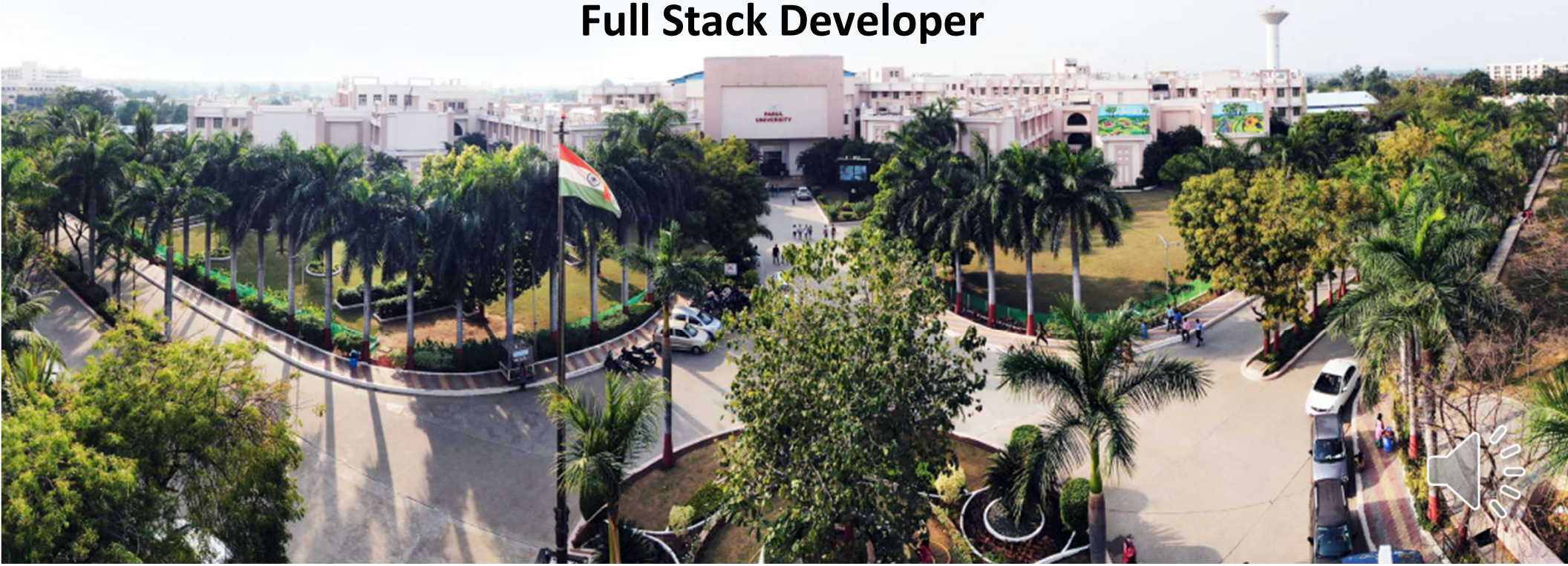
Parul® University

MEAN Full Stack

Amir Hussain
Cyber security trainer
&
Full Stack Developer



Parul® University

DIGITAL LEARNING CONTENT

CHAPTER-3rd

Introduction

Parul® University

Node.JS

Node JS is an open-source and cross-platform runtime environment built on Chrome's V8 JavaScript engine for executing JavaScript code outside of a browser. It provides an event-driven, non-blocking (asynchronous) I/O and cross-platform runtime environment for building highly scalable server-side applications using JavaScript.

Node.js also provides a rich library of various JavaScript modules which simplifies the development of web applications using Node.js to a great extent.

Node.js = Runtime Environment + JavaScript Library

Parul® University

Node.JS

Node.js is an open-source and cross-platform runtime environment for executing JavaScript code outside a browser.

NodeJS is not a framework and it's not a programming language.

Node.js is used to build back-end services like **APIs** like Web App or Mobile App.

It's used in production by large companies such as Paypal, Uber, Netflix, Walmart, and so on.

Parul® University

Why do we use Node.JS

There are many reasons for which we prefer using NodeJs for the server side of our application, some of them are discussed in the following:

- NodeJs is built on Google Chrome's V8 engine, and for this reason its execution time is very fast and it runs very quickly.
- There are more than 50,000 bundles available in the Node Package Manager and for that reason developers can import any of the packages any time according to their needed functionality for which a lot of time is saved.
- As NodeJs do not need to wait for an API to return data , so for building real time and data intensive web applications, it is very useful. It is totally asynchronous in nature that means it is totally non-blocking.

Parul® University

Why do we use Node.JS

- The loading time for an audio or video is reduced by NodeJs because there is better synchronization of the code between the client and server for having the same code base.

- As NodeJs is open-source and it is nothing but a JavaScript framework , so for the developers who are already used to JavaScript, for them starting developing their projects with NodeJs is very easy.

Parul® University

Node.JS

FEATURE OF NODE.JS

1. **Asynchronous and Event Driven** − All APIs of Node.js library are asynchronous, that is, non-blocking. It essentially means a Node.js based server never waits for an API to return data. The server moves to the next API after calling it and a notification mechanism of Events of Node.js helps the server to get a response from the previous API call

2. **Very Fast** − Being built on Google Chrome's V8 JavaScript Engine, Node.js library is very fast in code execution.

Node.JS

FEATURE OF NODE.JS

3. **Single Threaded but Highly Scalable** − Node.js uses a single threaded model with event looping. Event mechanism helps the server to respond in a non-blocking way and makes the server highly scalable as opposed to traditional servers which create limited threads to handle requests. Node.js uses a single threaded program and the same program can provide service to a much larger number of requests than traditional servers like Apache HTTP Server.

4. **No Buffering** − Node.js applications never buffer any data. These applications simply output the data in chunks.

Node.JS

FEATURE OF NODE.JS

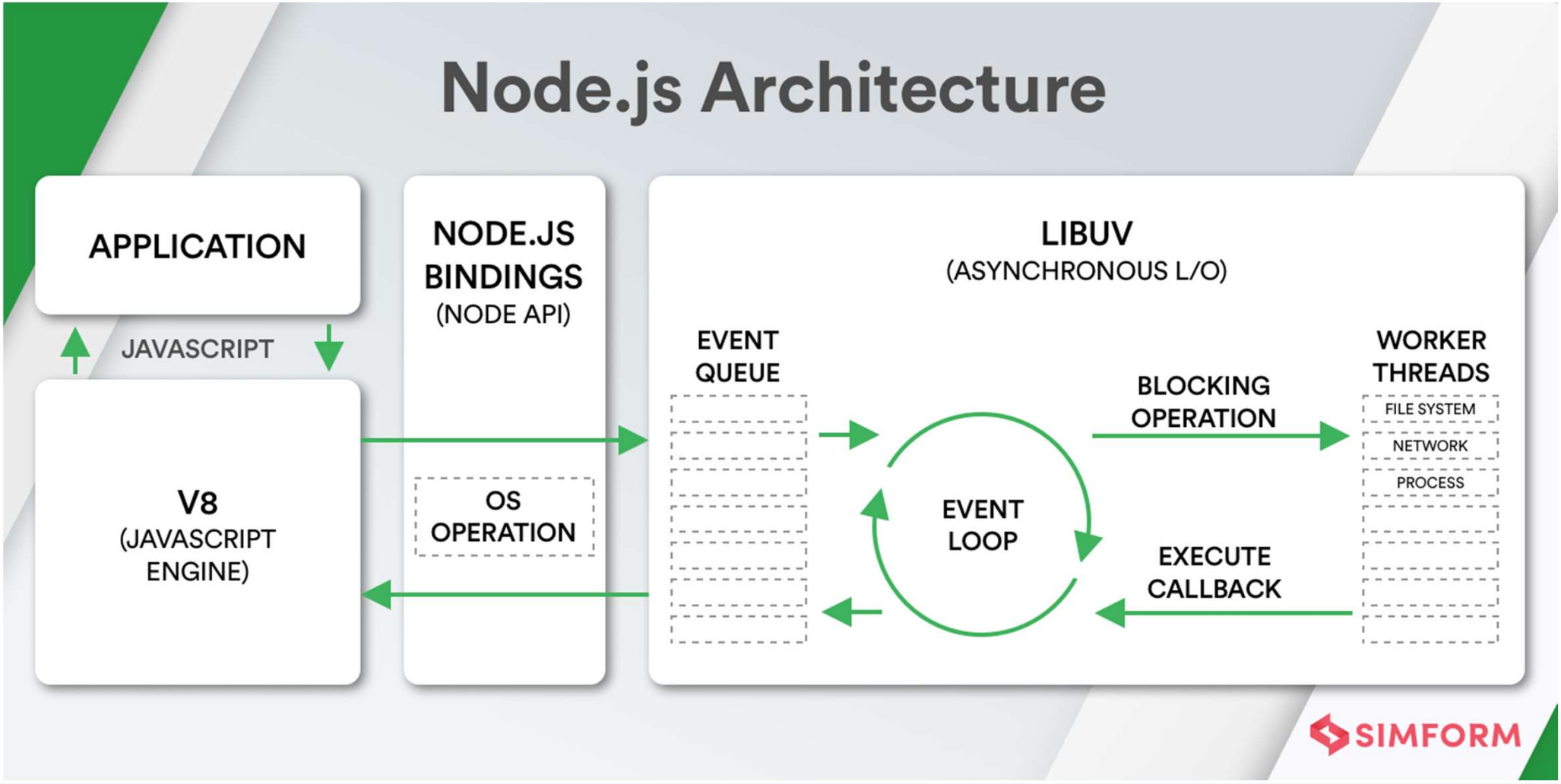
5. **Real time web apps:** Today the web has become much more about interaction. Users want to interact with each other in real-time. Chat, gaming, constant social media updates, collaboration tools, eCommerce websites, real-time tracking apps, marketplace- each of these features requires real-time communication between users, clients, and servers across the web.

6. **Compatibility on the cross platforms:** Different types of systems like Windows, UNIX, LINUX, MacOS and other mobile devices can use NodeJs. For generating a self-sufficient execution, it can be paired with any appropriate package.

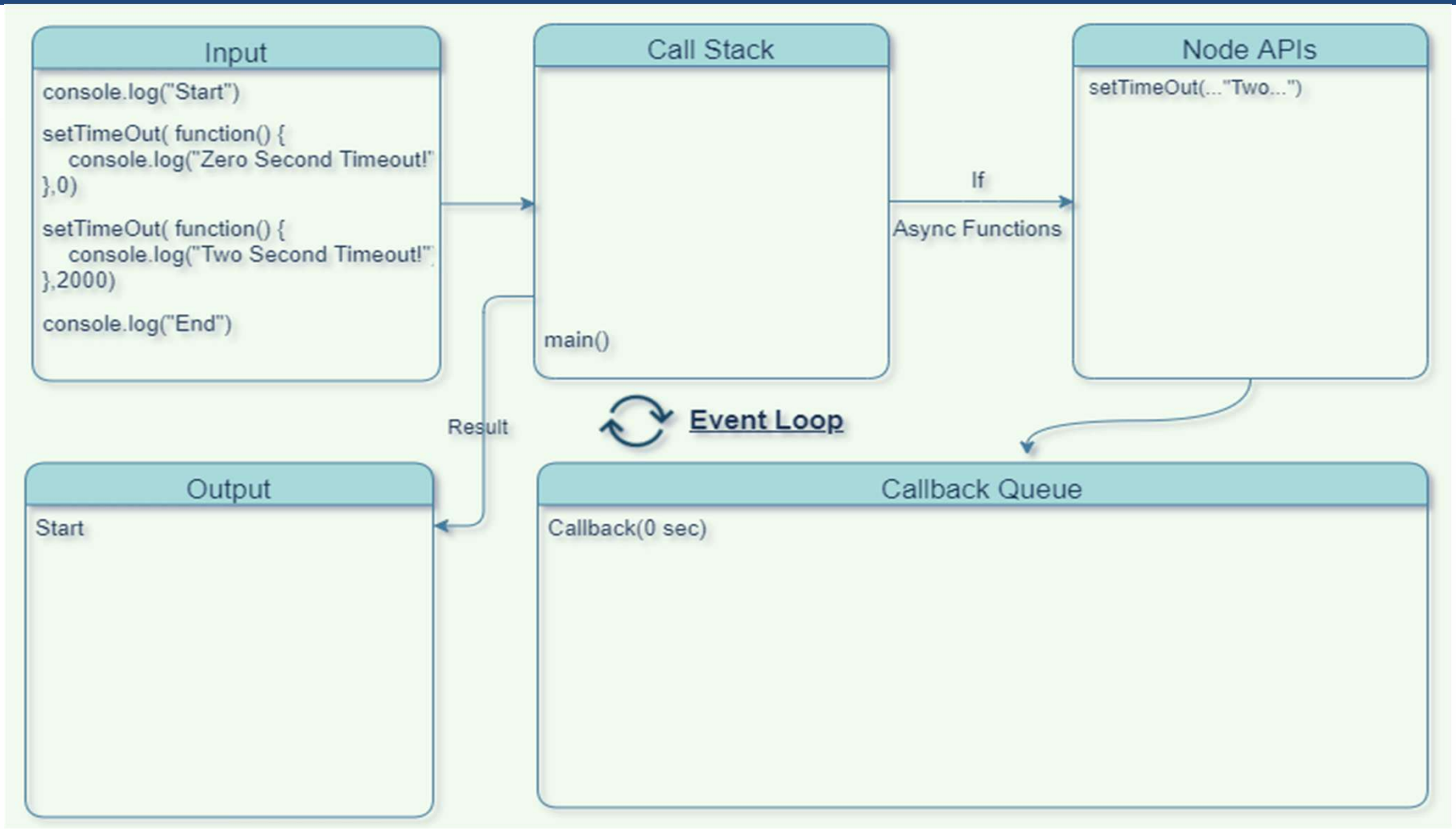
Parul® University

DIGITAL LEARNING CONTENT

Node.JS Architecture



Working of event loop



Parul® University

Working of event loop

The event loop in Node.js is a fundamental concept that allows it to handle asynchronous operations efficiently. Here's a concise explanation for your PowerPoint presentation

In essence, the event loop is the mechanism that allows Node.js to handle many operations concurrently, making it well-suited for scalable and high-performance applications.

Parul® University

Creating Node.js Application

Step 1 - Import Required Module

We use the **require** directive to load the http module and store the returned HTTP instance into an http variable as follows

```
var http= require("http");
```

Step 2 - Create Server

We use the created http instance and call **http.createServer()** method to create a server instance and then we bind it at port 8081 using the **listen** method associated with the server instance. Pass it a function with parameters request and response.

Creating Node.js Application

Step 3 - Testing Request & Response

Let's put step 1 and 2 together in a file called **main.js** and start our HTTP server as shown below −

```
const http = require('http');
const hostname = '127.0.0.1';
const port = 3000;
const server = http.createServer((req, res) => {
res.writeHead(200, { 'Content-Type': 'text/plain' });
res.end('Server running\n');
});

server.listen(port, hostname, () => {
    console.log(`Server running at http://${hostname}:${port}/`);
});
```

Creating Node.js Application

Step 3 - Testing Request & Response

Let's put step 1 and 2 together in a file called **main.js** and start our HTTP server as shown below −

```
const http = require('http');
const hostname = '127.0.0.1';
const port = 3000;
const server = http.createServer((req, res) => {
res.writeHead(200, { 'Content-Type': 'text/plain' });
res.end('Server running\n');
});

server.listen(port, hostname, () => {
    console.log(`Server running at http://${hostname}:${port}/`);
});
```

Parul®
University

NPM

NPM is the default package manager for Node.js, a runtime environment for executing JavaScript code server-side. NPM allows developers to discover, install, and manage third-party packages (libraries, frameworks, tools, etc.) for their Node.js applications. It simplifies the process of integrating external modules and tools into a Node.js project, making it a fundamental tool for Node.js developers.

Key Features of NPM

1. Developers can use NPM to install packages from the NPM registry.

For example:

npm install package-name

NPM

2. Dependency Management:
NPM automatically manages dependencies, ensuring that the required libraries and tools are installed and accessible for a given project.

3. Version Control:
NPM allows developers to specify version ranges or exact versions of packages to ensure consistency and compatibility within a project.

4. Scripting:
NPM provides a scripting mechanism through the "scripts" field in the package.json file. Developers can define custom scripts, such as running tests or starting the application, and execute them with simple commands.

NPM

5. Global and Local Installations:

Packages can be installed globally to make them available system-wide or locally within a specific project.

1. To install a package locally and add it to the package.json file.
   npm install package-name --save
2. Global Installations:
   For packages needed system-wide:
   npm install -g package-name

.

Parul® University

NodeJs Modules

In Node.js, a module is a reusable block of code that encapsulates related functionality and can be used to organize and structure a Node.js application. Modules in Node.js promote modular programming, making it easier to manage and maintain large codebases by breaking them down into smaller, more manageable pieces.

Types of Modules in Node.js:

1. Core Modules:
Core modules are built-in modules that come with the Node.js installation. They provide essential functionality for common tasks.
Example: fs (file system), http (HTTP server/client), path (path manipulation), etc.
Core modules are loaded using require without specifying a path, e.g., const fs = require('fs');

NodeJs Modules

2. Local Modules:

Local modules are modules created by developers to organize and encapsulate specific functionality within their Node.js application.
A local module is typically a separate JavaScript file that can be loaded into other files using require.
Example: Suppose you have a file named myModule.js:

```
const myFunction = () => {
  console.log('This is a custom module function.');
};

module.exports = myFunction;
```

Parul®
University

NodeJs Modules

2. Local Modules:

You can use this module in another file like this**:**

```
const myModule = require('./myModule');
myModule(); // This is a custom module function.
```

NodeJs Modules

3. Third-Party Modules:

Third-party modules are modules created by external developers and shared through the Node Package Manager (NPM).

Developers can use NPM to install third-party modules and include them in their projects.

Example: express for building web applications, lodash for utility functions, etc.
Installation: npm install module-name
Usage: const moduleName = require('module-name');

Parul®
University

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 is an open source framework developed and maintained by the Node.js foundation.

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

To install Express and add it to our package.json file, use the following command

npm install –save express

Parul®
University

Express.JS

To start developing our first app using Express. Create a new file called **index.js** and type the following in it.

```
const express = require('express');
const app = express();
app.get('/', (req, res) => { res.send('Hello, World!'); });
const port = 3000;
app.listen(port, () => { console.log(`Server running at http://localhost:${port}`); });
```

Save the file, go to your terminal and start application it will show “Hello, World!”

Parul® University

ExpressJS - HTTP Methods

S.No.	Method & Description
1	**GET** The GET method requests a representation of the specified resource. Requests using GET should only retrieve data and should have no other effect.
2	**POST** The POST method requests that the server accept the data enclosed in the request as a new object/entity of the resource identified by the URI.
3	**PUT** The PUT method requests that the server accept the data enclosed in the request as a modification to existing object identified by the URI. If it does not exist then the PUT method should create one.
4	**DELETE** The DELETE method requests that the server delete the specified resource.

Parul® University

DIGITAL LEARNING CONTENT

Express Middleware

Middleware in Node.js, particularly in the context of web frameworks like Express, is a function that has access to the request, response, and the next middleware function in the application's request-response cycle. Middleware functions can perform various tasks, modify the request or response objects, end the request-response cycle, and call the next middleware in the stack

Lets discuss an example…

Express Middleware

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

//Middlewares
app.use(express.json());
app.use(express.urlencoded({ extended: true }));

// Custom middleware to log incoming requests
app.use((req, res, next) => {
  console.log(`[${new Date().toISOString()}] ${req.method} ${req.url}`);
  next();
});

app.get('/', (req, res) => {
  res.send('Hello, World!');
});

const port = 3000;
app.listen(port, () => {
  console.log(`Server running at http://localhost:${port}`);
});
```

Parul® University

ExpressJS - Form data

Forms are an integral part of the web. Almost every website we visit offers us forms that submit or fetch some information for us. 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

npm install –save multer

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.

ExpressJS Database

We keep receiving requests, but end up not storing them anywhere. We need a Database to store the data. For this, we will make use of the NoSQL database called **MongoDB**.

In order to use Mongo with Express, we need a client API for node. There are multiple options for us, but for this tutorial, we will stick to mongoose. Mongoose is used for **document Modeling** in Node for MongoDB. For document modeling, we create a **Model** (much like a **class** in document oriented programming), and then we produce **documents** using this Model (like we create **documents of a class** in OOP). All our processing will be done on these "documents", then finally, we will write these documents in our database.

ExpressJS Database

We keep receiving requests, but end up not storing them anywhere. We need a Database to store the data. For this, we will make use of the NoSQL database called **MongoDB**.

In order to use Mongo with Express, we need a client API for node. There are multiple options for us, but for this tutorial, we will stick to mongoose. Mongoose is used for **document Modeling** in Node for MongoDB. For document modeling, we create a **Model** (much like a **class** in document oriented programming), and then we produce **documents** using this Model (like we create **documents of a class** in OOP). All our processing will be done on these "documents", then finally, we will write these documents in our database.

Parul® University

DIGITAL LEARNING CONTENT

ExpressJS Middleware

Express.js middleware is a crucial aspect of building web applications with the MEAN (MongoDB, Express.js, AngularJS, Node.js) stack. Middleware functions in Express.js play a pivotal role in handling requests, modifying the request and response objects, and enabling various features within your application.

Middleware functions in Express.js 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. Middleware can perform tasks, modify the request or response, end the request-response cycle, and call the next middleware in the stack.

Parul® University

ExpressJS Middleware

Middleware in Express are functions that come into play **after the server receives the request and before the response is sent to the client**. They are arranged in a chain and are called in sequence.

We can use middleware functions for different types of processing tasks required for fulfilling the request like database querying, making API calls, preparing the response, etc, and finally calling the next middleware function in the chain

Parul® University

ExpressJS Middleware

Middleware functions take three arguments: the request object (request), the response object (response), and optionally the next() middleware function:

```
const express = require('express');
const app = express();
function middlewareFunction(request, response, next){
  ...
  next()
}

app.use(middlewareFunction)
```

An exception to this rule is error handling middleware which takes an error object as the fourth parameter. We call app.use() to add a middleware function to our Express application.

Type of ExpressJS Middleware

Application-level Middleware:

Application-level middleware is bound to the app object and executes for every request to the application.

```
app.use(function(req, res, next) {
  // middleware logic
  next();
});
```

Application-level middleware which runs for all routes in an app object

Type of ExpressJS Middleware

Router-level Middleware:

Router-level middleware is bound to an instance of express.Router() and executes for routes defined on that router

```
var router = express.Router();
router.use(function(req, res, next) {
  // middleware logic
  next();
});
```

Router level middleware which runs for all routes in a router object

Type of ExpressJS Middleware

Error-handling Middleware:

Error-handling middleware is defined with four parameters, and it gets invoked when an error occurs during the request-response cycle.

```
app.use(function(err, req, res, next) {
  // error-handling logic
  res.status(500).send('Something went wrong!');
});
```

Error handling middleware for handling errors

Type of ExpressJS Middleware

Built-in Middleware:

Express provides several built-in middleware functions that can be easily integrated into your application

- **express.json(): Parses incoming requests with JSON payloads.**
- **express.urlencoded(): Parses incoming requests with URL-encoded payloads.**
- **express.static(): Serves static files**

Type of ExpressJS Middleware

Third-party Middleware:

- Developers can use third-party middleware to add additional functionality to their applications.
- Example (using **body-parser** for parsing JSON or **multer** file upload):

```
const bodyParser = require('body-parser');
app.use(bodyParser.json());
```

- Third-party middleware maintained by the community

Example of ExpressJS Middleware

Authentication Middleware:
Verifies if the user is authenticated before allowing access to protected routes.

```
function authenticate(req, res, next) {
  if (req.isAuthenticated()) {
    return next();
  }
  res.redirect('/login');
}
app.get('/dashboard', authenticate, function(req, res) {
  res.render('dashboard');
});
```

Middleware functions are executed in the order they are declared in your application code.
To move to the next middleware in the stack, the next() function must be called.

Parul® University

DIGITAL LEARNING CONTENT

ExpressJS Routing

Routing in Express involves defining routes for handling different HTTP requests and specifying the corresponding logic to execute when a request matches a particular route. Express uses a chain of middleware functions to handle requests, and each route can have its own set of middleware

Routing defines the way in which the client requests are handled by the application endpoints.

Implementation of routing in Node.js:

There are two ways to implement routing in node.js which are listed below:

- By Using Framework
- Without using Framework

Parul® University

DIGITAL LEARNING CONTENT

ExpressJS Routing

Using Framework:

Node has many frameworks to help you to get your server up and running. The most popular is Express.js.

Routing with Express in Node:

Express.js has an “app” object corresponding to HTTP. We define the routes by using the methods of this “app” object. This app object specifies a callback function called when a request is received. We have different methods of in-app objects for different types of requests.

ExpressJS Routing

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

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

app.listen(3000)
```

ExpressJS Routing

1. Basic Routing:

Use app.get(), app.post(), app.put(), app.delete(), etc., to define routes based on HTTP methods

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

app.get('/', (req, res) => {
  res.send('Hello, this is the homepage!');
});

app.post('/users', (req, res) => {
  res.send('Creating a new user');
});

// Additional HTTP methods can be used for different routes
```

ExpressJS Routing

2. Route Parameters:

Route parameters are placeholders for values that are part of the URL.

Example:
Access route parameters using req.params.

```
app.get('/users/:id', (req, res) => {
  const userId = req.params.id;
  res.send(`User ID: ${userId}`);
});
```

ExpressJS Routing

3. Handling Multiple Route Parameters:

Example:
Define routes with multiple parameters

```
app.get('/users/:category/:id', (req, res) => {
  const category = req.params.category;
  const userId = req.params.id;
  res.send(`Category: ${category}, User ID: ${userId}`);
});
```

ExpressJS Routing

4. **Route Middleware:**

Middleware functions that have access to the request, response, and next function in the application's request-response cycle.

Example:

Implement middleware for route-specific tasks.

```
const logRequest = (req, res, next) => {
  console.log(`Received request for ${req.url}`);
  next();
};

app.get('/users', logRequest, (req, res) => {
  res.send('List of users');
});
```

ExpressJS Routing

5. Express Router:

An isolated instance of middleware and routes, used to structure and organize the application.

Example:

Use express.Router() to create modular routers.

```
const express = require('express');
const router = express.Router();

router.get('/', (req, res) => {
  res.send('Router homepage');
});

router.get('/about', (req, res) => {
  res.send('About page');
});

module.exports = router;
```

ExpressJS Routing

6. Express.js Route Redirects:

Redirect clients to a different route.

Example:
Redirect from one route to another.

```
app.get('/old-route', (req, res) => {
  res.redirect('/new-route');
});
```

Parul® University

Promise in JS

Promises are a powerful feature in JavaScript used to handle asynchronous operations. They provide a clean and structured way to work with asynchronous code, making it easier to manage.

A Promise in JavaScript represents the eventual completion or failure of an asynchronous operation and its resulting value. Promises have three states: pending, fulfilled, and rejected. They are especially useful when dealing with tasks like fetching data from an API, reading files, or making database queries.

Promise in JS example

```
// Function that returns a Promise to simulate fetching data
const fetchData = () => {
  return new Promise((resolve, reject) => {
    // Simulating an asynchronous operation (e.g., fetching data from an API)
    setTimeout(() => {
      // Simulating a condition to demonstrate resolving or rejecting the promise
      const shouldReject = Math.random() < 0.2; // 20% chance of rejection
```

Promise in JS example

```
if (shouldReject) {
        reject(new Error('Failed to fetch data'));
      } else {
        const data = { message: 'Data fetched successfully' };
        resolve(data);
      }
    }, 1000); // Simulating a delay of 1 second
  });
};
fetchData()
  .then((data) => {
    console.log(data.message); // Output: Data fetched successfully
  })
  .catch((error) => {
    console.error(error.message); // Output: Failed to fetch data
  });
```

async and await

async and await are features in JavaScript that simplify working with asynchronous code, making it look more like synchronous code.

An async function always returns a Promise. Inside an async function, you can use the await keyword to wait for a Promise to resolve.

Inside an async function, you can use await to pause the execution of the function until the Promise is resolved.

async and await

```
async function fetchDataWithAwait() {
  try {
    const data = await fetchData();
    console.log(data.message); // Output: Data fetched successfully
  } catch (error) {
    console.error(error.message);
  }
}

fetchDataWithAwait();
```

Using try and catch blocks, you can handle errors in a cleaner way compared to using .then() and .catch()

```
async function fetchDataWithTryCatch() {
  try {
    const data = await fetchData();
    console.log(data.message);
  } catch (error) {
    console.error('Error:', error.message);
  }
}

fetchDataWithTryCatch();
```

Parul® University

DIGITAL LEARNING CONTENT

Authentication and security with Passport.js

DIGITAL LEARNING CONTENT

Parul® University

www.paruluniversity.ac.in