



SERVER-SIDE DEVELOPMENT WITH NODE.JS

Server-side Development with Node.js

© 2024 Aptech Limited

All rights reserved.

No part of this book may be reproduced or copied in any form or by any means – graphic, electronic, or mechanical, including photocopying, recording, taping, or storing in information retrieval system or sent or transferred without the prior written permission of copyright owner Aptech Limited.

All trademarks acknowledged.

APTECH LIMITED

Contact E-mail: ov-support@onlinevarsity.com

Edition 1 – 2024



Onlinevarsity



Preface

Node.js is an open-source JavaScript runtime environment that facilitates server-side application development using JavaScript. It supports both synchronous and asynchronous programming paradigms. Node.js eradicates the burden of managing thread concurrency by allowing a single server thread to efficiently handle concurrent connections. It can be used to build effective back-end APIs for mobile and Web applications. In this Learner's Guide, you will learn the basics of server-side scripting required to create Web applications. You will also learn to use different modules and packages available with various Node.js frameworks such as Express.js. You will learn to create fully functional Web applications and deploy them using Render.

This book is the result of a concentrated effort of the Design Team, which is continuously striving to bring you the best and the latest in Information Technology. The process of design has been a part of the ISO 9001 certification for Aptech-IT Division, Education Support Services. As part of Aptech's quality drive, this team does intensive research and curriculum enrichment to keep it in line with industry trends.

We will be glad to receive your suggestions.

Design Team



**MANY
COURSES
ONE
PLATFORM**



Onlinevarsity App for Android devices

Download from Google Play Store

Table of Contents

Sessions

Session 1 – Introduction to Server-side Development

Session 2 – Node.js Essentials

Session 3 – Modules and Packages in Node.js

Session 4 – Built-in Modules

Session 5 – More Built-in and Local Modules

Session 6 – Introduction to Express.js Framework

Session 7 – Asynchronous and Synchronous Programming

Session 8 – RESTful and HTTP APIs

Session 9 - Integration of Node.js with MongoDB

Session 10 – Building Websites Using Node.js

Session 11 – Node.js in Action (Application Development) – I

Session 12 – Node.js in Action (Application Development) – II

Appendix A

Appendix B

Appendix C



SESSION 1

INTRODUCTION TO SERVER-SIDE DEVELOPMENT

Learning Objectives

In this session, students will learn to:

- Outline the client-server architecture
- Explain the significance of server-side programming
- Explain a Web server and its architecture
- Describe Node.js, its features, and its installation process

Today, Web applications have become an integral part of everyone's lives. Web applications are used to perform tasks such as shopping for various items, buying groceries, maintaining health data, booking travel tickets and hotels, and banking. These Web applications work on the client-server architecture model. In this architecture, data processing happens on the server, so the load on the users' device, such as a computer or mobile device, is minimal.

The client-server architecture provides security features, such as login authentication, to prevent unauthorized access to users' data. It also provides personalization features, which personalize the data displayed based on the user preferences. This feature enhances the user experience with the Web application. For the Web application to work, server-side programming is employed to create programs that process the user requests and respond to user actions.

This session will provide an overview of the client-server architecture and the importance of server-side programming in the smooth execution of Web applications. It will also introduce Node.js, which is a server-side programming environment that is widely used to develop Web applications. Finally, it will explain the steps to install Node.js.

1.1 What is Client-Server Architecture?

Consider that a user goes to an ATM to withdraw some cash. The user inserts the card into the ATM machine, which, in turn, reads the card, and prompts the user to enter the PIN. When the user enters a valid PIN, the user can perform various tasks such as viewing the balance amount in the account or withdrawing an amount. When the user selects the cash withdrawal option, the user is prompted to enter the amount required. If that amount is more than the amount available in the account, the ATM machine will display an error. If the amount is less than the amount available in the account, the required cash will be dispensed. Multiple users use the same ATM machine to access accounts in various banks.

In this case, the user interacts with the ATM machine. Behind the scenes, there are a number of processes happening. When the user enters the card, the ATM machine reads the card and sends the card details to the bank's server. The server, in turn, checks if the card is valid. If the card is valid, the server sends a signal to the ATM machine, which then prompts the user for the PIN. After the user enters the PIN, the PIN is sent to the server for validation. Only if the server sends a signal stating that PIN is valid, the ATM machine will display various options which user can use to interact with the bank account. When the user wants to withdraw cash and enters amount to withdraw, that amount is sent to the server to check if there is enough balance in the account. Based on the response that the server sends, the ATM machine dispenses the amount or gives an error.

Here, the ATM machine and the banks' server together form the client-server architecture.

Figure 1.1 depicts the client-server architecture.

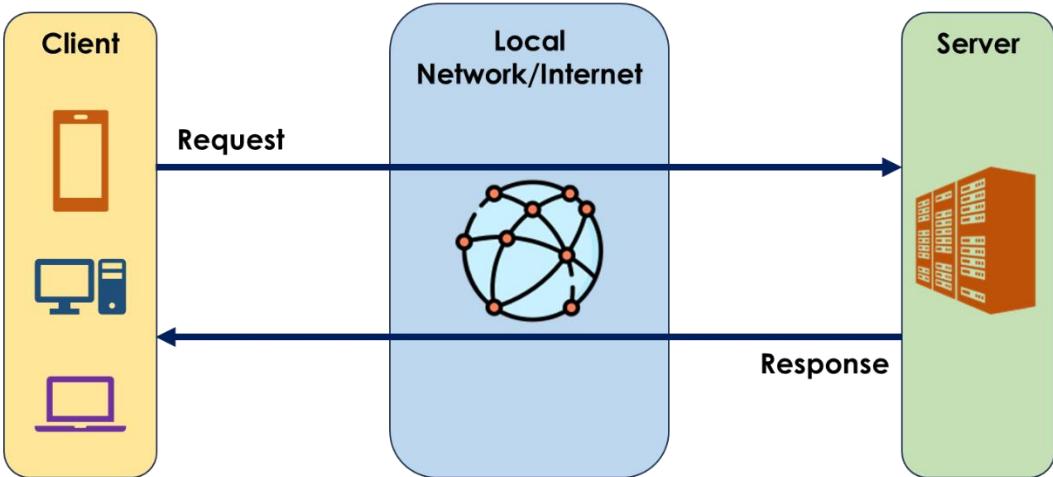


Figure 1.1: Client-Server Architecture

As shown in Figure 1.1 the main components of client-server architecture are as follows:

Client	Server	Network
Known as workstations or frontend. Used by users to provide instructions for performing various tasks. Connects with the server to deliver the user instructions and pass on the responses from the server to the user. Is governed by policies defined by the server. Can be desktop, laptop, tab, or mobile.	Known as backend. Is a centralized repository of files, programs, databases, and network policies. Has large storage space and memory to cater to requests coming in from several clients. Can be a file server, mail server, database server, Web server, or domain controller.	Is the medium through which clients and servers connect. Is made up of various networking devices that facilitate the communication between clients and server. Includes networking devices such as hubs, repeaters, bridges, and routers.

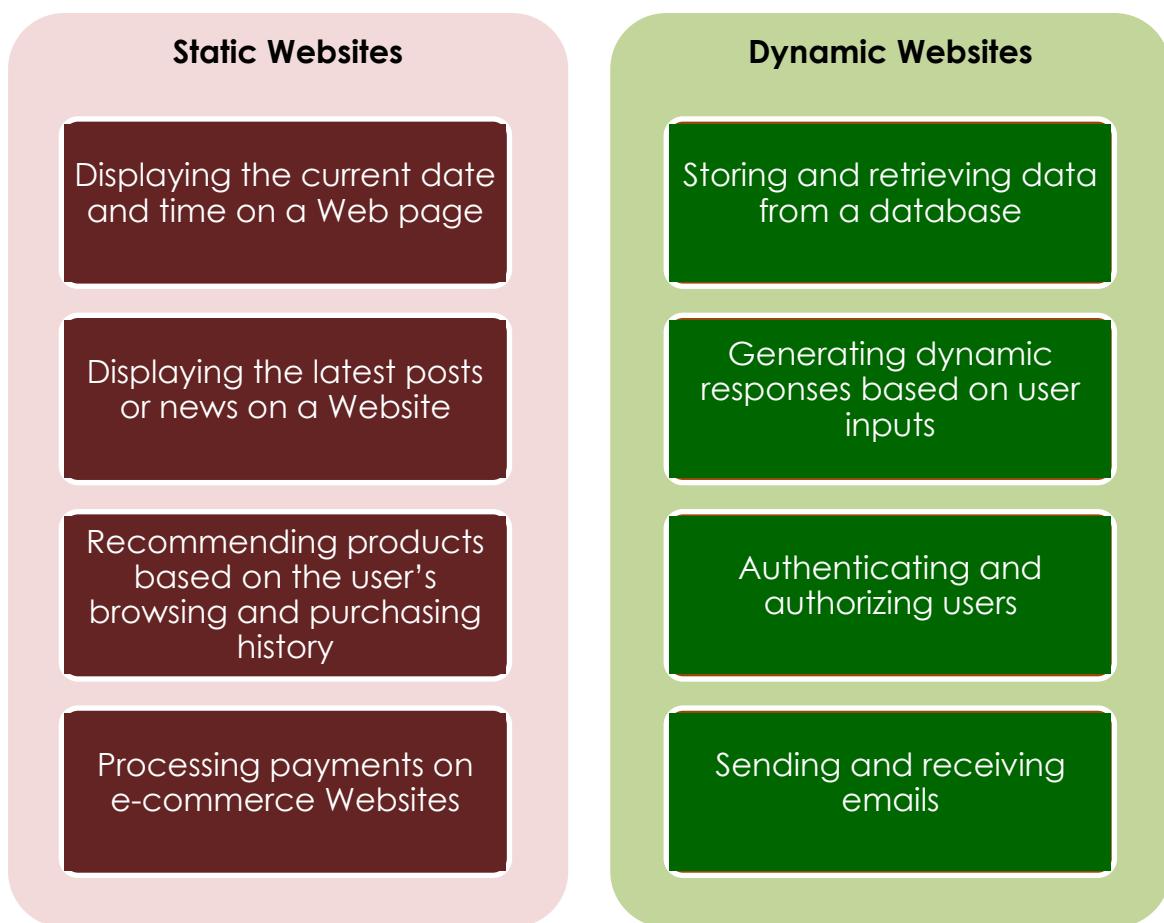
1.2 Server-side Programming

As discussed already, the server performs tasks that it receives as instructions from users through the client. To perform these tasks, various programs run on the server.

Developing these programs that run on the server is termed **server-side programming**.

To understand this better, consider the example of the ATM. There are programs that run on server for authenticating users' card and PIN, checking balance in the account, and updating the balance in the account after the cash withdrawal. This is also referred to as back-end processing. The user is not required to know what is happening on the server. Users are only required to provide instructions through the client.

Server-side programming is used to add dynamic features to static Websites or to create dynamic Web pages. Examples of server-side programming uses are as follows:



Server-side Technologies/Programming Languages

Various programming languages and technologies can be used to create server-side programs.

Some of these are as follows:

Python	It is a simple programming language that can be used to create Web applications. It is also extensively used in the field of data science and machine learning.
PHP	It is a scripting language widely used in the development of Web applications. It provides a large library of functions and classes that help make the applications fast and efficient.
Java	It is a programming language that can be used to create reliable and scalable Web applications. It is widely used to develop enterprise applications.
Node.js	It is a JavaScript runtime environment that facilitates developers to use JavaScript for server-side programming. It is lightweight and scalable. It is often used for building real-time Web applications.
Ruby	It is a dynamic programming language that is expressive and readable. It is popularly used for the development of Web applications.
C#	It is a general-purpose programming language that is used to develop Windows-based applications and Web applications.

Advantages and Disadvantages of Server-side Programming

Table 1.1 describes the advantages and disadvantages of server-side programming.

Advantages	Disadvantages
Enhanced Security Server-side programs are run on the server and never sent to the client. Therefore, the code cannot be intercepted or exploited by an attacker. Even if users	Complexity Server-side programs must be able to handle a variety of tasks, such as database interaction, user authentication,

Advantages		Disadvantages	
	try to view the code, they can only view the client-side code and not the server-side code. This feature enhances the security of the Website or Web application.		and error handling. This requirement can make the program development complex.
High Scalability	Server-side programs can be scaled up to handle the increasing number of users and requests. This feature makes the server-side programs ideal for enterprise applications or Websites that experience heavy user traffic.	Performance Overhead	The server is required to process the data before sending the response to the client. This activity can add some performance overhead to a Website or Web application.
High Performance	Server-side programs run on the server. As a result, not all the code is downloaded to the client for execution. In addition, complex activities are completed in less time and steps. This feature improves the performance of the Website or Web application.	Cost	Applications that use server-side programs may require high-end and expensive servers to handle the data processing activities. This requirement can increase the cost of implementation of the Website or the Web application.
Adaptive Flexibility	Server-side programs can be used to implement a wide range of applications that provide several features such as generating dynamic	Vulnerability	Server-side programs can be vulnerable to security attacks, such as SQL injection or Cross-Site

Advantages	Disadvantages
<p>content, authenticating users, and interacting with databases. In addition, server-side programs are not browser-dependent, so program developers do not have to worry about the browser versions.</p>	<p>Scripting (XSS). However, these vulnerabilities can be handled by applying best practices and using secure coding techniques.</p>

Table 1.1: Advantages and Disadvantages of Server-side Programming

1.3 Introduction to Web Server

Servers can be of different types such as mail servers, DNS servers, and Web servers. Each server has a specific purpose.

A Web server is one that has the content required for a Website and has the logic for processing the data received from the client. A Web server stores files related to Websites. When a request for a Web page arrives through a browser, the Web server responds by returning the requested Web page to the Web browser.

A Web server has both hardware and software components to serve its purpose. The hardware connects the Web server to the network or Internet and facilitates the exchange of data with various clients.

The software component is responsible for processing the client requests and responding to them with the required information or data. The clients connect with a Web server using Hypertext Transfer Protocol (HTTP). This protocol is the guideline for the client for sending the requests and for the Web server for responding to the requests.

Functions that a Web server performs include:

Web servers physically store the Website content including Web pages, videos, and images.

Storing Website Content

When the Web server receives a request from a client, it processes the requests, retrieves the required content, and delivers the content to the client.

Delivering Website Content

The Web server hosts the Website or Web application. It also facilitates the installation, configuration, and maintenance of the Website or Web application.

Maintaining the Website or Web Application

The Web server handles multiple client requests simultaneously. This is accomplished by using the multi-threading technique or other similar techniques.

Managing Access to the Website or Web Application

1.3.1 Web Server Architecture

Web servers receive single or multiple requests from the clients simultaneously. The aim of the Web server is to handle these requests and respond to them with less wait time. Web server architecture is of two types based on the way the requests are handled:

- Concurrent approach
- Single-process-event-driven approach

Concurrent Approach

In the concurrent approach type of architecture, the Web server creates a thread for each request it receives. This facilitates simultaneous response to multiple requests. For example, consider that 10 friends plan to play online games and access the same gaming Website simultaneously. In this approach, the Web server that hosts the game will create ten separate threads, each thread serving one of the players. This ensures quick, real-time responses to the actions taken by the players, thereby enhancing player experience.

However, this approach requires large investments in hardware, such as CPU and memory, to handle multiple processing threads. The concurrent approach is best suited for Websites that experience heavy user traffic, such as social media, large e-commerce Websites, and gaming Websites.

Single-Process-Event-Driven Approach

In this single-process-event-driven approach, the Web servers use a single thread to handle all requests. The requests are handled in a non-blocking manner. For example, consider that there are ten users who are accessing a shopping Website. The users perform tasks such as browsing the catalog, viewing product details, and making purchases. In this approach, a single thread is used to handle user requests one-by-one as they come in, ensuring that one request processing does not block processes of other requests. The single-process-event-driven approach is best suited for Websites that experience low user traffic, such as small e-commerce Websites and Websites that provide information about companies.

1.3.2 Working of Web Servers

Figure 1.2 shows the working of Web servers.

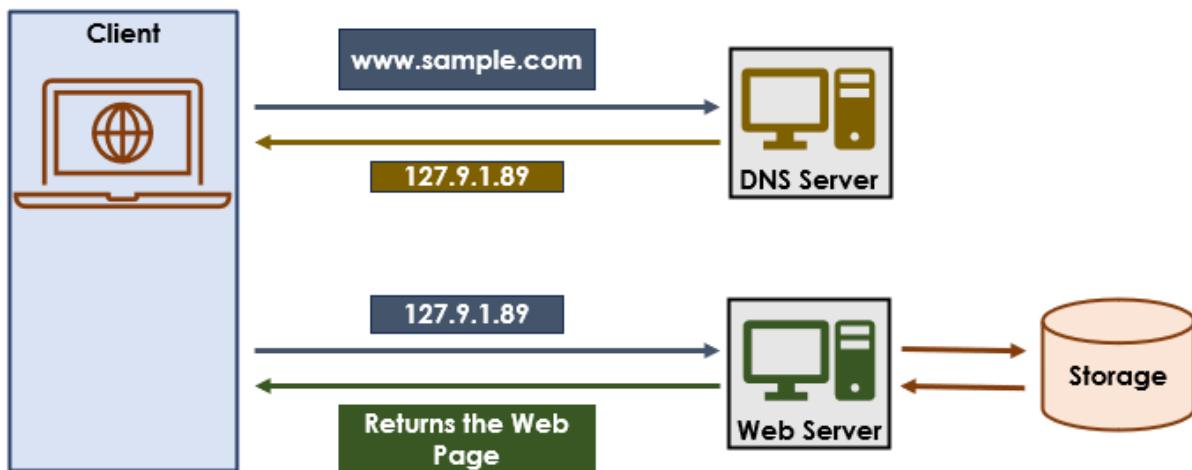


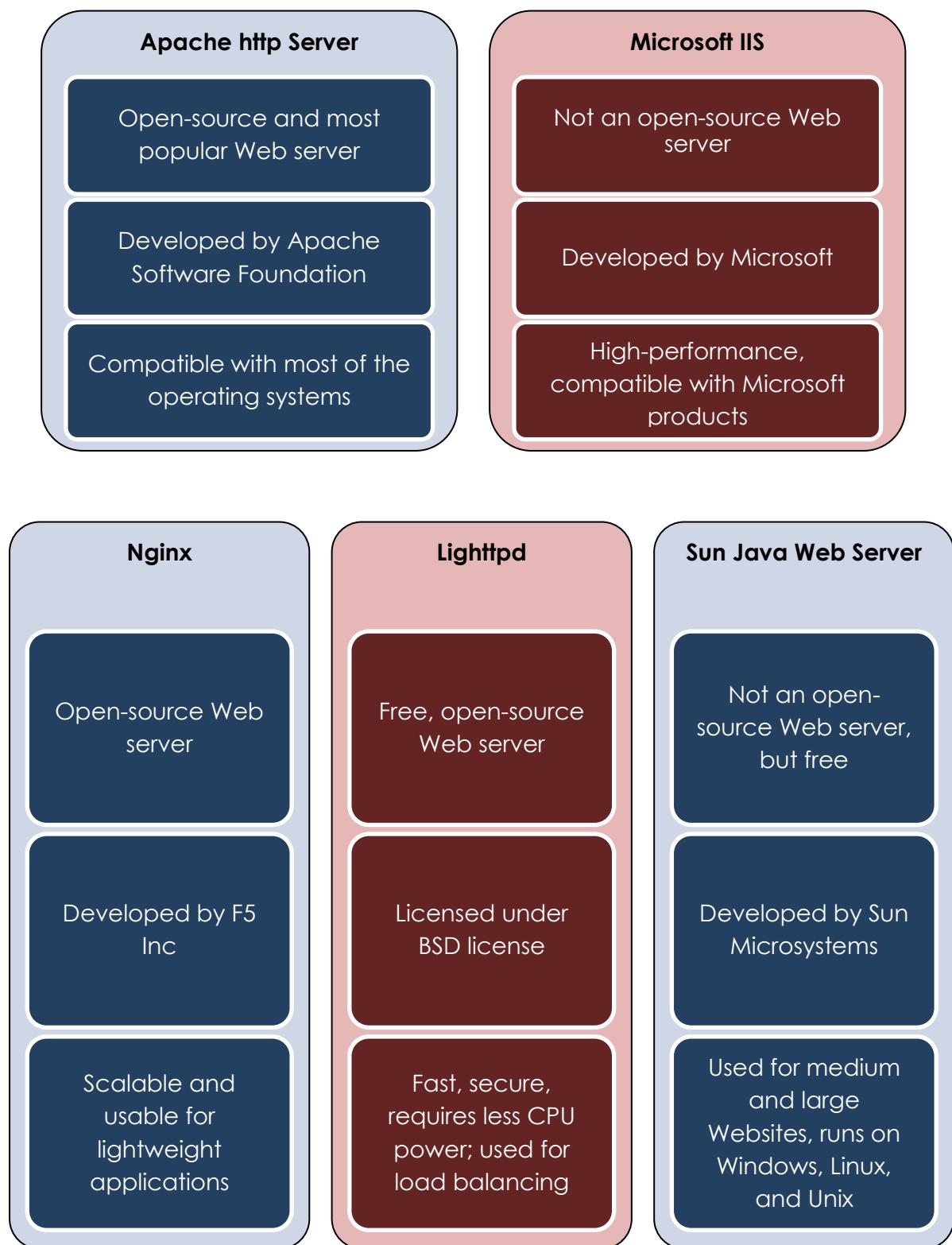
Figure 1.2: Working of a Web Server

As shown in Figure 1.2, a user enters a Universal Resource Locator (URL), for example, www.sample.com, in the Web browser on the client. The client sends this URL over the network to the DNS server. The DNS server checks for the URL in the database and returns the associated IP address to the client. The client then sends the IP address to the Web server. Finally, the Web server locates the file, image, or video in its storage and returns the requested content to the client. The client then displays the content received in the browser window.

1.3.3 Types of Web Servers

There are many types of Web servers. Each is designed for a specific purpose. Some are open-source, while some are not.

Some of the Web servers include:



1.4 Introduction to Node.js

Node.js is a runtime environment for developing server-side programs using JavaScript. It is used to develop Web applications. It is open-source and is

compatible across numerous platforms, including Windows, Linux, Mac OS, and Unix. It is built on Google Chrome's JavaScript engine (V8 Engine). It efficiently executes JavaScript code using a Just-in-Time compiler instead of an interpreter.

1.4.1 Features of Node.js

Node.js has some distinct features that make it the most preferred runtime environment for developing and implementing Web applications. These features are as follows:

- It is easier for developers to adapt**

Most of the Web application developers are well-versed in the use of JavaScript coding. Node.js is also written in JavaScript, making it easier for the developers to adapt to it. JavaScript is also used to develop the front end for Web applications. A combination of JavaScript and Node.js helps developers to create full-stack Web applications by using only JavaScript to develop the client-side user interface and server-side programs. In addition, Node.js is compatible with various operating systems, such as Windows, Mac OS, Linux, or mobile platforms. This feature makes Node.js a preferred platform for developing and implementing Web applications.

- It facilitates enhanced user experience**

Node.js has a non-blocking way of executing the instructions it receives and returning the response. It uses a single thread in which all the requests are queued and processed one after another as they come in. This way of handling requests ensures that there is no wait time for processing any of the requests. This asynchronous way of processing data provides enhanced performance of the Website or Web application. The fast performance, in turn, enhances the user experience when using the Website or Web application.

- It reduces the cost of investing in server hardware**

Node.js uses an event-based approach. In this approach, an event loop handles the operations asynchronously. The event loop listens for events and then, calls the associated callback function. An event could be any action, for example, the clicking of a button by a user is an event. The callback function associated with the event provides the responses to these events. For the callback function to be executed, fewer resources and less memory on the server side are required. This feature ensures a reduction in the cost of investment in server resources.

- It is a scalable runtime environment**

As the number of users accessing the Website or Web application increases, the processing power of the server must also be increased to handle the users.

Increasing the processing power in terms of memory, CPU, and other resources can be expensive. With Node.js, this scalability is in-built because it uses the event-driven approach of handling user requests. Applications running on Node.js are scalable as the environment provides the elasticity and flexibility to adjust or customize according to the demanding situations.

- **It facilitates quick data streaming**

Sequential data handling of input and output is termed data streaming. Node.js reads or writes chunks of data, processes it and sends it back without holding it in memory. This feature reduces the overhead of buffering data and makes the streaming process faster.

1.4.2 Node.js Installation

To install Node.js, perform these steps:

1. Open a browser and navigate to the URL:
<https://nodejs.org/en/download>

The Web page opens as shown in Figure 1.3.

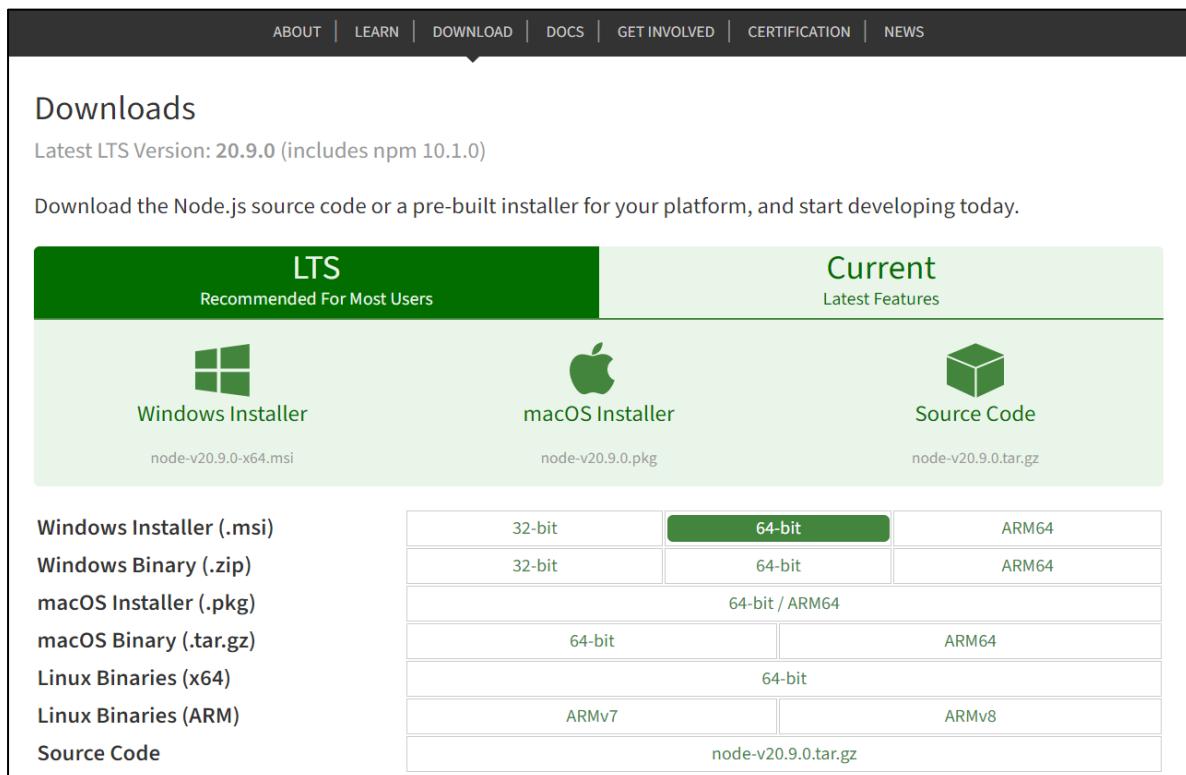


Figure 1.3: Downloading Node.js Installer

2. Based on the operating system installed on the computer, download the associated .msi file by clicking it.
3. On the computer, navigate to the folder where the file is downloaded.
4. To start the installation, double-click the .msi file.

The Node.js Setup wizard launches and the Welcome page is displayed, as shown in Figure 1.4.

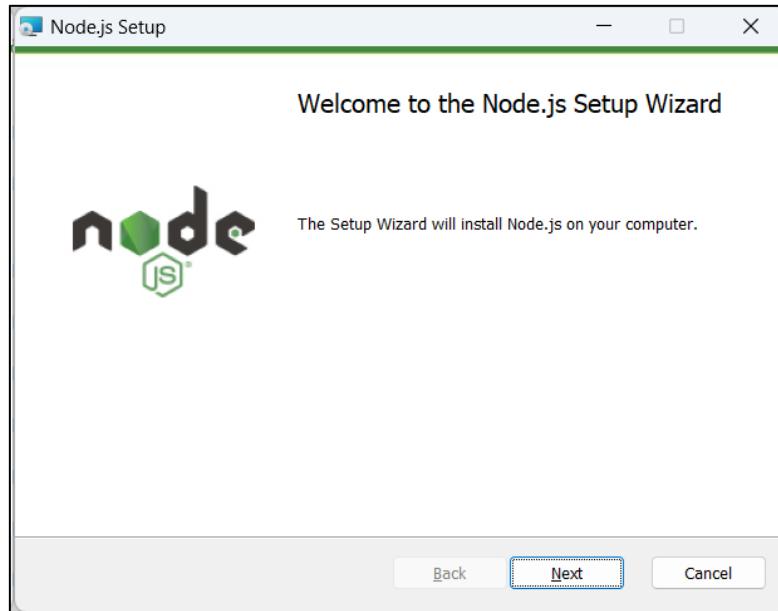


Figure 1.4: Node.js Setup Wizard – Welcome Page

5. On the Welcome page of the wizard, click **Next**.
The End-User License Agreement page of the wizard opens.
6. On this page, select the **I accept the terms in the License Agreement** check box and click **Next**, as shown in Figure 1.5.

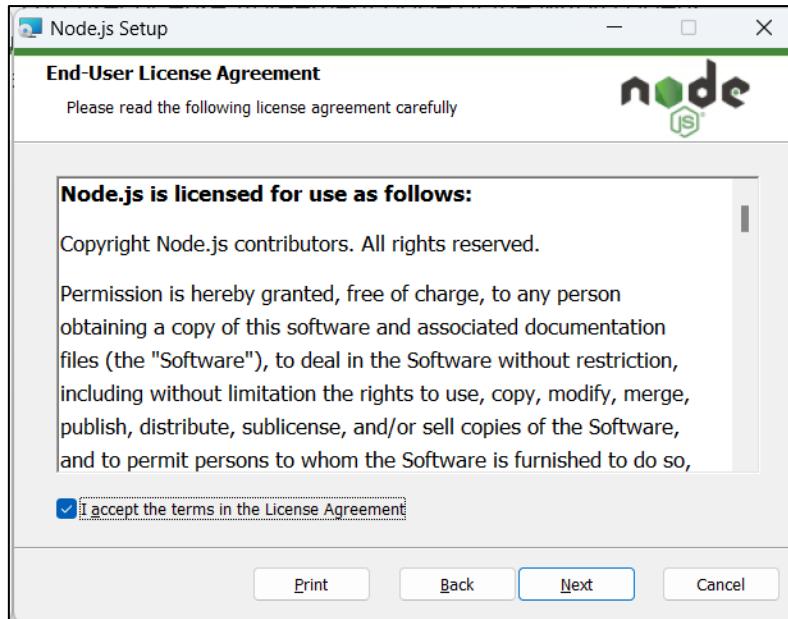


Figure 1.5: Accepting the End-User License Agreement

The Destination Folder page of the wizard opens, as shown in Figure 1.6.

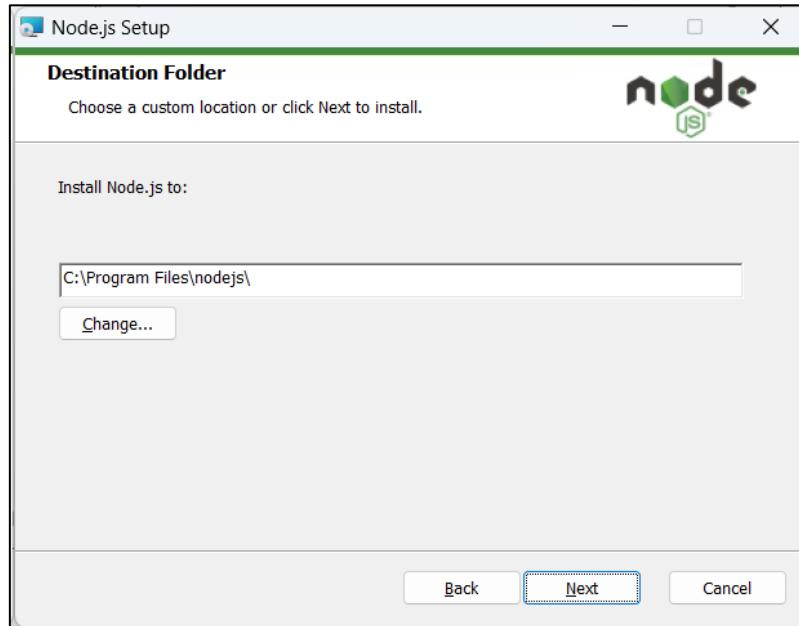


Figure 1.6: Selecting Destination Folder

- To install Node.js on the default path, click **Next**.
The Custom Setup page of the wizard opens, as shown in Figure 1.7.

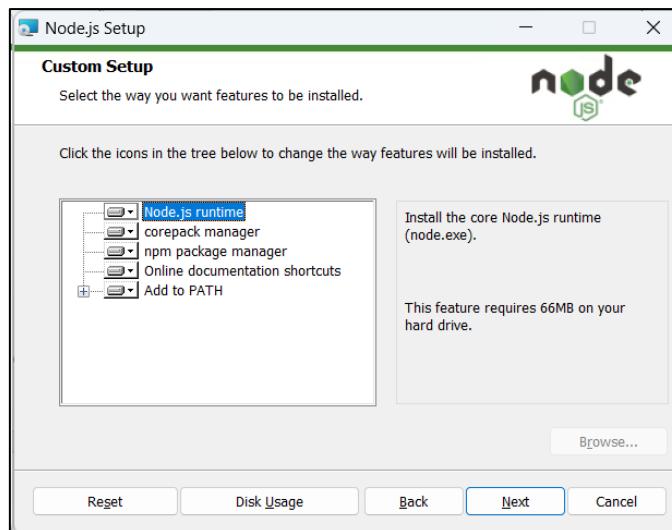


Figure 1.7: Selecting the Features to Install

- To install all the features on the computer, click **Next**.
The Tools for Native Modules page of the wizard opens, as shown in Figure 1.8.

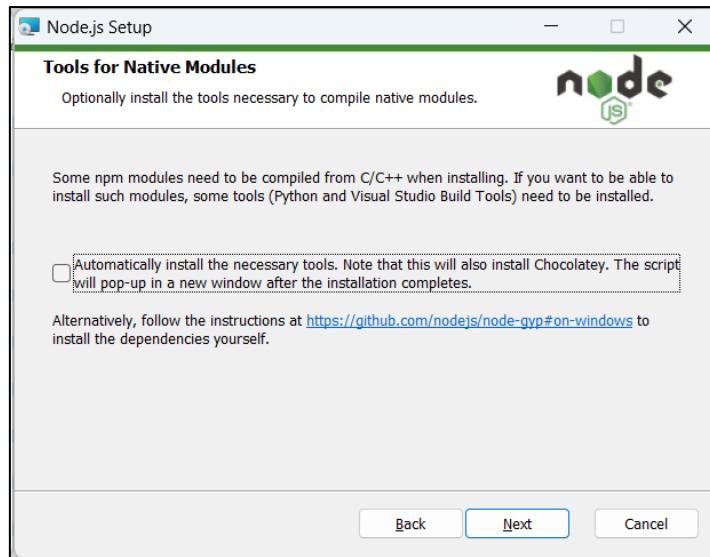


Figure 1.8: Installing Necessary Tools

9. To continue without installing the additional tools, click **Next**.
The Ready to Install Node.js page of the wizard opens, as shown in Figure 1.9.

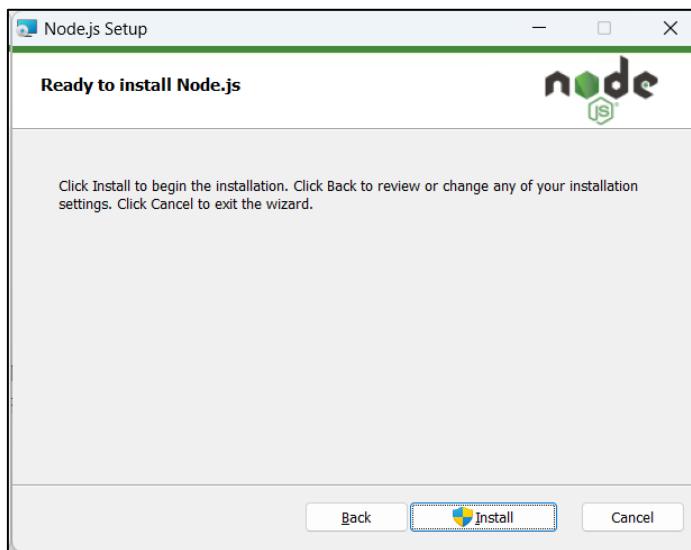


Figure 1.9: Installing Node.js

10. To install Node.js, click **Install**.
Node.js is installed on the computer. After the installation is complete, the Completed Node.js Wizard page opens, as shown in Figure 1.10.

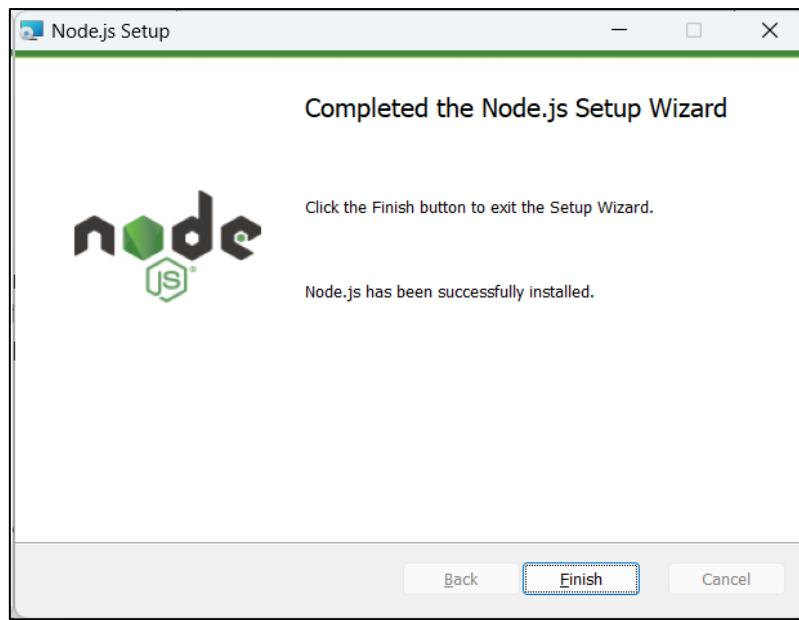


Figure 1.10: Completing the Installation

11. To close the wizard, click **Finish**.
12. To verify that Node.js has been installed, open a Command Prompt window.
13. At the command prompt, type the command as:

```
node -v
```

The version of Node.js that has been installed will be displayed as shown in Figure 1.11.

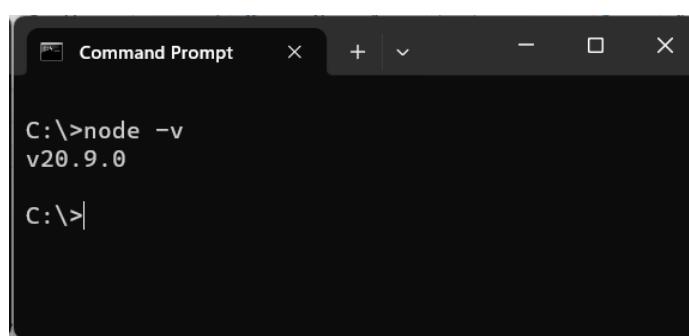


Figure 1.11: Checking Version of Node.js

To create scripts/programs for Node.js, one can use any text editor or Integrated Development Environment (IDE). Visual Studio (VS) Code is recommended for code development as it is free and also offers support to run and debug Node.js applications.

Developers can download and install VS Code using this link:
<https://code.visualstudio.com/>

Set the **Path** environment variable using Control Panel to include the folder for Node.js as shown in Figure 1.12. The Path variable requires to be set only once.

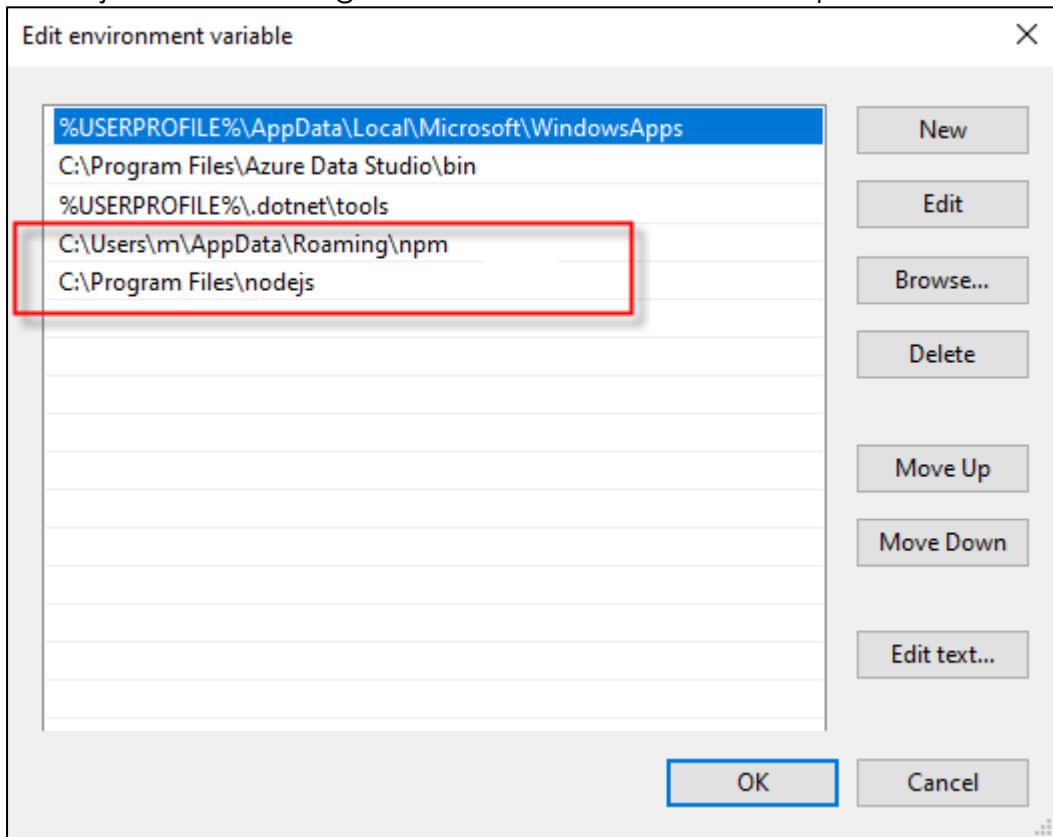


Figure 1.12: Setting Path for Node.js

Now, developers can launch VS Code and use it to write, run, and debug Node.js applications.

1.5 Summary

- Client-server architecture allows the server to serve requests from multiple clients.
- Client, server, and network are the components of the client-server architecture.
- The DNS server and Web server are the two servers that help in fetching the Web page when a request is made.
- Server-side scripting allows data processing to be done on the server side, thereby keeping the processing part hidden from the client.
- Java, PHP, Python, Node.js, Ruby, and C# are some of the server-side scripting tools.
- Node.js is a single-threaded, runtime environment for running JavaScript.
- Ease of working with JavaScript, scalability, and fast streaming of data makes Node.js a preferred runtime environment for Web applications.
- Visual Studio Code can be used as an IDE for Node.js applications.

Test Your Knowledge

1. A public library catalog system is managed using client-server architecture. Visitors search for books using a computer terminal. Results are displayed on the screen. In this library scenario, which function does the server perform?
 - a) It allows visitors to search for books
 - b) It displays book information on the screen
 - c) It stores and manages the catalog data
 - d) It serves as a client for the visitors
2. In Web development what is the difference between server-side and client-side programming?
 - a) Code execution on the client's side is handled using server-side programming
 - b) Process requests on the Web server are handled using server-side programming
 - c) Server databases are managed using client-side programming
 - d) Client-side programming controls the server's response to requests
3. A Web server architecture that can efficiently handle thousands of simultaneous connections is required for an online gaming platform. Which Web server architecture approach will be used for this situation?
 - a) Blocking Approach
 - b) Concurrent Approach
 - c) Single-Process-Event-Driven Approach
 - d) Non-blocking Approach
4. In concurrent approach how task is assigned to each thread?
 - a) Each task is assigned to one thread
 - b) Multiple tasks are assigned to a single thread
 - c) Tasks are assigned to the threads based on the availability
 - d) Based on the server configuration tasks are decided
5. What are the important features of Node.js?
 - a) Asynchronous Nature
 - b) Sharding
 - c) Minimum Buffering
 - d) Replication of Data

Answers to Test Your Knowledge

1	c
2	b
3	c
4	a
5	a, c

Try It Yourself

1. Install Node.js version 20.7.0 on a Windows system.
2. Verify the installation of Node.js on the Windows system.
3. Prepare a presentation with at least five points recommending why Node.js is good for server-side programming.



SESSION 2

NODE.JS ESSENTIALS

Learning Objectives

In this session, students will learn to:

- Describe the architecture of Node.js
- List benefits and limitations of Node.js applications
- Explain the concept of console-based applications in Node.js
- Explain major programming constructs in Node.js
- Compare the roles of Node Package Manager (NPM) and Node Version Manager (NVM) in Node.js
- Describe global objects and their use in Node.js applications

Node.js is a server-side environment that uses JavaScript to perform request-response operations. It has its own set of benefits and limitations in the computing domain. Interactive and non-interactive command-line environments are popular in Node.js. Programming essentials such as variables, multiline expressions, data types, functions, and arrays are part of Node.js application development. In Node.js, Node Package Manager (NPM) and Node Version Manager (NVM) provide information on packages and versions, respectively.

This session will cover the architecture of Node.js in detail, including the components and workflow. It will explore the benefits and limitations of Node.js applications. The session will describe the concept of a console-based

application and Read Print Evaluate Loop (REPL) environment. It will also explain the important programming constructs, such as data types, functions, and arrays, used in Node.js applications. Finally, the session will conclude by comparing the roles of NPM and NVM in Node.js and using global objects in Node.js applications.

2.1 Introduction to Node.js Architecture

Node.js is an efficient runtime environment based on JavaScript. It is a single-threaded platform in which a single main thread handles all the requests from the clients. Node.js was developed on Google Chrome's V8 JavaScript engine.

Figure 2.1 depicts the visual representation of the Node.js architecture.

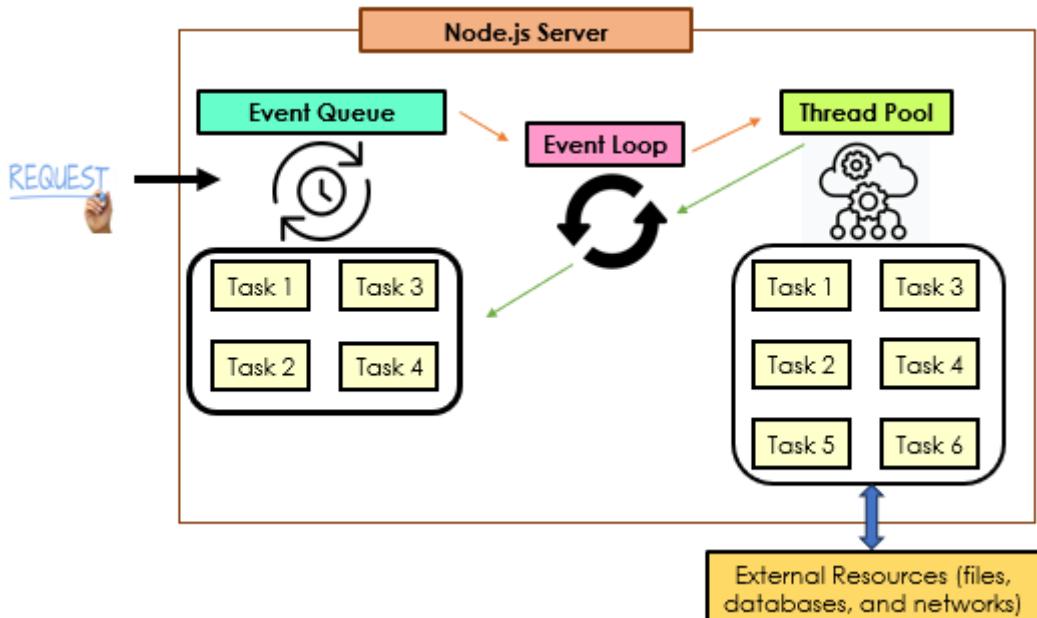
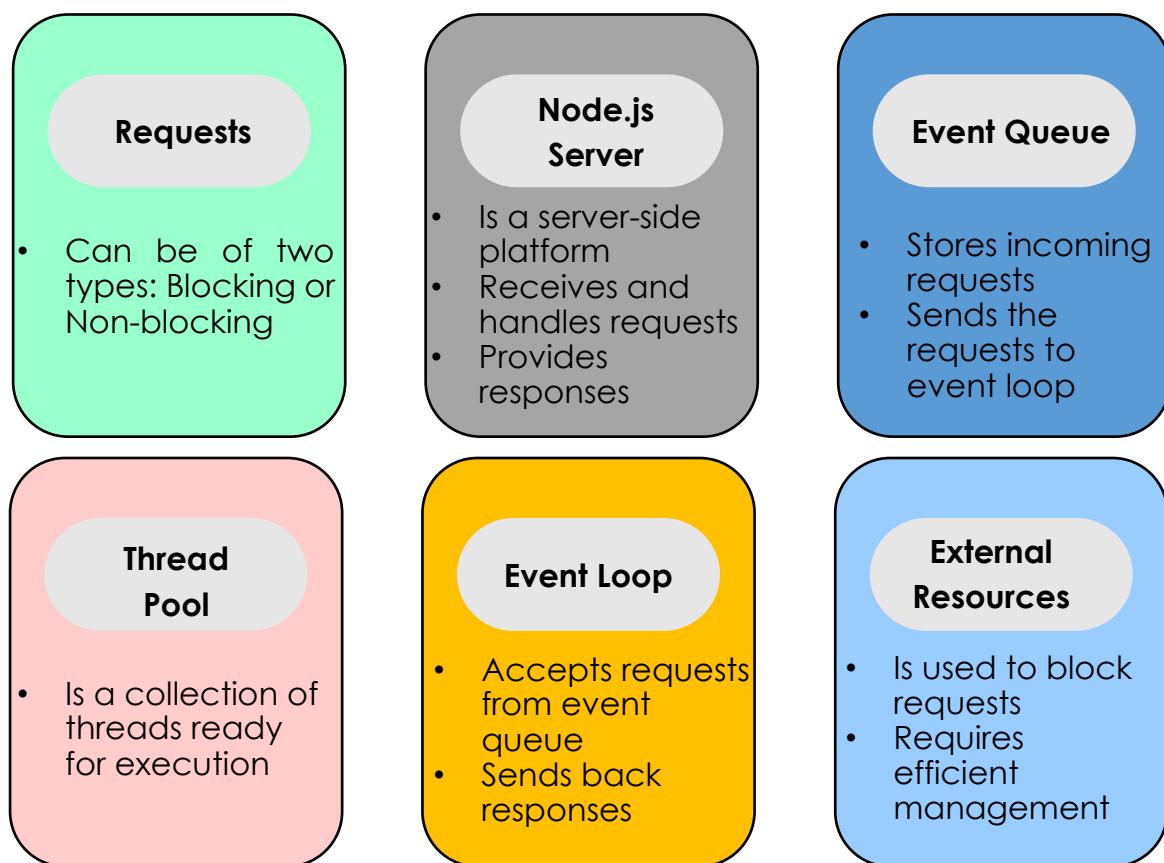


Figure 2.1: Node.js Architecture

2.1.1 Components of Node.js Architecture

Components included in the Node.js architecture are as follows:

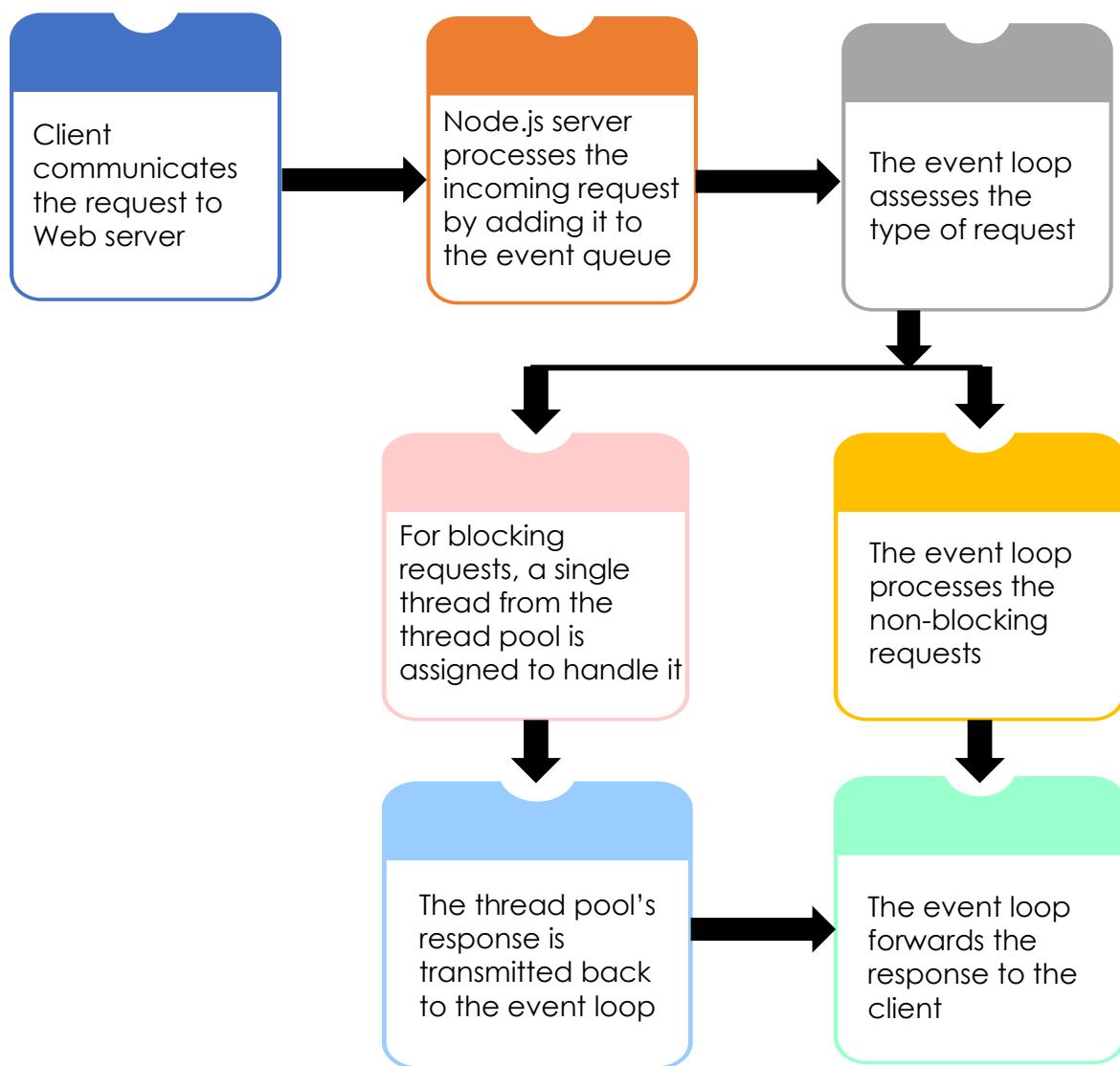


2.1.2 Workflow of Node.js

Node.js follows the concurrent mode of executing tasks. This means that it does not wait for the current task to end and then, begin the execution of the next task. There is a wastage of CPU power if Node.js waits for each task to end before proceeding with the next task.

When the client sends a request to the server, a thread is assigned to that request. The thread is responsible for processing the request from start to end. The thread contains specific instructions to process the request and generate a response. The response generated by the thread is sent to the event loop. The event loop is in charge of transmitting the response back to the client.

Here is the visual representation of the workflow of Node.js.



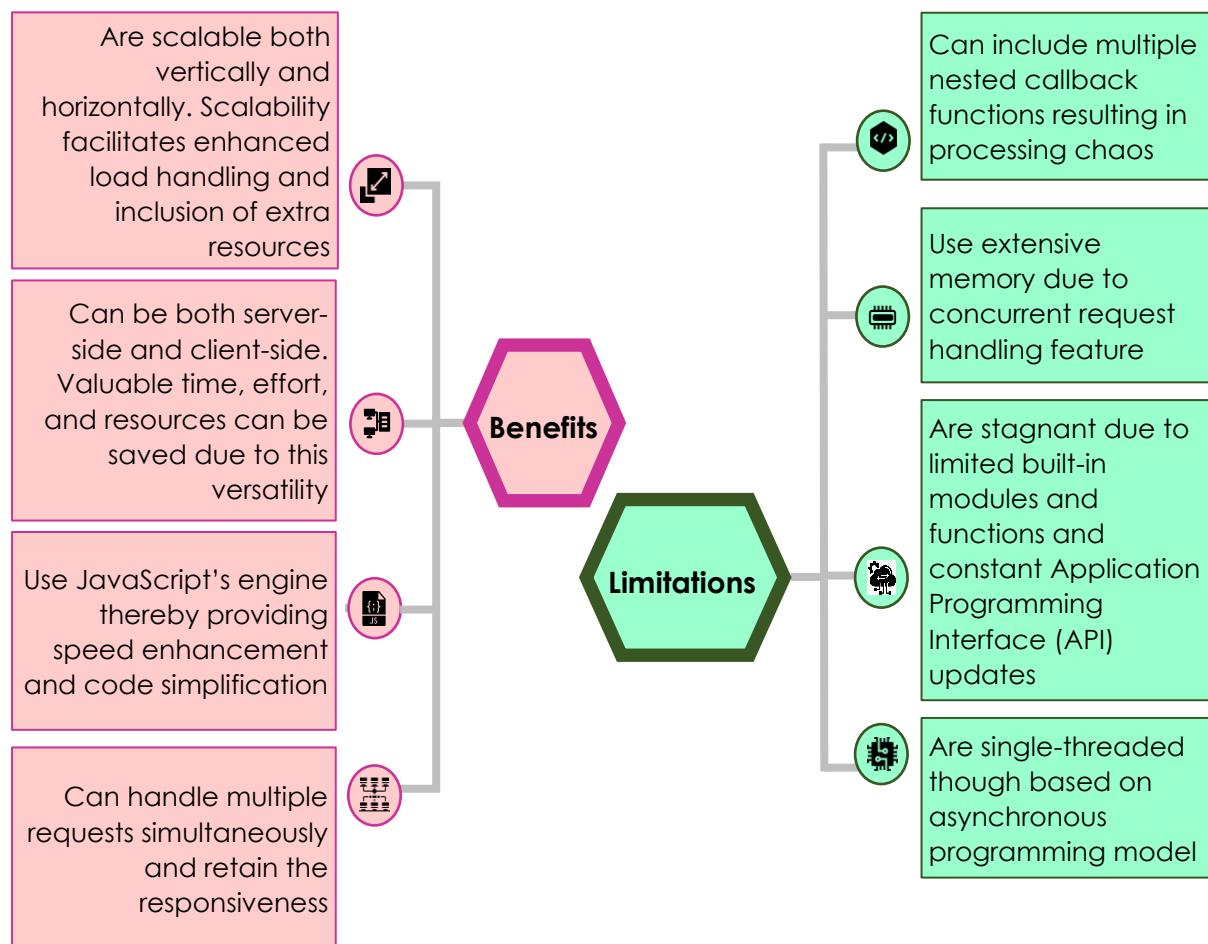
A set of specific instructions to the server is known as a thread in Node.js.

2.2 Benefits and Limitations of Node.js

Software developers who are well-versed with JavaScript will find it quite easy to comprehend Node.js in their applications. They can develop programs using JavaScript for both front-end and back-end applications. However, Node.js comes with its own set of limitations as well.

Benefits and limitations of Node.js applications are presented here.

Node.js applications:



2.3 Console-based Application in Node.js

There are two ways to develop Node.js applications, namely console-based and Web-based. Console-based Node.js applications run at the command prompt. These applications do not have a Graphical User Interface (GUI).

The `console` module in Node.js has the `console.log` function that helps to display custom messages on the VS Code terminal or command prompt.

Consider an example code where a custom message using `console.log` function will be displayed on the terminal.

Perform following steps:

1. Open a text file in VS Code.
2. Type the given code in the file.

```
console.log('This session covers the essentials of  
Node.js');
```

In the example code, a custom message is provided as the argument for `console.log` function.

3. Save the file as `sample_display.js` in a folder on the local system.
4. Open the command prompt window and navigate to the required folder. Alternatively, open VS Code Terminal. Ensure that the environment path variable includes the path for the Node.js installation.
5. To run the saved program, at the command prompt or VS Code Terminal, type following command and press Enter.

```
node sample_display.js
```

Figure 2.2 displays the output of the code.

```
This session covers the essentials of Node.js
```

Figure 2.2: Console Output

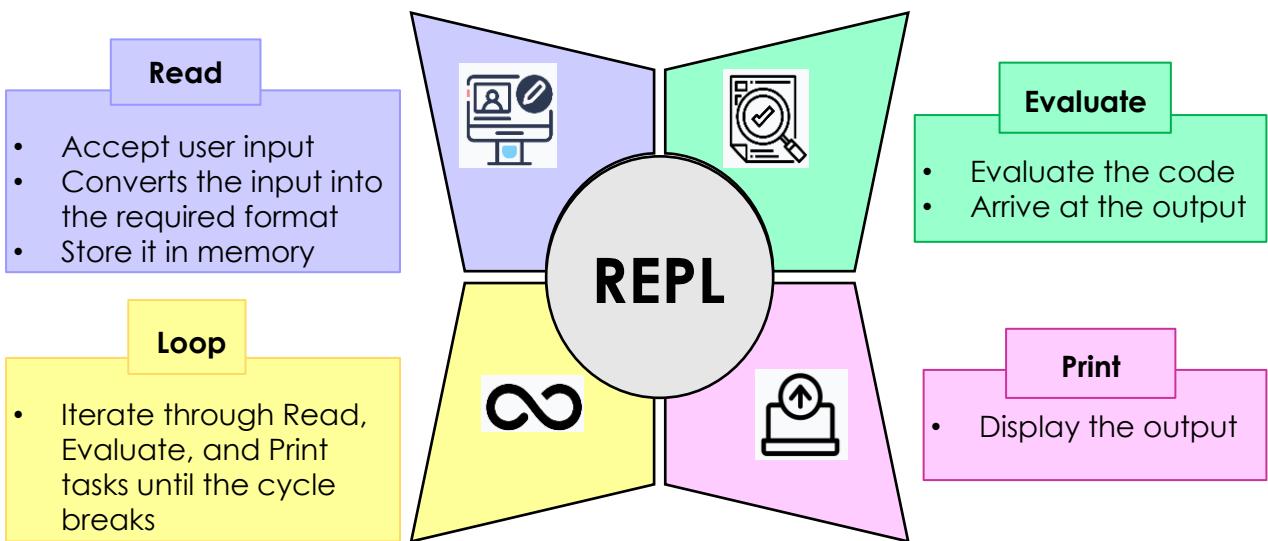
Observe the program output. The custom message is displayed on the command prompt or Terminal.

2.3.1 Overview of REPL

In Node.js, REPL stands for Read Evaluate Print Loop. REPL resembles a Windows command-line environment or a Unix/Linux shell, installed with Node.js on the system.

The conventional way of executing a Node.js program is to run the saved `.js` file at the command prompt. However, using REPL, small code snippets can be directly executed at the REPL prompt without creating `.js` files.

Each of the four parts in REPL plays a specific role. Tasks performed by these four parts are as follows:



To activate the REPL prompt in Windows, perform following steps:

1. Open the command prompt window.
2. At the command prompt, type the given command and press Enter.

```
node
```

Figure 2.3 displays the command output at the command prompt.

```
Welcome to Node.js v21.1.0.  
Type ".help" for more information.  
>
```

Figure 2.3: Command Output to Launch REPL

3. The REPL command prompt appears.
4. Begin executing the Node.js commands.

Working with Simple Expressions

Simple mathematical expressions can be executed directly at the REPL command prompt.

Consider an example to understand this. Perform the given steps:

1. Activate the REPL prompt in Windows.
2. Type the mathematical expression at the REPL prompt and press Enter.

```
15+80*5
```

Figure 2.4 displays the output of the mathematical expression.

```
Welcome to Node.js v21.1.0.  
Type ".help" for more information.  
> 15+80*5  
415
```

Figure 2.4: Output of Mathematical Expression

Observe the output. The result of the mathematical expression is displayed on the terminal. A separate .js file has not been created to store or process the mathematical expression.

Similar to simple mathematical expressions, string expressions can also be executed at the REPL command prompt.

Consider an example where a simple string expression using a concatenation operator is executed at the REPL command prompt.

Perform following steps:

1. Activate the REPL prompt in Windows.
2. Type the given string expression at the REPL prompt and press Enter.

```
'Welcome' + ' ' + 'to Node.js'
```

Figure 2.5 displays the output of the string expression.

```
> 'Welcome' + ' ' + 'to Node.js'  
'Welcome to Node.js'
```

Figure 2.5: Output of String Expression

Observe the output. The concatenation operator joins two strings and produces a single string. The result of the string expression is displayed on the terminal.

Working with Variables

A variable is a programming entity that provides a name to a specific location in system's memory. A value can be stored in a variable and that value can be

used in programs through the variable. In Node.js code, use the `var` keyword to declare and assign a value to the variable.

Consider an example where variables are used to demonstrate an addition operation.

Perform following steps:

1. Activate the REPL prompt in Windows.
2. Type the statements given in Code Snippet 1 at the REPL prompt one by one. Ensure to press Enter at the end of each statement for execution.

Code Snippet 1:

```
var a = 5
var b = 7
var c = a + b
console.log ('The sum of a and b is:',c);
```

In Code Snippet 1, values are stored in variables `a` and `b` respectively. Variable `c` stores the sum of variables `a` and `b`. The result of addition is displayed as the output.

Figure 2.6 displays the output of the addition operation.

```
> var a = 5
undefined
> var b = 7
undefined
> var c = a + b
undefined
> console.log ('The sum of a and b is:',c);
The sum of a and b is: 12
undefined
```

Figure 2.6: Output of Addition Using Variables

Observe the output, which is, The sum of 5 and 7 is 12.

Note that by default the return value of the commands is displayed as `undefined` after the execution of each command. This can be suppressed by setting the `ignoreUndefined` option to `true` when starting an REPL instance as given in following command:

```
node -e "require('repl').start({ignoreUndefined: true})"
```

The default value is `false`.

Working with Multiline Expressions

The REPL environment is able to identify complete expressions at the prompt. If the Enter key is pressed at the end of an incomplete expression, REPL automatically adds ellipsis (...) on the next line. The ellipsis is an indication to continue the code.

Consider an example that demonstrates the use of REPL multiline expressions with the help of a do-while loop.

Perform following steps:

1. Activate the REPL prompt in Windows.
2. Type the given statements in Code Snippet 2 at the REPL prompt one by one. Ensure to press Enter at the end of each statement for continuity.

Code Snippet 2:

```
var a = 5
var b = 7
var i = 0
do{
    var c = a + b
    i++
    a++
    b++
    console.log ('Sum of a and b in
iteration',i,'is', c)
}while (i<3)
```

In Code Snippet 2, variables `a`, `b`, and `i` are declared and assigned values. Inside the do-while loop:

- Variables `a` and `b` are added.
- The result is stored in variable `c`.
- The values of `a`, `b`, and `i` are incremented by 1.
- The result of addition obtained in each iteration is displayed on the terminal.

The do-while loop continues to execute until the value of `i` is less than 3.

Figure 2.7 displays the output of the multiline expressions.

```

> var a = 5
undefined
> var b = 7
undefined
> var i = 0
undefined
> do{
... var c = a + b
... i++
... a++
... b++
... console.log ('Sum of a and b in iteration', i, 'is', c)
... }while (i<3)
Sum of a and b in iteration 1 is 12
Sum of a and b in iteration 2 is 14
Sum of a and b in iteration 3 is 16
undefined

```

Figure 2.7: Output of Multiline Expressions

Observe the output. Ellipsis are automatically generated to continue the code. Results of the iterations are displayed as 12, 14, and 16.

Table 2.1 lists some of the commonly used commands in Node.js REPL.

Commands	Description
Ctrl + C/Ctrl + c	Stops the current command's execution
Ctrl + C twice/Ctrl + c twice/Ctrl + d	Exits the REPL environment
up/down keys	Helps to check the command history and update the earlier commands
tab keys	Displays a list of current commands
.help	Shows a list of all available commands
.break/.clear	Exits from multiline expressions
.save <filename>	Saves the current REPL session to a file provided in the command
.load <filename>	Loads the specified file contents into the current REPL session

Table 2.1: REPL Commands

2.4 Node.js Programming Constructs

Different types of programming constructs are available in Node.js. These constructs enable the developers to create varied forms of applications.

Some of the programming constructs in Node.js are as follows:

Data Types

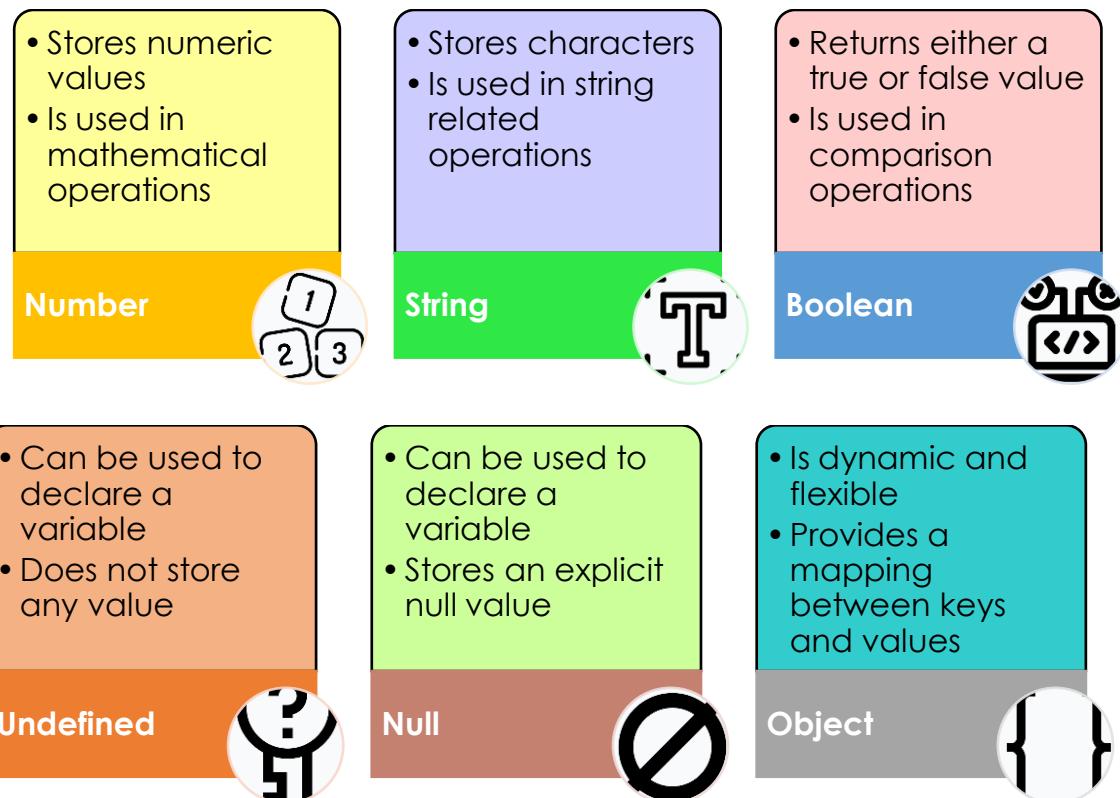
Functions

Arrays

2.4.1 Overview of Data Types

Data types in a programming language are important entities. The type of value stored in a variable signifies the importance of a data type. Variables can store different types of values such as text, numbers, decimals, and so on. For each variable, a data type can be assigned.

Here are various data types that can be used in Node.js applications.



Working with Number Data Type

The `number` data type can take both positive and negative values. Consider an example code in Code Snippet 3 where variables are declared using number data types and arithmetic operations are demonstrated.

Code Snippet 3:

```
var a = 40;  
var b = 10.5;
```

```
var c = -4;
console.log('Value of a is:',a);
console.log('Value of b is:',b);
console.log('Value of c is:',c);
console.log('Sum of a and c is :' , (a+c));
console.log('Subtracting b from c is:' , (c - b));
console.log('Product of c and b is :' , (c * b));
console.log('Quotient of a/b is :' , (a / b));
console.log('Remainder of a/b is :' , (a % b));
```

In Code Snippet 3, although `var` is used, variables `a`, `b`, and `c` are inferred to be of number data type based on the content they hold. Various mathematical operations such as addition, subtraction, multiplication, and division are carried out in the program.

Figure 2.8 displays the output of Code Snippet 3.

```
Value of a is: 40
Value of b is: 10.5
Value of c is: -4
Sum of a and c is : 36
Subtracting b from c is: -14.5
Product of c and b is : -42
Quotient of a/b is : 3.8095238095238093
Remainder of a/b is : 8.5
```

Figure 2.8: Output of Code Snippet 3

Observe the program output. The results of the mathematical operations are displayed on the console.

Working with String Data Type

To assign a value to a string data type variable, use either single quotes or double quotes. The variable becomes a string variable only after the value in the required format is assigned to it.

Consider an example code in Code Snippet 4 where string concatenation is demonstrated.

Code Snippet 4:

```
var str1 = 'Welcome ';
var str2 = 'to Node.js Session 2';
console.log('First string is:',str1);
console.log('Second string is:',str2);
console.log('Concatenated string is:',str1+str2);
```

In Code Snippet 4, two string variables are declared. These two strings are concatenated using the + concatenation operator.

Figure 2.9 displays the output of Code Snippet 4.

```
First string is: Welcome
Second string is: to Node.js Session 2
Concatenated string is: Welcome to Node.js Session 2
```

Figure 2.9: Output of Code Snippet 4

Observe the program output. The first string, second string, and the concatenated string are displayed.

Working with Boolean Data Type

The Boolean data type helps the developers to build comparison and conditional logic in Node.js applications.

Consider an example code in Code Snippet 5 where Boolean data type variables are demonstrated.

Code Snippet 5:

```
var i = false;
var a = 6;
var b = 9;
console.log('The Boolean value of variable i is:', i);
console.log('The value of variable a is:', a);
console.log('The value of variable b is:', b);
console.log('The values of variables a and b are the same:', a == b);
console.log('The value of variable a is not the same as value of variable b:', a != b);
console.log('The Boolean value of !i is:', !i);
```

In Code Snippet 5, variable *i* is declared using a Boolean data type. Operators such as equal to, not equal to, and not are used to compare the values.

Figure 2.10 displays the output of Code Snippet 5.

```
The Boolean value of variable i is: false
The value of variable a is: 6
The value of variable b is: 9
The values of variables a and b are the same: false
The value of variable a is not the same as value of variable b: true
The Boolean value of !i is: true
```

Figure 2.10: Output of Code Snippet 5

Observe the program output. The Boolean values are displayed as true or false.

Working with Undefined Data Type

Consider an example code in Code Snippet 6 where undefined data type variables are demonstrated.

Code Snippet 6:

```
var number1;
console.log('Value of number1 is:',number1);
var number2;
console.log('Value of number2 is:',number2);
```

In Code Snippet 6, two variables are declared without any values.

Figure 2.11 displays the output of Code Snippet 6.

```
Value of number1 is: undefined
Value of number2 is: undefined
```

Figure 2.11: Output of Code Snippet 6

Observe the program output. Since there are no values assigned to the variables, the output shows undefined.

Working with Null Data Type

Consider an example code in Code Snippet 7 where null data type variables are demonstrated.

Code Snippet 7:

```
var number1 = null;
console.log("Value of number1 is:",number1);
var number2 = null;
console.log("Value of number2 is:",number2);
```

In Code Snippet 7, two variables are declared and assigned null values.

Figure 2.12 displays the output of Code Snippet 7.

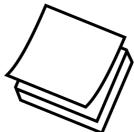
```
Value of number1 is: null
Value of number2 is: null
```

Figure 2.12: Output of Code Snippet 7

Observe the program output. The values are displayed as null.

Working with Object Data Type

Names and values play a crucial role in an object data type. Each variable declared as an object will have name:value pairs. Names must be unique. Each name will have a corresponding value. However, values can be repeated. Commas separate the name:value pairs and they are presented in curly braces. Another term for name:value pairs is object properties.



Name:value pairs for the object data type are also known as key:value pairs in Node.js.

Consider an example code in Code Snippet 8 where object data type is demonstrated.

Code Snippet 8:

```
var student = {  
    Name: 'David Franklin',  
    Age: 12,  
    Grade: 8  
};  
console.log('Name:',student.Name);  
console.log('Age:',student.Age);  
console.log('Grade:',student.Grade);
```

In Code Snippet 8, the variable `student` is declared as an object data type with the key:value pairs. Keys are `Name`, `Age`, and `Grade`. Values are `David Franklin`, `12`, and `8`.

Figure 2.13 displays the output of Code Snippet 8.

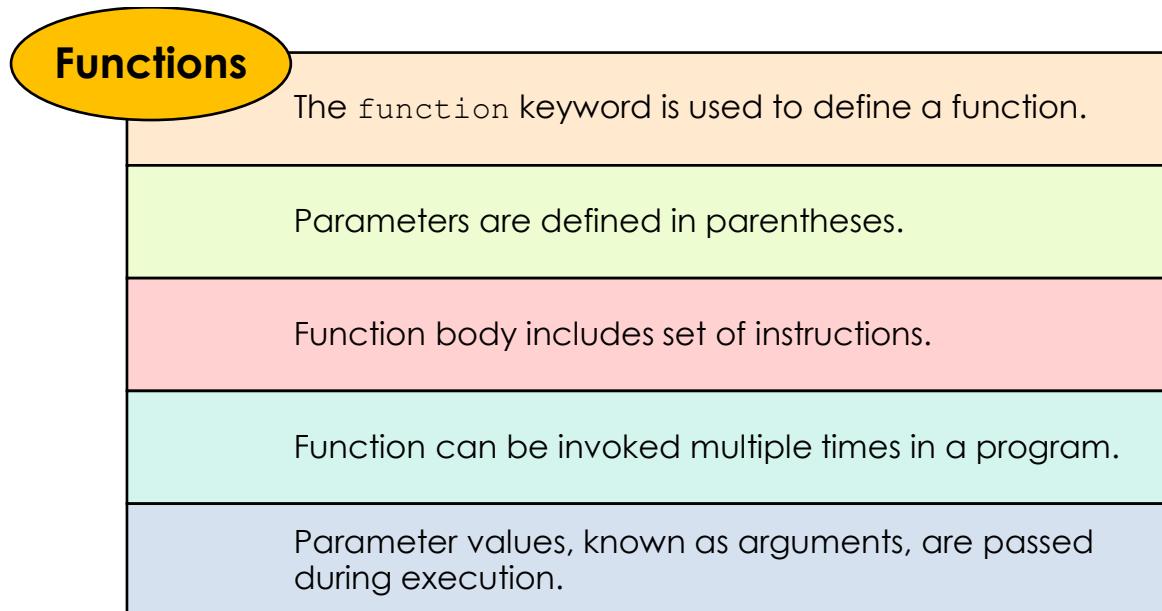
```
Name: David Franklin  
Age: 12  
Grade: 8
```

Figure 2.13: Output of Code Snippet 8

Observe the program output. The keys and the values of the object are displayed.

2.4.2 Overview of Functions

Functions are an important programming construct in Node.js. There are two types of functions, built-in and user-defined.



Consider an example code in Code Snippet 9 where a user-defined function is created and executed.

Code Snippet 9:

```
function student(name,age,grade,mark1, mark2) {  
    console.log('Name:',name);  
    console.log('Age:',age);  
    console.log('Grade:',grade);  
    var total_marks=mark1+mark2;  
    console.log('Total Marks:',total_marks)  
}  
student('David Franklin',13,8,78,85);
```

In Code Snippet 9, a user-defined function with the name `student` is defined. The function calculates total marks of a student and has five parameters. Values for these parameters are passed as arguments when the function is called.

Figure 2.14 displays the output of Code Snippet 9.

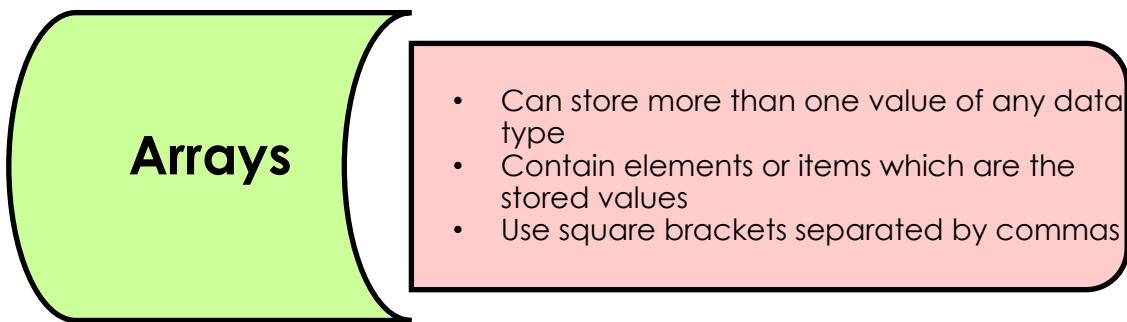
```
Name: David Franklin  
Age: 13  
Grade: 8  
Total Marks: 163
```

Figure 2.14: Output of Code Snippet 9

Observe the program output. The name, age, grade, and marks are passed as arguments when the function is called. Total marks is calculated and the values of the object are displayed.

2.4.3 Overview of Arrays

Arrays are another frequently used programming construct in Node.js.



Consider an example code in Code Snippet 10 where an array is declared and demonstrated.

Code Snippet 10:

```
var arr1 = [76,80,85,90,92];  
var sum=0, avg=0;  
for (var i = 0; i < arr1.length; i++) {  
    sum=sum+arr1[i]  
    avg=sum/5  
}  
console.log('Average marks of five subjects:',avg);
```

In Code Snippet 10, an array is declared, which stores the marks of five subjects obtained by a student. The total marks and average marks obtained by a student are calculated.

Figure 2.15 displays the output of Code Snippet 10.

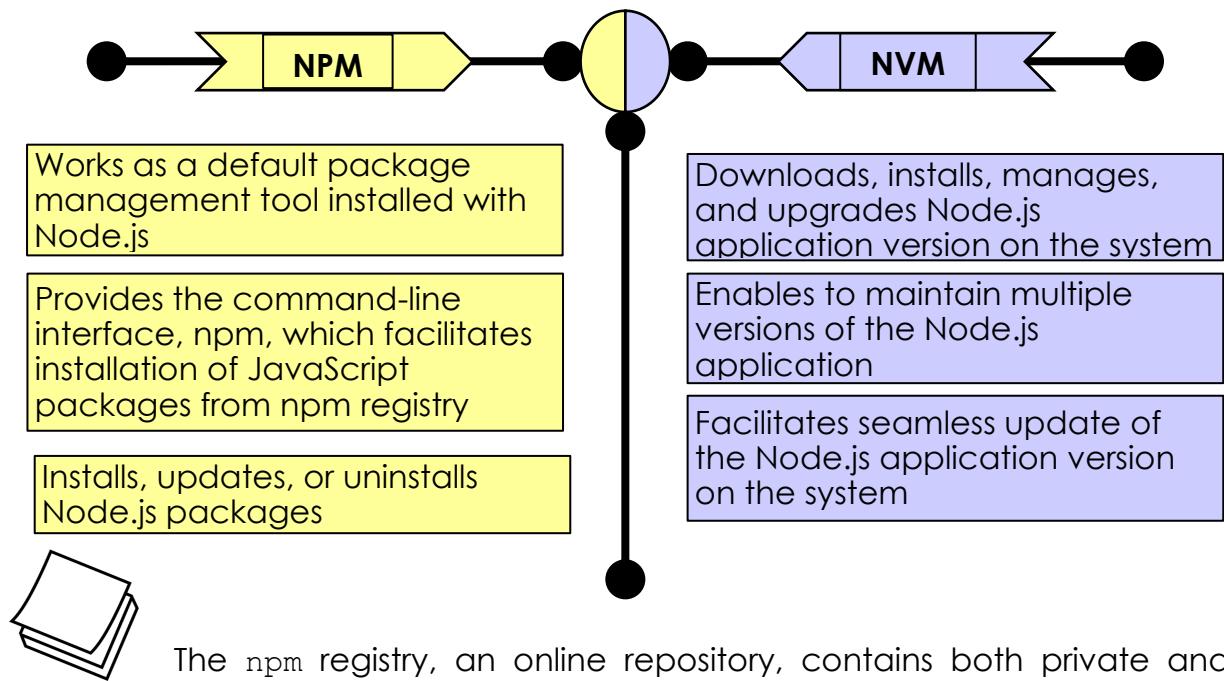
```
Average marks of five subjects: 84.6
```

Figure 2.15: Output of Code Snippet 10

Observe the program output. The average marks calculated is displayed.

2.5 NPM and NVM

Node Package Manager (NPM) and Node Version Manager (NVM) are important tools for managing Node.js and JavaScript development.

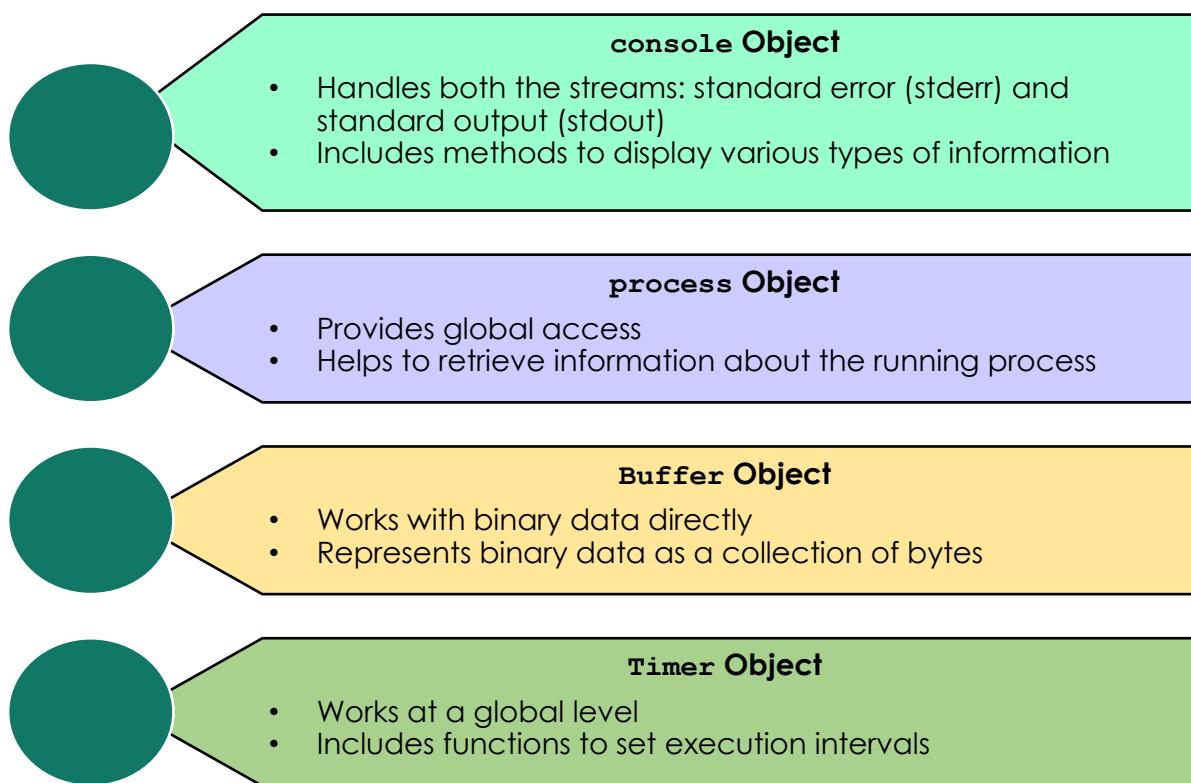


The `npm` registry, an online repository, contains both private and public packages.

2.6 Global Objects

There are certain objects in Node.js that are global in nature. Such objects can be used directly in Node.js applications. There is no compulsion to import any module before using the global objects.

Some of the commonly used global objects in Node.js are as follows:



Working with Global Objects

Here are some code snippets on the global objects.

console object

Consider an example code in Code Snippet 11 where the `console` object is demonstrated.

Code Snippet 11:

```
const console = require('console');

console.log('console.log is used to display a simple message');

number = 100
if (number > 150) {
  console.log('Number is less than 100');
}
else {
  console.error('Error message: Number is greater than 100');
}

console.error('console.error is used to display an error message');
```

```

a = 110;
b = 0;
a = a/b;

function divbyzero(a, b) {

    if (a == 0) {
        console.warn(`Result is ${b}`);
    }
    else {
        console.warn(`Divide by zero is not possible`);
    }
}
divbyzero(a, b);

console.warn('console.warn is used to display a warning message');

console.info('Node.js follows the concurrent mode of executing the tasks');

console.info('console.info is used to display an informational message');

```

In Code Snippet 11, various methods from the `console` class are included. Each method has its own functionality.

Figure 2.16 displays the output of Code Snippet 11.

```

console.log is used to display a simple message
Error message: Number is greater than 100
console.error is used to display an error message
Divide by zero is not possible
console.warn is used to display a warning message
Node.js follows the concurrent mode of executing the tasks
console.info is used to display an informational message

```

Figure 2.16: Output of Code Snippet 11

Observe the program output. The messages are displayed on the `console`.
process object

Consider an example code in Code Snippet 12 where the `process` object is demonstrated.

Code Snippet 12:

```
console.log(`Current directory: ${process.cwd()}`);
console.log(`Command-line arguments: ${process.argv}`);
```

In Code Snippet 12, the `process.cwd` method is used to access the current working directory of the program. The `process` object `argv` is used to print the command-line arguments supplied to the Node.js program using the `process.argv` property.

Figure 2.17 displays the output of Code Snippet 12.

```
C:\Node Js>node process_global.js
Current directory: C:\Node Js
Command-line arguments: C:\Program Files\nodejs\node.exe,C:\Node Js\process_global.js
```

Figure 2.17: Output of Code Snippet 12

Observe the program output. The current working directory and the command-line arguments are displayed.

Buffer object

Consider an example code in Code Snippet 13 where the `Buffer` object is demonstrated.

Code Snippet 13:

```
var buf_obj = Buffer.from('Learning');
console.log(buf_obj);
```

In Code Snippet 13, the string 'Learning' is converted into binary data.

Figure 2.18 displays the output of Code Snippet 13.

```
<Buffer 4c 65 61 72 6e 69 6e 67>
```

Figure 2.18: Output of Code Snippet 13

Observe the program output. The binary form of the string data is displayed.

Timer object

Consider an example code in Code Snippet 14 where the `Timer` object is demonstrated.

Code Snippet 14:

```
function display() {  
    console.log('Welcome to Session 2 of Node.js');  
}  
const timeinterval = setInterval(display, 1000);  
setTimeout(() => {clearInterval(timeinterval);}, 3000);
```

In Code Snippet 14, the `display` method is executed each second. The `setInterval` method executes the `display` method continuously in milliseconds, according to the intervals specified.

After three seconds, the `clearInterval` method is called with the interval ID given by the `setInterval` method. The `clearInterval` method terminates the continued execution of the `display` method.

Figure 2.19 displays the output of Code Snippet 14.

```
Welcome to Session 2 of Node.js  
Welcome to Session 2 of Node.js
```

Figure 2.19: Output of Code Snippet 14

Observe the program output. The messages are displayed only twice as the `display` function is terminated.

2.7 Summary

- Node.js, a JavaScript-based runtime environment, has various components to handle the client request, complete the execution, and provide the response.
- Node.js manages multiple client requests simultaneously.
- Node.js is single-threaded and follows an asynchronous programming model.
- REPL environment reads the input, evaluates the code, prints the output, and iterates through the tasks.
- Console-based applications perform their tasks in command-line environments.
- NVM manages the versions of Node.js application.
- NPM installs, updates, and uninstalls Node.js packages and modules.
- Global objects in Node.js are available throughout the lifetime of an application.

Test Your Knowledge

1. What is the primary function of the event loop in Node.js?
 - a) Storing client requests
 - b) Processing client requests with blocking I/O operations
 - c) Handling external resources
 - d) Sending responses to clients
2. Which of the following are benefits of building server-side applications in Node.js?
 - a) Easy scalability
 - b) Concurrent Request Handling
 - c) Single-Threaded Limitation
 - d) Memory Usage
3. Which method will you use to display the given text on console?

Node.js is an efficient runtime environment based on JavaScript. on the console.

 - a) `console.display()`
 - b) `console.write()`
 - c) `console.log()`
 - d) `console.message()`
4. Which of the following environment will you use to quickly test some JavaScript code?
 - a) Node.js IDE
 - b) Web browser console
 - c) REPL
 - d) Command prompt
5. Which keyword is used to declare variables in JavaScript?
 - a) `let`
 - b) `const`
 - c) `def`
 - d) `var`

Answers to Test Your Knowledge

1	d
2	a, b
3	c
4	c
5	d

Try It Yourself

1. Write a program to perform arithmetic operations on the given set of numbers. Initialize three variables `x`, `y`, and `z` with the values `27`, `7.2`, and `-3`, respectively. Perform the given operations and display the results:
 - a. Display the sum of `x` and `z`
 - b. Display the result of subtracting `y` from `z`
 - c. Display the product of `z` and `y`
 - d. Display the quotient of `x` divided by `y`
 - e. Display the remainder when `x` is divided by `y`
2. Write a program to perform operations on boolean values and numeric variables. Initialize a boolean variable `i` with the value `false` and two numeric variables `x` and `y` with the values `10` and `1`. Perform the given operations and display the results:
 - a. Display if the value of `x` is the same as the value of `y`
 - b. Check and display if the value of `x` is not same as the value of `y`
 - c. Display the boolean value of the negation of variable `i`
3. Write a program to perform string concatenation. Initialize two string variables, `text1` with the value `Welcome to` and `text2` with the value `JavaScript Programming`. Perform the given operations and display the results:
 - a. Display the value of the `text1` variable
 - b. Display the value of the `text2` variable
 - c. Concatenate the two strings `text1` and `text2` and display the result
4. Write a program to display information about cricketers. Define a JavaScript object named `cricket` with properties such as `Name`, `Run`, and `Wickets`. Perform the given operations and display the results:
 - a. Display the `Name` of the cricketer
 - b. Display the `Run` of the cricketer
 - c. Display the number of `Wickets` taken by team

5. Create a function named `salesPersonInMarketing` that takes the given parameters:

`name`: The name of the salesperson.

`age`: The age of the salesperson.

`experience`: The years of experience in sales.

`salesInQuarter1`: The sales achieved in the first quarter.

`salesInQuarter2`: The sales achieved in the second quarter.

The function should perform the given operations and display the results:

- a. Display the `name` of the salesperson
 - b. Display the `age` of the salesperson
 - c. Display the `experience` in sales
 - d. Calculate and display the total sales achieved in both quarters
6. Write a program to calculate the average test scores of a cricketer across five test matches performed in the World Cup. You have an array named `testScoresMatch` that has the test scores of the cricketer. Calculate the average test score and display the result.
- a. Create an array `testScoresMatch` with the test scores of a cricketer using this data [111, 55, 303, 324, 400]
 - b. Calculate the total `sum` of test scores
 - c. Calculate and display the `average` test score



SESSION 3

MODULES AND PACKAGES IN NODE.JS

Learning Objectives

In this session, students will learn to:

- Explain what modules are and its three types
- Explain the significance of packages
- Describe the purpose of Web framework and utility functions

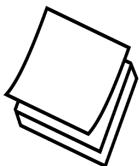
Modules are a ready-to-use, time-tested, reusable collection of functions or methods that can be used by other applications to make the process of development easier.

Packages are a collection of one or more modules grouped together as a single unit that can be published and installed in Node.js applications. Web frameworks are a collection of libraries and tools that provide specific features. Using Web frameworks in Node.js applications simplifies the development process and reduces the necessity to repeat the same code in multiple places.

This session will explain different types of modules and Web frameworks that can be used to build Node.js applications. It will explain how to install and use packages. Finally, it will discuss the importance of utility functions.

3.1 Modules in Node.js

Modules can be thought of as an application where functions or methods are wrapped up together. They are available as a ready-to-use instant application. Modules must be imported into the code to use the functions defined in them. The advantage of modules is that once defined, they can be reused inside any number of Node.js program files.



A method is also a function that is associated with an object. However, a function is not associated with any object. For example, a GET method can be defined for a HTTP Request or Response object. Require is a Node.js function that is used to import modules in a Node.js program.

Different types of modules in Node.js are follows:



3.1.1 Core Modules

Built-in modules are referred to as core modules, which are part of the Node.js platform. The built-in modules get installed along with the Node.js installation.

Table 3.1 lists some of the core modules.

Name of the Module	Description
http	Provides methods to create an HTTP server in Node.js.
assert	Provides methods with a set of assertion functions for testing.
fs	Provides methods to handle file system.
path	Provides methods to deal with file paths.
process	Provides methods to get information and control about the current Node.js process.
os	Provides methods to get information about the operating system.
querystring	Provides methods for parsing and formatting URL query strings.
url	Provides methods for URL resolution and parsing.

Table 3.1: Built-in Modules

3.1.2 Local Modules

A local module is a .js file that includes user-defined functions within it. It is created locally within a Node.js application and can be distributed across projects using the Node Package Manager (NPM). For example, a local module can be created to contain a function that returns the current date and time and can be distributed across projects.

3.1.3 Third-Party Modules

Third-party modules are modules that are not native to Node.js. They are created by some third party for use by other Web applications and are available online. They can be installed using the NPM.

Table 3.2 lists some of the modules of third-party modules.

Name of the Module	Description
Mongoose	It is a MongoDB Object data modeling library. It converts code from MongoDB to a Node.js server.
Express	It is a library for building Web and mobile applications. It manages the routing of a server.
Angular	It is a library to build robust Web applications. It is based on the MVC model.
React	It is a library for building the front-end interface of applications. It aids in creating creative and complex user interfaces.

Table 3.2: Third-Party Modules

An advantage of using a third-party module is that there is no overhead for writing the code and testing it. It is tested and available for instant use.

3.2 Working with Modules

Node.js provides the `require` function for importing modules. The syntax for using the `require` function is as follows:

```
var_module_name = require('module_name')
```

Here, `var_module_name` is the name of the variable to which the instance of the imported module is assigned. `module_name` is the name of the module that is being imported such as `os`, `http`, and `fs`.

Code Snippet 1 shows an example code that imports the built-in module `fs` using the `require` function and assigns it to the variable named `var_fs`. The code then prints a message to the console.

Code Snippet 1:

```
var _fs=require('fs');
console.log('An instance of File system object created');
```

To execute the code, type the code in VS Code and save it with the file name code-snippet1.js. Then, execute the .js file using the command as in:

```
node code-snippet1.js
```

Figure 3.1 shows the output of the execution of Code Snippet 1.

```
C:\Users\write\Sample code>node code-snippet1.js
An instance of File system object created
```

Figure 3.1: Execution of code-snippet1.js

After importing the module into the Node.js code, start using the methods in the module by accessing them with the instance of the module. For example, the `fs` module provides a set of methods to perform the given actions on a file:

- Opening a file
- Writing to a file
- Appending a file
- Reading a file
- Deleting a file

Table 3.3 describes these methods.

Method	Description
<code>fs.open</code>	Performs the specified operation on the specified file such as read, write, and append. If the file does not exist, creates a new file. Opens the file in the operation mentioned such as read or write or read and write. Creates the file if it does not exist.
<code>fs.writeFile</code>	Performs the write operation on a file. Creates a new file if the file does not exist. Overwrites the file if it already exists.
<code>fs.appendFile</code>	Appends the given contents to a file if it already exists. Else, creates a new file and appends the contents to it.
<code>fs.readFile</code>	Reads the contents of a file.
<code>fs.unlink</code>	Removes the file from the directory.

Table 3.3: File Handling Methods

3.2.1 Opening a File

The syntax for opening or creating a new file using the `open` method is as follows:

```
fs.open(path, flags[, mode], callback)
```

Here:

- `path` denotes the path of the file to open.
- `flags` denotes the operation for which the file is to be opened. The flags can be used to open the file for the read or write operation.
- `mode` specifies the mode of the file in which it has to be opened.
- `callback` is a callback function that is called after the file opening operation is complete.

Code Snippet 2 shows the code to create a file using the `fs.open` method. The code creates an empty text file named `filecontent.txt` in write mode.

Code Snippet 2:

```
var fs = require('fs');
fs.open('filecontent.txt', 'w', function (err) {
  if (err) throw err;
  console.log("File created successfully.")
});
```

In Code Snippet 2, `function(err)` is a parameter of the `fs.open` method. This is the callback function which is called if an error occurs as the `err` parameter is specified. This callback function throws the error specified by the `err` parameter. If no error occurs when opening the file, the `File created successfully` message is written to the console. Type the code in VS Code and save the file as `openfile.js`.

Figure 3.2 shows the output of the execution of the file.

```
C:\Users\write\Sample code>node openfile.js
File created successfully.
```

Figure 3.2: Execution of `openfile.js`

Figure 3.3 shows the contents of `filecontent.txt`, opened using Notepad.

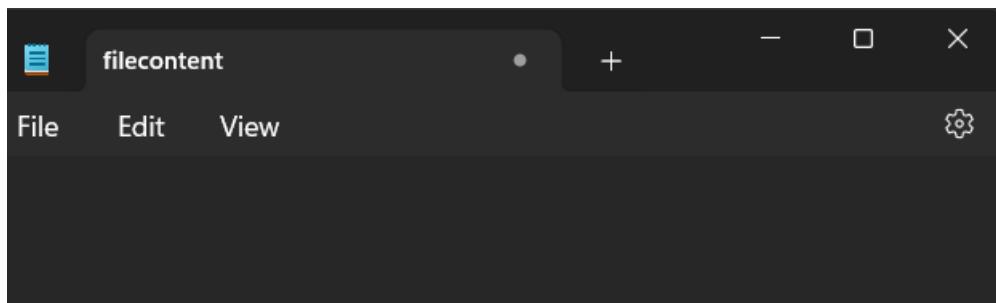


Figure 3.3: Contents of the filecontent.txt File

3.2.2 Writing to a File

The syntax for creating a file using the `writeFile` method is:

```
fs.writeFile(filename, data, [encoding],  
           [callback_function])
```

Here:

- `filename` denotes the name of the file.
- `data` is the content to be written to the file.
- `encoding` is an optional parameter that represents the way the data is represented inside the file system. ASCII, Base64, UTF64 are the common encoding formats. UTF64 is a commonly used encoding format for representing text files internally.
- `callback_function` is an optional parameter that specifies a callback function, which is called after the contents are written to the file. This function takes two parameters:
 - `err`: Is an `error` object that contains the error message if any error occurs.
 - `data`: Is a `buffer` or `string` that contains the file content.

Code Snippet 3 shows the code to create a file named `nodefile.txt` and write a custom message into it.

Code Snippet 3:

```
var fs = require('fs');  
fs.writeFile('nodefile.txt', 'Welcome to Node.js  
Programming. In this session, we learn about File  
System module.', function (err) {  
  if (err) throw err;  
  console.log('File is written successfully.');//  
});
```

In Code Snippet 3, the filename, data to be written, and the callback function are passed as parameters to the `writeFile` method. If an error occurs, the callback function is called by passing the error code, `err`. If the method executes successfully, then a message is displayed on the console.

Type the code in VS Code and save it with the file name `creatfile.js`. Execute the code. Figure 3.4 shows the output of the execution of code snippet 3.

```
C:\Users\write\Sample code>node creatfile.js  
File is written successfully.
```

Figure 3.4: Execution of `creatfile.js`

To verify that the text file is created, type the `dir *.txt` command at the command prompt. Find the file listed in the current directory.

Figure 3.5 shows the `nodefile.txt` file listed in the current directory.

```
C:\Users\write\Sample code>dir *.txt  
Volume in drive C is Windows-SSD  
Volume Serial Number is 26B1-8262  
  
Directory of C:\Users\write\Sample code  
  
09-11-2023 15:16 0 filecontent.txt  
03-11-2023 07:33 136 in the post..txt  
09-11-2023 16:22 83 nodefile.txt  
13-10-2023 05:19 0 openfile.txt  
13-10-2023 11:34 160 readfile.js.txt  
13-10-2023 15:09 42 sampleopen.txt  
13-10-2023 11:44 93 sampleread.txt  
13-10-2023 04:25 51 samplewrite.txt  
 8 File(s) 565 bytes  
 0 Dir(s) 74,353,127,424 bytes free
```

Figure 3.5: File Listed in the Directory

Figure 3.6 shows the contents of the `nodefile.txt` file.

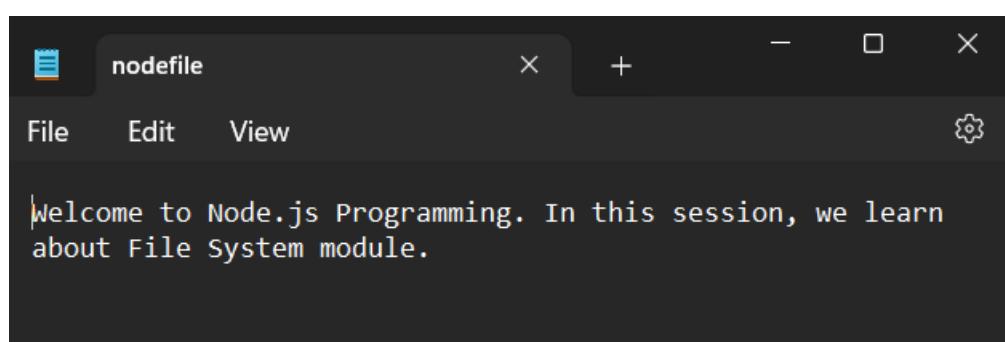


Figure 3.6: Contents of the File `nodefile.txt`

3.2.3 Appending to a File

Appending is adding contents to the end of the file. The `appendFile` method adds the given data to the end of a file. If the file does not exist, it creates a new file.

The syntax for appending a file using the `appendFile` method is:

```
fs.appendFile(filename, data, [options], callback)
```

Here:

- `filename` denotes the name of the file.
- `data` is the content to be written to the file.
- `options` is an optional parameter that represents encoding, mode, or flag.
- `callback_function` is an optional parameter that specifies a callback function, which is called after the contents are written to the file. This function takes two parameters:
 - `err`: Is an `error` object that contains the error message if any error occurs.
 - `data`: Is a `buffer` or `string` that contains the file content.

Code Snippet 4 shows the code to append a file.

Code Snippet 4:

```
var fs = require('fs');
var data="Here we create, read and write contents to the
file."
fs.appendFile('nodefile.txt', data, function (err)
{
    if (err) throw err;
    console.log("File Appended successfully.")
});
fs.readFile('nodefile.txt', function(err, data)
{
    if (err) throw err;
    console.log(data.toString('utf8'))
});
```

In Code Snippet 4, the `appendFile` method appends the string content stored in the variable named `data` to the `nodefile.txt` file. The `readFile` method then, reads and displays the contents of the `nodefile.txt` file to the console.

Save the code in a file named appendfile.js and execute the file. Figure 3.7 shows the output of the execution. The output displays contents of the nodefile.txt file with the appended contents.

```
C:\Users\write\Sample code>node appendfile.js
Welcome to Node.js Programming. In this session, we learn about File System module. Here we
create, read and write contents to the file.
File Appended successfully.
```

Figure 3.7: Execution of appendfile.js

Figure 3.8 shows the content of the text file nodefile.txt where the text data is appended.

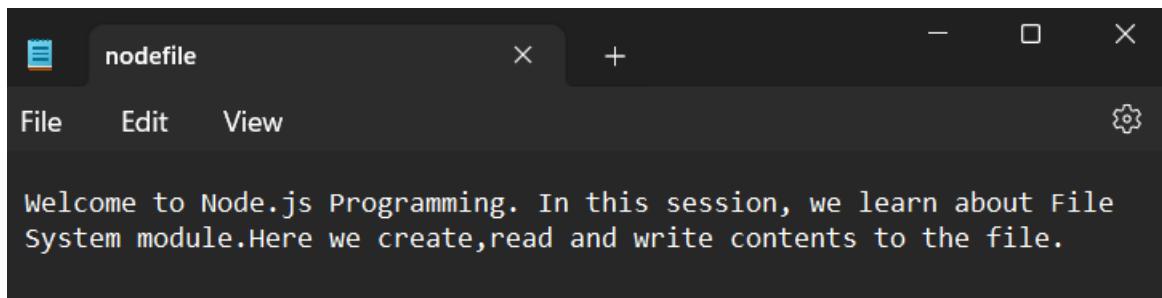


Figure 3.8: Contents of the File nodefile.txt

3.2.4 Reading a File

To read the contents of the text files, use the `readFile` method of the `fs` module.

The syntax of the `readFile` method is:

```
fs.readFile(filename, callback)
```

Here, `filename` specifies the path of the file to read. `callback` is the callback function that is called after the reading operation is complete.

Code Snippet 5 shows the code to read a file.

Code Snippet 5:

```
var fs = require('fs');
fs.readFile('nodefile.txt', function (err) {
    if (err) throw err;
    console.log("File read successfully.")
});
```

The code reads a text file called `nodefile.txt`.

To execute the code, save the code with the file name `readfile.js` and execute the file. Figure 3.9 shows the output of the execution. Contents of the `nodefile.txt` file are displayed in the console.

```
C:\Users\write\Sample code>node readfile.js
Welcome to Node.js Programming. In this session, we learn about File System module. Here we
create, read and write contents to the file.
```

Figure 3.9: Execution of `readfile.js`

3.2.5 Deleting a File

The `fs.unlink` and `fs.rmdir` methods help in deleting a file.

The syntax for deleting a file using the `unlink` method is:

```
fs.unlink(filename, callback)
```

Here, `filename` denotes the path of the file to delete. `callback` is an optional parameter that specifies a function to be called after the delete operation is complete.

Code Snippet 6 shows the code to delete a file.

Code Snippet 6:

```
var fs = require("fs");
fs.unlink('filecontent.txt', function(err) {
    if (err) throw err;
    console.log("File deleted successfully!");
});
```

The code calls the `unlink` method to delink the specified file from the current directory thereby deleting it from the directory.

Save the code with the filename `deletefile.js`. Execute the file. The file `filecontent.txt` is used as a sample file for the deletion process. Once the code executes, the file gets deleted. Figure 3.10 shows the execution of the Code Snippet 6.

```
C:\Users\write\Sample code>node deletefile.js
File deleted successfully!
```

Figure 3.10: Execution of `deletefile.js`

Figure 3.11 shows the execution of the `dir *.txt` command to show the listing of the `filecontent.txt` file in the current directory before and after the execution of the code in `deletefile.js`.

```
C:\Users\write\Sample code>dir *.txt
Volume in drive C is Windows-SSD
Volume Serial Number is 26B1-8262

Directory of C:\Users\write\Sample code

09-11-2023  16:41          0 filecontent.txt
03-11-2023  07:33          136 in the post..txt
09-11-2023  18:21          134 nodefile.txt
13-10-2023  05:19          0 openfile.txt
13-10-2023  11:34          160 readfile,js.txt
09-11-2023  17:42          195 sampleopen.txt
13-10-2023  11:44          93 sampleread.txt
13-10-2023  04:25          51 samplewrite.txt
                           8 File(s)           769 bytes
                           0 Dir(s)   74,326,167,552 bytes free

C:\Users\write\Sample code>node deletefile.js
File deleted successfully!

C:\Users\write\Sample code>dir *.txt
Volume in drive C is Windows-SSD
Volume Serial Number is 26B1-8262

Directory of C:\Users\write\Sample code

03-11-2023  07:33          136 in the post..txt
09-11-2023  18:21          134 nodefile.txt
13-10-2023  05:19          0 openfile.txt
13-10-2023  11:34          160 readfile,js.txt
09-11-2023  17:42          195 sampleopen.txt
13-10-2023  11:44          93 sampleread.txt
13-10-2023  04:25          51 samplewrite.txt
                           7 File(s)           769 bytes
                           0 Dir(s)   74,327,367,680 bytes free
```

Figure 3.11: Directory Listing

3.3 Packages

A package is a container of modules. A package groups all related modules together as a single unit so that they can be imported into applications.

To import a package into your application, use the `require` function. The functions within the modules of a package can be accessed using the instance that is returned by the `require` function.

3.3.1 Installation of Packages

To install packages, use the `npm` command-line tool. This command-line tool is installed with the Node.js installation. To verify the installation of `npm`, type the command `npm -v` at the command prompt.

Figure 3.12 shows the output of verifying the version of the `npm` tool.

```
C:\Users\write>npm -v  
10.1.0
```

Figure 3.12: Verifying the Version of npm

To update an older version of `npm` to the latest version, use the command as in:

```
npm install npm -g
```

Here,

- `npm` is the name of the command-line tool that runs the command.
- `install` is the command that installs the specified package.
- `g` is the flag to denote that `npm` should be installed globally on the system, rather than locally in the current project directory.

Install Packages Locally

To install packages locally in the current project directory, use the command as in:

```
npm install <package_name>
```

Here, `package_name` is the name of the package to be installed. For example, the command to install Angular CLI locally in your current project directory is:

```
npm install @angular/cli
```

Figure 3.13 shows the execution of this command at the command prompt and the installation of Angular packages.

```
C:\Users\write>npm install @angular/cli  
added 244 packages in 1m
```

Figure 3.13: Installation of the Package ‘angular’

Locally installed packages are available only to the Node.js applications present within the respective local directory. By default, NPM places the modules of a package into the local `node_modules` directory. To verify, type `dir` at the command prompt. Find the `node_modules` directory listed as shown in Figure 3.14.

```
C:\Users\write>dir  
Volume in drive C is Windows-SSD  
Volume Serial Number is 26B1-8262  
  
Directory of C:\Users\write  
  
09-11-2023 18:39 <DIR> .  
06-03-2023 20:28 <DIR> ..  
09-05-2021 13:16 <DIR> .android  
14-09-2021 11:25 <DIR> .fcc  
19-10-2023 10:55 <DIR> .ipynb_checkpoints  
28-06-2023 19:28 <DIR> .ipython  
07-08-2023 09:40 <DIR> .jupyter  
03-09-2023 10:26 <DIR> .matplotlib  
21-09-2023 12:49 13 .node_repl_history  
27-09-2023 11:25 <DIR> .vscode  
01-12-2020 18:39 <DIR> 3D_Objects  
02-12-2020 13:18 <DIR> ansel  
09-11-2023 17:25 381 appendfile.js  
19-10-2023 10:53 3,698 code.txt  
06-03-2023 20:39 <DIR> Contacts  
07-08-2023 11:52 366 create_table.py  
09-11-2023 18:38 151 deeltefile.js  
09-11-2023 18:40 151 deletefile.js  
03-04-2023 18:09 <DIR> Documents  
09-11-2023 12:02 <DIR> Downloads  
06-03-2023 20:39 <DIR> Favorites  
28-08-2023 19:18 417 filexttest.pkl  
28-08-2023 19:18 945 filextrain.pkl  
28-08-2023 19:18 241 fileytest.pkl  
28-08-2023 19:18 285 fileytrain.pkl  
19-07-2023 15:46 <DIR> flask_app  
17-06-2023 17:09 2,578,580 get-pip.py  
06-03-2023 20:39 <DIR> Links  
06-03-2023 20:39 <DIR> Music  
09-11-2023 20:08 <DIR> node_modules  
09-11-2023 05:56 <DIR> OneDrive  
09-11-2023 20:08 128,278 package-lock.json  
09-11-2023 20:08 187 package.json
```

Figure 3.14: `node_modules` Directory

Install Packages Globally

To enable Node.js applications present across the system to access a package, install the package globally by turning on the global mode. Append -g to the npm install command. For example, to install Angular CLI globally on the system, use the command as in:

```
npm install -g @angular/cli
```

3.3.2 Update Packages

To update a package, use the npm update command as in:

3.3.3 Uninstall Pack

```
npm update <packagename>
```

To uninstall a package, use the npm uninstall command as in:

```
npm uninstall <packagename>
```

Figure 3.15 shows the execution of this command at the command prompt and the uninstallation of Angular packages.

```
C:\Users\write>npm uninstall @angular/cli  
removed 244 packages, and audited 1 package in 1s  
found 0 vulnerabilities
```

Figure 3.15: Uninstalling a Package

3.4 Web Framework

A Web framework is a set of libraries or tools used for the speedy development of Web applications. They eliminate the requirement to write repetitive code. Frameworks act as a template to design the User Interface (UI) of Web applications. They also provide libraries for database access, session management, and other common tasks. Figure 3.16 shows some of the frameworks used with Node.js.

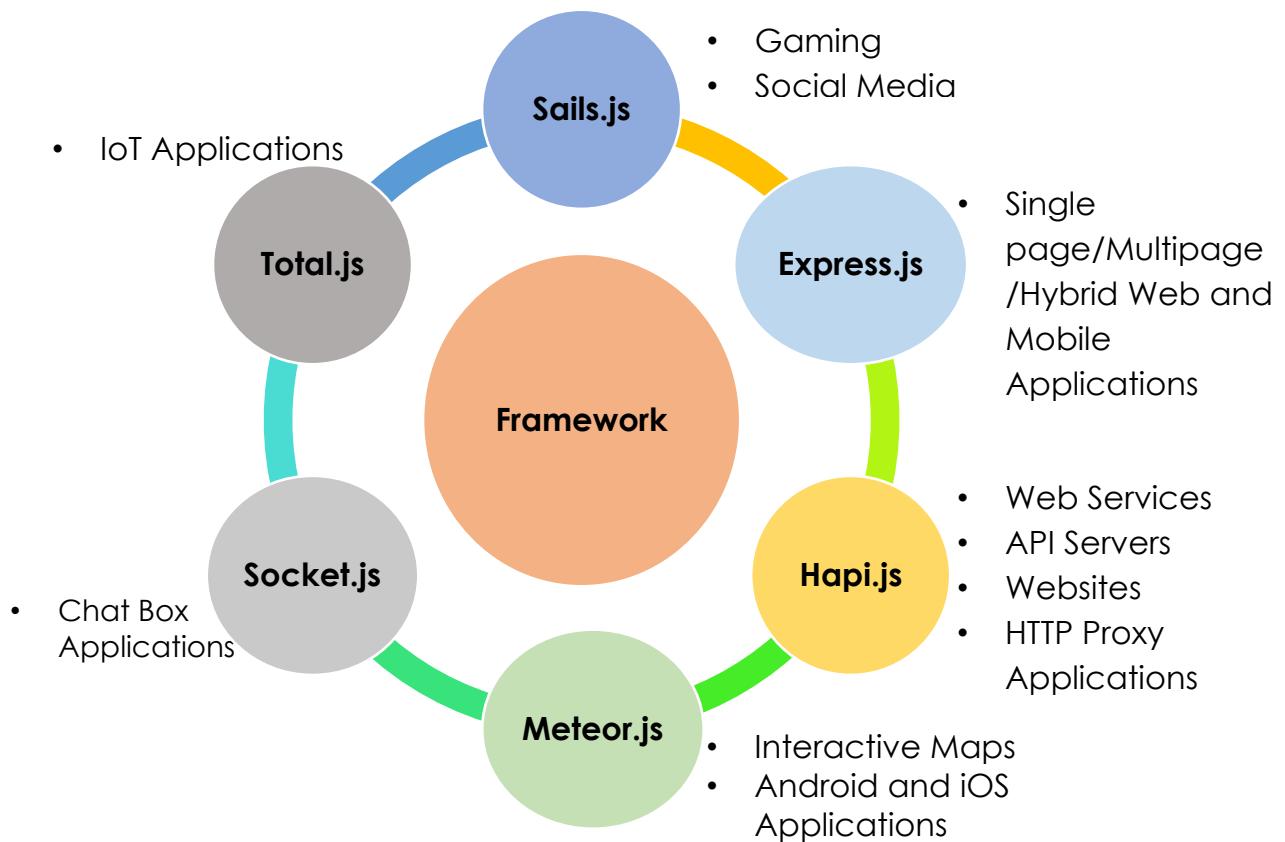


Figure 3.16: Frameworks of Node.js

A node framework is a Web framework that supports the development of applications using Node.js. Node frameworks allows developers to use JavaScript to develop both front-end and back-end of a Web application. There are three types of framework models in Node.js. They are as depicted in Figure 3.17.

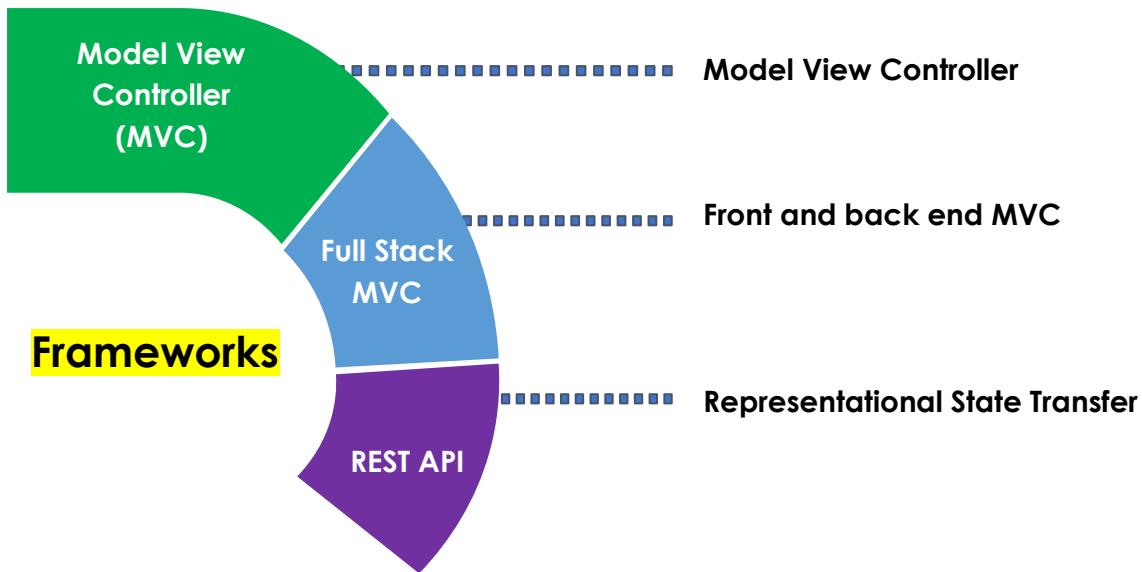


Figure 3.17: Types of Framework Models

3.4.1 MVC Framework

MVC stands as an abbreviation for the term, Model-View-Controller. This model splits the code logic into three parts, namely Model, View, and Controller. The Model component contains the data and business logic. The View component contains the logic to handle the user interface and the output. The Controller component contains the logic to handle the user inputs and coordinate with the Model and the View.

Express.js is modelled on the MVC Framework is Express.js. Figure 3.18 illustrates the working of the MVC model.

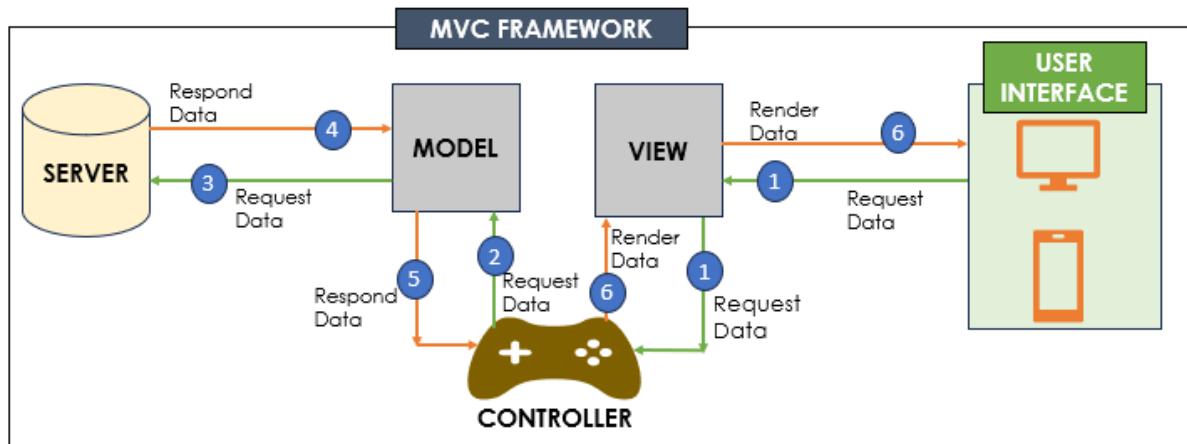


Figure 3.18: MVC Framework

In the MVC Framework model:

1. Users send requests to the Controller through the View.
2. The Controller in turn sends the requests to the Model.
3. The Model requests the database server for the required data.
4. The database server sends the required data as response to the Model.
5. The Model sends the response to the Controller.
6. The Controller in turn uses the View for rendering the response.

Thus, the Controller acts as a bridge between Model and View, controlling the flow of information.

3.4.2 Full Stack MVC Framework

A full stack MVC Framework model supports both front-end and back-end application development. It offers special libraries and tools for the development and management of applications.

3.4.3 REST API Frameworks

REST stands for Representational State Transfer. REST API defines a set of rules for communication between a server and a client using the HTTP methods. By using REST APIs, developers can create and use REST Web services in Node.js applications.

Simplicity and ease of use makes REST accessible to a wide range of developers. Moreover, developers are already familiar with the HTTP methods such as get, put, post, and delete that are part of the REST API.

3.5 Utility Functions

Utility functions are general-purpose functions that perform common or basic tasks required for the application development. Node.js provides several utility functions that are wrapped into different utility module as shown in Figure 3.19.

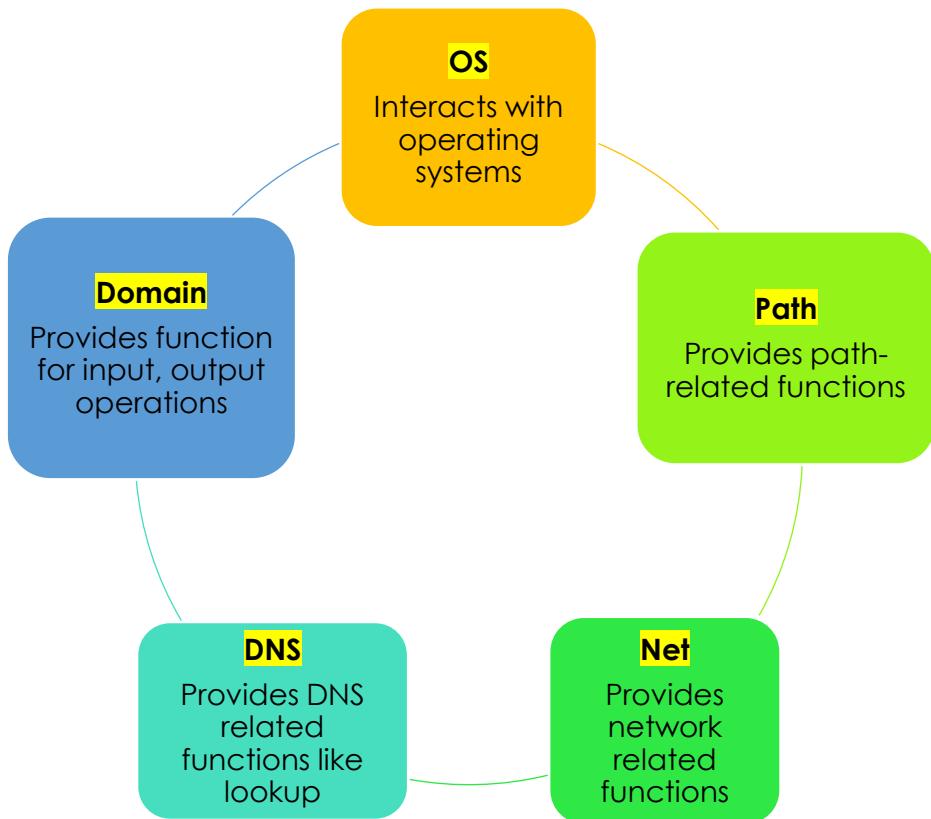


Figure 3.19: Utility Modules

3.5.1 Working with OS Module

Table 3.4 lists the methods of the os module.

Name of the Method	Features
os.type	Returns the name of the operating system
os.platform	Returns the platform of the operating system
os.totalmem	Returns the size of total system memory in bytes

Table 3.4: Methods of OS Module

Code Snippet 7 shows the code to print the operating system name, the platform, and the total amount of system memory in bytes using the methods of the os module.

Code Snippet 7:

```
const os = require('os');
console.log('Operating System type : ' + os.type());
console.log('Platform : ' + os.platform());
console.log('Total memory : ' + os.totalmem() + ' bytes.');
```

Save the code in a file name `osmethod.js` and execute the file.

Figure 3.20 shows the output of this code.

```
C:\Users\write\Sample code>node osmethod.js
Operating System type : Windows_NT
Platform : win32
Total memory : 8343445504 bytes.
```

Figure 3.20: Execution of `osmethod.js`

3.5.2 Working with Path Module

Table 3.5 lists the methods of the `Path` module.

Name of the Method	Features
<code>path.join(path1, path2, ...)</code>	Joins the given strings and creates a new string that can be used as a directory or path
<code>path.parse(path)</code>	Returns an object that provides the path-related details such as root, directory, filename, and file extension
<code>path.dirname(path)</code>	Returns the directories under the given path
<code>path.extname(filename or path)</code>	Returns the extension of a file or file in the path

Table 3.5: Methods of Path Module

Code Snippet 8 shows the code that uses the `Path` module. The code uses the `path.parse` method to display information about the root, base, directory, filename, and extension of file paths `/nodejs/sample code/sampleread.txt` and `/nodejs/sampleread.js`.

Code Snippet 8:

```
const path = require('path');
path1 = path.parse('/nodejs/sample
code/sampleread.txt');
console.log(path1);
path2 = path.parse('/nodejs/sampleread.js');
console.log(path2);
```

Figure 3.21 shows the output of execution of Code Snippet 8.

```
C:\Users\write\Sample code>node pathmethod.js
{
  root: '/',
  dir: '/nodejs/sample code',
  base: 'sampleread.txt',
  ext: '.txt',
  name: 'sampleread'
}
{
  root: '/',
  dir: '/nodejs',
  base: 'sampleread.js',
  ext: '.js',
  name: 'sampleread'
}
```

Figure 3.21: Information About Files

3.6 Summary

- Modules are ready-to-use collections of functions.
- Three types of modules are core modules, local modules, and third-party modules.
- Core modules are built-in modules that come with the installation of Node.js.
- Packages are collections of one or more modules, which are installed using the NPM tool.
- Frameworks are a collection of libraries and tools that speed up the application development.
- Utility functions perform general-purpose tasks that can be useful for communicating with operating systems and working with files and directories.

Test Your Knowledge

1. You are a Web developer in a pharmaceutical company. Which module will you use to create a Web server for handling HTTP requests?
 - a) fs
 - b) http
 - c) os
 - d) None of these

2. Which of the following are built-in modules in Node.js?
 - a) assert
 - b) querystring
 - c) express
 - d) mongoose

3. What will be output of the given code?

```
var_fs=require('fs');
console.log('Packages are a collection of modules');
```

 - a) Displays Nothing
 - b) Displays fs module is not recognized
 - c) Shows error in the code
 - d) Displays Packages are a collection of modules

4. You are working on a file management project as a lead developer. You want to check if a file already exists and overwrite the data in it. If the file does not exist then, you want to create a file and write the data to it. Which method will you use to perform the given scenario?
 - a) fs.writeFile
 - b) fs.appendFile
 - c) fs.open
 - d) None of these

5. You are required to communicate with a remote server to fetch and send data. Which framework will you choose to code without worrying about the network issues?
 - a) REST API Frameworks
 - b) MVC Framework
 - c) Full Stack MVC Framework
 - d) Other Framework

Answers to Test Your Knowledge

1	b
2	a, b
3	d
4	a
5	a

Try It Yourself

1. You are working on an admin file management application. Your application must create admin profile, update admin information, read admin data, append new information, and delete admin profile. Write a program using the file system module.
 - a) Create a file named `Admin.txt` and write the text Admin profile created successfully into it.
 - b) Create a file named `adminOpenFile.txt` with the text Admin file open successfully. Using a Node.js script, open this text file, read, and display its contents.
 - c) Further, in the script, append the text Admin data appended successfully to admin open file to the `adminOpenFile.txt` file. Read and display its contents.
 - d) Add code in the script to delete the `adminOpenFile.txt` and display a message Admin File deleted successfully.
2. Write a program to display the operating system, platform of the operating system, and size of total system memory in bytes.
3. Write a program to display all the directories in a specified path.
4. Write a program to display the version of `npm` installed in your system.



SESSION 4

BUILT-IN MODULES

Learning Objectives

In this session, students will learn to:

- Describe built-in modules in Node.js
- Explain `http` module to create an HTTP server and how to use it
- Explain how to use the `url` module to parse and work with URLs
- Explain how to use the `events` module to handle asynchronous events

Node.js comes with a variety of built-in modules that are readily available for use without any additional installations. These modules provide essential functionality to Node.js applications and are included as a part of the core library of Node.js. The built-in modules serve as a foundation for building a wide range of applications, from Web servers to file systems, and much more.

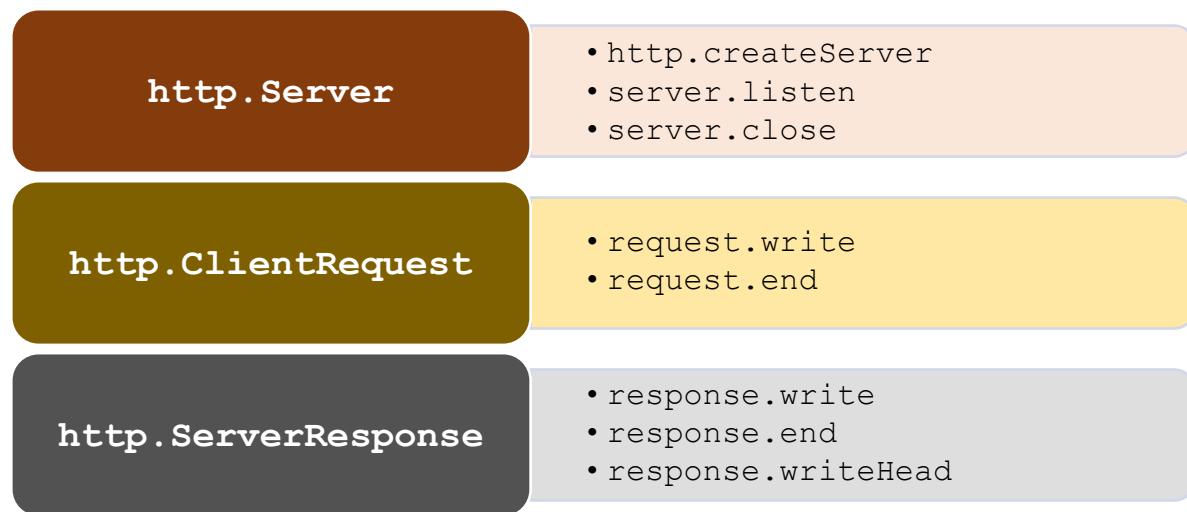
Built-in modules are optimized for the Node.js runtime and so are faster than third-party modules. These modules are maintained by the Node.js team and therefore, are reliable and secure.

This session will discuss three important built-in modules: `http`, `url`, and `events`.

4.1 http Module

The `http` module in Node.js allows users to create HTTP servers and make HTTP requests. This module provides various classes and methods that help implement interactions with HTTP protocols in an organized and efficient manner.

The most commonly used classes and their methods in the `http` module are as follows:



4.1.1 `http.Server` Class

The `http.Server` class is a core part of the `http` module. It represents an HTTP server that is designed to handle incoming HTTP requests and generate responses for clients.

Table 4.1 describes methods of the `http.Server` class.

Method	Description
<code>http.createServer([options,] requestListener)</code>	This method creates an HTTP server. It accepts an <code>options</code> object and a <code>requestListener</code> function. The <code>options</code> object is optional. This object includes two internal objects <code>IncomingRequest</code> and <code>ServerResponse</code> . These <code>options</code> object is used to specify the classes that will be used for these two internal objects. The <code>requestListener</code> function accepts two parameters, one is the <code>request</code> from the client and the other is the <code>response</code> to the <code>request</code> .

Method	Description
	This function is invoked for every incoming HTTP request.
<code>server.listen(port, [hostname], [backlog], [callback])</code>	This method binds and listens to the specified <code>port</code> with a <code>hostname</code> where the <code>hostname</code> is optional. Additionally, users can provide a <code>backlog</code> value and a <code>callback</code> function, which is executed when the server begins listening for incoming connections.
<code>server.close([callback])</code>	This method stops the server from accepting new connections. Optionally, users can provide a <code>callback</code> function that is executed when the server is closed for new connections.

Table 4.1: Methods of `http.server` Class

4.1.2 `http.ClientRequest` class

The `http.ClientRequest` class represents an outbound HTTP request. Instances of this class are used to define the specifics of an HTTP request, including the `request` method, headers, and data to be sent to the server.

Table 4.2 describes methods of the `http.ClientRequest` class.

Method	Description
<code>request.write(chunk[, encoding][, callback])</code>	This method writes data to the request body. It allows users to provide optional <code>encoding</code> and <code>callback</code> functions, which will be executed when the data has been successfully written to the request body.
<code>request.end([data][, encoding][, callback])</code>	This method signals that the request is complete and sends any data contained in the optional <code>data</code> parameter. Additionally, users can specify the <code>encoding</code> and an optional <code>callback</code> function that will be executed when the request is completed.

Table 4.2: Methods of `http.ClientRequest` Class

4.1.3 `http.ServerResponse` class

The `http.ServerResponse` class is generated automatically by an HTTP server in Node.js. It is used to send HTTP responses to clients. Instances of this class are provided as the second argument to the `request` callback function.

Table 4.3 shows methods of the `http.ServerResponse` class.

Method	Description
<code>response.write(chunk, [encoding], [callback])</code>	This method is used to write the response to be sent to the client. Small portions of the data to be sent are contained in the <code>chunk</code> parameter. The method is called multiple times until all the chunks of data are sent. Additionally, the method can contain the optional <code>encoding</code> parameter and the <code>callback</code> function. The <code>encoding</code> parameter specifies how data is encoded. The <code>callback</code> function is executed when the complete response data is written.
<code>response.end([data], [encoding], [callback])</code>	This method signals the end of the response. It uses the optional <code>data</code> parameter to include any remaining data to be sent as part of the response body. It can also specify the encoding used for the data and provide a <code>callback</code> function that is executed when the response is complete.
<code>response.writeHead(statusCode, [statusMessage], [headers])</code>	This method combines the setting of the status code, status message, and headers of the response into one function. These settings are used to set the HTTP status code, associated status message, and response headers, respectively, of the response.

Table 4.3: Methods of `http.ServerResponse` Class

4.2 Creating a Simple HTTP server

Consider that a user wants to create a simple HTTP server that responds with a plain text message `Welcome to Node.js HTTP Module for GET requests.` To do so, the user must perform these steps:

Creating the Server

1. Open a text editor, such as VS Code or Notepad.

2. To include the `http` module, type the code as:

```
const http = require('http');
```

This makes the `http` module available for use in the script.

3. To create an HTTP server using the `http.createServer` method, continue typing the code in Code Snippet 1 into the same file.

Code Snippet 1:

```
const server = http.createServer((req, res) => {
    // Request handling code here
});
```

In Code Snippet 1, the `req` parameter represents the incoming request and the `res` parameter represents the response of the server.

4. To handle different types of HTTP requests (GET, POST, and more) by examining the `req.method` property, in the `callback` function of the server, type the code given in Code Snippet 2.

Code Snippet 2:

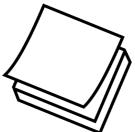
```
const server = http.createServer((req, res) => {
    if (req.method === 'GET') {
        // Handle GET request
    });
});
```

5. To set the format of the response from the server to plain text and specify the status code as 200, in the `if` block, add the code in Code Snippet 3.

Code Snippet 3:

```
res.writeHead(200, { 'Content-Type': 'text/plain' });
res.end('Welcome to Node.js HTTP Module \n');
```

In Code Snippet 3, the `res.writeHead` method is used to set the HTTP status code and headers and the `res.end` method is used to send the response body. The user can also use the `setHeader` method instead of the `writeHead` method to define the response type of the server.



In Node.js, HTTP headers are essential for specifying the type of response, plain text, HTML, JSON, or any other format, based on the requirements.

6. To specify the port on which the server should listen for the requests, after the `http.createServer` method, add the code in Code Snippet 4.

Code Snippet 4:

```
const port = 3000;
server.listen(port, () => {
  console.log(`Server is listening on port ${port}`);
});
```

Code Snippet 4 sets up the server to listen on port 3000 and prints a message to the console when the server is running.

7. Save the file as `servercreate.js`.

Starting the Server

1. To start the server, at the command prompt, type the command as:

```
node servercreate.js
```

The command executes and displays the message, as shown in Figure 4.1 indicating that the server is running and listening on port 3000.

```
C:\NodeJS Programs>node servercreate.js  
Server is listening on port 3000
```

Figure 4.1: Starting the Server

2. To test the server, open a Web browser and navigate to the URL: <http://localhost:3000>.

The message Welcome to Node.js HTTP Module for GET requests is displayed, as shown in Figure 4.2.

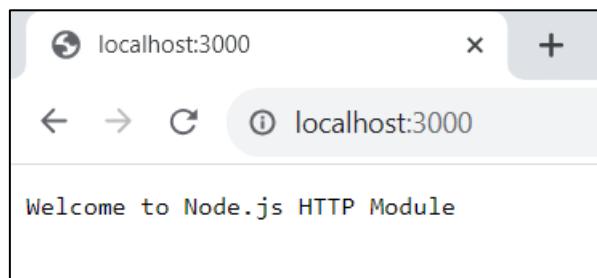


Figure 4.2: Response Message

4.3 Create an HTML Response for the HTTP Server

To get an HTML response from the HTTP server, users must include an HTTP header with the content type as text/html. To do so, perform these steps:

1. Type the code shown in Code Snippet 5 in the text editor.

Code Snippet 5:

```
const http = require('http');
const server = http.createServer((req, res) => {
  if (req.method === 'GET' && req.url === '/html') {
    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.write('<b>Welcome to Node.js</b><br/>');
    res.write('<span style="font-weight:bold; color:Red">In  
this session, we learn about HTTP module.</span>');
    res.end();
  } else {
    res.writeHead(404, { 'Content-Type': 'text/plain' });
    res.end('404 Not Found');
  }
});
```

```
const port = 3000;
server.listen(port, () => {
  console.log(`Server is listening on port ${port}`);
});
```

In Code Snippet 5, a server is created that serves an HTML page to the clients. Different routes can be defined based on the URL of the incoming request using the `req.url` property. In Code Snippet 5, the property is set such that if the requested URL is the root (`/html`), the server will send a response with an HTML page. Otherwise, the server will send a response with a `404 Not Found` message.

2. Save the file as `serverhtml.js`.
3. Open the Command Prompt window.
4. To start the server, at the Command Prompt, type the command as:

```
node serverhtml.js
```

The command executes and displays the message, as shown in Figure 4.3 indicating that the server is running and listening on port 3000.

```
C:\NodeJS Programs>node serverhtml.js
Server is listening on port 3000
```

Figure 4.3: Starting the Server

5. To test the server, open a Web browser and navigate to the URL:
<http://localhost:3000/html>.

Since, the URL is the root (`/html`), the message `Welcome to Node.js` In this session we learn about HTTP module is displayed as shown in Figure 4.4.

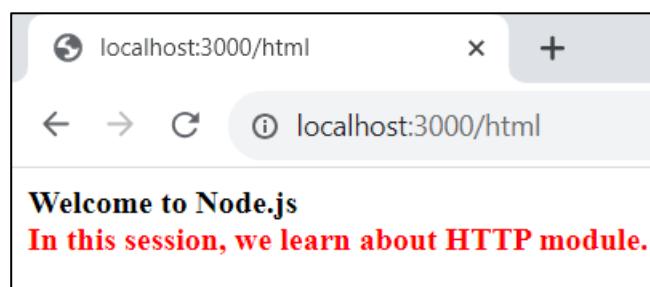


Figure 4.4: Response Message

6. Now, navigate to the URL:

<http://localhost:3000/>

Since, the URL is not the root, the 404 Not Found message is displayed, as shown in Figure 4.5.

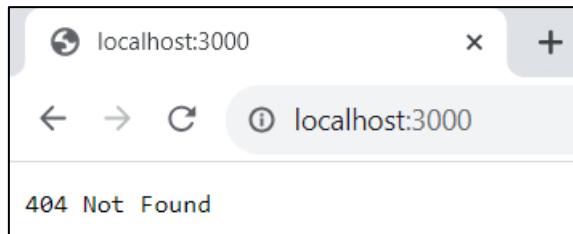


Figure 4.5: 404 Not Found Message

4.4 Create an HTTP Client Request Using the GET Method

Users can create an HTTP client request to send a GET request to a remote server. To do this, the users can use the JSONPlaceholder API as the remote server. This API provides a set of sample data for testing HTTP requests.

To create a GET request to the specified remote server (mentioned in hostname and path) and log the response data to the console, the user must perform these steps:

1. Type the code given in Code Snippet 6 in the text editor.

Code Snippet 6:

```
const http = require('http');
const hostname = 'jsonplaceholder.typicode.com';
const path = '/todos/61';
const options = {
  hostname: hostname,
  path: path,
  method: 'GET',
};
// Create an HTTP request
const req = http.request(options, (res) => {
  let data = '';
  res.on('data', (chunk) => {
    data += chunk;
  });
  res.on('end', () => {
    console.log(data);
  });
});
```

```
    });
  });

req.on('error', (error) => {
  console.error(error);
});

// Send the GET request
req.end();
```

Code Snippet 6 defines an `options` object that specifies the details of the HTTP request, including `hostname`, `path`, and `method`.

The `http.request(options, callback)` method is used to create an HTTP request where the `options` object contains the request details and the `callback` function will be executed when a response is received.

The `response` callback function listens for data chunks using the `res.on('data', ...)` event and accumulates the data.

When the response is complete, the `end` event is raised. At this point, the received data is sent to the console. The `error` event is raised on the `request` object to handle any errors that might occur.

2. Save the file as `requestget.js`.
3. To make a GET request to the specified remote server, in the Command Prompt window, at the command prompt, execute the command as:

```
node requestget.js
```

The command executes and displays the response data from the remote server, as shown in Figure 4.6.

```
C:\NodeJS Programs>node requestget.js
{
  "userId": 4,
  "id": 61,
  "title": "odit optio omnis qui sunt",
  "completed": true
}
```

Figure 4.6: Displays the Response Data

To summarize, developers can use the `http` module in their backend application to accept and present data through Web services.

For example, when developing a flight reservation system, the developers can use the `http` request object to:

- Send `GET` request to a route in the application for searching flights with search parameters such as origin, destination, and dates as part of the query string.
- Send `POST` requests to save the details of a specific flight booking and request an external gateway to receive payment for the flight booking.
- Send `GET` request to retrieve reservation details of a specific flight booking.
- Send `POST` request to cancel a specific flight booking.

Similarly, developers can use the `http` response object to:

- Receive a JSON response containing the flight details matching the search from the server.
- Receive a JSON response confirming the success or failure of the flight booking after interaction with external gateways for payment.
- Receive a JSON response with reservation details of a specific flight booking.
- Receive a JSON response confirming the success or failure of cancellation of a specific flight booking.

4.5 URL Module

The Uniform Resource Locator (URL) module in Node.js provides utilities for parsing and working with URLs. It allows users to:



4.5.1 Parsing the URL

The `URL` method is used to parse a URL string to create a new `URL` object. This method takes an input, which can be an absolute URL, (`https://www.sample.com/Sample1`) or a relative URL (`/Sample1`) with the base (`https://www.sample.com`). The `URL` object contains various components of the URL, such as protocol, hostname, port, pathname, and search parameters.

For example, consider that the user wants to parse the URL string: `https://www.jsonplaceholder.typicode.com/todos/61`

To do so, the user must perform these steps:

1. Type the code in Code Snippet 7 in the text editor:

Code Snippet 7:

```
const parsedUrl = new URL  
('https://www.jsonplaceholder.typicode.com/todos/61');  
console.log(parsedUrl);
```

In Code Snippet 7, the `URL` constructor is used to parse the URL string provided. The result of this parsing operation is stored in the `parsedURL` object, which contains various components of the URL, including protocol, hostname, port, and pathname.

Subsequently, the contents of the `parsedURL` object are displayed on the console using the line of code, `console.log(parsedUrl)`.

2. Save the file as `parseurl.js`.
3. To parse the given URL string, in the Command Prompt, type:

```
node parseurl.js
```

The command executes and displays the response data from the remote server, as shown in Figure 4.7.

```
C:\NodeJS Programs>node parseurl.js  
URL {  
  href: 'https://www.jsonplaceholder.typicode.com/todos/61',  
  origin: 'https://www.jsonplaceholder.typicode.com',  
  protocol: 'https:',  
  username: '',  
  password: '',  
  host: 'www.jsonplaceholder.typicode.com',  
  hostname: 'www.jsonplaceholder.typicode.com',  
  port: '',  
  pathname: '/todos/61',  
  search: '',  
  searchParams: URLSearchParams {},  
  hash: ''  
}
```

Figure 4.7: Response Data From Remote Server

The output shown in Figure 4.7 can also be obtained if the code is executed using Terminal mode in the VS Code.

4.5.2 Building the URL

The `url.format` method is used to create a URL string from a URL object or URL components.

Consider that the user wants to create a URL string using these URL components:

- `protocol: 'https:'`,
- `hostname: 'www.jsonplaceholder.typicode.com'`,
- `port: '8080'`,
- `pathname: '/todos/61'`

To do so, the user must perform these steps:

1. Type the code given in Code Snippet 8 in the text editor.

Code Snippet 8:

```
const url = require('url');
const urlObject = {
  protocol: 'https:',
  hostname: 'www.jsonplaceholder.typicode.com',
  port: '8080',
  pathname: '/todos/61'
};
const urlString = url.format(urlObject);
console.log(urlString);
```

Code Snippet 8, begins by importing the URL module using the `require` function, assigning it to a constant variable `url`.

Then, it creates an URL object called `urlObject` by specifying various components of the URL, including the `protocol ('https:')`, `hostname ('www.jsonplaceholder.typicode.com')`, `port ('8080')`, and `pathname ('/todos/61')`.

The `url.format` method is used to create a complete URL string from the individual components.

Finally, the URL string generated is displayed on the console.

2. Save the file as `buildurl.js`.

- To build the URL string, in the Command Prompt window, at the Command Prompt, execute the command as:

```
node buildurl.js
```

The command executes and displays the response data from the remote server, as shown in Figure 4.8.

```
C:\NodeJS Programs>node buildurl.js
https://www.jsonplaceholder.typicode.com:8080/todos/61
```

Figure 4.8: Response Data From Remote Server

4.5.3 Retrieving Specific Components of the URL

Users can also retrieve specific components of the URL using methods, such as:

Hash	• url.hash
Host	• url.host
Host Name	• url.hostname
Href	• url.href
Origin	• url.origin
Password	• url.password
Path Name	• url.pathname
Port	• url.port
Protocol	• url.protocol

For example, consider that the user wants to retrieve the protocol, hostname, and pathname for the specified URL. To do so, the user must perform these steps:

- Type the code given in Code Snippet 9 in the text editor.

Code Snippet 9:

```
const parsedUrl = new URL  
('https://www.jsonplaceholder.typicode.com/todos/61');  
console.log(parsedUrl.protocol);  
console.log(parsedUrl.hostname);  
console.log(parsedUrl.pathname);
```

Code Snippet 9, begins by creating an URL object called `parsedURL` by passing the URL provided to the `URL` constructor. Then, the specific components of the URL are displayed on the console.

2. Save the file as `getURLdetails.js`.
3. To build the URL string, in the Command Prompt window, at the Command Prompt, execute the command as:

```
node getURLdetails.js
```

The command executes and displays the response data from the remote server, as shown in Figure 4.9.

```
C:\NodeJS Programs>node getURLdetails.js  
https:  
www.jsonplaceholder.typicode.com  
/todos/61
```

Figure 4.9: Response Data From Remote Server

Consider the flight reservation system discussed in the `http` module. When developing a flight reservation system, the developers can use the `url` module to:

- Handle incoming requests with encoded data in query strings or paths.
- Build dynamic query strings defining the search criteria for flight booking.
- Parse incoming request URLs to obtain the search criteria for flight booking.
- Generate confirmation pages or booking summary pages with appropriate URLs.
- Redirect users to payment gateways or external services with properly constructed URLs.
- Create links to e-mail confirmations or itinerary details.

4.6 Events Module

The events module provides the `EventEmitter` class, which serves as the foundation for working with events in Node.js. Event emitters are objects that can emit named events, allowing servers to listen to those events and respond to them asynchronously. Users can also create custom event emitters by extending the `EventEmitter` class.

The `events` module is an event-driven architecture, which helps users to work with events and event emitters.

This `events` module allows users to:

Build applications where various components, such as objects or modules, communicate by emitting and listening for events.

Register event listeners that respond to I/O events, such as reading a file, receiving an HTTP request, or handling incoming data from a socket.

Handle user interactions, such as button clicks, mouse movements, and keyboard inputs.

Build real-time applications, such as chat systems, multiplayer games, and data streaming applications.

Notify clients of real-time changes or updates.

Create custom events specific to the requirements of the application to represent various states or actions and trigger them in specific conditions.

Table 4.4 shows the methods of the `events` module.

Methods	Description
<code>on(event, listener)</code>	Registers a listener function to be executed whenever the specified event is emitted.
<code>once(event, listener)</code>	Registers a one-time listener function that will be automatically removed after it is invoked the first time.
<code>removeListener(event, listener)</code>	Removes a specific listener from the listener array for the specified event.
<code>setMaxListeners(n)</code>	Sets the maximum number of listeners that can be added for an event. The default is 10.
<code>emit(event[, ...args])</code>	Emits the specified event with an optional list of arguments, triggering the registered listener functions for that event.

Table 4.4: Methods of events Module

4.6.1 Creating an Event-Driven Notification System

Consider that a user wants to create a notification system where events are used to trigger notifications when specific conditions are met. To do so, the user must perform these steps:

1. Type the code given in Code Snippet 10 in the text editor.

Code Snippet 10:

```
const EventEmitter = require('events');
class NotifyEmitter extends EventEmitter {}
const notifyEmitter = new NotifyEmitter();
// Register event listener for sending notifications
notifyEmitter.on('sendNotification', (message) => {
    console.log(`Sending notification: ${message}`);
});
// Simulate events triggering notifications
setTimeout(() => {
    notifyEmitter.emit('sendNotification', 'Meeting
Reminder');
}, 2000);
setTimeout(() => {
    notifyEmitter.emit('sendNotification', 'Reminder to
call client');
}, 4000);
setTimeout(() => {
    notifyEmitter.emit('sendNotification', 'Product
Delivery Reminder');
}, 6000);
```

Code Snippet 10 creates a custom event emitter class `NotifyEmitter` by extending the `EventEmitter` class from the `events` module. This class will serve as the foundation for the notification system.

Then, an instance of the `NotifyEmitter` class, named `notifyEmitter`, is created. This instance is used to work with event handling within the application.

The code uses the `.on(eventName, listener)` method to register an event listener. In this case, an event listener is registered for the `sendNotification` event.

This method takes two arguments:

- `eventName`: Specify the `eventName`, such as `sendNotification`.
- `listener`: Define the `listener` function, such as the message that executes when the event is emitted.

When the `sendNotification` event is emitted, the associated listener function executes, receiving a message as an argument and logging the message to the console.

To simulate events triggering notifications, the `setTimeout` function is used. After specific time intervals (in this case, after 2, 4, and 6 seconds), the `.emit(eventName, [args])` method is used to emit the `sendNotification` event. Each time, a different notification message is passed as the second argument.

2. Save the file as `eventemit.js`.
3. To trigger the event `sendNotification`, in the Command Prompt window, at the Command Prompt, execute the command as :

```
node eventemit.js
```

The command executes and simulates a notification system where events trigger notifications based on specific conditions, as shown in Figure 4.10.

```
C:\NodeJS Programs>node eventemit.js
Sending notification: Meeting Reminder
Sending notification: Reminder to call client
Sending notification: Product Delivery Reminder
```

Figure 4.10: Notification System

The `notifyEmitter.emit('sendNotification', message)` line triggers the `sendNotification` event, and the registered listener responds by sending the notification message to the console.

4.6.2 Managing Event Listeners in the `sendNotification` Event

Users can register multiple listeners for the same event. In case of multiple listeners, all of them will be invoked in the order they were registered.

The `.once(eventName, listener)` method registers a listener that is automatically removed after it is invoked once.

To register multiple listeners for the `sendNotification` event using `.once` method, perform these steps:

1. Type the code given in Code Snippet 11 in the text editor.

Code Snippet 11:

```
const EventEmitter = require('events');
class NotifyEmitter extends EventEmitter {}
const notifyEmitter = new NotifyEmitter();
// Register event listener for sending notifications
notifyEmitter.on('sendNotification', (message) => {
    console.log(`Sending notification1: ${message}`);
});
notifyEmitter.on('sendNotification', (message) => {
    console.log(`Sending notification2: ${message}`);
});
notifyEmitter.once('sendNotification', (message) => {
    console.log(`Sending notification-once: ${message}`);
});
notifyEmitter.emit('sendNotification', 'Meeting Reminder');
notifyEmitter.emit('sendNotification', 'Reminder to call client');
notifyEmitter.emit('sendNotification', 'Product Delivery Reminder');
```

2. Save the file as `eventonce.js`.
3. To trigger the `sendNotification` event, in the Command Prompt window, at the Command Prompt, execute the command as:

```
node eventonce.js
```

The command executes and displays the response, as shown in Figure 4.11.

```
C:\NodeJS Programs>node eventonce.js
Sending notification1: Meeting Reminder
Sending notification2: Meeting Reminder
Sending notification-once: Meeting Reminder
Sending notification1: Reminder to call client
Sending notification2: Reminder to call client
Sending notification1: Product Delivery Reminder
Sending notification2: Product Delivery Reminder
```

Figure 4.11: Response Data From Notification System

Figure 4.11 shows that two listeners are registered with `.on` method. These listeners respond to every `sendNotification` event, while the listener registered with the `.once` method responds only to the first `sendNotification` event. After that, this listener is automatically removed.

Consider the flight reservation system discussed in the `http` module. When developing a flight reservation system, the developers can use the `events` module to:

- Listen and handle incoming client requests.
- Update the availability of flights in real-time when a flight booking is in progress.
- Initiate updates related to a flight booking such as seat booking or meal booking.
- Send reminders for upcoming flights.
- Send reminders for Web check-in.

4.7 Summary

- Node.js built-in modules offer essential functionality for a wide range of tasks, including Web servers, file systems, and more, with benefits including performance, reliability, and interoperability.
- The Node.js `http` module simplifies the creation of HTTP servers and requests, providing various methods and classes for interacting with HTTP protocols.
- Common classes in the `http` module in Node.js include `http.Server`, `http.ClientRequest`, and `http.ServerResponse`.
- The Node.js `URL` module offers utilities for parsing and manipulating URLs.
- The Node.js `events` module uses the `EventEmitter` class to facilitate event-driven programming for building applications and handling I/O, user interactions, real-time systems, and custom events.

Test Your Knowledge

1. Which method is used to bind and listen on the specified port of the HTTP server?
 - a) `http.createServer`
 - b) `server.listen`
 - c) `server.close`
 - d) `request.write`

2. You are working on a Node.js application. You want to send an HTTP request to your remote server including the `request` method, headers, and data to be sent to the server. Which method from the `http.ClientRequest` class will you use?
 - a) `request.end([data] [, encoding] [, callback])`
 - b) `response.write(chunk, [encoding], [callback])`
 - c) `request.write(chunk [, encoding] [, callback])`
 - d) `response.writeHead(statusCode, [statusMessage], [headers])`

3. What will be the output of the given code in the browser?

```
const http = require('http');
const server = http.createServer((req, res) => {
    if (req.method === 'GET') {

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

    }
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('Node.js comes with a variety of built-in
modules');
});

server.listen(3000, () => {
    console.log(`Server is listening on port 3000`);
});
```

 - a) It will display Server is listening on port 3000
 - b) It will display Node.js comes with a variety of built-in modules
 - c) Error in the code
 - d) It will display nothing

In the given code, add the missing statements:

```
const http = require('http');
const server = http.createServer((req, res) => {
    if (req.method === 'GET' && req.url === '/html') {
        res.writeHead(200, { 'Content-Type': 'text/html' });
        res.write('<b>You are working on a
Node.js</b><br/>');
        res.write('<span style="font-weight:bold;
color:Blue">Now you have started working on
HTTP</span>');
        res.end();
    } else {
        // Add missing statements
    }
});
const port = 3000;
server.listen(port, () => {
    console.log(`Server is listening on port ${port}`);
});

a) res.writeHead(404, { 'Content-Type': 'text/plain' });
    res.end('404 Not Found');
b) res.end();
c) Nothing to add to this code
d) console.log(`Server is listening on port ${port}`);
```

4. What is the use of the `url.parse` method in Node.js?

- a) It is used to generate URLs
- b) It is used to parse a URL string into an object
- c) It is used to validate the structure of the URL
- d) It is used to convert an object to a URL

Answers to Test Your Knowledge

1	b
2	c
3	b
4	a
5	b

Try It Yourself

1. Create a simple HTTP server that responds with a plain text message: You have started working on HTTP server.
 - a) Import the `http` module.
 - b) Create an HTTP Server.
 - c) Handle Requests.
 - d) Send a Response You have started working on HTTP server.
 - e) Specify the listening Port as 3000.
 - f) Display the result on the browser.
2. You are working on an application in which you must get the response from the HTTP server in HTML format.
 - a) Import the `http` module.
 - b) Create an HTTP Server.
 - c) Check the HTTP request method and URL of the incoming request.
 - d) Add `if` condition, check if HTTP the request is a `get` method and check if the request URL is equal to `/html`.
 - e) Set status code to 200 and the response from the HTTP server is in plain text.
 - f) Add the response You are working on a `Node.js` application. Use font color as Blue and font weight as Bold.
 - g) If the requested URL is not the root (`/html`), then display 404 Not found.
 - h) Specify the port as 3000.
 - i) Display the result on the browser.
3. You are working on a chat application and you must create an Event-Driven Notification System.
 - a) Create a custom event emitter class `ChatEmitter`.
 - b) Create an instance of the `ChatEmitter` class, named `chatEmitter`.
 - c) Register an event listener for:
 - Customer joined
 - Customer left
 - New message
 - d) Simulate events triggering notifications using 3, 5, and 7 seconds.
 - e) Display the result on the browser.



SESSION 5

MORE BUILT-IN AND LOCAL MODULES

Learning Objectives

In this session, students will learn to:

- Explain the purpose and usage of `querystring` in Node.js
- Describe how to send emails using Node.js
- Describe the process of uploading files using Node.js
- Explain the concept of local Modules and their application in Node.js

Node.js includes several built-in and third-party modules, each tailored to simplify specific tasks. For example, the `querystring` module simplifies the parsing and manipulation of query strings within Universal Resource Locators (URLs), making data management in Web applications more straightforward.

The `nodemailer` module allows users to send emails programmatically in Node.js, enhancing communication within Node.js applications. Node.js also offers tools to simplify the process of uploading files, a common requirement in Web applications.

Local modules enable users to create custom code modules for better organization and reusability in Web projects.

5.1 `querystring` Module

Node.js provides the `querystring` module which is used to format and parse URLs. It is a valuable tool for passing data from one page to another in Web applications.

In simpler terms, this module simplifies working with URLs, allowing the user to add extra data in the form of key-value pairs to a URL. These pairs correspond to the specific information that is passed to a Web Server as part of an HTTP request. The information is embedded in the URL, typically after the `?`.

Table 5.1 shows two examples of query strings.

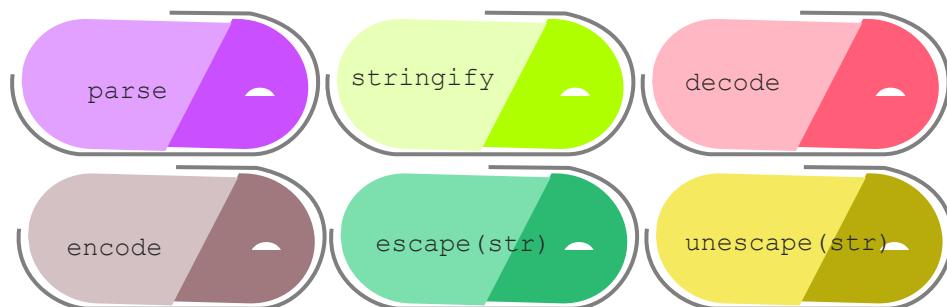
URL	Query String
https://exampleofquery.com/string/is? example=Query	example=Query
https://exampleofquery.com/path/of/ the/page/is?example=Query&partof= module	example=Query&partof=module

Table 5.1: Sample Query Strings

5.1.1 `querystring` Methods

Node.js offers a valuable resource in the form of the `querystring` module, which extends beyond URL parsing.

Some of the essential methods in the `querystring` module are as follows:



These methods are valuable to Web developers, providing tools to efficiently manage data embedded within URLs.

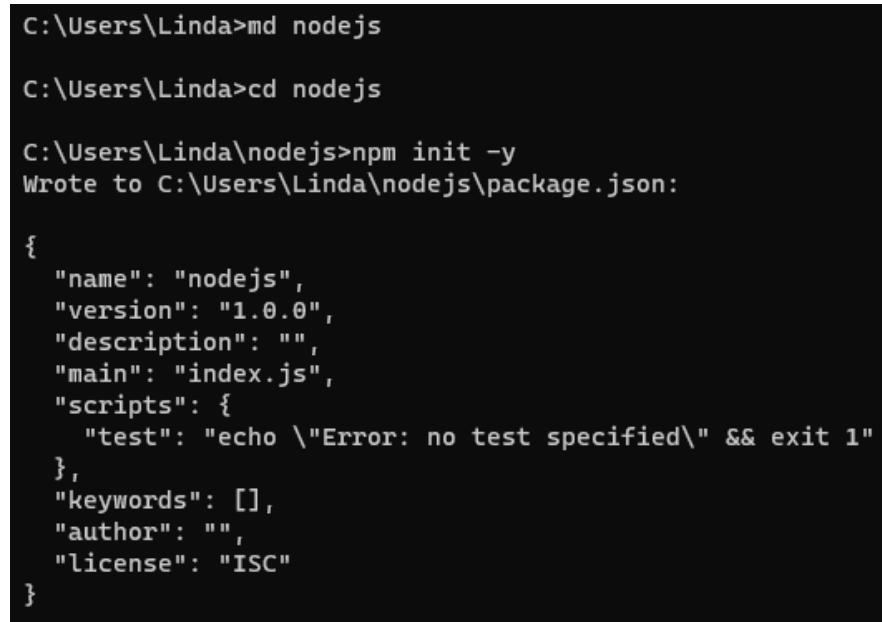
The first step to explore the use of each of these methods is to set up a basic Node.js application.

To set up a basic Node.js application:

1. Open the command prompt and navigate to a folder to create the Node.js project. If an existing Node.js project folder does not exist, create one.
2. To initiate a new Node.js project, run the command:

```
npm init -y
```

The `-y` flag is used to skip manual entry of project details. The command executes as shown in Figure 5.1.



```
C:\Users\Linda>md nodejs
C:\Users\Linda>cd nodejs
C:\Users\Linda\Nodejs>npm init -y
Wrote to C:\Users\Linda\Nodejs\package.json:

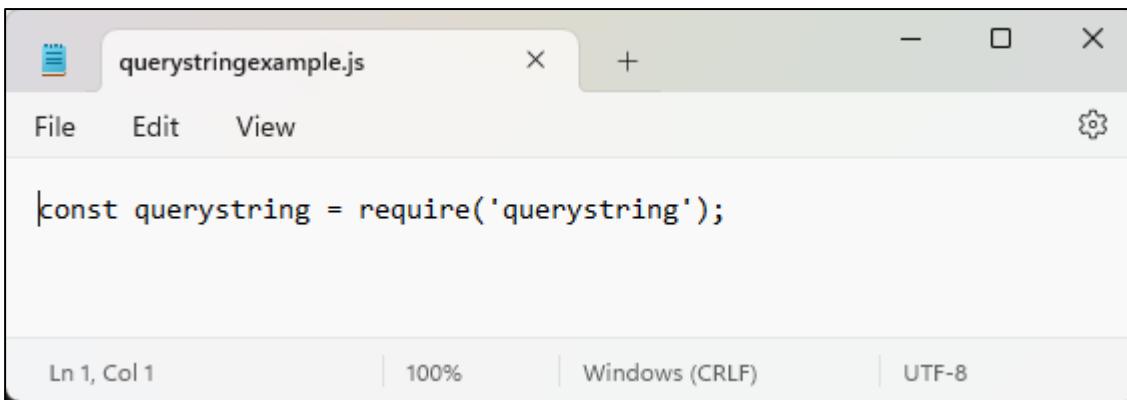
{
  "name": "nodejs",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \\"Error: no test specified\\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Figure 5.1: Basic package.json File

The command generates a basic `package.json` file, serving as the foundation of any Node.js project.

3. Within the project folder, create a new JavaScript file named `querystringexample.js`. The first line of the `querystringexample.js` file should include the import statement for the `querystring` module, as shown in Figure 5.2.

```
const querystring = require('querystring');
```



The screenshot shows a code editor window with a tab labeled "querystringexample.js". The main area contains the following code:

```
const querystring = require('querystring');
```

The status bar at the bottom indicates "Ln 1, Col 1" for the cursor position, "100%" for the zoom level, "Windows (CRLF)" for the line endings, and "UTF-8" for the encoding.

Figure 5.2: querystringexample.js File

4. Save the file.

The Node.js project is successfully set up. Now, the methods of the `querystring` module can be used.

5.1.2 `querystring.parse` Method

The `querystring.parse` method is utilized for parsing the URL query string into an object that contains key-value pairs. It is important to note that the resulting object is not a JavaScript object. Therefore, standard object methods such as `obj.toString` or `obj.hasOwnProperty` cannot be used with the object returned by the `parse` method.

By default, the method assumes the latest UTF-8 encoding format unless an alternative encoding format is specified. Only if the project has specific requirements for an alternative character encoding should the `decodeURIComponent` option be considered. However, it is advisable to stick with UTF-8 encoding, which is the standard and encompasses all international characters.

The syntax for the `querystring.parse` method is:

```
querystring.parse( str[, sep[, eq[, options]]] )
```

The `parse` method accepts four parameters:

str (Required)

This parameter specifies the query string that has to be parsed.

Sep (Optional)

This parameter specifies the substring that delimits the key-value pairs in the query string. The default and commonly used value is `&`.

Eq (Optional)

This parameter specifies the substring used to separate keys and values in the query string. The default and commonly used value is `=`.

Options (Optional)

This parameter specifies the object field used to modify the method's behaviour. It includes:

decodeURIComponent: A function that specifies the encoding format within the query string. The default value is `querystring.unescape()`.

maxKeys: A number that determines the maximum number of keys to be parsed. A value of 0 removes all counting limits, allowing the parsing of any number of keys. The default value is set at 1000.

The six options that can be used in the `querystring.parse` method are as follows:

- **Parsing a URL query string**

The `querystring.parse` method is used to parse a query string into a JavaScript object. Code Snippet 1 lists the code to do this.

Code Snippet 1:

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

let queryStringURL = "name=John&age=30&city>New+York";
let parsingObj = querystring.parse(queryStringURL);
console.log("Parsed Query 1:", parsingObj);
```

In Code Snippet 1, the `queryStringURL` variable contains the key-value pairs separated by the `&` symbol. The result is stored in the `parsingObj` variable. To try this example:

1. Open VS Code.
2. Type the code given in Code Snippet 1.
3. Save the file as `querystringexample.js`.
4. Open Command Prompt and run the command as:

```
node querystringexample.js
```

The parsed object is printed on the console as shown in Figure 5.3.

```
C:\Users\Linda\nodejs>node querystringexample.js
Parsed Query 1: [Object: null prototype] { name: 'John', age: '30', city: 'New York' }
```

Figure 5.3: Parsed Object

- **Stringifying an object into a URL query string**

The `querystring.stringify` method is used to convert an object into a URL query string. Code Snippet 2 lists the code to do this.

Code Snippet 2:

```
const querystring = require("querystring");
let dataObj = { name: "Alice", age: 25, city: "Los
Angeles" };
let queryString = querystring.stringify(dataObj);
console.log("\nStringified Query 2:", queryString);
```

In Code Snippet 2, a JavaScript object `dataObj` has the key-value pairs. The result is stored in the `queryString` variable and printed on the console as shown in Figure 5.4.

```
C:\Users\Linda\nodejs>node objtoqs.js  
Stringified Query 2: name=Alice&age=25&city=Los%20Angeles
```

Figure 5.4: Stringed Query

- **Parsing a query string with custom delimiters**

The `querystring.parse` method can be used to parse custom query strings. Code Snippet 3 lists the code to do this.

Code Snippet 3:

```
const querystring = require("querystring");  
  
queryStringURL = "itemA|1&itemB|2&itemC|3";  
parsingObj = querystring.parse(queryStringURL, "&", "|");  
console.log("\nParsed Query 3:", parsingObj);
```

In Code Snippet 3, a custom-formatted query string has the `&` character as the separator between key-value pairs and the `|` character as the delimiter between keys and values. This custom string is stored in the `queryStringURL` variable. The `querystring.parse` method uses the separator and custom delimiter to parse the given query string. The parsed data is stored in the `parsingObj` variable as an object. The result is printed on the console as shown in Figure 5.5.

```
C:\Users\Linda\nodejs>node customqs.js  
Parsed Query 3: [Object: null prototype] { itemA: '1', itemB: '2', itemC: '3' }
```

Figure 5.5: Parsed Object with Custom Delimiters

- **Parsing a query string with `maxKeys` set to 1**

The `maxKeys` option limits the number of keys parsed from the query string. Code Snippet 4 lists the code to demonstrate this.

Code Snippet 4:

```
const querystring = require("querystring");
queryStringURL =
"fruit=apple&fruit=banana&fruit=cherry&fruit=orange";
parsingObj = querystring.parse(queryStringURL, "&", "=", {
maxKeys: 1 });
console.log("\nParsed Query 4:", parsingObj);
```

In Code Snippet 4, the variable `queryStringURL` contains a query string with multiple key-value pairs. The `&` character separates pairs and the `=` character separates keys and values. Additionally, the `maxKeys` option is set to 1. Thus, when this code is executed, only one key-value pair will be parsed and the rest will be ignored. The result is stored in the `parsingObj` variable and printed on the console as shown in Figure 5.6.

```
C:\Users\Linda\nodejs>node maxkeys.js
Parsed Query 4: [Object: null prototype] { fruit: 'apple' }
```

Figure 5.6: Parsed Object with `maxKeys` Set to 1

Similarly, in Code Snippet 5, the `maxkeys` option is used to set the limit of key-value pairs to 2.

Code Snippet 5:

```
const querystring = require("querystring");
queryStringURL =
"fruit=apple&fruit=banana&fruit=cherry&fruit=orange";
parsingObj = querystring.parse(queryStringURL, "&", "=", {
maxKeys: 2 });
console.log("\nParsed Query 5:", parsingObj);
```

The result is printed on the console as shown in Figure 5.7.

```
C:\Users\Linda\nodejs>node maxkeys2.js
Parsed Query 5: [Object: null prototype] { fruit: [ 'apple', 'banana' ] }
```

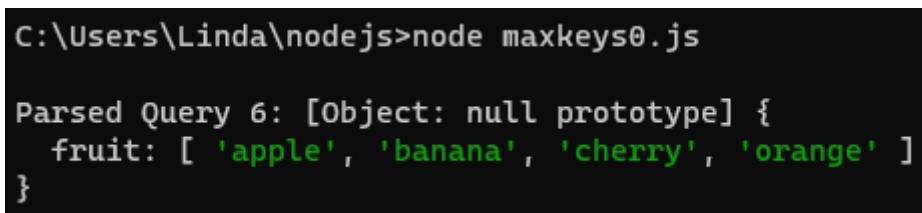
Figure 5.7: Parsed Object with `maxKeys` Set to 2

If the `maxKeys` option is set to 0, then all the key-value pairs will be parsed. Code Snippet 6 lists the code to demonstrate this.

Code Snippet 6:

```
const querystring = require("querystring");
queryStringURL =
"fruit=apple&fruit=banana&fruit=cherry&fruit=orange";
parsingObj = querystring.parse(queryStringURL, "&", "=", {
maxKeys: 0 });
console.log("\nParsed Query 6:", parsingObj);
```

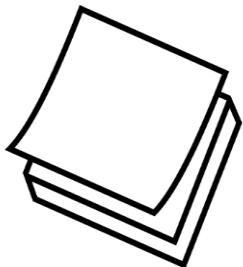
The result is printed on the console, as shown in Figure 5.8.



```
C:\Users\Linda\nodejs>node maxkeys0.js

Parsed Query 6: [Object: null prototype] {
  fruit: [ 'apple', 'banana', 'cherry', 'orange' ]}
```

Figure 5.8: Parsed Object with `maxKeys` Set to 0



The `querystring.unescape` method is involved in decoding key-value pairs in a query string. In practice, the `querystring.unescape` is not directly used in the code. The `querystring.parse` method handles this decoding process, making it unnecessary to use `querystring.unescape` separately.

5.1.3 `querystring.stringify` Method

The `querystring.stringify` method is used to produce a query string from a given object that contains the key value pairs. This method works exactly opposite to the `querystring.parse` method.

The resultant query string may contain strings, numbers, and/or Boolean values. The process of converting an object to a query string is termed serialization of the object.

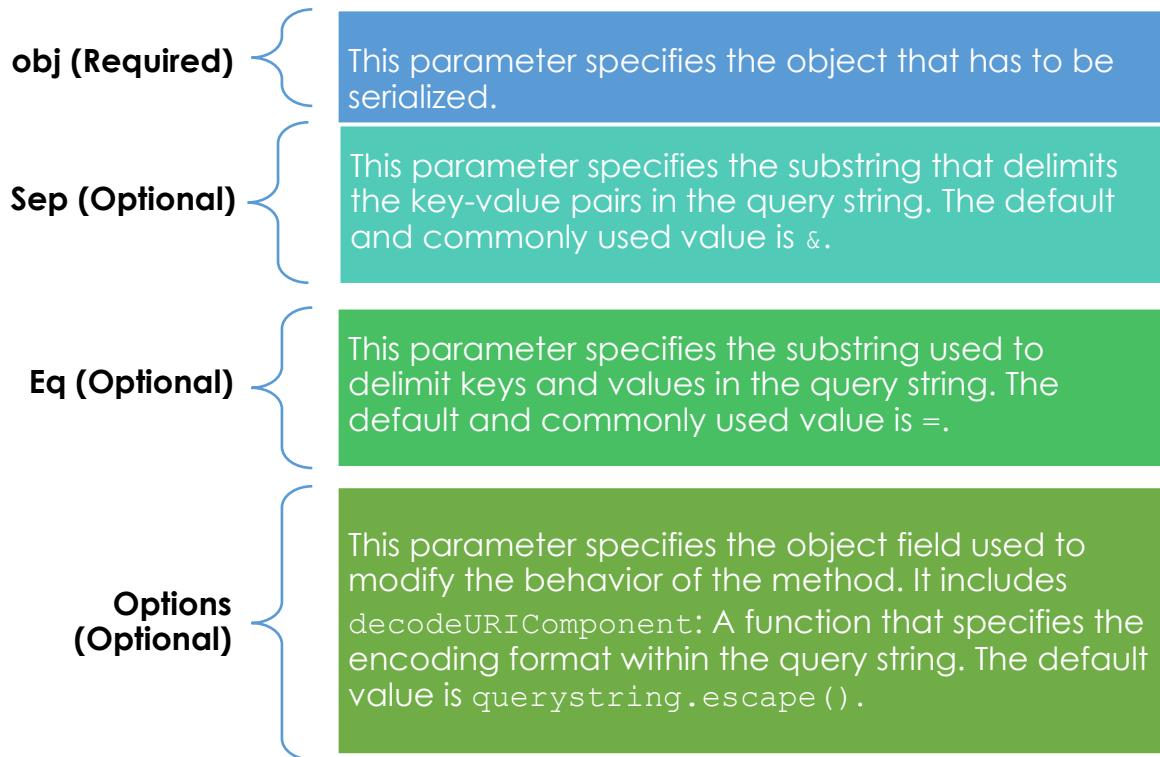
By default, the method assumes the use of the UTF-8 encoding format unless an alternative encoding format is specified. However, adhering to UTF-8 encoding is advisable as it is the standard encoding format encompassing all

international characters. In cases requiring alternative character encoding, the `decodeURIComponent` option should be used.

The syntax of the `querystring.stringify` method is:

```
querystring.stringify( obj[, sep[, eq[, options]]] )
```

The `querystring.stringify` method accepts four parameters:



The `querystring` module in Node.js allows the users to convert JavaScript objects into URL query strings in different ways.

Code Snippet 7 lists the code to demonstrate the default behavior of the `stringify` method when converting objects into URL strings. The code defines a user object, `user1` and initializes it with distinct properties such as `username`, `activity status`, and `roles`.

Code Snippet 7:

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

let user1 = {
  username: "Alice",
  active: true,
  roles: ["developer", "designer", "manager"],
};

let queryString1 = querystring.stringify(user1);
console.log("Query String 1:", queryString1);
```

The result serializes the `user1` object into a query string, using `&` as the separator and `=` to separate key-value pairs. The resulting query string is printed to the console as shown in Figure 5.9.

```
C:\Users\Linda\nodejs>node stringify.js
Query String 1: username=Alice&active=true&roles=developer&roles=designer&roles=manager
```

Figure 5.9: Stringed Query

Code Snippet 8 uses custom separators and delimiters to achieve object serialization. This showcases the flexibility of the `querystring` module in accommodating different formatting requirements.

Code Snippet 8:

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

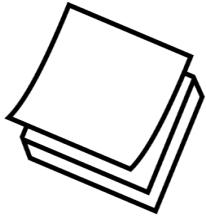
let user1 = {
  username: "Alice",
  active: true,
  roles: ["developer", "designer", "manager"],
};

let queryString1 = querystring.stringify(user1, ", ", ":");
console.log("Query String 1:", queryString1);
```

The result serializes the user1 object into a query string, using a comma followed by a space (,) as the separator and a colon : to separate key-value pairs. The resulting query string is printed to the console as shown in Figure 5.10.

```
C:\Users\Linda\nodejs>node stringifycus.js
Query String 1: username:Alice, active:true, roles:developer, roles:designer, roles:manager
```

Figure 5.10: Stringed Query With Custom Separators



The `querystring.escape` method is used to encode key-value pairs in a query string. In practice, the `querystring.escape` method is not used directly in the code. The `querystring.stringify` method takes care of this encoding process, making it unnecessary to use `querystring.escape` separately.

5.1.4 `querystring.decode` Method

The `querystring.decode` method is nothing, but an alias for the `querystring.parse` method.

The `decode` method takes an input and processes it. The result is a JavaScript object, where each key becomes a property. The associated values can be single values or arrays when there are multiple values for the same key.

Code Snippet 9 shows the code to decode an URL into JavaScript object using the `decode` method.

Code Snippet 9:

```
// Import the querystring module
const querystring = require("querystring");

// Define a custom URL query string to be parsed
let customQueryString =
"product=smartphone&brand=Samsung&brand=Apple&available=true
";

// Use the decode() method on the string
let parsedData = querystring.decode(customQueryString);

// Print the parsed object to the console
console.log("Parsed Data:", parsedData);
```

In Code Snippet 9,

- The `querystring` module is imported to access its functionality.
- A custom URL query string variable, `customQueryString`, is defined and initialized with a value. This query string value is made up of key-value pairs separated by &.
- The `decode` method is used to parse the data.
- The parsed object is displayed on the console.

The processed data is printed to the console as shown in Figure 5.11.

```
C:\Users\Linda\nodejs>node decode.js
Parsed Data: [Object: null prototype] {
  product: 'smartphone',
  brand: [ 'Samsung', 'Apple' ],
  available: 'true'
}
```

Figure 5.11: Decoded Data

5.1.5 `querystring.encode` Method

The `querystring.encode` method is nothing, but an alias for `querystring.stringify` method.

The `querystring.encode` method is used to serialize an object into a URL query string. This method converts the properties of an object into key-value pairs, using the default separators & for pairs and = for separating keys and values.

Code Snippet 10 shows the code to encode a JavaScript object using the `encode` method.

Code Snippet 10:

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

let userData = {
  username: "Alice",
  active: true,
  roles: ["developer", "designer", "manager"],
};

let queryString1 = querystring.encode(userData);
console.log("Serialized Data 1:", queryString1);
```

In Code Snippet 10,

- The `querystring` module is imported to access its functionality.
- A JavaScript object, `userData`, is defined and initialized with values.
- The `encode` method is used to string the data.
- The stringed query is displayed on the console.

The stringed query is displayed on the console as shown in Figure 5.12.

```
C:\Users\Linda\nodejs>node encode.js  
Serialized Data 1: username=Alice&active=true&roles=developer&roles=designer&roles=manager
```

Figure 5.12: Serialized Data

5.2 nodemailer Module

Node.js itself does not have a built-in email module. If users want to work with email in Node.js, they must typically use a third-party library such as `nodemailer`.

`nodemailer` is a widely used library that simplifies the process of sending emails from Node.js applications. To install `nodemailer`, perform these steps:

1. Open a Command Prompt window.
2. To install `nodemailer`, at the Command Prompt, run the command as:

```
npm install nodemailer
```

This command downloads and installs the `nodemailer` module on the local system as shown in Figure 5.12.

```
C:\NodeJS Programs>npm install nodemailer  
added 1 package in 986ms
```

Figure 5.12: Installing nodemailer

After `nodemailer` is successfully installed, users can use it in their Node.js application.

To create an application to send an email from a Node.js application, perform these steps:

1. Open a VS Code file.
2. Import the nodemailer module, using the code as:

```
var nodemailer = require('nodemailer');
```

3. To set up Gmail as the email transport, in the same file, type the code as given in Code Snippet 11.

Code Snippet 11:

```
var emailClient = nodemailer.createTransport({  
  service: 'gmail',  
  auth: {  
    user: 'your_email@gmail.com',  
    pass: 'your_password'  
  }  
});
```

In Code Snippet 11, the `nodemailer.createTransport` function sets up the email transport. Here, the Gmail service is configured as email transport. To configure this transport, the user must provide the Gmail email address and password for authentication.



Specifying the password directly in the code is not recommended. Instead, it is better to use application-specific passwords or OAuth tokens for enhanced security.

3. To specify the contents of the email, in the same file, type the code as given in Code Snippet 12.

Code Snippet 12:

```
var emailContent = {  
  from: 'sender@gmail.com',  
  to: 'recipient@example.com',  
  subject: 'Hello from Node.js',  
  text: 'This is a test email from Node.js.'  
};
```

In Code Snippet 12, the `emailContent` object specifies email details, such as the sender's email address (`from`), the recipient's email address (`to`), the email subject (`subject`), and the email body text (`text`).

4. To send the email, in the same file, type the code as given in Code Snippet 13.

Code Snippet 13:

```
emailClient.sendMail(emailContent, function(error,
info) {
  if (error) {
    console.log('Error: ' + error);
  } else {
    console.log('Email sent: ' + info.response);
  }
})
```

In Code Snippet 13, the `emailClient.sendMail` function sends the email. It requires the `emailContent` object as a parameter and a callback function to manage the result.

5. Save the file as `sendEmail.js`.
6. To run the application, open a Command Prompt window.
7. At the prompt, run the command as:

```
node senEmail.js
```

The application sends the email and displays an output as shown in Figure 5.13.

```
C:\NodeJS Programs>node sendEmail.js
Email sent: 250 2.0.0 OK 1700653891 q6-20020a17090a2e0600b00268b439a0cbsm1206910pj.d.23 - gsmtp
```

Figure 5.13: Email Message Successfully Sent

If there is any issue, such as incorrect username and password provided to log on the email service, the application will display an error as shown in Figure 5.14.

```
C:\NodeJS Programs>node sendEmail.js
Error: Error: Invalid login: 535-5.7.8 Username and Password not accepted. Learn more at
535 5.7.8 https://support.google.com/mail/?p=BadCredentials w13-20020a1709027b8d00b001c9b35287aesm9637219pll.88 - gsmtp
```

Figure 5.14: Error in Sending Email Message

Users can also use `nodemailer` to send emails to a group of recipients using Node.js. To do this, the email addresses must be specified in the `to` property by separating each email address with commas, as shown in Code Snippet 14.

Code Snippet 14:

```
var emailDetails = {
  from: 'your_email@gmail.com',
  to: 'recipient1@example.com, recipient2@example.com',
  subject: 'Hello from Node.js',
  text: 'This is a test email from Node.js.'
};
```

5.3 Uploading Files

Many Web applications allow users to upload files, such as images, documents, or other types of media. Node.js, however, lacks a built-in module dedicated to file upload processing. When it comes to managing file uploads in Node.js, users can use third-party modules, such as `formidable`.

The `formidable` module simplifies the process of receiving and processing uploaded files in Node.js applications. To use the `formidable` module in a Node.js application, perform these steps:

1. To install the `formidable` module, open a Command Prompt window.
2. At the Command Prompt, run the command as:

```
npm install formidable
```

This command downloads and installs the `formidable` module on the local system, as shown in Figure 5.15.

```
C:\NodeJS Programs>npm install formidable
added 6 packages, and audited 8 packages in 2s
1 package is looking for funding
  run `npm fund` for details
found 0 vulnerabilities
```

Figure 5.15: Installing `formidable` Module

After `formidable` is successfully installed, users can use it in their Node.js application.

To create an application to allow users to upload files from a Node.js application, perform these steps:

1. To create an HTML form that the users can use to upload a file, open a file, and type the code given in Code Snippet 15.

Code Snippet 15:

```
const var_http = require('http');

var_http.createServer(function (req, res)
{
    res.writeHead(200, {'Content-Type': 'text/html'});

    res.write('<form action="submitFile" method="post" enctype="multipart/form-data">');
    res.write('<b>Select file to upload:</b><br><br><input type="file" name="uploadfile"><br><br>');
    res.write('<input type="submit">');
    res.write('</form>');

    return res.end();
}).listen(3000, console.log("The HTTP Server is listening on port 3000."));
```

The code in Code Snippet 15, creates an HTTP server using the `createServer` method. This server displays an HTML file that shows a label `Select a file to upload:`, with a **Choose File** button to choose the file to upload and a Submit button to upload the file. The elements of the form are created using the `write` method of the `res` object. The server is also configured to listen on port 3000 using the `listen` method of the HTTP server.

2. To configure the HTTP server to parse the uploaded file, in the same file, after the declaration of the `var_http` variable, type the code as:

```
const var_formidable = require('formidable');
```

This code creates an instance of the formidable module, which is required to handle the file upload.

To parse the uploaded file when the user clicks the **Submit** button on the form, in the same file, after the opening braces of `var_http.createServer(function (req, res,)` add the code given in Code Snippet 16.

Code Snippet 16:

```
if (req.url == '/submitFile')
{
    var form = new var_formidable.IncomingForm();
    form.parse(req, function (err, fields, files)
    {
        if (err)
        {
            console.error('No file selected',
                         err);
            res.end('No file selected !!!');
            return;
        }
        else
        {
            res.write('File Submitted');
            res.end();
        }
    });
}
```

The code in Code Snippet 16 uses the `parse` function of the `formidable` instance to parse the uploaded file. It also checks for any error when parsing the file, such as a file not uploaded. It displays the message `File Submitted` if the parsing is successful. If an error is encountered, it returns the message `No file selected !!!`.

3. Move the code for displaying the upload file form into the `else` block as shown in Code Snippet 17.

Code Snippet 17:

```
else
{
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.write('<form action="submitFile" method="post"');
    res.write('enctype="multipart/form-data">');
    res.write('<b>Select file to upload:</b><br><br>');
    res.write('<input type="file" name="uploadfile"><br><br>');
    res.write('<input type="submit">');
    res.write('</form>');
    return res.end();
}
}).listen(3000, console.log("The HTTP Server is
listening on port 3000."));
```

4. Save the file as `uploadFile.js`.
5. To run the application, open a Command Prompt window.
6. At the command prompt, run the command as:

```
node uploadFile.js
```

This command starts the HTTP server that listens on port 3000 and displays an output as shown in Figure 5.16.

```
C:\NodeJS Programs>node uploadFile.js
The HTTP Server is listening on port 3000.
|
```

Figure 5.16: Starting the HTTP Server

7. Open a Web browser window and navigate to the URL <http://localhost:3000>.

The browser window displays the HTML page with the form to upload the file as shown in Figure 5.17.

A screenshot of a web browser window titled "localhost:3000". The address bar shows the same URL. The main content area displays a form with the instruction "Select file to upload:". Below this is a "Choose File" button with the text "No file chosen" next to it. At the bottom of the form is a "Submit" button.

Figure 5.17: Form to Upload File

8. Click **Choose File** and from the local system, select a file to upload. The file name appears next to the **Choose File** button as shown in Figure 5.18.

A screenshot of a web browser window titled "localhost:3000". The address bar shows the same URL. The main content area displays a form with the instruction "Select file to upload:". Below this is a "Choose File" button with the text "Picture1.jpg" next to it. At the bottom of the form is a "Submit" button.

Figure 5.18: Selecting File to Upload

9. Click **Submit**.

The file is uploaded and the message is displayed as shown in Figure 5.19.

A screenshot of a web browser window titled "localhost:3000/submitFile". The address bar shows the same URL. The main content area displays the message "File Submitted" in green text.

Figure 5.19: File Submitted

10. Open a new tab in the browser window and navigate to the URL <http://localhost:3000>.

The browser window displays the HTML page with the form to upload the file.

11. Click **Submit** without selecting any file.

The error message is displayed in the browser window as shown in Figure 5.20.

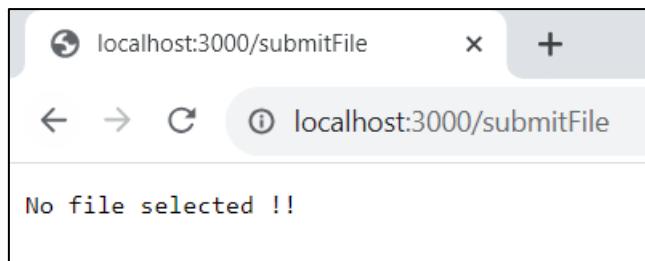


Figure 5.20: Error Message

5.4 Local Modules

Node.js developers often want to reuse pieces of code to save time. They achieve this by using local modules—custom-made modules created locally to fulfil specific requirements.

Local modules help in organizing the code into smaller, reusable parts, thereby simplifying the management and maintenance of the application. These modules also help implement encapsulation by hiding the code details and only exposing the functionality.

The local modules are created in a separate `.js` file and then, imported into the application, as required. These modules must be exported for them to be available for import into other applications. To export the modules, the `module.exports` function is used. This is a special function provided in Node.js, by default. This function is used to make a module available to other application. The application can import the exposed modules using the `require` method.

For example, consider the code in Code Snippet 18.

Code Snippet 18:

```
const MyFunc =  
{  
    msg: "This is a sample of using local modules.",  
  
    addNum: function (a, b)  
    {  
        const add_result = a + b;  
        console.log("The result is: ", a, "+", b, "=",  
                   add_result);  
    },  
    subNum: function (a, b)  
    {  
        if (a>=b)  
        {  
            const sub_result = a - b;  
            console.log("The result is: ", a, "-", b, "=",  
                       sub_result);  
        }  
        else  
        {  
            const err = "Cannot subtract"  
            console.log("The result is: ", a, "-", b, "=",  
                       err);  
        }  
    }  
}  
  
module.exports = MyFunc;
```

Code Snippet 18 creates a module named `MyFunc`. This module includes one variable `msg`, which stores a message. It also includes two functions `addNum` and `subNum`. The `addNum` function takes two parameters: `a` and `b`. It then, adds the parameters and stores the result in a variable named `add_result`. Finally, it displays a message. Similarly, the `subNum` function takes two parameters: `a` and `b`. It then, checks if `a` is larger than the `b`. If yes, it subtracts the second parameter from the first parameter and stores the result of the operation in a variable named `sub_result` and displays a message. If `a` is smaller than `b`, the message stored in the variable named `err` is displayed. Finally, the code uses the `module.exports` object to expose the `MyFunc` module.

This file is saved as `MyFunc.js`.

Next, consider the code in Code Snippet 19.

Code Snippet 19:

```
const localmod = require("./MyFuncs.js");

console.log(localmod.msg);

localmod.addNum(10, 40);

localmod.subNum(40, 10);

localmod.subNum(10, 40);
```

Code Snippet 19 uses the `require` method to import the `MyFunc` module. It then calls the variables and functions included in the module.

This file is saved as `useLocalModule.js`.

To see the working of the local modules, open a Command Prompt window. Then, at the Command Prompt, run the command as:

```
node useLocalModule.js
```

The application executes and displays the output as shown in Figure 5.21.

```
C:\NodeJS Programs>node useLocalModule.js
This is a sample of using local modules.
The result is: 10 + 40 = 50
The result is: 40 - 10 = 30
The result is: 10 - 40 = Cannot subtract
```

Figure 5.21: Working of Local Modules

5.5 Summary

- The `querystring` module simplifies the parsing and manipulation of query strings within URLs.
- The essential methods of the `querystring` module are `parse`, `stringify`, `decode`, `encode`, `escape(str)`, and `unescape(str)`.
- Node.js relies on third-party libraries such as `nodemailer` for email-related tasks, allowing users to send emails programmatically and improve communication within applications.
- Node.js lacks a built-in module for file uploads but supports the usage of third-party modules such as `formidable`, which simplifies the process of receiving and handling uploaded files in Node.js applications.
- Local Modules enable users to create custom code modules for better organization and reusability in applications.
- The `module.exports` object is a fundamental part of Node.js, included in every JavaScript file by default.

Test Your Knowledge

1. Which of the following module does not come under Built-in modules?

- a) Request Module
- b) Email Module
- c) Local Module
- d) Export Module

2. What is the purpose of `Querystring` module in Built-in modules?

- a) Ability to send emails programmatically in Node.js
- b) Simplify the process of uploading files
- c) Create custom code modules
- d) Simplify the parsing and manipulation of query strings within URLs

3. What is the correct output of the given code:

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

let foodObj = { cuisine: "Italian", food: "Bruschetta",
city: "Italy" };
let queryString = querystring.stringify(foodObj);
console.log("\nStringified Query 2:", queryString);
```

- a) Stringified Query 2:
`cuisine=Italian&food=Bruschetta&city=Italy`
- b) Stringified Query:
`cuisine=Italian&food=Bruschetta&city=Italy`
- c) Error in the code
- d) Stringified Query 11:
`cuisine=Italian&food=Bruschetta&city=Italy`

4. Identify the missing statements in the given code in order to get this output:

Query String 1:
playerName=John&active=true&hobby=cricket&hobby=football

Code:

```
const querystring = require("querystring");
let // Add missing code here
};
```

```
let queryString1 = querystring.stringify(player1);
console.log("Query String 1:", queryString1);

player2 = {
    playerName: "Anna",
    active: false,
    hobby: ["golf", "ice hockey "],
};
```

- a) player1 = {
 playerName: "John",
 active: true,
 hobby: ["cricket", "football"],
- b) player2 = {
 playerName: "Lee",
 active: true,
 hobby: ["cricket", "football"],
- c) player1 = {
 playerName: "John",
 active: false,
 hobby: ["cricket", "football"],
- d) player2 = {
 playerName: "Lee",
 active: false,
 hobby: ["cricket", "football"],

5. What is the use of `querystring.encode` method in Node.js?

- a) This method is used to decode the object
- b) This method is used to serialize an object into a URL query string
- c) This method is used in encoding key-value pairs
- d) None of these

Answers to Test Your Knowledge

1	a
2	d
3	a
4	a
5	b

Try It Yourself

1. You are working on a Node.js application. You want to parse and extract data from URL query string.
 - a) Import the `querystring` module.
 - b) Parse the query string using this data:
 1. `playerName: Jack`
 2. `city: Canada`
 3. `hobby: snowboarding`
 - c) Display parsed data in the console.
2. Create a program to convert JavaScript objects into URL query strings.
 - a) Define two book objects, `book1` and `book2`, each with distinct properties such as `bookName`, `activity status`, and `typeOfBook` using the given data:

```
book1
bookName= Captain Chuckle
activity status= false
typeOfBook= Drama
```



```
book2
bookName= Funny Bones
activity status= active
typeOfBook= Comedy
```
 - b) To serialize the object, use `book2`.
 - c) Use the `stringify` method on the object.
 - d) Display the result on the console.

3. Write a Node.js application to encode the data:

```
playerName: Jack,  
city: Canada,  
hobby: snowboarding
```

Also add code to decode the data:

```
String_decode:  
playerName=Jack&city=Canada&hobby=snowboarding&available=  
true
```

- a) Import the `querystring` module.
- b) Define a custom URL query string to be parsed.
- c) Use the `decode` method on the `String_decode`.
- d) Print the parsed object to the console.
- e) Define a custom object to be serialized.
- f) Use the `encode` method on the object.
- g) Display the result in the console.

4. You have created a program in Node.js. You want to send an email from your application.

- a) Install `Nodemailer` in your system.
- b) Import the `Nodemailer`.
- c) Create an email transport.
- d) Define the email content subject as `Node.js Working` and add body text as `email sent successfully`.
- e) Send the email.

5. You are creating Web application using Node.js and you want to implement file upload feature using Node.js.

- a) Install `Formidable` module in your system.
- b) Import the `Formidable` module.
- c) Create a form to upload a file.
- d) Set the response header and write the HTML form.
- e) Parse the uploaded file.
- f) Display the message, `File uploaded successfully`.



SESSION 6

INTRODUCTION TO EXPRESS.JS FRAMEWORK

Learning Objectives

In this session, students will learn to:

- Describe the Express.js framework
- Explain the architecture of Express.js
- List the advantages of using Express.js
- Explain the working of Express.js

Frameworks are tools that help in building applications. Frameworks are similar to blueprints of houses. These blueprints provide the plan for building the house including the layout and various measurements of the rooms. Following the plans ensure fast construction and reduces the chances of errors, thereby enhancing the quality of construction. The blueprints also use the language, structure, and symbols that are understood by all builders globally, thereby facilitating easy collaboration among builders, as required. Similarly, frameworks provide libraries and specify rules and guidelines that the developers can use to develop applications faster with minimal errors. This improves the quality of the code in their applications. Frameworks also provide a common language and structure to organize the code, making it easier for developers to collaborate with each other.

Express.js is one such framework that can be used for Node.js programming.

This session explains the Express.js framework, its architecture, working, and advantages.

6.1 What is Express.js?

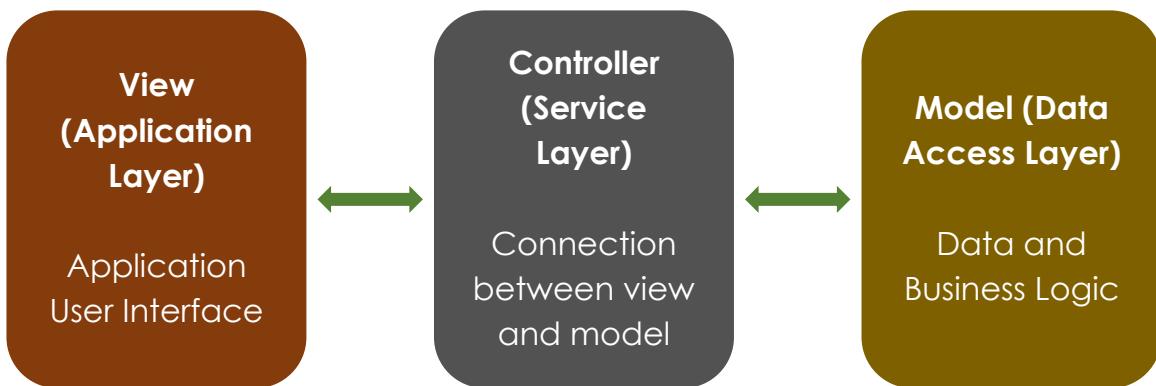
Express.js is a framework that is used for server-side programming with Node.js runtime environment. This framework provides features and libraries that can be used to develop Web applications, mobile applications, and APIs. Express.js can handle different types of Hypertext Transfer Protocol (HTTP) requests, generate dynamic Hypertext Markup Language (HTML), provide middleware functions, and serve static files. Express.js is an unopinionated framework that does not restrict developers to follow a prescribed way of developing applications. It allows developers to be flexible and innovative with their application development using the tools and techniques of their choice. Developers can also extend Express.js to add customized functionalities as required.



Middleware provides services such as authentication, security, and transaction processing to the applications.

6.2 Express.js Architecture

The Express.js framework is made up of three interconnected components: view, controller, and model, spread across three layers: application, service, and data access, respectively.



The **view** component makes up the application layer. It is responsible for generating the HTML content that is sent to the client.

The **model** component makes up the data access layer. It handles the data and the business logic and interacts with the underlying data sources.

The **controller** component makes up the service layer and provides the logic that acts as a bridge between the view and model components.

In Express.js the HTTP requests and responses are handled by the middleware. The middleware is a software that has access to the HTTP requests and responses and includes the next function that is used to move from one request to another. The middleware performs various functions including authentication, validation, and error handling. These functions can be chained together to form a pipeline that handles the requests and responses in a specific order.

6.3 Working of Express.js

Figure 6.1 shows the working sequence of the components of Express.js.

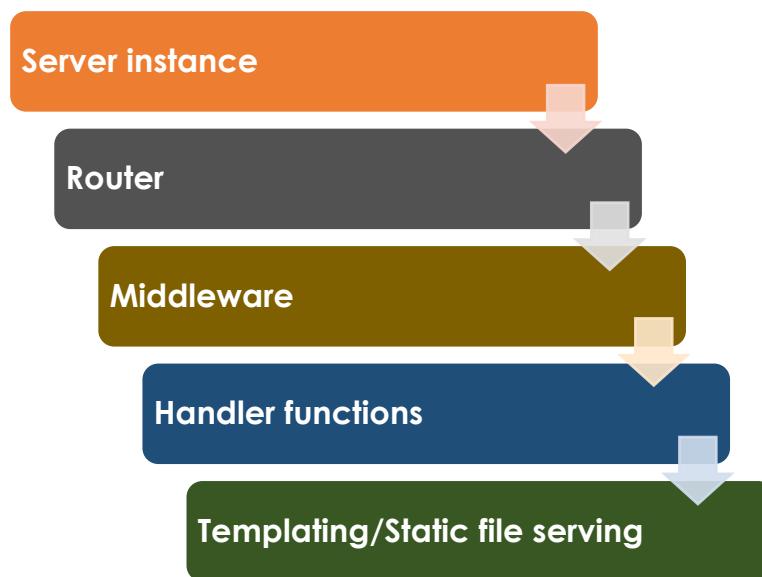


Figure 6.1: Components of Express.js

As shown in Figure 6.1, Express.js creates a server instance first. This instance listens for requests from the client on the specified port. When the instance receives a request, it passes it on to the router. The router, in turn, examines the request to identify the Universal Resource Locator (URL) and the HTTP method, such as POST, GET, DELETE, or PUT. Based on the URL and HTTP method, the router determines the handler function to which the request must be passed. The router adds the required details of the handler function and passes on the request to the middleware.

The middleware performs various tasks on the request including authentication, data compression, logging, and then passes on the request to the handler function. The handler function then, processes the request and generates the response. If required, the function involves a templating engine to generate dynamic HTML. The handler function can also access static files, such as images, Cascading Style Sheets (CSS), or JavaScript files, as required for generating the response. After the response is generated, it is sent to the server instance, which, in turn, sends the response to the client.

6.4 Why Use Express.js?

Express.js is a very popular choice with developers for developing Web applications and APIs. This is because it provides various advantages that makes it easier for the developer to build simple and standalone to complex and distributed Web applications. The advantages of using Express.js are:

- **It is a minimalist framework**

The Express.js framework includes a minimal set of core features. This makes the applications lightweight and efficient. It also allows developers to build applications quickly.

- **It is extensible**

The Express.js framework allows developers to add custom functionalities with ease. It also provides support for various third-party plugins and modules that can be used to extend the functionality of the framework.

- **It provides enhanced performance**

The Express.js framework is built on top of Node.js. Therefore, it inherits the event-driven, non-blocking model of Node.js, thereby providing enhanced performance and scalability to handle large number of connections concurrently.

- **It provides router and middleware support**

The Express.js framework uses routers that can handle different types of HTTP requests. It also uses middleware to intercept and modify the requests and responses by performing tasks such as logging, authentication, error handling, and data compression.

- **It is compatible with several databases**

The Express.js framework allows developers to work with various popular databases, including MongoDB, MySQL, and PostgreSQL, based on the application requirements. Developers can perform Create, Read, Update, and Delete (CRUD) operations on the databases and manage database transactions.

- **It is compatible with templating engines**

The Express.js framework can work with various templating engines, such as EJS and Pug, to render dynamic HTML content.

- **It can serve static files**

The Express.js framework can serve static files including CSS, images, JavaScript files directly from the server.

- **It is easy to use**

The Express.js framework includes easy and intuitive APIs that developers can learn and use easily. This framework is a popular choice among beginners and experienced developers.

6.5 Installing Express.js

Before installing Express.js, developers must ensure that Node.js has been installed.

To install Express.js, first create a folder to store all the Express.js applications. Express.js must be installed in this folder. Next, open a Command Prompt window and navigate to the newly created folder.

It is recommended that developers must install the `init` package before installing Express.js, although it is not mandatory. Installing the `init` package is especially useful when collaborating with other developers. This is because, the `init` package:

- Creates the `package.json` file, which handles the project dependencies and metadata.
- Stores the project information in the `package.json` file, making it easier to identify and share the project with others.
- Facilitates better project management and organization by providing a consistent and standardized project setup, making collaboration easier.

To install the `init` package, at the command prompt, execute the command as:

```
npm install init
```

Figure 6.2 shows the output of this command.

```
C:\MyApps>npm install init
added 2 packages in 376ms
```

Figure 6.2: Installing the `init` Package

To install Express.js, at the command prompt, execute the command as:

```
npm install express
```

Figure 6.3 shows the output of this command.

```
C:\MyApps>npm install express  
added 62 packages, and audited 65 packages in 4s  
11 packages are looking for funding  
  run `npm fund` for details  
  
found 0 vulnerabilities
```

Figure 6.3: Installing the express Package

To verify the installation of Express.js, at the command prompt, execute the command as:

```
npm ls express
```

The output of this command shows the version of Express.js installed, as shown in Figure 6.4.

```
C:\MyApps>npm ls express  
MyApps@ C:\MyApps  
'-- express@4.18.2
```

Figure 6.4: Version of Express.js

6.6 request and response Objects

The `request` and `response` objects in Express.js play an important role in handling and serving HTTP requests. The `request` object handles the HTTP requests coming from the clients. It provides information about the request, such as the URL, HTTP method, HTTP header, request body, and IP of the client. All this information helps the developers to handle the requests effectively by taking informed decisions. On the other hand, the `response` object is responsible for generating the responses for the requests and sending them back to the client.

Table 6.1 lists some key properties of the `request` and `response` objects.

Property	Description
request object	
body	Holds the data sent by the client. It provides the payload of the request in JSON or form-encoded format.
headers	Holds all the HTTP headers included in the HTTP requests sent by the client. It provides information, such as content type

Property	Description
	and authorization, that helps determine the preferences and capabilities of the client.
ip	Holds the IP address of the client from where the request has been sent. It provides information about the client network and location.
method	Provides information about which HTTP method has been used to send the client request. The method could be GET, POST, PUT, or DELETE.
params	Holds the route parameters that have been extracted from the URL. These parameters help determine the route for processing the client request.
query	Contains the query strings appended to the URL included in the client request. These query strings are stored in key-value pairs.
url	Stores the complete URL included in the request sent by the client. It contains the structure of the URL including the path, query string, and fragment.
response object	
app	Holds the reference to the instance of the Express.js application. It can be used to access application-level settings and functionalities related to the response.
charset	Specifies the encoding that is used for the response. This encoding ensures that the client can correctly interpret the response.
headers	Holds all the HTTP headers to be included in the HTTP response that will be sent to the client. It allows access to settings such as Content-Type, Content-Length, and Cache-Control.
statusCode	Specifies the status of the HTTP response. It takes the values such as 200 for OK or 404 for Not Found depending on the result of the request processing.
type	Specifies the content type of the response being sent to the client. It takes the values such as text/html, image.png, or application/json.

Table 6.1: Properties of request and response Objects

Table 6.2 lists the key methods of the `request` and `response` objects.

Method	Description
request object	
accepts(types)	Used to determine if the client is capable of handling the content format that it has requested
is (types)	Used to determine if the client is capable of processing the content format that it has requested
get(key)	Used to access specific header or cookie values
param(name)	Used to access specific route parameter using its name
signedCookie(name, secret)	Used to determine the value of the signed cookie using the secret key
unmount	Used to disconnect the request from the router
query(name)	Used to access the query string using its name
response object	
download(path, filename)	Used to provide files, such as images and documents, for download to the client
json(data)	Used to convert response to the JSON format and send it to the client
redirect(url)	Used to redirect the client to a different URL
render(view, locals)	Used to generate dynamic HTML content using the specified template engine and <code>locals</code> object
send(data)	Used to send specified data to the client as response
set(name, value)	Used to set the header of the response using the specified name and value
status(code)	Used to set the status of the HTTP response

Table 6.2: Methods of `request` and `response` Objects

6.7 GET and POST Methods

In any Web application, two common methods used to send requests are `GET` and `POST`. Table 6.3 provides the differences between these two methods.

GET Method	POST Method
This method is used to send small amount of data to the server.	This method is used to send large amount of data to the server.
This method appends the data to the URL when sending the request.	This method encloses the data in the request body when sending the request.
The requests sent using this method are stored in the browser history and browser log. They can be bookmarked.	The requests sent using this method are not stored in the browser history or browser log. They cannot be bookmarked.
The requests made using this method can only include ASCII characters.	The requests made using this method can include all types of data, including files and images.
The requests made using this method can be cached.	The requests made using this method cannot be cached.
This method offers low security to the request data. So, the request data can be intercepted and hacked easily.	This method offers high security to the request data. So, the request data cannot be easily intercepted or hacked.
This method is generally used to get data from a particular resource, such as getting product lists and user profiles.	This method is generally used to submit data to a particular resource, such as sending a form, uploading a file, editing existing data, or creating accounts.

Table 6.3: GET and POST Methods

Express.js provides a very simple way to handle these requests. It provides the `app.get` function to define handlers for `GET` requests and the `app.post` function to define handlers for `POST` requests.

To handle `POST` requests, developers must install an additional package, `body-parser`. To install this package, perform these steps:

1. Open a Command Prompt window.
2. At the Command Prompt, navigate to the folder where Express.js is installed.

- At the Command Prompt, run the installation command as:

```
npm i express body-parser
```

The command executes and installs the package as shown in Figure 6.5.

```
C:\MyApps>npm i express body-parser
added 2 packages, changed 2 packages, and audited 67 packages in 1s
11 packages are looking for funding
  run 'npm fund' for details

found 0 vulnerabilities
```

Figure 6.5: Installing the `body-parser` Package

Now, the system is ready for handling GET and POST requests. For example, consider that there is a Web page `StudentResult` on the local host. This page provides two text boxes to enter the name of the student and marks of the student, respectively. There is also a **Get Result** button on the page. When this button is clicked, the response is received based on the marks of the student. To achieve this using Express.js, perform these steps:

- To create a Web page, open the VS Code or Notepad application, and type the code given in Code Snippet 1.

Code Snippet 1:

```
<!DOCTYPE html>
<html>
  <title> Student Result </title>
  <body>
    <form action="/result" method="POST">
      Student Name: <input type="text"
name="stuName">
      <br>
      <br>
      Student Marks: <input type="text"
name="stuMarks">
      <br>
      <br>
      <input type="submit" value="Get Result">
    </form>
  </body>
</html>
```

2. In the folder where Express.js has been installed, save this Web page as StudentResult.html.

The next step is to create the Express.js application that will use the `app.get` method to open the `StudentResult.html` page when the users navigate to the localhost server. The user can then enter the student's name and marks and click the **Get Result** button. The application will then use the `app.post` method to display the result based on the marks. If the marks is more than 60, it will display `Result: Passed`, else it will display `Result: Failed`.

To create the Express.js application, perform these steps:

1. Open a new file.
2. To create an Express Web server, create a body-parser instance and configure the server to listen on port 3000. To do this, in the file, add the code given in Code Snippet 2.

Code Snippet 2:

```
//Creating an Express Web server
const var_express = require('express');
const var_app = var_express();

//Creating a body-parser instance
const var_parser = require('body-parser');
var_app.use(var_parser.json())
var_app.use(var_parser.urlencoded({ extended: false }))

//Configuring the Web server to listen on port 3000
const port = 3000;
var_app.listen(port, () => {
  console.log(`My Express App is running at
http://localhost:${port}`);
});
```

3. To send the `StudentResult.html` page using the `app.get` method, in the same file, after `const port = 3000`, add the code as shown in Code Snippet 3.

Code Snippet 3:

```
//Serving the StudentResult.html page using app.get()
var_app.get('/', (req, res) => {res.sendFile(__dirname +
'/StudentResult.html')});
```

4. To send the student's result as a response using the `app.post` method, in the same file, after the `app.get` method, type the code as shown in Code Snippet 4.

Code Snippet 4:

```
//Sending the result based on the marks using app.post()
var app.post('/result', (req, res) => {
  let stuName = req.body.stuName;
  let stuMarks = req.body.stuMarks;
  if (stuMarks > 60) {
    res.send('<p> Student Name: ' + stuName + '<p> Student
Marks: ' + stuMarks + '<p> Student Result: Passed');
  }
  else {
    res.send('<p> Student Name: ' + stuName + '<p> Student
Marks: ' + stuMarks + '<p> Student Result: Failed');
  }
  console.log(response);
})
```

5. In the same folder where Express.js is installed, save the file as `server.js`.
6. To execute the application, in the Command Prompt window, navigate to the folder where Express.js is installed.
7. At the command prompt, run the command as:

```
node server.js
```

The command executes and creates the Web server, as shown in Figure 6.6.

```
C:\MyApps>node server.js
My Express App is running at http://localhost:3000
|
```

Figure 6.6: Web Server Created

8. Open a browser.
9. Navigate to the URL, `http://localhost:3000`.

The `StudentResult.html` page opens as shown in Figure 6.7.

Student Result

localhost:3000

Student Name:

Student Marks:

Get Result

Figure 6.7: StudentResult Page

10. Enter the student's name and marks as shown in Figure 6.8.

Student Result

localhost:3000

Student Name: James

Student Marks: 50

Get Result

Figure 6.8: Student Details Entered

11. Click **Get Result**.

The result is displayed as shown in Figure 6.9.

localhost:3000/result

localhost:3000/result

Student Name: James

Student Marks: 50

Student Result: Failed

Figure 6.9: Result: Failed

12. Open another browser tab and navigate to the URL:
<http://localhost:3000/>.

13. Enter the student details as shown in Figure 6.10.

A screenshot of a web browser window titled "Student Result". The address bar shows "localhost:3000". The form contains two input fields: "Student Name: Kevin" and "Student Marks: 75", followed by a "Get Result" button.

Figure 6.10: Student Details Entered

14. Click Get Result.

The result is displayed as shown in Figure 6.11.

A screenshot of a web browser window titled "localhost:3000/result". The address bar shows "localhost:3000/result". The page displays the entered student details: "Student Name: Kevin", "Student Marks: 75", and "Student Result: Passed".

Figure 6.11: Result: Passed

The characteristic of the middleware is to move to the next stacked request after responding to the current request. To understand this better, perform these steps:

1. Open a file using VS Code or Notepad and add the code given in Code Snippet 5.

Code Snippet 5:

```
const var_express = require('express');
const var_app = var_express();
const var_port = 3000;
// Middleware function
let var_count = 1;
var_app.use((req, res, next) => {
  console.log('This is a simple middleware that logs a message
for request:' + var_count);
  var_count = var_count+1;
  next(); // Call next to pass control to the next middleware
or route handler
});
```

```
var_app.get('/', (req, res) => res.send('Welcome to My Express App'));
var_app.listen(var_port, () => {
  console.log(`My Express App is running at http://localhost:${var_port}`);
});
```

2. In the folder where Express.js is installed, save the file as middlewareSample.js.
3. Open the Command Prompt window and navigate to the folder where Express.js is installed.
4. At the command prompt, run the command as:

```
node middlewareSample.js
```

The command starts a server as shown in Figure 6.12.

```
C:\MyApps>node middlewareSample.js
My Express App is running at http://localhost:3000
```

Figure 6.12: Server Started

5. Open a browser window and navigate to the URL, <http://localhost:3000>.

The Web page shows the message as shown in Figure 6.13.

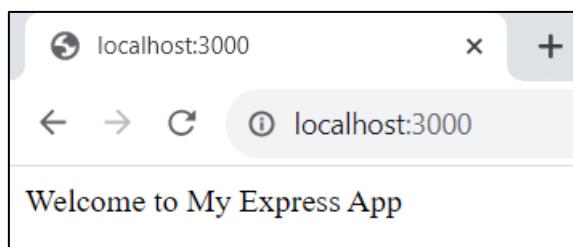


Figure 6.13: Browser Output at Port 3000

6. Switch to the Command Prompt window.

The message is displayed as shown in Figure 6.14.

```
C:\MyApps>node middlewareSample.js
My Express App is running at http://localhost:3000
This is a simple middleware that logs a message for request:1
```

Figure 6.14: Message Logged for First Request

7. Switch to the browser window and refresh the page.

8. Switch back to the Command Prompt window.

The second message is displayed as shown in Figure 6.15.

```
C:\MyApps>node middlewareSample.js
My Express App is running at http://localhost:3000
This is a simple middleware that logs a message for request:1
This is a simple middleware that logs a message for request:2
|
```

Figure 6.15: Message Logged for Second Request

With every refresh, the request number is incremented as shown in Figure 6.16.

```
C:\MyApps>node middlewareSample.js
My Express App is running at http://localhost:3000
This is a simple middleware that logs a message for request:1
This is a simple middleware that logs a message for request:2
This is a simple middleware that logs a message for request:3
This is a simple middleware that logs a message for request:4
This is a simple middleware that logs a message for request:5
|
```

Figure 6.16: Message Logged for Subsequent Requests

This shows how the middleware moves to the next request in the queue.

6.8 Summary

- Express.js is a framework that is used for server-side programming with Node.js runtime environment.
- Express.js can handle different types of HTTP requests, generate dynamic HTML, provide middleware functions, and serve static files.
- The Express.js framework is made up of three interconnected components—view, controller, and model—spread across three layers—application, service, and data access, respectively.
- The view component is responsible for generating the HTML content that is sent to the client.
- The model component handles the data and the business logic and interacts with the underlying data sources.
- The controller component provides the logic that acts as a bridge between the view and model components.
- The order in which components of Express.js are involved is server instance → router → middleware → handler functions → templating/static file serving.
- Express.js is a minimalist, extensible framework that is easy to use and adopt.
- It provides enhanced performance and supports several databases and templating engines.
- The `request` object handles the HTTP requests coming from the clients and provides information, such as the URL, HTTP method, HTTP header, request body, and client IP.
- The `response` object is responsible for generating the responses for the requests and sending them back to the client.
- Express.js provides the `app.get` function to define handlers for GET requests and the `app.post` function to define handlers for POST requests.

Test Your Knowledge

1. What is the purpose of middleware?
 - a) It is a bridge between databases
 - b) It is a single library
 - c) It is a tool that can only be used in React
 - d) It is used to handle HTTP requests and responses

2. You are working on a Web application. Your client wants you to add Express.js to your application. What is the correct sequence of components you will use in your application?
 - i. Middleware
 - ii. Router
 - iii. Server instance
 - iv. Handler functions
 - v. Templating/Static file serving
 - a) i, ii, iii, iv, v
 - b) iv, v, iii, ii, i
 - c) iii, ii, i, iv, v
 - d) iii, i, ii, iv, v

3. Consider the given code. Add the missing code to complete the application.

```
const var_express = require('express');
const var_app = express();
const port = 3000;

// Add missing code here
app.listen(port, () => {
  console.log(`My Express App is running at
http://localhost:${port}`);
});
```


 - a) app.get(req, res) => res.send('This is an example of express.');
 - b) app.get('/html', (req, res) => res.send('This is an example of express. '));
 - c) app.get('/', (req, res) => res.send('This is an example of express.'));
 - d) app.get('/', (res) => res.send('This is an example of express.'));

4. What is the purpose of `url` property in HTTP?

- a) It stores encoding data
- b) It stores route parameters
- c) It stores HTTP headers
- d) It stores the complete URL included in the request sent by the client

5. What will be the output of the given code:

```
const var_express = require('express');
const var_app = express();
const port = 3000;

app.get('/', (req, res) => res.send('Alan loves to play
cricket'));

app.listen(3000, () => {
  console.log(`Express App is running on port 3000.`);
});
```

- a) It will display Alan loves to play cricket
- b) It will display Express App is running on port 3000
- c) It will display an error
- d) None of these

Answers to Test Your Knowledge

1	d
2	c
3	c
4	d
5	a

Try It Yourself

1. Create a simple Web application for a company to display their Employee Information such as Employee Name, Employee ID, and Employee Designation using Express.js as backend.
 - a) Install Express.js and the body-parser Package.
 - b) Create a Web page as EmployeeInfo.html.
 - c) Create an app.js file.
 - d) Inside the app.js file create an Express Web server.
 - e) Create a body-parser instance.
 - f) Configure the Web server to listen on port 3000.
 - g) Serve the EmployeeInfo.html page using the app.get method.
 - h) Send the result based on the employee information using the app.post method.
 - i) Display the result on the Web page.
2. Create a simple Web server using Express and use a middleware in the code. Display the message, My first Web server working successfully, for each request.



SESSION 7

ASYNCHRONOUS AND SYNCHRONOUS PROGRAMMING

Learning Objectives

In this session, students will learn to:

- Compare asynchronous and synchronous programming models
- Explain asynchronous programming model using callback, Promise, and async/await in Node.js
- Describe synchronous programming model in Node.js
- Explain the event loop architecture in Node.js

Asynchronous and synchronous programming models help developers create robust and event-driven applications. Asynchronous applications do not depend on the sequential approach of execution. Multiple tasks can be executed simultaneously. Synchronous applications follow sequential execution of tasks, one at a time.

This session will provide an overview of asynchronous and synchronous programming models. It will compare the features of both these models. The session will cover in detail the asynchronous programming model using callback, Promise, and async/await in Node.js. The session will also elaborate on the concept of synchronous programming model in Node.js.

The session will conclude with a detailed explanation of the event loop architecture in Node.js.

7.1 Introduction to Asynchronous Programming

A Node.js program can have multiple tasks for execution. The execution of the tasks depends on the adopted programming model. There are two generic programming models, asynchronous and synchronous.

An asynchronous programming model follows the non-blocking architecture, which allows a program to initiate multiple tasks at the same time. Each task runs concurrently without waiting for the other task to be completed. The application output provided to the user is faster. The application remains responsive throughout the execution cycle.

Consider an example of an online form application based on an asynchronous programming model. Figure 7.1 shows the visual representation of the simultaneous task execution in the application.

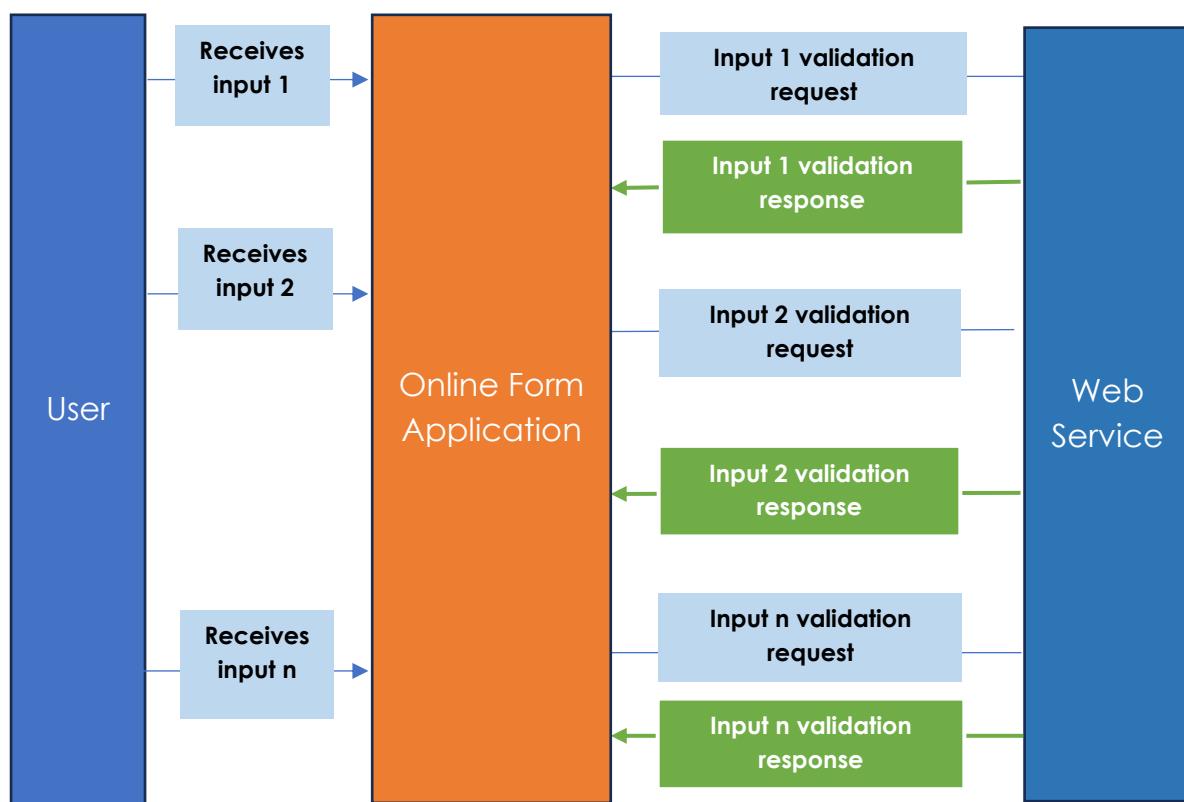
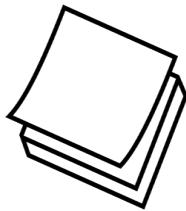


Figure 7.1: Asynchronous Task Execution

The form application continues to accept user inputs while it concurrently sends the received inputs to a Web service for validation. The Web service processes the validation request and immediately sends the response back to the application. This way, the application remains responsive throughout by continuously running the user interface in the foreground. Simultaneously, the application requests the Web service to perform the validation tasks in the background.

Asynchronous programming is used in both front-end and back-end applications. Different ways to implement asynchronous programming are by using the callback function, Promise object, await/async keywords, and so on.



Node.js is a single-threaded, asynchronous JavaScript runtime environment. The JavaScript code is executed in a single thread at a time. The asynchronous operations such as input request, network request, reading from a file, or writing to a file are executed simultaneously.

7.2 Introduction to Synchronous Programming

The synchronous programming model follows the blocking architecture that executes each task in a program in a sequential manner. The first task completes its execution and generates the output. The control then returns to the program for the execution of the second task. The second task is executed and the execution series goes on in a similar manner. If any one of the tasks consumes more time, the entire application may become unresponsive until the task is completed.

Consider the online form application example based on the synchronous programming model. Figure 7.2 shows the visual representation of the sequential task execution in the application.

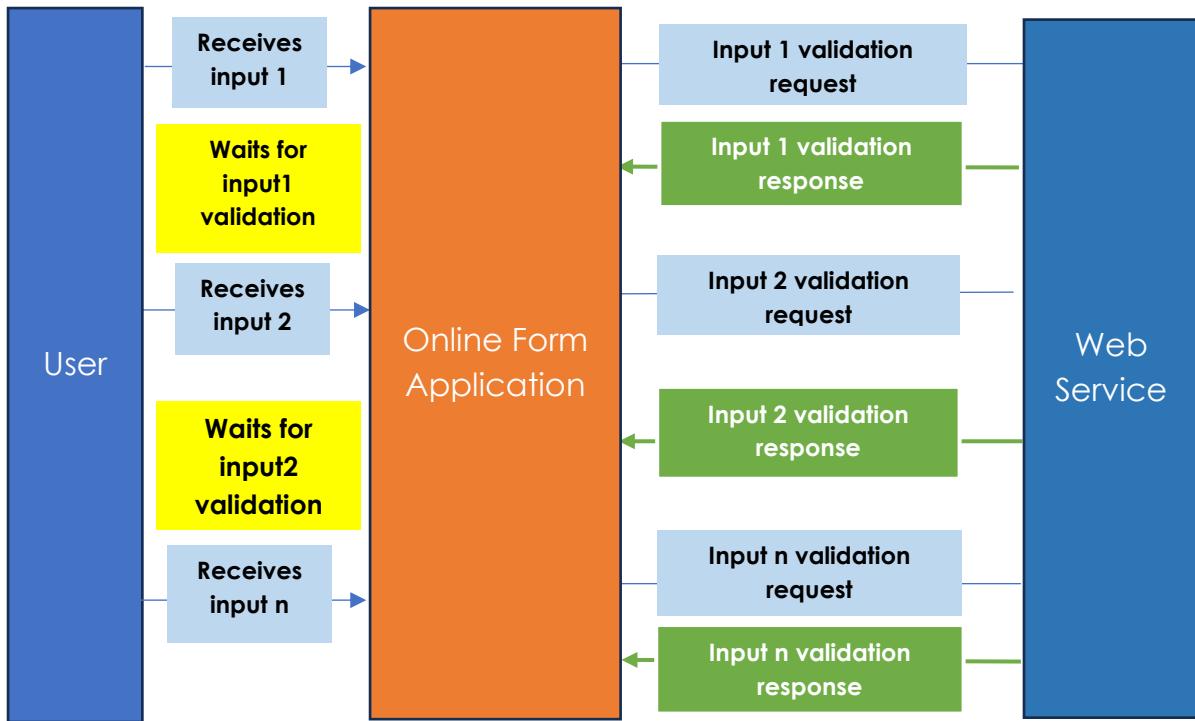
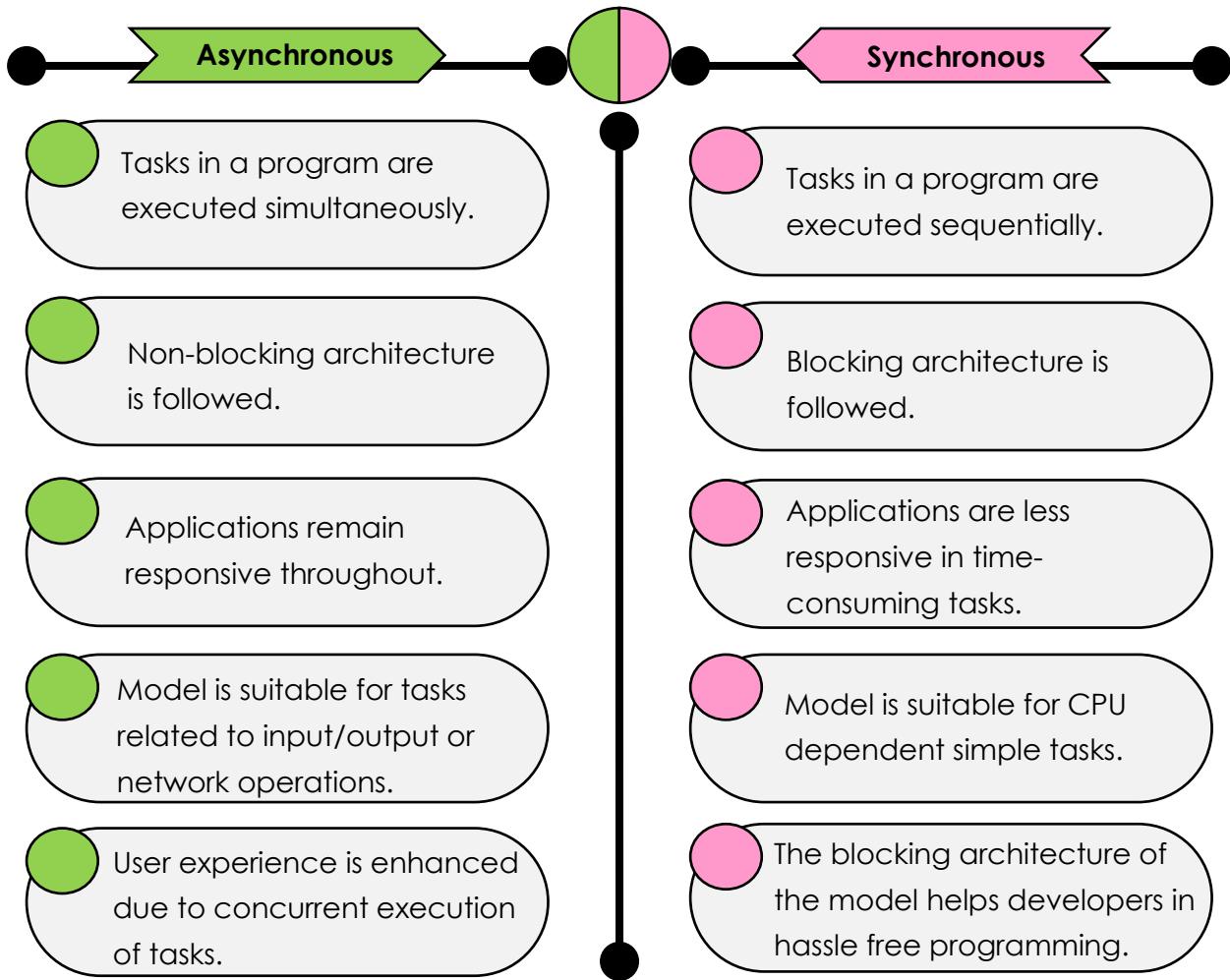


Figure 7.2: Synchronous Task Execution

The tasks in the form application are executed one at a time in a sequence. The application waits for the current input validation operation to complete before it moves on to receive the next input from the user. In this model, the order of executing the tasks is critical and important in the application.

7.3 Asynchronous vs. Synchronous Programming

As software developers, it is crucial to understand the difference between asynchronous and synchronous programming models.



7.4 Asynchronous Programming in Node.js

Node.js, preferably, follows the asynchronous programming model. Different ways to implement the asynchronous programming model in Node.js are as follows:

- Using the `callback` function
- With the `Promise` object
- With the help of `async` and `await` keywords

7.4.1 Asynchronous Programming in Node.js Using callback Function

A callback function is a function that is passed as an argument to another function. When the outer function finishes its task, it calls the callback function to handle the result of its operation. Thus, developers can use callback functions to implement asynchronous programming. Results of asynchronous operations can be handled without blocking the execution of the rest of the code.

Consider an example code in Code Snippet 1 where the text file reading operation is demonstrated using the callback function.

Perform the given steps:

1. Create a text file using VS Code or Notepad. Enter the contents in the sample text file:

```
Welcome to Node.js programming!
```

```
This session covers asynchronous and synchronous  
programming models.
```

2. Save the text file with the name `nodefile.txt` on the local system.
3. Type the code given in Code Snippet 1 in the editor.

Code Snippet 1:

```
const fs = require('fs');  
fs.readFile('nodefile.txt', 'utf8', (err, data) =>  
{  
    if (err) {  
        console.error(err);  
        return;  
    }  
    console.log(data);  
});  
console.log('Topics to be discussed in this session  
are:');  
console.log('1. Asynchronous Programming');  
console.log('2. Synchronous Programming');
```

The example code imports the `fs` module to perform the read operation on a text file. The `fs.readFile` function from the `fs` module reads the text file `nodefile.txt` asynchronously. The file reading operation takes place in the background while the remaining code gets executed simultaneously.

The last argument of the `fs.readFile` function is a callback function. The callback function is triggered when the background task of reading the file is completed. There are two arguments for the callback function.

The first argument `err` stores any potential error information if the task execution throws any error. The second argument `data` stores the contents of the text file. Once the `fs.readFile` function completes the

reading of the text file contents, the callback function displays the corresponding output on the console.

4. Save Code Snippet 1 with the file name `asyncprg.js`. Ensure to save the code in the same folder where the text file is saved.
5. Open the command prompt and navigate to the required folder.
6. To run the saved program, at the command prompt, type the command and press Enter:

```
node asyncprg.js
```

Figure 7.3 displays the output of Code Snippet 1.

```
Topics to be discussed in this session are:  
1. Asynchronous Programming  
2. Synchronous Programming  
Welcome to Node.js programming!  
  
This session covers asynchronous and synchronous programming models.
```

Figure 7.3: Output of Code Snippet 1

Observe the program output. The file reading operation is running in the background. Therefore, the rest of the code after the callback function block is executed and the messages provided in the `console.log` functions are displayed first. Node.js does not stop the execution of the `console.log` functions even though the file reading operation is happening in the background. After the file reading operation is completed, contents of the text file are displayed.

- **Working of the callback Function in Code Snippet 1**

The request to read the contents of the text file is submitted to the operating system. On receiving the request, the operating system starts processing and sends the response. There should be an entity that can handle the response sent by the operating system and that entity is the callback function. The callback function is capable of handling the operating system's response.

When the request-response operations are in progress, the main thread of Node.js continues to execute other instructions in the program. Therefore, the `console.log` functions are executed, and the outputs are displayed.

- **Operational Flow of Code Snippet 1**

Figure 7.4 shows the visual representation of the operational flow of Code Snippet 1.

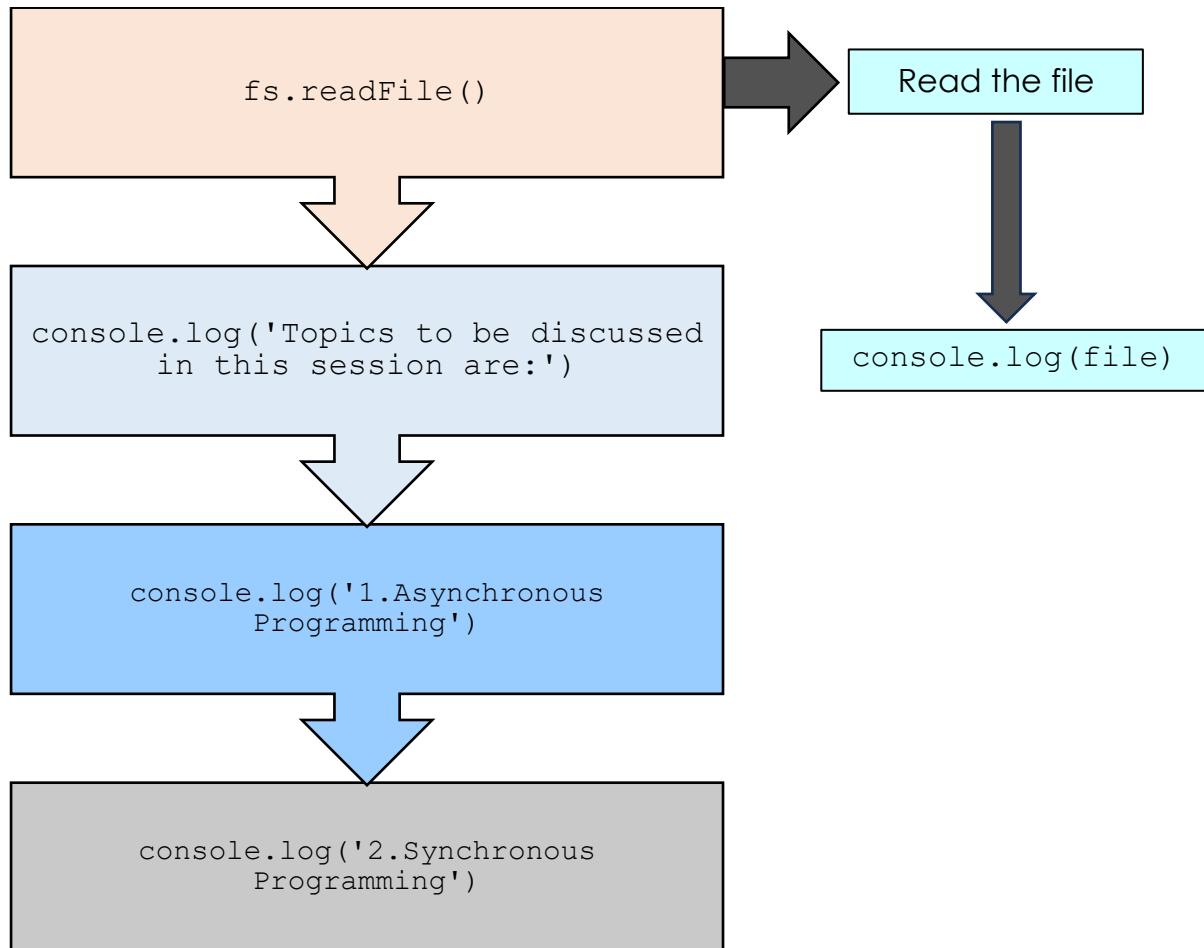


Figure 7.4: Operational Flow of Code in Code Snippet 1

7.4.2 Asynchronous Programming in Node.js Using Promise

An asynchronous Node.js program can have multiple or nested callback functions for simultaneous execution of tasks. This kind of arrangement, also known as a **callback hell**, in a program can lead to chaos and disorganization. A JavaScript object known as `Promise` solves the problem of disorganized callback functions in a program.

The object, `Promise`, handles the execution of multiple tasks in an organized manner in a program. A task can just be started, or already completed successfully, or rejected due to an error. Therefore, the `Promise` object has three states as shown in Figure 7.5.

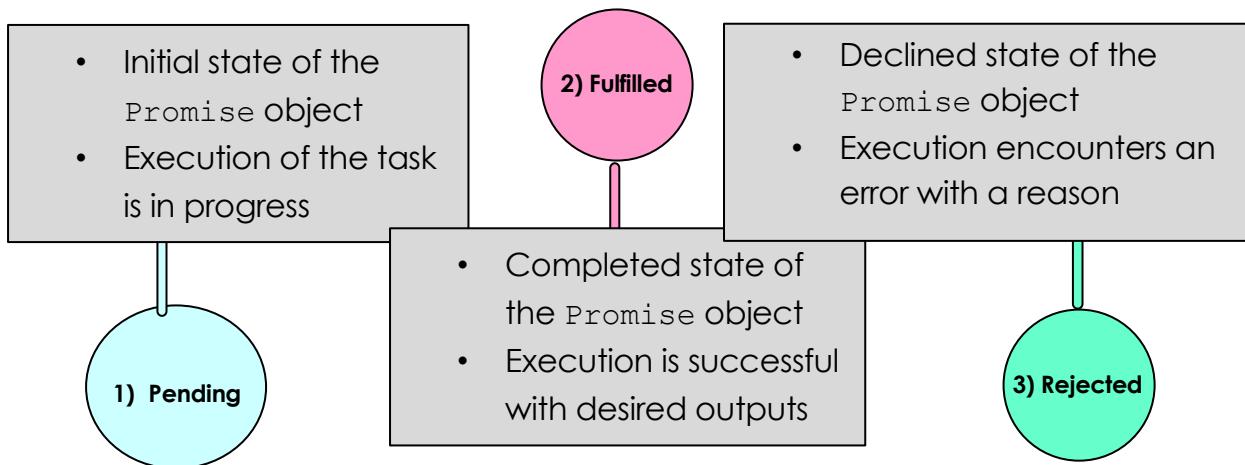


Figure 7.5: States of the Promise object

Consider an example code in Code Snippet 2 where the file reading operation is demonstrated using the `Promise` object.

Code Snippet 2:

```

const fs = require('fs');
const promise = new Promise(function (resolve, reject) {
  fs.readFile('nodefile.txt', 'utf8', (err, data) => {
    if (err) {
      return reject (err)
    }
    resolve(data)
  })
})
promise
.then ((data) => console.log(data))
.catch((err) => console.log(err))

```

In Code Snippet 2:

- The `fs` module is used to handle the file read operation. The `Promise` constructor is used to create a new `Promise` object.
- The first argument to the constructor is a callback function. This callback function is called the `executor` function, which is to be executed by the constructor. The `executor` function accepts two functions as parameters.
- The first parameter is the `resolve` function, which creates the `Promise` object in a fulfilled or completed state. The second parameter is the `reject` function, which declines the creation of the `Promise` object with an error definition.
- The `fs.readFile` function reads the text in the `nodefile.txt` file asynchronously. The file reading operation takes place in the background while the remaining program tasks are being executed. If the

`nodefile.txt` file is available, the `Promise` object creation is successful, and the object moves to the resolved state. The contents of the text file is transmitted to the `.then` function and the output is displayed on the console.

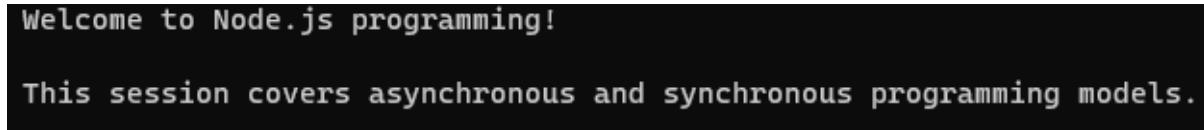
- If the `nodefile.txt` file is not available, the `Promise` object creation is unsuccessful and the object moves to the rejected state. An error is generated and handled by the `.catch` function and the error information is displayed on the console.

Save Code Snippet 2 with the file name, `asyncpromise.js`. Use the `nodefile.txt` file created earlier.

Type the command and press Enter:

```
node asyncpromise.js
```

Figure 7.6 displays the output of Code Snippet 2.



```
Welcome to Node.js programming!  
This session covers asynchronous and synchronous programming models.
```

Figure 7.6: Output of Code Snippet 2

Observe the program output. Since the text file is available, the program has not generated an error and the contents of the text file are displayed.

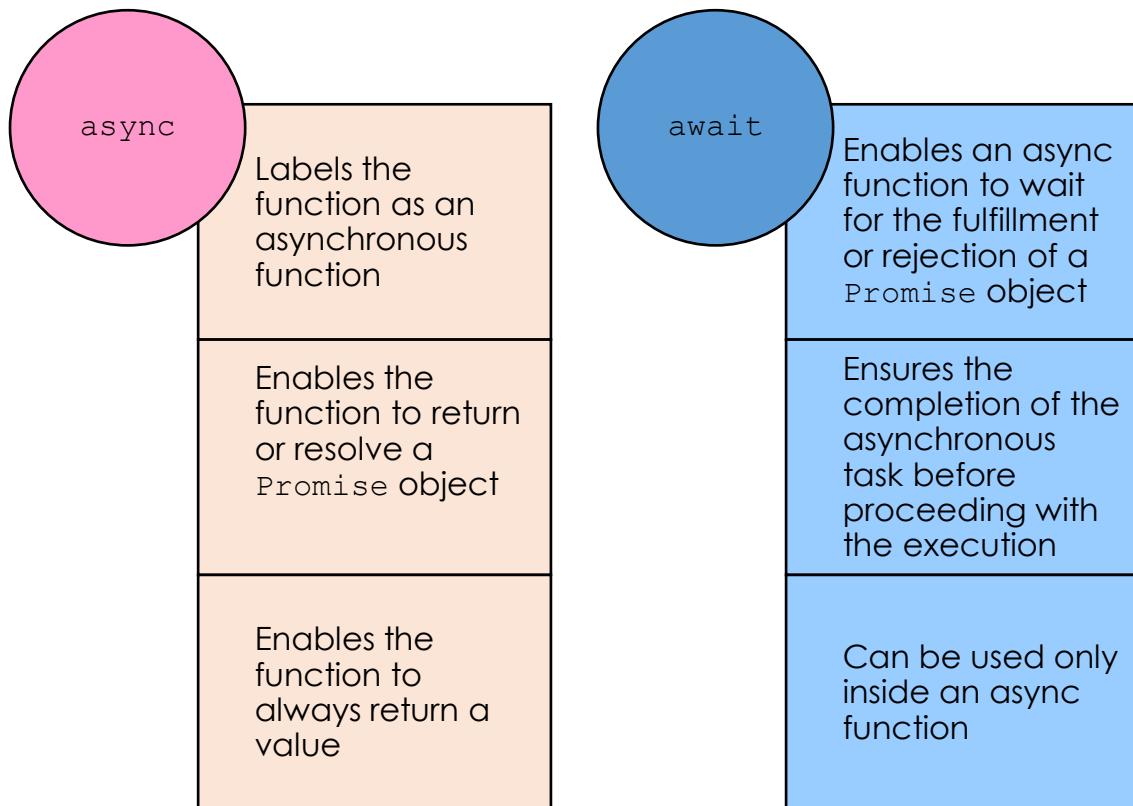
Delete the text file from the folder and run the program again. The program will throw a runtime error displaying the details of the error.

To understand the purpose of the promise object, consider the example of a restaurant. The chefs in the kitchen of the restaurant receive orders continuously for various dishes from the waiters. Here, the preparation of a dish is an asynchronous task. A waiter is a promise object who takes orders from the customers and passes them on to the chefs. Chefs are the asynchronous processors who execute the asynchronous task of preparing the dish thereby fulfilling the promises. Why is this whole process asynchronous? Multiple orders will be received and processed by multiple chefs simultaneously without waiting for the completion of any one particular task. In case a certain dish is not available, the waiter obtains an updated order from the customer. Thus, a promise is either fulfilled or rejected to create another promise.

To conclude, the `Promise` object handles both successful and unsuccessful asynchronous operations in a program.

7.4.3 Asynchronous Programming in Node.js Using `async/await`

The keywords `async` and `await` act as an added advantage for callback function and `Promise` object in Node.js. These keywords make an asynchronous program appear similar to a synchronous program thereby facilitating clarity in the sequence of events.



Consider the example code in Code Snippet 3 where the file reading operation is demonstrated using the `async` and `await` keywords.

Code Snippet 3:

```
const fs = require('fs');
const filepromise = function () {
  const promise = new Promise(function (resolve, reject) {
    fs.readFile('nodefile.txt', 'utf8', (err, data) => {
      if (err) {
        return reject (err)
      }
      resolve(data)
    })
  });
  return promise;
};
async function readFileAsync() {
  try {
    const data = await filepromise();
```

```
        console.log('File content:');
        console.log(data);
    } catch (error) {
        console.error('Error:', error.message);
    }
}
readFileAsync();
```

In the example code, the `fs` module is used to handle the file read operation. The `Promise` constructor is used to create a new `Promise` object.

An asynchronous function with the name `readFileAsync` is declared using the `async` keyword. The `await` keyword is used inside the `async` function. The keyword pauses the execution of the `async` function until the fulfillment of the `Promise` object. The function `filepromise` returns a `Promise` object.

If the program is able to read the text file successfully, the `Promise` object is resolved, and the contents of the text file are displayed as the output. The `try` block handles the successful execution of the function. If the text file is not available, an error is generated at runtime, and the `catch` block handles the error.

Either the `try` block or the `catch` block is executed depending on the fulfillment or rejection of the `Promise` object. The code initiates the file read operation by invoking the `readFileAsync` function.

Save Code Snippet 3 with the file name, `fileawait.js`. Type the command and press Enter:

```
node fileawait.js
```

Figure 7.7 displays the output of Code Snippet 3.

```
File content:
Welcome to Node.js programming!

This session covers asynchronous and synchronous programming models.
```

Figure 7.7: Output of Code Snippet 3

Observe the program output. The `Promise` object is successfully resolved, the `try` block is executed, and the text file contents are displayed on the console.

7.5 Synchronous Programming in Node.js

A synchronous Node.js program executes the instructions in a sequential way. This approach can lead to blockade of critical instructions and make the program unresponsive.

Consider an example code in Code Snippet 4 where the text file reading operation is demonstrated using the synchronous programming model.

Code Snippet 4:

```
const fs = require('fs');
const data = fs.readFileSync('nodefile.txt', 'utf8');
console.log(data);
console.log('Topics to be discussed in this session are:');
console.log('1.Aynchronous Programming');
console.log('2.Synchronous Programming');
```

In the example code, the `fs` module is used to perform the file read operation. The synchronous function `fs.readFileSync` reads the contents of the text file, `nodefile.txt`, which is passed as the function argument. The program is executed sequentially. Therefore, the `console.log` statements are executed only after the function `fs.readFileSync` completes the file reading operation.

Save Code Snippet 4 with the file name `syncprg.js`. Type the command and press Enter:

```
node syncprg.js
```

Figure 7.8 displays the output of Code Snippet 4.

```
Welcome to Node.js programming!

This session covers asynchronous and synchronous programming models.

Topics to be discussed in this session are:
1.Aynchronous Programming
2.Synchronous Programming
```

Figure 7.8: Output of Code Snippet 4

Observe the program output. The contents of the text file are displayed first and then the messages provided in the `console.log` functions are displayed.

Operational Flow of Code Snippet 4

Figure 7.9 shows the visual representation of the operational flow of Code Snippet 4.

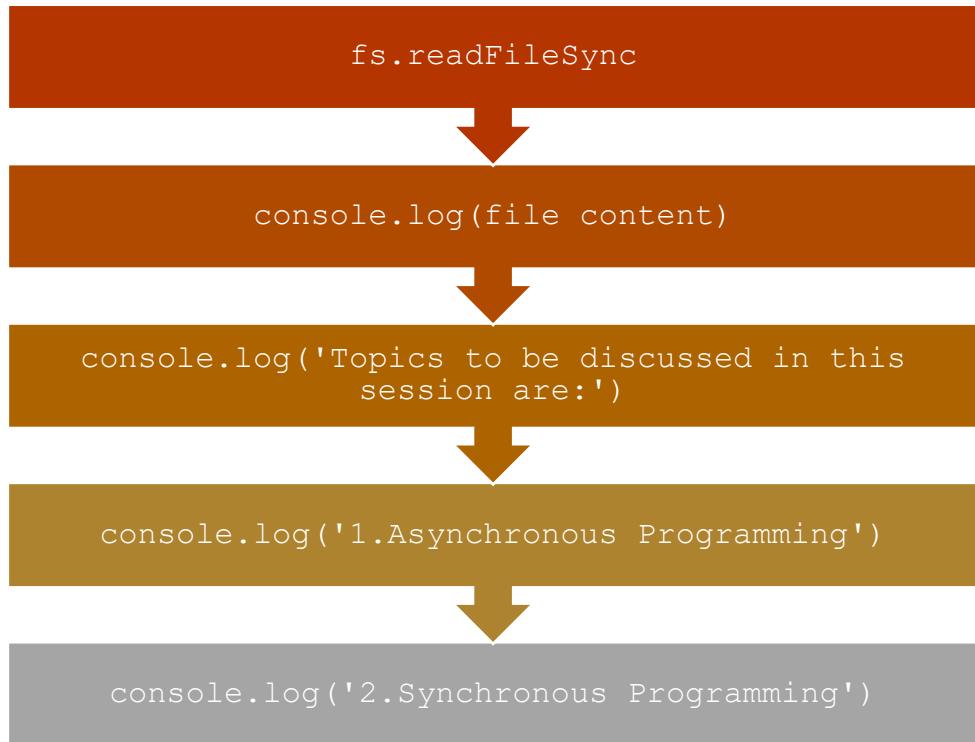


Figure 7.9: Operational Flow of Code in Code Snippet 4

7.6 Introduction to Event Loop in Node.js

The asynchronous programming model of Node.js is also known as the Input/Output (I/O) non-blocking model. An event loop manages the asynchronous and non-blocking operations in Node.js. The event loop keeps on running continuously until there are asynchronous operations in the program.

7.6.1 Working of an Event Loop

Figure 7.10 shows a diagrammatic representation of the working of an event loop on the Node.js server.

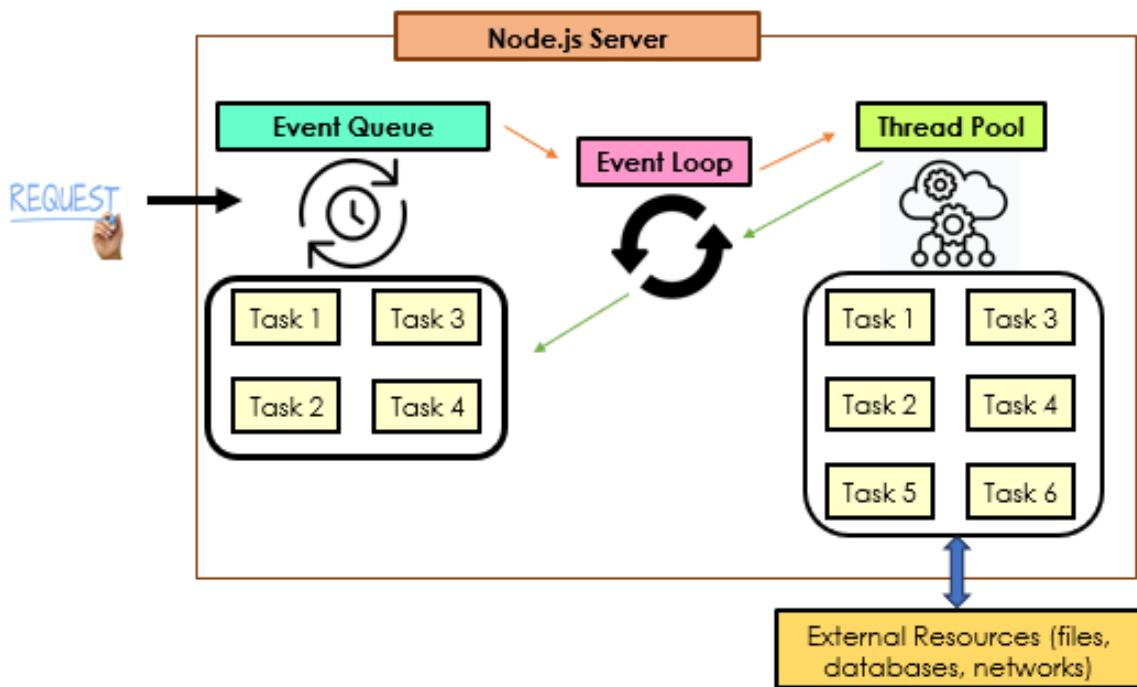
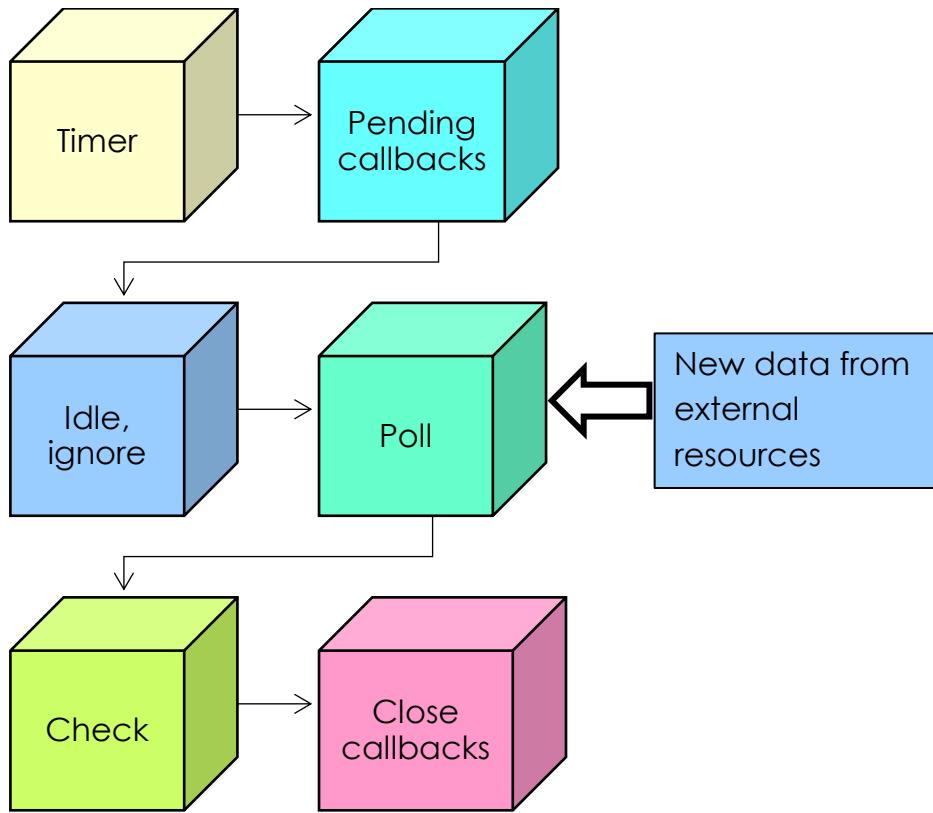


Figure 7.10: Working of Event Loop

The components of Node.js server that are responsible for the looping the events are as follows:

- **Event queue:** This component is also known as the callback queue or message queue. Event queue is a data structure that stores asynchronous tasks or callbacks. The event loop picks up the task from the event queue for execution.
- **Event loop:** This loop is the most important component of Node.js server. The loop keeps on checking the event queue for any tasks to execute. The event loop consists of six phases. Each phase has a set of tasks to perform.

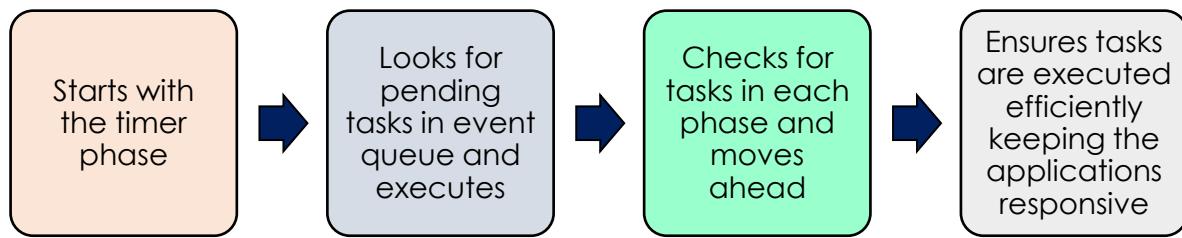


Event loop phases are as follows:

- **Timer:** This phase checks for any timers that have attained their specified duration. The timer module is a part of the Node.js framework. The timer module executes callback functions according to the specified intervals. There are two global functions, such as `setTimeout` and `setInterval`, in the timer module. The `setTimeout` function facilitates the setting up of a timer that can execute a specific task upon the expiry of the timer. The `setInterval` function helps to execute the tasks repeatedly, after a specific interval.
- **Pending callbacks:** In this phase, the pending callback functions, related to input/output, file handling, and so on, are executed from the event queue.
- **Idle/ignore:** This phase is usually dedicated to the not-so-important tasks in the event loop. Internal purpose tasks or background tasks such as garbage collection are performed in the idle phase.
- **Poll:** This phase is entrusted with the task of checking new input/output events and processes.
- **Check:** This phase starts after the poll phase becomes idle. The `setImmediate` function, a timer, executes the callback functions in the check phase.

- **Close callbacks:** This phase executes callback functions that are primarily involved in the clean-up tasks such as closing the database connections or terminating the server connections.
- **Thread pool:** The thread pool is in charge of executing the asynchronous operations. The Node.js server informs the thread pool of an asynchronous task. The asynchronous task runs as a separate thread in the thread pool. Concurrently, the main operations run as a separate thread.

To conclude, the event loop:



7.6.2 Event Loop Example

Consider an example code for an event loop in Code Snippet 5.

Code Snippet 5:

```

console.log('Timer begins');
setTimeout(() => {
  console.log('Timer 1: Executed after 2 seconds');
}, 2000);
setTimeout(() => {
  console.log('Timer 2: Executed after 1 second');
}, 1000);
console.log('Timer Ends');
  
```

In the example code, two timers are set using the `setTimeout` function. The first timer, Timer 1, will execute after two seconds. The second timer, Timer 2, will execute after one second. The tasks are executed asynchronously. Therefore, timers run in the background and are added to the event queue when they are ready for execution. The event loop checks the event queue and picks up the callback functions for Timer 1 and Timer 2, respectively.

Save Code Snippet 5 with the file name, eventlooppgr.js. Type the command and press Enter:

```
node eventlooppgr.js
```

Figure 7.11 displays the output of Code Snippet 5.

```
Timer begins
Timer Ends
Timer 2: Executed after 1 second
Timer 1: Executed after 2 seconds
```

Figure 7.11: Output of Code Snippet 5

Observe the program output. The first `console.log` function is executed. Meanwhile, the callback functions are running in the background. Therefore, Node.js continues to execute the second `console.log` function. It then, executes Timer 2 followed by Timer 1.

7.7 Summary

- In an asynchronous programming model, tasks are executed simultaneously.
- The synchronous programming model follows the sequential execution of tasks.
- Node.js relatively follows the asynchronous programming model with the help of a callback function, Promise object, and `async/await` keywords.
- The callback function executes the asynchronous tasks in the background while other tasks are executed without any blockade.
- The `Promise` object acts as a complementary addition to the callback function.
- The `async` keyword ensures that the `async` function resolves a `Promise` object.
- The `await` keyword should be used inside an `async` function only.
- The event loop architecture in Node.js consists of six phases such as timer, pending callbacks, idle, ignore, poll, check, and close callbacks.
- An event loop maintains the responsiveness of the applications ensuring efficient execution of asynchronous tasks.

Test Your Knowledge

1. Which approach do the applications using synchronous programming follow?
 - a) Bottom-up Approach
 - b) Top-down Approach
 - c) Routing Approach
 - d) Sequential Approach
2. In asynchronous programming, how is a program executed?
 - a) Sequentially
 - b) Concurrently
 - c) Multiple tasks executing at once
 - d) None of the these
3. Which of the following functions can be used to perform asynchronous programming in Node.js applications?
 - a) reverse Function
 - b) callback Function
 - c) addition Function
 - d) Promise Function
4. Which of the following is not a state of the `Promise` object?
 - a) Fulfilled
 - b) Rejected
 - c) Pending
 - d) Stop

5. Add the missing line in the given code:

```
const fs = require('fs');
// Add missing line here
if (err) {
  console.error(err);
  return;
}
console.log(data);
});
console.log('You have started working on asynchronous
programming.');

a) fs.readFile('program.txt', 'utf8', (err, data) => {
b) fs.readFile('program.txt', 'utf11', (err) =>
c) fs.readFile('program.txt', 'utf8', (data) =>
d) fs.readFile('program.txt', 'utf8' =>
```

Answers to Test Your Knowledge

1	b
2	a
3	b, d
4	d
5	a

Try It Yourself

1. Create a Node.js application to read a text file.
 - a) Create a text file using Notepad. Enter the given content in the read Text file:
 - i. Now, you have started working on a Node.js application.
 - ii. You have learned how to perform asynchronous and synchronous programming.
 - b) Save the text file with the name, `readTextFile.txt` in a particular folder on the local system.
 - c) Save the code with the file name `program.js`.
 - d) Display the result in the console.
2. Create a Node.js application to read a file using the `async` and `await` functions.
 - a) Create a text file using Notepad. Enter the given content in the text file:
 - i. Mangos are Yellow.
 - ii. Strawberries are Red.
 - iii. Raspberries are Red.
 - b) Save the text file as `fruit.txt` on the local system.
 - c) Define a function that will return `Promise`.
 - d) Create an asynchronous function for reading file.
 - e) Save the code as `programAsycn.js`.
 - f) Display the result in the console.
3. Create a simple program and display the given statement in the console using the timer module. Display Statement 1 after 3 seconds and Statement 2 after 4 seconds.

Statement 1: Fruits are beneficial for your health.
Statement 2: Fruits are low in calories.



SESSION 8

RESTFUL AND HTTP APIs

Learning Objectives

In this session, students will learn to:

- Describe an API and its purpose
- List the common types of APIs in Node.js and their functions
- Explain RESTful and HTTP APIs and their features
- Explain how to use RESTful API and HTTP API to manage data
- Identify the advantages and disadvantages of RESTful and HTTP APIs
- Distinguish between RESTful APIs and HTTP APIs

Application Programming Interfaces (APIs) translate a user's instructions into a format, which can be interpreted by the system to process the request and send back the response.

For example, an e-commerce application (say, App1) provides an option to pay through credit cards. To ensure the smooth transaction of the payment, App1 must talk to the credit card company's application (say, App2) that handles the authentication and processes payments made using credit cards. Each credit card company will have its payment processing application. For talking to App2, APP1 must use an API. The credit card company usually provides this API. This API will act as a middleman between App1 and App2 to

complete the transaction for the user.

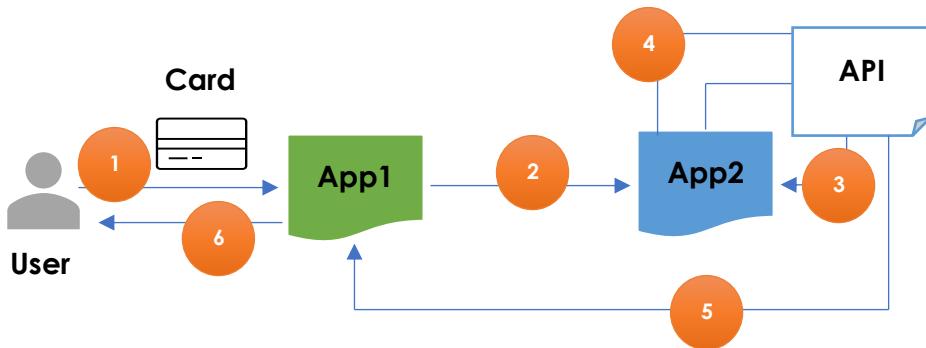


Figure 8.1: Payment Processing System

Figure 8.1 shows workflow of the payment processing system. In this case:

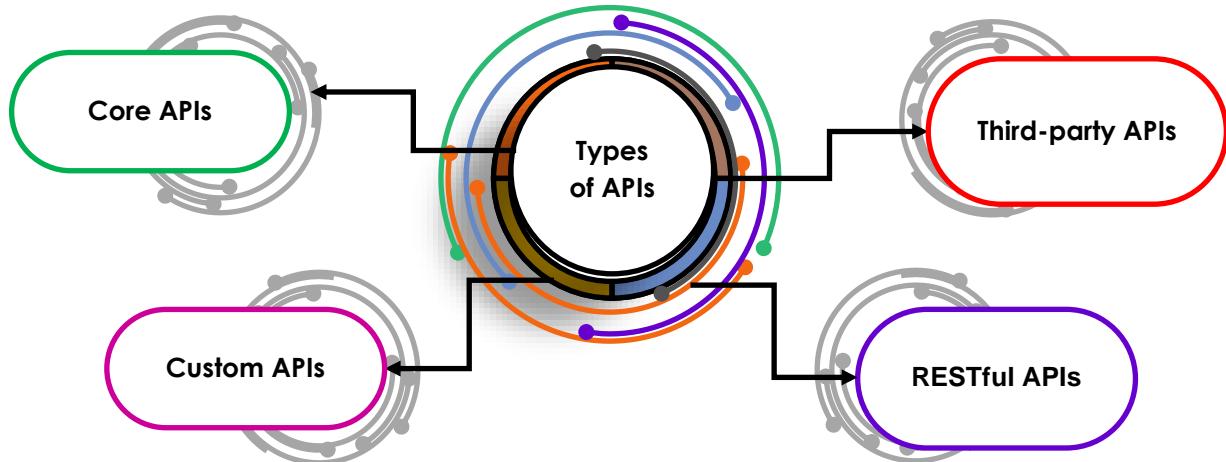
1. The user enters the card details on the payment Web page.
2. App1 sends the data to App2.
3. At App2, the details are received by the API, which, in turn, translates the data into a format that is understood by App2.
4. After the payment is processed, the API takes the status from App2 and translates it into a language that App1 understands.
5. App2 then, sends the payment status to App1.
6. App1 then, displays the status of the payment on the Web page for the user.

This session introduces common types of APIs in Node.js and their applications. It also lists features of Representational State Transferful (RESTful) API and HyperText Transfer Protocol (HTTP) API and their benefits and drawbacks. Finally, it explains applications of RESTful and HTTP APIs and differentiating features between RESTful and HTTP APIs.

8.1 Introduction to APIs

As already discussed, an API facilitates communication and interaction between various software applications through a set of rules and protocols. The main objective of an API is to organize the requests and their responses to enable the developers to extract data from other software applications.

Depending upon scope and usage, the APIs supported by Node.js are as follows:



- **Core APIs:** These APIs are provided by Node.js and are available as built-in modules, such as a filesystem module or HTTP module. The core APIs provide basic foundation blocks of an application. For example, consider that a user wants to build an e-commerce Website. Here, core APIs will provide basic functionalities that allow customers to browse the product catalog, handle shopping carts, manage product inventory, manage orders, and manage payments.
- **Third-Party APIs:** As the name suggests, these APIs are provided by other developers or companies that the users can integrate into their application to provide specialized features. These APIs are installed using Node Package Manager (`npm`). Some examples of third-party APIs include mongoose for connecting to MongoDB databases, Express.js to use Web frameworks, and Passport.js to handle authentication. In the e-commerce application, users can use third-party APIs to provide the product review feature. Customers can use this feature to provide product reviews and view reviews added by other customers.
- **Custom APIs:** These APIs are used to develop new functionality required by an application. The users can use the `module.exports` object to create functionality and make it available to other parts of the application. In an e-commerce application, custom APIs can be used to provide customers with personalized recommendations based on their browsing and purchasing history.
- **RESTful APIs:** As the name suggests, these APIs work on the principles of the Representational State Transfer (REST) architecture. They are type of HTTP APIs that use HTTP requests to access and manage data. These APIs define how data is shared between different parts of an application. In the e-commerce application, RESTful APIs can be used to ensure that data is transmitted in a standardized format such as JSON or XML.

The developer can choose a specific API based on the requirements of an application.

8.2 HTTP APIs

Node.js supports the HTTP API that uses HTTP protocol to facilitate communication and exchange of data between systems. These APIs use HTTP methods, such as GET, POST, PUT, and DELETE, for sending requests to the server. They use common formats, such as JSON and XML, to send back responses to the client.

Some of the features of HTTP APIs are as follows:

Client-server Model	They maintain clear distinction between client and server. In this model, the client is responsible for requesting actions or data and the server is responsible for processing and responding to the requests.
Layered Architecture	They allow multiple intermediary systems to participate in the communications between the client and server. This architecture offers flexibility by supporting distribution of processing responsibilities among various systems.
Caching	They allow intermediate systems, such as Web server and proxy servers, to cache responses for frequently requested data. This feature helps improve the performance of the application by reducing the load on the server.
Uniform Interface	They use a consistent interface based on the common HTTP methods and standard Uniform Resource Identifiers (URI) to work with data. This uniform interface makes it easy for developers to develop and maintain RESTful APIs applications.
Standard Media Types	They use standard media types, such as JSON and XML, for exchange of data between client and server. This feature makes HTTP API applications to be interoperable and compatible across diverse platforms.
Resource-centric Approach	They treat data as resources. These resources are identified by URIs and can be manipulated using HTTP methods. This feature makes the API structure easy to understand and implement.
Stateless Interactions	In these APIs, the server does not maintain any information related to client-specific state. This means that each request contains all the information required to understand and process the request, without any dependency on previous or next request.

Common methods used by HTTP APIs are as follows:

GET	POST	PUT	DELETE	PATCH
Used to send a request to retrieve data from the server	Used to send a request with the required information to the server to create a new resource	Used to send a request with the required information to the server to update an existing resource	Used to send a request to the server to remove a resource	Used to send a request to the server to partially update a resource

8.3 Advantages and Disadvantages of HTTP APIs

Table 8.1 lists some of the advantages and disadvantages of HTTP APIs.

Advantages		Disadvantages	
Simplicity	These APIs are easy to understand, design, implement, and maintain. This is because they use standard HTTP methods and common data formats.	Complexity	These APIs are not suitable for working with complex queries or data aggregations that require processing multiple requests simultaneously.
Scalability	These APIs are scalable owing to their stateless nature and resource-oriented architecture. This feature allows these APIs to adapt to changing requirements without significant architectural changes.	Performance Limitation	These APIs work on the request-response cycle and use HTTP methods that are less expressive. This feature puts a limit on the functionality and complexity of applications. Therefore, these APIs are not suitable for applications that require real-time streaming or high-frequency transactions.

Advantages		Disadvantages	
Efficiency	These APIs provide efficient performance because responses to frequent requests can be cached, thereby reducing the load on the server, and providing quick responses to such requests.	Security Concerns-	These APIs depend on HTTP protocol for authentication and data encryption. However, these services provided by HTTP are weak and prone to attacks and breaches.
Flexibility	These APIs can easily adapt to varying requirements. They also support multiple platforms and languages.	Limited Handling	These APIs use HTTP status codes to convey errors. These codes do not provide enough information that can be used for identifying errors and troubleshooting them. So, developers are required to implement additional mechanisms for debugging.
Portability	These APIs are platform-independent because they use standard HTTP protocol and common data formats.	Limited Standardization	These APIs do not have any official specifications or validations. This can lead to inconsistencies and ambiguities among different implementations.

Table 8.1: Advantages and Disadvantages of HTTP APIs

8.4 RESTful APIs

A RESTful API is a Web API that uses HTTP protocol to exchange data between applications. It is a subset of HTTP API that follows the principles and limitations of the REST architecture. It is used to design networked applications, especially Web services. Similar to HTTP APIs, RESTful APIs use HTTP methods, such as GET, POST, PUT, and DELETE, to perform operations on the resources.

RESTful APIs provide the same features, advantages, and disadvantages as HTTP APIs.

8.5 RESTful APIs vs. HTTP APIs

The RESTful and HTTP APIs have many common features, advantages, and disadvantages. However, both differ from each other in certain aspects, such as design principles and standards,

Table 8.2 lists the differences between these two APIs.

	RESTful API	HTTP API
Architectural styles	RESTful APIs follow specific architectural limitations and design principles of REST, which include resource-based strategy, standardized HTTP request methods, stateless communication between client and server, and uniform interfaces.	HTTP APIs do not follow the REST principles of design. It is flexible to suit the requirements of the applications, define the endpoints, and use the HTTP methods without many restrictions.
URL patterns	RESTful APIs are resource-based, where each URL represents a resource or collection of resources. The URL includes a resource identifier (ID) to specify the resource that must be accessed or manipulated.	HTTP APIs are more flexible, where the endpoints are not strictly resource-based and include resource identifiers that can be structured depending on the design of the API.
Design standardization	All RESTful APIs have standardized design patterns and fixed architectural styles.	HTTP APIs do not have the same level of standardization and show considerable variation in design patterns and architectural styles.
Manageable Granularity	RESTful APIs enforce granularity in code making it easy for developers to develop and maintain applications.	HTTP APIs do not enforce granularity in code, making it difficult

	RESTful API	HTTP API
Versioning	RESTful APIs enforce versioning.	to develop complex applications. HTTP APIs do not enforce versioning.
Hypermedia As The Engine Of Application State (HATEOAS)	In RESTful APIs, HATEOAS is a design principle. This principle emphasizes on the use of hypermedia links in the response to guide clients through the API and discover available resources and actions.	In HTTP APIs, HATEOAS is an optional element that is implemented in specific applications as required.

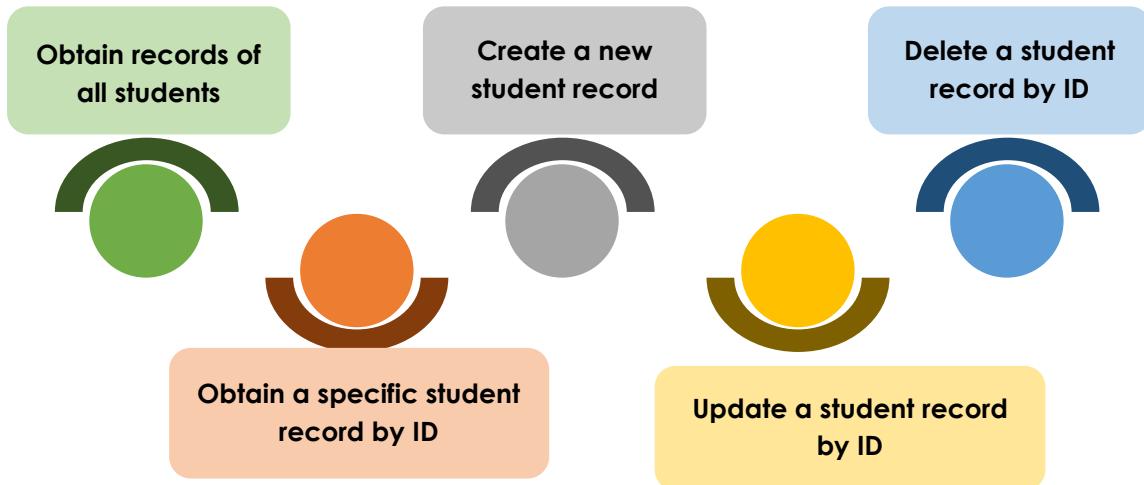
Table 8.2: Differences Between RESTful and HTTP APIs

8.6 Using HTTP APIs with Express.js

Express.js is an open-source Node.js Web application framework to create Websites and APIs. This framework handles different HTTP requests depending on the requirements.

Consider that a user wants to create a simple HTTP API to manage the data of students in a Node.js application using the Express.js framework.

This application will allow users to perform following tasks:



To run the application, the user wants to use cURL as the client to interact with the server. cURL is a tool that can be used to send requests to the server and receive responses from the server. To use cURL, users must have a command-line interface, in this case, Node.js, and an Internet connection.

To create this application, perform following steps:

1. Ensure that Node.js and Express.js are installed.
2. Open VS Code or a Notepad file.

3. To import the Express.js framework, create an instance of the Express.js application, and specify the port to be used for the Web application. Then, in the file, type the code given in Code Snippet 1.

Code Snippet 1:

```
const express = require('express');
const app = express();
const port = 3000;

app.use(express.json());
```

In Code Snippet 1, `require` function is used to import Express.js framework. Then, an instance of the Express application is created and stored in the variable `app`. Finally, the port to be used for this application is specified as 3000.

4. To add some sample student data, in the same file, append the code given in Code Snippet 2.

Code Snippet 2:

```
// Sample data for students
let students = [
  { id: 1, name: 'David Miller', age: 12, grade: 'A' },
  { id: 2, name: 'Adam Smith', age: 13, grade: 'B' },
  { id: 3, name: 'John Willams', age: 12, grade: 'A' },
];
let nextStudentId = 4;
```

In Code Snippet 2, an array named `students` is created to store sample data of the students. Every student represents an object configured with `ID`, `name`, `age`, and `grade` properties. The variable `nextStudentId` initializes a value to assign unique IDs to new students. In this code, this variable is initialized to 4 as the first three students are already in the array.

5. To view details of all the students, in the same file, append the code given in Code Snippet 3.

Code Snippet 3:

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

The code in Code Snippet 3 sends student details in JSON format when the endpoint of the request is /students.

6. To view the details of a student with a specific ID, in the same file, append the code given in Code Snippet 4.

Code Snippet 4:

```
// Get a specific student by ID
app.get('/students/:id', (req, res) => {
  const id = parseInt(req.params.id);
  const student = students.find((s) => s.id === id);
  if (student) {
    res.json(student);
  } else {
    res.status(404).json({ message: 'Student not found' });
  }
});
```

Code Snippet 4 stores the ID received from the client in the `id` variable. It then searches for the student with the ID stored in the `id` variable and stores the student details found in the `student` variable. Finally, it uses an `if-else` block to return the student details if the student exists or return an error message if the student does not exist.

7. To add details of a new student, in the same file, append the code given in Code Snippet 5.

Code Snippet 5:

```
// Create a new student
app.post('/students', (req, res) => {
  const { name, age, grade } = req.body;
  const student = { id: nextStudentId++, name, age, grade };
  students.push(student);
  res.status(201).json(student);
});
```

Code Snippet 5, first parses and stores the student details received from the client in an array. It then creates a student using the details in the array and the ID stored in the `nextStudentId` variable and then, increments this variable. It finally sends the new student details to the client in JSON format.

8. To update the details of a student with a specific ID, in the same file, append the code given in Code Snippet 6.

Code Snippet 6:

```
// Update a student by ID

app.put('/students/:id', (req, res) => {
  const id = parseInt(req.params.id);
  const student = students.find((s) => s.id === id);

  if (!student)
  {
    res.status(404).json({ message: 'Student not found' });
  }
  else
  {
    const { name, age, grade } = req.body;
    student.name = name;
    student.age = age;
    student.grade = grade;
    res.json(student);
  }
});
```

Code Snippet 6 parses the ID received from the client and stores it in the `id` variable. It then uses the ID in the `id` variable to search for the student. It stores the details of the student found in the variable `student`. It then checks if the `student` variable exists. If it does not exist, an error message is sent back to the client. If the `student` variable exists, then the student details are replaced with the details received in the client request. These updated student details are then sent back to the client in JSON format.

9. To delete the details of a student with a specific ID, in the same file, append the code given in Code Snippet 7.

Code Snippet 7:

```
// Delete a student by ID

app.delete('/students/:id', (req, res) => {
  const id = parseInt(req.params.id);
  const index = students.findIndex((s) => s.id === id);
  if (index !== -1) {
    students.splice(index, 1);
    res.json({ message: 'Student deleted' });
  } else {
    res.status(404).json({ message: 'Student not found' });
  }
});
```

Code Snippet 7 parses the ID received from the client and stores it in the `id` variable. It then, uses the ID in the `id` variable to search for the index of the student. It stores the index of the student found in the variable `index`. It then, checks if the `index` variable has a value that is not equal to `-1`. If the condition is met, the `splice` method is used to remove the student details at the specified index. The updated student details are then, sent back to the client in JSON format. If the condition is not met, an error message is sent back to the client.

- To start the server and configure it to listen to port 3000, in the same file, append the code given in Code Snippet 8.

Code Snippet 8:

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

- Save the file as `studentHTTPapi.js` in the same folder where Express.js is installed.

Table 8.3 lists the standard HTTP methods and the endpoints defined in the code.

Operation	HTTP Method	Endpoint
To retrieve all student records	GET	app.get('/students', (req, res)
To retrieve a student record based on the ID	GET	app.get('/students/:id', (req, res)
To add a new student record in the database	POST	app.post('/students', (req, res)
To update the details of a student based on the ID	PUT	app.put('/students/:id', (req, res)
To delete a student record based on the ID	DELETE	app.delete('/students/:id', (req, res)

Table 8.3: HTTP Methods and Endpoints Defined for Application

- To start the server, open a Command Prompt window.

13. At the Command Prompt, run the command as follows:

```
node studentHTTPapi.js
```

The command starts the server. Once the server is up and running, a message is logged to the console indicating that the server is listening on the specified port, as shown in Figure 8.2.

```
C:\MyApps>node studentHTTPapi.js
Server is running on port 3000
```

Figure 8.2: Starting the Server

The server is now ready to receive requests and send responses.

14. To use curl to interact with the Web server, open another Command Prompt window.
15. To obtain the records of all the students, at the command prompt, run the command as follows:

```
curl http://localhost:3000/students
```

This command displays information about all the students as shown in Figure 8.3.

```
C:\MyApps>curl http://localhost:3000/students
[{"id":1,"name":"David Miller","age":12,"grade":"A"
}, {"id":2,"name":"Adam Smith","age":13,"grade":"B"}
, {"id":3,"name":"John Williams","age":12,"grade":"A"
}]
```

Figure 8.3: Information of all the Students

16. To obtain the records of a particular student by specifying the ID, at the Command Prompt, run the command as follows:

```
curl http://localhost:3000/students/2
```

The command specifies the ID as 2. The details of the student with ID as 2 are displayed as shown in Figure 8.4.

```
C:\MyApps>curl http://localhost:3000/students/2
{"id":2,"name":"Adam Smith","age":13,"grade":"B"}
```

Figure 8.4: Information of Student with Specified ID

17. To create a new student record, at the command prompt, run the command as:

```
curl -X POST -H "Content-Type: application/json" -d
"{"name": "Michael Faraday", "age": 11, "grade":
"A"}" http://localhost:3000/students
```

In this example, the `POST` request is used to create a new student record, with the information as follows:

- Name: Michael Faraday
- Age: 11
- Grade: A
- ID: 4

The command creates the new student record and displays the details as shown in Figure 8.5.

```
C:\MyApps>curl -X POST -H "Content-Type: application/json" -d
"{"name": "Michael Faraday", "age": 11,
"grade": "A"}" http://localhost:3000/students
{"id":4,"name":"Michael Faraday","age":11,"grade":"A"}
```

Figure 8.5: Information of New Student

18. To check whether this student record has been created in the database, at the Command Prompt, run the command as follows:

```
curl http://localhost:3000/students
```

This command executes and displays the records of all the students as shown in Figure 8.6.

```
C:\MyApps>curl http://localhost:3000/students
[{"id":1,"name":"David Miller","age":12,"grade":"A"}, {"id":2,"name":"Adam Smith","age":13,"grade":"B"}, {"id":3,"name":"John Williams","age":12,"grade":"A"}, {"id":4,"name":"Michael Faraday","age":11,"grade":"A"}]
```

Figure 8.6: New Student Added in the Database

19. To update a student record with a specific ID, at the Command Prompt, run the command as follows:

```
curl -X PUT -H "Content-Type: application/json" -d
"{"name": "David Willams", "age": 11, "grade": "A"}" http://localhost:3000/students/3
```

In this example, the `PUT` request is used to update the information of the student with `ID` as 3. In this request, the name of the student is changed from John to David, and age is changed from 12 to 11. The command is executed, and the updated student information is displayed as shown in Figure 8.7.

```
C:\MyApps>curl -X PUT -H "Content-Type: application/
json" -d "{\"name\": \"David Willams\", \"age\": 11,
\"grade\": \"A\"}" http://localhost:3000/students/3
{"id":3,"name":"David Willams","age":11,"grade":"A"}
```

Figure 8.7: Updating Student Information

20. To check whether this student record has been updated in the database, at the Command Prompt, run the command as follows:

```
curl http://localhost:3000/students
```

This command executes and displays the records of all the students as shown in Figure 8.8.

```
C:\MyApps>curl http://localhost:3000/students
[{"id":1,"name":"David Miller","age":12,"grade":"A"},
 {"id":2,"name":"Adam Smith","age":13,"grade":"B"}, {
 "id":3,"name":"David Willams","age":11,"grade":"A"}, {
 "id":4,"name":"Michael Faraday","age":11,"grade":"A"
}]
```

Figure 8.8: Student Information Updated in the Database

21. To delete a student record with a specific ID, at the Command Prompt, run the command as follows:

```
curl -X DELETE http://localhost:3000/students/4
```

In this example, the `DELETE` request is used to delete student record where the `ID` is 4. The command deletes the record and displays a message as shown in Figure 8.9.

```
C:\MyApps>curl -X DELETE http://localhost:3000/students/4
{"message":"Student deleted"}
```

Figure 8.9: Deleting Student Record

22. To check whether the student record has been deleted from the database, at the command prompt, run the command as follows:

```
curl http://localhost:3000/students
```

This command executes and displays the records of all the students as shown in Figure 8.10.

```
C:\MyApps>curl http://localhost:3000/students
[{"id":1,"name":"David Miller","age":12,"grade":"A"}, {"id":2,"name":"Adam Smith","age":13,"grade":"B"}, {"id":3,"name":"David Williams","age":11,"grade":"A"}]
```

Figure 8.10: Student Record Deleted from the Database

8.7 Using RESTful APIs with Express.js

Now, consider that the user wants to update the same student application to validate the data entered to create a new student. This can be done using RESTful API. In this case, a schema must be created and the data passed by the client must be verified against the schema.

To do this, the user must perform these steps:

1. Ensure that Node.js and Express.js are installed.
2. To use schema, open a Command Prompt window.
3. Navigate to the same folder where Express.js is installed.
4. To install the schema package, at the command prompt, run the command as follows:

```
npm install json-schema express-jsonschema
```

The command installs the package as shown in Figure 8.11.

```
C:\MyApps>npm install json-schema express-jsonschema
added 1 package, and audited 70 packages in 2s
11 packages are looking for funding
  run `npm fund` for details
found 0 vulnerabilities
```

Figure 8.11: Installing the Schema Package

- To install the validator package, at the Command Prompt, run the command as follows:

```
npm install express-validator
```

The command installs the package as shown in Figure 8.12.

```
C:\MyApps>npm install express-validator
added 3 packages, and audited 73 packages in 1s
11 packages are looking for funding
  run `npm fund` for details
found 0 vulnerabilities
```

Figure 8.12: Installing the Validator Package

- To create the schema, open VS Code or a Notepad file and type the code given in Code Snippet 8.

Code Snippet 8:

```
{
  "$schema": "https://json-schema.org/draft/2019-
09/schema",
  "$id": "studentSchema",
  "title": "Students",
  "description": "Student records",
  "type": "object",
  "properties": {
    "id": {
      "type": "integer",
      "description": "Unique identifier for the
student",
      "required": true
    },
    "name": {
      "type": "string",
      "description": "Full name of the student",
      "required": true
    },
    "age": {
      "type": "integer",
```

```

        "description": "Age of the student",
        "required": true

    },
    "grade":
    {
        "type": "string",
        "description": "Grade of the student",
        "required": true
    }
}

```

In this schema, the details related to the students are defined as given in Table 8.4.

Detail	Data Type	Required
id	integer	true
name	string	true
age	integer	true
grade	string	true

Table 8.4: Schema Definitions

7. Save the file as `studentSchema.json` in the same folder where `Express.js` is installed.
8. Make a copy of the file `studentHTTPapi.js` and name it as `studentRESTfulapi.js`.
9. Open the `studentRESTfulapi.js` file.
10. To import the validator and reference the `studentSchema.json` file, update the code as given in Code Snippet 9.

Code Snippet 9:

```

const express = require('express');
const app = express();
const port = 3000;
const { check, validationResult } = require('express-validator');
const studentSchema = require('./studentSchema.json');
app.use(express.json());

```

The additional code in Code Snippet 9, specifies that the `express-validator` must extract the validation errors from the request and store it in the `validationResult` object. Additionally, the `studentSchema.json` file is imported and stored in the `studentSchema` object.

11. To update the code for creating a new student so that the data can be validated, update the code as given in Code Snippet 10.

Code Snippet 10:

```
/ Create a new student
app.post('/students', [
  // Using check for validation
  check('name').isString(),
  check('age').isInt({ min: 9, max: 13}),
  check('grade').isString(),
], (req, res) => {
  const errors = validationResult(req);

  if (!errors.isEmpty()) {
    //Handling validation errors
    return res.status(400).json({ message: 'Errors in
entered data'});
  } else {
    const { name, age, grade } = req.body;
    const student = { id: nextStudentId++, name, age,
grade };
    students.push(student);
    res.status(201).json(student);
  }
});
```

In Code Snippet 10, the `check` function is used to validate if the data type of the details provided is the same as given in Table 8.4. The data type is checked using the `isString` and `isInt` methods. The code also checks if the age specified is between 9 and 13 by specifying the `min` property as 9 and the `max` property as 13. If all the data entered passes the validation, the new student is created and the new student details are displayed. Otherwise, an error message, `Errors in entered data`, is displayed.

12. Retaining the rest of the code as is, save the file.
13. To start the server, open a Command Prompt window.

14. At the command prompt, run the command as:

```
node studentRESTfulapi.js
```

The command starts the server. Once the server is up and running, a message is logged to the console indicating that the server is listening on the specified port, as shown in Figure 8.13.

```
C:\MyApps>node studentRESTfulapi.js  
Server is running on port 3000
```

Figure 8.13: Starting the Server

15. To create a new student record, at the Command Prompt, run the command as follows:

```
curl -X POST -H "Content-Type: application/json" -d  
"{"name": "Michael Faraday", "age": 11, "grade":  
"A"}" http://localhost:3000/students
```

The data provided in this command passes the validation rule. Therefore, the student is added to the database, as shown in Figure 8.14.

```
C:\MyApps>curl -X POST -H "Content-Type: application/json" -d  
"{"name": "Michael Faraday", "age": 11, "grade": "A"}"  
http://localhost:3000/students  
{"id":4, "name": "Michael Faraday", "age":11, "grade": "A"}
```

Figure 8.14: Adding a New Student

16. To check whether this student record has been created in the database, at the Command Prompt, run the command as follows:

```
curl http://localhost:3000/students
```

This command executes and displays the records of all the students as shown in Figure 8.15.

```
C:\MyApps>curl http://localhost:3000/students  
[{"id":1, "name": "David Miller", "age":12, "grade": "A"},  
 {"id":2, "name": "Adam Smith", "age":13, "grade": "B"},  
 {"id":3, "name": "John Williams", "age":12, "grade": "A"},  
 {"id":4, "name": "Michael Faraday", "age":11, "grade": "A"}]
```

Figure 8.15: New Student Added in the Database

17.To create another new student record, at the Command Prompt, run the command as follows:

```
curl -X POST -H "Content-Type: application/json" -d
"{"name": "Tony Allen", "age": 15, "grade": "A}"
http://localhost:3000/students
```

In this command, the age is specified as 15, which is more than the maximum age, 13. Therefore, this data does not pass the validation and an error message is displayed, as shown in Figure 8.16.

```
C:\MyApps>curl -X POST -H "Content-Type: application
/json" -d "{\"name\": \"Tony Allen\", \"age\": 15, \
\"grade\": \"A\"}" http://localhost:3000/students
>{"message": "Errors in entered data"}
```

Figure 8.15: Error Message Displayed Due to Incorrect age

18.To create another new student record, at the Command Prompt, run the command as follows:

```
curl -X POST -H "Content-Type: application/json" -d
"{"name": "Tony Allen", "age": 11}"
http://localhost:3000/students
```

In this command, the grade is not specified. However, as per the schema, grade is a required detail. Therefore, this data does not pass the validation, and an error message is displayed, as shown in Figure 8.17.

```
C:\MyApps>curl -X POST -H "Content-Type: application
/json" -d "{\"name\": \"Tony Allen\", \"age\": 11}"
http://localhost:3000/students
>{"message": "Errors in entered data"}
```

Figure 8.17: Error Message Displayed Due to Missing grade

8.8 Summary

- APIs translate a user's instructions into a format, which can be interpreted by the system to process the request and send back the response.
- An API facilitates communication and interaction between software applications through a set of rules and protocols.
- Depending upon the scope and usage, Node.js supports several APIs, such as core, third-party, custom, and RESTful APIs.
- HTTP API uses the HTTP protocol to facilitate communication and exchange of data between the systems.
- A RESTful API is an HTTP API that follows the principles and limitations of the REST architectural style to design networked applications.

Test Your Knowledge

1. You have created a simple Web application for a library. You want to find the details of a book using its name. Complete the code:

```
app.get('/books/:name', (req, res) => {
    const bookName = req.params.name;

    // Add missing code

    if (book) {
        res.json(book);
    } else {
        res.status(404).json({ message: 'Book not found' });
    }
});
```

a) const book = books.find((b) => b.name.toLowerCase())
b) const bookName = req.params.name;
 const book = books.find((b) => b.name.toLowerCase()
 ==== bookName.toLowerCase());
c) books.find(bookName)
d) const bookName = req.params.name.find(Name)

2. You are working on a Web application that must interact with a server to retrieve information. Which architecture will be useful for designing APIs to ensure simplicity, scalability, and statelessness?

a) RESTful APIs
b) Custom APIs
c) Routing APIs
d) Sequential APIs
3. You are working on a Web application. While communicating with the server using HTTP APIs which method will you use to update the existing information on the server?

a) GET
b) POST
c) DELETE
d) PUT

4. You are responsible for maintaining an online application for a hotel. In the hotel, there was a party and the manager entered wrong name of a guest, by mistake. You are required to update the guest list with the correct name. Which of the following command you will use to achieve this?

- a) Update Query
- b) curl -X PUT -H "Content-Type: application/json" -d '{"key1": "value1", "key2": "value2"}' http://your-api-endpoint
- c) curl -X POST "Content-Type: application/json" -d '{"key1": "value1", "key2": "value2"}' http://your-api-endpoint
- d) DELETE Query

5. Which of the following HTTP methods will you use to remove resources from the server?

- a) GET
- b) POST
- c) DELETE
- d) PUT

Answers to Test Your Knowledge

1	b
2	a
3	d
4	b
5	c

Try It Yourself

1. Create a simple RESTful API for managing employee details using Node.js and Express.

Table 8.5 lists the sample data for employees.

ID	Name	Age	Designation
1	Annie	27	Developer
2	Jack	30	Analyst
3	Paul	31	Developer
4	Harry	29	Sr. Analyst

Table 8.5: Employees Data

Install Express.js framework. Write a Node.js program to create the API for managing employee information.

- a) Create a schema for the employee information.
- b) Create an array using the data in Table 8.5.
- c) Define GET route to fetch all employee details.
- d) Define GET route for fetching the details of a specific employee by ID.
- e) Add details of a new employee.
- f) Validate the data entered.
- g) Update an employee by ID.
- h) Delete an employee by ID.
- i) Display the results on the console.
- j) Filter the employee data based on a specified Designation from the server when a GET request is made using a custom method.



SESSION 9

INTEGRATION OF NODE.JS WITH MONGODB

Learning Objectives

In this session, students will learn to:

- Describe the process to connect Node.js with MongoDB
- Demonstrate the CRUD operations on a MongoDB database using Node.js
- Explain the building of CRUD APIs in Node.js

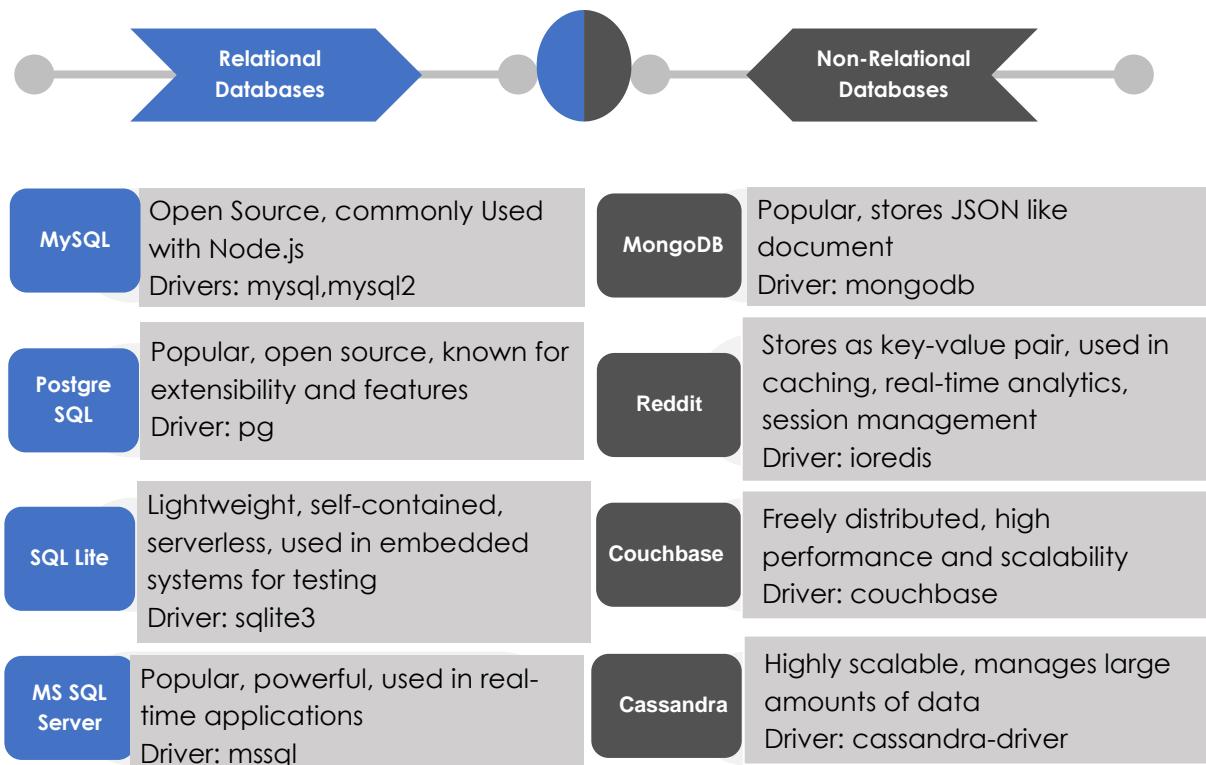
Database is an important part of any software application. It provides the space for storing the information or data. The arrangement of data in the database is subject to the requirements of the system. Once the data is arranged, various operations can be carried out on the data. To perform these operations, an interaction is essential between the data and the application. This session will explain steps to install MongoDB database and make it interact with Node.js to perform required tasks.

9.1 Introduction to Database Connectivity

Data is an integral part of any application that must be collected, stored, and updated periodically for future reference. To facilitate data management in a Web application there must be interaction between the database layer and the presentation layer. This interaction is enabled through a component

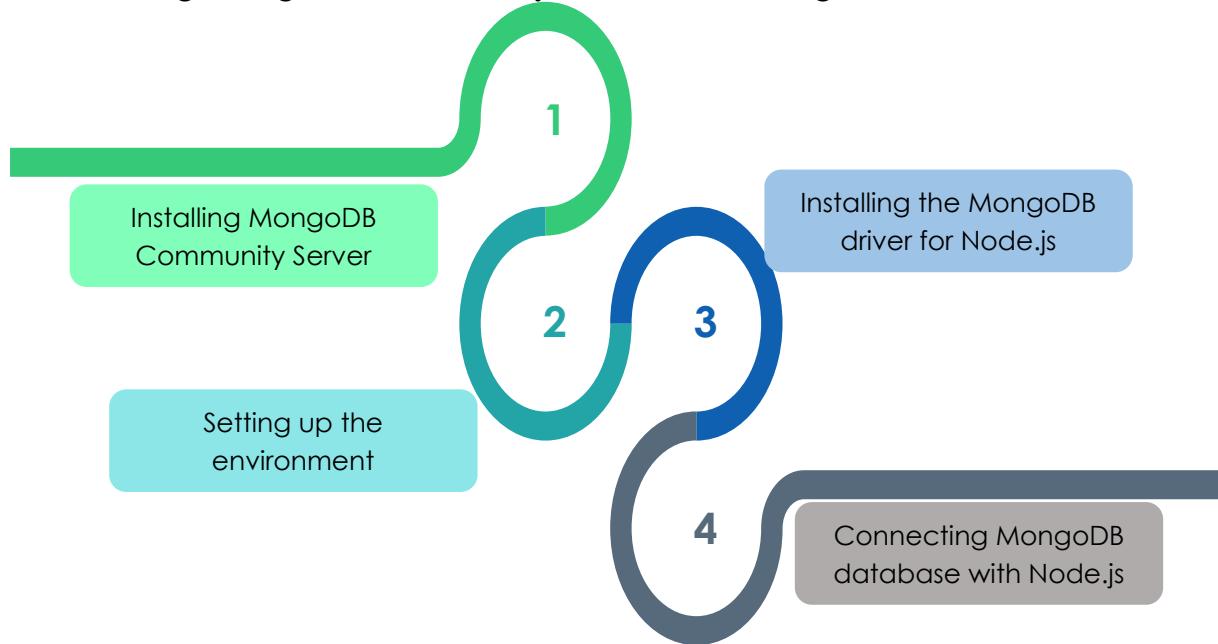
referred to as a driver, which acts as an intermediate communication point between the database and the presentation part or the front end.

Node.js is a runtime environment that supports a wide range of relational and NoSQL databases. It facilitates the interaction between the database layer and the presentation layer. It has a variety of libraries and modules that make the interaction with the databases easy. These libraries provide the methods and functions for storing, updating, and retrieving data. Developers can select a database system based on the application requirements and install the required database drivers using the Node Package Manager (npm). Some of the common databases supported by Node.js are as follows:



9.2 Connecting MongoDB with Node.js

Connecting MongoDB with Node.js involves following tasks:



Installing MongoDB Community Server

To install the MongoDB Community Server on Windows systems, perform following steps:

1. Open a browser and navigate to the URL:
<https://www.mongodb.com/try/download/community>
The MongoDB Community Server Download page opens.
2. Scroll down and click **Select Package** under the **Community Server** section.
3. On this page, scroll down and ensure that:
 - In the **Version** drop-down, **7.0.3 (current)** is selected.
 - In the **Platform** drop-down, **Windows x64** is selected.
 - In the **Package** drop-down, **msi** is selected.
4. Click **Download**.
5. After the download is complete, run the installer.

The **MongoDB 7.0.3 2008R2Plus SSL (64 bit) Setup** wizard opens as shown in Figure 9.1.



Figure 9.1: Welcome Page of the MongoDB Installation Wizard

6. Click **Next**.

The **End-User License Agreement** page of the wizard opens as shown in Figure 9.2.

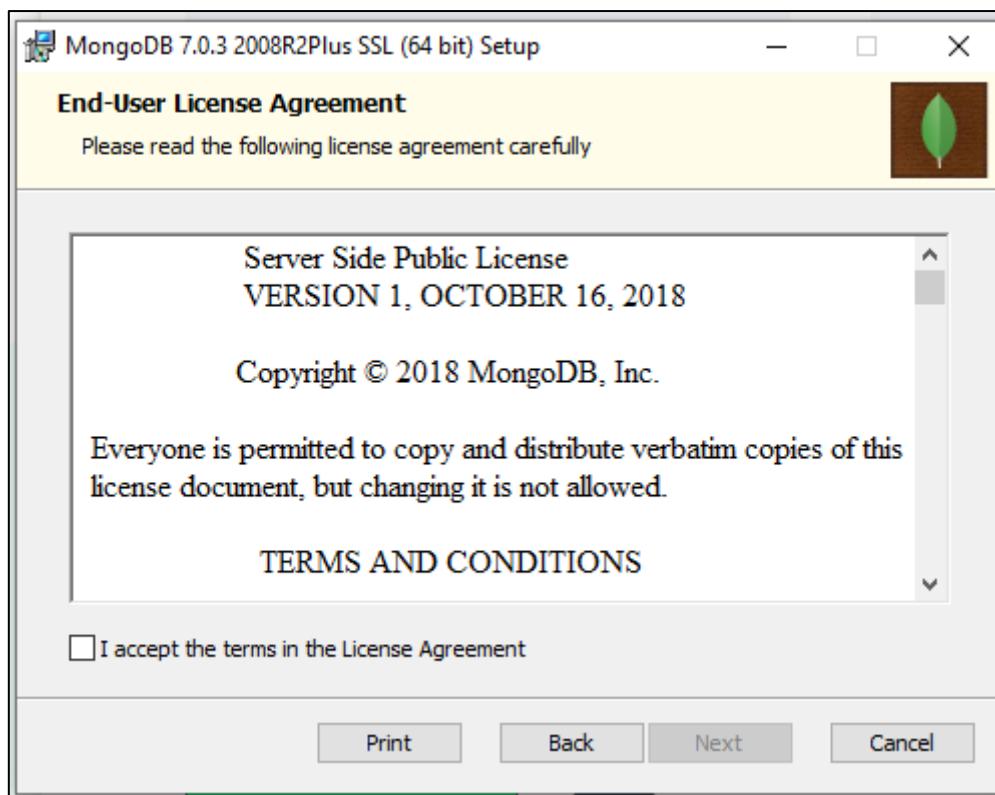


Figure 9.2: End-User License Agreement Page

7. To proceed, select the **I accept the terms in the License Agreement** check box.
8. Click **Next**.

The **Choose Setup Type** page of the wizard opens as shown in Figure 9.3.

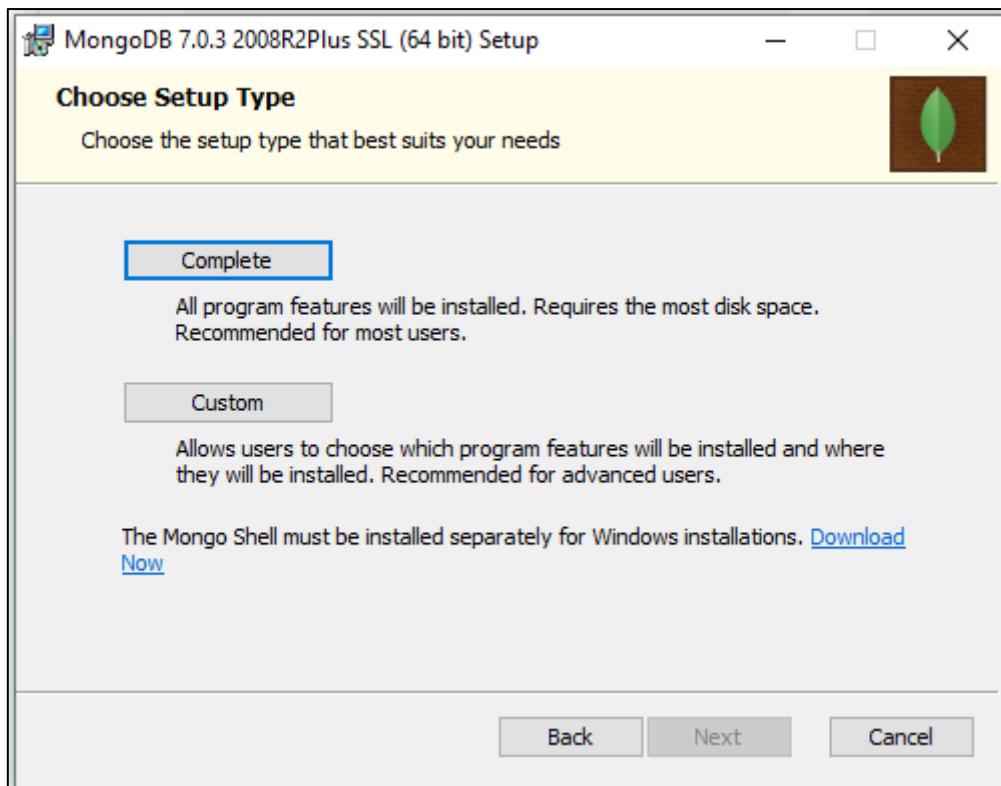


Figure 9.3: Choose Setup Type Page

9. Click **Complete**.

The **Service Configuration** page of the wizard opens as shown in Figure 9.4.

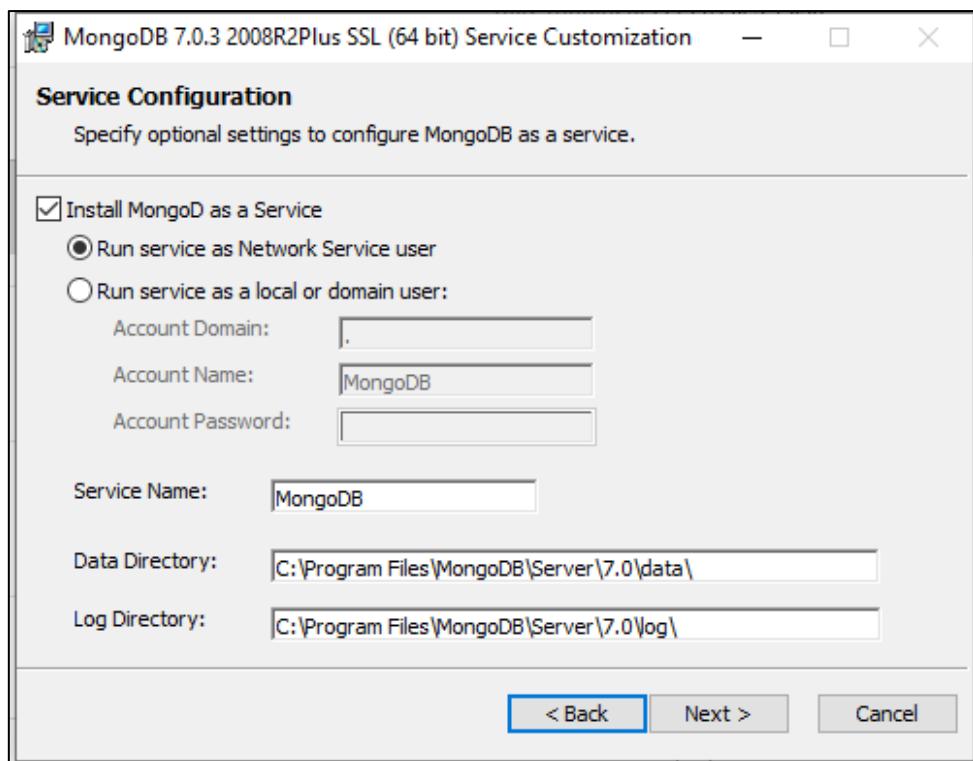


Figure 9.4: Service Configuration Page

10. Click **Next**.

The **Install MongoDB Compass** page of the wizard opens as shown in Figure 9.5.

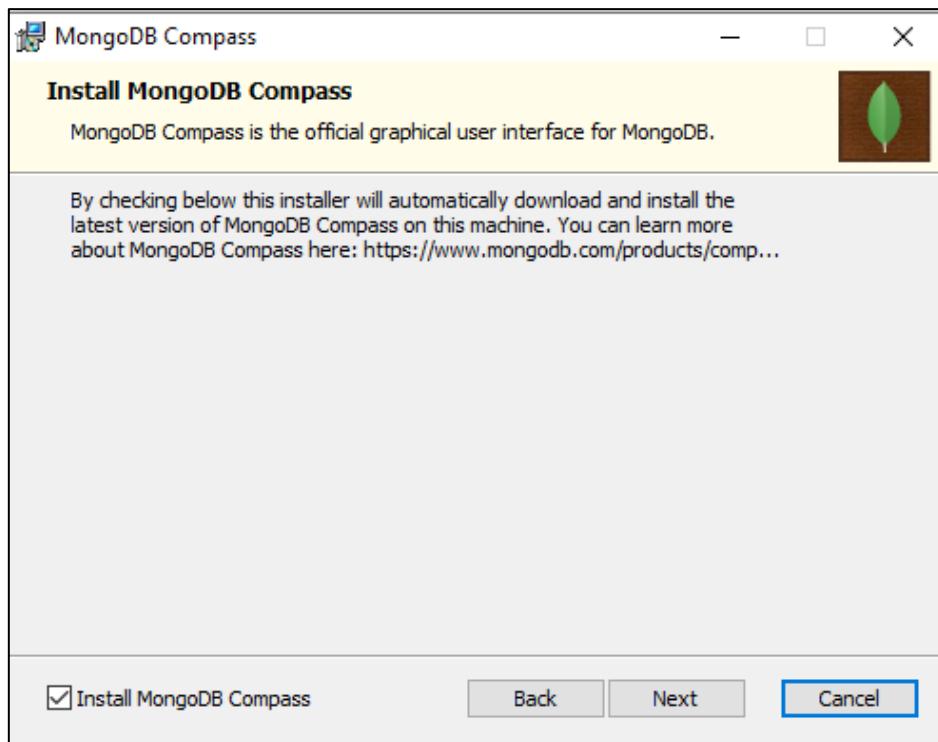


Figure 9.5: Install MongoDB Compass Page

11. Ensure that the **Install MongoDB Compass** checkbox is selected.

12. Click **Next**.

The **Ready to install MongoDB 7.0.3 2008R2Plus SSL (64 bit)** page of the wizard opens as shown in Figure 9.6.

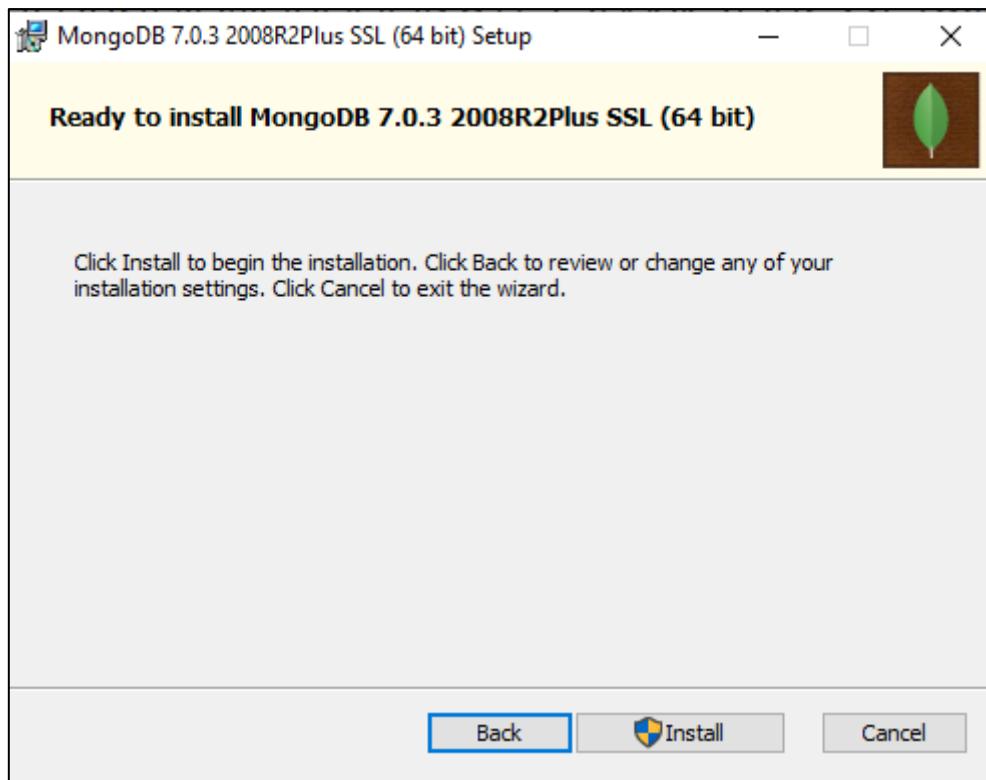


Figure 9.6: Ready to Install Page

13. Click **Install**.

The **Installing MongoDB 7.0.3f 2008R2Plus SSL (64 bit)** page of the wizard opens as shown in Figure 9.7.

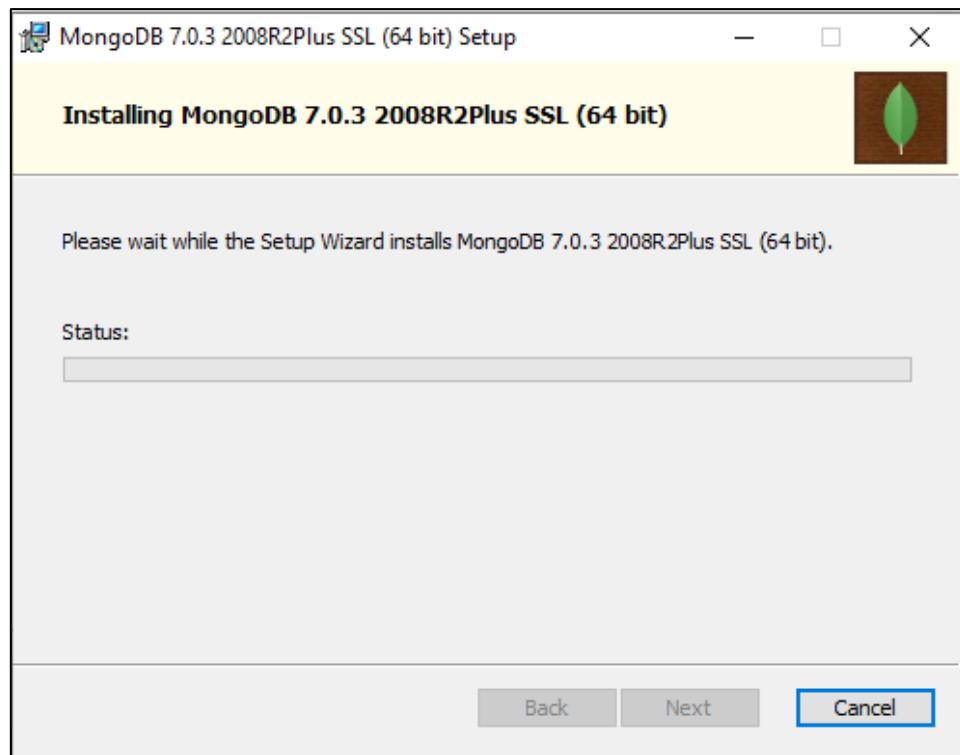


Figure 9.7: Installation in Progress Page

The **Completed the MongoDB 7.0.3 2008R2Plus SSL (64 bit) Setup Wizard** page opens as shown in Figure 9.8.

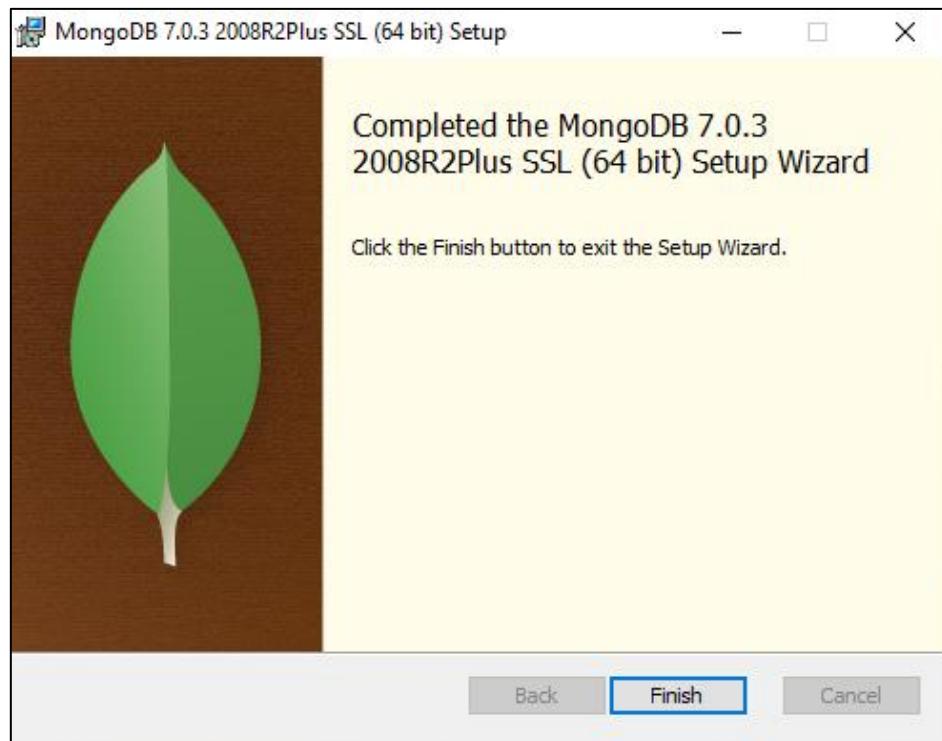


Figure 9.8: Installation Completion Page

14. Click **Finish**.

Setting up the Environment

To set up the environment, perform following steps:

1. Open Windows Command Prompt (cmd.exe) as an **Administrator**.
2. Create the data directory where MongoDB will store all the data, as shown in Figure 9.9.

```
C:\>md "\data\db"  
C:\>
```

Figure 9.9: Creating Directories



MongoDB listens for connections from clients only on port 27017, by default. The default storage path for data is /data/db directory.

To start the **MongoDB** database, run the command:

```
"C:\Program Files\MongoDB\Server\6.0\bin\mongod.exe" --  
dbpath="c:\data\db"
```

The command executes as shown in Figure 9.10.

```
C:\Users\Linda>"C:\Program Files\MongoDB\Server\6.0\bin\mongod.exe" --  
dbpath="c:\data\db"  
{"t": {"$date": "2023-12-12T18:04:53.687+05:30"}, "s": "I", "c": "NETWORK",  
"id": 4915701, "ctx": "thread1", "msg": "Initialized wire specification",  
"attr": {"spec": {"incomingExternalClient": {"minWireVersion": 0, "maxWireVersion": 17},  
"incomingInternalClient": {"minWireVersion": 0, "maxWireVersion": 17},  
"outgoing": {"minWireVersion": 6, "maxWireVersion": 17}, "isInternalClient": true}}}
```

Figure 9.10: Starting the MongoDB Database

The `--dbpath` option specifies the database directory. If the MongoDB database server is running correctly, the message: **waiting for connections** is displayed as shown in Figure 9.11.

```
{"t":{"$date":"2023-12-12T18:04:57.015+05:30"},"s":"I", "c":"FTDC",  
  "id":20631, "ctx":"ftdc","msg":"Unclean full-time diagnostic da  
ta capture shutdown detected, found interim file, some metrics may ha  
ve been lost","attr":{"error":{"code":0,"codeName":"OK"}}}  
{ "t": {"$date": "2023-12-12T18:04:57.034+05:30"}, "s": "I", "c": "NETWORK",  
  "id": 23015, "ctx": "listener", "msg": "Listening on", "attr": {"addr  
ess": "127.0.0.1"} }  
{ "t": {"$date": "2023-12-12T18:04:57.034+05:30"}, "s": "I", "c": "NETWORK",  
  "id": 23016, "ctx": "listener", "msg": "Waiting for connections", "a  
ttr": {"port": 27017, "ssl": "off"} }
```

Figure 9.11: Waiting for Connections Message

Now, the MongoDB instance `mongod` is successfully running.



Add `C:\Program Files\MongoDB\Server\6.0\bin` to the system path. This allows executing the `mongod` instance instantly without having to navigate to this folder each time to run the instance.

Installing the MongoDB Driver for Node.js

After installing the MongoDB server and setting up the necessary environment, the driver for Node.js to access the MongoDB database must be installed.

To install native MongoDB drivers using npm, run the command:

```
npm install mongodb
```

The command executes as shown in Figure 9.12.

```
C:\Users\Linda\nodejs>npm install mongodb  
added 12 packages, and audited 13 packages in 3s  
found 0 vulnerabilities
```

Figure 9.12: Installation of MongoDB Driver

Connecting MongoDB Database with Node.js

After installing the driver, connect the MongoDB database with Node.js.

To connect to a MongoDB database, perform following steps:

1. Open VS Code or Notepad and type the code as shown in Code Snippet 1.

Code Snippet 1:

```
const { MongoClient } = require('mongodb')
const client = new MongoClient('mongodb://localhost:27017')
client.connect()
  .then(() => {
    console.log('Connected to MongoDB Successfully!')
    console.log('Exiting from MongoDB')
    client.close()
  })
  .catch(error => console.log('Failed to connect to MongoDB!', error))
```

2. Save this code as `connectmongo.js`.

Table 9.1 describes Code Snippet 1.

Code	Description
<code>const { MongoClient } = require('mongodb')</code>	Imports the MongoClient class from the MongoDB package to initiate the connection with a MongoDB database using the MongoDB Node.js driver
<code>const client = new MongoClient('mongodb://localhost:27017')</code>	Creates a new MongoDB client instance by specifying the connection URL
<code>client.connect()</code>	Initiates the connection to the MongoDB database and returns a promise object
<code>.then(() => { console.log() })</code>	Displays success messages to exhibit that the connection has been established successfully
<code>client.close()</code>	Closes the connection to the MongoDB database and frees up the resources
<code>.catch(error => console.log())</code>	Displays any errors while establishing the connection

Table 9.1: Description of Code Snippet 1

To run `connectmongo.js`, type following command in the Command Prompt:

```
node connectmongo.js
```

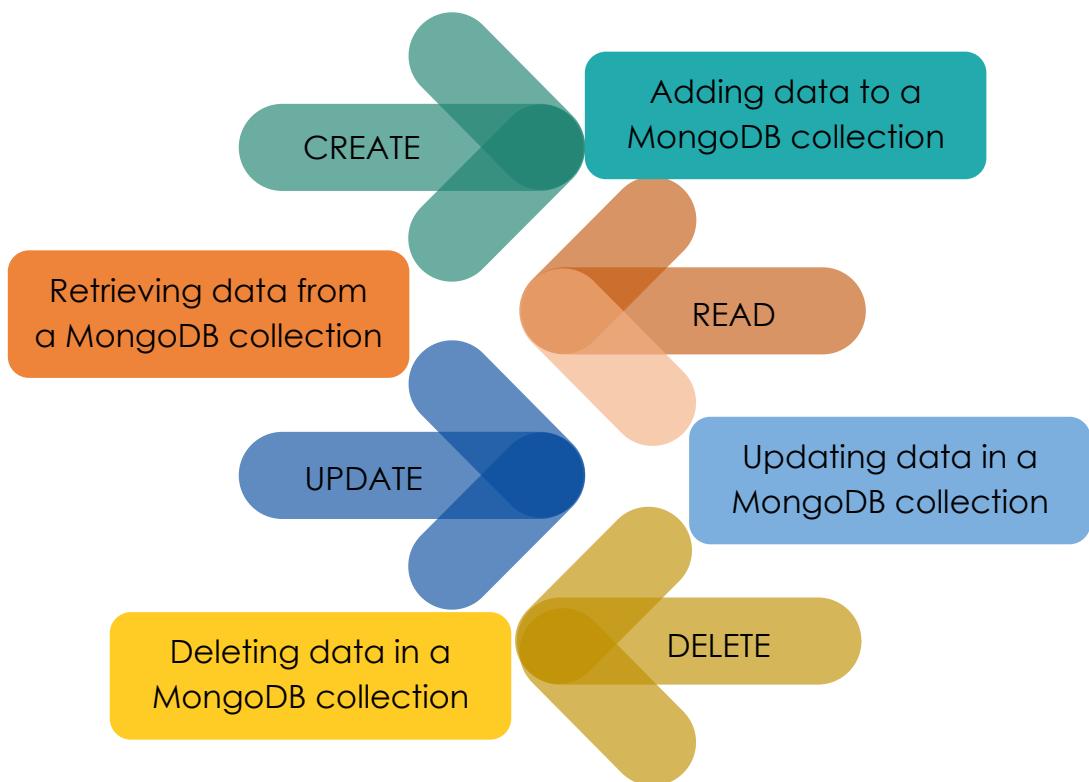
Once the server is up and running, a message is logged to the console to indicate that the connection to MongoDB is successful, as shown in Figure 9.13.

```
C:\Users\Linda\nodejs>node connectmongo.js
Connected to MongoDB Successfully!
Exiting from MongoDB
```

Figure 9.13: Console Displaying Successful Connection to MongoDB

9.3 Performing Create, Read, Update, and Delete (CRUD) Operations in Node.js

Developers can perform CRUD operations to manage data in a database.



In a MongoDB database, data is organized into collections and each document in a collection represents a data record.

9.3.1 Create Operation

Let us create a MongoDB database named, `Library`. This database will contain the `books` collection with three documents.

1. Open VS Code or Notepad and type the code given in Code Snippet 2.

Code Snippet 2:

```
const { MongoClient } = require('mongodb');
const url = 'mongodb://localhost:27017';
const databaseName = 'Library';
```

```

const collectionName = 'books';

const dataToInsert = [
  { Book_id:101, Book_name: 'The Secret of The House On The Hill', Author_name: 'Julia Harris', Price: '$10', Age_group:'15+', Book_type:'Thriller' },
  { Book_id:2001, Book_name: 'The Court Of The Evil', Author_name: 'Nolan Evans', Price: '$17', Age_group:'10+',Book_type:'Young-Adult' },
  { Book_id:3001, Book_name: 'When You Wish Upon A Star', Author_name: 'Ashlyn Clark', Price: '$18.99', Age_group:'18+',Book_type:'Romance' },
];
const client = new MongoClient(url);
client.connect()
  .then(() => {
    const db = client.db(databaseName);
    const collection = db.collection(collectionName);
    return collection.insertMany(dataToInsert);
  })
  .then(result => {
    console.log(` ${result.insertedCount} documents inserted into the collection in the "${databaseName}" database.`);
  })
  .catch(err => {
    console.error('Failed to connect to Database:', err);
  })
  .finally(() => {
    client.close();
  });

```

2. Save the code as `createop.js`.

The code:

- Imports the `MongoClient` class and defines the connection details, names of the database and collection, and an array of documents to insert into the collection.
- Creates a new `MongoClient` instance and establishes a connection to the MongoDB server specified in the URL using the `connect` method.
- Accesses the database and collection using the `success` callback argument in the `then` block.
- Creates a reference to the database and collection using the `client.db(databaseName)` and `db.collection(collectionName)`.
- Inserts the `dataToInsert` array into the collection using the `insertMany` method.
- Handles the results to log the number of documents that were successfully inserted using another `then` block.

- Handles the errors thrown during connecting or inserting using the `catch` block.
- Ensures that the MongoDB client connection is closed using the `finally` block.

To execute the code, type following command in the Command Prompt:

```
node createop.js
```

A message is logged in the console to indicate that three documents are inserted into the `books` collection of the `Library` database, as shown in Figure 9.14.

```
C:\Users\Linda\nodejs>node createop.js
3 documents inserted into the collection in the
"Library" database.
```

Figure 9.14: Console Displaying Insertion of Documents into the Collection

9.3.2 Read Operation

Let us view the three documents inserted into the `books` collection of the `Library` database.

1. Open VS Code or Notepad and type the code given in Code Snippet 3.

Code Snippet 3:

```
const { MongoClient } = require('mongodb');
const url = 'mongodb://localhost:27017';
const databaseName = 'Library';
const collectionName = 'books';
const client = new MongoClient(url);
client.connect()
  .then(() => {
    const db = client.db(databaseName);
    const collection = db.collection(collectionName);
    return collection.find({}).toArray();
  })
  .then(docs => {
    console.log(`Found ${docs.length} documents in the
"${collectionName}" collection:`);
    console.log(docs);
  })
  .catch(err => {
    console.error('Failed to connect to Database:', err);
  })
}
```

```
.finally(() => {
  client.close();
});
```

2. Save the code as `readop.js`.

The code:

- Imports the `MongoClient` class and defines the connection details, names of the database and collection, and an array of documents.
- Creates a new `MongoClient` instance and establishes a connection to the MongoDB server specified in the URL using the `connect` method.
- Accesses the `Library` database and the `books` collection using the `success` callback in the `then` block.
- Retrieves all the documents in the collection using the `find` method and converts the result to an array of documents using the `toArray` method.
- Handles the retrieved documents and logs them to the console using another `then` block.
- Manages any errors thrown during connecting or querying using the `catch` block.
- Ensures that the MongoDB client connection is closed using the `finally` block.

To execute the code, type following command in the Command Prompt:

```
node readop.js
```

Find the three documents retrieved from the `books` collections of the `Library` database, as shown in Figure 9.15.

```
C:\Users\Linda\nodejs>node readop.js
Found 3 documents in the "books" collection:
[
  {
    _id: new ObjectId('65785deb18ee8801b5712490'),
    Book_id: 101,
    Book_name: 'The Secret of The House On The Hill',
    Author_name: 'Julia Harris',
    Price: '$10',
    Age_group: '15+',
    Book_type: 'Thriller'
  },
  {
    _id: new ObjectId('65785deb18ee8801b5712491'),
    Book_id: 2001,
    Book_name: 'The Court Of The Evil',
    Author_name: 'Nolan Evans',
    Price: '$17',
    Age_group: '10+',
    Book_type: 'Young-Adult'
  },
  {
    _id: new ObjectId('65785deb18ee8801b5712492'),
    Book_id: 3001,
    Book_name: 'When You Wish Upon A Star',
    Author_name: 'Ashlyn Clark',
    Price: '$18.99',
    Age_group: '18+',
    Book_type: 'Romance'
  }
]
```

Figure 9.15: Console Displaying the Retrieved Documents

9.3.3 Update Operation

Let us update `Price` and `Age_group` values in the document having `Book_id` as 2001. The new `Price` value should be \$20 and the new `Age_group` value should be 12+.

1. Open VS Code or Notepad and type the code given in Code Snippet 4.

Code Snippet 4:

```
const { MongoClient } = require('mongodb');
const url = 'mongodb://localhost:27017';
```

```

const databaseName = 'Library';
const collectionName = 'books';
const client = new MongoClient(url);
client.connect()
  .then(() => {
    const db = client.db(databaseName);
    const collection = db.collection(collectionName);
    // Define the filter criteria to match the document to
    update
    const filter = { Book_id:2001};
    const update = {
      $set: {
        Price: '$20',
        Age_group: '12+',
      },
    };
    return collection.updateOne(filter, update);
  })
  .then(result => {
    if (result.matchedCount === 0) {
      console.log('No matching document found for the
      update.');
    } else {
      console.log('Document updated:', result.modifiedCount);
    }
  })
  .catch(err => {
    console.error('Failed to connect to Database:', err);
  })
  .finally(() => {
    client.close();
  });
})

```

2. Save the code as updateop.js.

The code:

- Imports the MongoClient class and defines the connection details, names of the database and collection, and an array of documents.
- Creates a new MongoClient instance and establishes a connection to the MongoDB server specified in the URL, using the connect method.
- Accesses the Library database and the books collection, using the success callback in the then block.
- Defines the criteria to identify the document to be modified using the filter object.
- Specifies the modifications to be applied using the update object and the \$set operator.
- Updates the document that matches the filter criteria, using the updateOne method.

- Inserts a result callback to check if a matching document is found for updating and to log the number of documents modified.
- Manages any errors thrown during connecting using the `catch` block.
- Ensures that the MongoDB client connection is closed using the `finally` block.

To execute the code, type following command in the Command Prompt:

```
node updateop.js
```

Find the number of documents that are updated in the `books` collections of the `Library` database as shown in Figure 9.16.

```
C:\Users\Linda\nodejs>node updateop.js
Document updated: 1
```

Figure 9.16: Console Displaying the Number of Updated Documents

To verify if the document with `Book_id` as `2001` has been updated, retrieve all the documents in the `books` collection by executing `readop.js`.

Type following command in the Command Prompt:

```
node readop.js
```

The `Price` and `Age_group` values have been updated in the specified document as shown in Figure 9.17.

```
C:\Users\Linda\nodejs>node readop.js
Found 3 documents in the "books" collection:
[
  {
    _id: new ObjectId('65785deb18ee8801b5712490'),
    Book_id: 101,
    Book_name: 'The Secret of The House On The Hill',
    Author_name: 'Julia Harris',
    Price: '$10',
    Age_group: '15+',
    Book_type: 'Thriller'
  },
  {
    _id: new ObjectId('65785deb18ee8801b5712491'),
    Book_id: 2001,
    Book_name: 'The Court Of The Evil',
    Author_name: 'Nolan Evans',
    Price: '$20',
    Age_group: '12+',
    Book_type: 'Young-Adult'
  },
  {
    _id: new ObjectId('65785deb18ee8801b5712492'),
    Book_id: 3001,
    Book_name: 'When You Wish Upon A Star',
    Author_name: 'Ashlyn Clark',
    Price: '$18.99',
    Age_group: '18+',
    Book_type: 'Romance'
  }
]
```

Figure 9.17: Console Displaying the Updated Values in the Document

9.3.4 Delete Operation

Let us delete the document having Book_id:2001.

1. Open VS Code or Notepad and type the code given in Code Snippet 5.

Code Snippet 5:

```
const { MongoClient } = require('mongodb');
const url = 'mongodb://localhost:27017';
const databaseName = 'Library';
const collectionName = 'books';
const client = new MongoClient(url);
```

```

client.connect()
  .then(() => {
    const db = client.db(databaseName);
    const collection = db.collection(collectionName);
    const filter = { Book_id: 2001 };
    return collection.deleteOne(filter);
  })
  .then(result => {
    if (result.deletedCount === 0) {
      console.log("Book not found");
    } else{
      console.log("Book deleted successfully")
    }
  })
  .catch(err => {
    console.error('Failed to connect to Database:', err);
  })
  .finally(() => {
    client.close();
  });

```

2. Save the code as deleteop.js.

The code:

- Imports the `MongoClient` class and defines the connection details, names of the database and collection, and an array of documents.
- Creates a new `MongoClient` instance and establishes a connection to the MongoDB server specified in the URL, using the `connect` method.
- Accesses the `Library` database and the `books` collection, using the `success` callback in the `then` block.
- Defines the criteria to identify the document to be deleted, using the `filter` object.
- Deletes the document that matches the `filter` criteria, using the `deleteOne` method.
- Inserts a `result` callback to check if a matching document is found for deleting and to log the number of documents deleted.
- Manages any errors thrown during connecting using the `catch` block.
- Ensures that the MongoDB client connection is closed using the `finally` block.

To execute the code, type following command in the Command Prompt:

```
node deleteop.js
```

The documents that matched with the specific criteria are deleted from the `books` collections in the `Library` database, as shown in Figure 9.18.

```
C:\Users\Linda\nodejs>node deleteop.js
Book deleted successfully
```

Figure 9.18: Console Displaying the Message of Document Deletion

Verify if the document with Book_id: 2001 has been deleted by retrieving the existing documents in the books collection.

Type following command in the Command Prompt:

```
node readop.js
```

The document with Book_id as 2001 has been deleted from the books collection, as shown in Figure 9.19.

```
C:\Users\Linda\nodejs>node deleteop.js
Book deleted successfully

C:\Users\Linda\nodejs>node readop.js
Found 2 documents in the "books" collection:
[
  {
    _id: new ObjectId('65785deb18ee8801b5712490'),
    Book_id: 101,
    Book_name: 'The Secret of The House On The Hill',
    Author_name: 'Julia Harris',
    Price: '$10',
    Age_group: '15+',
    Book_type: 'Thriller'
  },
  {
    _id: new ObjectId('65785deb18ee8801b5712492'),
    Book_id: 3001,
    Book_name: 'When You Wish Upon A Star',
    Author_name: 'Ashlyn Clark',
    Price: '$18.99',
    Age_group: '18+',
    Book_type: 'Romance'
  }
]
```

Figure 9.19: Console Displaying the Existing Documents After Deletion

9.4 Building CRUD APIs in Node.js

API stands for Application Programming Interface. Let us build a basic Web API to perform the CRUD operations on the `books` collection of the Library MongoDB database using Node.js and Express.js environments.

Node.js and Express.js have already been installed. Now, complete following steps:

23. Open VS Code or Notepad and type the code given in Code Snippet 6.

Code Snippet 6:

```
const express = require('express');
const { MongoClient, ObjectId } = require('mongodb');
const app = express();
const port = 3000;
const url = 'mongodb://localhost:27017';
const databaseName = 'Library';
const collectionName = 'books';
app.use(express.json());
// Create - Add a new book
app.post('/books', (req, res) => {
    const newBook = req.body;
    MongoClient.connect(url)
        .then(client => {
            const db = client.db(databaseName);
            const collection = db.collection(collectionName);
            return collection.insertOne(newBook);
        })
        .then(result => {
            res.status(201).json(newBook);
        })
        .catch(error => {
            // Handle any other errors, and log detailed error
            information
            console.error('Error during book insertion:', error);
            res.status(500).json({ error: 'Failed to add a new book' });
        });
    });
});

// Read - Get all books
app.get('/books', (req, res) => {
    MongoClient.connect(url)
        .then(client => {
            const db = client.db(databaseName);
            const collection = db.collection(collectionName);
            return collection.find({}).toArray();
        })
        .then(result => {
            res.status(200).json(result);
        })
        .catch(error => {
            console.error('Error during book retrieval:', error);
            res.status(500).json({ error: 'Failed to get books' });
        });
});
```

```

        })
        .then(books => {
            res.status(200).json(books); // Respond with the list of
books
        })
        .catch(error => {
            res.status(500).json({ error: 'Failed to retrieve books' });
        });
    });
}

// Read - Get a single book by Book_id
app.get('/books/:id', (req, res) => {
    const bookId = req.params.id;
    MongoClient.connect(url)
        .then(client => {
            const db = client.db(databaseName);
            const collection = db.collection(collectionName);
            return collection.findOne({ Book_id: parseInt(bookId) });
        });
    })
    .then(book => {
        if (!book) {
            console.error(`Book with Book_id ${bookId} not
found.`);
            res.status(404).json({ error: 'Book not found' });
        } else {
            res.status(200).json(book);
        }
    })
    .catch(error => {
        console.error('Error during book retrieval:', error);
        res.status(500).json({ error: 'Failed to retrieve the
book' });
    });
});

// Update - Modify a book by Book_id
app.put('/books/:id', (req, res) => {
    const bookId = req.params.id;
    const updatedBook = req.body;
    MongoClient.connect(url)
        .then(client => {
            const db = client.db(databaseName);
            const collection = db.collection(collectionName);
            return collection.updateOne({ Book_id: parseInt(bookId)
}, { $set: updatedBook });
        })
        .then(result => {

```

```

        if (result.matchedCount === 0) {
            res.status(404).json({ error: 'Book not found' });
        } else {
            res.status(200).json({ message: 'Book updated successfully' });
        }
    })
    .catch(error => {
        res.status(500).json({ error: 'Failed to update the book' });
    });
}

// Delete - Remove a book by Book_id
app.delete('/books/:id', (req, res) => {
    const bookId = req.params.id;
    MongoClient.connect(url)
        .then(client => {
            const db = client.db(databaseName);
            const collection = db.collection(collectionName);
            return collection.deleteOne({ Book_id: parseInt(bookId) });
        })
        .then(result => {
            if (result.deletedCount === 0) {
                res.status(404).json({ error: 'Book not found' });
            } else {
                res.status(200).json({ message: 'Book deleted successfully' });
            }
        })
        .catch(error => {
            res.status(500).json({ error: 'Failed to delete the book' });
        });
    app.listen(port, () => {
        console.log(`Server is running on port ${port}`);
    });
}

```

24. Save the code as `libraryapi.js`.

Table 9.2 describes the packages and the constants defined in Code Snippet 6.

Packages/Constants	Description
<code>const express = require('express');</code>	Imports the required packages, namely: <ul style="list-style-type: none"> • <code>express</code> to create the API

Packages/Constants	Description
const { MongoClient, ObjectId } = require('mongodb');	<ul style="list-style-type: none"> • MongoClient to connect with the database • ObjectId to work with MongoDB ObjectIds
const app = express(); const port = 3000; const url = 'mongodb://localhost:27017'; const dbName = 'Library'; const collectionName = 'books';	Creates an Express application instance named app on the specified port number to define the API routes and manage HTTP requests. Further, defines the MongoDB connection details, such as the url, dbName, and collectionName.

Table 9.2: Packages and Constants in Code Snippet 6

Table 9.3 discusses the methods and properties used in Code Snippet 6 to perform the CRUD operations.

Methods/Properties	Description
app.use(express.json())	Parses incoming data requests in JSON format with the express.json middleware.
app.listen()	Binds and listens to the connections on the specified host and port.
app.post()	Routes the HTTP post requests to the specified path. Adds new books to the database.
app.get()	Routes the HTTP get requests to the specified path. Retrieves the book from the collection using the book_id parameter.
app.put()	Routes the HTTP put requests to the specified path. Modifies the details of the book using the book_id parameter.
app.delete()	Routes the HTTP delete requests to the specified path. Deletes the book using the book_id parameter.
req.body	Property of the Express.js req object that populates the parsed data.

Methods/Properties	Description
req.params.id	Property of the Express.js req.params object that holds the <code>id</code> value passed in the route parameter. Used for fetching, deleting, and updating book details.
res.status()	Sets the HTTP status for the response corresponding to the success or failure of the operation.

Table 9.3: Methods and Properties to Perform the CRUD Operations

To execute `libraryapi.js`, type following command in the Command Prompt:

```
node libraryapi.js
```

Find the Express server up and running. A message is logged to the console to indicate that the server is running on the specified port, as shown in Figure 9.20.

```
C:\Users\Linda\nodejs>node libraryapi.js
Server is running on port 3000
```

Figure 9.20: Console Displaying the Port Number of the Server

It is now possible to send HTTP requests to the specified API endpoints. Generally, the requests can be made using tools, such as curl, Postman, or Web browser.

Use `curl` to access the API endpoints and perform following tasks:

View all the books
Access a specific book
Add (Create) a new book
Read a book
Update a book
Delete a book

Type the corresponding commands in the Command Prompt to perform the respective tasks. Note that, before opening a new Command Prompt, confirm that the server is running on port 3000.

1. To view the books in the books collection, type:

```
curl http://localhost:3000/books
```

Figure 9.21 displays the output of the execution of this command.

```
C:\Users\Linda\nodejs>curl http://localhost:3000/books
[{"_id": "65785deb18ee8801b5712490", "Book_id": 101, "Book_name": "The Secret of The House On The Hill", "Author_name": "Julia Harris", "Price": "$10", "Age_group": "15+", "Book_type": "Thriller"}, {"_id": "65785deb18ee8801b5712492", "Book_id": 3001, "Book_name": "When You Wish Upon A Star", "Author_name": "Ashlyn Clark", "Price": "$18.99", "Age_group": "18+", "Book_type": "Romance"}]
```

Figure 9.21: Information of all the Books

Figure 9.22 shows all the documents displayed in the browser.

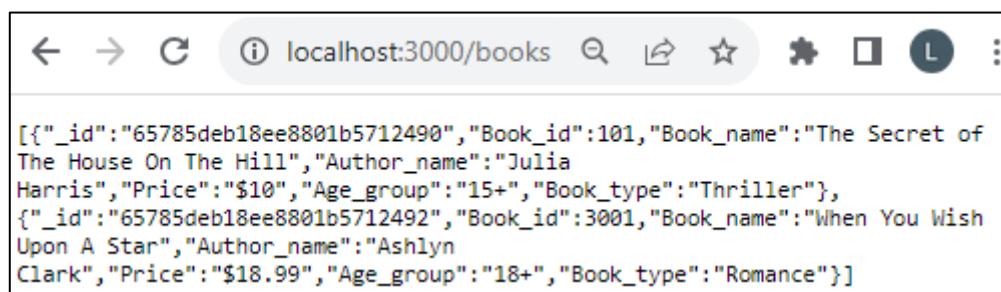


Figure 9.22: Information of all Books in the Browser

2. To access details of a specific book by ID, for instance, 101, type:

```
curl http://localhost:3000/books/101
```

Figure 9.23 displays the output of the execution of this command.

```
C:\Users\Linda\nodejs>curl http://localhost:3000/books/101
{"_id": "65785deb18ee8801b5712490", "Book_id": 101, "Book_name": "The Secret of The House On The Hill", "Author_name": "Julia Harris", "Price": "$10", "Age_group": "15+", "Book_type": "Thriller"}
```

Figure 9.23: Retrieval of the Specified Book Information

Figure 9.24 shows the document of the Book_id as 101 displayed in the browser.

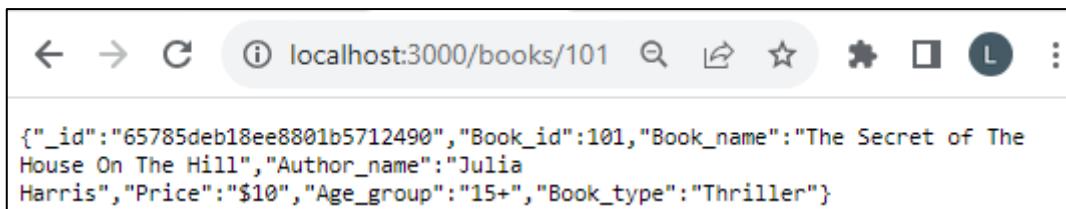


Figure 9.24: Document of Book ID 101

3. To add a book to the books collection, type:

```
curl -X POST -H "Content-Type: application/json" -d
"{"Book_id":1088, "Book_name": "Death and Me",
"Author_name": "Noah Wilson", "Price": "$16",
"Age_group": "16+", "Book_type": "Thriller"}"
http://localhost:3000/books
```

The command adds a book with following details:

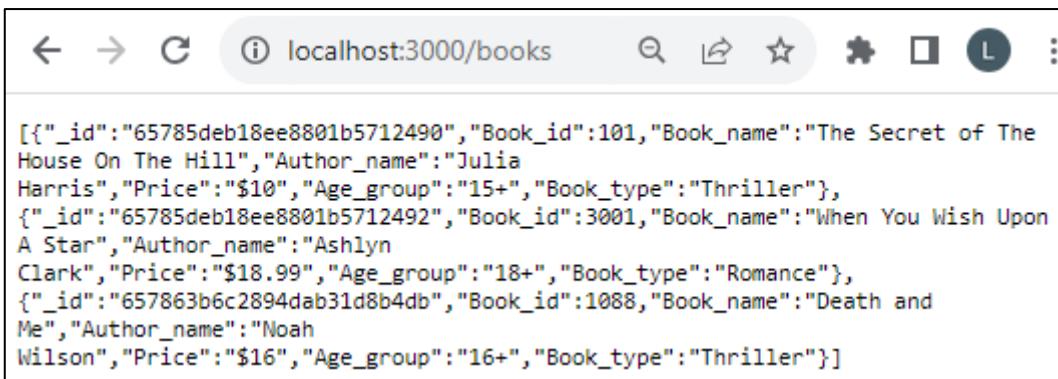
- Book_id: 1088
- Book_name: Death and Me
- Author_name: Noah Wilson
- Price: \$16
- Age_group: 16+
- Book_type: Thriller

Figure 9.25 displays the output of the execution of this command.

```
C:\Users\Linda\nodejs>curl -X POST -H "Content-Type: application/json" -d
"{"Book_id":1088, "Book_name": "Death and Me",
"Author_name": "Noah Wilson", "Price": "$16",
"Age_group": "16+", "Book_type": "Thriller"}"
http://localhost:3000/books
{"Book_id":1088, "Book_name": "Death and Me", "Author_name": "Noah Wilson", "Price": "$16", "Age_group": "16+", "Book_type": "Thriller", "_id": "657863b6c2894dab31d8b4db"}
```

Figure 9.25: Addition of a New Book

Figure 9.26 shows the documents in the books collection after the add operation displayed in the browser.



```
[{"_id": "65785deb18ee8801b5712490", "Book_id": 101, "Book_name": "The Secret of The House On The Hill", "Author_name": "Julia Harris", "Price": "$10", "Age_group": "15+", "Book_type": "Thriller"}, {"_id": "65785deb18ee8801b5712492", "Book_id": 3001, "Book_name": "When You Wish Upon A Star", "Author_name": "Ashlyn Clark", "Price": "$18.99", "Age_group": "18+", "Book_type": "Romance"}, {"_id": "657863b6c2894dab31d8b4db", "Book_id": 1088, "Book_name": "Death and Me", "Author_name": "Noah Wilson", "Price": "$16", "Age_group": "16+", "Book_type": "Thriller"}]
```

Figure 9.26: Documents after Add Operation

4. To update the information of a specific book by ID, type:

```
curl -X PUT -H "Content-Type: application/json" -d
"{"Book_id":1088, "Book_name": "Death and Me",
"Author_name": "Noah Wilson", "Price": "$20",
"Age_group": "18+", "Book_type": "Thriller}"
http://localhost:3000/books/1088
```

The command updates the details of the book with `book_id: 1088`, as follows:

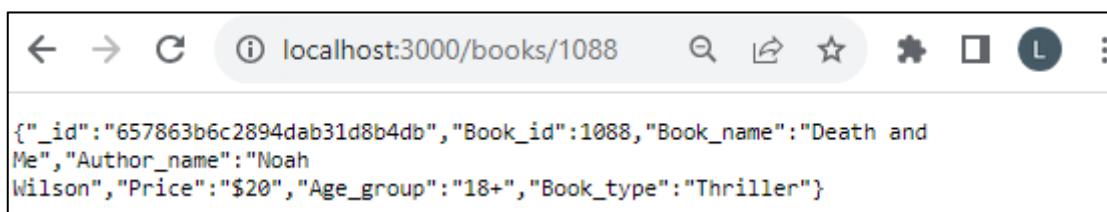
- `Price: $20`
- `Age_group: 18+`

Figure 9.27 displays the output of the execution of this command.

```
C:\Users\Linda\nodejs>curl -X PUT -H "Content-Type: application/json" -d
"{"Book_id":1088, "Book_name": "Death and Me",
"Author_name": "Noah Wilson", "Price": "$20",
"Age_group": "18+", "Book_type": "Thriller}"
http://localhost:3000/books/1088
{"message": "Book updated successfully"}
```

Figure 9.27: Retrieval of the Specified Book Information

Figure 9.28 shows the updated document displayed in the browser.



```
{"_id": "657863b6c2894dab31d8b4db", "Book_id": 1088, "Book_name": "Death and Me", "Author_name": "Noah Wilson", "Price": "$20", "Age_group": "18+", "Book_type": "Thriller"}
```

Figure 9.28: Updated Document

5. To delete a specific book by ID, type:

```
curl -X DELETE http://localhost:3000/books/101
```

The command uses the `DELETE` request to delete an existing book with `book_id: 101`.

Figure 9.29 displays the output of the execution of this command.

```
C:\Users\Linda\nodejs>curl -X DELETE http://localhost:3000/books/101
{"message": "Book deleted successfully"}
```

Figure 9.29: Deletion of the Book

Figure 9.30 shows the documents in the `books` collection after the delete operation displayed in the browser.

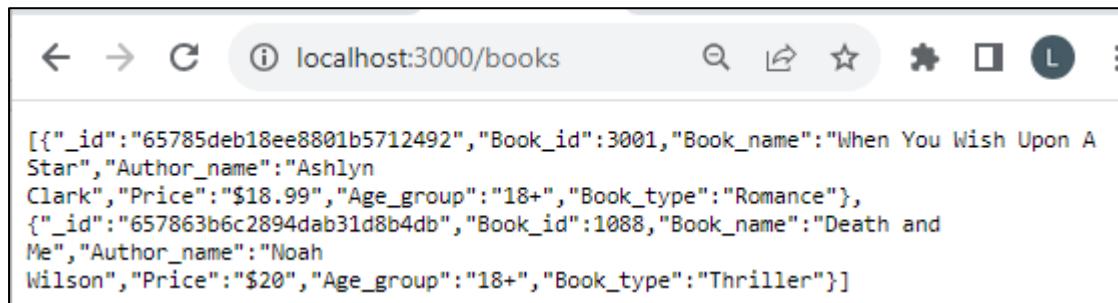


Figure 9.30: Documents after Delete Operation

9.5 Summary

- Using Node.js developers can connect with both relational and non-relational databases.
- MongoDB is a non-relational database.
- Express is a Node.js Web framework that can be used to create Web applications and APIs.
- The MongoDB Node.js Driver is the communication tool that is internally used for communication between Node.js and MongoDB.
- Libraries, such as `mongodb` and `express` can be imported into the code with the `require` keyword.
- Command line tools, such as `curl` help in transferring data over network protocols, such as HTTP.

Test Your Knowledge

1. What type of database does Node.js support?
 - a) MySQL
 - b) Oracle
 - c) MongoDB
 - d) All of these

2. In which format does Node.js exchange data?
 - a) CSV
 - b) JSON
 - c) XML
 - d) Text

3. You have started working on a MongoDB database. Which is the correct way to import the `MongoClient` class?
 - a) `const { mongodb } = require('mongodb')`
 - b) `const { MongoClient } = require('mongodb')`
 - c) `const { mongoDB } = require('MongoClient')`
 - d) `const { MongoClient } = require('mongoDB')`

4. Which tool is used to access API by sending HTTP requests to the endpoint?
 - a) Express
 - b) Init
 - c) Curl
 - d) mongodb

5. Which operation will you use to add a new record or document?
 - a) Create
 - b) Read
 - c) Update
 - d) Delete

Answers to Test Your Knowledge

1	d
2	b
3	d
4	c
5	a

Try It Yourself

1. Create a simple Web application for the art gallery and perform CRUD operations on the `GalleryOne` in the `Art` database. Table 9.4 shows the data for this collection.

ID	Artist Name	Art Type	Price
1	Annie	Painting	120\$
2	Jack	Sculpture	100\$
3	Paul	Pop art	130\$
4	Harry	Abstract art	190\$

Table 9.4: Data for GalleryOne

- a) Import the necessary packages.
- b) Create a MongoDB database named, `Art`.
- c) Add a collection named `GalleryOne`.
- d) Insert the four documents given in Table 9.4 into this collection.
- e) View the inserted documents.
- f) Update the `Price` and `Artist Name` in the document with `ID` as 4 to 210\$ and Grace respectively.
- g) Delete the document with `ID` as 3.
- h) Display the results using curl as well as a Web browser.



SESSION 10

BUILDING WEBSITES USING NODE.JS

Learning Objectives

In this session, students will learn to:

- Explain the process of creating a Web application using Node.js
- Describe the steps to connect the Web application to a MongoDB cloud database
- List the steps to upload the Web application to the GitHub repository
- Explain the procedure to deploy the Web application on Render from a GitHub repository

Node.js can be used to create a complete Website. The front-end Web application can be connected to a back-end database. GitHub and Render are crucial in the Web deployment process using Node.js.

This session begins with explaining the process of retrieving the connection string of a MongoDB cloud database. It then, progresses to describe the process of creating the Web application in Node.js. The session explains the procedure of uploading the Web application to the GitHub repository. It elaborates on the steps to deploy the application on Render from a GitHub repository. Finally, the session concludes by illustrating the functionality testing of the application in the Web browser.

10.1 Overview

Node.js can be used to create both front-end and back-end applications. In this session, a simple library application is created using Node.js and a

MongoDB cloud database called Library. Figure 10.1 shows the characteristics of the library application.

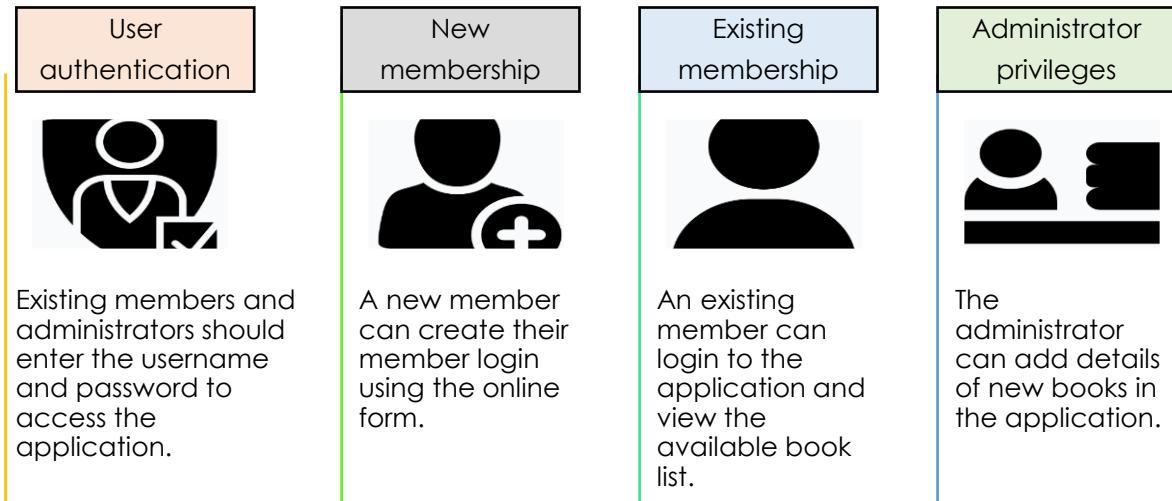
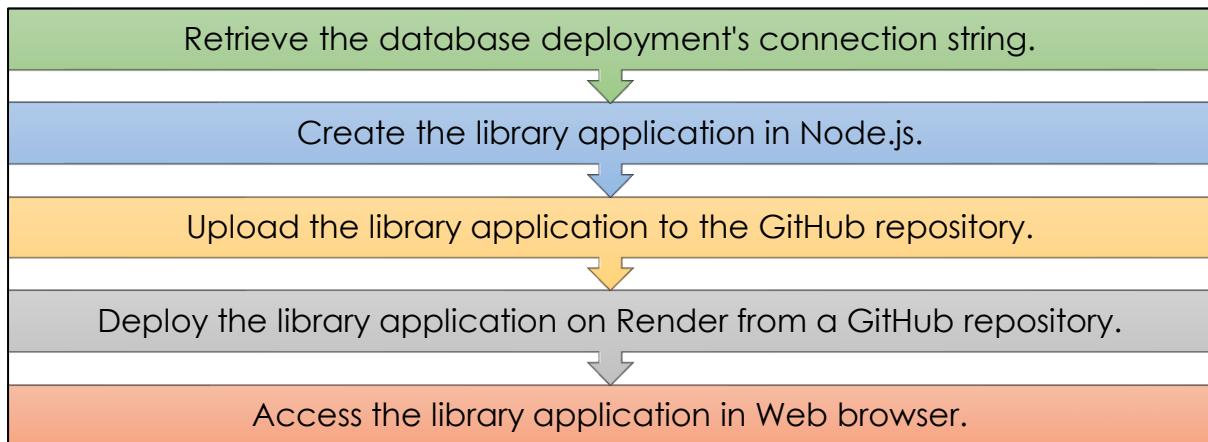


Figure 10.1: Library Application

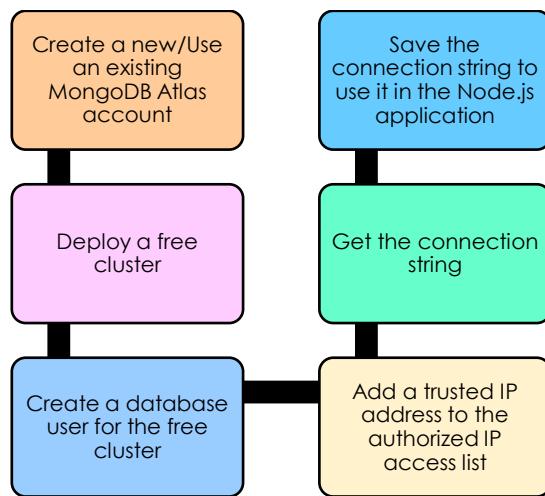
Steps in creating the library application are as follows:



10.2 Retrieving the Connection String from MongoDB Cloud

The prerequisite in creating the library application is to deploy a database in MongoDB cloud and retrieve the database deployment's connection string. The connection string is then used to connect the Library application with the cloud database for storing, retrieving, and processing data.

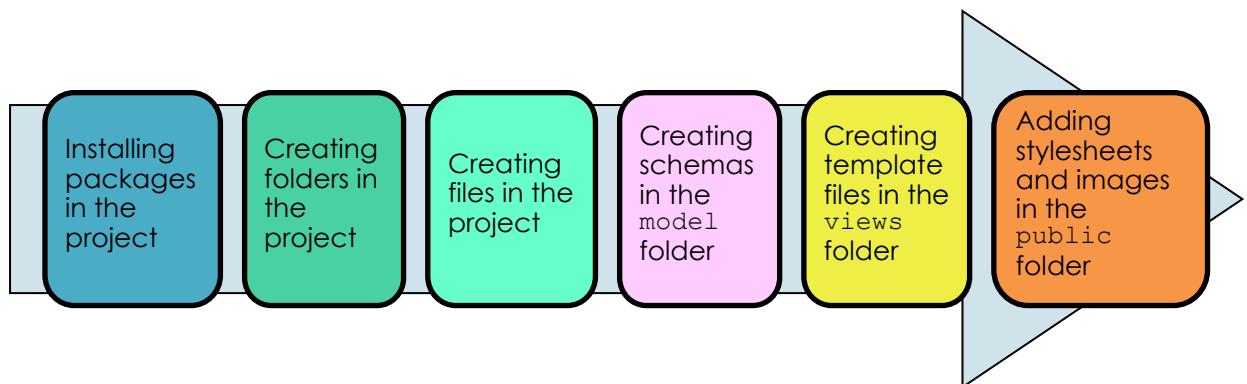
Here are the steps to be executed for achieving the task:



Refer to the Appendix A for a detailed description on how to obtain the connection string from the MongoDB cloud.

10.3 Creating the Library Application in Node.js

The next step is to create the library application in Node.js that involves following actions:



10.3.1 Installing Packages in the Project

To install the required Node.js packages, perform following tasks:

1. Create a directory for the application. To do so:
 - a. Open Command Prompt.
 - b. Navigate to the local drive where Node.js is installed.
 - c. Navigate to the directory to create the application folder.
 - d. Create a directory named `library_application` and change to that directory.

Figure 10.2 displays the directory path.

```
C:\Users\Linda\nodejs>cd library_application
```

Figure 10.2: Creation of a New Directory

2. Create the package.json file to initialize a new Node.js project for the application. Type the command at the prompt and press Enter.

```
npm init -y
```

The command creates the package.json file with the default values in the application's root directory, library_application. Figure 10.3 displays the output of the command.

```
C:\Users\Linda\nodejs\library_application>npm init -y
Wrote to C:\Users\Linda\nodejs\library_application\package.json:

{
  "name": "library_application",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \\"Error: no test specified\\\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Figure 10.3: Creation of a New Node.js Project

The information in Figure 10.4 is stored in the package.json file as specified in the image.

3. Install the required packages in the application's directory.
Table 10.1 lists the purpose of the seven packages that must be installed in the application folder.

Package	Description
passport	Passport is a middleware in Node.js that offers authentication strategies using username-password, Twitter, Facebook, Instagram, and so on
express	A Web application framework that facilitates rapid application development
ejs	A JavaScript library for Embedded JavaScript (EJS) Templating used to generate Hypertext Markup Language (HTML) templates

Package	Description
mongoose	An Object Data Modeling (ODM) library for MongoDB used to facilitate communication with the MongoDB database and perform database operations
body-parser	A Node.js middleware used to parse the body of HTTP POST requests
express-session	A session middleware to manage the session data
passport-local	A package used to implement user authentication process
passport-local-mongoose	A mongoose plug-in that helps in user authentication and session management

Table 10.1: Packages to be Added to the Project

All these packages can be installed using a single command. To install the packages, type the command at the prompt and press Enter:

```
npm install passport express ejs mongoose body-parser
express-session passport-local passport-local-mongoose
```

Figure 10.4 displays the output of the command.

```
C:\Users\Linda\nodejs\library_application>npm install express ejs
mongoose body-parser express-session passport-local passport-local-
mongoose

added 112 packages, and audited 113 packages in 2s

14 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

Figure 10.4: Installation of Packages

After installing the packages, observe the structure of the library_application folder as shown in Figure 10.5.

```
C:\Users\Linda\nodejs\library_application>dir
Volume in drive C is Windows
Volume Serial Number is 66C5-B39B

Directory of C:\Users\Linda\nodejs\library_application

11/30/2023  06:56 PM    <DIR>          .
11/30/2023  06:36 PM    <DIR>          ..
11/30/2023  06:58 PM    <DIR>          node_modules
11/30/2023  06:58 PM            43,889 package-lock.json
11/30/2023  06:58 PM            467 package.json
                           2 File(s)       44,356 bytes
                           3 Dir(s)  275,915,255,808 bytes free
```

Figure 10.5: Listing of the library_application Folder

The node_modules folder and package-lock.json file are automatically created when npm command is executed to install various packages in the project. The package-lock.json file includes the complete details of all the packages installed in the project. It should not be manually edited. Also, the package.json file is automatically updated with the version numbers of the installed packages as shown in Figure 10.6.



Figure 10.6: package.json File With Version Numbers

4. Open the package.json file in VS Code. Edit the file to update the value of the main field, and to include the engines and node fields as shown in Code Snippet 1.

Code Snippet 1:

```
{
  "name": "library_app",
  "version": "1.0.0",
```

```

"description": "",
"main": "app.js",
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1"
},
"keywords": [],
"author": "",
"license": "ISC",
"dependencies": {
  "body-parser": "^1.20.2",
  "ejs": "^3.1.9",
  "express": "^4.18.2",
  "express-session": "^1.17.3",
  "mongoose": "^8.0.0",
  "passport": "^0.6.0",
  "passport-local": "^1.0.0",
  "passport-local-mongoose": "^8.0.0"
},
"engines": {
  "node": "20.7.0"
}
}

```

Save and close the updated package.json file.

10.3.2 Creating Folders in the Project

To organize the project structure, create the required folders and subfolders as shown in Figure 10.7.

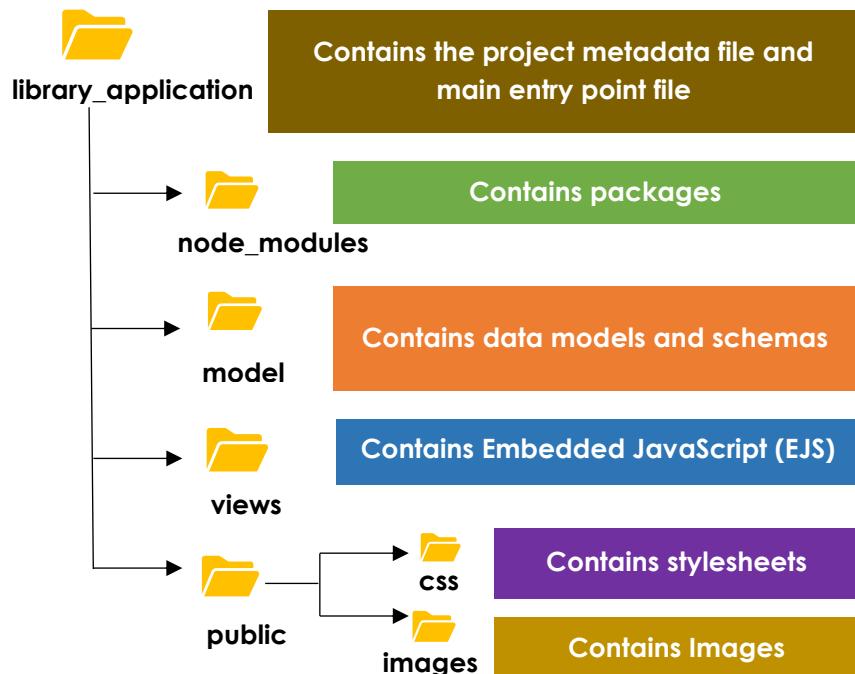


Figure 10.7: Project Folder Structure

Figure 10.8 shows the listing of the `library_application` folder.

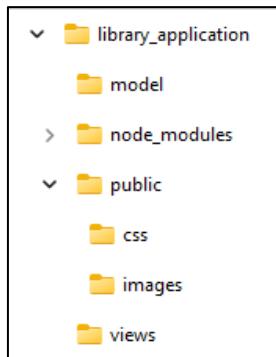


Figure 10.8: Listing of the `library_application` Folder

10.3.3 Creating Files in the Project

To define the user interface design, logic, and functionality of the project, create the required source code files under the specified project folders.

- **Creating Schemas in the `model` Folder**

To define the schemas for the Library database collections, create two models, `User.js` and `Book.js`, in the `model` folder. The schemas represent the structure of the books and users collections that are to be stored in the Library database. Models enable the developers to create, update, and retrieve data in the application.

1. Create `User.js`

Code Snippet 2 shows the code for `User.js`. Save the code with the file name `User.js` in the `model` folder.

Code Snippet 2:

```
const mongoose = require('mongoose')
const Schema = mongoose.Schema
const passportLocalMongoose = require('passport-local-mongoose');
var User = new Schema({
  username: {
    type: String
  },
  password: {
    type: String
  }
})
User.plugin(passportLocalMongoose);
module.exports = mongoose.model('User', User)
```

Operations demonstrated in the `User.js` file are as follows:

- The `mongoose` package is imported and used for interacting with the MongoDB database.
- The statement `const Schema = mongoose.Schema` imports the `Schema` class from the `mongoose` package. The `Schema` class is used to define the structure of the collections that will be stored in the database.
- The statement `var User = new Schema({ })` defines the schema for the `User` model. The schema specifies two fields, `username` and `password`, and their data types.
- The plugin `passportLocalMongoose` is added to the `User` schema for authentication purposes.
- `mongoose.model('User', User)` creates a `model` object called `User` based on the `User` schema. Mongoose creates a collection based on the `User` model. It automatically assigns the name of the collection as `users`, which is a plural, lowercased version of the model's name. The `module.exports` statement exports the `User` model as a module. Exporting the `model` object enables the developers to access the `User` model in any other file or project.

2. Create Book.js

Code Snippet 3 creates a `Book` model for storing book details in the `Library` database. The code creates a `Schema` object called `bookSchema`. It then creates a `model` called `Book` based on the `bookSchema`. As models represent collections, Mongoose creates a collection based on the `Book` model in the `Library` database and automatically assigns the name of the collection as `books`. The code exports the `Book` model to make it accessible in another file or project.

Save the code with the file name `Book.js` in the `model` folder.

Code Snippet 3:

```
const mongoose = require("mongoose");
const bookSchema = new mongoose.Schema({
  Book_id: Number,
  Book_name: String,
  Author_name: String,
  Price: String, // Assuming price is a number
  Age_group: String,
  Book_type: String,
});
const Book = mongoose.model("Book", bookSchema);
module.exports = Book;
```

After creating the schema files, the structure of the `model` folder appears as shown in Figure 10.9.

Name	Date modified	Type	Size
Book	08-11-2023 20:34	JS File	1 KB
User	08-11-2023 20:11	JS File	1 KB

Figure 10.9: Listing of the `model` Folder

- **Creating Template Files in the `views` Folder**

To render HTML pages for the application, create the template files or `.ejs` files in the `views` folder. The `.ejs` files are used to provide the HTML markups by embedding the required JavaScript in the main `Node.js` application file, `app.js`.

1. Create `register.ejs`

Code Snippet 4 demonstrates the HTML code for the registration form for new library members. The completed form, when submitted, will be delivered to the `/register` endpoint for further processing. Save the code with the file name `register.ejs` in the `views` folder.

Code Snippet 4:

```
<!DOCTYPE html>
<html>
<head>
  <link rel="stylesheet" type="text/css"
  href="/css/register.css">
  <title>Sign up for Library Member</title>
</head>
<body>
  <div class="container">
    
    <h1>Sign up form for Library Member</h1>
    <form action="/register" method="POST" class="form-
table">
      <table>
        <tr>
          <td><label for="username">Username</label></td>
          <td><input type="text" name="username" id="username"
placeholder="Username" autocomplete="off"></td>
        </tr>
        <tr>
          <td><label for="password">Password</label></td>
```

```

<td><input type="password" name="password"
id="password" placeholder="Password"></td>
</tr>
<tr>
<td></td>
<td><button>Sign-UP</button></td>
</tr>
</table>
</form>
<p><a href="/logout">Home Page</a></p>
</div>
</body>
</html>

```

2. Create login.ejs

Code Snippet 5 demonstrates the HTML code for the application's login form. The completed form, when submitted, will be delivered to the /login endpoint for further processing. Save the code with the file name login.ejs in the views folder.

Code Snippet 5:

```

<!DOCTYPE html>
<html>
<head>
<link rel="stylesheet" type="text/css"
href="/css/register.css">
<title>Member Login</title>
</head>
<body>
<div class="container">

<h1>Member login</h1>
<form action="/login" method="POST">
<table>
<tr>
<td><label for="username">Username</label></td>
<td><input type="text" name="username" id="username"
placeholder="Username" autocomplete="off"></td>
</tr>
<tr>
<td><label for="password">Password</label></td>
<td><input type="password" name="password"
id="password" placeholder="Password"></td>
</tr>
<tr>
<td></td>
<td><button>Login</button></td>
</tr>

```

```

</table>
</form>
    <p><a href="/logout">Home Page</a></p>
</div>
</body>
</html>

```

3. Create home.ejs

Code Snippet 6 demonstrates the HTML code for the application's home page. On the home page of the application, the user can:

- Sign up as a new member
- Login as an existing member
- Login as an administrator

Save the code with the file name `home.ejs` in the `views` folder.

Code Snippet 6:

```

<!DOCTYPE html>
<html>
<head>
    <link rel="stylesheet" type="text/css"
    href="/css/home.css">
    <title>Member Login</title>
</head>
<body>
<div class="container">
<div class="right">
<h1>Library Home page</h1>
<table align="center" style="width:50%">
<tr>
    <td align="center"><a href="/register">Sign_up for
    Member User</a></td>
    </tr>
<tr>
    <td align="center"><a href="/login">Member
    Login</a></td>
    </tr>
<tr>
    <td align="center"><a href="/admin">Administrator
    Login</a></td>
    </tr>
</table>
</div>
<div class="left">
    
</div>

```

```
</div>
</body>
</html>
```

4. Create admin-login.ejs

Code Snippet 7 demonstrates the HTML code for the administrator's login page. Administrators can enter their username and password in the login form. The completed form, when submitted, will be sent to the /admin-login endpoint for further processing.

Save the code with the file name `admin-login.ejs` in the `views` folder.

Code Snippet 7:

```
<!DOCTYPE html>
<html>
<head>
<link rel="stylesheet" type="text/css"
href="/css/register.css">
<title>Admin Login</title>
</head>
<body>
<div class="container">
    
    <h1>Admin Login</h1>
    <form action="/admin-login" method="post">
        <table>
            <tr>
                <td><label for="username">Username</label></td>
                <td><input type="text" name="username" id="username"
placeholder="Username" autocomplete="off"></td>
            </tr>
            <tr>
                <td><label for="password">Password</label></td>
                <td><input type="password" name="password"
id="password" placeholder="Password"></td>
            </tr>
            <tr>
                <td></td>
                <td><button>Login</button></td>
            </tr>
        </table>
    </form>
    <p><a href="/logout">Home Page</a></p>
    </div>
</body>
</html>
```

5. Create admin-dashboard.ejs

Code Snippet 8 demonstrates the HTML code for the admin dashboard page. On this page, the administrators can add new book details to the library's books collection. On clicking the **Add Book** button, the completed form will be sent to the /admin-dashboard/add-book endpoint for further processing.

Save the code with the file name `admin-dashboard.ejs` in the `views` folder.

Code Snippet 8:

```
<!DOCTYPE html>
<html>
<head>
  <link rel="stylesheet" type="text/css"
  href="/css/admindash.css">
  <title>Admin Dashboard</title>
</head>
<body>
  <div class="container">
    <form action="/admin-dashboard/add-book"
method="post">
      <table> <tr>
        <td></td>
        <td><h1>Create Book Detail</h1></td>
      </tr>
      <tr>
        <td><label for="Book_id">BookID:</label></td>
        <td><input type="number" name="Book_id" required
autocomplete="off"></td>
      </tr>
      <tr>
        <td><label for="Book_name">Book Name:</label></td>
        <td><input type="text" name="Book_name" required
autocomplete="off"></td>
      </tr>
      <tr>
        <td><label for="Author_name">Author
Name:</label></td>
        <td><input type="text" name="Author_name" required
autocomplete="off"></td>
      </tr>
      <tr>
        <td><label for="Price">Price:</label></td>
        <td><input type="text" name="Price" required
autocomplete="off"></td>
      </tr>
```

```

<tr>
<td><label for="Age_group">Age Group:</label></td>
<td><input type="text" name="Age_group" required
autocomplete="off"></td>
</tr>
<tr>
<td><label for="Book_type">Book Type:</label></td>
<td><input type="text" name="Book_type" required
autocomplete="off"></td>
</tr>
</table>
<div class="center-button-container">
<button class="center-button" type="submit">Add
Book</button>
</div>
</form>
</div>
<p><a href="/logout">Logout</a></p>
</body>
</html>

```

6. Create booklist.ejs

Code Snippet 9 demonstrates the HTML code of a view that displays the list of books retrieved from the books collection.

The use of embedded JavaScript (`<% %>`) shows that:

- Data is retrieved and displayed dynamically by a server-side application.
- Book data is injected into the HTML template at runtime.

Save the code with the file name `booklist.ejs` in the `views` folder.

Code Snippet 9:

```

<!DOCTYPE html>
<html>
<head>
    <link rel="stylesheet" type="text/css"
href="/css/bookstyle.css">
    <title>List of Books</title>
</head>
<body>
<div class="content">
<div class="container">
    <div class="column1">
        <h1>List of Books</h1>
        <table class="center">
<tr>

```

```

<th>Book ID</th>
<th>Book </th>
<th>Author </th>
<th>Price</th>
<th>Age Group</th>
<th>Book Type</th>
</tr>
<% books.forEach(function(book) { %>
  <tr>
    <td><%= book.Book_id %></td>
    <td><%= book.Book_name %></td>
    <td><%= book.Author_name %></td>
    <td><%= book.Price %></td>
    <td> <%= book.Age_group %></td>
    <td> <%= book.Book_type %></td>
  </tr>
<% }); %>
</table>
</div>
</div>
<p><a href="/logout">Logout</a></p>
</body>
</html>

```

7. Create error.ejs

Code Snippet 10 demonstrates the HTML code for the error page of member logins.

The error page:

- Displays error messages to members.
- Provides an option to return to the login page.

Save the code with the file name `error.ejs` in the `views` folder.

Code Snippet 10:

```

<!-- error.ejs -->
<!DOCTYPE html>
<html>
<head>
  <link rel="stylesheet" type="text/css"
  href="/css/error.css">
  <title>Error</title>
</head>
<body>
<div class="container">
  <h1>Error</h1>
  <p><%= errorMessage %></p>

```

```
<p><a href="/login">Go back to login</a></p>

</div>
</body>
</html>
```

8. Create admin-error.ejs

Code Snippet 11 demonstrates the HTML code for the error page of administrator login. The error page:

- Displays error messages to administrators
- Provides an option to return to the admin login page

Save the code with the file name `admin-error.ejs` in the `views` folder.

Code Snippet 11:

```
<!-- error.ejs -->
<!DOCTYPE html>
<html>
<head>
<link rel="stylesheet" type="text/css"
href="/css/error.css">
<title>Error</title>
</head>
<body>
<div class="container">

<h1>Error</h1>
<p><%= errorMessage %></p>
<p><a href="/admin">Go back to admin login</a></p>
</div>
</body>
</html>
```

After the creation of `.ejs` files, find the structure of the `views` folder as shown in Figure 10.10.

Name	Date modified	Type	Size
admin-dashboard	08-11-2023 21:47	EJS File	2 KB
admin-error	08-11-2023 22:05	EJS File	1 KB
admin-login	08-11-2023 21:43	EJS File	1 KB
booklist	08-11-2023 21:53	EJS File	1 KB
error	08-11-2023 22:00	EJS File	1 KB
home	08-11-2023 22:07	EJS File	1 KB
login	08-11-2023 21:39	EJS File	2 KB
register	08-11-2023 21:40	EJS File	2 KB

Figure 10.10: Listing of the `views` Folder

- **Adding Stylesheets and Images in the `public` Folder**

Next, include the required `.css` and image files in the `public\css` and `public\images` folders, respectively.

1. Create `register.css`

Code Snippet 12 shows the stylesheet for the new member registration form. Save the code with the file name `register.css` in the `public\css` folder.

Code Snippet 12:

```
/* register.css */
body {
    margin: 0;
    padding: 0;
    font-family: Arial, sans-serif;
    background-color: lightblue; /* Background color for
the entire page */
}
h1 {
    border: 2px #eee solid;
    color: brown;
    text-align: center;
    padding: 10px;
    margin: 0; /* Add this line to remove any default
margin */
}
.container {
    display: flex;
    flex-direction: column; /* Change to a column layout
*/
    justify-content: center;
```

```

align-items: center;
height: 100vh;
text-align: center; /* Center-align the content */
}
.form-table {
background-color: #fff; /* Background color for the
form container */
padding: 20px;
border-radius: 8px;
box-shadow: 0 4px 6px rgba(0, 0, 0, 0.1);
}
table {
border-collapse: collapse;
width: 100%;
}
table td {
padding: 10px;
}

input {
width: 100%;
padding: 10px;
margin: 5px 0;
border: 1px solid #ccc;
border-radius: 4px;
}

button {
width: 100%;
padding: 10px;
background-color: #007BFF; /* Button background color
*/
color: #fff; /* Button text color */
border: none;
border-radius: 4px;
cursor: pointer;
}

button:hover {
background-color: #0056b3; /* Button background color
on hover */
}

```

2. Create `home.css`

Code Snippet 13 shows the stylesheet for the application's home page. Save the code with the file name `home.css` in the `public\css` folder.

Code Snippet 13:

```
table, th, td {  
    border: 1px solid;  
    font-size: 30px;  
}  
  
body {  
    background-color: lightblue;  
}  
h1 {  
    border: 2px #eee solid;  
    color: brown;  
    text-align: center;  
    padding: 10px;  
}  
.center-image {  
    display: block;  
    margin: 0 auto;  
    max-width: 100%;  
    height: auto;  
}  
  
.container {  
    height: 75%;  
    overflow: hidden;  
}  
.right {  
    width: 800px;  
    float: right;  
    margin-top: 100px;  
}  
.left {  
    margin-top: 125px;  
    width: auto;  
    overflow: hidden;  
}
```

3. Create error.css

Code Snippet 14 shows the stylesheet for the application's error pages. Save the code with the file name `error.css` in the `public\css` folder.

Code Snippet 14:

```
.container {  
    display: flex;  
    flex-direction: column;  
    align-items: center;  
    width: 100%; /* Set the container width to 100% */
```

```
max-width: 1000px; /* Optionally, set a maximum width  
for the container */  
margin: 0 auto; /* Center the container horizontally */  
}  
}  
body {  
margin: 0;  
padding: 0;  
font-family: Arial, sans-serif;  
background-color: lightblue;  
display: flex;  
flex-direction: column;  
align-items: center;  
justify-content: flex-start;  
min-height: 100vh;  
}
```

4. Create bookstyle.css

Code Snippet 15 shows the stylesheet for displaying the book details. Save the code with the file name `bookstyle.css` in the `public\css` folder.

Code Snippet 15:

```
container {  
display: grid;  
grid-template-columns: 100px 150px 200px;  
text-align: center;  
}  
.column1 {  
float: left;  
width: 90%;  
padding: 10px;  
height: auto;  
background-color: lightblue;  
}  
body {  
background-color: grey;  
}  
.content {  
width: 100%;  
position: absolute;  
top: 10%;  
}  
.center {  
display: table;  
height: 100%;  
width: 100%;  
}  
h2 {
```

```
border: 2px #eee solid;
color: Red;
text-align: center;
padding: 10px;
}
```

5. Create admindash.css

Code Snippet 16 shows the stylesheet for administrator dashboard. Save the code with the file name `admindash.css` in the `public\css` folder.

Code Snippet 16:

```
/* admin-dashboard.css */
body {
    margin: 0;
    padding: 0;
    font-family: Arial, sans-serif;
    background-color: lightblue;
    display: flex;
    flex-direction: column;
    align-items: center;
    justify-content: center; /* Center both horizontally
and vertically */
    min-height: 100vh;
}

.container {
    display: flex;
    flex-direction: column;
    align-items: center;
    width: 100%;
    max-width: 1000px;
    margin: 0 auto;
}

h1 {
    border: 2px #eee solid;
    color: brown;
    text-align: center;
    padding: 10px;
    margin: 0;
}

form {
    width: 60%;
}

table {
    border-collapse: collapse;
```

```

width: 100%;
margin: 20px auto;
}

table td {
padding: 10px;
}

label {
font-weight: bold;
display: block;
text-align: left;
}

input {
width: 100%;
padding: 10px;
margin: 5px 0;
border: 1px solid #ccc;
border-radius: 4px;
}

.center-button-container {
text-align: center; /* Center horizontally */
margin-top: 20px; /* Adjust the top margin as required */
}

.center-button {
width: 150px; /* Adjust the width as required */
height: 50px; /* Adjust the height as required */
background-color: #007BFF; /* Blue background color */
color: #fff;
border: none;
cursor: pointer;
}

```

6. Add Images

Download the image file, book1.jpeg, from https://commons.wikimedia.org/wiki/File:Books_HD%288314929977%29.jpg and upload it to the public\images folder.

After the creation of .css files, find the structure of the public\css folder as shown in Figure 10.11.

Name	Date modified	Type	Size
admindash	09-11-2023 14:48	Cascading Sty...	2 KB
bookstyle	09-11-2023 14:36	Cascading Sty...	1 KB
error	09-11-2023 14:33	Cascading Sty...	1 KB
home	09-11-2023 14:53	Cascading Sty...	1 KB
register	09-11-2023 14:53	Cascading Sty...	2 KB

Figure 10.11: Listing of the public\css Folder

- **Creating the app.js File**

Finally, create the main Node.js application file, app.js. This file acts as the main entry point in the library application. In this file, the core logic of the application is handled. Routes and middleware are also defined.

Consider the code in Code Snippet 17 for app.js file.

Code Snippet 17:

```
const express = require("express");
const mongoose = require("mongoose");
const passport = require("passport");
const bodyParser = require("body-parser");
const LocalStrategy = require("passport-local");
const passportLocalMongoose = require("passport-local-mongoose");
const User = require("./model/User");
const Book = require("./model/Book");
const app = express();
mongoose.connect(`mongodb+srv://lindalarrissa91:linda91@cluster0.gktucwf.mongodb.net/Library?retryWrites=true&w=majority`);
app.set("view engine", "ejs");
app.use(bodyParser.urlencoded({ extended: true }));
app.use(
  require("express-session")({
    secret: "Rio is a dog",
    resave: false,
    saveUninitialized: false,
  })
);
app.use(passport.initialize());
app.use(passport.session());
app.use(express.static(__dirname + "/public"));
```

```
// Add the admin local strategy
passport.use(
  "admin-local",
  new LocalStrategy(function (username, password, done) {
    if (username === "Admin" && password === "12345") {
      return done(null, { username: "Aptech" });
    }
    return done(null, false, { message: "Incorrect admin username or password" });
  })
);
passport.serializeUser(function (user, done) {
  // Here, you might want to serialize user data if required (e.g., user.id)
  done(null, user);
});

passport.deserializeUser(function (user, done) {
  // Implement deserialization logic here if required
  done(null, user);
});
// Showing home page
app.get("/", function (req, res) {
  res.render("home");
});
// Showing register form
app.get("/register", function (req, res) {
  res.render("register");
});

// Handling user signup
app.post("/register", async (req, res) => {
  const user = await User.create({
    username: req.body.username,
    password: req.body.password,
  });
  res.redirect("/");
});

// Showing login form
app.get("/login", function (req, res) {
  res.render("login");
});

// Handling user login
app.post("/login", async function (req, res) {
  try {
    const user = await User.findOne({ username: req.body.username });
    if (user) {
      const result = req.body.password === user.password;
```

```
if (result) {
  const books = await Book.find({});
  res.render("booklist", { books: books });
} else {
  res.render("error", { errorMessage: "Password doesn't match" });
}
} else {
  res.render("error", { errorMessage: "User doesn't exist" });
}
} catch (error) {
  res.render("error", { errorMessage: "An error occurred" });
}
});

// Admin login route
app.get("/admin", function (req, res) {
  res.render("admin-login");
});
// Admin login form
app.post(
  "/admin-login",
  passport.authenticate("admin-local", {
    successRedirect: "/admin-dashboard",
    failureRedirect: "/admin-error",
  })
);
// Admin error route
app.get("/admin-error", function (req, res) {
  res.render("admin-error", { errorMessage: "Incorrect admin username or password" });
});
// Admin dashboard route
app.get("/admin-dashboard", function (req, res) {
  if (req.isAuthenticated()) {
    res.render("admin-dashboard");
  } else {
    res.redirect("/admin");
  }
});
// Get book details from the form
app.post("/admin-dashboard/add-book", function (req, res) {
  if (req.isAuthenticated()) {
    const bookDetails = {
      Book_id: req.body.Book_id,
      Book_name: req.body.Book_name,
      Author_name: req.body.Author_name,
      Price: req.body.Price,
      Age_group: req.body.Age_group,
      Book_type: req.body.Book_type,
    };
  }
});
```

```

// Create a new book in the "books" collection
Book.create(bookDetails)
  .then((newBook) => {
    console.log("Book added successfully:", newBook);
    res.redirect("/admin-dashboard");
  })
  .catch((err) => {
    console.error("Failed to add the book:", err);
    res.status(500).json({ error: "Failed to add the book" });
  });
} else {
  res.redirect("/admin");
}
});

// Handling user logout
app.get("/logout", function (req, res) {
  req.logout(function (err) {
    if (err) {
      return next(err);
    }
    res.redirect("/");
  });
});
function isLoggedIn(req, res, next) {
  if (req.isAuthenticated()) return next();
  res.redirect("/login");
}
const port = process.env.PORT || 3000;
app.listen(port, function () {
  console.log("Server Has Started!");
});

```

Save the code with the file name `app.js` in the `library_application` folder. The operations demonstrated in the `app.js` file are as follows:

- **Importing packages:** Various Node.js packages such as `express`, `mongoose`, `passport`, `body-parser`, `passport-local`, and `passport-local-mongoose` are imported.
- **Importing schemas:** Two data models namely, `User` and `Book` from the `local model` directory are imported.
- **Creating express application:** The statement `const app = express()` creates the `express` application.
- **Connecting the database:** The statement `mongoose.connect(`mongodb+srv://lindalarrissa91:linda91@cluster0.gktucwf.mongodb.net/Library?retryWrites=true&w=majority`)` connects the application to the MongoDB database, `Library`, using the connection string.
- **Setting the view engine:** The statement `app.set("view engine", "ejs")` is used to set the view engine to EJS.

- **Setting up the middleware:** Packages such as body-parser, express-session, and passport-authenticate are used to set up the middleware.
- **Defining the local strategy:** A local strategy is defined for authenticating the administrators. If the username Admin and password 12345 match, the function returns an administrator object. Else, it returns an error message.
- **Serializing and deserializing user functions:** Functions are defined for serializing and deserializing user objects. These functions are used to manage user sessions.
- **Defining routes:** The application defines several routes for handling HTTP requests. The routes are:
 - /: This is the home page route.
 - /register: This is for registering a new member.
 - /login: This is for authenticating an existing member.
 - /admin: This is the administrator login route.
 - /admin-login: This is for handling administrator username and password.
 - /admin-error: This is for displaying administrator login errors.
 - /admin-dashboard: This is the administrator dashboard route.
 - /admin-dashboard/add-book: This is for handling the addition of a new book by the administrator.
 - /logout: This is the member/administrator logout route.
- **Handling new member registration:** On the /register route, new member registration is handled with the provided username and password. The username and password are added to the users collection in the Library database.
- **Handling existing member login:** On the /login route, the existing member login is handled. It checks if the member exists and whether the password matches. If successful, it retrieves a list of books from the database and displays the booklist from the views folder. Else, it displays an error message.
- **Handling administrator login and dashboard:** The administrator login is handled with a separate strategy using the administrator username and password. If the authentication is successful, the administrator is redirected to the admin dashboard. If not, an error message is displayed.
- **Managing admin dashboard and adding books:** The admin dashboard route is protected by authentication. Administrators can add new books by submitting a form. These books are added to the books collection in the Library database and the administrator is redirected to the dashboard.
- **Managing member/administrator logout process:** The member can log out by accessing the /logout route, which clears the session and redirects them to the home page.

- **Listening for requests:** The application listens on a specified port, either from the environment variable or port 3000. A message is logged when the server starts.

This completes the coding for the application.

10.4 Uploading the Library Application to the GitHub Repository

The developed application must be loaded to the Internet for it to be accessible globally. For this purpose, the GitHub repository is used. It can store codes, files, version histories, and so on, of applications. To upload the library application to the GitHub repository, create an exclusive repository space in GitHub.

Refer to Appendix B for a detailed description on how to:

- Create a repository in GitHub account
- Upload the application resources to the repository

10.5 Deploying the Library Application on Render from a GitHub Repository

Render is a cloud computing platform that helps developers to deploy their applications on cloud. The library application is uploaded to the GitHub repository. To deploy the application on Render from the GitHub repository, perform the given steps:

1. To access the **Render** platform, in the browser, navigate to the URL <https://render.com/>, and click **GET STARTED FOR FREE** as shown in Figure 10.12.

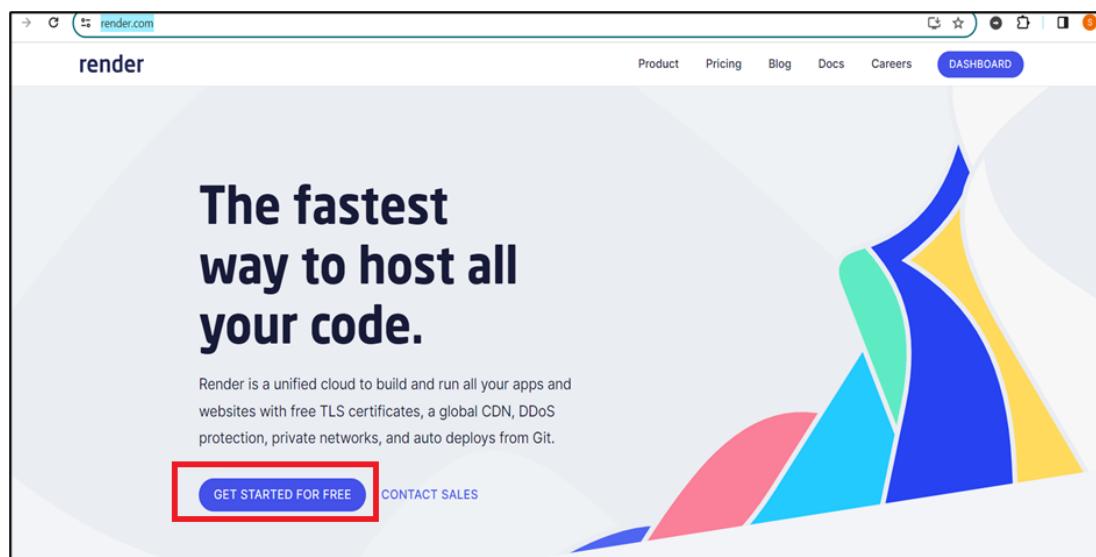


Figure 10.12: Home Page of Render

2. To sign in with the GitHub account, on the **Sign up for Render** page, type the given details, and click **Sign in** as shown in Figure 10.13.

Here, for example, a GitHub account is used to sign into Render with the email as Linda.larrissa91@gmail.com and password as **linda91@**.

- In the **Email** box, type the Github account id.
- In the **Password** box, type the password.

If an existing account is used to sign in to Render, then provide the email id, password and click **Sign in**.

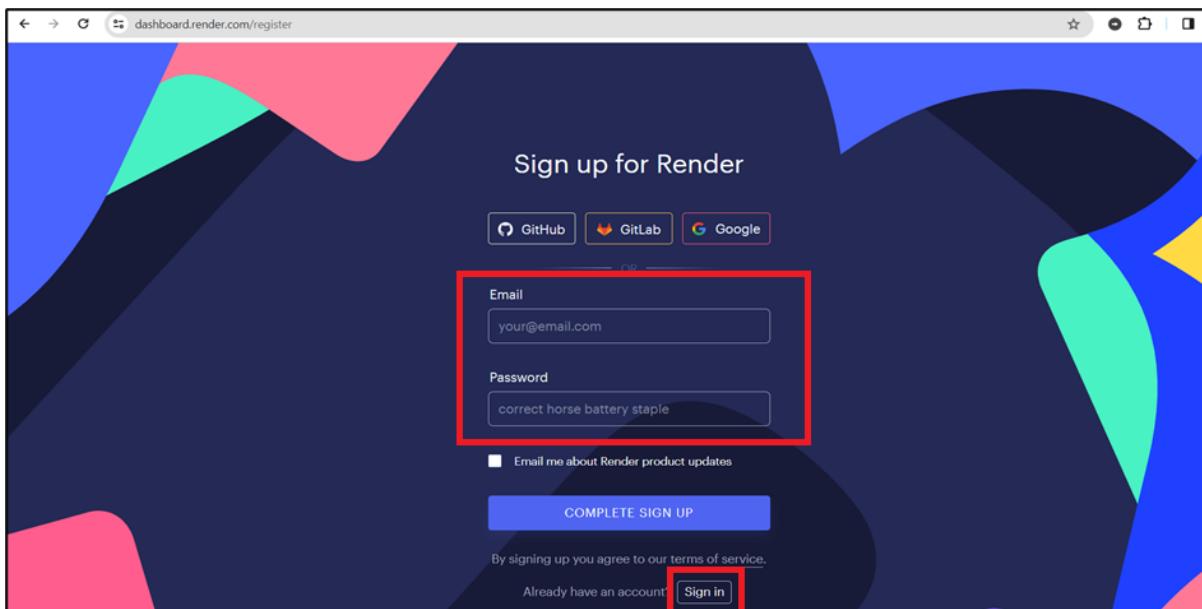


Figure 10.13: Sign up for Render Page

3. To complete the sign up process, on the **Sign up for Render** page, click **COMPLETE SIGN UP** as shown in Figure 10.14.

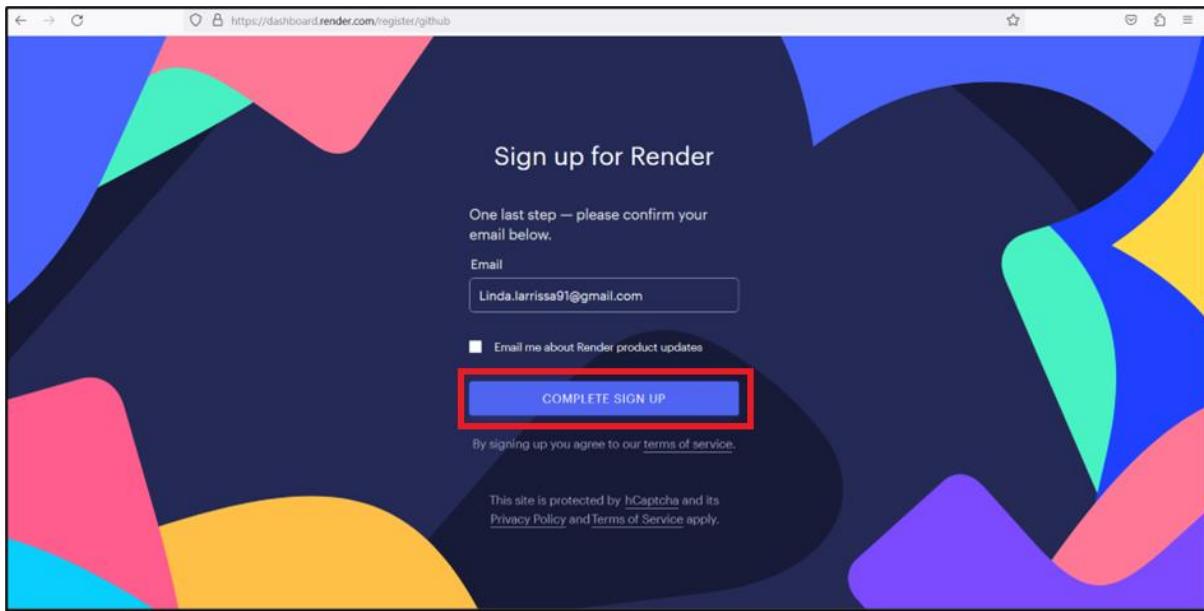


Figure 10.14: Sign up for Render Page – Email Confirmation

4. To activate the **Render** account, click the link in the email as instructed on the **Almost there** page as shown in Figure 10.15.

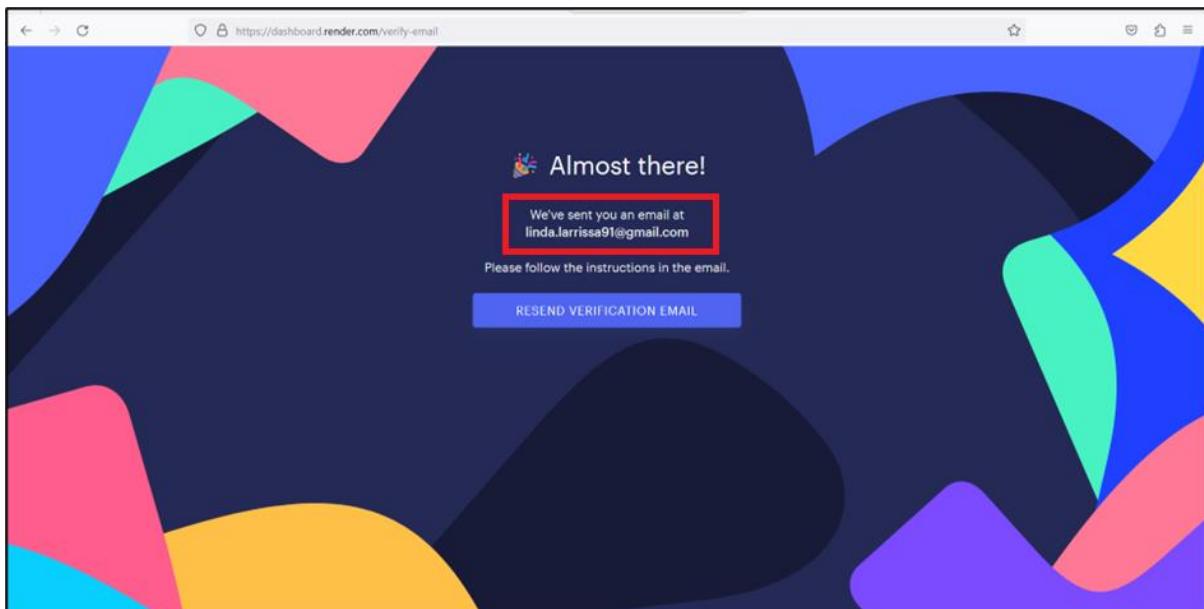


Figure 10.15: Almost there Page

5. To continue on **Render**, on the **Tell us about yourself** page, fill in the necessary details, and click **CONTINUE TO RENDER** as shown in Figure 10.16.

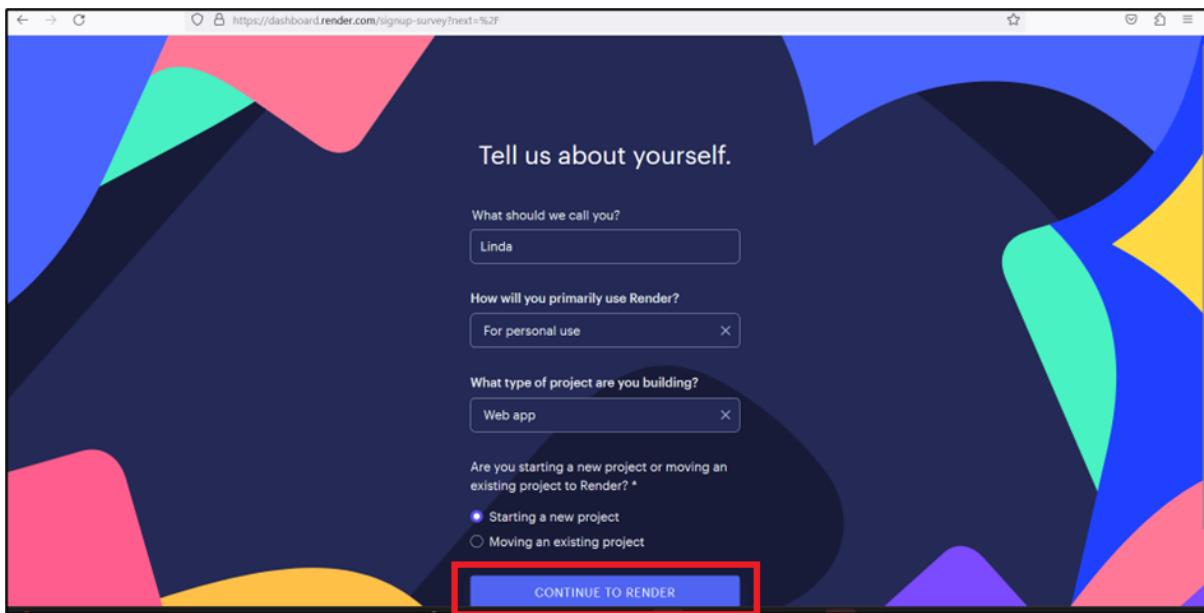


Figure 10.16: Tell us about yourself Page

6. To create a new Web service, on the **Render Dashboard** page, under **Web Services**, click **New Web Service** as shown in Figure 10.17.

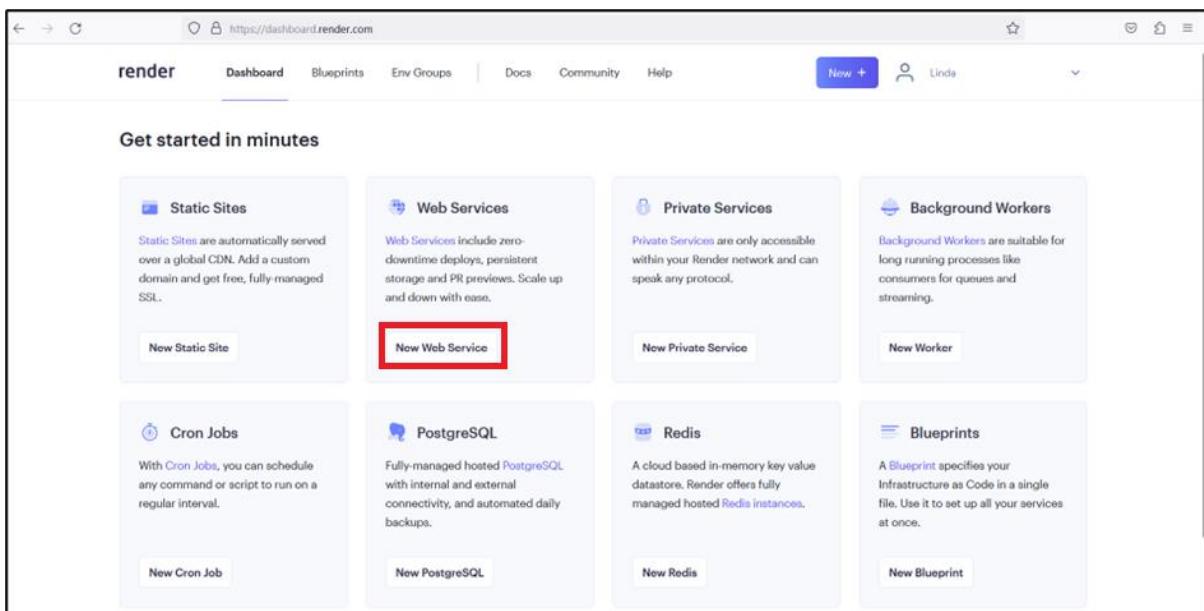


Figure 10.17: Render Dashboard Page

7. To deploy the Web service, on the **Create a new Web Service** page, click **Build and deploy from a Git repository**, and click **Next** as shown in Figure 10.18.

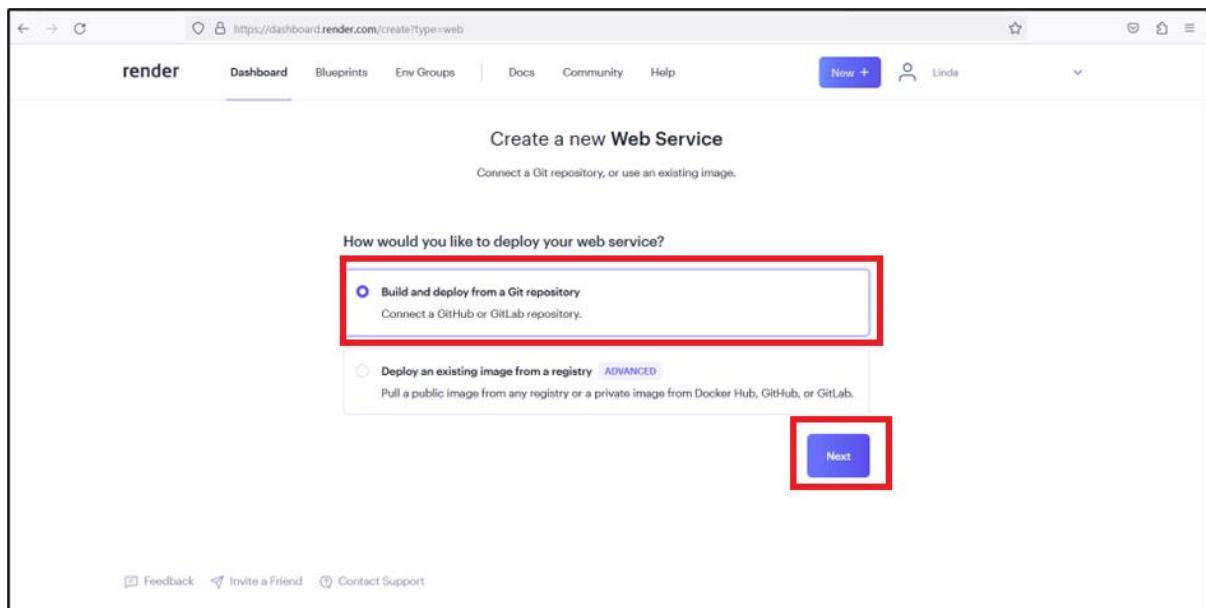


Figure 10.18: Create a new Web Service Page

8. To connect a repository, on the **Connect a repository** page, click **Connect GitHub** as shown in Figure 10.19.

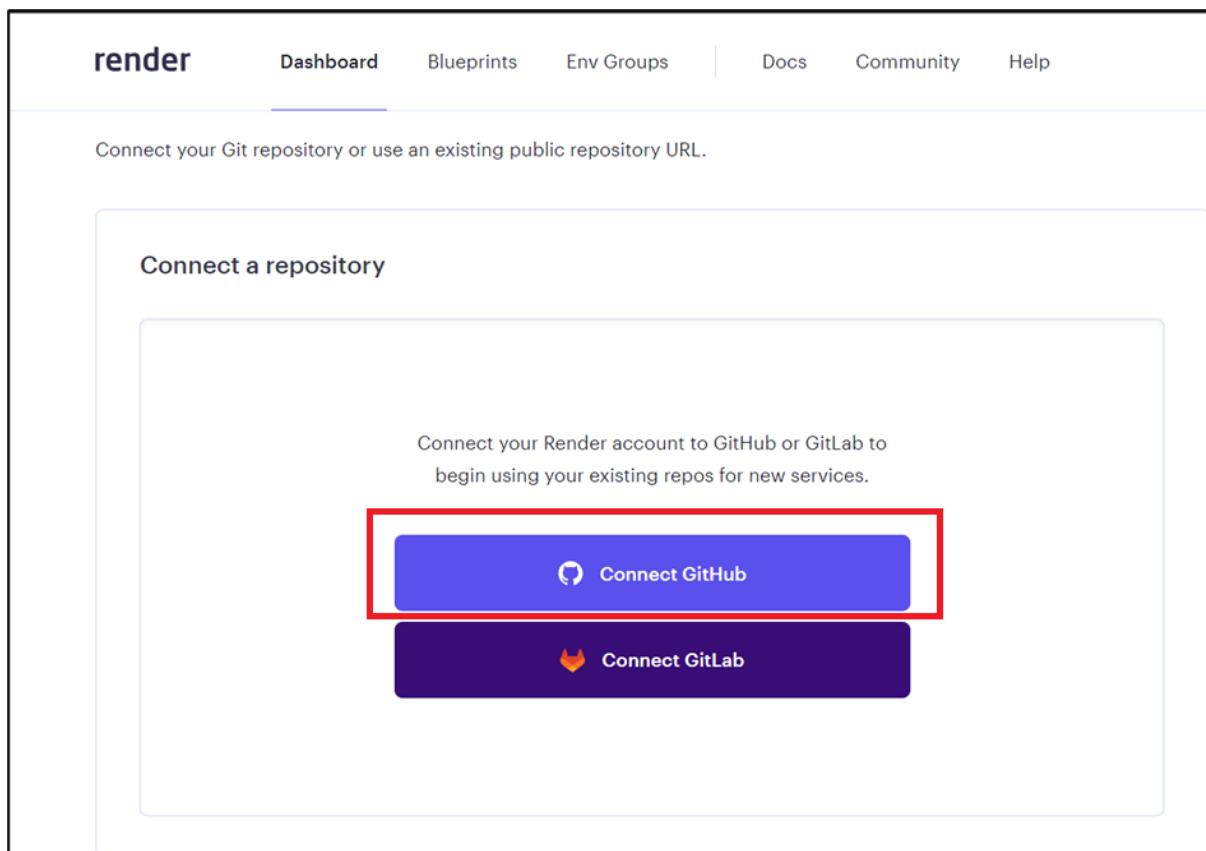


Figure 10.19: Connect a Repository Page

9. To authorize Render, on the **Render by Render would like permission to** page, click **Authorize Render** as shown in Figure 10.20.

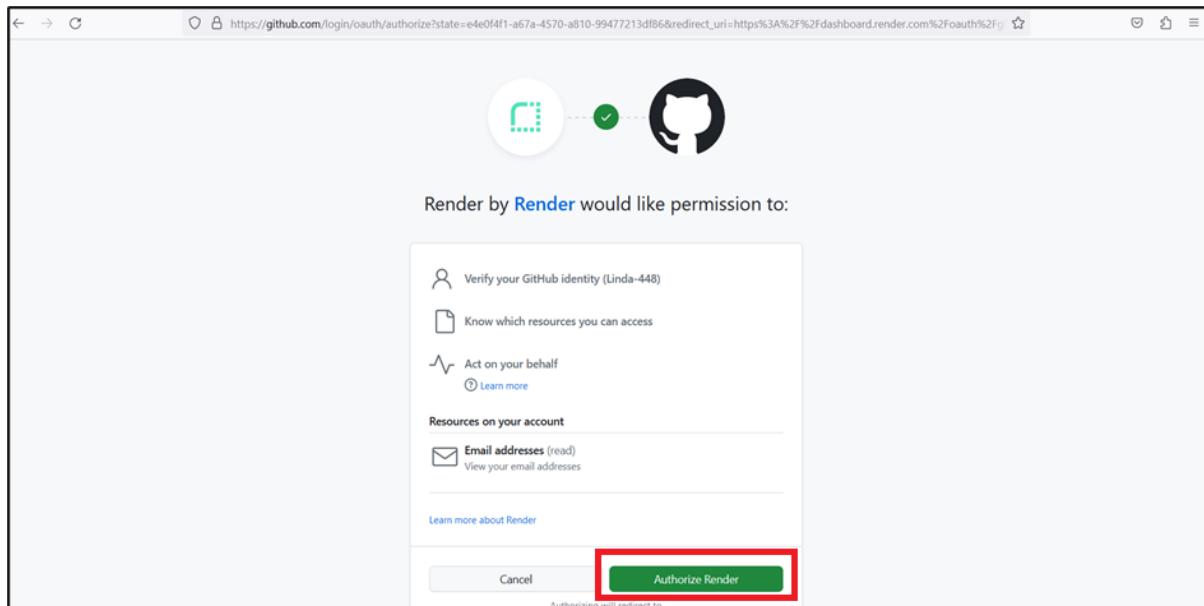


Figure 10.20: Render by Render would like permission to Page

10. To install **Render** on the GitHub account, on the **Install Render** page, click **Install** as shown in Figure 10.21.

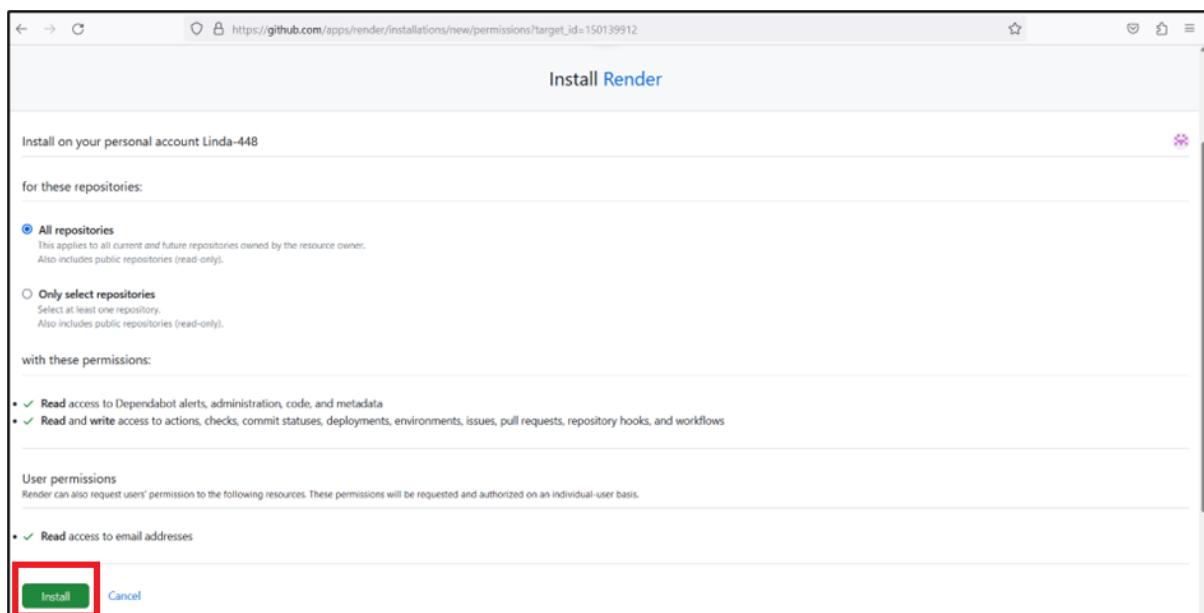


Figure 10.21: Install Render Page

11. To connect the repository, on the **Create a new Web Service** page, click **Connect** as shown in Figure 10.22.

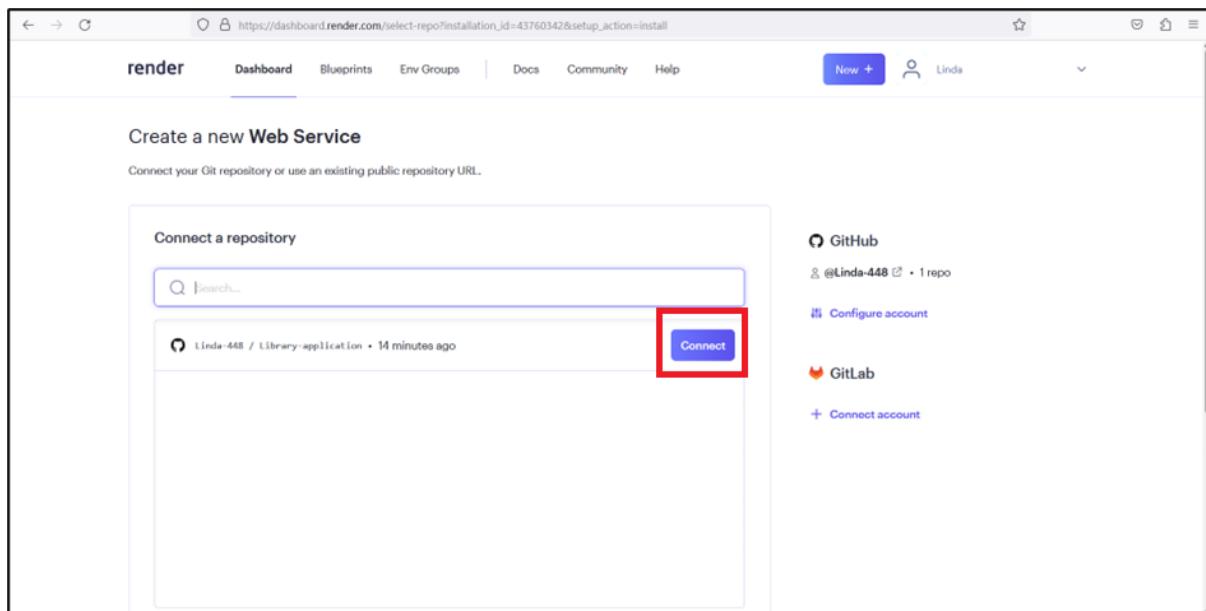


Figure 10.22: Connect the Repository

12. To provide a unique name for the Web service, on the **Render Dashboard** page, in the **Name** box, type **Library-app** as shown in Figure 10.23.

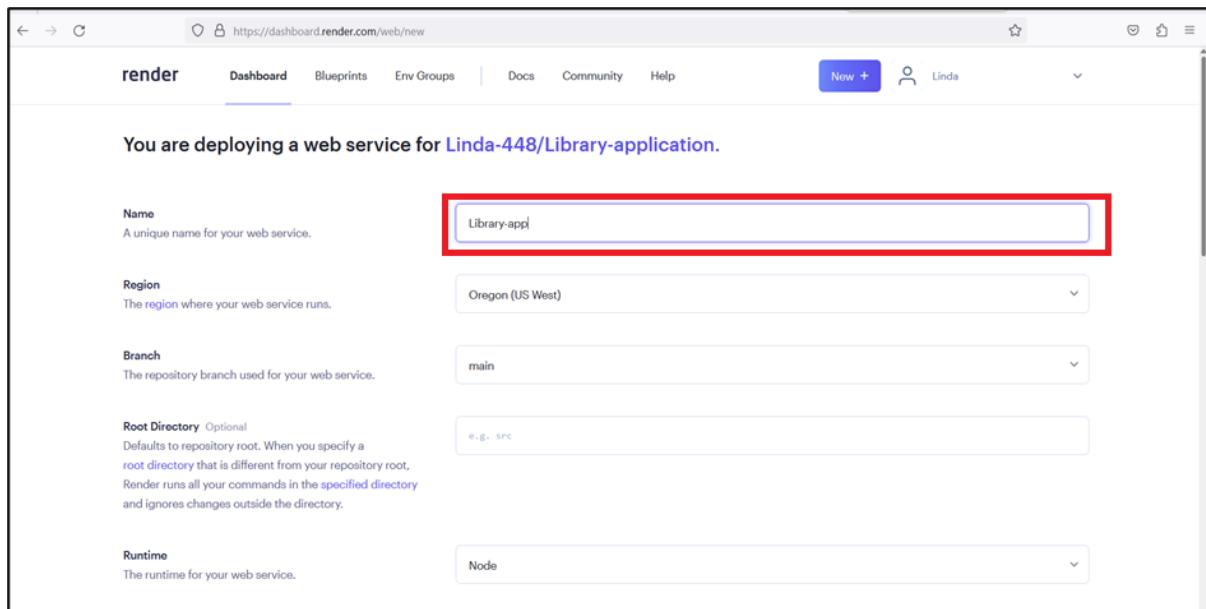


Figure 10.23: Web Service Name

13. To provide the start command, on the **Render Dashboard** page, scroll down and in the **Start Command** box, type **node app.js** as shown in Figure 10.24.

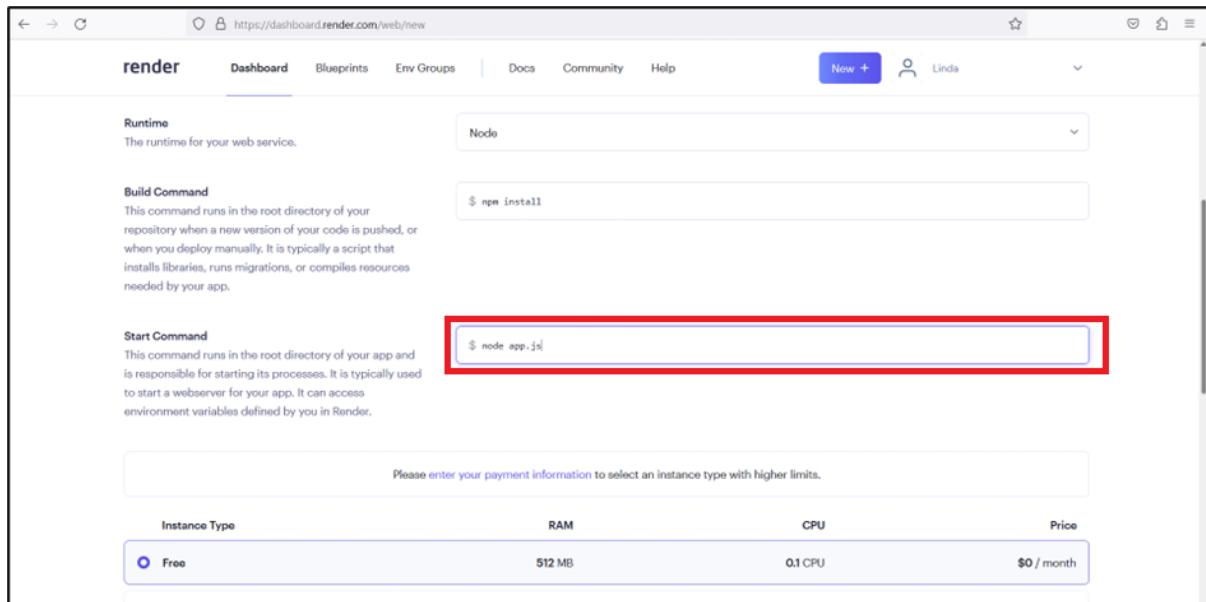


Figure 10.24: Start Command for Web Service

14. To create the Web service, on the **Render Dashboard** page, scroll down, and click **Create Web Service** as shown in Figure 10.25.

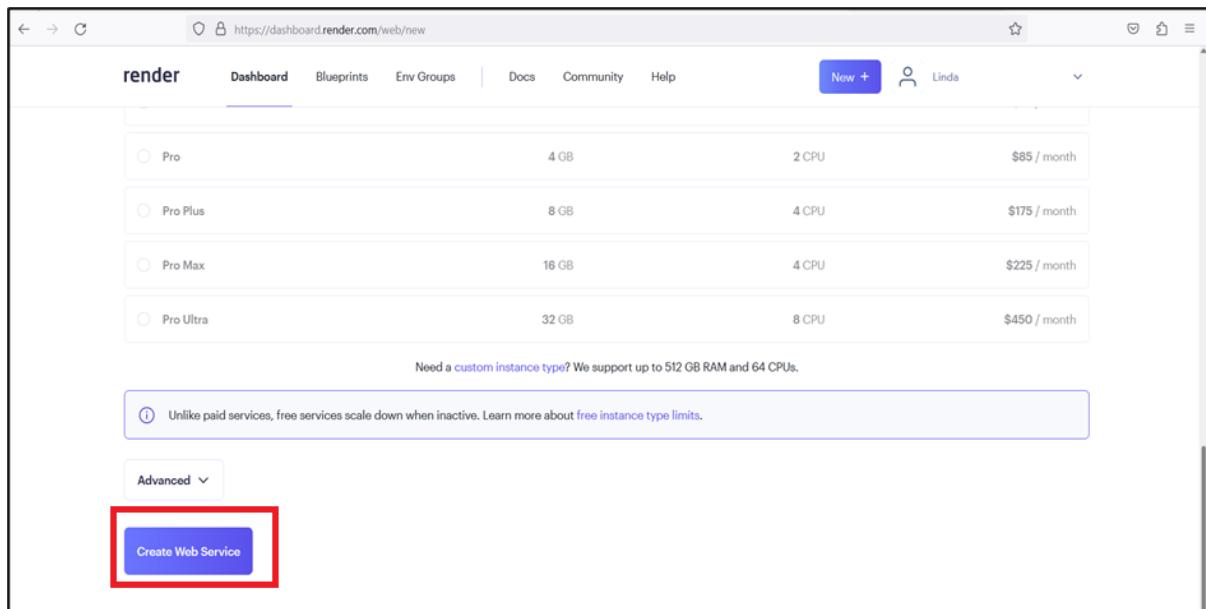


Figure 10.25: Create Web Service on Render Dashboard Page

15. Notice that the **Library-app** is deployed and the server has started as shown in Figure 10.26.

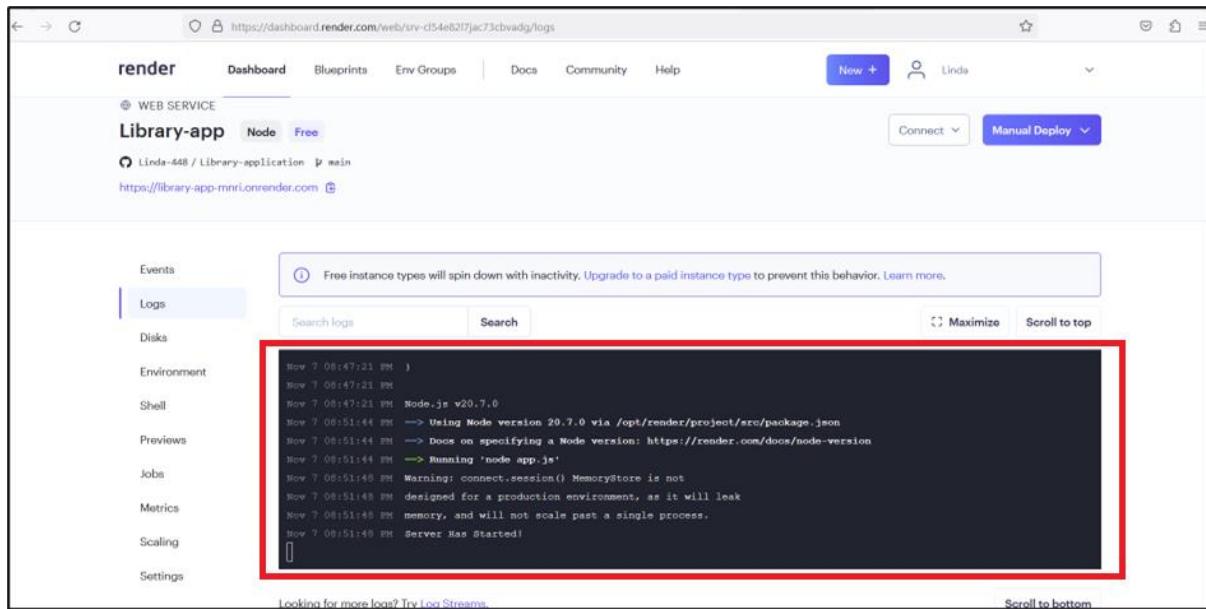


Figure 10.26: Deployment of Application

16. To identify the IP address of Render for the Library application, on the **Render Dashboard** page, click the **Connect** drop-down list as shown in Figure 10.27.

The static outbound IP addresses are displayed. It is not safe to use the IP address 0.0.0.0/0, which was specified while connecting the application to MongoDB database. Therefore, use any one IP address from the list.

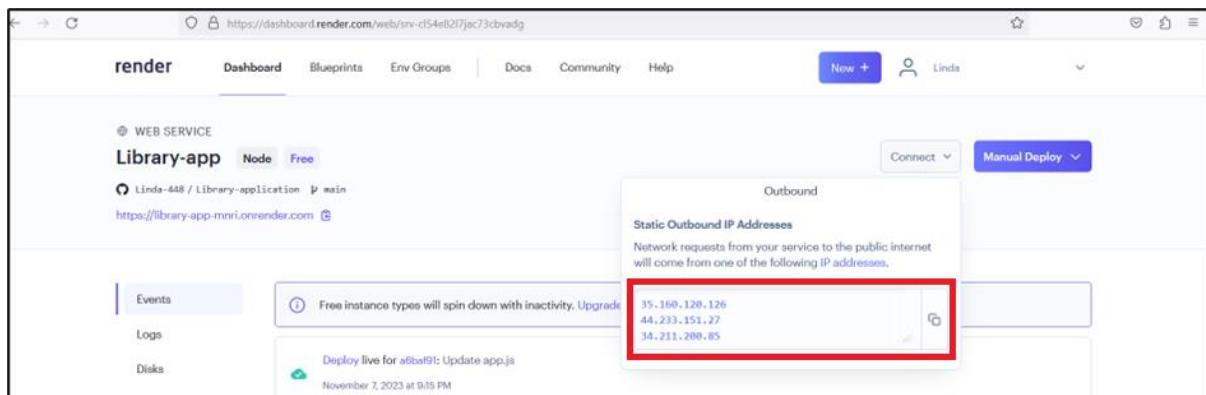


Figure 10.27: IP Addresses on Render Dashboard Page

17. To update the IP address, navigate to the MongoDB cloud account. The **Database Deployments** page is active.

On the left, click **Network Access**.

On the **Network Access** page, click **Edit** as shown in Figure 10.28.

The screenshot shows the MongoDB Atlas Network Access page. On the left, there's a sidebar with categories like Project 0, Deployment, Services, Security, and Network Access (which is selected). The main area has tabs for IP Access List, Peering, and Private Endpoint. Under IP Access List, it says 'You will only be able to connect to your cluster from the following list of IP Addresses:' followed by a table. The table has columns for IP Address, Comment, Status, and Actions. One row shows '0.0.0.0/0 (includes your current IP address)' with 'Status' as 'Active' and 'Actions' showing 'EDIT' and 'DELETE' buttons, where 'EDIT' is highlighted with a red box.

Figure 10.28: Network Access Page

18. To edit the IP address from 0.0.0.0/0 to 35.160.120.126, on the **Edit IP Access List Entry** page, in the **Access List Entry** box, type **35.160.120.126**.

To specify a comment, in the **Comment** box, type **IP of Render**, and click **Confirm** as shown in Figure 10.29.

This is a modal dialog titled 'Edit IP Access List Entry'. It contains instructions about client connections and two input fields: 'Access List Entry' with the value '35.160.120.126' and 'Comment' with the value 'IP of Render'. At the bottom are 'Cancel' and 'Confirm' buttons, with 'Confirm' highlighted with a red box.

Figure 10.29: Edit IP Access List Entry Page

19. The IP address is updated as shown in Figure 10.30.

The screenshot shows the MongoDB Atlas Network Access page for the 'SMITHA'S ORG - 2023-11-07 > PROJECT 0' cluster. The 'IP Access List' tab is selected. It displays two IP addresses: '49.37.363.87/32 (includes your current IP address)' and '35.160.120.126/32'. The second IP address, '35.160.120.126/32', is highlighted with a red box. A green button labeled '+ ADD IP ADDRESS' is visible at the top right.

Figure 10.30: IP Address Updated on Network Access Page

20. To view the status of the **Library-app** project, navigate to **Render**, and click **Dashboard** as shown in Figure 10.31.

The screenshot shows the Render Dashboard page. The 'Dashboard' tab is selected and highlighted with a red box. The page displays an 'Overview' section with a search bar and a table showing the status of the 'Library-app' project. The table includes columns for NAME, STATUS, TYPE, RUNTIME, REGION, and LAST DEPLOYED. The 'Library-app' entry shows it is 'Deployed' as a 'Web Service' in the 'Node' runtime, located in the 'Oregon' region, and was last deployed 12 minutes ago.

Figure 10.31: Project Status on Render Dashboard Page

21. To copy the URL, on the **Render Dashboard** page, click **Library-app**, select the URL, and press **Ctrl + C** as shown in Figure 10.32.

The URL, <https://library-app-mnri.onrender.com>, is copied to the clipboard.

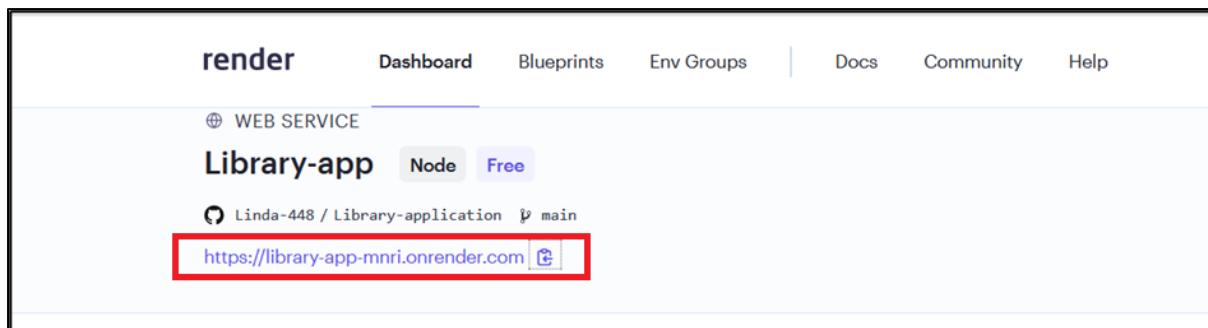


Figure 10.32: URL on Render Dashboard Page

10.6 Accessing the Library Application in Web Browser

The library application is now deployed and is ready to be accessed in the Web browser using the URL.

Perform the given steps:

1. Open the Web browser and paste the copied URL. The home page of the library application is displayed.

To create a new member, click **Sign_up for Member User** as shown in Figure 10.33.



Figure 10.33: Home Page of Library Application

2. The sign up form for a new library member is displayed.

To provide the username, in the **Username** box, type **Richard**.

To provide the password, in the **Password** box, type **1234**, and click **Sign-UP** as shown in Figure 10.34.

The screenshot shows a web browser window with the URL <https://library-app-mnri.onrender.com/register>. The page features a decorative background image of books. A central form is titled "Sign up form for Library Member". It contains two input fields: "Username" with the value "Richard" and "Password" with the value "****". Below the password field is a red rectangular box containing the "Sign-UP" button. At the bottom left of the form is a link "Home Page".

Figure 10.34: Sign Up Form for Library Member

3. To login as an administrator, on the home page, click **Administrator Login** as shown in Figure 10.35.



Figure 10.35: Home Page of Library Application

4. To provide the administrator username, on the **Admin Login** page, in the **Username** box, type **Admin**.
To provide the password, in the **Password** box, type **12345**, and click **Login** as shown in Figure 10.36.

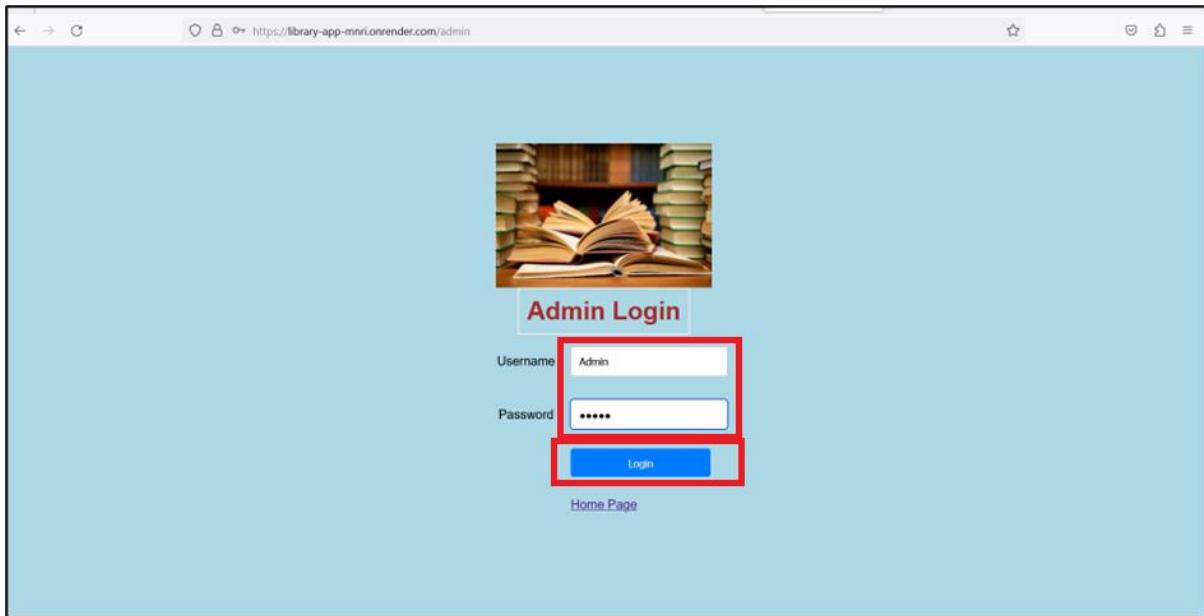


Figure 10.36: Admin Login Page of Library Application

5. To add the new book details, on the **Create Book Detail** page, type the given details, and click **Add Book** as shown in Figure 10.37.
In the **BookID** box, type **1211**.
In the **Book Name** box, type **The Secret of The House On The Hill**.
In the **Author Name** box, type **Julia Harris**.
In the **Price** box, type **\$10**.
In the **Age Group** box, type **15+**.
In the **Book Type** box, type **Thriller**.
The book details are added in the database.

Add more book details, if required or click **Logout** to return to the home page.

Create Book Detail

BookID: 1211

Book Name: The Secret Of The House On The Hill

Author Name: Julia Harris

Price: \$10

Age Group: 15+

Book Type: Thriller

Add Book

Figure 10.37: Create Book Detail Page of Library Application

6. To login as an existing member, on the **Library Home** page, click **Member Login** as shown in Figure 10.38.



Figure 10.38: Library Home Page

7. To provide the existing username, in the **Username** box, type **Richard**. To provide the password, in the **Password** box, type **1234**, and click **Login** as shown in Figure 10.39.

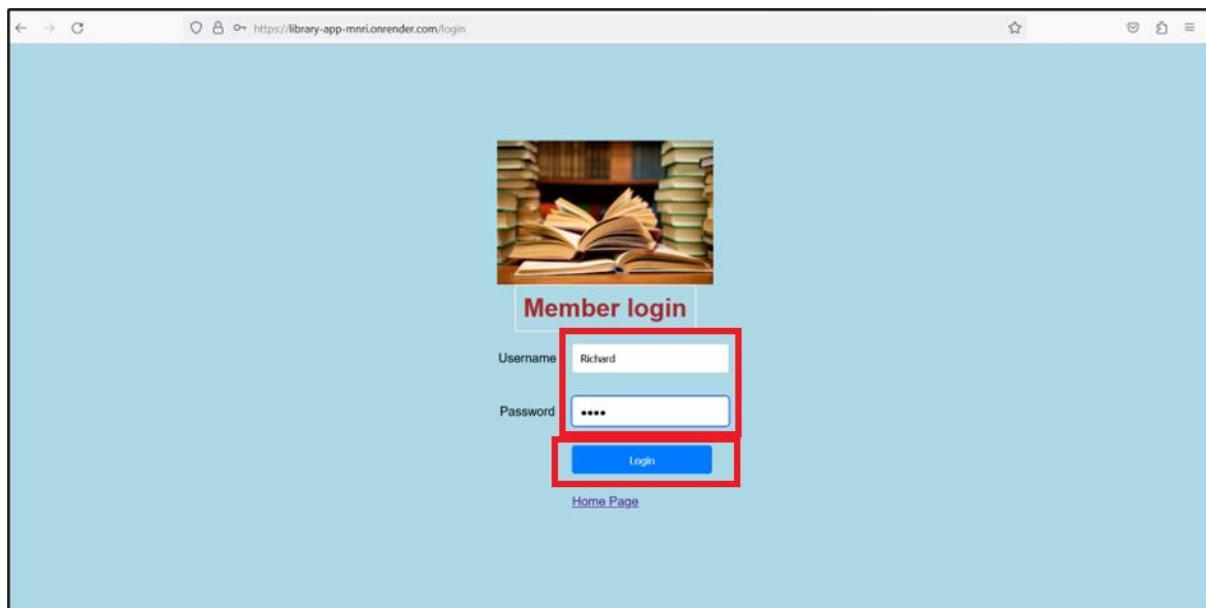


Figure 10.39: Member Login Page

8. On the **List of Books** page, the list of available books are displayed for the member as shown in Figure 10.40.

List of Books						
Book ID	Book	Author	Price	Age Group	Book Type	
1211	The Secret of The House On The Hill	Julia Harris	\$10	15+	Thriller	

Figure 10.40: Book List on List of Books Page

9. To return to the home page of the library application, on the **List of Books** page, click **Logout** as shown in Figure 10.41.

List of Books						
Book ID	Book	Author	Price	Age Group	Book Type	
1211	The Secret of The House On The Hill	Julia Harris	\$10	15+	Thriller	

Figure 10.41: Library Home Page

10.7 Summary

- The connection string created in the MongoDB cloud database connects the Web application to MongoDB cloud.
- The Node.js packages are installed in the Web application using the npm commands.
- Some of the important tasks in Web application development are:
 - Setting up the view engine and middleware
 - Defining the local strategy and routes
 - Serializing and deserializing user functions
 - Listening for requests
- Schemas and stylesheets should be created in the respective Web application folders.
- The Web application can be uploaded to the GitHub repository for better collaboration and version control.
- Render can be used to deploy the Web applications on cloud.

Test Your Knowledge

1. Arrange the steps to create a simple Web application in sequence.
 - i. Create a Web application in Node.js
 - ii. Connect Web application to database
 - iii. Upload the application to the GitHub Repository
 - iv. Deploy the application on Render
 - v. Access the application on the Web browser
 - a) ii, i, iii, iv, and v
 - b) i, ii, iii, iv, and v
 - c) v, iv, iii, ii, and i
 - d) i, ii, iii, v, and iv
2. Which command will you use to create a new project in the application folder?
 - a) npm init
 - b) npm init -y
 - c) npm init -x
 - d) npm
3. Which package in Node.js is used to parse the body of HTTP POST requests?
 - a) encryption
 - b) body
 - c) parse
 - d) body-parser
4. Which package is used to perform authentication using username and password in Node.js applications?
 - a) passport-local
 - b) local
 - c) authentication
 - d) ejs
5. What is the purpose of the `npm install ejs` command?
 - a) To install EJS run time environment
 - b) To install JavaScript package
 - c) To install the server
 - d) To install the EJS library

Answers to Test Your Knowledge

1	a
2	b
3	d
4	a
5	d

Try It Yourself

1. Create a simple Web application for the **Passport Portal**. Perform the given operations and deploy the application on **Render**.
 - Add a new profile and refer to the parameters such as Name, Age, Bank Name, and ID Proof.
 - Delete a profile.
 - Update a profile.
 - a) Import the necessary packages.
 - b) Create a database named, Passport.
 - c) Add the `Passport_Collection` collection to the Passport database and insert 3 documents as given in Table 10.2.

Name	Age	Bank Name	ID Proof
Alex	30	Bank of America	Green Card
John	21	Citigroup	Voter's Registration
Harry	26	U.S. Bancorp	Green Card

Table 10.2: Passport Profiles

- a) View the inserted documents.
- b) Update the Name Harry to Charlie.
- c) Delete a document with the Name as Alex.
- d) Display the results on the browser.
- e) Deploy the application on **Render**.



SESSION 11

NODE.JS IN ACTION (APPLICATION DEVELOPMENT) - I

Learning Objectives

In this session, students will learn to:

- Explain the process of creating a Web application using Node.js
- Describe the steps to connect the Web application to a MongoDB database

Wonderland Widescreen is a Web-based movie application created using Node.js, HTML, and related technologies. This application will have five primary functionalities:

- Login as administrator or guest
- Add a movie
- Book seats for a movie
- Delete a movie
- Update seats for a movie

The process of complete creation and deployment of this Web application is covered across this session and the next. This session includes steps to create major part of the application using Node.js with a MongoDB database. The rest of the application creation and its deployment is covered in the next session.

11.1 Overview

In this session, a simple movie application is created using Node.js, HTML, and a MongoDB cloud database called MovieCollectionCluster.

Create a collection named MovieCollection in a database MovieCollectionCluster.

Figure 11.1 shows characteristics of the Wonderland Widescreen application, which are as follows:

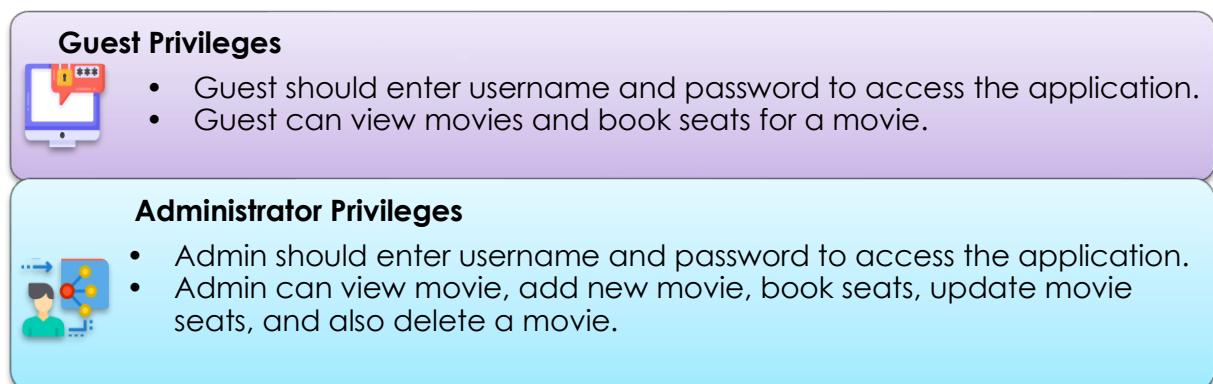


Figure 11.1: Wonderland Widescreen Application

Figure 11.2 shows the site map of the Web application with each page and its description mentioned.

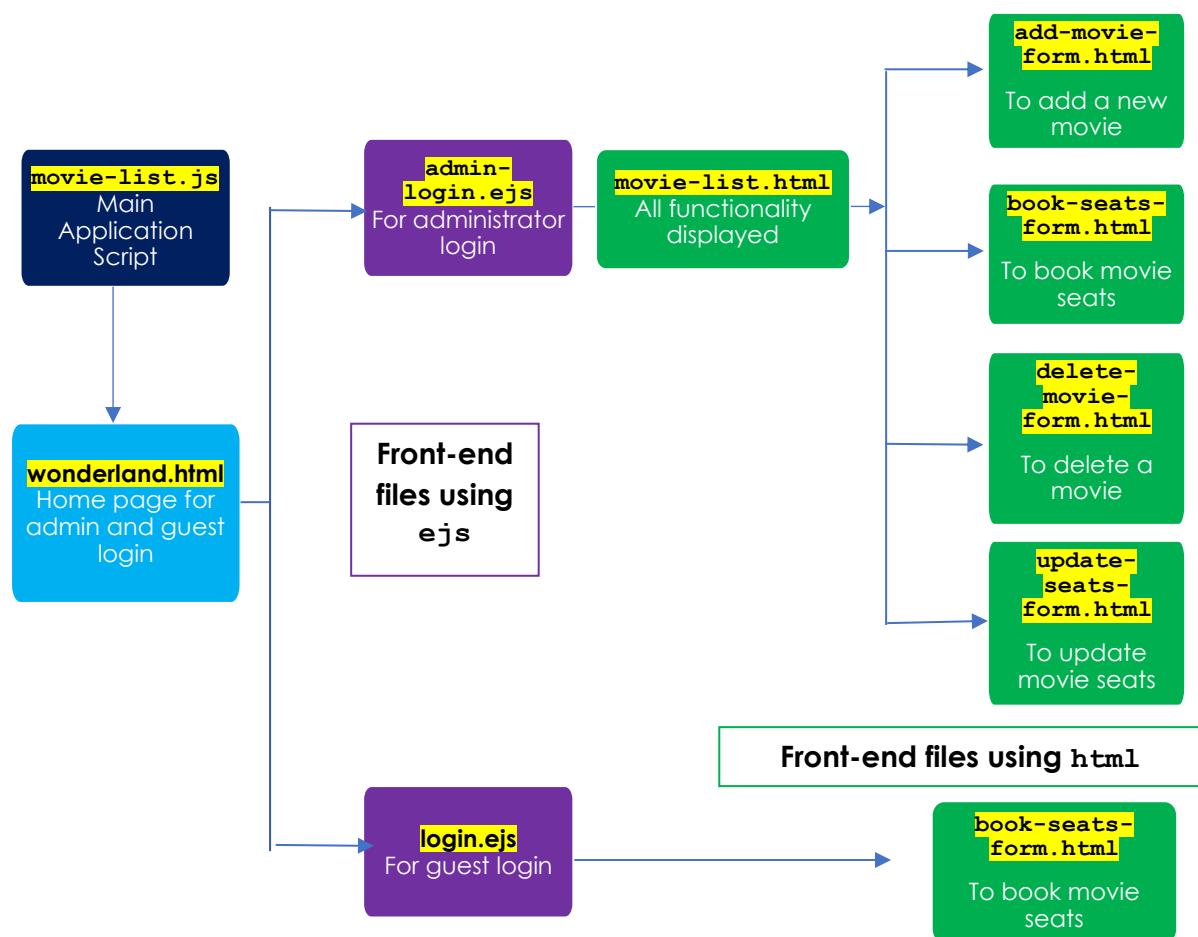
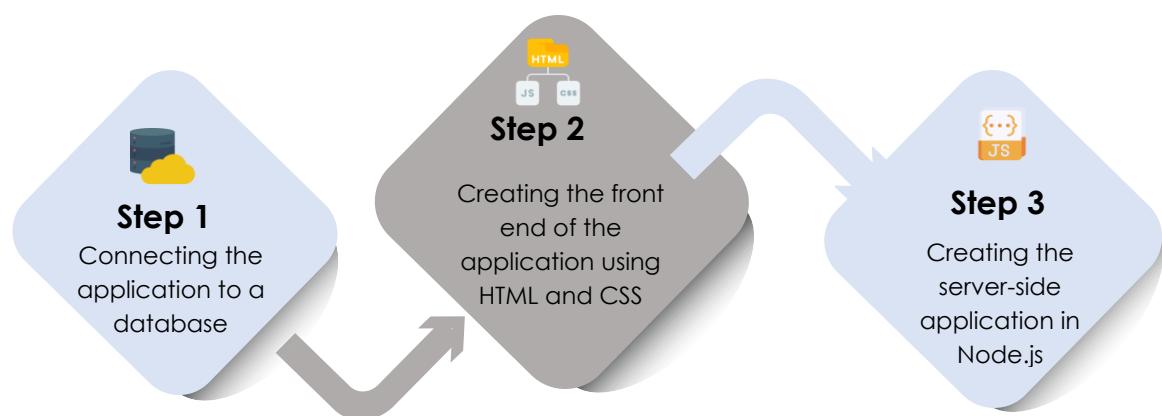


Figure 11.2: Site Map



In this session, three Web pages, movie-list.html, add-movie-form.html, and book-seats-form.html are created. Also, the main application entry point movie-list.js is created. The creation of the remaining pages such as delete-movie-form.html and update-seats-form.html is covered in the next session.

Steps to create the Wonderland Widescreen application are as follows:



11.2 Retrieving the Connection String from MongoDB Cloud

The prerequisite to create the Wonderland Widescreen application is to deploy a database in the MongoDB cloud and retrieve the database deployment's connection string. The connection string is then used to connect the Wonderland Widescreen Node.js application with the cloud database for storing, retrieving, and processing data. Refer to Appendix A to learn how to do this.

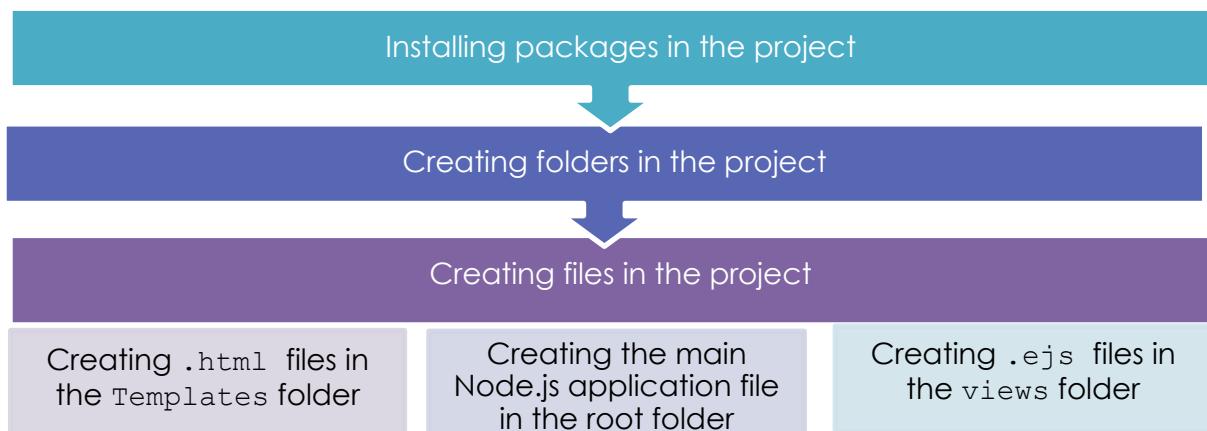
The connection string for MovieCollection appears as follows:

```
mongodb+srv://<<username>>:<<password>>@moviecollectioncluster.4x9rauo.mongodb.net/moviecollectioncluster?retryWrites=true&w=majority
```

Note that username and password placeholders in the connection string must be substituted with actual username and password provided by developer when creating the cloud database in MongoDB.

11.3 Creating the Wonderland Widescreen Application in Node.js

After the database deployment's connection string has been retrieved, one must create the Wonderland Widescreen application in Node.js which involves following steps:



11.3.1 Installing Packages in the Project

The first step is to install some important Node.js packages. One can accomplish this as follows:

1. Create a directory for the application to be developed and navigate to it:
 - a. Open Command Prompt.
 - b. Navigate to the local drive where Node.js is installed.
 - c. Navigate to the directory to create the application folder.
 - d. Create a directory named `movie-application` and change to that directory.

Figure 11.3 displays the directory path.

```
C:\Node Js>cd movie-application
```

Figure 11.3: Creation of the Application Directory

2. Create the `package.json` file to initialize a new Node.js project for the application.

3. Install the required packages such as express, body-parser, express-session, passport, and passport-local in the application directory using the commands:

```
npm install express
npm install body-parser
npm install express-session
npm install passport
npm install passport-local
```

Alternatively, you can install all these packages using a single npm command where the names of the packages are separated by spaces.

4. After installing the packages, node_modules folder and package-lock.json file are automatically created. The package-lock.json file includes the complete details of all the packages installed in the project. It should not be manually edited. Also, the package.json file is automatically updated with the version numbers of the installed packages as shown in Figure 11.4.



The screenshot shows a code editor displaying a package.json file. The file content is as follows:

```
[{"name": "movie-application", "version": "1.0.0", "description": "", "main": "index.js", "scripts": {"test": "echo \\\"Error: no test specified\\\" && exit 1"}, "keywords": [], "author": "", "license": "ISC", "dependencies": {"body-parser": "^1.20.2", "ejs": "^3.1.9", "express": "^4.18.2", "express-session": "^1.17.3", "mongoose": "^8.0.3", "passport-local": "^1.0.0", "passport-local-mongoose": "^8.0.0"}}
```

Figure 11.4: package.json File With Version Numbers

5. Open the package.json file in VS Code or Notepad. Edit the file to update the value of the main field to include the engines and node fields as shown in Code Snippet 1.

Code Snippet 1:

```
{  
  "name": "movie-application",  
  "version": "1.0.0",  
  "description": "",  
  "main": "movie-list.js",  
  "scripts": {  
    "test": "echo \\\"Error: no test specified\\\" && exit 1"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "dependencies": {  
    "body-parser": "^1.20.2",  
    "ejs": "^3.1.9",  
    "express": "^4.18.2",  
    "express-session": "^1.17.3",  
    "mongoose": "^8.0.3",  
    "passport-local": "^1.0.0",  
    "passport-local-mongoose": "^8.0.0"  
  },  
  "engines": {  
    "node": ">=18.17.1 <19.0.0"  
  }  
}
```

6. Save and close the updated package.json file.

11.3.2 Creating Folders in the Project

To organize the project structure, create the required folders and subfolders such as Templates and views as shown in Figure 11.5.

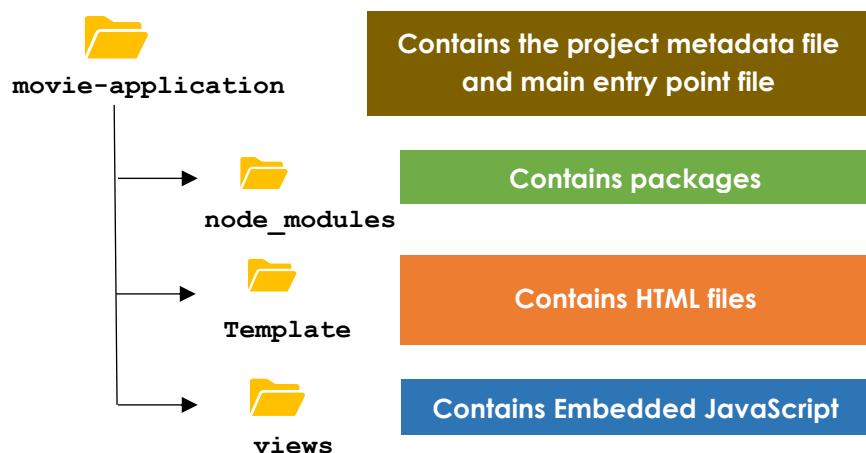


Figure 11.5: Project Folder Structure

The `node_modules` folder is created automatically when the packages are installed.

Figure 11.6 shows the listing of the `movie-application` folder.

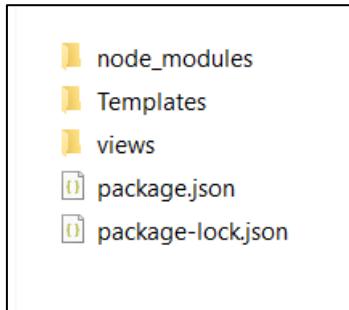


Figure 11.6: movie-application Folder

11.4 Creating Files in the Project

To define the user interface design, logic, and functionality of the project, create the required source code files under the specified project folders.

11.4.1 Creating .html files in the Templates Folder

The next step is to create front-end files using HTML.

➤ `movie-list.html`

Figure 11.7 shows the design for the `movie-list.html` page.



Figure 11.7: movie-list.html

Code Snippet 2 shows the code for movie-list.html.

Code Snippet 2:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Movie List</title>
    <link rel="stylesheet" href="style.css" />
  </head>
  <style>
    html {
      height: 100%;
    }
    body {
      text-align: center;
      background:
        url("https://cdn.pixabay.com/photo/2017/01/12/17/30/warm-
and-cozy-1975215_1280.jpg");
      background-attachment: fixed;
      background-position: relative;
      background-repeat: no-repeat;
      background-size: cover;
      height: 100%;
      margin: 0;
      place-items: center;
      position: fixed;
      font-size: 20px;
    }
    .container {
      top: 30%;
      left: 50%;
      width: 50em;
      height: 18em;
      transform: translate(-50%, -50%);
      position: fixed;
      color: rgb(255, 255, 255);
      font-size: 20px;
    }
    .button {
      background-color: #000;
      border: 0.5px solid crimson;
      border-radius: 10px;
      color: #fff;
      padding: 8px;
      font-size: 20px;
      cursor: pointer; /* Pointer/hand icon */
      /* display:block; */
    }
    .movie-details-box {
```

```
margin-top: 20px;
padding: 20px;
background-color: rgba(0, 0, 0, 0.8);
color: white;
border-radius: 10px;
box-shadow: 0 0 10px rgba(0, 0, 0, 0.5);
display: none; /* Initially hide the box */
}

.movie-category-box {
margin: 0 auto; /* Center horizontally */
margin-top: 20px;
padding: 20px;
background-color: rgba(0, 0, 0, 0.8);
color: white;
border-radius: 10px;
box-shadow: 0 0 10px rgba(0, 0, 0, 0.5);
display: none; /* Initially hide the box */
}

.button.action-button {
background-color: #8b0000;
border: 3px solid black;
cursor: pointer;
}

.button.comedy-button {
background-color: #8b0000;
border: 3px solid black;
cursor: pointer;
}

.button.drama-button {
background-color: #8b0000;
border: 3px solid black;
cursor: pointer;
}

/* Style for movie list items */
#movie-category ul {
list-style-type: none;
padding: 0;
}

#movie-category ul li {
cursor: pointer;
margin-bottom: 5px; /* Adjust as required */
/* background-color: #8fa5ac; */
}

h4 {
```

```

        color: #fcfbfb;

    }
</style>
<body>
    <div class="container">
        <h1>Welcome to the Wonderland Widescreen</h1>

        <a href="add-movie-form.html"
            ><button class="button">Add Movie</button></a>
        >
        <a href="book-seats-form.html"
            ><button class="button">Book seats</button></a>
        >
        <a href="delete-movie-form.html"
            ><button class="button">Delete Movie</button></a>
        >
        <a href="update-seats-form.html"
            ><button class="button">Update seats</button></a>
        >
        <div id="movie-list"></div>
        <br />

        <div class="button-group">
            <h4>Click each category
            to view movies.&#128071;</h4>
            <button class="button action-button"
            onclick="getMovies('Action')">
                Action
            </button>
            <button class="button comedy-button"
            onclick="getMovies('Comedy')">
                Comedy
            </button>
            <button class="button drama-button"
            onclick="getMovies('Drama')">
                Drama
            </button>
        </div>

        <!-- Create a new div for displaying movie details
-->
        <div class="movie-category-box" id="movie-
category"></div>

        <!-- Create a new div for displaying movie details
-->
        <div class="movie-details-box" id="movie-
details"></div>
        <br /><br />
    </div>

```

```

<script>
    async function getMovies(category) {
        const movieCategoryDiv =
document.getElementById("movie-category");
        try {
            const response = await fetch(`/get-
movies?category=${category}`);
            const data = await response.json();
            displayMovies(data, category);
        } catch (error) {
            console.error("Error fetching movies:", error);
        }
    }

    async function getMovieDetails(movieName) {
        try {
            const response = await fetch(`/get-movie-
details?name=${movieName}`);
            const data = await response.json();
            displayMovieDetails(data, movieName);
        } catch (error) {
            console.error("Error fetching movie details:", error);
        }
    }

    function displayMovies(movies, category) {
        const movieCategoryDiv =
document.getElementById("movie-category");
        movieCategoryDiv.innerHTML = "";

        if (movies.length === 0) {
            movieCategoryDiv.textContent = `No movies found
for category: ${category}`;
        } else {
            const ul = document.createElement("ul");
            ul.classList.add("movie-list"); // Add a class
to the list

            movies.forEach((movie) => {
                const li = document.createElement("li");
                const movieName = movie["Movie name"];
                li.textContent = movieName;
                li.classList.add("movie-list-item"); // Add a
class to the list item

                li.addEventListener("click", () => {
                    getMovieDetails(movieName);
                });
                ul.appendChild(li);
            });
        }
    }
}

```

```

        });
        movieCategoryDiv.appendChild(ul);
        movieCategoryDiv.style.display = "block"; // Show the box when movies are available
    }
}

function displayMovieDetails(details, movieName) {
    const movieDetailsDiv =
document.getElementById("movie-details");
    movieDetailsDiv.innerHTML = "";

    if (details) {
        const description = details["Description"];
        const actors = details["Actors"];
        const detailsText = `
            <div><strong>Movie Name:</strong>
${movieName}</div>
            <div><strong>Description:</strong>
${description}</div>
            <div><strong>Actors:</strong>
${actors}</div>
        `;
        movieDetailsDiv.innerHTML = detailsText;

        movieDetailsDiv.style.display = "block"; // Show the box when details are available
    }
}
</script>
</body>
</html>

```

The `movie-list.html` form:

- Uses HTML and JavaScript.
- Has four button controls created using `href` tag to provide main functionality.
 - **Add Movie** Button: When the user clicks this button the application is redirected to the `add-movie-form.html` page.
 - **Book seats** Button: When the user clicks this button the application is redirected to the `book-seats-form.html` page.
 - **Delete Movie** Button: When the user clicks this button, the application is redirected to the `delete-movie-form.html` page.
 - **Update seats** Button: When the user clicks this button, the application is redirected to the `update-seats-form.html` page.

- Has three buttons with `onclick` events to denote movie categories. When the user clicks these buttons, available movies in that category are displayed.
 - **Action:** When this button is clicked, the `getMovies` function is called with `Action` as argument.
 - **Comedy:** When this button is clicked, the `getMovies` function is called with `Comedy` as argument.
 - **Drama:** When this button is clicked, the `getMovies` function is called with `Drama` as argument.
- Displays the **Movie names** in a list using the `movie-category` division.
- Displays the **Movie details** using the `movie-details` division.
- Defines the `getMovies(category)` function to fetch movies from the MongoDB server based on the specific category. Here, the `fetch` function is used to make request to server using `/get-movies` endpoint with the specified category.
- Defines the `getMovieDetails(movieName)` function to fetch the details of a movie from MongoDB server based on the selected movie name. This uses `/get-movie-details` endpoint.
- Defines the `displayMovies(movies, category)` function for displaying list of movies in a specified category. This function uses the `movie-category` division to display a dynamically created unordered list of movies.
- Defines the `displayMovieDetails(details, movieName)` function that uses the `movie-details` division to display details of a specific movie.

Save the code with the file name `movie-list.html` in the `Templates` folder.

➤ **add-movie-form.html**

Figure 11.8 shows the design for the `add-movie-form.html` page.

The screenshot displays a web form titled "Add Movie Details" set against a background gradient from red to yellow. The form consists of six input fields: "Movie Name", "Screen No", "Show Time", "Available Seats", "Actors Name", and "Movie Description". Below these is a dropdown menu labeled "Select a Category ▾". At the bottom is a large blue "SUBMIT" button.

Field	Type	Description
Movie Name	Text	Input field for movie name
Screen No	Text	Input field for screen number
Show Time	Text	Input field for show time
Available Seats	Text	Input field for available seats
Actors Name	Text	Input field for actors name
Movie Description	Text	Input field for movie description
Select a Category	Dropdown	Category selection dropdown
SUBMIT	Button	Large blue submit button

Figure 11.8: add-movie-form.html

Code Snippet 3 shows the code for `add-movie-form.html`. The form that accepts the details of a new movie stores the movie details entered by the user to the `MovieCollectionCluster` database.

Code Snippet 3:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width,
initial-scale=1.0" />
    <title>Add Movie Details</title>
    <style>
      html {
        height: 100%;
      }
    </style>
  </head>
  <body>
```

```
body {
    margin: 0;
    padding: 0;
    background:
        url("https://cdn.pixabay.com/photo/2018/02/27/15/40/background-3185765_1280.jpg");
        /* background:
        url("https://cdn.pixabay.com/photo/2015/05/13/07/07/playmobil-765110_1280.jpg"); */
        background-attachment: fixed;
        background-position: relative;
        background-repeat: no-repeat;
        background-size: cover;
}

.login-box {
    position: absolute;
    top: 50%;
    left: 50%;
    width: 400px;
    padding: 40px;
    transform: translate(-50%, -50%);
    background: rgba(226, 53, 157, 0.5);
    box-sizing: border-box;
    box-shadow: 0 15px 25px rgba(226, 53, 157, 0.5);
    border-radius: 10px;
    font-size: 20px;
}

.login-box h2 {
    margin: 0 0 30px;
    padding: 0;
    color: #520340;
    text-align: center;
}

.login-box .user-box {
    position: relative;
}

.login-box .user-box input {
    width: 100%;
    padding: 10px 0;
    font-size: 16px;
    color: #fff;
    margin-bottom: 30px;
    border: none;
    border-bottom: 1px solid #fff;
    outline: none;
    background: transparent;
}
```

```

.login-box .user-box label {
    position: absolute;
    top: 0;
    left: 0;
    padding: 10px 0;
    font-size: 16px;
    color: #fff;
    pointer-events: none;
    transition: 0.5s;
}

.login-box .user-box input:focus ~ label,
.login-box .user-box input:valid ~ label {
    top: -20px;
    left: 0;
    color: #520340;
    font-size: 12px;
}

.login-box-button {
    position: relative;
    display: inline-block;
    padding: 10px 20px;
    color: #ee0754;
    font-size: 16px;
    text-decoration: none;
    text-transform: uppercase;
    overflow: hidden;
    transition: 0.5s;
    margin-top: 40px;
    letter-spacing: 4px;
    background: #520340;
    border-radius: 5px;
    box-shadow: 0 0 5px #520340, 0 0 25px #520340, 0 0
50px #520340,
        0 0 100px #520340;
}

.login-box-button:hover {
    background: #fff;
    color: #520340;
}

</style>
</head>
<body>
    <div class="login-box">
        <h2>Add Movie Details</h2>
        <form id="movie-form" action="/add-movie"
method="POST">
            <div class="user-box">

```

```

        <input type="text" id="movie-name" name="Movie
name" required="" />
            <label>Movie Name</label>
        </div>
        <br />
        <div class="user-box">
            <input type="text" id="screen-no" name="Screen no"
required="" />
                <label>Screen No</label>
            </div>
            <br />
            <div class="user-box">
                <input type="text" id="show-time" name="Show time"
required="" />
                    <label>Show Time</label>
                </div>
                <br />
                <div class="user-box">
                    <input
                        type="number"
                        id="available-seats"
                        name="Available seats"
                        required=""
                    />
                    <label>Available Seats</label>
                </div>
                <br />
                <div class="user-box">
                    <input type="text" id="actors-names" name="Actors"
required="" />
                        <label>Actors Name</label>
                    </div>
                    <br />
                    <div class="user-box">
                        <input
                            type="text"
                            id="movie-description"
                            name="Description"
                            required=""
                        />
                        <label>Movie Description</label>
                    </div>
                    <br />
                    <div class="user-box">
                        <!--          <input type="text" id="movie-
Category" name="Category" required="" -->
                            <label>Movie Category</label> -->
                            <select id="movie-Category" name="Category"
required="">
                                <option value="" disabled selected>Select a
Category</option>

```

```
<option value="Action">Action</option>
<option value="Comedy">Comedy</option>
<option value="Drama">Drama</option>
<!-- Add more options as required -->
</select>
<br />
</div>
<button type="submit" class="login-box-button">
    <span></span>
    <span></span>
    <span></span>
    <span></span>
    Submit
</button>
</form>
</div>
</body>
</html>
```

The add-movie-form.html form:

- Uses HTML.
- Defines input fields for movie details such as Movie Name, Screen No, Show Time, Available Seats, Actors Name, Movie Description, and category.
- Uses the /add-movie route.

Save the code with the file name add-movie-form.html in the Templates folder.

➤ **book-seats-form.html**

Figure 11.9 shows the design for the book-seats-form.html page.



Figure 11.9: book-seats-form.html

Code Snippet 4 shows the code for book-seats-form.html. This form helps users to book seats for a movie.

Code Snippet 4:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>Book Seats</title>
    <style>
      html {
        height: 100%;
      }
      body {
        margin: 0;
        /* padding:0;
        background:
url("https://cdn.pixabay.com/photo/2018/02/27/15/40/backgroun
d-3185765_1280.jpg"); */
        /* background:
url("https://cdn.pixabay.com/photo/2015/05/13/07/07/playmobil
-765110_1280.jpg"); */
        background:
url("https://cdn.pixabay.com/photo/2017/01/12/17/30/warm-and-
cozy-1975215_1280.jpg");
        background-attachment: fixed;
        background-position: relative;
        background-repeat: no-repeat;
        background-size: cover;
      }
    </style>
  </head>
  <body>
    <div>
      <h2>Movie List</h2>
      <button>Action</button>
      <button>Comedy</button>
      <button>Drama</button>
    </div>
    <div>
      <h2>Book Your Seats</h2>
      <form>
        <label>Select a movie <input type="button" value="▼" /></label>
        <input type="text" value="Seats to Book" />
        <input type="button" value="SUBMIT" />
      </form>
    </div>
  </body>
</html>
```

```
.login-box {
    position: absolute;
    top: 50%;
    left: 50%;
    width: 400px;
    padding: 50px;
    transform: translate(-50%, -50%);
    background: rgba(16, 15, 15, 0.5);
    box-sizing: border-box;
    box-shadow: 0 15px 25px rgba(80, 80, 80, 0.5);
    border-radius: 10px;
}

.login-box h2 {
    margin: 0 0 30px;
    padding: 0;
    color: #f3eff2;
    text-align: center;
}

.login-box .user-box {
    position: relative;
}

.login-box .user-box input {
    width: 100%;
    padding: 10px 0;
    font-size: 20px;
    color: #fff;
    margin-bottom: 30px;
    border: none;
    border-bottom: 1px solid #fff;
    outline: none;
    background: transparent;
}

.login-box .user-box label {
    position: absolute;
    top: 0;
    left: 0;
    padding: 10px 0;
    font-size: 16px;
    color: #fff;
    pointer-events: none;
    transition: 0.5s;
}

.login-box .user-box input:focus ~ label,
.login-box .user-box input:valid ~ label {
    top: -20px;
    left: 0;
```

```
        color: #f9f9f9;
        font-size: 15px;
    }

    .login-box-button {
        position: relative;
        display: inline-block;
        padding: 10px 20px;
        color: #f8f5f6;
        font-size: 16px;
        text-decoration: none;
        text-transform: uppercase;
        overflow: hidden;
        transition: 0.5s;
        margin-top: 40px;
        letter-spacing: 4px;
        background: #520306;
        border-radius: 5px;
        box-shadow: 0 0 5px #520306, 0 0 25px #520306, 0 0
50px #520306,
            0 0 100px #520306;
    }

    .login-box-button:hover {
        background: #fff;
        color: #520340;
    }

/*Category button action, comedy and drama*/

.button {
    background-color: #000;
    border: 0.5px solid crimson;
    border-radius: 10px;
    color: #fff;
    padding: 8px;
    font-size: 20px;
    cursor: pointer; /* Pointer/hand icon */
    /* display:block; */
}

.movie-details-box {
    margin-top: 20px;
    padding: 20px;
    background-color: rgba(0, 0, 0, 0.8);
    color: white;
    border-radius: 10px;
    box-shadow: 0 0 10px rgba(0, 0, 0, 0.5);
    display: none; /* Initially hide the box */
}
```

```
.movie-category-box {  
    margin: 0 auto; /* Center horizontally */  
    margin-top: 20px;  
    padding: 20px;  
    background-color: rgba(0, 0, 0, 0.8);  
    color: white;  
    border-radius: 10px;  
    box-shadow: 0 0 10px rgba(0, 0, 0, 0.5);  
    display: none; /* Initially hide the box */  
}  
  
.button.action-button {  
    background-color: #8b0000;  
    border: 3px solid black;  
    cursor: pointer;  
}  
  
.button.comedy-button {  
    background-color: #8b0000;  
    border: 3px solid black;  
    cursor: pointer;  
}  
  
.button.drama-button {  
    background-color: #8b0000;  
    border: 3px solid black;  
    cursor: pointer;  
}  
  
/* Style for movie list items */  
#movie-category ul {  
    list-style-type: none;  
    padding: 0;  
}  
  
#movie-category ul li {  
    cursor: pointer;  
    margin-bottom: 5px; /* Adjust as required */  
    /* background-color: #8fa5ac; */  
}  
  
.container {  
    top: 40%; /*Display category buttons from top*/  
    left: 20%;  
    width: 20em; /*width of the black box*/  
    height: 18em;  
    transform: translate(-50%, -50%);  
    position: fixed;  
    color: rgb(255, 255, 255);  
    font-size: 20px;  
    cursor: pointer;
```

```

        }
    </style>
</head>
<body>
<div class="container">
    <h2>Movie List</h2>
    <div id="movie-list"></div>

    <div class="button-group">
        <button class="button action-button"
onclick="getMovies('Action')">
            Action
        </button>
        <button class="button comedy-button"
onclick="getMovies('Comedy')">
            Comedy
        </button>
        <button class="button drama-button"
onclick="getMovies('Drama')">
            Drama
        </button>
    </div>

    <!-- Create a new div for displaying movie details -->
    <div class="movie-category-box" id="movie-
category1"></div>

    <!-- Create a new div for displaying movie details -->
    <div class="movie-details-box" id="movie-
details1"><br /><br /><br />
    </div>

<div class="login-box">
    <h2>Book Your Seats</h2>
    <form id="seats-form" action="/book-seats"
method="POST">
        <div class="user-box">
            <select id="movie-category" name="Movie name">
                <option value="">Select a movie</option>
            </select>
        </div>

        <br /><br />
        <div class="user-box">
            <input
                type="number"
                id="seats-to-book"
                name="seats-to-book"
                required
            />

```

```

        <label>Seats to Book</label>
    </div>
    <button type="submit" class="login-box-button">
        <span></span>
        <span></span>
        <span></span>
        <span></span>
        Submit
    </button>
</form>
<br /><br />
</div>
<script>
    async function getMovieList() {
        const movieCategoryDiv =
document.getElementById("movie-category");
        try {
            //const response = await fetch(`/get-all-movies?category=${'Action'}`);
            const response = await fetch(`/get-all-movies`);
            const data = await response.json();
            displayMovieList(data);
        } catch (error) {
            console.error("Error fetching movies:", error);
        }
    }

    async function getMovieDetails(movieName) {
        console.log(movieName);
        try {
            const response = await fetch(`/get-movie-details?name=${movieName}`);
            const data = await response.json();
            displayMovieDetails(data, movieName);
        } catch (error) {
            console.error("Error fetching movie details:", error);
        }
    }

    function displayMovieDetails(details, moviename) {
        console.log(details);
        console.log(moviename);
        const movieDetailsDiv =
document.getElementById("movie-details");
        movieDetailsDiv.innerHTML = "";

        if (details) {
            const description = details["Description"];
            const actors = details["Actors"];

```

```

        const detailsText = `Description:
${description}<br>Actors: ${actors}`;
        movieDetailsDiv.innerHTML = detailsText;
    }
}

function displayMovieList(movies) {
    const movieCategoryDiv =
document.getElementById("movie-category");
    // movieCategoryDiv.innerHTML = '';

    if (movies.length === 0) {
        movieCategoryDiv.textContent = `No movies found`;
    } else {
        movies.forEach((movie) => {
            let option = document.createElement("option");
            option.setAttribute("value", movie["Movie
name"]);
            let optionText =
document.createTextNode(movie["Movie name"]);
            option.appendChild(optionText);

            movieCategoryDiv.appendChild(option);
        });
    }

    movieCategoryDiv.style.display = "block"; // Show
//the box when movies are available
}
}

// Category button code

async function getMovies(category) {
    const movieCategoryDiv =
document.getElementById("movie-category1");
    try {
        const response = await fetch(`/get-
movies?category=${category}`);
        const data = await response.json();
        displayMovies(data, category);
    } catch (error) {
        console.error("Error fetching movies:", error);
    }
}

async function getMovieDetails(movieName) {
    try {
        const response = await fetch(`/get-movie-
details?name=${movieName}`);
        const data = await response.json();

```

```

        displayMovieDetails1(data, movieName);
    } catch (error) {
        console.error("Error fetching movie details:", error);
    }
}

function displayMovies(movies, category) {
    const movieCategoryDiv =
document.getElementById("movie-category1");
    movieCategoryDiv.innerHTML = "";

    if (movies.length === 0) {
        movieCategoryDiv.textContent = `No movies found for category: ${category}`;
    } else {
        const ul = document.createElement("ul");
        ul.classList.add("movie-list"); // Add a class to
// the list

        movies.forEach((movie) => {
            const li = document.createElement("li");
            const movieName = movie["Movie name"];
            li.textContent = movieName;
            li.classList.add("movie-list-item"); // Add a
// class to the list item

            li.addEventListener("click", () => {
                getMovieDetails(movieName);
            });
            ul.appendChild(li);
        });
        movieCategoryDiv.appendChild(ul);
        movieCategoryDiv.style.display = "block";
        // Show the box when movies are available
    }
}

function displayMovieDetails1(details, movieName) {
    const movieDetailsDiv =
document.getElementById("movie-details1");
    movieDetailsDiv.innerHTML = "";

    if (details) {
        const description = details["Description"];
        const actors = details["Actors"];
        const detailsText = `
            <div><strong>Movie Name:</strong>
${movieName}</div>
            <div><strong>Description:</strong>
${description}</div>
`;
        movieDetailsDiv.innerHTML = detailsText;
    }
}

```

```

        <div><strong>Actors:</strong> ${actors}</div>
    `;
    movieDetailsDiv.innerHTML = detailsText;

    movieDetailsDiv.style.display = "block";
    // Show the box when details are available
}
}
// Call the function to fetch movies when
// the page loads
document.addEventListener("DOMContentLoaded",
getMovieList());
</script>
</body>
</html>

```

The book-seats-form.html form:

- Uses HTML and JavaScript.
- Defines three buttons to filter the movie names by category and defines the `onclick` event for these buttons **Action**, **Comedy**, and **Drama**.
- Displays movie names in a list using the `movie-category1` division.
- Displays movie details using the `movie-details1` division.
- Defines the `getMovieList` function that fetches the movie names from the MongoDB server using `/get-all-movies` endpoint. This function calls the `displayMovieList` function to list the movie names on the page.
- Defines the `getMovies` function that fetches the movie based on the specific category using `/get-movies` endpoint. This function calls the `displayMovies` function to show the list of movies in a specific category.
- Defines the `getMovieDetails` function that fetches details of a selected movie using `/get-movie-details` endpoint. This function calls the `displayMovieDetails1` function to display the details of the selected movie.

- Defines the `displayMovies` function displays a list of movies in a specified container. This function:
 - Checks if there are any movies available for the specific category.
 - Creates an unordered list (`ul`) by iterating through the list of movies for the specified category. For each movie it creates a list item (`li`) and sets the text of the `li` to the movie name. The function then, attaches an event listener to the list item. When a list item is clicked, the function calls the `getMovieDetails` function with the selected movie name. Each list item is appended to the unordered list.
 - Appends the unordered list to the movie category container.

Save the code with the file name `book-seats-form.html` in the `Templates` folder.

➤ `movie-list.js`

Finally, create the main Node.js application file, `movie-list.js`. This file acts as the main entry point for the movie application. In this file, the core logic of the application is handled. Routes and middleware are also defined.

Consider the code in Code Snippet 5 for `movie-list.js` file.

Code Snippet 5:

```
const express = require("express");
const { MongoClient } = require("mongodb");
const bodyParser = require("body-parser");
const session = require("express-session");
const passport = require("passport");
const LocalStrategy = require("passport-local");
const path = require("path"); // Import the path module
// const uri =
// 
"mongodb+srv://<>: <>@moviecollectioncluster.4x9rauo.mongodb.net/moviecollectioncluster?retryWrites=true&w=majority"; //MongoDB Atlas Link
//Connect to MongoDB using MongoDB Compass localhost
const uri = "mongodb://127.0.0.1:27017/Movie";
// MongoDB connection URI
const client = new MongoClient(uri);
const app = express();

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

async function main() {
```

```

try {
    await client.connect();
    console.log("Connected to MongoDB");

    const database = client.db();
    const collection =
database.collection("MovieCollection");
    // Adjust the collection name

    // Set up the views folder
    app.set("views", path.join(__dirname, "views"));

    app.get("/", (req, res) => {
        console.log("Received request for /");
        // Serve the HTML page for displaying movie lists
        res.sendFile(__dirname + "/Templates/movie-list.html");
    });

    // Serve Add_movie-form.html
    app.get("/add-movie-form.html", (req, res) => {
        console.log("Received request for /add-movie-
form.html");
        // Serve the HTML form
        res.sendFile(__dirname + "/Templates/add-movie-
form.html");
    });

    // Serve book-seats-form.html
    app.get("/book-seats-form.html", (req, res) => {
        res.sendFile(__dirname + "/Templates/book-seats-
form.html");
    });

    app.get("/get-movies", async (req, res) => {
        const category = req.query.category;
        // Get the selected category from query parameters

        try {
            // Fetch movies of the selected category
            // from the database
            const movies = await collection.find({ Category:
category }).toArray();
            res.status(200).json(movies);
        } catch (error) {
            console.error("Error fetching movies:", error);
            res.status(500).json({ error: "Failed to fetch
movies" });
        }
    });

    app.get("/get-all-movies", async (req, res) => {

```

```

        try {
            // Fetch movies of the selected category from the
            database
            const movies = await collection.find().toArray();
            res.status(200).json(movies);
        } catch (error) {
            console.error("Error fetching movies:", error);
            res.status(500).json({ error: "Failed to fetch
movies" });
        }
    });

    app.get("/get-movie-details", async (req, res) => {
        const movieName = req.query.name; // Get the selected
// movie name from query parameters

        try {
            // Fetch movie details based on the selected
            // movie name from the database
            const movie = await collection.findOne({ "Movie
name": movieName });
            if (movie) {
                res.status(200).json({
                    Description: movie["Description"],
                    Actors: movie["Actors"],
                });
            } else {
                res.status(404).json({ error: "Movie not found" });
            }
        } catch (error) {
            console.error("Error fetching movie details:",
error);
            res.status(500).json({ error: "Failed to fetch movie
details" });
        }
    });

    // Handle the form submission and add a movie
    app.post("/add-movie", async (req, res) => {
        try {
            // Insert a new movie document into the MongoDB
            collection
            await collection.insertOne(req.body);
            console.log("Added movie to the database");
            // Display an alert on the same page
            res.send(
                '<script>alert("Movie added successfully");
window.location.href = "/";</script>'
            );
        } catch (error) {
            console.error("Error adding movie:", error);
        }
    });
}

```

```

        // Render an error message on the same page
        res.status(500).send("<h2>Failed to add the
movie</h2>");
    }
})
// Handle booking seats
app.post("/book-seats", async (req, res) => {
    try {
        // Retrieve movie information from the request
        const movieNameToBook = req.body["Movie name"];
        const seatsToBook = parseInt(req.body["seats-to-
book"]);
        // Check if the movie exists
        const existingMovie = await collection.findOne({
            "Movie name": movieNameToBook,
        });
        if (!existingMovie) {
            console.log("Movie not found in the database.");
            return res.send("Movie not found in the
database.");
        }
        // Retrieve the available seats from the existing
        movie document
        const availableSeats = existingMovie["Available
Seats"]; //Here field name should be same with databse field
name
        if (seatsToBook <= availableSeats) {
            // Calculate the updated available seats
            const updatedAvailableSeats = availableSeats -
seatsToBook;
            // Update the document with new available seats
            const result = await collection.updateOne(
                { "Movie name": movieNameToBook },
                { $set: { "Available Seats":
updatedAvailableSeats } } //Here field name should be same
with databse field name
            );
            if (result.modifiedCount === 1) {
                // Display an alert message on the same page
                const alertMessage = `Booking successful for
${seatsToBook} seat(s) in ${movieNameToBook}`;
                return res.send(`

<script>
    alert("${alertMessage}");
    window.location.href = "${"/"}`;

```

```

        `);
    } else {
        console.log("Failed to update available seats");
        const errorMessage = "Failed to update available
seats";
        // Display an alert on the same page
        return res.send(`

<script>
    alert("${errorMessage}");
    window.location.href = "${"/"}";
</script>
`);
    }
} else {
    console.log(
        `Not enough seats available for ${seatsToBook}
seat(s) in ${movieNameToBook}`
    );
    return res.send(
        `Not enough seats available for ${seatsToBook}
seat(s) in ${movieNameToBook}` // This will display message
    );
}
} catch (error) {
    console.error("Error booking seats:", error);
    return res.status(500).send("Failed to book seats");
}
);
} catch (error) {
    console.error("Error connecting to MongoDB:", error);
}
}

main().catch(console.error);

app.listen(process.env.PORT || 3000, process.env.IP ||
"0.0.0.0", () => {
    console.log("Server is running");
});

```

The movie-list.js file:

- **Imports packages:** Various Node.js packages such as express, mongoose, passport, body-parser, passport-local, and passport-local-mongoose are imported.
- **Creates express application:** The statement const app = express() creates the express application.

- **Connects to the database:** The statement `(mongodb+srv://prachigupta8585:prachi8585@moviecollectioncluster.4x9rauo.mongodb.net/moviecollectioncluster?retryWrites=true&w=majority)` connects the application to the MongoDB database, Library, using the connection string.
- **Sets up the middleware:** Packages such as `body-parser`, `express-session`, and `passport-authenticate` are used to set up the middleware.
- **Defines routes:** The application defines several routes for handling Hypertext Transfer Protocol (HTTP) requests. The routes are:
 - `/`: This is the home page route.
 - `/get-movies`: This is to fetch movies of the selected category from the database.
 - `/get-all-movies`: This is to fetch all movie names from the database.
 - `/get-movie-details`: This is to fetch movie details based on the selected movie name from the database.
 - `/add-movie`: This is to handle form submission and add a new movie.
 - `/book-seats`: This is to handle booking seats.
- **Listens for requests:** The application listens on the specified port, either from the environment variable or port 3000. A message is logged when the server starts.

Save the code with the file name `movie-list.js` in the `movie-application` folder.

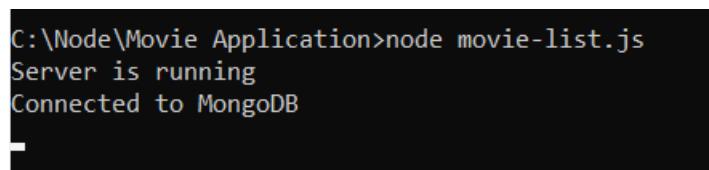
11.5 Running the Application

To view the application in the browser:

1. Open Command Prompt.
2. Navigate to the movie application folder.
3. Execute the main application file using the command:

```
node movie-list.js
```

Figure 11.10 shows the output of this command.



```
C:\Node\Movie Application>node movie-list.js
Server is running
Connected to MongoDB
```

Figure 11.10: Application in the Browser

4. Open a browser window.
5. Type the URL:

```
localhost:3000
```

Since the Web application is not fully developed yet, the page only displays the output as shown in Figure 11.11.



Figure 11.11: Application in the Browser

The application is not yet fully complete hence, further steps to add more pages and deploy the application are covered in the next session.

11.6 Summary

- Creation of a complete Web application involves a combination of several different client-side and server-side technologies.
- The connection string created in the MongoDB cloud database connects the Wonderland Widescreen application to the MongoDB cloud.
- Front-end files are created using HTML and JavaScript.
- The main application entry point is a JavaScript file that must be kept in the root folder of the application.
- HTML files are kept in the `Templates` folder under the root folder.
- When Node packages are installed, they are automatically stored in the `node_modules` folder under the root folder.

Test Your Knowledge

1. Which of the following files will you edit if an error is encountered while deploying the Node.js application on Render?
 - a) package-lock.json
 - b) package.json
 - c) app.js
 - d) render.json

2. Consider the code given here.

```
const express = require('express');
const app = express();
const ejs = require('ejs');
app.set('view engine' , 'ejs');
app.get('/', (req , res) => {
    const data = {
        Employee_designation: 'Analyst',
        Employee_ID: 'SP123',
        Company_Location: 'Ontario'
    };
    res.render('index', data) ;
});
app.listen(3000, () => {
    console.log(' Server started on port 3000 ');
});
```

Which of the following code snippets will you use to display Employee_Designation and Employee_Location in the index.html file?

- a) <body>
 <h1> <%= Employee_designation %> </h1>
 <h2> <%= Company_Location %> </h3>
 </body>
- b) <body>
 <h1> <%= Employee %> </h1>
 <h2> <%= Location %> </h3>
 </body>
- c) <body>
 <h1> <%= Employee_designation %> </h1>
 </body>
- d) <body>
 <h1> <%= designation %> </h1>
 <h2> <%= Location %> </h3>
 </body>

3. Which of the following middleware applications helps you add admin and user login to your Web application?
- a) passport-local
 - b) local
 - c) authentication
 - d) ejs
4. You are working on user authentication for your Web application. If incorrect credentials are entered for the admin login, the application must display a message on a pop-up. Which of the following will you use to achieve this?
- a) <script>alert("Incorrect Admin username or password")</script>
 - b) alert("Incorrect Admin username or password")
 - c) res.send("Incorrect Admin username or password")
 - d) res.send('<script>alert("Incorrect Admin username or password"); window.location.href = "/";</script>');
5. Which function will you use to store the user's ID, username, or any other information?
- a) passport.use
 - b) serializeUser and deserializeUser
 - c) passport
 - d) serialize

Answers to Test Your Knowledge

1	b
2	a
3	a
4	d
5	b

Try It Yourself

1. Create a simple Web application for the **Student Management System**. Perform the given operations.
 - a) Create an application directory.
 - b) Import the necessary packages.
 - c) Create a database named Student. In this database add the `Student_Collection` collection and insert 3 documents as given in Table 11.1.

Name	Age	Marks
Alex	21	93
John	20	80
Harry	22	79

Table 11.1: Student_Collection Data

- d) Create an HTML form to view all the information in the inserted documents in a list format.
- e) Create an HTML form to add a new student to the collection. Use input text boxes to accept the details of the new student. Add a **Submit** button to add the information to the MongoDB cloud database.
- f) Create an HTML form to update student information. In this page, add a listbox to list the names of all the students. When a student is selected in the listbox, age and marks information of the student must be fetched in two fields. The user can edit the data in these two fields. Add a **Submit** button to update the database with the edited information in the fields.
- g) Create a home page for the application to contain two buttons **Add** and **Update**. When these buttons are clicked the application must redirect the user to the corresponding forms.
- h) Update the `package.json` file to include the main application name. Also, include the `engines` and `node` fields.



SESSION 12

NODE.JS IN ACTION (APPLICATION DEVELOPMENT) - II

Learning Objectives

In this session, students will learn to:

- List the steps to upload the Web application to the GitHub repository
- Explain the procedure to deploy the Web application on Render from a GitHub repository

This session carries forward the creation and deployment of the movie Web application from the previous session. It explains additional steps to create pages for the Web application and covers in detail how to deploy the **Wonderland Widescreen** Node.js application using Render.

12.1 Creating More Pages for the Web Application

More pages that are to be created for the Web application are as follows:

- delete-movie-form.html
- update-seats-form.html
- admin-login.ejs
- login.ejs

Let us create these pages and place them in relevant folders for the application to run properly.

➤ **delete-movie-form.html**

Figure 12.1 shows the design for the delete-movie-form.html page.

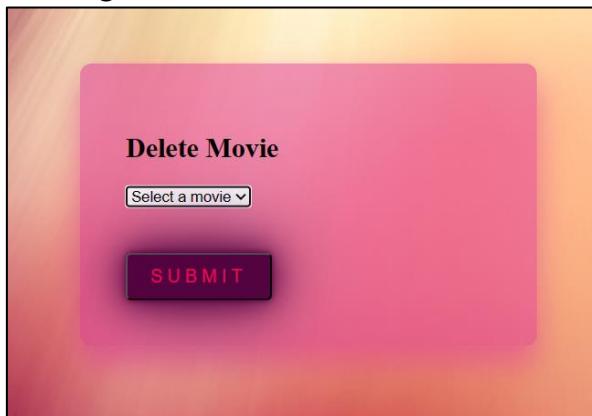


Figure 12.1: delete-movie-form.html

Code Snippet 1 shows the code for delete-movie-form.html.

Code Snippet 1:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>Delete Movie</title>
    <style>
      html {
        height: 100%;
      }
      body {
        margin: 0;
        padding: 0;
        background:
url("https://cdn.pixabay.com/photo/2018/02/27/15/40/backgroun
d-3185765_1280.jpg");
        background-attachment: fixed;
        background-position: relative;
        background-repeat: no-repeat;
        background-size: cover;
      }

      /* For the center container */
      .center-container {
        position: absolute;
        top: 50%;
        left: 50%;
        transform: translate(-50%, -50%);
      }
    </style>
  </head>
  <body>
    <div class="center-container">
      <h2>Delete Movie</h2>
      <form>
        <label>Select a movie <input type="button" value="▼" /></label>
        <br/>
        <input type="submit" value="SUBMIT" />
      </form>
    </div>
  </body>
</html>
```

```
/* login box */
.login-box {
    width: 400px;
    padding: 40px;
    background: rgba(226, 53, 157, 0.5);
    box-sizing: border-box;
    box-shadow: 0 15px 25px rgba(226, 53, 157, 0.5);
    border-radius: 10px;
}

.login-box h1 {
    margin: 0 0 30px;
    padding: 0;
    color: #520340;
    text-align: center;
}

/* User input box */
.login-box .user-box {
    position: relative;
}

.login-box .user-box input {
    width: 100%;
    padding: 10px 0;
    font-size: 16px;
    color: #fff;
    margin-bottom: 30px;
    border: none;
    border-bottom: 1px solid #fff;
    outline: none;
    background: transparent;
}

.login-box .user-box label {
    position: absolute;
    top: 0;
    left: 0;
    padding: 10px 0;
    font-size: 16px;
    color: #fff;
    pointer-events: none;
    transition: 0.5s;
}

.login-box .user-box input:focus ~ label,
.login-box .user-box input:valid ~ label {
    top: -20px;
    left: 0;
    color: #520340;
```

```

        font-size: 12px;
    }

    /* Submit button */
    .login-box-button {
        position: relative;
        display: inline-block;
        padding: 10px 20px;
        color: #ee0754;
        font-size: 16px;
        text-decoration: none;
        text-transform: uppercase;
        overflow: hidden;
        transition: 0.5s;
        margin-top: 40px;
        letter-spacing: 4px;
        background: #520340;
        border-radius: 5px;
        box-shadow: 0 0 5px #520340, 0 0 25px #520340, 0 0
50px #520340,
            0 0 100px #520340;
    }

    .login-box-button:hover {
        background: #fff;
        color: #520340;
    }

```

</style>

</head>

<body>

```

<div class="center-container">
    <div class="login-box">
        <h2>Delete Movie</h2>

        <form id="movie-form" action="/delete-movie"
method="POST">
            <div class="user-box">
                <!-- Dropdown for selecting a movie to delete -->
                <select id="movie-category" name="Movie name">
                    <option value="">Select a movie</option>
                </select>
            </div>
            <!-- Submit button to delete a movie -->
            <button type="submit" class="login-box-button">
                <span></span>
                <span></span>
                <span></span>
                <span></span>
                Submit
            </button>
        </form>

```

```

        </div>
    </div>
    <script>
        async function getMovieList() {
            // To get the movie category dropdown
            const movieCategoryDiv =
document.getElementById("movie-category");
            try {
                // To fetch the movie name from the database
                const response = await fetch(`/get-all-movies`);
                const data = await response.json();

                // To display the fetched movie name list
                displayMovieList(data);
            } catch (error) {
                // display error
                console.error("Error fetching movies:", error);
            }
        }

        // Function to display the movie list in the dropdown
        function displayMovieList(movies) {
            // To get the movie category dropdown
            const movieCategoryDiv =
document.getElementById("movie-category");

            if (movies.length === 0) {
                // Display a message if no movies are found in the
                // database
                movieCategoryDiv.textContent = `No movies found`;
            } else {
                // Fetch the movie name and display in the dropdown
                movies.forEach((movie) => {
                    let option = document.createElement("option");
                    option.setAttribute("value", movie["Movie
name"]);
                    let optionText =
document.createTextNode(movie["Movie name"]);
                    option.appendChild(optionText);

                    movieCategoryDiv.appendChild(option);
                });
            }

            movieCategoryDiv.style.display = "block";
            // Show the box when movies are available
        }
    }
    // Call the function to fetch movies when the
    // page loads
    document.addEventListener("DOMContentLoaded",
getMovieList());

```

```
</script>
</body>
</html>
```

The code in `delete-movie-form.html`:

- Uses HTML and JavaScript.
- Has `/delete-movie` as the endpoint.
- Uses a drop-down list to allow users select a movie name from the list.
- Defines the `getMovieList` function to fetch all movie names from the MongoDB server using the `/get-all-movies` endpoint.
- Defines the `displayMovieList` function to display the list of movie names in a drop-down list. This function creates `<option>` elements for each movie and appends them to the dropdown.
- Uses the `addEventListener` event listener method to call the `getMovieList` function when the page is loaded.

Save the code with the file name `delete-movie-form.html` in the `Templates` folder.

➤ **update-seats-form.html**

Figure 12.2 shows the design for the `update-seats-form.html` page.

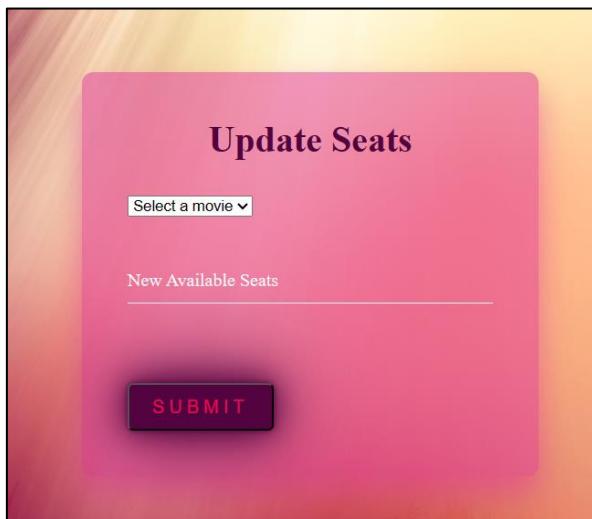


Figure 12.2: update-seats-form.html

Code Snippet 2 shows the code for `update-seats-form.html`. This form updates the movie seats in the database.

Code Snippet 2:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>Update Seats</title>

    <style>
      /* Height of the HTML document */
      html {
        height: 100%;
      }

      body {
        margin: 0;
        padding: 0;
        background:
url("https://cdn.pixabay.com/photo/2018/02/27/15/40/background-3185765_1280.jpg");
        background-attachment: fixed;
        background-position: relative;
        background-repeat: no-repeat;
        background-size: cover;
      }

      /* For the center container */
      .center-container {
        position: absolute;
        top: 50%;
        left: 50%;
        transform: translate(-50%, -50%);
      }

      /* Login box */
      .login-box {
        width: 400px;
        padding: 40px;
        background: rgba(226, 53, 157, 0.5);
        box-sizing: border-box;
        box-shadow: 0 15px 25px rgba(226, 53, 157, 0.5);
        border-radius: 10px;
      }

      /* Heading inside the login box */
      .login-box h1 {
        margin: 0 0 30px;
        padding: 0;
        color: #520340;
        text-align: center;
      }
    </style>
  </head>
  <body>
    <div class="center-container">
      <h1>Update Seats</h1>
      <form>
        <input type="text" placeholder="Enter Name" />
        <input type="text" placeholder="Enter Email" />
        <input type="text" placeholder="Enter Password" />
        <input type="submit" value="Update Seats" />
      </form>
    </div>
  </body>
</html>
```

```

/* User input box */
.login-box .user-box {
    position: relative;
}

/* User input */
.login-box .user-box input {

}

/* Label above the user input */
.login-box .user-box label {
    /* Label styling */
}

/* Button styling */
.login-box-button {

}

/* Hover effect for the button */
.login-box-button:hover {

}
</style>
</head>
<body>
    <div class="center-container">
        <div class="login-box">
            <h1>Update Seats</h1>

            <form id="seats-form" action="/update-seats"
method="POST">
                <!-- Dropdown for selecting a movie -->
                <div class="user-box">
                    <select id="movie-category" name="Movie name">
                        <option value="">Select a movie</option>
                    </select>
                </div>
                <br /><br />

                <!-- Input for entering new available seats -->
                <div class="user-box">
                    <input
                        type="number"
                        id="available-seats"
                        name="Available Seats"
                        required=""
                    />
                    <label>New Available Seats</label>
                </div>

                <!-- Submit button -->

```

```

<button type="submit" class="login-box-button">
    <span></span>
    <span></span>
    <span></span>
    <span></span>
    Submit
</button>
</form>
</div>
</div>

<script>
    // Function to fetch movie name from database
    async function getMovieList() {
        const movieCategoryDiv =
document.getElementById("movie-category");
        try {
            const response = await fetch(`/get-all-movies`);
            const data = await response.json();
            displayMovieList(data);
        } catch (error) {
            console.error("Error fetching movies:", error);
        }
    }

    // Function to display the list of movie names
    // in the dropdown
    function displayMovieList(movies) {
        const movieCategoryDiv =
document.getElementById("movie-category");

        if (movies.length === 0) {
            movieCategoryDiv.textContent = `No movies found`;
        } else {
            movies.forEach((movie) => {
                let option = document.createElement("option");
                option.setAttribute("value", movie["Movie
name"]);
                let optionText =
document.createTextNode(movie["Movie name"]);
                option.appendChild(optionText);

                movieCategoryDiv.appendChild(option);
            });
        }
        movieCategoryDiv.style.display = "block";
        // Show the box when movies are available
    }
}

```

```

        // Call the function to fetch movies when
        // the page loads
        document.addEventListener("DOMContentLoaded",
getMovieList());
    </script>
</body>
</html>

```

The update-seats-form.html form:

- Uses HTML and JavaScript.
- Has /update-seats as the endpoint.
- Uses a drop-down list to select a movie.
- Uses an input field to accept available seats.
- Defines the getMovieList function to fetch movie names from the database and uses the /get-all-movies endpoint. This function uses the displayMovieList function to display movie names in a dropdown.
- Defines the displayMovieList(movies) function to display the movie names fetched from the database in a drop-down.

Save the code with the file name update-seats-form.html in the Templates folder.

12.2 Adding Home Page With User Authentication

The application developed so far must be anchored using the home page, wonderland.html. This page redirects the user to a specific page depending on whether the user is a guest or an administrator.

➤ **wonderland.html**

Figure 12.3 shows the design of the wonderland.html page.



Figure 12.3: wonderland.html

Code Snippet 3 shows the code for wonderland.html page.

Code Snippet 3:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Movie List</title>
</head>
<style>
html {
    height: 100%;
}
body {
    text-align: center;
    background:
url("https://cdn.pixabay.com/photo/2017/01/12/17/30/warm-
and-cozy-1975215_1280.jpg");
    background-attachment: fixed;
    background-position: relative;
    background-repeat: no-repeat;
    background-size: cover;
    height: 100%;
    margin: 0;
    place-items: center;
    position: fixed;
    font-size: 20px;
}
.container {
    top: 30%;
    left: 50%;
    width: 50em;
    height: 18em;
    transform: translate(-50%, -50%);
    position: fixed;
    color: rgb(255, 255, 255);
    font-size: 20px;
}
.button {
    background-color: #000;
    border: .5px solid crimson;
    border-radius: 10px;
    color: #fff;
    padding: 8px;
    font-size: 20px;
    cursor:pointer; /* Pointer/hand icon */
    /* display:block; */
}
</style>
```

```

<body>
<div class="container">
    <h1>Welcome to the Wonderland Widescreen</h1>
    <br><br><br>
    <a href="/views/admin-login.ejs"><button
class="button"> Admin Login</button></a>
    <a href="/views/login.ejs"><button class="button">
Guest Login</button></a>
</div>
</body>
</html>

```

This is a simple HTML page that displays two buttons, one each for the administrator and the guest logins. Save the code with the file name `wonderland.html` in the `Templates` folder.

If the user logs in as an administrator, the endpoint will be `admin-login.ejs`. However, if the user logs in as a guest, the endpoint will be `login.ejs`. These files must be created and stored in the `views` folder.

The `.ejs` files are used to provide the HTML markups by embedding the required JavaScript code in the main Node.js application file, `movie-list.js`.

➤ **admin-login.ejs**

Figure 12.4 shows the design of the `admin-login.ejs` page.

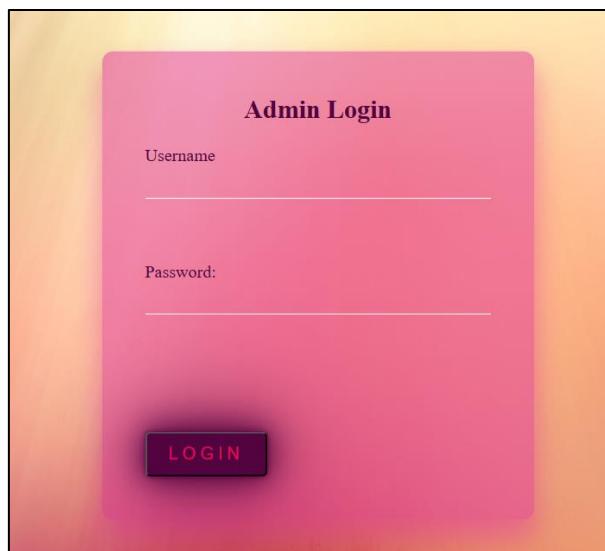


Figure 12.4: admin-login.ejs

Code Snippet 4 shows the code for the `admin-login.ejs` page.

Code Snippet 4:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width,
initial-scale=1.0" />
    <title>Admin Login</title>
    <style>
      html {
        height: 100%;
      }

      body {
        margin: 0;
        padding: 0;
        background:
url("https://cdn.pixabay.com/photo/2018/02/27/15/40/background-3185765_1280.jpg");
        background-attachment: fixed;
        background-position: relative;
        background-repeat: no-repeat;
        background-size: cover;
      }

      .login-box {
        position: absolute;
        top: 50%;
        left: 50%;
        width: 400px;
        padding: 40px;
        transform: translate(-50%, -50%);
        background: rgba(226, 53, 157, 0.5);
        box-sizing: border-box;
        box-shadow: 0 15px 25px rgba(226, 53, 157, 0.5);
        border-radius: 10px;
      }

      .login-box h2 {
        margin: 0 0 30px;
        padding: 0;
        color: #520340;
        text-align: center;
      }

      .user-box {
        position: relative;
        margin-bottom: 20px;
      }
    </style>
  </head>
  <body>
    <div class="login-box">
      <h2>Admin Login</h2>
      <form>
        <div>
          <label>Username:</label>
          <input type="text" name="username" value="admin" required>
        </div>
        <div>
          <label>Password:</label>
          <input type="password" name="password" value="password" required>
        </div>
        <div>
          <input type="checkbox" name="remember"> Remember Me
        </div>
        <div>
          <input type="submit" value="Login" style="background-color: #520340; color: white; border: none; padding: 10px; border-radius: 5px; font-weight: bold; width: 100%; height: 40px;">
        </div>
      </form>
    </div>
  </body>
</html>
```

```
.user-box input {
    width: 100%;
    padding: 10px 0;
    font-size: 16px;
    color: #fff;
    margin-bottom: 30px;
    border: none;
    border-bottom: 1px solid #fff;
    outline: none;
    background: transparent;
}

.user-box label {
    position: absolute;
    top: 0;
    left: 0;
    padding: 10px 0;
    font-size: 20px; /* Adjust the font size here */
    color: #fff;
    pointer-events: none;
    transition: 0.5s;
}

.user-box input:focus ~ label,
.user-box input:valid ~ label,
.user-box input:not(:placeholder-shown) ~ label {
    top: -20px;
    font-size: 16px; /* Adjust the font size here */
    color: #520340;
}

.login-box-button {
    position: relative;
    display: inline-block;
    padding: 10px 20px;
    color: #ee0754;
    font-size: 16px;
    text-decoration: none;
    text-transform: uppercase;
    overflow: hidden;
    transition: 0.5s;
    margin-top: 40px;
    letter-spacing: 4px;
    background: #520340;
    border-radius: 5px;
    box-shadow: 0 0 5px #520340, 0 0 25px #520340, 0 0
50px #520340,
        0 0 100px #520340;
}

.login-box-button:hover {
```

```

        background: #fff;
        color: #520340;
    }
</style>
</head>
<body>
    <div class="center-container">
        <div class="login-box">
            <h2>Admin Login</h2>
            <!-- Add a form for admin login -->
            <form id="loginForm" action="/admin-login"
method="POST">
                <div class="user-box">
                    <input type="text" id="username" name="username"
required />
                    <label for="username">Username</label>
                </div>
                <br />
                <div class="user-box">
                    <input type="password" id="password"
name="password" required />
                    <label for="password">Password:</label>
                </div>
                <br />
                <button type="submit" class="login-box-
button">Login</button>
            </form>
        </div>
    </div>
</body>
</html>

```

The `admin-login.ejs` file:

- Uses `/admin-login` as the endpoint.
- Uses two input fields one for accepting the username and another for accepting the password.
- Uses a submit button to transfer the execution to the endpoint.

Save the code with the file name `admin-login.ejs` in the `views` folder.

➤ **login.ejs**

Figure 12.5 shows the design of the `login.ejs` page.

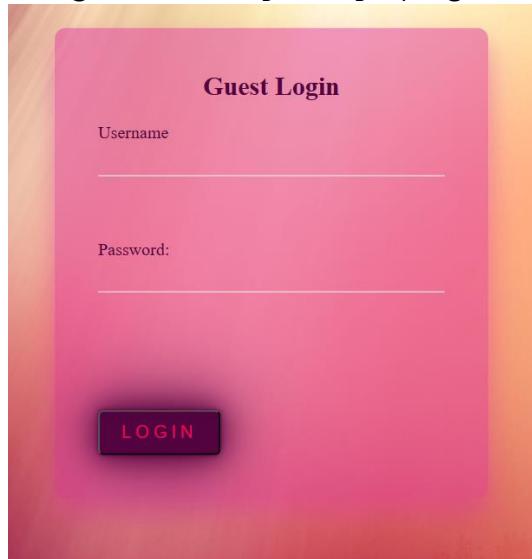


Figure 12.5: login.ejs

Code Snippet 5 shows the code for `login.ejs` page.

Code Snippet 5:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width,
initial-scale=1.0" />
    <title>Guest Login</title>
    <style>
      html {
        height: 100%;
      }

      body {
        margin: 0;
        padding: 0;
        background:
url("https://cdn.pixabay.com/photo/2018/02/27/15/40/backgroun
d-3185765_1280.jpg");
        background-attachment: fixed;
        background-position: relative;
        background-repeat: no-repeat;
        background-size: cover;
      }
    </style>
  </head>
  <body>
    <div>
      <h2>Guest Login</h2>
      <form>
        <label>Username</label>
        <input type="text" />
        <br/>
        <label>Password:</label>
        <input type="password" />
        <br/>
        <input type="submit" value="Login" />
      </form>
    </div>
  </body>
</html>
```

```
.login-box {
    position: absolute;
    top: 50%;
    left: 50%;
    width: 400px;
    padding: 40px;
    transform: translate(-50%, -50%);
    background: rgba(226, 53, 157, 0.5);
    box-sizing: border-box;
    box-shadow: 0 15px 25px rgba(226, 53, 157, 0.5);
    border-radius: 10px;
}

.login-box h2 {
    margin: 0 0 30px;
    padding: 0;
    color: #520340;
    text-align: center;
}

.user-box {
    position: relative;
    margin-bottom: 20px;
}

.user-box input {
    width: 100%;
    padding: 10px 0;
    font-size: 16px;
    color: #fff;
    margin-bottom: 30px;
    border: none;
    border-bottom: 1px solid #fff;
    outline: none;
    background: transparent;
}

.user-box label {
    position: absolute;
    top: 0;
    left: 0;
    padding: 10px 0;
    font-size: 20px; /* Adjust the font size here */
    color: #fff;
    pointer-events: none;
    transition: 0.5s;
}

.user-box input:focus ~ label,
.user-box input:valid ~ label,
.user-box input:not(:placeholder-shown) ~ label {
```

```

        top: -20px;
        font-size: 16px; /* Adjust the font size here */
        color: #520340;
    }

    .login-box-button {
        position: relative;
        display: inline-block;
        padding: 10px 20px;
        color: #ee0754;
        font-size: 16px;
        text-decoration: none;
        text-transform: uppercase;
        overflow: hidden;
        transition: 0.5s;
        margin-top: 40px;
        letter-spacing: 4px;
        background: #520340;
        border-radius: 5px;
        box-shadow: 0 0 5px #520340, 0 0 25px #520340, 0 0
50px #520340,
        0 0 100px #520340;
    }

    .login-box-button:hover {
        background: #fff;
        color: #520340;
    }

```

</style>

</head>

<body>

```

<div class="center-container">
    <div class="login-box">
        <h2>Guest Login</h2>
        <!-- Add a form for guest login -->
        <form id="loginForm" action="/user-local"
method="POST">
            <div class="user-box">
                <input type="text" id="username" name="username"
required />
                <label for="username">Username</label>
            </div>
            <br />
            <div class="user-box">
                <input type="password" id="password"
name="password" required />
                <label for="password">Password:</label>
            </div>
            <br />
            <button type="submit" class="login-box-
button">Login</button>

```

```
</form>
</div>
</div>
</body>
</html>
```

The `login.ejs` file:

- Uses `/user-local` as the endpoint.
- Uses two input fields one for accepting the username and another for accepting the password.
- Uses a submit button to transfer the execution to the endpoint.

Save the code with the file name `login.ejs` in the `views` folder.

12.3 Updating the Main Application File

Finally, update the main Node.js application file, `movie-list.js`. This file acts as the main entry point in the movie application. In this file, the core logic of the application is handled. Routes and middleware are also defined.

➤ **movie-list.js**

Consider the code in Code Snippet 6 for `movie-list.js` file. Add the functionality for user authentication, update, and delete. Note that the code to be added is given in bold.

Code Snippet 6:

```
const express = require("express");
const { MongoClient } = require("mongodb");
const bodyParser = require("body-parser");
const express = require("express");
const { MongoClient } = require("mongodb");
const bodyParser = require("body-parser");
const session = require("express-session");
const passport = require("passport");
const mongoose = require('mongoose');
const LocalStrategy = require("passport-local");
const path = require("path"); // Import the path module
// const uri =
// "mongodb+srv://prachigupta8585:prachi8585@moviecollectioncluster.4x9rauo.mongodb.net/moviecollectioncluster?retryWrites=true&w=majority"; //MongoDB Atlas Link

//Connect to MongoDB using MongoDB Compass localhost
//const uri = 'mongodb://127.0.0.1:27017/Movie'; // MongoDB connection URI
```

```

const client = new MongoClient(uri);
const app = express();

app.set("view engine", "ejs");

// Use session middleware
app.use(
  session({
    secret: "abc",
    resave: false,
    saveUninitialized: true,
  })
);

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

// Initialize passport middleware
app.use(passport.initialize());
app.use(passport.session());

// Middleware to check if the user is authenticated
function isAuthenticated(req, res, next) {
  if (req.session && req.session.authenticated) {
    return next();
  } else {
    res.redirect("/login");
  }
}

// Add the admin local strategy
passport.use(
  "admin-local",
  new LocalStrategy(function (username, password, done) {
    if (username === "Admin" && password === "12345") {
      return done(null, { username: "Aptech" });
    }
    return done(null, false, {
      message: "Incorrect admin username or password",
    });
  })
);

passport.serializeUser(function (user, done) {
  done(null, user);
});

passport.deserializeUser(function (user, done) {
  done(null, user);
});

// Add the User name and the password for the guest login

```

```

const users = [
  { id: 1, username: "abc", password: "123" },
  { id: 2, username: "user1", password: "user" },
];

passport.use(
  "user-local",
  new LocalStrategy(function (username, password, done) {
    const user = users.find((u) => u.username === username);
    if (!user) {
      return done(null, false, { message: "Incorrect
username." });
    }
    if (user.password !== password) {
      return done(null, false, { message: "Incorrect
password." });
    }
    return done(null, user);
  })
);

passport.serializeUser(function (user, done) {
  done(null, user.id);
});

passport.deserializeUser(function (id, done) {
  const user = users.find((u) => u.id === id);
  done(null, user);
});

async function main() {
  try {
    await client.connect();
    console.log("Connected to MongoDB");

    const database = client.db();
    const collection =
database.collection("MovieCollection"); // Adjust the
collection name

    // Set up the views folder
    app.set("views", path.join(__dirname, "views"));

    app.get("/", (req, res) => {
      console.log("Received request for /");
      // Serve the HTML page for displaying the login page
for admin and guest
      res.sendFile(__dirname + "/Templates/wonderland.html");
    });

    // Admin get and post
  }
}

```

```

// Admin login route
app.get("/views/admin-login.ejs", function (req, res) {
  console.log("entered into admin-login page");
  res.render("admin-login");
});

// Admin login form
app.post(
  "/admin-login",
  passport.authenticate("admin-local", {
    // successRedirect: path.join(__dirname, 'Templates',
'add-movie-form.html'),
    successRedirect: "/admin-dashboard",
    failureRedirect: "/admin-error",
  })
);

// Admin error route
app.get("/admin-error", function (req, res) {
  console.log("getting an admin-error page");
  res.send(
    '<script>alert("Incorrect Admin username or
password"); window.location.href = "/";</script>'
  );
});

// Admin dashboard route
app.get("/admin-dashboard", function (req, res) {
  console.log("Entered into admin dashboard page");
  res.sendFile(__dirname + "/Templates/movie-list.html");
});

// User login route
app.get("/views/login.ejs", function (req, res) {
  console.log("Entered into user-login page.");
  res.render("login");
});

// User login form
app.post(
  "/user-local",
  passport.authenticate("user-local", {
    // successRedirect: path.join(__dirname, 'Templates',
'add-movie-form.html'),
    successRedirect: "/user-dashboard",
    failureRedirect: "/user-error",
  })
);

// User error route
app.get("/user-error", function (req, res) {

```

```

        console.log("Entered into user login error.");
        // res.render("admin-error", { errorMessage: "Incorrect
admin username or password" });
        // Display an alert on the same page
        res.send(
            '<script>alert("Incorrect username or password");
window.location.href = "/";</script>'
        );
    });

    // user dashboard route
    app.get("/user-dashboard", function (req, res) {
        console.log("Entered into user dashboard page");
        res.sendFile(__dirname + "/Templates/book-seats-
form.html");
    });

    // Serve add-movie-form.html
    app.get("/add-movie-form.html", (req, res) => {
        console.log("Received request for /add-movie-
form.html");
        // Serve the HTML form
        res.sendFile(__dirname + "/Templates/add-movie-
form.html");
    });

    // Serve book-seats-form.html
    app.get("/book-seats-form.html", (req, res) => {
        res.sendFile(__dirname + "/Templates/book-seats-
form.html");
    });

    // Serve delete-movie-form.html
    app.get("/delete-movie-form.html", (req, res) => {
        res.sendFile(__dirname + "/Templates/delete-movie-
form.html");
    });

    // Serve update-seats-form.html
    app.get("/update-seats-form.html", (req, res) => {
        res.sendFile(__dirname + "/Templates/update-seats-
form.html");
    });

    app.get("/get-movies", async (req, res) => {
        const category = req.query.category; // Get the
selected category from query parameters

        try {
            // Fetch movies of the selected category from the
database

```

```

        const movies = await collection.find({ Category:
category }).toArray();
        res.status(200).json(movies);
    } catch (error) {
        console.error("Error fetching movies:", error);
        res.status(500).json({ error: "Failed to fetch
movies" });
    }
});

app.get("/get-all-movies", async (req, res) => {

    try {
        // Fetch movies of the selected category from the
database
        const movies = await collection.find().toArray();
        res.status(200).json(movies);
    } catch (error) {
        console.error("Error fetching movies:", error);
        res.status(500).json({ error: "Failed to fetch
movies" });
    }
};

app.get("/get-movie-details", async (req, res) => {
    const movieName = req.query.name; // Get the selected
movie name from query parameters

    try {
        // Fetch movie details based on the selected movie
name from the database
        const movie = await collection.findOne({ "Movie
name": movieName });
        if (movie) {
            res.status(200).json({
                Description: movie["Description"],
                Actors: movie["Actors"],
            });
        } else {
            res.status(404).json({ error: "Movie not found" });
        }
    } catch (error) {
        console.error("Error fetching movie details:",
error);
        res.status(500).json({ error: "Failed to fetch movie
details" });
    }
};

// Handle the form submission and add a movie
app.post("/add-movie", async (req, res) => {

```

```

        try {
            // Insert a new movie document into the MongoDB
            collection
            await collection.insertOne(req.body);
            console.log("Added movie to the database");
            // Display an alert on the same page and redirect to
            admin dashboard
            res.send('<script>alert("Movie added successfully");
            window.location.href = "/admin-dashboard";</script>');
        } catch (error) {
            console.error("Error adding movie:", error);
            // Render an error message on the same page
            res.status(500).send("<h2>Failed to add the
            movie</h2>");
        }
    });

// Handle booking seats
app.post("/book-seats", async (req, res) => {
    try {
        // This checks if the user is an admin or a guest based
        on the authentication
        const isAdmin = req.isAuthenticated() &&
        req.user.username === "Aptech";

        // Retrieve movie information from the request
        const movieNameToBook = req.body["Movie name"];
        const seatsToBook = parseInt(req.body["seats-to-book"]);

        // Check if the movie exists
        const existingMovie = await collection.findOne({ "Movie
        name": movieNameToBook });

        if (!existingMovie) {
            console.log("Movie not found in the database.");
            return res.send("Movie not found in the database.");
        }

        // Retrieve the available seats from the existing movie
        document
        const availableSeats = existingMovie["Available Seats"];
        //Check field name

        if (seatsToBook <= availableSeats) {
            // Calculate the updated available seats
            const updatedAvailableSeats = availableSeats -
            seatsToBook;

            // Update the document with new available seats
            const result = await collection.updateOne(
                { "Movie name": movieNameToBook },

```

```

        { $set: { "Available Seats": updatedAvailableSeats } }
    } //Check field name
);

// This will redirect to the route based on the users
// role admin or user using the conditional (ternary) operator
// concept
const redirectRoute = isAdmin ? '/admin-dashboard' :
'/user-dashboard';

if (result.modifiedCount === 1) {
    // Display an alert and redirect to the appropriate
route
    const alertMessage = `Booking successful for
${seatsToBook} seat(s) in ${movieNameToBook}`;
    return res.send(`

        <script>
            alert("${alertMessage}");
            window.location.href = "${redirectRoute}";
        </script>
    `);
} else {
    console.log("Failed to update available seats");
    const errorMessage = "Failed to update available
seats";
    // Display an alert on the same page
    return res.send(`

        <script>
            alert("${errorMessage}");
            window.location.href = "${redirectRoute}";
        </script>
    `);
}
} else {
    console.log(`Not enough seats available for
${seatsToBook} seat(s) in ${movieNameToBook}`);
    return res.send(`Not enough seats available for
${seatsToBook} seat(s) in ${movieNameToBook}`);
}
} catch (error) {
    console.error("Error booking seats:", error);
    return res.status(500).send("Failed to book seats");
}
});

// Handle delete movie
app.post("/delete-movie", async (req, res) => {
    const movieNameToDelete = req.body["Movie name"];

    try {
        // Check if the movie exists

```

```

const existingMovie = await collection.findOne({
  "Movie name": movieNameToDelete,
});

if (!existingMovie) {
  console.log("Movie not found in the database");
  res.send("Movie not found in the database");
} else {
  // Delete the movie from the collection
  const result = await collection.deleteOne({
    "Movie name": movieNameToDelete,
  });

  if (result.deletedCount === 1) {
    console.log("Movie deleted successfully");
    // Display an alert on the same page and redirect
    to admin dashboard
    res.send(
      '<script>alert("Movie deleted successfully");'
      window.location.href = "/admin-dashboard";</script>');
  } else {
    console.log("Failed to delete the movie");
    // Display an alert on the same page
    res.send(
      '<script>alert("Failed to delete the movie");'
      window.location.href = "/";</script>'
    );
  }
}

} catch (error) {
  console.error("Error deleting the movie:", error);
  res.status(500).send("Failed to delete the movie");
}
});

// Handle updating available seats
app.post("/update-seats", async (req, res) => {
  const movieNameToUpdate = req.body["Movie name"];
  const newAvailableSeats = parseInt(req.body["Available
  Seats"]); // Make sure the field name matches your MongoDB
  collection

  try {
    // Check if the movie exists
    const existingMovie = await collection.findOne({
      "Movie name": movieNameToUpdate,
    });

    if (!existingMovie) {
      console.log("Movie not found in the database");
      // Display an alert on the same page
    }
  }
});

```

```

        res.send(
            '<script>alert("Movie not found in the
database"); window.location.href = "/";
</script>'
        );
    } else {
        // Update the available seats for the movie
        console.log("Existing movie:", existingMovie);

        const result = await collection.updateOne(
            { _id: existingMovie._id }, // Use _id to
        uniquely identify the movie
            { $set: { "Available Seats": newAvailableSeats }
        } // Make sure the field name matches your MongoDB collection
        );

        console.log("Update operation result:", result);

        if (result.modifiedCount === 1) {
            const alertMessage = `Updated available seats for
${movieNameToUpdate} successfully`;
            console.log(alertMessage);
            // Display an alert on the same page and redirect
            to admin dashboard
            res.send(
                '<script>alert("${alertMessage}");
window.location.href = "/admin-dashboard";
</script>');
            } else {
                console.log("Failed to update available seats");
                res.status(500).send("Failed to update available
seats");
            }
        }
    } catch (error) {
        console.error("Error updating available seats:",
error);
        res.status(500).send("Failed to update available
seats");
    }
});

//logout
app.get("/logout", function (req, res) {
    console.log("Logout page activated.");
    res.sendFile(__dirname + "/Templates/wonderland.html");
});
} catch (error) {
    console.error("Error connecting to MongoDB:", error);
}
}

main().catch(console.error);

```

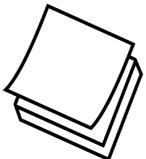
```
app.listen(process.env.PORT || 3000, process.env.IP ||
"0.0.0.0", () => {
  console.log("Server is running");
});
```

The movie-list.js file:

- Uses `app.use` to set up session middleware using the `express-session` package. Here,
 - `secret` is a key used to sign the session ID cookie.
 - `resave` and `saveUninitialized` are session options.
- Initializes and uses `passport` middleware for handling user authentication.
- Defines the `isAuthenticated` function to check for user authentication and redirect the user to the login page if the user is not authenticated.
- Uses the admin-local and user-local `passport` strategies to authenticate the credentials.
- Uses the `serialize` and `deserialize` functions to define how user objects are serialized into the session and deserialized from the session.
- Defines additional routes for handling HTTP requests. The routes are:
 - `/delete-movie`: This route is used for handling delete movie.
 - `/update-seats`: This route is used for handling updating available seats.
 - `/logout`: This route is used to logout from the application.
- Admin login routes:
 - `/views/admin-login.ejs`: This route is used to redirect to the admin login page.
 - `/admin-login`: This route is used to handle admin login form submission.
 - `/admin-error`: This route is used to handle login errors such as incorrect credentials.
 - `/admin-dashboard`: This route is used to redirect to the admin dashboard.
- User login routes:
 - `/views/login.ejs`: This route is used to redirect to the login page of guest.
 - `/user-local`: This route is used to handle guest form submission.
 - `/user-error`: This route is used to handle login errors such as incorrect credentials.

- /user-dashboard: This route is used to redirect to the guest dashboard.
- Manages the logout process by accessing the /logout route, which clears the session and redirects them to the home page.

Save the code. This completes the coding for the application.



To add the **Logout** button, in the movie-list.html page add this code before the ending of div tag:

```
<a href="/logout"><button class="button">Logout</button>
</a>
```

12.4 Uploading the Application to the GitHub Repository

To be accessible globally, the application must be deployed on the Internet. The GitHub repository serves as a global storage that hosts the application files. This repository can store codes, files, and version histories of applications stored on it. To upload the movie application to the GitHub repository, create an exclusive repository space for the application in GitHub.

Refer to Appendix B for a detailed description of how to:

- Create a repository in GitHub.
- Upload the application resources to the repository.

12.5 Deploying the Application on Render from a GitHub Repository

Render is a cloud computing platform that helps developers to deploy their applications on the cloud. Refer to session 10 to deploy the application using Render. Deploy the movie application uploaded to the GitHub repository using Render. When the deployment is complete, copy the URL from the Dashboard. The URL, <https://movie-application-o2ed.onrender.com> is copied to the clipboard.

12.6 Accessing the Application in Web Browser

The Wonderland Widescreen application is now deployed and is ready to be accessed in the Web browser using the URL.

Perform the given steps:

1. Open the Web browser and paste the copied URL. The home page of the movie application is displayed as shown in Figure 12.6.



Figure 12.6: Home Page of Movie Application

Guest Login

2. Click **Guest Login**.
3. The guest login form is displayed as shown in Figure 12.7.

A screenshot of a guest login form. The title "Guest Login" is at the top. Below it is a "Username" field containing "User1". Below that is a "Password" field containing "user", with a small eye icon to its right. At the bottom is a "LOGIN" button.

Figure 12.7: Guest Login Page

4. To provide the username, in the **Username** box, type **user1**.
5. To provide the password, in the **Password** box, type **user**, and click **Login**. The **Book Your Seats** page appears.
6. To book a movie, click **Action**, **Comedy**, or **Drama** button. The movie list appears as shown in Figure 12.8. Select a movie to see the description of the movie.

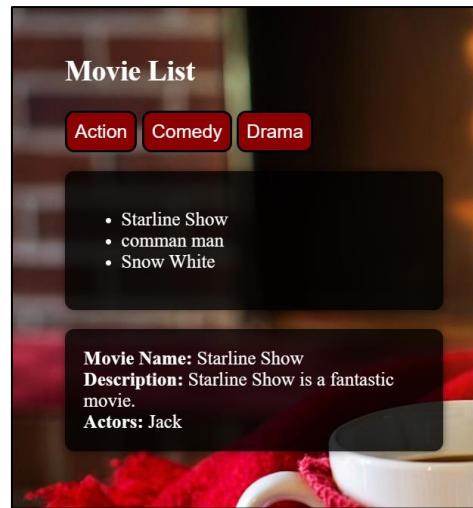


Figure 12.8: Book Your Seats - Movie List Section

7. Then, select a movie from the drop-down and enter the number of seats as shown in Figure 12.9.

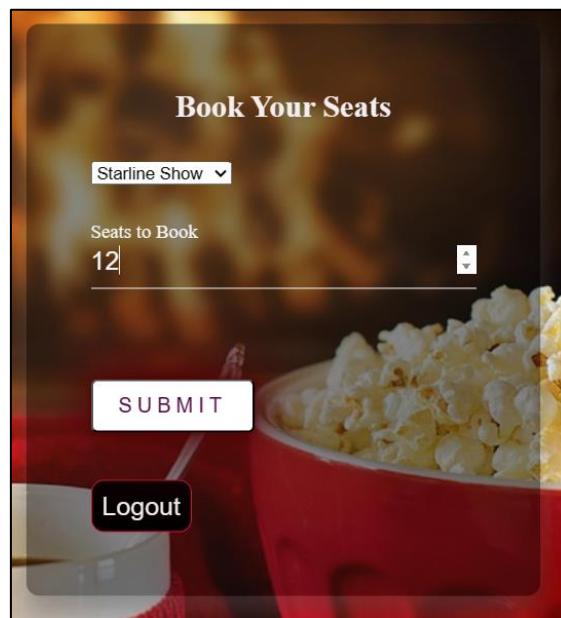


Figure 12.9: Book Your Seats

8. Click **SUBMIT**.

Figure 12.10 shows the message that will appear after the seats are booked and the data is updated to the database.

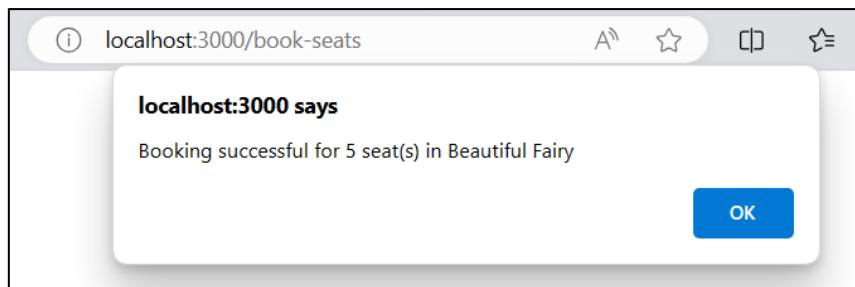


Figure 12.10: Booking Seats Confirmation

Admin Login

9. Click **Admin Login** as shown in earlier Figure 12.6.
The admin login form is displayed as shown in Figure 12.11.
10. To provide the username, in the **Username** box, type **Admin**.
11. To provide the password, in the **Password** box, type **12345**, and click **Login**.

A screenshot of an "Admin Login" form. The form has a pink-to-orange gradient background. It contains two input fields: "Username" with the value "Admin" and "Password" with the value "12345". Below the password field is a visibility icon. At the bottom is a dark purple "LOGIN" button.

Figure 12.11: Admin Login Form

Admin login is done successfully and the dashboard appears as shown in Figure 12.12.



Figure 12.12: Admin Login Dashboard

Add Movie

12. Now, to add a new movie click **Add Movie**. The **Add movie Details** form is displayed as shown in Figure 12.13.

Add Movie Details

Movie Name
Beautiful Fairy

Screen No
2

Show Time
11.00

Available Seats
120

Actors Name
Jack and Rose

Movie Description
Beautiful Fairy is a Drama movie.

Figure 12.13: Add Movie Details Page

13. Enter the details in the **Add Movie Details** page as shown in Figure 12.13.
14. Click **SUBMIT**.

Figure 12.14 shows the message that will appear after the movie is added to the database.

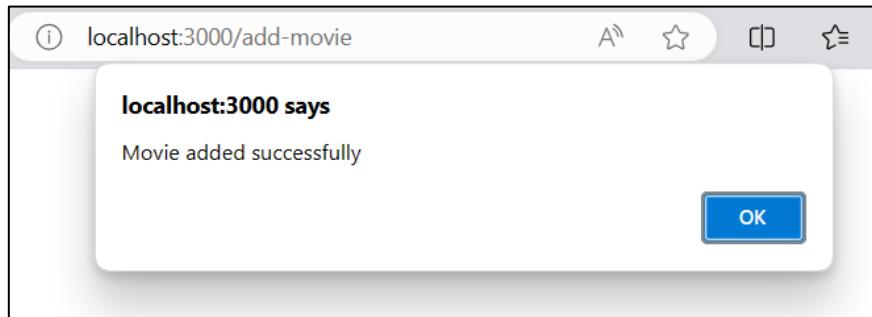


Figure 12.14: Add Movie Confirmation

Book Seats

15. To book a movie click **Book seats** as shown in earlier Figure 12.12.



Booking the seats functionality is the same for both guest and admin login.

Update Seats

16. To update movie seat, click **Update seats** as shown in earlier Figure 12.12.

Update Seats form is displayed as shown in Figure 12.15.

A screenshot of a web form titled "Update Seats". At the top left is a dropdown menu showing "Beautiful Fairy". Below it is a text input field labeled "New Available Seats" containing the value "50". To the right of the input field is a small up-and-down arrow indicating it's a dropdown. At the bottom center is a large blue "SUBMIT" button.

Figure 12.15: Update Seats Page

17. Enter the details in the **Update Seats** page as shown in Figure 12.15 and then click **SUBMIT**.

Figure 12.16 shows the message that will appear after the movie is updated in the database.

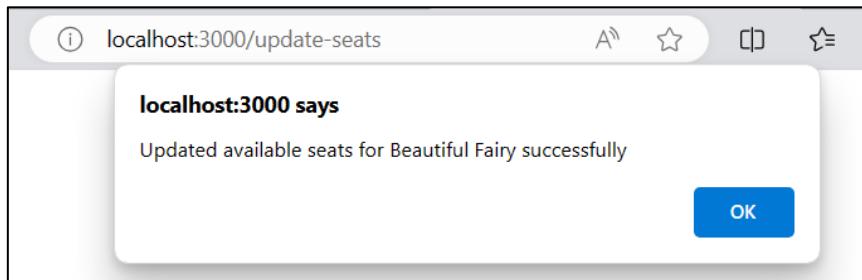


Figure 12.16: Update Movie Confirmation

Delete Movies

18. To delete a movie, click **Delete Movie** as shown in earlier Figure 12.12.

Delete Movie form is displayed as shown in Figure 12.17.

A screenshot of a "Delete Movie" form. The title "Delete Movie" is at the top. Below it is a dropdown menu containing the option "Beautiful Fairy". At the bottom is a "SUBMIT" button.

Figure 12.17: Delete Movie Page

19. Enter the details in the **Delete Movies** page as shown in Figure 12.17 and click **SUBMIT**.

Figure 12.18 shows the message that will appear after the movie is deleted from the database.

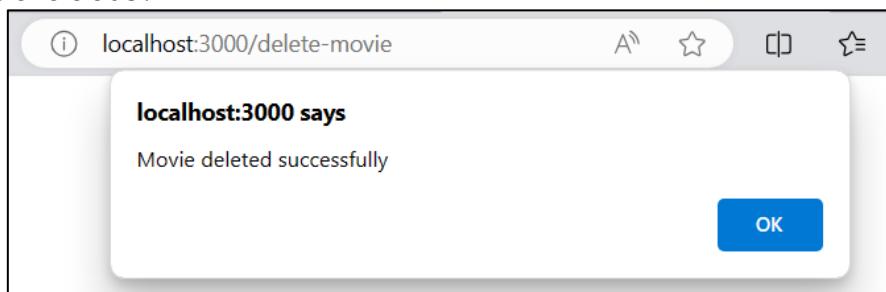


Figure 12.18: Delete Movie Confirmation

12.7 Summary

- Embedded JavaScript files are used to provide the HTML markups. The scripts required are embedded in the main Node.js application file.
- The `.ejs` files are stored in the `views` folder.
- Middleware for maintaining user session and authentication is added to the main Node.js application file using the `passport` middleware.
- `app.use` from the `express-session` package is used to set up session middleware.
- A Web application can be uploaded to the GitHub repository for better collaboration and version control.
- Render can be used to deploy Web applications on the cloud.

Test Your Knowledge

1. Which of the following accounts are required to deploy an application on Render?

- a) Gmail
- b) GitHub
- c) Node.js
- d) Express

2. Consider the given code snippet.

```
app.get('/', (req , res) => {
  const data = {
    Cricketer_Name: 'Chris Gayle',
    Cricketer_Hobby: 'Cricket',
  };
  res.render('index', data) ;
});
```

Which of the following code snippets will you use to display Cricketer_Name and Cricketer_Hobby in a .html file?

- a) <body>
 <h1> <%= Cricketer_Name %> </h1>
 <h2> <%= Cricketer_Hobby %> </h3>
 </body>
- b) <body>
 <h1> <% Cricketer_Name %> </h1>
 <h2> <% Cricketer_Hobby %> </h3>
 </body>
- c) <body>
 <h1> </= Cricketer_Name /> </h1>
 <h2> </= Cricketer_Hobby /> </h3>
 </body>
- d) <body>
 <h1> <%= &Cricketer_Name %> </h1>
 <h2> <%= &Cricketer_Hobby %> </h3>
 </body>

3. Which option will you choose on the dashboard to deploy a Web application on Render?

- a) Static Sites
- b) Private Services
- c) New Web Service
- d) Blueprints

4. Which command will you enter in the start command section while deploying a Web application on Render?

- a) npm file_name.js
- b) npm install
- c) npm file_name.ejs
- d) npm file_name.html

5. What does res.status(500) represent in Node.js?

- a) Completed successfully
- b) Not found
- c) Exceptional
- d) Internal server error

Answers to Test Your Knowledge

1	a, b
2	a
3	c
4	a
5	d

Try It Yourself

1. Refer to the **Student Management System** application developed in Session 11.
 - a) Create an HTML form to delete a student selected from a list of students. Add a listbox to list all the student names. Allow the user to select a student from the list. Add a **Delete** button to delete the selected student.
 - b) On the home page, add a button **Delete**. When this button is clicked the application must redirect the user to the delete form.
 - c) Implement user authentication using admin and guest login.
 - a. Create an admin login page. The user should be redirected to the admin dashboard.
 - b. Include the add, update, and delete functionality for the admin dashboard.
 - c. Create a student login page. The user should be redirected to the student dashboard.
 - d. Include only the update functionality for the student login from which the students can update marks.
 - d) Create a Github repository for the application.
 - e) Upload the application to the Github repository.
 - f) Deploy the application using Render.
 - g) Execute the application from a browser.

Appendix A

RETRIEVE CONNECTION STRING FROM MONGODB ATLAS

MongoDB Atlas is a cloud-based service that requires an account to access its servers. With an Atlas account, the Atlas dashboard can be accessed to deploy MongoDB clusters, create database users, and manage databases.

Create an Atlas Account

To create an Atlas account, use any one of the three options:

GitHub account

Google account

Email address



Google account is the preferred method. However, email address can also be used.

To create the Atlas account:

1. Open the browser and navigate to the URL:

<https://www.mongodb.com/cloud/atlas/register>

The **MongoDB Atlas Landing** page opens as shown in Figure 1. To sign up using the preferred method or Google account, scroll down and click **Sign up with Google**.

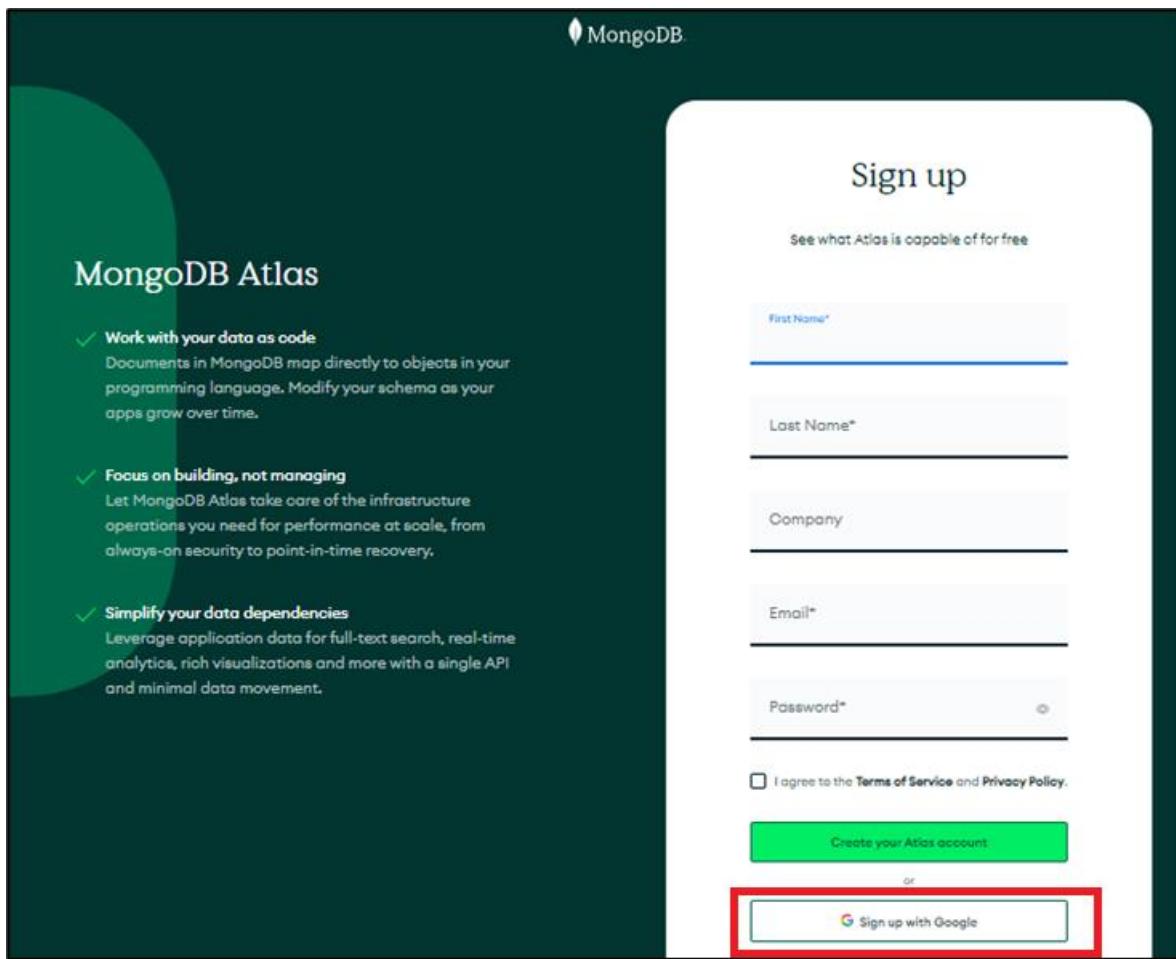


Figure 1: MongoDB Atlas Landing Page

2. The **Accept Privacy Policy and Terms of Service** page opens as shown in Figure 2. To understand how the data is collected, used, and shared, review the privacy policy and terms of service.

To proceed, select the **I accept the Privacy Policy and Terms of Service** check box.

Click **Submit**.

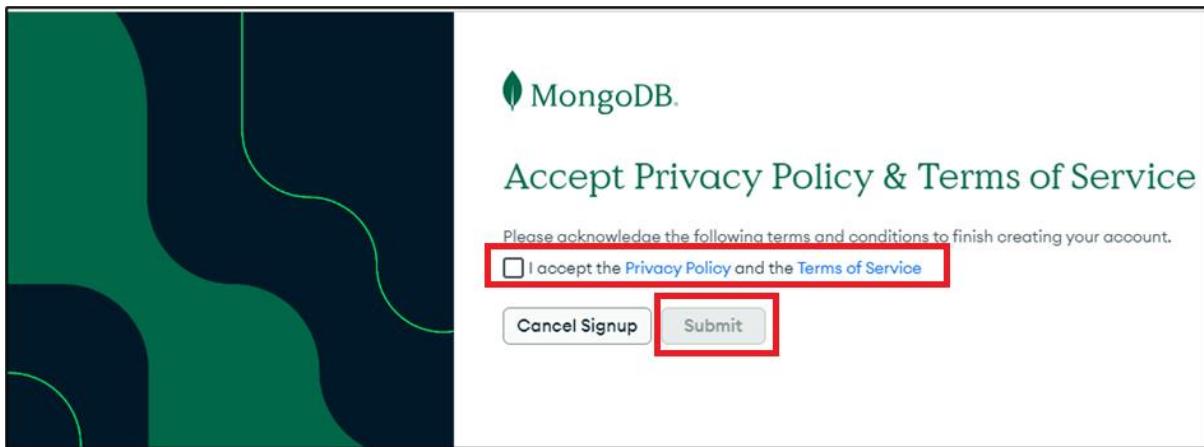


Figure 2: Accept Privacy Policy and Terms of Service Page

3. This completes the sign-up process and welcomes the user to the account as shown in Figure 3.

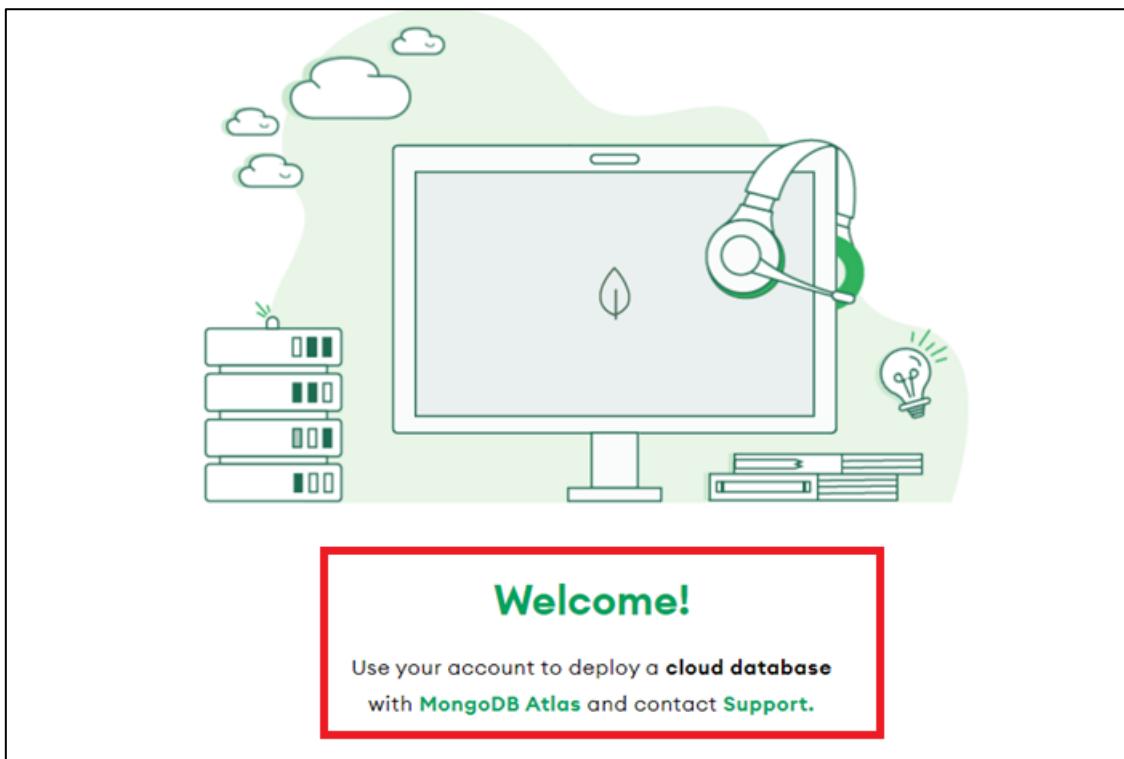


Figure 3: Welcome Page

4. Atlas automatically creates a default organization and a project. For first-time users, Atlas prompts them to respond to a few queries related to the user and the project as shown in Figure 4.

Respond to all the queries appropriately and click **Finish**. This completes the Atlas account creation process.

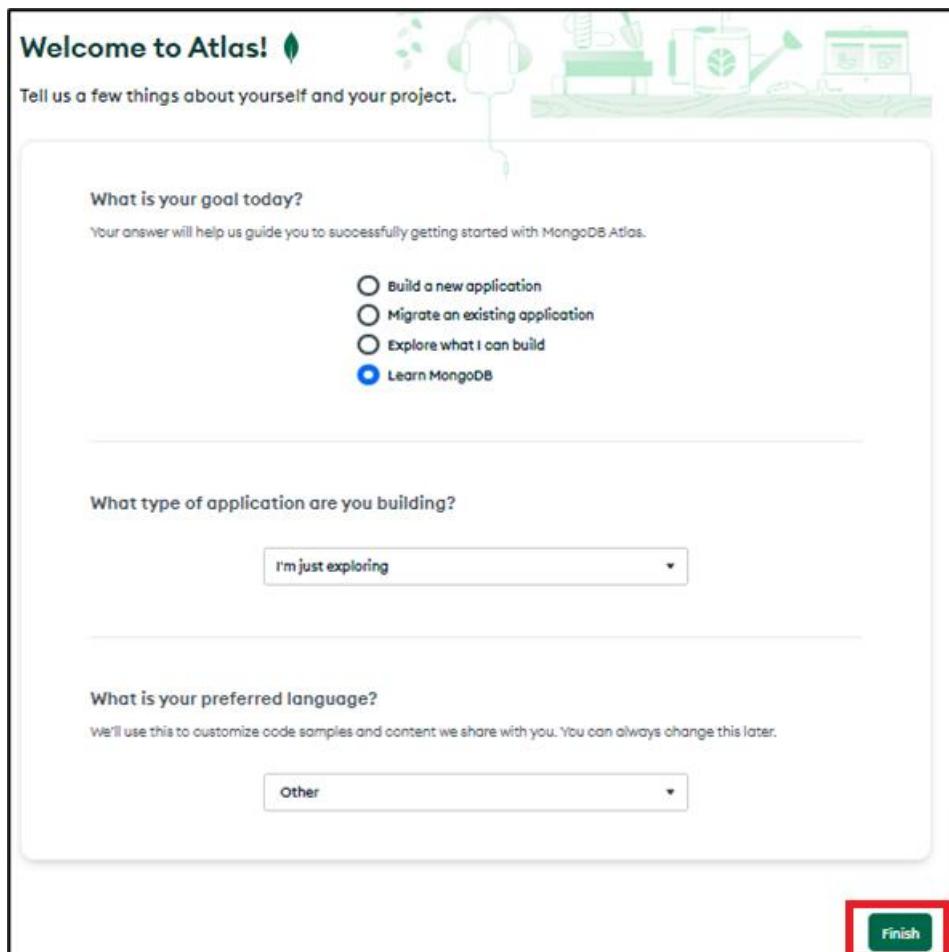


Figure 4: Welcome to Atlas Page

Deploy a Free Cluster

After creating an Atlas account, the next step is to create a cluster. A cluster is a group of servers that work together to store and manage data. To create a MongoDB cluster in Atlas, the user must specify the required size and configuration, and Atlas takes care of the rest.

M0 clusters are a free, entry-level option for MongoDB users who are learning the database or developing small, proof-of-concept applications. These clusters are limited in terms of storage and features, but they are a great way to get started with MongoDB without any upfront costs. These clusters are small-scale, but they never expire and they provide access to a subset of Atlas features.

To deploy M0 cluster:

1. From the **Deploy your database** page shown in Figure 5, select the **M0** option.

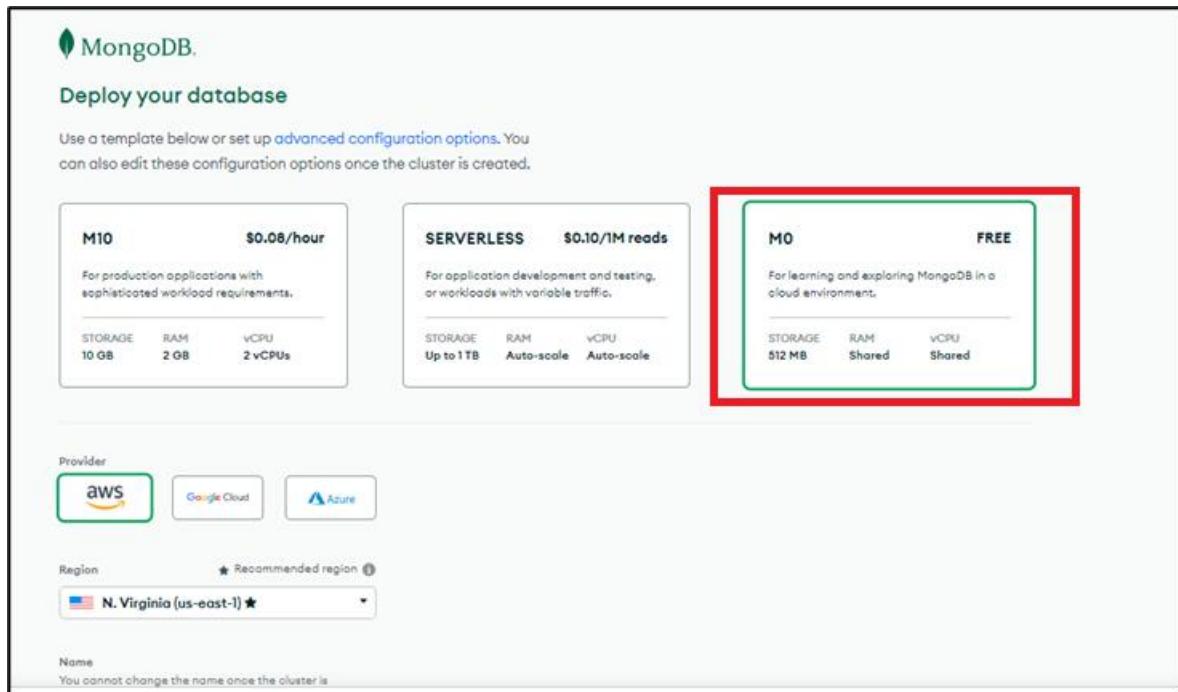


Figure 5: Deploy your database Page

2. Scroll down to make more selections as shown in Figure 6.
 - Select the preferred **Provider**.
Atlas supports M0 free clusters on Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure. In this case, the **AWS** option is selected.
 - Select the preferred **Region**.
Atlas displays only the cloud provider regions that support M0 free clusters. Here, **N. Virginia (us-east-1)** is selected.
 - Enter a name for the cluster in the **Name** box.
Users can specify any name for the cluster, as long as it contains American Standard Code for Information Interchange or ASCII letters, numbers, and hyphens. Retain **Cluster0** as given here.
 - Click **Create**.

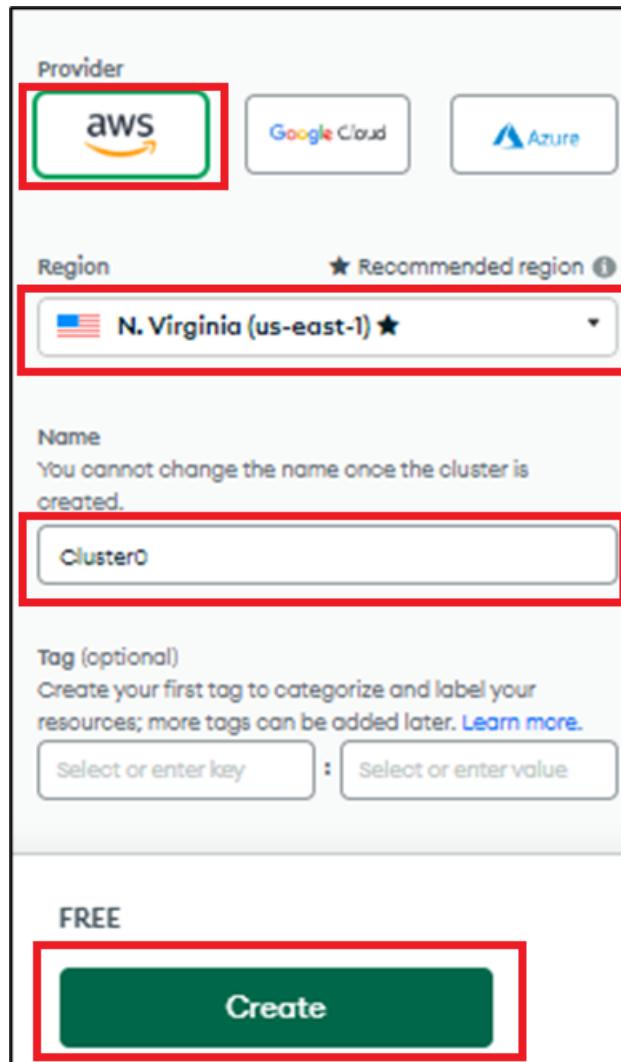


Figure 6: Create Cluster

3. The **Security Quickstart** page opens as shown in Figure 7 with the success message.

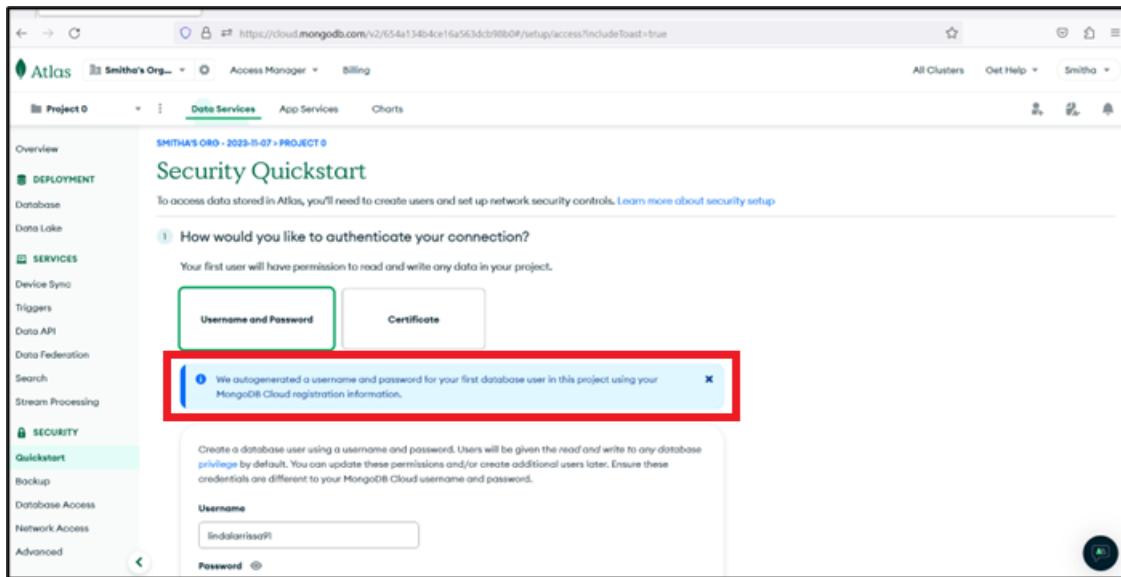


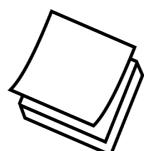
Figure 7: Success Message on the Security Quickstart Page

This completes the deployment of free cluster M0.

Create a Database User for the M0 Cluster

To protect data, Atlas requires users to authenticate as valid MongoDB database users before they can access the cluster. To allow users to access the MongoDB database on the cluster, a database user must be created after deploying the cluster. The database user can then be granted permission to access the cluster and the database hosted on the cluster. The database user accounts can be created for different users and roles, and each account can be granted different permissions.

There is a difference between database users and Atlas users in MongoDB Atlas:



A database user account can ONLY be created by a user who has either Organization Owner or Project Owner permission.

MongoDB Atlas autogenerateds a username and password for the first database user in the project using the already submitted MongoDB Cloud registration information.

To manually add a user account:

1. On the **Security Quickstart** page, as shown in Figure 8, enter the new username and credentials.
 - For this session, enter the username as **lindalarrissa91** and password as **linda91**. This username and password combination is used to grant a user the access to databases and collections in the cluster within the Atlas project.
 - Click **Create User**. This creates a database user.

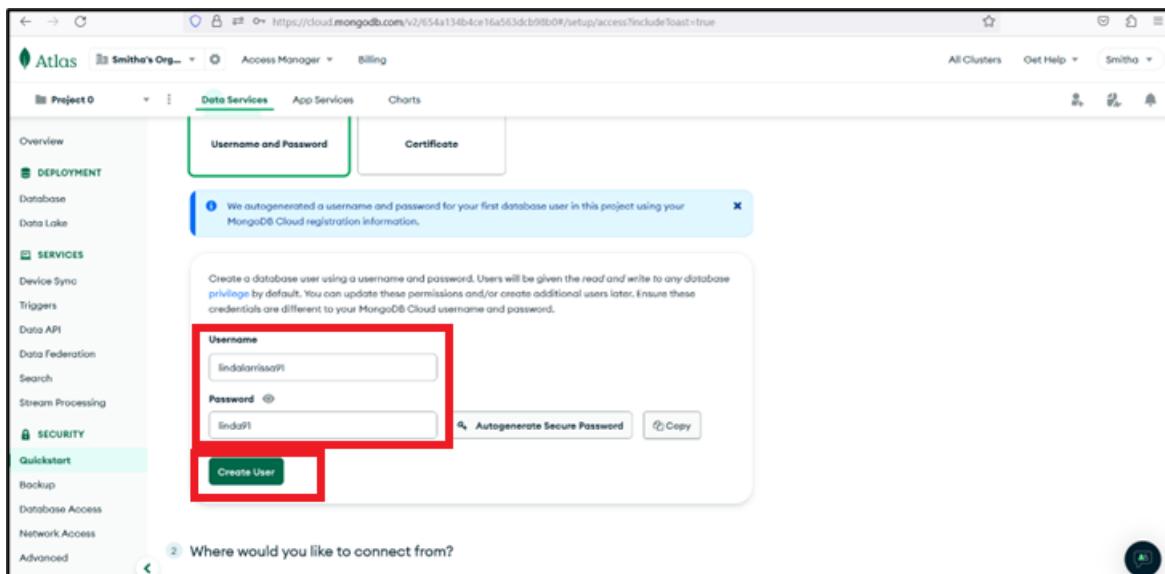


Figure 8: Create User

Add a Trusted IP Address to the Authorized IP Access List

After creating a database user, the deployed cluster is ready to be connected. Each device that connects to a network is assigned a unique identifier called an Internet Protocol or IP address. It is used to identify and locate devices on the network, and to route data between devices.

In MongoDB Atlas, an IP access list is a list of trusted IP addresses that are allowed to connect to a MongoDB Atlas cluster. To secure the MongoDB Atlas cluster, access should be restricted only to specified IP addresses. By limiting access to specific IPs, unauthorized connections can be prevented so as to ensure that only trusted sources can connect to the cluster.

To add an IP Address:

1. For this session, under **Add entries to your IP Access List**, in the **IP Address** box, enter **0.0.0.0/0**, and click **Add Entry** as shown in Figure 9.

The specified IP Address can be retrieved after deploying the application on the Render platform. Therefore, a temporary IP address is entered that allows connections from any network.

Click **Finish and Close**.

Add entries to your IP Access List

Only an IP address you add to your Access List will be able to connect to your project's clusters. You can manage existing IP entries via the [Network Access Page](#).

IP Address	Description
Enter IP Address 0.0.0.0/0	Enter description

Add My Current IP Address

Add Entry

IP Access List	Description
0.0.0.0/0	 EDIT REMOVE

Finish and Close

Figure 9: Add entries to your IP Access List Page

2. The **Congratulations on setting up access rules!** message box appears. Click **Go to Overview** as shown in Figure 10.

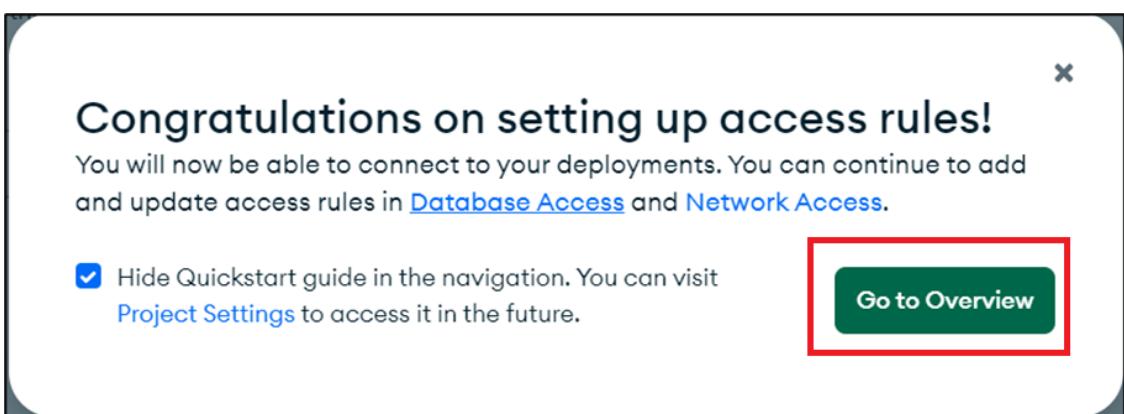


Figure 10: Congratulations on setting up access rules Message

Get the Connection String

To retrieve the connection string of the database deployment:

1. Click **Database** in the upper-left corner of the Atlas screen.

The **Database Deployments** page opens as shown in Figure 11. To deploy the desired deployment, click **Connect** for **Cluster0**.

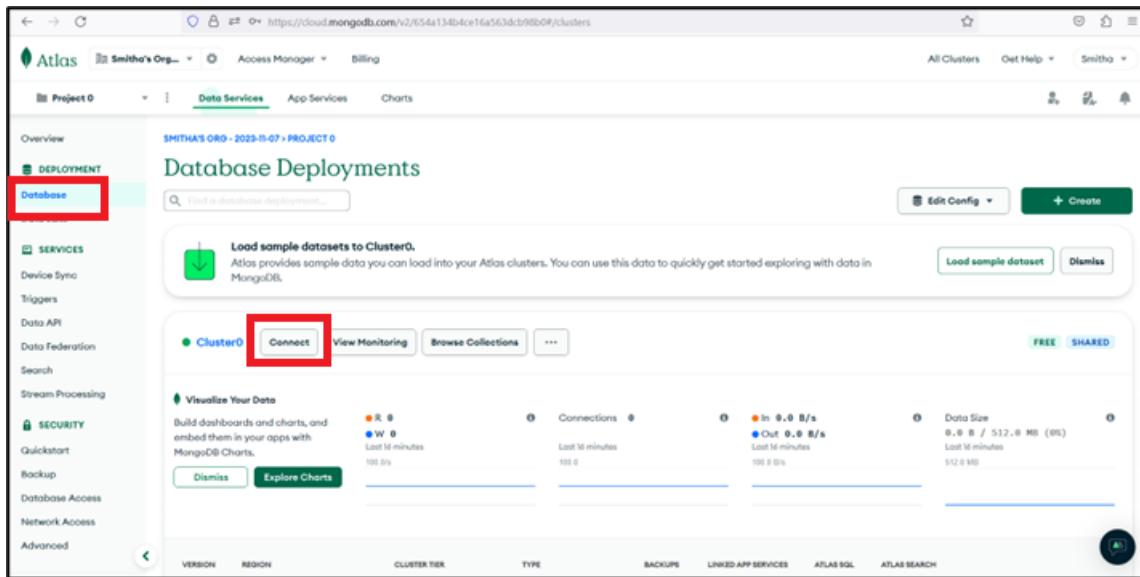


Figure 11: Database Deployments Page

2. The **Choose a connection method** page opens as shown in Figure 12. To connect using the MongoDB driver, under **Connect to your application**, click **Drivers**.

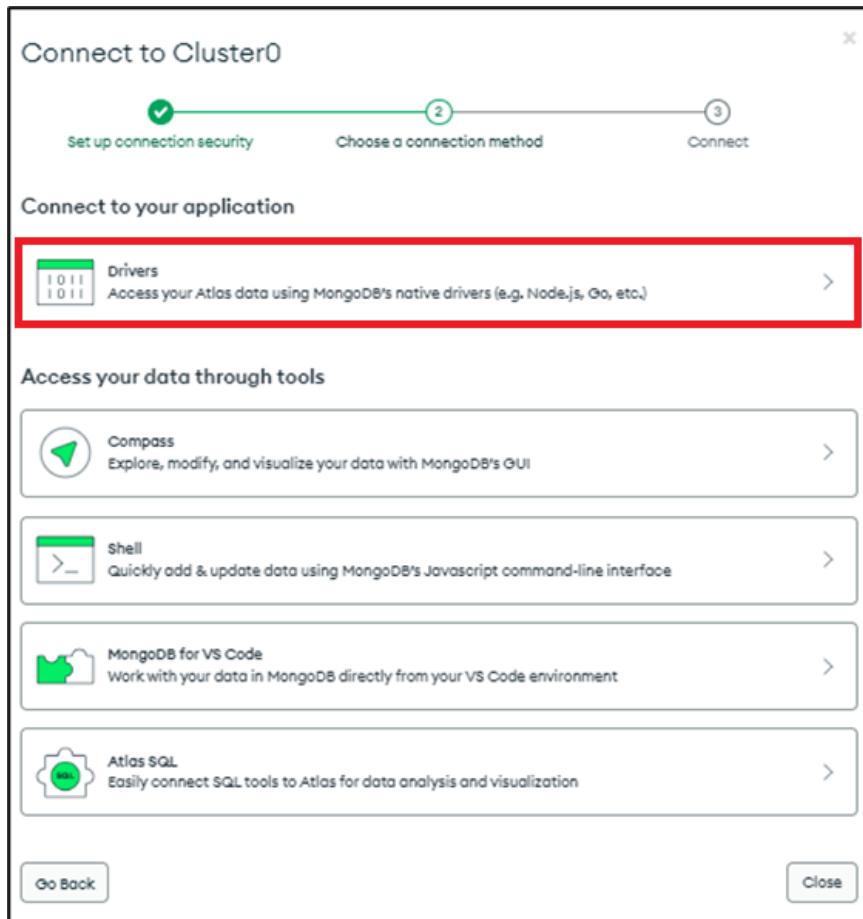


Figure 12: Choose a connection method Page

3. The **Connect** page opens as shown in Figure 13.

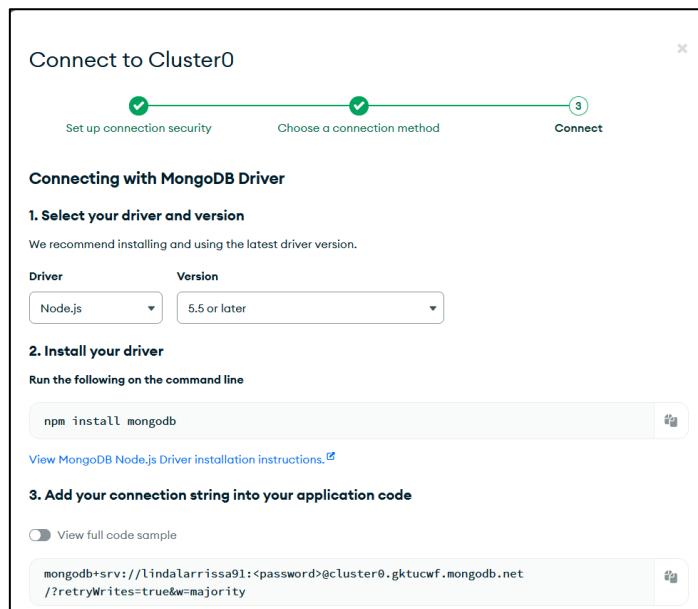


Figure 13: Connect Page

4. To select the driver and version, under **Select your driver and version**, click the **Driver** drop-down list, and select **Node.js** as shown in Figure 14.

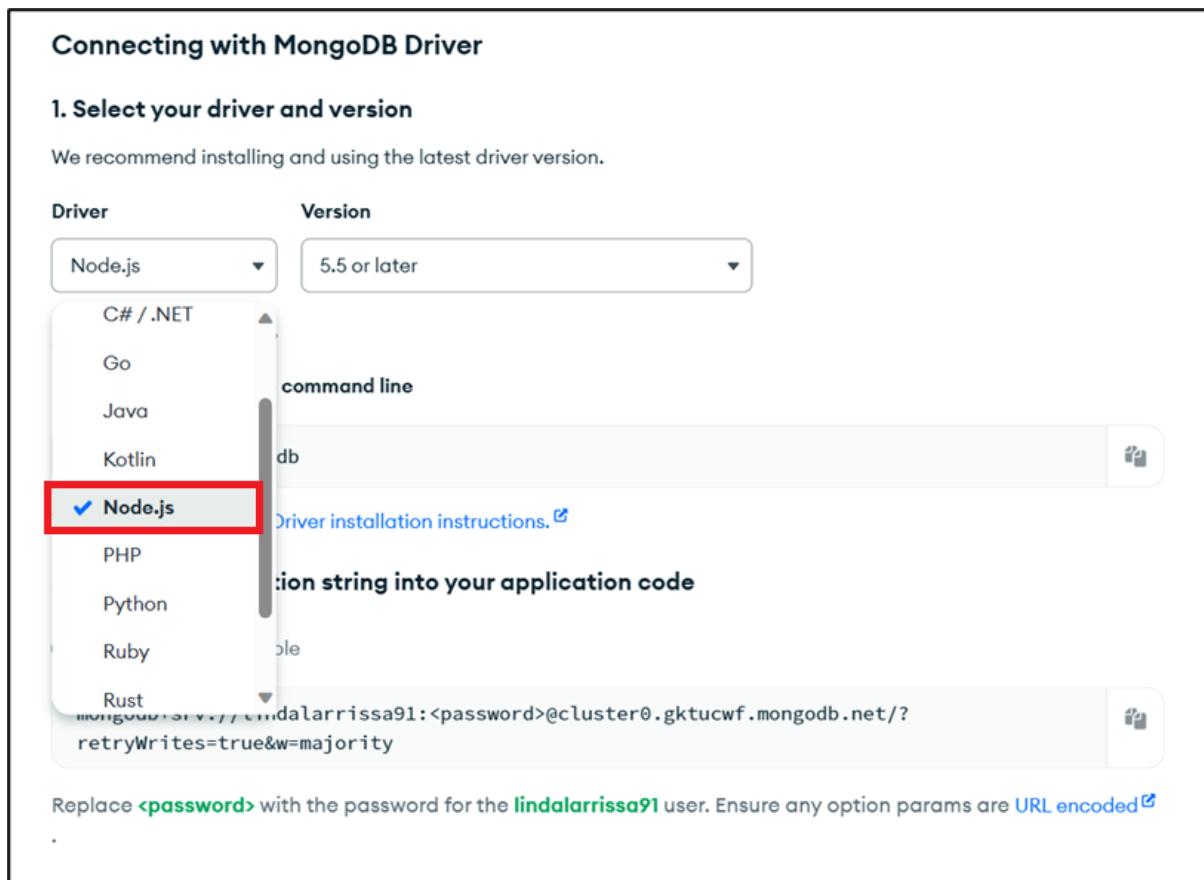


Figure 14: Select your driver and version Page

5. The connection string is displayed. It should be copied, edited, and saved to be used in the application.

To copy the connection string:

- Under **Add your connection string into your application code**, select the connection string as shown in Figure 15.
- Press **Ctrl + c**. Paste the connection string in a Notepad file and save it for further use.
- To close the **Connect to Cluster0** dialog box, click **Close**.

2. Install your driver

Run the following on the command line

```
npm install mongodb
```

[View MongoDB Node.js Driver installation instructions.](#)

3. Add your connection string into your application code

View full code sample

```
mongodb+srv://lindalarrissa91:<password>@cluster0.gktucwf.mongodb.net/?retryWrites=true&w=majority
```

Replace <password> with the password for the **lindalarrissa91** user. Ensure any option params are [URL encoded](#).

RESOURCES

[Get started with the Node.js Driver](#) [Node.js Starter Sample App](#)
[Access your Database Users](#) [Troubleshoot Connections](#)

[Go Back](#) Close

Figure 15: Copy the Connection String

Edit the Connection String

The copied connection string cannot be directly used in the application. Edit the connection string to include the password and database name. In the connection string:

- Replace <password> with linda91.
- Type the database name, Library, before the question mark.

The edited connection string is as follows:

mongodb+srv://lindalarrissa91:**linda91**@cluster0.gktucwf.mongodb.net/**Library**?retryWrites=true&w=majority

Use the edited connection string in the library application.

Appendix B

UPLOAD LIBRARY APPLICATION TO GITHUB REPOSITORY

Upload the library application to GitHub repository by performing the given steps:

1. Ensure to install the latest version of GitHub on the local system. To do so, navigate to <https://git-scm.com/downloads>, and complete the installation.

To open GitHub on the browser, navigate to <https://github.com/>.

The **Let's build from here** page appears as shown in Figure 1.

To create a new account, on the upper-right of **Let's build from here** page, click **Sign up**.

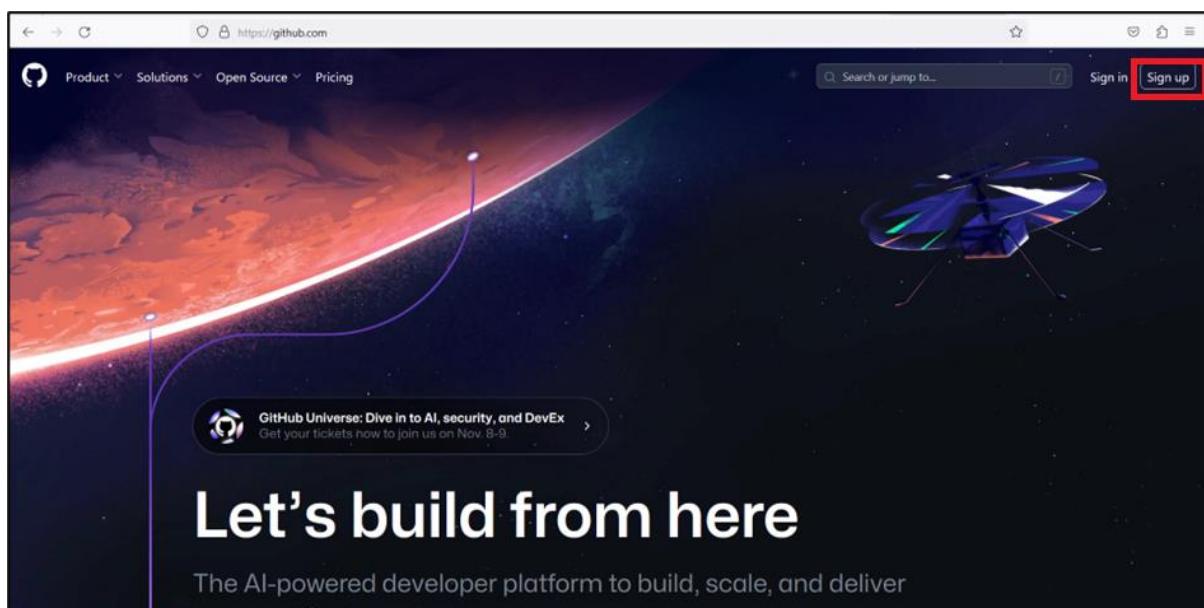


Figure 1: Let's build from here Page

2. Complete the account creation process by filling in the necessary details.

The new GitHub account is created and the dashboard is displayed as shown in Figure 2. To create a new repository, on the left of GitHub dashboard, click **Create repository**.

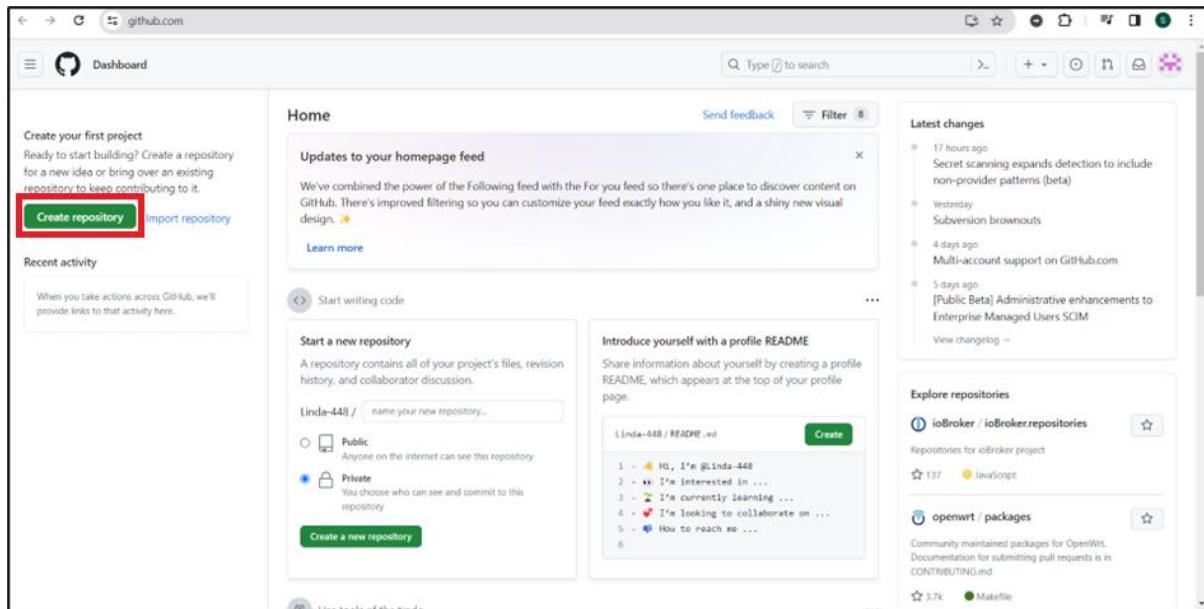


Figure 2: GitHub Dashboard

- To provide the repository name, on the **Create a new repository** page, in the **Repository name** box, type **Library application** as shown in Figure 3.

The screenshot shows the 'Create a new repository' form. The 'Repository name' field is filled with 'Library application', which is highlighted with a red box. Other fields include 'Owner' set to 'Linda-448', 'Description (optional)', 'Visibility' set to 'Public', 'Initialize this repository with' options, 'Add .gitignore' dropdown, and 'Choose a license' dropdown.

Figure 3: Create a new repository Page

- To finish the process of creating a new repository, on the **Create a new repository** page, scroll down, and click **Create repository** as shown in Figure 4.

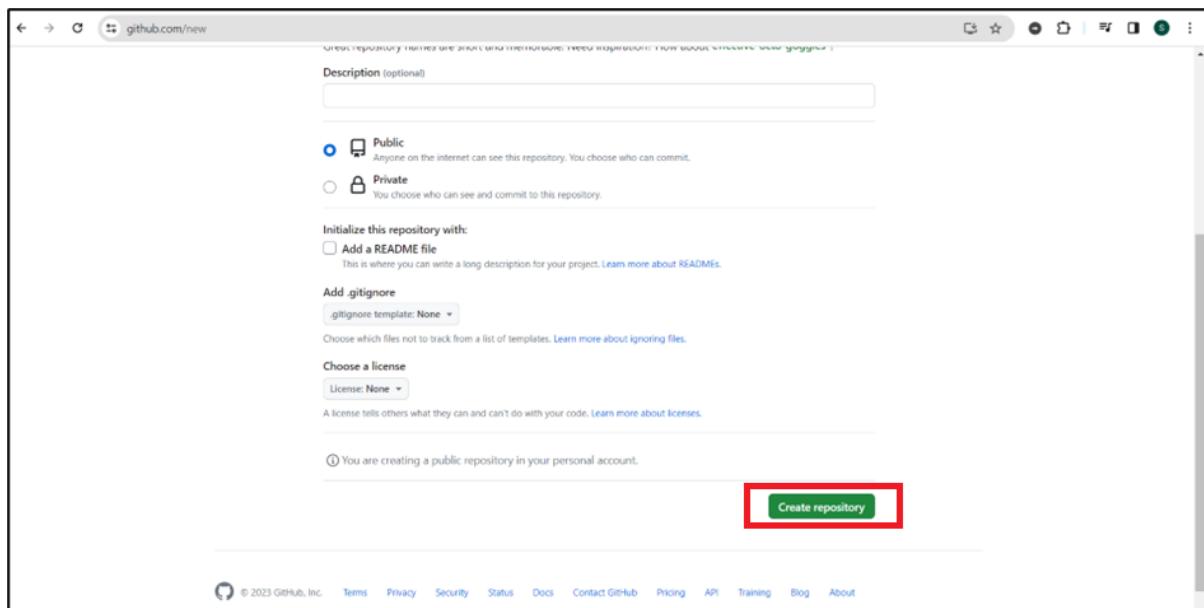


Figure 4: Creating a New Repository

5. The new repository, **Library-application**, is created as shown in Figure 5.

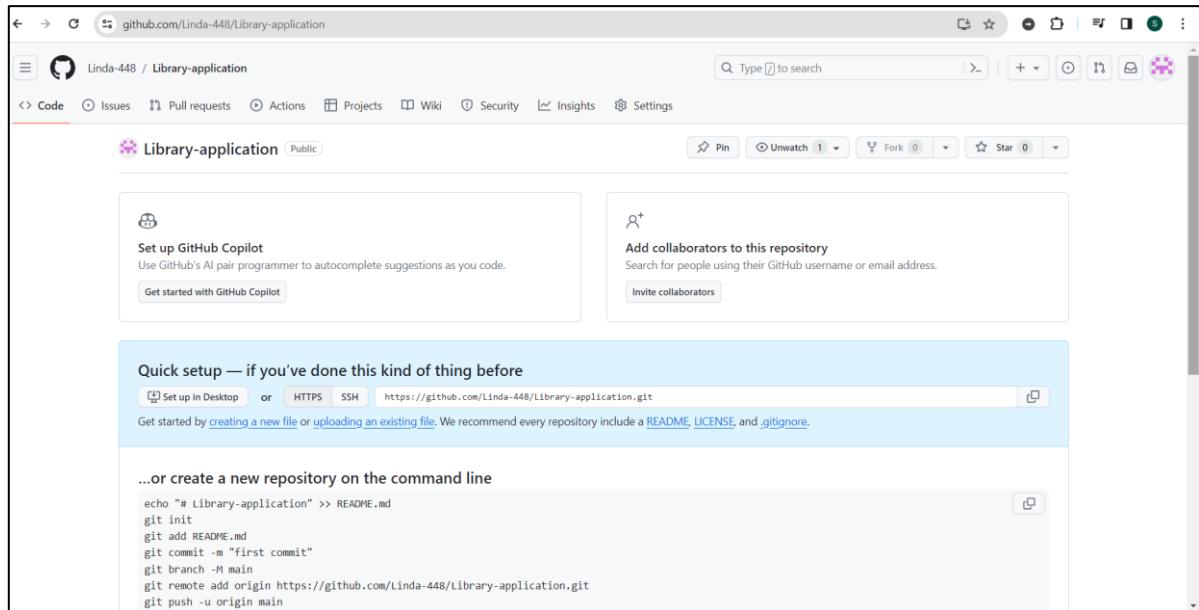


Figure 5: Library-application Repository

6. To create the new repository using the command prompt, open the Node.js prompt, and navigate to the `library_application` folder.

7. To initialize an empty Git repository, at the Command Prompt, type the command, and press **Enter** as shown in Figure 6.

```
git init
```

```
C:\NodeJS\library_application>git init  
Initialized empty Git repository in C:/NodeJS/library_application/.git/
```

Figure 6: Initializing the Application

8. To add all the application resources in the **library_application** folder, type the commands one by one by pressing **Enter** as shown in Figures 7, 8, and 9.

```
git add model  
git add views  
git add public  
git add app.js
```

```
C:\NodeJS\library_application>git add model  
C:\NodeJS\library_application>git add views  
C:\NodeJS\library_application>git add public  
C:\NodeJS\library_application>git add app.js
```

Figure 7: Adding the Folders in the Root Directory

```
git add package.json  
git add package-lock.json
```

```
C:\NodeJS\library_application>git add package.json
warning: LF will be replaced by CRLF in package.json.
The file will have its original line endings in your working directory

C:\NodeJS\library_application>git add package-lock.json
warning: LF will be replaced by CRLF in package-lock.json.
The file will have its original line endings in your working directory
```

Figure 8: Adding the Files in the Root Directory

```
git add node_modules
```

```
C:\NodeJS\library_application>git add node_modules
warning: LF will be replaced by CRLF in node_modules/.bin/ejs.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in node_modules/.bin/ejs.ps1.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in node_modules/.bin/jake.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in node_modules/.bin/jake.ps1.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in node_modules/.bin/mime.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in node_modules/.bin/mime.ps1.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in node_modules/.package-lock.json.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in node_modules/@mongodb-js/saslprep/LICENSE.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in node_modules/@mongodb-js/saslprep/dist/.esm-wrapper.mjs.
The file will have its original line endings in your working directory
```

Figure 9: Adding the Packages

9. To save the updates done to the repository, type the command and press **Enter** as shown in Figure 10.

```
git commit -m "first commit"
```

```
C:\NodeJS\library_application>git commit -m "first commit"
[master (root-commit) 46f60a0] first commit
 1895 files changed, 384987 insertions(+)
 create mode 100644 app.js
 create mode 100644 model/Book.js
 create mode 100644 model/User.js
 create mode 100644 node_modules/.bin/ejs
 create mode 100644 node_modules/.bin/ejs.cmd
 create mode 100644 node_modules/.bin/ejs.ps1
 create mode 100644 node_modules/.bin/jake
 create mode 100644 node_modules/.bin/jake.cmd
 create mode 100644 node_modules/.bin/jake.ps1
 create mode 100644 node_modules/.bin/mime
 create mode 100644 node_modules/.bin/mime.cmd
 create mode 100644 node_modules/.bin/mime.ps1
 create mode 100644 node_modules/.package-lock.json
 create mode 100644 node_modules/@mongodb-js/saslprep/LICENSE
 create mode 100644 node_modules/@mongodb-js/saslprep/dist/.esm-wrapper.mjs
 create mode 100644 node_modules/@mongodb-js/saslprep/dist/code-points-data.d.ts
 create mode 100644 node_modules/@mongodb-js/saslprep/dist/code-points-data.d.ts.map
 create mode 100644 node_modules/@mongodb-js/saslprep/dist/code-points-data.js
 create mode 100644 node_modules/@mongodb-js/saslprep/dist/code-points-data.js.map
 create mode 100644 node_modules/@mongodb-js/saslprep/dist/code-points-src.d.ts
```

Figure 10: Committing the Changes to GitHub Repository

- 10.To rename the old branch name to `main`, type the command, and press **Enter** as shown in Figure 11.

```
git branch -M main
```

```
C:\NodeJS\library_application>git branch -M main
```

Figure 11: Renaming the Branch

- 11.To add a remote repository for the existing repository at the given URL, type the command and press **Enter** as shown in Figure 12.

```
git remote add origin https://github.com/Linda-448/Library-application.git
```

```
C:\NodeJS\library_application>git remote add origin https://github.com/Linda-448/Library-application.git
```

Figure 12: Adding a Remote Repository

- 12.To add the upstream reference for the branch that is successfully pushed, type the command and press **Enter** as shown in Figure 13.

```
git push -u origin main
```

```
C:\NodeJS\library_application>git push -u origin main
```

Figure 13: Adding Upstream Reference

13. The **Connect to GitHub** page appears. To connect to GitHub, on the **Connect to GitHub** page, click **Sign in with your browser** as shown in Figure 14.

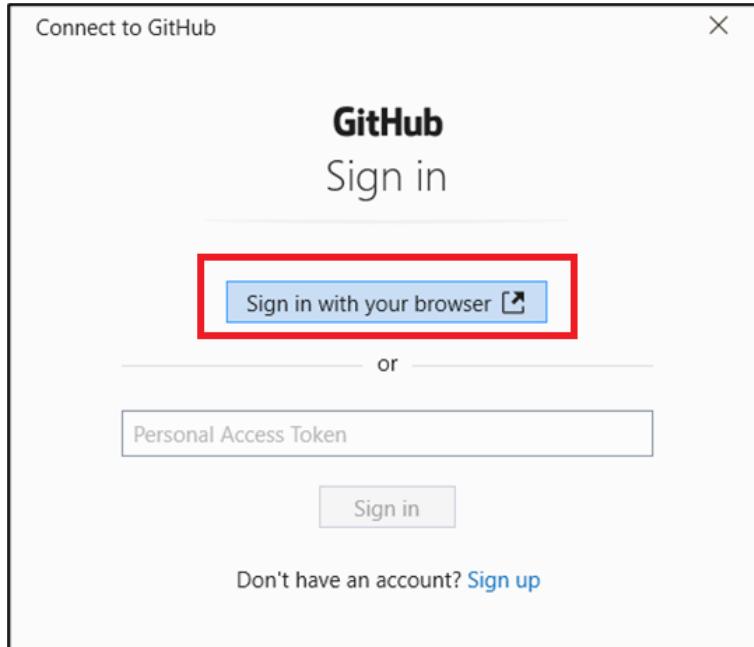


Figure 14: Connect to GitHub Page

14. To authorize the Git Ecosystem, on the **Authorize Git Credential Manager** page, click **Authorize git-ecosystem** as shown in Figure 15.

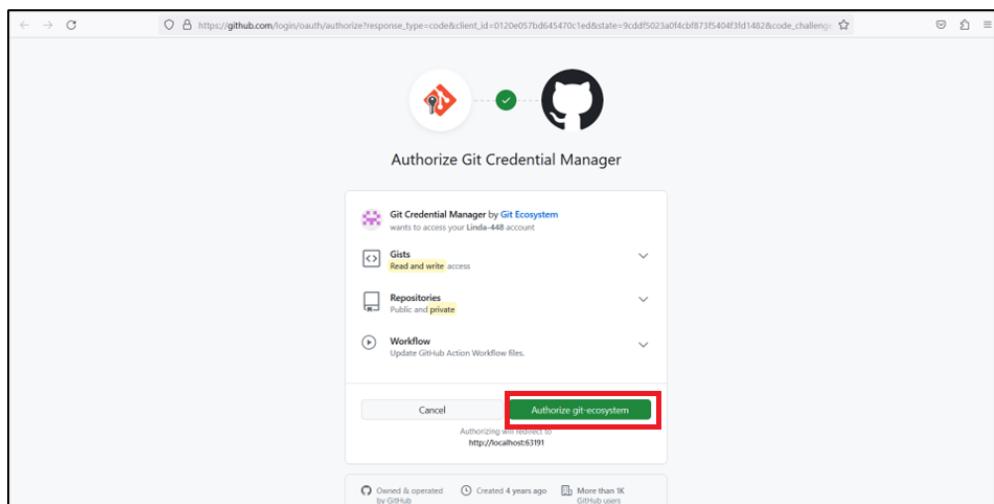


Figure 15: Authorize Git Credential Manager Page

15. On the **Authentication Succeeded** page, the message “**Authentication Succeeded**” is displayed as shown in Figure 16.

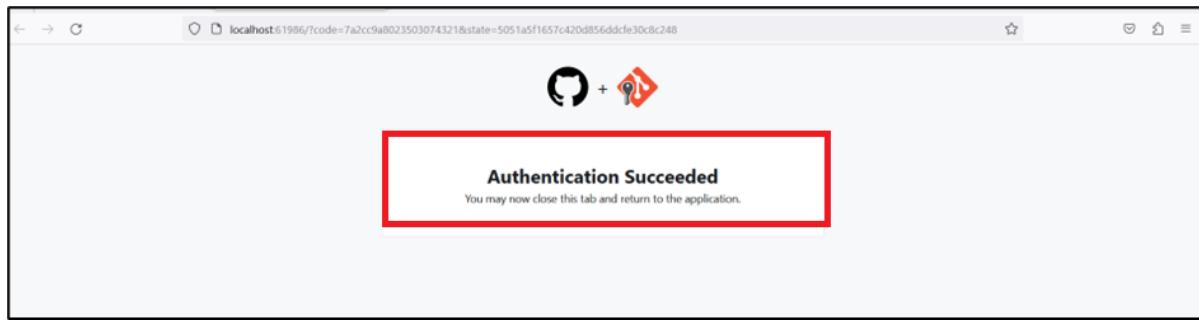


Figure 16: Authentication Succeeded Page

On some computers, the Git Credential Manager may not work properly or may be blocked by a firewall. In such cases, the developer can use following command and substitute the password parameter with the value of a personal access token:

```
git remote set-url origin  
https://username:password@github.com/organization/repo.git
```

Refer to the following link to know how to create a Personal Access Token:

<https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/managing-your-personal-access-tokens>

16. To access the newly created repository, navigate to the GitHub dashboard, and click **Library-application** as shown in Figure 17.

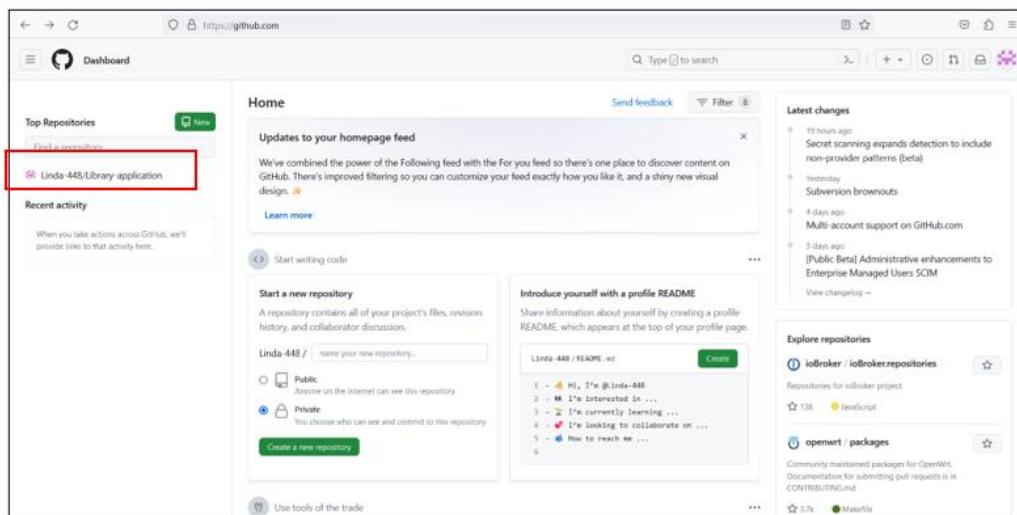


Figure 17: GitHub Dashboard Page

17. The **Library-application** repository is created and all the resources are

uploaded as shown in Figure 18.

The library application is successfully uploaded to the GitHub repository.

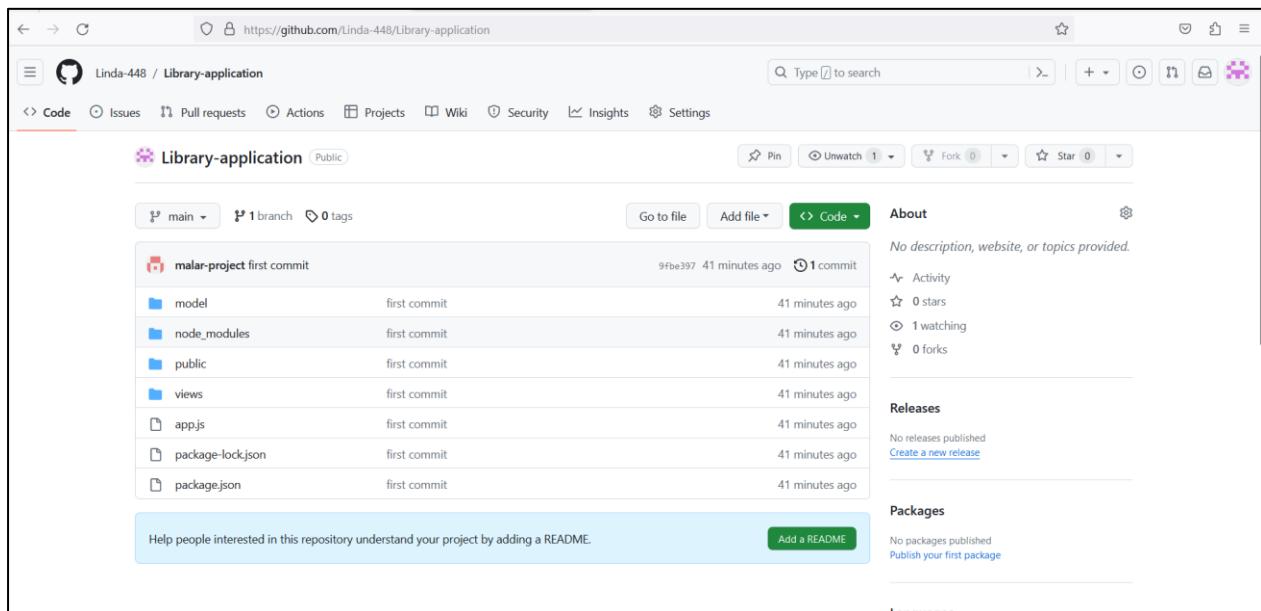


Figure 18: Library Application Resources on the GitHub

Appendix C

Sr. No.	Case Studies																												
1.	<p>Create a simple hospital management application using Node.js. Table 1 lists some sample patient data.</p> <table border="1"> <thead> <tr> <th>Patient_Name</th><th>Age</th><th>Diagnosis</th><th>Treatment_Cost (\$)</th></tr> </thead> <tbody> <tr> <td>John</td><td>34</td><td>Malaria</td><td>1177</td></tr> <tr> <td>Emily</td><td>29</td><td>Influenza</td><td>2167</td></tr> <tr> <td>Lara</td><td>23</td><td>Mpox</td><td>3423</td></tr> <tr> <td>Emma</td><td>50</td><td>Typhoid Fever</td><td>1233</td></tr> <tr> <td>Jack</td><td>45</td><td>Malaria</td><td>1177</td></tr> <tr> <td>Mike</td><td>40</td><td>Zika Virus</td><td>4562</td></tr> </tbody> </table> <p>Table 1: Patient Details</p> <ul style="list-style-type: none"> a. Create a function for managing patients using Patient_Name, Age, Diagnosis, and Treatment_Cost parameters. b. Save the treatment costs of five patients from Table 1 in an array. Then, use a for loop to calculate the total treatment cost and average treatment cost. c. Create an empty patientRecordFile.txt file. Write the first three patient details from Table 1 to this file. d. Read and display contents of patientRecordFile.txt. e. Append details of Mike to patientRecordFile.txt. f. Delete patientRecordFile.txt file. 	Patient_Name	Age	Diagnosis	Treatment_Cost (\$)	John	34	Malaria	1177	Emily	29	Influenza	2167	Lara	23	Mpox	3423	Emma	50	Typhoid Fever	1233	Jack	45	Malaria	1177	Mike	40	Zika Virus	4562
Patient_Name	Age	Diagnosis	Treatment_Cost (\$)																										
John	34	Malaria	1177																										
Emily	29	Influenza	2167																										
Lara	23	Mpox	3423																										
Emma	50	Typhoid Fever	1233																										
Jack	45	Malaria	1177																										
Mike	40	Zika Virus	4562																										
2.	<ul style="list-style-type: none"> a. Create a simple HTTP server that responds with a plain text message, Welcome to the Evergreen Health Care Hospital. b. Create an event-driven login notification system by registering an event listener for the events listed in Table 2. <table border="1"> <thead> <tr> <th>Events</th><th>Time (in secs)</th></tr> </thead> <tbody> <tr> <td>Patient Admitted</td><td>100</td></tr> <tr> <td>Preliminary Checkup Completed</td><td>150</td></tr> <tr> <td>Doctor Visit</td><td>800</td></tr> </tbody> </table> <p>Table 2: Event Details</p> <ul style="list-style-type: none"> c. Parse the query string using this data: Patient_Name: Jack Age: 23 Diagnosis: Food-borne and water-borne 	Events	Time (in secs)	Patient Admitted	100	Preliminary Checkup Completed	150	Doctor Visit	800																				
Events	Time (in secs)																												
Patient Admitted	100																												
Preliminary Checkup Completed	150																												
Doctor Visit	800																												

	<p>Extract data from the URL query string and display parsed data in the console.</p> <p>d. Convert JavaScript objects into URL query strings.</p> <ul style="list-style-type: none"> Define two patient objects, <code>patient1</code> and <code>patient2</code>, each with distinct properties such as <code>Patient_Name</code>, <code>Activity_Status</code>, and <code>Patient_Condition</code> using the given data: <pre> patient1 Patient_Name = Jack Activity_Status = false Patient_Condition = stable </pre> <pre> patient2 Patient_Name = Daniel Activity_Status = false Patient_Condition = Critical </pre> <ul style="list-style-type: none"> To serialize the object, use <code>patient2</code>. Use the <code>stringify</code> method on the object. Display the result on the console. <p>e. Encode the given patient data:</p> <pre> Patient_Name = Jack Age = 40 Patient_Condition = stable </pre> <p>f. Decode the URL</p> <pre>patientName=Lara&patientGender=Female&AverageGlucoseLevel=229&available=true</pre> <p>g. Send an email to the Medical Insurance company with the subject <code>Medical Claim</code> and add body text as <code>Hi I am Jack and I want to submit a medical claim.</code></p> <p>h. Create a simple Web application using Express.js as the backend to display patient information. Refer to Table 1 for patient details.</p>
3.	<p>a. Create a text file using Notepad with the given statements:</p> <p>Statement 1: Jack's condition is not stable yet. Statement 2: Now, Jack is feeling a little better.</p> <ul style="list-style-type: none"> Perform read operation on the text file using the <code>async</code> and <code>await</code> functions.

- Display Statement 1 after three seconds and statement 2 after four seconds.
- b. Create a simple RESTful API for patient details. This RESTful API manages patient details using Node.js and Express using Table 1.
- Create a new patient.
 - Update the patient's name John to Kevin.
 - Delete a patient whose age is 50.
 - Display the results on the console.
 - Filter the patient data based on a specified diagnosis such as Malaria.
 - Add a new patient using a custom method and non-standard URL.
 - Update the patient's name Mike to Mark using a custom method and non-standard URL.
 - Delete a patient whose age is greater than 44 using a custom method and non-standard URL.
 - Display the results on the console.
- c. Perform the CRUD operations with the data given in Table 1.
- Create a MongoDB database Patient and collection Patient_Collection and insert the first three documents given in Table 1 into the Patient_Collection.
 - View the inserted documents.
 - Update Name: Lara as Anna and age of Emily as 56.
 - Delete a document where Name is equal to Emily.
 - Display the results.