.use(function(req, res, next){
   console.log("Start");
   next();
});

//Route handler
app.get('/', function(req, res, next){
   res.send("Middle");
   next();
});

app.use('/', function(req, res){
   console.log('End');
});

app.listen(3000);
```

When we visit '/' after running this code, we receive the response as **Middle**and on our console

```
Start

End
```

Middleware Summary



ExpressJS Templating Engine

- 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.

ExpressJS Templating Engine

- 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');
```

ExpressJS Templating Engine

- 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');
});
```

ExpressJS Templating Engine

- 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>
```

Features of PUG – Self-study:

https://www.tutorialspoint.com/expressjs/expressjs_templating.htm

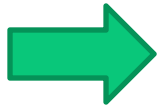
ExpressJS Static Files

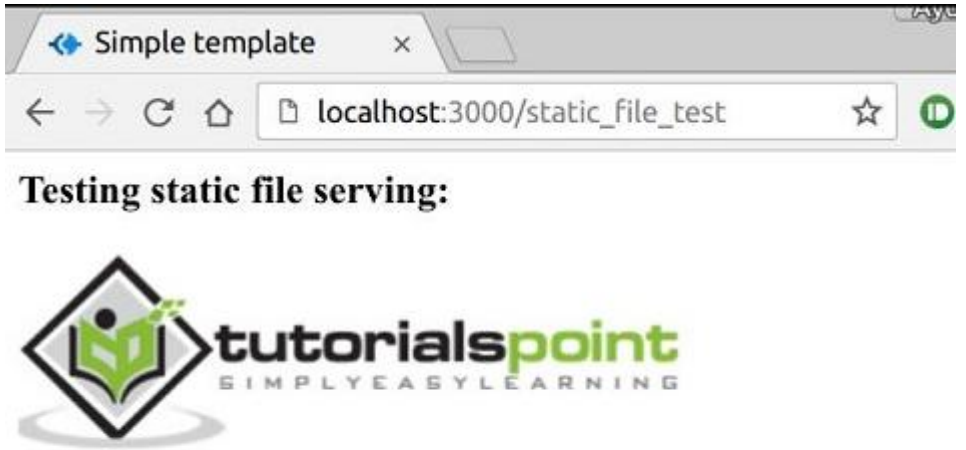
- Static files are files that clients download as they are from the server. Create a new directory, **public**
- Express, by default does not allow you to serve static files
- You need to enable it using the following built-in middleware
- `app.use(express.static('public'));`
- Note that the root route is now set to your public dir, so all static files you load will be considering public as root.

ExpressJS Static Files

- 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
```





ExpressJS Static Files

- 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);
```

ExpressJS Static Files

- 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.

ExpressJS Form data

- Forms are an integral part of the web.
- To get started with forms, we will first install the *body-parser*(for parsing JSON and url-encoded data) and multer(for parsing multipart/form data) middleware
- To install the *body-parser* and *multer*, go to your terminal and use −
- npm install --save body-parser multer

https://www.tutorialspoint.com/expressjs/expressjs_form_data.htm

ExpressJS Form data

- Replace your **index.js** file contents with the following code

https://www.tutorialspoint.com/expressjs/expressjs_form_data.htm

```
var express = require('express');
var bodyParser = require('body-parser');
var multer = require('multer');
var upload = multer();
var app = express();

app.get('/', function(req, res){
   res.render('form');
});

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

// for parsing application/json
app.use(bodyParser.json());

// for parsing application/xwww-
app.use(bodyParser.urlencoded({ extended: true }));
//form-urlencoded

// for parsing multipart/form-data
app.use(upload.array());
app.use(express.static('public'));

app.post('/', function(req, res){
   console.log(req.body);
   res.send("recieved your request!");
});
app.listen(3000);
```

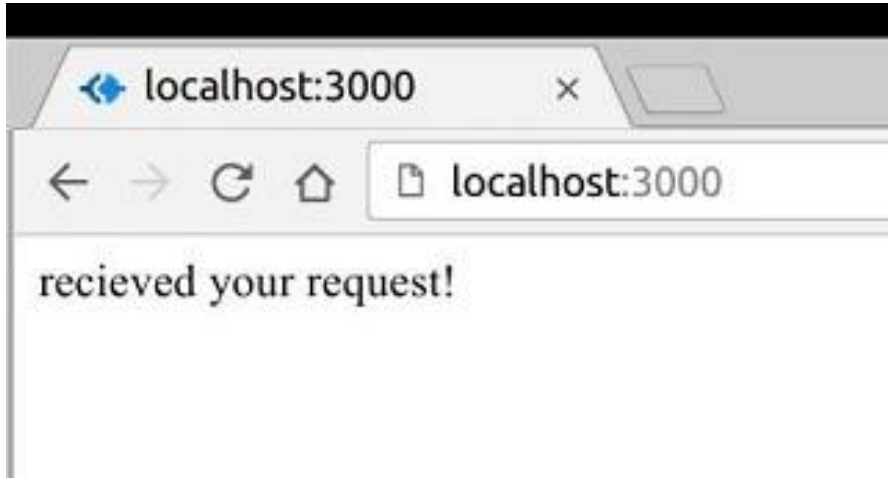
ExpressJS Form data

- After importing the body parser and multer, we will use the **body-parser** for parsing json and x-www-form-urlencoded header requests, while we will use **multer** for parsing multipart/form-data.
- Let us create an html form to test this out. Create a new view called **form.pug** with the following code −

```
html
html
   head
      title Form Tester
   body
      form(action = "/", method = "POST")
         div
            label(for = "say") Say:
            input(name = "say" value = "Hi")
         br
         div
            label(for = "to") To:
            input(name = "to" value = "Express forms")
         br
         button(type = "submit") Send my greetings
```

ExpressJS Form data

- Run your server using the following.
- `nodemon index.js`
- Now go to localhost:3000/ and fill the form as you like, and submit it. The following response will be displayed



ExpressJS Form data

- Have a look at your console; it will show you the body of your request as a JavaScript object as in the following screenshot

- The **req.body** object contains your parsed request body. To use fields from that object, just use them like normal JS objects.
- This is the most recommended way to send a request.

ExpressJS Cookies

- Cookies are simple, small files/data that are sent to client with a server request and stored on the client side
- Every time the user loads the website back, this cookie is sent with the request.
- This helps us keep track of the user's actions.
- The following are the numerous uses of the HTTP Cookies −
  - Session management
  - Personalization(Recommendation systems)
  - User tracking
- To use cookies with Express, we need the cookie-parser middleware. To install it, use the following code −
- `npm install --save cookie-parser`

ExpressJS Cookies

- Now to use cookies with Express, we will require the **cookie-parser**.
- cookie-parser is a middleware which *parses cookies attached to the client request object*
- To use it, we will require it in our **index.js** file; this can be used the same way as we use other middleware
- Here, we will use the following code.

```
var cookieParser = require('cookie-parser');
app.use(cookieParser());
```

ExpressJS Cookies

- cookie-parser parses Cookie header and populates **req.cookies** with an object keyed by the cookie names. To set a new cookie, let us define a new route in your Express app like −

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

app.get('/', function(req, res){
   res.cookie('name', 'express').send('cookie set'); //Sets name = express
});

app.listen(3000);
```

ExpressJS Cookies

- To check if your cookie is set or not, just go to your browser, fire up the console, and enter −
- `console.log(document.cookie);`
- You will get the output like (you may have more cookies set maybe due to extensions in your browser) −
- `"name = express"`
- The browser also sends back cookies every time it queries the server. To view cookies from your server, on the server console in a route, add the following code to that route
- `console.log('Cookies: ', req.cookies);`
- Next time you send a request to this route, you will receive the following output.
- `Cookies: { name: 'express' }`

Self-study

- Adding cookies with expiry date
- Deleting existing cookies
- https://www.tutorialspoint.com/expressjs/expressjs_cookies.htm

- ExpressJS Sessions
- https://www.tutorialspoint.com/expressjs/expressjs_sessions.htm

Thank you!

Dr Ganesh Neelakanta Iyer

ni_amrita@cb.amrita.edu
ganesh.vigneswara@gmail.com

React and Node/Express - Samples

- https://www.freecodecamp.org/news/create-a-react-frontend-a-node-express-backend-and-connect-them-together-c5798926047c/

- https://blog.cloudboost.io/learn-how-to-create-a-simple-blog-with-react-node-c05fa6889de3