SGOU
Higher Education for All

Web Programming

Course Code: M26CA01SE
Master of Computer Application
Skill Enhancement Course
Self Learning Material



SREENARAYANAGURU OPEN UNIVERSITY

SREENARAYANAGURU OPEN UNIVERSITY

The State University for Education, Training and Research in Blended Format, Kerala

SREENARAYANAGURU OPEN UNIVERSITY

Vision

To increase access of potential learners of all categories to higher education, research and training, and ensure equity through delivery of high quality processes and outcomes fostering inclusive educational empowerment for social advancement.

Mission

To be benchmarked as a model for conservation and dissemination of knowledge and skill on blended and virtual mode in education, training and research for normal, continuing, and adult learners.

Pathway

Access and Quality define Equity.

Web Programming

Course Code: M26CA01SE

Semester- I

Skill Enhancement Course
Master of Computer Application
2026-27 Admission onwards
Self Learning Material

(With Model Question Paper Sets)

SREENARAYANAGURU
OPEN UNIVERSITY

SREENARAYANAGURU OPEN UNIVERSITY

The State University for Education, Training and Research in Blended Format, Kerala

SREENARAYANAGURU
OPEN UNIVERSITY

Web Programming

Course Code: M26CA01SE
Semester- I
Skill Enhancement Course
Master of Computer Application

Academic Committee

Prof. (Dr.) Sabu M.K.
Dr. G. Santhoshkumar T
Prof. (Dr.) Vinod Chandra S.S.
Dr. Vinod P.
Dr.Lajish V.L.
Sreekanth M.S.
Dr. Vivek P.
Dr. Arun K.S.
Dr. Abdul Jebbar P.

Development of the Content

Pankaj Kumar G.
A. A. Aswathy

Review and Edit

Sreeraj R.

Scrutiny

Shamin S.
Greeshma P. P.
Sreerekha V. K.
Dr. Kanitha D.K.
Subi Priya Laxmi S. B. N.
Aswathy V. S.

Cover Design

Jineshkumar S

Co-ordination

Head,
School of Communication &
Information Studies

Scan this QR Code for reading the SLM on a digital device.

First Edition
October 2026

Copyright
© Sreenarayanaguru Open University

ISBN 978-81-181574-6-5

All rights reserved. No part of this work may be reproduced in any form, by mimeograph or any other means, without permission in writing from Sreenarayanaguru Open University. Printed and published on behalf of Sreenarayanaguru Open University by Registrar, SGOU, Kollam.

www.sgou.ac.in

Visit and Subscribe our Social Media Platforms

MESSAGE FROM VICE CHANCELLOR

Dear learner,

It gives me immense pleasure and a deep sense of pride to warmly welcome you to Sreenarayanaguru Open University—a vibrant and progressive institution committed to transforming lives through inclusive, flexible, and high-quality education.

Established in September 2020 as a forward-looking initiative of the Government of Kerala, the University stands as a beacon of opportunity for learners seeking to advance their academic and professional aspirations through the open and distance learning mode. Guided by our foundational principle that "access and quality define equity," we are steadfast in our mission to democ-ratize education while upholding uncompromising academic standards.

Our university is inspired by the timeless vision and philosophy of Sree Narayana Guru, whose ideals of knowledge, equality, and social transformation continue to guide our academic journey. His enduring legacy instils in us the responsibility to create an educational environment that empowers individuals, nurtures critical thinking, and contributes meaningfully to society.

Understanding the dynamic needs of contemporary learners, we have adopted a robust and learner-centric blended learning model, seamlessly integrating Self-Learning Materials, Academic Counselling, and Advanced Digital Learning Platforms. This holistic approach ensures flexibility without sacrificing academic depth, enabling you to learn at your own pace while remaining meaningfully connected to a vibrant academic ecosystem.

The Master of Computer Applications (MCA) programme you are embarking upon is carefully designed to position you at the forefront of the digital revolution. It uniquely blends strong theoretical foundations with practical, industry-oriented competencies. The curriculum emphasizes algorithmic thinking, system design, programming, database management, networking, and emerging technologies such as artificial intelligence, data science, and cloud computing.

What sets this programme apart is its forward-thinking design, which offers:

- Opportunities for skill enhancement aligned with industry needs
- Multidisciplinary learning pathways for broader intellectual development
- Exposure to innovative and emerging technology domains
- Multiple specialization options tailored to evolving career landscapes
- A strong focus on employability, entrepreneurship, and global career readiness

Our Self-Learning Materials are meticulously developed by experts, enriched with contemporary case studies, real-world applications, and practical insights to ensure clarity, engagement, and relevance. We are committed to equipping you not just with knowledge, but with the confidence and competence to excel in a competitive global environment.

At Sreenarayanaguru Open University, you are never alone in your learning journey. Our dedicated learner support system is designed to provide continuous academic guidance, timely assistance, and effective grievance redressal. We encourage you to actively engage with us, share your concerns, and make the most of the resources and support available to you.

As you begin this important phase of your academic journey, I urge you to embrace learning with curiosity, discipline, and determination. The world of technology is ever-evolving, and your willingness to adapt, innovate, and grow will define your success.

Remember, this is not just a programme—it is a pathway to transforming your future. I wish you a fulfilling learning experience and a successful, inspiring career ahead.

Regards,



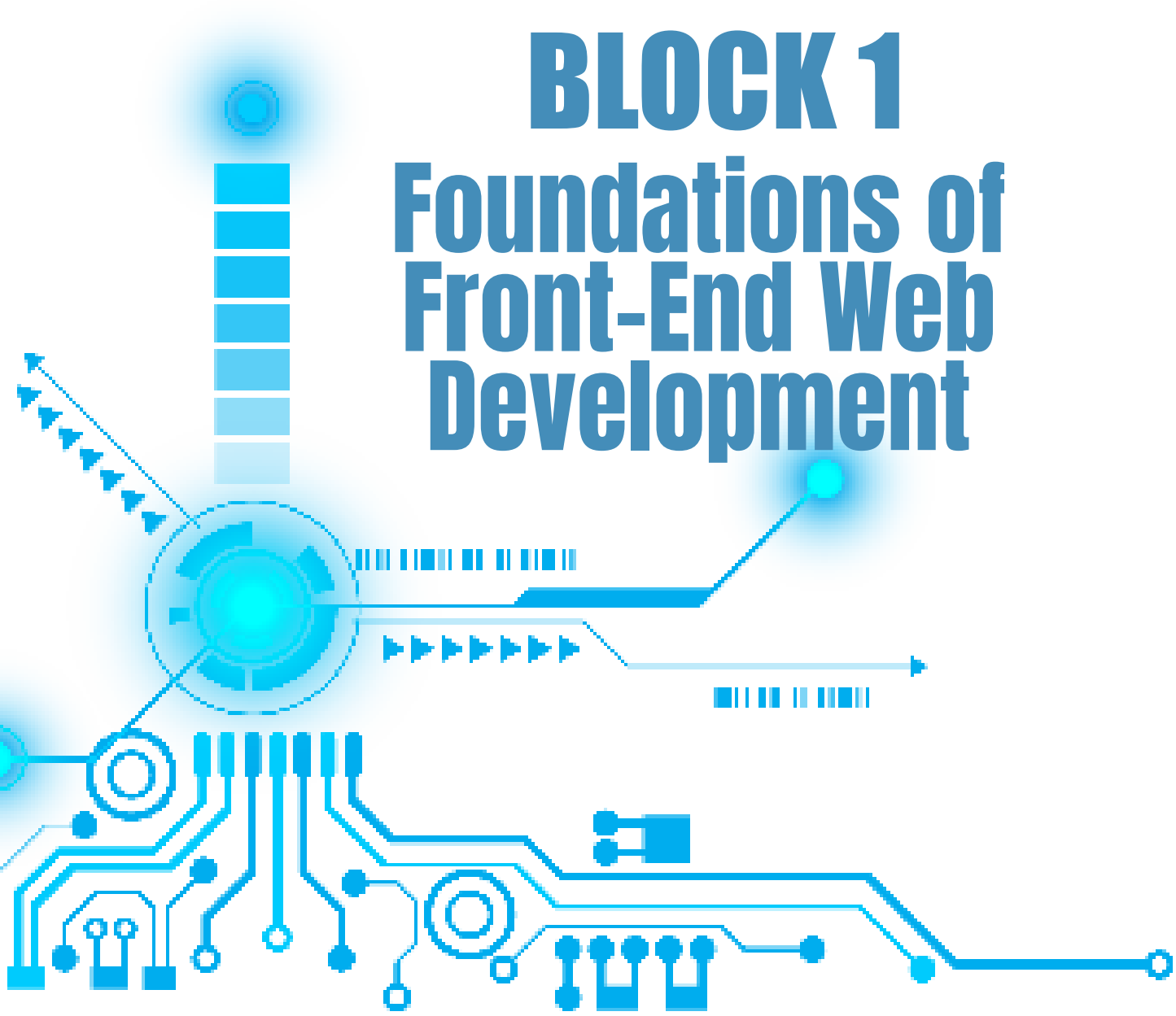
Dr. Jagathy Raj V. P.

01-10-2026

Contents

BLOCK	**1**	**Foundations of Front-End Web Development**	**1**
UNIT	1	Fundamentals of HTML and Web Page	2
UNIT	2	CSS Fundamentals and Responsive Web Design: Page Development	22
UNIT	3	Javascript Essentials and Dom Manipulation Development	49
UNIT	4	Event Handling, Form Validation, and Asynchronous JavaScript	82
BLOCK	**2**	**Full-Stack Web Application Development**	**114**
UNIT	1	Node.JS Fundamentals and Restful Web Services	115
UNIT	2	Database Integration and REST API Development	151
UNIT	3	React Fundamentals and Component-Based Development	185
UNIT	4	React Routing, Backend Integration, and Full-Stack Application Development	218
Model Question Paper Sets			**253**

BLOCK 1
Foundations of Front-End Web Development



1 UNIT Fundamentals of HTML and Web Page

Learning Outcomes

After completion of this unit, the learner will be able to:

- explain the interaction between web browsers and web servers in delivering web content
- describe the structure and components of an HTML document
- apply semantic HTML elements to organize and structure web content
- construct webpages using headings, paragraphs, lists, images, tables, and hyperlinks
- select appropriate HTML5 input types for collecting different types of information from users
- differentiate between the GET and POST methods used for submitting form data

Background

Kerala is known for its diverse tourist destinations, ranging from the hill stations of Munnar and Wayanad to the backwaters of Alappuzha, the wildlife attractions of Thekkady, and the coastal destinations of Kovalam. A tourist planning a trip may want to know about the location, attractions, activities, accommodation and travel options associated with each destination.

Traditionally, such information could be obtained from printed brochures or travel offices. Today, much of this information is accessed through websites and mobile applications.

Imagine a tourist visiting a website called Kerala Explorer. The home page displays a photograph of Munnar, a list of popular destinations, navigation links, a table of travel information and a form through which visitors can submit an enquiry.

How does the browser know which text represents the heading, which information represents a list, which content is an image and which element is a form?

How does the information reach the tourist's browser in the first place?

These questions lead us to the foundations of web development.

A web application involves communication between a client, usually a web browser, and a server. HTML provides the basic structure through which information can be organized and presented as a webpage.

In this unit, the Kerala Explorer tourism portal will be used as a running illustration. The concepts introduced here will later provide the foundation for adding styling, interactivity, server-side processing, databases and modern frontend technologies.

Keywords

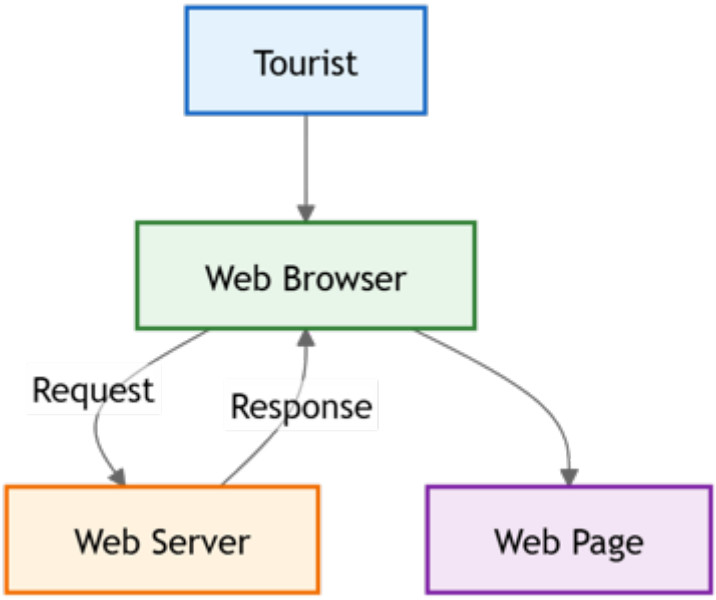
Browser–Server Communication, HTML Document Structure, Semantic HTML, HTML Content Elements, Hyperlinks and Navigation, HTML Forms

Discussion

1.1.1 How Browsers and Servers Communicate

Imagine that a tourist wants to explore Munnar through Kerala Explorer. The tourist opens a web browser and enters the address of the tourism website.

A webpage is typically retrieved from a web server when requested by a client. The web browser initiates the interaction by sending an HTTP request for a particular resource. The server processes the request and returns an appropriate response, which the browser interprets and renders for the user. In the Kerala Explorer application, selecting the Munnar destination causes the browser to request the corresponding resource from the server. The returned HTML document provides the structural information required for the browser to render the destination page. The browser needs to communicate with the system hosting the website and request the required resources.



In this interaction, the web browser acts as the client and the web server acts as the system responsible for processing the request and returning the appropriate response. The browser receives the response and displays the webpage.

Web Browser

A web browser is a software application used to access and display web resources.

Web Server

A web server is a software system, often running on a server computer, that receives HTTP requests from clients and returns appropriate web resources or responses.

Kerala Explorer Illustration

Suppose Kerala Explorer provides separate webpages for:

- Munnar
- Wayanad
- Alappuzha
- Thekkady
- Kovalam

When the tourist selects Munnar, the browser requests the corresponding resource. The server processes the request and sends an appropriate response. The browser then presents the Munnar webpage.

Kerala Explorer – Illustration

A tourist selects Munnar from the Kerala Explorer website. The browser sends a request for the Munnar page. The web server processes the request and returns the required resources. The browser interprets the response and displays the Munnar webpage.

Student Engagement

Develop two or three objective questions from this section and write their answers.

1.1.2 Anatomy of an HTML Document

Suppose Kerala Explorer wants to create a webpage describing Munnar. The page needs a title, destination name and description.

How can these components be organized so that the browser can interpret their roles?

HTML provides a structured way of representing this information.

Consider the following document:

```
<!DOCTYPE html>
<html>
<head>
   <title>Explore Munnar</title>
</head>
<body>
   <h1>Munnar</h1>
   <p>
      Explore the scenic hill destination of Munnar.
   </p>
</body>
</html>
```

The document contains several important components.

1. DOCTYPE Declaration

<!DOCTYPE html>

The <!DOCTYPE html> declaration tells the browser that the document should be interpreted using the HTML standard and enables standards mode.

2. HTML Element

```
<html>
  ...
</html>
```

The <html> element represents the root element of the HTML document.

3. Head Element

```
<head>
  <title>Explore Munnar</title>
</head>
```

The <head> section contains information associated with the document.

4. Title Element

```
<title>Explore Munnar</title>
```

The <title> element specifies the title of the document, which is typically displayed in the browser tab and may also be used by bookmarks and other browser features.

5. Body Element

```
<body>
  <h1>Munnar</h1>
  <p>
    Explore the scenic hill destination of Munnar.
  </p>
</body>
```

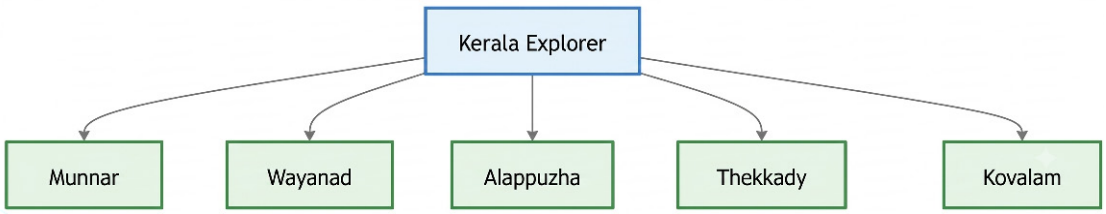
The <body> element contains the main content of the webpage.

Definition

An HTML document is a structured document containing HTML elements that describe the organization and content of a webpage.

Kerala Explorer Illustration

The same basic document structure can be used for different destination pages:



The destination-specific content changes, while the basic HTML structure remains similar.

Think About It

If every destination page contains a destination name, description and image, how can HTML communicate the different roles of these pieces of information?

1.1.3 Semantic HTML Elements

Consider the home page of Kerala Explorer. It may contain the website name, navigation menu, featured destinations, destination information and footer details.

These parts have different purposes. HTML provides semantic elements to communicate the meaning or role of different sections.

Consider the following example:

```
<header>
   <h1>Kerala Explorer</h1>
</header>
<main>
   <section>
      <h2>Popular Destinations</h2>
   </section>
   <article>
      <h2>Discover Munnar</h2>
      <p>Explore the scenic beauty of Munnar.</p>
   </article>
</main>
<footer>
   <p>Kerala Explorer</p>
</footer>
```

1. Header

The <header> element represents introductory or header content.

2. Main

The <main> element represents the principal content of the webpage.

3. Section

The <section> element represents a thematic section of content.

4. Article

The <article> element represents an independent piece of content.

5. Footer

The <footer> element represents footer information.

Definition

Semantic HTML refers to the use of HTML elements that communicate the meaning or purpose of the content they contain.

Kerala Explorer Illustration

A destination page can be organized as:

```
<header>
   Kerala Explorer
</header>
<main>
   <section>
      Destination Information
   </section>
   <article>
      Travel Experience
   </article>
</main>
<footer>
   Contact Information
</footer>
```

The semantic structure makes the purpose of different parts of the webpage clearer.

Value Addition

A webpage is not consumed only by users visually inspecting the screen. Its structure may also be interpreted by search engines and assistive technologies. Meaningful HTML structure therefore contributes to the broader quality and accessibility of webpage content.

1.1.4 HTML Content Elements

A tourism webpage needs to present information in several forms.

For example, a Munnar webpage may contain:

Munnar

Munnar is a popular hill destination.

Popular Experiences

- Explore tea plantations
- Visit scenic viewpoints
- Enjoy nature walks

HTML provides different elements for representing these types of information.

1.1.4.1 Headings

Headings organize content into a hierarchy.

```
<h1>Kerala Explorer</h1>

<h2>Popular Destinations</h2>

<h3>Munnar</h3>
```

HTML provides heading levels from <h1> to <h6>.

For example:

```
<h1>Kerala Tourist Destinations</h1>

<h2>Munnar</h2>

<h3>Popular Attractions</h3>
```

The hierarchy helps organize information within the webpage.

1.1.4.2 Paragraphs

The <p> element represents a paragraph.

```
<p>
   Munnar is a scenic hill destination known
   for its landscapes and tea plantations.
</p>
```

A destination webpage can contain multiple paragraphs describing attractions, activities and travel information.

1.1.4.3 Lists

Suppose Kerala Explorer wants to display popular activities in Munnar.

An unordered list can be created using <ul>:

```
<ul>
   <li>Explore tea plantations</li>
   <li>Visit scenic viewpoints</li>
   <li>Enjoy nature walks</li>
</ul>
```

An ordered list can be created using <ol>:

```
<ol>
   <li>Select a destination</li>
   <li>Choose travel dates</li>
   <li>Submit an enquiry</li>
</ol>
```

An unordered list is appropriate when the order of items is not significant. An ordered list is appropriate when the items represent a sequence. The list can be created using the following HTML elements:

- <p> represents paragraphs.
- creates an unordered list.
- creates an ordered list.
- represents an individual list item.

1.1.5 Images

Tourism websites depend heavily on visual information. A tourist exploring Munnar may want to see photographs of its landscape before deciding to visit.

The <img> element is used to embed an image.

```
<img src="munnar.jpg"
    alt="Scenic landscape of Munnar">
```

The src attribute specifies the image source.

The alt attribute provides alternative text describing the image.

For example:

```
<img src="alappuzha.jpg"
   alt="Backwaters of Alappuzha">
```

The alternative text can communicate the purpose or description of the image when the image itself cannot be accessed.

> **Think About It**
>
> Suppose the image of Munnar cannot be displayed. What information should the webpage provide so that the visitor can still understand what the image represents?

1.1.6 Hyperlinks and Navigation

A tourism website normally contains several related webpages. A tourist may move from the home page to Munnar, Wayanad or Alappuzha.

HTML provides hyperlinks through the <a> element.

```
<a href="munnar.html">
   Explore Munnar
</a>
```

The href attribute specifies the destination of the hyperlink.

1.1.6.1 Internal Linking

An internal link connects pages or resources within the same website.

```
<a href="wayanad.html">
   Explore Wayanad
```

```
</a>
```

If wayanad.html belongs to the Kerala Explorer website, the link is an internal link.

1.1.6.2 External Linking

An external link points to a resource outside the current website.

```
<a href="https://example.com">
    Tourism Information
</a>
```

example.com is a special domain name provided for examples and demonstrations. It is not meant to represent a real website or a particular company.

Navigation

A tourism website can provide a navigation menu.

```
<nav>
    <a href="index.html">Home</a>
    <a href="munnar.html">Munnar</a>
    <a href="wayanad.html">Wayanad</a>
    <a href="alappuzha.html">Alappuzha</a>
    <a href="thekkady.html">Thekkady</a>
</nav>
```

Navigation provides a convenient way for users to move between different webpages.

1.1.7 Tables

Tourists may need structured information before planning a journey.

Kerala Explorer can present destination information using a table.

```
<table>
    <tr>
        <th>Destination</th>
        <th>Type</th>
        <th>Major Attraction</th>
    </tr>
```

```
    <tr>
      <td>Munnar</td>
      <td>Hill destination</td>
      <td>Tea plantations</td>
    </tr>
    <tr>
      <td>Alappuzha</td>
      <td>Backwater destination</td>
      <td>Backwaters</td>
    </tr>
    <tr>
      <td>Thekkady</td>
      <td>Wildlife destination</td>
      <td>Wildlife experiences</td>
    </tr>
</table>
```

The major table elements are:

- <table> – defines a table;
- <tr> – defines a table row;
- <th> – defines a heading cell; and
- <td> – defines a data cell.

1.1.8 HTML Forms

Imagine that a tourist has explored Kerala Explorer and wants more information about a trip to Munnar.

The website may ask for:

- name;
- email address;
- preferred destination;
- travel date; and
- interests.

HTML forms provide a mechanism for collecting such information.

A simple form can be written as:

```
<form>
   <label for="name">Name:</label>
   <input id=”name” type="text" >
   <label for="email">Email:</label>
   <input id=”email” type="email">
   <button type="submit">
      Submit Enquiry
   </button>
</form>
```

1.1.8.1 Text Input

A text input can be used to collect a visitor's name.

```
<input type="text"
       name="visitorName">
```

1.1.8.2 Password Input

A password input is used when password information needs to be entered.

```
<input type="password"
       name="password">
```

1.1.8.3 Radio Buttons

Suppose a tourist has to select one preferred destination.

```
<input type="radio"
       name="destination"
       value="munnar">
```

Munnar

```
<input type="radio"
       name="destination"
       value="wayanad">
```

Wayanad

Radio buttons are suitable when the user is expected to select one option from a group.

1.1.8.4 Checkboxes

Suppose a tourist wants to select multiple interests.

```
<input type="checkbox"
    name="interest"
    value="nature">
```

Nature

```
<input type="checkbox"
    name="interest"
    value="wildlife">
```

Wildlife

```
<input type="checkbox"
    name="interest"
    value="culture">
```

Culture

Checkboxes allow multiple selections.

1.1.8.5 Select

A drop-down list can be created using <select>.

```
<select name="destination">
  <option>Munnar</option>
  <option>Wayanad</option>
  <option>Alappuzha</option>
  <option>Thekkady</option>
</select>
```

1.1.8.6 Textarea

A tourist may want to provide a detailed enquiry.

```
<textarea name="message"></textarea>
```

The <textarea> element is suitable for longer textual input.

1.1.9 Form Submission: GET and POST

Consider a tourist searching for destinations on Kerala Explorer.

The tourist enters Munnar into a search field and selects the Search button. The information needs to be submitted for processing.

Two commonly used form submission methods are GET and POST.

GET Method

A form can use GET as follows:

```
<form method="get">
   <input type="text"
          name="destination">
   <button type="submit">
      Search
   </button>
</form>
```

A tourism search form is an example of an interaction where GET can be useful because the request represents retrieving information.

POST Method

A form can use POST as follows:

```
<form method="post">
   <input type="text"
          name="name">
   <input type="email"
          name="email">
   <button type="submit">
      Submit Enquiry
   </button>
</form>
```

A tourism enquiry form may use POST when submitting information to be processed by the server.

Comparison

Table 1.1.1 Comparison of GET and POST Methods

GET	POST
Commonly associated with retrieving information	Commonly associated with submitting information
Data is associated with the request URL	Data is sent as part of the request
Useful for many search-type interactions	Useful for many submission-type interactions

The appropriate method depends on the purpose of the interaction and how the server is expected to process the request. The use of POST does not by itself provide security or encryption. Security of form data depends on mechanisms such as HTTPS and appropriate server-side security practices.

1.1.10 HTML5 Input Types

A tourism enquiry form may need different types of information, such as an email address, travel date or a preference represented using a range.

HTML provides specialized input types for such information.

1.1.10.1 Email

```
<input type="email"
    name="email">
```

The email input indicates that the field is intended for an email address.

1.1.10.2 Date

```
<input type="date"
    name="travelDate">
```

The date input provides a control for entering a date.

1.1.10.3 Range

```
<input type="range"
    name="budget">
```

The range input provides a range-based control.

A tourism enquiry form can therefore include:

```
<form method="post">
   <label for=”email”>Email:</label>
   <input id=”email” type="email"
         name="email">
   <label for=”date”>Travel Date:</label>
   <input id=”date” type="date"
         name="travelDate">
   <label for=”budget”>Budget Preference:</label>
   <input id=”budget” type="range"
         name="budget">
   <button type="submit">
      Submit Enquiry
   </button>
</form>
```

Think About It

A tourist may access the same tourism website using a desktop computer, tablet or smartphone.

Should the webpage be designed with only one type of device in mind?

This question leads naturally to the next unit, where CSS and responsive web design will be introduced.

Summarised Overview

Browsers communicate with web servers to obtain web resources, while HTML provides the structural foundation of webpages. HTML documents follow a defined structure, and semantic elements communicate the purpose of different sections of a webpage. Headings, paragraphs, and lists help organize textual information, while images provide visual information and hyperlinks connect webpages and other resources. Tables are used to represent structured information, and forms enable websites to collect information from visitors. GET and POST are important methods used for submitting form data. HTML5 further provides specialized input types, such as email, date, and range, which support more appropriate and user-friendly data entry.

Assignments

Design a Kerala Tourism Webpage

The Kerala Tourism Department wants to develop a prototype webpage for exploring tourist destinations.

Design an HTML webpage for Kerala Explorer that presents information about selected Kerala tourist destinations.

The webpage should include:

- A semantic HTML structure
- Website heading
- Navigation links
- Destination descriptions
- At least one destination image
- A list of attractions or activities
- A table containing destination information
- A tourist enquiry form
- Appropriate HTML5 input types
- A suitable form submission method

Tasks

1. Develop the HTML page.
2. Identify the major HTML elements used.
3. Explain why semantic elements are appropriate for structuring the webpage.
4. Explain the choice of form controls.
5. Explain the choice of GET or POST for the form.

Suggested submission: HTML source code accompanied by a brief explanation of the design decisions.

Reference

1. Felke-Morris, T. (2012). *Web development and design foundations with HTML5.* Pearson Education.
2. Nixon, R. (2018). *Learning PHP, MySQL & JavaScript: With jQuery, CSS & HTML5*. " O'Reilly Media, Inc.".
3. Duckett, J. (2011). *HTML & CSS: design and build websites* (Vol. 15). Indianapolis, IN: Wiley.

Suggested Reading

1. Keith, Jeremy, and Rachel Andrew. *HTML5 for web designers*. Vol. 1. New York, NY: A Book Apart, 2010.
2. Pilgrim, M. (2010). *HTML5: up and running: dive into the future of web development.* " O'Reilly Media, Inc.".
3. Wempen, F. (2012). *Start Here! Learn HTML5*. Pearson Education.

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

2 UNIT

CSS Fundamentals and Responsive Web Design:Page Development

Learning Outcomes

After completion of this unit, the learner will be able to:

- explain the role of CSS in controlling the presentation, styling, and layout of HTML webpages
- apply inline, internal, and external stylesheets along with CSS selectors and specificity rules to style HTML elements appropriately
- apply CSS properties for colors, typography, backgrounds, spacing, dimensions, and the box model to create well-structured webpage layouts
- apply CSS display properties and Flexbox to construct one-dimensional webpage layouts
- apply CSS Grid to construct two-dimensional webpage layouts with appropriate rows, columns, and alignment
- design responsive and mobile-friendly webpages using viewport settings, media queries, and fluid images

Background

In the previous unit, Kerala Explorer was developed using HTML to provide the structural foundation of the tourism website.

The HTML page can contain a website heading, navigation menu, destination descriptions, images, lists, tables and forms. However, a webpage created only with HTML may appear plain and may not provide an effective visual experience for visitors.

Imagine opening the Kerala Explorer website and seeing the following:

Kerala Explorer

Home Munnar Wayanad Alappuzha Thekkady

Explore Kerala

Munnar

Munnar is a scenic hill destination...

[Image]

Popular Experiences

- • Tea plantations
- • Scenic viewpoints
- • Nature walks

Tourist Enquiry

Name: _____

Email: _____

The information is available, but the page does not yet communicate the visual identity of a tourism website.

A tourism website may need:

- attractive colours;
- suitable fonts;
- spacing between sections;
- visually distinct navigation;
- properly sized images;
- cards for destinations;
- flexible layouts;
- alignment of content;
- different layouts for desktop and mobile devices.

These presentation requirements can be addressed using CSS (Cascading Style Sheets).

CSS separates the presentation of a webpage from the structure provided by HTML. It allows developers to control colours, typography, spacing, borders, dimensions, positioning and responsive layouts.

For Kerala Explorer, CSS can transform a basic HTML document into a visually organized tourism interface.

The unit also introduces responsive web design, which enables webpages to adapt to different screen sizes and devices. This is important because tourists may access Kerala Explorer using desktop computers, tablets or smartphones.

Keywords

CSS Fundamentals, CSS Selectors and Specificity, CSS Box Model and Layout, Flexbox, CSS Grid, Responsive Web Design

Discussion

1.2.1 Role of CSS in Web Development

Consider the basic Kerala Explorer webpage developed in Unit 1.

HTML determines what the page contains:

```
<h1>Kerala Explorer</h1>
<p>
    Discover the beautiful tourist destinations of Kerala.
</p>
```

However, HTML alone does not determine the complete visual appearance of these elements.

CSS can be used to specify how these elements should appear.

For example:

```
h1 {
    color: darkgreen;
```

```
    font-size: 36px;
}
```

The CSS rule changes the appearance of the <h1> element.

CSS can therefore be viewed as a mechanism for describing the presentation and layout of HTML content.

Definition

CSS (Cascading Style Sheets) is a stylesheet language used to describe the presentation, appearance and layout of documents written in HTML.

Kerala Explorer Illustration

Suppose Kerala Explorer contains three destination sections:

Munnar

Wayanad

Alappuzha

HTML can provide the structure, while CSS can control:

- destination heading colour;
- font size;
- background colour;
- spacing;
- borders;
- image dimensions;
- alignment; and
- overall layout.

Thus, HTML and CSS perform complementary roles.

Think About It

HTML provides the structure of the Kerala Explorer webpage. What problems might occur if every visual property had to be specified directly inside each HTML element?

1.2.2 Applying CSS to an HTML Document

CSS can be applied to an HTML document in three common ways: inline CSS, internal CSS and external CSS.

1.2.2.1 Inline CSS

Inline CSS is written directly inside an HTML element using the style attribute.

For example:

```
<h1 style="color: darkgreen;">
   Kerala Explorer
</h1>
The style applies directly to that particular element.
Another example is:
<p style="font-size: 18px; color: #444;">
   Discover the beautiful destinations of Kerala.
</p>
```

Inline CSS can be useful when a style is required for a specific element. However, using inline styles extensively can make HTML difficult to maintain.

1.2.2.2 Internal CSS

Internal CSS is written inside a <style> element, normally within the <head> section of an HTML document.

```
<!DOCTYPE html>
<html>
<head>
   <title>Kerala Explorer</title>
   <style>
      h1 {
         color: darkgreen;
      }
      p {
         font-size: 18px;
      }
```

```
    </style>
</head>
<body>
    <h1>Kerala Explorer</h1>
    <p>
        Explore Kerala tourist destinations.
    </p>
</body>
</html>
```

Internal CSS is useful when styles are intended primarily for a single webpage.

1.2.2.3 External CSS

External CSS places the styles in a separate .css file.

For example, a file named style.css may contain:

```
body {
    font-family: Arial, sans-serif;
}
h1 {
    color: darkgreen;
}
p {
    font-size: 18px;
}
```

The HTML document can link the stylesheet using:

```
<link rel="stylesheet" href="style.css">
```

An external stylesheet is particularly useful when several webpages need to share the same visual design.

For example, Kerala Explorer may contain:

index.html

munnar.html

wayanad.html

alappuzha.html

thekkady.html

style.css

The same style.css file can be linked to multiple destination pages.

Comparison

Table 1.2.1 Location of CSS and its Suitable Use

Method	Location of CSS	Suitable Use
Inline	Inside HTML element	Styling a specific element
Internal	Inside <style> in the HTML document	Styling one webpage
External	Separate .css file	Sharing styles across multiple webpages

Kerala Explorer – Design Decision

If Munnar, Wayanad, Alappuzha and Thekkady pages must have the same navigation bar, typography and colour scheme, an external stylesheet is more appropriate than repeating the same styles on every webpage.

1.2.3 CSS Selectors

A CSS rule identifies the HTML elements to which a style should be applied.

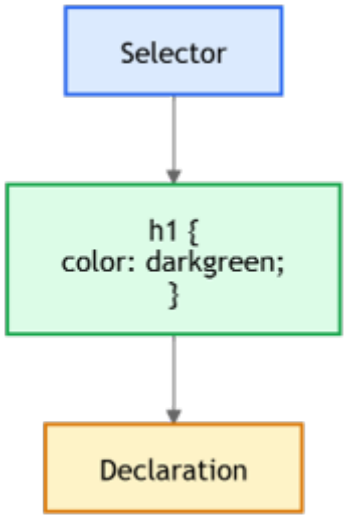
Consider:

```
h1 {
    color: darkgreen;
}
```

Here, h1 is the selector.

The selector identifies the HTML element to which the declaration applies.

A CSS rule can be represented as:



1.2.3.1 Element Selector

An element selector targets HTML elements based on their element name.

```
p {
    font-size: 18px;
}
```

This applies the style to paragraph elements.

1.2.3.2 Class Selector

A class selector targets elements having a particular class.

HTML:

```
<div class="destination">
    <h2>Munnar</h2>
    <p>Explore the scenic beauty of Munnar.</p>
</div>
```

CSS:

```
.destination {
    background-color: #f2f8f2;
    padding: 20px;
}
```

The period . identifies a class selector.

1.2.3.3 ID Selector

An ID selector targets an element with a particular id.

HTML:

```
<section id="featured">
    <h2>Featured Destination</h2>
</section>
```

CSS:

```
#featured {
    background-color: #e8f5e9;
}
```

The # symbol identifies an ID selector.

Selector Example

A Kerala Explorer page may therefore use:

```
h1 {
    color: darkgreen;
}
.destination {
    padding: 20px;
}
#featured {
    background-color: #e8f5e9;
}
```

1.2.4 CSS Specificity

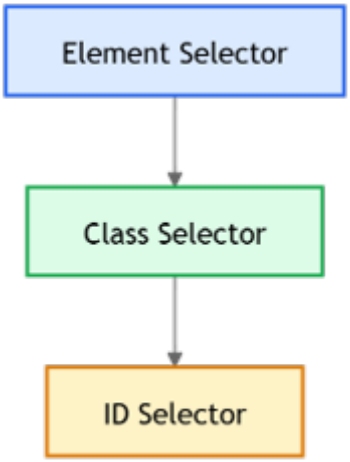
Suppose a Kerala Explorer destination heading receives styles from different CSS rules.

```
h2 {
    color: green;
}
.destination-title {
    color: blue;
}
```

```
The HTML element may be:
<h2 class="destination-title">
    Munnar
</h2>
```

Both rules can potentially apply to the same element. CSS uses specificity as one of the mechanisms for determining which rule takes precedence.

In the simplified specificity hierarchy considered here, ID selectors have greater specificity than class selectors, which in turn have greater specificity than element selectors.



For example:

```
h2 {
    color: green;
}
.destination-title {
    color: blue;
}
#munnar-title {
    color: brown;
}
HTML:
<h2 id="munnar-title"
    class="destination-title">
    Munnar
</h2>
```

The ID selector has greater specificity than the class and element selectors.

Definition

CSS specificity is a mechanism used by the CSS cascade to determine which applicable rule has greater priority when multiple rules target the same element.

Specificity becomes important when a webpage contains many styles and selectors.

Think About It

If two CSS rules target the same Munnar heading but specify different colours, how can a developer determine which rule will be applied?

1.2.5 Colors, Typography and Backgrounds

A tourism website depends strongly on visual presentation.

Kerala Explorer can use CSS to control the colour, typography and background of different sections.

Colors

CSS supports different ways of representing colours.

For example:

```
h1 {
    color: darkgreen;
}
```

A hexadecimal colour value can also be used:

```
h1 {
    color: #176b3a;
}
```

Backgrounds

A background colour can be applied to a section.

```
.destination {
    background-color: #f4f8f4;
}
```

A background image can also be specified:

```
.hero {
    background-image: url("munnar.jpg");
    background-size: cover;
    background-position: center;
}
```

Typography

Typography controls the appearance of textual content.

```
body {
    font-family: Arial, sans-serif;
    font-size: 16px;
    line-height: 1.6;
}
h1 {
    font-size: 36px;
    font-weight: bold;
}
```

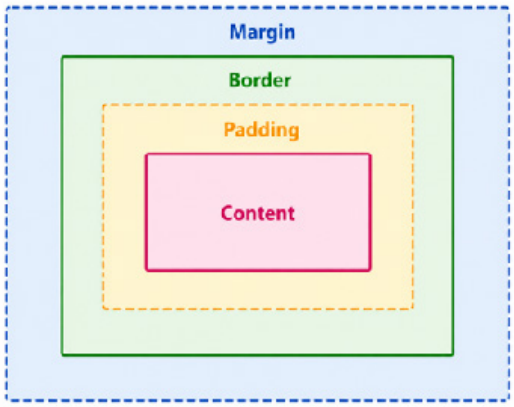
The properties include:

- font-family;
- font-size;
- font-weight; and
- line-height.

For example, Kerala Explorer can use larger typography for the main website heading and smaller typography for supporting descriptions.

1.2.6 CSS Box Model

Consider a destination card on Kerala Explorer.



The CSS box model describes how the size and spacing of an element are organized.

The four main components are:

- Content
- Padding
- Border
- Margin

Content

The content is the actual text, image or other material inside the element.

Padding

Padding creates space between the content and the border.

```
.destination {
    padding: 20px;
}
```

Border

A border surrounds the padding and content.

```
.destination {
    border: 1px solid #cccccc;
}
```

Margin

Margin creates space outside the element.

```
.destination {
    margin: 20px;
}
```

These properties can be combined:

```
.destination {
    margin: 20px;
    border: 1px solid #cccccc;
    padding: 20px;
}
```

Kerala Explorer Illustration

Suppose each destination is presented as a card.

The **content** contains the destination name and description.

The **padding** creates internal space.

The **border** visually separates the card.

The **margin** creates space between neighbouring cards.

1.2.7 Display Properties

The display property controls how an element participates in layout.

Common display values include:

- block
- inline
- inline-block
- none

A block-level element normally occupies the available width and starts on a new line.

```
section {
    display: block;
}
```

An inline element normally occupies only the space required by its content.

```
a {
    display: inline;
}
```

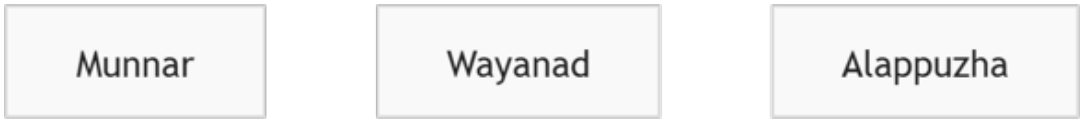
An element can also be hidden using:

```
.menu {
    display: none;
}
```

The display property becomes particularly important when creating layouts using Flexbox and Grid.

1.2.8 Flexbox for One-Dimensional Layouts

Suppose Kerala Explorer wants to display three destination cards in a horizontal row:



Flexbox provides a convenient way to arrange elements along a single dimension: a row or a column.

Consider the HTML:

```
<div class="destinations">
    <div class="card">
        <h3>Munnar</h3>
        <p>Hill destination</p>
    </div>
    <div class="card">
        <h3>Wayanad</h3>
        <p>Nature destination</p>
    </div>
    <div class="card">
        <h3>Alappuzha</h3>
        <p>Backwater destination</p>
```

```
    </div>
</div>
CSS:
.destinations {
    display: flex;
}
.card {
    padding: 20px;
}
```

The parent container becomes a flex container.

Flex Direction

The direction can be controlled using flex-direction.

```
.destinations {
    display: flex;
    flex-direction: row;
}
```

A column layout can be created using:

```
.destinations {
    display: flex;
    flex-direction: column;
}
```

Justifying Content

The justify-content property controls alignment along the main axis.

```
.destinations {
    display: flex;
    justify-content: space-between;
}
```

Aligning Items

The align-items property controls alignment along the cross axis.

```
.destinations {
    display: flex;
    align-items: center;
}
```

Wrapping

If the available width is insufficient, flex items can be allowed to wrap.

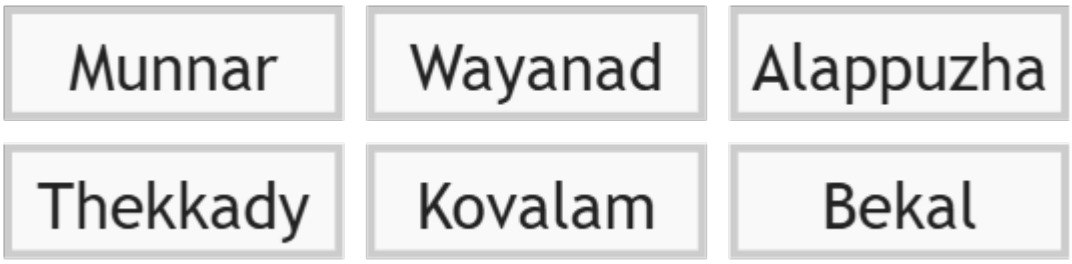
```
.destinations {
    display: flex;
    flex-wrap: wrap;
}
```

Definition

Flexbox is a CSS layout model designed primarily for arranging elements along a single dimension.

1.2.9 CSS Grid for Two-Dimensional Layouts

Suppose Kerala Explorer wants to display six destinations in a structured grid:



CSS Grid is designed for two-dimensional layouts involving rows and columns.

Consider:

```
.destination-grid {
    display: grid;
    grid-template-columns: repeat(3, 1fr);
    gap: 20px;
}
```

The HTML can contain destination cards:

```
<div class="destination-grid">
    <div class="card">Munnar</div>
    <div class="card">Wayanad</div>
    <div class="card">Alappuzha</div>
    <div class="card">Thekkady</div>
    <div class="card">Kovalam</div>
    <div class="card">Bekal</div>
</div>
```

The grid-template-columns property defines the column structure.

The gap property creates space between grid items.

Responsive Grid

The number of columns can be changed for smaller screens using a media query:

```
    .destination-grid {
        display: grid;
        grid-template-columns: repeat(3, 1fr);
        gap: 20px;
    }
@media (max-width: 768px) {
   .destination-grid {
       grid-template-columns: repeat(2, 1fr);
   }
}
```

Definition

CSS Grid is a two-dimensional layout system that allows elements to be arranged using rows and columns.

Table 1.2.2 Comparison of Flexbox and CSS Grid

Flexbox	CSS Grid
Primarily one-dimensional	Two-dimensional
Row or column	Rows and columns
Useful for navigation and linear arrangements	Useful for card and page layouts
Content-oriented layouts	Structured grid-oriented layouts

1.2.10 Responsive Web Design

A tourist may access Kerala Explorer using:

- a desktop computer;
- a laptop;
- a tablet; or
- a smartphone.

These devices have different screen dimensions.

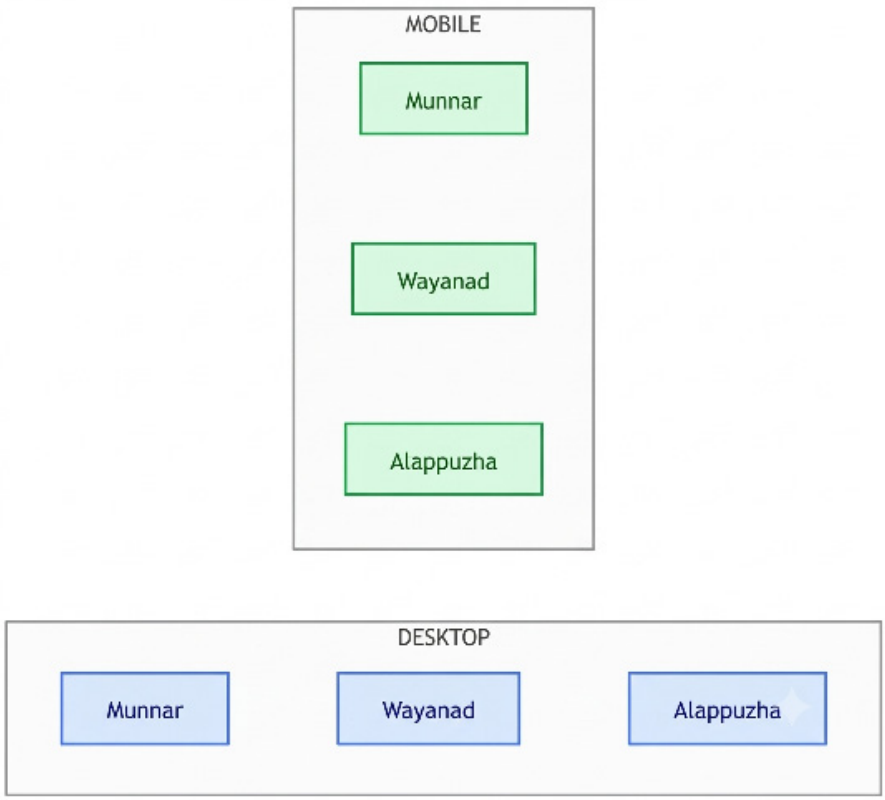
A layout designed only for a large desktop screen may become difficult to use on a smartphone.

Responsive Web Design is an approach in which webpage layouts and content adapt to different screen sizes and devices.

A responsive design may change:

- number of columns;
- image dimensions;
- navigation layout;
- spacing;
- typography; and
- alignment.

Kerala Explorer Illustration



The content remains available, but the layout adapts to the available screen width.

1.2.11 Viewport Meta Tag

A mobile browser needs information about how the webpage should be scaled and displayed.

The viewport meta tag is commonly placed inside the <head> section:

<meta name="viewport"

content="width=device-width, initial-scale=1.0">

The width=device-width setting makes the viewport width correspond to the device's screen width.

The initial-scale=1.0 setting specifies the initial zoom level.

A responsive Kerala Explorer page can therefore begin with:

```
<!DOCTYPE html>
<html>
<head>
   <meta name="viewport"
        content="width=device-width, initial-scale=1.0">
   <title>Kerala Explorer</title>
</head>
<body>
   <h1>Kerala Explorer</h1>
</body>
</html>
```

1.2.12 Media Queries

A responsive design often needs different CSS rules at different screen widths.

CSS media queries allow styles to be applied when specified conditions are satisfied.

For example:

```
    .destination-grid {
        display: grid;
        grid-template-columns: repeat(3, 1fr);
        gap: 20px;
    }
@media (max-width: 768px) {
   .destination-grid {
      grid-template-columns: repeat(2, 1fr);
   }
}
@media (max-width: 480px) {
```

```
    .destination-grid {
      grid-template-columns: 1fr;
    }
}
```

The design therefore changes progressively:

Large screen → 3 columns

Tablet → 2 columns

Mobile → 1 column

Definition

A media query is a CSS mechanism used to apply styles conditionally according to characteristics such as viewport width.

Kerala Explorer Application

The desktop version of Kerala Explorer may show three destination cards per row. On a tablet it may show two cards, while on a smartphone it may show one card per row.

The same HTML content can therefore support different layouts through CSS.

Think About It

Should a developer create completely separate HTML pages for desktop, tablet and mobile devices when the content is essentially the same?

1.2.13 Fluid Images

Large fixed-width images may create horizontal scrolling on smaller screens.

Consider:

```
img {
    width: 800px;
}
```

An image with a fixed width of 800 pixels may not fit comfortably on a narrow smartphone screen.

A common responsive approach is:

```
img {
    max-width: 100%;
    height: auto;
}
```

This allows the image to scale down when necessary while maintaining its aspect ratio.

For Kerala Explorer:

```
<img src="munnar.jpg"
     alt="Scenic landscape of Munnar">
```

with:

```
img {
   max-width: 100%;
   height: auto;
}
```

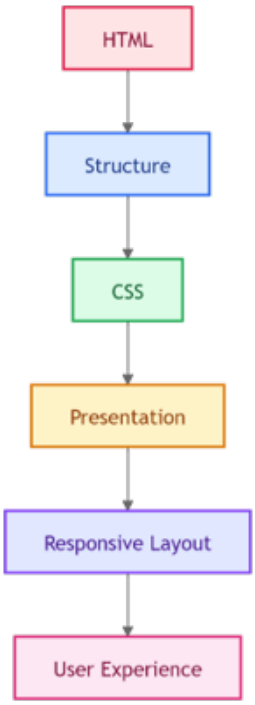
allows the image to adapt to the width available within its containing element.

Think About It

Suppose Kerala Explorer adds six more destinations. Should the developer manually create a separate CSS layout for every destination, or should the existing reusable card layout be used?

Kerala Explorer – From Structure to Experience

HTML gives Kerala Explorer its content structure, while CSS gives that structure a visual presentation.



The separation of structure and presentation makes it easier to modify the visual design without rewriting the underlying content.

Think About It

A tourist may access Kerala Explorer while travelling using a smartphone with a narrow screen.

If the destination cards, images and navigation are designed only for a large desktop screen, what usability problems might occur?

This question highlights why responsive web design is an important part of modern web development.

Summarised Overview

CSS controls the presentation and layout of HTML webpages and can be applied using inline styles, internal stylesheets, and external stylesheets. CSS selectors identify the HTML elements to which styles are applied, while specificity determines the priority when multiple applicable selectors compete. CSS can be used to control colours, typography, backgrounds, and other visual properties. The CSS box model consists of content, padding, border, and margin, while the display property controls how elements participate in the layout. Flexbox supports one-dimensional layouts, whereas CSS Grid supports two-dimensional layouts. Responsive web design enables webpages to adapt to different screen sizes. The viewport meta tag supports appropriate rendering on mobile devices, while media queries allow CSS rules to change according to viewport conditions. Fluid images help prevent layout problems on smaller screens. Combining CSS Grid, Flexbox, media queries, and fluid images helps create responsive and mobile-friendly webpages.

Assignments

Design a Responsive Kerala Tourism Webpage

The Kerala Explorer tourism portal wants to improve the webpage developed in Unit 1 by introducing a visually attractive and responsive design.

Develop a CSS-based presentation for the Kerala Explorer webpage.

Demonstrate the webpage at desktop, tablet and mobile screen sizes and explain the layout changes produced by the media queries.

The webpage should:

- use an external CSS stylesheet;
- apply suitable colours and typography;
- style the navigation menu;
- create destination cards;
- apply the CSS box model;
- use Flexbox for an appropriate one-dimensional layout;
- use CSS Grid for displaying tourist destinations;
- include responsive images;
- include the viewport meta tag;
- use media queries for different screen sizes; and
- provide a mobile-friendly layout.

Tasks

1. Develop the HTML structure for the Kerala Explorer webpage.
2. Develop an external CSS stylesheet.
3. Apply suitable selectors and CSS properties.
4. Explain the use of the CSS box model in the destination cards.
5. Identify where Flexbox is used and justify its use.
6. Identify where CSS Grid is used and justify its use.
7. Develop media queries for tablet and mobile layouts.
8. Explain how the webpage adapts to different screen sizes.

Suggested submission: HTML and CSS source code accompanied by a brief explanation of the design and responsive layout decisions.

Reference

1. Duckett, J. (2011). *HTML & CSS: design and build websites* (Vol. 15). Indianapolis, IN: Wiley.
2. Meyer, E., & Weyl, E. (2023). *CSS: The Definitive Guide.* " O'Reilly Media, Inc.".
3. Robbins, J. N. *Learning Web Design.* O'Reilly Media.

Suggested Reading

1. Frain, Ben. *Responsive Web Design with HTML5 and CSS: Build future-proof responsive websites using the latest HTML5 and CSS techniques.* Packt Publishing Ltd, 2025.
2. Attardi, Joe. "Modern CSS." Berkeley, CA Apress 1 (2020).
3. Grant, K. J. (2024). *CSS in Depth*. Simon and Schuster.

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

3 UNIT

Javascript Essentials and Dom Manipulation Development

Learning Outcomes

After completion of this unit, the learner will be able to:

- explain the role of JavaScript in developing interactive webpages
- apply variables, data types, operators and control structures to solve client-side programming problems
- develop JavaScript functions using appropriate scope and parameter-passing techniques
- apply modern JavaScript features such as array methods, object literals, destructuring, spread/rest operators, arrow functions and template literals
- select and manipulate HTML elements using the Document Object Model (DOM) and create, modify and remove webpage elements dynamically using JavaScript

Background

In the previous two units, Kerala Explorer was developed using HTML and CSS.

HTML provided the structure of the tourism website, while CSS controlled its presentation and responsive layout. The website can now display destination information in an attractive manner across different devices.

However, consider what happens when a tourist interacts with the website.

A tourist may want to:

1. select a tourist destination;
2. view additional information about Munnar;

3. calculate the estimated travel budget;
4. display a personalized greeting;
5. filter destinations according to preferences;
6. add a new destination card;
7. remove an unwanted destination;
8. change the content of a webpage without reloading it.

HTML and CSS alone cannot provide this level of behaviour.

For example, suppose Kerala Explorer contains three destination cards:

Munnar

Wayanad

Alappuzha

A tourist clicks a View Details button associated with Munnar. The website should respond immediately by displaying additional information.

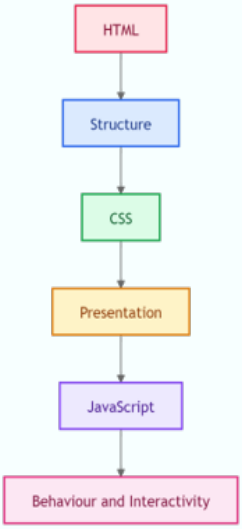
Similarly, a tourist may select Wildlife as an interest and expect Kerala Explorer to display only destinations associated with wildlife experiences.

Such behaviour requires programming logic within the webpage.

JavaScript is a programming language commonly used to add behaviour and interactivity to webpages.

JavaScript can process data, make decisions, perform calculations, respond to user interactions and modify the content displayed in a webpage.

The interaction between HTML, CSS and JavaScript can therefore be viewed as:



In this unit, JavaScript will be introduced through the Kerala Explorer application. The concepts will progress from basic JavaScript syntax to manipulating webpage content through the Document Object Model (DOM).

Keywords

JavaScript Fundamentals, Modern JavaScript Syntax, Arrays and Objects, Functions and Scope, DOM Manipulation, Dynamic Webpage Content

Discussion

1.3.1 Role of JavaScript in Web Development

Consider the Kerala Explorer webpage developed in the previous units.

HTML can create a button:

```
<button>
    View Munnar Details
</button>
```

CSS can make the button visually attractive:

```
button {
    padding: 10px 20px;
    background-color: darkgreen;
    color: white;
}
```

But what should happen when the tourist clicks the button?

JavaScript can define the behaviour associated with the interaction.

For example:

```
const message = "Munnar is a scenic hill destination.";
console.log(message);
```

```
    JavaScript can also modify webpage content.
    <p id="destinationInfo">
        Select a destination to view details.
    </p>
document.getElementById("destinationInfo").textContent =
    "Munnar is known for its scenic landscapes and tea plantations.";
```

The content displayed in the paragraph can therefore be changed using JavaScript.

Definition

JavaScript is a programming language used to add logic, behaviour and interactivity to webpages and web applications.

Kerala Explorer Illustration

JavaScript can enable Kerala Explorer to:

- respond to button clicks;
- display destination information;
- calculate values;
- process user-selected information
- filter destinations;
- update webpage content; and
- dynamically add or remove destination elements.

Thus, HTML, CSS and JavaScript perform complementary roles.

Table 1.3.1 Web Technology and its Primary Role

Technology	Primary Role
HTML	Structure
CSS	Presentation
JavaScript	Behaviour and interactivity

Think About It

Suppose a Kerala Explorer webpage contains a button labelled View Details. Which technology should determine what happens when the tourist clicks the button?

1.3.2 Variables and Constants

A JavaScript program often needs to store information for later use.

For example, Kerala Explorer may need to store the name of a selected destination.

let destination = "Munnar";

The variable destination stores the value "Munnar".

JavaScript provides let and const for declaring variables and constants.

Using let

A variable declared using let can be assigned a new value.

let destination = "Munnar";

destination = "Wayanad";

console.log(destination);

The final value is "Wayanad".

Using const

A variable declared using const cannot be reassigned to another value after initialization.

const websiteName = "Kerala Explorer";

console.log(websiteName);

An attempt to reassign it would result in an error.

const websiteName = "Kerala Explorer";

websiteName = "Kerala Travel";

Therefore, const is suitable when a binding should not be reassigned, while let is appropriate when the value needs to change.

Naming Variables

Meaningful variable names improve readability.

Prefer:

let selectedDestination = "Munnar";

over:

let x = "Munnar";

Kerala Explorer Example

```
let selectedDestination = "Munnar";
let numberOfVisitors = 4;
const websiteName = "Kerala Explorer";
```

These declarations represent information that could be used by the tourism application.

1.3.3 JavaScript Data Types

A JavaScript program works with different types of data.

For example, Kerala Explorer may need to store:

```
let destination = "Munnar";
let visitors = 4;
let hasBooking = false;
```

These values represent different types of data.

String

A string represents textual data.

```
let destination = "Munnar";
```

Number

Numbers represent numeric values.

```
let visitors = 4;
let estimatedBudget = 25000;
```

Boolean

A Boolean represents either true or false.

```
let hasBooking = false;
```

Undefined

A variable that has been declared but has not been assigned a value has the value undefined.

```
let selectedDestination;
console.log(selectedDestination);
```

Null

null can be used to explicitly represent the absence of a value.

```
let selectedDestination = null;
```

Object

Objects can represent collections of related information.

```
const destination = {
    name: "Munnar",
    type: "Hill destination",
    state: "Kerala"
};
```

Array

An array can store a collection of values.

```
const destinations = [
    "Munnar",
    "Wayanad",
    "Alappuzha"
];
```

The typeof operator can be used to inspect the type of a value.

```
console.log(typeof "Munnar");
console.log(typeof 25000);
console.log(typeof true);
```

1.3.4 Operators

Operators allow JavaScript to perform operations on values.

Arithmetic Operators

Arithmetic operators can be used for calculations.

```
let accommodation = 5000;
let travel = 3000;
let food = 2000;
```

```
let total = accommodation + travel + food;
console.log(total);
```

Other arithmetic operators include:

+ Addition

- Subtraction

* Multiplication

/ Division

% Remainder

Comparison Operators

Comparison operators compare values.

```
let visitors = 4;
console.log(visitors > 2);
console.log(visitors === 4);
console.log(visitors !== 2);
```

The result of a comparison is a Boolean value.

Logical Operators

Logical operators combine conditions.

```
let hasTicket = true;
let hasValidDate = true;
console.log(hasTicket && hasValidDate);
```

Common logical operators include:

&& Logical AND

|| Logical OR

! Logical NOT

Assignment Operators

Assignment operators assign values to variables.

```
let budget = 10000;
budget += 5000;
console.log(budget);
```

Kerala Explorer Example

Suppose Kerala Explorer calculates the total estimated cost:

```
let travelCost = 3000;
let accommodationCost = 5000;
let foodCost = 2000;
let totalCost =
  travelCost +
  accommodationCost +
  foodCost;
console.log(totalCost);
```

1.3.5 Control Flow

A program often needs to make decisions.

Suppose Kerala Explorer wants to display a message depending on the selected destination.

An if statement can be used.

```
let destination = "Munnar";
if (destination === "Munnar") {
   console.log("Explore the tea plantations of Munnar.");
}
```

if...else

Different actions can be performed depending on a condition.

```
let destination = "Wayanad";
if (destination === "Munnar") {
   console.log("Explore the hill landscapes of Munnar.");
} else {
   console.log("Explore another Kerala destination.");
}
```

else if

Multiple conditions can be evaluated.

```
let destination = "Alappuzha";
if (destination === "Munnar") {
    console.log("Hill destination");
} else if (destination === "Alappuzha") {
    console.log("Backwater destination");
} else if (destination === "Thekkady") {
    console.log("Wildlife destination");
} else {
    console.log("Other destination");
}
```

Loops

A loop can be used when an operation needs to be repeated.

For example:

```
const destinations = [
    "Munnar",
    "Wayanad",
    "Alappuzha"
];
for (let i = 0; i < destinations.length; i++) {
    console.log(destinations[i]);
}
```

The loop processes each destination.

for...of

A for...of loop provides a convenient way to iterate over values in an iterable such as an array.

```
const destinations = [
    "Munnar",
    "Wayanad",
    "Alappuzha"
];
for (const destination of destinations) {
    console.log(destination);
}
```

Kerala Explorer Illustration

A tourism website may use control flow to determine which destination information should be displayed or which destinations should be processed.

1.3.6 Functions and Scope

A function groups instructions that perform a particular task.

Suppose Kerala Explorer repeatedly needs to calculate an estimated trip cost.

Instead of writing the calculation repeatedly, a function can be defined.

```
function calculateTotal(travel, accommodation, food) {
    return travel + accommodation + food;
}
const total = calculateTotal(3000, 5000, 2000);
console.log(total);
```

The function receives values through parameters and returns a result.

Function Parameters

Consider:

```
function greetTourist(name) {
    return "Welcome to Kerala Explorer, " + name;
}
console.log(greetTourist("Anu"));
```

Here, name is a parameter.

Function Scope

Variables declared inside a function are generally local to that function.

```
function showDestination() {
    let destination = "Munnar";
    console.log(destination);
}
showDestination();
```

The variable destination belongs to the function's local scope.

A variable declared outside the function has a wider scope:

```
let destination = "Munnar";
function showDestination() {
    console.log(destination);
}
showDestination();
```

Kerala Explorer Example

```
function getDestinationMessage(destination) {
    return "Explore " + destination + " with Kerala Explorer.";
}
console.log(
    getDestinationMessage("Munnar")
);
```

Functions make programs easier to organize, reuse and maintain.

1.3.7 Arrays

Kerala Explorer needs to manage collections of destinations.

An array can store multiple values:

```
const destinations = [
    "Munnar",
    "Wayanad",
```

```
    "Alappuzha",
    "Thekkady"
];
```

Individual values can be accessed using an index.

console.log(destinations[0]);

The first element is "Munnar".

The length of the array can be obtained using:

```
console.log(destinations.length);
Arrays can contain objects as well.
const destinations = [
    {
        name: "Munnar",
        type: "Hill destination"
    },
    {
        name: "Alappuzha",
        type: "Backwater destination"
    }
];
```

This representation is more useful for a tourism application because each destination can contain multiple properties.

1.3.8 Array Methods: map(), filter() and reduce()

Modern JavaScript provides methods that make it easier to process arrays.

map()

The map() method creates a new array by applying a function to each element.

Suppose Kerala Explorer needs destination labels:

```
const destinations = [
    "Munnar",
```

```
    "Wayanad",
    "Alappuzha"
];
const labels = destinations.map(
    destination => "Visit " + destination
);
console.log(labels);
```

The resulting array contains:

Visit Munnar

Visit Wayanad

Visit Alappuzha

filter()

The filter() method creates a new array containing elements that satisfy a condition.

```
const destinations = [
    {
        name: "Munnar",
        type: "Hill"
    },
    {
        name: "Thekkady",
        type: "Wildlife"
    },
    {
        name: "Wayanad",
        type: "Hill"
    }
];
```

```
const hillDestinations =
   destinations.filter(
      destination => destination.type === "Hill"
   );
console.log(hillDestinations);
```

The result contains Munnar and Wayanad.

reduce()

The reduce() method processes an array and produces a single accumulated result.

Suppose Kerala Explorer stores estimated costs:

```
const costs = [
   5000,
   3000,
   2000
];
const total = costs.reduce(
   (sum, cost) => sum + cost,
   0
);
console.log(total);
```

The result is:

10000

Kerala Explorer Application

These methods can be useful when processing tourism data:

- map() → transform destination data;
- filter() → select destinations matching a condition;
- reduce() → calculate a total.

1.3.9 Object Literals

A tourism destination contains several related properties.

Instead of storing each property separately:

```
let name = "Munnar";
let type = "Hill destination";
let state = "Kerala";
an object can group the related information.
const destination = {
   name: "Munnar",
   type: "Hill destination",
   state: "Kerala"
};
```

The properties can be accessed using dot notation:

```
console.log(destination.name);
console.log(destination.type);
```

Bracket notation can also be used:

```
console.log(destination["name"]);
```

Array of Destination Objects

A realistic tourism application can maintain multiple destination objects.

```
const destinations = [
   {
      name: "Munnar",
      type: "Hill destination"
   },
   {
      name: "Wayanad",
      type: "Nature destination"
   },
```

```
    {
        name: "Alappuzha",
        type: "Backwater destination"
    }
];
```

This data structure can later be used to generate destination cards dynamically.

1.3.10 Destructuring

Suppose a destination object contains several properties:

```
const destination = {
    name: "Munnar",
    type: "Hill destination",
    state: "Kerala"
};
Instead of repeatedly writing:
console.log(destination.name);
console.log(destination.type);
object destructuring can be used.
const {
    name,
    type
} = destination;
console.log(name);
console.log(type);
```

Destructuring extracts values from an object into variables.

Array Destructuring

Destructuring can also be used with arrays.

```
const destinations = [
    "Munnar",
    "Wayanad",
    "Alappuzha"
];
const [
    firstDestination,
    secondDestination
] = destinations;
console.log(firstDestination);
console.log(secondDestination);
```

Kerala Explorer Application

Destructuring can make code that processes destination data more readable.

```
const destination = {
    name: "Munnar",
    type: "Hill destination"
};
const {
    name,
    type
} = destination;
console.log(
    `${name} is a ${type}.`
);
```

1.3.11 Spread and Rest Operators

JavaScript uses the ... syntax for both spread and rest operations, depending on the context.

Spread Operator

Suppose Kerala Explorer has two destination arrays:

```
const hillDestinations = [
    "Munnar",
    "Wayanad"
];
const otherDestinations = [
    "Alappuzha",
    "Thekkady"
];
```

The spread operator can combine them:

```
const allDestinations = [
    ...hillDestinations,
    ...otherDestinations
];
console.log(allDestinations);
```

The spread syntax expands the elements of the arrays into the new array.

Rest Parameter

The rest parameter collects multiple arguments into an array.

```
function calculateTotal(...costs) {
    return costs.reduce(
        (sum, cost) => sum + cost,
        0
    );
}
console.log(
    calculateTotal(3000, 5000, 2000)
);
```

Here, ...costs collects the supplied arguments into an array.

1.3.12 Arrow Functions

Arrow functions provide a concise syntax for defining functions.

A traditional function can be written as:

```
function greet(name) {
    return "Welcome to " + name;
}
```

The equivalent arrow function is:

```
const greet = (name) => {
    return "Welcome to " + name;
};
```

For a simple expression, the function can be shortened:

```
const greet =
    name => "Welcome to " + name;
```

Arrow functions are particularly useful with array methods.

For example:

```
const destinations = [
    "Munnar",
    "Wayanad",
    "Alappuzha"
];
const labels = destinations.map(
    destination => "Visit " + destination
);
```

1.3.13 Template Literals

JavaScript can construct strings using template literals.

A template literal is enclosed using backticks.

```
const destination = "Munnar";
```

```
const message =
  `Explore ${destination} with Kerala Explorer.`;
console.log(message);
```

The ${...} syntax allows expressions to be embedded within the string.

For example:

```
const destination = "Munnar";
const visitors = 4;
const message =
  `${visitors} visitors selected ${destination}.`;
console.log(message);
```

This is particularly useful when generating dynamic webpage content.

Kerala Explorer Example

```
const destination = {
  name: "Munnar",
  type: "Hill destination"
};
const cardTitle =
  `${destination.name} - ${destination.type}`;
console.log(cardTitle);
```

DOM Manipulation

1.3.14 Understanding the Document Object Model

HTML represents the structure of a webpage.

When a browser loads an HTML document, it creates a structured representation of the document known as the Document Object Model (DOM).

Consider:

```
<body>
  <h1>Kerala Explorer</h1>
  <p>Munnar</p>
</body>
```

The browser represents the document as a hierarchy of nodes.

JavaScript can access and modify these nodes.

Definition

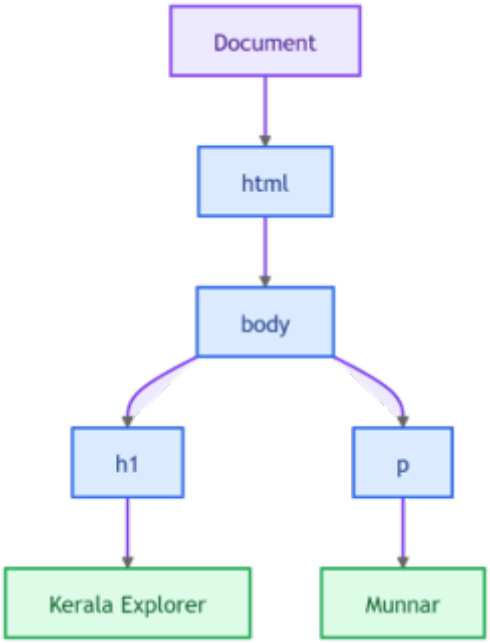
The Document Object Model (DOM) is a programming interface that represents the structure of an HTML document as a tree of objects that scripts can access and manipulate.

This allows JavaScript to modify parts of the webpage dynamically without requiring the entire page to be reloaded.

Kerala Explorer Illustration

Suppose the webpage initially displays:

Selected Destination



Munnar

JavaScript can modify the displayed destination when the tourist selects Wayanad.

The DOM therefore acts as the connection between the HTML document and JavaScript code.

1.3.15 Selecting DOM Elements

Before modifying an element, JavaScript needs to select it.

Consider:

<h1 id="title">

Kerala Explorer

</h1>

The element can be selected using getElementById().

const title =

document.getElementById("title");

querySelector()

The querySelector() method selects the first element matching a CSS selector.

const title =

document.querySelector("#title");

A class selector can also be used:

const card =

document.querySelector(".destination-card");

querySelectorAll()

The querySelectorAll() method selects all elements matching a CSS selector.

const cards =

document.querySelectorAll(".destination-card");

This is useful when Kerala Explorer contains multiple destination cards.

Kerala Explorer Example

```
<h2 class="destination-name">
   Munnar
</h2>
<h2 class="destination-name">
   Wayanad
</h2>
const destinations =
   document.querySelectorAll(
```

```
    ".destination-name"
  );
console.log(destinations);
```

1.3.16 Modifying Content

Once an element has been selected, JavaScript can modify its content.

Consider:

```
<p id="destinationInfo">
  Select a destination.
</p>
JavaScript can change the text using textContent.
const info =
  document.getElementById(
    "destinationInfo"
  );
info.textContent =
  "Explore the scenic beauty of Munnar.";
```

The displayed content changes dynamically.

innerHTML

The innerHTML property can be used when HTML markup needs to be inserted.

```
info.innerHTML =
  "<strong>Munnar</strong> is a hill destination.";
```

When inserting content, developers should carefully distinguish between plain text and HTML markup, particularly when the content comes from users or external sources.

Kerala Explorer Example

```
<h2 id="destinationName">
  Destination
</h2>
<p id="description">
```

```
  Select a destination.
</p>
const destinationName =
  document.getElementById(
    "destinationName"
  );
const description =
  document.getElementById(
    "description"
  );
destinationName.textContent = "Munnar";
description.textContent =
  "Explore the scenic hill landscapes and tea plantations of Munnar.";
```

1.3.17 Modifying Attributes

HTML elements contain attributes that provide additional information.

Consider:

```
<img id="destinationImage"
     src="munnar.jpg"
     alt="Scenic landscape of Munnar">
JavaScript can modify an attribute using setAttribute().
const image =
  document.getElementById(
    "destinationImage"
  );
image.setAttribute(
  "src",
  "wayanad.jpg"
);
```

```
image.setAttribute(
    "alt",
    "Landscape of Wayanad"
);
```

The image source and alternative text are changed dynamically.

An attribute can also be read:

```
const source =
    image.getAttribute("src");
console.log(source);
```

Kerala Explorer Application

A single destination card can therefore display different destination images based on the tourist's selection.

1.3.18 Modifying Styles

JavaScript can also modify the style of an element.

Consider:

```
<p id="message">
    Welcome to Kerala Explorer.
</p>
```

The style can be changed using the style property.

```
const message =
  document.getElementById("message");
message.style.color = "darkgreen";
message.style.fontWeight = "bold";
```

The appearance of the selected element changes dynamically.

However, when several style changes are required, modifying CSS classes is often more maintainable.

For example:

```
.highlight {
    background-color: #e8f5e9;
    padding: 15px;
}
```

JavaScript can add the class:

```
message.classList.add("highlight");
```

A class can be removed using:

```
message.classList.remove("highlight");
```

Kerala Explorer Application

A selected destination card could be highlighted when the tourist chooses it.

1.3.19 Creating DOM Nodes Dynamically

Suppose Kerala Explorer receives a new destination and needs to display it on the webpage.

JavaScript can create an HTML element dynamically.

```
const card =
    document.createElement("article");
```

Content can then be added:

```
card.textContent = "Kovalam";
```

A class can be assigned:

```
card.classList.add(
    "destination-card"
);
```

The new element can be inserted into the webpage.

```
const container =
    document.querySelector(
        ".destination-grid"
    );
```

```
container.appendChild(card);
```

The complete example is:

```
const container =
    document.querySelector(
        ".destination-grid"
    );
const card =
    document.createElement("article");
card.classList.add(
    "destination-card"
);
card.textContent = "Kovalam";
container.appendChild(card);
```

A new destination card is therefore created without manually editing the original HTML document.

1.3.20 Removing DOM Nodes

JavaScript can also remove elements from the DOM.

Suppose Kerala Explorer contains a destination card that needs to be removed.

```
const card =
    document.querySelector(
        ".destination-card"
    );
card.remove();
```

The selected element is removed from the webpage.

An element can also be removed from its parent using:

container.removeChild(card);

The remove() method provides a direct way to remove an element.

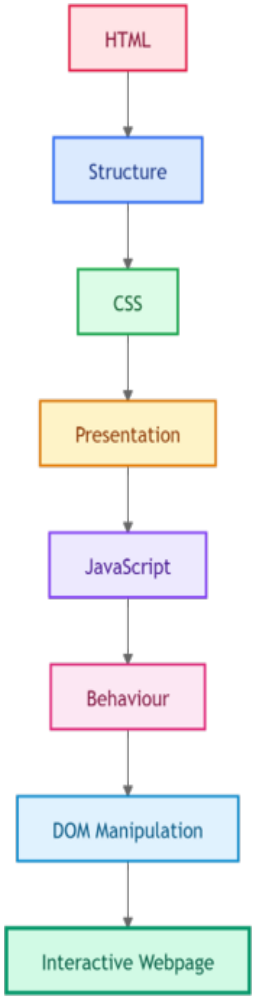
Kerala Explorer Application

Dynamic removal can be useful when:

- a destination is no longer available;
- a user removes a selected destination;
- a filter hides unwanted content; or
- dynamically generated content needs to be cleared.

Kerala Explorer – From Static Page to Interactive Application

The development of Kerala Explorer can now be viewed as a progression:



HTML defines what the webpage contains.

CSS determines how it looks.

JavaScript determines how it behaves.

The DOM provides the programming interface through which JavaScript can access and modify the webpage.

Think About It

A tourist selects Munnar and then Wayanad from a destination selector. The webpage changes the displayed information without reloading the entire page.

Why might this provide a better user experience than navigating to a completely new webpage every time?

Summarised Overview

JavaScript adds behaviour and interactivity to webpages and provides programming features for developing dynamic web applications. The let keyword is used for variables whose bindings may be reassigned, while const is used for bindings that should not be reassigned. JavaScript supports various data types, including strings, numbers, Boolean values, objects, and arrays, along with arithmetic, comparison, logical, and assignment operators. Conditional statements and loops control program execution, while functions organize reusable programming logic and scope determines where variables can be accessed. Arrays store collections of values and provide methods such as map(), filter(), and reduce() for common array-processing operations. Object literals represent structured data using properties and values, while destructuring extracts values from arrays and objects. Spread syntax expands values into another structure, rest parameters collect multiple function arguments, arrow functions provide concise function syntax, and template literals support dynamic string construction. The Document Object Model (DOM) represents an HTML document as a structured tree, allowing JavaScript to select elements and modify their content, attributes, and styles. JavaScript can also create and remove DOM nodes dynamically, enabling interactive webpage behaviour.

Assignments

Develop an Interactive Kerala Tourism Webpage

The Kerala Explorer tourism portal wants to enhance its responsive webpage by introducing JavaScript-based interactivity.

Develop an interactive webpage that allows visitors to select a Kerala tourist destination and dynamically display information about the selected destination.

The webpage should:

- use an appropriate HTML structure;
- use CSS for basic presentation;
- store destination information using JavaScript objects or arrays;
- use appropriate variables and constants;
- use functions to organize the JavaScript logic;
- use at least one array method such as map(), filter() or reduce();
- use an appropriate DOM selection method;
- dynamically modify webpage content;
- dynamically modify at least one attribute or style;
- create or remove at least one DOM element dynamically; and
- provide an interactive destination-selection mechanism.

Tasks

1. Design the HTML structure for the Kerala Explorer webpage.
2. Create JavaScript data representing at least five Kerala tourist destinations.
3. Implement a destination selection mechanism.
4. Display the selected destination dynamically using DOM manipulation.
5. Use an appropriate function to organize the interaction.
6. Use at least one modern JavaScript feature such as an arrow function, destructuring, spread/rest or template literal.
7. Use an array method such as map(), filter() or reduce().
8. Create at least one webpage element dynamically using JavaScript.
9. Provide an option to remove dynamically generated content.
10. Explain how JavaScript interacts with the DOM in your application.

Suggested submission: HTML, CSS and JavaScript source code accompanied by a brief explanation of the data structures, JavaScript features and DOM manipulation techniques used.

Reference

1. Haverbeke, M. (2011). *Eloquent javascript: A modern introduction to programming*. No Starch Press.

2. Duckett, J. (2014). *Javascript and jquery: Interactive front-end web development.* Wiley Publishing.

3. Nixon, R. (2018). *Learning PHP, MySQL & JavaScript: With jQuery, CSS & HTML5*. " O'Reilly Media, Inc.".

4. Felke-Morris, T. (2012). *Web development and design foundations with HTML5.* Pearson Education.

Suggested Reading

1. Duckett, J. *JavaScript and JQuery: Interactive Front-End Web Development.* Wiley.

2. Haverbeke, M. *Eloquent JavaScript: A Modern Introduction to Programming*. No Starch Press.

3. Nixon, R. *Learning PHP, MySQL & JavaScript: With jQuery, CSS & HTML5.* O'Reilly Media.

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

4 UNIT

Event Handling, Form Validation, and Asynchronous JavaScript

Learning Outcomes

After completion of this unit, the learner will be able to:

- explain the role of events and event handling in creating interactive webpages
- apply mouse and keyboard event handling using JavaScript, including event listeners and control of default browser behaviour
- develop client-side form validation with appropriate validation feedback
- apply JavaScript timers, Promises, async/await, and the event loop to handle asynchronous operations
- parse and process JSON data and use fetch() to retrieve data from APIs while handling responses and errors appropriately
- display dynamically retrieved data on a webpage using DOM manipulation

Background

In the previous unit, Kerala Explorer was transformed from a static webpage into an interactive webpage using JavaScript and DOM manipulation.

JavaScript was used to select elements, modify content, change attributes and styles, and dynamically create or remove DOM elements.

However, consider a more realistic tourism website.

A tourist may:

- click a destination card;

- move the mouse over an image;
- type a destination name;
- submit a tourist enquiry form;
- enter an invalid email address;
- request information from a remote server;
- wait while destination data is being retrieved;
- see a loading message;
- receive an error message when the server is unavailable; or
- view newly retrieved destination information without reloading the page.

These interactions require JavaScript to respond to events and handle operations that may take time to complete.

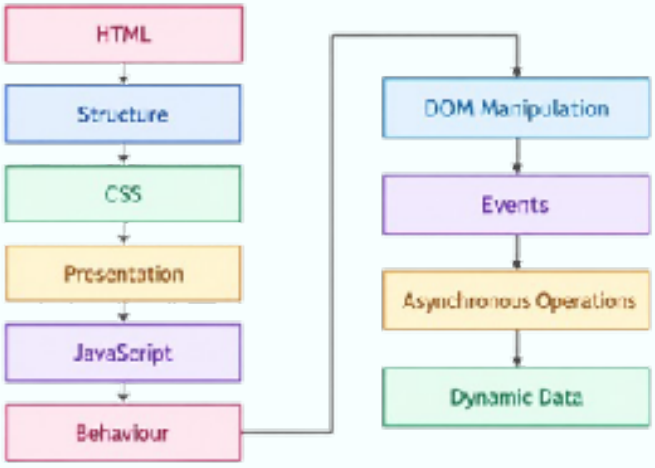
For example, when a tourist clicks the View Details button, JavaScript should execute a specific function.

Similarly, when the tourist submits an enquiry form, the browser should first validate the entered information before the form is processed.

When destination information is stored on a remote server, the webpage cannot assume that the information will be available immediately. JavaScript must handle the operation while allowing the webpage to remain responsive.

This introduces the concepts of event handling, form validation and asynchronous JavaScript.

The development of Kerala Explorer can now be viewed as:



Keywords

Event Handling, Form Validation, Timers, Promises, Async/Await, Event Loop, JSON, Fetch API, API Response Handling, Dynamic Data Display

Discussion

1.4.1 Understanding Events in JavaScript

A webpage becomes interactive when it can respond to actions performed by the user or by the browser.

For example, a tourist may click a button:

```
<button id="viewDetails">
    View Munnar Details
</button>
```

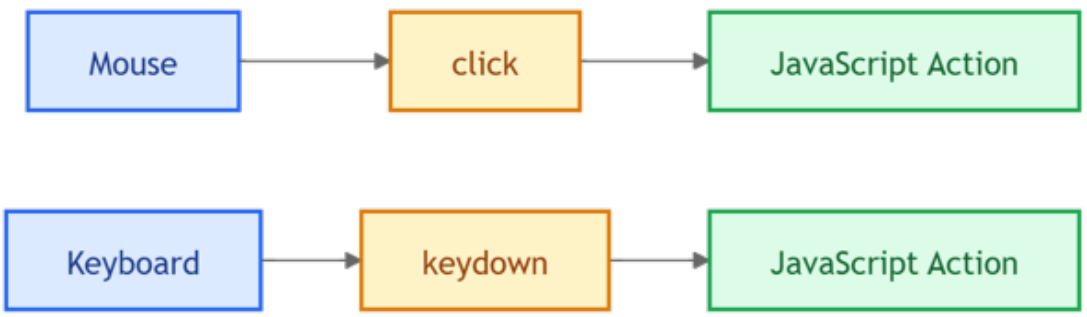
The click is an event.

JavaScript can respond to this event and perform an action.

```
function showMessage() {
    console.log("Munnar details requested.");
}
```

Events can be generated by different actions.

Examples include:



Definition

An event is an occurrence or action detected by the browser that can be handled using JavaScript.

Examples include:

- clicking a button;
- pressing a keyboard key;
- moving the mouse;
- submitting a form; and
- changing the value of a form control.

Kerala Explorer Example

When a tourist clicks View Details, Kerala Explorer can display additional information about the selected destination.

1.4.2 Mouse Events

Mouse events occur when a user interacts with webpage elements using a mouse or similar pointing device.

Common mouse events include:

1. click
2. dblclick
3. mouseenter
4. mouseleave
5. mouseover
6. mouseout

Consider a Kerala Explorer button:

```
<button id="viewDetails">
    View Details
</button>
```

A click event can be handled using JavaScript.

```
const button =
    document.getElementById("viewDetails");
button.addEventListener(
    "click",
    function () {
        console.log(
            "View Details button clicked."
        );
    }
);
```

The function is executed when the button is clicked.

Mouse Enter

A destination card can respond when the pointer enters the card.

```
const card =
    document.querySelector(
        ".destination-card"
    );
card.addEventListener(
    "mouseenter",
    function () {
        console.log(
            "Pointer entered the destination card."
        );
    }
);
```

Kerala Explorer Application

Mouse events can be used to:

- ♦ display additional information;

- highlight a destination;
- show a tooltip;
- respond to a button click; or
- change the visual state of a destination card.

1.4.3 Keyboard Events

Keyboard events occur when users interact with the webpage using a keyboard.

Common keyboard events include:

- keydown
- keyup

Consider a search field:

```
<input
  id="destinationSearch"
  type="text"
  placeholder="Search destination">
```

A keyboard event can be detected using:

```
const search =
  document.getElementById(
    "destinationSearch"
  );
search.addEventListener(
  "keydown",
  function (event) {
    console.log(
      "Key pressed:",
      event.key
    );
  }
);
```

The event.key property identifies the key that was pressed.

Kerala Explorer Example

Suppose a tourist types:

Munnar

into a destination search box.

JavaScript can detect each key press and use the entered information to perform an operation.

1.4.4 Using addEventListener()

The addEventListener() method is commonly used to attach an event handler to an element.

Its general form is:

```
element.addEventListener(
    "event",
    function () {
        // action
    }
);
```

For example:

```
const button =
    document.getElementById("viewDetails");
button.addEventListener(
    "click",
    function () {
        console.log(
            "Destination details requested."
        );
    }
);
```

The method separates the event type from the function that should execute.

Reusable Function

The function can be defined separately:

```
function showDestination() {
    console.log(
        "Displaying destination details."
    );
}
const button =
    document.getElementById("viewDetails");
button.addEventListener(
    "click",
    showDestination
);
```

This can improve readability when the event-handling logic becomes larger.

Kerala Explorer Application

A destination card, button or search field can have its own event listener.

1.4.5 The Event Object

When an event occurs, the browser provides information about the event through an event object.

For example:

```
button.addEventListener(
    "click",
    function (event) {
        console.log(event.type);
    }
);
```

The event.type property identifies the event.

For keyboard events:

```
search.addEventListener(
    "keydown",
    function (event) {
        console.log(event.key);
    }
);
```

The event object therefore provides information that can help JavaScript determine how to respond.

1.4.6 Preventing Default Browser Behaviour

Some HTML elements have default browser behaviour.

For example, submitting a form normally causes the browser to submit the form.

Consider:

```
<form id="enquiryForm">
    <input
        id="name"
        type="text"
        placeholder="Your name">
    <button type="submit">
        Submit Enquiry
    </button>
</form>
```

JavaScript can intercept the submission:

```
const form =
    document.getElementById(
        "enquiryForm"
    );
form.addEventListener(
```

```
    "submit",
    function (event) {
        event.preventDefault();
        console.log(
            "Form submission intercepted."
        );
    }
);
```

The preventDefault() method prevents the browser's default action.

Kerala Explorer Application

This is useful when Kerala Explorer needs to:

- receive the form submission;
- validate the input;
- display feedback; and
- submit the information only after validation succeeds.

Form Validation

1.4.7 Client-Side Form Validation

A tourism website may contain an enquiry form:

```
<form id="enquiryForm">
    <input
        id="name"
        type="text"
        placeholder="Name">
    <input
        id="email"
        type="email"
        placeholder="Email">
```

```
    <button type="submit">
      Send Enquiry
    </button>
</form>
```

Client-side validation improves usability and provides immediate feedback, but server-side validation is still required when data is submitted to a server.

Client-side validation performs these checks in the browser.

For example:

```
const form =
  document.getElementById(
    "enquiryForm"
  );
form.addEventListener(
  "submit",
  function (event) {
    event.preventDefault();
    const name =
      document.getElementById(
        "name"
      ).value.trim();
    if (name === "") {
      console.log(
        "Please enter your name."
      );
    }
  }
);
```

The trim() method removes unnecessary whitespace from the beginning and end of the entered text.

Kerala Explorer Application

The enquiry form can validate:

- name;
- email;
- selected destination;
- number of visitors; and
- enquiry message.

1.4.8 Providing Validation Feedback

Validation should not merely detect an error; it should communicate the problem clearly to the user.

HTML:

```
<form id="enquiryForm">
  <input
    id="name"
    type="text"
    placeholder="Name">
  <p id="nameError"></p>
  <button type="submit">
    Send Enquiry
  </button>
</form>
```

JavaScript:

```
const nameInput =
  document.getElementById("name");
const nameError =
  document.getElementById(
    "nameError"
  );
```

```
if (nameInput.value.trim() === "") {
    nameError.textContent =
        "Please enter your name.";
}
```

A successful validation can also provide feedback:

```
nameError.textContent =
    "Name entered successfully.";
```

Better Feedback

A CSS class can be used to visually indicate an error:

```
.error {
    color: red;
    font-size: 14px;
}
```

JavaScript:

```
nameError.classList.add("error");
```

1.4.9 Validating an Email Address

The email field can use HTML's built-in email input type:

```
<input
    id="email"
    type="email"
    placeholder="Email address">
```

JavaScript can also perform a simple client-side check.

```
const email =
    document.getElementById(
        "email"
    ).value.trim();
if (email === "") {
    console.log(
```

```
        "Email address is required."
      );
    }
```

A simple pattern can be used when a basic format check is required:

```
const emailPattern =
  /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
  if (!emailPattern.test(email)) {
    console.log(
      "Enter a valid email address."
    );
  }
```

This provides a basic format check; it does not prove that the email address actually exists.

1.4.10 Asynchronous JavaScript

1.4.10.1 Synchronous and Asynchronous Operations

JavaScript normally executes statements in sequence.

For example:

```
console.log("First");
console.log("Second");
console.log("Third");
```

The output is:

```
First
Second
Third
```

This is a simple synchronous sequence.

However, some operations may take time to complete.

Examples include:

- waiting for a timer;
- retrieving data from a server;

- reading external information; and
- waiting for an asynchronous operation to finish.

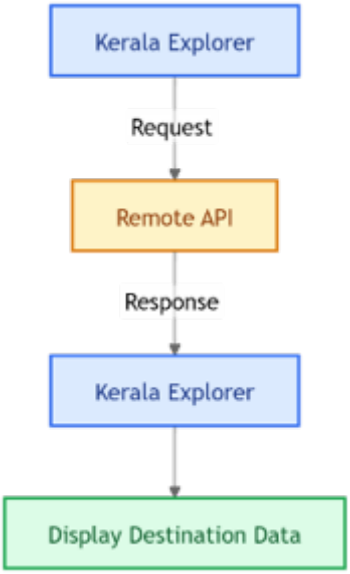
JavaScript provides mechanisms for handling such operations without unnecessarily blocking other work.

Definition

An asynchronous operation is an operation whose completion may occur later, allowing other JavaScript work to continue while the operation is pending.

Kerala Explorer Example

Suppose destination information is stored on a remote server.



The webpage needs to wait for the response, but it should remain responsive while waiting.

1.4.10.2 setTimeout()

The setTimeout() function schedules a function to execute after a specified delay.

For example:

```
console.log("Start");
setTimeout(
  function () {
    console.log(
      "Destination information ready."
    );
```

```
    },
    2000
);
console.log("End");
```

The output begins with:

```
Start
End
```

Destination information ready.

The callback is scheduled for execution after approximately two seconds.

Kerala Explorer Application

A loading message could be displayed temporarily:

```
console.log(
    "Loading destination information..."
);
setTimeout(
    function () {
        console.log(
            "Destination information loaded."
        );
    },
    2000
);
```

1.4.10.3 setInterval()

The setInterval() function repeatedly executes a function at a specified interval.

For example:

```
let count = 0;
const timer =
    setInterval(
```

```
        function () {
          count++;
          console.log(count);
          if (count === 3) {
            clearInterval(timer);
          }
        },
        1000
    );
```

The function executes approximately every second until clearInterval() stops it.

Kerala Explorer Application

A tourism website could use an interval to rotate promotional messages:

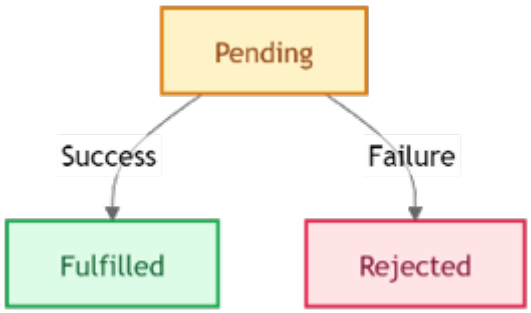
```
const messages = [
    "Explore Munnar",
    "Discover Wayanad",
    "Experience Alappuzha"
];
let index = 0;
const timer =
    setInterval(
        function () {
            console.log(
                messages[index]
            );
            index++;
            if (index === messages.length) {
                clearInterval(timer);
            }
        },
        3000
    );
```

1.4.10.4 Understanding Promises

When an asynchronous operation is started, its result may not be available immediately.

A Promise represents the eventual completion or failure of an asynchronous operation.

A Promise has three commonly described states:



Consider:

```
const destinationData =
    new Promise(
        function (resolve, reject) {
            const success = true;
            if (success) {
                resolve(
                    "Destination data loaded."
                );
            } else {
                reject(
                    "Unable to load data."
                );
            }
        }
    );
```

The Promise can be handled using:

```
destinationData
    .then(
        function (message) {
            console.log(message);
```

```
    }
  )
  .catch(
    function (error) {
      console.log(error);
    }
  );
```

Definition

A Promise is an object representing the eventual completion or failure of an asynchronous operation.

1.4.10.5 async and await

The async and await keywords provide a readable way to work with Promises.

An asynchronous function is declared using async.

```
async function loadDestination() {
  return "Munnar";
}
```

An await expression waits for a Promise to settle within an asynchronous function.

```
async function loadDestination() {
  const destination =
    await Promise.resolve(
      "Munnar"
    );
  console.log(destination);
}
loadDestination();
```

Error Handling

try...catch can be used with async and await.

```
async function loadDestination() {
  try {
```

```
        const destination =
            await Promise.resolve(
                "Munnar"
            );
        console.log(destination);
    } catch (error) {
        console.log(
            "Unable to load destination."
        );
    }
}
```

Kerala Explorer Application

When retrieving destination information from an API, async/await can make the code easier to read.

1.4.10.6 Understanding the Event Loop

JavaScript uses an event-driven execution model.

Consider:

```
console.log("A");
setTimeout(
    function () {
        console.log("B");
    },
    0
);
console.log("C");
```

The output is:

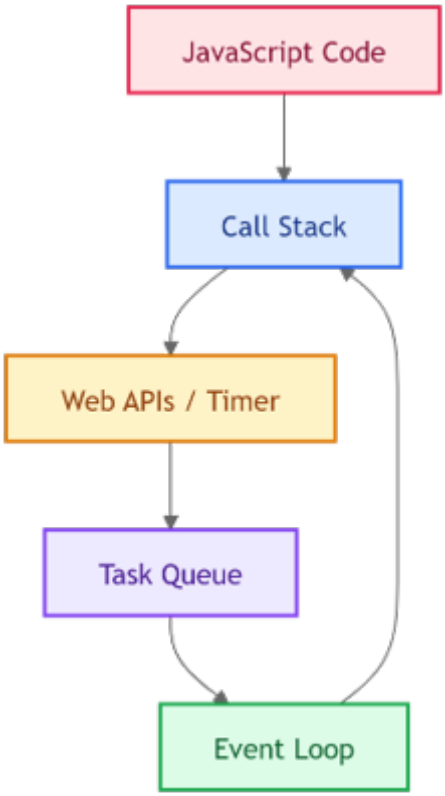
A

C

B

Although the timer delay is zero milliseconds, its callback does not execute immediately.

A simplified view is:



The event loop helps coordinate asynchronous callbacks with JavaScript's execution mechanism.

Kerala Explorer Illustration

When Kerala Explorer starts retrieving destination information, JavaScript can continue processing other work while waiting for the asynchronous operation to complete.

1.4.11 Understanding JSON

Tourism information exchanged between a webpage and a server can be represented using JSON (JavaScript Object Notation).

Example:

```
{
  "name": "Munnar",
  "type": "Hill destination",
  "state": "Kerala"
}
```

A collection of destinations can be represented as:

```
[
  {
    "name": "Munnar",
    "type": "Hill destination"
  },
  {
    "name": "Wayanad",
    "type": "Nature destination"
  },
  {
    "name": "Alappuzha",
    "type": "Backwater destination"
  }
]
```

JSON is commonly used for exchanging structured data between applications.

1.4.12 Parsing JSON

JavaScript provides methods for converting between JSON text and JavaScript values.

1. **JSON.parse()**

   - JSON.parse() converts a JSON-formatted string into a JavaScript value.

```
const jsonData =
  '{"name":"Munnar","type":"Hill destination"}';
const destination =
  JSON.parse(jsonData);
console.log(destination.name);
```

2. **JSON.stringify()**

   - JSON.stringify() converts a JavaScript value into a JSON string.

```
const destination = {
    name: "Munnar",
    type: "Hill destination"
};
const jsonData =
    JSON.stringify(destination);
console.log(jsonData);
```

1.4.13 Fetching Data Using fetch()

The fetch() function can be used to request resources from a server.

A simplified example is:

```
fetch(
    "https://example.com/api/destinations"
);
```

fetch() returns a Promise representing the response to the request.

A common pattern is:

```
fetch(
    "https://example.com/api/destinations"
)
    .then(
        response =>
            response.json()
    )
    .then(
        data => {
            console.log(data);
        }
    );
```

The first .then() converts the response body into JavaScript data using response.json().

Using async/await

The same operation can be written using async/await:

```
async function loadDestinations() {
    const response =
        await fetch(
            "https://example.com/api/destinations"
        );
    const data =
        await response.json();
    console.log(data);
}
```

Kerala Explorer Application

A remote API could provide destination information such as:

Destination Name

Destination Type

Description

Image

Popular Activities

The webpage can retrieve this information and display it dynamically.

1.4.14 Handling Responses and Errors

A successful HTTP request does not automatically mean that the requested resource was available.

The response should therefore be checked.

```
async function loadDestinations() {
    try {
        const response =
            await fetch(
                "https://example.com/api/destinations"
```

```
            );
        if (!response.ok) {
            throw new Error(
                "Unable to retrieve destinations."
            );
        }
        const data =
            await response.json();
        console.log(data);
    } catch (error) {
        console.log(
            "Error:",
            error.message
        );
    }
}
```

The response.ok property can be checked before processing the response.

Why Error Handling Matters

A remote request can fail because:

- the server is unavailable;
- the requested resource does not exist;
- the network connection fails; or
- the response cannot be processed as expected.

A good application should provide appropriate feedback instead of leaving the user uncertain about what happened.

1.4.15 Displaying Dynamic API Data

The data retrieved from an API can be displayed by creating or updating DOM elements.

Suppose the API returns:

```
[
  {
    "name": "Munnar",
    "type": "Hill destination"
  },
  {
    "name": "Wayanad",
    "type": "Nature destination"
  },
  {
    "name": "Alappuzha",
    "type": "Backwater destination"
  }
]
```

HTML:

```
<section>
  <h2>Popular Destinations</h2>
  <div id="destinationList"></div>
</section>
```

JavaScript:

```
async function loadDestinations() {
  const response =
    await fetch(
      "https://example.com/api/destinations"
    );
  if (!response.ok) {
    throw new Error(
      "Unable to load destinations."
    );
```

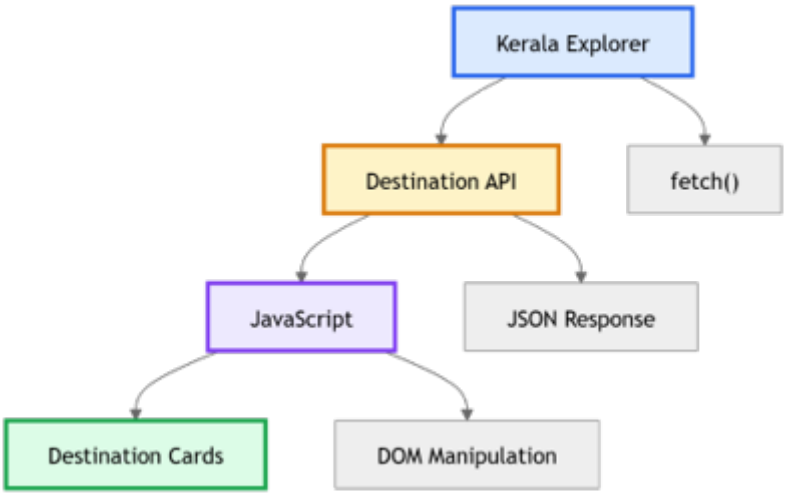
```
    }
    const destinations =
      await response.json();
    const container =
      document.getElementById(
        "destinationList"
      );
    destinations.forEach(
      destination => {
        const card =
          document.createElement(
            "article"
          );
        card.classList.add(
          "destination-card"
        );
        card.innerHTML = `
          <h3>${destination.name}</h3>
          <p>${destination.type}</p>
        `;
        container.appendChild(card);
      }
    );
}
```

The function can be called using:

```
loadDestinations();
```

When inserting data obtained from external or untrusted sources, avoid using innerHTML with untrusted content; prefer safer DOM methods such as textContent where HTML markup is not required.

Kerala Explorer Data Flow

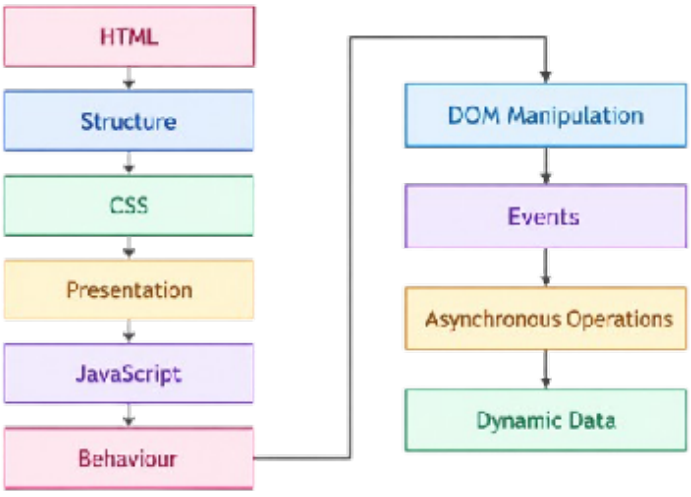


This represents the transition from static webpage content to dynamically retrieved content.

> **Think About It**
>
> Suppose the enquiry information must now be sent to a remote server instead of being displayed only in the browser. Which asynchronous JavaScript mechanism could be used to send the data and receive a response?

Kerala Explorer – From Interaction to Dynamic Data

The development of Kerala Explorer can now be represented as:



At this stage, Kerala Explorer is no longer simply a collection of static webpages.

It can:

- respond to user actions;
- validate information;
- communicate with external services;
- process asynchronous responses; and
- display dynamically retrieved information.

Think About It

A tourist submits an enquiry while the destination information is being retrieved from a remote server.

Why should the webpage continue responding to the tourist instead of appearing completely unresponsive while waiting for the server?

This situation demonstrates the importance of asynchronous JavaScript.

Summarised Overview

JavaScript events represent actions or occurrences detected by the browser and include mouse events such as click and mouseenter and keyboard events such as keydown and keyup. The addEventListener() method attaches event handlers to elements, while the event object provides information about the event and preventDefault() can be used to prevent default browser behaviour. Client-side validation checks user input before processing, and validation feedback communicates input errors to users. JavaScript supports asynchronous programming using setTimeout() to schedule functions after a delay, setInterval() to repeatedly execute functions, and Promises to represent the eventual completion or failure of asynchronous operations. The async and await keywords simplify working with asynchronous operations, while the event loop coordinates asynchronous callbacks with JavaScript execution. JSON provides a structured format for data exchange, with JSON.parse() converting JSON text into JavaScript data and JSON.stringify() converting JavaScript data into JSON text. The fetch() method retrieves resources from a server, and API responses should be checked and handled appropriately. Errors can be handled using suitable mechanisms such as try...catch. Retrieved data can then be displayed dynamically through DOM manipulation, enabling webpages to respond to user actions and server-provided data.

Assignments

Develop a Dynamic Kerala Tourism Enquiry and Destination Page

The Kerala Explorer tourism portal wants to enhance its website by introducing event-driven interaction, client-side validation and dynamically retrieved destination information.

Develop a webpage that allows tourists to submit an enquiry and view destination information retrieved from an API.

The webpage should:

- contain an enquiry form;
- handle form submission using addEventListener();
- prevent the browser's default form submission;
- validate required fields on the client side;
- provide meaningful validation feedback;
- use at least one mouse or keyboard event;
- use async/await;
- use fetch() to retrieve destination data;
- process JSON data;
- handle unsuccessful responses or errors; and
- display the retrieved destination information dynamically using DOM manipulation.

Tasks

1. Design the HTML structure for the Kerala Explorer enquiry page.
2. Implement event handling using addEventListener().
3. Prevent the default form submission behaviour.
4. Validate the tourist's name, email and selected destination.
5. Display suitable feedback for invalid input.
6. Create an asynchronous function to retrieve destination information.

7. Use fetch() to request JSON data.
8. Check the response before processing the data.
9. Use try...catch to handle errors.
10. Display the retrieved destination information dynamically on the webpage.
11. Explain how the event loop and asynchronous JavaScript contribute to the responsiveness of the application.

Suggested submission: HTML, CSS and JavaScript source code accompanied by a brief explanation of the event handling, validation and asynchronous programming techniques used.

Reference

1. Haverbeke, M. *Eloquent JavaScript: A Modern Introduction to Programming.* No Starch Press.
2. Duckett, J. *JavaScript and JQuery: Interactive Front-End Web Development.* Wiley.
3. Nixon, R. *Learning PHP, MySQL & JavaScript: With jQuery, CSS & HTML5.* O'Reilly Media.
4. Felke-Morris, T. *Web Development and Design Foundations with HTML5.* Pearson.

Suggested Reading

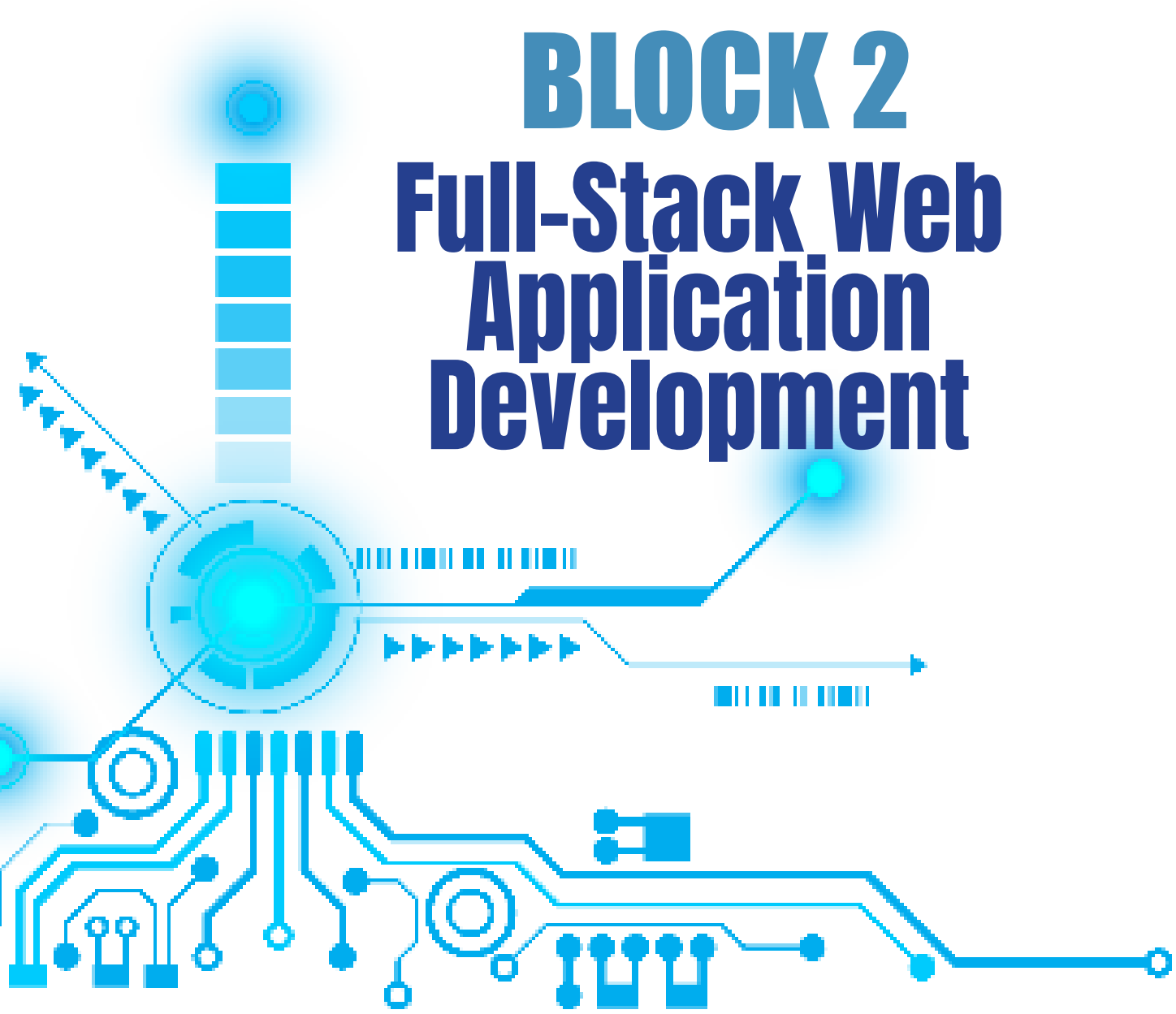
1. Duckett, J. *JavaScript and JQuery: Interactive Front-End Web Development.* Wiley.
2. Haverbeke, M. *Eloquent JavaScript: A Modern Introduction to Programming.* No Starch Press.
3. Nixon, R. *Learning PHP, MySQL & JavaScript: With jQuery, CSS & HTML5.* O'Reilly Media.

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

BLOCK 2

Full-Stack Web Application Development



1 UNIT

Node.JS Fundamentals and Restful Web Services

Learning Outcomes

After completion of this unit, the learner will be able to:

- explain the role of Node.js in server-side web application development and describe its event-driven programming model
- manage Node.js projects using npm and package.json, and apply built-in modules such as fs, path, and http
- develop a basic web server using Node.js and process client-server communication using request and response objects
- create Express.js applications and configure routes to implement GET, POST, PUT, and DELETE operations
- apply Express.js features such as static file serving, middleware, and JSON request processing using express.json()
- design RESTful routes using appropriate HTTP methods and status codes, and implement suitable error-handling middleware

Background

In Block I, Kerala Explorer was developed as an interactive frontend application.

The website can now:

1. display Kerala tourist destinations;
2. respond to user interactions;
3. validate tourist enquiries;

4. retrieve dynamic information;

5. process JSON data; and

6. communicate with external APIs.

However, consider what happens when a tourist submits an enquiry.

The information entered by the tourist needs to be sent somewhere for processing.

Similarly, destination information may need to be stored and retrieved from a server rather than being permanently written inside the webpage.

For example, Kerala Explorer may need to provide:

GET /destinations

GET /destinations/1

POST /destinations

PUT /destinations/1

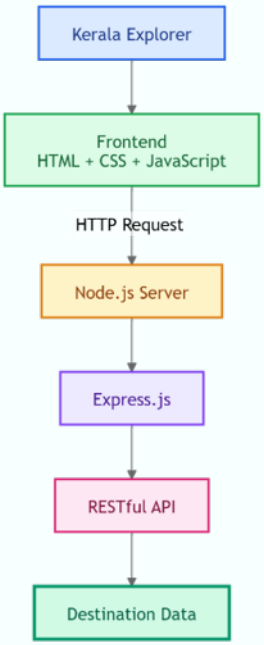
DELETE /destinations/1

These requests need to be received and processed by a server-side application.

This is where Node.js can be used.

Node.js allows JavaScript to execute outside the browser and provides an environment for developing server-side applications.

The development of Kerala Explorer can now be viewed as:



The focus of this unit is therefore to understand how a JavaScript-based backend can receive requests, process them and return responses.

Keywords

Node.js Fundamentals, Event-Driven Architecture, npm and Project Management, Node.js Built-in Modules, Express.js, Routing, Request and Response, Middleware, RESTful API Design, HTTP Status Codes

Discussion

2.1.1 Understanding Node.js

JavaScript was originally designed to execute inside a web browser.

For example:

console.log("Welcome to Kerala Explorer");

The browser provides the environment in which this JavaScript code executes.

However, JavaScript can also be executed outside the browser using Node.js.

Definition

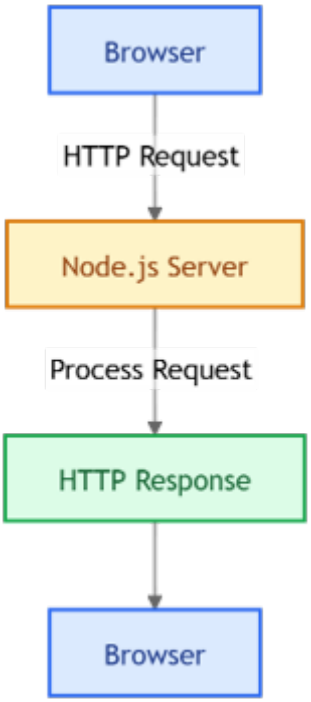
Node.js is a JavaScript runtime environment that allows JavaScript code to execute outside the browser.

Node.js can be used to develop:

- web servers;
- REST APIs;
- backend applications;
- command-line tools; and
- network applications.

Kerala Explorer Example

Suppose Kerala Explorer needs a backend server.



The browser remains responsible for the frontend, while Node.js can handle server-side operations.

Why Node.js?

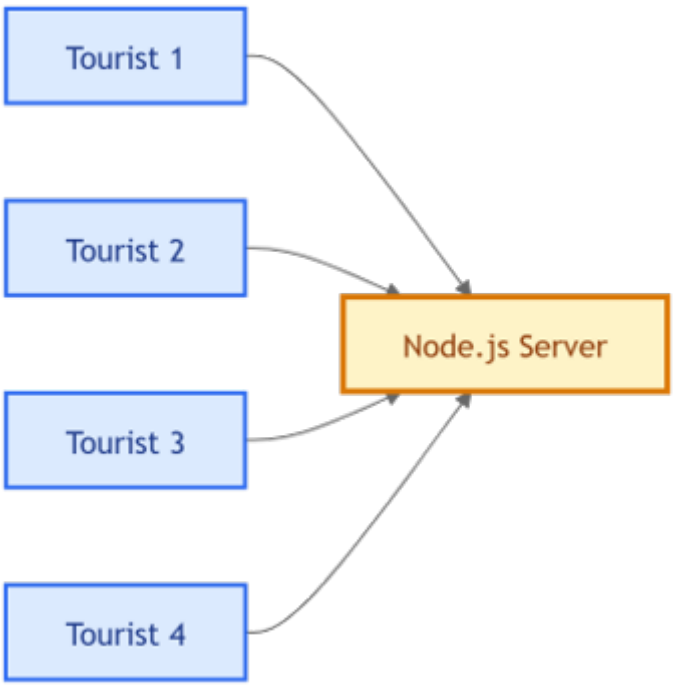
Node.js is useful for web development because:

1. it uses JavaScript on the server;
2. it supports event-driven programming;
3. it provides built-in modules for common server-side tasks;
4. it has a large package ecosystem through npm; and
5. it can be used with frameworks such as Express.js.

2.1.2 Node.js and the Event-Driven Model

A server may receive many requests.

For example, Kerala Explorer may simultaneously receive requests from tourists looking for destination information.



Node.js uses an event-driven model to handle operations.

Node.js uses an event-driven, non-blocking I/O model in which operations such as file and network I/O can be handled asynchronously, allowing the application to continue processing other work while waiting for the operation to complete.

The following simple example illustrates that a delayed callback does not prevent subsequent synchronous statements from executing.

Simple Example

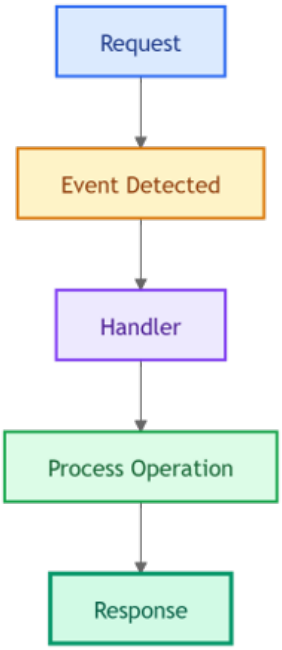
```
    console.log("Server started");
    setTimeout(
      function () {
        console.log(
          "Destination data processed"
        );
      },
      2000
    );
console.log("Server is ready");
```

The output begins with:

```
Server started
Server is ready
Destination data processed
```

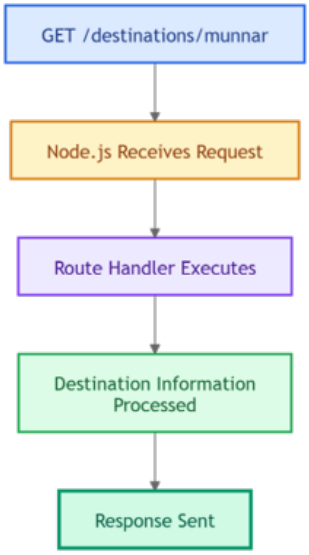
The delayed operation does not prevent the following statement from executing.

Event-Driven Flow



Kerala Explorer Example

When a tourist requests Munnar information:



2.1.3 npm and package.json

A Node.js project often requires external packages.

For example, Kerala Explorer can use Express.js to simplify server development.

The Node Package Manager (npm) is used to manage packages in Node.js projects.

A project can be initialized using:

npm init

or:

npm init -y

The command creates a package.json file.

package.json

The package.json file contains information about a Node.js project.

A simple example is:

```
{
  "name": "kerala-explorer-api",
  "version": "1.0.0",
  "description": "Backend API for Kerala Explorer",
  "main": "server.js",
  "scripts": {
    "start": "node server.js"
  }
}
```

The file can contain:

- project name;
- version;
- description;
- entry point;
- scripts; and
- dependencies.

Installing Express

Express can be installed using:

npm install express

The installed package is recorded in the project's dependencies.

Running the Application

If the project contains:

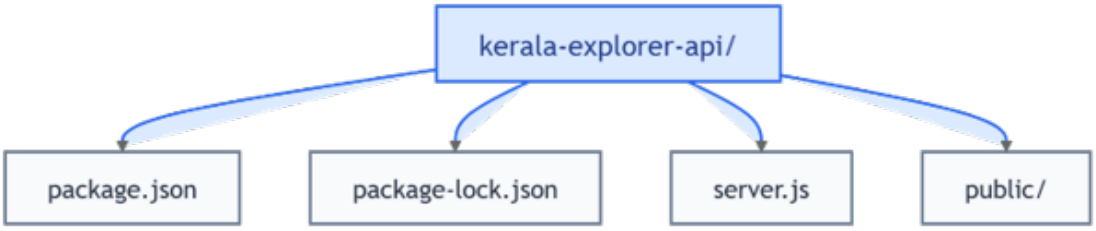
```
{
  "scripts": {
    "start": "node server.js"
  }
}
```

the application can be started using:

npm start

Kerala Explorer Project Structure

A simple backend project may look like:



2.1.4 Node.js Modules

Node.js applications can be organized into modules.

A module contains reusable functionality that can be used by another part of an application.

Node.js provides several built-in modules.

This unit focuses on:

- fs
- path
- http

These modules provide functionality without requiring separate installation.

2.1.5 File System Module – fs

The fs module provides functionality for working with files and directories.

It can be imported using:

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

For example, a file can be created using:

```
const fs = require("fs");
fs.writeFileSync(
   "destinations.txt",
   "Munnar\nWayanad\nAlappuzha"
);
```

The file contains:

```
Munnar
Wayanad
Alappuzha
```

A file can be read using:

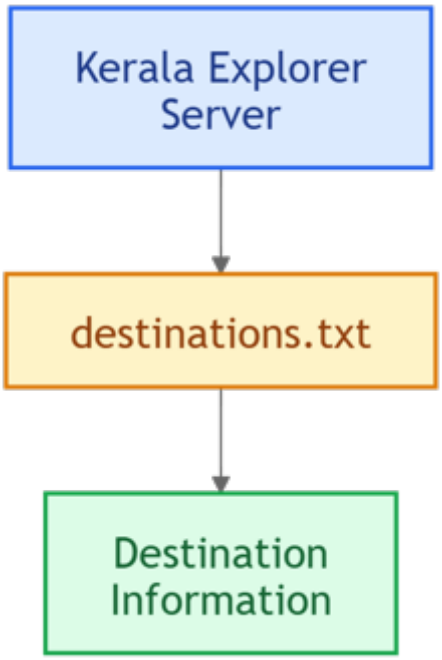
```
const data =
   fs.readFileSync(
      "destinations.txt",
      "utf8"
   );
console.log(data);
```

The Sync methods perform file operations synchronously and are used here for simplicity. In applications where non-blocking execution is important, asynchronous file-system methods can be used.

Kerala Explorer Application

The fs module can be used for file-based operations such as reading or writing local application files.

For example:



2.1.6 Path Module – path

File and directory paths can vary between operating systems.

Node.js provides the path module to work with file paths.

It can be imported using:

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

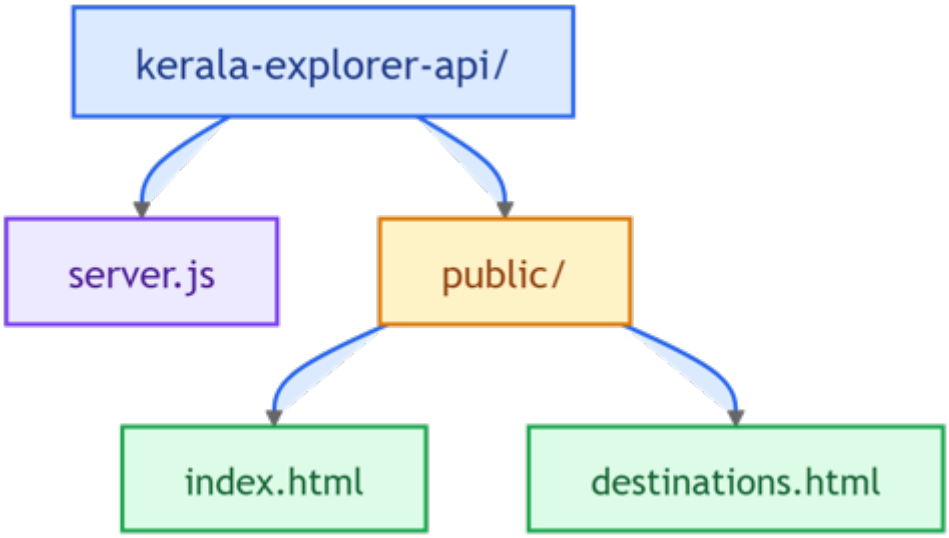
For example:

```
const filePath =
  path.join(
    __dirname,
    "public",
    "destinations.html"
  );
console.log(filePath);
```

The path.join() method combines path segments appropriately.

Kerala Explorer Example

Suppose the application contains:



The path to destinations.html can be created using:

```
const destinationPage =
    path.join(
        __dirname,
        "public",
        "destinations.html"
    );
```

2.1.7 HTTP Module

Node.js provides the built-in http module for creating HTTP servers.

It can be imported using:

```
const http = require("http");
A simple server can be created using:
const http = require("http");
const server =
    http.createServer(
        function (request, response) {
            response.writeHead(
```

```
            200,
            {
              "Content-Type":
                "text/plain"
            }
        );
        response.end(
          "Welcome to Kerala Explorer"
        );
      }
  );
server.listen(
  3000,
  function () {
    console.log(
      "Server running on port 3000"
    );
  }
);
```

When the server is running, a client can send a request to:

http://localhost:3000

The server responds with:

Welcome to Kerala Explorer

Request and Response

The server receives:

Request

and sends:

Response

The basic flow is:

Browser

|

| HTTP Request

↓

Node.js HTTP Server

|

| HTTP Response

↓

Browser

2.1.8 Express.js Framework

Introduction to Express.js

Although Node.js provides the http module, building a complete web application directly with the http module can require considerable code.

Express.js provides a framework that simplifies the development of web servers and APIs.

Express.js can be installed using:

```
npm install express
```

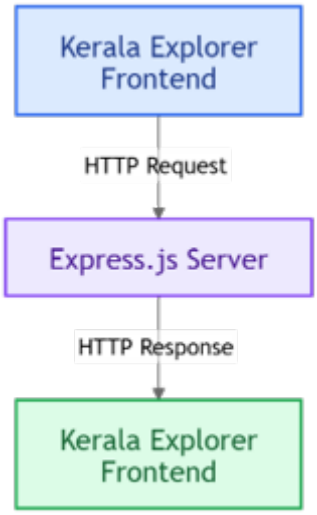
A basic Express application is:

```
const express = require("express");
const app = express();
app.listen(
   3000,
   function () {
      console.log(
         "Kerala Explorer server running."
      );
   }
);
```

The express() function creates an Express application.

Kerala Explorer Application

The application can act as the backend server for the tourism website.



2.1.9 Creating an Express Application

A basic Kerala Explorer server can be created as follows:

```
const express = require("express");
const app = express();
app.listen(
    3000,
    function () {
        console.log(
            "Kerala Explorer server is running on port 3000."
        );
    }
);
```

The server can be started using:

node server.js

The console displays:

Kerala Explorer server is running on port 3000.

Using a Route

A route can be added to respond to requests:

```
app.get(
    "/",
    function (request, response) {
        response.send(
            "Welcome to Kerala Explorer"
        );
    }
);
```

Now a GET request to / receives the response.

2.1.10 Express Routing

Understanding Routes

A route determines how an application responds to a particular HTTP method and URL path.

For example:

GET /destinations

can be used to retrieve destination information.

Another route can be:

GET /destinations/munnar

to retrieve information about Munnar.

Basic Route

```
app.get(
    "/destinations",
    function (request, response) {
        response.send(
            "Kerala tourist destinations"
        );
    }
);
```

The general structure is:

```
app.METHOD(
   PATH,
   HANDLER
);
```

For example:

```
app.get(
   "/about",
   function (request, response) {
      response.send(
         "About Kerala Explorer"
      );
   }
);
```

Kerala Explorer Routes

A tourism API may contain:

/destinations

/destinations/munnar

/destinations/wayanad

/destinations/alappuzha

2.1.11 GET Routing

The GET method is commonly used to retrieve data.

For example:

```
app.get(
   "/destinations",
   function (request, response) {
      response.json([
         {
            name: "Munnar",
```

```
            type: "Hill destination"
        },
        {
            name: "Wayanad",
            type: "Nature destination"
        },
        {
            name: "Alappuzha",
            type: "Backwater destination"
        }
    ]);
  }
);
```

The response is JSON data.

A request:

GET /destinations

can therefore return:

```
[
  {
    "name": "Munnar",
    "type": "Hill destination"
  },
  {
    "name": "Wayanad",
    "type": "Nature destination"
  },
  {
    "name": "Alappuzha",
    "type": "Backwater destination"
  }
]
```

2.1.12 POST Routing

The POST method is commonly used to create a new resource.

Suppose Kerala Explorer allows an administrator to add a new destination.

```
app.post(
    "/destinations",
    function (request, response) {
        response.send(
            "New destination created"
        );
    }
);
```

The client can send:

POST /destinations

The server processes the request and creates the resource.

Kerala Explorer Example

POST /destinations

```
Request data:
    {
        "name": "Kovalam",
        "type": "Coastal destination"
    }
```

The server can process this information.

2.1.13 PUT Routing

The PUT method can be used to update an existing resource.

For example:

```
    app.put(
        "/destinations/:id",
        function (request, response) {
```

```
            response.send(
              "Destination updated"
            );
        }
    );
```

The route:

PUT /destinations/1

can represent an update to destination 1.

Kerala Explorer Example

PUT /destinations/1

Request data:

```
    {
        "name": "Munnar",
        "type": "Hill destination",
        "popularFor": "Tea plantations"
    }
```

2.1.14 DELETE Routing

The DELETE method is commonly used to remove a resource.

For example:

```
app.delete(
    "/destinations/:id",
    function (request, response) {
        response.send(
            "Destination deleted"
        );
    }
);
```

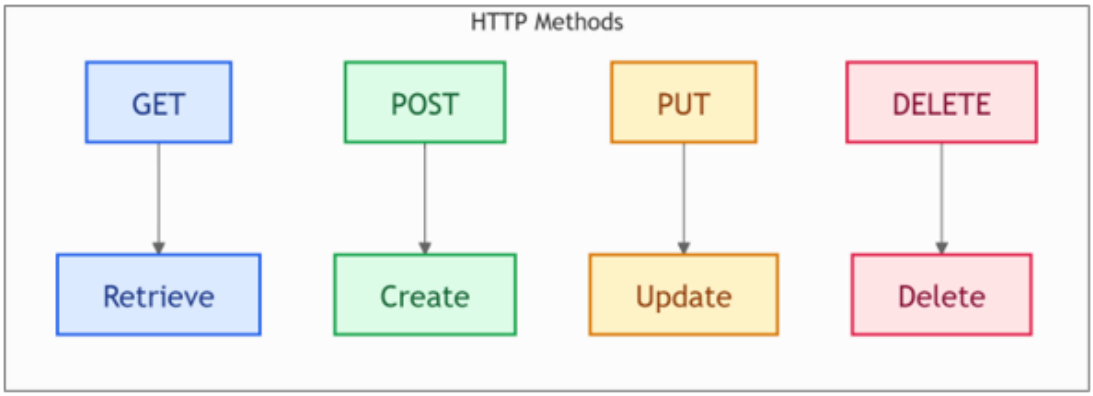
A request such as:

DELETE /destinations/1

can request removal of destination 1.

RESTful Resource Operations

The four HTTP methods can be summarized as:



Kerala Explorer Example

GET /destinations

POST /destinations

PUT /destinations/:id

DELETE /destinations/:id

These routes provide a basic RESTful structure.

2.1.15 Request and Response Objects

An Express route handler receives two important objects:

request

response

For example:

```
app.get(
  "/destinations",
  function (request, response) {
    console.log(request.method);
    response.send(
```

```
        "Destination information"
      );
    }
);
```

Request Object

The request object contains information about the incoming request.

For example:

request.method

provides the HTTP method.

The URL can be accessed through:

request.url

Response Object

The response object is used to send a response to the client.

For example:

```
response.send(
   "Welcome to Kerala Explorer"
);
JSON can be returned using:
response.json(
   {
      name: "Munnar",
      type: "Hill destination"
   }
);
```

Kerala Explorer Example

```
app.get(
   "/destination",
   function (request, response) {
```

```
    response.json(
        {
            name: "Munnar",
            type: "Hill destination"
        }
    );
}
);
```

2.1.16 Route Parameters

A route parameter allows part of the URL to represent a value.

For example:

```
app.get(
    "/destinations/:id",
    function (request, response) {
        const id =
            request.params.id;
        response.send(
            `Destination ID: ${id}`
        );
    }
);
```

If the client requests:

GET /destinations/3

the value of request.params.id is:

3

Kerala Explorer Example

```
app.get(
    "/destinations/:id",
    function (request, response) {
```

```
    const id =
        request.params.id;
    response.json(
        {
            message:
                "Requested destination",
            id: id
        }
    );
  }
);
```

This approach allows a single route to handle multiple destination IDs.

2.1.17 Serving Static Files

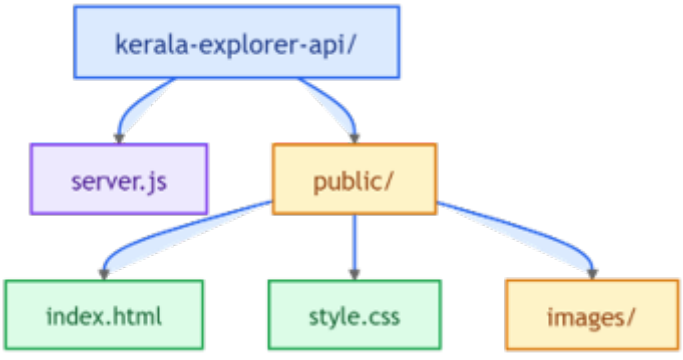
Serving Static Files

A tourism application may contain static resources such as:

- HTML files;
- CSS files;
- JavaScript files;
- images.

Express provides middleware for serving static files.

Suppose the project contains:



The public directory can be served using:

```
app.use(
    express.static("public")
);
```

The browser can then request:

```
/
```

or:

```
/style.css
```

and Express can serve the corresponding static resources.

Kerala Explorer Application

The same server can therefore provide both:

Static Frontend
+
REST API

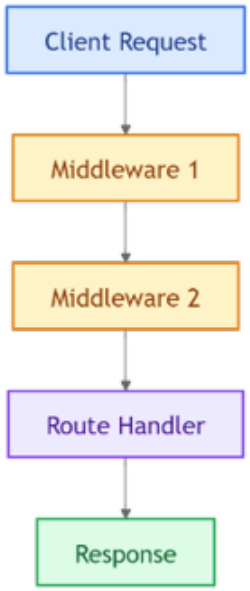
2.1.18 Middleware

Understanding Middleware

A request may need to pass through several processing steps before reaching the final route handler.

These processing functions are called middleware.

A simplified flow is:



Definition

Middleware is a function that has access to the request object, response object and the next middleware function in the request-response cycle.

A simple middleware function is:

```
app.use(
    function (request, response, next) {
        console.log(
            "Request received"
        );
        next();
    }
);
```

The next() function passes control to the next middleware or route handler.

Kerala Explorer Example

Middleware can be used to log requests:

```
app.use(
    function (request, response, next) {
        console.log(
            request.method,
            request.url
        );
        next();
    }
);
```

If a tourist requests:

GET /destinations

the console may display:

GET /destinations

2.1.19 Processing JSON Request Data

When a client sends JSON data to a server, the Express application needs to parse the request body.

Express provides:

```
express.json()
```

It can be enabled using:

```
app.use(
    express.json()
);
```

Consider a POST request containing:

```
{
    "name": "Kovalam",
    "type": "Coastal destination"
}
```

The route can access the data using:

```
app.post(
    "/destinations",
    function (request, response) {
        console.log(
            request.body
        );
        response.json(
            {
                message:
                    "Destination received",
                destination:
                    request.body
            }
        );
    }
);
```

Kerala Explorer Example

The client sends:

```
{
    "name": "Thekkady",
    "type": "Wildlife destination"
}
```

The server can access:

request.body.name

and:

request.body.type

2.1.20 REST API Design

Designing RESTful Routes

A RESTful API organizes application data as resources.

For Kerala Explorer, a resource can be:

destinations

A suitable RESTful design is:

Table 2.1.1 HTTP Methods

Operation	HTTP Method	Route
Get all destinations	GET	/destinations
Get one destination	GET	/destinations/:id
Create destination	POST	/destinations
Update destination	PUT	/destinations/:id
Delete destination	DELETE	/destinations/:id

The URL represents the resource, while the HTTP method represents the intended operation.

Avoiding Action-Based URLs

Instead of designing:

GET /getDestinations

POST /createDestination

POST /deleteDestination

a RESTful design uses:

GET /destinations

POST /destinations

DELETE /destinations/:id

This provides a consistent resource-oriented structure.

Kerala Explorer API

/destinations

can represent the collection of tourism destinations.

/destinations/1

can represent a specific destination.

2.1.21 HTTP Status Codes

An HTTP response contains a status code that indicates the result of a request.

Common status codes include:

200 OK

201 Created

400 Bad Request

404 Not Found

500 Internal Server Error

200 OK

Indicates that the request was successfully processed.

```
response.status(200).json(
    {
        message:
            "Destinations retrieved successfully"
    }
);
```

201 Created

Used when a new resource has been created.

```
response.status(201).json(
    {
        message:
            "Destination created successfully"
    }
);
```

400 Bad Request

Indicates that the request contains invalid information.

```
response.status(400).json(
    {
        error:
            "Destination name is required"
    }
);
```

404 Not Found

Used when the requested resource cannot be found.

```
response.status(404).json(
    {
        error:
            "Destination not found"
    }
);
```

500 Internal Server Error

Indicates an unexpected server-side problem.

```
response.status(500).json(
    {
        error:
            "Internal server error"
    }
);
```

Kerala Explorer Application

If a tourist requests:

GET /destinations/999

and destination 999 does not exist, the API can return:

```
{
   "error": "Destination not found"
}
```

with:

404 Not Found

2.1.22 Error-Handling Middleware

An Express application may encounter errors while processing requests.

Instead of handling errors separately in every route, Express provides a mechanism for centralized error handling.

An error-handling middleware function has four parameters:

```
function (
   error,
   request,
   response,
   next
) {
   // error handling
}
```

A simple example is:

```
app.use(
   function (
      error,
      request,
      response,
      next
```

```
    ) {
        console.error(error);
        response.status(500).json(
            {
                error:
                    "Something went wrong."
            }
        );
    }
);
```

Kerala Explorer Example

Suppose an unexpected error occurs while retrieving destination information.

The error-handling middleware can return:

```
{
    "error": "Unable to process the request."
}
```

instead of leaving the client without a meaningful response.

Important Placement

Error-handling middleware should be placed after the application's routes and other middleware that may generate errors.

A simplified structure is:

```
const express = require("express");
const app = express();
app.use(express.json());
app.get(
    "/destinations",
    function (request, response) {
        response.json(
            {
                message:
                    "Destination data"
```

```
            }
        );
    }
);
app.use(
    function (
        error,
        request,
        response,
        next
    ) {
        response.status(500).json(
            {
                error:
                    "Internal server error"
            }
        );
    }
);
```

app.listen(3000);

Think About It

Suppose Kerala Explorer has 1,000 tourist destinations.

Would storing all destination information directly inside server.js be a suitable long-term solution?

The next unit addresses this problem by introducing database integration with MongoDB and PostgreSQL.

Summarised Overview

Node.js is a JavaScript runtime environment that enables JavaScript to be executed outside the browser and uses an event-driven programming model. npm is used to manage Node.js packages, while package.json describes a Node.js project, including its dependencies and scripts. Node.js provides built-in modules such as fs, path, and http; the fs module supports file-system operations, the path module provides utilities for handling file and directory paths, and the http module can be used to create HTTP servers. Express.js is a framework for developing Node.js web applications and APIs, where routes handle client requests using HTTP methods such as GET for retrieving resources, POST for creating resources, PUT for updating resources, and DELETE for removing resources. Request and response objects support client-server communication, while route parameters can represent resource identifiers. Express can serve static files using express.static() and process requests using middleware during the request-response cycle. The express.json() middleware parses JSON request bodies. RESTful APIs organize application data as resources, while HTTP status codes communicate the result of requests and error-handling middleware provides centralized error processing. REST APIs enable frontend applications such as Kerala Explorer to communicate with backend services.

Assignments

Develop a Kerala Explorer RESTful Web Service

The Kerala Explorer tourism portal now requires a backend service to manage tourist destination information.

Develop a Node.js and Express.js application that provides RESTful services for Kerala tourist destinations.

The application should:

- use Node.js;
- use npm and package.json;
- use Express.js;

- implement RESTful routes;
- support GET, POST, PUT and DELETE operations;
- process JSON request data;
- use route parameters;
- return appropriate HTTP status codes;
- implement middleware;
- implement error-handling middleware; and
- return destination information in JSON format.

Tasks

1. Create a Node.js project using npm.
2. Configure package.json.
3. Install Express.js.
4. Create an Express server.
5. Create a /destinations resource.
6. Implement a GET route to retrieve all destinations.
7. Implement a GET route to retrieve a destination by ID.
8. Implement a POST route to add a new destination.
9. Implement a PUT route to update a destination.
10. Implement a DELETE route to remove a destination.
11. Use express.json() to process JSON request bodies.
12. Add middleware to log incoming requests.
13. Implement suitable HTTP status codes.
14. Add error-handling middleware.
15. Test the API using a REST client such as Postman or Thunder Client.

Suggested REST API

GET /destinations

GET /destinations/:id

POST /destinations

PUT /destinations/:id

DELETE /destinations/:id

Suggested submission

Submit:

- package.json;
- server.js;
- screenshots of API testing;
- sample request and response data; and
- a brief explanation of the RESTful design.

Reference

1. Zammetti, F. (2020). *Modern Full-Stack Development: Using TypeScript, React, Node. js, Webpack, and Docker*. Apress.

2. Northwood, C. (2018). *The full stack developer: your essential guide to the everyday skills expected of a modern full stack web developer*. Apress.

3. Subramanian, V. (2019). *Pro Mern Stack: Full Stack Web App Development with Mongo, Express, React and Node*. Apress.

4. Dayley, B., Dayley, B., & Dayley, C. (2017). *Node. js, MongoDB and Angular Web Development: The definitive guide to using the MEAN stack to build web applications*. Addison-Wesley Professional.

Suggested Reading

1. Hoque, S. (2020). *Full-Stack React Projects: Learn MERN stack development by building modern web apps using MongoDB, Express, React, and Node. js*. Packt Publishing Ltd.

2. Koroliova, E. W. I. (2018). *MERN quick start guide: build web applications with MongoDB, Express. js, React, and Node*. Packt Publishing Ltd.

3. Bugl, D. (2024). *Modern full-stack React projects: Build, maintain, and deploy modern web apps using MongoDB, Express, React, and Node. js.* Packt Publishing Ltd.

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

2 UNIT Database Integration and REST API Development

Learning Outcomes

After completion of this unit, the learner will be able to:

- explain the need for database integration and persistent data storage in web applications
- differentiate between document-oriented and relational databases using MongoDB and PostgreSQL
- explain MongoDB collections, documents, schemas, and models
- apply Mongoose to perform CRUD operations on MongoDB data
- apply SQL and the pg library to access and manage data in PostgreSQL
- develop a database-backed REST API using Express.js with MongoDB or PostgreSQL

Background

In Block II – Unit 1, Kerala Explorer was extended from a frontend application into a basic backend service using Node.js and Express.js.

The application could provide routes such as:

GET /destinations
GET /destinations/:id
POST /destinations
PUT /destinations/:id
DELETE /destinations/:id

However, the destination information in the previous unit was stored in an in-memory array.

For example:

```
const destinations = [
  {
    id: 1,
    name: "Munnar",
    type: "Hill destination"
  },
  {
    id: 2,
    name: "Wayanad",
    type: "Nature destination"
  }
];
```

This approach is suitable for demonstrating REST API concepts, but it is not suitable for a real tourism application.

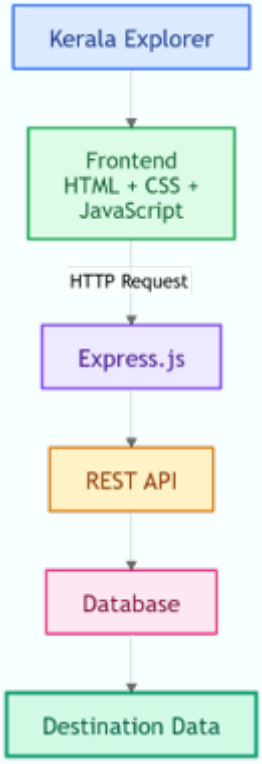
Consider a Kerala tourism portal containing information about:

1. Munnar;
2. Wayanad;
3. Alappuzha;
4. Thekkady;
5. Kovalam;
6. Varkala;
7. Kumarakom;
8. Fort Kochi; and
9. many other destinations.

If the server is restarted, information stored only in memory can be lost.

A database provides persistent storage for application data.

The architecture can now be extended as follows:



Two important types of databases introduced in this unit are MongoDB and PostgreSQL

MongoDB is a document-oriented database, whereas PostgreSQL is a relational database.

The unit therefore introduces both approaches and demonstrates how they can be connected to a Node.js and Express.js application.

Keywords

Database, Database Integration, MongoDB, Collection, Mongoose, ODM, Schema, Model, CRUD, PostgreSQL, Relational Database, Query, REST API, Persistence

Discussion

2.2.1 Need for Database Integration

A web application often needs to store information permanently.

For Kerala Explorer, the application may need to store:

Destination Name, Destination Type, District, Description, Best Time to Visit, Popular Attractions

If this information is stored in JavaScript variables:

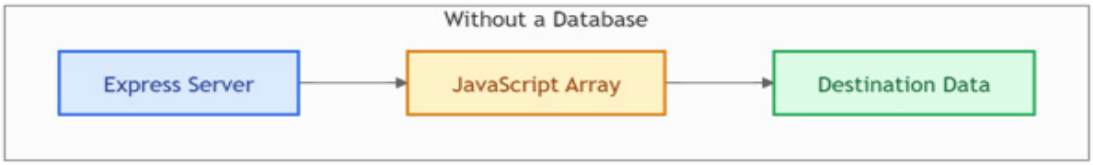
```
const destinations = [
  {
    name: "Munnar",
    type: "Hill destination"
  }
];
```

the data exists only while the server is running.

This is called in-memory storage.

A database provides persistent storage.





The first approach allows data to remain available even after the server is restarted.

Kerala Explorer Example

Suppose an administrator adds:

Name: Varkala

Type: Coastal destination

District: Thiruvananthapuram

The information should be stored permanently so that it can later be retrieved through:

GET /destinations

2.2.2 Types of Databases

Databases can be broadly categorized into different types.

This unit introduces two database approaches: the document-oriented database MongoDB and the relational database PostgreSQL.

MongoDB stores data as documents, whereas PostgreSQL stores data using tables and relationships.

Simple Comparison

Table 2.2.1 Comparison of MongoDB and PostgresSQL

MongoDB	PostgreSQL
Document-oriented	Relational
Documents	Rows
Collections	Tables
Flexible document structure	Defined table structure
Uses MongoDB query operations	Uses SQL

Both can be used to build database-backed web applications.

Kerala Explorer

The same destination information can be represented differently.

MongoDB:

```
{
  "name": "Munnar",
  "type": "Hill destination",
  "district": "Idukki"
}
```

PostgreSQL:

destinations

id	name	type	district
1	Munnar	Hill	Idukki

2.2.3 Introduction to MongoDB

MongoDB is a document-oriented database system.

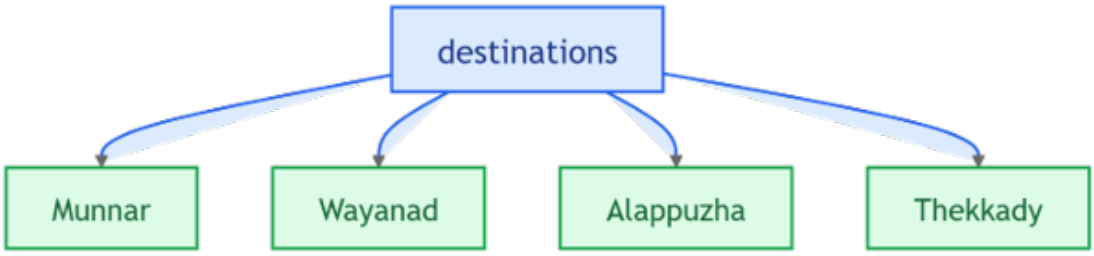
Instead of storing information in rows and columns, MongoDB stores data as documents.

A document is represented using a JSON-like structure.

For example:

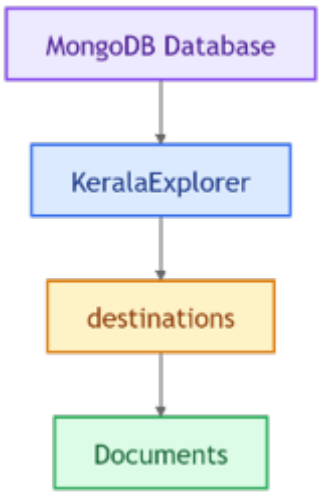
```
{
   "name": "Munnar",
   "type": "Hill destination",
   "district": "Idukki"
}
```

A collection can contain multiple documents.



Kerala Explorer Example

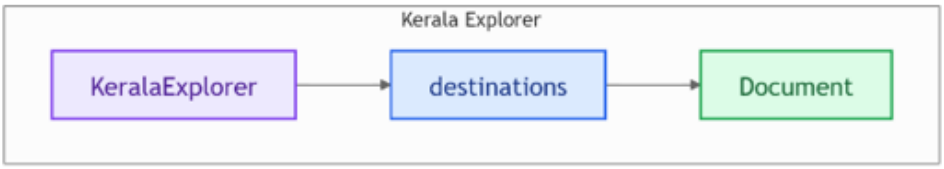
A destinations collection can store information about Kerala tourist destinations.

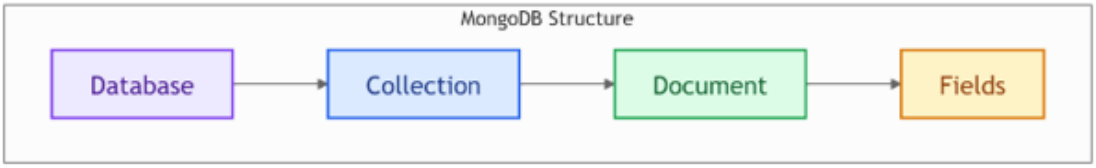


2.2.4 MongoDB Documents and Collections

MongoDB organizes data using databases, collections, and documents.

The structure can be represented as:





A destination document may be:

```
{
   "name": "Wayanad",
   "type": "Nature destination",
   "district": "Wayanad",
   "bestTime": "October to May"
}
```

Another document can be:

```
{
   "name": "Alappuzha",
   "type": "Backwater destination",
   "district": "Alappuzha",
   "bestTime": "November to February"
}
```

Unlike a relational table, individual MongoDB documents can contain different fields.

2.2.5 MongoDB and JSON-Like Documents

MongoDB stores documents internally in BSON (Binary JSON), while the documents are commonly represented in a JSON-like format when working with applications and tools.

For example:

```
{
  "name": "Thekkady",
  "type": "Wildlife destination",
  "district": "Idukki",
  "popularFor": [
    "Periyar Wildlife Sanctuary",
    "Boating",
    "Wildlife"
  ]
}
```

The document can contain:

- strings;
- numbers;
- Boolean values;
- arrays; and
- nested objects.

This makes document-oriented storage suitable for data that may have varying structures.

Kerala Explorer Example

A destination can contain a list of attractions:

```
{
  "name": "Munnar",
  "type": "Hill destination",
  "attractions": [
    "Tea plantations",
    "Mattupetty Dam",
    "Eravikulam National Park"
  ]
}
```

2.2.6 Introduction to Mongoose

Node.js applications need a way to work conveniently with MongoDB.

Mongoose is an Object Data Modeling (ODM) library for MongoDB and Node.js applications.

It provides features such as:

- schemas;
- models;
- validation;
- database interaction; and
- CRUD operations.

A Mongoose package can be installed using:

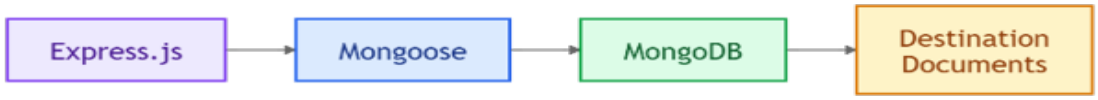
```
npm install mongoose
```

The module can be imported using:

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

Kerala Explorer

Mongoose can act as a bridge between the Express application and MongoDB.



2.2.7 Connecting Node.js to MongoDB

Mongoose provides a connection mechanism for MongoDB.

A connection can be created using:

```
const mongoose =
    require("mongoose");
mongoose.connect(
    "mongodb://127.0.0.1:27017/keralaExplorer"
);
```

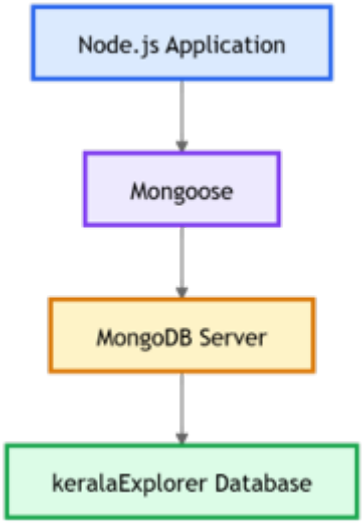
The database name is:

keralaExplorer

A more complete example is:

```
const mongoose =
    require("mongoose");
mongoose.connect(
    "mongodb://127.0.0.1:27017/keralaExplorer"
)
.then(
    function () {
        console.log(
            "MongoDB connected"
        );
    }
)
.catch(
    function (error) {
        console.error(
            "MongoDB connection failed",
            error
        );
    }
  }
```

Connection Flow



2.2.8 Defining a Mongoose Schema

A schema defines the structure of documents managed by a Mongoose model.

For Kerala Explorer:

```
const mongoose =
    require("mongoose");
const destinationSchema =
    new mongoose.Schema({
        name: {
            type: String,
            required: true
        },
        type: {
            type: String,
            required: true
        },
        district: {
            type: String,
            required: true
        }
    });
```

The schema describes fields such as:

name

type

district

Adding More Fields

A destination can contain:

```
const destinationSchema =
  new mongoose.Schema({
    name: String,
    type: String,
    district: String,
    description: String,
    bestTime: String
  });
```

2.2.9 Creating a Mongoose Model

A model provides an interface for interacting with documents in a MongoDB collection.

For example:

```
const Destination =
  mongoose.model(
    "Destination",
    destinationSchema
  );
```

The model can then be used to create and retrieve destinations.

For example:

```
const destination =
  new Destination({
    name: "Munnar",
    type:
```

```
        "Hill destination",
    district:
        "Idukki"
});
```

The document can be saved using:

```
destination.save();
```

Structure



2.2.10 MongoDB CRUD Operations

2.2.10.1 Create Operation

The Create operation adds a new document to MongoDB.

```
Using Mongoose:
  const destination =
    await Destination.create({
      name: "Varkala",
      type:
        "Coastal destination",
      district:
        "Thiruvananthapuram"
    });
```

The document is stored in the database.

Kerala Explorer Example

```
POST /destinations
Request:
    {
      "name": "Varkala",
```

```
    "type": "Coastal destination",
    "district": "Thiruvananthapuram"
}
```

The Express route can create the document.

```
app.post(
    "/destinations",
    async function (
        request,
        response
    ) {
        const destination =
            await Destination.create(
                request.body
            );
        response.status(201).json(
            destination
        );
    }
);
```

2.2.10.2 Read Operation

The Read operation retrieves data from MongoDB.

All destinations can be retrieved using:

```
const destinations =
    await Destination.find();
```

An Express route can return the results:

```
app.get(
    "/destinations",
    async function (
```

```
        request,
        response
    ) {
        const destinations =
            await Destination.find();
        response.status(200).json(
            destinations
        );

    }
);
```

A specific destination can be retrieved using:

```
const destination =
    await Destination.findById(
        request.params.id
    );
```

2.2.10.3 Update Operation

The Update operation modifies an existing document.

For example:

```
const destination =
    await Destination.findByIdAndUpdate(
        request.params.id,
        request.body,
        {
            new: true
        }
    );
```

An Express route can be:

```
app.put(
```

```
    "/destinations/:id",
    async function (
        request,
        response
    ) {
        const destination =
            await Destination.findByIdAndUpdate(
                request.params.id,
                request.body,
                {
                    new: true
                }
            );
        response.status(200).json(
            destination
        );
    }
);
```

Kerala Explorer Example

A destination can be updated using:

```
PUT /destinations/65abc123
with:
    {
        "type": "Hill and tea plantation destination"
    }
```

2.2.10.4 Delete Operation

The Delete operation removes a document.

For example:

```
    const destination =
      await Destination.findByIdAndDelete(
        request.params.id
      );
An Express route can be:
    app.delete(
      "/destinations/:id",
      async function (
        request,
        response
      ) {
        await Destination.findByIdAndDelete(
          request.params.id
        );
        response.status(200).json(
          {
            message:
              "Destination deleted"
          }
        );
      }
    );
```

2.2.11 Introduction to PostgreSQL

PostgreSQL is a relational database management system.

A relational database organizes information using tables.

For Kerala Explorer, destination information can be stored in a table:

destinations

with columns such as:

id

name

type

district

best_time

Example:

id	name	type	district
1	Munnar	Hill destination	Idukki
2	Wayanad	Nature destination	Wayanad
3	Alappuzha	Backwater destination	Alappuzha

Relational Structure



2.2.11.1 Tables, Rows and Columns

A relational table consists of rows and columns.

Consider:

destinations

id	name	type	district
1	Munnar	Hill destination	Idukki
2	Wayanad	Nature destination	Wayanad
3	Thekkady	Wildlife destination	Idukki

Each row represents a destination.

Each column represents an attribute.

For example:

name

stores the destination name.

district

stores the district.

2.2.11.2 SQL and SELECT

SQL (Structured Query Language) is used to interact with relational databases.

To retrieve all destinations:

```
SELECT *
FROM destinations;
```

To retrieve selected columns:

```
SELECT
    name,
    district
FROM destinations;
```

To retrieve destinations from Idukki:

```
SELECT *
FROM destinations
WHERE district = 'Idukki';
```

Kerala Explorer Example

The query:

```
SELECT *
FROM destinations;
```

returns all destination records.

2.2.11.3 INSERT

The INSERT statement adds a new row.

For example:

```
INSERT INTO destinations
    (name, type, district)
VALUES
        (
            'Varkala',
            'Coastal destination',
```

```
        'Thiruvananthapuram'
    );
```

This adds Varkala to the destination table.

Kerala Explorer REST API

A POST request:

POST /destinations

can result in an SQL INSERT operation.



2.2.11.4 UPDATE

The UPDATE statement modifies existing rows.

For example:

```
UPDATE destinations
SET
    type = 'Hill and tea plantation destination'
WHERE
    name = 'Munnar';
```

The WHERE clause identifies the record to update.

Kerala Explorer

A PUT request:

PUT /destinations/1

can result in an SQL UPDATE.

2.2.11.5 DELETE

The DELETE statement removes records.

For example:

```
DELETE FROM destinations
WHERE name = 'Varkala';
```

The WHERE clause limits the deletion to the specified record.

Kerala Explorer

A DELETE request:

DELETE /destinations/4

can result in an SQL DELETE operation.

2.2.12 PostgreSQL with Node.js

2.2.12.1 Using the pg Library

Node.js applications can communicate with PostgreSQL using the pg library.

It can be installed using:

```
npm install pg
```

The library can be imported using:

```
const {
    Pool
} = require("pg");
```

A connection pool can be created using:

```
const pool =
    new Pool({
        user: "postgres",
        host: "localhost",
        database:
            "kerala_explorer",
        password:
            "your_password",
        port: 5432
    });
```

Note

The password shown above is only a placeholder. In a real application, database credentials should not be hard-coded in source code.

2.2.12.2 Connecting to PostgreSQL

A connection can be tested using:

```
pool.query(
    "SELECT NOW()",
    function (
        error,
        result
    ) {
        if (error) {
            console.error(
                error
            );
            return;
        }
        console.log(
            result.rows
        );
    }
);
```

The database connection flow is:



Kerala Explorer Example

The server can request:

SELECT *

FROM destinations;

PostgreSQL returns the rows.

Express then converts the result into a JSON response.

2.2.12.3 Executing Queries from Node.js

A query can be executed using:

```
const result =
  await pool.query(
    "SELECT * FROM destinations"
  );
console.log(
  result.rows
);
```

The returned rows can be sent through Express:

```
app.get(
  "/destinations",
  async function (
    request,
    response
  ) {
    const result =
      await pool.query(
        "SELECT * FROM destinations"
      );
    response.status(200).json(
      result.rows
    );
  }
);
```

Kerala Explorer API

A request:

GET /destinations

follows:



Database-Backed REST API

2.2.12.4 Connecting Express with MongoDB

A database-backed REST API combines the concepts from Units 1 and 2.

The architecture becomes:



For example:

GET /destinations

can retrieve documents from MongoDB.

```
app.get(
   "/destinations",
   async function (
      request,
      response
   ) {
      const destinations =
         await Destination.find();
      response.json(
         destinations
      );
   }
);
```

2.2.12.5 Connecting Express with PostgreSQL

The same REST API idea can be implemented using PostgreSQL.



For example:

```
app.get(
    "/destinations",
    async function (
        request,
        response
    ) {
        const result =
            await pool.query(
                "SELECT * FROM destinations"
            );
        response.json(
            result.rows
        );
    }
);
```

The frontend does not need to know whether MongoDB or PostgreSQL is being used.

It communicates with the REST API.

2.2.13 MongoDB and PostgreSQL: Comparison

MongoDB and PostgreSQL can both support Kerala Explorer, but their data models are different.

Table 2.2.2 Comparison between MongoDB and PostgreSQL

Feature	MongoDB	PostgreSQL
Database type	Document-oriented	Relational
Main structure	Collections and documents	Tables and rows
Query approach	MongoDB operations	SQL
Node.js library	Mongoose	pg
Structure	Flexible document structure	Defined table structure
Example	Destination document	Destination row

MongoDB Representation

```
{
  "name": "Munnar",
  "type": "Hill destination",
  "district": "Idukki"
}
```

PostgreSQL Representation

id	name	type	district
1	Munnar	Hill destination	Idukki

Important Observation

The frontend can remain unchanged:



The database implementation can change behind the API.

2.2.14 Building a To-Do REST API with Express and MongoDB

A To-Do REST API can be developed using Express.js, MongoDB, and Mongoose. The API allows users to create, view, update, and delete to-do tasks.

2.2.14.1 To-Do Model

Create a file named Todo.js inside the models folder.

```
const mongoose = require("mongoose");
const todoSchema = new mongoose.Schema({
  title: {
    type: String,
    required: true
  },
  completed: {
    type: Boolean,
```

```
        default: false
    }
});
module.exports =
  mongoose.model("Todo", todoSchema);
```

2.2.14.2 REST API

```
const express = require("express");
const mongoose = require("mongoose");
const Todo = require("./models/Todo");
const app = express();
// Middleware to read JSON data from requests
app.use(express.json());
// Connect to the MongoDB database
mongoose.connect(
  "mongodb://localhost:27017/todo_app"
);
// GET: Retrieve all to-do tasks
app.get("/todos", async function (request, response) {
  const todos = await Todo.find();
  response.json(todos);
});
// POST: Create a new to-do task
app.post("/todos", async function (request, response) {
  const todo = await Todo.create({
    title: request.body.title,
    completed: request.body.completed
  });
  response.status(201).json(todo);
```

```
});
// PUT: Update an existing to-do task
app.put("/todos/:id", async function (request, response) {
    const todo =
        await Todo.findByIdAndUpdate(
            request.params.id,
            request.body,
            {
                new: true,
                runValidators: true
            }
        );
    if (!todo) {
        return response.status(404).json({
            error: "To-do task not found"
        });
    }
    response.json(todo);
});
// DELETE: Delete a to-do task
app.delete("/todos/:id", async function (request, response) {
    const todo =
        await Todo.findByIdAndDelete(
            request.params.id
        );
    if (!todo) {
        return response.status(404).json({
            error: "To-do task not found"
```

```
        });
    }
    response.json({
        message: "To-do task deleted successfully"
    });
});
// Start the Express server
app.listen(3000, function () {
    console.log(
        "To-Do API running on port 3000"
    );
});
```

2.2.14.3 Testing the To-Do REST API Using Postman

Postman is a REST client used to send HTTP requests to an API and examine the responses. The To-Do API can be tested by sending requests to http://localhost:3000/todos.

Step 1: Start the API server

Run the following command in the terminal:

node server.js

The server starts on port 3000.

Step 2: Create a To-Do Task

In Postman, select POST and enter:

http://localhost:3000/todos

Select Body → raw → JSON and enter:

```
{
    "title": "Complete Web Technology Assignment",
    "completed": false
}
```

Click Send. The newly created task is returned in the response.

Step 3: Retrieve To-Do Tasks

Select GET and enter:

http://localhost:3000/todos

Click Send to retrieve all stored tasks.

Step 4: Update a To-Do Task

Copy the _id of an existing task from the GET response.

Select PUT and enter:

http://localhost:3000/todos/<id>

For example:

http://localhost:3000/todos/65abc123...

Select Body → raw → JSON and enter:

```
{
  "title": "Complete Web Technology Assignment",
  "completed": true
}
```

Click Send to update the task.

Step 5: Delete a To-Do Task

Select DELETE and enter:

http://localhost:3000/todos/<id>

Click Send. The selected task is deleted from the database.



Summarised Overview

Database integration enables web applications to store and manage data persistently. MongoDB is a document-oriented database that stores data as documents organized into collections, while PostgreSQL is a relational database that stores data in tables consisting of rows and columns. Mongoose provides an ODM-based approach for defining schemas and models and performing CRUD operations with MongoDB.

PostgreSQL can be accessed from Node.js using the pg library and SQL operations such as SELECT, INSERT, UPDATE, and DELETE. Database-backed REST APIs allow Express.js applications to create, retrieve, update, and delete persistent application data. MongoDB or PostgreSQL can be integrated with Express.js to develop applications such as to-do or note-taking systems, and REST clients such as Postman or Thunder Client can be used to test the API endpoints.



Assignments

Suppose an administrator adds 500 Kerala tourist destinations through the REST API.

If the server is restarted, should all the destinations disappear?

Why is database storage necessary for a real-world tourism application?

Develop a Database-Backed Kerala Explorer REST API

The Kerala Explorer application developed in Unit 1 currently stores destination information in an in-memory array.

Extend the application by integrating a database.

Develop a RESTful API that stores and manages Kerala tourist destinations using MongoDB with Mongoose or PostgreSQL with the pg library.

The application should:

- use Node.js and Express.js;
- connect to MongoDB or PostgreSQL;
- store Kerala tourist destination information;
- implement CRUD operations;
- provide RESTful routes;
- process JSON request data;
- return appropriate HTTP status codes;
- handle database errors; and
- return destination information as JSON.

Suggested Data

Each destination may contain:

Name, type, district, description, bestTime

Suggested REST API

GET /destinations

GET /destinations/:id

POST /destinations

PUT /destinations/:id

DELETE /destinations/:id

Tasks

1. Create the Node.js project.
2. Install Express and the required database package.
3. Configure the database connection.
4. Create the destination schema/model or database table.
5. Insert sample Kerala tourist destinations.
6. Implement GET to retrieve all destinations.
7. Implement GET to retrieve one destination.
8. Implement POST to add a destination.
9. Implement PUT to update a destination.
10. Implement DELETE to remove a destination.
11. Handle records that do not exist.
12. Return appropriate HTTP status codes.
13. Handle database errors.
14. Test all endpoints using Postman or Thunder Client.
15. Document the API requests and responses.

Suggested Submission

Submit:

- package.json;
- server.js;
- database schema/model;
- sample database records;
- API testing screenshots;
- sample requests and responses; and
- a brief explanation of the database design.

Reference

1. Zammetti, F. (2020). *Modern Full-Stack Development: Using TypeScript, React, Node. js, Webpack, and Docker.* Apress.

2. Northwood, C. (2018). *The full stack developer: your essential guide to the everyday skills expected of a modern full stack web developer*. Apress.

3. Subramanian, V. (2019). *Pro Mern Stack: Full Stack Web App Development with Mongo, Express, React and Node*. Apress.

4. Dayley, B., Dayley, B., & Dayley, C. (2017). *Node. js, MongoDB and Angular Web Development: The definitive guide to using the MEAN stack to build web applications*. Addison-Wesley Professional.

Suggested Reading

1. Hoque, S. (2020). *Full-Stack React Projects: Learn MERN stack development by building modern web apps using MongoDB, Express, React, and Node. js*. Packt Publishing Ltd.

2. Koroliova, E. W. I. (2018). *MERN quick start guide: build web applications with MongoDB, Express. js, React, and Node*. Packt Publishing Ltd.

3. Bugl, D. (2024). *Modern full-stack React projects: Build, maintain, and deploy modern web apps using MongoDB, Express, React, and Node. js.* Packt Publishing Ltd.

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

3 UNIT React Fundamentals and Component-Based Development

Learning Outcomes

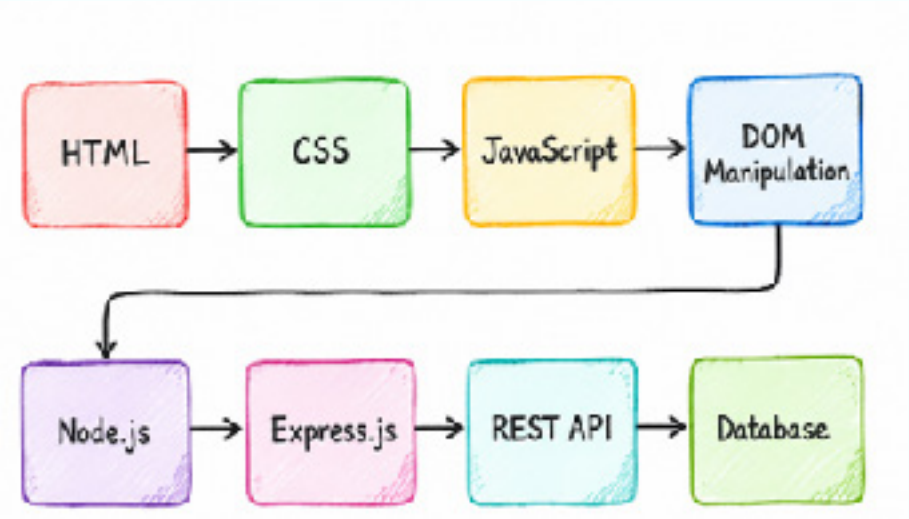
After completion of this unit, the learner will be able to:

- explain the role of React in developing user interfaces and describe the role of the Virtual DOM
- set up React applications using tools such as Create React App or Vite and apply JSX syntax to create user interface elements
- develop reusable functional components and pass data between components using props
- manage component state using useState and apply conditional rendering, lists, and keys to display dynamic content
- handle user interactions using React synthetic events and develop controlled form components for handling user input
- apply lifting state up to share and manage state between components

Background

In the previous units, Kerala Explorer was extended from a static webpage into a complete backend-enabled application.

The development so far can be viewed as:



The frontend developed in Block I works, but as the application becomes larger, managing the user interface directly with HTML and DOM manipulation can become difficult.

Consider the Kerala Explorer application.

The page may contain:

1. a navigation bar;
2. a destination list;
3. destination cards;
4. a destination search box;
5. a destination details section;
6. a tourist enquiry form;
7. buttons;
8. messages; and
9. other interactive elements.

If all these elements are maintained in one large HTML and JavaScript file, modifying one part of the interface can become difficult.

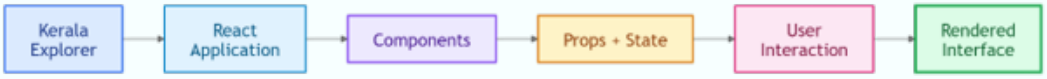
For example, the destination card may need to be reused in several places.

Instead of writing the complete markup separately for every destination, a reusable component can represent the common structure.

This is the idea behind component-based development.

React provides a way to build user interfaces using reusable components.

The Kerala Explorer frontend can now be represented as:



The next unit will connect this React frontend to the Express backend and database. Therefore, API fetching and React Router are intentionally kept for Unit 4.

Keywords

React, Component, JSX, Virtual DOM, Functional Component, Props, State, useState, Conditional Rendering, Synthetic Event, Controlled Component, Lifting State Up

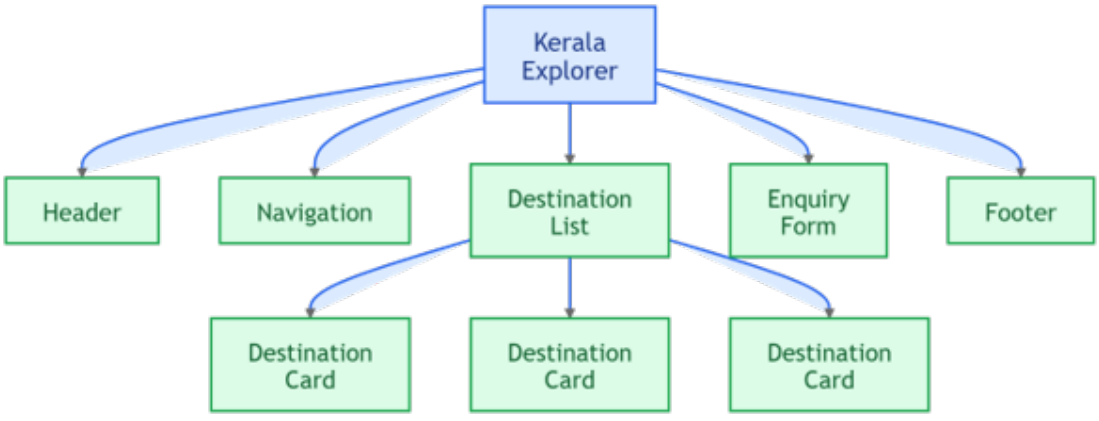
Discussion

2.3.1 Understanding React

React is a JavaScript library for building user interfaces.

React allows a user interface to be divided into reusable components.

For example, Kerala Explorer can be divided into:



Each component can represent a specific part of the interface.

A simple React component is:

```
function Welcome() {
    return (
        <h1>
            Welcome to Kerala Explorer
        </h1>
    );
}
```

The component returns the user interface that should be displayed.

React and Traditional JavaScript

In traditional JavaScript, a developer may directly select and modify DOM elements.

For example:

```
document.getElementById(
    "destinationName"
).textContent = "Munnar";
```

In React, the interface is described through components and data.

```
function Destination() {
    const name = "Munnar";
    return (
        <h2>{name}</h2>
    );
}
```

React manages the process of updating the displayed interface.

2.3.2 Setting up a React Application

A React application can be created using tools such as Create React App or Vite. Vite is a modern build tool commonly used for new React applications. Create React App is an older project setup tool and may be encountered in existing React projects.

For example, a Vite project can be created using:

npm create vite@latest

The setup process asks for project information such as:

- Project name
- Framework
- Variant

For a React application, select:

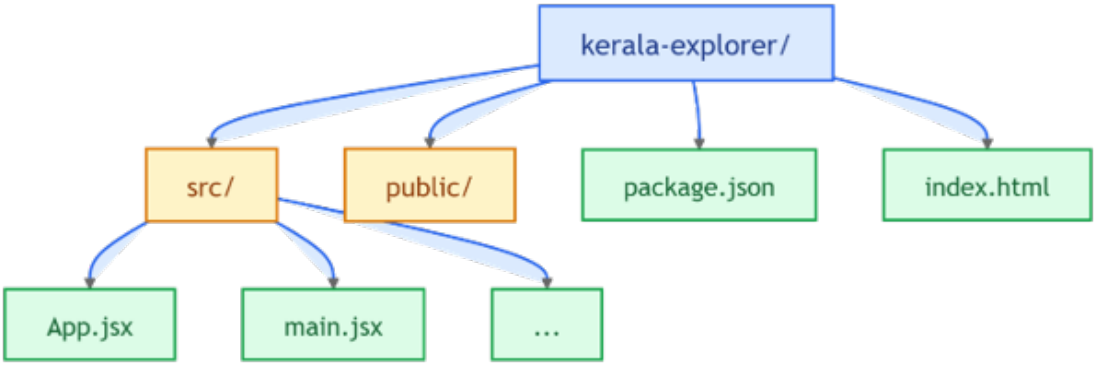
Framework: React

A project can then be installed and started using:

npm install

npm run dev

A typical project structure is:



Kerala Explorer

The React application can become the new frontend for the existing Kerala Explorer backend.

The backend integration will be introduced in Unit 4.

2.3.3 JSX Syntax

React commonly uses JSX to describe the user interface.

JSX allows HTML-like syntax to be written inside JavaScript.

For example:

```
function Destination() {
    return (
        <div>
            <h2>Munnar</h2>
            <p>
                Explore the scenic hills
                of Munnar.
            </p>
        </div>
    );
}
```

JSX looks similar to HTML, but it is written as part of JavaScript code.

JavaScript Expressions in JSX

JavaScript expressions can be inserted using curly braces.

```
function Destination() {
    const name = "Munnar";
    return (
        <h2>
            {name}
        </h2>
    );
}
```

The value of name is displayed inside the heading.

JSX Attributes

JSX can contain attributes:

```
<img
   src="munnar.jpg"
   alt="Landscape of Munnar"
/>
Some attributes use JSX-specific names.
For example:
<div className="destination-card">
instead of:
<div class="destination-card">
```

2.3.4 Rendering Elements

React renders components and their returned JSX into the webpage.

A simple component is:

```
function Welcome() {
   return (
      <h1>
         Explore Kerala
      </h1>
   );
}
```

The component can be used inside another component:

```
function App() {
   return (
      <Welcome />
   );
}
```

The App component renders the Welcome component.

Rendering a Value

```
function Destination() {
    const destination =
        "Munnar";
    return (
        <h2>
            Explore {destination}
        </h2>
    );
}
```

The rendered result is conceptually:

Explore Munnar

2.3.5 Understanding the Virtual DOM

The browser represents the structure of a webpage using the DOM (Document Object Model). When the interface changes, the browser may need to update parts of the DOM. React uses a virtual DOM representation to help determine what needs to change in the rendered interface. React compares the changes and updates the relevant part of the rendered interface.

A simplified view is:



Consider:

<h2>Munnar</h2>

If the destination changes to:

<h2>Wayanad</h2>

React determines the relevant change and updates the rendered interface.

Important Idea

The virtual DOM is an internal representation used by React to help manage UI updates. It should not be considered a replacement for the browser's actual DOM.

Kerala Explorer

Suppose the selected destination changes from Munnar to Wayanad. React can update the relevant part of the interface instead of requiring the developer to manually manipulate every DOM element.

Components

2.3.6 Functional Components

A functional component is a JavaScript function that returns JSX.

For example:

```
function DestinationCard() {
  return (
    <article>
      <h2>Munnar</h2>
      <p>
        A scenic hill destination.
      </p>
    </article>
  );
}
```

The component can be used as:

```
<DestinationCard />
```

Creating Multiple Components

Kerala Explorer can be divided into:

```
function Header() {
  return (
    <header>
      <h1>
        Kerala Explorer
      </h1>
    </header>
```

```
  );
}
function Footer() {
  return (
    <footer>
      <p>
        Kerala Explorer
      </p>
    </footer>
  );
}
```

The main component can combine them:

```
function App() {
  return (
    <>
      <Header />
      <main>
        <h2>
          Popular Destinations
        </h2>
      </main>
      <Footer />
    </>
  );
}
```

2.3.7 Reusable Components

A component becomes more useful when it can display different data without rewriting its structure.

Consider three destinations Munnar, Wayanad, Alappuzha.

Instead of creating three separate components, one DestinationCard component can be reused.

```
function DestinationCard() {
  return (
    <article>
      <h2>
        Munnar
      </h2>
      <p>
        Hill destination
      </p>
    </article>
  );
}
```

The next step is to make the component receive destination information using props.

Props and State

2.3.8 Passing Data Using Props

Props are values passed from a parent component to a child component.

For example:

```
function DestinationCard(
  props
) {
  return (
    <article>
      <h2>
        {props.name}
      </h2>
```

```
            <p>
                {props.type}
            </p>
        </article>
    );
}
```

The parent component can provide values:

```
function App() {
    return (
        <>
            <DestinationCard
                name="Munnar"
                type="Hill destination"
            />
            <DestinationCard
                name="Wayanad"
                type="Nature destination"
            />
        </>
    );
}
```

The same component is reused with different values.

Destructuring Props

Instead of:

```
function DestinationCard(
    props
) {
    return (
        <h2>
```

```
            {props.name}
        </h2>
    );
}
```

props can be destructured:

```
function DestinationCard(
    {
        name,
        type
    }
) {
    return (
        <article>
            <h2>
                {name}
            </h2>
            <p>
                {type}
            </p>
        </article>
    );
}
```

Kerala Explorer

This allows one component (DestinationCard) to represent all locations (Munnar, Wayanad, Alappuzha, Thekkady, Varkala)

2.3.9 Understanding State

Props are received from a parent component.

State represents data that belongs to a component and may change during its use.

For example, Kerala Explorer may need to remember the currently selected destination.



When the user selects another destination, eg Munnar to Wayanad the component state changes.

React then updates the relevant interface.

Props vs State

Table 2.3.1 Props vs State

Props	State
Passed from parent	Managed by component
Used to receive data	Used to store changing data
Read by child	Can change through state updates
Helps component reuse	Supports interactive behaviour

2.3.10 Managing State with useState

React provides the useState Hook for managing local component state.

For example:

```
import {
    useState
} from "react";
function DestinationSelector() {
    const [
        destination,
        setDestination
    ] = useState("Munnar");
    return (
        <div>
            <h2>
                {destination}
            </h2>
            <button
                onClick={
                    function () {
```

```
                    setDestination(
                      "Wayanad"
                    );
                  }
                }
              >
                View Wayanad
              </button>
            </div>
          );
        }
```

The state variable is:

destination

The function used to update it is:

setDestination

The initial value is:

"Munnar"

General Syntax

```
const [
  value,
  setValue
] = useState(
  initialValue
);
```

Kerala Explorer

The selected destination can be stored as state:

```
const [
  selectedDestination,
  setSelectedDestination
] = useState("Munnar");
```

When the state changes, React re-renders the component.

2.3.11 Conditional Rendering

Sometimes a component should display different content depending on a condition.

For example:

```
function DestinationInfo(
    {
        selected
    }
) {
    if (!selected) {
        return (
            <p>
                Select a destination.
            </p>
        );
    }
    return (
        <p>
            Explore {selected}.
        </p>
    );
}
```

Conditional rendering can also use the ternary operator:

```
function Message(
    {
        loggedIn
    }
) {
    return (
        <p>
            {
                loggedIn
                    ? "Welcome to Kerala Explorer."
```

```
                : "Please sign in."
            }
        </p>
    );
}
```

Kerala Explorer

If no destination has been selected:

Select a destination.

If Munnar has been selected:

Explore Munnar.

Lists and Keys

2.3.12 Rendering Lists

Kerala Explorer may contain an array of destinations:

```
const destinations = [
    {
        id: 1,
        name: "Munnar"
    },
    {
        id: 2,
        name: "Wayanad"
    },
    {
        id: 3,
        name: "Alappuzha"
    }
];
```

React can use map() to create a component for each destination.

```
function DestinationList() {
  return (
    <section>
      {
        destinations.map(
          function (
            destination
          ) {
            return (
              <h2
                key={
                  destination.id
                }
              >
                {
                  destination.name
                }
              </h2>
            );
          }
        )
      }
    </section>
  );
}
```

The map() method transforms each array element into a piece of JSX.

Kerala Explorer

The data:

Munnar

Wayanad

Alappuzha

can become:

Munnar

Wayanad

Alappuzha

in the rendered interface.

2.3.13 Keys in React Lists

When rendering a list, React uses a key to identify individual items.

For example:

```
{
   destinations.map(
      function (
         destination
      ) {
         return (
            <DestinationCard
               key={
                  destination.id
               }
               name={
                  destination.name
               }
            />
         );
      }
   )
}
```

Here:

key = destination.id

identifies each destination.

A key should be stable and unique among the items in the list.

Why Keys Matter

Suppose the list changes:

Munnar

Wayanad

Alappuzha

becomes:

Munnar

Thekkady

Wayanad

Alappuzha

Keys help React identify which items correspond to existing elements and which item has been added.

Events and Forms

2.3.14 Synthetic Events

React provides an event system for handling user interactions.

For example:

```
function DestinationButton() {
  function handleClick() {
    console.log(
      "Destination selected"
    );
  }
  return (
    <button
      onClick={
        handleClick
      }
    >
      View Details
    </button>
  );
}
```

The onClick property specifies the function to execute when the button is clicked.

React events provide a consistent event interface across supported environments.

Event Object

The event handler can receive an event object:

```
function SearchBox() {
    function handleChange(
        event
    ) {
        console.log(
            event.target.value
        );
    }
    return (
        <input
            onChange={
                handleChange
            }
        />
    );
}
```

Kerala Explorer

A tourist clicking:

View Details

can trigger an event handler that updates the interface.

2.3.15 Handling Form Input

Kerala Explorer may contain a tourist enquiry form.

```
function EnquiryForm() {
    return (
        <form>
            <label for=”name”>
                Name:
            </label>
            <input
                id=”name”
                type="text"
```

```
            />
            <button
               type="submit"
            >
               Submit
            </button>
         </form>
      );
   }
```

A React component can respond to changes in the input field.

```
function EnquiryForm() {
   function handleChange(
      event
   ) {
      console.log(
         event.target.value
      );
   }
   return (
      <input
         type="text"
         onChange={
            handleChange
         }
      />
   );
}
```

The value entered by the user is available through:

event.target.value

2.3.16 Controlled Components

A controlled component is a form element whose value is controlled by React state.

For example:

```
import {
    useState
} from "react";
function EnquiryForm() {
    const [
        name,
        setName
    ] = useState("");
    function handleChange(
        event
    ) {
        setName(
            event.target.value
        );
    }
    return (
        <form>
            <label for=”name”>
                Name:
            </label>
            <input
            id=”name”
                type="text"
                value={name}
                onChange={
                    handleChange
                }
```

```
            />
            <p>
                Name: {name}
            </p>
        </form>
    );
}
```

The relationship is:



Kerala Explorer

The tourist's name can therefore be maintained as React state.

If the tourist enters:

Anu

the state becomes:

name = "Anu"

and the component renders the updated value.

2.3.17 Handling Form Submission

A form can handle submission using the onSubmit event.

```
function EnquiryForm() {
    function handleSubmit(
        event
    ) {
        event.preventDefault();
        console.log(
            "Enquiry submitted"
        );
    }
    return (
```

```
        <form
          onSubmit={
            handleSubmit
          }
        >
          <input
            type="text"
          />
          <button
            type="submit"
          >
            Send Enquiry
          </button>
        </form>
    );
}
```

The preventDefault() method prevents the browser's normal form submission behaviour.

This is similar to the event-handling concept introduced in Block I, but the event is now handled within a React component.

Sharing State

2.3.18 Understanding Lifting State Up

Consider a Kerala Explorer interface containing:

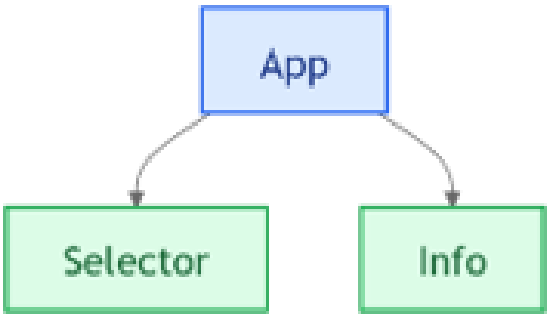
DestinationSelector

DestinationInfo

The selected destination is required by both components.

If the state is stored separately inside each component, the components may not have access to the same selected value.

Instead, the state can be moved to their common parent.

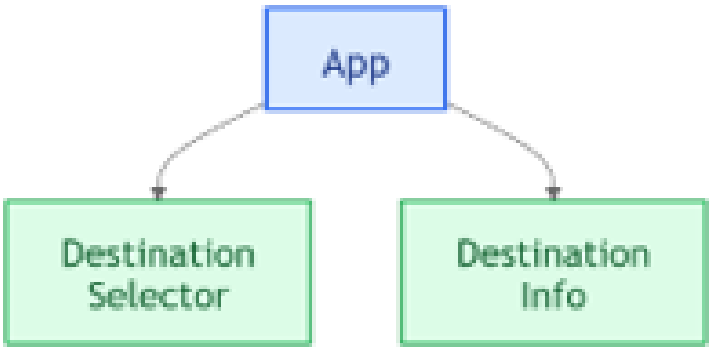


The parent maintains the state and passes the required information to the children.

This technique is called lifting state up.

2.3.19 Lifting State Up in Kerala Explorer

Suppose the application contains:



The selected destination can be stored in App.

```
import {
    useState
} from "react";
function App() {
    const [
        selectedDestination,
        setSelectedDestination
    ] = useState(
        "Munnar"
    );
    return (
        <>
```

```
            <DestinationSelector
               selected={
                  selectedDestination
               }
               onSelect={
                  setSelectedDestination
               }
            />
            <DestinationInfo
               destination={
                  selectedDestination
               }
            />
         </>
      );
   }
```

The selector receives:

selected

onSelect

The information component receives:

destination

Data Flow



The child component does not directly modify the parent's state.

Instead, it calls the function provided by the parent.

Interventions for Value Addition

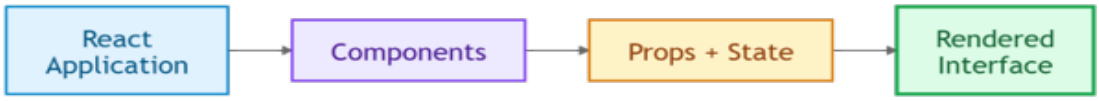
Kerala Explorer – From Pages to Components

The architecture of Kerala Explorer has now changed.

Earlier:



With React:



A destination card can be represented once:

DestinationCard

and reused for:

Munnar

Wayanad

Alappuzha

Thekkady

Varkala

This is the central idea of component-based development.

Think About It

Suppose Kerala Explorer contains 100 destination cards.

Would it be better to write 100 separate pieces of interface code, or create one reusable DestinationCard component and provide each destination as data?

The component-based approach separates the structure of the interface from the data displayed by the interface.

Summarised Overview

React is a JavaScript library for building user interfaces, and React applications can be created using tools such as Create React App or Vite. JSX provides an HTML-like syntax for describing React interfaces, with JavaScript expressions embedded using {}. React uses a virtual DOM representation to help manage interface updates. Functional components are JavaScript functions that return JSX and can be reused to represent common interface structures. Props are used to pass data from parent components to child components, while state represents data that can change during component use and can be managed using useState. Conditional rendering displays different content based on conditions, while map() can be used to render collections of data and keys help React identify items in rendered lists. React provides an event system for handling user interactions, and controlled components use React state to manage form values, with onChange handling input changes and onSubmit handling form submission. Lifting state up moves shared state to a common parent, allowing props to pass state and update functions between components. Component-based development supports the creation of reusable, organized, and maintainable user interfaces.

Assignments

Develop a Component-Based Kerala Explorer Application

The Kerala Explorer tourism portal wants to redesign its frontend using React.

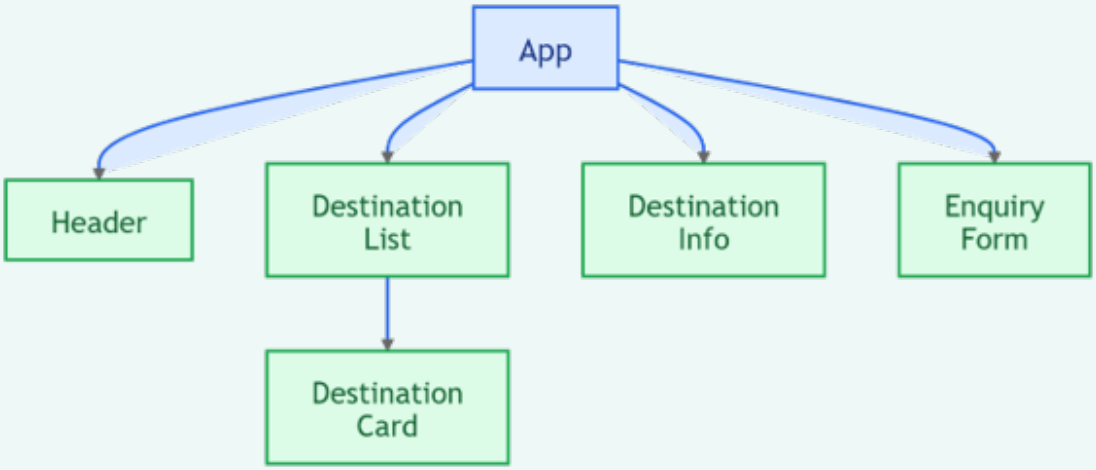
Develop a React application that displays Kerala tourist destinations using reusable components.

The application should:

- be created using React and Vite or Create React App;
- use JSX;
- contain reusable functional components;
- pass destination information using props;
- maintain selected destination using useState;

- conditionally display destination information;
- render destination cards using map();
- use suitable keys for list items;
- handle button events using React events;
- contain a controlled tourist enquiry form;
- handle form input using React state; and
- demonstrate lifting state up between components.

Suggested Component Structure



Suggested Destination Data

```
const destinations = [
  {
    id: 1,
    name: "Munnar",
    type: "Hill destination"
  },
  {
    id: 2,
    name: "Wayanad",
    type: "Nature destination"
  },
  {
    id: 3,
```

```
        name: "Alappuzha",
        type: "Backwater destination"
    },
    {
        id: 4,
        name: "Thekkady",
        type: "Wildlife destination"
    }
];
```

Tasks

1. Create the React project.
2. Create an App component.
3. Create reusable destination components.
4. Pass destination data using props.
5. Render destination cards using map().
6. Assign suitable keys to list items.
7. Add a View Details button.
8. Use useState to store the selected destination.
9. Display destination information conditionally.
10. Create a controlled tourist enquiry form.
11. Handle form input using onChange.
12. Handle form submission using onSubmit.
13. Demonstrate lifting state up where appropriate.
14. Explain the component structure and data flow.

Suggested Submission

Submit:

- React project source code;
- component files;
- screenshots of the application;
- sample destination data; and
- a brief explanation of props, state, events and component communication.

Note: API integration is not required in this assignment. It will be introduced in Block II – Unit 4.

Reference

1. Zammetti, F. (2020). *Modern Full-Stack Development: Using TypeScript, React, Node.js, Webpack, and Docker.* Apress.
2. Northwood, C. (2018). *The full stack developer: your essential guide to the everyday skills expected of a modern full stack web developer.* Apress.
3. Subramanian, V. (2019). *Pro Mern Stack: Full Stack Web App Development with Mongo, Express, React and Node.* Apress.
4. Dayley, B., Dayley, B., & Dayley, C. (2017). *Node.js, MongoDB and Angular Web Development: The definitive guide to using the MEAN stack to build web applications.* Addison-Wesley Professional.

Suggested Reading

1. Hoque, S. (2020). *Full-Stack React Projects: Learn MERN stack development by building modern web apps using MongoDB, Express, React, and Node. js*. Packt Publishing Ltd.
2. Koroliova, E. W. I. (2018). *MERN quick start guide: build web applications with MongoDB, Express. js, React, and Node*. Packt Publishing Ltd.

3. Bugl, D. (2024). *Modern full-stack React projects: Build, maintain, and deploy modern web apps using MongoDB, Express, React, and Node.js.* Packt Publishing Ltd.

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

4 UNIT

React Routing, Backend Integration, and Full-Stack Application Development

Learning Outcomes

After completion of this unit, the learner will be able to:

- explain the purpose of side effects and API communication in React applications
- apply the useEffect Hook to retrieve and manage data from REST APIs
- apply loading and error handling techniques when retrieving API data
- configure client-side routing using React Router and navigate between application views
- integrate a React frontend with an Express backend and a MongoDB or PostgreSQL database
- develop and deploy a basic full-stack web application using React, Express.js, and a database

Background

In the previous three units, Kerala Explorer was developed layer by layer.

In Unit 1, a backend was created using Node.js and Express.js.

In Unit 2, persistent data storage was introduced using MongoDB and PostgreSQL.

In Unit 3, the frontend was redesigned using React.

However, the React application developed in Unit 3 uses locally defined destination data.

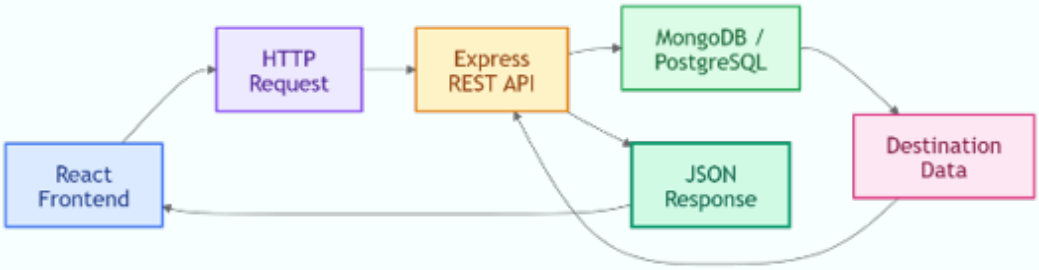
For example:

```
const destinations = [
  {
    id: 1,
    name: "Munnar",
    type: "Hill destination"
  },
  {
    id: 2,
    name: "Wayanad",
    type: "Nature destination"
  }
];
```

A real tourism application should not depend on hard-coded data.

The React frontend should retrieve destination information from the backend.

The complete architecture therefore becomes:



The application may also contain multiple views:

Home

Destinations

Destination Details

Enquiry

About

React Router can be used to navigate between these views without requiring a traditional full-page navigation.

Thus, this unit brings together the concepts developed throughout Block II to create a complete full-stack Kerala Explorer application.

Keywords

React Side Effects, API Integration, Loading and Error Handling, Client-Side Routing, React Router, Navigation, Backend Integration, Full-Stack Architecture, Application Deployment

Discussion

2.4.1 Understanding Side Effects in React

A React component primarily describes what should appear in the user interface.

However, an application may also need to perform operations outside the direct rendering process.

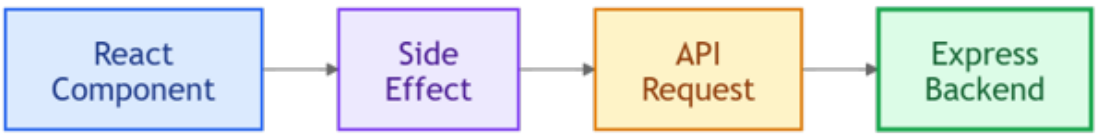
Examples include:

1. requesting data from a server;
2. interacting with browser APIs;
3. setting up timers; and
4. performing other external operations.

Such operations are commonly referred to as side effects.

For Kerala Explorer, retrieving destination information from the Express backend is a side effect.

The flow is:



The returned data can then be used to update the React interface.

Kerala Explorer Example

Initially, the application may display:

Loading destinations...

After the API responds:

Munnar

Wayanad

Alappuzha

Thekkady

The interface therefore changes according to the result of the external operation.

2.4.2 The useEffect Hook

React provides the useEffect Hook for performing side effects in functional components.

It can be imported using:

```
import {
    useEffect
} from "react";
```

A simple example is:

```
useEffect(
    function () {
        console.log(
            "Component rendered"
        );
    }
);
```

The function passed to useEffect() contains the operation that React should perform as an effect.

Basic Structure

```
useEffect(
    function () {
        // Side effect
    },
    []
);
```

The second argument is called the dependency array.

An empty dependency array is commonly used when an effect should run after the component's initial rendering.

Kerala Explorer

A component can use:

```
useEffect(
   function () {
      // Fetch destinations
   },
   []
);
```

to initiate retrieval of destination data.

2.4.3 Fetching Data from an API

The React frontend can communicate with the Express backend using the browser's fetch() function.

Suppose the Express API provides:

GET /destinations

The React component can request the data using:

```
fetch(
   "http://localhost:3000/destinations"
);
The response can be converted into JSON using:
response.json();
A simple example is:
useEffect(
   function () {
      fetch(
         "http://localhost:3000/destinations"
      )
      .then(
         function (response) {
```

```
            return response.json();
        }
    )
    .then(
        function (data) {
            console.log(data);
        }
    );
},
[]
);
```

Data Flow



Kerala Explorer Example

The API may return:

```
[
  {
    "id": 1,
    "name": "Munnar",
    "type": "Hill destination"
  },
  {
    "id": 2,
    "name": "Wayanad",
    "type": "Nature destination"
  }
]
```

React can use this data to display destination cards.

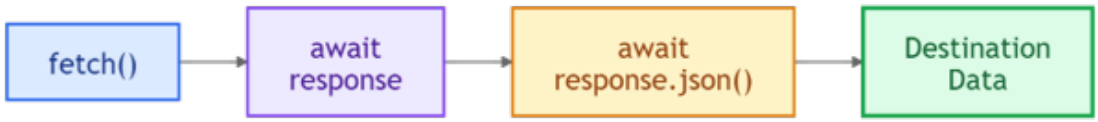
2.4.4 Fetching API Data with async/await

The same API request can be written using async/await.

```
useEffect(
   function () {
      async function loadDestinations() {
         const response =
            await fetch(
               "http://localhost:3000/destinations"
            );
         const data =
            await response.json();
         console.log(data);
      }
      loadDestinations();
   },
   []
);
```

The async function allows asynchronous operations to be written in a sequential style.

The flow is:



Kerala Explorer

The retrieved destination data can later be stored using useState.

```
const [
   destinations,
   setDestinations
] = useState([]);
```

Then:

setDestinations(data);

updates the state.

2.4.5 Loading and Error States

Managing Loading State

API communication may take some time.

During this period, the application should provide feedback to the user.

For example:

Loading destinations...

A loading state can be created using useState.

```
const [
  loading,
  setLoading
] = useState(true);
```

After data is successfully retrieved:

setLoading(false);

The component can conditionally display:

```
if (loading) {
  return (
    <p>
      Loading destinations...
    </p>
  );
}
```

Data Flow



2.4.6 Handling API Errors

An API request may fail because:

- the server is unavailable;
- the network connection fails;
- the API returns an error; or
- the requested resource cannot be found.

An error state can be created using:

```
const [
  error,
  setError
] = useState("");
```

The API request can be placed inside a try...catch block.

```
try {
  const response =
    await fetch(
      "http://localhost:3000/destinations"
    );
  if (!response.ok) {
    throw new Error(
      "Unable to retrieve destinations"
    );
  }
  const data =
    await response.json();
  setDestinations(data);
}
catch (error) {
  setError(
    error.message
  );
```

```
}
The interface can display:
if (error) {
   return (
      <p>
         {error}
      </p>
   );
}
```

Kerala Explorer

If the backend is unavailable, the interface can display:

Unable to retrieve destinations.

Please try again later.

Instead of displaying a blank page.

2.4.7 Displaying API Data

Once data is retrieved, it can be stored in React state.

```
const [
   destinations,
   setDestinations
] = useState([]);
After fetching:
setDestinations(data);
The data can be displayed using map().
{
   destinations.map(
      function (
         destination
      ) {
         return (
```

```
            <article
              key={
                destination.id
              }
            >
              <h2>
                {destination.name}
              </h2>
              <p>
                {destination.type}
              </p>
            </article>
          );
        }
      )
    }
```

The complete flow is:



2.4.8 Complete Data Fetching Component

The concepts can be combined into a reusable component.

```
import {
  useEffect,
  useState
} from "react";
function DestinationList() {
 // Define state
  const [ destinations, setDestinations ] = useState([]);
  const [ loading, setLoading ] = useState(true);
```

```
  const [ error,  setError] = useState("");
//Fetch and manage the API response
  useEffect(
    function () {
      async function loadDestinations() {
        try {
          const response =
            await fetch(
              "http://localhost:3000/destinations"
            );
          if (!response.ok) {
            throw new Error(
              "Unable to retrieve destinations"
            );
          }
          const data =
            await response.json();
          setDestinations(
            data
          );
        }
        catch (error) {
          setError( error.message );
        }
        finally {
          setLoading( false );
        }
      }
      loadDestinations();
    },
    []
```

```
    );
//display loading error
    if (loading) {
        return  <p>  Loading destinations...  </p>;
    }
    if (error) {
        return  <p>   {error}   </p>;
    }
    return (
        <section>
            <h2>
                Kerala Destinations
            </h2>
            {
                destinations.map(
                    function (   destination    ) {
                        return (
                            <article
                                key={
                                    destination.id
                                }
                            >
                                <h3>
                                    {
                                        destination.name
                                    }
                                </h3>
                                <p>
                                    {
                                        destination.type
                                    }
```

```
                        </p>
                    </article>
                );
            }
        )
    }
</section>
);
}
```

export default DestinationList;

The component now demonstrates:



2.4.9 React Router

Need for Client-Side Routing

Kerala Explorer can contain several views:

Home

Destinations

About

Contact

A traditional website may load a separate webpage for each URL.

A React application can instead use client-side routing to display different components based on the URL.

For example:

/ → Home

/destinations → Destinations

/about → About

/contact → Contact

The browser remains within the React application while the displayed component changes.

Client-Side Routing



2.4.10 Setting Up React Router

React Router can be installed using:

npm install react-router-dom

The application can import routing components:

```
import {
    BrowserRouter,
    Routes,
    Route
} from "react-router-dom";
```

A basic router can be configured as:

```
function App() {
    return (
        <BrowserRouter>
            <Routes>
                <Route
                    path="/"
                    element={
                        <Home />
                    }
                />
                <Route
                    path="/destinations"
                    element={
                        <Destinations />
                    }
                />
            </Routes>
        </BrowserRouter>
    );
}
```

2.4.11 Route and Component Mapping

A Route connects a URL path with a React component.

For example:

```
<Route
   path="/about"
   element={
      <About />
   }
/>
```

When the user navigates to:

/about

the About component is rendered.

A Kerala Explorer application can define:

```
<Routes>
   <Route
      path="/"
      element={
         <Home />
      }
   />
   <Route
      path="/destinations"
      element={
         <Destinations />
      }
   />
   <Route
      path="/about"
      element={
         <About />
```

```
      }
    />
    <Route
      path="/contact"
      element={
        <Contact />
      }
    />
</Routes>
```

2.4.12 Navigation Using Link

React Router provides the Link component for navigation.

It can be imported using:

```
import {
  Link
} from "react-router-dom";
A navigation bar can be created using:
function Navigation() {
  return (
    <nav>
      <Link to="/">
        Home
      </Link>
      <Link to="/destinations">
        Destinations
      </Link>
      <Link to="/about">
        About
      </Link>
      <Link to="/contact">
        Contact
```

```
        </Link>
      </nav>
    );
}
```

The user can select:

Destinations

to navigate to:

/destinations

Why Link?

Using Link allows navigation within the React application without relying on traditional full-page navigation.

2.4.13 Dynamic Routes

A destination details page may require a different URL for each destination.

For example:

/destinations/1

/destinations/2

/destinations/3

React Router can define a dynamic route:

```
<Route
  path="/destinations/:id"
  element={
    <DestinationDetails />
  }
/>
```

The :id portion represents a route parameter.

The application can therefore use one component to represent many destinations.

Kerala Explorer



2.4.14 useNavigate for Programmatic Navigation

Sometimes navigation needs to occur after an action rather than after clicking a Link.

React Router provides the useNavigate Hook.

It can be imported using:

```
import {
   useNavigate
} from "react-router-dom";
A component can use:
function DestinationCard() {
   const navigate =
      useNavigate();
   function handleClick() {
      navigate(
         "/destinations/1"
      );
   }
   return (
      <button
         onClick={
            handleClick
         }
```

```
        >
            View Details
        </button>
    );
}
```

When the button is clicked, the application navigates to:

/destinations/1

Kerala Explorer Example

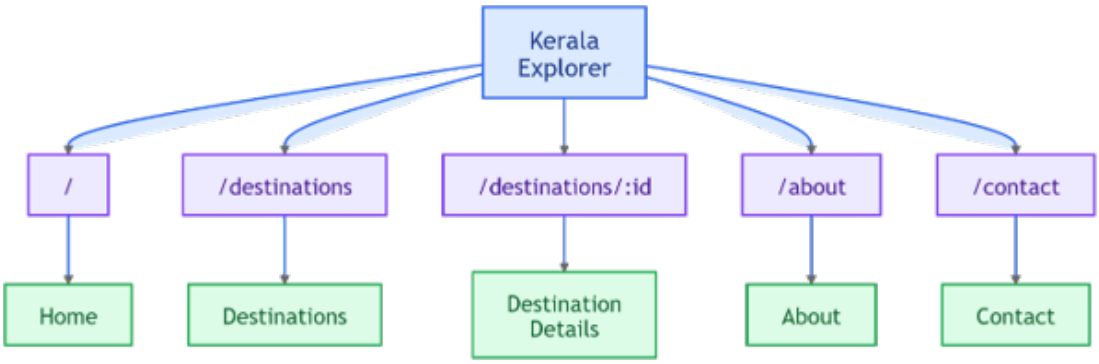
A destination card can navigate to the corresponding details page.

```
navigate(
    `/destinations/${destination.id}`
);
```

2.4.15 Multi-View Application

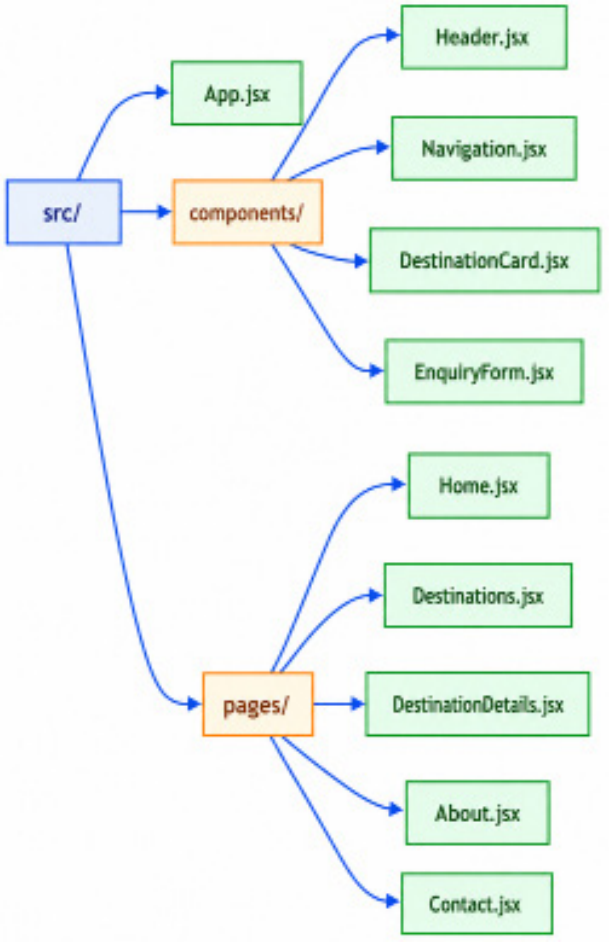
Designing the Kerala Explorer Routes

The application can now contain multiple views.



This provides a structured navigation system.

Suggested Component Structure



2.4.16 React Router Example

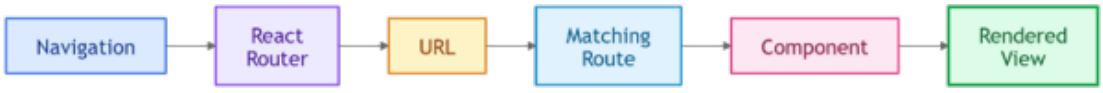
```
import {
    BrowserRouter, Routes, Route, Link} from "react-router-dom";
//Create the View Components
function Home() {
    return <h1> Welcome to Kerala Explorer </h1>;
}
function Destinations() {
    return <h1> Kerala Destinations </h1>;
}
function About() {
    return <h1> About Kerala Explorer </h1>;
```

```
    }
    function Contact() {
      return  <h1> Contact Kerala Explorer  </h1>;
    }
//Create Navigation Links
    function Navigation() {
      return (
        <nav>
          <Link to="/">  Home </Link>
          <Link to="/destinations">  Destinations  </Link>
          <Link to="/about"> About </Link>
          <Link to="/contact"> Contact </Link>
        </nav>
      );
    }
//Define the Routes
    function App() {
      return (
        <BrowserRouter>
          <Navigation />
          <Routes>
            <Route
              path="/"
              element={
                <Home />
              }
            />
            <Route
              path="/destinations"
              element={
                <Destinations />
```

```
                }
            />
            <Route
                path="/about"
                element={
                    <About />
                }
            />
            <Route
                path="/contact"
                element={
                    <Contact />
                }
            />
        </Routes>
    </BrowserRouter>
  );
}
```

export default App;

Application Flow



2.4.17 Backend Integration

Connecting React to Express

The React application developed in Unit 3 can now communicate with the Express API developed in Units 1 and 2.

The architecture becomes:



For example:

GET /destinations

can be sent from React to Express.

The Express server retrieves the destination unu from the database and returns JSON.

During development, the React frontend and Express backend may run on different ports. For example, the React development server may run on port 5173, while the Express server runs on port 3000. Because these represent different origins, the backend must allow requests from the frontend. You will need to install and use the CORS package.

Kerala Explorer



2.4.18 React Destination List with Backend Data

The destination list from Unit 3 used locally defined data.

Now it can retrieve data from the backend.

```
import {
    useEffect,
    useState
} from "react";
function Destinations() {
    const [
        destinations,
        setDestinations
    ] = useState([]);
    const [
        loading,
        setLoading
    ] = useState(true);
    const [
        error,
```

```
    setError
] = useState("");
useEffect(
    function () {
        async function loadData() {
            try {
                const response =
                    await fetch(
                        "http://localhost:3000/destinations"
                    );
                if (!response.ok) {
                    throw new Error(
                        "Failed to load destinations"
                    );
                }
                const data =
                    await response.json();
                setDestinations(
                    data
                );
            }
            catch (error) {
                setError(
                    error.message
                );
            }
            finally {
                setLoading(
                    false
                );
            }
```

```
        }
        loadData();
    },
    []
);
if (loading) {
    return (
        <p>
            Loading destinations...
        </p>
    );
}
if (error) {
    return (
        <p>
            {error}
        </p>
    );
}
return (
    <section>
        <h1>
            Kerala Destinations
        </h1>
        {
            destinations.map(
                function (
                    destination
                ) {
                    return (
                        <article
```

```
                        key={
                            destination.id
                        }
                    >
                        <h2>
                            {
                                destination.name
                            }
                        </h2>
                        <p>
                            {
                                destination.type
                            }
                        </p>
                    </article>
                );
            }
        )
    }
</section>
);
}
```

export default Destinations;

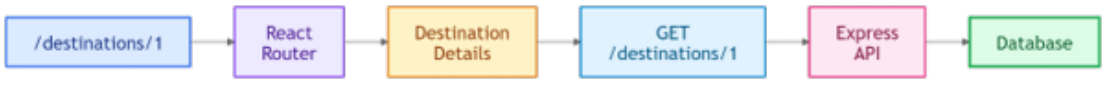
2.4.19 Fetching a Destination by ID

The destination details page can use a dynamic route:

/destinations/:id

The route component can retrieve the corresponding destination from the backend.

The architecture is:



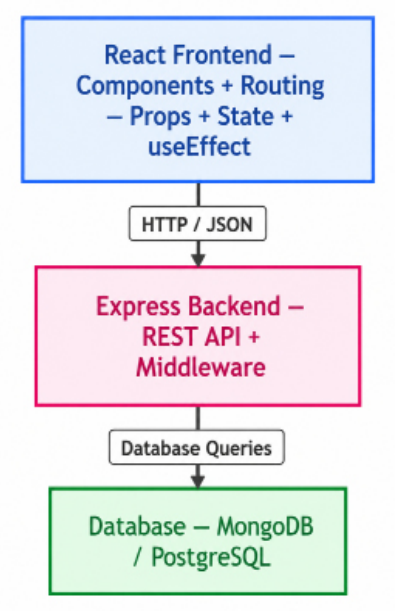
A destination details component can use the route parameter.

```
import {
    useParams
} from "react-router-dom";
The ID can be obtained using:
const {
    id
} = useParams();
The component can then request:
fetch(
    `http://localhost:3000/destinations/${id}`
);
```

2.4.20 Complete Full-Stack Application

Full-Stack Architecture

The complete Kerala Explorer application now contains three major layers.



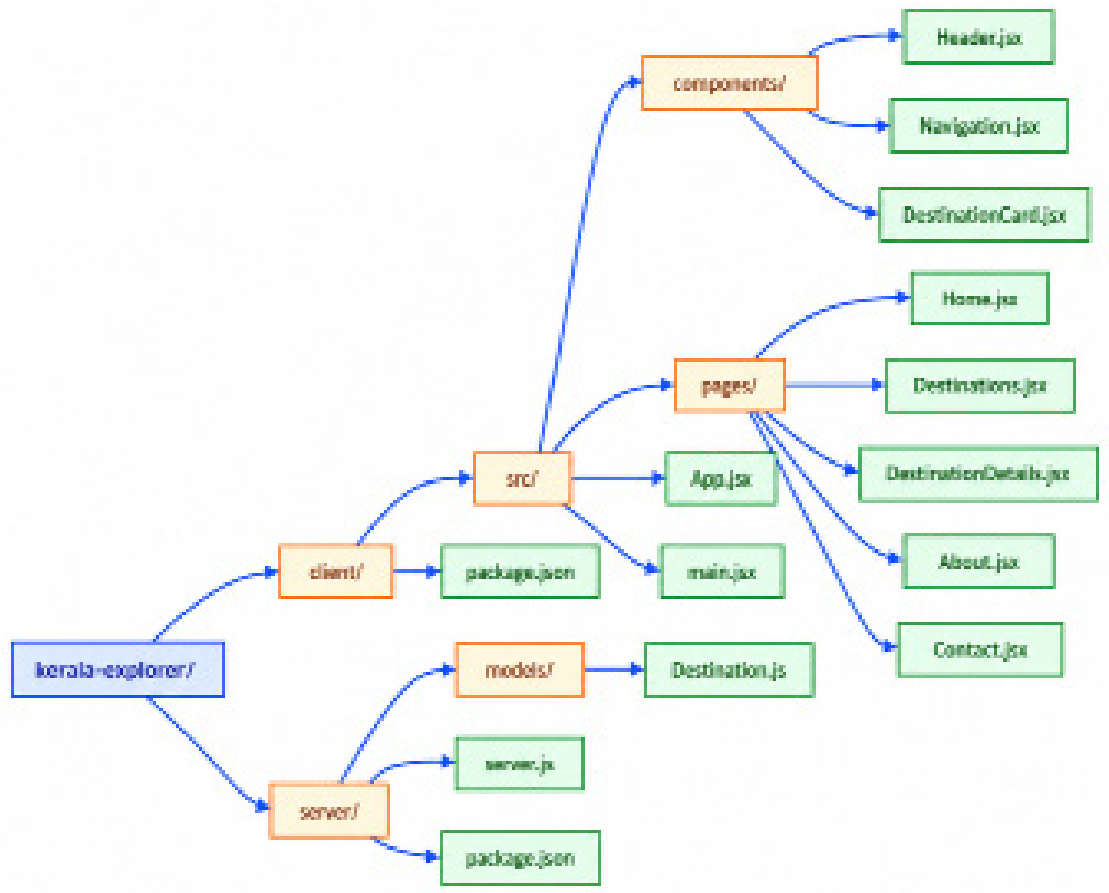
This is a full-stack web application.

Table 2.4.1 Responsibilities of Each Layer

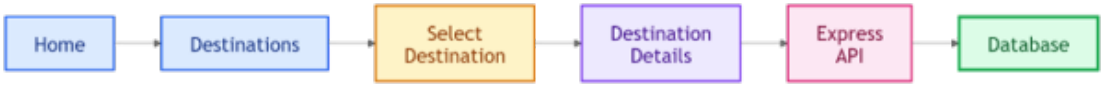
Layer	Responsibility
React	User interface and client-side interaction
Express	API and server-side request processing
MongoDB / PostgreSQL	Persistent data storage

2.4.21 Kerala Explorer Application Structure

A complete project can be organized as:



Application Flow



Think About It

Suppose the Kerala Explorer application has:

- 500 tourist destinations;
- multiple destination categories;
- destination detail pages;
- an enquiry form;
- an Express REST API; and
- a MongoDB database.

Which part of the application should be responsible for:

1. displaying the interface?
2. handling HTTP requests?
3. storing destination information?
4. navigating between views?
5. retrieving destination data after a page is loaded?

The concepts introduced in this unit provide the answers.

Summarised Overview

React applications may need to perform operations outside the rendering process, such as retrieving data from external systems. The useEffect Hook can be used to perform such side effects, including fetching data from an Express REST API. Retrieved JSON data can be stored in React state and displayed dynamically, while loading and error states provide appropriate feedback during API communication. React Router enables client-side routing by mapping URL paths to React components and provides features such as BrowserRouter, Routes, Route, Link, dynamic routes, and useNavigate for navigation between views. A React frontend can communicate with an Express backend through HTTP requests, while the backend interacts with MongoDB or PostgreSQL for persistent data storage. A full-stack application combines these frontend, backend, and database layers into a single system. The completed Kerala Explorer application demonstrates this integration, and deployment platforms such as Render or Railway can be used to make the application available in a hosted environment.

Assignments

Develop the Complete Kerala Explorer Full-Stack Application

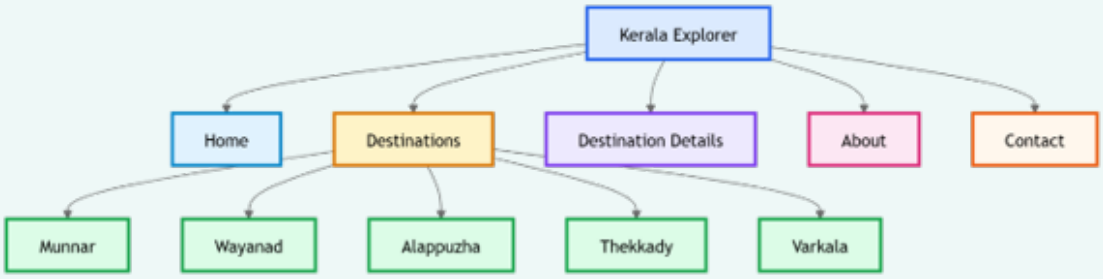
The Kerala Explorer application has been developed progressively throughout Block II.

Extend the application into a complete full-stack web application using – React, Express.js, MongoDB or PostgreSQL

The application should:

- use React for the frontend;
- use React Router for client-side navigation;
- use useEffect for API-related side effects;
- retrieve destination data from the Express backend;
- display loading feedback while data is being retrieved;
- display an appropriate error message when the API request fails;
- display destination information dynamically;
- provide a destination details view;
- use dynamic routing for destination IDs;
- use Link for navigation;
- use useNavigate where programmatic navigation is appropriate;
- use Express.js for the backend;
- use MongoDB or PostgreSQL for persistent storage;
- provide RESTful destination endpoints; and
- demonstrate the complete frontend → backend → database architecture.

Suggested Application Structure



Suggested Routes

/

/destinations

/destinations/:id

/about

/contact

Suggested API

GET /destinations

GET /destinations/:id

POST /destinations

PUT /destinations/:id

DELETE /destinations/:id

Tasks

1. Create the React frontend.
2. Configure React Router.
3. Create Home, Destinations, Destination Details, About and Contact views.
4. Create reusable components.
5. Connect the React frontend to the Express API.
6. Use useEffect to retrieve destination data.
7. Store API data using React state.
8. Implement loading feedback.
9. Implement API error handling.

10. Display destinations dynamically.
11. Implement dynamic destination routes.
12. Retrieve a destination using its ID.
13. Configure the Express backend.
14. Connect the backend to MongoDB or PostgreSQL.
15. Test all REST API endpoints.
16. Test navigation between React views.
17. Test loading and error states.
18. Prepare the application for deployment.
19. Deploy the application using a suitable platform such as Render or Railway.
20. Demonstrate the complete working application.

Suggested Submission

Submit:

- React frontend source code;
- Express backend source code;
- database schema/model;
- sample destination data;
- screenshots of the application;
- screenshots of API testing;
- screenshots of React Router navigation;
- deployed application details;
- sample API requests and responses; and
- a brief explanation of the full-stack architecture.

Reference

1. Subramanian, V. *Pro MERN Stack: Full Stack Web App Development with Mongo, Express, React, and Node*. Springer.
2. Zammetti, F. *Modern Full Stack Development: Using TypeScript, React, Node.js, Webpack, and Docker.* Springer.
3. Northwood, C. *The Full Stack Developer: Your Essential Guide to the Everyday Skills Expected of a Modern Full Stack Web Developer*. Springer.

Suggested Reading

1. Subramanian, V. *Pro MERN Stack: Full Stack Web App Development with Mongo, Express, React, and Node*. Springer.
2. Northwood, C. *The Full Stack Developer: Your Essential Guide to the Everyday Skills Expected of a Modern Full Stack Web Developer.* Springer.
3. Zammetti, F. *Modern Full Stack Development: Using TypeScript, React, Node.js, Webpack, and Docker*. Springer.
4. Haverbeke, M. *Eloquent JavaScript: A Modern Introduction to Programming*. No Starch Press.
5. Dayley, B. *Node.js, MongoDB, and Angular Web Development*. Pearson.

Space for Learner Engagement for Objective Questions

Learners are encouraged to develop objective questions based on the content in the paragraph as a sign of their comprehension of the content. The Learners may reflect on the recap bullets and relate their understanding with the narrative in order to frame objective questions from the given text. The University expects that 1 - 2 questions are developed for each paragraph. The space given below can be used for listing the questions.

Model Question Paper Sets

SREENARAYANAGURU OPEN UNIVERSITY

MODEL QUESTION PAPER (SET- 01)

QP CODE: …… Reg. No:.....

Name: ……

FIRST SEMESTER – MASTER OF COMPUTER APPLICATIONS (MCA) EXAMINATION

SKILL ENHANCEMENT COURSE

M26CA01SE – WEB PROGRAMMING

POST-GRADUATE PROGRAMME (PG)

2026–27 Admission Onwards

Time: 3 Hours Max. Marks: 70

Section A

Answer any ten of the following questions. Each answer should be in two or three valid points. Each question carries 2 marks.

(10x2=20)

1. What is the role of a web browser in web page development?
2. Differentiate between internal and external hyperlinks.
3. What is the purpose of a media query in CSS?
4. Define responsive web design.
5. What is destructuring in JavaScript?
6. What is the purpose of template literals?
7. What is the role of fetch() in JavaScript?
8. What is the purpose of JSON.parse()?
9. What is Express.js?
10. What is the purpose of express.json() middleware?
11. What is an HTTP status code? Give two examples.

12. Differentiate between SQL tables and MongoDB collections.

13. What is the purpose of the pg library in Node.js?

14. What is the Virtual DOM in React?

15. What is the purpose of the useEffect hook?

Section B

Answer any six of the following questions. Provide adequate valid points with clear explanations, proportionate to the marks allotted. Each question carries 5 marks.

(6x5=30)

16. Explain HTML forms and their different form controls. Discuss text, password, radio, checkbox, select, and textarea elements.

17. Explain the use of CSS Flexbox and Grid for page layout. Illustrate their differences with suitable examples.

18. Discuss JavaScript functions, scope, control flow, and object literals with suitable examples.

19. Explain how JavaScript can dynamically create, modify, and remove HTML elements using DOM manipulation.

20. Explain the process of setting up an Express.js application. Discuss routing, request and response objects, and serving static files.

21. Explain the process of connecting a Node.js application to PostgreSQL using the pg library. Illustrate how SQL queries are executed and results are retrieved.

22. Explain MongoDB CRUD operations using Mongoose. Discuss the purpose of schemas and models with suitable examples.

23. Explain React event handling and controlled components. Discuss how form inputs are managed using state.

24. Explain the purpose of useEffect in React. Discuss how it can be used to fetch data from a backend API and handle loading and error states.

Section C

Answer any two of the following questions. Provide adequate valid points with clear explanations, proportionate to the marks allotted. Each question carries 10 marks.

(2X10=20)

25. Explain the different HTML content elements and their applications in web page development. Discuss headings, paragraphs, lists, images, tables, hyperlinks, navigation, and HTML forms with suitable examples.

26. Discuss the role of JavaScript in developing interactive web applications. Explain variables, data types, operators, control flow, functions, array methods, destructuring, spread/rest operators, and arrow functions with suitable examples.

27. Explain the architecture and working of a full-stack web application using React, Express.js, and MongoDB or PostgreSQL. Discuss how the frontend communicates with the backend and how the backend interacts with the database.

28. A university plans to develop a student management application using React. Explain how functional components, props, state, controlled forms, conditional rendering, and list rendering can be used to develop the frontend. Discuss how React Router, useEffect, backend API integration, and deployment using Render or Railway contribute to the development of the ap plication.

SREENARAYANAGURU OPEN UNIVERSITY

MODEL QUESTION PAPER (SET- 02)

QP CODE: …… Reg. No:.....

Name: ……

FIRST SEMESTER – MASTER OF COMPUTER APPLICATIONS (MCA) EXAMINATION

SKILL ENHANCEMENT COURSE

M26CA01SE – WEB PROGRAMMING

POST-GRADUATE PROGRAMME (PG)

2026–27 Admission Onwards

Time: 3 Hours Max. Marks: 70

Section A

Answer any ten of the following questions. Each answer should be in two or three valid points. Each question carries 2 marks.

(10x2=20)

1. What is the purpose of semantic HTML elements? Give two examples.
2. Differentiate between the GET and POST methods in HTML forms.
3. What is CSS specificity?
4. Explain the CSS box model.
5. Differentiate between let and const in JavaScript.
6. What is the purpose of the map() method in JavaScript?
7. What is an event listener in JavaScript?
8. What is the purpose of async and await in JavaScript?
9. What is npm? State its role in Node.js development.
10. What is middleware in Express.js?
11. Differentiate between HTTP GET and POST methods in REST APIs.

12. What is a Mongoose model?

13. What is JSX in React?

14. What is the purpose of props in React?

15. What is conditional rendering in React?

Section B

Answer any six of the following questions. Provide adequate valid points with clear explanations, proportionate to the marks allotted. Each question carries 5 marks.

(6x5=30)

16. Explain the structure of an HTML document. Discuss the purpose of the DOCTYPE, html, head, title, and body elements.

17. Discuss the different types of CSS selectors with suitable examples. Explain how selectors are used to apply styles to HTML elements.

18. Explain JavaScript array methods map(), filter(), and reduce() with suitable examples.

19. Discuss JavaScript event handling. Explain how mouse and keyboard events can be handled using addEventListener() and how preventDefault() works.

20. Explain the event-driven and non-blocking I/O model of Node.js. Discuss the role of the built-in fs, path, and http modules.

21. Explain the different HTTP methods used in RESTful web services. Discuss PUT and DELETE operations with suitable examples.

22. Compare MongoDB and PostgreSQL based on their data organization, structure, and applications. Explain the situations in which each database may be preferred.

23. Explain how React functional components are created and reused. Discuss the role of props and state in component-based development.

24. Explain how React Router supports navigation in a single-page application. Discuss the purpose of Route, Link, and useNavigate().

Section C

Answer any two of the following questions. Provide adequate valid points with clear explanations, proportionate to the marks allotted. Each question carries 10 marks.

(2X10=20)

25. Explain how CSS can be used to design an attractive and responsive website. Discuss inline, internal, and external CSS, selectors, specificity, colours, typography, backgrounds, the box model, and responsive layout techniques with suitable examples.

26. Explain asynchronous JavaScript programming. Discuss callbacks using setTimeout() and setInterval(), Promises, the event loop, async/await, JSON handling, and fetching data from external APIs with suitable examples.

27. A college wants to develop a RESTful API for managing student records using Node.js and Express.js. Explain the steps involved in creating the application. Discuss project setup, routing, middleware, request handling, response handling, HTTP methods, status codes, and error handling.

28. A web application needs to store and manage student records in a database. Explain how database integration can be implemented using either MongoDB with Mongoose or PostgreSQL with the pg library. Discuss database connectivity, data models or tables, and the implementation of create, read, update, and delete operations. Explain how the API endpoints can be tested using Postman or Thunder Client.

സർവ്വകലാശാലാഗീതം

വിദ്യയാൽ സ്വതന്ത്രരാകണം
വിശ്വപൗരരായി മാറണം
ഗ്രഹപ്രസാദമായ് വിളങ്ങണം
ഗുരുപ്രകാശമേ നയിക്കണേ

കൂരിരുട്ടിൽ നിന്നു ഞങ്ങളെ
സൂര്യവീഥിയിൽ തെളിക്കണം
സ്നേഹദീപ്തിയായ്വിളങ്ങണം
നീതിവൈജയന്തി പാറണം

ശാസ്ത്രവ്യാപ്തിയെന്നുമേകണം
ജാതിഭേദമാകെ മാറണം
ബോധരശ്മിയിൽ തിളങ്ങുവാൻ
ജ്ഞാനകേന്ദ്രമേ ജ്വലിക്കണേ

കുരീപ്പുഴ ശ്രീകുമാർ

SREENARAYANAGURU OPEN UNIVERSITY

Regional Centres

Kollam
Sreenarayanaguru Open University
Headquarters Building
BSNL Bhavan, Vellayittambalam,
Kollam, Kerala, Pin : 691012
Ph: 9188912554
email:rckollam@sgou.ac.in

Tripunithura
Govt. College
Tirpunithura, Ernakulam
Kerala, Pin : 682301
Ph: 04842927436
email:rcedirector@sgou.ac.in

Thalassery
Govt. Brennen College
Dharmadam, Thalassery
Kannur, Pin : 670106
Ph: 04952990494
email:rctdirector@sgou.ac.in

Pattambi
Sree Neelakanta Govt. Sanskrit College
Pattambi, Palakkad
Kerala, Pin : 679303
Ph: 04662912009
email:rcpdirector@sgou.ac.in

Kozhikode
Govt. Arts and Science College
Meenchantha, Kozhikode,
Kerala, Pin : 673002
Ph: 04952920228
email:rckdirector@sgou.ac.in

SAY NO TO RAGGING

BEFORE YOU EVEN THINK OF RAGGING



Humiliation

Blacklisting

Ruined Career

Download

ANTI RAGGING

App

Don't just stand and watch. Stop Ragging! Show Character

Visit UGC Website i.e. www.ugc.Ac.in & www antinaging.in to UGC Anti Ragging regulations.

Ministry of Education
Government of India

Are You Being Ragged ?

विश्वविद्यालय अनुदान आयोग
University Grants Commission
quality higher education for all

DON'T LET IT BE TOO LATE

SAY NO TO DRUGS

LOVE YOURSELF AND ALWAYS BE HEALTHY



SREENARAYANAGURU OPEN UNIVERSITY

The State University for Education, Training and Research in Blended Format, Kerala

Web Programming

COURSE CODE : M26CA01SE

Sreenarayanaguru Open University
Kollam, Kerala Pin- 691601, email: info@sgou.ac.in, www.sgou.ac.in Ph: +91 474 2966841

ISBN 978-81-181574-6-5