Class 6: Implementing Search Functionality & Place Order

SESSION OVERVIEW:

By the end of this session, students will be able to:

- Implement search functionality for dynamically filtering food items using JavaScript.
- Understand and apply search algorithms like debouncing and throttling to improve search performance.
- Utilize DOM manipulation to display filtered search results in real-time as the user types.
- Create a smooth user experience by integrating a "Place Order" feature with a modal that allows users to input their seat number before confirming their order.
- Learn how to handle user interactions, such as clicking the "Place Order" button and handling order submission through the modal.

SESSION TOPICS: Implementing Search Functionality & Place Order

1. **Search Functionality:**
    - **Adding a Search Bar:** Learn how to implement a search bar on the menu page to filter food items dynamically as the user types.
    - **Search Algorithms (Debouncing and Throttling):**
        - Understand the concept of debouncing to optimize search function calls by reducing the frequency of search updates.
        - Explore throttling techniques to limit the number of times the search function is triggered in rapid succession.
    - **Dynamic Search Results:**
        - Master the use of DOM manipulation to update the search results in real-time without reloading the page, providing a smooth user experience.
2. **Place Order Functionality:**
    - **Implementing "Place Order" Button:** Learn how to create a functional "Place Order" button that triggers a modal when clicked.
    - **Modal Implementation:** Understand how to design and display a modal asking the user to input their seat number after clicking the "Place Order" button.
    - **Order Confirmation:**
        - Learn how to show a success message (e.g., "Order placed successfully") after the user enters their seat number.
3. **Responsive Design for Search & Order:**
    - **Optimizing for Different Devices:** Understand how to design the search bar and "Place Order" button for responsiveness across desktop, tablet, and mobile views.

    - **CSS Flexbox/Grid:** Learn how to use Flexbox and Grid to ensure the layout adapts properly to different screen sizes.
    - **Media Queries:** Implement media queries to adjust the design for various device sizes, improving the user experience on smaller screens.

Introduction to Search Functionality

Understand how to add a search bar to a page, filter items dynamically as users type, and optimize performance using techniques like debouncing and throttling.

Page Components Overview:

1. **Search Bar:**
    - A search bar is an input field where users type their search queries (e.g., food names).
    - It filters items on the page as users type, providing instant feedback without needing to reload the page.
2. **Debouncing & Throttling:**
    - **Debouncing:** It limits the rate at which the search function is executed by waiting for the user to stop typing for a certain amount of time before initiating the search. This reduces unnecessary searches when the user is still typing.
    - **Throttling:** It limits how often the search function is executed over time (e.g., every 500ms), preventing too many searches in a short period, which can overload the system.
3. **Why are these needed?**
    - Both techniques help improve performance, especially when making requests to an API or processing large data sets. They ensure that the search function isn't triggered too frequently, which can slow down the page.
4. **Search Results:**
    - As the user types, the page dynamically updates to show only the items that match the search input.
    - The items could be filtered based on product names, categories, or keywords.

HTML Structure:
Add the div at the top of all categories for the rendering search items.

```
<div class="ourmenu-category" id="search-items">
           <div class="ourmenu-items">
           </div>
```

```
</div>
```

Dynamic Search Results Using JavaScript

Html Code:

```
<input class="search-input" placeholder="What do you want to search..." />
```

Javascript code:

```
function searchItems(query) {
    const searchContainer =
document.getElementById('search-items').querySelector('.ourmenu-items');
    searchContainer.innerHTML = ''; // Clear any previous search results

    const filteredItems = foodItems.filter(item =>
        item.title.toLowerCase().includes(query.toLowerCase())
    );

    if (filteredItems.length > 0) {
        const itemsHTML = filteredItems.map(createMenuItem).join('');
        searchContainer.innerHTML = itemsHTML;
    } else {
        searchContainer.innerHTML = '<p>No items found.</p>';
    }
}
```

Explanation:

- Filters items from the `foodItems` array that match the search query.
- Dynamically updates the DOM with the filtered results.

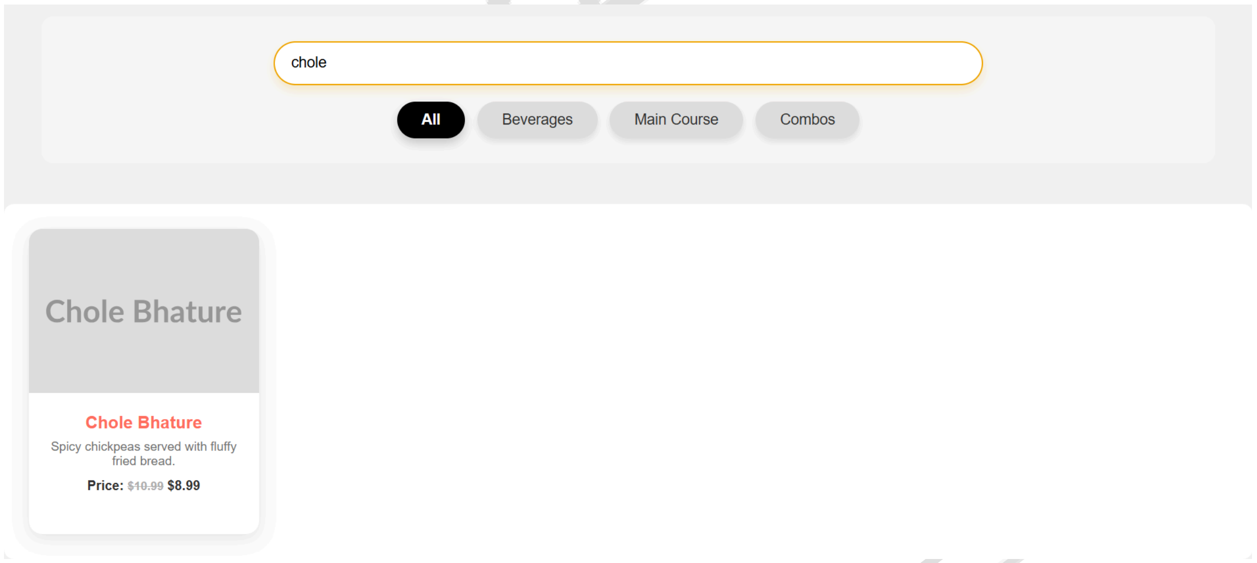
Implementing Debouncing:

- Introduce debouncing as a technique to optimize performance by limiting the number of API calls made during searches.

```
let timeout = null;
document.querySelector('.search-input').addEventListener('input', (event) => {
    const query = event.target.value;
    console.log(query)
    if (query) {
        // Clear the previous timeout if it exists
        if (timeout) {
            clearTimeout(timeout);
        }
        // Set a new timeout
        timeout = setTimeout(() => {
            searchItems(query);
        }, 1000); // Wait for 1000ms before searching
    } else {
        // Optionally clear the search results if input is empty

document.getElementById('search-items').querySelector('.ourmenu-items').innerHTML
= '';
    }
});
```

Output:



Explanation:

- **Purpose of Debouncing:**
    1. Reduces the number of times the `searchItems` function is called while the user is typing.
    2. Waits for the user to stop typing for a specific time (e.g., 1 second) before executing the function.
- **How It Works:**
    1. Each time the `input` event is triggered, it clears any previous `timeout` to reset the timer.

2. After the user stops typing for 1 second (`1000ms`), the `searchItems` function is called with the search query.

Place Order Functionality:

First, let's create the basic HTML structure of our modal for place order containing input for entering the seat number at cart page.

HTML structure that defines the overall layout of the Modal.

```
  <div id="seatModal" class="modal">
        <div class="modal-content">
            <span class="close-button" onclick="closeModal()">&times;</span>
            <h2>Enter Your Seat Number</h2>
          <div class="modal-button">
             <input type="number" id="seatNumber" placeholder="Seat Number" />
             <button class="place-order-button" onclick="placeOrder()">Place
Order</button>
           </div>
        </div>
    </div>
```

CSS Style:

```
.modal {

    display: none; /* Hidden by default */

    position: fixed; /* Stay in place */

    z-index: 1000; /* Sit on top */

    left: 0;

    top: 0;

    width: 100%; /* Full width */

    height: 100%; /* Full height */
    background-color: rgba(0, 0, 0, 0.5); /* Black w/ opacity */

}



/* Modal Content */

.modal-content {

    background-color: #fefefe;

    margin: 15% auto; /* 15% from the top and centered */

    padding: 20px;

    border: 1px solid #888;



    width: 300px; /* Could be more or less, depending on screen size */

    border-radius: 8px; /* Rounded corners */

    box-shadow: 0 4px 8px rgba(0, 0, 0, 0.2); /* Shadow effect */

}

.modal-button{



    display: flex;

    justify-content: flex-start;

    align-items: center;

    justify-content: space-between;

    /* flex-direction: column; */

}

/* Close Button */
```

```css
.close-button {

    color: #aaa;

    float: right;

    font-size: 28px;

    font-weight: bold;

    cursor: pointer;

}


/* Close button hover effect */

.close-button:hover,

.close-button:focus {

    color: black;

    text-decoration: none;

    cursor: pointer;

}


/* Input field */

input[type="number"] {

    /* width: 100%;   */

    padding: 10px; /* Some padding */

    /* margin: 10px 0; Space above and below */

    border: 1px solid #ccc; /* Gray border */

    border-radius: 4px; /* Rounded corners */
}


/* Place Order Button */

.place-order-button {

    background-color: #4CAF50; /* Green */

    color: white;

    padding: 10px 15px;

    align-self: flex-end;

    border: none;

    border-radius: 4px; /* Rounded corners */

    cursor: pointer;

    float: right; /* Align to the right */

}


.place-order-button:hover {

    background-color: #45a049; /* Darker green on hover */

}
```

Javascript:
This functionality is designed to capture the user's seat number when they place order. Upon clicking the "proceed to checkout" button, a modal appears prompting the user to enter their seat number.

```
function openModal() {

    document.getElementById("seatModal").style.display = "block";

}
```

```
// Function to close the modal

function closeModal() {

    document.getElementById("seatModal").style.display = "none";

}

// Function to place the order

function placeOrder() {

    const seatNumber = document.getElementById("seatNumber").value;

    if (seatNumber) {

        alert(`Order placed for seat number: ${seatNumber}`);

        closeModal(); // Close the modal after placing the order

    } else {

        alert("Please enter a seat number.");

    }

}

// Close the modal when clicking outside of it

window.onclick = function(event) {

    const modal = document.getElementById("seatModal");

    if (event.target === modal) {

        closeModal();

    }

}
```

Explanation:

This functionality manages a **Seat Number Modal** for placing orders:

1. **Open Modal**: Triggered when the "Place Order" button is clicked, allowing users to enter their seat number.
2. **Close Modal**: Users can close the modal by clicking a close button, outside the modal, or after submitting an order.
3. **Place Order**: Users input a seat number. If valid, the order is confirmed, and the modal closes; otherwise, an alert prompts them to enter the number.

Final output:
Ourmenu.html
Cart.html
cartstyle.css

Concept Indepths:

Search Functionality

Search functionality is a cornerstone of modern applications, enabling users to quickly locate desired information. A well-designed search mechanism enhances user experience, reduces friction, and ensures application scalability. Let's delve into the concepts and techniques used in search functionality, ranging from the basics to advanced implementations.

---

1. Basic Concepts of Search Functionality

1. **Static Search**:
    - Searches through predefined static data stored on the client-side (e.g., arrays, JSON).
    - **Example**: Searching through a local list of items in a JavaScript array.
2. **Dynamic Search**:
    - Retrieves and filters data dynamically from a database or API, enabling real-time updates.
    - Uses asynchronous calls (e.g., `fetch()` or `axios`) to interact with backend services.
3. **Case-Insensitive Search**:
    - Converts both the search query and data to a common format (e.g., lowercase) to handle case differences effectively.

---

2. Optimizing Search Performance

While basic search can be effective for small datasets, it becomes inefficient as data grows. Optimization techniques include:

1. **Debouncing**:
   - **Definition**: A method to delay the execution of a function until after a specified period of user inactivity.
   - **Use Case**: Prevents frequent API calls during continuous typing.
   - **Industry Example**: E-commerce websites like Amazon use debouncing to improve performance and reduce server load.
2. **How It Works**:
   - Waits for a user to pause typing before sending a search query.
3. **Throttling**:
   - **Definition**: Ensures a function executes only once within a specific time interval, regardless of how often it's triggered.
   - **Use Case**: Limits API requests in high-frequency user actions.
   - **Industry Example**: Applications like Google Maps throttle map panning to ensure smooth navigation.
4. **Key Difference from Debouncing**:
   - Debouncing waits until user activity stops; throttling ensures periodic execution during activity.
5. **Caching**:
   - **Definition**: Storing previously fetched data to avoid redundant API calls.
   - **Use Case**: Improves performance for repetitive searches.
   - **Example**: Search engines cache recent queries to enhance response times.

---

3. Advanced Techniques in Search Functionality

1. **Search Indexing**:
   - **Definition**: Creating an optimized data structure (e.g., a search tree or inverted index) for faster search operations.
   - **Technology**: Tools like Elasticsearch, Apache Solr, or Algolia provide full-text search capabilities with advanced indexing.
   - **Use Case**: Essential for large-scale applications handling vast datasets (e.g., e-commerce, job portals).
2. **Fuzzy Search**:
   - **Definition**: Enables approximate matches to handle typos or similar words.
   - **Industry Trend**: Social media platforms like LinkedIn or Facebook use fuzzy search to account for misspellings in names or terms.
   - **Implementation**: Levenshtein Distance or tools like ElasticSearch provide out-of-the-box fuzzy search.
3. **AI-Powered Search (Semantic Search)**:

   - **Definition**: Leverages machine learning and natural language processing (NLP) to understand user intent.
   - **Technology**: Tools like OpenAI embeddings, Google BERT, or vector-based search algorithms.
   - **Industry Application**: Content platforms like YouTube use semantic search to recommend videos based on query context rather than exact matches.
4. **Faceted Search**:
   - **Definition**: Allows users to refine searches using filters (e.g., price, category).
   - **Example**: E-commerce platforms like eBay use faceted search for better navigation.

---

4. Search UI/UX Best Practices

1. **Instant Feedback**:
   - Display results in real-time as users type to improve interactivity.
2. **Error Tolerance**:
   - Provide suggestions for typos or incomplete queries.
3. **Highlighting Matches**:
   - Highlight the matching terms in results to enhance clarity.
4. **Loading Indicators**:
   - Show spinners or messages during data fetching to manage user expectations.

Debouncing

Debouncing and throttling are techniques used in JavaScript to manage how often a function executes in response to high-frequency events, such as keystrokes, scrolling, or resizing. Both aim to improve performance and ensure a smoother user experience, but they work differently and are suited to distinct use cases.

---

1. What is Debouncing?

Debouncing is a technique to delay the execution of a function until a specified period of inactivity has elapsed. This ensures the function is triggered only after the user stops performing an action, such as typing or resizing.

How It Works:

- When an event (e.g., `input`) fires, the debounce function clears the previously set timer and starts a new one.

- If the user continues to trigger the event (e.g., keeps typing), the timer resets every time.
- The actual function runs only after the timer completes without any new events.

Use Case:

- **Search Functionality**: Avoid making an API call on every keystroke; instead, wait until the user has stopped typing.
- **Window Resizing**: Trigger layout adjustments only after the user finishes resizing the browser window.

Advantages:

- Prevents unnecessary function calls.
- Reduces computational load, especially for heavy operations like API calls or DOM updates.

Example:

```
let debounceTimer;

function debounce(func, delay) {
    return function(...args) {
        clearTimeout(debounceTimer); // Clear the previous timer
        debounceTimer = setTimeout(() => func(...args), delay); // Set a new
timer
    };
}

const searchItems = debounce((query) => {
    console.log(`Searching for: ${query}`);
    // Simulate API call or data filtering
}, 1000);

document.querySelector('.search-input').addEventListener('input', (event) => {
    searchItems(event.target.value);
});
```

- **Explanation**: The `searchItems` function executes only after the user stops typing for 1000ms.

Throttling

What is Throttling?

Throttling limits the execution of a function to once every specified interval, regardless of how many times the event occurs. Unlike debouncing, throttling ensures the function runs at regular intervals, even during continuous activity.

How It Works:

- When an event triggers, the function executes immediately (or after a delay) and ignores subsequent triggers until the interval has elapsed.

Use Case:

- **Scrolling Events**: Calculate positions or load content only at regular intervals during scrolling.
- **Mouse Move Events**: Track the position of a moving cursor without updating excessively.

Advantages:

- Ensures the function executes periodically during continuous events.
- Provides consistent updates while avoiding excessive computation.

Example:

```
function throttle(func, interval) {
    let lastExecuted = 0;

    return function(...args) {
        const now = Date.now();
        if (now - lastExecuted >= interval) {
            func(...args);
            lastExecuted = now;
        }
    };
}
```

```
const logScrollPosition = throttle(() => {
    console.log(`Scroll position: ${window.scrollY}`);
}, 500);

window.addEventListener('scroll', logScrollPosition);
```

- **Explanation**: The `logScrollPosition` function logs the scroll position at most once every 500ms, even if the user scrolls continuously.

Real-World Industry Trends

1. **Debouncing in Search Engines**:
    - Platforms like Google Search use debouncing to optimize query handling and reduce server load, ensuring a seamless user experience.
2. **Throttling in Social Media Feeds**:
    - Apps like Instagram throttle infinite scrolling to control the rate at which new content loads, balancing performance and responsiveness.
3. **Hybrid Approaches**:
    - Advanced systems often combine debouncing and throttling to balance precision and periodicity. For example, debouncing might control API calls while throttling updates UI components.
4. **ElasticSearch and OpenSearch**:
    - Widely adopted for scalable and robust search systems.
    - Offers features like real-time indexing, distributed architecture, and powerful analytics.
5. **GraphQL for Search**:
    - Allows precise data fetching with customizable query structures.
    - **Trend**: Increasing adoption due to its flexibility and performance advantages over REST.
6. **AI-Powered Suggestions**:
    - Personalization using AI models to predict user intent.
    - **Example**: Netflix's search feature suggests shows based on viewing history and preferences.
7. **Progressive Web Applications (PWA)**:
    - Search functionality in PWAs ensures offline capabilities with cached data.

Follow tech blogs like Smashing Magazine, and Dev.to to stay current with industry trends