Browser Events

CS6.302 - SOFTWARE SYSTEM DEVELOPMENT

18/09/2025

Static Documents vs. Interactive Applications

1 The Web Pages

- HTML provides structure, CSS provides style.
- The user is a passive consumer. They can only read and scroll.
- Interaction is limited to clicking a link to load a new page.

2 The Web Applications

- The user is an active participant.
- Clicks, key presses, and mouse movements trigger custom logic.
- The page can change without reloading. Data is updated, UI is redrawn, and feedback is instant.

What is an Event? The Formal Definition

An **event** is – a signal fired by the browser when something significant happens, which the developer can run some code in response to.

These "**occurrences**" can be triggered by:

- **User Interaction**: click, keydown, mousemove
- **Browser API Actions**: A file finishes downloading, a video starts playing.
- **Document Lifecycle**: The page has finished parsing (DOMContentLoaded), an image has loaded.

Event Interaction: The Three Core Components

1. **The Event Target:**
   The DOM node on which the event occurred. This could be an element (<button>), the “*document*” itself, or the “*window*”.

2. **The Event Listener (or Observer):**
   A function that is registered to "listen" for a specific type of event on a specific target.

3. **The Event Handler (or Callback):**
   The actual block of code (the function) that is executed when the event is detected by the listener.

Event Interaction: Two Ways to Register Listeners

1. **Inline Event Handlers**

```
<button onclick="myFunction()">Click Me</button>
```

Problems: Mixes HTML and JavaScript, difficult to maintain, can only assign one handler per event.

2. **addEventListener() (The Modern Best Practice):**

```
const myButton = document.querySelector('#myBtn');

myButton.addEventListener('click', myFunction);
```

Benefits:

- Clean Separation of Concerns: HTML is for structure, JS is for behavior.
- Flexibility: Can add multiple listeners for the same event to one element.
- Power: Gives access to advanced features like event propagation controls.

The event Object

- When a handler is executed, the browser automatically passes it a single argument: the ***event*** object.
- This object is a treasure trove of contextual data about the interaction.

- **Key Properties:**
  - ***event.target:*** The element that originated the event.
  - ***event.currentTarget***: The element the listener is attached to. (They can be different!)
  - ***event.type***: The type of event (e.g., "click").
  - ***event.timeStamp***: The time the event occurred.
  - ***event.key, event.code***: For keyboard events.
  - ***event.clientX, event.clientY***: Mouse coordinates relative to the viewport.

Mouse Events

- ***click***: The user presses and releases the primary mouse button. (The workhorse)
- ***dblclick***: The user clicks twice in rapid succession.
- ***mousedown / mouseup:*** Fired separately for the press and release actions. Useful for drag interactions.
- ***mouseenter / mouseleave:*** Fires when the pointer enters or leaves the element's boundaries. Perfect for hover effects. Does not bubble.
- ***mouseover / mouseout***: Similar to above, but fires again for child elements. This can be tricky!
- ***mousemove***: Fires continuously whenever the mouse is moving over the element. (Use with caution - can be performance-intensive!)

Keyboard Events

- ***keydown****:* Fires when the user presses a key. Fires repeatedly if the key is held down.
- ***keyup****:* Fires when the user releases a key.
- **Useful event Properties:**
  - *event.key:* The value of the key pressed (e.g., "a", "Enter", "Shift").
  - *event.code:* The "physical" key on the keyboard (e.g., "KeyA", "Enter", "ShiftLeft").
  - *event.shiftKey*, *event.ctrlKey*, *event.altKey:* Booleans that are true if the respective modifier key was held down.
- *keypress i*s a legacy event; avoid it in favor of keydown.

Form & Input Events

- ***onsubmit***: Fires on the <form> element when it is submitted (either by a button or pressing Enter).
- ***oninput***: Fires on an ***<input>, <select>, or <textarea>*** element immediately after its value changes. The modern standard for real-time feedback.
- ***onchange***: Fires when the value is committed, which for text inputs means when the element loses focus (on blur).
- ***onfocus***: Fires when an element becomes the active element.
- ***onblur***: Fires when an element loses focus.

Document & Window Events

- These events are attached to the “***document”*** or **“window”** object and relate to the page as a whole.
- **DOMContentLoaded**: Fires on “*document”* when the HTML has been fully loaded and parsed. This is the safest and most common time to run your initial JavaScript setup code.
- **onload**: Fires on window after DOMContentLoaded and all assets (CSS, images, etc.) have finished loading.
- **onscroll**: Fires on window or a scrollable element every time the user scrolls. Can fire dozens of times per second!
- **onresize**: Fires on window when the browser window is resized.

Advanced Interaction: The Drag & Drop API

- Drag and Drop is not a single event, but a choreographed sequence of events.
- It allows users to intuitively move elements around the page or transfer data.
- Two Key Players:
  - T**he Draggable Element**: The item being dragged. Must have the HTML attribute ***draggable="true"***.
  - **The Drop Zone**: The area where the draggable element can be dropped.
- This API is the foundation for features like Trello boards, file uploaders, and WYSIWYG editors.

Advanced Interaction: Draggable Element's Story

The three events fire on the element that is being dragged:

1. **ondragstart**
    a. Fires once when the user begins dragging the element.
    b. This is the setup phase. It's where you can specify what data is being dragged.
2. **ondrag**
    a. Fires repeatedly as the user continues to move the mouse while dragging.
    b. Useful for continuous feedback, but use with caution as it fires very frequently.
3. **ondragend**
    a. Fires once when the user releases the mouse button, ending the drag.
    b. This event fires regardless of whether the drop was successful or not. It is the cleanup phase.

Advanced Interaction: The Drop Zone

These events fire on the potential drop target element(s):

- **ondragenter**: Fires when a dragged item first enters the boundaries of the drop zone. (Use this to apply a highlight style).
- **ondragover**: Fires repeatedly as the item is being dragged over the drop zone.
  - CRITICAL: You must call event.preventDefault() in this event's handler to allow a drop!
- **ondragleave**: Fires when the dragged item leaves the drop zone's boundaries. (Use this to remove the highlight style).
- **ondrop**: Fires when the item is released over a valid drop zone. This is where you handle the main logic.
  - You must also call event.preventDefault() here to stop the browser's default handling (like opening the data as a link).

Advanced Interaction: The DataTransfer Object

How does the drop zone know what was dropped on it?

- The *event* object for drag events contains a special property: ***dataTransfer***.
- This object acts as a temporary storage for the data being dragged.
  - event.dataTransfer.setData(format, data)
    - Used in the *dragstart* handler.
    - You attach the data you want to transfer, like the ID of the dragged element.
    - **Example**: event.dataTransfer.setData('text/plain', event.target.id);
  - event.dataTransfer.getData(format)
    - Used in the *drop* handler.
    - You retrieve the data that was set at the start of the drag.
    - **Example**: const elementId = event.dataTransfer.getData('text/plain');

Advanced Interaction: The Full Sequence

A successful drag and drop operation follows this typical flow:

1. The Draggable Element (draggable="true"):
    a. *ondragstart*: You call *event.dataTransfer.setData()* to store the element's ID.
2. On the Drop Zone Element:
    a. *ondragenter*: Fires. You add a CSS class to highlight the drop zone.
    b. *ondragover*: Fires repeatedly. You must call event.preventDefault().
    c. *ondrop*: Fires on release. You call *event.dataTransfer.getData()* to get the ID, find the element, and move it into the drop zone.
3. Back on the Draggable Element:
    a. *ondragend*: Fires. You can perform any necessary cleanup.

Example : https://codepen.io/sanketAd/pen/XJXrYNg

Event Propagation: Bubbling and Capturing

- An event doesn't just happen on one element. It takes a journey through the **DOM**.
- This journey has three phases:
  - **Capturing Phase**: The event travels from the window down to the target's parent elements.
  - **Target Phase**: The event reaches the event.target itself.
  - Bubbling Phase: The event travels back up from the target to the window.
- By default, all event listeners operate in the BUBBLING phase.

Example: https://javascript.info/article/bubbling-and-capturing/bubble-target/

Event Propagation: Controlling the Flow

- The “event” object gives us methods to control this flow.
- **event.stopPropagation()**
  - Stops the event from traveling any further in its current phase (capturing or bubbling).
  - **Use Case**: You have a clickable element inside another clickable element. When you click the inner one, you don't want the outer element's click handler to also fire.
- **event.preventDefault()**
  - Stops the browser's default action for that event.
  - **Use Cases:**
    - Preventing a <form> from doing a full-page reload on “*submit*”.
    - Preventing a link <a> from navigating to a new URL on “*click*”.

Event Propagation: Event Delegation

- **The Problem:** Imagine a list with 100 items, each needing a click handler. Or a list where new items are added dynamically. Attaching a listener to every single item is inefficient.
- The Solution: **Event Delegation**
  - Attach a single event listener to the parent container (e.g., the <ul>).
  - When an event bubbles up from a child (an <li>), the parent's listener catches it.
  - Inside the handler, use event.target to identify which specific child was clicked.
- **Benefits**:
- Massively improves performance and reduces memory usage.
- Automatically works for elements added to the container later!

Example: https://en.js.cx/article/event-delegation/bagua/

Summary & Key Takeaways

- Events turn static documents into interactive applications.
- Always use “***addEventListener()***” for a clean separation of concerns and more power.
- The ***event*** object is your essential toolkit for understanding the context of an interaction.
- Events ***propagate*** through the DOM (bubbling is the default and most useful phase).
- Use ***preventDefault()*** to control default browser actions and ***stopPropagation()*** to control the event flow.
- Leverage Event Delegation for efficient and scalable event handling.

Resources

- https://javascript.info/web-components
- https://javascript.info/events
- https://javascript.info/onload-ondomcontentloaded
- https://aws.amazon.com/what-is/event-listener/
- https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener
- https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Events
- https://www.freecodecamp.org/news/dom-events-and-javascript-event-listeners/
- https://javascript.info/event-delegation

Thank You!