**NODE JS NOTES**

**WEEK 1**

**DAY1**

### Node JS:  
  As an asynchronous event driven JavaScript runtime, Node.js is designed to build scalable network applications.  
  In the context of Node.js, blocking and non-blocking refer to the behavior of I/O operations (like reading from a file or making a network request) in the event-driven,   
  asynchronous, and non-blocking I/O paradigm that Node.js is built upon.  
  \* Blocking I/O:  
In traditional synchronous programming, when an I/O operation is performed, the program execution is halted until the operation is completed. This means if you read a file or make a network request, the program will wait until the file is read or the request is completed before it can do anything else. This is called blocking I/O because it blocks the execution of the program until the I/O operation is finished. const fs = require('fs');  
const data = fs.readFileSync('file.txt'); // This is a blocking I/O operation  
console.log(data.toString());  
console.log('Program Ended'); \* Non-blocking I/O:  
In contrast, Node.js uses non-blocking I/O operations. When a non-blocking I/O operation is initiated, the program continues to execute the next line of code without waiting for the I/O operation to finish. A callback function is typically provided that gets executed once the I/O operation is completed. This approach allows Node.js to handle many connections concurrently. const fs = require('fs'); fs.readFile('file.txt', (err, data) => { // This is a non-blocking I/O operation  
    if (err) throw err;  
    console.log(data.toString());  
}); console.log('Program Ended'); In this example, the readFile function is non-blocking. It takes a callback function as an argument, which will be executed when the file reading operation is completed. The program continues executing the next line (console.log('Program Ended');) without waiting for the file reading operation to finish.  
The non-blocking nature of Node.js is one of its key features, enabling highly scalable applications where many I/O operations can be handled simultaneously without blocking the execution flow. It's essential to understand this asynchronous behavior when working with Node.js applications. Node.js takes the event model a bit further. It presents an event loop as a runtime construct instead of as a library. In other systems there is always a blocking call to start the event-loop. Typically behavior is defined through callbacks at the beginning of a script and at the end starts a server through a blocking call like EventMachine::run(). In Node.js there is no such start-the-event-loop call. Node.js simply enters the event loop after executing the input script. Node.js exits the event loop when there are no more callbacks to perform. This behavior is like browser JavaScript — the event loop is hidden from the user. ##  The "harmony" Runtime flag:   
The "Harmony" flag in the context of JavaScript refers to the set of experimental features and proposed ECMAScript (ES) standards that were in the process of being developed when ES6 (ECMAScript 2015) was under development.   
The "Harmony" flag in the context of JavaScript refers to the set of experimental features and proposed ECMAScript (ES) standards that were in the process of being developed when ES6 (ECMAScript 2015) was under development. In the early stages of development, many features that are now a standard part of JavaScript (such as arrow functions, classes, and modules) were marked as experimental and were not enabled by default in JavaScript engines to avoid breaking existing code. To experiment with these features, developers had to enable specific flags in JavaScript engines, and one of those flags was called the "Harmony" flag. For example, in Node.js, you could enable experimental features using the --harmony flag when running your scripts. An example command might look like this:  
node --harmony myscript.js   
node.green website:  provides an excellent overview over supported ECMAScript features in various versions of Node.js, based on kangax's compat-table. Summary:  \* 1. Node JS is constantly upgraded, meaning you can do a lot of cool things like creating your own web server, insisting on cool game features, and efficiency in JS code. Node.js is designed to build scalable network applications.  
\* 2. You use the "command line interface" to access it. That's where node.js files are indicated. eg. cmd, terminal, powershell etc.   
\* 3.  We use it to create server-side applications.   
\* 4. Modules in NodeJS are the same as JavaScript libraries.  
\* 5. NPM: Node Package Manager; helps install packages and libraries within NodeJS.   
\* 6. It contains all the files you need, for a package.

**DAY 2**

**Node.js Libraries**

Often referred to simply as “node.js module” or “node.js package is a collection of reusable js code that can be easily incorporated into your node.js application.

The purpose of the libraries is to provide specific functionality or solve common programming problems or tasks, allowing developers to save time and effort by not having to reinvent the wheel.

Node.js libraries are typically distributed through the node package manager(npm) and can be installed using the npm command line tool.

**Types of libraries.**

**Express**

A minimalist web app framework that simplifies the process of building web applications and APIs. Express.js provides a set of powerful tools and middleware to handle routing, HTTP requests, and responses.

Install this library using the following syntax.

npm install express.

Now, let's create a simple Express.js server:

const express = require('express');

const app = express();

const PORT = 3000;

app.get('/', (req, res) => {​​​​​​​​

res.send('Hello, world!');}​​​​​​​​);

app.listen(PORT, () => {​​​​​​​​

console.log(`Server is running on port ${​​​​​​​​PORT}​​​​​​​​`);

}​​​​​​​​);

**Lodash Lodash** is a modern JavaScript utility library providing utility functions for common programming tasks. It simplifies working with arrays, strings, objects, numbers, and more.

To install Lodash, run:

npm install lodash Here's an example of using Lodash to find the intersection of two arrays:

const \_ = require('lodash');

const array1 = [1, 2, 3, 4]; const array2 = [3, 4, 5, 6];

const intersection = \_.intersection(array1, array2); console.log(intersection); // Output: [3, 4]

**Mongoose**

 Mongoose is an Object Data Modeling (ODM) library for MongoDB and Node.js. It provides a simple, schema-based solution to model application data and manage database operations.

Install Mongoose using:

npm install mongoose Here's an example of defining a schema and connecting to MongoDB using Mongoose:

const mongoose = require('mongoose');

mongoose.connect('mongodb://localhost/test', {​​​​​​​   useNewUrlParser: true,   useUnifiedTopology: true, }​​​​​​​);

const userSchema = new mongoose.Schema({​​​​​​​   name: String,   age: Number,   email: String, }​​​​​​​);

const User = mongoose.model('User', userSchema);

const user = new User({​​​​​​​   name: 'John Doe',   age: 30,   email: 'john@example.com', }​​​​​​​);

user.save().then(() => console.log('User saved successfully'));

**Passport.js**is a popular authentication middleware for Node.js that supports various authentication strategies, including OAuth, OpenID, and more.

To install Passport.js, run:

npm install passport Here's an example of using Passport.js with a local authentication strategy:

const express = require('express'); const passport = require('passport'); const LocalStrategy = require('passport-local').Strategy;

const app = express();

passport.use(   new LocalStrategy((username, password, done) => {​​​​​​​     // Replace with your own authentication logic     if (username === 'admin' && password === 'password') {​​​​​​​       return done(null, {​​​​​​​ id: 1, username: 'admin' }​​​​​​​);     }​​​​​​​     return done(null, false);   }​​​​​​​) );

app.use(passport.initialize()); app.use(passport.session());

app.post('/login', passport.authenticate('local'), (req, res), () => {​​​​​​​   res.send('Logged in!'); }​​​​​​​);

app.listen(3000, () => {​​​​​​​   console.log('Server is running on port 3000'); }​​​​​​​);

**Socket.IO** is a real-time communication library for Node.js that enables bidirectional event-based communication between the server and the client. It is perfect for chat applications, real-time analytics, and more.

To install Socket.IO, run:

npm install socket.io Here's an example of a simple chat server using Socket.IO:

const express = require('express'); const http = require('http'); const socketIO = require('socket.io');

const app = express(); const server = http.createServer(app); const io = socketIO(server);

io.on('connection', (socket) => {​​​​​​​   console.log('A user connected');

  socket.on('message', (msg) => {​​​​​​​     io.emit('message', msg);   }​​​​​​​);

  socket.on('disconnect', () => {​​​​​​​     console.log('A user disconnected');   }​​​​​​​); }​​​​​​​);

server.listen(3000, () => {​​​​​​​   console.log('Server is running on port 3000'); }​​​​​​​);

**Async**

 Async is a powerful utility module for working with asynchronous JavaScript. It provides a collection of higher-order functions for managing control flow and handling asynchronous operations.

To install Async, run:

npm install async Here's an example of using Async's waterfall function to execute multiple asynchronous functions in sequence:

 const async = require('async');

async.waterfall(

  [

    (callback) => {

      setTimeout(() => {

        console.log('Task 1');

        callback(null, 'Task 1 result');

      }, 1000);

  },

    (task1Result, callback) => {

      setTimeout(() => {

        console.log('Task 2');

        callback(null, 'Task 2 result');

      }, 1000);

    },

  ],

(err, result) => {

    if (err) {

      console.error(err);

    } else {

      console.log('All tasks completed:', result);

    }

  }

);

**Request** is a popular HTTP client library for Node.js that simplifies making HTTP requests. It supports various features like OAuth signing, streaming, and more.

Note: The Request library has been deprecated, but you can still use it or switch to alternatives like Axios, Got, or Node-fetch.

To install Request, run:

npm install request Here's an example of making a GET request using the Request library:

const request = require('request');

request('https://api.example.com/data', (error, response, body) => {​​​​​​​   if (!error && response.statusCode === 200) {​​​​​​​     console.log(body);   }​​​​​​​ }​​​​​​​);

**Winston** is a versatile logging library for Node.js, designed to be flexible and extensible. It supports multiple transports, which means you can log messages to various destinations, such as console, files, or remote services.

To install Winston, run:

npm install winston Here's an example of basic logging using Winston:

const winston = require('winston');

const logger = winston.createLogger({​​​​​​​   level: 'info',   format: winston.format.simple(),   transports: [     new winston.transports.Console(),     new winston.transports.File({​​​​​​​ filename: 'combined.log' }​​​​​​​),   ], }​​​​​​​);

logger.info('Hello, Winston!');

 9. Nodemon Nodemon is a utility that monitors for changes in your source code and automatically restarts your Node.js application during development. This saves you the hassle of manually restarting the server every time you make changes.

To install Nodemon globally, run:

npm install -g nodemon Now, you can start your Node.js application with Nodemon using the following command:

nodemon app.js

10. dotenv dotenv is a zero-dependency module that loads environment variables from a .env file into process.env. Storing configuration in the environment is a best practice, as it keeps sensitive data like API keys and passwords out of your source code.

To install dotenv, run:

npm install dotenv Create a .env file in your project's root directory:

API\_KEY=myapikey SECRET=mysecret Now, load the environment variables in your Node.js application:

require('dotenv').config();

console.log('API Key:', process.env.API\_KEY); console.log('Secret:', process.env.SECRET

**DAY 3**

**Daily Notes - Sprites**

 In JavaScript, "sprites" typically refer to graphical objects or images that can be manipulated and displayed on a web page or within a game. Sprites are often used in 2D game development and interactive web applications to represent characters, objects, or other elements that can move around the screen.

 Activity 1 - Applying new Concepts

Okay in activity 1 it's where we focused more on resizing the pictures for our game, the asphalt(road) and the car ,defined the default background color for our game’s canvas. And we also created the client folder where all of these sub folders were created.

**Activity 2 - Inside the client folder**

 Client folder,the code handles the updating of player information in a multiplayer game, ensuring that players are correctly created, updated, and deleted based on data received from the server. It manages the state of other players in the game and keeps the client's representation of the game world in sync with the server's data.

**Activity 3 - Code Snippet**

 This code is likely used to control the rotation of a game object or sprite based on user input from the 'A' and 'D' keys, with the direction and speed of rotation determined by the state of these keys and the this.speed property. The code snippet suggests that the game object has a physics body (this.sprite.body) that can be manipulated to control its angular velocity (rotation speed).

**My Views on the Day**

 All the content of the day were important   
All the activities were beneficial   
None of the activities was too easy   
All the activities need time

**Daily Notes - Day 3 Reflections**

 Sprites are a fundamental element in game development and play a crucial role in creating visually engaging 2D and sometimes even 3D games. Here are some key uses and benefits of sprites in game development:   
   
Character and Object Representation   
Animation   
Efficiency   
Versatility   
Retro and Indie Games   
Efficient Memory Usage   
Customization

**DAY 4**

**Activity 1 - Assigning a Sprite to a player**

Assigning a sprite to a player in a game typically involves associating a visual representation (the sprite) with a game object or character (the player). The specific steps to assign a sprite to a player can vary depending on the game development platform or engine you're using, but I'll provide a general outline of the process:

**Create or Import the Sprite:**

You need to have a sprite for your player character. This can be created using graphic design software or obtained from a library of pre-made sprites. Import the sprite into your game development environment.

**Create a Player Object:**

In your game development environment, create an object or character that represents the player. This is typically done by creating a game object or entity and defining its properties and behaviours.

**Assign the Sprite to the Player Object:**

Attach the sprite you imported in step 1 to the player object. The specific way to do this varies by game engine, but it usually involves selecting the sprite as a visual component of the player object.

**Define Sprite Animations (if needed):**

If your player character has animations (e.g., walking, running, jumping), you'll need to define these animations in your game development environment. This may involve creating a sprite sheet with multiple frames for each animation.

**Control the Player Object:**

Implement the game logic and controls for the player object. This includes code for movement, interaction, and any other player-specific behaviours.

**Display the Player Object:**

In the game's rendering loop, display the player object with its assigned sprite. This typically involves rendering the sprite at the player's current position in the game world.

**Activity 2 - Managing the game server**

Managing a game server involves various tasks and responsibilities to ensure that the server runs smoothly, provides a good gaming experience for players, and remains secure. The specific steps and processes may vary depending on the game, hosting platform, and your server's requirements, but here are some general guidelines for managing a game server:

**Choose the Right Hosting Provider:**

Select a hosting provider that offers reliable server hosting services tailored to your game's requirements.

Consider factors like server location, server performance, scalability, and customer support.

**Server Configuration:**

Configure the server hardware and software to meet the needs of your game. This includes selecting the right server specifications, operating system, and game server software.

**Install and Update Game Server Software:**

Install the game server software on your server, ensuring it's the latest version.

Regularly update the game server software to patch security vulnerabilities and add new features.

**Manage Server Files and Mods:**

Organize game files and mods if applicable. Many games support mods and custom content, so ensure they are properly installed and managed.

**Monitor Server Performance:**

Implement server monitoring tools to keep an eye on server performance, such as CPU and memory usage, network traffic, and latency.

**Server Security:**

Implement security measures to protect your game server from unauthorized access, DDoS attacks, and cheating.

Regularly update and patch the server and its dependencies.

Use strong authentication methods and enforce password policies.

**Backup and Recovery:**

Regularly back up game data and server configurations to prevent data loss.

Establish a disaster recovery plan to quickly restore the server in case of a catastrophic failure.