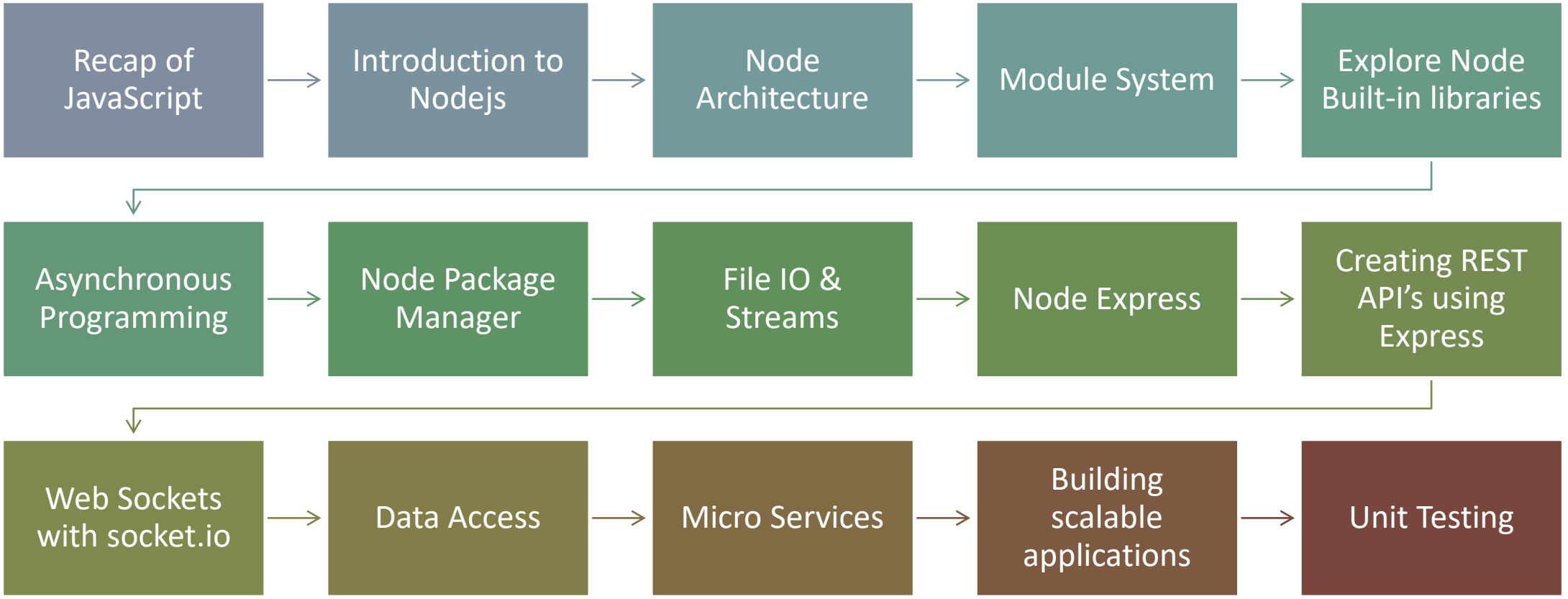
Node.js

ANIL JOSEPH

Timmins Training Consulting

Agenda


Recap of JavaScript
Introduction to Nodejs
Node Architecture
Module System
Explore Node Built-in libraries
Asynchronous Programming
Node Package Manager
File IO & Streams
Node Express
Creating REST API's using Express
Web Sockets with socket.io
Data Access
Micro Services
Building scalable applications
Unit Testing

Http Module

The HTTP module provides the ability to create HTTP servers and clients,

It’s the essential module for building web applications and services.

Key Features

- Creating an HTTP Server
- Handling HTTP Requests and Responses
- Creating an HTTP Client

This is a low-level API over which we can build a web framework

Express.js

Express.js is a minimal and flexible Node.js web application framework.

Provides a robust set of features for web and mobile applications.

Designed to make it easy to build web applications and APIs.

Key Features of Express.js:

- Middleware
  - Intercepts the request and response objects.
- Routing
  - Express provides a robust routing mechanism to handle different HTTP methods
- Template Engines
  - Express supports various template engines like Pug, EJS, and Handlebars, which allow you to generate HTML dynamically.
- Error Handling
  - Simple mechanisms to handle errors in your application.
- Static Files
  - Express makes it easy to serve static files, such as images, CSS, and JavaScript files, using built-in middleware.

Middlewares

- Intercept the request and response before being processed by the handler.
- Middleware can execute code, modify the request and response objects, end the request-response cycle, and call the next middleware function.

Types of Middleware

- Application-Level Middleware:
  - Defined directly on the app object using app.use or app.METHOD.
  - Can be executed for every request or for specific routes.
- Router-Level Middleware:
  - Similar to application-level middleware but applied to instances of express.Router().
  - Useful for modularizing and organizing route-specific middleware.
- Error-Handling Middleware:
  - Special type of middleware that handles errors in the application.
  - Defined with four arguments: err, req, res, and next.
- Built-In Middleware:
  - Provided by Express.js,
  - Examples: express.static for serving static files, express.json for parsing JSON bodies, and express.urlencoded for parsing URL-encoded bodies.

Template Engines

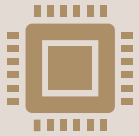
- Template engines in web development are tools that allow you to create HTML pages dynamically by combining static templates with dynamic data.
- Benefits
  - Separation of Concerns
  - Reusability
  - Easy Maintenance
  - Simplified Development

Template Engines

- EJS (Embedded JavaScript)
  - EJS is a simple and popular template engine that uses plain JavaScript to generate HTML. It allows embedding JavaScript code directly within the template.
- Pug(Jade)
  - Pug is known for its clean, whitespace-sensitive syntax. It uses indentation to define HTML structure, making the templates concise.
- Handlebars
  - Handlebars is an extension of Mustache with a focus on keeping the templates logic-less. It uses a double curly brace syntax for placeholders.

REST API

A REST (Representational State Transfer) API is a set of rules and conventions for building and interacting with web services.



REST APIs allow different software systems to communicate over the internet using standard HTTP methods.

REST is an architectural style, not a protocol or standard, and it leverages the principles of the web.

Key Principles of REST

Stateless	Client Server	Uniform Interface	Resource-based	Representation	Cacheable
• The server does not store any client context between requests.	• The client and server are separate entities.	• A consistent and standardized way to interact with resources.	• Everything in a RESTful system is considered a resource, which can be accessed and manipulated using URLs.	• Resources can have multiple representations	• Responses must define themselves as cacheable or non-cacheable

HTTP Methods in REST

GET: Retrieve a resource or a collection of resources.

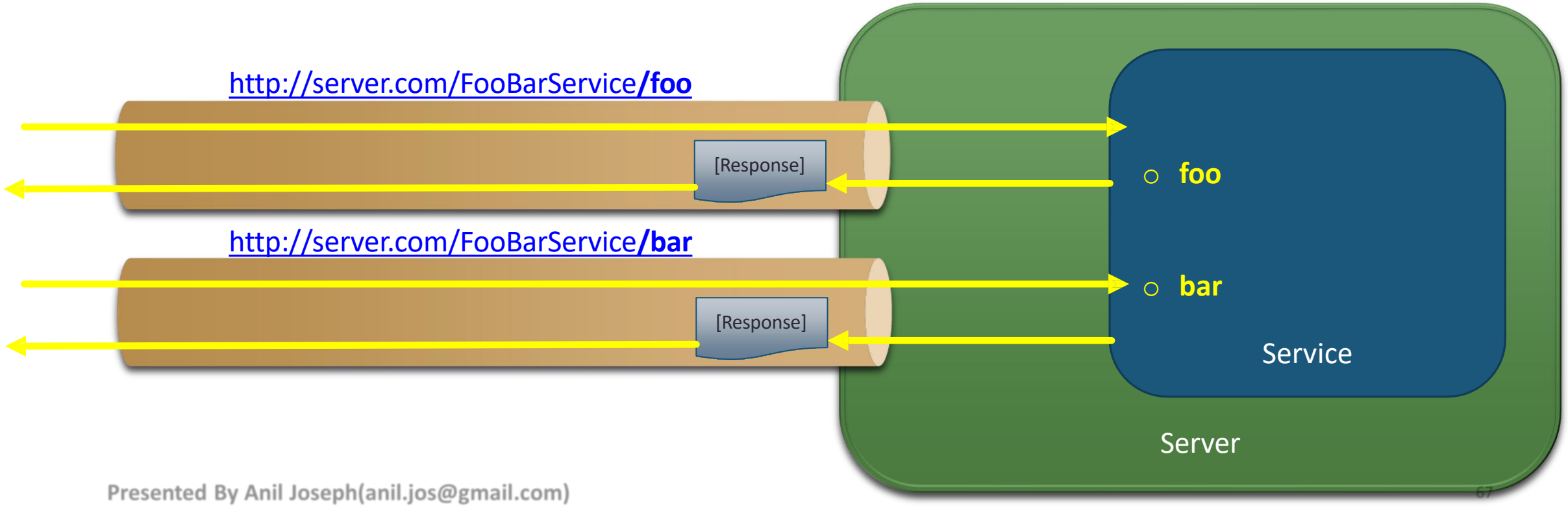
POST: Create a new resource.

PUT: Update an existing resource or create a resource if it does not exist.

DELETE: Remove a resource.

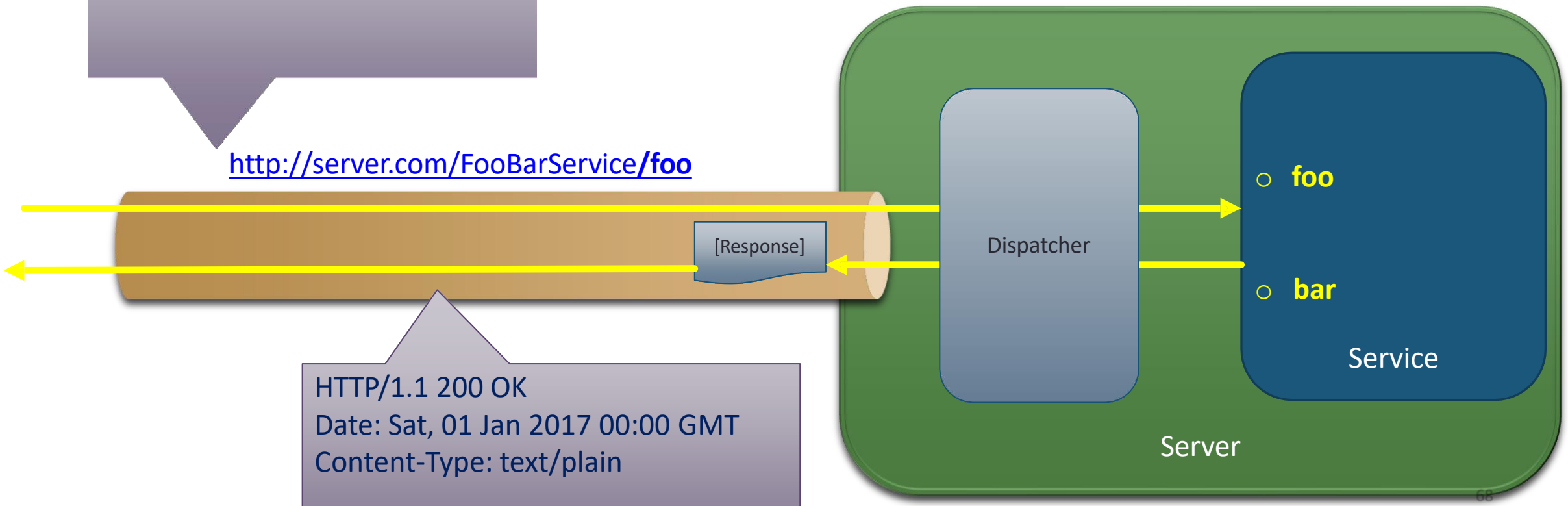
PATCH: Partially update a resource.

REST Services



REST Services

HTTP/1.1 200 OK
Date: Sat, 01 Jan 2017 00:00 GMT
Content-Type: text/plain



HTTP/1.1 200 OK
Date: Sat, 01 Jan 2017 00:00 GMT
Content-Type: text/plain

{“name”: “anil joseph”}

MongoDB

- MongoDB is a NoSQL database
- Stores data in flexible, JSON-like documents.
- Unlike traditional relational databases, which use tables and rows to store data, MongoDB uses collections and documents.

MongoDB advantages

- **Flexibility:**
  - MongoDB's document model allows for dynamic schemas, meaning you can store data without predefined structures.
  - Each document can have a different set of fields, making it easier to evolve your data model as your application requirements change.
- **Scalability:**
  - MongoDB is designed to scale horizontally by distributing data across multiple servers.
  - It supports sharding, which allows you to partition your data across a cluster of machines, ensuring high availability and scalability.

MongoDB advantages

- **Performance:**
  - MongoDB's document model allows for efficient read and write operations.
  - Indexes can be created to improve query performance
  - Supports rich query capabilities, including filtering, sorting, and aggregations.
- **Ease of Use:**
  - MongoDB provides a simple and intuitive query language based on JavaScript Object Notation (JSON).
  - Drivers for most popular programming languages, enabling seamless integration with various applications.
- **Rich Ecosystem:**
  - MongoDB comes with a robust ecosystem of tools and services,
  - MongoDB Atlas (a managed cloud database service),
  - MongoDB Compass (a graphical user interface for MongoDB),
  - Libraries for data modeling, migration, and analytics.

Key Concepts

- Database:
  - A container for collections.
  - Each database has its own set of files on the file system.
- Collection:
  - A grouping of MongoDB documents, similar to a table in a relational database.
- Document:
  - A record in a collection, represented as a BSON (Binary JSON) object.
  - Each document contains one or more fields, and each field can hold data of various types, such as strings, numbers, arrays, and even nested documents.
- Schema:
  - While MongoDB is schema-less, you can enforce data validation rules to ensure data integrity.

Common Use-cases

- Content Management Systems:
  - Storing and managing content with diverse structures and frequent changes.
- Real-Time Analytics:
  - Handling large volumes of data and providing real-time insights.
- Internet of Things (IoT):
  - Managing data from numerous devices with varying data structures.
- Mobile and Web Applications:
  - Storing user profiles, session data, and other application-specific information.

Socket.io

Socket.IO is a library that enables real-time, bidirectional, and event-based communication between web clients and servers.

It is built on top of WebSockets.

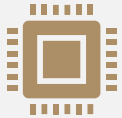
Socket.io advantages

- Real-Time Communication:
  - Socket.IO allows real-time communication, without the need for the client to make periodic requests (polling).
- Bidirectional Communication:
  - Communication flows both ways; the server can send messages to the client, and the client can send messages to the server.
- Event-Based:Communication is based on events.
  - Both the client and server can emit and listen for events, making the interaction straightforward and intuitive.

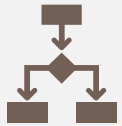
Socket.io Features

- WebSocket Fallback:
  - Socket.IO uses WebSockets if they are available.
  - Falls back to other technologies like long-polling if WebSockets are not supported.
- Automatic Reconnection:
  - If the connection between the client and server is lost, Socket.IO will automatically attempt to reconnect, handling transient network issues gracefully.
- Binary Support:
  - Socket.IO supports binary data transmission, which is useful for applications needing to send files, images, or other binary data.

Node: Options to perform CPU intensive applications



child_process



cluster

worker_threads

Workers

- Worker threads provide a way to run JavaScript code in parallel, taking advantage of multi-core systems.
- Allows for more efficient execution of CPU-intensive tasks.
  - Offloads them to separate threads
  - Avoids the blocking of the main event loop.
- Introduced in Node.js 10.5.0
- Worker threads have their own event loop.
- Worker threads communicates with the main thread and other workers using the messaging system

When to Use Worker Threads

- CPU-Intensive Tasks:
  - Tasks such as image processing, complex calculations, or data parsing that require significant CPU resources.
- Parallel Execution:
  - Situations where tasks can be executed in parallel to improve performance and reduce overall execution time.

Sharing data between workers

- SharedArrayBuffer
  - SharedArrayBuffer is a special type of array buffer that allows data to be shared between threads.
  - Unlike ArrayBuffer, which can only be accessed by one thread, SharedArrayBuffer can be accessed by multiple threads simultaneously.
- Atomics
  - Atomics is a global object that provides atomic operations for accessing and modifying shared memory.
  - Atomic operations are crucial in multi-threaded environments to avoid race conditions and ensure data consistency.

Atomics

- Atomics.add(typedArray, index, value)
  - Adds a value to the element at the specified index and returns the previous value.
  - Example: Atomics.add(sharedArray, 0, 1);
- Atomics.load(typedArray, index)
  - Returns the value at the specified index.Example:
  - Atomics.load(sharedArray, 0);
- Atomics.store(typedArray, index, value)
  - Stores the given value at the specified index and returns the value.
  - Example: Atomics.store(sharedArray, 0, 10);

Child Process

- the child_process module allows you to create and manage subprocesses.
- These subprocesses can execute shell commands or other Node.js scripts.
- Enables to perform tasks concurrently and handle CPU-intensive operations outside of the main event loop.

Child Process

- Spawning Processes:
  - You can spawn a new process to run a command or execute a script.
- Communication:
  - Child processes can communicate with the parent process through standard input (stdin), standard output (stdout), and standard error (stderr).
- Event-Driven:
  - Child processes emit events that can be handled to monitor their execution and manage their lifecycle.

Child Process

- Executing Shell Commands:
  - Child processes are ideal for running shell commands or executing external scripts and programs.
- Parallel Processing:
  - Performing CPU-intensive tasks in parallel to avoid blocking the main event loop.
- Process Isolation:
  - Running untrusted or resource-heavy tasks in isolated environments.
- Node.js Scripts:
  - Spawning other Node.js scripts to distribute workload.

Child Process: methods

- spawn:
  - Launches a new process with a given command.
- exec:
  - Executes a command in a shell and buffers the output.
- execFile:
  - Similar to exec but more efficient for executing files directly.
- fork:
  - Special case of spawn specifically for spawning new Node.js processes.

Workers vs. Child Process

- Memory and Resource Management:
  - Worker Threads: Share the same memory space, which allows for efficient data sharing but requires careful synchronization.
  - Child Processes: Have separate memory spaces, which provides isolation and prevents one process from affecting another.
- Communication:
  - Worker Threads: Communicate through shared memory and message passing.
  - Child Processes: Communicate through IPC (Inter-Process Communication) channels, such as standard input/output or message passing.

Workers vs. Child Process

- Overhead:
  - Worker Threads: Lower overhead due to shared resources.
  - Child Processes: Higher overhead due to separate resources.
- Execution Environment:
  - Worker Threads: Suitable for running Node.js code.
  - Child Processes: Suitable for running any type of code, including non-Node.js code.

Cluster Module

- The cluster module in Node.js is used to create child processes (workers) that share the same server port.
- Useful for taking advantage of multi-core systems to improve the performance of network servers.
- Using the cluster module, you can create multiple instances of your application that run in parallel, thus handling more requests simultaneously.

PM2(Process Manager 2)

- PM2 (Process Manager 2) is a popular production-grade process manager for Node.js applications.
- It helps manage and monitor applications, ensuring that they run smoothly in production environments.

Main Features of PM2

- **Process Management**:
  - Start, stop, restart, and delete applications with ease.
  - Manage multiple applications or instances of an application.
  - Support for clustering to take advantage of multi-core systems.
- **Monitoring**:
  - Real-time monitoring of applications with key metrics like CPU and memory usage.
  - Built-in log management to view and manage application logs.

Main Features of PM2

- **Process Reliability**:
  - Automatically restarts crashed or unresponsive applications.
  - Zero-downtime reloads to update applications without downtime.
- **Deployment**:
  - Integrated deployment workflow to automate deployment processes.

PM2

- Install: **npm install pm2 –g**
- Start an Application: **pm2 start app.js**
- Start an application(cluster): **pm2 start app.js -i max**
- List All Processes: **pm2 list**
- Monitor Processes:**pm2 monit**
- Restart an Application: **pm2 restart app.js**
- Stop an Application: **pm2 stop app.js**
- Delete an Application:**pm2 delete app.js**
- Reload an Application Without Downtime: **pm2 reload app.js**

PM2

- Start the application using the configuration file:
  - **pm2 start ecosystem.config.js --env production**
- View logs of an application:
  - pm2 logs my-app
- Stream logs in real-time:
  - pm2 logs

Unit Testing Frameworks

- Mocha:
  - A flexible testing framework that provides test structure and reporting.
- Jest:
  - A zero-config testing framework developed by Facebook, particularly good for React applications.
- Jasmine:
  - A behavior-driven development framework for testing JavaScript code.

Unit Testing

- Test Suites:
  - Collections of related tests. In most frameworks, these are defined using describe blocks.
- Test Cases:
  - Individual tests that check a specific piece of functionality. These are defined using it or test blocks.
- Assertions:
  - Statements that check if a condition is true. If the assertion fails, the test fails. Common assertion libraries include Chai (used with Mocha) and the built-in assertions in Jest.

Containers

- A container is a lightweight, standalone, executable package that includes everything needed to run a piece of software, including the code, runtime, system tools, libraries, and settings.
- Containers are isolated from each other and the host system, ensuring that they run consistently regardless of the environment.

Characteristics of Containers

- Isolation:
  - Containers provide process and file system isolation. They use Linux namespaces and cgroups to achieve this isolation.
- Portability:
  - Containers encapsulate all dependencies, making them portable across different environments (development, testing, production).
- Efficiency:
  - Containers share the host system's kernel, allowing them to be more efficient in terms of resource usage compared to traditional virtual machines.

Container vs. Virtual Machine

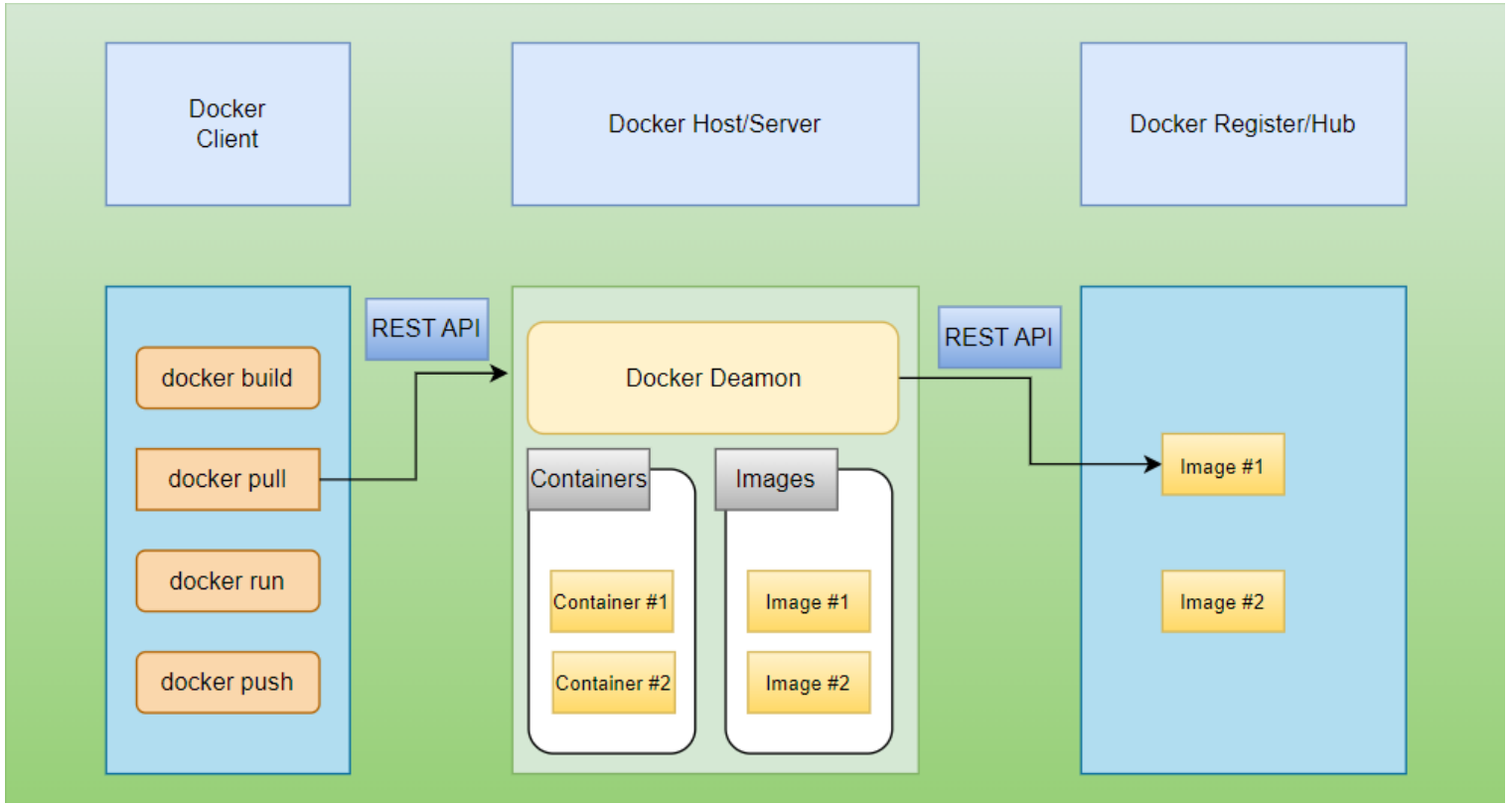
- A virtual machine is an emulation of a physical computer.
- VMs run a complete operating system (OS) along with a hypervisor or virtual machine monitor (VMM) that manages and allocates resources from the host machine to each VM.
- Each VM runs its own OS, which can be different from the host OS.
- VMs are completely isolated from each other. Each VM has its own kernel.
- VMs require more system resources (CPU, memory, storage) because each VM runs a full OS.
- VMs have longer boot times because the entire OS needs to start.

Docker

Docker is an open-source platform that automates the deployment, scaling, and management of applications within containers.

It provides the ability to package an application and its dependencies into a container.

Docker Architecture



Components of Docker

- Docker Engine
  - The core component of Docker, responsible for running and managing containers
  - The Docker daemon (dockerd) is the background service responsible for managing Docker containers, images, networks, and storage volumes.
  - The daemon listens for Docker API requests and performs the requested actions, such as starting, stopping, and managing containers.
- Docker Client:
  - The Docker client is a command-line interface (CLI) that users interact with to communicate with the Docker daemon.
  - Commands like docker run, docker build, and docker pull are issued from the Docker client.
  - The Docker client can communicate with multiple Docker daemons.

Components of Docker

- Docker Images:
  - Read-only templates used to create containers.
  - An image includes the application code, libraries, dependencies, and other required files.
  - Images are built from a Dockerfile.
  - Images are stored in a registry and can be pulled to create containers.
- Docker Containers:
  - Runtime instances of Docker images.
  - Containers are isolated, lightweight, and include everything needed to run an application.

Components of Docker

- Docker Registry:
  - A Docker registry is a storage and distribution system for Docker images.
  - The most common registry is Docker Hub, which provides public and private repositories.
  - Users can push images to and pull images from a Docker registry.
- Dockerfile:
  - A file containing a set of instructions on how to build a Docker image.
  - It specifies the base image, application code, dependencies, and commands to run the application.

Components of Docker

- Docker Compose:
  - A tool for defining and running multi-container Docker applications using a docker-compose.yml file.
  - It simplifies the process of managing multiple containers.

What is DevOps?

- DevOps is a set of practices that combines software development (Dev) and IT operations (Ops) to shorten the systems development life cycle and provide continuous delivery with high software quality.

Plan(Dev)

Code(Dev)

Build(Dev)

Test(Dev)

Release(DevOps)

Deploy(Ops)

Operate(Ops)

Monitor(Ops)

Continuous Integration

- Continuous Integration (CI) is a software development practice where developers frequently integrate their code changes into a shared repository, often multiple times a day.
- Each integration is automatically verified by building the code and running automated tests to detect integration errors as quickly as possible.
- Improved Code Quality:
  - Regular testing helps maintain high code quality.
- Reduced Integration Issues:
  - Early detection of integration problems.
- Faster Feedback:
  - Immediate feedback on code changes.

Continuous Delivery

- Continuous Delivery (CD) is a practice where code changes are automatically prepared for release to production.
- It builds on CI by deploying code changes to testing and production environments after the build stage.
- CD ensures software can be reliably released at any time.

CI/CD

CI: Focuses on integrating and testing code frequently to catch issues early.

CD: Focuses on automating the release process so that the codebase is always ready for deployment.

Infrastructure as Code (IaC):

- A practice where infrastructure is provisioned and managed using code and software development techniques, such as version control and continuous integration.
- This allows for the automation of infrastructure management processes.

DevOps Tools

 Version Control:	Git, GitHub, GitLab
 CI/CD:	Jenkins, CircleCI, Travis CI, GitHub actions, Gitlab CI/CD
 Configuration Management:	Ansible, Chef, Puppet
 Containerization:	Docker, Kubernetes
 Monitoring:	Prometheus, Grafana, Nagios

Github Actions

- GitHub Actions is a powerful automation tool integrated directly into GitHub repositories.
- It enables developers to automate workflows, including continuous integration (CI) and continuous deployment (CD), right from their GitHub repositories.

Github Actions Features

- Workflow Automation:
  - Event-Driven: GitHub Actions can trigger workflows based on specific events (e.g., push, pull request, release).
  - These events can be tailored to fit the needs of the repository.
  - Custom Workflows: Users can define custom workflows using YAML syntax, specifying different jobs, steps, and actions to be performed.
- Continuous Integration (CI):
  - Automated Testing: Automatically run tests on different environments and configurations to ensure code quality and functionality.
  - Build Automation: Automate the build process for different platforms and environments, ensuring that the code is always in a deployable state.

Github Actions Features

- Continuous Deployment (CD):
  - Automated Deployments: Deploy applications to various environments (e.g., staging, production) automatically after successful builds and tests.
  - Environment Management: Manage different deployment environments and ensure smooth rollouts with proper versioning and rollback mechanisms.
- Extensibility
- Parrallel Execution
- Secrets Management
- Monitoring and Logs

Anil Joseph

anil.jos@gmail.com

WhatsApp : +91 98331 69784

Thank You