Introduction to NodeJS,
https://youtu.be/OJZRQQ9ip9g

Introduction to Node.JS

Node.js is a powerful and flexible serverside platform built on Google Chrome's V8 JavaScript engine. It's designed to build scalable network applications.

Overview

Definition: Node.js is an opensource, crossplatform runtime environment for executing JavaScript code serverside.

Purpose: Primarily used to build network programs such as web servers, making it a good choice for many types of projects.

Advantages:

Nonblocking, eventdriven architecture ensures optimal throughput and scaling.

Uses JavaScript, making it easy to learn for those familiar with the language.

Rich ecosystem powered by npm, providing access to thousands of packages.

Architecture

Eventdriven: Everything that happens in Node.js is in reaction to an event, which enables high scalability.

Singlethreaded: Utilizes a singlethreaded model with event looping. This event mechanism helps the server to respond in a nonblocking way.

Nonblocking I/O operations: This allows Node.js to handle thousands of concurrent connections, resulting in high performance.

Key Features

Asynchronous and Event Driven: All APIs of the Node.js library are asynchronous, i.e., nonblocking. It essentially means a Node.js based server never waits for an API to return data.

Fast Execution: Node.js uses the V8 JavaScript Engine which makes it very fast in code execution.

Single Programming Language: You can write both clientside and serverside code in JavaScript using Node.js.

Large Ecosystem: npm, the Node.js package ecosystem is the largest ecosystem of open source libraries.

Core Modules

Node.js includes a set of builtin modules that you can use without any further installation:

HTTP: Launch a server, send requests.

URL: Parse URL strings.

File System (fs): Interact with the file system.

Events: Handle events.

Stream: Handle streaming data.

Others: `os`, `path`, `querystring`, etc.

npm Node Package Manager

Overview: npm is the world s largest software registry. Opensource developers use npm to share software, and many organizations use it to manage private development.

Features:

Over 800,000 libraries available.

Facilitates version control and dependency management.

Common Use Cases

Web servers and RESTful APIs

Realtime services like Chat applications and live updates

Complex singlepage applications (SPAs)

I/O bound Applications

Data streaming applications

Utility scripts

Setting up a Node.js Project

1. Installation: Download and install Node.js from the official website.
2. Creating a Project:

   Use `npm init` to create a `package.json` file which holds metadata relevant to the project.

   Install packages using `npm install <package_name>`.
3. Development:

   Write your JavaScript code in files such as `app.js`.

   Use `node app.js` to run your application.

Best Practices

Use asynchronous code wherever possible.

Handle all errors in callbacks and promises.

Stick to the small module principle: keep modules small and focused on a single responsibility.

Test your code thoroughly.

Keep your codebase readable and maintainable with proper documentation.

Deploying to Node, NPM,
https://youtu.be/0N-Pzclf6Zc

Deploying to Node, NPM

Deploying a React.js application involves several steps, including setting up a build process, configuring a server, and deploying your application to a hosting provider.

1. Prepare Your React.js Application:

Ensure that your React.js application is properly structured with components, styles, and any necessary assets.

Make sure you have a package.json file in the root directory of your project. If not, create one by running npm init and following the prompts.

2. Install Necessary Dependencies:

Ensure you have Node.js and npm installed on your development machine.

Install React and other dependencies required for your project using npm. For example:

```
npm install react reactdom
```

Additionally, install any other dependencies your project requires, like reactrouterdom, axios, etc., and add them to your package.json file.

3. Configure Your Build Process:

Set up a build process to compile your React code into a format suitable for deployment. Most React projects use tools like Webpack or Create React App for this purpose.

If you're using Create React App, building your project for production is as simple as running:

```
npm run build
```

This command creates an optimized production build of your app in the build folder.

4. Set Up a Node.js Server:

You'll need a server to host your React application. Node.js is a popular choice for this purpose due to its flexibility and ease of use.

Create a new file (e.g., server.js) in the root directory of your project to define your server logic.

Use Express.js, a minimal and flexible Node.js web application framework, to set up a basic server. Install it using npm:

```
npm install express
```

In server.js, set up a basic Express server to serve your static React files:

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

app.use(express.static(path.join(__dirname, 'build')));

app.get('/', function (req, res) {
  res.sendFile(path.join(__dirname, 'build', 'index.html'));
});

const port = process.env.PORT || 3000;
app.listen(port, () => {
  console.log(Server is running on port ${port});
});
```

5. Deploy Your Application:

Choose a hosting provider for your application. Popular options include Heroku, Netlify, Vercel, AWS, and DigitalOcean.

Follow the specific instructions provided by your hosting provider to deploy your application. Typically, this involves creating an account, setting up a new project, and linking it to your Git repository.

If you're deploying to Heroku, for instance, you'll need to:

Install the Heroku CLI.

Log in to your Heroku account using the CLI.

Navigate to your project directory in the terminal.

Initialize a new Git repository if you haven't already (git init).

Add your Heroku remote (heroku git:remote a yourappname).

Deploy your application by pushing your code to the Heroku remote (git push heroku master).

6. Monitor Your Deployment:

After deploying your application, it's essential to monitor its performance and stability.

Set up logging and error tracking to identify and troubleshoot any issues that may arise.

Use tools like New Relic, Sentry, or LogRocket to monitor your application's performance and user experience.

7. Continuous Deployment (Optional):

To streamline your deployment process, consider setting up continuous integration and continuous deployment (CI/CD) pipelines.

CI/CD pipelines automate the build, test, and deployment processes, allowing you to deploy changes to your application quickly and reliably.

By following these steps, you can successfully deploy your React.js application using Node.js and npm. Remember to test your deployment thoroughly to ensure everything is working as expected before making your application live.

1. What is npm?

npm is a package manager for JavaScript, meaning it helps you manage project dependencies and share JavaScript modules with other developers. It's also used to manage private packages and adapt packages for your own projects.

2. npm Registry

The npm registry is an online database of public and private packages, which are reusable pieces of JavaScript code. This registry allows developers to upload their packages for others to use, and to download packages that others have created.

3. npm CLI

npm provides a command line interface (CLI) that lets you interact with the npm registry. Using the CLI, you can install, update, and manage packages in your projects. Common commands include:

npm install <package>: Installs a package.

npm update <package>: Updates a package to the latest version.

npm uninstall <package>: Removes a package from your project.

4. package.json

The package.json file is a key part of any project using npm. It holds metadata relevant to the project and manages project dependencies. This file specifies the packages your project depends on and their versions, ensuring consistency across environments.

5. Node_modules

When you install packages via npm, they are stored in the node_modules directory within your project. This folder contains all the packages (and their dependencies) that are listed in your package.json file.

6. Semantic Versioning

npm uses semantic versioning (semver) for packages. This versioning scheme has three parts: major, minor, and patch (e.g., 1.0.2). It helps manage dependencies effectively, specifying which versions are compatible and how updates affect compatibility:

Major version changes indicate breaking changes.

Minor version updates introduce new features that are backward compatible.

Patch versions are for bug fixes and minor changes that don't affect compatibility.

7. npm Scripts

npm allows you to define scripts in your package.json file, which you can execute with npm commands. These scripts can automate common tasks such as testing, building, and deploying applications.

8. npm and Node.js

While npm is closely associated with Node.js, it's also used in frontend JavaScript development. It integrates well with build tools like Webpack and Babel, enhancing JavaScript processing and application bundling.

9. Security

npm includes features to help secure JavaScript projects. It can audit packages for security vulnerabilities and automatically update packages to secure versions.

10. npm Inc.

npm is also a company: npm, Inc., which manages the npm project and the npm registry. It provides additional services like npm Pro, npm Teams, and npm Enterprise, which offer features like private packages and improved team collaboration tools.

Introduction to ExpressJS,
https://youtu.be/SSvVP_8lIP8

Express JS

Express.js, commonly referred to as Express, is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications. It's an essential part of the MEAN (MongoDB, Express, Angular, Node.js) and MERN (MongoDB, Express, React, Node.js) stack environments, facilitating the development of serverside logic and APIs.

Key Features of Express.js

1. Simplicity: Express simplifies the process of building serverside applications with Node.js. It handles the complexities of the underlying HTTP server, allowing you to focus on writing your application's specific logic.

2. Middleware: Middleware functions are functions that have access to the request object (req), the response object (res), and the next middleware function in the application’s requestresponse cycle. These functions can execute any code, make changes to the request and response objects, end the requestresponse cycle, and call the next middleware function. This architecture is useful for executing code, making modifications to the request and response objects, ending the requestresponse cycle, and adding functionality to the Express app.

3. Routing: Express provides a sophisticated router that allows you to manage different HTTP routes via a simple and intuitive API. You can define routes based on URL patterns and HTTP methods, and implement them using callbacks and middleware strategies.

4. Performance: Express is designed for performance. It provides mechanisms for handling HTTP requests in a fast manner, which makes it an ideal choice for building efficient applications.

5. Scalability: With its lightweight nature and middleware capabilities, Express can handle a large number of simultaneous connections, making it suitable for high traffic applications.

6. Community and Ecosystem: Being one of the most popular Node.js frameworks, Express has a large community of developers and a vast ecosystem of middleware and plugins that can be integrated to extend its capabilities.

Basic Concepts

Application: The central part of an Express app. It's created by calling express() and is used to configure the server, by registering routes, middleware, and starting the server.

Request Objects: Represent the HTTP request and have properties for the request query string, parameters, body, HTTP headers, etc.

Response Objects: Represent the HTTP response that an Express app sends when it gets an HTTP request.

Routes: Define the control flow for different endpoints and HTTP methods. They match incoming requests and delegate control to the appropriate function.

Example Usage

Here's a basic example of a simple Express server:

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

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

app.listen(3000, () => {
```

```
  console.log('Server is running on http://localhost:3000');
});
```

This server listens on port 3000 and will respond with "Hello World!" for requests to the root URL.

Conclusion

Express.js is highly regarded for its performance and flexibility in building web applications, particularly APIs. It's minimalistic, yet powerful, providing just the right amount of functionality needed to make serverside development easier and more enjoyable. As a React.js professional, integrating Express into your fullstack projects can facilitate better control over the serverside logic and data handling, enhancing your React applications.

Life cycle and routing of Express App, deploying to Node, https://youtu.be/T7Eezc5m2UQ

Life Cycle of an Express And Deploying it to the Node

1. Initialization:

   Import Modules: Import required modules using require().

   javascript

   ```
   const express = require('express');
   const bodyParser = require('bodyparser');
   const mongoose = require('mongoose');
   ```

   Create App Instance: Create an instance of the Express application.

   javascript

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

2. Middleware Setup:

   ApplicationLevel Middleware: Apply middleware using app.use().

   javascript

   ```
   app.use(bodyParser.json());
   app.use(express.static('public'));
   ```

   RouterLevel Middleware: Apply middleware to specific routers.

   javascript

   ```
   const router = express.Router();
   router.use((req, res, next) => {
     console.log('Request URL:', req.originalUrl);
     next();
   });
   ```

3. Routing:

Define Routes: Use app.get(), app.post(), etc., to define routes.

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

4. Error Handling:

ErrorHandling Middleware: Define middleware for handling errors.

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

5. Server Setup:

Listen for Requests: Start the server using app.listen().

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

Routing in Express

1. Basic Routing:

GET Route: Define a route for GET requests.

```
javascript
app.get('/users', (req, res) => {
  res.send('GET request to the users page');
```

```
});
```

POST Route: Define a route for POST requests.

```
javascript
app.post('/users', (req, res) => {
  res.send('POST request to the users page');
});
```

2. Route Parameters:

Defining Route Parameters: Use colon : to define route parameters.

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

3. Query Parameters:

Accessing Query Parameters: Use req.query to access query parameters.

```
javascript
app.get('/search', (req, res) => {
  res.send(Search Query: ${req.query.q});
});
```

4. Router Object:

Create Router: Create an instance of the router and define routes.

```
javascript
const router = express.Router();
router.get('/profile', (req, res) => {
  res.send('User profile');
```

```
});
app.use('/user', router);
```

5. Nested Routes:

Defining Nested Routes: Nest routes within a parent router.

```
javascript
router.get('/:userId/posts', (req, res) => {
  res.send(Posts for user ${req.params.userId});
});
```

Deploying an Express App to Node

1. Preparation:

Ensure Environment Variables: Set environment variables for configuration.

```
export PORT=3000
export NODE_ENV=production
```

Install Dependencies: Ensure all dependencies are installed.

```
npm install
```

2. Create a Production Build:

Transpile Code (if using TypeScript or Babel): Transpile code if necessary.

```
npm run build
```

3. Configure Server:

Environment Configuration: Configure server settings based on environment.

javascript

```
const PORT = process.env.PORT || 3000;
```

4. Deploy:

Choose a Hosting Provider: Choose a hosting provider like Heroku, AWS, or DigitalOcean.

Deploy to Heroku:

Create a Procfile:

```
web: node server.js
```

Initialize Git and Deploy:

```
git init
heroku create
git add .
git commit m "Initial commit"
git push heroku master
```

Deploy to AWS EC2:

Set Up EC2 Instance: Launch an EC2 instance and SSH into it.

Install Node.js and Git:

```
sudo aptget update
sudo aptget install nodejs
sudo aptget install git
```

Clone Repository and Install Dependencies:

```
git clone <repositoryurl>
cd <repository>
npm install
```

Start Application:

```
node server.js
```

Deploy to DigitalOcean:

Set Up Droplet: Create a droplet and SSH into it.

Install Node.js and Git:

```
sudo aptget update
sudo aptget install nodejs
sudo aptget install git
```

Clone Repository and Install Dependencies:

```
git clone <repositoryurl>
cd <repository>
npm install
```

Start Application:

```
node server.js
```

Handling request and response parameters, https://youtu.be/RLUGZeblo5M

Handling Request and Response Parameters in Express.js

Express.js is a web application framework for Node.js, designed for building web applications and APIs. Handling request and response parameters efficiently is crucial for developing robust applications. Below are key concepts and methods for managing these parameters.

1. Request Parameters

Request parameters are pieces of data sent by the client to the server. They can be extracted from different parts of the request, such as the URL, query string, body, or headers.

a. URL Parameters

Definition: Parts of the URL that capture values at specified positions.

Usage: Often used for identifying resources.

Syntax: Defined in the route with a colon (:) prefix.

Example:

```
js
// Route definition
app.get('/user/:id', (req, res) => {
  // Accessing URL parameter
  const userId = req.params.id;
  res.send(User ID is ${userId});
});
```

In this example, if the client requests /user/123, the userId would be 123.

b. Query Parameters

Definition: Keyvalue pairs in the URL after the ?.

Usage: Used to filter or sort data.

Syntax: Appended to the URL with ? and separated by &.

Example:

```
js
// Route definition
app.get('/search', (req, res) => {
  // Accessing query parameters
  const term = req.query.term;
  res.send(Search term is ${term});
});
```

In this example, if the client requests /search?term=express, the term would be express.

c. Body Parameters

Definition: Data sent by the client in the body of the request, often in POST or PUT requests.

Usage: Used for creating or updating resources.

Middleware: Bodyparsing middleware like express.json() or express.urlencoded() is needed to parse the body.

Example:

```
js
// Middleware to parse JSON body
app.use(express.json());

// Route definition
app.post('/login', (req, res) => {
  // Accessing body parameters
  const username = req.body.username;
  const password = req.body.password;
  res.send(Username is ${username} and Password is ${password});
});
```

In this example, the body of the request might be {"username":"user1", "password":"pass1"}.

d. Headers

Definition: Metadata sent with the request.

Usage: For authentication tokens, content type, etc.

Example:

```
js
// Route definition
app.get('/headers', (req, res) => {
  // Accessing headers
  const contentType = req.headers['contenttype'];
  res.send(ContentType is ${contentType});
});
```

In this example, if the client sets the header ContentType: application/json, it can be accessed via req.headers['contenttype'].

2. Response Parameters

Response parameters are sent by the server to the client. Express.js provides various methods to set these parameters.

a. Status Codes

Definition: Numeric codes indicating the result of the request.

Usage: To indicate success, redirection, client errors, or server errors.

Example:

```
js
// Route definition
app.get('/success', (req, res) => {
  res.status(200).send('Success');
```

```
});
app.get('/notfound', (req, res) => {
  res.status(404).send('Not Found');
});
```

In this example, 200 indicates success, and 404 indicates the resource was not found.

b. Response Body

Definition: The main content returned by the server.

Methods:

- res.send(): Sends a response of various types.
- res.json(): Sends a JSON response.
- res.sendFile(): Sends a file.

Example:

```
js
// Sending plain text
app.get('/text', (req, res) => {
  res.send('Hello World');
});

// Sending JSON
app.get('/json', (req, res) => {
  res.json({ message: 'Hello World' });
});
```

c. Headers

Definition: Additional metadata sent with the response.

Methods:

res.set(): Sets a response header.

res.type(): Sets the ContentType header.

Example:

```
js
// Setting headers
app.get('/setheader', (req, res) => {
  res.set('CustomHeader', 'CustomValue');
  res.send('Header Set');
});

// Setting content type
app.get('/settype', (req, res) => {
  res.type('text/plain');
  res.send('Plain Text Response');
});
```

3. Middleware for Handling Parameters

Middleware functions are used to process request and response objects. They can modify request and response objects, end the requestresponse cycle, or call the next middleware.

Example:

```
js
// Middleware function to log request parameters
app.use((req, res, next) => {
  console.log('Request URL:', req.originalUrl);
  console.log('Request Method:', req.method);
  next();
});
```

```
// Route definition
app.get('/', (req, res) => {
  res.send('Middleware logged the request');
});
```

Conclusion

Handling request and response parameters efficiently in Express.js involves understanding how to extract and manipulate different types of parameters. By leveraging the flexibility of Express.js and using appropriate middleware, you can build powerful and scalable web applications.

Reading configuration parameters.,
https://youtu.be/_c2sz8cLyhw

Reading Configuration Parameters in Express.js

When developing applications with Express.js, managing configuration parameters is crucial for maintaining flexibility and security. Configuration parameters can include things like database connection strings, API keys, or server settings. Here's a detailed guide on how to read and manage configuration parameters in an Express.js application.

1. Environment Variables

Environment variables are a standard way to manage configuration outside of your codebase, which enhances security and flexibility. You can use the dotenv package to load environment variables from a .env file into process.env.

Setup dotenv:

1. Install the dotenv package:

```
sh
npm install dotenv
```

2. Create a .env file in the root of your project:

```
env
PORT=3000
DATABASE_URL=mongodb://localhost:27017/mydatabase
API_KEY=yourapikey
```

3. Load environment variables in your application:

```
js
require('dotenv').config();

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

const port = process.env.PORT || 3000;
const databaseUrl = process.env.DATABASE_URL;
const apiKey = process.env.API_KEY;
```

```
app.listen(port, () => {
    console.log(Server is running on port ${port});
    console.log(Connected to database at ${databaseUrl});
});
```

2. Configuration Files

For more structured configurations, you might use a dedicated configuration file. This approach is useful when you have different configurations for development, testing, and production environments.

Using Configuration Files:

1. Create a config directory with files like default.json, production.json, and development.json.

```
json
// config/default.json
{
  "port": 3000,
  "databaseUrl": "mongodb://localhost:27017/mydatabase",
  "apiKey": "yourapikey"
}
```

2. Install the config package:

```
sh
npm install config
```

3. Load configuration in your application:

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

```
const port = config.get('port');
const databaseUrl = config.get('databaseUrl');
const apiKey = config.get('apiKey');

app.listen(port, () => {
   console.log(Server is running on port ${port});
   console.log(Connected to database at ${databaseUrl});
});
```

3. CommandLine Arguments

You can also pass configuration parameters via commandline arguments using process.argv.

Using CommandLine Arguments:

1. Start your application with arguments:

```
sh
node app.js port=3000 databaseUrl=mongodb://localhost:27017/mydatabase apiKey=yourapikey
```

2. Parse commandline arguments in your application:

```
js
const express = require('express');
const app = express();
const yargs = require('yargs/yargs');
const { hideBin } = require('yargs/helpers');
const argv = yargs(hideBin(process.argv)).argv;

const port = argv.port || 3000;
const databaseUrl = argv.databaseUrl;
const apiKey = argv.apiKey;

app.listen(port, () => {
```

```
    console.log(Server is running on port ${port});
    console.log(Connected to database at ${databaseUrl});
});
```

4. Summary

Environment Variables: Use the dotenv package to load configurations from a .env file.

Configuration Files: Use the config package to manage different configurations for various environments.

CommandLine Arguments: Use process.argv and libraries like yargs to parse commandline arguments.

Each method has its use case and can be combined for a robust configuration management system in your Express.js application. By separating configuration from your code, you enhance the maintainability and security of your application.

Lab Programs ,

https://youtu.be/cLk2BoTvx-c

Lab programs

/*4. Write a NodeJS program that accepts a Port from the user and runs a Node server at the port */

```
// Importing the required modules
const express = require('express');
const app = express();

// Getting the port from command-line arguments
const port = process.argv[2] || 3000; // Default to port 3000 if no port is provided

// Define a simple route
app.get('/', (req, res) => {
    res.send('Hello, world!');
});

// Start the server and listen on the specified port
app.listen(port, () => {
    console.log(`Server is running on port ${port}`);
});
```

to run:

node server.js 8080

/5.*Write a Node js program that reads a file and display the content on the screen */

```
// Importing the required modules
const fs = require('fs');
const path = require('path');

// Getting the file path from command-line arguments
const filePath = process.argv[2];

// Check if the file path is provided
if (!filePath) {
    console.error('Please provide a file path as a command-line argument.');
    process.exit(1);
}

// Resolve the file path to handle relative paths correctly
const resolvedPath = path.resolve(filePath);

// Read the file asynchronously
fs.readFile(resolvedPath, 'utf8', (err, data) => {
    if (err) {
        console.error(`Error reading file: ${err.message}`);
        process.exit(1);
    }
    // Display the content on the screen
```

```
    console.log(`Content of the file (${resolvedPath}):\n`);
    console.log(data);
});
```

How to run :

node server.js filepath/file.txt

/5.*Write a Node js program that reads a file and display the content on the screen(express) */

```
// Importing the required modules
const express = require('express');
const fs = require('fs');
const path = require('path');

// Create an instance of an Express application
const app = express();

// Getting the port and file path from command-line arguments
const port = process.argv[2] || 3000;
const filePath = process.argv[3];

// Check if the file path is provided
if (!filePath) {
    console.error('Please provide a file path as a command-line argument.');
    process.exit(1);
```

```
}

// Define a route to serve the file content
app.get('/', (req, res) => {
    // Resolve the file path to handle relative paths correctly
    const resolvedPath = path.resolve(filePath);

    // Read the file asynchronously
    fs.readFile(resolvedPath, 'utf8', (err, data) => {
        if (err) {
            console.error(`Error reading file: ${err.message}`);
            res.status(500).send('Error reading file');
            return;
        }
        // Send the content of the file as the response
        res.send(`<pre>${data}</pre>`);
    });
});

// Start the server and listen on the specified port
app.listen(port, () => {
    console.log(`Server is running on port ${port}`);
});

node server.js 8080 filepath/file.txt
```