SGOU
Higher Education for All

WEB PROGRAMMING LAB

COURSE CODE: M26CA01PC

Master of Computer Applications
Practical Core 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

SGOU

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 Lab

Course Code: M26CA01PC

Semester - I

SGOU

Practical Core Course
Postgraduate Programme
Master of Computer Applications
2026-27 Admission onwards
Self Learning Material

SREENARAYANAGURU
OPEN UNIVERSITY

SREENARAYANAGURU OPEN UNIVERSITY

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

SREENARAYANAGURU
OPEN UNIVERSITY

WEB PROGRAMMING LAB

Course Code: M26CA01PC
Semester- I
Practical Core Course
Master of Computer Applications

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.

Review and Edit

Sreerej 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

Jobin J.

Co-ordination

Head, School of ICS

SGOU

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

Edition
October 2026

Copyright
© Sreenarayanaguru Open University

ISBN 978-81-181574-7-2

9 788118 157472

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.m

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 democratize 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.

Warm regards,



Prof. (Dr.) Jagathy Raj V. P.
Vice Chancellor

SGOU

Contents

Cycle I	**Client-Side Web Programming**	**1**
Experiment 1	HTML5 Web Page Development	7
Experiment 2	CSS Styling and Responsive Layout Design	22
Experiment 3	Developing JavaScript Programs for Dynamic Web Pages	36
Experiment 4	Developing Interactive Web Pages Using DOM and Events	48
Experiment 5	Form Validation and Asynchronous Web Programming	
	Objectives	58
Cycle II	**Full-Stack Web Programming**	**70**
Experiment 1	Server-Side Application Development Using Node.js and Express	77
Experiment 2	RESTful API Development	84
Experiment 3	Database Integration and CRUD Operations	97
Experiment 4	Building User Interfaces Using React	114
Experiment 5	React Routing and Full-Stack Web Application Development	126

SGOU

CYCLE 1

Client-Side Web Programming



SGOU

Introduction

Client-side web programming refers to the development of webpages and web-based interfaces that are processed and displayed by a web browser. It enables developers to structure webpage content, apply visual styles, create dynamic behaviour, handle user interactions, validate user input, and retrieve data asynchronously.

This cycle introduces the fundamental technologies and concepts required for client-side web development. The experiments progress from creating structured webpages using HTML5 to styling and responsive design using CSS, programming dynamic webpages using JavaScript, manipulating webpage elements through the DOM, handling events, validating forms, and performing asynchronous operations.

The cycle consists of five experiments covering HTML5, CSS, JavaScript, DOM and events, and asynchronous web programming.

Objectives

1. Develop structured webpages using HTML5 elements.
2. Apply CSS to style webpages and create responsive layouts.
3. Implement JavaScript programs for dynamic webpage development.
4. Manipulate webpage elements using the DOM and handle user events.
5. Implement client-side form validation.
6. Perform asynchronous operations using JavaScript.
7. Process JSON data and retrieve dynamic data using the Fetch API.

Scope

This cycle provides hands-on experience in client-side web programming. Students will:

- Develop webpages using HTML5.
- Structure webpage content using semantic elements.
- Apply CSS styles and responsive layout techniques.
- Write JavaScript programs using fundamental and modern JavaScript features.
- Manipulate HTML elements dynamically using the DOM.

SGOU

- Handle mouse and keyboard events using event listeners.
- Implement client-side form validation.
- Work with timers, Promises, and async/await.
- Process JSON data.
- Retrieve and display dynamic data using the Fetch API.

Tools and Software

1. Code Editor

Visual Studio Code (VS Code) can be used to create and edit HTML, CSS, and JavaScript files. It provides syntax highlighting, code completion, and extensions that support web development.

2. Web Browser

A modern web browser is required to execute and test webpages developed during the laboratory sessions.

Examples include:

- Google Chrome
- Mozilla Firefox
- Microsoft Edge

The browser can also be used to inspect webpage elements and view JavaScript output during development.

3. Browser Developer Tools

Browser Developer Tools provide facilities for testing and debugging client-side webpages. They can be used to:

- Inspect HTML elements.
- Examine CSS styles.
- View JavaScript output.
- Identify errors in JavaScript programs.
- Observe webpage behaviour during user interaction.

4. HTML5

HTML5 is used to create and structure webpages using semantic elements, headings, lists, tables, images, hyperlinks, forms, input controls, and validation attributes.

SGOU

5. CSS

CSS (Cascading Style Sheets) is used to style webpages and create responsive layouts using selectors, colors, typography, backgrounds, the box model, Flexbox, CSS Grid, and media queries.

6. JavaScript

JavaScript is used to develop dynamic and interactive webpages. The laboratory uses JavaScript for variables, operators, control structures, functions, arrays, objects, arrow functions, template literals, DOM manipulation, event handling, form validation, and asynchronous programming.

7. Internet / Local Web Resources

An internet connection may be required when working with the Fetch API to retrieve dynamic data from web resources. Local HTML, CSS, and JavaScript files can also be executed directly in a web browser for the client-side experiments.

8. Live Server

Live Server is used to run HTML, CSS, and JavaScript files through a local development server. It automatically reloads the webpage when changes are saved and is particularly useful for experiments involving the Fetch API and JSON data.

General Procedure

Step 1: Create an Experiment Folder

Create a separate folder for each experiment.

```
WebProgrammingLab
│
└── Cycle_I
    ├── Experiment_1
    ├── Experiment_2
    ├── Experiment_3
    ├── Experiment_4
    └── Experiment_5
```

Keeping separate folders helps organize the programs and supporting files for each experiment.

Step 2: Open the Folder in VS Code

Open Visual Studio Code and select:

SGOU

File → Open Folder

Choose the folder corresponding to the experiment being performed.

Step 3: Create the Required Files

Create the files specified in the experiment.

For example:

Experiment_1

└── index.html

For an experiment involving separate CSS:

Experiment_2

├── index.html

└── style.css

For an experiment involving JSON data:

Experiment_5

├── index.html

└── destinations.json

Step 4: Enter and Save the Program

Enter the program in the appropriate file and save the files using:

Ctrl + S

Check that the file extensions are correct, such as .html, .css, .js, and .json.

Step 5: Run the Webpage

For basic client-side webpages, the HTML file can be opened directly in a web browser.

Alternatively, use Live Server from VS Code:

Right-click index.html → Open with Live Server

The webpage will open in the browser through a local address such as:

http://127.0.0.1:5500/

Step 6: Test the Program

Interact with the webpage and verify that the expected output is produced.

Depending on the experiment, test:

- HTML elements and forms

SGOU

- CSS styling and responsive layouts
- JavaScript-generated content
- DOM manipulation
- Mouse and keyboard events
- Form validation
- Asynchronous operations
- JSON data retrieval

Step 7: Debug Errors

If the expected output is not obtained, use the browser's Developer Tools.

Press:

F12

or:

Ctrl + Shift + I

The Console can be used to identify JavaScript errors, while the Elements section can be used to inspect HTML and CSS.

SGOU

Experiment 1

HTML5 Web Page Development

Objectives

- Understand the basic structure of an HTML5 document.
- Use HTML5 semantic elements to organize webpage content.
- Create webpages using headings, lists, tables, images, and hyperlinks.
- Develop HTML5 forms using different input controls.
- Apply HTML5 validation attributes to form controls.

1.1.1 Theory

HTML5 is the standard markup language used to structure content on webpages. It provides elements for organizing text, images, hyperlinks, tables, forms, and other webpage content.

An HTML5 webpage is composed of elements represented using tags. These elements define the structure and meaning of the content displayed by a web browser.

The basic structure of an HTML5 document is:

```
<!DOCTYPE html>
<html>
<head>
   <title>My Web Page</title>
</head>
<body>
   <h1>Welcome to My Web Page</h1>
   <p>This is an HTML5 webpage.</p>
</body>
</html>
```

The <!DOCTYPE html> declaration specifies that the document uses HTML5. The

SGOU

<html> element represents the complete HTML document. The <head> section contains information about the document, while the <body> contains the content displayed in the browser.

1.1.1.1 HTML5 Semantic Elements

Semantic elements provide meaningful information about the structure and purpose of webpage content. They help organize a webpage into clearly defined sections.

Common semantic elements include:

- <header> – represents introductory or header content.
- <nav> – represents navigation links.
- <main> – represents the main content of the webpage.
- <section> – represents a section of related content.
- <article> – represents independent content.
- <aside> – represents related or supplementary content.
- <footer> – represents footer information.

Example:

```
<header>
   <h1>Kerala Explorer</h1>
</header>
<nav>
   <a href="#">Home</a>
   <a href="#">Destinations</a>
</nav>
<main>
   <section>
      <h2>Popular Destinations</h2>
      <p>Explore popular destinations in Kerala.</p>
   </section>
```

SGOU

```
    <article>
        <h2>Munnar</h2>
        <p>Munnar is a popular hill station.</p>
    </article>
    <aside>
        <p>Explore more destinations.</p>
    </aside>
</main>
<footer>
    <p>Kerala Explorer</p>
</footer>
```

1.1.1.2 Headings

HTML provides six levels of headings, from <h1> to <h6>. Headings are used to organize webpage content.

```
<h1>Main Heading</h1>
<h2>Section Heading</h2>
<h3>Subsection Heading</h3>
<h4>Heading 4</h4>
<h5>Heading 5</h5>
<h6>Heading 6</h6>
```

The <h1> element represents the highest-level heading, while <h6> represents the lowest-level heading.

1.1.1.3 Lists

Lists are used to organize related information.

Ordered List

An ordered list displays items in a numbered sequence.

```
<ol>
    <li>Munnar</li>
```

SGOU

```
    <li>Wayanad</li>
    <li>Varkala</li>
</ol>
```

Unordered List

An unordered list displays items using bullet points.

```
<ul>
    <li>Munnar</li>
    <li>Wayanad</li>
    <li>Varkala</li>
</ul>
```

1.1.1.4 Tables

Tables are used to display information in rows and columns.

The main elements used to create a table are:

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

Example:

```
<table border="1">
    <tr>
        <th>Destination</th>
        <th>District</th>
        <th>Category</th>
    </tr>
    <tr>
        <td>Munnar</td>
```

SGOU

```
        <td>Idukki</td>
        <td>Hill Station</td>
    </tr>
    <tr>
        <td>Alappuzha</td>
        <td>Alappuzha</td>
        <td>Backwaters</td>
    </tr>
</table>
```

1.1.1.5 Images

The <img> element is used to display images on a webpage.

```
<img src="munnar.jpg" alt="Munnar Hills">
```

The src attribute specifies the location of the image, while the alt attribute provides alternative text for the image.

1.1.1.6 Hyperlinks

Hyperlinks are used to navigate from one webpage or resource to another.

The <a> element is used to create a hyperlink.

```
<a href="https://example.com">Visit Website</a>
```

The href attribute specifies the destination of the hyperlink.

1.1.1.7 Forms

HTML forms are used to collect information from users.

A form is created using the <form> element.

```
<form>
    <label for="name">Name:</label>
    <input type="text" id="name">
    <label for="email">Email:</label>
    <input type="email" id="email">
```

SGOU

```
    <input type="submit" value="Submit">
</form>
```

A form can contain different input controls depending on the type of information that needs to be collected.

1.1.1.8 Input Controls

HTML5 provides different input controls for collecting user information.

Common input types include:

```
<input type="text">
<input type="email">
<input type="password">
<input type="number">
<input type="date">
<input type="tel">
```

Radio buttons can be used when the user needs to select one option from a group.

```
<input type="radio" name="gender" value="male"> Male
<input type="radio" name="gender" value="female"> Female
```

Checkboxes can be used when multiple options can be selected.

```
<input type="checkbox" value="Munnar"> Munnar
<input type="checkbox" value="Wayanad"> Wayanad
```

A selection list can be created using <select> and <option>.

```
<select>
    <option>Munnar</option>
    <option>Wayanad</option>
    <option>Varkala</option>
</select>
```

A <textarea> can be used to collect multiline input.

```
<textarea rows="4" cols="40"></textarea>
```

SGOU

1.1.1.9 Validation Attributes

HTML5 provides validation attributes that help ensure that the information entered by the user satisfies specified conditions.

Important validation attributes include:

required

Specifies that the input field must be completed.

<input type=”text” required>

min

Specifies the minimum permitted value.

<input type=”number” min=”18”>

max

Specifies the maximum permitted value.

<input type=”number” max=”60”>

minlength

Specifies the minimum number of characters.

<input type=”text” minlength=”3”>

maxlength

Specifies the maximum number of characters.

<input type=”text” maxlength=”20”>

pattern

Specifies a pattern that the input must follow.

<input type=”text” pattern=”[A-Za-z]+”>

These attributes allow the browser to perform basic validation when a form is submitted.

1.1.2 Implementation

Develop an HTML5 webpage that presents information about tourist destinations and provides a form for collecting visitor information.

SGOU

The webpage should contain:

- Semantic HTML5 elements
- Headings
- Lists
- A table
- Images
- Hyperlinks
- A form
- Different input controls
- HTML5 validation attributes

Program

```
<!DOCTYPE html>
<html>
<head>
   <title>Kerala Explorer</title>
</head>
<body>
<header>
   <h1>Kerala Explorer</h1>
   <p>Explore the beautiful destinations of Kerala</p>
</header>
<nav>
   <a href="#destinations">Destinations</a> |
   <a href="#places">Popular Places</a> |
   <a href="#registration">Visitor Registration</a>
</nav>
<main>
```

SGOU

```
<section id=”destinations”>
    <h2>Popular Destinations</h2>
    <ul>
        <li>Munnar</li>
        <li>Wayanad</li>
        <li>Alappuzha</li>
        <li>Varkala</li>
    </ul>
</section>
<section>
    <h2>Travel Activities</h2>
    <ol>
        <li>Explore hill stations</li>
        <li>Visit the backwaters</li>
        <li>Enjoy the beaches</li>
    </ol>
</section>
<section id=”places”>
    <h2>Destination Details</h2>
    <table border=”1”>
        <tr>
            <th>Destination</th>
            <th>District</th>
            <th>Category</th>
        </tr>
        <tr>
            <td>Munnar</td>
            <td>Idukki</td>
```

SGOU

```
            <td>Hill Station</td>
        </tr>
        <tr>
            <td>Wayanad</td>
            <td>Wayanad</td>
            <td>Hill Station</td>
        </tr>
        <tr>
            <td>Alappuzha</td>
            <td>Alappuzha</td>
            <td>Backwaters</td>
        </tr>
        <tr>
            <td>Varkala</td>
            <td>Thiruvananthapuram</td>
            <td>Beach</td>
        </tr>
    </table>
</section>
<section>
    <h2>Munnar</h2>
    <img src="munnar.jpg"
        alt="Munnar Hills"
        width="300">
    <p>
        Munnar is a popular hill station known for its scenic landscapes.
    </p>
    <a href="https://www.keralatourism.org/">
        Visit Kerala Tourism
```

SGOU

```
    </a>
</section>
<section id=”registration”>
    <h2>Visitor Registration</h2>
    <form>
        <label for=”name”>Name:</label>
        <input type=”text”
            id=”name”
            name=”name”
            required
            minlength=”3”
            maxlength=”30”>
        <br><br>
        <label for=”email”>Email:</label>
        <input type=”email”
            id=”email”
            name=”email”
            required>
        <br><br>
        <label for=”phone”>Phone:</label>
        <input type=”tel”
            id=”phone”
            name=”phone”
            required>
        <br><br>
        <label for=”age”>Age:</label>
        <input type=”number”
            id=”age”
            name=”age”
```

SGOU

```
        min="18"
        max="60"
        required>
    <br><br>
    <label for="date">Travel Date:</label>
    <input type="date"
        id="date"
        name="date"
        required>
    <br><br>
    <p>Gender:</p>
    <input type="radio"
        id="male"
        name="gender"
        value="male">
    <label for="male">Male</label>
    <input type="radio"
        id="female"
        name="gender"
        value="female">
    <label for="female">Female</label>
    <br><br>
    <p>Preferred Destinations:</p>
    <input type="checkbox"
        id="munnar"
        name="destination"
        value="Munnar">
    <label for="munnar">Munnar</label>
```

SGOU

```
        <input type="checkbox"
            id="wayanad"
            name="destination"
            value="Wayanad">
        <label for="wayanad">Wayanad</label>
        <input type="checkbox"
            id="alappuzha"
            name="destination"
            value="Alappuzha">
        <label for="alappuzha">Alappuzha</label>
        <br><br>
        <label for="comments">Comments:</label>
        <br>
        <textarea id="comments"
            name="comments"
            rows="4"
            cols="40"></textarea>
        <br><br>
        <input type="submit" value="Register">
        <input type="reset" value="Reset">
      </form>
    </section>
</main>
<footer>
    <p>&copy; 2026 Kerala Explorer</p>
</footer>
</body>
</html>
```

SGOU

Output

Explore the beautiful destinations of Kerala

Destinations | Popular Places | Visitor Registration

Popular Destinations

- Munnar
- Wayanad
- Alappuzha
- Varkala

Travel Activities

1. Explore hill stations
2. Visit the backwaters
3. Enjoy the beaches

Destination Details

Destination	District	Category
Munnar	Idukki	Hill Station
Wayanad	Wayanad	Hill Station
Alappuzha	Alappuzha	Backwaters
Varkala	Thiruvananthapuram	Beach

Munnar



Munnar is a popular hill station known for its scenic landscapes.

Visit Kerala Tourism

Visitor Registration

Name:

Email:

Phone:

Age:

Travel Date: dd-mm-yyyy

Gender:

Male Female

Preferred Destinations:

Munnar Wayanad Alappuzha

Comments:

Register Reset

© 2026 Kerala Explorer

SGOU

1.1.3 Lab Practice Questions

1. Develop an HTML5 webpage using semantic elements and different levels of headings.

2. Create an HTML5 webpage to display a list of items using ordered and unordered lists.

3. Develop an HTML5 webpage to display information using a table.

4. Create an HTML5 webpage containing an image and hyperlinks.

5. Develop an HTML5 form using different input controls such as text, email, number, date, radio buttons, checkboxes, selection lists, and text areas.

6. Apply appropriate HTML5 validation attributes to the form controls.

7. Develop an HTML5 webpage that combines semantic elements, headings, lists, tables, images, hyperlinks, and a form with appropriate input controls and validation attributes.

SGOU

Experiment 2

CSS Styling and Responsive Layout Design

Objectives

- Understand the basic concepts and purpose of CSS.
- Apply CSS selectors to HTML elements.
- Apply colors, typography, and backgrounds to webpage elements.
- Apply the CSS box model to control the dimensions and spacing of elements.
- Design webpage layouts using Flexbox.
- Design webpage layouts using CSS Grid.
- Develop responsive layouts using media queries.

SGOU

1.2.1 Theory

1.2.1.1 Introduction to CSS

CSS stands for Cascading Style Sheets. It is used to control the presentation and visual appearance of HTML webpages. While HTML provides the structure of a webpage, CSS is used to define how that structure is displayed.

CSS can be used to specify colors, fonts, spacing, dimensions, backgrounds, and layouts.

A CSS rule generally consists of a selector and a declaration block.

```
selector {
    property: value;
}
```

Example:

```
h1 {
    color: blue;
    font-size: 30px;
}
```

Here, h1 is the selector, color and font-size are properties, and blue and 30px are their corresponding values.

1.2.1.2 CSS Selectors

Selectors are used to identify the HTML elements to which CSS styles should be applied.

Element Selector

An element selector selects HTML elements based on their tag name.

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

The style is applied to all <p> elements.

Class Selector

A class selector is used to select elements having a specific class.

```
.destination {
    color: green;
}
```

The class is applied in HTML using the class attribute.

```
<p class="destination">Munnar</p>
```

ID Selector

An ID selector selects an element using its unique id.

```
#title {
    text-align: center;
}
```

Example:

```
<h1 id="title">Kerala Explorer</h1>
```

1.2.1.3 Colors

CSS provides properties for applying colors to webpage elements.

The color property specifies the color of text.

```
p {
    color: blue;
```

SGOU

```
}
```

The background-color property specifies the background color of an element.

```
section {
    background-color: lightgray;
}
```

Colors can be specified using names, hexadecimal values, RGB values, and other supported color representations.

1.2.1.4 Typography

Typography refers to the appearance and arrangement of text on a webpage.

Common CSS properties include:

- font-family
- font-size
- font-weight
- font-style
- text-align

Example:

```
h1 {
    font-family: Arial, sans-serif;
    font-size: 32px;
    font-weight: bold;
    text-align: center;
}
```

These properties can be used to control the appearance of headings and other textual content.

1.2.1.5 Backgrounds

CSS provides properties for controlling the background of HTML elements.

Example:

```
body {
 background-color: #f2f2f2;
```

SGOU

```
}
```

A background image can also be specified using the background-image property.

```
section {
    background-image: url("background.jpg");
}
```

Background properties can be applied to individual webpage elements.

1.2.1.6 CSS Box Model

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

Every element can be considered as a box consisting of:

1. Content
2. Padding
3. Border
4. Margin

The structure can be represented as:

Content → Padding → Border → Margin

Content

The content area contains the actual text or other content of the element.

Padding

Padding is the space between the content and the border.

```
section {
    padding: 20px;
}
```

Border

The border surrounds the content and padding.

```
section {
    border: 1px solid black;
}
```

SGOU

Margin

Margin specifies the space outside the border.

```
section {
    margin: 20px;
}
```

Example:

```
.card {
    width: 300px;
    padding: 20px;
    border: 2px solid black;
    margin: 15px;
}
```

1.2.1.7 Flexbox

Flexbox, or the Flexible Box Layout, is a CSS layout model used to arrange elements efficiently within a container.

A flex container is created by applying:

```
.container {
    display: flex;
}
```

Example:

```
<div class="container">
    <div>Munnar</div>
    <div>Wayanad</div>
    <div>Varkala</div>
</div>
.container {
    display: flex;
    gap: 20px;
}
```

SGOU

Flexbox provides properties for controlling the direction, alignment, and distribution of elements.

Important properties include:

- flex-direction
- justify-content
- align-items
- gap
- flex-wrap

Example:

```
.container {
    display: flex;
    justify-content: space-between;
    align-items: center;
    gap: 20px;
}
```

1.2.1.8 CSS Grid

CSS Grid is a two-dimensional layout system used to arrange elements in rows and columns.

A grid container is created using:

```
.container {
    display: grid;
}
```

Columns can be defined using grid-template-columns.

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

SGOU

Example:

```
<div class=”container”>
    <div>Munnar</div>
    <div>Wayanad</div>
    <div>Alappuzha</div>
</div>
```

Grid is useful when webpage elements need to be arranged in a structured row-and-column layout.

1.2.1.9 Media Queries

Media queries allow CSS styles to be applied according to characteristics of the device or viewport.

They are commonly used to create responsive webpages.

Example:

```
@media (max-width: 600px) {
    .container {
        grid-template-columns: 1fr;
    }
}
```

The style inside the media query is applied when the viewport width is 600 pixels or less.

Media queries allow layouts to adapt to different screen sizes.

1.2.2 Implementation

Develop a Styled and Responsive Webpage Using CSS

Develop a webpage for displaying tourist destinations. Apply CSS selectors, colors, typography, backgrounds, the box model, Flexbox, CSS Grid, and media queries to create a responsive layout.

SGOU

HTML

```
<!DOCTYPE html>
<html>
<head>
<title>Kerala Explorer</title>
<link rel=”stylesheet” href=”style.css”>
</head>
<body>
<header>
<h1>Kerala Explorer</h1>
<p>Explore the beauty of Kerala</p>
</header>
<nav>
<a href=”#”>Home</a>
<a href=”#”>Destinations</a>
<a href=”#”>Contact</a>
</nav>
<main>
<section class=”intro”>
<h2>Popular Destinations</h2>
<p>Discover some of the popular destinations in Kerala.</p>
</section>
<section class=”destination-grid”>
<div class=”card”>
<h3>Munnar</h3>
<p>A beautiful hill station known for its scenic landscapes.</p>
</div>
```

SGOU

```
        <div class=”card”>
            <h3>Wayanad</h3>
            <p>A destination known for its hills, forests and wildlife.</p>
        </div>
        <div class=”card”>
            <h3>Alappuzha</h3>
            <p>Known for its backwaters and houseboats.</p>
        </div>
        <div class=”card”>
            <h3>Varkala</h3>
            <p>A coastal destination known for its beach and cliffs.</p>
        </div>
    </section>
</main>
<footer>
    <p>&copy; 2026 Kerala Explorer</p>
</footer>
</body>
</html>
```

CSS

```
* {
    box-sizing: border-box;
}
body {
    margin: 0;
    font-family: Arial, sans-serif;
    background-color: #f2f2f2;
```

SGOU

```
    color: #333333;
}
header {
    background-color: #2e7d32;
    color: white;
    text-align: center;
    padding: 30px;
}

header h1 {
    font-size: 32px;
}
nav {
    display: flex;
    justify-content: center;
    gap: 30px;
    background-color: #333333;
    padding: 15px;
}
nav a {
    color: white;
    text-decoration: none;
}
.intro {
    margin: 20px;
    padding: 20px;
    border: 1px solid #cccccc;
```

SGOU

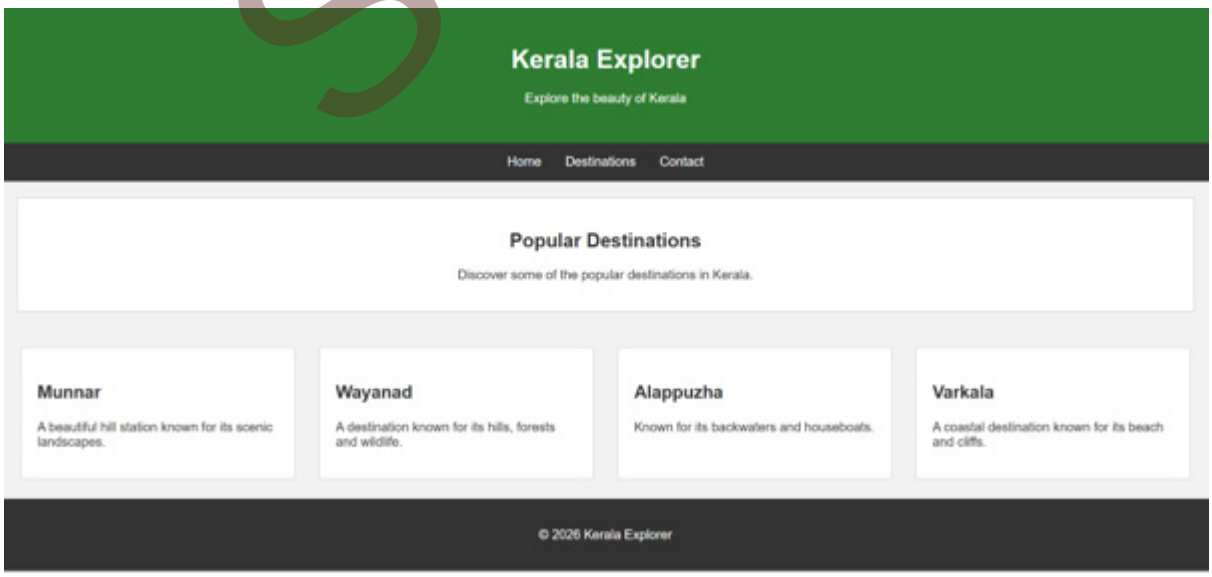
```
    background-color: white;
    text-align: center;
}
.destination-grid {
    display: grid;
    grid-template-columns: repeat(4, 1fr);
    gap: 20px;
    padding: 20px;
}
.card {
    background-color: white;
    padding: 20px;
    border: 1px solid #cccccc;
    margin: 5px;
}
.card h3 {
    font-size: 22px;
}
footer {
    text-align: center;
    padding: 20px;
    background-color: #333333;
    color: white;
}
@media (max-width: 768px) {
    .destination-grid {
```

SGOU

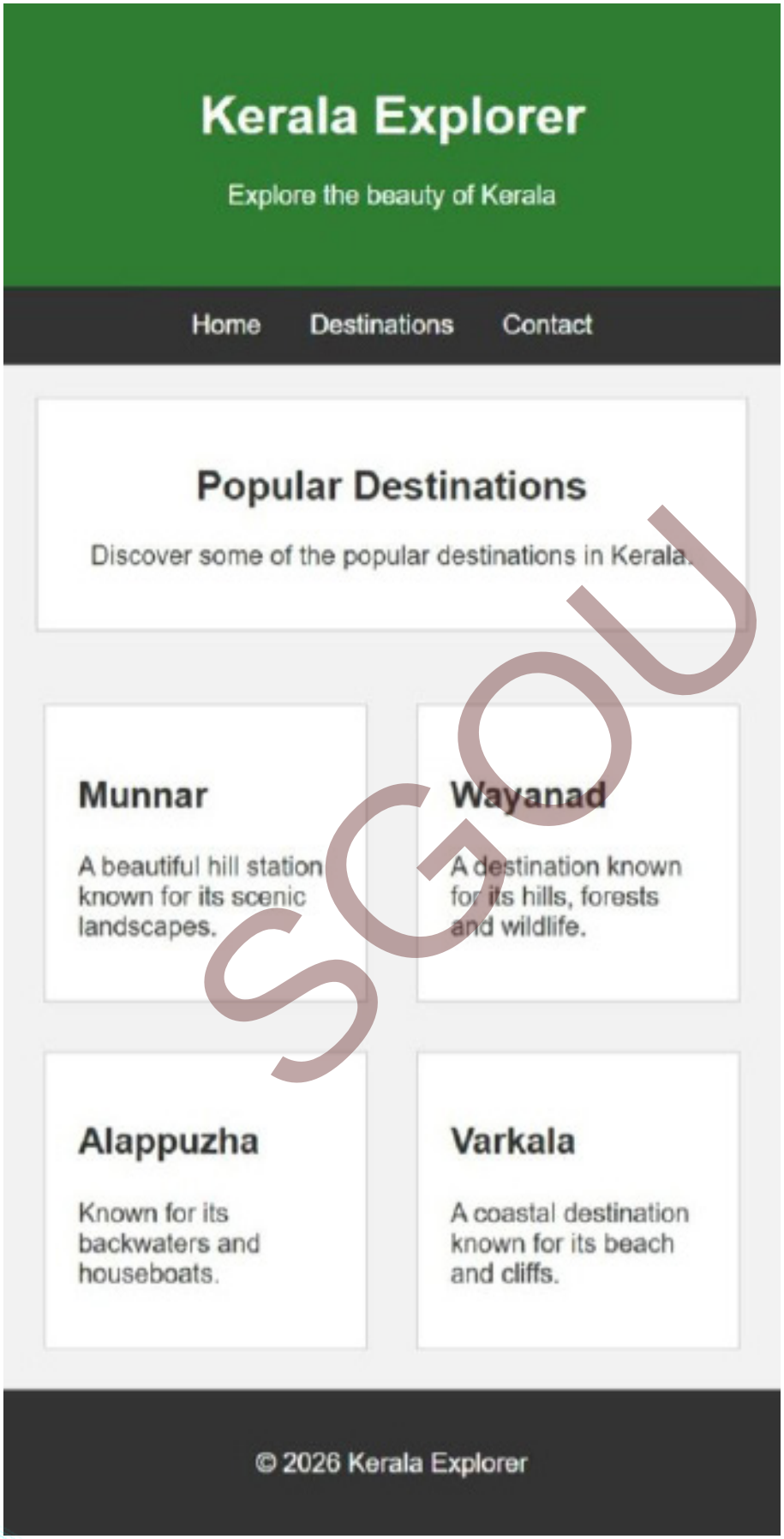
```
        grid-template-columns: repeat(2, 1fr);
    }
    nav {
        flex-wrap: wrap;
    }
}
@media (max-width: 480px) {

    .destination-grid {
        grid-template-columns: 1fr;
    }
    header h1 {
        font-size: 24px;
    }
}
```

Output



SGOU


Kerala Explorer
Explore the beauty of Kerala
Home
Destinations
Contact
Popular Destinations
Discover some of the popular destinations in Kerala.
Munnar
A beautiful hill station known for its scenic landscapes.
Wayanad
A destination known for its hills, forests and wildlife.
Alappuzha
Known for its backwaters and houseboats.
Varkala
A coastal destination known for its beach and cliffs.
© 2026 Kerala Explorer

SGOU

1.2.3 Lab Practice Questions

1. Apply CSS selectors to style different elements of an HTML webpage.
2. Design a webpage using CSS colors, typography, and backgrounds.
3. Create a webpage demonstrating the CSS box model using content, padding, border, and margin.
4. Design a webpage layout using Flexbox.
5. Design a webpage layout using CSS Grid with multiple rows and columns.
6. Develop a responsive webpage using CSS media queries.
7. Develop a webpage that combines CSS selectors, colors, typography, backgrounds, the box model, Flexbox, CSS Grid, and media queries.

SGOU

Experiment 3

Developing JavaScript Programs for Dynamic Web Pages

Objectives

- Understand the use of variables and data values in JavaScript.
- Apply different operators to perform computations and comparisons.
- Implement decision-making and repetitive operations using control structures.
- Define and use functions in JavaScript programs.
- Create and manipulate arrays to store collections of values.
- Create and access objects to represent structured data.
- Implement modern JavaScript features such as arrow functions and template literals.

SGOU

1.3.1 Theory

1.3.1.1 Introduction to JavaScript

JavaScript is a programming language used to add dynamic behaviour to webpages. It can be used to perform calculations, process data, implement decision-making, repeat operations, and organize information using arrays and objects.

JavaScript code can be included in an HTML document using the <script> element.

```
<script>
    let name = “Munnar”;
    console.log(name);
</script>
```

JavaScript programs can also be written in a separate .js file and connected to an HTML document.

```
<script src=”script.js”></script>
```

1.3.1.2 Variables

Variables are used to store values in a JavaScript program.

JavaScript provides the keywords let, const, and var for declaring variables.

1. let

let is used to declare a variable whose value can be changed.

```
let age = 20;
age = 21;
```

2. const

const is used to declare a variable whose binding cannot be reassigned.

```
const college = "ABC College";
```

3. var

var is an older method of declaring variables.

var marks = 85;

Example:

```
let name = "Anu";
let age = 20;
const course = "MCA";
console.log(name);
console.log(age);
console.log(course);
```

SGOU

1.3.1.3 Operators

Operators are symbols used to perform operations on values.

1. Arithmetic Operators

Arithmetic operators perform mathematical operations.

```
let a = 10;
let b = 5;
console.log(a + b);
console.log(a - b);
```

```
console.log(a * b);
console.log(a / b);
console.log(a % b);
```

Common arithmetic operators include:

- + Addition
- - Subtraction
- * Multiplication
- / Division
- % Modulus

2. Relational Operators

Relational operators compare values.

```
let a = 10;
let b = 20;
console.log(a > b);
console.log(a < b);
console.log(a >= b);
console.log(a <= b);
```

3. Logical Operators

Logical operators are used to combine conditions.

```
let age = 25;
let hasID = true;
console.log(age >= 18 && hasID);
```

Common logical operators are:

- && AND
- || OR
- ! NOT

4. Assignment Operators

Assignment operators are used to assign or update values.

SGOU

```
let marks = 50;
marks + = 10;
marks - = 5;
```

1.3.1.4 Control Structures

Control structures determine the order in which statements are executed.

1. Conditional Statement

The if statement executes a block when a condition is true.

```
let marks = 75;
if (marks >= 50) {
    console.log("Pass");
}
```

a. if-else

```
let marks = 45;
if (marks >= 50) {
    console.log("Pass");
} else {
    console.log("Fail");
}
```

b. else-if

Multiple conditions can be evaluated using else if.

```
let marks = 85;
if (marks >= 90) {
    console.log("A+");
} else if (marks >= 80) {
    console.log("A");
} else if (marks >= 70) {
```

SGOU

```
    console.log("B");
} else {
    console.log("C");
}
```

c. switch

The switch statement can be used when a value needs to be compared against multiple cases.

```
let choice = 2;
switch (choice) {
    case 1:
        console.log("Home");
        break;
    case 2:
        console.log("Destinations");
        break;
    default:
        console.log("Invalid choice");
}
```

SGOU

2. Loops

Loops are used to execute a block of statements repeatedly.

a. for Loop

```
for (let i = 1; i <= 5; i++) {
    console.log(i);
}
```

b. while Loop

```
    let i = 1;
while (i <= 5) {
    console.log(i);
    i++;
}
```

c. do-while Loop

```
let i = 1;
do {
    console.log(i);
    i++;
} while (i <= 5);
```

1.3.1.5 Functions

A function is a reusable block of code designed to perform a specific task.

A function can be defined using the function keyword.

```
function add(a, b) {
    return a + b;
}
let result = add(10, 20);
console.log(result);
```

Here, a and b are parameters, while the values supplied during the function call are arguments.

Functions improve code organization and allow the same operation to be performed multiple times.

1.3.1.6 Arrays

An array is used to store multiple values in a single variable.

SGOU

```
let destinations = [“Munnar”, “Wayanad”, “Varkala”];
console.log(destinations[0]);
console.log(destinations[1]);
Array indexing begins from 0.
// Array elements can be accessed and modified using their index.
destinations[1] = “Alappuzha”;
Arrays can also be processed using loops.
for (let i = 0; i < destinations.length; i++) {
    console.log(destinations[i]);
}
```

1.3.1.7 Objects

Objects are used to represent related information using properties and values.

```
let destination = {
    name: “Munnar”,
    district: “Idukki”,
    type: “Hill Station”
};
console.log(destination.name);
console.log(destination.district);
// Properties can also be accessed using bracket notation.
console.log(destination[“name”]);
```

Objects are useful for representing structured information.

1.3.1.8 Arrow Functions

Arrow functions provide a shorter syntax for defining functions.

A traditional function can be written as:

SGOU

```
function add(a, b) {
    return a + b;
}
```

The same function can be written using an arrow function:

```
const add = (a, b) => {
    return a + b;
};
```

For a simple expression, the function can be written more concisely:

```
const add = (a, b) => a + b;
```

1.3.1.9 Template Literals

Template literals provide a convenient way to create strings containing variables or expressions.

They are enclosed using backticks `.

```
let name = "Anu";
let age = 20;
let message = `Name: ${name}, Age: ${age}`;
console.log(message);
```

The ${} syntax is used to insert variable values or expressions into a template literal.

1.3.2 Implementation

1. Develop a JavaScript program that processes information about tourist destinations and demonstrates the JavaScript concepts covered in this experiment.

```
<!DOCTYPE html>
<html>
<head>
    <title>JavaScript Dynamic Webpage</title>
```

SGOU

```
</head>
<body>
    <h1>Kerala Explorer</h1>
    <div id=”output”></div>
    <script>
        // Variables
        let destination = “Munnar”;
        let district = “Idukki”;
        let adults = 80;
        let children = 40;
        // Operators
        let totalVisitors = adults + children;
        let averageVisitors = totalVisitors / 2;
        // Control Structure
        let category;
        if (totalVisitors >= 100) {
            category = “Popular Destination”;
        } else {
            category = “Less Visited Destination”;
        }
        // Function
        function calculateTotal(a, b) {
            return a + b;
        }
        let total = calculateTotal(adults, children);
        // Array
        let destinations = [
```

SGOU

```
    "Munnar",
    "Wayanad",
    "Alappuzha",
    "Varkala"
];
// Object
let place = {
    name: destination,
    district: district,
    type: "Hill Station"
};
// Arrow Function
const calculateAverage = (a, b) => (a + b) / 2;
let average = calculateAverage(adults, children);
// Control Structure with Loop
let destinationList = "";
for (let i = 0; i < destinations.length; i++) {
    destinationList += `<li>${destinations[i]}</li>`;
}
// Template Literal
let output = `
    <h2>${place.name}</h2>
    <p><strong>District:</strong> ${place.district}</p>
    <p><strong>Type:</strong> ${place.type}</p>
    <p><strong>Adults:</strong> ${adults}</p>
    <p><strong>Children:</strong> ${children}</p>
```

SGOU

```
        <p><strong>Total Visitors:</strong> ${total}</p>
        <p><strong>Average:</strong> ${average}</p>
        <p><strong>Category:</strong> ${category}</p>
        <h3>Other Destinations</h3>
        <ul>
          ${destinationList}
        </ul>
      `;
      // Display the generated content
      document.getElementById("output").innerHTML = output;
    </script>
  </body>
  </html>
```

Output

Kerala Explorer

Munnar

District: Idukki

Type: Hill Station

Adults: 80

Children: 40

Total Visitors: 120

Average: 60

Category: Popular Destination

Other Destinations

- Munnar
- Wayanad
- Alappuzha
- Varkala

SGOU

1.3.3 Lab Practice Questions

1. Write a JavaScript program to declare variables and perform arithmetic and relational operations on two numbers.

2. Write a JavaScript program using conditional statements to determine the result of a student based on marks.

3. Write a JavaScript program using loops to display a sequence of numbers.

4. Write a JavaScript program using a function to calculate the total and average of a set of values.

5. Write a JavaScript program to store and display a list of tourist destinations using an array.

6. Write a JavaScript program to store and display the details of a tourist destination using an object.

7. Rewrite a given JavaScript function using arrow function syntax and display the result using a template literal.

8. Develop a JavaScript program that combines variables, operators, control structures, functions, arrays, objects, arrow functions, and template literals.

SGOU

Experiment 4

Developing Interactive Web Pages Using DOM and Events

Objectives

- Understand the Document Object Model (DOM).
- Access and select HTML elements using JavaScript.
- Dynamically modify HTML elements and their content.
- Dynamically modify element properties and styles.
- Create interactive webpages using JavaScript.
- Handle mouse events using event listeners.
- Handle keyboard events using event listeners.

1.4.1 Theory

1.4.1.1 Document Object Model

The Document Object Model (DOM) represents an HTML document as a hierarchical structure of objects. Each HTML element in the webpage is represented as an object that can be accessed and manipulated using JavaScript.

For example, the following HTML:

```
<body>
  <h1>Kerala Explorer</h1>
  <p>Explore Kerala.</p>
</body>
```

can be represented conceptually as:

Document

```
|
└── HTML
    |
    └── Body
        ├── H1
        └── P
```

SGOU

JavaScript can use the DOM to access these elements and dynamically change their content, properties, and styles.

1.4.1.2 Selecting HTML Elements

JavaScript provides methods for selecting HTML elements.

getElementById()

Selects an element using its id.

let heading = document.getElementById(“title”);

getElementsByClassName()

Selects elements using their class name.

let cards = document.getElementsByClassName(“card”);

getElementsByTagName()

Selects elements using their tag name.

let paragraphs = document.getElementsByTagName(“p”);

querySelector()

Selects the first element that matches a CSS selector.

let heading = document.querySelector(“h1”);

querySelectorAll()

Selects all elements that match a CSS selector.

let paragraphs = document.querySelectorAll(“p”);

1.4.1.3 Modifying HTML Content

The innerHTML property can be used to read or modify the HTML content of an element.

Example:

<h2 id=”title”>Welcome</h2>

document.getElementById(“title”).innerHTML =

“Welcome to Kerala Explorer”;

The text displayed by the heading is changed dynamically.

SGOU

1.4.1.4 Modifying Element Properties

JavaScript can be used to modify the properties or attributes of HTML elements.

For example, the src property of an image can be changed dynamically.

```
document.getElementById("placeImage").src =
   "wayanad.jpg";
```

Similarly, attributes can be modified using methods such as setAttribute().

```
document.getElementById("placeImage")
   .setAttribute("alt", "Wayanad");
```

1.4.1.5 Modifying Styles

JavaScript can dynamically change the CSS properties of HTML elements.

Example:

```
document.getElementById("title").style.fontSize =
   "32px";
document.getElementById("title").style.color =
   "blue";
```

CSS classes can also be modified using the classList property.

```
element.classList.add("highlight");
element.classList.remove("highlight");
```

1.4.1.6 Events

An **event** is an action or occurrence that can be detected by a webpage.

Examples include:

- Mouse click
- Mouse movement
- Mouse entering an element
- Key press
- Key release

JavaScript can respond to these events and perform appropriate actions.

SGOU

1.4.1.7 Event Listeners

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

Syntax:

element.addEventListener("event", function);

Example:

```
document.getElementById("button")
   .addEventListener("click", function() {
      alert("Button clicked");
   });
```

When the button is clicked, the specified function is executed.

1.4.1.8 Mouse Events

Mouse events occur when the user interacts with webpage elements using a mouse.

Common mouse events include:

- click
- dblclick
- mouseover
- mouseout
- mousedown
- mouseup

Example:

```
document.getElementById("place")
   .addEventListener("mouseover", function() {
      this.style.fontSize = "25px";
   });
```

The event listener responds when the mouse pointer moves over the element.

1.4.1.9 Keyboard Events

Keyboard events occur when the user interacts with the webpage using the keyboard.

SGOU

Common keyboard events include:

keydown

keyup

Example:

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

The event.key property identifies the key pressed by the user.

1.4.2 Implementation

Develop interactive web pages by dynamically manipulating HTML elements and handling mouse and keyboard events using event listeners.

Develop an interactive webpage for displaying tourist destination information. Use JavaScript to dynamically modify HTML content and styles and handle mouse and keyboard events using event listeners.

Program

```
<!DOCTYPE html>
<html>
<head>
   <title>Interactive Kerala Explorer</title>
   <style>
      body {
         font-family: Arial, sans-serif;
         text-align: center;
      }
      #place {
         width: 400px;
         margin: 20px auto;
```

SGOU

```
            padding: 20px;
        border: 1px solid black;
    }
    #placeImage {
        width: 250px;
    }
    .highlight {
        background-color: lightblue;
        font-size: 22px;
    }
    input {
        padding: 8px;
        margin: 10px;
    }
    button {
        padding: 10px 20px;
    }
  </style>
</head>
<body>
  <h1 id="title">Kerala Explorer</h1>
  <div id="place">
    <h2 id="placeName">Munnar</h2>
    <img id="placeImage"
        src="munnar.jpg"
        alt="Munnar">
    <p id="description">
```

SGOU

```
        Munnar is a beautiful hill station in Kerala.
    </p>
    <button id=”changeButton”>
        Change Destination
    </button>
</div>
<p id=”message”>
    Move the mouse over the destination.
</p>
<label for=”name”>Enter your name:</label>
<input type=”text”
       id=”name”
       placeholder=”Type your name”>
<p id=”keyboardMessage”></p>
<script>
    // Selecting HTML elements
    const place = document.getElementById(“place”);
    const placeName = document.getElementById(“placeName”);
    const placeImage = document.getElementById(“placeImage”);
    const description = document.getElementById(“description”);
    const message = document.getElementById(“message”);
    const button = document.getElementById(“changeButton”);
    const nameInput = document.getElementById(“name”);
    const keyboardMessage =
        document.getElementById(“keyboardMessage”);
    // Click event
```

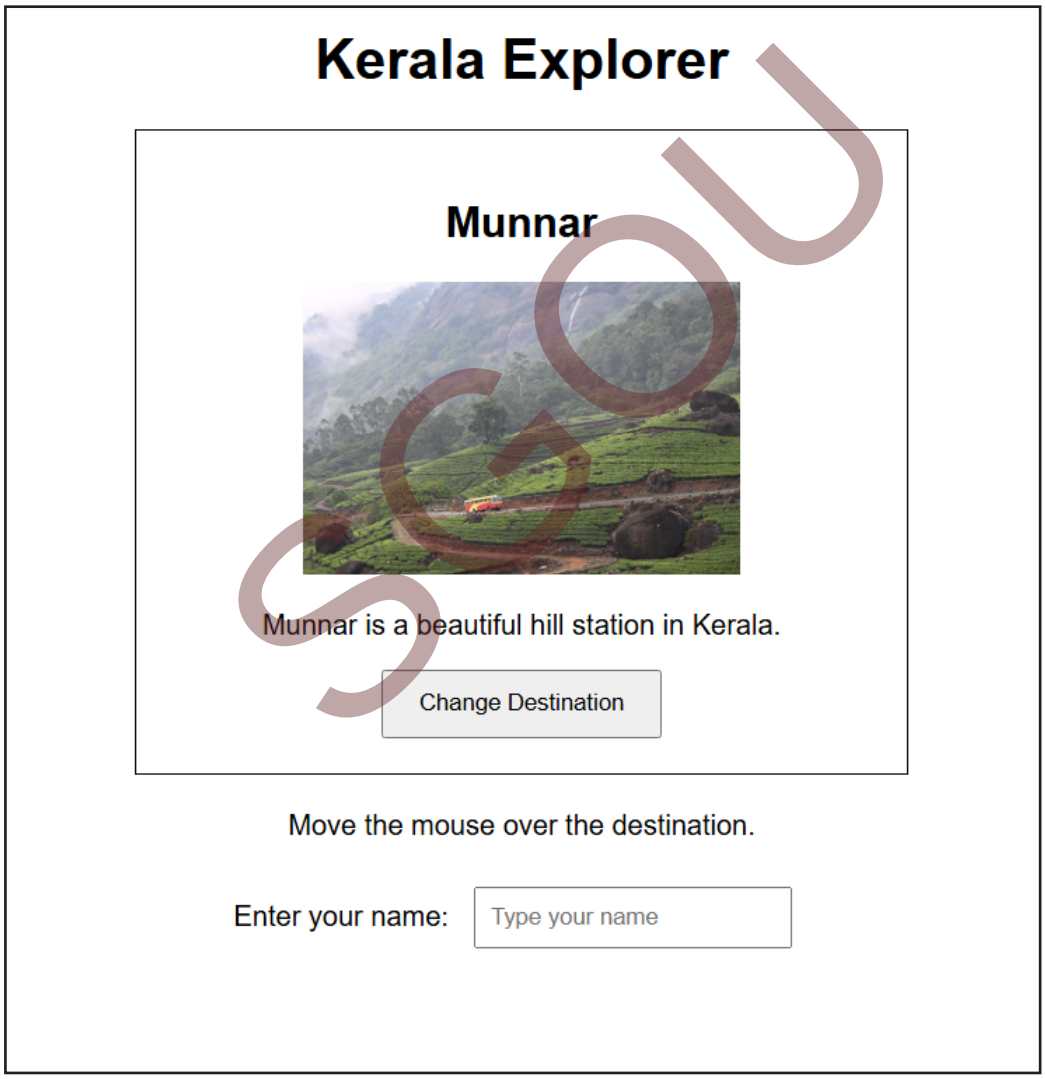
SGOU

```
button.addEventListener("click", function() {
    placeName.innerHTML = "Wayanad";
    description.innerHTML =
        "Wayanad is known for its hills, forests and wildlife.";
    placeImage.src = "wayanad.jpg";
    placeImage.setAttribute(
        "alt",
        "Wayanad"
    );
    message.innerHTML =
        "Destination changed to Wayanad.";
});
// Mouseover event
place.addEventListener("mouseover", function() {
    place.classList.add("highlight");
    message.innerHTML =
        "You are viewing a destination.";
});
// Mouseout event
place.addEventListener("mouseout", function() {
    place.classList.remove("highlight");
    message.innerHTML =
        "Move the mouse over the destination.";
});
// Keyboard event
nameInput.addEventListener("keydown", function(event) {
```

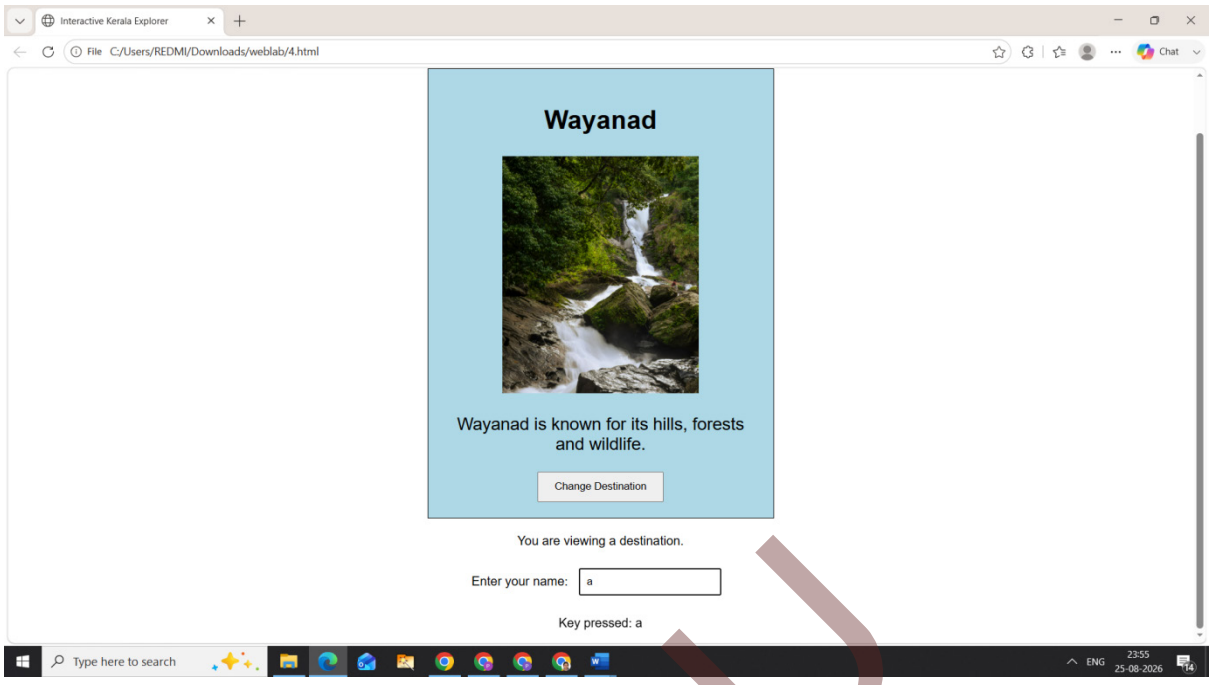
SGOU

```
            keyboardMessage.innerHTML =
                `Key pressed: ${event.key}`;
        });
    </script>
</body>
</html>
```

Output



SCOU



1.4.3 Lab Practice Questions

1. Develop a webpage that dynamically changes the content of an HTML element when a button is clicked.

2. Develop a webpage that changes an image and its properties using JavaScript.

3. Create an interactive webpage that changes the style of an element when the mouse pointer moves over it.

4. Develop a webpage that responds to click, mouseover, and mouseout events using event listeners.

5. Develop a webpage that displays the key pressed by the user using a keyboard event listener.

6. Create an interactive webpage that combines DOM manipulation with mouse and keyboard events.

SGOU

Experiment 5

Form Validation and Asynchronous Web Programming

Objectives

- Implement client-side form validation using JavaScript.
- Use timers to execute JavaScript operations after a specified interval.
- Understand and implement Promises for asynchronous operations.
- Use async and await to handle asynchronous operations.
- Process data represented in JSON format.
- Use the Fetch API to retrieve data from a web resource.
- Retrieve and display dynamic data on a webpage.

1.5.1 Theory

1.5.1.1 Client-Side Form Validation

Client-side form validation checks user input in the browser before the form data is submitted.

JavaScript can be used to check whether the entered values satisfy specified conditions.

Consider the following form:

```
<form id=”registrationForm”>
   <input type=”text” id=”name”>
   <input type=”email” id=”email”>
   <button type=”submit”>Submit</button>
</form>
```

JavaScript can be used to validate the form:

```
document.getElementById(“registrationForm”)
   .addEventListener(“submit”, function(event) {
      let name = document.getElementById(“name”).value;
```

SGOU

```
    if (name === "") {
        event.preventDefault();
        alert("Name is required.");
    }
});
```

The preventDefault() method prevents the normal form submission when the validation condition is not satisfied.

1.5.1.2 Timers

JavaScript provides timers for executing code after a specified period or repeatedly at a specified interval.

setTimeout()

setTimeout() executes a function after a specified delay.

```
setTimeout(function() {
    console.log("Message displayed after 2 seconds");
}, 2000);
```

The delay is specified in milliseconds.

setInterval()

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

```
setInterval(function() {
    console.log("Message displayed repeatedly");
}, 1000);
```

A timer can be stopped using clearInterval().

1.5.1.3 Promises

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

A Promise can have three states:

- **Pending** – the operation is still in progress.
- **Fulfilled** – the operation completed successfully.
- **Rejected** – the operation failed.

SGOU

Example:

```
let result = new Promise(function(resolve, reject) {
    let success = true;
    if (success) {
        resolve("Operation successful");
    } else {
        reject("Operation failed");
    }
});
result
    .then(function(message) {
        console.log(message);
    })
    .catch(function(error) {
        console.log(error);
    });
```

The then() method handles a successful result, while catch() handles an error.

1.5.1.4 Async and Await

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

An async function always returns a Promise.

The await keyword pauses the execution of an asynchronous function until the Promise is resolved.

Example:

```
function getMessage() {
    return new Promise(function(resolve) {
        setTimeout(function() {
            resolve("Data received");
        }, 2000);
    });
```

SGOU

```
}
async function displayMessage() {
    let message = await getMessage();
    console.log(message);
}
displayMessage();
```

Using async/await makes asynchronous code easier to read and understand.

1.5.1.5 JSON

JSON (JavaScript Object Notation) is a lightweight format used for representing and exchanging structured data.

A JSON object consists of key-value pairs.

Example:

```
{
    “name”: “Munnar”,
    “district”: “Idukki”,
    “type”: “Hill Station”
}
```

JSON data can be converted into a JavaScript object using JSON.parse().

```
let jsonData = ‘{“name”:”Munnar”,”district”:”Idukki”}’;
let place = JSON.parse(jsonData);
console.log(place.name);
```

A JavaScript object can be converted into a JSON string using JSON.stringify().

```
let place = {
    name: “Munnar”,
    district: “Idukki”
};
let jsonString = JSON.stringify(place);
console.log(jsonString);
```

SGOU

1.5.1.6 Fetch API

The Fetch API provides a mechanism for making network requests and retrieving resources asynchronously.

A basic Fetch API request is:

```
fetch("data.json")
    .then(response => response.json())
    .then(data => {
        console.log(data);
    })
    .catch(error => {
        console.log(error);
    });
```

The fetch() function returns a Promise.

The response.json() method processes the response as JSON data.

The retrieved data can then be used to dynamically display information on a webpage.

1.5.1.7 Fetch API with Async/Await

The Fetch API can also be used with async/await.

```
async function loadData() {
    try {
        let response = await fetch("data.json");
        let data = await response.json();
        console.log(data);
    } catch (error) {
        console.log(error);
    }
}
loadData();
```

This approach provides a readable way of handling asynchronous data retrieval.

SGOU

1.5.2 Implementation

Develop a webpage that collects visitor information, performs client-side validation, uses a timer, processes JSON data, and retrieves dynamic data using the Fetch API.

Program

HTML file: index.html

```
<!DOCTYPE html>
<html>
<head>
   <title>Kerala Explorer</title>
</head>
<body>
   <h1>Kerala Explorer</h1>
   <h2>Visitor Registration</h2>
   <form id="registrationForm">
      <label for="name">Name:</label>
      <input type="text"
          id="name"
          required>
      <br><br>
      <label for="email">Email:</label>
      <input type="email"
          id="email"
          required>
      <br><br>
      <label for="age">Age:</label>
      <input type="number"
          id="age"
          min="18"
          max="60"
```

SGOU

```
        required>
    <br><br>
    <button type="submit">Submit</button>
</form>
<p id="message"></p>
<h2>Popular Destinations</h2>
<div id="destinations">
    Loading destinations...
</div>
<script>
    // Client-side form validation
    document.getElementById("registrationForm")
        .addEventListener("submit", function(event) {
            event.preventDefault();
            let name =
                document.getElementById("name").value;
            let email =
                document.getElementById("email").value;
            let age =
                document.getElementById("age").value;
            let message =
                document.getElementById("message");
            if (name === "") {
                message.innerHTML =
                    "Please enter your name.";
                return;
            }
            if (email === "") {
```

SGOU

```
            message.innerHTML =
                "Please enter your email.";
            return;
        }
        if (age < 18 || age > 60) {
            message.innerHTML =
                "Age must be between 18 and 60.";
            return;
        }
        message.innerHTML =
            `Welcome, ${name}. Registration successful.`;
    });
// Timer using setTimeout()
setTimeout(function() {
    document.getElementById("message").innerHTML +=
        "<br>Thank you for visiting Kerala Explorer.";
}, 3000);
// Promise
function getDestinationMessage() {
    return new Promise(function(resolve, reject) {
        let success = true;
        if (success) {
            resolve("Destination data is ready.");
        } else {
            reject("Unable to load destination data.");
        }
    });
}
```

SGOU

```
getDestinationMessage()
    .then(function(message) {
        console.log(message);
    })
    .catch(function(error) {
        console.log(error);
    });
// Async/Await
async function displayPromiseMessage() {
    let message =
        await getDestinationMessage();
    console.log(message);
}
displayPromiseMessage();
// Fetch API with Async/Await
async function loadDestinations() {
    try {
        let response =
            await fetch("destinations.json");
        let data =
            await response.json();
        let output = "<ul>";
        for (let place of data) {
            output += `
                <li>
                    <strong>${place.name}</strong>
                    - ${place.district}
                </li>
```

SGOU

```
            `;
        }
        output += "</ul>";
        document.getElementById("destinations")
          .innerHTML = output;
      } catch (error) {
        document.getElementById("destinations")
          .innerHTML =
          "Unable to load destination data.";
      }
    }
    loadDestinations();
  </script>
</body>
</html>
```

JSON File: destinations.json

```
[
  {
    "name": "Munnar",
    "district": "Idukki"
  },
  {
    "name": "Wayanad",
    "district": "Wayanad"
  },
  {
    "name": "Alappuzha",
    "district": "Alappuzha"
```

SGOU

```
    },
    {
      "name": "Varkala",
      "district": "Thiruvananthapuram"
    }
]
```

Output



SGOU

1.5.3 Lab Practice Questions

1. Develop a webpage that performs client-side validation of a registration form.

2. Write a JavaScript program using setTimeout() to display a message after a specified delay.

3. Write a JavaScript program that creates and handles a Promise using then() and catch().

4. Develop a JavaScript program using async and await to perform an asynchronous operation.

5. Create a JSON object containing information about tourist destinations and display its contents on a webpage.

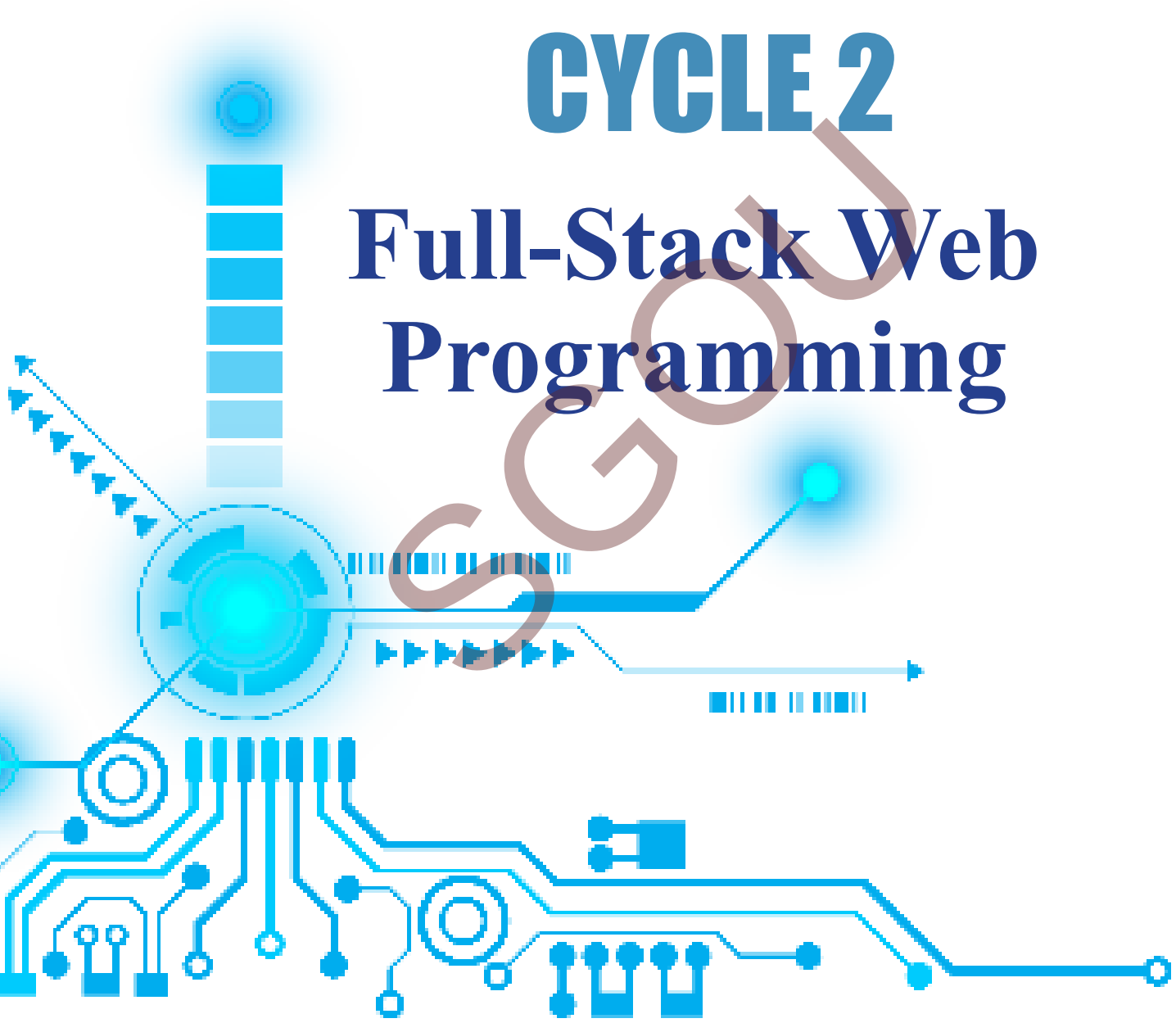
6. Develop a webpage that retrieves JSON data using the Fetch API and displays the retrieved information.

7. Develop a webpage combining client-side form validation, timers, Promises, async/await, JSON processing, and the Fetch API.

SGOU

CYCLE 2

Full-Stack Web Programming



SGOU

Introduction

Full-stack web programming involves the development of web applications by integrating client-side interfaces with server-side applications and databases. It enables applications to process user requests, provide data through APIs, store and manage information, and present the resulting data through interactive user interfaces.

Cycle II introduces the server-side and full-stack technologies required to develop complete web applications. The cycle begins with server-side application development using Node.js and Express, followed by the development of RESTful APIs. It then introduces database integration and CRUD operations, followed by the development of user interfaces using React. The cycle concludes with React Router and full-stack web application development, integrating the frontend, backend, REST API, and database.

The five experiments provide a progressive approach to full-stack web development, moving from server-side programming to complete full-stack application integration.

Objectives

After completing this cycle, the learner will be able to:

1. Develop server-side applications using Node.js and Express.
2. Create RESTful APIs using appropriate HTTP methods and status codes.
3. Integrate databases with web applications and implement CRUD operations.
4. Develop user interfaces using React functional components.
5. Apply props, state, event handling, controlled forms, and conditional rendering in React applications.
6. Implement client-side routing using React Router.
7. Integrate React, Express, REST APIs, and databases to develop full-stack web applications.

Scope

Cycle II covers the development of full-stack web applications using server-side technologies, APIs, databases, and React-based user interfaces.

The scope includes:

- Server-side application development using Node.js and Express.
- Creating and handling routes, requests, responses, and middleware.

SGOU

- Developing RESTful APIs.
- Working with HTTP methods and status codes.
- Testing REST API endpoints.
- Integrating applications with MongoDB or PostgreSQL.
- Implementing CRUD operations.
- Developing React user interfaces using functional components and JSX.
- Working with props and state.
- Handling events and developing controlled forms.
- Implementing conditional rendering.
- Implementing routing using React Router.
- Integrating the frontend, backend, REST API, and database to develop a full-stack web application.

Tools and Software

1. Visual Studio Code

Visual Studio Code (VS Code) is used as the primary code editor for developing Node.js, Express, React, and other source files used in the laboratory.

2. Node.js

Node.js provides the runtime environment for executing JavaScript on the server side. It is used for developing server-side applications in Cycle II.

3. npm

npm (Node Package Manager) is used to initialize Node.js projects and install and manage the packages required for server-side and React application development.

4. Express

Express is a Node.js web framework used to develop server-side applications, routes, APIs, and middleware.

5. Postman / API Testing Tool

Postman or another suitable API testing tool can be used to send HTTP requests to RESTful API endpoints and examine the responses.

6. MongoDB / PostgreSQL

A database system is required for database integration and CRUD operations.

SGOU

The laboratory may use either:

- MongoDB, or
- PostgreSQL

depending on the database implementation adopted for the course.

7. React

React is used to develop user interfaces using functional components, JSX, props, state, event handling, controlled forms, and conditional rendering.

8. React Router

React Router is used to implement routing and navigation between different views or pages in React applications.

9. Web Browser

A modern web browser is required to run and test React applications and interact with the developed web applications.

Examples include:

- Google Chrome
- Mozilla Firefox
- Microsoft Edge

General Procedure

Cycle II experiments focus on full-stack web programming using Node.js, Express, REST APIs, databases, React, and React Router. These experiments require separate backend and frontend setups depending on the experiment.

Step 1: Create a Separate Folder for Each Experiment

Organize the Cycle II experiments in separate folders.

```
WebProgrammingLab
|
└── Cycle_II
    ├── Experiment_1
    ├── Experiment_2
    ├── Experiment_3
    ├── Experiment_4
    └── Experiment_5
```

SGOU

Step 2: Open the Experiment Folder in VS Code

Open **Visual Studio Code** and select:

File → Open Folder

Choose the folder corresponding to the experiment.

Step 3: Initialize the Required Project

For Node.js and Express experiments, initialize the project using:

npm init -y

Install the required packages using npm install.

For example:

npm install express

For database experiments, install the required database library such as Mongoose or the appropriate PostgreSQL library.

For React experiments, create a React project using Vite and install the required packages.

npm create vite@latest project-name -- --template react

cd project-name

npm install

Step 4: Create the Required Program Files

Create the files specified for the experiment.

For example, a backend application may contain:

```
Experiment_1
├── app.js
├── package.json
└── node_modules
```

A React application may contain:

```
react-app
├── src
│   ├── App.jsx
│   └── main.jsx
├── package.json
└── node_modules
```

SGOU

Step 5: Run Node.js / Express Applications

For Node.js and Express experiments, run the server using:

node app.js

The terminal will display a message such as:

Server running at http://localhost:3000

Open the corresponding URL in a browser or test the API using Postman or Thunder Client.

Step 6: Run React Applications

For React experiments, start the development server using:

npm run dev

The terminal displays a local URL such as:

http://localhost:5173

Open the URL in a web browser to view the application.

Step 7: Run Full-Stack Applications

For the final full-stack experiment, the backend and frontend must run simultaneously.

Run the backend in one terminal:

node app.js

Run the React frontend in another terminal:

npm run dev

The React frontend communicates with the Express backend through the REST API.

Step 8: Test REST API Endpoints

Use Postman or Thunder Client to test API endpoints.

Verify:

- GET requests
- POST requests
- PUT/PATCH requests
- DELETE requests
- HTTP status codes
- JSON responses

SGOU

Step 9: Verify Database Operations

For database experiments, ensure that the database service or cloud database connection is available before running the application.

Verify that:

- New records are created.
- Existing records can be retrieved.
- Records can be updated.
- Records can be deleted.

Step 10: Debug Errors

Use the VS Code terminal to identify backend errors and the browser Developer Tools to identify frontend errors.

Press:

F12

or:

Ctrl + Shift + I

Check the Console and Network tabs when troubleshooting React and API communication.

SGOU

Experiment 1

Server-Side Application Development Using Node.js and Express

Objectives

- Understand the role of Node.js in server-side web application development.
- Create and execute a basic Node.js application.
- Use npm to manage a Node.js project.
- Develop a server-side application using Express.
- Create routes to handle client requests.
- Handle HTTP requests and responses using Express.
- Understand and implement middleware in an Express application.

SGOU

2.1.1 Theory

2.1.1.1 Node.js

Node.js is a JavaScript runtime environment used to execute JavaScript outside a web browser. It enables JavaScript to be used for server-side application development.

A Node.js application can receive requests from clients, process the requests, and send appropriate responses.

A simple Node.js program is:

```
console.log("Hello from Node.js");
```

The program can be executed using:

node app.js

2.1.1.2 npm

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

A Node.js project can be initialized using:

npm init

This creates a package.json file containing information about the project and its dependencies.

A package can be installed using:

npm install package-name

2.1.1.3 Express

Express is a web application framework for Node.js. It provides features for creating server-side applications, defining routes, handling requests and responses, and using middleware.

Express can be installed using:

npm install express

A basic Express application can be created as follows:

```
const express = require("express");
const app = express();
app.listen(3000, () => {
   console.log("Server running on port 3000");
});
```

The application starts a server that listens for incoming requests on port 3000.

2.1.1.4 Routing

Routing determines how an application responds to requests made to specific URLs and HTTP methods.

An Express route has the general form:

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

For example:

```
app.get("/", (req, res) => {
   res.send("Welcome to Kerala Explorer");
});
```

Here:

- app.get() defines a GET route.
- / represents the route path.
- req represents the request.

SGOU

- res represents the response.

Multiple routes can be defined in an Express application.

```
app.get("/", (req, res) => {
   res.send("Home Page");
});
app.get("/destinations", (req, res) => {
   res.send("Destinations Page");
});
```

2.1.1.5 Request and Response

An HTTP request is sent by a client to a server to request a resource or perform an operation.

Express provides the request (req) object to access information associated with the incoming request.

The response (res) object is used by the server to send a response to the client.

Example:

```
app.get("/hello", (req, res) => {
   res.send("Hello from the server");
});
```

The server receives a request for /hello and sends a response using res.send().

2.1.1.6 Middleware

Middleware functions are functions that execute during the request-response cycle of an Express application.

Middleware can perform operations before a request reaches the final route handler.

A middleware function generally has access to:

- Request object (req)
- Response object (res)
- Next function (next)

SGOU

Example:

```
app.use((req, res, next) => {
    console.log("Request received");
    next();
});
```

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

Middleware can be used for common processing tasks within an Express application.

2.1.2 Implementation

1. Develop a server-side application for Kerala Explorer using Node.js and Express. The application should demonstrate server creation, routing, request/response handling, and middleware.

Step 1: Create the Project

```
// Create a project directory and initialize the Node.js project.
mkdir kerala-explorer
cd kerala-explorer
npm init -y
```

Step 2: Install Express

```
npm install express
```

Step 3: Create app.js

```
const express = require("express");
const app = express();
const PORT = 3000;
// Middleware
app.use((req, res, next) => {
    console.log(
        `${req.method} request received for ${req.url}`
    );
    next();
```

SGOU

```
});
// Home route
app.get("/", (req, res) => {
  res.send(`
    <h1>Kerala Explorer</h1>
    <p>Welcome to Kerala Explorer.</p>
    <a href="/destinations">
      View Destinations
    </a>
  `);
});
// Destinations route
app.get("/destinations", (req, res) => {
  res.send(`
    <h1>Popular Destinations</h1>
    <ul>
      <li>Munnar</li>
      <li>Wayanad</li>
      <li>Alappuzha</li>
      <li>Varkala</li>
    </ul>
  `);
});
// About route
app.get("/about", (req, res) => {
  res.send(`
```

SGOU

```
    <h1>About Kerala Explorer</h1>
    <p>
      Explore popular destinations across Kerala.
    </p>

    `);
});
// Start server
app.listen(PORT, () => {
    console.log(
      `Server running at http://localhost:${PORT}`
    );
});
```

Step 4: Run the Application

Execute:

node app.js

The terminal displays:

Server running at http://localhost:3000

Open the following address in a web browser:

http://localhost:3000

The application responds according to the requested route:

/ → Home Page

/destinations → Destinations Page

/about → About Page

The middleware executes when a request is received and displays the request method and URL in the terminal.

SGOU

Output

http://localhost:3000/

Kerala Explorer

Welcome to Kerala Explorer.

View Destinations

http://localhost:3000/destinations

Popular Destinations

- Munnar
- Wayanad
- Alappuzha
- Varkala

2.1.3 Lab Practice Questions

1. Develop a Node.js application that displays a simple message.
2. Create an Express application with a route that displays a welcome message.
3. Develop an Express application with separate routes for Home, About, and Contact pages.
4. Create an Express route that displays a list of tourist destinations.
5. Develop an Express application that demonstrates request and response handling.
6. Implement middleware in an Express application to display information about incoming requests.
7. Develop an Express application combining routing, request/response handling, and middleware.

SGOU

Experiment 2

RESTful API Development

Objectives

- Understand the concept and purpose of RESTful APIs.
- Identify resources in a REST-based application.
- Develop RESTful API endpoints using Express.
- Use appropriate HTTP methods for different operations.
- Understand and apply HTTP status codes.
- Test REST API endpoints using Postman or a similar API testing tool.

2.2.1 Theory

2.2.1.1 REST

REST (Representational State Transfer) is an architectural style used for designing web services that communicate over HTTP.

In a RESTful application, data is represented as resources. Each resource is identified using a URL.

For example, in a tourism application:

/destinations

can represent the collection of destinations.

A particular destination can be represented as:

/destinations/1

The client communicates with the server using HTTP requests, and the server returns an appropriate response.

2.2.1.2 RESTful API

An **API (Application Programming Interface)** provides a way for different software applications to communicate with each other.

A RESTful API uses HTTP methods to perform operations on resources.

SGOU

For example:

HTTP Method	Operation	Example
GET	Retrieve data	/destinations
POST	Create data	/destinations
PUT	Update data	/destinations/1
PATCH	Partially update data	/destinations/1
DELETE	Delete data	/destinations/1

2.2.1.3 GET Method

The GET method is used to retrieve information from a server.

Example:

```
app.get("/destinations", (req, res) => {
  res.json([
    {
      id: 1,
      name: "Munnar"
    },
    {
      id: 2,
      name: "Wayanad"
    }
  ]);
});
```

A request to:

GET /destinations

returns the available destinations.

2.2.1.4 POST Method

The POST method is used to create a new resource.

Example:

```
app.post("/destinations", (req, res) => {
```

SGOU

```
  res.status(201).json({
    message: “Destination created”
  });
});
```

A POST request generally contains data in the request body.

2.2.1.5 PUT Method

The PUT method is used to update an existing resource.

Example:

```
app.put(“/destinations/:id”, (req, res) => {
  res.json({
    message: `Destination ${req.params.id} updated`
  });
});
```

The :id represents a route parameter identifying the resource.

2.2.1.6 PATCH Method

The PATCH method is used to partially update an existing resource.

Example:

```
app.patch(“/destinations/:id”, (req, res) => {
  res.json({
    message: `Destination ${req.params.id} partially updated`
  });
});
```

2.2.1.7 DELETE Method

The DELETE method is used to remove a resource.

Example:

```
app.delete(“/destinations/:id”, (req, res) => {
  res.json({
    message: `Destination ${req.params.id} deleted`
  });
```

SGOU

});

2.2.1.8 HTTP Status Codes

HTTP status codes indicate the result of an HTTP request.

Some commonly used status codes are:

Status Code	Meaning
200	OK
201	Created
400	Bad Request
404	Not Found
500	Internal Server Error

For example:

```
res.status(201).json({
    message: “Destination created”
});
```

The status code 201 indicates that a resource has been successfully created.

2.2.1.9 Testing REST APIs

REST API endpoints can be tested using tools such as Postman or similar API testing tools.

A request can be configured by specifying:

- HTTP method
- URL
- Request body, when required
- Headers, when required

The response can then be examined to verify:

- Status code
- Response data
- Response structure

2.2.2 Implementation

Develop and Test a RESTful API Using Express

SGOU

Develop a RESTful API for managing tourist destinations. The API should support retrieving, creating, updating, and deleting destinations.

Step 1: Create the Project

```
mkdir destination-api
cd destination-api
npm init -y
npm install express
```

Step 2: Create app.js

```
const express = require("express");
const app = express();
const PORT = 3000;
app.use(express.json());
let destinations = [
  {
    id: 1,
    name: "Munnar",
    district: "Idukki"
  },
  {
    id: 2,
    name: "Wayanad",
    district: "Wayanad"
  }
];
// GET - Retrieve all destinations
app.get("/destinations", (req, res) => {
  res.status(200).json(destinations);
```

SGOU

```
});
// GET - Retrieve a particular destination


app.get("/destinations/:id", (req, res) => {
    const id = Number(req.params.id);
    const destination =
        destinations.find(place => place.id === id);
    if (!destination) {
        return res.status(404).json({
            message: "Destination not found"
        });
    }
    res.status(200).json(destination);
});
// POST - Create a new destination
app.post("/destinations", (req, res) => {
    const newDestination = {
        id: destinations.length + 1,
        name: req.body.name,
        district: req.body.district
    };
    destinations.push(newDestination);
    res.status(201).json(newDestination);
});
// PUT - Update a destination
app.put("/destinations/:id", (req, res) => {
    const id = Number(req.params.id);
    const destination =
```

SGOU

```
    destinations.find(place => place.id === id);
  if (!destination) {
    return res.status(404).json({
      message: "Destination not found"
    });
  }
  destination.name = req.body.name;
  destination.district = req.body.district;
  res.status(200).json(destination);
});
// DELETE - Delete a destination
app.delete("/destinations/:id", (req, res) => {
  const id = Number(req.params.id);
  const index =
    destinations.findIndex(place => place.id === id);
  if (index === -1) {
    return res.status(404).json({
      message: "Destination not found"
    });
  }
  destinations.splice(index, 1);
  res.status(200).json({
    message: "Destination deleted successfully"
  });
});
// Start server
app.listen(PORT, () => {
  console.log(
```

SGOU

```
    `Server running at http://localhost:${PORT}`
  );
});
```

Step 3: Run the Application

```
node app.js
The server runs at:
http://localhost:3000
```

Step 4: Test the API

The endpoints can be tested using Postman or another API testing tool.

GET — Retrieve All Destinations

Method:

GET

URL:

http://localhost:3000/destinations

Expected status:

200 OK

Response:

```
[
  {
    "id": 1,
    "name": "Munnar",
    "district": "Idukki"
  },
  {
    "id": 2,

    "name": "Wayanad",
    "district": "Wayanad"
  }
]
```

SGOU

POST — Create a Destination

Method:

POST

URL:

http://localhost:3000/destinations

Request Body:

```
{
  "name": "Alappuzha",
  "district": "Alappuzha"
}
```

Expected status:

201 Created

PUT — Update a Destination

Method:

PUT

URL:

http://localhost:3000/destinations/1

Request Body:

```
{
  "name": "Munnar Hills",
  "district": "Idukki"
}
```

Expected status:

200 OK

DELETE — Delete a Destination

Method:

DELETE

URL:

http://localhost:3000/destinations/2

Expected status:

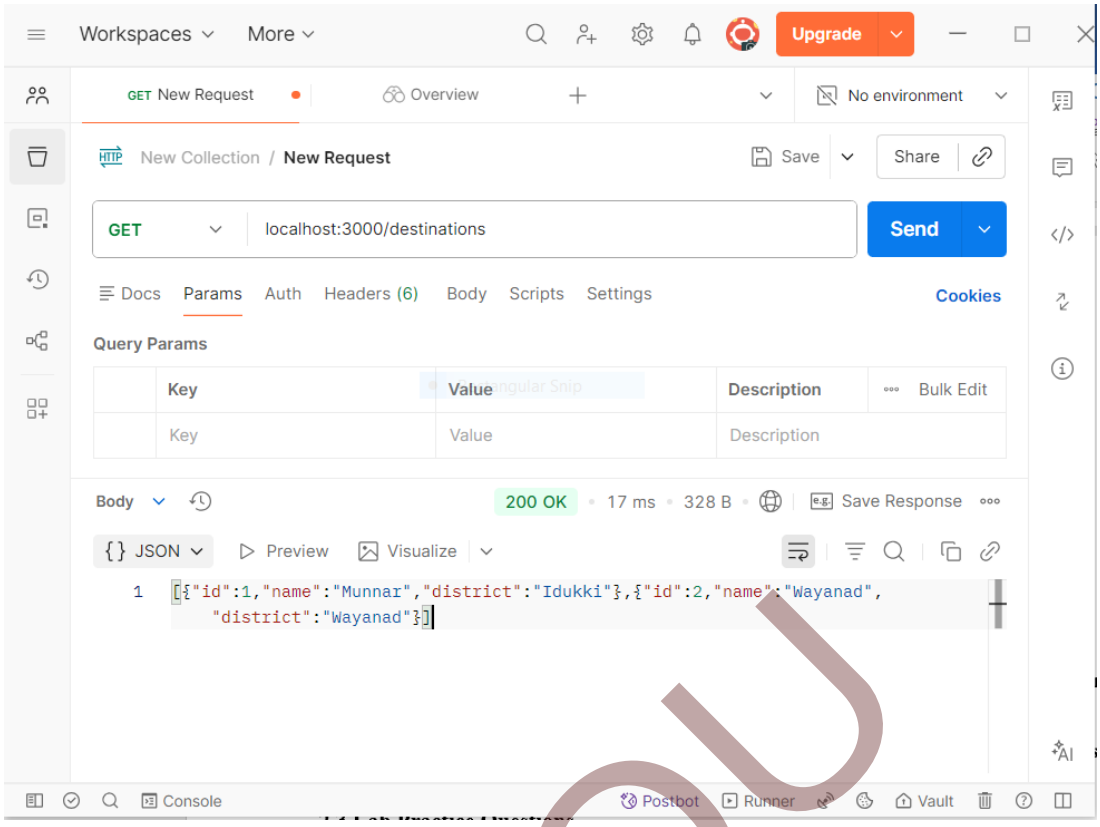
200 OK

Output

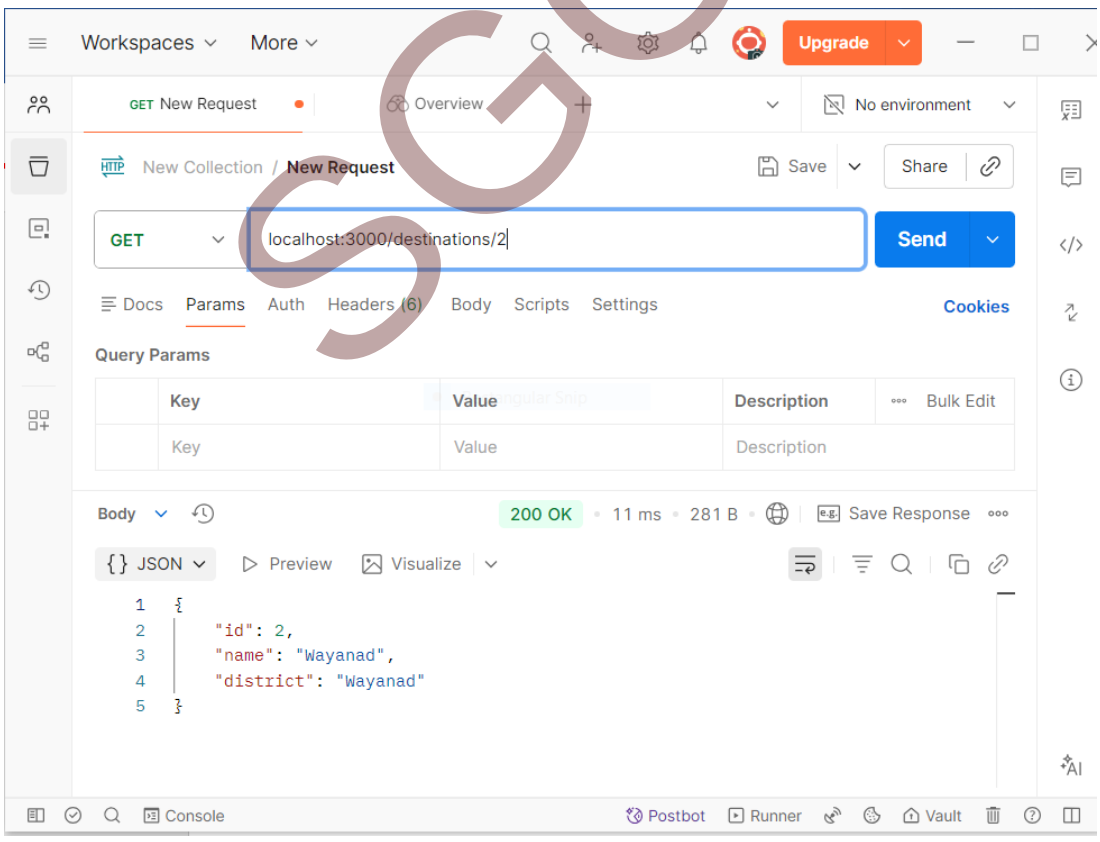
The RESTful API successfully provides endpoints for:

GET /destinations

SGOU

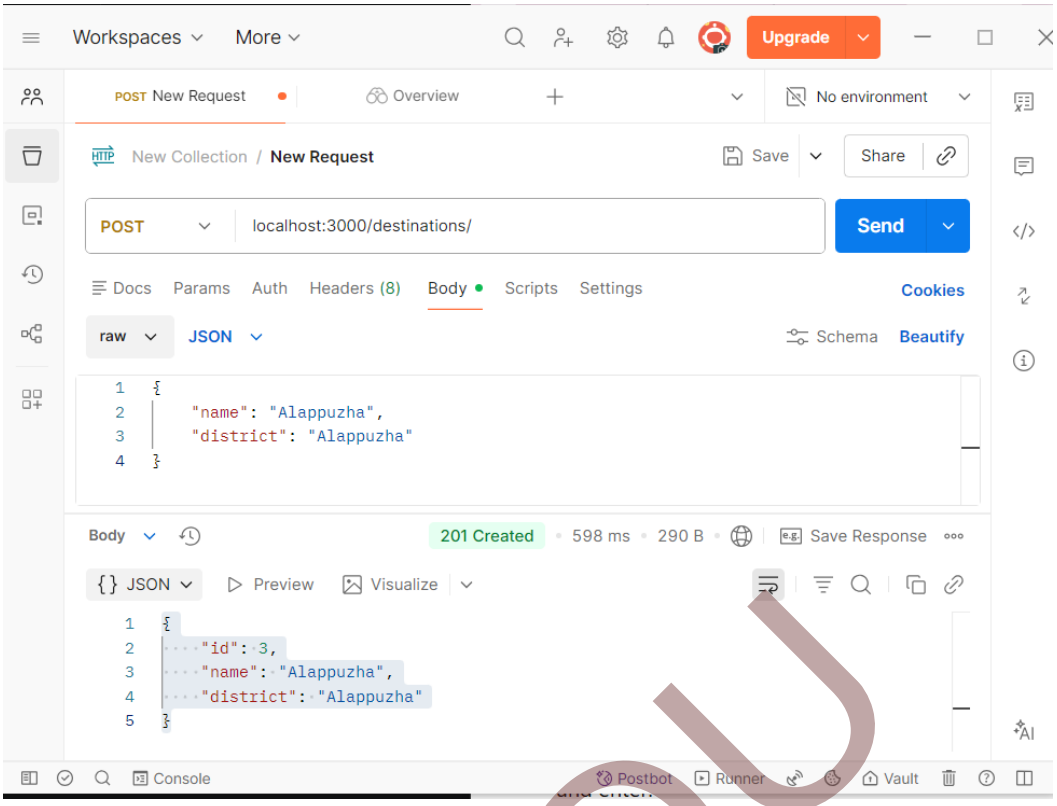


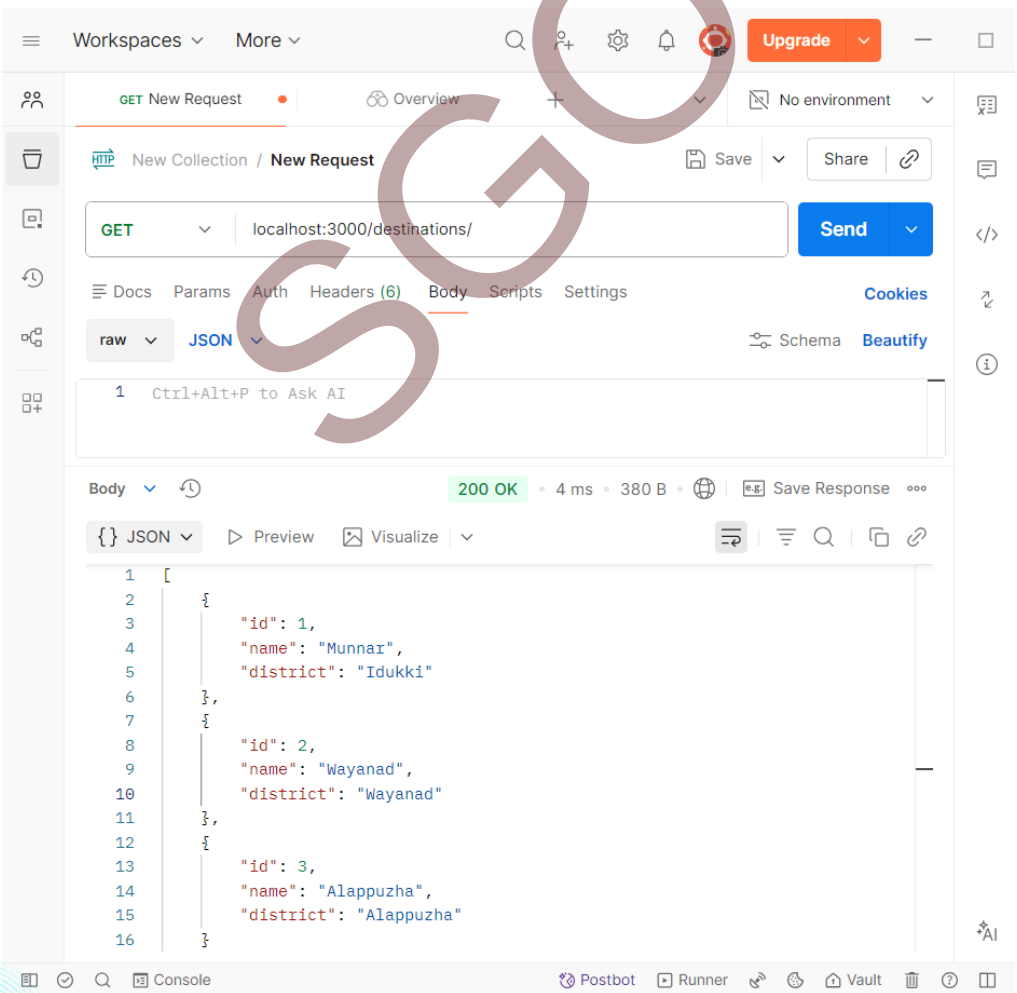
GET /destinations/:id



POST /destinations

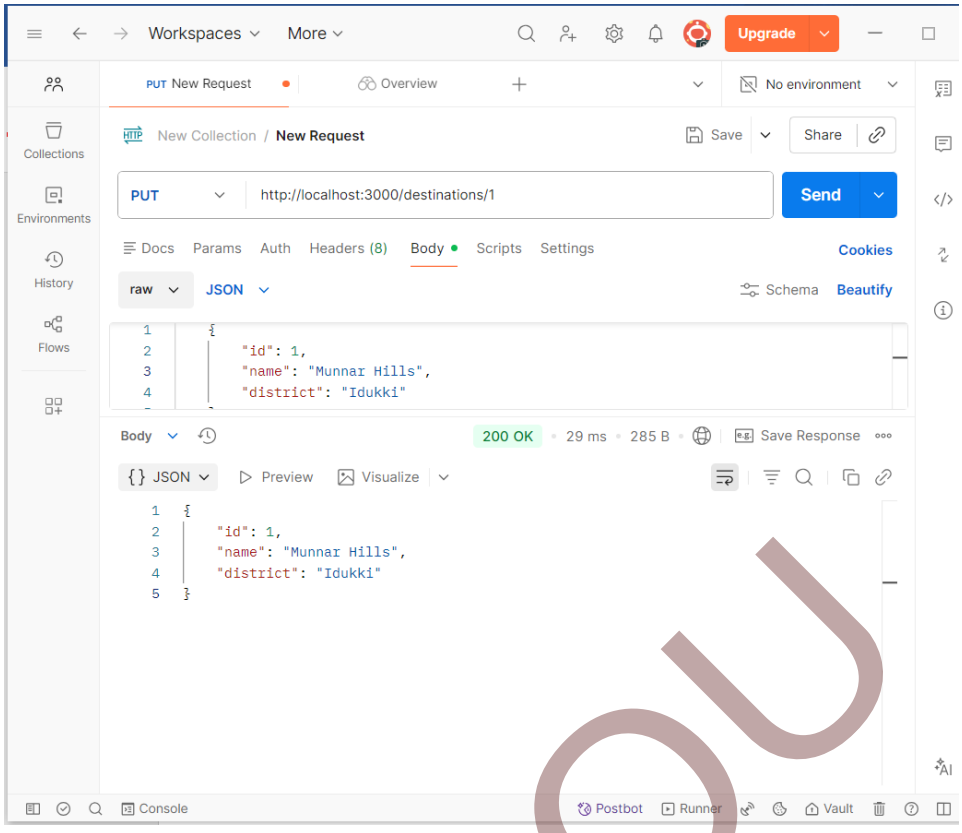
SGOU


Workspaces
More
Upgrade
POST New Request
Overview
No environment
New Collection / New Request
Save
Share
POST
localhost:3000/destinations/
Send
Docs Params Auth Headers (8) Body Scripts Settings
Cookies
raw JSON
Schema Beautify
{
"name": "Alappuzha",
"district": "Alappuzha"
}
Body
201 Created
598 ms
290 B
Save Response
JSON Preview Visualize
{
"id": 3,
"name": "Alappuzha",
"district": "Alappuzha"
}
Console
Postbot
Runner
Vault

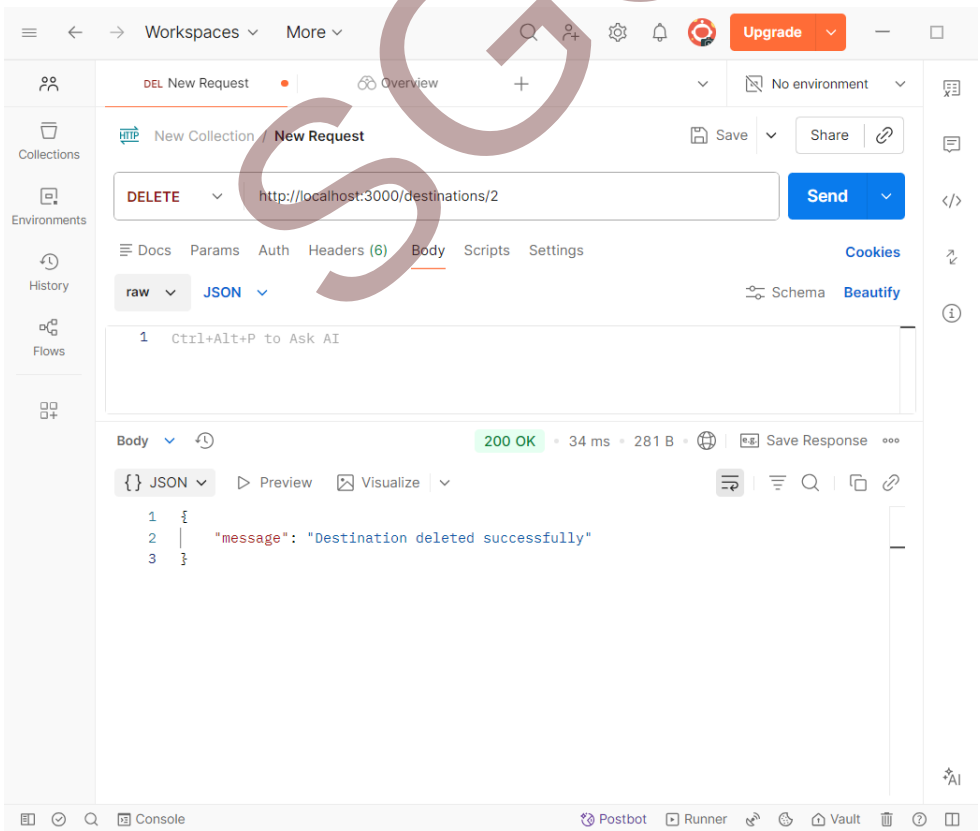

Workspaces
More
Upgrade
GET New Request
Overview
No environment
New Collection / New Request
Save
Share
GET
localhost:3000/destinations/
Send
Docs Params Auth Headers (6) Body Scripts Settings
Cookies
raw JSON
Schema Beautify
Ctrl+Alt+P to Ask AI
Body
200 OK
4 ms
380 B
Save Response
JSON Preview Visualize
[
{
"id": 1,
"name": "Munnar",
"district": "Idukki"
},
{
"id": 2,
"name": "Wayanad",
"district": "Wayanad"
},
{
"id": 3,
"name": "Alappuzha",
"district": "Alappuzha"
}
Console
Postbot
Runner
Vault

SGOU

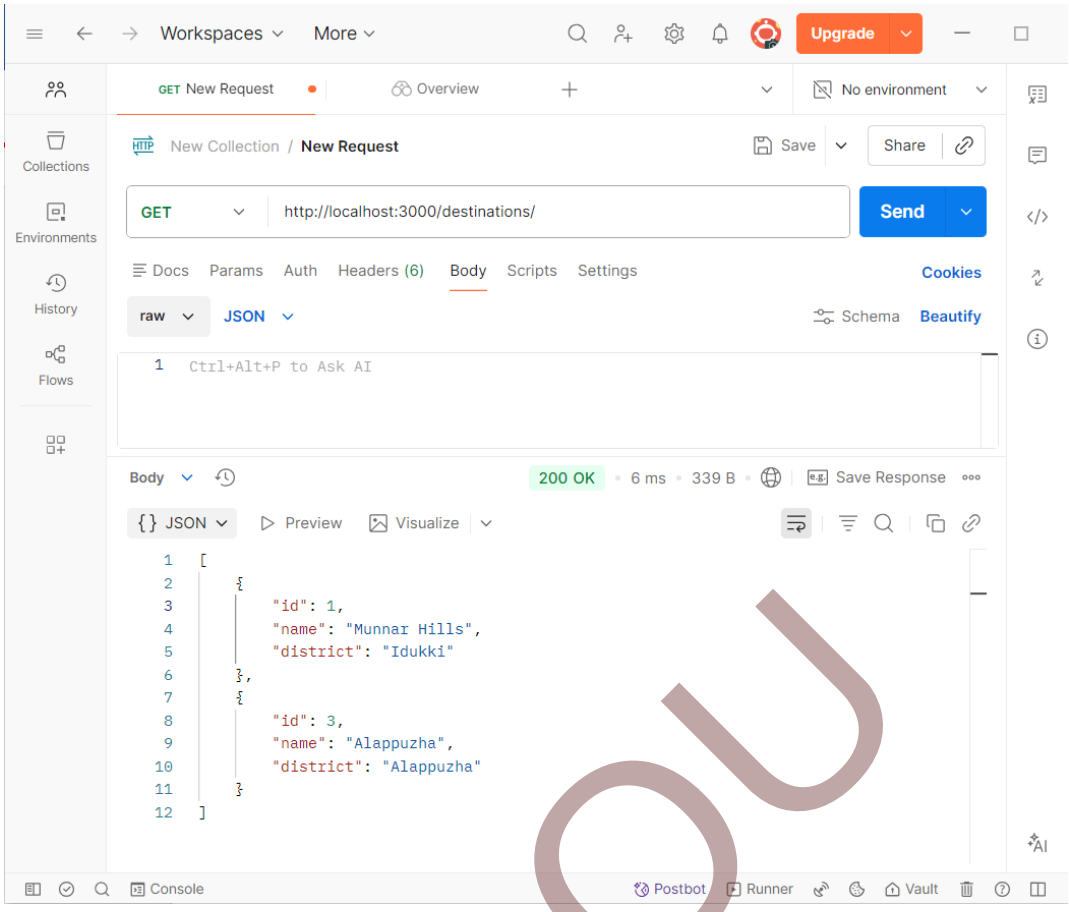
PUT /destinations/:id



DELETE /destinations/:id



SGOU



The API returns appropriate HTTP status codes and JSON responses. The endpoints can be verified using Postman or a similar API testing tool.

Thus, the implementation covers RESTful APIs, HTTP methods, HTTP status codes, and API endpoint testing.

2.2.3 Lab Practice Questions

1. Develop a RESTful API using Express to retrieve a list of students.
2. Create a REST API endpoint using POST to add a new record.
3. Develop an API using PUT to update an existing resource.
4. Develop an API using DELETE to remove a resource.
5. Implement appropriate HTTP status codes for successful and unsuccessful API requests.
6. Test GET, POST, PUT, and DELETE endpoints using Postman or a similar API testing tool.

SGOU

Experiment 3

Database Integration and CRUD Operations

Objectives

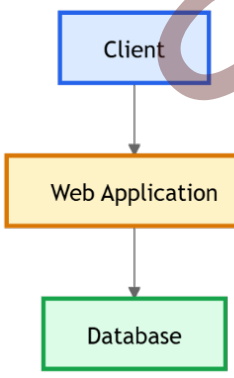
- Understand the role of databases in web applications.
- Connect a web application to a database.
- Store application data in a database.
- Retrieve records from a database.
- Create, read, update, and delete records using a web application.
- Implement CRUD operations through server-side application code.

2.3.1 Theory

2.3.1.1 Database Integration

A database is used to store and manage application data persistently. In a web application, the server-side application communicates with the database to store new information, retrieve existing information, modify records, and remove records.

The basic interaction can be represented as:



The server-side application acts as an intermediary between the client and the database.

For this experiment, the database can be MongoDB or PostgreSQL.

2.3.1.2 CRUD Operations

CRUD represents the four basic operations performed on stored data:

SGOU

Operation	Purpose
Create	Add a new record
Read	Retrieve existing records
Update	Modify an existing record
Delete	Remove a record

These operations allow a web application to manage data throughout its lifecycle.

2.3.1.3 Create

The Create operation adds a new record to the database.

For example, a tourism application may create a new destination record containing:

Name

District

Category

2.3.1.4 Read

The Read operation retrieves records stored in the database.

A web application can retrieve:

- All destination records.
- A particular destination record.

2.3.1.5 Update

The **Update** operation modifies an existing record.

For example, the district or category of an existing destination can be changed.

2.3.1.6 Delete

The **Delete** operation removes an existing record from the database.

For example, a destination that is no longer required can be removed from the database.

2.3.2 Implementation

Develop a server-side application that manages tourist destination records using Node.js, Express, and MongoDB.

SGOU

Step 1: Create the Project

mkdir destination-db

cd destination-db

npm init -y

npm install express mongoose

Step 2: Create app.js

```
const express = require("express");
const mongoose = require("mongoose");
const app = express();
const PORT = 3000;
app.use(express.json());
// Database connection
mongoose.connect("mongodb://127.0.0.1:27017/kerala_explorer")
  .then(() => {
    console.log("Database connected");
  })
  .catch((error) => {
    console.log("Database connection failed:", error);
  });
// Schema
const destinationSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true
  },
  district: {
    type: String,
    required: true
  },
```

SGOU

```
    category: {
      type: String,
      required: true
    }
});
// Model
const Destination =
    mongoose.model("Destination", destinationSchema);
// CREATE
app.post("/destinations", async (req, res) => {
    try {
      const destination =
        await Destination.create(req.body);
      res.status(201).json(destination);
    } catch (error) {
      res.status(400).json({
        message: "Unable to create destination"
      });
    }
});
// READ
app.get("/destinations", async (req, res) => {
    try {
      const destinations =
        await Destination.find();
      res.status(200).json(destinations);
    } catch (error) {
      res.status(500).json({
        message: "Unable to retrieve destinations"
```

SGOU

```
    });
  }
});
// UPDATE
app.put("/destinations/:id", async (req, res) => {
  try {
    const destination =
      await Destination.findByIdAndUpdate(
        req.params.id,
        req.body,
        {
          new: true,
          runValidators: true
        }
      );
    if (!destination) {
      return res.status(404).json({
        message: "Destination not found"
      });
    }
    res.status(200).json(destination);
  } catch (error) {
    res.status(400).json({
      message: "Unable to update destination"
    });
  }
});
// DELETE
```

SGOU

```
app.delete("/destinations/:id", async (req, res) => {
  try {
    const destination =
      await Destination.findByIdAndDelete(
        req.params.id
      );
    if (!destination) {
      return res.status(404).json({
        message: "Destination not found"
      });
    }
    res.status(200).json({
      message: "Destination deleted successfully"
    });
  } catch (error) {
    res.status(400).json({
      message: "Unable to delete destination"
    });
  }
});
// Start server
app.listen(PORT, () => {
  console.log(
    `Server running at http://localhost:${PORT}`
  );
});
```

Step 3: Start MongoDB

Ensure that MongoDB is running on the system.

SGOU

The application connects to:

mongodb://127.0.0.1:27017/kerala_explorer

The database is named:

kerala_explorer

Step 4: Test the CRUD Operations

The endpoints can be tested using Postman or another API testing tool.

Create a Destination

POST

http://localhost:3000/destinations

Request body:

```
{
  "name": "Munnar",
  "district": "Idukki",
  "category": "Hill Station"
}
```

The destination is stored in the database.

Read Destinations

GET

http://localhost:3000/destinations

The stored destination records are retrieved from the database.

Update a Destination

PUT

http://localhost:3000/destinations/<destination-id>

Request body:

```
{
  "name": "Munnar Hills",
  "district": "Idukki",
```

SGOU

"category": "Hill Station"

}

The selected database record is updated.

Delete a Destination

DELETE

http://localhost:3000/destinations/<destination-id>

The selected destination record is removed from the database.

Alternative Version

First create a table using the following query :

```
CREATE TABLE destinations (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    district TEXT NOT NULL,
    category TEXT NOT NULL
);
This version uses postgres
First install pg using:
npm install pg
const express = require("express");
const cors = require("cors");
const { Pool } = require("pg");
const app = express();
const PORT = 3000;
const pool = new Pool({
  connectionString:
    "postgresql://postgres:<password>@localhost:5432/kerala_explorer"
});
```

SGOU

```
app.use(cors());
app.use(express.json());
// CREATE
app.post("/destinations", async (req, res) => {
  const { name, district, category } = req.body;
  try {
    const result = await pool.query(
      `INSERT INTO destinations (name, district, category)
       VALUES ($1, $2, $3)
       RETURNING id, name, district, category, id AS _id`,
      [name, district, category]
    );
    res.status(201).json(result.rows[0]);
  } catch (error) {
    console.log(error);
    res.status(400).json({
      message: "Unable to create destination"
    });
  }
});
// READ all
app.get("/destinations", async (req, res) => {
  try {
    const result = await pool.query(
      `SELECT id, name, district, category, id AS _id
       FROM destinations
       ORDER BY id`
```

SGOU

```
        );
        res.status(200).json(result.rows);
    } catch (error) {
        console.log(error);
        res.status(500).json({
            message: "Unable to retrieve destinations"
        });
    }
});
// READ single destination
app.get("/destinations/:id", async (req, res) => {
    try {
        const result = await pool.query(
            `SELECT id, name, district, category, id AS _id
             FROM destinations
             WHERE id = $1`,
            [req.params.id]
        );
        if (result.rows.length === 0) {
            return res.status(404).json({
                message: "Destination not found"
            });
        }
        res.status(200).json(result.rows[0]);
    } catch (error) {
        console.log(error);
        res.status(400).json({
```

SGOU

```
            message: “Unable to retrieve destination”
        });
    }
});
// UPDATE
app.put(“/destinations/:id”, async (req, res) => {
    const { name, district, category } = req.body;
    try {
        const result = await pool.query(
            `UPDATE destinations
             SET name = $1, district = $2, category = $3
             WHERE id = $4
             RETURNING id, name, district, category, id AS _id`,
            [name, district, category, req.params.id]
        );
        if (result.rows.length === 0) {
            return res.status(404).json({
                message: “Destination not found”
            });
        }
        res.status(200).json(result.rows[0]);
    } catch (error) {
        console.log(error);
        res.status(400).json({
            message: “Unable to update destination”
        });
    }
```

SGOU

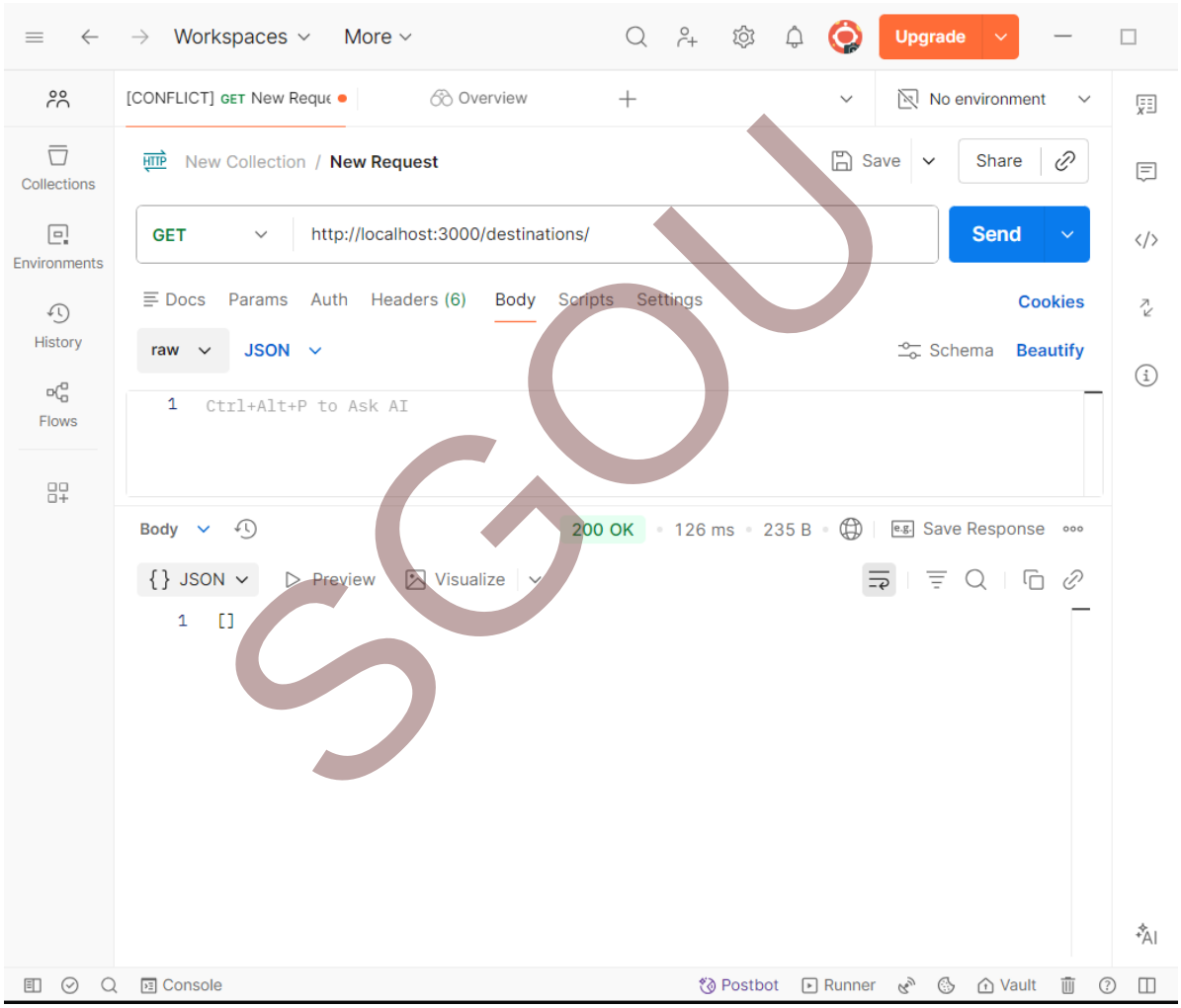
```
});
// DELETE
app.delete("/destinations/:id", async (req, res) => {
  try {
    const result = await pool.query(
      `DELETE FROM destinations
       WHERE id = $1
       RETURNING id`,
      [req.params.id]
    );
    if (result.rows.length === 0) {
      return res.status(404).json({
        message: "Destination not found"
      });
    }
    res.status(200).json({
      message: "Destination deleted successfully"
    });
  } catch (error) {
    console.log(error);
    res.status(400).json({
      message: "Unable to delete destination"
    });
  }
});
// Start server
app.listen(PORT, () => {
  console.log(
```

SGOU

```
        `Server running at http://localhost:${PORT}`
    );
});
```

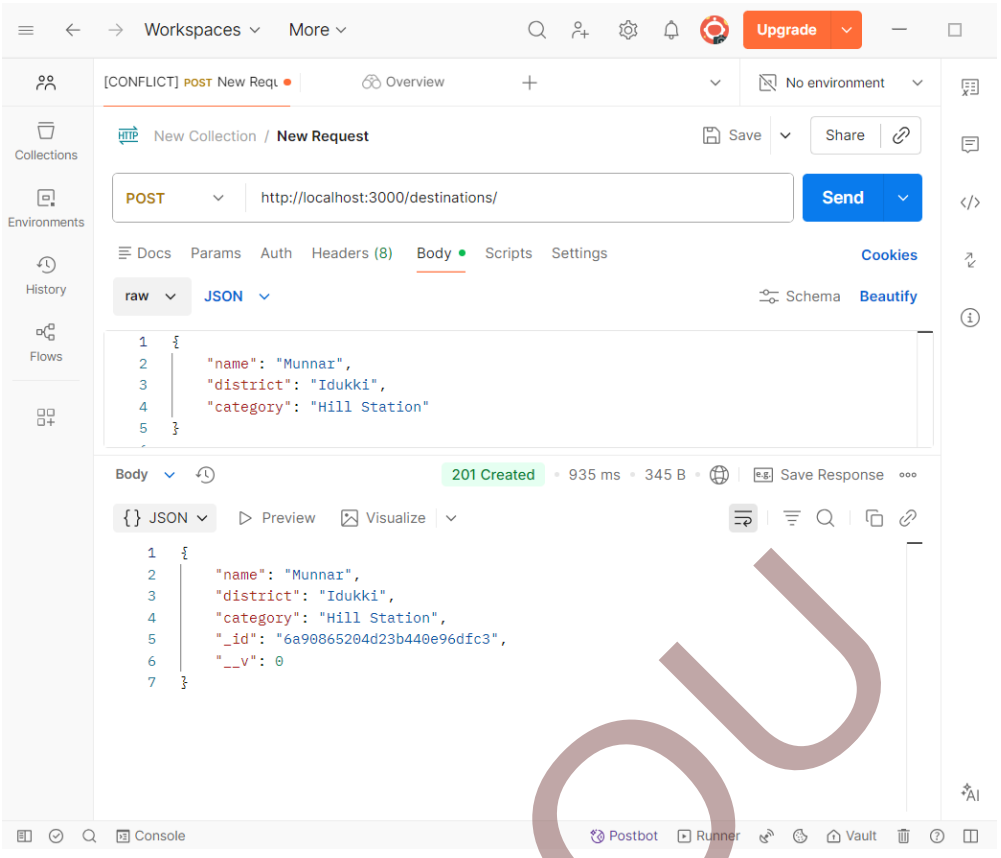
Output

The application successfully performs the four CRUD operations:

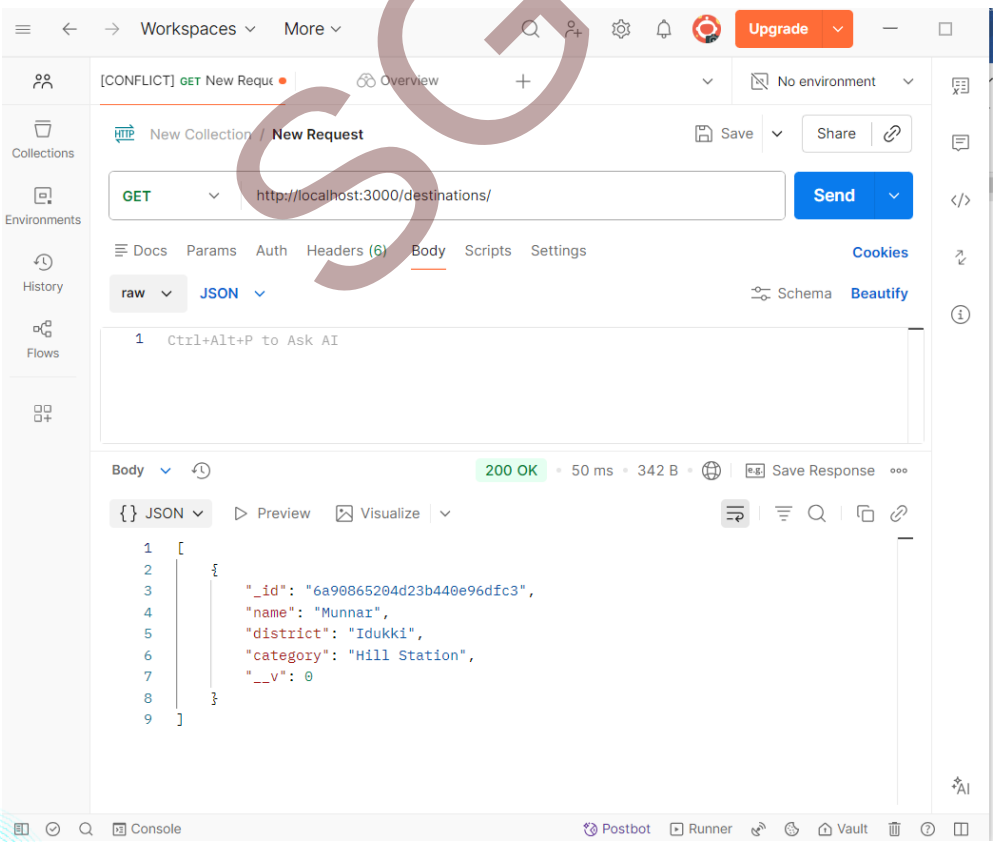


SGOU

CREATE -Store destination in database

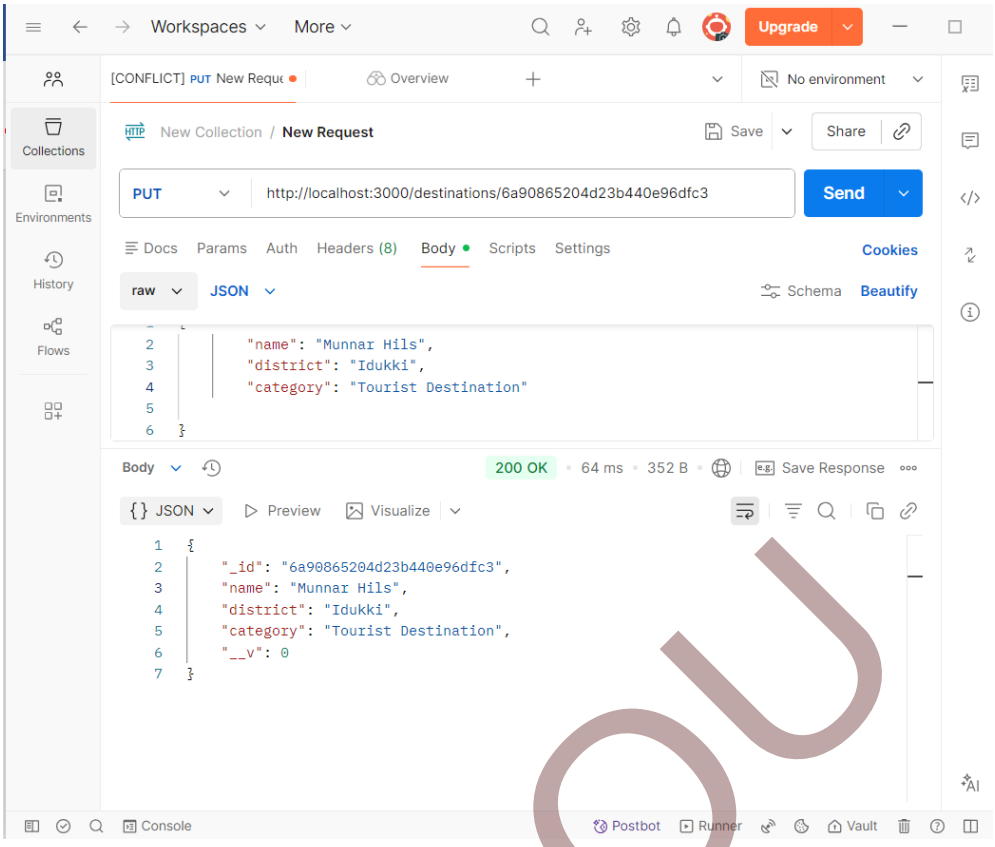


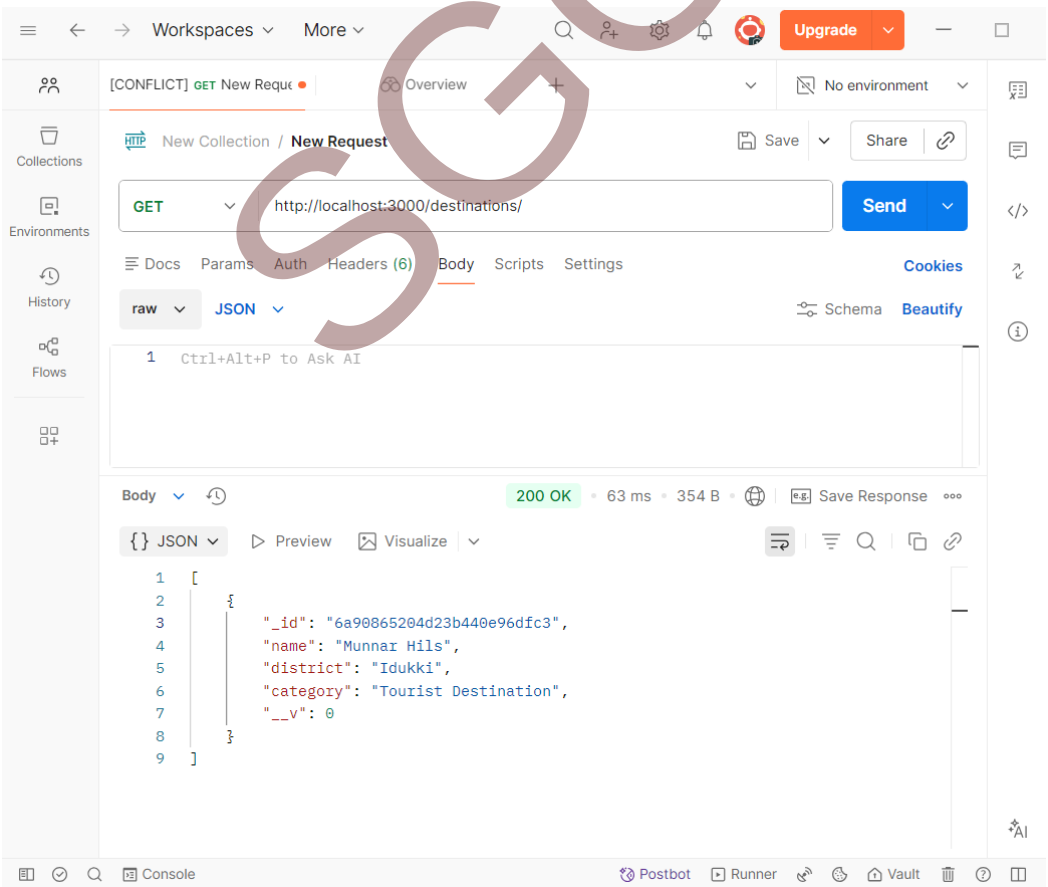
READ- Retrieve destinations



SGOU

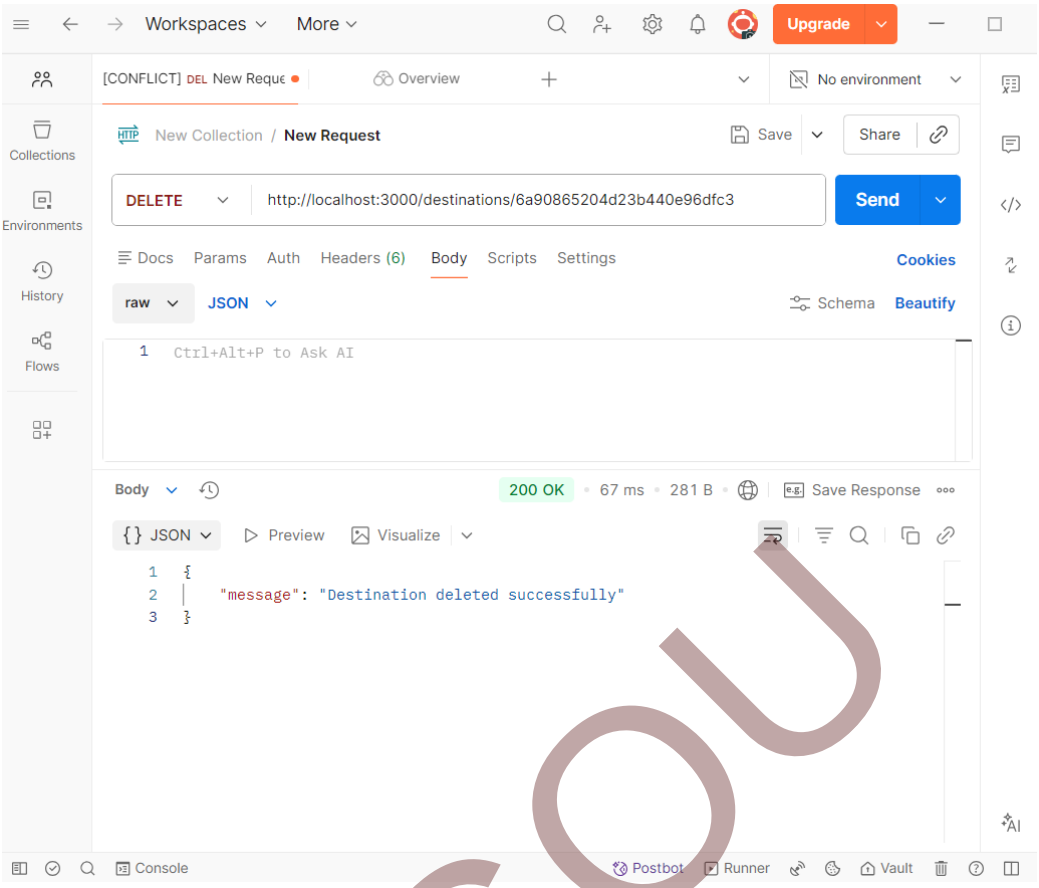
UPDATE- Modify destination

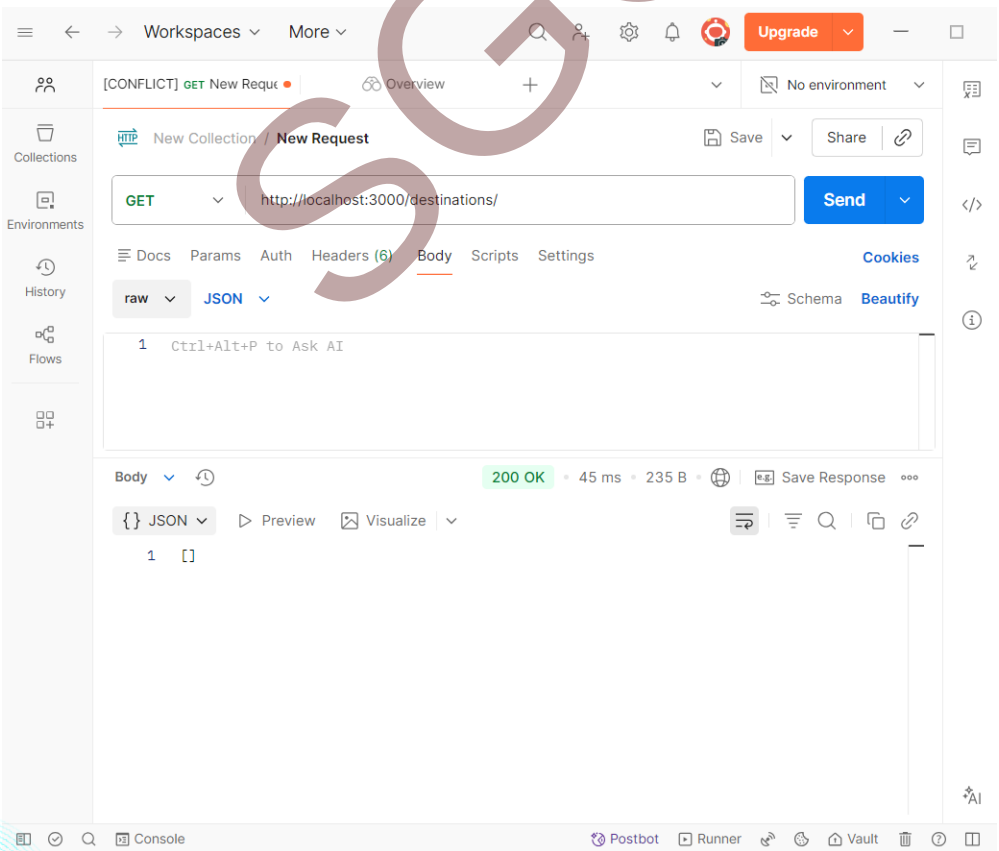




SGOU

DELETE- Remove destination





SGOU

The application therefore demonstrates database integration and CRUD operations using a Node.js and Express application with MongoDB.

2.3.3 Lab Practice Questions

1. Develop a web application that connects to a database and stores student records.

2. Implement the Create operation to add a new record to the database.

3. Implement the Read operation to retrieve records from the database.

4. Implement the Update operation to modify an existing record.

5. Implement the Delete operation to remove a record from the database.

6. Develop a web application that performs all four CRUD operations on a selected resource.

7. Integrate a database with an Express application and implement CRUD operations for managing records.

SGOU

Experiment 4

Building User Interfaces Using React

Objectives

- Understand the basic concepts of React.
- Create user interfaces using functional components.
- Use JSX to describe the structure of React interfaces.
- Pass data between components using props.
- Manage component data using state.
- Handle user interactions using event handling.
- Develop controlled forms using React state.
- Implement conditional rendering in React applications.

2.4.1 Theory

2.4.1.1 Introduction to React

React is a JavaScript library used for developing user interfaces. It allows an application interface to be divided into reusable components.

A React application is built by combining components to create the complete user interface.

A simple React component is:

```
function Welcome() {
   return <h1>Welcome to Kerala Explorer</h1>;
}
```

The component can be rendered as part of a React application.

2.4.1.2 Functional Components

A functional component is a JavaScript function that returns JSX representing a part of the user interface.

SGOU

Example:

```
function Destination() {
  return (
    <div>
      <h2>Munnar</h2>
      <p>Idukki District</p>
    </div>
  );
}
```

Components can be reused within an application.

```
function App() {
  return (
    <div>
      <Destination />
      <Destination />
    </div>
  );
}
```

2.4.1.3 JSX

JSX (JavaScript XML) is a syntax used in React to describe the structure of user interfaces.

Example:

```
const element = (
  <div>
    <h1>Kerala Explorer</h1>
    <p>Explore Kerala destinations.</p>
  </div>
);
```

JSX allows HTML-like elements to be written within JavaScript code.

SGOU

JavaScript expressions can be included inside JSX using curly braces.

```
let name = “Munnar”;
function Destination() {
   return <h2>{name}</h2>;
}
```

2.4.1.4 Props

Props are used to pass data from one component to another.

Example:

```
function Destination(props) {
   return (
      <div>
         <h2>{props.name}</h2>
         <p>{props.district}</p>
      </div>
   );
}
```

The component can receive values from its parent component.

```
<Destination
   name=”Munnar”
   district=”Idukki”
/>
```

Props are read-only and are used to provide information to a component.

2.4.1.5 State

State represents data that can change during the execution of a React component.

The useState Hook can be used to create and update state.

```
import { useState } from “react”;
function Counter() {
   const [count, setCount] = useState(0);
   return (
```

SGOU

```
      <div>
        <p>Count: {count}</p>
        <button onClick={() =>
          setCount(count + 1)
        }>
          Increase
        </button>
      </div>
    );
}
```

When the state changes, React updates the corresponding part of the user interface.

2.4.1.6 Event Handling

React allows user interactions to be handled through event handlers.

For example, a button click can be handled using onClick.

```
function Message() {
  function showMessage() {
    alert("Welcome!");
  }
  return (
    <button onClick={showMessage}>
      Click Me
    </button>
  );
}
```

Event handling can be used for actions such as button clicks and form interactions.

2.4.1.7 Controlled Forms

In a controlled form, the value of a form element is controlled by React state.

SGOU

Example:

```
import { useState } from “react”;
function Form() {
   const [name, setName] = useState(“”);
   return (
      <input
         value={name}
         onChange={(event) =>
            setName(event.target.value)
         }
      />
   );
}
```

The input value is stored in the component’s state. Whenever the user enters a value, the state is updated.

2.4.1.8 Conditional Rendering

Conditional rendering allows different content to be displayed based on a condition.

Example:

```
function Status({ available }) {
   if (available) {
      return <p>Destination available</p>;
   }
   return <p>Destination unavailable</p>;
}
```

Conditional expressions can also be used directly within JSX.

```
{available
   ? <p>Available</p>
   : <p>Not Available</p>
}
```

SGOU

2.4.2 Implementation

Develop a React User Interface for Displaying and Managing Destination Information.

Develop a React application that displays destination information using functional components and JSX. The application should demonstrate props, state, event handling, controlled forms, and conditional rendering.

App.jsx

```
import { useState } from “react”;

// Functional component using props

function Destination({ name, district, type }) {

   return (

      <div className=”destination”>

         <h2>{name}</h2>

         <p>District: {district}</p>

         <p>Type: {type}</p>

      </div>

   );

}

// Main component

function App() {

   // State

   const [showForm, setShowForm] = useState(false);

   const [name, setName] = useState(“”);

   const [message, setMessage] = useState(“”);

   // Event handling and controlled form

   function handleSubmit(event) {

      event.preventDefault();
```

SGOU

```
        setMessage(
            `Welcome, ${name}!`
    );
}
return (
    <div>
        <h1>Kerala Explorer</h1>
        <h2>Popular Destinations</h2>
        {/* Components using props */}
        <Destination
            name=”Munnar”
            district=”Idukki”
            type=”Hill Station”
        />
        <Destination
            name=”Wayanad”
            district=”Wayanad”
            type=”Hill Station”
        />
        <Destination
            name=”Alappuzha”
            district=”Alappuzha”
            type=”Backwaters”
        />
        {/* Event handling */}
```

SGOU

```
<button
   onClick={() =>
      setShowForm(!showForm)
   }
>
   {showForm
      ? “Hide Form”
      : “Show Form”}
</button>
{/* Conditional rendering */}
{showForm && (
   <form onSubmit={handleSubmit}>
      <h2>Visitor Form</h2>
      {/* Controlled form */}
      <label>
         Name:
      </label>
      <input
         type=”text”
         value={name}
         onChange={(event) =>
            setName(event.target.value)
         }
      />
      <button type=”submit”>
```

SGOU

```
                Submit
              </button>
            </form>
          )}
          {/* Conditional display of message */}
          {message && (
            <p>{message}</p>
          )}
        </div>
    );
}
export default App;
```

Output

The simplest setup is:

npm create vite@latest react-lab -- --template react

Then:

cd react-lab

npm install

npm run dev

You will get a local address such as:

http://localhost:5173/

Open that address in the browser.

Then replace the contents of:

src/App.jsx

SGOU

Kerala Explorer

Popular Destinations

Munnar

District: Idukki

Type: Hill Station

Wayanad

District: Wayanad

Type: Hill Station

Alappuzha

District: Alappuzha

Type: Backwaters

Show Form

Kerala Explorer

Popular Destinations

Munnar

District: Idukki

Type: Hill Station

Wayanad

District: Wayanad

Type: Hill Station

Alappuzha

District: Alappuzha

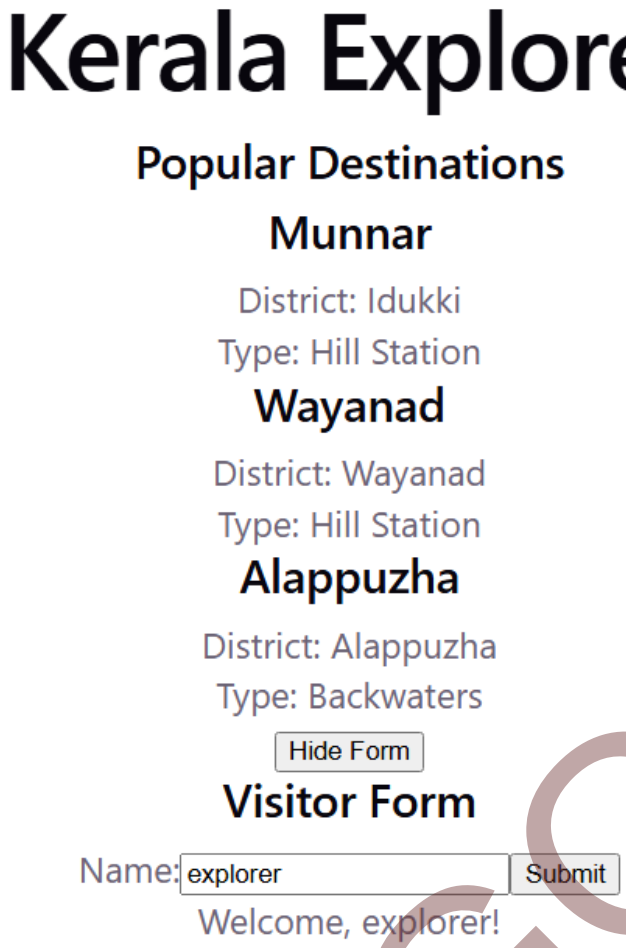
Type: Backwaters

Hide Form

Visitor Form

Name: Submit

SGOU



The implementation demonstrates functional components, JSX, props, state, event handling, controlled forms, and conditional rendering.

2.4.3 Lab Practice Questions

Develop a React application using a functional component to display information about a student.

1. Create a React application that passes information from a parent component to a child component using props.

2. Develop a React component that uses state to maintain and update a value.

3. Create a React application that responds to a button click using event handling.

4. Develop a controlled form using React state to accept and display user information.

SGOU

5. Create a React component that displays different content based on a condition.

6. Develop a React application combining functional components, JSX, props, state, event handling, controlled forms, and conditional rendering.

SGOU

Experiment 5

React Routing and Full-Stack Web Application Development

Objectives

- Understand routing in React applications.
- Configure routes using React Router.
- Navigate between different views of a React application.
- Connect a React frontend with an Express backend.
- Communicate with backend REST APIs from the React application.
- Integrate the frontend, backend, and database.
- Develop a basic full-stack web application.

2.5.1 Theory

2.5.1.1 React Router

A React application can contain multiple views or pages. React Router is used to implement client-side routing and navigation between these views.

React Router allows different URL paths to display different React components.

For example:

/ → Home

/destinations → Destinations

/about → About

2.5.1.2 Installing React Router

React Router can be installed in a React project using:

npm install react-router-dom

2.5.1.3 Routes

Routes define which component should be displayed for a particular URL path.

SGOU

Example:

```
import {
    BrowserRouter,
    Routes,
    Route
} from “react-router-dom”;
Routes can then be defined as:
<BrowserRouter>
    <Routes>
        <Route
            path=”/”
            element={<Home />}
        />
        <Route
            path=”/destinations”
            element={<Destinations />}
        />
    </Routes>
</BrowserRouter>
```

2.5.1.4 Navigation

The Link component can be used to navigate between routes without reloading the complete webpage.

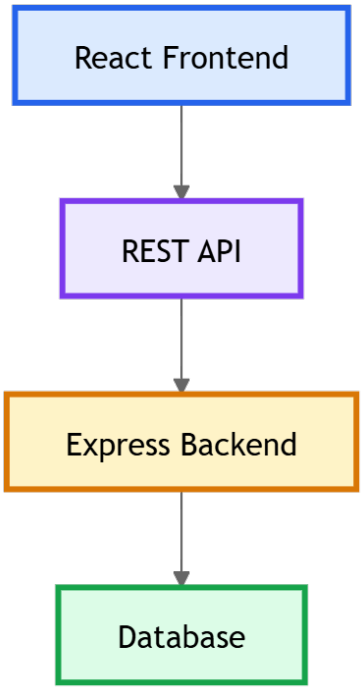
```
import { Link } from “react-router-dom”;
<Link to=”/”>Home</Link>
<Link to=”/destinations”>
    Destinations
</Link>
```

SGOU

2.5.1.5 Full-Stack Integration

A full-stack web application consists of the frontend, backend, and database.

The components communicate as follows:



The React frontend sends HTTP requests to the Express backend. The backend processes the request and communicates with the database. The result is then returned to the React application.

2.5.1.6 React and REST API Integration

The React frontend can communicate with a REST API using the Fetch API.

Example:

```
fetch("http://localhost:3000/destinations")
  .then(response => response.json())
  .then(data => {
    console.log(data);
  });
```

The retrieved data can then be displayed by React.

2.5.2 Implementation

Implement routing using React Router and integrate the frontend, backend, REST APIs, and database to build a full-stack web application.

SGOU

Develop a basic full-stack application using:

- React for the frontend
- React Router for navigation
- Express for the backend
- REST API for communication
- MongoDB for database storage

The application will provide separate pages for Home and Destinations and retrieve destination information from the backend.

Part A — Backend

Create a backend project:

```
mkdir kerala-explorer-backend
cd kerala-explorer-backend
npm init -y
npm install express mongoose cors
```

Create app.js:

```
const express = require("express");
const mongoose = require("mongoose");
const cors = require("cors");
const app = express();
const PORT = 3000;
app.use(cors());
app.use(express.json());
// Database connection
mongoose.connect(
    "mongodb://127.0.0.1:27017/kerala_explorer"
)
.then(() => {
    console.log("Database connected");
```

SGOU

```
})
.catch((error) => {
    console.log(
        “Database connection failed:”,
        error
    );
});
const destinationSchema = new mongoose.Schema({
    name: {
        type: String,
        required: true
    },
    district: {
        type: String,
        required: true
    },
    category: {
        type: String,
        required: true
    }
});
// Model
const Destination =
    mongoose.model(“Destination”, destinationSchema);
// CREATE
app.post(“/destinations”, async (req, res) => {
    try {
        const destination =
```

SGOU

```
            await Destination.create(req.body);
        res.status(201).json(destination);
    } catch (error) {
        res.status(400).json({
            message: "Unable to create destination"
        });
    }
});
// READ
app.get("/destinations", async (req, res) => {
    try {
        const destinations =
            await Destination.find();
        res.status(200).json(destinations);
    } catch (error) {
        res.status(500).json({
            message: "Unable to retrieve destinations"
        });
    }
});
// READ single destination
app.get("/destinations/:id", async (req, res) => {
    try {
        const destination =
            await Destination.findById(req.params.id);
        if (!destination) {
            return res.status(404).json({
                message: "Destination not found"
```

SGOU

```
        });
      }
      res.status(200).json(destination);
    } catch (error) {
      res.status(400).json({
        message: "Unable to retrieve destination"
      });
    }
});
// UPDATE
app.put("/destinations/:id", async (req, res) => {
  try {
    const destination =
      await Destination.findByIdAndUpdate(
        req.params.id,
        req.body,
        {
          new: true,
          runValidators: true
        }
      );
    if (!destination) {
      return res.status(404).json({
        message: "Destination not found"
      });
    }
    res.status(200).json(destination);
  } catch (error) {
```

SGOU

```
        res.status(400).json({
            message: “Unable to update destination”
        });
    }
});
// DELETE
app.delete(“/destinations/:id”, async (req, res) => {
    try {
        const destination =
            await Destination.findByIdAndDelete(
                req.params.id
            );
        if (!destination) {
            return res.status(404).json({
                message: “Destination not found”
            });
        }
        res.status(200).json({
            message: “Destination deleted successfully”
        });
    } catch (error) {

        res.status(400).json({
            message: “Unable to delete destination”
        });
    }
});
// Start server
```

SGOU

```
app.listen(PORT, () => {
    console.log(
        `Server running at http://localhost:${PORT}`
    );
});
```

Run the backend:

node app.js

The backend runs at:

http://localhost:3000

Alternative Version

First create a table using the following query :

```
CREATE TABLE destinations (
        id SERIAL PRIMARY KEY,
        name TEXT NOT NULL,
        district TEXT NOT NULL,
        category TEXT NOT NULL
    );
```

This uses postgres

First install pg using:

npm install pg

```
const express = require("express");
const cors = require("cors");
const { Pool } = require("pg");
const app = express();
const PORT = 3000;
const pool = new Pool({
    connectionString:
        "postgresql://postgres:<password>@localhost:5432/kerala_explorer"
```

SGOU

```
});
app.use(cors());
app.use(express.json());
// CREATE
app.post("/destinations", async (req, res) => {
  const { name, district, category } = req.body;
  try {
    const result = await pool.query(
      `INSERT INTO destinations (name, district, category)
       VALUES ($1, $2, $3)
       RETURNING id, name, district, category, id AS _id`,
      [name, district, category]
    );
    res.status(201).json(result.rows[0]);
  } catch (error) {
    console.log(error);
    res.status(400).json({
      message: "Unable to create destination"
    });
  }
});
// READ all
app.get("/destinations", async (req, res) => {
  try {
    const result = await pool.query(
      `SELECT id, name, district, category, id AS _id
       FROM destinations
       ORDER BY id`
```

SGOU

```
        );
        res.status(200).json(result.rows);
    } catch (error) {
        console.log(error);
        res.status(500).json({
            message: "Unable to retrieve destinations"
        });
    }
});
// READ single destination
app.get("/destinations/:id", async (req, res) => {
    try {
        const result = await pool.query(
            `SELECT id, name, district, category, id AS _id
             FROM destinations
             WHERE id = $1`,
            [req.params.id]
        );
        if (result.rows.length === 0) {
            return res.status(404).json({
                message: "Destination not found"
            });
        }
        res.status(200).json(result.rows[0]);
    } catch (error) {
        console.log(error);
        res.status(400).json({
            message: "Unable to retrieve destination"
```

SGOU

```
        });
    }
});
// UPDATE
app.put("/destinations/:id", async (req, res) => {
    const { name, district, category } = req.body;
    try {
        const result = await pool.query(
            `UPDATE destinations
             SET name = $1, district = $2, category = $3
             WHERE id = $4
             RETURNING id, name, district, category, id AS _id`,
            [name, district, category, req.params.id]
        );
        if (result.rows.length === 0) {
            return res.status(404).json({
                message: "Destination not found"
            });
        }
        res.status(200).json(result.rows[0]);
    } catch (error) {
        console.log(error);
        res.status(400).json({
            message: "Unable to update destination"
        });
    }
});
// DELETE
app.delete("/destinations/:id", async (req, res) => {
```

SGOU

```
    try {
        const result = await pool.query(
            `DELETE FROM destinations
             WHERE id = $1
             RETURNING id`,
            [req.params.id]
        );
        if (result.rows.length === 0) {
            return res.status(404).json({
                message: "Destination not found"
            });
        }
        res.status(200).json({
            message: "Destination deleted successfully"
        });
    } catch (error) {
        console.log(error);
        res.status(400).json({
            message: "Unable to delete destination"
        });
    }
});
// Start server
app.listen(PORT, () => {
    console.log(
        `Server running at http://localhost:${PORT}`
    );
});
```

SGOU

Part B — React Frontend

Create a React project using Vite:

npm create vite@latest kerala-explorer -- --template react

Move into the project:

cd kerala-explorer

npm install

npm install react-router-dom

Start the application:

npm run dev

src/App.jsx

```
import {
  BrowserRouter,
  Routes,
  Route,
  Link,
  useNavigate,
 useParams
} from "react-router-dom";
import { useEffect, useState } from "react";
const API_URL = "http://localhost:3000/destinations";
// Home Component
function Home() {
  return (
    <div>
      <h1>Kerala Explorer</h1>
      <p>
        Explore and manage popular destinations
        of Kerala.
      </p>
```

SGOU

```
      </div>
    );
}
```

// Destinations Component

```
function Destinations() {
    const [destinations, setDestinations] = useState([]);
    const [loading, setLoading] = useState(true);
    const loadDestinations = () => {
        fetch(API_URL)
            .then(response => response.json())
            .then(data => {
                setDestinations(data);
                setLoading(false);
            })
            .catch(error => {
                console.log(error);
                setLoading(false);
            });
    };
    useEffect(() => {
        loadDestinations();
    }, []);
    // DELETE
    const deleteDestination = (id) => {
        fetch(`${API_URL}/${id}`, {
            method: "DELETE"
        })
```

SGOU

```
        .then(response => response.json())
        .then(() => {
          loadDestinations();
        })
        .catch(error => {
          console.log(error);
        });
    };
    return (
      <div>
        <h1>Destinations</h1>
        <Link to="/add">
          <button>Add Destination</button>
        </Link>
        {loading ? (
          <p>Loading destinations...</p>
        ) : destinations.length === 0 ? (
          <p>No destinations available.</p>
        ) : (
          <table border="1" cellPadding="10">
            <thead>
              <tr>
                <th>Name</th>
                <th>District</th>
                <th>Category</th>
                <th>Actions</th>
              </tr>
            </thead>
```

SGOU

```
<tbody>
  {destinations.map(destination => (
    <tr key={destination._id}>
      <td>
        {destination.name}
      </td>
      <td>
        {destination.district}
      </td>
      <td>
        {destination.category}
      </td>
      <td>
        <Link
          to={`/edit/${destination._id}`}
        >
          <button>
            Edit
          </button>
        </Link>

        <button
          onClick={() =>
            deleteDestination(
              destination._id
            )
          }
        >
```

SGOU

```
                                Delete
                            </button>
                        </td>
                    </tr>
                ))}
            </tbody>
        </table>
    )}
    </div>
  );
}
// Add Destination Component
function AddDestination() {
  const navigate = useNavigate();
  const [form, setForm] = useState({
    name: "",
    district: "",
    category: ""
  });
  const handleChange = (event) => {
    setForm({
      ...form,
      [event.target.name]: event.target.value
    });
  };
  // CREATE
  const handleSubmit = (event) => {
    event.preventDefault();
```

SGOU

```
    fetch(API_URL, {
      method: “POST”,
      headers: {
        “Content-Type”: “application/json”
      },
      body: JSON.stringify(form)
    })
      .then(response => response.json())
      .then(() => {
        navigate(“/destinations”);
      })
      .catch(error => {
        console.log(error);
      });
};
return (
  <div>
    <h1>Add Destination</h1>
    <form onSubmit={handleSubmit}>
      <label>Name:</label>
      <input
        type=”text”
        name=”name”
        value={form.name}
        onChange={handleChange}
        required
      />
      <br /><br />
```

SGOU

```
            <label>District:</label>
            <input
              type="text"
              name="district"
              value={form.district}
              onChange={handleChange}
              required
            />
            <br /><br />
            <label>Category:</label>
            <input
              type="text"
              name="category"
              value={form.category}
              onChange={handleChange}
              required
            />
            <br /><br />
            <button type="submit">
              Add Destination
            </button>
          </form>
        </div>
    );
}
// Edit Destination Component
function EditDestination() {
  const { id } = useParams();
```

SGOU

```
const navigate = useNavigate();
 const [form, setForm] = useState({
    name: “”,
    district: “”,
    category: “”
 });
 // READ - Load selected destination
 useEffect(() => {
    fetch(`${API_URL}/${id}`)
       .then(response => response.json())
       .then(data => {
          setForm({
             name: data.name,
             district: data.district,
             category: data.category
          });
       })
       .catch(error => {
          console.log(error);
       });
 }, [id]);
 const handleChange = (event) => {
    setForm({
       ...form,
       [event.target.name]: event.target.value
    });
 };
 // UPDATE
```

SGOU

```
const handleSubmit = (event) => {
    event.preventDefault();
    fetch(`${API_URL}/${id}`, {
        method: "PUT",
        headers: {
            "Content-Type": "application/json"
        },
        body: JSON.stringify(form)
    })
        .then(response => response.json())
        .then(() => {
            navigate("/destinations");
        })
        .catch(error => {
            console.log(error);
        });
};
return (
    <div>
        <h1>Edit Destination</h1>
        <form onSubmit={handleSubmit}>
            <label>Name:</label>
            <input
                type="text"
                name="name"
                value={form.name}
                onChange={handleChange}
                required
```

SGOU

```
            />
            <br /><br />
            <label>District:</label>
            <input
               type="text"
               name="district"
               value={form.district}
               onChange={handleChange}
               required
            />
            <br /><br />
            <label>Category:</label>
            <input
               type="text"
               name="category"
               value={form.category}
               onChange={handleChange}
               required
            />
            <br /><br />
            <button type="submit">
               Update Destination
            </button>
         </form>
      </div>
   );
}
// Main Application
```

SGOU

```
function App() {
  return (
    <BrowserRouter>
      <nav>
        <Link to="/">
          Home
        </Link>
        {" | "}
        <Link to="/destinations">
          Destinations
        </Link>
      </nav>
      <hr />
      <Routes>
        <Route
          path="/"
          element={<Home />}
        />
        <Route
          path="/destinations"
          element={<Destinations />}
        />
        <Route
          path="/add"
          element={<AddDestination />}
        />
        <Route
          path="/edit/:id"
```

SGOU

```
                element={<EditDestination />}
            />
          </Routes>
        </BrowserRouter>
    );
}
export default App;
```

Part C — Database Data

Add some destination records to the MongoDB database using the API developed in Experiment 3.

For example:

```
{
    "name": "Munnar",
    "district": "Idukki",
    "category": "Hill Station"
}
```

Additional records may be added similarly.

Part D — Run the Full-Stack Application

Run the backend first:

node app.js

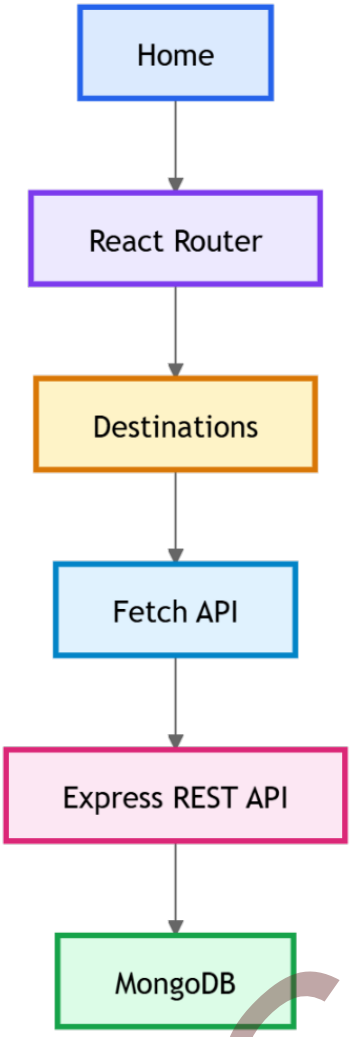
Then run the React frontend:

npm run dev

Open the React application in the browser using the URL provided by Vite.

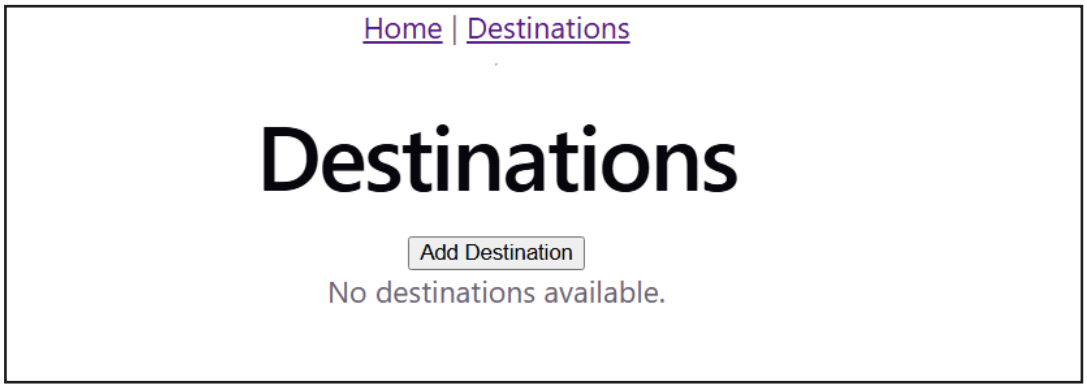
SGOU

The application provides:

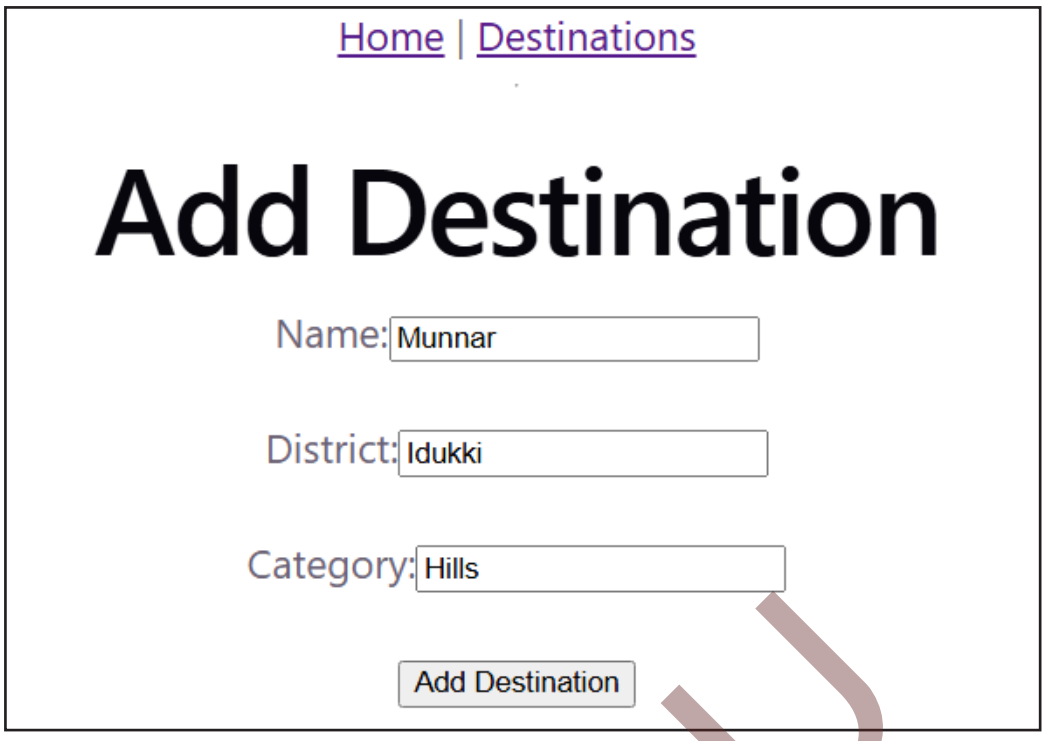


Output

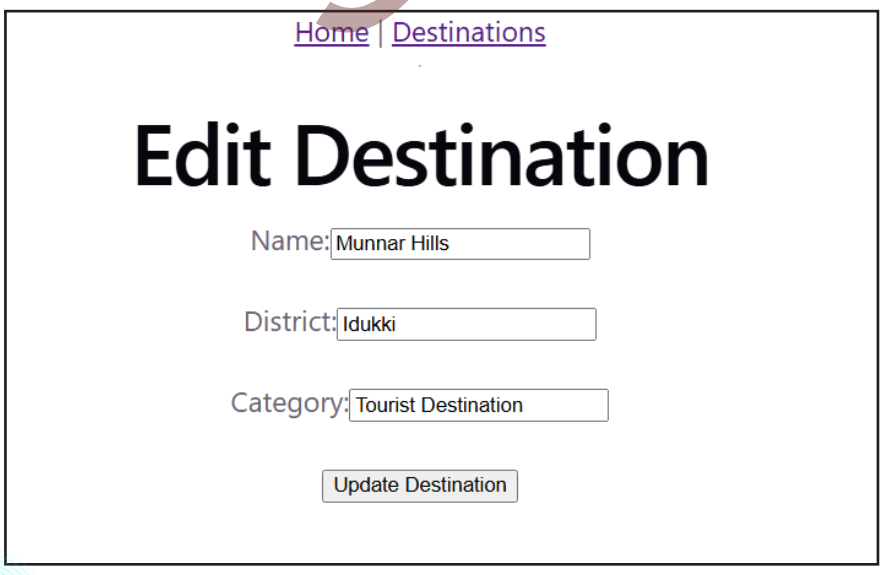
The **Home** route displays the home page of Kerala Explorer.



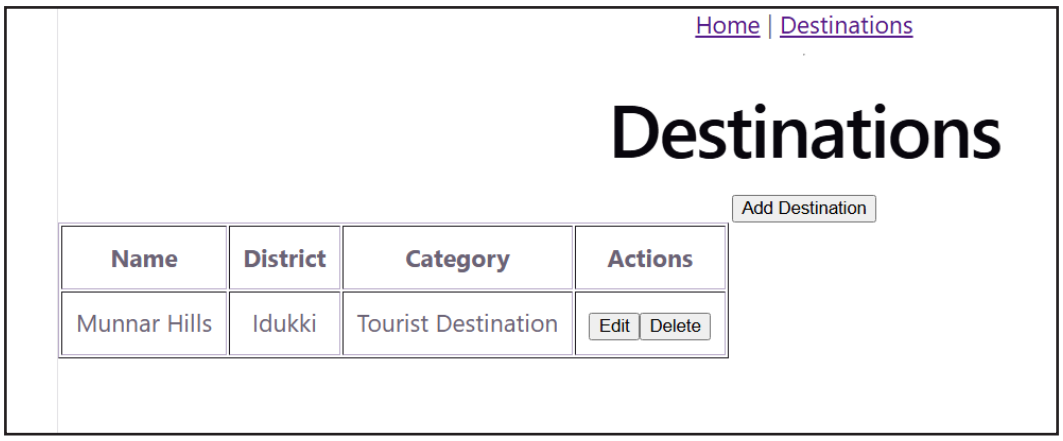
SGOU

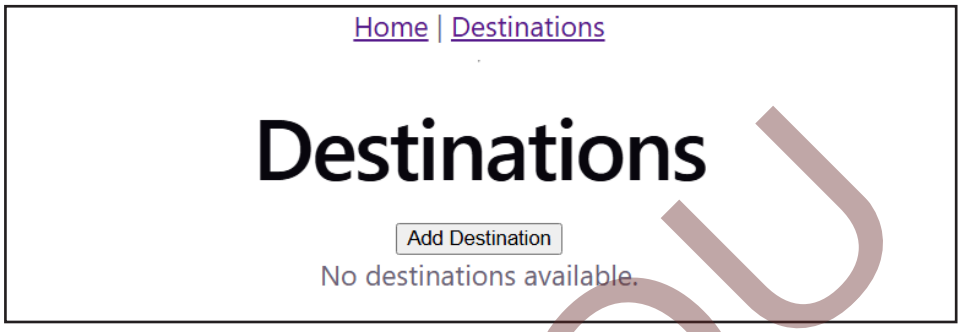

Home | Destinations
Add Destination
Name: Munnar
District: Idukki
Category: Hills
Add Destination


Home | Destinations
Destinations
Add Destination
Name
District
Category
Actions
Munnar
Idukki
Hills
Edit
Delete


Home | Destinations
Edit Destination
Name: Munnar Hills
District: Idukki
Category: Tourist Destination
Update Destination

SGOU





2.5.3 Lab Practice Questions

1. Create a React application with separate routes for Home and About pages.
2. Implement navigation between React components using React Router.
3. Develop a React application that retrieves data from a REST API and displays it.
4. Integrate a React frontend with an Express backend.
5. Connect a React application to a REST API that retrieves records from a database.
6. Develop a full-stack application integrating React, React Router, Express, REST API, and a database.
7. Develop a full-stack web application with multiple React routes that retrieves and displays database information through a REST API.

SGOU

Appendix

1.1 Integrated Example: Kerala Explorer (Block 1: Unit 1)

The concepts introduced in this unit can now be combined to construct a simple tourism webpage.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <!-- Defines the character encoding -->
    <meta charset="UTF-8">
    <!-- Makes the webpage responsive on different devices -->
    <meta name="viewport"
        content="width=device-width, initial-scale=1.0">
    <!-- Title displayed in the browser tab -->
    <title>Kerala Explorer</title>
</head>
<body>
    <!-- Header section of the webpage -->
    <header>
        <h1>Kerala Explorer</h1>
        <p>Explore the beautiful destinations of Kerala</p>
        <!-- Navigation menu -->
        <nav>
            <a href="#destinations">Destinations</a> |
            <a href="#experiences">Experiences</a> |
            <a href="#enquiry">Tourist Enquiry</a>
        </nav>
    </header>
    <!-- Main content of the webpage -->
```

SGOU

```
<main>
  <!-- Section displaying popular tourist destinations -->
  <section id="destinations">
    <h2>Popular Destinations</h2>
    <!-- Image representing a destination -->
    <img src="munnar.jpg"
      alt="Scenic landscape of Munnar"
      width="400">
    <p>
      Kerala offers a variety of destinations known for
      their natural beauty, culture and wildlife.
    </p>
    <!-- Table used to represent structured information -->
    <table border="1">
      <!-- Table header row -->
      <tr>
        <th>Destination</th>
        <th>Description</th>
      </tr>
      <!-- Destination: Munnar -->
      <tr>
        <td>Munnar</td>
        <td>Hill station</td>
      </tr>
      <!-- Destination: Wayanad -->
      <tr>
        <td>Wayanad</td>
        <td>Nature destination</td>
```

SGOU

```
        </tr>
        <!-- Destination: Alappuzha -->
        <tr>
          <td>Alappuzha</td>
          <td>Backwater destination</td>
        </tr>
        <!-- Destination: Thekkady -->
        <tr>
          <td>Thekkady</td>
          <td>Wildlife destination</td>
        </tr>
      </table>
    </section>
    <!-- Section containing tourist experiences -->
    <section id="experiences">
      <h2>Popular Experiences</h2>
      <!-- Unordered list used to organize information -->
      <ul>
        <li>Explore tea plantations</li>
        <li>Enjoy Kerala backwaters</li>
        <li>Visit wildlife sanctuaries</li>
        <li>Experience local culture</li>
      </ul>
    </section>
    <!-- Section containing the tourist enquiry form -->
    <section id="enquiry">
      <h2>Tourist Enquiry</h2>
      <!-- Form used to collect information from visitors -->
```

SGOU

```
<form method="post">
    <!-- Text input for visitor's name -->
    <label for="name">Name:</label>
    <input type="text"
        id="name"
        name="name">
    <!-- Email input for visitor's email address -->
    <label for="email">Email:</label>
    <input type="email"
        id="email"
        name="email">
    <!-- Dropdown list for selecting a destination -->
    <label for="destination">
      Preferred Destination:
    </label>
    <select id="destination"
         name="destination">
      <option value="Munnar">Munnar</option>
      <option value="Wayanad">Wayanad</option>
      <option value="Alappuzha">Alappuzha</option>
      <option value="Thekkady">Thekkady</option>
    </select>
    <!-- HTML5 date input -->
    <label for="travelDate">
      Travel Date:
    </label>
    <input type="date"
        id="travelDate"
```

SGOU

```
                name="travelDate">
            <!-- Submit button -->
            <button type="submit">
              Submit Enquiry
            </button>
          </form>
        </section>
    </main>
    <!-- Footer section of the webpage -->
    <footer>
      <p>Kerala Explorer</p>
    </footer>
</body>
</html>
```

The example combines the major concepts discussed in this unit.

The <header> identifies the website and contains navigation. The <main> element contains the primary tourism information. Individual <section> elements organize the destination content, destination information and enquiry form.

The image provides visual information about Munnar. The list organizes tourist experiences. The table presents structured destination information. The form collects information from the visitor.

1.2 Building a Mobile-Friendly Kerala Explorer Layout (Block 1: Unit 2)

The concepts introduced in this unit can now be combined to create a responsive destination layout.

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
```

SGOU

```
    <!-- Specifies the character encoding used by the webpage -->
    <meta charset="UTF-8">
    <!-- Makes the webpage responsive on different screen sizes -->
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <!-- Title displayed in the browser tab -->
    <title>Kerala Explorer</title>
    <!-- Links the external CSS stylesheet -->
    <link rel="stylesheet" href="style.css">
</head>
<body>
    <!-- Header section containing the website title and navigation -->
    <header>
        <!-- Main heading of the webpage -->
        <h1>Kerala Explorer</h1>
        <!-- Navigation menu -->
        <nav>
            <!-- Navigation links -->
            <a href="#">Home</a>
            <a href="#">Munnar</a>
            <a href="#">Wayanad</a>
            <a href="#">Alappuzha</a>
        </nav>
    </header>
    <!-- Main content of the webpage -->
    <main>
        <!-- Section containing all destination cards -->
        <section class="destination-grid">
            <!-- Card for Munnar -->
```

SGOU

```
<article class="destination-card">
    <!-- Image representing Munnar -->
    <img src="munnar.jpg"
        alt="Scenic landscape of Munnar">
    <!-- Destination name -->
    <h2>Munnar</h2>
    <!-- Short description of Munnar -->
    <p>
        Explore the scenic hill landscapes
        and tea plantations of Munnar.
    </p>
</article>
<!-- Card for Wayanad -->
<article class="destination-card">
    <!-- Image representing Wayanad -->
    <img src="wayanad.jpg"
        alt="Landscape of Wayanad">
    <!-- Destination name -->
    <h2>Wayanad</h2>
    <!-- Short description of Wayanad -->
    <p>
        Discover the natural beauty and
        attractions of Wayanad.
    </p>
</article>
<!-- Card for Alappuzha -->
<article class="destination-card">
    <!-- Image representing Alappuzha -->
```

SGOU

```
            <img src="alappuzha.jpg"
                alt="Backwaters of Alappuzha">
            <!-- Destination name -->
            <h2>Alappuzha</h2>
            <!-- Short description of Alappuzha -->
            <p>
                Experience the scenic backwaters
                and traditional houseboats.
            </p>
        </article>
    </section>
</main>
<!-- Footer section -->
<footer>
    <!-- Copyright/website identification text -->
    <p>Kerala Explorer</p>
</footer>
</body>
</html>
```

SGOU

Analysis of the Example

The integrated example combines several CSS concepts introduced in this unit.

The external stylesheet separates presentation from HTML structure. The universal selector is used to establish consistent box sizing. The header and navigation are styled using colours, spacing and Flexbox.

The destination cards are arranged using CSS Grid. Three columns are displayed on larger screens, two columns on medium screens and one column on smaller screens.

The images are fluid because their width is controlled relative to their containing elements.

The media queries modify the layout according to viewport width. As a result, the same

Kerala Explorer webpage can provide a more appropriate layout across desktop, tablet and smartphone screens.

The box-sizing: border-box declaration makes the declared width and height of an element include its content, padding and border. This can make layout dimensions easier to control, particularly when building responsive layouts.

1.3 Integrated Example: Interactive Kerala Explorer (Block 1: Unit 3)

Interactive Destination Selector — HTML

```
<!DOCTYPE html>
<html lang=”en”>
<head>
    <!-- Specifies the character encoding -->
    <meta charset=”UTF-8”>
    <!-- Makes the webpage responsive on different devices -->
    <meta name=”viewport”
        content=”width=device-width, initial-scale=1.0”>
    <!-- Title displayed in the browser tab -->
    <title>Kerala Explorer</title>
</head>
<body>
    <!-- Main heading of the application -->
    <h1>Kerala Explorer</h1>
    <!-- Label for the destination dropdown -->
    <label for=”destination”>
        Select a destination:
    </label>
    <!-- Dropdown list for selecting a destination -->
    <select id=”destination”>
        <!-- Destination options -->
        <option value=”Munnar”>
```

SGOU

```
            Munnar
        </option>
        <option value="Wayanad">
            Wayanad
        </option>
        <option value="Alappuzha">
            Alappuzha
        </option>
    </select>
    <!-- Button that calls the JavaScript function when clicked -->
    <button onclick="showDetails()">
        View Details
    </button>
    <!-- Section used to display the selected destination details -->
    <section>
        <!-- JavaScript will update this heading -->
        <h2 id="destinationName">
            Destination
        </h2>
        <!-- JavaScript will update this paragraph -->
        <p id="destinationInfo">
            Select a destination to view details.
        </p>
    </section>
    <!-- Links the external JavaScript file -->
    <script src="script.js"></script>
</body>
</html>
```

SGOU

Interactive Destination Selector — JavaScript

```
// Select the dropdown element from the HTML
const destinationSelect =
    document.getElementById("destination");
// Select the heading where the destination name
// will be displayed
const destinationName =
    document.getElementById("destinationName");
// Select the paragraph where the destination
// description will be displayed
const destinationInfo =
    document.getElementById("destinationInfo");
// Store destination descriptions in a JavaScript object
// The destination name is used as the key
const descriptions = {
    Munnar:
        "Munnar is known for its scenic hill landscapes and tea plantations.",
    Wayanad:
        "Wayanad is known for its natural landscapes, forests and wildlife.",
    Alappuzha:
        "Alappuzha is known for its backwaters and traditional houseboats."
};
// Function executed when the "View Details" button is clicked
function showDetails() {
    // Get the value of the currently selected option
    const selectedDestination =
        destinationSelect.value;
    // Update the heading with the selected destination
```

SGOU

```
    destinationName.textContent =
        selectedDestination;
    // Get and display the corresponding description
    // from the descriptions object
    destinationInfo.textContent =
        descriptions[selectedDestination];
}
```

Analysis of the Example

The HTML provides the structure of the interactive application. The <select> element allows the user to choose a Kerala destination, while the <button> provides an interaction point. The <h2> and <p> elements act as output areas where JavaScript displays the selected destination and its description.

The JavaScript program accesses these HTML elements using document. getElementById(). The destination descriptions are stored in a JavaScript object, where each destination is used as a key and its description is stored as the corresponding value.

When the user selects a destination and clicks View Details, the showDetails() function is executed. The selected value is obtained using the .value property of the <select> element.

For example:

const selectedDestination = destinationSelect.value;

If the user selects **Munnar**, selectedDestination contains:

"Munnar"

JavaScript then uses this value to retrieve the corresponding description:

descriptions[selectedDestination]

which is equivalent to:

descriptions["Munnar"]

Finally, the webpage is updated using the textContent property:

destinationName.textContent = selectedDestination;

destinationInfo.textContent = descriptions[selectedDestination];

Therefore, the displayed content changes dynamically without reloading the webpage or manually changing the HTML document.

SGOU

1.4 Integrated Example: Interactive Kerala Explorer Enquiry (Block 1: Unit 4)

The concepts introduced in this unit can now be combined into a simple Kerala Explorer enquiry form.

HTML

```
<!DOCTYPE html>
<html lang=”en”>
<head>
   <!-- Specifies the character encoding -->
   <meta charset=”UTF-8”>
   <!-- Makes the webpage responsive -->
   <meta name=”viewport”
       content=”width=device-width, initial-scale=1.0”>
   <!-- Title displayed in the browser tab -->
   <title>Kerala Explorer</title>
</head>
<body>
   <!-- Main heading -->
   <h1>Kerala Explorer</h1>
   <!-- Enquiry form -->
   <form id=”enquiryForm”>
      <!-- Name input -->
      <label for=”name”>
         Name:
      </label>
      <input
         id=”name”
         type=”text”
         placeholder=”Enter your name”
```

SGOU

```
    >
<!-- Area for displaying name validation error -->
<p id="nameError"></p>
<!-- Email input -->
<label for="email">
    Email:
</label>
<input
    id="email"
    type="email"
    placeholder="Enter your email"
>
<!-- Area for displaying email validation error -->
<p id="emailError"></p>
<!-- Destination selection -->
<label for="destination">
    Destination:
</label>
<select id="destination">
    <!-- Default option -->
    <option value="">
        Select a destination
    </option>
    <option value="Munnar">
        Munnar
    </option>
    <option value="Wayanad">
        Wayanad
```

SGOU

```
        </option>
          <option value="Alappuzha">
            Alappuzha
          </option>
        </select>
        <!-- Area for displaying destination validation error -->
        <p id="destinationError"></p>
        <!-- Submit button -->
        <button type="submit">
          Send Enquiry
        </button>
    </form>
    <!-- Area for displaying the final form message -->
    <p id="formMessage"></p>
    <!-- Links the external JavaScript file -->
    <script src="script.js"></script>
</body>
</html>
```

SGOU

Analysis of the Example

The HTML provides the structure of the enquiry form.

The JavaScript program attaches a submit event listener to the form. When the tourist submits the form, preventDefault() prevents the browser's default submission behaviour.

The entered values are retrieved from the form controls. JavaScript then validates the name, email address and selected destination.

If an error is found, an appropriate message is displayed next to the relevant field.

If all values are valid, a confirmation message is displayed dynamically.

This example combines:

1. DOM selection;

2. event handling;
3. addEventListener();
4. preventDefault();
5. form validation;
6. regular expressions;
7. conditional logic;
8. DOM content modification; and
9. template literals.

2.1 Integrated Example: Kerala Explorer REST API (Block 2: Unit 1)

The concepts introduced in this unit can now be combined to create a basic REST API for Kerala Explorer.

The API will support:

GET /destinations

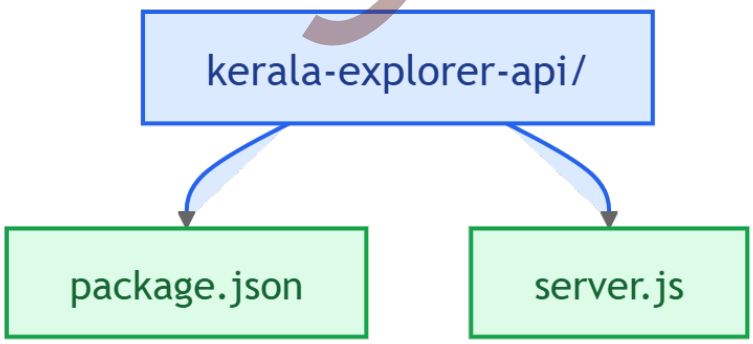
GET /destinations/:id

POST /destinations

PUT /destinations/:id

DELETE /destinations/:id

Project Structure



Step 1 – Initialize the Project

mkdir kerala-explorer-api

cd kerala-explorer-api

SGOU

npm init -y

Step 2 – Install Express

npm install express

Step 3 – Create server.js

```
const express = require("express");
const app = express();
app.use(
   express.json()
);
const destinations = [
   {
      id: 1,
      name: "Munnar",
      type: "Hill destination"
   },
   {
      id: 2,
      name: "Wayanad",
      type: "Nature destination"
   },
   {
      id: 3,
      name: "Alappuzha",
      type: "Backwater destination"
   }
];
```

GET – Retrieve All Destinations

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

SGOU

```
    function (request, response) {

        response.status(200).json(
            destinations
        );
    }
);
```

GET – Retrieve One Destination

```
app.get(
    "/destinations/:id",
    function (request, response) {
        const id =
            Number(request.params.id);
        const destination =
            destinations.find(
                function (item) {
                    return item.id === id;
                }
            );
        if (!destination) {
            return response.status(404).json(
                {
                    error:
                        "Destination not found"
                }
            );
        }
        response.status(200).json(
            destination
```

SGOU

```
    );
  }
);
```

POST – Create a Destination

```
app.post(
  "/destinations",
  function (request, response) {
    const destination = {
      id:
        destinations.length + 1,
      name:
        request.body.name,
      type:
        request.body.type
    };
    destinations.push(
      destination
    );
    response.status(201).json(
      destination
    );
  }
);
```

PUT – Update a Destination

```
app.put(
  "/destinations/:id",
  function (request, response) {
    const id =
      Number(request.params.id);
```

SGOU

```
    const destination =
      destinations.find(
        function (item) {
          return item.id === id;
        }
      );
    if (!destination) {
      return response.status(404).json(
        {
          error:
            "Destination not found"
        }
      );
    }
    destination.name =
      request.body.name;
    destination.type =
      request.body.type;
    response.status(200).json(
      destination
    );
  }
);
```

DELETE – Remove a Destination

```
app.delete(
  "/destinations/:id",
  function (request, response) {
    const id =
      Number(request.params.id);
```

SGOU

```
    const index =
      destinations.findIndex(
        function (item) {
          return item.id === id;
        }
      );
    if (index === -1) {
      return response.status(404).json(
        {
          error:
            "Destination not found"
        }
      );
    }
    const deleted =
      destinations.splice(
        index,
        1
      );
    response.status(200).json(
      deleted[0]
    );
  }
);
```

Error-Handling Middleware

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

SGOU

```
        response,
        next
    ) {
        console.error(error);
        response.status(500).json(
            {
                error:
                    “Internal server error”
            }
        );
    }
);
```

Start the Server

```
app.listen(
    3000,
    function () {
        console.log(
            “Kerala Explorer API running on port 3000”
        );
    }
);
```

Complete server.js

```
const express = require(“express”);
const app = express();
app.use(
    express.json()
);
const destinations = [
    {
```

SGOU

```
        id: 1,
        name: “Munnar”,
        type: “Hill destination”
    },
    {
        id: 2,
        name: “Wayanad”,
        type: “Nature destination”
    },
    {
        id: 3,
        name: “Alappuzha”,
        type: “Backwater destination”
    }
];
app.get(
    “/destinations”,
    function (request, response) {
        response.status(200).json(
            destinations
        );
    }
);
app.get(
    “/destinations/:id”,
    function (request, response) {
        const id =
            Number(request.params.id);
        const destination =
```

SGOU

```
        destinations.find(
            function (item) {
                return item.id === id;
            }
        );
    if (!destination) {
        return response.status(404).json(
            {
                error:
                    "Destination not found"
            }
        );
    }
    response.status(200).json(
        destination
    );
  }
);
let nextId = 4;
app.post(
  "/destinations",
  function (request, response) {
    const destination = {
        id:
            nextId++,
        name:
            request.body.name,
        type:
            request.body.type
```

SGOU

```
        };
        destinations.push(
            destination
        );
        response.status(201).json(
            destination
        );
    }
);
app.put(
    "/destinations/:id",
    function (request, response) {
        const id =
            Number(request.params.id);
        const destination =
            destinations.find(
                function (item) {
                    return item.id === id;
                }
            );
        if (!destination) {
            return response.status(404).json(
                {
                    error:
                        "Destination not found"
                }
            );
        }
        destination.name =
```

SGOU

```
            request.body.name;
        destination.type =
            request.body.type;
        response.status(200).json(
            destination
        );
    }
);
app.delete(
    "/destinations/:id",
    function (request, response) {
        const id =
            Number(request.params.id);
        const index =
            destinations.findIndex(
                function (item) {
                    return item.id === id;
                }
            );
        if (index === -1) {
            return response.status(404).json(
                {
                    error:
                        "Destination not found"
                }
            );
        }
        const deleted =
            destinations.splice(
```

SGOU

```
            index,
            1
        );
      response.status(200).json(
        deleted[0]
      );
  }
);
app.use(
  function (
    error,
    request,
    response,
    next
  ) {
    console.error(error);
    response.status(500).json(
      {
        error:
          "Internal server error"
      }
    );
  }
);
app.listen(
  3000,
  function () {
    console.log(
      "Kerala Explorer API running on port 3000"
```

SGOU

```
        );
    }
);
```

Testing the API

The following requests can be used to test the API:

GET

http://localhost:3000/destinations

GET

http://localhost:3000/destinations/1

POST

http://localhost:3000/destinations

Request body:

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

Update:

PUT

http://localhost:3000/destinations/1

Request body:

```
{
    "name": "Munnar",
    "type": "Hill and tea plantation destination"
}
```

Delete:

DELETE

http://localhost:3000/destinations/3

Analysis of the Example

The example demonstrates how Kerala Explorer can move from a frontend-only application to a basic backend service.

SGOU

The Express application:

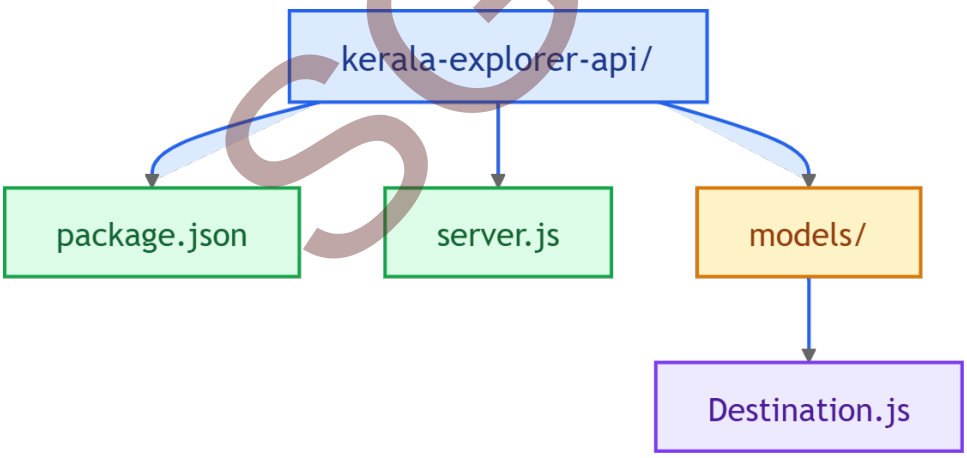
- creates an HTTP server;
- parses JSON request bodies;
- defines RESTful routes;
- retrieves destination information;
- creates new destinations;
- updates existing destinations;
- deletes destinations;
- returns appropriate status codes; and
- provides centralized error handling.

The destination data is currently stored in an in-memory array.

2.2 Integrated Example: Kerala Explorer with MongoDB (Block 2: Unit 2)

The concepts introduced in this unit can now be combined to create a database-backed Kerala Explorer REST API using Node.js, Express.js, Mongoose, MongoDB

Project Structure



Step 1 – Install Packages

npm install express mongoose

Step 2 – Create the Model

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

SGOU

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

Step 3 – Create server.js

```
const express =
    require("express");
const mongoose =
    require("mongoose");
const Destination =
    require(
        "./models/Destination"
    );
```

SGOU

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

Step 4 – Connect to MongoDB

```
mongoose.connect(
    “mongodb://127.0.0.1:27017/keralaExplorer”
)
.then(
    function () {
        console.log(
            “MongoDB connected”
        );
    }
)
.catch(
    function (error) {
        console.error(
            “Database connection failed”,
            error
        );
    }
);
```

Step 5 – GET All Destinations

```
app.get(
    “/destinations”,
    async function (
        request,
```

SGOU

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

Step 6 – POST a Destination

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

Step 7 – GET One Destination

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

SGOU

```
    request,
    response
  ) {
    const destination =
      await Destination.findById(
        request.params.id
      );
    if (!destination) {
      return response.status(404).json(
        {
          error:
            "Destination not found"
        }
      );
    }
    response.status(200).json(
      destination
    );
  }
);
```

Step 8 – Update a Destination

```
app.put(
  "/destinations/:id",
  async function (
    request,
    response
  ) {
    const destination =
      await Destination.findByIdAndUpdate(
```

SGOU

```
            request.params.id,
            request.body,
            {
              new: true
            }
          );
        if (!destination) {
          return response.status(404).json(
            {
              error:
                "Destination not found"
            }
          );
        }
        response.status(200).json(
          destination
        );
    }
);
```

Step 9 – Delete a Destination

```
app.delete(
    "/destinations/:id",
    async function (
        request,
        response
    ) {
        const destination =
            await Destination.findByIdAndDelete(
                request.params.id
```

SGOU

```
        );
      if (!destination) {
        return response.status(404).json(
          {
            error:
              "Destination not found"
          }
        );
      }
      response.status(200).json(
        {
          message:
            "Destination deleted successfully"
        }
      );
    }
);
app.use(
  function (
    error,
    request,
    response,
    next
  ) {
    console.error(error);
    response.status(500).json({
      error:
        "Internal server error"
    });
```

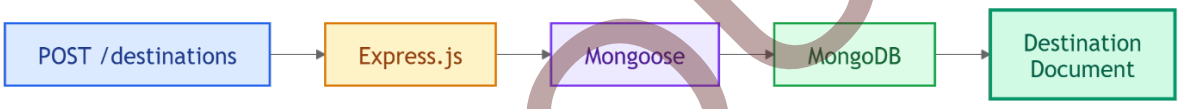
SGOU

```
    }
);
```

Step 10 – Start the Server

```
app.listen(
    3000,
    function () {
        console.log(
            "Kerala Explorer API running on port 3000"
        );
    }
);
```

Complete Flow



For retrieving data:



Testing the API

Testing Database-Backed REST Endpoints

The API can be tested using a REST client such as **Postman** or **Thunder Client**.

GET All Destinations

GET

http://localhost:3000/destinations

POST a Destination

POST

http://localhost:3000/destinations

Request body:

```
{
    "name": "Kumarakom",
```

SGOU

"type": "Backwater destination",

"district": "Kottayam"

}

GET One Destination

GET

http://localhost:3000/destinations/<id>

PUT a Destination

PUT

http://localhost:3000/destinations/<id>

Request body:

{

"type": "Backwater and bird sanctuary destination"

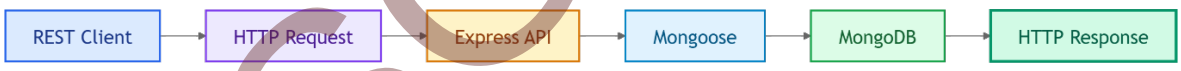
}

DELETE a Destination

DELETE

http://localhost:3000/destinations/<id>

Testing Flow



Interventions for Value Addition

From Temporary Data to Persistent Data

In Unit 1, Kerala Explorer used:

const destinations = [

{

id: 1,

name: "Munnar"

}

];

The information existed only in server memory.

In this unit, data is stored in the database and is therefore persisted beyond the

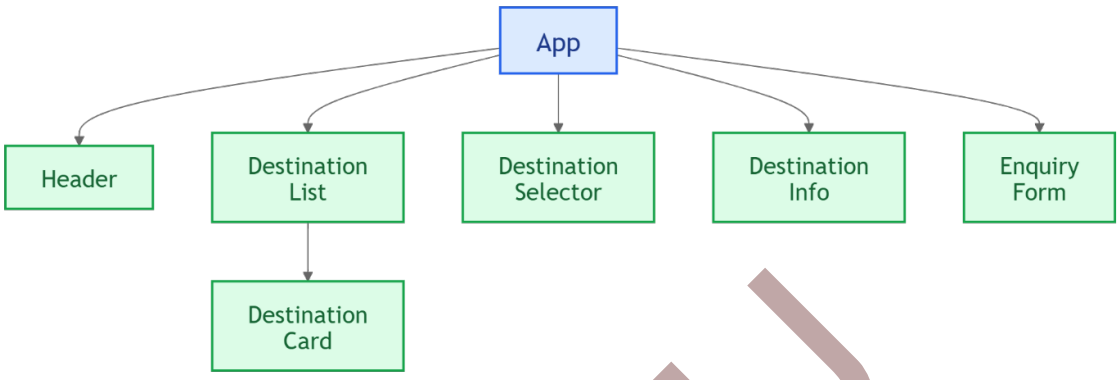
SGOU

lifetime of the server process.

2.3 Integrated Example: Kerala Explorer React Destination Explorer (Block 2: Unit 3)

The concepts introduced in this unit can be combined into a small React application.

The application will contain:



The example demonstrates:

- functional components;
- JSX;
- props;
- state;
- useState;
- conditional rendering;
- lists;
- keys;
- synthetic events;
- controlled form input; and
- lifting state up.

Complete Example

```
import {
  useState
```

SGOU

```
} from “react”;
const destinations = [
  {
    id: 1,
    name: “Munnar”,
    type: “Hill destination”,
    description:
      “Explore tea plantations and scenic viewpoints.”
  },
  {
    id: 2,
    name: “Wayanad”,
    type: “Nature destination”,
    description:
      “Discover forests, waterfalls and wildlife.”
  },
  {
    id: 3,
    name: “Alappuzha”,
    type: “Backwater destination”,
    description:
      “Experience scenic backwaters and houseboats.”
  }
];
function Header() {
  return (
```

SGOU

```
        <header>
          <h1>
            Kerala Explorer
          </h1>
          <p>
            Discover the beauty of Kerala.
          </p>
        </header>
    );
}
function DestinationCard(
    {
      destination,
      onSelect
    }
) {
    return (
      <article>
        <h2>
          {destination.name}
        </h2>
        <p>
          {destination.type}
        </p>
        <button
          onClick={
```

SGOU

```
                function () {
                  onSelect(
                    destination
                  );
                }
              }
            >
              View Details
            </button>
          </article>
        );
      }
      function DestinationList(
        {
          destinations,
          onSelect
        }
      ) {
        return (
          <section>
            <h2>
              Popular Destinations
            </h2>
            {
              destinations.map(
                function (
```

SGOU

```
                destination
            ) {
                return (
                    <DestinationCard
                        key={
                            destination.id
                        }
                        destination={
                            destination
                        }
                        onSelect={
                            onSelect
                        }
                    />
                );
            }
        )
    }
</section>
);
}
function DestinationInfo(
{
    destination
}
) {
```

SGOU

```
    if (!destination) {
      return (
        <p>
          Select a destination
          to view details.
        </p>
      );
    }
    return (
      <section>
        <h2>
          {destination.name}
        </h2>
        <p>
          {destination.description}
        </p>
      </section>
    );
}
function EnquiryForm() {
    const [
      name,
      setName
    ] = useState("");
    function handleSubmit(
      event
```

SGOU

```
) {
    event.preventDefault();
    alert(
      `Thank you, ${name}.`
    );
}
return (
    <form
      onSubmit={
        handleSubmit
      }
    >
      <label>
        Name:
      </label>
      <input
        type="text"
        value={name}
        onChange={
          function (event) {
            setName(
              event.target.value
            );
          }
        }
      />
```

SGOU

```
            <button
              type="submit"
            >
              Send Enquiry
            </button>
          </form>
        );
      }
      function App() {
        const [
          selectedDestination,
          setSelectedDestination
        ] = useState(null);
        return (
          <>
            <Header />
            <main>
              <DestinationList
                destinations={
                  destinations
                }
                onSelect={
                  setSelectedDestination
                }
              />
              <DestinationInfo
                destination={
```

SGOU

```
                selectedDestination
            }
          />
          <EnquiryForm />

          </main>
        </>
    );
  }
export default App;
```

How the Example Works

The App component stores:

selectedDestination

using useState.

The DestinationList receives the destination array through props.

Each destination is converted into a DestinationCard using map().

The key property identifies each card.

When the tourist clicks View Details, the child component calls the onSelect function received through props.

The parent updates:

selectedDestination

The updated value is then passed to DestinationInfo.

If no destination is selected, the component displays:

Select a destination to view details.

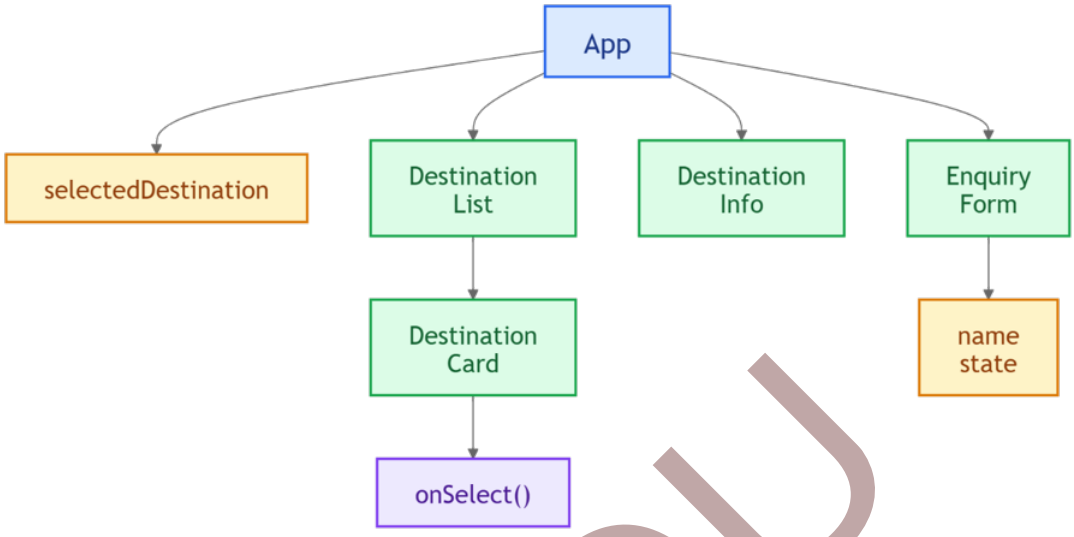
Once a destination is selected, its information is displayed.

SGOU

The EnquiryForm maintains the tourist's name using useState.

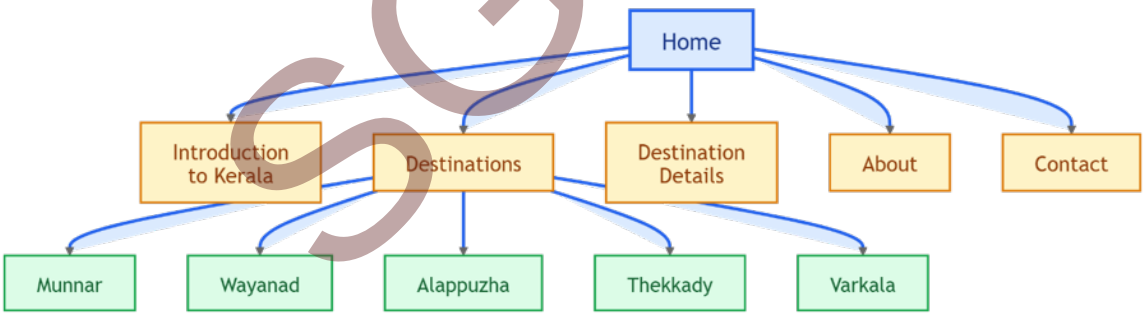
The input is therefore a controlled component.

Component Data Flow



2.4 Integrated Kerala Explorer Application (Block 2: Unit 4)

The final application can provide the following views:



The destination list is obtained from the backend.

The destination details page retrieves information using a destination ID.

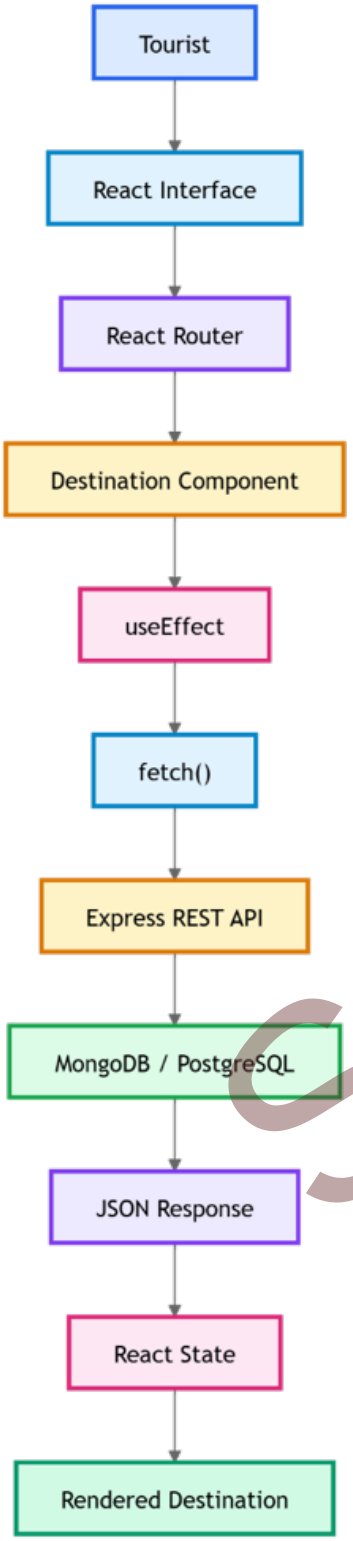
The navigation bar uses React Router.

The Express backend provides RESTful endpoints.

The database stores destination information.

SGOU

Complete Flow



SGOU

Deployment

Introduction to Deployment

Developing an application on a local computer makes it accessible through addresses

such as:

http://localhost:5173

for the React development server and:

http://localhost:3000

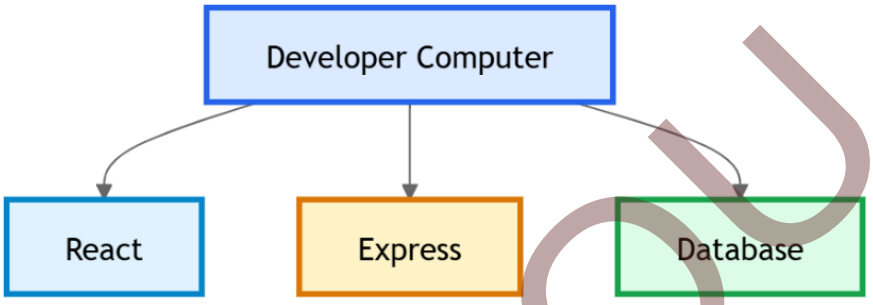
for the Express server.

However, a real application needs to be made available through a hosted environment.

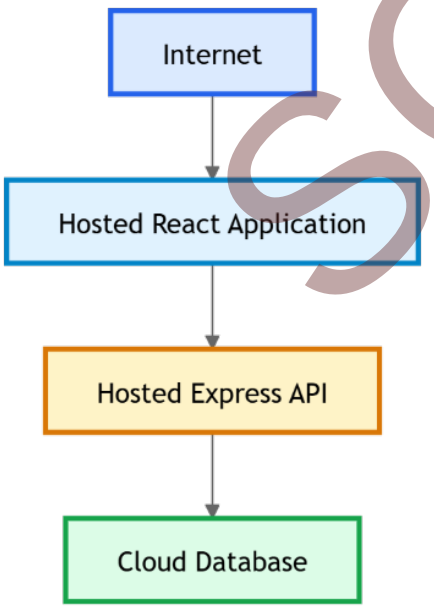
This process is called deployment.

A deployed Kerala Explorer application can be accessed by users through the Internet.

Local Development



Deployed Application

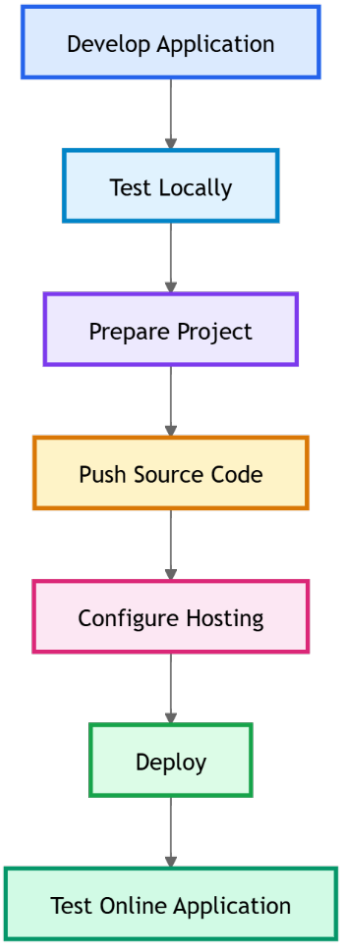


Basic Deployment Using Render or Railway

Platforms such as Render and Railway can be used for basic deployment of web applications.

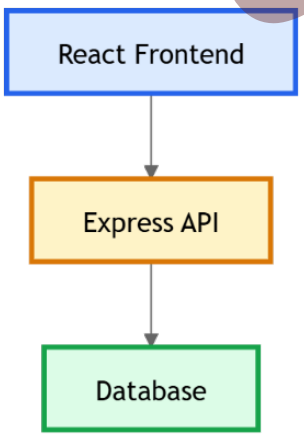
SGOU

A simplified deployment process is:



Before deployment, the application should be tested locally.

For example:



The frontend should use the deployed backend URL rather than:

http://localhost:3000

SGOU

Important Configuration

Database credentials and other environment-specific information should not be exposed directly in source code.

For example:

DATABASE_URL

MONGODB_URI

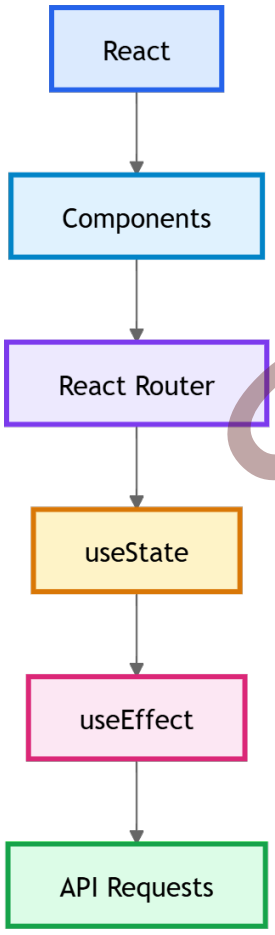
can be maintained as environment configuration.

Integrated Example

Complete Kerala Explorer Full-Stack Application

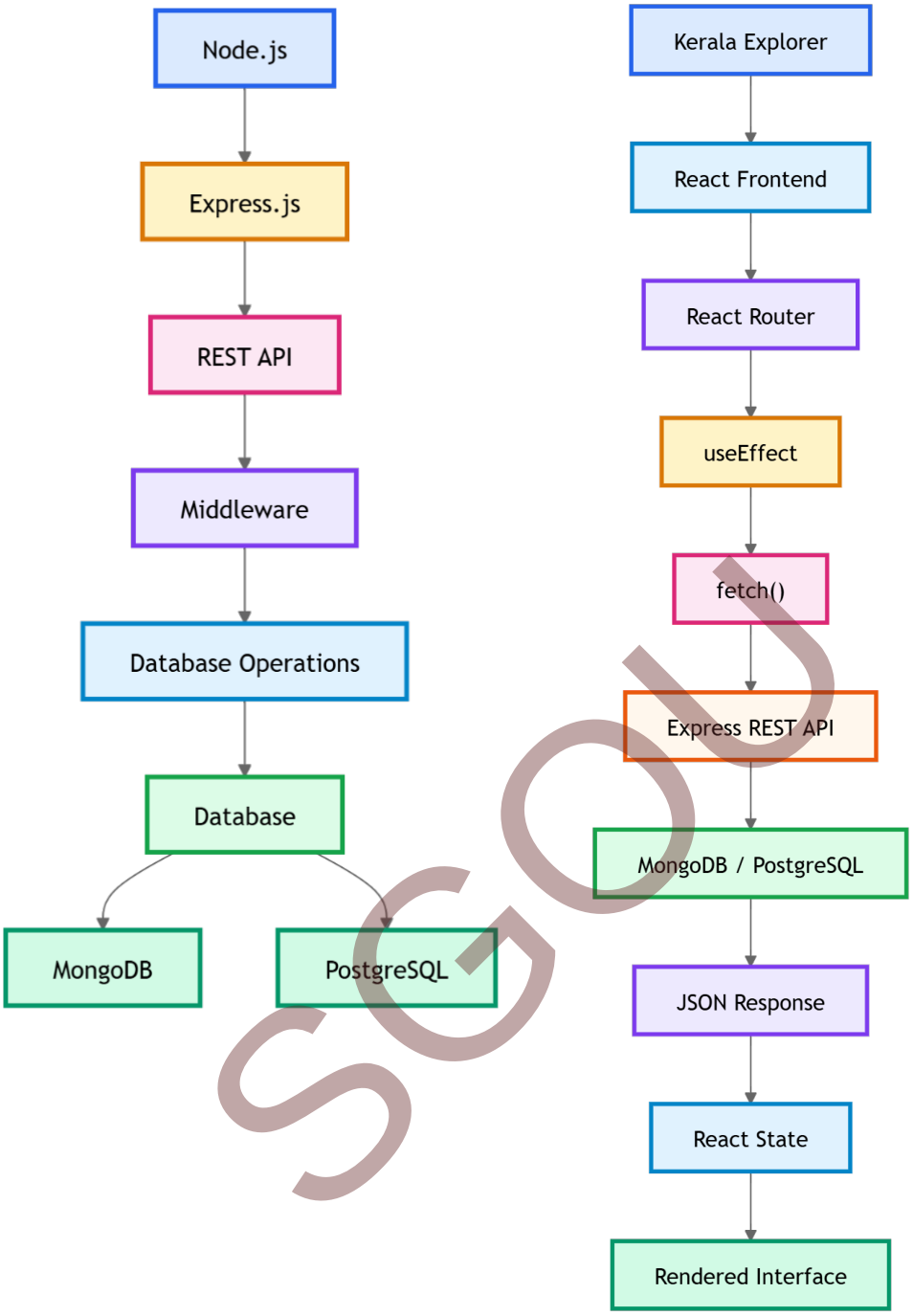
The complete application developed throughout Block II can now be summarized as follows.

Frontend



Backend

SGOU



Complete Architecture

SGOU

Example Request

A tourist opens:

/destinations

React Router displays the destination component.

The component uses useEffect() to request:

GET /destinations

The Express server receives the request.

The backend retrieves destination records from the database.

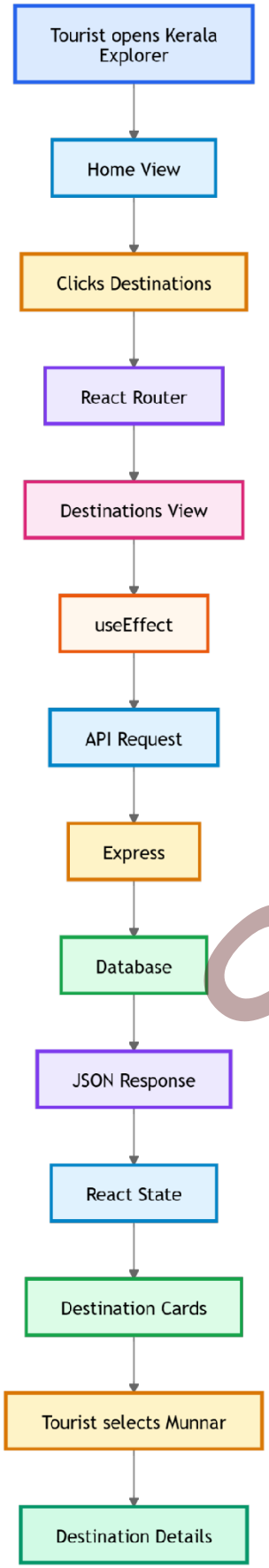
The server returns JSON:

```
[
  {
    “id”: 1,
    “name”: “Munnar”,
    “type”: “Hill destination”
  },
  {
    “id”: 2,
    “name”: “Wayanad”,
    “type”: “Nature destination”
  }
]
```

React stores the data in state and renders destination cards.

SGOU

Complete User Journey



SGOU

Kerala Explorer – From Frontend to Full Stack

The development of Kerala Explorer across Block II can be viewed as a progressive integration of different technologies. In Unit 1, Node.js and Express are used to develop the REST API. Unit 2 introduces MongoDB and PostgreSQL to provide persistent data storage. In Unit 3, React is introduced to develop the frontend using reusable components and state management. Unit 4 brings these concepts together by introducing React Router, useEffect, and API integration to create a complete full-stack application.

The final architecture of Kerala Explorer consists of a React frontend, an Express backend, and a MongoDB or PostgreSQL database. This demonstrates how different technologies can work together to develop a complete full-stack web application.

SGOU

സർവ്വകലാശാലാഗീതം

വിദ്യയാൽ സ്വതന്ത്രരാകണം
വിശ്വപൗരരായി മാറണം
ഗ്രഹപ്രസാദമായ് വിളങ്ങണം
ഗുരുപ്രകാശമേ നയിക്കണേ

കൂരിരുട്ടിൽ നിന്നു ഞങ്ങളെ
സൂര്യവീഥിയിൽ തെളിക്കണം
സ്നേഹദീപ്തിയായ് വിളങ്ങണം
നീതിവൈജയന്തി പാറണം

ശാസ്ത്രവ്യാപ്തിയെന്നുമേകണം
ജാതിഭേദമാകെ മാറണം
ബോധരശ്മിയിൽ തിളങ്ങുവാൻ
ജ്ഞാനകേന്ദ്രമേ ജ്വലിക്കണേ

കുരീപ്പുഴ ശ്രീകുമാർ

SGOU

SREENARAYANAGURU OPEN UNIVERSITY

Regional Centres

Kozhikode
Govt. Arts and Science College
Meenchantha, Kozhikode,
Kerala, Pin: 673002
Ph: 04952920228
email: rckdirector@sgou.ac.in

Thalassery
Govt. Brennen College
Dharmadam, Thalassery,
Kannur, Pin: 670106
Ph: 04902990494
email: rctdirector@sgou.ac.in

Tripunithura
Govt. College
Tripunithura, Ernakulam,
Kerala, Pin: 682301
Ph: 04842927436
email: rcedirector@sgou.ac.in

Pattambi
Sree Neelakanta Govt. Sanskrit College
Pattambi, Palakkad,
Kerala, Pin: 679303
Ph: 04662912009
email: rcpdirector@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

SGOU

Web Programming Lab

COURSE CODE: M26CA01PC

SGOU

Sreenarayanaguru Open University
Kollam, Kerala Pin- 691601, email: info@sgou.ac.in, www.sgou.ac.in Ph: +91 474 2966841

ISBN 978-81-181574-7-2