**INSTITUT D'ENSEIGNEMENT SUPÉRIEUR DE RUHENGERI**

*Accredited by Ministerial Order N° 005/2010/Mineduc of 16 June 2010*

**FACULTY OF SCIENCES AND INFORMATION TECHNOLOGY**

**DEPARTMENT OF MSWE**

*Scientia et Lux*

# Report for Advanced web development

**By:** **ABDULRAHIM Fadul** **MSWE002/24**

**DUSINGIZIMANA Jean Paul** **MSWE004/24**

**Supervisor: Dr. MUSABE Jean Bosco**

**Musanze, February 2025**



B.P. 155
Ruhengeri
Rwanda
**T** : +(250) 07 88 90 30 30
: +(250) 08 88 90 30 32
**E** :info@ines.ac.rw
**W** : www.ines.ac.rw

# LIST OF ABREVIATIONS

**Env**: environment

**Cd**: change directory

**CSS**: cascading style sheet

**App**: application

**CDN**: Content delivery network

**JS**: JavaScript

NPM: Node Package Manager

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# REPORT I: Report on Collaboration GitHub

## 1.1 Introduction

GitHub is a powerful platform for version control and collaborative development, widely used by developers to manage code, track changes, and facilitate teamwork. This report provides a comprehensive overview of the GitHub repository, including setup, workflow, key commands used, and reflections on the project. Fro this work the repository link is msweweb.dev

1.2 Repository setups and commands

### 1.2.1 Cloning the Repository:

To clone a Git repository, there are steps to follow this allow the user to copy someone's repository

Copy the Repository URL:

Get the repository URL from the Git hosting service (e.g., GitHub, GitLab, Bitbucket). It usually looks like:

1 git clone https://github.com/Dusipaul/msweweb.dev.git

Or with SSH git@github.com/Dusipaul/msweweb.dev.git

### 1.2.2 Navigating to the Project Directory:

In Terminal or CMD **Navigate to the Desired Directory:**

- Go to the folder where you want to clone the repository:

```
cd msweweb.dev
```

**1.2.3 Installing Dependencies:** If the project involves a specific technology stack (e.g., Node.js):

```
npm install
```

### 1.3 Git Workflow and collaboration

#### 1.3.1 Branching Strategy

- **Creating a Feature Branch:**

```
git checkout -b feature/branch-name
```

- **Committing Changes:**

```
git add .
git commit -m "Commit message"
```

- **Pushing Changes to Remote:**

```
git push origin feature/branch-name
```

#### 1.3.2 Merging Changes

- **Switching to Main Branch:**

```
git checkout main
```

- **Pulling Latest Changes:**

```
git pull origin main
```

- **Merging Feature Branch:**

```
git merge feature/branch-name
```

- **Resolving Conflicts:** Handled manually in the code editor, followed by:

```
git add . git add* (if you want to add all)
git commit -m "Resolved merge conflicts"
```

#### 1.3.3 Finalizing Changes

- **Pushing Merged Changes to Remote:**

```
git push origin main
```

- **Cleaning Up Branches:**

```
git branch -d feature/branch-name
```

### 1.4. Reflections

- **Challenges Faced:**
  - Merge conflicts and how they were resolved.
  - Managing contributions from multiple team members.
- **Lessons Learned:**
  - Importance of clear commit messages and regular pull requests.
  - Effective use of branching strategies to avoid conflicts.

### 1.5. Conclusion

This Task provided valuable insights into collaborative development using Git and GitHub. The documented workflow ensured a smooth integration of changes and highlighted the importance of version control best practices.

# REPORT II: Tailwind CSS

## 2.1. Introduction

This report provides an overview of Task #2, which involved building a responsive web page using Tailwind CSS. The page was collaboratively created with showcases key Tailwind features such as responsive design, interactive states, and utility-first styling.

## 2.2. Code Analysis

The provided HTML code implements a simple yet effective web page layout using Tailwind CSS. Key sections of the code include:

### 2.2.1 Header Section

The header utilizes responsive typography with Tailwind's `text-4xl`, `md:text-5xl`, and `lg:text-6xl` classes, ensuring the title scales well across devices. Centering and color styling are achieved with `text-center` and `text-blue-600` classes.

### 2.2.2 Card Grid

The main content features a grid layout, transitioning from a single column on small screens (`grid-cols-1`) to a three-column layout on large screens (`lg:grid-cols-3`). Each card includes a hover effect (`hover:shadow-xl`) and smooth transitions (`transition-shadow duration-300`).

### 2.2.3 Feature Highlights

- **Responsive Design:** Utilized `md:` and `lg:` breakpoints.
- **Flexbox & Grid:** Implemented using `grid` and `flex` classes.
- **Spacing and Sizing:** Classes like `p-8`, `mb-4`, and `px-4` offer precise control over spacing.
- **Colors and Typography:** Consistent use of Tailwind’s color palette and font styling (`text-xl`, `font-bold`).
- **Interactive States:** Hover and focus effects on buttons (`hover:bg-blue-600`, `focus:ring-2`).
- **Transitions and Effects:** The `transition-shadow` and `duration-300` classes contribute to a polished, interactive feel.

### 2.2.4 Footer

The footer remains simple and centered, utilizing Tailwind's margin and text utilities

(`mt-12`, `text-center`, `text-gray-600`).

## 2.3. Reflections

#### 2.2.3.1 Challenges

- Balancing design simplicity with responsiveness.
- Ensuring consistent styling across all components.

#### 2.2.3.2 Lessons Learned

- The utility-first approach of Tailwind CSS significantly speeds up styling.
- Responsive design is straightforward with Tailwind’s breakpoint system.
- Managing hover and focus states directly in HTML avoids complex CSS files.

## 2.3 Design and implementation

This enumerates the features of this demo app designed with Tailwind CSS. It provides the following:

1. **Responsive Layout for large screens**

The app is fully responsive, ensuring it adapts seamlessly to various screen sizes, including mobile, tablet, and desktop views.

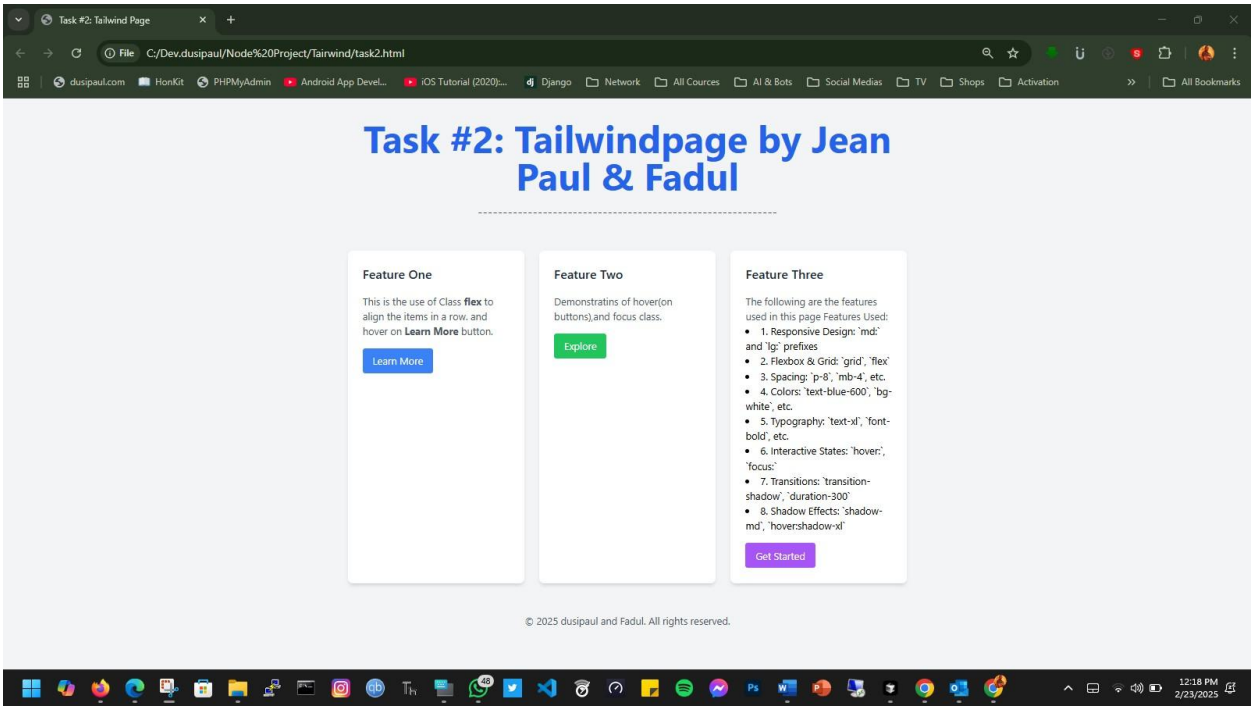



**Figure 1: Responsive Layout large window**

2. **Responsive Layout for small and medium screens**

For small and medium screens, a Responsive Layout ensures that the app remains functional and visually appealing. Key considerations include:

**Mobile-First Approach:** The app is designed with smaller screens in mind first, progressively enhancing the layout for larger screens.

**Grid System:** Utilizing Tailwind CSS's grid and flex utilities (`grid-cols-1`, `md:grid-cols-2`, etc.) to dynamically adjust content arrangement.

**Navigation:** Implementing a collapsible sidebar or a hamburger menu for easier navigation on small screens (`md:hidden` for mobile menus).

**Adaptive Components:** Buttons, cards, and text elements adjust size and padding based on screen size (`p-2 md:p-4`, `text-sm md:text-base`)(Taking Flight with Tailwind CSS, n.d.).

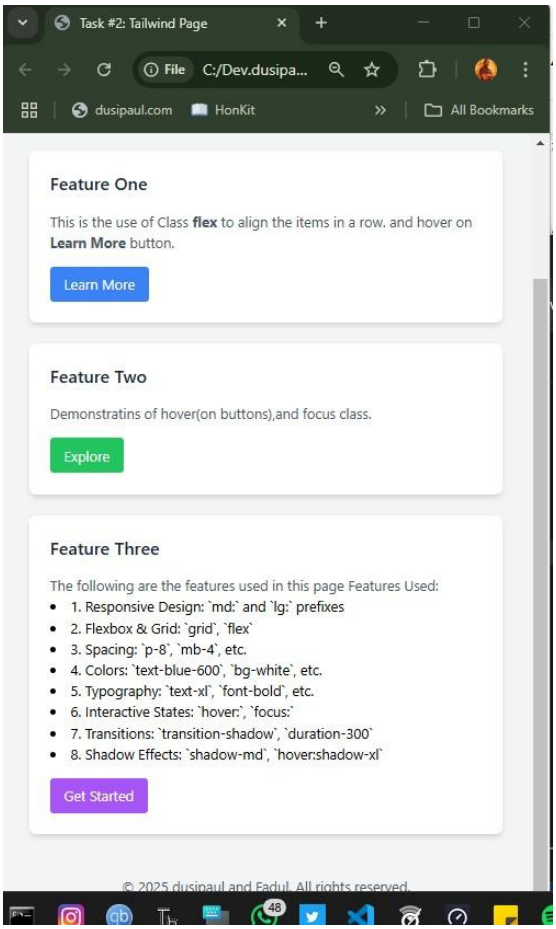



**Figure 2:Responsive Layout for small and medium screens**

**Customizable Components**: Tailwind's utility-first approach allows easy customization of elements such as buttons, navigation bars, and cards by applying different utility classes.

**Intuitive User Interface**: The app prioritizes user experience with clean, modern aesthetics and easy navigation, with elements designed to enhance usability.

**Dark Mode Support**: The app includes a built-in dark mode toggle, providing users with the option to switch between light and dark themes for a more personalized experience.

**Performance Optimization**: Tailwind CSS's purging mechanism ensures that only the necessary CSS is included in the final build, optimizing the app's performance and loading times.

**Easy Integration with JavaScript**: The design integrates well with JavaScript frameworks, making it flexible for dynamic features such as interactivity and data handling.

**Accessibility Features**: The app includes accessible design elements, ensuring compliance with WCAG standards for users with disabilities.

**Scalable Design**: The app's design is scalable, allowing easy expansion and addition of new features while maintaining consistency across the platform.

## 5. Conclusion

Task #2 demonstrated the efficiency and flexibility of Tailwind CSS in building modern, responsive web pages. The project's collaborative nature also underscored the importance of clear code structure and consistent use of utility classes for maintainability.

# REPORT III: Bootstrap

## 3.1 Bootstrap Overview

### 3.1.1 History of Bootstrap

Bootstrap is a sleek, intuitive, and powerful mobile first front-end framework for faster and easier web development. It uses HTML, CSS and JavaScript. Bootstrap was developed by **Mark Otto and Jacob Thornton at Twitter**. It was released as an open-source product in August 2011 on GitHub.

### 3.1.2 Why use Bootstrap

- ✓ Mobile first approach: Since Bootstrap 3, the framework consists of Mobile first styles throughout the entire library instead of in separate files.
- ✓ Browser Support: It is supported by all popular browsers.


**Figure 3: Browser support and responsive for different platforms**

- ✓ Easy to get started: With just the knowledge of HTML and CSS anyone can get started with Bootstrap. Also the Bootstrap official site has a good documentation.
- ✓ Responsive design: Bootstrap's responsive CSS adjusts to Desktops, Tablets and Mobiles. More about responsive design in the IV Bootstrap Responsive Design

### 3.2 Bootstrap Environment Setup

there are two ways to set up bootstrap env in respective project:

#### 3.2.1. Bootstrap via CDN

Create Your Project Folder

Create a folder for your project and initialize it (optional) with Git:

```
mkdir task3

cd task3

git init
```

Add Bootstrap via CDN (Quick Setup)

Create an `index.html` file and add the Bootstrap CDN:

```
<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Bootstrap Setup</title>

<! -- Bootstrap CSS CDN -->

<linkhref="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css"rel
="stylesheet">

</head>

--------------------------------------------------

<body>

<! -- Bootstrap JS (Optional) -->

<script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/js/bootstrap.bundle.min.js"></s
cript>

</body>
```

### 3.2.2. Install Bootstrap via npm

Better for building tool like **Webpack**, **Vite**, or **Gulp**, this approach is better:

```
npm init -y

npm install bootstrap

<link rel="stylesheet"
href="node_modules/bootstrap/dist/css/bootstrap.min.css">

<script
src="node_modules/bootstrap/dist/js/bootstrap.bundle.min.js"
defer></script>
```

(Bootstrap Tutorial, n.d.)

### 3.4 Design and implementation

Bootstrap includes six default breakpoints, sometimes referred to as *grid tiers*, for building responsively. These breakpoints can be customized if you're using our source Sass files. Each breakpoint was chosen to comfortably hold containers whose widths are multiples of 12. Breakpoints are also representative of a subset of common device sizes and viewport dimensions—they don't specifically target every use case or device. Instead, the ranges provide a strong and consistent foundation to build on for nearly any device (Bootstrap, n.d.).
Take a look on the following Bootstrap code:

```
$grid-breakpoints: (
  xs: 0,
  sm: 576px,
  md: 768px,
  lg: 992px,
  xl: 1200px,
  xxl: 1400px
);
```

**Table 1: Bootstrap includes six default breakpoints**

| Breakpoint | Class infix | Dimensions |
|---|---|---|
| X-Small | *None* | <576px |
| Small | sm | ≥576px |
| Medium | md | ≥768px |
| Large | lg | ≥992px |
| Extra large | xl | ≥1200px |
| Extra extra large | xxl | ≥1400px |

### 3.4.1 Demo app medium screen sizes

This describer the design of this demo app that is developed in bootstrap by showing the screen shoots of the demo app. In Bootstrap, medium screen sizes (laptop and tablets) are defined as devices with a screen width of 768px and up, but less than 992px, and are targeted with classes prefixed with md (e.g., .col-md-4).

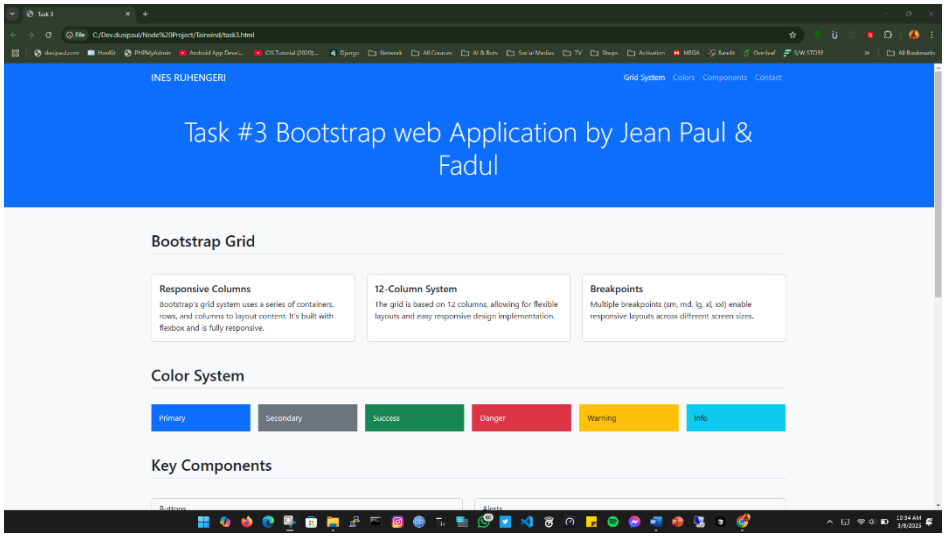
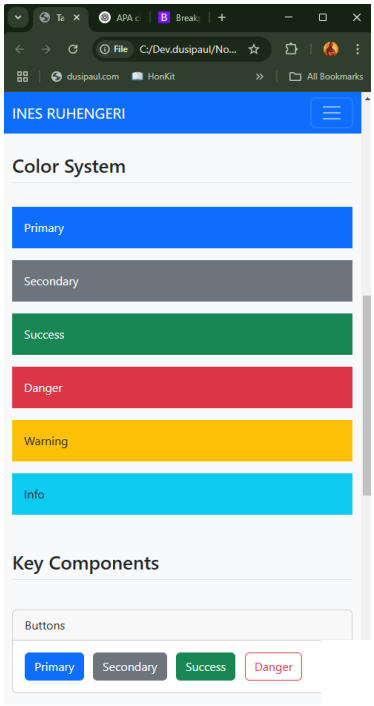



**Figure 5: Medium Screen size**

## 3.5 Conclusion




A Simple layout with two columns. We want the columns to be split 25%/75% for small devices.

**Tip:** Small devices are defined as having a screen width from **768 pixels to 991 pixels**.

For small devices we will use the .col-sm-* classes.

We will add the following classes to our two columns:

```
<div class="col-sm-3">....</div>
<div class="col-sm-9">....</div>
```

**Figure 4: Small Screen size**

Bootstrap is a powerful, mobile-first front-end framework that simplifies web development through its responsive grid system, pre-designed components, and extensive browser support. Developed by Mark Otto and Jacob Thornton, it has evolved into an essential tool for modern web design, providing developers with an efficient way to create visually appealing and functional websites.

This report covered the history of Bootstrap, its key advantages, and the steps for setting up a Bootstrap environment using both CDN and npm installation. Additionally, the importance of responsive design and Bootstrap's default breakpoints were highlighted, showcasing how the framework adapts to various screen sizes.

By leveraging Bootstrap, developers can build flexible, responsive, and well-structured web applications with minimal effort. Its ease of use, strong community support, and continuous updates make it a reliable choice for both beginners and experienced web developers.

# REPORT IV: JavaScript DOM Manipulation and Styling

## IV.1 Introduction

JavaScript is a powerful programming language that enables dynamic interactions and manipulations on web pages. One of its core functionalities is the manipulation of the Document Object Model (DOM), which represents the structure of an HTML document. Through DOM manipulation, developers can dynamically update content, modify styles, and handle user interactions efficiently. This report explores JavaScript's DOM manipulation capabilities, focusing on element selection, event handling, and styling techniques.

## IV 2: JavaScript DOM Manipulation

### 2.1 Selecting Elements

JavaScript provides multiple methods to select and interact with HTML elements. Some commonly used selectors include:

- **getElementById()**: Retrieves an element using its unique `id`.

```
let element = document.getElementById("myElement");
```

- **getElementsByClassName()**: Returns a collection of elements with a specified class name.

```
let elements = document.getElementsByClassName("myClass");
```

- **getElementsByTagName()**: Selects elements by their tag name (e.g., `div`, `p`, `h1`).

```
let paragraphs = document.getElementsByTagName("p");
```

- **querySelector()**: Returns the first matching element using a CSS selector.

```
let firstElement = document.querySelector(".myClass");
```

- **querySelectorAll()**: Selects all matching elements using a CSS selector.

```
let allElements = document.querySelectorAll(".myClass");
```

### 2.2 Event Handling

Event handling is a crucial aspect of interactive web applications. JavaScript allows developers to listen to and respond to various user interactions such as clicks, key presses, and form submissions. Some common event-handling methods include:

- **addEventListener()**: Adds an event listener to an element.

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

- **Key Events**: Detects key presses to provide real-time interaction.

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

- **Mouse Events**: Responds to mouse actions.

```
document.getElementById("box").addEventListener("mouseover", function()
{
    this.style.backgroundColor = "yellow";
});
```

### 2.3 Styling Elements

JavaScript enables dynamic styling of HTML elements through the `style` property and CSS classes.

- **Modifying Inline Styles**:

```
document.getElementById("text").style.color = "blue";
document.getElementById("text").style.fontSize = "20px";
```

- **Adding and Removing CSS Classes**:

```
document.getElementById("box").classList.add("highlight");
document.getElementById("box").classList.remove("highlight");
```

- **Toggling Classes**:

```
document.getElementById("box").classList.toggle("hidden");
```

## IV 3 Design and implementation

The design of JavaScript's DOM manipulation revolves around structuring an interactive web page that dynamically updates based on user input. A simple web interface is created where users can interact with buttons, text fields, and visual elements, demonstrating various DOM manipulation techniques.

### 3.2 Implementation Steps

1. **Setting Up the HTML Structure**:
    - Create an HTML file with elements such as buttons, input fields, and divs.

```
<button id="changeText">Change Text</button>
<p id="text">Hello, World!</p>
```

3. **Applying JavaScript for Interactivity**:
    - Write JavaScript to modify the text when the button is clicked.

```
document.getElementById("changeText").addEventListener("click",
function() {
    document.getElementById("text").innerText = "Text Updated!";
});
```

6. **Adding Dynamic Styling**:
    - Modify CSS styles dynamically.

```
document.getElementById("text").style.color = "red";
```

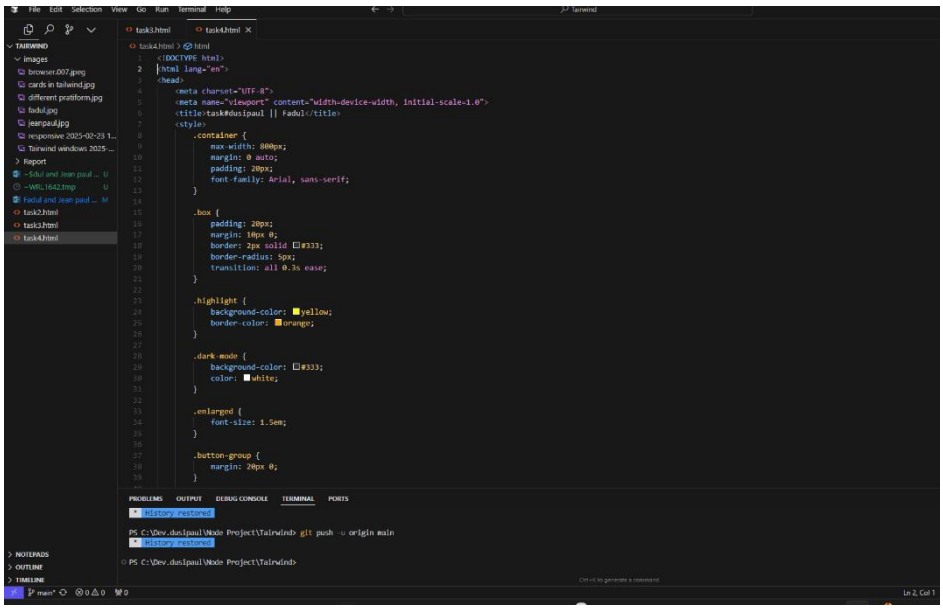


**Figure 5: Dynamic Styling**

7. **Event Handling for User Inputs**:
    - Capture keyboard inputs and display them.

```
8. document.addEventListener("keyup", function(event) {
9.     console.log("Key Released: " + event.key);
   });
```

A small app that: Page 2 of 5 ▪ Selects and modifies elements on the page dynamically. ▪ Demonstrates adding, removing, and toggling CSS classes on elements. ▪ Includes interactive elements (e.g., buttons to change styles or content).

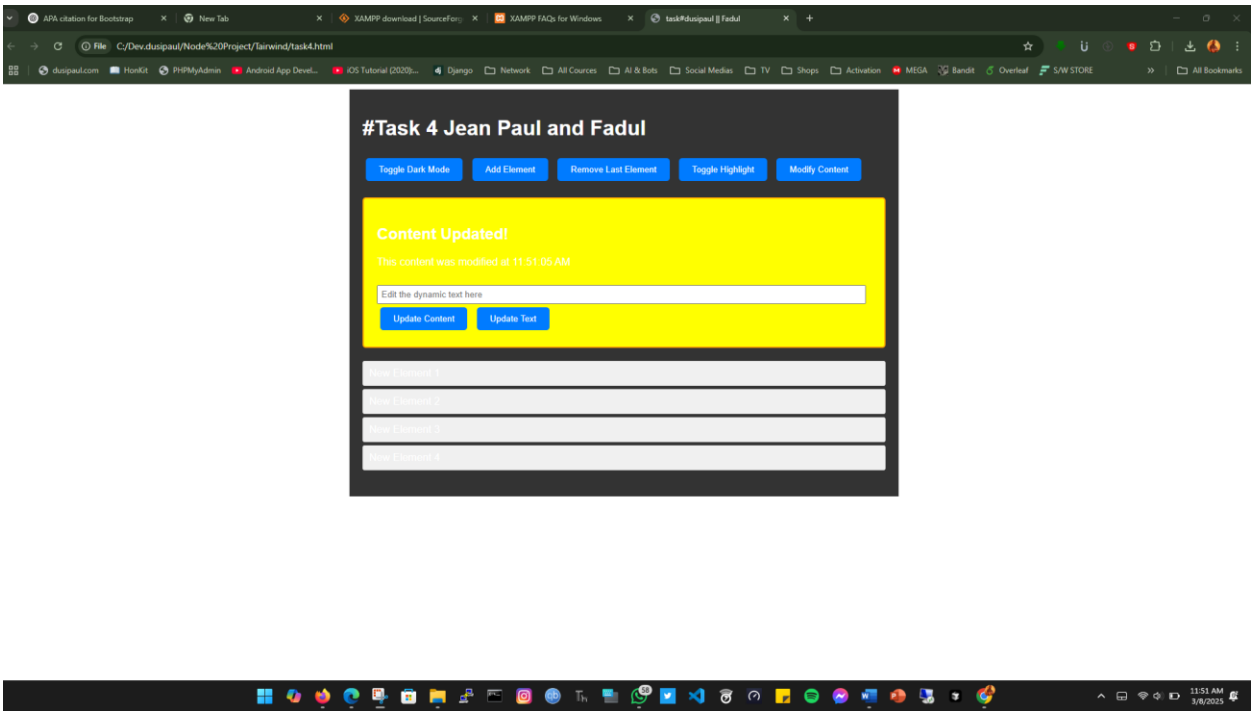



**Figure 6: User Inputs and Manipulation**

## IV 4: Conclusion

JavaScript's DOM manipulation capabilities provide developers with extensive control over web pages, enabling them to create dynamic and interactive user experiences. By using element selectors, event handlers, and styling methods, developers can modify content, handle user interactions, and style elements efficiently. Understanding these techniques is fundamental for building modern, responsive web applications. Mastery of DOM manipulation empowers developers to enhance user experience and optimize website functionality effectively.

# REPORT V. Node.js Application with File Handling and Image Upload

## 5.0. File Handling in Node.js

### 5.1.1 Overview

File handling in Node.js involves reading, writing, and deleting files using the built-in `fs` module. In this application, file handling is used to manage image uploads, display stored images, and delete images when requested.

## 5.2 File Handling in the Provided Application

The application uses the `fs` module in the following ways:

### 5.2.1. Reading Files (Displaying Images in the Gallery)

- The application reads the contents of the `public/uploads/` directory to list the uploaded images.
- The `fs.readdir()` function retrieves the list of uploaded image files and passes them to the `index.ejs` template for rendering.

```
app.get("/", (req, res) => {
  const uploadDir = "public/uploads/";
  fs.readdir(uploadDir, (err, files) => {
    if (err) throw err;
    res.render("index", { images: files });
  });
});
```

#### 5.2.2. Deleting Files

- Users can delete images from the gallery by clicking a delete button.
- The `fs.unlink()` function removes the specified file from the `uploads` directory.

```
app.post("/delete/:filename", (req, res) => {
  const      filePath      =      path.join(__dirname,      "public/uploads/",
req.params.filename);
  fs.unlink(filePath, (err) => {
    if (err) throw err;
    res.redirect("/");
  });
});
```

### 5.3. Image Upload & Storage

Storage Method Used

This application uses **local storage** for image uploads. The images are saved in the `public/uploads/` directory.

Uploading Images with Multer

The `multer` middleware is used to handle file uploads. It stores uploaded images in the `public/uploads/` directory and renames files with a timestamp to prevent duplicates.

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

const storage = multer.diskStorage({
  destination: (req, file, cb) => {
    cb(null, "public/uploads/"); // Image storage directory
  },
  filename: (req, file, cb) => {
    cb(null, Date.now() + "-" + file.originalname); // Unique file naming
  },
});

const upload = multer({ storage });
```

- **`destination`**: Defines where the uploaded images are stored.
- **`filename`**: Renames files by adding a timestamp.

**Handling Image Uploads**

The `/upload` route processes the file upload and stores it in the `uploads` directory.

```
app.post("/upload", upload.single("image"), (req, res) => {
  res.redirect("/");
});
```

- Users upload images via an HTML form (`enctype="multipart/form-data"`).
- After uploading, the user is redirected to the homepage to view the updated gallery.

### 5.4. Does This Application Support the Required Features?

Allows users to upload images

The upload form in `index.ejs` lets users select an image and upload it.

Images are stored in the `public/uploads/` directory.

Stores images and displays them in a gallery

The `index.ejs` file dynamically lists images stored in `public/uploads/` and displays them in a Bootstrap-powered gallery.

**Provides image preview and delete functionality**

Each uploaded image is shown in the gallery with a preview (`<img>` tag).

Users can delete images by clicking a delete button.

### HTML Structure for Image Display (`index.ejs`)

```
<% images.forEach(image => { %>
  <div class="col">
    <div class="card h-100">
      <img src="/uploads/<%= image %>" class="card-img-top" alt="<%= image %>">
      <div class="card-body">
        <form action="/delete/<%= image %>" method="POST" class="d-grid">
          <button type="submit" class="btn btn-danger btn-sm">Delete</button>
        </form>
      </div>
    </div>
  </div>
<% }) %>
```

- Displays all images stored in `uploads/` using `<img>`.
- Provides a delete button for each image.

# REPORT VI. Node.js Application with Google Social Login Using Passport.js

## VI.1. Overview of the Application

The uploaded file (`app.js`) is a Node.js application that implements Google social login using Passport.js. The application allows users to authenticate using their Google accounts and displays their profile information after successful login. The application is built using the following technologies:

- **Node.js:** A JavaScript runtime environment used to build server-side applications.
- **Express.js:** A web framework for Node.js that simplifies the creation of web servers and routes.
- **Passport.js:** An authentication middleware for Node.js that supports various authentication strategies, including Google OAuth.
- **Google OAuth 2.0:** A protocol used for authentication and authorization, allowing users to log in using their Google accounts.

The application is designed to be simple yet functional, providing a seamless user experience for authentication and profile management.

### VI.2. Key Components of the Application

The application consists of several key components, each serving a specific purpose:

1. Environment Configuration:

- The application uses the `dotenv` package to load environment variables from a `.env` file. This ensures that sensitive information, such as Google OAuth credentials and session secrets, is kept secure and not hard-coded into the application.

2. Express.js Setup:

- The application initializes an Express.js server and configures it to use sessions for maintaining user authentication state. The session is secured using a secret key stored in the environment variables.

3. Passport.js Configuration:

Passport.js is configured to use the Google OAuth 2.0 strategy for authentication. The Google OAuth credentials (client ID, client secret, and callback URL) are retrieved from the environment variables.

The `serializeUser` and `deserializeUser` methods are implemented to store and retrieve user information from the session.

4. Routes:

- The application defines several routes to handle different functionalities:
    - **Root Route (`/`)**: Displays a login page with a button to log in using Google.
    - **Google Authentication Route (`/auth/google`)**: Initiates the Google OAuth authentication process.
    - **Google Callback Route (`/auth/google/callback`)**: Handles the callback from Google after successful authentication and redirects the user to the profile page.
    - **Profile Route (`/profile`)**: Displays the user's profile information, including their name, email, and profile picture, after successful login.
    - **Logout Route (`/logout`)**: Logs the user out by ending their session and redirecting them to the login page.

5. User Interface:

- The application uses simple HTML and CSS to create a visually appealing user interface. The login and profile pages are styled using Bootstrap and custom CSS to provide a modern and responsive design.

### VI.3. Detailed Explanation of the Code

**1. Environment Variables and Module Imports:**

- The application starts by loading environment variables using `dotenv` and importing the necessary modules, including Express, Passport, and the Google OAuth strategy.

**2. Express and Session Configuration:**

- The Express app is initialized, and session middleware is configured to store user authentication data securely. The session secret is retrieved from the environment variables.

**3. Passport Initialization and Google Strategy:**

- Passport is initialized and configured to use the Google OAuth 2.0 strategy. The strategy is set up with the client ID, client secret, and callback URL from the environment variables. The `serializeUser` and `deserializeUser` methods are implemented to manage user sessions.

**4. Routes:**

**Root Route (`/`)**: This route renders a simple HTML page with a "Login with Google" button. The page is styled using Bootstrap and custom CSS to create a visually appealing login interface.

**Google Authentication Route (`/auth/google`)**: This route initiates the Google OAuth process by redirecting the user to Google's authentication page.

**Google Callback Route (`/auth/google/callback`)**: This route handles the callback from Google after the user has authenticated. If authentication is successful, the user is redirected to the profile page.

**Profile Route (`/profile`)**: This route displays the user's profile information, including their name, email, and profile picture. If the user is not authenticated, they are redirected to the login page.

**Logout Route (`/logout`)**: This route logs the user out by ending their session and redirecting them to the login page.

**5. Server Initialization:**

- The application listens on a specified port (defaulting to 3000 if no port is provided in the environment variables) and logs a message to the console indicating that the server is running.

---

## VI.4. How the Application Works

1. **User Visits the Login Page:**
   - When the user visits the root URL (`/`), they are presented with a login page containing a "Login with Google" button.
2. **User Initiates Google Login:**
   - Clicking the "Login with Google" button redirects the user to Google's authentication page, where they can log in using their Google account.
3. **Google Authentication and Callback:**

- After the user logs in, Google redirects them back to the application's callback URL (`/auth/google/callback`). The application verifies the authentication and stores the user's profile information in the session.

4. **User is Redirected to the Profile Page:**
   - If authentication is successful, the user is redirected to the profile page (`/profile`), where their profile information (name, email, and profile picture) is displayed.
5. **User Logs Out:**
   - The user can log out by clicking the "Logout" button on the profile page. This ends their session and redirects them back to the login page.

## REPORT VII: React Core Concepts in the Task Dashboard App

This report covers the core React concepts used in the Task Dashboard App: **Components, Props, and State**. These concepts are fundamental to understanding how React applications are structured and how data flows between different parts of the app.

# 1. Components

### VII.0 What Are Components?

Components are the building blocks of a React application. They are reusable, self-contained pieces of code that represent a part of the user interface (UI). Components can be thought of as custom HTML elements that encapsulate both structure (HTML) and behavior (JavaScript).

### VII.1 Components in the Task Dashboard App

The Task Dashboard App is built using several components, each responsible for a specific part of the UI:

**1. Dashboard**

- The main parent component that manages the state (tasks) and renders other components.
- Acts as the container for the entire app.

**2. TaskCounter**

- Displays statistics about the tasks (e.g., completed, pending, and urgent tasks).
- Receives the `tasks` array as a **prop** to calculate and display the counts.

**3. TaskList**

- Displays a list of tasks with their title, description, priority, and status.
- Receives the `tasks` array, `onToggleStatus`, and `onDeleteTask` functions as **props** to render tasks and handle user interactions.

**4. AddTaskForm**

- A form that allows users to add new tasks.
- Receives the `onAddTask` function as a **prop** to add a new task to the `tasks` array.

**5. Card, CardHeader, CardTitle, CardContent**

- Reusable UI components from a custom library (e.g., `./components/ui/card`).
- Used to create consistent and styled card layouts.

## Why Use Components?

- **Reusability**: Components like `TaskCounter` and `TaskList` can be reused in other parts of the app or in other projects.
- **Separation of Concerns**: Each component handles a specific part of the UI, making the code easier to maintain and debug.
- **Modularity**: Components can be developed and tested independently.

---

### VII.2. Props

## What Are Props?

Props (short for "properties") are a way to pass data from a **parent component** to a **child component**. They are **read-only** and allow components to be dynamic and reusable.

## Props in the Task Dashboard App

Props are used extensively in the app to pass data and functions between components:

### 1. Passing Data

The `Dashboard` component passes the `tasks` array to the `TaskCounter` and `TaskList` components:

```
<TaskCounter tasks={tasks} />
<TaskList tasks={tasks} />
```

### 2. Passing Functions

The `Dashboard` component passes functions like `onAddTask`, `onToggleStatus`, and `onDeleteTask` to child components to handle user interactions:

```
<AddTaskForm onAddTask={handleAddTask} />
<TaskList onToggleStatus={handleToggleStatus} onDeleteTask={handleDeleteTask}
/>
```

### 3. Using Props in Child Components

Child components like `TaskCounter` and `TaskList` use the props they receive to render data and handle events:

```
const TaskCounter = ({ tasks }) => {
  const completed = tasks.filter((task) => task.status === "completed").length;
  // ...
};

const TaskList = ({ tasks, onToggleStatus, onDeleteTask }) => {
  return (
    <div>
      {tasks.map((task, index) => (
        <div key={index}>
          <h3>{task.title}</h3>
          <button onClick={() => onToggleStatus(index)}>Toggle Status</button>
          <button onClick={() => onDeleteTask(index)}>Delete</button>
        </div>
      ))}
    </div>
  );
};
```

## Why Use Props?

- **Data Flow**: Props enable **unidirectional data flow** from parent to child components.
- **Reusability**: Components like `TaskList` can be reused with different data by passing different props.
- **Separation of Concerns**: Props allow components to remain independent and focused on their specific tasks.

---

**VII3. State**

## What Is State?

State is a built-in React feature that allows components to **manage and track data** that can change over time. Unlike props, state is **mutable** and is managed within the component itself (or its parent component).

## State in the Task Dashboard App

The `Dashboard` component uses **state** to manage the list of tasks (`tasks`) and update the UI when tasks are added, toggled, or deleted.

### 1. State Initialization

The `tasks` state is initialized using the `useState` hook:

```
const [tasks, setTasks] = useState([
  { title: "Complete Project Proposal", status: "completed", priority: "high"
},
  { title: "Review Code Changes", status: "pending", priority: "medium" },
  // ...
]);
```

### 2. Updating State

The `setTasks` function is used to update the `tasks` state when a new task is added, a task's status is toggled, or a task is deleted:

```
const handleAddTask = (newTask) => {
  setTasks([...tasks, newTask]);
};

const handleToggleStatus = (index) => {
  const updatedTasks = [...tasks];
  updatedTasks[index].status =
    updatedTasks[index].status === "completed" ? "pending" : "completed";
  setTasks(updatedTasks);
};

const handleDeleteTask = (index) => {
  const updatedTasks = tasks.filter((_, i) => i !== index);
  setTasks(updatedTasks);
};
```

### 3. Passing State as Props

The `tasks` state is passed as a prop to child components (`TaskCounter` and `TaskList`) to render the data:

```
<TaskCounter tasks={tasks} />
<TaskList tasks={tasks} />
```

## Why Use State?

- **Dynamic UI**: State allows the UI to update dynamically in response to user interactions (e.g., adding, toggling, or deleting tasks).
- **Centralized Data Management**: The `Dashboard` component acts as the **single source of truth** for the `tasks` data, making it easier to manage and debug.
- **Reactivity**: When **state changes**, React automatically **re-renders** the affected components, ensuring the UI stays in sync with the data.

## VII.4**Conclusion**

The Task Dashboard App demonstrates the core React concepts of **components, props, and state**:

1. **Components** break the UI into reusable and modular pieces.
2. **Props** enable data and functions to flow from parent to child components.
3. **State** allows the app to manage and update dynamic data, ensuring the UI stays reactive and up-to-date.

# REFERENCE

Web Dev Simplified. (2023, May 5). *Learn Tailwind CSS In 20 Minutes* [Video]. YouTube. https://www.youtube.com/watch?v=elgqxmdVms8

Bootstrap. (n.d.). *Grid system*. Bootstrap. Retrieved March 8, 2025, from https://getbootstrap.com/docs/5.0/layout/grid/#sass