Class 7: Implementing Search Filters with Advanced JavaScript

Topics Covered

1. CRUD Operations in JavaScript
2. Advanced DOM Manipulation
3. Debouncing and Throttling

Objective

Enhance the product_page.html to include a search filter that allows users to filter products based on various criteria. We will implement the search functionality and incorporate advanced JavaScript techniques for better performance and user experience.

CRUD Operations

1. Read Operation

Purpose: Retrieve and display product data from an external source (e.g., products.json file).

Implementation Details:

- **Fetching Data:**

  Explanation:

  - fetch('products.json'): Sends an HTTP request to retrieve the products.json file.
  - .then(response => response.json()): Converts the response into a JSON object.
  - .then(data => displayProducts(data)): Calls the displayProducts function with the fetched data.

2. Create Operation

Purpose: Add a new product to the shopping cart.

Implementation Details:

- **Adding to Cart:**

  Explanation:

  - localStorage.getItem('cart'): Retrieves the current cart from local storage or initializes an empty array if not present.
  - cart.find(item => item.name === product.name): Checks if the product already exists in the cart.

- existingProduct.quantity += quantity: Updates the quantity if the product is already in the cart.
- cart.push(product): Adds the new product to the cart if it's not already present.
- localStorage.setItem('cart', JSON.stringify(cart)): Saves the updated cart to local storage.

3. Update Operation

Purpose: Modify the quantity of an existing product in the cart.

Implementation Details:

- **Updating Quantity:**
  - **Integrated in the addToCart function:**
    - If the product is already in the cart, its quantity is updated (existingProduct.quantity += quantity).
    - This ensures that the cart reflects the latest quantity of each product.

4. Delete Operation

Purpose: (Not implemented on the product page; generally involves removing a product from the cart.)

Note: Although the delete operation is not required here, in a typical scenario, it would involve removing an item from the cart array and updating local storage.

Summary

- **Read Operation:** Retrieves and displays product data from a JSON file.
- **Create Operation:** Adds new products to the product page or updates existing product quantities.
- **Update Operation:** (Embedded within the Create operation) Adjusts product quantities to the product page.
- **Delete Operation:** Not implemented here but usually involves removing products from the cart.

Advanced DOM Manipulation

1. **Dynamic Content Insertion**

Explanation:

- **Creating Elements:**
  - **document.createElement('div')**: Creates a new div element for each product card.
  - **card.classList.add('col-lg-3', 'col-md-4', 'col-sm-6', 'mb-4')**: Adds responsive Bootstrap classes to the div for proper layout.

    - **card.innerHTML**: Sets the inner HTML of the card with product details and "Add to Cart" button.
- **Appending to DOM:**
    - **productList.appendChild(card)**: Adds the new card to the product-list container, updating the DOM.

2. Event Handling

Explanation:

- **Adding Event Listeners:**
    - **document.querySelectorAll('.add-to-cart')**: Selects all "Add to Cart" buttons.
    - **.forEach(button => {...})**: Iterates over each button to attach an event listener.
- **Event Handling:**
    - **event.preventDefault()**: Prevents the default action of the link.
    - **event.target.getAttribute('data-product')**: Retrieves the product data stored in the data-product attribute.
    - **event.target.previousElementSibling**: Accesses the quantity input field adjacent to the button.

3. **Local Storage Manipulation**

Explanation:

- **Managing Cart Data:**
    - **localStorage.getItem('cart')**: Retrieves the current cart from local storage or initializes it as an empty array if not present.
    - **cart.find(item => item.name === product.name)**: Checks if the product already exists in the cart.
    - **existingProduct.quantity += quantity**: Updates the quantity of an existing product.
    - **cart.push(product)**: Adds a new product to the cart if it doesn't exist.
- **Saving to Local Storage:**
    - **localStorage.setItem('cart', JSON.stringify(cart))**: Saves the updated cart to local storage.

Summary

In this implementation, advanced DOM manipulation techniques are used to:

1. **Dynamically Insert Content:** Generate and insert product cards into the page based on data fetched from products.json.
2. **Handle User Events:** Attach event listeners to interactive elements (e.g., "Add to Cart" buttons) to manage user interactions.
3. **Manipulate Local Storage:** Store and manage cart data using the browser's local storage to persist user selections across page reloads.

These techniques ensure a dynamic, interactive, and user-friendly e-commerce product page.

Debouncing and Throttling

Debouncing and throttling are techniques used to optimize performance in scenarios where events are fired frequently, such as user input, scrolling, or resizing. They help reduce the number of times a function is executed, which can improve performance and user experience.

1. Debouncing

What is Debouncing?

Debouncing ensures that a function is not called until a certain amount of time has passed since the last time it was invoked. This is particularly useful for input fields where you want to wait until the user has stopped typing before performing an action like a search.

When to Use Debouncing?

- **Input Fields:** When handling user input, such as live search or autocomplete.
- **Resize Events:** When resizing the browser window and you want to wait until resizing is complete before executing code.

2. Throttling

What is Throttling?

Throttling ensures that a function is called at most once in a specified time interval. This technique is useful for tasks that need to be executed repeatedly but not too frequently, such as handling scroll or resize events.

When to Use Throttling?

- **Scroll Events:** When tracking user scroll position or implementing infinite scrolling.
- **Resize Events:** When handling window resizing.

Search Functionality

Objective: The search functionality allows users to filter products by their name and description. When a user types in the search box, the product list dynamically updates to show only the products that match the search query.

Implementation Steps:

1. **Fetch Products:**
    - The products are fetched from a products.json file and stored in a global variable for filtering.

2. **Handle Search Input:**
    - The handleSearch function filters products based on the search query, checking both the product name and description.

3. **Display Products:**
    - The displayProducts function creates product cards and appends them to the product list. It updates the DOM with the filtered products.

Filter Functionality Integration

Create a search input and a price range dropdown. Both elements should be in the same line, with the search input taking up most of the width and the price range dropdown occupying 25%.

JavaScript for Filtering:

- Implement a function to filter products based on the search input and the selected price range. This function will update the displayed products dynamically.

Explanation:

- **applyFilters Function:** This function retrieves the search query and selected price range, then filters the products accordingly.
- **Search Query:** Obtained from the search input field.
- **Price Range:** Obtained from the price-range dropdown. If a price range is selected, it splits the range into minimum and maximum values.
- **Filter Logic:** Filters the window.products array based on whether the product name or description includes the search query and whether the product price falls within the selected price range.
- **Display Products:** Calls the displayProducts function to update the displayed products based on the filtered results.

Event Listeners: Attach event listeners to the search input and price range dropdown to trigger the applyFilters function when their values change.

Debouncing

Objective: Debouncing ensures that the search function is called only after the user has stopped typing for a specified period. This reduces the number of times the search function is executed, improving performance.

Implementation:

1. **Debounce Function:**
    - Creates a delay before executing the function to allow the user to finish typing.

```
function debounce(func, delay) {
    let timerId;
    return function (...args) {
        clearTimeout(timerId);
        timerId = setTimeout(() => func.apply(this, args), delay);
    };
}
```

Attach Debounced Search:

- Uses the debounce function to delay the execution of handleSearch when the user types in the search input.

```
const searchInput = document.getElementById('search');
searchInput.addEventListener('input', debounce(function () {
    handleSearch(this.value);
}, 300)); // Debounce delay of 300ms
```

Throttling

Objective: Throttling ensures that the search function is executed at most once in a specified period. It is useful for cases where you want to limit the number of times a function is called over time.

Implementation:

1. **Throttle Function:**
    - Limits the rate at which the function is executed, ensuring it's only called at most once every specified interval.

```
function throttle(func, limit) {
    let lastFunc;
    let lastRan = 0;
    return function (...args) {
        const context = this;
        const now = Date.now();
        if (now - lastRan >= limit) {
            func.apply(context, args);
            lastRan = now;
        } else {
            clearTimeout(lastFunc);
            lastFunc = setTimeout(function () {
                func.apply(context, args);
```

```
                lastRan = now;
            }, limit - (now - lastRan));
        }
    };
}
```

Attach Throttled Search:

- Uses the throttle function to limit the rate of execution for handleSearch when the user types in the search input.

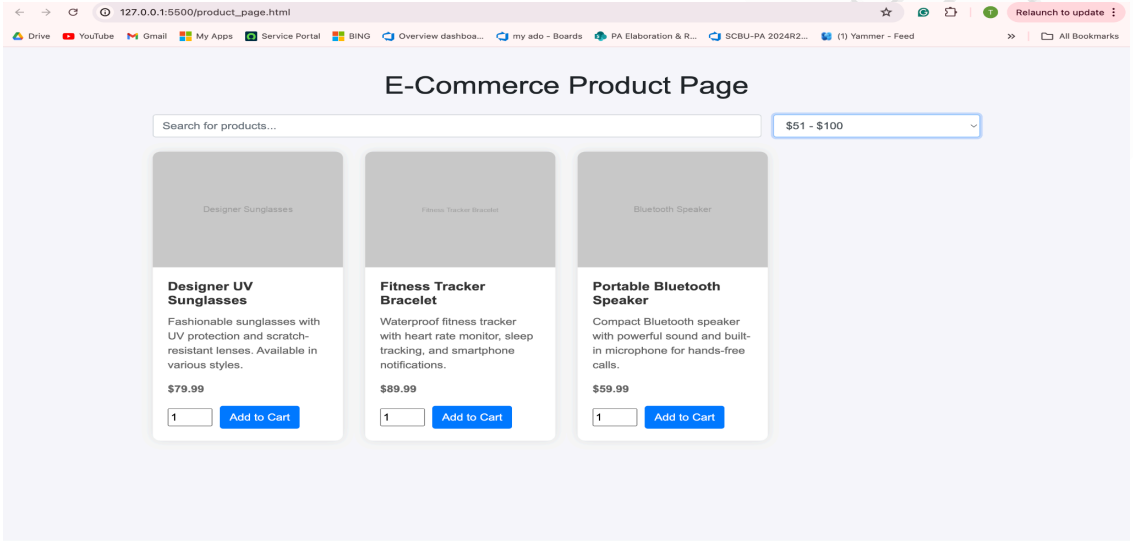
```
// Uncomment to use throttling instead of debouncing
// searchInput.addEventListener('input', throttle(function () {
//     handleSearch(this.value);
// }, 300)); // Throttle limit of 300ms
```

Summary

- **Search Functionality:** Filters and displays products based on user input in the search box, considering both product titles and descriptions.
- **Debouncing:** Delays the execution of the search function until the user has stopped typing for a specified period, improving performance by reducing the number of search operations.
- **Throttling:** Limits the rate at which the search function is executed, ensuring it is only called at most once per specified time interval, which can be useful for controlling the load on the system.

These techniques enhance user experience and performance by optimizing how frequently the search function is triggered and executed.

Output Screenshot:



Interview and FAQ References

CRUD Operations in JavaScript

Understanding CRUD Operations: CRUD (Create, Read, Update, Delete) operations are the basic functions for managing data in applications. In JavaScript, these operations are commonly performed on arrays or objects and can be integrated with APIs for persistent storage.

Interview Questions and Answers:

1. **What are CRUD operations, and how are they used in JavaScript?**
    - **Answer:** CRUD operations refer to Create, Read, Update, and Delete functionalities. In JavaScript, they are used to manage and manipulate data structures such as arrays and objects. For instance, Array.push() is used for Create, Array.find() for Read, Object.assign() for Update, and Array.splice() for Delete.
2. **How do you add a new item to an array in JavaScript?**

Answer: Use the `push()` method to add a new item to the end of an array. Example:

```
let arr = [1, 2, 3];
arr.push(4); // arr is now [1, 2, 3, 4]
```

How can you update an existing item in an array?

- **Answer:** Access the item by its index and assign a new value. Example:

```
let arr = [1, 2, 3];
arr[1] = 5; // arr is now [1, 5, 3]
```

How do you remove an item from an array?

- **Answer:** Use the splice() method to remove an item by index. Example:

```
let arr = [1, 2, 3];
arr.splice(1, 1); // arr is now [1, 3]
```

Explain how you would perform CRUD operations with an API in JavaScript.

- **Answer:** Use fetch() or axios to make HTTP requests to the API. For example, to create a new record, send a POST request; to update, send a PUT request; to delete, send a DELETE request; and to read, send a GET request.

```
// Create
fetch('https://api.example.com/items', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({ name: 'New Item' })
});

// Read
fetch('https://api.example.com/items')
  .then(response => response.json())
  .then(data => console.log(data));

// Update
fetch('https://api.example.com/items/1', {
  method: 'PUT',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({ name: 'Updated Item' })
});

// Delete
fetch('https://api.example.com/items/1', {
  method: 'DELETE'
});
```

Advanced Topics:

- **How do you handle CRUD operations with local storage in JavaScript?**
  - **Answer:** Use localStorage to store data in key-value pairs. Convert data to JSON before storing and parse it when retrieving.

```
// Create
localStorage.setItem('item', JSON.stringify({ name: 'New Item' }));

// Read
let item = JSON.parse(localStorage.getItem('item'));

// Update
item.name = 'Updated Item';
localStorage.setItem('item', JSON.stringify(item));

// Delete
localStorage.removeItem('item');
```

Advanced DOM Manipulation

Understanding Advanced DOM Manipulation: Advanced DOM manipulation involves complex interactions with the Document Object Model, including dynamic updates, animations, and performance optimizations.

codingninjas

Interview Questions and Answers:

1. **What is the difference between innerHTML and textContent?**
    - **Answer:** innerHTML parses and renders HTML content, while textContent retrieves or sets plain text without HTML formatting. Use textContent for plain text and innerHTML for HTML content.
2. **How can you create and insert a new element into the DOM?**
    - **Answer:** Create an element using document.createElement(), set its properties, and insert it into the DOM using methods like appendChild() or insertBefore(). Example:

```
let newDiv = document.createElement('div');
newDiv.textContent = 'Hello World';
document.body.appendChild(newDiv);
```

How can you efficiently handle a large number of DOM updates?

- **Answer:** Minimize reflows and repaints by using techniques like document fragments, batching updates, or using requestAnimationFrame for animations.

```
// Using a document fragment
let fragment = document.createDocumentFragment();
let newDiv = document.createElement('div');
newDiv.textContent = 'Hello World';
fragment.appendChild(newDiv);
document.body.appendChild(fragment);
```

Explain how you would implement drag-and-drop functionality using the DOM.

- **Answer:** Use dragstart, dragover, and drop events to handle dragging and dropping elements. Manage the DataTransfer object to store and retrieve data during the drag-and-drop operation.

```
document.addEventListener('dragstart', (event) => {
  event.dataTransfer.setData('text/plain', event.target.id);
});

document.addEventListener('dragover', (event) => {
  event.preventDefault();
});

document.addEventListener('drop', (event) => {
  event.preventDefault();
  const data = event.dataTransfer.getData('text/plain');
  const droppedElement = document.getElementById(data);
  event.target.appendChild(droppedElement);
});
```

Advanced Topics:

- **How does the MutationObserver API improve DOM manipulation?**
    - **Answer:** MutationObserver allows you to watch for changes in the DOM and execute code when specific mutations occur, providing a more efficient way to detect and respond to DOM changes compared to polling or frequent event listeners.

```
const observer = new MutationObserver((mutations) => {
  mutations.forEach(mutation => {
    console.log('Mutation detected:', mutation);
  });
});

observer.observe(document.body, { childList: true, subtree: true });
```

Debouncing and Throttling

Understanding Debouncing and Throttling:

- **Debouncing:** Ensures that a function is executed only after a specified delay, reducing the number of calls made during continuous input (e.g., search input).
- **Throttling:** Limits the number of times a function can be executed over a specified period, ensuring it is called at most once in that period (e.g., scroll events).

Interview Questions and Answers:

1. **What is debouncing, and how is it implemented in JavaScript?**
    - **Answer:** Debouncing delays the execution of a function until after a specified period has elapsed since the last call. It is implemented using a timer that resets on each call.

```
function debounce(func, delay) {
  let timerId;
  return function(...args) {
    clearTimeout(timerId);
    timerId = setTimeout(() => func.apply(this, args), delay);
  };
}

// Usage example
const debouncedFunction = debounce(() => console.log('Debounced!'), 300);
window.addEventListener('resize', debouncedFunction);
```

What is throttling, and how is it different from debouncing?

- **Answer:** Throttling ensures a function is executed at most once every specified interval, regardless of how often it is triggered. Unlike debouncing, which delays execution until after a pause, throttling enforces a regular execution schedule.

```
function throttle(func, limit) {
  let lastFunc;
  let lastRan = 0;
  return function(...args) {
    const now = Date.now();
    if (now - lastRan >= limit) {
      func.apply(this, args);
      lastRan = now;
    } else {
      clearTimeout(lastFunc);
      lastFunc = setTimeout(() => {
        func.apply(this, args);
        lastRan = now;
      }, limit - (now - lastRan));
    }
  };
}

// Usage example
const throttledFunction = throttle(() => console.log('Throttled!'), 300);
window.addEventListener('scroll', throttledFunction);
```

1. **How do you decide whether to use debouncing or throttling for a particular situation?**
    - **Answer:** Use debouncing when you want to execute a function after a pause, such as in search input or resizing events. Use throttling when you need to ensure periodic execution, like handling scroll or mousemove events.

Advanced Topics:

- **What are the performance considerations when using debouncing and throttling?**
    - **Answer:** Debouncing and throttling can significantly reduce the number of function executions and improve performance by minimizing expensive operations. However, they also introduce a delay or limit execution frequency, which might affect responsiveness. Choose based on the use case requirements and performance testing.

These notes provide a comprehensive overview of CRUD operations in JavaScript, advanced DOM manipulation techniques, and the concepts of debouncing and throttling, with practical examples and advanced topics for a deeper understanding.