Web Application Development Lab Report Lab 3 Exercise

Contents

1. Exercise 1.1: Interactive Form Validator

Function 1: validateUsername(username)

Trigger Code:

```
document.getElementById('username').addEventListener('input', function() {
```

How It Works:

Step 1: Receives username string as parameter.

Step 2: Checks if length is between 4-20 characters:

```
if(username.length < 4 || username.length > 20){
    return false;
}
```

Step 3: Uses regex pattern to verify alphanumeric characters only:

```
const alphanumericRegex = /^[A-Za-z0-9]+$/;
return alphanumericRegex.test(username);
```

Step 4: Returns *true* if valid, *false* if invalid.

Interaction Flow:

User types in input → *input* event fires → Event listener gets value → Calls validateUsername() → Returns true/false → Event listener calls showError() or clearError() → CSS classes update border color.

Function 2: validateEmail(email)

Trigger Code:

```
document.getElementById('email').addEventListener('input', function() {
```

How It Works:

Step 1: Receives email string as parameter.

Step 2: Uses regex to validate email format (requires @ and . symbols):

```
const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
return emailRegex.test(email);
```

Interaction Flow:

User types in input → *input* event fires → Event listener gets value → Calls validateEmail()→ Regex tests pattern → Returns result → Updates UI with border color and message

Function 3: validatePassword(password)

Trigger Code:

```
document.getElementById('password').addEventListener('input', function() {
```

How It Works:

Step 1: Checks minimum length of 8 characters:

```
if (password.length < 8) {
    return false;
}
```

Step 2: Uses regex with lookahead assertions to verify:

```
// Check for at least one uppercase letter and one number
const passRegex = /^(?=.*[A-Z])(?=.*\d).{8,}$/;
return passRegex.test(password);
```

Interaction Flow:

User types in input → *input* event fires → Event listener gets value → Calls validatePassword()→ Validates length and pattern → Returns true/false → Updates UI and enables/disables submit button

Function 4: validatePasswordMatch(pass1, pass2)

Trigger Code:

```
document.getElementById('confirmPassword').addEventListener('input', function() {
```

How It Works:

Step 1: Receives both password strings as parameters.

Step 2: Checks if both exist:

Step 3: Compares using strict equality:

```
if (!pass1 || !pass2) {
    return false;
}

// Check if they match exactly
return pass1 === pass2;
```

Interaction Flow:

User types in confirm field → Gets both password values → Compares them → Shows match/mismatch message

Function 5: showError(fieldId, message)

Trigger Code:

```
showError('username', 'Username must be 4-20 alphanumeric characters');
```

```
showError('email', 'Please enter a valid email address');
```

```
showError('password', 'Password must be at least 8 characters with upp
```

```
showError('confirmPassword', 'Passwords do not match');
```

How It Works:

Step 1: Gets the input element using getElementById:

```
const inputElement = document.getElementById(fieldId);
```

Step 2: Manipulates CSS classes to show red border:

```
// Apply invalid styling (red border)
inputElement.classList.add('invalid');
inputElement.classList.remove('valid');
```

Step 3: Gets error message element by concatenating ID:

```
// Get the error message element
const errorElement = document.getElementById(fieldId + 'Error');
```

Step 4: Sets error message text:

```
errorElement.textContent = message;
```

Step 5: Makes error visible:

```
errorElement.classList.add('show');
```

HTML-CSS-JavaScript Connection:

JavaScript adds **invalid** class → CSS rule *.invalid { border-color: #dc3545; }* applies red border → JavaScript adds **show** class → CSS rule *.error-message.show { display: block; }* makes message visible

Function 6: clearError(fieldId)

Trigger Code:

```
clearError('username');
```

```
clearError('email');
```

```
clearError('password');
```

```
clearError('confirmPassword');
```

How It Works:

Step 1: Gets input element and adds green border:

```
// Get the input element
const inputElement = document.getElementById(fieldId);

// Apply valid styling (green border)
inputElement.classList.add('valid');
inputElement.classList.remove('invalid');
```

Step 2: Clears error message and hides it:

```
// Clear the error text
errorElement.textContent = '';

// Hide the error message
errorElement.classList.remove('show');
```

HTML-CSS-JavaScript Connection:

JavaScript adds **valid** class → CSS applies green border → Removes **show** class → CSS hides error message

Function 7: validateForm()

Trigger Code:

```
validateForm();
```

In each ActionEvent.

How It Works:

Step 1: Gets all input values:

```
const username = document.getElementById('username').value;
const email = document.getElementById('email').value;
const password = document.getElementById('password').value;
const confirmPassword = document.getElementById('confirmPassword').value;
```

Step 2: Validates each field:

```
// Validate each field (returns true/false for each)
const isUsernameValid = validateUsername(username);
const isEmailValid = validateEmail(email);
const isPasswordValid = validatePassword(password);
const isPasswordMatchValid = validatePasswordMatch(password, confirmPassword)
```

Step 3: Combines results using AND operator:

```
// Check if ALL fields are valid
const isFormValid = isUsernameValid && isEmailValid &&
                    isPasswordValid && isPasswordMatchValid;
```

Step 4: Enables/disables submit button:

```
const submitBtn = document.getElementById('submitBtn');
submitBtn.disabled = !isFormValid;
```

2. Project 2: Shopping Cart System

Function 1: addToCart(productId)

Trigger Code:

```
<button class="add-to-cart-btn" onclick="addToCart(${product.id})">
    Add to Cart
</button>
```

How It Works:

Step 1: Finds product in products array:

```
const product = products.find(p => p.id === productId);
```

Array Method: *find()* searches array and returns first matching item where *p.id === productId*

Step 2: Checks if product already in cart:

```
// Step 2: Check if product already exists in cart
const existingItem = cart.find(item => item.id === productId);
```

Step 3a: If exists, increase quantity:

```
if (existingItem) {
    // Product already in cart - increase quantity
    existingItem.quantity += 1;
```

Step 3b: If not exists, add to cart:

```
    existingItem.quantity += 1;
} else {
    // Product not in cart - add it with quantity 1
    cart.push({
        id: product.id,
        name: product.name,
        price: product.price,
        image: product.image,
        quantity: 1
    });
}
```

Step 4: Update display:

```
    // Step 4: Update the cart display
    renderCart();
```

Interaction Flow:

User clicks "Add to Cart" → onclick triggers addToCart(productId) → Finds product → Checks if in cart → Adds or updates quantity → Calls renderCart() → Updates HTML → Calls calculateTotal() → Updates cart count badge

Function 2: removeFromCart(itemId)

Trigger Code:

```
<button onclick="removeFromCart(${item.id})"
        style="margin-left: 10px; background: #dc3545; color: white; padding:
    Remove
</button>
```

How It Works:

Step 1: Filters cart to exclude the item:

```
// Use filter to keep all items EXCEPT the one with matching id
cart = cart.filter(item => item.id !== itemId);
```

Array Method: *filter()* creates new array keeping only items where *item.id !== itemId* (not equal to removed item)

Step 2: Update display:

```
// Update the cart display
renderCart();
```

Interaction Flow:

User clicks "Remove" → Filters cart array → Reassigns cart without the item → Updates HTML display → Recalculates total

Function 3: updateQuantity(productId, change)

Trigger Code:

```
<button onclick="updateQuantity(${item.id}, -1)">-</button>
<span>${item.quantity}</span>
<button onclick="updateQuantity(${item.id}, 1)">+</button>
```

How It Works:

Step 1: Finds item in cart:

```
// Find the item in cart
const item = cart.find(item => item.id === productId);
```

Step 2: Updates quantity:

```
// Update quantity
item.quantity += change;
```

Step 3: Checks if quantity becomes 0:

```
// Remove item if quantity becomes 0 or negative
if (item.quantity <= 0) {
    removeFromCart(productId);
} else {
    // Update display
    renderCart();
}
```

Interaction Flow:

User clicks +/- button → Passes +1 or -1 → Finds item → Adds change to quantity → If 0, removes item → Otherwise updates display

Function 4: calculateTotal()

Trigger Code:

```
// Calculate and display total
calculateTotal();
```

How It Works:

Step 1: Calculates total price using reduce:

```
const total = cart.reduce((sum, item) => {
    return sum + (item.price * item.quantity);
}, 0);
```

Array Method: *reduce()* accumulates a single value by iterating through array.

- *sum* - Running total (starts at 0)
- *item* - Current cart item
- Returns sum + (price × quantity) for each item

Step 2: Updates total display:

```
document.getElementById('cartTotal').textContent = total.toFixed(2);
```

toFixed(2) formats number to 2 decimal places (e.g., 1234.50)

Step 3: Calculates total item count:

```
const totalItems = cart.reduce((sum, item) => sum + item.quantity, 0);
```

Step 4: Updates cart badge:

```
document.getElementById('cartCount').textContent = totalItems;
```

Interaction Flow:

Called by renderCart() → Loops through cart array → Multiplies price × quantity → Sums all subtotals → Formats to 2 decimals → Updates HTML textContent → Counts total items → Updates badge

Function 5: renderProducts()

Trigger Code:

```
renderProducts();
```

How It Works:

Step 1: Gets the products container:

```
const productsGrid = document.getElementById('productsGrid');
```

Step 2: Creates HTML for each product using map:

```
productsGrid.innerHTML = products.map(product => `
    <div class="product-card">
        <div class="product-image">${product.image}</div>
        <div class="product-name">${product.name}</div>
        <div class="product-price">$${product.price.toFixed(2)}</div>
        <button class="add-to-cart-btn" onclick="addToCart(${product.id})">
            Add to Cart
        </button>
    </div>
`).join('');
```

Array Method: *map()* transforms each product into HTML string.

Template Literals: Backticks (`) allow embedding variables with ${variable}.

join(): Combines array of HTML strings into single string.

HTML-CSS-JavaScript Connection:

JavaScript creates HTML with class="product-card" → CSS rule *.product-card {...}* applies styling → onclick attribute connects button to addToCart() function

Function 6: renderCart()

Trigger Code:

```
renderCart();
```

Called after any cart changes.

How It Works:

Step 1: Gets cart container:

```
const cartItems = document.getElementById('cartItems');
```

Step 2: Checks if cart is empty:

```
if (cart.length === 0) {
    // Cart is empty
    cartItems.innerHTML = '<p style="padding: 20px; text-align: center;">Your
```

Step 3: If not empty, creates HTML for each item:

```
} else {
    // Display cart items
    cartItems.innerHTML = cart.map(item => `
        <div class="cart-item">
            <div>
                <span style="font-size: 24px;">${item.image}</span>
                <strong>${item.name}</strong> - $${item.price.toFixed(2)}
            </div>
            <div class="quantity-controls">
                <button onclick="updateQuantity(${item.id}, -1)">-</buttor
                <span>${item.quantity}</span>
                <button onclick="updateQuantity(${item.id}, 1)">+</button>
                <button onclick="removeFromCart(${item.id})"
                        style="margin-left: 10px; background: #dc3545; col
                    Remove
                </button>
            </div>
        </div>
    `).join('');
```

Step 4: Calls calculateTotal to update prices:

```
// Calculate and display total
calculateTotal();
```

Interaction Flow:

Cart array changes → renderCart() called → Checks if empty → Creates HTML for each item → Embeds onclick handlers in buttons → Sets innerHTML → CSS styles appear → Calls calculateTotal() → Updates totals and badge

Function 7: toggleCart()

Trigger Code:

```
<div class="cart-icon" onclick="toggleCart()">
    🛒 Cart
```

How It Works:

Step 1: Gets cart section:

```
const cartSection = document.getElementById('cartSection');
```

Step 2: Checks current display state:

```
// Toggle visibility
if (cartSection.style.display === 'none') {
    cartSection.style.display = 'block';
} else {
    cartSection.style.display = 'none';
}
```

Interaction Flow:

User clicks cart icon → toggleCart() called → Checks current display → If hidden, shows cart → If visible, hides cart