# Server-Side Programming - Comprehensive Student Guide

## ****Slide 2: Introduction to Server-Side Programming****

### What is Server-Side Programming?

Server-side programming is like the **"brain behind the website"**. While front-end (client-side) deals with what users see and interact with, server-side handles all the heavy lifting behind the scenes.

**Think of it like a restaurant:**

* **Client-side** = The dining area, menu, waiters (what customers see)
* **Server-side** = The kitchen, chef, inventory management (what customers don't see but makes everything work)

### Key Characteristics:

* **Runs on the server** - Code executes on powerful computers in data centers, not on user's browser
* **Handles business logic** - Processes data, makes decisions, enforces rules
* **Manages data** - Stores, retrieves, and manipulates information in databases
* **Provides security** - Protects sensitive information and validates user access

### Real-World Use Cases Explained:

**1. Querying the database**

* When you search for a product on Amazon, server-side code searches through millions of products in their database
* Example: User searches "laptop" → Server queries database → Returns matching products

**2. Operations over databases**

* Creating user accounts, updating profiles, deleting posts
* Example: When you update your Facebook status, server-side code saves it to their database

**3. Access/Write a file on server**

* Uploading profile pictures, downloading documents, generating reports
* Example: When you upload a photo to Instagram, server-side code saves it to their file system

**4. Interact with other servers**

* Payment processing, weather data, social media integration
* Example: When you pay with PayPal, your website's server communicates with PayPal's servers

**5. Structure web applications**

* Organizing code, managing user sessions, routing requests
* Example: When you visit different pages on a website, server-side code determines what content to show

**6. Process user input**

* Validating forms, sanitizing data, processing submissions
* Example: When you fill out a contact form, server-side code validates your email and saves the message

### Popular Server-Side Languages:

* **JavaScript (Node.js)** - Same language as front-end, great for full-stack development
* **Python** - Easy to learn, popular for data science and web development
* **PHP** - Powers many websites including WordPress
* **Java** - Enterprise-level applications, very robust
* **Ruby** - Known for developer-friendly syntax
* **C#** - Microsoft's language for Windows-based applications

## ****Slide 3: Key Features of Server-Side Programming****

### 1. Security - The Digital Bodyguard

**Authentication (Who are you?)**

* **Login/Logout systems**: Verifying user credentials
* **Session management**: Keeping users logged in across pages
* **Password hashing**: Storing passwords securely (never in plain text)
* **Two-factor authentication**: Adding extra security layers

**Real Example**: When you log into your bank account, the server:

1. Checks your username/password against the database
2. Creates a secure session
3. Ensures only you can access your account information

**Input Validation and Sanitization**

* **Preventing malicious code**: Blocking harmful scripts
* **Data type checking**: Ensuring numbers are numbers, emails are valid
* **SQL injection prevention**: Protecting database from attacks
* **Cross-site scripting (XSS) protection**: Preventing malicious scripts

**Example**: If someone tries to enter <script>alert('hack')</script> in a form, server-side code removes or neutralizes it.

### 2. Database Management - The Digital Filing Cabinet

**Reading and Writing Data**

* **CRUD Operations**: Create, Read, Update, Delete
* **User profiles**: Storing personal information, preferences, history
* **Content management**: Posts, comments, articles, media files
* **Transaction records**: Orders, payments, logs

**Database Types:**

* **SQL Databases** (MySQL, PostgreSQL): Structured data with relationships
	+ Good for: E-commerce, banking, traditional business applications
* **NoSQL Databases** (MongoDB, Redis): Flexible, unstructured data
	+ Good for: Social media, real-time applications, big data

**Real Example**: Twitter's database management

* Stores billions of tweets
* Manages user relationships (followers/following)
* Handles likes, retweets, comments
* Provides real-time search across all content

### 3. Dynamic Content Generation - The Personalization Engine

**What is Dynamic Content?** Content that changes based on:

* **User identity**: Personalized recommendations
* **User behavior**: Recently viewed items
* **Time/date**: Current news, weather updates
* **Location**: Local events, nearby restaurants
* **Device type**: Mobile vs desktop layouts

**Examples:**

* **Netflix homepage**: Shows different movies based on your viewing history
* **Amazon product recommendations**: "Customers who bought this also bought..."
* **News websites**: Different headlines based on your location and interests
* **Social media feeds**: Personalized timeline based on your connections and activity

### 4. API Development - The Communication Bridge

**What are APIs?** APIs (Application Programming Interfaces) are like waiters in a restaurant:

* They take your order (request)
* Communicate with the kitchen (server/database)
* Bring back your food (response)

**RESTful APIs** follow specific rules for communication:

* **GET**: Retrieve data (like reading a menu)
* **POST**: Create new data (like placing an order)
* **PUT**: Update existing data (like modifying an order)
* **DELETE**: Remove data (like canceling an order)

**Real Example**: Mobile app communication

* Instagram app sends API request: "Get posts from users I follow"
* Server processes request, queries database
* Returns JSON data with posts, images, comments
* App displays the feed to user

## ****Slide 4: Overview of Server-Side Languages (Node.js Focus)****

### What Do Server-Side Languages Handle?

**Business Logic**

* **Decision making**: If user is premium, show ad-free content
* **Calculations**: Computing taxes, discounts, shipping costs
* **Workflow management**: Order processing, approval workflows
* **Rule enforcement**: Age restrictions, access permissions

**Database Operations**

* **Connection management**: Establishing secure database connections
* **Query optimization**: Making database searches faster
* **Data relationships**: Managing how different data pieces connect
* **Backup and recovery**: Ensuring data safety

**API Management**

* **Request routing**: Directing different requests to appropriate handlers
* **Response formatting**: Converting data to JSON, XML, or other formats
* **Rate limiting**: Preventing abuse by limiting requests per user
* **Version control**: Managing different API versions

**Authentication Systems**

* **User registration**: Creating new accounts
* **Password recovery**: Reset password functionality
* **Role-based access**: Different permissions for users, admins, moderators
* **OAuth integration**: Login with Google, Facebook, etc.

**File Operations**

* **Upload handling**: Processing images, documents, videos
* **File storage**: Organizing files in folders, cloud storage
* **Compression**: Reducing file sizes for faster loading
* **Format conversion**: Converting images, generating thumbnails

**Server Communication**

* **HTTP requests**: Communicating with other websites/services
* **Email services**: Sending notifications, newsletters
* **SMS/Push notifications**: Mobile app notifications
* **Third-party integrations**: Payment gateways, mapping services

### Why Node.js?

**JavaScript Everywhere**

* Same language for front-end and back-end
* Easier for developers to learn and maintain
* Shared code between client and server
* Single development team can handle full project

**Popular Frameworks**

* **Express.js**: Web application framework
* **NestJS**: Enterprise-level applications
* **Koa.js**: Modern, lightweight framework

## ****Slide 5: Node.js Deep Dive****

### What is Node.js?

**Technical Definition**: Node.js is a server-side runtime environment that allows developers to use JavaScript to build scalable and efficient web applications.

**Simple Explanation**: Node.js lets you use JavaScript (normally only for websites) to build server applications, like creating APIs, handling databases, and managing file operations.

### Built on Chrome's V8 JavaScript Engine

**What is V8?**

* Google's open-source JavaScript engine
* Same engine that powers Chrome browser
* Compiles JavaScript directly to machine code
* Extremely fast performance

**Why This Matters:**

* JavaScript runs as fast as compiled languages like C++
* Continuous performance improvements from Google
* Proven reliability (powers billions of web pages)

### Ideal Use Cases:

**1. Fast, Asynchronous, Scalable Network Applications**

* **Chat applications**: WhatsApp Web, Slack, Discord
* **Real-time gaming**: Multiplayer online games
* **Collaborative tools**: Google Docs, Figma, Notion
* **Live streaming**: Twitch chat, YouTube live comments

**2. Web Servers for HTTP Requests/Responses**

* **E-commerce sites**: Handling product catalogs, shopping carts
* **Content management**: WordPress alternatives, blogging platforms
* **Corporate websites**: Company portals, customer dashboards

**3. REST APIs**

* **Mobile app backends**: Instagram, TikTok APIs
* **Microservices**: Breaking large applications into smaller services
* **Integration APIs**: Connecting different software systems

**4. Real-time Applications**

* **Online gaming**: Real-time multiplayer games
* **Live dashboards**: Stock trading, analytics dashboards
* **IoT applications**: Smart home devices, sensor networks

**5. IoT (Internet of Things) Solutions**

* **Smart home systems**: Controlling lights, temperature, security
* **Industrial monitoring**: Factory equipment, environmental sensors
* **Wearable devices**: Fitness trackers, smartwatches

### Node.js Architecture Explained:

**Single-threaded Event Loop**

* **Traditional servers**: Create new thread for each request (heavy memory usage)
* **Node.js approach**: One thread handles all requests using events (lightweight)

**Analogy**:

* **Traditional**: Like having a separate cashier for each customer (expensive)
* **Node.js**: Like having one very efficient cashier who can handle multiple orders simultaneously

**Callback-based and Promise-based**

* **Callbacks**: Functions that run after an operation completes
* **Promises**: Modern way to handle asynchronous operations
* **Async/Await**: Even cleaner syntax for handling asynchronous code

**Module System**

* **Built-in modules**: File system, HTTP, crypto, path utilities
* **NPM modules**: Over 1 million packages available
* **Custom modules**: Create reusable code components

**Database Compatibility**

* **NoSQL**: MongoDB, CouchDB, Redis
* **SQL**: MySQL, PostgreSQL, SQLite
* **Cloud databases**: AWS DynamoDB, Firebase, Google Cloud Firestore

## ****Slide 6: Key Features of Node.js****

### 1. Unified Language

**Benefits:**

* **Learning curve**: Developers only need to master one language
* **Code sharing**: Common utilities can be used on both client and server
* **Team efficiency**: Front-end developers can contribute to back-end
* **Faster development**: No context switching between languages

**Real Example**: A developer building an e-commerce site can:

* Use JavaScript for website interactivity (client-side)
* Use Node.js for handling orders and payments (server-side)
* Share data validation code between both sides

### 2. Single-threaded Architecture

**How it Works:**

Traditional Server (Multi-threaded):

Request 1 → Thread 1 (2MB memory)

Request 2 → Thread 2 (2MB memory)

Request 3 → Thread 3 (2MB memory)

1000 requests = 2GB memory usage

Node.js (Single-threaded):

All requests → Event Loop → Single Thread

1000 requests = ~20MB memory usage

**Event Loop Process:**

1. Request comes in
2. If it's I/O operation (database, file read), delegate to system
3. Continue processing other requests
4. When I/O completes, run callback
5. Send response back to client

**Lightweight Concurrency:**

* Can handle 10,000+ concurrent connections
* Much lower memory usage per connection
* Perfect for I/O-intensive applications

### 3. NPM Ecosystem

**What is NPM?**

* **Node Package Manager**: World's largest software registry
* **Over 1 million packages**: Pre-built solutions for common problems
* **Easy installation**: npm install package-name
* **Dependency management**: Automatically handles package dependencies

**Popular Packages:**

* **Express.js**: Web framework (50M+ weekly downloads)
* **Lodash**: Utility functions
* **Moment.js**: Date/time manipulation
* **Mongoose**: MongoDB object modeling
* **Socket.io**: Real-time communication
* **Passport.js**: Authentication strategies

**Benefits:**

* **Faster development**: Don't reinvent the wheel
* **Community support**: Millions of developers contributing
* **Quality packages**: Popular packages are well-tested
* **Regular updates**: Security patches and new features

### 4. Fast & Efficient Performance

**V8 Engine Optimization:**

* **Just-in-time compilation**: Converts JavaScript to machine code
* **Hidden class optimization**: Optimizes object property access
* **Inline caching**: Speeds up property lookups
* **Garbage collection**: Automatic memory management

**Performance Benchmarks:**

* **Netflix**: Reduced startup time by 70% switching to Node.js
* **PayPal**: 2x faster response times compared to Java
* **Walmart**: Handles 500M page views per month

### 5. Asynchronous, Non-blocking I/O

**Blocking vs Non-blocking:**

**Blocking (Traditional):**

1. Read file from disk (wait 100ms)

2. Process file (wait 50ms)

3. Save to database (wait 200ms)

Total time: 350ms for one request

**Non-blocking (Node.js):**

1. Start reading file → Continue to next request

2. Start processing file → Continue to next request

3. Start saving to database → Continue to next request

Handle multiple requests simultaneously

**Real-world Impact:**

* **Improved scalability**: Handle more users with same hardware
* **Better performance**: Faster response times
* **Cost efficiency**: Reduced server requirements

### 6. Event-Driven Architecture

**How Events Work:**

// When user uploads file

fileUpload.on('progress', (bytesUploaded) => {

 console.log(`Uploaded: ${bytesUploaded} bytes`);

});

// When upload completes

fileUpload.on('complete', (file) => {

 console.log('File uploaded successfully');

 processImage(file);

});

// If error occurs

fileUpload.on('error', (error) => {

 console.log('Upload failed:', error);

});

**Event Types:**

* **HTTP requests**: New user visits website
* **Database operations**: Query completed
* **File operations**: File read/write finished
* **Timer events**: Scheduled tasks execution
* **Custom events**: Application-specific events

### 7. Cross-Platform Compatibility

**Supported Platforms:**

* **Windows**: Desktop and server versions
* **Linux**: All major distributions (Ubuntu, CentOS, Debian)
* **macOS**: Development and production environments
* **Docker**: Containerized deployments
* **Cloud platforms**: AWS, Google Cloud, Azure, Heroku

**Benefits:**

* **Development flexibility**: Work on any operating system
* **Deployment options**: Choose best hosting environment
* **Team collaboration**: Different OS preferences supported
* **Cost optimization**: Use existing infrastructure

## ****Slides 7-10: RESTful Web Services and APIs****

### What are RESTful Web Services?

**REST = Representational State Transfer**

**Simple Explanation**: REST is like a universal language that allows different applications to communicate with each other over the internet.

**Restaurant Analogy:**

* **Menu** = API documentation (what's available)
* **Order** = API request (what you want)
* **Kitchen** = Server (processes your order)
* **Meal** = API response (what you get back)

### REST Constraints Explained:

### 1. Client-Server Architecture

**Separation of Concerns:**

* **Client (Frontend)**: Handles user interface and user interactions
* **Server (Backend)**: Manages business logic and data storage

**Benefits:**

* **Independent development**: Front-end and back-end teams can work separately
* **Easy to maintain**: Changes to UI don't affect server logic
* **Scalable**: Can have multiple clients (web, mobile, desktop) using same server
* **Technology flexibility**: Client can use React, server can use Node.js

**Real Example**: Instagram

* **Mobile app** (iOS/Android client) communicates with Instagram servers
* **Web interface** (browser client) uses same server APIs
* **Third-party apps** can also integrate using same API

### 2. Statelessness

**What it means**: Server doesn't remember previous requests from the client.

**Every request must include:**

* **Authentication token**: Who is making the request
* **All necessary data**: What operation to perform
* **Context information**: Any relevant details

**Example:**

❌ Stateful (Bad):

Request 1: "Login as john@email.com"

Request 2: "Get my profile" (server remembers it's John)

✅ Stateless (Good):

Request 1: "Login as john@email.com" → Returns token: "abc123"

Request 2: "Get profile for token abc123"

**Advantages:**

* **Scales easily**: No need to store session data
* **No session management**: Simpler server architecture
* **Improved reliability**: Server restart doesn't lose session data
* **Better performance**: No memory used for session storage

### 3. Cacheability

**What is Caching?** Storing frequently requested data in fast-access memory to avoid repeated server calls.

**Cache Levels:**

* **Browser cache**: User's browser stores images, CSS, JavaScript
* **CDN cache**: Content Delivery Networks store static files globally
* **Server cache**: Database query results stored in memory
* **Database cache**: Frequently accessed data kept in RAM

**HTTP Cache Headers:**

Cache-Control: max-age=3600 (Cache for 1 hour)

Cache-Control: no-cache (Always validate with server)

ETag: "abc123" (Version identifier)

**Benefits:**

* **Reduces server load**: Fewer requests to process
* **Faster response times**: Data served from cache is much faster
* **Better user experience**: Pages load quickly
* **Cost savings**: Less bandwidth and server resources used

### 4. Uniform Interface

**Consistent API Design:**

**Resource Identification:**

✅ Good URLs:

GET /users/123 (Get user with ID 123)

GET /users/123/posts (Get posts by user 123)

GET /posts/456 (Get post with ID 456)

❌ Bad URLs:

GET /getUserById?id=123

GET /getPostsForUser?userId=123

**HTTP Methods:**

GET /users → Get all users

GET /users/123 → Get specific user

POST /users → Create new user

PUT /users/123 → Update entire user

PATCH /users/123 → Update part of user

DELETE /users/123 → Delete user

**Self-Descriptive Messages:**

{

 "id": 123,

 "name": "John Doe",

 "email": "john@email.com",

 "created\_at": "2024-01-15T10:30:00Z",

 "\_links": {

 "self": "/users/123",

 "posts": "/users/123/posts",

 "friends": "/users/123/friends"

 }

}

### 5. Layered System

**Architecture Layers:**

Client → Load Balancer → API Gateway → Microservices → Database

**Each Layer's Role:**

* **Load Balancer**: Distributes requests across multiple servers
* **API Gateway**: Handles authentication, rate limiting, logging
* **Microservices**: Specialized services for different functions
* **Caching Layer**: Redis/Memcached for fast data access
* **Database Layer**: Persistent data storage

**Benefits:**

* **Improved scalability**: Add/remove layers as needed
* **Enhanced security**: Multiple security checkpoints
* **Better performance**: Caching and load balancing
* **Easier maintenance**: Update individual layers without affecting others

### 6. Code on Demand (Optional)

**What it means**: Server can send executable code to extend client functionality.

**Examples:**

* **JavaScript widgets**: Sending interactive components
* **Browser plugins**: Adobe Flash, Java applets (legacy)
* **Progressive Web Apps**: Service workers for offline functionality
* **Dynamic forms**: Form validation rules sent from server

**Modern Implementation:**

// Server sends JavaScript code

{

 "data": {...},

 "clientCode": "function validateForm(data) { ... }"

}

// Client executes the code

eval(response.clientCode);

validateForm(formData);

**Benefits:**

* **Reduces client complexity**: Client doesn't need all possible functionality built-in
* **Dynamic functionality**: Add features without updating client
* **Flexibility**: Customize behavior based on user or context

## ****Slide 11: Simple Node.js Server Example****

Let's break down this code step by step:

const http = require('http');

const data = {

 message: 'Hello, this is simple data from Node.js!',

 author: 'Karthik',

 version: '1.0',

 status: 'success',

 time: new Date()

};

const server = http.createServer((req, res) => {

 res.setHeader('Content-Type', 'application/json');

 res.end(JSON.stringify(data));

});

server.listen(3000, () => {

 console.log('Server running at http://localhost:3000/');

});

### Code Explanation:

**1. Import HTTP Module:**

const http = require('http');

* require('http'): Imports Node.js built-in HTTP module
* This module provides functionality to create HTTP servers and clients
* No installation needed - it's part of Node.js core

**2. Define Response Data:**

const data = {

 message: 'Hello, this is simple data from Node.js!',

 author: 'Karthik',

 version: '1.0',

 status: 'success',

 time: new Date()

};

* Creates a JavaScript object with sample data
* new Date(): Creates current timestamp
* This data will be sent to all clients as JSON

**3. Create HTTP Server:**

const server = http.createServer((req, res) => {

 // Request handler function

});

* createServer(): Creates a new HTTP server instance
* Takes a callback function that runs for each request
* req: Request object (contains client data)
* res: Response object (used to send data back)

**4. Set Response Headers:**

res.setHeader('Content-Type', 'application/json');

* Tells the client that response will be JSON format
* Browser/client knows how to handle the response
* Other common content types: 'text/html', 'text/plain', 'image/png'

**5. Send Response:**

res.end(JSON.stringify(data));

* JSON.stringify(): Converts JavaScript object to JSON string
* res.end(): Sends the response and closes the connection
* Client receives the JSON data

**6. Start the Server:**

server.listen(3000, () => {

 console.log('Server running at http://localhost:3000/');

});

* listen(3000): Server starts listening on port 3000
* Callback function runs when server successfully starts
* localhost:3000: Local address where server is accessible

### What Happens When You Run This Code:

1. **Start the server**: node server.js
2. **Server starts listening**: On port 3000
3. **Open browser**: Go to http://localhost:3000
4. **Server receives request**: Browser sends HTTP GET request
5. **Server processes request**: Runs the callback function
6. **Server sends response**: JSON data with proper headers
7. **Browser displays**: JSON data (formatted by browser)

### Expected Output in Browser:

{

 "message": "Hello, this is simple data from Node.js!",

 "author": "Karthik",

 "version": "1.0",

 "status": "success",

 "time": "2024-08-03T14:30:45.123Z"

}

### Enhancing the Basic Server:

**Handle Different Routes:**

const server = http.createServer((req, res) => {

 const url = req.url;

 if (url === '/') {

 res.setHeader('Content-Type', 'application/json');

 res.end(JSON.stringify(data));

 } else if (url === '/about') {

 res.setHeader('Content-Type', 'text/html');

 res.end('<h1>About Page</h1>');

 } else {

 res.statusCode = 404;

 res.end('Page not found');

 }

});

**Handle Different HTTP Methods:**

const server = http.createServer((req, res) => {

 const method = req.method;

 if (method === 'GET') {

 // Handle GET requests

 } else if (method === 'POST') {

 // Handle POST requests

 }

});

## ****Slides 12-14: Database Integration (MySQL)****

### Why Database Integration?

**Real Applications Need Data Persistence:**

* **User accounts**: Store login credentials, profiles, preferences
* **Content**: Posts, comments, articles, media files
* **Business data**: Orders, inventory, transactions, analytics
* **Session data**: Shopping carts, user activities, temporary data

### Step-by-Step MySQL Integration:

### Step 1: Create Project Directory

mkdir mysql-demo

cd mysql-demo

**What this does:**

* Creates a new folder for your project
* Keeps your project organized and separate from other projects
* Good practice: one folder per project

### Step 2: Initialize Node.js Project

npm init -y

**What this creates:**

* package.json file: Contains project metadata
* Project name, version, description
* Dependencies list (packages your project uses)
* Scripts for running/testing your project

**Sample package.json:**

{

 "name": "mysql-demo",

 "version": "1.0.0",

 "description": "Node.js MySQL integration demo",

 "main": "index.js",

 "scripts": {

 "start": "node index.js",

 "dev": "nodemon index.js"

 },

 "dependencies": {}

}

### Step 3: Install MySQL Package

npm install mysql

**What this does:**

* Downloads MySQL driver for Node.js
* Adds dependency to package.json
* Creates node\_modules folder with package files
* Allows your Node.js code to communicate with MySQL database

**Alternative Modern Package:**

npm install mysql2

# mysql2 is faster and supports promises/async-await

### Step 4: Require MySQL Module

const mysql = require('mysql');

**What this does:**

* Imports MySQL functionality into your code
* Makes database connection and query functions available
* Must be at the top of your JavaScript file

### Step 5: Create Database Connection

const db = mysql.createConnection({

 host: 'localhost', // Database server location

 user: 'root', // Database username

 password: '', // Database password (empty for local dev)

 database: 'testdb' // Database name to connect to

});

**Connection Parameters Explained:**

* **host**: Where MySQL server is running
	+ localhost: Your local computer
	+ 192.168.1.100: Another computer on network
	+ database.example.com: Remote server
* **user**: MySQL username (default: 'root')
* **password**: MySQL password (set during MySQL installation)
* **database**: Which database to use (must exist in MySQL)

### Step 6: Connect to Database

db.connect((err) => {

 if (err) {

 console.log('Database connection failed:', err);

 } else {

 console.log('MySQL Connected!');

 }

});

**Error Handling:**

* **Connection successful**: Code continues execution
* **Connection failed**: Common issues:
	+ MySQL server not running
	+ Wrong username/password
	+ Database doesn't exist
	+ Firewall blocking connection

### Step 7: Execute INSERT Query

// Insert new user into database

db.query("INSERT INTO users (name, email) VALUES (?, ?)",

 ['John Doe', 'john@example.com'],

 (err, result) => {

 if (err) {

 console.log('Insert failed:', err);

 } else {

 console.log('User inserted with ID:', result.insertId);

 }

});

**Parameterized Queries:**

* ? placeholders prevent SQL injection attacks
* Values are safely escaped and inserted
* Much safer than string concatenation

**INSERT Query Breakdown:**

* **Table**: users (must exist in database)
* **Columns**: name, email (must exist in table)
* **Values**: ['John Doe', 'john@example.com']
* **Result**: Contains insertId (new record's ID)

### Step 8: Execute SELECT Query

// Retrieve users from database

db.query('SELECT \* FROM users', (err, rows) => {

 if (err) {

 console.log('Select failed:', err);

 } else {

 console.log('User Data:', rows);

 // rows is an array of user objects

 }

});

**SELECT Query Results:**

// Example result:

[

 { id: 1, name: 'John Doe', email: 'john@example.com' },

 { id: 2, name: 'Jane Smith', email: 'jane@example.com' }

]

### Complete Working Example:

const mysql = require('mysql');

// Database connection configuration

const db = mysql.createConnection({

 host: 'localhost',

 user: 'root',

 password: '',

 database: 'testdb'

});

// Connect to database

db.connect((err) => {

 if (err) {

 console.log('Connection failed:', err);

 return;

 }

 console.log('MySQL Connected!');

 // Create users table if it doesn't exist

 const createTable = `

 CREATE TABLE IF NOT EXISTS users (

 id INT AUTO\_INCREMENT PRIMARY KEY,

 name VARCHAR(100) NOT NULL,

 email VARCHAR(100) UNIQUE NOT NULL,

 created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

 )

 `;

 db.query(createTable, (err) => {

 if (err) {

 console.log('Table creation failed:', err);

 } else {

 console.log('Users table ready');

 // Insert sample data

 insertUser('John Doe', 'john@example.com');

 insertUser('Jane Smith', 'jane@example.com');

 // Retrieve and display data

 setTimeout(() => {

 getAllUsers();

 }, 1000);

 }

 });

});

// Function to insert new user

function insertUser(name, email) {

 const query = 'INSERT INTO users (name, email) VALUES (?, ?)';

 db.query(query, [name, email], (err, result) => {

 if (err) {

 console.log('Insert error:', err.message);

 } else {

 console.log(`User ${name} inserted with ID: ${result.insertId}`);

 }

 });

}

// Function to get all users

function getAllUsers() {

 db.query('SELECT \* FROM users', (err, rows) => {

 if (err) {

 console.log('Select error:', err);

 } else {

 console.log('All users:');

 rows.forEach(user => {

 console.log(`ID: ${user.id}, Name: ${user.name}, Email: ${user.email}`);

 });

 }

 });

}

// Graceful shutdown

process.on('SIGINT', () => {

 console.log('\nClosing database connection...');

 db.end(() => {

 console.log('Database connection closed.');

 process.exit(0);

 });

});

### Database Best Practices:

**1. Connection Pooling:**

const mysql = require('mysql');

const pool = mysql.createPool({

 connectionLimit: 10,

 host: 'localhost',

 user: 'root',

 password: '',

 database: 'testdb'

});

// Use pool instead of single connection

pool.query('SELECT \* FROM users', callback);

**2. Environment Variables:**