Resurse suplimentare pe care recomand să le parcurgi

Ce sunt evenimentele?

Evenimentele sunt semnale generate de browser atunci când "ceva se întâmplă" cu pagina sau cu elementele sale. Acestea pot fi:

- Interacțiuni ale utilizatorului (click-uri, apăsări de taste, mișcări ale mouse-ului)
- Modificări în pagină (încărcare, redimensionare)
- Modificări în starea elementelor (schimbări de focus, validări de formular)

Evenimentele ne permit să construim interfețe interactive, detectând aceste acțiuni și răspunzând la ele cu cod JavaScript.

Modelul evenimentelor în JavaScript

Sistemul de evenimente din JavaScript funcționează pe baza conceptului de "event handlers" (gestionari de evenimente):

1. Înregistrarea - Specificăm ce elemente ascultăm și pentru ce evenimente
2. Declanșarea - Utilizatorul sau browserul declanșează un eveniment
3. Execuția - Funcția noastră de gestionare (event handler) este executată

Tipuri comune de evenimente

Evenimentele mouse-ului

- `click` - Declanșat când utilizatorul face click pe un element
- `dblclick` - Declanșat la dublu click
- `mousedown`/`mouseup` - Declanșate când butonul mouse-ului este apăsat/eliberat
- `mouseover`/`mouseout` - Declanșate când cursorul intră/iese dintr-un element

- `mouseenter`/`mouseleave` - Similar cu mouseover/mouseout, dar nu se propagă la elementele copil
- `mousemove` - Declanșat când cursorul se mișcă în interiorul unui element

Evenimentele tastaturii

- `keydown` - Declanșat când o tastă este apăsată
- `keyup` - Declanșat când o tastă este eliberată
- `keypress` - Declanșat când o tastă care produce un caracter este apăsată (depreciat)

Evenimentele formularelor

- `submit` - Declanșat când un formular este trimis
- `reset` - Declanșat când un formular este resetat
- `change` - Declanșat când valoarea unui element de input se schimbă și pierde focus-ul
- `input` - Declanșat imediat ce valoarea unui element de input se schimbă
- `focus`/`blur` - Declanșate când un element primește/pierde focus-ul
- `select` - Declanșat când text dintr-un element input sau textarea este selectat

Evenimentele documentului/ferestrei

- `load` - Declanșat când pagina a terminat de încărcat
- `DOMContentLoaded` - Declanșat când DOM-ul este complet încărcat, fără a aștepta resursele
- `resize` - Declanșat când fereastra browserului este redimensionată
- `scroll` - Declanșat când utilizatorul derulează pagina sau un element cu overflow
- `beforeunload`/`unload` - Declanșate când utilizatorul părăsește pagina

Adăugarea event listener-ilor

Există mai multe moduri de a adăuga event listeners în JavaScript:

1. Atributul HTML (nerecomandată)

```html
<button onclick="alert('Buton apăsat!')">Click Me</button>
```

Dezavantaje:

- Amestecă HTML cu JavaScript, încălcând principiul separării responsabilităților
- Limitează posibilitățile de gestionare a evenimentelor
- Poate crea probleme de performanță și mentenanță

2. Proprietăți de evenimente DOM

```
const button = document.querySelector('#myButton')
button.onclick = function () {
  alert('Buton apăsat prin proprietate DOM!')
}
```

Avantaje:

- Separă JavaScript de HTML
- Simplu și direct

Dezavantaje:

- Permite doar un singur handler per eveniment per element (suprascrie handlere anterioare)
- Nu permite capturarea evenimentelor (doar faza bubbling)

3. Metoda addEventListener() (recomandată)

```
const button = document.querySelector('#myButton')
button.addEventListener('click', function () {
  alert('Buton apăsat cu addEventListener!')
})

// Adăugarea mai multor handlere pentru același eveniment
button.addEventListener('click', function () {
  console.log('Și acest handler va fi executat!')
})
```

Avantaje:

- Permite atașarea mai multor handlere pentru același eveniment
- Oferă control asupra fazei evenimentului (capturing vs. bubbling)
- Metodă modernă și recomandată pentru toate proiectele noi

Obiectul eveniment

Când un event handler este executat, el primește automat un **obiect eveniment** ca argument. Acest obiect conține informații despre evenimentul declanșat:

```
const button = document.querySelector('#myButton')
button.addEventListener('click', function (event) {
  console.log('Eveniment declanșat:', event.type)
  console.log('Element țintă:', event.target)
  console.log('Coordonate click:', event.clientX, event.clientY)
})
```

Proprietăți comune ale obiectului eveniment

- `event.type` - Tipul evenimentului (ex: "click", "keydown")
- `event.target` - Elementul care a declanșat evenimentul
- `event.currentTarget` - Elementul la care este atașat handler-ul (diferit de target în timpul propagării)
- `event.clientX`/`event.clientY` - Coordonatele mouse-ului (relative la viewport)
- `event.pageX`/`event.pageY` - Coordonatele mouse-ului (relative la document)
- `event.key`/`event.code` - Informații despre tasta apăsată (pentru evenimentele keyboard)
- `event.preventDefault()` - Metodă pentru a preveni comportamentul implicit al browser-ului
- `event.stopPropagation()` - Metodă pentru a opri propagarea evenimentului

Propagarea evenimentelor (Event Propagation)

Când un eveniment este declanșat pe un element, el parcurge 3 faze:

1. **Capturing Phase** - Evenimentul coboară de la document către elementul țintă
2. **Target Phase** - Evenimentul ajunge la elementul țintă
3. **Bubbling Phase** - Evenimentul urcă înapoi de la elementul țintă către document

```html
<div id="outer">
  <div id="inner">
    <button id="button">Click me</button>
  </div>
</div>
```

```javascript
// Exemplu de propagare a evenimentului

// Capturing phase (jos către țintă)
document.querySelector('#outer').addEventListener(
  'click',
  function () {
    console.log('Outer div - Capturing phase')
  },
  true
) // al treilea parametru true activează faza de capturing

document.querySelector('#inner').addEventListener(
  'click',
  function () {
    console.log('Inner div - Capturing phase')
  },
  true
)

// Target și Bubbling phase (de la țintă în sus)
document.querySelector('#button').addEventListener('click', function () {
  console.log('Button clicked - Target phase')
})

document.querySelector('#inner').addEventListener('click', function () {
  console.log('Inner div - Bubbling phase')
})

document.querySelector('#outer').addEventListener('click', function () {
  console.log('Outer div - Bubbling phase')
})
```

Când se face click pe buton, ordinea de afișare va fi:

1. "Outer div - Capturing phase"
2. "Inner div - Capturing phase"
3. "Button clicked - Target phase"
4. "Inner div - Bubbling phase"
5. "Outer div - Bubbling phase"

Oprirea propagării

În unele cazuri, vrem să oprim propagarea evenimentului:

```
document.querySelector('#button').addEventListener('click', function (event) {
  console.log('Button clicked')
  event.stopPropagation() // Oprește propagarea evenimentului
  // Event handlers de pe div-urile părinte nu vor fi executate
})
```

Prevenirea comportamentului implicit

Multe elemente HTML au comportamente implicite (ex: link-urile navighează către URL-ul din href, formularele se trimit). Putem preveni aceste comportamente:

```
// Prevenirea navigării la apăsarea unui link
document.querySelector('a').addEventListener('click', function (event) {
  event.preventDefault() // Previne comportamentul implicit
  console.log('Link-ul a fost apăsat, dar navigarea a fost blocată')
})

// Prevenirea trimiterii unui formular
document.querySelector('form').addEventListener('submit', function (event) {
  event.preventDefault() // Previne trimiterea formularului
  console.log('Formular trimis, dar acțiunea implicită a fost blocată')
  // Aici putem adăuga logica noastră personalizată
})
```

Delegarea evenimentelor (Event Delegation)

Delegarea evenimentelor este o tehnică eficientă bazată pe propagarea (bubbling) evenimentelor. În loc să atașăm event listeners pentru fiecare element individual, putem atașa un singur listener la un părinte comun:

```html
<ul id="taskList">
  <li>Sarcina 1</li>
  <li>Sarcina 2</li>
  <li>Sarcina 3</li>
  <!-- Mai multe elemente pot fi adăugate dinamic -->
</ul>
```

```javascript
// Fără delegare (ineficient pentru liste dinamice)
const items = document.querySelectorAll('#taskList li')
items.forEach((item) => {
  item.addEventListener('click', function () {
    console.log('Item clicked:', this.textContent)
  })
})

// Cu delegare (eficient, funcționează și pentru elemente adăugate dinamic)
document.querySelector('#taskList').addEventListener('click', function (event) {
  if (event.target.tagName === 'LI') {
    console.log('Item clicked:', event.target.textContent)
  }
})
```

Avantaje ale delegării:

- Reduce numărul de event listeners
- Funcționează pentru elemente adăugate dinamic
- Consumă mai puțină memorie
- Cod mai curat și mai ușor de întreținut

Evenimente personalizate (Custom Events)

JavaScript permite și crearea de evenimente personalizate:

```
// Crearea unui eveniment personalizat
const evenimentPersonalizat = new CustomEvent('itemAdded', {
  detail: {
    id: 123,
    name: 'Produs nou',
  },
  bubbles: true,
})

// Declanșarea evenimentului
document.querySelector('#productList').dispatchEvent(evenimentPersonalizat)

// Ascultarea evenimentului
document
  .querySelector('#productList')
  .addEventListener('itemAdded', function (event) {
    console.log('Produs adăugat:', event.detail.name, 'cu ID:', event.detail.id)
  })
```

Eliminarea event listener-ilor

Pentru a preveni memory leaks, în special în aplicații complexe, este bine să eliminăm event listeners când nu mai avem nevoie de ei:

```
function clickHandler() {
  console.log('Buton apăsat')
}

const button = document.querySelector('#myButton')

// Adăugare listener
button.addEventListener('click', clickHandler)

// Eliminare listener (trebuie să folosim aceeași funcție de referință)
button.removeEventListener('click', clickHandler)
```

Pentru funcții anonime, trebuie să păstrăm o referință la funcție:

```
const button = document.querySelector('#myButton')
const handlerFn = function () {
  console.log('Buton apăsat')
}

button.addEventListener('click', handlerFn)

// Mai târziu
button.removeEventListener('click', handlerFn)
```

Modele de evenimente comune

1. Eveniment click pentru toggle

```
// Toggle pentru afișare/ascundere
document.querySelector('#toggleButton').addEventListener('click', function () {
  const content = document.querySelector('#content')
  if (content.style.display === 'none' || content.style.display === '') {
    content.style.display = 'block'
  } else {
    content.style.display = 'none'
  }
})

// Alternativ, folosind classList.toggle
document.querySelector('#toggleButton').addEventListener('click', function () {
  document.querySelector('#content').classList.toggle('hidden')
})
```

2. Validare formular la submit

```
document.querySelector('#myForm').addEventListener('submit', function (event) {
  const email = document.querySelector('#email').value
  const password = document.querySelector('#password').value
  let isValid = true

  // Resetare mesaje de eroare
  document
    .querySelectorAll('.error-message')
    .forEach((el) => (el.textContent = ''))

  // Validare email
  if (!email.includes('@')) {
    document.querySelector('#emailError').textContent = 'Email invalid!'
    isValid = false
  }

  // Validare parolă
  if (password.length < 8) {
    document.querySelector('#passwordError').textContent =
      'Parola trebuie să aibă minim 8 caractere!'
    isValid = false
  }

  // Dacă formularul nu este valid, prevenim trimiterea
  if (!isValid) {
    event.preventDefault()
  }
})
```

3. Event listeners pentru tastatură

```
// Monitorizare tastatură pentru un joc simplu
document.addEventListener('keydown', function (event) {
  const player = document.querySelector('#player')
  const currentLeft = parseInt(window.getComputedStyle(player).left) || 0
  const currentTop = parseInt(window.getComputedStyle(player).top) || 0
  const step = 10 // pixeli per apăsare

  switch (event.key) {
    case 'ArrowUp':
      player.style.top = currentTop - step + 'px'
      break
    case 'ArrowDown':
      player.style.top = currentTop + step + 'px'
      break
    case 'ArrowLeft':
      player.style.left = currentLeft - step + 'px'
      break
    case 'ArrowRight':
      player.style.left = currentLeft + step + 'px'
      break
  }
})
```

4. Drag and drop

```
// Element care poate fi tras
const draggable = document.querySelector('#draggable')
let offsetX,
  offsetY,
  isDragging = false

draggable.addEventListener('mousedown', function (event) {
  isDragging = true
  offsetX = event.clientX - draggable.getBoundingClientRect().left
  offsetY = event.clientY - draggable.getBoundingClientRect().top
  draggable.style.cursor = 'grabbing'
})

document.addEventListener('mousemove', function (event) {
  if (!isDragging) return

  const x = event.clientX - offsetX
  const y = event.clientY - offsetY

  draggable.style.left = x + 'px'
  draggable.style.top = y + 'px'
})

document.addEventListener('mouseup', function () {
  isDragging = false
  draggable.style.cursor = 'grab'
})
```

Debouncing și Throttling pentru evenimente frecvente

Pentru evenimente care se declanșează frecvent (resize, scroll, input), este bine să folosim tehnici pentru a limita numărul de execuții ale handler-ului:

Debouncing

Execută funcția doar după ce utilizatorul se oprește din acțiunea respectivă:

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

// Utilizare
const efficientResize = debounce(function () {
  console.log('Window resized - debounced!')
  // Cod care se execută doar la 300ms după ultima redimensionare
}, 300)

window.addEventListener('resize', efficientResize)
```

Throttling

Limitează numărul de execuții la un interval minim de timp:

```
function throttle(func, limit) {
  let inThrottle
  return function (...args) {
    if (!inThrottle) {
      func.apply(this, args)
      inThrottle = true
      setTimeout(() => {
        inThrottle = false
      }, limit)
    }
  }
}

// Utilizare
const efficientScroll = throttle(function () {
  console.log('Window scrolled - throttled!')
  // Cod care se execută maxim o dată la 300ms
}, 300)

window.addEventListener('scroll', efficientScroll)
```

Cele mai bune practici pentru evenimentele DOM

1. **Folosește addEventListener în loc de proprietăți on{event}**
    - Permite multiple handlere
    - Oferă mai mult control
2. **Folosește delegarea evenimentelor când este posibil**
    - Reduce numărul de event listeners
    - Funcționează pentru elemente adăugate dinamic
3. **Evită evenimente inline în HTML**
    - Menține separarea dintre HTML și JavaScript
    - Evită potențiale probleme de securitate
4. **Folosește debounce/throttle pentru evenimente frecvente**
    - Optimizează performanța
    - Reduce numărul de calcule inutile
5. **Elimină event listeners când nu mai sunt necesare**
    - Previne memory leaks
    - Eliberează resurse
6. **Codifică funcțiile event handler separat**
    - Îmbunătățește mentenanța
    - Permite reutilizarea codului