UNIT 4 - BACK END TECHNOLOGIES

NODE JS

Introduction to Node.js

- Node.js is an open-source, cross platform JS runtime environment that enables developers to execute JS code outside of a browser, typically on the server side.
- Widely used for building scalable, high-performance applications, especially web servers and API's.
- Built on Google's V8 engine

Features :-

1) Open-source and Cross platform.
2) Asynchronous & event-driven Architecture.
3) Modern JS support.
3) Non-blocking I/O.
4) Rich set of Built-in Modules.
5) Same language for Backend & Frontend

Node.js Events

- Node.js applications are event-driven.
- The core 'events' module allows you to create, fire and listen for custom events.
- An event handler is nothing but a callback function called when an event is triggered.

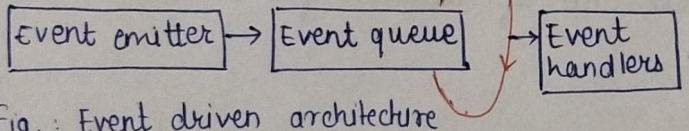
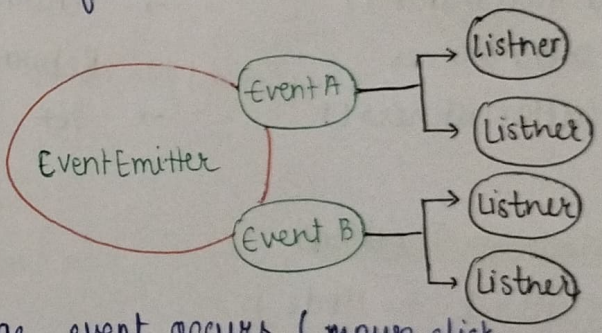


Fig.: Event driven architecture in Node.js.



- Every time when user interacts with webpage, event occurs (mouse click, key pressed). When events are triggered, some functions associated with these events get executed.

i) When some events occur, an event handler function is called. (call back function)
ii) The main program listens every event getting triggered and calls the associated event handler for that event.

CONCEPT OF EVENT-EMITTER

- It is a core class provided by 'events' module that enables objects to emit named events, that can be listened by other parts of application.
- EventEmitter allows to:

-) Emit (trigger) named events.
-) Attach one/more listners (callback functions) to those events.
-) Handle events synchronously as they occur.

eg:

```
const EventEmitter = require ('events'); // Import 'events' module
const eventEmitter = new EventEmitter (); // Create instance of EventEmitter
// Create event listner
eventEmitter.on ("OrderReceived", (orderId) => {
  console.log (" Order received ! Order ID : ${orderId}");
  // save to db.
});
// Trigger (Emit) the event
eventEmitter.emit ('orderReceived', 1234);
```

my event name

output - Order received ! Order ID : 1234.

Methods in Event Listners

on (event, listner)	→ Adds a listner for specified event.
emit (event, [...])	→ Emits specified event, optionally passing arguments to listners.
once (event, listner)	→ Adds one-time listner that is removed after first execution.
removeListners ()	→ Removes a specific listner for an event.
removeAllListners ()	→ Removes all listners for an event or all events.
listnerCounts ()	→ Returns no. of listeners.
eventNames	→ Returns array of event names.
setMaxListeners ()	→ Set max. no. of listeners for an event.

Node.js Functions

Callback in Node.js

- A callback is a function passed as an argument to another function.
- In Node.js callbacks are asynchronous operations.

Working :

- The main function starts asynchronous operation (like reading a file)
- Once the operation finishes, Node.js invokes callback function.
- This allows the program to continue executing other code while waiting for the operation to complete.

- eg:

```
const fs = require ('fs');
fs.readFile ('example.txt', 'utf8', (err, data) => {
  if (err) {
    console.error ('Error reading file:', err);
    return;
  }
  console.log ('File content :', data);
});
```

Async reading file with callback

the callback : (err, data) => {...} is called after read operation is completed.

Built-in Modules.

fs → file system operations.
http → create HTTP servers & clients.
path → includes methods to deal with file paths.
os → info. about OS.
events → event driven programming via EventEmitter
url → URL parsing & formatting.
buffer → Handling binary data
vm → run code in virtual machine.

File System

Node.js provides a built-in module called fs (File System) that allows you to interact with the file system: read, write, update, delete and manage files and directories.

Importing fs : const fs = require ('fs')

The File System in Node.js enables a simple API with which we can work with files, in both synchronous and asynchronous way.

1) Read a file

Async

```
fs.readFile ('example.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log (data);
});
```

Sync

```
const data = fs.readFileSync ('example.txt',
                              'utf8');
console.log (data);
```

2) Write a file

```
fs.writeFile ('example.txt', 'Hello, Node.js!', (err) => {
  if (err) throw err;
  Console.log ("File written successfully!");
});
```

3) Append to a file

```
fs.appendFile ('example.txt', '\n Appended text', (err) => {
  if (err) throw err;
  Console.log ('content append');
});
```

4) Rename a file

```
fs.rename ('example.txt', 'newname.txt', (err) => {
  if (err) throw err;
  console.log ('File renamed!');
});
```

5) Delete a file

```
fs.unlink ('newfilename.txt', (err) => {
if (err) throw err;
console.log ('File deleted');
});
```

NPM (Node Package Manager)

- npm is a default package manager for Node.js, used to install, manage, and share Javascript packages or modules.

It is:

→ A command line tool for installing and managing Node.js packages.

→ A repository of open source packages (modules/libraries).

- npm plays a central role in Node.js development as default package manager.

Functions:

1) Dependency Management - npm helps install, update and remove packages required for Node.js project. By running npm install, all the dependencies listed in project's package.json file are downloaded and made available, ensuring consistency.

2) Vast Package Repository - npm provides access to large collection of open-source packages, making it easy to add new features.

3) Project Organization - npm keeps track of all project dependencies in a file called package.json.

4) Task Automation - npm allows running scripts for tasks like building, testing, and starting the project.

5) Version Control - It manages package versions to avoid compatibility issues.

6) Easy Sharing - Developers can share their own packages with others using npm.

eg. npm install mongodb
npm install express

Handling Data I/O in Node.js

usually handled via:

1) files (fs module) —— (already done)
2) Streams
3) HTTP request and response
4) Console (stdin/stdout)

↳ eg:

READ
```
process.stdin.on('data', (data)=> {
console.log(`${data.toString().trim()}`);
});
```

WRITE
```
process.stdout.write("Enter msg");
```

Create HTTP Server

HTTP server ? : listens for request & responds with data like JSON, HTML or plain text.

In Node.js, HTTP server is built using built-in module http module - no installation needed.

eg.
```
const http = require ('http');
const server = http.createServer ((req, res) => {
res.writeHead (200, {'Content-Type' : 'text/plain'});
res.end ("Hello, World!");
});
server.listen (3000, () => {
console.log ("server is running");
});
```

→ save this file as server.js
→ run it with node server.js

Create Socket Server.

Socket Server ? : allows real time 2-way communication between client and the server using Web socket.

eg. npm install socket.io

```
// create HTTP server and install required
const http = require ('http');
const socketio = require ('socket.io');
// HTTP server creation.
const server = http.createServer();
const io = socketio (server);

~~Socket on~~ io.on ('connection', (socket) => {
console.log ('New user connected');

// Listen for message from client
socket.on ('message', (msg) => {
console.log ('Received :', msg);
// ack
socket.emit ('message', 'server received : ${msg}');
});
// Handle disconnection
socket.on ('disconnect', () => {
console.log ('User disconnected');
});
});
// start server
server.listen (3000, () => {
console.log ('Websocket Running');
});
```

PM2 - Microservices

Microservices ? : An architecture where your app is divided into small, independent services - each handling a specific task & communicating via API's

PM2 ? : It is a production process manager for Node.js apps.

It : 1) Runs multiple apps / services (Process management)
2) Restarts them automatically on crash. (Zero-downtime Reloads)
3) Supports logging, monitoring and clustering. (Centralized logging & Monitoring)
4) Makes it easy to manage microservices.
5) JSON configuration.

PM2 streamlines the deployment, scaling & monitoring of Node.js microservices, making it easier to manage complex distributed systems with minimal downtime and maximum visibility.

Install : npm install pm2 -g

Start all services :
```
pm2 start app.json
pm2 reload app.json
pm2 stop app.json
pm2 delete app.json
```

P49 : methods of console log in Node.js

1) console.log() → print normal output.
eg. console.log("Hello"); >> Hello

2) console.warn() → used to display warnings.
eg. console.warn("This is warning"); >> Outputs in yellow

3) console.error() → display error messages.
eg. console.error("something went wrong"); >> Outputs in red.

4) console.table() → displays tabular data in table format.
eg.
```
console.table([
  {name: "Alice", age:25},
  {name: "Bob", age:30}
]);
```
>>

(Index)	name	age
0	Alice	25
1	Bob	30

5) console.dir() → displays object with original depth & formatting.
eg. console.dir({name: "Node.js", type: "Javascriptruntime"}, {depth: 1});

6) console.assert() → writes msg only if assertion fails (ie condn false)
eg: console.assert(5>10, "Assertion failed"); >> Assertion failed.

EXPRESS JS

Introduction to ExpressJS

- It is minimal and flexible Node.JS web application framework that provides robust set of features for building web and mobile applications.
- It is a part of MEAN stack (MongoDB, Express.js, Angular, Node.js) and is widely used for building RESTful APIs and web applications.

Features:
- → Fast and lightweight
- → Middleware support
- → Routing
- → Template engines
- → RESTful APIs
- → Error Handling.

Configure Routes

- ExpressJS uses routing to determine how an applications responds to client request for a particular route, path or URL's and HTTP methods.
- Syntax: app.METHOD(PATH, HANDLER)
  - app: instance of express
  - METHOD: HTTP method (get, post, put, delete)
  - PATH: route to server for specific webpage.
  - HANDLER: callback function that is executed when route is matched.

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

app.get ('/', (req, res) => {
 res.send ('Wlcm to Home Page');
 });
app.get ('/about', (req, res) => {
 res.send ('About Us');
 });
app.post ('/contact', (req, res) => {
 res.post ('Contact Us');
});
app.listen (3000, () => console.log ('server running') };
```

Template Engines

This allows you to generate dynamic HTML pages by merging static templates with data from your applications. At runtime, variables within template are replaced with actual values, and resulting HTML is sent to the client.

Working : i) You create a template file containing placeholders for dynamic content.
eg. Welcome <%= username %>

ii) When route is accessed, Express renders the template, injecting data into the placeholders. eg. {username : "Rahul"}

iii) The final HTML is sent to the browser. eg. Welcome Rahul.

Popular Template Engines :

EJS (Embeded JS), Pug, Handlebars, Mustache, Nunjunks, Dust.js

Setting Up Template Engine :

i) Install engine via npm

```
npm install ejs
```

ii) Configure express to use engine

```
const express = require ('express');
const app = express();
app.set ('view engine', 'ejs');
```

iii) Create view file

index.ejs

iv) Render the template in a route

```
app.get ('/', (req, res) => {
res.render ('index', {name: 'Express User'
                      });
});
```

Output >> A rendered HTML page saying:

Hello, Express User

Express as Middleware

- Middleware functions are the core building blocks that process request & response in application's request - response cycle.
- Middleware functions are the functions that have access to the request object (req), the response object (res), and the next middleware function in the stack (next).
- They can : Execute any code.
  Modify the request & response object.
  End the request - response cycle.
  Call the next middleware function in the stack.
- Working : Middleware functions are executed in the order they are added to the Express app.

Each middleware can pass control to the next middleware using next() or end the cycle by sending response.
Middleware can be applied globally, to specific routes, or to specific routers.

eg :
```
const express = require ('express');
const app = express ();
const logger = function (req, res, next) {
console.log ('LOGGED');
next ();
};
app.use (Logger);
app.get ('/', (req, res) => {
   res.send ('Hello');
});
app.listen (3000);
```

Types of Middleware

1) Application level
2) Router level
3) Error - handling
4) Built-in
5) Third party - installed via npm

Serving Static files

- Express.js provides built-in middleware function express.static to serve static files such as images, CSS and JS to clients.
- Static files are directly sent to the client, ie don't change dynamically.

Basic static serving

app.use (express.static ('public')) ⇒ Result /hello.html → public/hello.html

REST HTTP Methods for APIs

- REST stands for Representational State Transfer. It is the set of rules that developers follow while creating API.
- HTTP method : POST GET PUT PATCH DELETE
  Curd operation : Create Read Update partial update delete
- eg.

```
const express = require ('express');
const app = express();
app.use(express.json());
let users = [{id:1, name:'Alice'}];

app.get('/users', (req, res) => res.json(users));

app.post('/users', (req, res) => {
   const user = {id: users.length+1, name: req.body.name};
   users.push(user);
   res.status(201).json(user);
});

app.put('/users/:id', (req, res) => {
 Const user = users.find(u => u.id == req.params.id);
 If (!user) return res.status(404).send('User not found');
 user.name = req.body.name;
 res.json(user);
});
app.delete('/users/:id', (req, res) => {
 users = users.filter(u => u.id != req.params.id);
 res.send('User deleted');
});
app.listen(3000, () => console.log("Server running"));
```

Applying Basic HTTP Authentication

To apply Basic authentication in express, use express-basic-auth middleware. This middleware checks incoming requests for HTTP Authorization header and prompts the browser for credentials if they are missed or incorrect.

If When a user accesses a protected route, the browser will prompt for a username and password.

- If the credentials are correct, the user can access the resource; otherwise they receive a 401 Unauthorized response.

MONGODB

NoSQL Basics (Not Only SQL) [Covered in DSBDA unit 3]

NoSQL db are designed to handle large volumes of unstructured or semi-structured data. NoSQL db use flexible data models to store data.
↳ [key value, document, column, graph]

Key features:

→ Schemaless / flexible schema
→ High scalable
→ Handles big data
→ High Performance
→ Flexible data model
→ Follows CAP theorem.

MongoDB Basics

Leading open source NoSQL db that stores data in a document-oriented format. Instead of tables MongoDB uses collections, and instead of rows it uses documents. Each document is a JSON like object.

Concepts:-

Db: A container for collections.
Collection: A group of MongoDB documents.
Document: Basic unit of data in MongoDB, stored as JSON-like obj with key-value pairs.
Field: A key-value pair in a document.

Features:-

→ Document Oriented
→ Flexible Schema
→ Replication
→ High Performance
→ High Scalability (Horizontal scaling using shreading)
→ Security
→ Cloud Support.
→ Integration & Tools

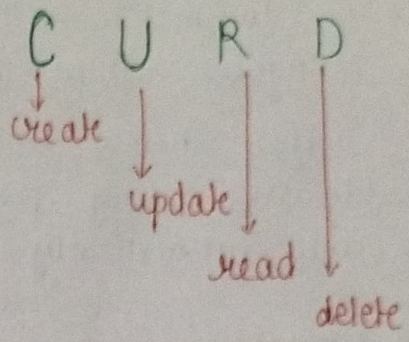
MongoDB - Node.js Communication

```
const {MongoClient} = require('mongodb');
const url = 'mongodb://localhost:27017'
const database = 'student';
const client = new MongoClient(url);

const dbConnect = async () => {
const result = await client.connect();
const db = result.db(database);
return db.collection('profile');
}
module.exports = dbConnect;
```

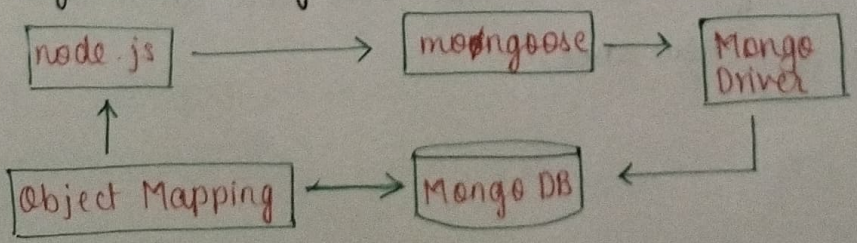
Node.js communicates with MongoDB using official MongoDB Node.js driver. This driver allows Node.js applications to connect to MongoDB database, perform CURD operations.

CURD operation using Node.js

```
const dbConnect = require ('./mongodb');
const express = require ('express');
const app = express();
app.use (express.json());
// Read or Get
app.get ('/getData', async (req, res) => {
  let result = await dbConnect ();
  result = await result.find().toArray();
  res.send (result);
});
// Create or Post
app.post ('/insertData', async (req, res) => {
  let result = await dbConnect ();
  result = await result.insertOne (req.body);
  res.send ("Data Inserted Successfully");
});
// Update or put
app.put ('/updateData/:name', async (req, res) => {
  let result = await dbConnect ();
  result = await result.updateOne ({name : req.params.name}, {$set : req.body})
  res.send ("Data Updated Successfully");
// Delete
app.delete ('/deleteData/:name', async (req, res) => {
let result = await dbConnect ();
result = await result.deleteOne ({name : req.params.name});
res.send ("Data Deleted Successfully");
});

app.listen (3000);
```



Mongoose ODM for Middleware



Mongoose is an Object Data Modeling (ODM) library for MongoDB and Node.js

Middleware (Hooks) are functions that run before or after certain Mongoose operations on documents or queries.
Mongoose methods like save, remove, updateOne, find, etc.
Working: Define schema using mongoose.Schema.
Attach middleware using .pre() or .post() on the schema.
Mongoose will automatically run the middleware when the operation is triggered.

Advanced MongoDB.

Features: Can Connect

1) Relationships: Data in different collection (tables) using two ways - Embedded & Referenced.

2) Covered Queries: It is a query in which all the fields used in the query are part of an index and are also included in the result.
ie MongoDB can answer the query using only index, without reading actual documents, making query much faster.

3) Measuring Indexes: It is imp to check how well indexes are helping queries. The explain() method in MongoDB shows how a query uses indexes & provides statics about query performance.

4) Map-Reduce: Used for complex data analysis.

5) Text Search Indexes: MongoDB provides text search indexes to search for specific words or phrases in string fields. This uses techniques like stemming & removing stop words to improve search results.

6) Rock Mongo: It is a web-based MongoDB administration tool. It provides user-friendly interface to manage db, collections, documents, and indexes.

7) GridFS: MongoDB specifications for storing & retrieving large files such as images, audio and video files.

8) Capped Collections: These are fixed-size, circular collections in MongoDB. When the allocated size is full, the oldest docs are automatically deleted to make space for new ones.