Far fronte agli eventi… in JavaScript - Parte 4

Prima o poi capita a tutti di sviluppare uno script complesso, se non una vera e propria webapp. Siamo abituati però a focalizzare il nostro codice più sugli elementi di input che non sul comportamento dell'applicazione. Ad esempio, per aggiungere un prodotto al carrello assegniamo al *listener* "submit" della form che contiene il pulsante "Aggiungi al carrello" un'unica funzione che contiene sia i controlli sul prodotto che i messaggi d'errore e di successo. Facciamo così perché usiamo soltanto gli eventi di default forniti dal browser.

Non sarebbe molto più leggibile, mantenibile, slegato dal markup un codice simile a questo?

Ebbene, è possibile farlo. Proprio così com'è scritto, se usate jQuery. Se invece usate JavaScript "vanilla", cioè puro e semplice, la sintassi è la seguente:

Il codice riportato in questo articolo è compatibile con tutti i principali browser tranne quelli molto vecchi, come Internet Explorer 8 e precedenti, per i quali esistono semplici fix facilmente reperibili.

Passare dei parametri

In JavaScript, per passare dei parametri alla funzione associata ad un *listener* è possibile usare il costruttore *CustomEvent* che supporta un secondo parametro, il cui valore è disponibile nell'attributo "detail" dell'oggetto evento.

Usando jQuery, invece, basta inserire un oggetto come secondo parametro, facendo passare la funzione come terzo parametro. Il valore del secondo parametro è disponibile nell'attributo "data" dell'oggetto evento.

In questo modo però passiamo il parametro **quando l'evento viene assegnato**. Di conseguenza, ogni volta in cui l'evento viene scatenato il valore del parametro sarà sempre lo stesso. Per gestire tutti i colori del nostro semaforo quindi dovremmo impostare tre *listener* (cioè tre "on cambiaColoreX"), che passano ognuno un colore diverso, il tutto gestito da una funzione che calcola quale colore mostrare e che di conseguenza scatena questo o quell'evento.

Piuttosto, sarebbe meglio spostare la logica sull'elemento stesso, creando un unico *listener* che gestisca il cambio colore in base al colore esistente. Avremmo quindi bisogno di passare un parametro **quando l'evento viene scatenato**. In jQuery possiamo approfittare del metodo trigger(), che ci dà la possibilità di inserire come secondo parametro un array o un oggetto.

Purtroppo, in JavaScript puro il metodo *dispatchEvent* non consente di passare parametri, perciò il custom event dev'essere creato “al volo”.

Dare un nome agli eventi

I nomi degli eventi dovrebbero contenere soltanto i caratteri alfanumerici, punto, due punti ed underscore (in jQuery quest’ultimo è riservato). Esistono due utili convenzioni per:

- **evitare che i *custom element* vengano sovrascritti**: a lungo termine, le nuove versioni di JavaScript potrebbero introdurre nuovi *listener*. Ad esempio, se il nome del nostro evento “addtocart” diventasse uno di quelli di default di JavaScript tanto quanto “click” sarebbe un bel problema! Per evitarlo, nel nome dell’evento **si usa il carattere “:”**, preferendo ad esempio una nomenclatura come “cart:add”.
- **individuare gli eventi della nostra applicazione**: quando si scrive uno script che viene usato assieme ad altri dalle provenienze più disparate (sempre nostra o anche da terze parti) come ad esempio un plugin di jQuery, può tornare utile individuare tutti gli eventi da esso impostati. Per esempio, potremmo voler disabilitare tutti gli eventi impostati da un nostro script. Per farlo, nel nome dell’evento **si usa il carattere “.”** seguito da un “namespace”. In pratica, usando la nomenclatura click.myplugin raggruppiamo uno specifico evento “click” nel gruppo ‘myplugin’. In seguito, scrivendo soltanto .off(‘.myplugin’) sarà possibile disabilitare tutti i *listener* del gruppo “myplugin”. I namespace non vengono riportati in *event.type*, ed è possibile usare più namespace contemporaneamente (proprio come quando in css si individua un elemento usando più classi dell’elemento stesso).

Non solo click: eventi quando meno te li aspetti

Il bello dei *custom events* è che possono gestire eventi che non vengono generati da un input dell’utente. Pensate ad esempio di voler mostrare in tempo reale una notifica nel browser del visitatore di un e-commerce che avvisa che un prodotto nella sua wishlist è appena tornato disponibile. Se non userete i web socket, la vostra scelta ricadrà su una chiamata ajax periodica, che a seconda del contenuto richiamerà una funzione. Qualcosa di simile a:

Ci sono anche dei casi in cui l’input dell’utente c’è ma potrebbe essere involontario: quando un utente tiene premuto un tasto, infatti, gli eventi associati a *keydown* (scatenato da qualsiasi tasto) e a *keypress* (scatenato dai tasti che producono un output, diversi quindi da control, alt, shift, esc, eccetera) vengono in realtà ripetuti più volte con un tempo che può essere cambiato via software e che cambia da tastiera a tastiera. Per quanto riguarda la mia, il tempo tra la pressione del tasto e la

prima ripetizione è di 500ms circa, mentre le altre avvengono ogni 33ms circa. In pratica una funzione associata a *keydown* viene eseguita secondo la seguente timeline:

Esecuzione 1 = 0ms
Esecuzione 2 = 500ms
Esecuzione 3 = 533ms
Esecuzione 4 = 566ms
Esecuzione 5 = 599ms
Esecuzione 6 = 632ms
// eccetera

La cosa di per sè è utile per alcune applicazioni, ma può comportare anche un paio di problemi.

Il primo è un problema di reattività nel caso in cui si voglia dare un'esperienza d'uso fluida, come ad esempio durante lo zoom di un'immagine, una chat, un gioco, un'applicazione per smart tv: quando ogni millisecondo è importante, doverne aspettare 500 può diventare un bel problema! Inoltre, molte applicazioni girano a 60fps, cioè si basano su un loop che si ripete ogni 17ms, il che significa che tra una ripetizione di *keydown* di quelle "veloci" (33ms) e l'altra perdiamo metà della reattività che potremmo avere.

Il secondo problema è per l'appunto che queste ripetizioni possono essere "inaspettate", causando grossi disagi ad esempio durante giochi online o aste, dove fare dalle 17 alle 30 puntate al secondo quando l'utente se ne aspetta soltanto una potrebbe procurare qualche fastidio...

Per gestire entrambe le problematiche, si usano *keydown* e *keyup* per salvare lo stato del tasti in una variabile (se premuto = true, se non premuto = false) che viene controllata a seconda delle proprie esigenze.

Volete cambiare la frequenza delle ripetizioni? Impostate un *setTimeout* o un *requestAnimationFrame* e al suo interno verificate ogni tot millisecondi lo stato del tasto.

Volete evitare le ripetizioni? Se possibile eseguite la funzione *onkeyup*, altrimenti eseguitela soltanto se lo stato del tasto era false ed ora è true.

Ma non basta usare qualche funzione?

Insomma, questi *custom events* sono molto belli, ma a molti potrebbero sembrare soltanto una complicazione. Servono davvero nel lavoro di tutti i giorni? Per i lavori più semplici direi proprio di no, ma al giorno d'oggi la complessità generale per lo sviluppo dei siti va aumentando, così come il numero di piattaforme sulle quali si può andare a sviluppare in JavaScript: meglio prepararsi a qualche cambiamento, provando a sviluppare applicazioni *event-driven* per capire se si tratta di un approccio adatto a noi e ai nostri progetti! Inoltre, un approccio basato sul comportamento può

essere molto interessante per chi si occupa di analisi delle attività compiute dagli utenti, e di conseguenza potrebbe aprire le porte a ghiotte collaborazioni.

Per ora è tutto?

Con questo articolo si conclude il ciclo di approfondimento sugli eventi JavaScript. Non è detto però che sia finita qui: se ritenete che ci siano altri aspetti di cui valga la pena parlare scrivetelo qui sotto e se sarà possibile aggiungeremo altri articoli all'argomento!