AKTU

B.Tech 5th Sem

GATEWAY CLASSES

CS IT & CS Allied

Web Technology

ONE SHOT Revision

(Crash Course)

Unit-4

Enterprise Java Bean Node.js with MongoDB

By Amol Sharma sir

- Pursuing Ph.D. from IIT (BHU)
- Wipro Certified Faculty (WCF)

Computer Science & Engineering , Information Technology & Computer Science Allied

UNIT-I : Introduction, Web Page Designing

UNIT-II : CSS, CSS Advanced

UNIT-III : Scripting, Networking

UNIT-IV : Enterprise Java Bean, Node.js, Node.js with Mongo DB

UNIT-V : Servlets, Java Server Pages (JSP)

(BCS-502 - Web Technology)

Unit-IV : Enterprise Java Bean, Node.js, Node.js with Mongo DB

AKTU : Syllabus

Enterprise Java Bean: Creating a JavaBeans, JavaBeans Properties, Types of beans, State ful Session bean, Stateless Session bean, Entity bean.

Node.js: Introduction, Environment Setup, REPL Terminal, NPM (Node Package Manager) Callbacks Concept, Events, Packaging, Express Framework, Restful API.

Node.js with Mongo DB: Mongo DB Create Database, Create Collection, Insert, delete, update, join, sort, query.

Web Technology

UNIT – IV:

Enterprise Java Bean: Creating a Java Bean, Java Beans Properties, Types of beans, Stateful Session bean, Stateless Session bean, Entity bean.

Node.js: Introduction, Environment Setup, REPL Terminal, NPM (Node Package Manager) Callbacks Concept, Events, Packaging, Express Framework, Restful API.

Node.js with MongoDB: MongoDB Create Database, Create Collection, Insert, delete, update, join, sort, query.

AKTU PYQs

Web Technology (AKTU 2022 – 23)

1. Illustrate Java Beans and their properties in detail.
2. Compare Stateful Session Bean and Stateless Session Bean.

Web Technology (AKTU 2021 – 22)

1. Describe the characteristics of JavaBeans.
2. Illustrate Java Bean. Explain characteristics of java Bean. Give example.

Web Technology (AKTU 2020 – 21)

1. Define Java Bean.
2. Explain and Compare Stateful Session Bean and Stateless Session Bean.
3. Explain Java Bean. Describe Java Bean properties and types in detail.

Web Technology (AKTU 2019 – 20)

1. Describe EJB. Explain EJB architecture. What are its various types?
2. Explain JavaBeans. Why they are used? Discuss setter and getter methods with Java code.

Web Technology (AKTU 2018 – 19)

No question on Java Beans

AKTU

B.Tech 5th Sem

GATEWAY CLASSES

CS IT & CS Allied

Web Technology

UNIT-4 Lecture-1

Today's Target

- Java Beans
  - Characteristics
  - Advantages
  - Properties
  - Creating a Java Bean
- AKTU PYQs

By Amol Sharma sir

- Pursuing Ph.D. from IIT (BHU)
- Wipro Certified Faculty (WCF)

Java Beans

"A Java Bean is a software component that has been designed to be reusable in a variety of different environments."

- Software components written in Java
- WORA (Write Once Run Anywhere)
- Configurable
- Begins 'Age of Component based Development'
- Bringing Engineering methods to Software Engineering (e.g. electronics…)

The JavaBeans API provides a framework for defining reusable, embeddable, modular software components.

Java Beans

Characteristics

Following are the unique characteristics that distinguish a JavaBean from other Java classes

- It provides a default, no-argument constructor.
- It should be serializable and that which can implement the Serializable interface.
- It may have a number of properties which can be read or written.
- It may have a number of "getter" and "setter" methods for the properties.

Java Beans

Advantages

- Beans can be reused.
- Java Beans facilitates Component based Software Development.
- A Bean obtains all the benefits of Java's "write-once, run-anywhere" paradigm.
- The properties, events, and methods of a Bean that are exposed to another application can be controlled.
- Auxiliary software can be provided to help configure a Bean. This software is only needed when the design-time parameters for that component are being set. It does not need to be included in the run-time environment.
- The state of a Bean can be saved in persistent storage and restored at a later time.
- A Bean may register to receive events from other objects and can generate events that are sent to other objects.

Java Beans

JavaBeans Properties

- Properties are attributes of a JavaBean that can be accessed or modified using getter and setter methods.
- Enable encapsulation and controlled access to data.

Types of JavaBeans Properties

Simple Property:	A property that holds a single value. (e.g., name, age).
Indexed Property:	A property that holds an array or list of values (e.g., list of courses).
Bound Property:	A property that notifies when its value changes.
Constrained Property:	A property that allows vetoing (rejecting) changes.

Java Beans

JavaBeans Naming Conventions or Design Patterns for Properties

Simple Properties

A simple property has a single value.

It can be identified by the following design patterns

Capitalize the first letter of every word in the method name after get or set.

Getter:	public <type> get<PropertyName>()
Setter:	public void set<PropertyName>(<type> value)

Example:

For name property, the methods will be getName() and setName().

Boolean Properties: Getter can be named as is<PropertyName>().

Example: For active property of boolean type, the getter method will be isActive().

Java Beans

JavaBeans Naming Conventions or Design Patterns for Properties

Indexed Properties

An indexed property consists of multiple values.

It can be identified by the following design patterns:

```
public <type> get<PropertyName>(int index);
public void set<PropertyName> (int index, <type> value);
public <type>[ ] get<PropertyName> ( );
public void set<PropertyName> (<type> values[ ]);
```

Example:

For course property, the methods will be

String getCourse(int index) and void setCourse(int index, String course).

String[] getCourse() and void setCourse(String[] courses).

Java Beans

Creating a JavaBean

1. Define the Class:

   Declare it public and implement Serializable.

2. Add Private Fields:

   Define private fields for properties.

3. Add Getters and Setters:

   Follow the JavaBean naming conventions.

4. Provide a No-Argument Constructor

Java Beans

Example:

```
import java.io.Serializable;

public class StudentBean implements Serializable {

    private String name;
    private int age;

    // No-argument Constructor
    public StudentBean() { }

    // Getter for name
    public String getName() {
        return name;
    }
```

Java Beans

Example:

```
    // Setter for name
    public void setName(String name) {
        this.name = name;
    }

    // Getter for age
    public int getAge() {
        return age;
    }

    // Setter for age
    public void setAge(int age) {
        this.age = age;
    }
}
```

AKTU

B.Tech 5th Sem

GW
GATEWAY CLASSES

CS IT & CS Allied

Web Technology

UNIT-4 Lecture-2

Today's Target

- Java Beans
  - Introspection
  - Enterprise Java Beans
  - Types of Beans
    - Session Beans
      - Stateful Session Beans
      - Stateless Session Beans
    - Message Driven Beans
    - Entity Beans
- AKTU PYQs

By Amol Sharma sir
- Pursuing Ph.D. from IIT (BHU)
- Wipro Certified Faculty (WCF)

Java Beans

Introspection

"*Introspection* is the process of analyzing a Bean to determine its capabilities."

This is an essential feature of the JavaBeans API because it allows another application, such as a design tool, to obtain information about a component.

The JavaBeans API for Introspection

BeanInfo	This interface allows to specify information about the properties, events, and methods of a Bean.
Introspector	The Introspector class provides a static method *getBeanInfo( )*. This method returns a BeanInfo object that can be used to obtain information about the Bean.
PropertyDescriptor	The PropertyDescriptor class manage and describe Bean properties.
MethodDescriptor	The MethodDescriptor class represents a Bean method.
EventSetDescriptor	The EventSetDescriptor class represents a set of Bean events.

Java Beans

Introspection

Example:

```java
import java.beans.*;

public class SimpleBeanIntrospection {
    public static void main(String[] args) throws IntrospectionException {
        BeanInfo beanInfo = Introspector.getBeanInfo(StudentBean.class);

        // Inspect Properties
        for (PropertyDescriptor property : beanInfo.getPropertyDescriptors()) {
            System.out.println("Property: " + property.getName());
        }
        // Inspect Methods
        for (MethodDescriptor method : beanInfo.getMethodDescriptors()) {
            System.out.println("Method: " + method.getName());
        }
        // Inspect Events
        for (EventSetDescriptor event : beanInfo.getEventSetDescriptors()) {
            System.out.println("Event: " + event.getName());
        }
    }
}
```

Java Beans

Enterprise Java Beans (EJB)

"EJB is a server-side component architecture for modular construction of enterprise applications."

- It extends the JavaBeans component model to support distributed computing.
- It is based on distributed component model i.e. It is a standard way of writing distributed components so that the written components can be used with the components you write in some other system.

Enterprise applications are large-scale software systems designed to support complex and critical business processes of an organization. They serve multiple users, often across different departments, and handle a lot of data.

Examples:

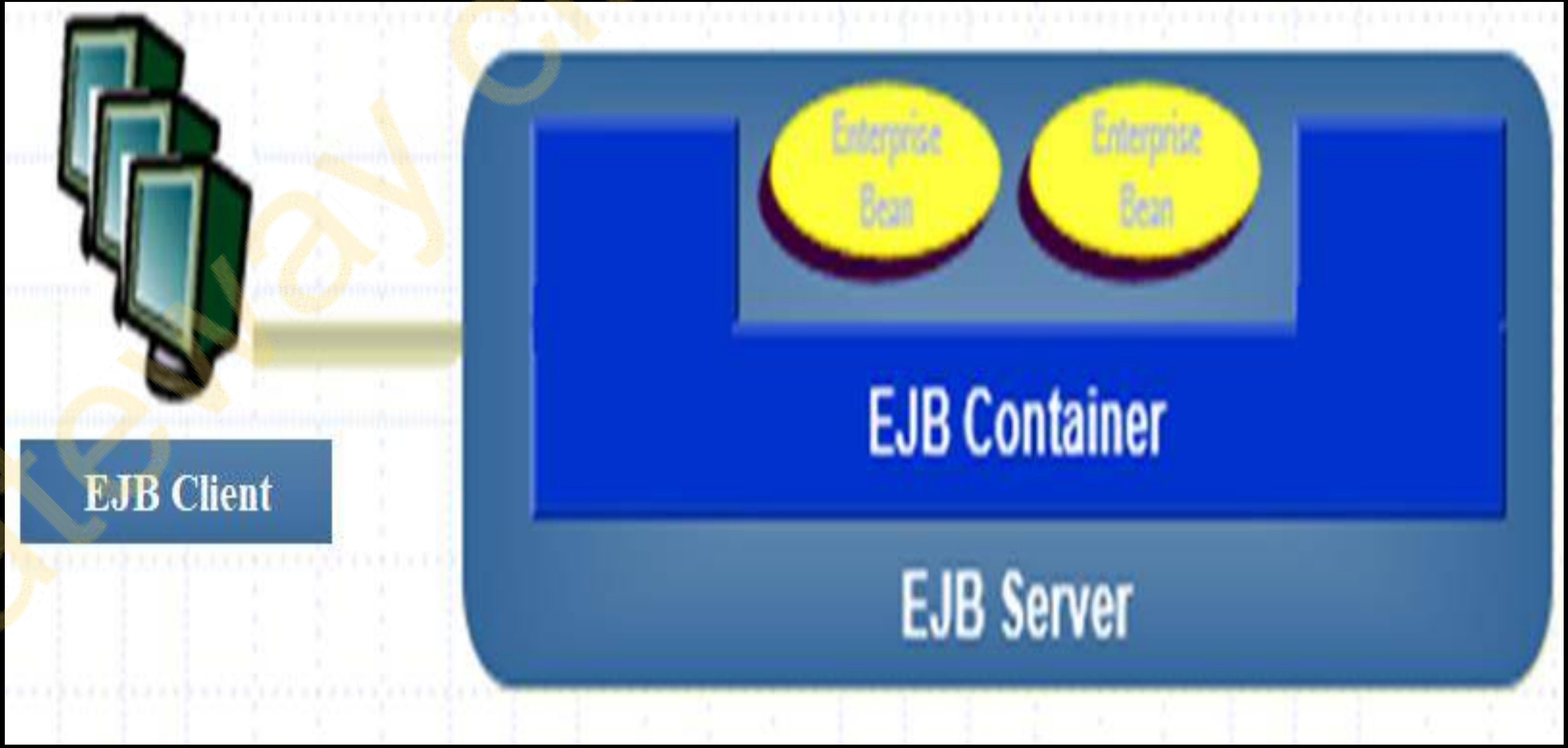
Banking Applications:	Used to manage accounts, loans, and transactions.
E-Commerce Platforms:	Websites like Amazon that handle orders, payments, and inventory.

Java Beans

EJB Architecture

The *EJB architecture* specifies the responsibilities of and interactions among following EJB entities:

1. EJB
2. EJB Container
3. EJB Server
4. EJB Client



Java Beans

EJB Architecture

The *EJB architecture* specifies the responsibilities of and interactions among following *EJB entities*:

1. EJB: EJBs are the core components that contain business logic, designed to be used by clients across a network.

2. EJB Container: EJB Container provides runtime support for EJBs. It acts as an intermediary between EJBs and the EJB server. It is responsible for handling services like lifecycle management, transaction management, security, and concurrency etc.

3. EJB Server: EJB Server provides the run time environment in which the EJB container runs. It manages low-level tasks like networking and resource allocation.

4. EJB Client: An EJB Client can be a web application, a standalone application, or even another EJB. The client interacts with the EJB server to access and use the functionality provided by the EJB.

Java Beans

Types of Enterprise Java Beans (EJB)

Following are the different types of Enterprise Java Beans:

1. Session Beans
   i. Stateful Session Bean
   ii. Stateless Session Bean
2. Message Driven Beans
3. Entity Beans

Java Beans

Types of Enterprise Java Beans (EJB)

Following are the different types of Enterprise Java Beans:

1. Session Beans

A Session Bean is a type of Enterprise JavaBean (EJB) that encapsulates business logic and provides services to clients. It is used to perform tasks such as computations, data processing, or managing client-specific business processes.
Session Beans are divided into two main types:

i. Stateful Session Bean

ii. Stateless Session Bean

Java Beans

Types of Enterprise Java Beans (EJB)

1. Session Beans

   i. Stateful Session Bean A Stateful Session Bean maintains the client’s state across multiple method invocations. This type of bean stores data specific to the client’s session, which means it can remember the client’s information during interactions.
   The *shopping cart* in an e-commerce application, a Stateful Session Bean can store the user's selections until the transaction is completed. The state is preserved throughout the client’s interaction with the server, and when the client is done, the bean is removed from memory.

   ii. Stateless Session Bean A Stateless Session Bean does not maintain any state between method calls. It is designed to perform business logic or computations without remembering any information about the client.
   Each method call is independent of others, and once the method completes, the bean can be reused by other clients.

Java Beans

Types of Enterprise Java Beans (EJB)

1. Session Beans

Stateful and Stateless Session Beans: Comparison

Stateful Session Bean	Stateless Session Bean
Maintains client-specific state across multiple method calls.	Does not maintain any client-specific state between method calls.
Each client has a dedicated bean instance with its own state.	Bean instances are shared among clients, with no memory of previous interactions.
Typically requires more resources due to the need to maintain state.	More efficient as it does not store any state, and the same instance can be reused.
Suitable for tasks requiring continuity, such as user sessions, shopping carts, or workflows.	Suitable for tasks that do not require continuity, such as stateless business logic, calculations, or simple data retrieval.
Less efficient for handling many clients simultaneously due to the need to maintain individual states.	Can handle many clients efficiently as the bean instance can be reused across multiple clients.
Requires more memory and resources to store the state of each client session.	More efficient since the bean instance can be pooled and reused by any client.
Shopping cart in an e-commerce site where the cart's contents are stored until checkout.	A payment processor that handles a transaction request without needing to store session data.

Java Beans

Types of Enterprise Java Beans (EJB)

2. Message Driven Beans

Message-Driven Beans are used to handle asynchronous messages. These beans are designed to process incoming messages without blocking the client, making them ideal for scenarios like *email notifications*. MDBs listen for messages in message queues and act as consumers of those messages, providing reliable and scalable communication in distributed applications.

3. Entity Beans *(Deprecated in EJB 3.0, replaced by JPA)*

Entity Beans were used to represent *persistent data* that could be stored in a relational database. Each Entity Bean was mapped to a database table, allowing automatic *database interactions* through methods like *create, read, update, and delete (CRUD)*.

However, Entity Beans were complex and required extensive configuration. In EJB 3.0, Entity Beans were replaced by *Java Persistence API (JPA)*, which offers a more lightweight and flexible approach to persistence.

AKTU PYQs

Web Technology (AKTU 2022 – 23)

1. Illustrate Java Beans and their properties in detail.
2. Compare Stateful Session Bean and Stateless Session Bean.

Web Technology (AKTU 2021 – 22)

1. Describe the characteristics of JavaBeans.
2. Illustrate Java Bean. Explain characteristics of java Bean. Give example.

Web Technology (AKTU 2020 – 21)

1. Define Java Bean.
2. Explain and Compare Stateful Session Bean and Stateless Session Bean.
3. Explain Java Bean. Describe Java Bean properties and types in detail.

Web Technology (AKTU 2019 – 20)

1. Describe EJB. Explain EJB architecture. What are its various types?
2. Explain JavaBeans. Why they are used? Discuss setter and getter methods with Java code.

Web Technology (AKTU 2018 – 19)

No question on Java Beans

AKTU

B.Tech 5th Sem

CS IT & CS Allied

Web Technology

UNIT-4 Lecture-3

Today's Target

- Node.js
  - Introduction,
  - Environment Setup,
  - REPL Terminal,
  - NPM (Node Package Manager),
  - Callbacks

By Amol Sharma sir

- Pursuing Ph.D. from IIT (BHU)
- Wipro Certified Faculty (WCF)

Web Technology

UNIT – IV:

Enterprise Java Bean: Creating a Java Bean, Java Beans Properties, Types of beans, Stateful Session bean, Stateless Session bean, Entity bean.

Node.js: Introduction, Environment Setup, REPL Terminal, NPM (Node Package Manager) Callbacks Concept, Events, Packaging, Express Framework, Restful API.

Node.js with MongoDB: MongoDB Create Database, Create Collection, Insert, delete, update, join, sort, query.

Node.js

Introduction

"Node.js is a runtime environment for executing JavaScript code outside of a web browser. It allows JavaScript to run on the server side, making it possible to build the backend of applications using JavaScript."

- Node.js is a server-side runtime environment built on Google Chrome's JavaScript Engine (V8 Engine).
- Node.js was developed by Ryan Dahl in 2009.
- Node.js is a free, open-source, cross-platform JavaScript runtime environment.
- Node.js may be used by developers to create servers, web apps, command line tools and scripts.
- With Node.js, developers use JavaScript for both frontend and backend i.e. single language for full stack.

Node.js

Introduction

Node.js, in its official documentation, is described as follows:

"As an asynchronous event-driven JavaScript runtime, Node.js is designed to build scalable network applications."

Features:

- Asynchronous and Event-Driven
- Non-Blocking I/O Model
- Lightweight and Efficient
- Single-Threaded but handles Concurrency

Node.js

Introduction

Features

- Asynchronous and Event-Driven

  Node.js doesn't wait for one task to finish before moving to the next. Instead, tasks are executed asynchronously, with the results handled later using events or callbacks.

- Non-Blocking I/O Model

  Input/Output operations (e.g., reading files or querying a database) do not block the execution of other tasks.

- Lightweight and Efficient

  Node.js uses minimal resources because it doesn't create a separate thread for every request. Best suited for data-intensive, real-time applications like streaming platforms.

- Single-Threaded but handles Concurrency

  Node.js is single-threaded (runs on one core of the CPU), but due to its event loop and non-blocking I/O, it can handle many connections concurrently.

Node.js

Introduction

Event Loop

The Event Loop in Node.js is the heart of its asynchronous, non-blocking behavior.

"The event loop is a mechanism in Node.js that manages and coordinates the execution of multiple operations (tasks) without blocking the main thread."

It continuously checks for tasks (e.g., callbacks, timers, or I/O operations) and processes them when they are ready:

1. Node.js uses a single thread to execute JavaScript code.
2. Tasks that are waiting (e.g., file reading or HTTP requests) are placed in a queue.
3. The event loop constantly checks if there are tasks in the queue that are ready to execute.
4. If a task is ready (e.g., file read is complete), the event loop picks it from the queue and runs its associated callback.

Node.js

Environment Setup

To set up environment for Node.js, we need the following tools on our system:

- The Node.js binary installer

  Node.js, being cross platform, can be installed on different OS platforms such as Windows, Linux, Mac OS X. Binaries for various OS platforms are available on the downloads page of the official website of Node.js: https://nodejs.org/en/download/package-manager.

- Node Package Manager (NPM)

  NPM is a tool that comes with Node.js. It helps you easily add and manage useful code packages (libraries) in your projects, so you don't have to write everything from scratch.

- IDE or Text Editor

  An IDE (like VS Code) or text editor (like notepad) is needed to write and manage your Node.js code effectively.

Node.js

REPL (Read-Evaluate-Print Loop) Terminal

The Node.js runtime includes a built-in interactive shell that allows you to execute instructions one at a time.
This interactive shell operates on the principle of REPL, which stands for Read, Evaluate, Print, and Loop.

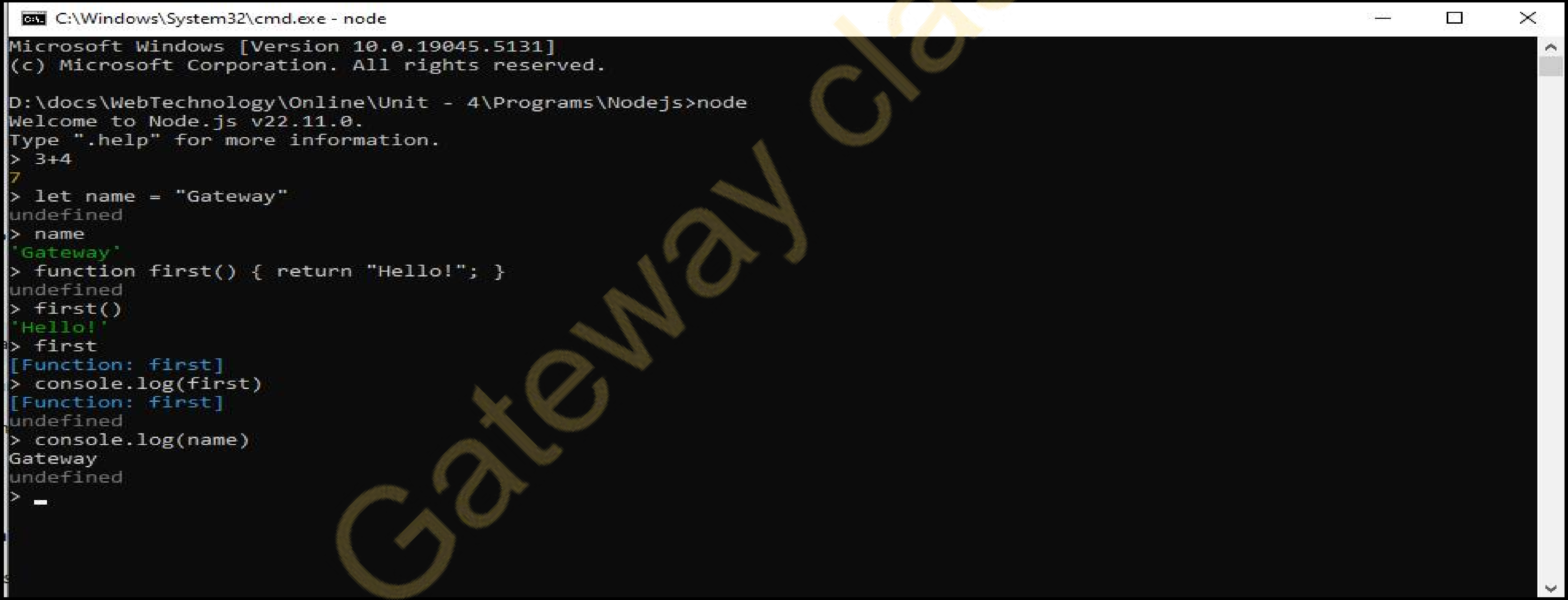
➢ R: Read –	Reads user input.
➢ E: Evaluate –	Evaluates the input.
➢ P: Print –	Prints the result.
➢ L: Loop –	Loops back to wait for the next command.

- ➢ To start the Node.js REPL on your computer, simply enter ‘node’ in the command terminal The Node.js prompt ‘>’ will appear.
- ➢ On the prompt, we may enter any valid JavaScript code, execute it and immediately see the result.
- ➢ Type .exit or press Ctrl + C twice to exit the REPL environment.

Node.js

REPL ((Read-Evaluate-Print Loop) Terminal

Examples



C:\Windows\System32\cmd.exe - node

```
Microsoft Windows [Version 10.0.19045.5131]
(c) Microsoft Corporation. All rights reserved.

D:\docs\WebTechnology\Online\Unit - 4\Programs\Nodejs>node
Welcome to Node.js v22.11.0.
Type ".help" for more information.
> 3+4
7
> let name = "Gateway"
undefined
> name
'Gateway'
> function first() { return "Hello!"; }
undefined
> first()
'Hello!'
> first
[Function: first]
> console.log(first)
[Function: first]
undefined
> console.log(name)
Gateway
undefined
> _
```

Node.js

NPM (Node Package Manager)

NPM is a tool that comes with Node.js to help to easily manage extra code (called packages) that we need in our application.

Think of it like an app store for developers where we can find and install useful code libraries.

- It helps us to add extra functionality to the project without writing everything from scratch.
- We can easily install code written by others, such as libraries for web servers, reading files, or connecting to databases.
- NPM also helps to manage versions of these libraries.

Node.js

NPM (Node Package Manager)

Basic NPM Commands

Install a library (package):

```
npm install <package-name>
```

This command downloads a specific package and its files to your project.

Install all required libraries:

(if the project already has a list of needed libraries)

```
npm install
```

This downloads all libraries listed in a special file called package.json. This file keeps track of all the libraries (and their versions) needed by the project.

Node.js

Callbacks

A callback is a function passed as an argument to another function. It is executed after the completion of an asynchronous operation.

- Node.js relies on callbacks for asynchronous operations.
- Callbacks enable non-blocking execution, allowing Node.js to handle lengthy processes without halting the main thread.
- Callbacks improve scalability by allowing Node.js to process multiple requests simultaneously.

Syntax:

*function function_name (argument, **callback_function**){ .. }*

Node.js

Callbacks

Example:

```
var fs = require("fs");
fs.readFile ('myFile.txt', function (err, data) {
        if (err) return console.error(err);
                console.log(data.toString());
});
for (let i=1;i<10;i++) {
        console.log("The value of  i is " + i);
}
```

Note: The *require()* function in Node.js is used to import modules (built-in or external) into an application to use their functionality.
'fs' module provides APIs for interacting with the file system to read, write, and manipulate files.

AKTU

B.Tech 5th Sem

GATEWAY CLASSES

CS IT & CS Allied

Web Technology

UNIT-4 Lecture-4

Today's Target

- Node.js
  - Events
    - Event Handling in Node.js
  - Express Framework
    - Features
    - Building a Web Application
      - Handling Get and Post

By Amol Sharma sir

- Pursuing Ph.D. from IIT (BHU)
- Wipro Certified Faculty (WCF)

Node.js

Events

*"An event in Node.js is a signal that something has occurred, such as user interaction, a file operation, or a network request. Node.js uses an **event-driven model**, where **callback functions** are executed in response to specific events emitted by the system or application."*

- When JavaScript is used on client side, the client-side events are handled in the browser (e.g., button clicks, mouse movements), and when it is used on the server side, the server-side events are handled by Node.js on the server (e.g., file I/O operations, HTTP requests/responses).

Node.js

Events

Event Handling in Node.js

- The event to be handled is attached to a callback.
- The Node.js environment listens for events and executes the attached callback function whenever the event occurs.
- Node.js has a built-in EventEmitter class (from the events module) to handle events.
  - The EventEmitter object is used to trigger (emit) events.
  - We can assign one or more callback functions (listeners) to an event type.
  - When an event is triggered, all its registered callbacks are executed.
  - Callbacks are fired in the order of registration.

Node.js

Events

Event Handling in Node.js

Example

```
const EventEmitter = require('events');

const eventEmitter = new EventEmitter();


// Define an event listener
eventEmitter.on('DoorBellRing', () => {
  console.log('Open the door.');
});


// Emit the event
eventEmitter.emit('DoorBellRing');
```

Creating an EventEmitter Object:
We create an instance of the EventEmitter class, which allows us to handle events in Node.js.

Defining an Event Listener:
We use the .on() method to register a listener for the event 'DoorBellRing’. When the event 'DoorBellRing' is emitted, the callback function is executed, printing "Open the door." to the console.

Emitting an Event:
The .emit() method triggers the 'DoorBellRing' event, causing the event listener to run and display the message.

Node.js

Express Framework

"Express is a lightweight and flexible web application framework for Node.js. It helps developers build web applications and APIs quickly by providing essential tools and features to manage HTTP requests, routes, and middleware."

Features

- Routing: Define routes to handle different HTTP requests (GET, POST, etc.) with ease.
- Middleware: Easily add functions that process requests before reaching the final route handler (e.g., authentication).
- RESTful API Support: Simplifies the creation of RESTful APIs by providing routing and HTTP method handling.
- Error Handling: Built-in error handling to manage issues effectively.
- Cross-Platform: Works on various platforms including Linux, Windows, and macOS.

Node.js

Express Framework

To build an Express application, we first need to set up our project environment and install the required tools.

- ➢ Initializing the Project (npm init)
- ➢ Installing Express (npm install express --save)

npm init initializes a new Node.js project by creating a package.json file.

Purpose:

Project Metadata: It gathers information like the project name, version, description, entry point, and author. This metadata is stored in package.json, which acts as a blueprint for the project.

Dependency Management: Tracks the packages (like express) installed in the project. Lists all dependencies under a dependencies section.

Version Control: Helps maintain the version of installed packages, ensuring consistency across environments.

Ease of Sharing: Developers can share the project using the package.json file. Others can use npm install to install all the required dependencies.

Node.js

Express Framework

Building a web application

npm install express --save installs the Express framework and adds it to project.

Purpose:

Install the Package: Downloads the Express framework from the npm registry. Adds it to the node_modules folder, making it available for your project.

Add as a Dependency: Updates package.json by listing Express under the dependencies section.
Example: "dependencies": { "express": "^4.18.2"}

Reusability: When sharing the project, others can use npm install to automatically install Express and other dependencies listed in package.json.

Node.js

Express Framework

Building a web application

```
// Step 1: Importing the Express module
const express = require('express');

// Step 2: Creating an Express application instance
const app = express();

// Step 3: Defining a route to handle GET requests on the root ("/") URL
app.get('/', (req, res) => {
    res.send('Hello, Express!');
});

// Step 4: Starting the server and listening on the specified port
app.listen(3000, () => {
    console.log('Server is running on http://localhost:3000');
});
```

Node.js

Express Framework

Building a web application

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

This line imports the Express module into our application. The express function is now available to create and manage our web server.

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

This creates an Express application instance and assigns it to the app variable. This instance will be used to define routes, middleware, and manage HTTP requests.

```
app.get('/', (req, res) => { .. });
```

app.get() defines a route for handling GET requests on the root URL (/). When someone visits http://localhost:3000/ in a browser, the callback function will be executed.
The callback function (req, res) handles two main objects: req and res.

Node.js

Express Framework

Building a web application

`req`

The request object contains information about the incoming request (such as the URL, query parameters, headers, etc.).

`res`

The response object is used to send response to the client. In this case, we use res.send() to send a simple "Hello, Express!" message back to the browser.

```
app.listen(3000, () => { … });
```

This line starts the Express server and tells it to listen for incoming requests on port 3000. Once the server is up and running, it will print a message to the console: "Server is running on http://localhost:3000". It makes the express app respond to requests.

Node.js

Express Framework

Building a web application – Handling Get and Post

login.html

```html
<!DOCTYPE html>
<head>
    <title>Login Page</title>
</head>
<body>
    <h2>Login Page</h2>
    <form action="/login" method="POST">
        <label for="username">Username:</label>
        <input type="text" id="username" name="username" required><br><br>
        <label for="password">Password:</label>
        <input type="password" id="password" name="password" required><br><br>
        <button type="submit">Login</button>
    </form>
</body>
</html>
```

Node.js

Express Framework

Building a web application – Handling Get and Post

```
  // Importing modules
const express = require('express');
const bodyParser = require('body-parser');
const path = require('path');

// Creating an Express app
const app = express();

// Middleware to parse form data
app.use(bodyParser.urlencoded({ extended: true }));

// Handle GET request to serve the login page
app.get('/', (req, res) => {
    res.sendFile(path.join(__dirname, 'login.html'));
});

// Handle POST request for form submission
app.post('/login', (req, res) => {
    const username = req.body.username;
    const password = req.body.password;
```

```
    // Basic validation logic
    if (username === 'admin' && password === 'password') {
        res.send(`You have successfully logged in.`);
    } else {
        res.send('Invalid username or password.');
    }
});

// Starting the server
app.listen(3000, () => {
    console.log(`Server running at http://localhost:3000`);
});
```

Node.js

Express Framework

Building a web application – Handling Get and Post

```
const bodyParser = require('body-parser’);
```

Imports the body-parser middleware, which parses incoming request bodies so you can access form data via req.body.

```
const path = require('path’);
```

Imports the path module, which helps work with file and directory paths, making it easier to serve static files like login.html.

```
app.use(bodyParser.urlencoded({ extended: true }));
```

Adds the body-parser middleware to the app to handle form submissions.

Node.js

Express Framework

Building a web application

```
app.get('/', (req, res) => { … });
```

Defines a GET route for the root (/) URL. When users navigate to the homepage, this route serves a response (e.g., the login page).

```
res.sendFile(path.join(__dirname, 'login.html’));
```

Sends the login.html file to the client as a response. The path.join ensures the correct file path by joining the current directory (__dirname) and the file name.

```
app.post(‘/login', (req, res) => { … });
```

Defines a POST route for the /login URL. When the login form is submitted, this route handles the incoming form data from the request body (req.body) and processes it for validating credentials.

AKTU

B.Tech 5th Sem

GATEWAY CLASSES

CS IT & CS Allied

Web Technology

UNIT-4 Lecture-5

Today's Target

- ➢ Node.js
  - ➢ Restful API,
  - ➢ Packaging.

By Amol Sharma sir

- Pursuing Ph.D. from IIT (BHU)
- Wipro Certified Faculty (WCF)

Node.js

Restful API

REST (REpresentational State Transfer)

"REST (Representational State Transfer) is an architectural style for designing networked applications. It uses a stateless, resource-based approach to enable communication between client and server via standard HTTP methods."

Representational

Resources (like data about users, products, etc.) are represented in a format such as JSON, XML, or HTML.

Example: A user's details can be sent as JSON:

```
{ "id": 1, "name": "Alice", "isOnline": true }
```

State

Refers to the current status of a resource at a specific time.

Example: A user's state is the combination of its attributes (id, name, isOnline) and their values (1, Alice, true).

REST is *stateless*, meaning the server does not maintain any memory of the client's state. Each request must include all necessary details for processing.

Node.js

Restful API

REST (REpresentational State Transfer)

Transfer

The representation of the resource (state) is transferred between client and server.

Example: The client sends an HTTP request:

```
GET /users/1  Host: api.example.com
```

The server transfers the user's data (representation) in the response:

```
{ "id": 1, "name": "Alice", "isOnline": true }
```

RESTful

"A RESTful system adheres to the principles and constraints of REST architecture, ensuring efficient, stateless, and uniform client-server communication with well-defined resource representations."

Node.js

Restful API

RESTful

Design Principles of RESTful Systems

Client-Server Separation: Server-side and client-side responsibilities are separated so that each side can be implemented independently of the other.

Statelessness: Every request from the client must contain all the information the server needs to fulfill it, as no client context is stored on the server.

Cacheability: Responses should include information about whether they are cacheable to improve efficiency and reduce server load.

Uniform Interface: A consistent and standardized way to interact with resources, such as using HTTP methods like GET, POST, PUT, DELETE.

Layered System: There may be a number of different intermediaries between client and server.

Code on Demand (Optional): Servers can send executable code (like JavaScript) as part of the response.

Resource Identification: Each resource should have a unique identifier (e.g., a URL) to access and manipulate it.

Node.js

Restful API

*"A **RESTful API** is an application programming interface (API) that follows the principles and constraints of **REST**. It allows **communication between a client and server** using **standard HTTP methods** while focusing on accessing and manipulating resources through **uniform** and **stateless interactions**."*

REST APIs should support *CRUD operations* for managing resources:

Create:	Enables clients to add new resources.
Read:	Allows clients to retrieve resource data.
Update:	Enables modification of existing resources.
Delete:	Allows clients to remove resources.

Node.js

Restful API

HTTP Methods in REST:

POST:	It corresponds to the CREATE operation in the CRUD and used to create a new resource. To create a new resource, you need certain data, it is included in the request as a data header.
GET:	It corresponds to the READ operation in the CRUD and used to retrieve an existing resource.
PUT:	It corresponds to the UPDATE operation in the CRUD and used to update an existing resource. The data required for update is included in the request body.
DELETE:	It corresponds to the DELETE operation in the CRUD and used to delete an existing resource.

Node.js

Restful API

Example - *Creating RESTful API*:

Consider the following data of courses in a file named *courses.json*:

```
{
    "course1" : {
        "name" : "Database Management System",
        "code" : "BCS-501",
        "id": 1
     },

     "course2" : {
       "name" : "Web Technology",
       "code" : "BCS-502",
       "id": 2
    },

     "course3" : {
        "name" : "Design and Analysis of Algorithm",
        "code" : "BCS-503",
        "id": 3
     }
 }
```

Node.js

Restful API

Example - *Creating RESTful API*:

Our API will expose the following endpoints for the clients to perform CRUD operations on the courses.json file which has the collection of resources on the server:

Sr.No.	Route/ URI	HTTP Method	Request body	Result
1	/	GET	empty	Show list of all the users.
2	/	POST	JSON String	Add details of new user.
3	/:id	DELETE	JSON String	Delete an existing user.
4	/:id	GET	empty	Show details of a user.
5	/:id	PUT	JSON String	Update an existing user

Node.js

Restful API

Example - *Creating RESTful API*:

Code:

```
var express = require('express');
var app = express();
var fs = require("fs");
var bodyParser = require('body-parser')
app.use( bodyParser.json() );
app.use(bodyParser.urlencoded({  extended: true }));

// Adding a resourse - CREATE
app.post('/', function (req, res) {
    fs.readFile( __dirname + "/" + "courses.json", 'utf8', function (err, data) {
       var courses = JSON.parse( data );
       var course = req.body.course4;
       courses["course"+course.id] = course
       res.end( JSON.stringify(courses));
    });
 })
```

Node.js

Restful API

Example - *Creating RESTful API*:

Code:

```
//Reading all resources - READ
 app.get('/', function (req, res) {
    fs.readFile( __dirname + "/" + "courses.json", 'utf8', function (err, data) {
       res.end( data );
    });
 })

 // Reading a resources - READ
 app.get('/:id', function (req, res) {
     fs.readFile( __dirname + "/" + "courses.json", 'utf8', function (err, data) {
        var courses = JSON.parse( data );
        var course = courses["course" + req.params.id]
        res.end( JSON.stringify(course));
     });
  })
```

Node.js

Restful API

Example - *Creating RESTful API*:

Code:

```
// Updating a reource- UPDATE
app.put("/:id", function(req, res) {
    fs.readFile( __dirname + "/" + "courses.json", 'utf8', function (err, data) {

        var courses = JSON.parse( data );
        var id = "course"+req.params.id;
        courses[id]=req.body;
        res.end( JSON.stringify(courses));
    })
 })
 // Deleting a reource- DELETE
 app.delete('/:id', function (req, res) {
    fs.readFile( __dirname + "/" + "courses.json", 'utf8', function (err, data) {
        data = JSON.parse( data );
        var id = "course"+req.params.id;
        var course = data[id];
        delete data[ "course"+req.params.id];
        res.end( JSON.stringify(data));
    });
 })
```

Node.js

Restful API

Example - *Creating RESTful API*:

Code:

```
var server = app.listen(5000, function () {
   console.log("Express App running at http://127.0.0.1:5000/");
})
```

Node.js

Packaging

"Packaging in Node.js is the process of preparing our application, along with its dependencies, into a format that can be easily shared, installed, and run by others."

- Preparing Node.js apps for distribution.
- Ensures easy installation and use by others.

package.json

- package.json is a JSON file that holds metadata about your project.
- It includes the name, version, dependencies (e.g., express), and other information.
- You create it using npm init.

Node.js

Packaging

Managing Dependencies

- Use npm install to add packages needed in the project (e.g., Express).
- Dependencies are listed in package.json, which allows others to install everything needed with npm install.

Packaging Tools

Packaging Tools allow you to bundle a Node.js application, along with the Node.js runtime, into a standalone executable that can be run on various platforms (like Windows, macOS, or Linux).

Examples: Nexe, pkg, etc.

Node.js

Packaging

Publishing a package

- Publishing a package in Node.js is a step in the packaging process where you share your application with others by making it available on the npm registry for easy installation via npm install.
- To publish a package, you first ensure that your package.json is set up correctly and then run npm publish command. This command uploads your package to npm, making it publicly available for others to use.

AKTU

B.Tech 5th Sem

GATEWAY CLASSES

CS IT & CS Allied

Web Technology

UNIT-4 Lecture - 6

Today's Target

- Node.js with MongoDB
  - MongoDB
  - Create Database,
  - Create Collection,
  - Insert.

By Amol Sharma sir

- Pursuing Ph.D. from IIT (BHU)
- Wipro Certified Faculty (WCF)

Web Technology

UNIT – IV:

Enterprise Java Bean: Creating a Java Bean, Java Beans Properties, Types of beans, Stateful Session bean, Stateless Session bean, Entity bean.

Node.js: Introduction, Environment Setup, REPL Terminal, NPM (Node Package Manager) Callbacks Concept, Events, Packaging, Express Framework, Restful API.

Node.js with MongoDB: MongoDB Create Database, Create Collection, Insert, delete, update, join, sort, query.

Node.js with MongoDB

MongoDB

"MongoDB is a NoSQL, document-oriented database that stores data in the form of JSON-like documents."

JSON (JavaScript Object Notation) is a lightweight way to store and exchange data using a simple structure of key-value pairs. It's easy for both humans to read and computers to process.

Key Concepts

Database:	A container for collections.	Equivalent to a database in RDBMS.
Collection:	A group of documents.	Equivalent to a table in RDBMS.
Document:	A record in a collection.	Equivalent to a row of a table in RDBMS.
	Stored in BSON (Binary JSON), an optimized storage format.	

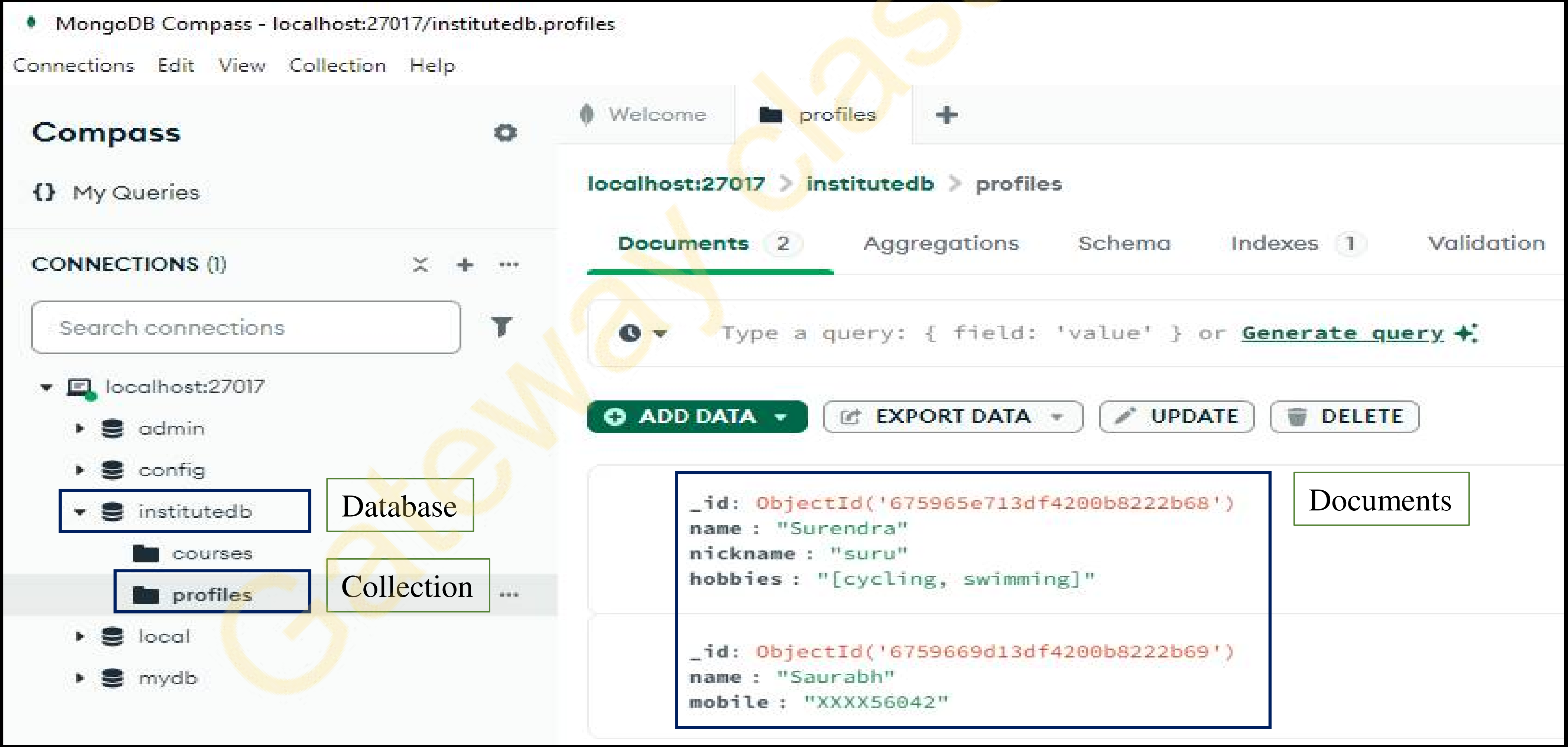
Features

- Schema-less database
- High performance for unstructured data
- Scalable, and distributed
- Popular for web and mobile applications

Node.js with MongoDB

MongoDB

Example:


MongoDB Compass - localhost:27017/institutedb.profiles

```
_id: ObjectId('675965e713df4200b8222b68')
name : "Surendra"
nickname : "suru"
hobbies : "[cycling, swimming]"

_id: ObjectId('6759669d13df4200b8222b69')
name : "Saurabh"
mobile : "XXXX56042"
```

Node.js with MongoDB

MongoDB

Advantages

- Handles large volumes of unstructured or semi-structured data.
  MongoDB can efficiently store and process diverse data formats without requiring a fixed schema, making it ideal for modern applications.
- Horizontal scaling with sharding.
  MongoDB supports sharding, allowing large datasets to be distributed across multiple servers, ensuring scalability and high availability.
- Faster development due to schema flexibility.
  The schema-less design of MongoDB enables developers to adapt to changing requirements quickly without database migrations.
- Ideal for real-time analytics.
  MongoDB's powerful querying capabilities and efficient storage make it suitable for analyzing large volumes of data in real time.

Used by companies like Facebook, Google, and Uber.

Node.js with MongoDB

Create Database

To create a database in MongoDB using Node.js, we start by creating a MongoClient object and specifying the connection URL, which includes both the IP address and port number. Using the db() method of the MongoClient object, we can access or create a database by providing its name. MongoDB will automatically create the database if it doesn't already exist and establish a connection to it.

Note: MongoDB only creates the database when we add a collection and insert data into it.

Create Collection

To create a collection in MongoDB using Node.js, we use the collection() method, which belongs to the db object. If the specified collection does not already exist, MongoDB will automatically create it when we insert the first document into it.

Insert Document

To insert a document into a collection, we use the insertOne() or insertMany() methods. These methods allow us to add single or multiple documents, respectively, into the collection. MongoDB will automatically create the collection when the first document is inserted, if it doesn't already exist.

Node.js with MongoDB

Example

```
const mongodb = require("mongodb");
mongoClient = mongodb.MongoClient;
const uri = "mongodb://localhost:27017/";
const client = new mongoClient(uri);

async function run() {
  try {
    const database = client.db('institutedb');
    const courses = database.collection('courses');

    const document = { name: 'WT', code: 'BCS-502' };
    const course = await courses.insertOne(document);
    console.log("1 document inserted in Collection '"+courses.collectionName+"' of Database
                                                   '"+database.databaseName+"'.");
  } finally {
    await client.close();
  }
}
run().catch(err=>{console.log("Error: "+err);});
```

Node.js with MongoDB

Example

1.Importing the MongoDB Library:

The program starts by importing the MongoDB library, which is required to interact with the MongoDB database.

```
const mongodb = require("mongodb");
```

2.Accessing the MongoClient:

The MongoClient object is extracted from the mongodb library. This object is used to connect to the MongoDB server and interact with the database.

```
mongoClient = mongodb.MongoClient;
```

3.Specifying the Connection URL:

The uri holds the connection string needed to connect to the MongoDB server. Here, the connection string "mongodb://localhost:27017/" tells the program to connect to the local MongoDB server running on port 27017.

```
const uri = "mongodb://localhost:27017/";
```

4.Creating a MongoClient Instance:

The client is created by calling the MongoClient constructor and passing the connection string (uri). This establishes a client object that will allow interaction with the MongoDB server.

```
const client = new mongoClient(uri);
```

Node.js with MongoDB

Example

5.Connecting to the Database:

The run() function is defined as an asynchronous function where the actual connection and operations with the database occur. Inside this function, we attempt to connect to the 'institutedb' Database:
The client.db('institutedb') method is called to connect to the database named 'institutedb'. This creates a reference to the database.

```
const database = client.db('institutedb’);
```

6.Creating a Collection:

To create or access a collection named 'courses', we use the collection() method of the database object. If the collection does not already exist, MongoDB creates it when the first document is inserted.

```
const courses = database.collection('courses’);
```

7.Inserting a Document:

A document is defined as an object with fields and values:

```
const document = { name: 'WT', code: 'BCS-502' };
```

This document is inserted into the 'courses' collection using the insertOne() method:

```
const course = await courses.insertOne(document);
```

After a successful insertion, the program prints a message to the console indicating that a document has been added to the 'courses' collection of the 'institutedb' database.

Node.js with MongoDB

Example

8.Closing the Connection:

After the database operation is complete, the program ensures that the client connection is closed by calling await client.close(). This step is important to release resources after completing the database interaction.

```
await client.close();
```

9.Handling Errors:

The program uses a catch block to handle any errors that might occur during the connection or database operations. If an error happens, it logs the error message to the console:

```
run().catch(err => { console.log("Error: " + err); });
```

AKTU

B.Tech 5th Sem

GATEWAY CLASSES

CS IT & CS Allied

Web Technology

UNIT-4 Lecture - 7

Today's Target

- ➢ Node.js with MongoDB
  - ➢ Insert
  - ➢ Update,
  - ➢ Delete,
  - ➢ Query (Find)
  - ➢ Sort,
  - ➢ Join.

By Amol Sharma sir

- Pursuing Ph.D. from IIT (BHU)
- Wipro Certified Faculty (WCF)

Node.js with MongoDB

Insert

"Insertion in MongoDB is used to add new documents to a collection."

Methods

MongoDB provides methods to insert one or more documents into a collection:

insertOne(document):	Inserts a single document into the collection.
insertMany(documents):	Inserts multiple documents into the collection.

Both methods return a response object with details about the inserted documents.

Examples

```
const result = await courses.insertOne(
{ name: "OOSD", code: "BCS-054" });
```

```
const result = await courses.insertMany([
  { name: "DBMS", code: "BCS-501" },
  { name: "WT", code: "BCS-502" },
  { name: "DAA", code: "BCS-503" }
]);
```

Node.js with MongoDB

Update

"Updation in MongoDB is used to modify existing documents in a collection."

Methods

MongoDB provides methods to *update a single document or multiple documents* using a filter:

updateOne(filter, update):	Searches for the first document matching the filter and updates it.
updateMany(filter, update):	Searches for all documents matching the filter and updates them.

These methods *return a response object* with details like the number of modified documents.

Examples

```
const result = await courses.updateOne(
    { name: "OOSD" },
    { $set: { name: "OOSD with C++" } }
);
```

```
const result = await courses.updateMany(
    { name: "OOSD" },
    { $set: { name: "OOSD with C++" } }
);
```

Node.js with MongoDB

Delete

"Deletion in MongoDB is used to remove documents from a collection."

Methods

MongoDB provides methods to *delete a single document or multiple documents* using a filter:

deleteOne(filter):	Searches for the first document matching the filter and deletes it.
deleteMany(filter):	Iterates through all documents matching the filter and deletes them.

These methods *return a response object* with details like the number of deleted documents.

Examples

```
const result = await courses.deleteOne({ name: "OOSD" });
```

```
const result = await courses.deleteMany({name: "OOSD"});
```

Node.js with MongoDB

Query (Find)

"Query in MongoDB is used to retrieve documents from a collection."

Methods

MongoDB provides methods to retrieve documents:

findOne():	Returns the first document in the collection.
find():	Returns all the documents in the collection.

These methods allow using query (filter) and projection objects:

Query (filter):	Specifies the filter criteria to match documents.
Projection:	Specifies which fields to include or exclude in the result.

These methods return a result object with details like the retrieved documents.

Examples

```
const result = await courses.findOne(
{ name: "OOSD" },
{ projection: { name: 1, _id: 0 } }
);
```

```
const result = await courses.find(
{ name: "OOSD" },
{ projection: { name: 1 } }
).toArray();
```

Node.js with MongoDB

Sort

"Sorting in MongoDB is used to order the documents based on specified field(s)."

Methods

MongoDB provides the sort() method to sort the documents:

sort():	Specifies the sorting order for the documents based on a field (or fields).
{field: 1}:	Sort in ascending order of the mentioned field.
{field: -1}:	Sort in descending order of the mentioned field.

Examples

```
const result = await courses.find(
{},
{projection:{_id:0,name:1,code:1}}
).sort({code:-1}).toArray();
```

```
const result = await courses.find(
{},
{projection:{_id:0,name:1,code:1}}
).sort({code:1}).toArray();
```

Node.js with MongoDB

Join

"In MongoDB, a join is performed using the $lookup operation to combine data from two collections."

Methods

MongoDB supports performing joins using the aggregate() method with the $lookup operation.

$lookup: Joins two collections by matching fields from one collection to another. It allows to merge data from the specified collections into a single result.

Example

```
const result = await courses.aggregate([
        {
          $lookup: {
            from: "courseAvailability", // The collection to join
            localField: "code",         // Field in the courses collection
            foreignField: "code",       // Field in the courseAvailability collection
            as: "availabilityDetails",  // Alias for the joined data
          },
        },
      ]).toArray();
```

Node.js with MongoDB

Example (Demonstrates the use of all the methos)

```
const mongodb = require("mongodb");
mongoClient = mongodb.MongoClient;

const uri = "mongodb://localhost:27017/";

const client = new mongoClient(uri);

async function run() {
  try {
    const database = client.db('institutedb');
    const courses = database.collection('courses');

    let result = await courses.insertOne({ name: "OOSD", code: "BCS-054" });
    console.log("Result [ insertOne() ] : " + JSON.stringify(result));

    result = await courses.insertMany([{ name: "DBMS", code: "BCS-501" },
    { name: "WT", code: "BCS-502" }, { name: "DAA", code: "BCS-503" }]);
    console.log("Result [ insertMany() ] : " + JSON.stringify(result));
```

Node.js with MongoDB

Example (Demonstrates the use of all the methos)

```
result = await courses.updateOne({ name: "OOSD" }, { $set: { name: "OOSD with C++" } });
console.log("Result [ updateOne() ] : " + JSON.stringify(result));

result = await courses.updateMany({ name: "DBMS" }, { $set: { name: "OOSD with C++" } });
console.log("Result [ updateMany() ] : " + JSON.stringify(result));

result = await courses.deleteOne({ name: "DAA" });
console.log("Result [ deleteOne() ] : " + JSON.stringify(result));

result = await courses.deleteMany({ name: "OOSD with C++" });
console.log("Result [ deleteMany() ] : " + JSON.stringify(result));

result = await courses.findOne();
console.log("Result [ findOne() ] : " + JSON.stringify(result));

result = await courses.find().toArray();
console.log("Result [ find() ] : " + JSON.stringify(result));
```

Node.js with MongoDB

Example (Demonstrates the use of all the methos)

```
    result = await courses.find({ name: "OOSD" }, { projection: { _id: 0, name: 1 }
}).toArray();
    console.log("Result [ find(query,projection) ] : " + JSON.stringify(result));

    result = await courses.find({}, { projection: { _id: 0, name: 1, code: 1 } }).sort({
code: -1 }).toArray();
    console.log("Result [ sort() ] : " + JSON.stringify(result));

    result = await courses.aggregate([
      {
        $lookup: {
          from: "courseAvailability", // The collection to join
          localField: "code",         // Field in the courses collection
          foreignField: "code",       // Field in the courseAvailability collection
          as: "availabilityDetails",  // Alias for the joined data
        },
      },
    ]).toArray();

    console.log("Result [ join ] : " + JSON.stringify(result));
```

Node.js with MongoDB

Example (Demonstrates the use of all the methos)

```
  } finally {
    // Ensures that the client will close when you finish/error
    await client.close();
  }
}
run().catch(err => { console.log("Error: " + err); });
```

Thank you