27th November 2019

Dispatching custom events

We can not only assign handlers, but also generate events from JavaScript.

Custom events can be used to create “graphical components”. For instance, a root element of our own JS-based menu may trigger events telling what happens with the menu: `open` (menu open), `select` (an item is selected) and so on. Another code may listen for the events and observe what’s happening with the menu.

We can generate not only completely new events, that we invent for our own purposes, but also built-in ones, such as `click`, `mousedown` etc. That may be helpful for automated testing.

Event constructor

Build-in event classes form a hierarchy, similar to DOM element classes. The root is the built-in Event class.

We can create `Event` objects like this:

```
1  let event = new Event(type[, options]);
```

Arguments:

- *type* – event type, a string like `"click"` or our own like `"my-event"`.
- *options* – the object with two optional properties:
  - `bubbles: true/false` – if `true`, then the event bubbles.
  - `cancelable: true/false` – if `true`, then the “default action” may be prevented. Later we’ll see what it means for custom events.

  By default both are false: `{bubbles: false, cancelable: false}`.

dispatchEvent

After an event object is created, we should “run” it on an element using the call `elem.dispatchEvent(event)`.

Then handlers react on it as if it were a regular browser event. If the event was created with the `bubbles` flag, then it bubbles.

In the example below the `click` event is initiated in JavaScript. The handler works same way as if the button was clicked:

```
<button id="elem" onclick="alert('Click!');">Autoclick</button>

<script>
  let event = new Event("click");
  elem.dispatchEvent(event);
</script>
```

event.isTrusted

There is a way to tell a “real” user event from a script-generated one.

The property `event.isTrusted` is `true` for events that come from real user actions and `false` for script-generated events.

Bubbling example

We can create a bubbling event with the name `"hello"` and catch it on `document`.

All we need is to set `bubbles` to `true`:

```
<h1 id="elem">Hello from the script!</h1>

<script>
  // catch on document...
  document.addEventListener("hello", function(event) { // (1)
    alert("Hello from " + event.target.tagName); // Hello from H1
  });

  // ...dispatch on elem!
  let event = new Event("hello", {bubbles: true}); // (2)
  elem.dispatchEvent(event);

  // the handler on document will activate and display the message.

</script>
```

Notes:

1. We should use `addEventListener` for our custom events, because `on<event>` only exists for built-in events, `document.onhello` doesn’t work.
2. Must set `bubbles:true`, otherwise the event won’t bubble up.

The bubbling mechanics is the same for built-in (`click`) and custom (`hello`) events. There are also capturing and bubbling stages.

MouseEvent, KeyboardEvent and others

Here’s a short list of classes for UI Events from the UI Event specification:

- `UIEvent`
- `FocusEvent`

- `MouseEvent`
- `WheelEvent`
- `KeyboardEvent`
- ...

We should use them instead of `new Event` if we want to create such events. For instance, `new MouseEvent("click")`.

The right constructor allows to specify standard properties for that type of event.

Like `clientX/clientY` for a mouse event:

```
let event = new MouseEvent("click", {
  bubbles: true,
  cancelable: true,
  clientX: 100,
  clientY: 100
});

alert(event.clientX); // 100
```

Please note: the generic `Event` constructor does not allow that.

Let's try:

```
let event = new Event("click", {
  bubbles: true, // only bubbles and cancelable
  cancelable: true, // work in the Event constructor
  clientX: 100,
  clientY: 100
});

alert(event.clientX); // undefined, the unknown property is ignored!
```

Technically, we can work around that by assigning directly `event.clientX=100` after creation. So that's a matter of convenience and following the rules. Browser-generated events always have the right type.

The full list of properties for different UI events is in the specification, for instance, MouseEvent.

Custom events

For our own, completely new events types like `"hello"` we should use `new CustomEvent`. Technically CustomEvent is the same as `Event`, with one exception.

In the second argument (object) we can add an additional property `detail` for any custom information that we want to pass with the event.

For instance:

```
<h1 id="elem">Hello for John!</h1>

<script>
```

```
  // additional details come with the event to the handler
  elem.addEventListener("hello", function(event) {
    alert(event.detail.name);
  });

  elem.dispatchEvent(new CustomEvent("hello", {
    detail: { name: "John" }
  }));
</script>
```

The `detail` property can have any data. Technically we could live without, because we can assign any properties into a regular `new Event` object after its creation. But `CustomEvent` provides the special `detail` field for it to evade conflicts with other event properties.

Besides, the event class describes "what kind of event" it is, and if the event is custom, then we should use `CustomEvent` just to be clear about what it is.

event.preventDefault()

Many browser events have a "default action", such as nagivating to a link, starting a selection, and so on.

For new, custom events, there are definitely no default browser actions, but a code that dispatches such event may have its own plans what to do after triggering the event.

By calling `event.preventDefault()`, an event handler may send a signal that those actions should be canceled.

In that case the call to `elem.dispatchEvent(event)` returns `false`. And the code that dispatched it knows that it shouldn't continue.

Let's see a practical example – a hiding rabbit (could be a closing menu or something else).

Below you can see a `#rabbit` and `hide()` function that dispatches `"hide"` event on it, to let all interested parties know that the rabbit is going to hide.

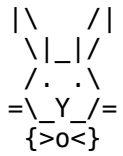
Any handler can listen for that event with `rabbit.addEventListener('hide',...)` and, if needed, cancel the action using `event.preventDefault()`. Then the rabbit won't disappear:

```
<pre id="rabbit">
  |\   /|
   \|_|/
   /. .\
  =\_Y_/=
   {>o<}
</pre>
<button onclick="hide()">Hide()</button>

<script>
  // hide() will be called automatically in 2 seconds
  function hide() {
    let event = new CustomEvent("hide", {
      cancelable: true // without that flag preventDefault doesn't work
    });
    if (!rabbit.dispatchEvent(event)) {
      alert('The action was prevented by a handler');
    } else {
      rabbit.hidden = true;
```

```
    }
  }

  rabbit.addEventListener('hide', function(event) {
    if (confirm("Call preventDefault?")) {
      event.preventDefault();
    }
  });
</script>
```



Hide()

Please note: the event must have the flag `cancelable: true`, otherwise the call `event.preventDefault()` is ignored.

Events-in-events are synchronous

Usually events are processed asynchronously. That is: if the browser is processing `onclick` and in the process a new event occurs, then it waits until the `onclick` processing is finished.

The exception is when one event is initiated from within another one.

Then the control jumps to the nested event handler, and after it goes back.

For instance, here the nested `menu-open` event is processed synchronously, during the `onclick`:

```
<button id="menu">Menu (click me)</button>

<script>
  menu.onclick = function() {
    alert(1);

    // alert("nested")
    menu.dispatchEvent(new CustomEvent("menu-open", {
      bubbles: true
    }));

    alert(2);
  };

  document.addEventListener('menu-open', () => alert('nested'));
</script>
```

Menu (click me)

The output order is: 1 → nested → 2.

Please note that the nested event `menu-open` fully bubbles up and is handled on the `document`. The propagation and handling of the nested event must be fully finished before the processing gets back to the outer code (`onclick`).

That's not only about `dispatchEvent`, there are other cases. JavaScript in an event handler can call methods that lead to other events – they are too processed synchronously.

If we don't like it, we can either put the `dispatchEvent` (or other event-triggering call) at the end of `onclick` or, maybe better, wrap it in zero-delay `setTimeout`:

```
<button id="menu">Menu (click me)</button>

<script>
  menu.onclick = function() {
    alert(1);

    // alert(2)
    setTimeout(() => menu.dispatchEvent(new CustomEvent("menu-open", {
      bubbles: true
    })));

    alert(2);
  };

  document.addEventListener('menu-open', () => alert('nested'));
</script>
```

Now `dispatchEvent` runs asynchronously after the current code execution is finished, including `mouse.onclick`, so event handlers are totally separate.

The output order becomes: 1 → 2 → nested.

Summary

To generate an event from code, we first need to create an event object.

The generic `Event(name, options)` constructor accepts an arbitrary event name and the `options` object with two properties:

- `bubbles: true` if the event should bubble.
- `cancelable: true` if the `event.preventDefault()` should work.

Other constructors of native events like `MouseEvent`, `KeyboardEvent` and so on accept properties specific to that event type. For instance, `clientX` for mouse events.

For custom events we should use `CustomEvent` constructor. It has an additional option named `detail`, we should assign the event-specific data to it. Then all handlers can access it as `event.detail`.

Despite the technical possibility to generate browser events like `click` or `keydown`, we should use with the great care.

We shouldn't generate browser events as it's a hacky way to run handlers. That's a bad architecture most of the time.

Native events might be generated:

- As a dirty hack to make 3rd-party libraries work the needed way, if they don't provide other means of interaction.
- For automated testing, to "click the button" in the script and see if the interface reacts correctly.

Custom events with our own names are often generated for architectural purposes, to signal what happens inside our menus, sliders, carousels etc.

Previous lesson Next lesson

Share

Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)