8. Node.js

What Is Backend Development?

- Frontend runs on **client's web browser** and the web browser's engine
  - Browser's engine communicates with the backend
- Backend runs on a **server**
- Backend developers writes the code that communicates with the browser's engine and the backend server
- Backend Responsibilities
  - Scalability → Application's ability to dynamically handle the load as it grows or shrinks
    - Load refers to the number of concurrent users, the number of transactions, and the amount of data transferred back and forth between the clients and servers
  - Security and prevention of malware attacks
  - Authentication

Key Components Of Backend Development

- **Server:** The hardware/software that provides resources, data, or services to clients over a network
- **Database:** A structured collection of data that can be easily accessed, managed, and updated
  - **SQL Databases:** Relational databases like MySQL, PostgreSQL, and SQLite
  - **NoSQL Databases:** Non-relational databases like MongoDB, Cassandra, and Redis
- **Runtime:** A mini operating system that provides the resources necessary for an application to run
  - The runtime is the infrastructure that supports the execution of a codebase
- **Server-Side Languages:** Programming languages used to build the back-end logic
  - Node.js, Python, Ruby, Java

Popular Server-side Frameworks

1. **Node.js**
   - JavaScript runtime with non-blocking I/O and rich npm ecosystem
   - Widely used for full-stack development
2. **Django**
   - Python framework with built-in features for rapid development
   - Highly secure and scalable, ideal for high-traffic sites
3. **Spring Boot**
   - Java framework for enterprise applications and microservices
4. **Laravel**
   - PHP framework with elegant syntax and robust features like routing and ORM
5. **Express.js**
   - Minimal and flexible Node.js framework for APIs and web apps
   - Core part of MEAN/MERN stacks with middleware support

Node.js Overview

- Node.js is **server-side runtime environment** that executes **javaScript** on a server
- It runs on the **V8** engine, developed by Google, is integrated into all Google Chrome browsers
    - Every piece of code you write needs to undergo processing and conversion into a machine-readable form, which is done using v8 in Node.js
    - V8 can operate on various operating systems like Linux, Windows, and Mac OSX
- **Event-Driven, Asynchronous, Non-Blocking, Single-Threaded:**
    - In a single-threaded environment, only one command is processed at any given time, while in a multi-threaded environment, multiple commands can be processed simultaneously
    - Despite being single-threaded, it excels in performance due to its asynchronous and non-blocking nature
        - This means that while a process is being executed, the program does not need to wait until it finishes
    - Node.js is event-driven, meaning that when it performs an input/output (I/O) operation, such as reading from the network or accessing a database or file system, an event is triggered
        - Instead of blocking the thread and consuming processor time while waiting, Node.js resumes operations. This non-blocking behavior enables the server to remain responsive and handle multiple tasks concurrently, akin to a multi-threaded environment

Workflow

1. User selects an option in the user interface, written in HTML and CSS
2. This action by the user triggers JavaScript code that implements the business logic on the client-side, for example, input validation
3. The JavaScript application makes a web service call over HTTP with a JSON data payload
4. The REST web service, which is part of a node.js application running on the node server, receives the HTTP request
5. The REST web service processes the request and returns the result to the client as a JSON payload over HTTP

Node.js Modules

Modules

- A file containing **JavaScript** code that serves a specific purpose
  - Every javaScript file is a module in node.js
- Allows developers to break down complex code into manageable chunks and promote reusability
- Modules can be single files or collections of files and folders

Module Specifications

- Conventions used to **create** packages in JavaScript for Node.js applications
- Most commonly used module specifications for node.js → CommonJS and ES modules
- Node.js defaults to treating JavaScript code as a CommonJS module
  - ES modules can be enabled by changing the file extension

Import vs. Require

- The **require** → used in CommonJS modules and can be called anywhere in the code
  - Dynamically bound → This means errors that occur when linking the function definition to the function call will not be identified until run-time
  - Synchronous → the modules will be loaded and processed in a linear fashion, one at a time
- The **import** → used in ES modules and must be at the beginning of the file
  - Statically bound → binding errors are identified at Compile time
  - Asynchronous → the modules can be processed simultaneously

require()

```
//export from a file named
//message.js
module.exports = 'Hello
  Programmers';
```

```
//import from the message.js
//file
let msg =
  require('./message.js');
console.log(msg);
```

import()

```
//export from file named module.mjs
const a = 1;
export { a as "myvalue" };
```

```
//import from module.mjs
import { myvalue } from
  module.mjs";
```

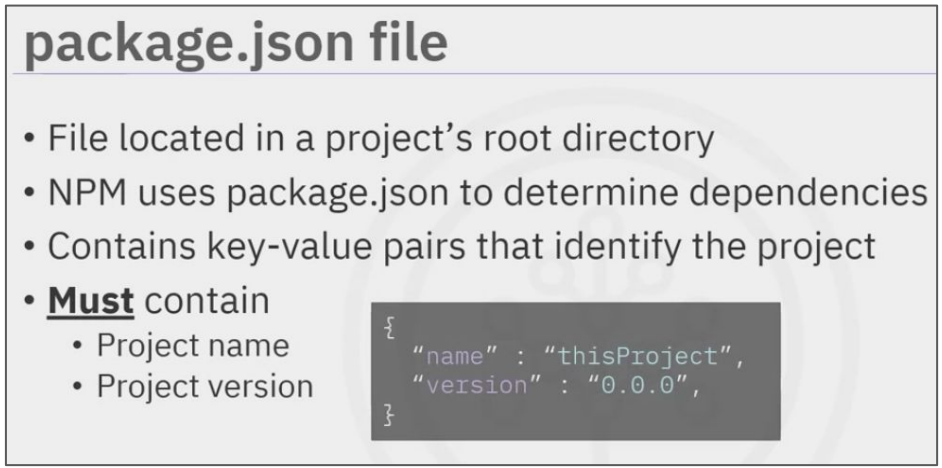
- Require() statement assumes that scripts have a file extension of **.js**
  - require() function creates an **object** representing the imported module
  - You can access exported properties from the module by retrieving them from the **instance** of the object
  - When you call require() with the name of a subdirectory, Node.js looks for a script file with the same name as the subdirectory
    - If the script file does not exist, the function assumes that the name is the name of a directory and looks for a script named index.js within that directory

There are **three** types of modules

- **Core**
  - They contain the minimal functionality needed to develop Node.js applications
  - The most important of the core modules are `http`, `path`, `fs`, `os`, `util`, `url`, and `querystring`
- **Local**
  - Modules written by you as part of creating your Node.js application
- **Third-party**
  - Available online and have been created by the back-end Node.js community
  - These libraries are available to use as stated per their licenses

Dependency Management in Node.js Using NPM

- Dependencies → Libraries or modules that a program **relies on** to function correctly
- Package manager → Ensures these dependencies are **managed** efficiently
    - It automates the process of finding, installing, upgrading, and removing packages and their dependencies
- NPM → Default package manager for node.js
- Every npm package requires a `package.json` file



Local npm install vs **Global** npm install

- Local → To install a package you want to use **within** your application
  - Run the local install command from the directory you want the package installed in
  - The local install is npm's default behavior
  - `npm install <package_name>`
    - Creates a directory named `node_modules` with the package and its dependencies
- Global → To install packages that need to be accessed by **any** application on the machine
  - `npm install -g <package_name>`

Creating a Web Server with Node.js

- To create an instance of a web server, use the **HTTP.createServer** function
  - The web server is stored in a variable called "server"
  - The createServer function takes in an optional callback function as a parameter
    - This callback function handles the incoming request message and provides an appropriate response message
    - After creating the server instance, you can set it to listen on a specific port, such as 8080

```
let server = http.createServer(function(request, response) {

  let body = "Hello world!";

  response.writeHead( 200, {

    'Content-Length': body.length,

    'Content-Type': 'text/plain'

  });

  response.end(body);

});

server.listen(8080);
```

Create a Node.js Server

1. Verify that node CLI is installed
   - `node --version`
2. Create an `index.js` file
3. Start the node server
   - `node index.js`
     - It will execute all the files imported through either import or require
4. Use the `curl` command to ping the application
   - The curl command is a command-line tool for transferring data to or from a server using various protocols, such as HTTP, HTTPS, FTP, etc
   - It's commonly used for making HTTP requests (GET, POST etc) and testing APIs
   - Exp: `curl localhost:8080`

Asynchronous I/O & Callback Functions

- When an application **blocks** (or **waits**) for a network operation to complete, that application **wastes** processing time on the server
- Node.js makes all network operations in a **non-blocking** manner
  - Every network operation returns immediately
  - To handle the **result** from a network call, write a **callback function** that Node.js calls when the network operation **completes**
- The `HTTP.request()` method accepts an optional callback function that is invoked immediately once a response is received
  - If the request is successful, a 'data' event is emitted on the response object, followed by an 'end' event when the response finishes
  - If the request fails, there is an 'error' event followed by the 'close' event

```
1   let options = {
2     host: 'w1.weather.gov',
3     path: '/xml/current_obs/KSFO.xml'
4   };
5
6   http.request( options,
7   function(response) {
8       let buffer = '';
9       let result = '';
10
11     response.on('data', function(chunk) {
12     buffer += chunk;
13   });
14
15   response.on('end', function() {
16      console.log(buffer);
17   });
18
19 }).end();
```

Example → Making an `HTTP.request()` call

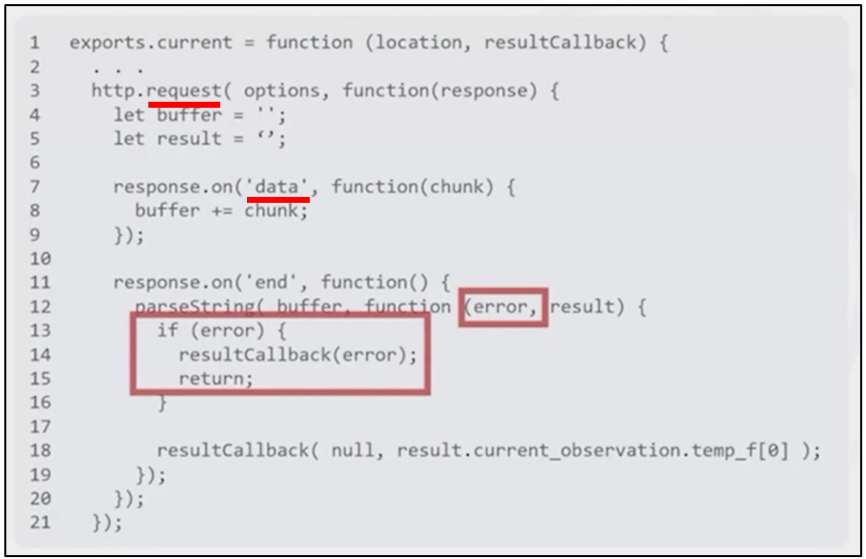
- The first parameter in the HTTP request function is an `options` variable
  - The options variable includes at least two variables: the hostname of the remote server, and a uniform resource locator (URL) resource path that you want to act upon
- The second parameter is a `callback function`
  - In this case, it is an anonymous function that receives one parameter: the response object
  - Invoked immediately once a response is received

Error Handling In Callback Functions

- The **first** parameter of a callback function is typically an **error** object
- If an error occurs, the callback function checks the first parameter and handles the error and cleans up any open network or database connections
- If no error is present, the callback processes the result, allowing the application to respond appropriately

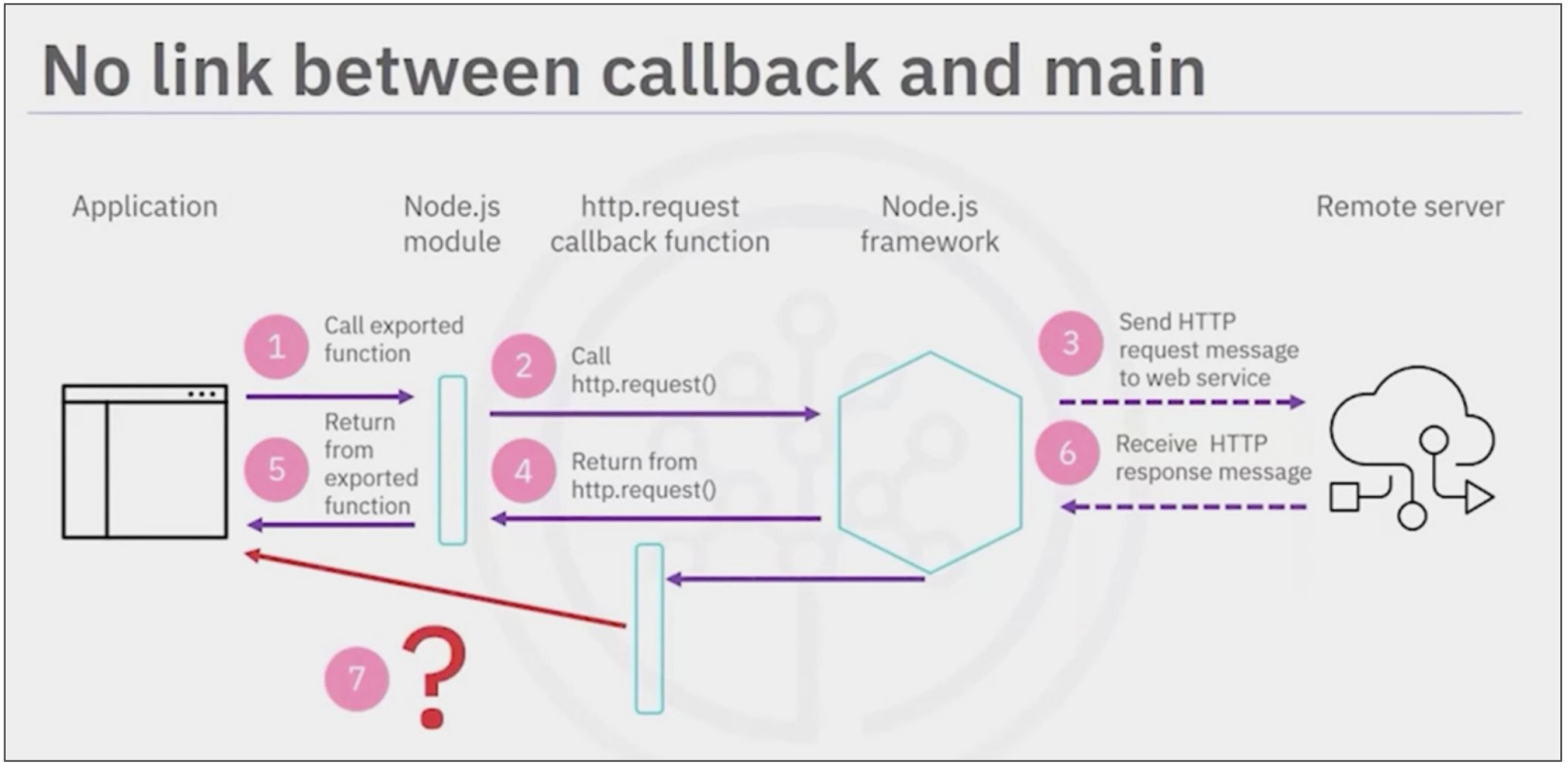
Example → Passing an error object to the callback function

- Recall how callback functions check the first parameter to see if an error condition occurred. Instead of printing the result in the console, you call the `resultCallback` callback function with the `error` object
- If no error occurred, you call the `resultCallback` function with `null` as the first parameter



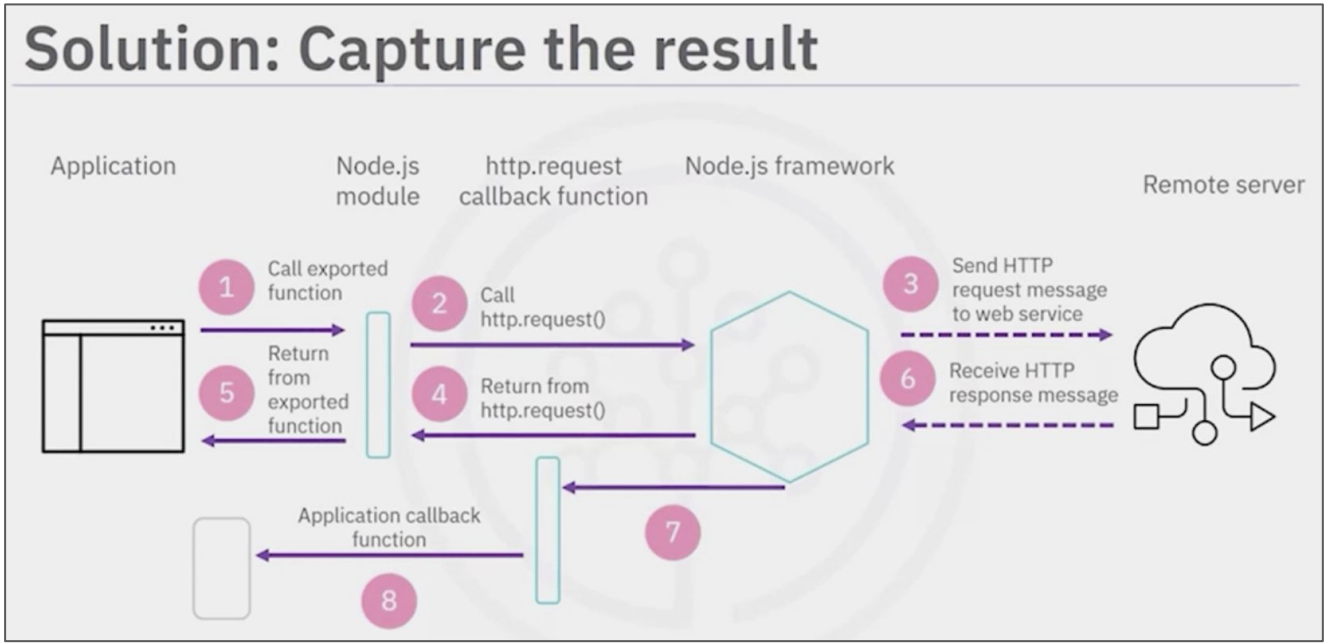
```
exports.current = function (location, resultCallback) {
  . . .
  http.request( options, function(response) {
    let buffer = '';
    let result = '';

    response.on('data', function(chunk) {
      buffer += chunk;
    });

    response.on('end', function() {
      parseString( buffer, function (error, result) {
        if (error) {
          resultCallback(error);
          return;
        }

        resultCallback( null, result.current_observation.temp_f[0] );
      });
    });
  });
```

Issue: In Passing Message From A Node.js Module To The Main Application



Solution

- At a **future** point, when the remote server sends back an HTTP response message
  - The Node.js framework calls the callback function defined by the Node.js module
  - This callback function calls **another** callback function defined by the main application



Issues With Callback

- A callback → A function that is passed as an argument to another function that executes the callback **based on** the result
  - They are basically functions that are executed only after a result is produced
  - Callbacks make sure that a function won't run before a prerequisite task is completed
- **Two** Major Issues
  - Nested Callbacks → Callback Hell
  - Inversion of Control (IOC)
- Mitigation strategies
  - Writing **comments**
  - **Breaking** functions into smaller, more manageable pieces
  - Utilizing **Promises** or **async/await** for better control over asynchronous operations
    - Detailed explanation with examples

1. Nested Callbacks

Example:

- Say you are making a cake
- These are the steps
  - Purchase cake ingredients. Combine ingredients. Bake the cake. Decorate the cake. Serve the cake
- These steps cannot be completed at the same time
  - In other words, these steps cannot be completed asynchronously

The pseudocode for the cake example

- Each subsequent function becomes the argument passed to the next function
- Every callback depends on and waits for the previous callback, thereby making a pyramid structure that affects the readability and maintainability of the code
- This nesting of callback functions is often referred to as “Callback Hell”

```
const makeCake = nextStep => {
  buyIngredients(function(shoppingList) {
    combineIngredients(bowl, mixer, function(ingredients){
      bakeCake(oven, pan, function(batter) {
        decorate(icing, function(cake) {
          nextStep(cake);
        });
      });
    });
  });
};
```

2. Inversion of Control (IoC)

- Happens when the execution **flow** is managed by **external** code
- Often when callbacks handover control to a **third party** service that may have bugs or unexpected behavior
    - Developers may need to implement additional logic to handle these potential errors
    - Ensure that third-party code does not:
        - Get called too many or too few times
        - Get called too early or too late
        - Pass back incorrect arguments

Promises

- A promise is an **object** returned by an **asynchronous** method
- Has **three** states: **Pending**, **Resolved**, or **Rejected**
  - The initial state of a promise is pending, and it remains in this state until the operation completes or fails
  - When the operation is successful, the promise is resolved, indicating that the expected result is available
  - If an error occurs during the operation, the promise is rejected, and an error message is provided
- Promises are particularly useful for handling **time-consuming** operations, such as API requests and I/O operations

User-Defined Promise

- A user-defined method can be defined to return a promise object

Example:

- If the filename is valid, the content of the file is read and displayed. In this case the promise is **resolved**
- If the filename is invalid, the promise is **rejected** and the error message is displayed

```
let prompt = require('prompt-sync')();
let fs = require('fs');

const methCall = new Promise((resolve,reject)=>{
  setTimeout(()=>
  {
    let filename = prompt('What is the name of the file ?');
    try {
      const data = fs.readFileSync(filename, {encoding:'utf8', flag:'r'});
      resolve(data);
    } catch(err) {
      reject(err)
    }
  },3000);
});

console.log(methCall);

methCall.then(
  (data) => console.log(data),
  (err) => console.log("Error reading file")
);
```

Axios Package

- The **axios** package in Node.js is commonly used to handle **REST APIs** i.e making HTTP requests (GET, PUT, POST etc)
- Returns a **promise** that can be resolved or rejected based on the request's success or failure
- Example: The promise object has a "then" method which is called after the promise is fulfilled. The catch is executed if the promise is rejected

```
const axios = require('axios').default;

const connectToURL = (url)=>{
    const req = axios.get(url);
    console.log(req);
    req.then(resp => {
        console.log("Fulfilled")
        console.log(resp.data);
    })
    .catch(err => {
        console.log("Rejected")
    });
}
```

Working With JSON

- JSON is the **standard** format for API **data exchange**
- A JSON object consists of **attribute-value pairs**

```
{
  "Company": "IBM",
  "Country": "USA",
  "Headquarters": "Armonk, New York"
}
```

- To convert a JSON string into a JavaScript object → use the `JSON.parse()` method
- To convert a JavaScript object back into a JSON string → use the `JSON.stringify()` method

Extending Node.js With Third-Party Packages

- Node.js has **limitations** in its default framework for building web servers, necessitating the use of **external** libraries for features like routing, authentication, and database connections
- Example:
  - `npm install xml2js`
  - To parse **XML** data into **JavaScript objects**, avoiding the inefficiencies of string matching
  - Simplifies data manipulation and access by converting XML elements into **JavaScript objects**

Node.js Vs Node Web Framework

Node.js

- Is **not** a framework
- It is a **runtime environment** that executes javaScript on a server

Node Web Framework

- Web frameworks are fundamental structure on which the web applications are built
- So, basically node web frameworks utilize the node.js runtime environment to run

Approaches to build back-ends that web frameworks can **employ**

- Model-View-Controller (MVC)
- REST API

Model-View-Controller (MVC)

- Divides an application into **three** components
  - **Model** → Responsible for managing the **data** of the application
    - It interacts with the database and handles the data logic
  - **View** → Responsible for rendering the **presentation of the data** that is passed to it by the model
  - **Controller** → Regulates the **flow of the data**
    - It is responsible for processing the data supplied to it by the user and sends that data to the model for manipulation or storage
- Example Web Frameworks: Koa, Django, Express, NestJS

REST API

- Enables **communication** between web services
- Allows client and server code to operate independently and using HTTP methods
- RESTful APIs are stateless
  - Stateless means the client does not need to know the state of the server, nor does the server need to know the state of the client for communication between the two
- The representation of the data transferred between client and server is usually a **JSON** payload but can also be HTML, XLT, Python, PHP, or plain text.

Express Web Application Framework

- Express is a web application framework built on the Node.js runtime environment, designed to simplify application organization and speed up development
- It provides robust mechanisms for integrating middleware and handling various HTTP request methods

Primary Uses of Express

- **Creating APIs** → Allowing interaction with the data layer and sending data back to clients in JSON format
- **Server-side rendering (SSR)** → Dynamically generating HTML, CSS, and JavaScript based on client data

```
1  const express = require('express')
2  const app = express()
3  const port = 3000
4  ...
5  let server = app.listen(port, function(){
6    console.log(`Listening on URL http://localhost:${port}`);
7  })
```

Steps To Download and Set Up Express

- Declare `Express` as a **dependency** in the `package.json`file of your Node.js **project**
  - The `package.json`file stores information about the contents of a Node.js module including five items:
    - Name
    - Version
    - Description
    - Main
    - Dependencies
- To download Express, run `npm install` in your project folder
  - When you run the command `npm install` with no parameters, the npm application scans your `package.json`file. It checks your `node_module`directory to see if any modules are missing. In this case, the Express web application framework is not in your current installation. The npm application downloads the Express framework and all of its dependent modules

Introduction to Middleware

- **Middleware** → A **function** in Express.js that processes **requests** and **responses**
  - Creating middleware is simple. It takes three parameters
    - Request
    - Response
    - Next
  - Define a method that takes these three parameters and then bind it with `app.use` or `router.use`
  - Useful for
    - Parsing requests (e.g., JSON, URL-encoded data)
    - Adding authentication and session management
    - Handling errors
  - Middleware functions have access to request and response objects and can be chained together to perform various tasks, such as authentication and error handling
- **Express** → A **messaging framework**
  - The front end of an application uses Express to facilitate communication with the back end without the front-end and back-end needing to use the same language
  - The front end communicates with the middleware, not directly with the back end

There are **five** types of middleware

- **Application-level**
  - Bound to the **entire** application using `app.use`
  - Processes **all client requests** to the application
  - Common Uses:
    - Authentication
    - Checking session information

```
app.use((req, res, next) => {
  console.log('Application-level middleware');
  next(); // Pass control to the next middleware
});
```

- **Router-level**
  - Bound to an instance of `express.Router()`
  - Applied to **specific routes** rather than all requests

```
const router = express.Router();
router.use('/route', (req, res, next) => {
  console.log('Router-level middleware for /route');
  next();
});
```

- **Error-handling**
  - Handles application errors
  - Can be bound to the entire app or specific routers
  - Must have **four parameters**: (err, req, res, next)
  - Even if next is not used, it must be included in the function signature.

```
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send('Something broke!');
});
```

- **Built-in**
  - Provided by Express.js for common tasks
    - express.static: Serves static files
    - express.json: Parses JSON data from requests
    - express.urlencoded: Parses URL-encoded data
  - Can be bound to the entire app or specific routers

- **Third-party**

Introduction To Routing

- **Routing** → A part of the code that **associates** an HTTP **request** (such as GET, POST, or DELETE) with (a **URL** and the **function**) that gets called to handle that URL
- Routing is crucial for handling requests to different endpoints, which can be GET, POST, PUT, or DELETE
    - Each request must be managed appropriately or return an error
- Routing can be handled
    - At the **application** level
    - At the **router** level
        - A router by itself is used for branching query handling and routing each query differently
        - Example: you are defining two routers. One is for an item and the other is for a user. All the requests that come with the item are handled by the itemRouter. The /item/about and /item/detail routes are handled. All the requests that come with /user are handled by userRouter

At Application Level

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

app.get("user/about/:id",(req,res)=>{
    res.send("Response about user "+req.params.id)
})

app.post("user/about/:id",(req,res)=>{
    res.send("Response about user "+req.params.id)
})

app.get("item/about/:id",(req,res)=>{
    res.send("Response about user "+req.params.id)
})

app.post("item/about/:id",(req,res)=>{
    res.send("Response about user "+req.params.id)
})

app.listen(3333, () => {
    console.log(`Listening at http://localhost:3333`)
})
```

At Router Level

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

let userRouter = express.Router()
let itemRouter = express.Router()

app.use("/item",itemRouter)
app.use("/user",userRouter)

userRouter.get("/about/:id",(req,res)=>{
    res.send("Response about user "+req.params.id)
})

userRouter.get("/details/:id",(req,res)=>{
    res.send("Details about user "+req.params.id)
})

itemRouter.get("/about/:id",(req,res)=>{
    res.send("Information about item "+req.params.id)
})

itemRouter.get("/details/:id",(req,res)=>{
    res.send("Details about item "+req.params.id)
})

app.listen(3333, () => {
    console.log(`Listening at http://localhost:3333`)
})
```

Introduction To Template Rendering

- Template rendering → The ability of the server to **fill** in **dynamic** content in the **HTML template**
- This example uses `express-react-views` which renders React components from the server
  - Set the view engine property, which is responsible for creating HTML from your views
    - Views are JSX code
    - The views are in a directory named `myviews`
    - The view engine will look for a JSX file named `index` in the `myviews` directory and pass the property `name` to it. The output rendered will have the name of the user

```
const express = require('express');
const app = new express();
const expressReactViews = require('express-react-views');

const jsxEngine = expressReactViews.createEngine();

app.set('view engine', 'jsx');

app.set('views', 'myviews');

app.engine('jsx',jsxEngine);

app.get("/:name",(req,res)=>{
    res.render('index', { name: req.params.name });
  });

app.listen(3333, () => {
    console.log(`Listening at http://localhost:3333`)
```

```
const React = require('react');

function HelloMessage(props) {
  return <div>Hello {props.name}</div>;
}

module.exports = HelloMessage;
```

Authentication & Authorization in Node.js

- Authentication **verifies** a user's **identity** through credentials, ensuring that only authorized users can access specific parts of the system
- Three popular authentication methods
  - Session-based
  - Token-based
  - Passwordless authentication

1. Session-Based

- **Log in** with your **credentials**, which are validated against a database
- If valid, the server creates a **unique** encrypted **session ID**, which is stored in
    - Database and As a browser cookie
- When you log out or after a certain period, the session ID is destroyed, logging you out from both the browser and the database

2. Token-Based

- **Most** Popular approach for authentication in node.js
- **Tokens**→ Often in the form of **JSON Web Tokens** (JOT)
  - JOT → Internet standard for creating encrypted JSON payload data
  - `npm install -save jsonwebtoken`
- Includes two key concepts
  - Token-based **authentication**
    - When you provide credentials and receive a token that validates your user credentials
  - Token-based **authorization**
    - When you use **that** token to access resources, ensuring the resource server knows which resources you're permitted to access
    - Occurs when a web application requests access to a protected resource, such as an API
      - During this process, the user authenticates with the authorization server, which then generates an access token that is sent back to the client and stored
      - With each HTTP request, the token is passed to the resource API server, carrying embedded permission details, eliminating the need to query the authorization server
- Advantages
  - More **scalable** as the token only needs to be stored on the client side. Also, since the server only needs to verify the token along with the user information, it is easier to handle multiple users

- Implementation → The application will have 2 APIs
  - A **POST** API for logging in
    - Returns a **web token** when the correct username and password are provided
  - A **GET** API for resource access
    - Requires a valid token in the Authorization header to access resources
    - Reads the Authorization header from the incoming API request
      - If no header is found within the request, then a Status code of 401 is sent back by the GET API with the message "No Token"

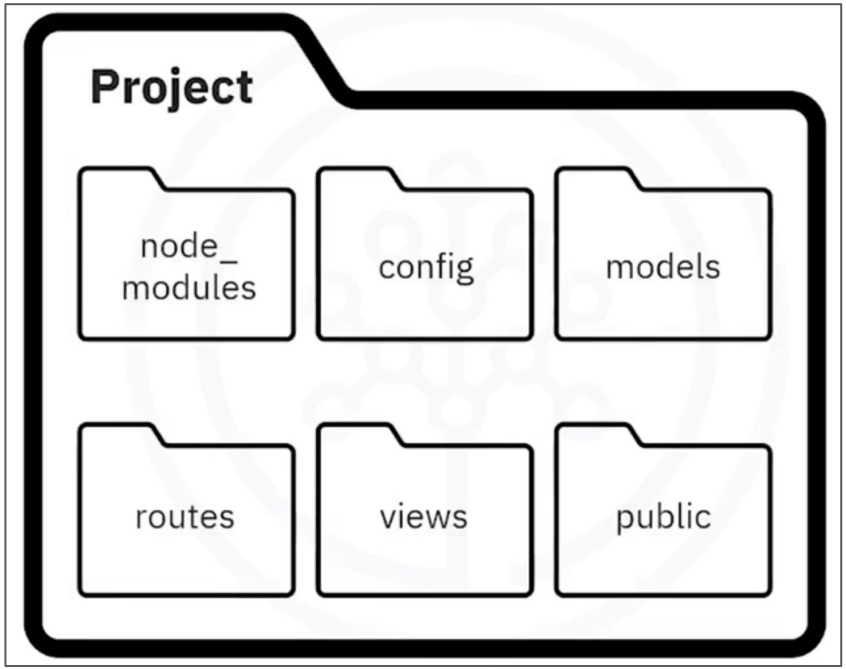
3. Password-less

- Eliminates the need for **traditional** passwords using methods like
  - Biometrics
  - Magic links sent to your email
  - One-time passcodes
- Often used in password recovery systems
- Achieved using **Public Key** and **Private Key** Encryption
  - When a user registers for the app, the user's device generates a private key/public key pair
  - The public key is used to **encrypt** messages, and the private key is used to **decrypt** them
  - The private key is stored on the user's device, and the public key is stored with the application
  - Anyone may access the public key, but the private key is only known to the client
  - When the user signs into the application, the application generates a login challenge, such as requesting biometrics, sending a "magic link," or sending a special code via SMS, encrypting it with the public key. The private key allows the message to be decrypted. The app then verifies the sign-in challenge and accepts the response to authorize the user

Code snippets in Express For The Above 3 Authentication Methods

Express Best Practices

- Suggested Express **Folder** Structure



- **node_modules** → Contains the application's modules and packages
  - It is automatically created after running the "npm install" command
- **config** → Contains configuration files such as
  - database connection configuration
  - an environment variables file
  - a credentials file containing the API keys for external services used by the application under development
- **models** → Contains the data models for the application
  - The files specify the type of datastore, such as relational or non-relational, and are defined by an object-relational mapping, or ORM, library
- **routes** → Contains all of the routes for the different entities in different files. It should have one file for each logical set of routes, such as one file for one type of resource
- **views** → Contains template files
  - A template dynamically writes HTML, CSS, and JavaScript to send back to the client. This approach makes it easier to generate user-specific user interfaces
- **public** → Contains all static content such as images, CSS, and JavaScript
  - It is often helpful to have a sub-folder for each type of content
- **project** → Contains a few files: app.js, routes.js, and package.json
  - app.js → main configuration file for your application
  - routes.js → central location to access all the routes in the application. This file requires or imports all the files in the routes folder and then exports them as a single module which is imported into the app.js file. This helps create a single point of entry for all the routes in the application.

Best Practices for Building APIs

- Use **nouns** as resource identifiers in API routes
- Ensure correct usage of **HTTP status codes** to communicate the outcome of requests
- Example: **HTTP routes** for post, get, patch, and delete using the word employee in the route

HTTP method and route	Action
POST/employee or PUT/employee:id	Create a new employee
GET/employees	Retrieve a list of employees
GET/employee/:id	Retrieve an employee
PATCH/employee/:id	Modify an employee record
DELETE/employee/:id	Remove an employee

Best Practice For Testing & npm

- Use **Black-box** testing for node.js REST APIs
  - Black-box testing means you test the code **without** looking at its **internal structure**. Test the system as a **whole**
  - **Mocha** → A JavaScript test framework that runs on Node.js, contains a simple module called `superTest`
    - `SuperTest` → Provides a way to test HTTP requests, which is exactly what you need when you want to black-box test REST APIs
- Regarding the use of NPM
  - Use the `npm init` → To initialize a node.js project
  - When using the `npm install` command to install dependencies, use the `-save` or the `-save -dev` attribute
    - It makes sure that if the application is moved to a different platform, the correct dependencies will be installed with it

Naming description	Example
Use lowercase for file names	myfile.js
Use camel case for variables	myVariable
Use lowercase separated by dashes for npm modules	my-npm-module
Use camel case when requiring npm modules	require(‘myModule’)

Express Cheat Sheet

- express.listen()
- express.get()
- express.post()
- express.use()
- etc..