Javascript (Https://Www.Sitepoint.Com/Javascript/) > March 25, 2015 > By Nick Salloum (https://www.sitepoint.com/author/nsalloum/)

Building a Custom Right-Click (Context) Menu with JavaScript

Web applications, as opposed to just informational websites, are being more and more utilised as the web continues to mature. Two cutting edge and leading examples of web applications are Gmail and Dropbox. As web apps continue to grow in functionality, accessibility, and usefulness, the need to streamline their efficiency also increases. An emerging (and highly useful) concept that's being utilised by the two formerly mentioned apps is a customised context menu. In this tutorial, we're going to cover a few things:

1. Define what a context menu actually is, and understand its purpose and position in web application architecture.
2. Use front-end code to build our very own custom context menu, from styling with CSS, to triggering with JavaScript
3. Top it all off with a little discussion on the practicalities of customised context menus, and some do's and don'ts when it comes to production-level implementations.

Let's dive in!

What is a Context Menu?

I'll borrow and tailor a definition from Wikipedia (https://en.wikipedia.org/wiki/Context_menu), as it covers it nicely:

> A context menu is a menu in a GUI that appears upon user interaction, such as a right-click mouse operation. A context menu offers a limited set of choices that are available in the current state, or context, of the operating system or application. Usually the available choices are actions related to the selected object.

On your computer, right-clicking on the desktop launches your OS's native context menu. From here, you can probably create a new folder, get some information, as well as other actions. When using a web browser, right-clicking on the page would launch that browser's default context menu. You'll be able to get page information, or view the source. Right-clicking on an image presents its own set of options too — you can save an image, open it in a new tab, copy it, among others. These are all default behaviours, but they did get built from the application makers once upon a time. The interesting thing to note here is how the available menu actions change depending on the situation, or context, in which you launch the menu.

Web applications are starting to deploy their own customized context menus to present users with relevant actions. Dropbox and Gmail are perfect examples of this, allowing you to perform actions like archiving, deleting, downloading, viewing, and so on. But how is it done? In a web browser, when a right-click action is performed, an event gets fired. This event is the `contextmenu` event (https://developer.mozilla.org/en-US/docs/Web/Events/contextmenu). To deploy a customized context menu, we'll need to prevent default behaviour, and then set up, trigger, and position our own menu. It's a bit of work, but we're going to do it step by step. Let's start by creating a basic application structure so that we have something real world-ish to play with.

Painting the Picture — a Task List App Example

Imagine that we're creating a good ol' task list application. Yes, it's a common example and you're probably sick of hearing about task list applications, but let's roll with it anyways because it always paints a clear picture. On the landing page of our application, we're likely to have a list of tasks that are yet to be completed. We can click on the task for more information, we can add new tasks, edit the task, and delete the task — a basic CRUD app if you will. With web apps being accessed by a variety of devices lately, we find ourselves implementing many gestures and actions to give them a more natural feeling. For example, swiping left might prompt a user to delete or archive a task.

We're accessing the same web apps on our laptops too though, so we can incorporate similar familiar gestures. Right-clicking and launching a context menu is a natural motion for users, and usually presents them with a list of actions. In Dropbox, those actions range from creating a new directory, to downloading or uploading files. In our app, let's imagine that launching the context menu by right-clicking on a task item allows us to view, edit, or delete this task. We'll use this as a backdrop on which we can build up a menu, and possibly pass in some data to our actions.

At the end of this tutorial, you'll see a working example on CodePen. Until then though, let's work together and build it up ourselves step by step, so you can see the progression. I'll use some modern day CSS for quick structural development, and I'll create a dummy task list with some `data-*` attributes. I'll also pull in Eric Meyer's CSS reset (http://meyerweb.com/eric/tools/css/reset/), and reset the `box-sizing` property to `border-box` on all properties like this:

```
 *,
*::before,
*::after {
  box-sizing: border-box;
}
```

I won't be prefixing any CSS styles, but the CodePen demo will leverage auto-prefixer. Let's get building!

Building our Base Structure

We use cookies to provide you the best possible experience of SitePoint. Want to know more? Read our Terms of Service (/legals) and Privacy Policy. (/privacy-policy). Have questions? Please contact support@sitepoint.com (mailto:support@sitepoint.com), we're happy to help!
Understood

We'll open up our HTML document as normal, throw in a header, a main content area, a dummy task list, and a footer. I'll pull in Font Awesome (http://fortawesome.github.io/Font-Awesome/) for a little extra visual pleasure, and I'll think ahead a little bit. I'll make sure each task contains a `data-id` attribute, which in the real world would be generated from a

database of some sort. Each task will also have an "actions" section, because we want actions to be as accessible as possible. Remember, the context menu provides one unique way to present a user with actions. Here's the important parts of the markup:

```
<body>
  <ul class="tasks">
    <li class="task" data-id="3">
      <div class="task__content">
        Go To Grocery
      </div>
      <div class="task__actions">
        <i class="fa fa-eye"></i>
        <i class="fa fa-edit"></i>
        <i class="fa fa-times"></i>
      </div>
    </li>
    <!-- more task items here... -->
  </ul>
  <script src="main.js"></script>
</body>
```

Remember, you can check out the CodePen demo to follow along with the full document structure. Let's take a look now at the CSS. If you're working along in CodePen, you can easily include auto-prefixer and CSS reset in the settings pane. If not, you'll have to include these manually, or set up your task runners according to your development environment. I also pulled in the Roboto font family and, as mentioned above, I've included Font Awesome. Remember, the focus of this demo is on the creation of the context menus, so the icon action indicators and the add-task button aren't going to work. Here are the relevant parts of the CSS so far:

```
/* tasks */

.tasks {
  list-style: none;
  margin: 0;
  padding: 0;
}

.task {
  display: flex;
  justify-content: space-between;
  padding: 12px 0;
  border-bottom: solid 1px #dfdfdf;
}

.task:last-child {
  border-bottom: none;
}
```

Again, you can grab the full set of styles from the CodePen demo. We're just focusing on the important parts here. Alright, now we have a basic little playground to work within. Let's start considering the actual context menu!

Beginning our Custom Context Menu — the Markup

Our menu will begin like most menus do these days — an unordered list nested inside a `nav` tag. Each action inside our menu will be a list item with a link. Each link will be responsible for a particular action. As I mentioned before, we'll want our context menu to be responsible for three actions:

1. Viewing a task
2. Editing a task
3. Deleting a task

Let's mark up our context menu like this:

```
<nav class="context-menu">
  <ul class="context-menu__items">
    <li class="context-menu__item">
      <a href="#" class="context-menu__link">
        <i class="fa fa-eye"></i> View Task
      </a>
    </li>
    <li class="context-menu__item">
      <a href="#" class="context-menu__link">
        <i class="fa fa-edit"></i> Edit Task
      </a>
    </li>
    <li class="context-menu__item">
      <a href="#" class="context-menu__link">
        <i class="fa fa-times"></i> Delete Task
      </a>
    </li>
  </ul>
</nav>
```

If you're wondering where to put this markup, let's leave it right before the closing `body` tag for now. There's a reason for this, and before we move on to our CSS, let's be aware of a couple things:

1. We ultimately want our context menu to show up wherever we right-click, meaning that it should be absolutely positioned, and out of the flow of the rest of the document. We don't want it to show up inside any relative container that will mess up the placement coordinates, hence the reason I left it at the bottom of the document.
2. We will ultimately want certain variables or attributes to be associated with the context menu. For example, when we click "delete task", we'll want to know which task to delete, and we'll only know this by picking up on which task was right-clicked on in the first place.

Let's continue with the CSS.

Styling our Custom Menu — the CSS

Off the bat, we know we want our menu to be absolutely positioned. Other than that, let's give a little extra style to make it look presentable. I'll also be setting the `z-index` to 10, but remember that in your application, you'll want the menu to be on top of all the content, so set the `z-index` accordingly. In the demo and source code, you'll find a lot of extra styles for some aesthetic pleasure, but the important thing to note here is the positioning and z-indexing of the context menu:

```
.context-menu {
  position: absolute;
  z-index: 10;
}
```

Before we move on to the JavaScript side of things, let's remember that, by default, the context menu should be hidden. We'll set the `display` to `none`, and create a helper "active" class for when we want to show it. And if you're wondering about positioning of the active context menu, we'll tackle that later on too. Here's my additional CSS:

```
.context-menu {
  display: none;
  position: absolute;
  z-index: 10;
}

.context-menu--active {
  display: block;
}
```

Now, it's time to get our context menu ready for action.

Deploying our Context Menu — the JavaScript

Let's start here by actually looking at how to register the `contextmenu` event. In our JavaScript, we'll open up a self-executing function, and catch the event on the entire document. We'll also log the event to the console so we can take a peek at some of the information that gets output:

```
(function() {

  "use strict";

  document.addEventListener( "contextmenu", function(e) {
    console.log(e);
  });

})();
```

If you open up the console and right-click anywhere on the document, you'll see that the event does indeed get logged. There's a lot of interesting information there that will come in handy for us, particularly positional coordinates. Before we prevent default behaviour though, let's take into consideration that on most parts of the document, we don't actually need

custom behaviour. We only want to deploy the custom context menu on our task items. With that in mind, let's build up a little bit according to the following steps:

1. We need to loop over each of our task items, and add the `contextmenu` event listener to them.
2. For each event listener on each task item, we will prevent default behaviour.
3. For now, let's just log the event and the element in question to the console.

Here's what we have so far:

```
(function() {

  "use strict";

  var taskItems = document.querySelectorAll(".task");

  for ( var i = 0, len = taskItems.length; i < len; i++ ) {
    var taskItem = taskItems[i];
    contextMenuListener(taskItem);
  }

  function contextMenuListener(el) {
    el.addEventListener( "contextmenu", function(e) {
      console.log(e, el);
    });
  }

})();
```

If you check out your console, you'll notice that a unique event gets triggered each time we click on a task item element. Let's take it a little further now, and implement the following two steps:

1. We'll prevent the default behaviour when we right-click on a task item.
2. We'll display the custom context menu by adding our helper class to it.

Before we do that though, let's add an ID of `context-menu` to our menu to make it easier to fetch, and let's also create a new variable called `menuState` in our JavaScript. We'll default this to `0`, and assume that `0` means off, and `1` means on. Finally, let's cache a variable for our active class and call it `active`. Here's my three additional variables:

```
var menu = document.querySelector("#context-menu");
var menuState = 0;
var active = "context-menu--active";
```

Now, let's revise our `contextMenuListener` function, and also add a `toggleMenuOn` to launch the menu:

```
function contextMenuListener(el) {
  el.addEventListener( "contextmenu", function(e) {
    e.preventDefault();
    toggleMenuOn();
  });
}

function toggleMenuOn() {
  if ( menuState !== 1 ) {
    menuState = 1;
    menu.classList.add(active);
  }
}
```

At this point, right clicking on a task item should deploy the menu! But a few things are still not working correctly. First, the menu isn't in the position we'd expect when we right-click on an item. This needs to be revised with some math. Second, there's no way to switch the menu off after it appears. If we observe default behaviour, we'll notice a couple things off the bat:

1. Clicking outside the menu causes the menu to go back to its inactive state.
2. Pressing the escape key (or `keycode` 27) also causes the menu to revert to an inactive state.

Let's take this a little further though. Because we have the default menu available on other parts of the app, users would also expect to be able to launch that menu as normal, even if the custom menu is open. So right-clicking outside of our context should close the custom menu, and open up the default one. This doesn't happen just yet. Let's tackle all of these open/close problems first, and then get to position.

Refactoring our Code

It's apparent right now that three main events will be responsible for triggering actions:

1. `contextmenu` – Checks states and deploys context menus.
2. `click` – Hides menu when applicable.
3. `keyup` – Responsible for keyboard actions. We'll only focus on the `ESC` key for this tutorial.

We're also going to need some helper functions, so let's add a section in our JavaScript for that too. Here's the first refactor:

```
(function() {

  "use strict";

  ////////////////////////////////////
  ////////////////////////////////////
  //
  // H E L P E R   F U N C T I O N S
  //
  ////////////////////////////////////
  ////////////////////////////////////

  /**
   * Some helper functions here.
   */

  ////////////////////////////////////
  ////////////////////////////////////
  //
  // C O R E   F U N C T I O N S
  //
  ////////////////////////////////////
  ////////////////////////////////////

  /**
   * Variables.
   */
  var taskItemClassName = 'task';
  var menu = document.querySelector("#context-menu");
  var menuState = 0;
  var activeClassName = "context-menu--active";

  /**
   * Initialise our application's code.
   */
  function init() {
    contextListener();
    clickListener();
    keyupListener();
  }

  /**
   * Listens for contextmenu events.
   */
```

```
  function contextListener() {

  }

  /**
   * Listens for click events.
   */
  function clickListener() {

  }

  /**
   * Listens for keyup events.
   */
  function keyupListener() {

  }

  /**
   * Turns the custom context menu on.
   */
  function toggleMenuOn() {
    if ( menuState !== 1 ) {
      menuState = 1;
      menu.classList.add(activeClassName);
    }
  }

  /**
   * Run the app.
   */
  init();

})();
```

This time, we won't be iterating over our task items. Instead, we'll listen for a document-wide `contextmenu` event, and check if that event occurred inside of our task item. That's the reason we now have the variable `taskItemClassName`. For that, we'll need our first helper function, `clickInsideElement`. It'll take two parameters:

1. The event itself, from which we can check the target or source element of the event.
2. A class name to compare against. If the event occurred inside an element that either has the class name, or on which the parent has the class name, we'll want to return that element.

Here's that first helper function:

```
function clickInsideElement( e, className ) {
  var el = e.srcElement || e.target;

  if ( el.classList.contains(className) ) {
    return el;
  } else {
    while ( el = el.parentNode ) {
      if ( el.classList && el.classList.contains(className) ) {
        return el;
      }
    }
  }

  return false;
}
```

Now we can get back to where we were before by editing our `contextListener` function to this:

```
function contextListener() {
  document.addEventListener( "contextmenu", function(e) {
    if ( clickInsideElement( e, taskItemClassName ) ) {
      e.preventDefault();
      toggleMenuOn();
    }
  });
}
```

With our helper function doing some dirty work for us now, and a `contextmenu` event being listened for on the entire document, we can now turn the menu off when that click occurs outside the context area, and when the custom menu is open. Let's add a `toggleMenuOff` function, and edit our `contextListener` function too:

```
function contextListener() {
  document.addEventListener( "contextmenu", function(e) {
    if ( clickInsideElement( e, taskItemClassName ) ) {
      e.preventDefault();
      toggleMenuOn();
    } else {
      toggleMenuOff();
    }
  });
}

function toggleMenuOff() {
  if ( menuState !== 0 ) {
    menuState = 0;
    menu.classList.remove(activeClassName);
  }
}
```

Go on now and right-click on a task item. Now right-click somewhere else on the document. Voila! The custom menu disappears, and the default one appears as normal. We can do something similar for our regular `click` events now:

```
function clickListener() {
  document.addEventListener( "click", function(e) {
    var button = e.which || e.button;
    if ( button === 1 ) {
      toggleMenuOff();
    }
  });
}
```

The above test is a little different than before, and that's because of a little quirk in Firefox. After the right-click button is released, Firefox triggers a `click` event also, but it's not the left-click code, so we have to make sure the click is specifically a left click. That way, the menu doesn't flash on and off when we right-click an item. Let's now do something similar again for the `keyup` event:

```
  function keyupListener() {
    window.onkeyup = function(e) {
      if ( e.keyCode === 27 ) {
        toggleMenuOff();
      }
    }
  }
```

Now we're successfully showing and hiding the menu as needed, and making the user experience as natural as possible. Let's move on to positioning the menu, and finally we'll look at a possible option for handling events inside the menu.

Positioning our Context Menu

Because of our current HTML and CSS set up, our menu is just appearing at the bottom of the screen. Naturally, we prefer it to appear right next to where we clicked. Let's make that happen. First, let's leverage another helper function that gets the exact coordinates of where we clicked, based on the event. I'll call it `getPosition`, and it handles some cross browser quirks to boost its accuracy:

```
function getPosition(e) {
  var posx = 0;
  var posy = 0;

  if (!e) var e = window.event;

  if (e.pageX || e.pageY) {
    posx = e.pageX;
    posy = e.pageY;
  } else if (e.clientX || e.clientY) {
    posx = e.clientX + document.body.scrollLeft +
                       document.documentElement.scrollLeft;
    posy = e.clientY + document.body.scrollTop +
                       document.documentElement.scrollTop;
  }

  return {
    x: posx,
    y: posy
  }
}
```

Our first step in positioning our menu is to prepare three variables. Let's add these to our variables block:

```
var menuPosition;
var menuPositionX;
var menuPositionY;
```

Now, let's create a function called `positionMenu` that accepts one argument — the event. For now, let's log the position result to the console:

```
function positionMenu(e) {
  menuPosition = getPosition(e);
  console.log(menuPosition);
}
```

We now can edit our `contextListener` function to begin the positioning process:

```
function contextListener() {
  document.addEventListener( "contextmenu", function(e) {
    if ( clickInsideElement( e, taskItemClassName ) ) {
      e.preventDefault();
      toggleMenuOn();
      positionMenu(e);
    } else {
      toggleMenuOff();
    }
  });
}
```

Toggle the context menu again, and check out the console. You'll notice that the position is logged, and available for us. We can use this to inline the `top` and `left` styles to our menu via JavaScript. Here's the updated `positionMenu` function:

```
function positionMenu(e) {
  menuPosition = getPosition(e);
  menuPositionX = menuPosition.x + "px";
  menuPositionY = menuPosition.y + "px";

  menu.style.left = menuPositionX;
  menu.style.top = menuPositionY;
}
```

Click around now and launch the context menu. It shows up wherever we click. That's awesome, but we have a couple things to consider:

1. What happens if the user's screen is a bit narrow, and the user clicks far to the right of the screen? The context menu will appear off screen, and the body will overflow.
2. What if a user resizes the window while the context menu is open? The same overflow issue will occur. Dropbox gets around this by hiding the x-overflow.

Let's tackle the first issue. We'll use JavaScript to determine the width and height of our menu, and check to make sure that the position of the menu won't overflow. If it does, we'll offset it by a few pixels, in a similar way that the default menu works. It's a bit of math, and requires some thought, but we can get through it step by step. First, we'll need to check the window width and height. Then, we'll need to find out the menu's width and height. Finally, we'll need to check if the difference of the clicked position and the window width plus offset is greater than the menu width, and the same for the height. If it is greater, we manually set the position, and if not, we proceed as normal. We'll start by caching two variables for the width and height:

```
var menuWidth;
var menuHeight;
```

Remember from earlier, our menu is hidden by default, so we won't be able to calculate the width and height yet. In any case, we're statically generating a menu, and in a real-world app like Dropbox, the contents of the menu may change depending on the context. It's a good idea then to calculate the width and height when the menu is deployed. Inside our `positionMenu` function, we can get the width and height like this:

```
menuWidth = menu.offsetWidth;
menuHeight = menu.offsetHeight;
```

Now let's prepare two more variables for the window's inner width and height:

```
var windowWidth;
var windowHeight;
```

Similarly, we can calculate them when we deploy our menu like this:

```
windowWidth = window.innerWidth;
windowHeight = window.innerHeight;
```

Finally, let's assume that we want it to only ever be as close as 4px to the edge of the window. We can compare our values like I mentioned before, and correctly position our menu. Here's what I came up with:

```
var clickCoords;
var clickCoordsX;
var clickCoordsY;

// updated positionMenu function
function positionMenu(e) {
  clickCoords = getPosition(e);
  clickCoordsX = clickCoords.x;
  clickCoordsY = clickCoords.y;

  menuWidth = menu.offsetWidth + 4;
  menuHeight = menu.offsetHeight + 4;

  windowWidth = window.innerWidth;
  windowHeight = window.innerHeight;

  if ( (windowWidth - clickCoordsX) < menuWidth ) {
    menu.style.left = windowWidth - menuWidth + "px";
  } else {
    menu.style.left = clickCoordsX + "px";
  }

  if ( (windowHeight - clickCoordsY) < menuHeight ) {
    menu.style.top = windowHeight - menuHeight + "px";
  } else {
    menu.style.top = clickCoordsY + "px";
  }
}
```

Our menu should now be behaving as expected, even if we trigger it on narrow window sizes! As mentioned, on window resize, there's still the possibility that it spills over. I mentioned before how Dropbox gets around this, but for us, let's add a final event listener to turn the menu off on window resize. In our `init` function, we can add this:

```
resizeListener();
```

And the function will look like this:

```
  function resizeListener() {
    window.onresize = function(e) {
      toggleMenuOff();
    };
  }
```

Awesome.

Attaching Events to the Context Menu Items

If your app is more complicated than this, and you have the need to generate the context menu content dynamically, then you'll have to take that into consideration. For our app, we have one menu with the same options. We can thus do a quick check to see what item has been clicked, and trigger some kind of action. Using Dropbox again as an example, if you right-click an item and press delete, a confirmation modal gets displayed. For our app, let's just store the current task item to a variable, and log to the console the `data-id` attribute that we set up in the beginning. We'll also log the corresponding action, so let's edit our context menu markup to include some data attributes:

```
<nav id="context-menu" class="context-menu">
  <ul class="context-menu__items">
    <li class="context-menu__item">
      <a href="#" class="context-menu__link" data-action="View">
        <i class="fa fa-eye"></i> View Task
      </a>
    </li>
    <li class="context-menu__item">
      <a href="#" class="context-menu__link" data-action="Edit">
        <i class="fa fa-edit"></i> Edit Task
      </a>
    </li>
    <li class="context-menu__item">
      <a href="#" class="context-menu__link" data-action="Delete">
        <i class="fa fa-times"></i> Delete Task
      </a>
    </li>
  </ul>
</nav>
```

We also have quite a few functions and objects to change and cache, so let's go through them all step by step. First of all, let's cache all the objects we need:

```
var contextMenuClassName = "context-menu";
var contextMenuItemClassName = "context-menu__item";
var contextMenuLinkClassName = "context-menu__link";
var contextMenuActive = "context-menu--active";

var taskItemClassName = "task";
var taskItemInContext;

var clickCoords;
var clickCoordsX;
var clickCoordsY;

var menu = document.querySelector("#context-menu");
var menuItems = menu.querySelectorAll(".context-menu__item");
var menuState = 0;
var menuWidth;
var menuHeight;
var menuPosition;
var menuPositionX;
var menuPositionY;

var windowWidth;
var windowHeight;
```

Some new ones to note are `taskItemInContext`, which will get assigned if we successfully right-click on a task item. We'll need this for logging the task item ID. Let's also take note of the various class names that got cached, which will make for easy editing of our functionality should we change our markup. Let's now move on to functionality.

Our initialiser function remains the same, but our first change lies in the `contextListener` function. Remember, we want to store the `taskItemInContext` variable if we right click on a task item. Our `clickInsideElement` helper actually returns an element if we have a match, so here's where we can set that up:

```
function contextListener() {
  document.addEventListener( "contextmenu", function(e) {
    taskItemInContext = clickInsideElement( e, taskItemClassName );

    if ( taskItemInContext ) {
      e.preventDefault();
      toggleMenuOn();
      positionMenu(e);
    } else {
      taskItemInContext = null;
      toggleMenuOff();
    }
  });
}
```

We reset it to `null` if our right-click is not on a task item. Let's move on to the `clickListener` function. Like I mentioned before, we just want to log some information to the console for simplicity purposes. Currently, when a click event is registered, we run a couple of checks and close off the menu. However, let's do a little chopping up of this function, and check if that click was an item in our context menu. If it was, then we'll perform our action, and close the menu after:

```
function clickListener() {
  document.addEventListener( "click", function(e) {
    var clickeElIsLink = clickInsideElement( e, contextMenuLinkClassName );

    if ( clickeElIsLink ) {
      e.preventDefault();
      menuItemListener( clickeElIsLink );
    } else {
      var button = e.which || e.button;
      if ( button === 1 ) {
        toggleMenuOff();
      }
    }
  });
}
```

You'll notice that a `menuItemListener` function gets called when we click a context menu item, so we'll assess that in a minute. The `keyupListener` and `resizeListener` functions remain unchanged. The same mostly goes for the `toggleMenuOn` and `toggleMenuOff` functions, the only difference being the change in variable name for the active class, providing better code readability:

```
function toggleMenuOn() {
  if ( menuState !== 1 ) {
    menuState = 1;
    menu.classList.add( contextMenuActive );
  }
}

function toggleMenuOff() {
  if ( menuState !== 0 ) {
    menuState = 0;
    menu.classList.remove( contextMenuActive );
  }
}
```

The `positionMenu` function remains unchanged too, and finally, here's the new `menuItemListener` function that accepts one argument:

```
function menuItemListener( link ) {
  console.log( "Task ID - " +
               taskItemInContext.getAttribute("data-id") +
               ", Task action - " + link.getAttribute("data-action"));
  toggleMenuOff();
}
```

That brings us to the end of our functionality…phew!

Some Notes and Considerations

Before wrapping up, let's take into account a couple things:

1. Throughout the entire article, I mentioned "right-click" as the event via which a context menu gets launched. Not everyone is a righty, and not everyone has the same mouse setup. Regardless of that, the `contextmenu` event acts in accordance with a user's mouse setup.
2. Another important point to note is that we've only considered full-fledged desktop applications. Users might be accessing your app or website via keyboard input or mobile tech, among others. You need to make sure that if you decide to alter default behaviour, you make it a consistently user-friendly experience across all types of accessibility.

With those things in mind, you should be well on your way to taking this component a step further.

The Big Question

I saved this final consideration for its own paragraph, because most importantly and above everything we've covered in this tutorial, there is the big question: Do you really need a custom context menu? Customising behaviours is cool and we've done some really interesting work here today, but you really have to make sure that your app will only absolutely benefit from something like this. Users expect certain things when they access the web. The perfect example would be if your app has a photo, and over that photo is your custom context menu. You should definitely keep in mind that users right-clicking on it would expect to be able to download it, copy the link, open the image in a new tab, and other standard options.

Browser Compatibility

This tutorial made use of some modern CSS and JavaScript, namely `flex` for layouts in CSS, and `classList` for class toggling in JavaScript. If you need backward compatibility with older browsers, I suggest you edit accordingly. This tutorial was tested in the following browsers:

> Chrome 39
> Safari 7
> Firefox 33

It's also worth mentioning here that there is a spec for a context menu/toolbar (https://html.spec.whatwg.org/multipage/forms.html#context-menus) in HTML5, but unfortunately browser support is just about none (http://caniuse.com/#feat=menu) so it's not something worth going into in much depth. Custom solutions are the only real option for the foreseeable future.

Wrap Up and Demo

If you've carefully considered everything and are 100% certain that your app could use this type of functionality, then go for it. We've covered a lot in this tutorial today, and you should now have a solid understanding of what a context menu is, how to implement it in your app, and some important considerations to factor in. I hope you enjoyed reading, and I'm looking forward to your comments!

For your reference, the codebase for this tutorial can be found on GitHub (https://github.com/callmenick/Custom-Context-Menu), and will be maintained and likely updated over time. For the iteration of code used in this tutorial and for a full working demo, check out the CodePen demo below.

Meet the author

Nick Salloum (https://www.sitepoint.com/author/nsalloum/) (https://twitter.com/nicksalloum_) (https://plus.google.com/+NickSalloum/) (https://www.facebook.com/callmenick1)

I'm a web designer & developer from Trinidad & Tobago, with a degree in Mechanical Engineering. I love the logical side of the web, and I'm an artist/painter at heart. I endorse progressive web techniques, and try to learn something every day. I try to impart my knowledge as much as possible on my personal blog, callmenick.com (http://www.callmenick.com/). I love food, I surf every weekend, and I have an amazing creative partnership with fellow mischief maker Elena (http://www.elenamolchanova.com/). Together, we run SAYSM (http://saysm.com/).

Start A Discussion (Http://Community.Sitepoint.Com/T/Building-A-Custom-Right-Click-Context-Menu-With-Javascript/116624)

- Versioning (/versioning/)
- Forums (/community/)
- References (/html-css/css/)

Legals

- Terms of Use (/legals/)
- Privacy Policy (/privacy-policy/)

- us/)
- Press Room (/press/)

- FAQ (https://sitepoint.zendesk.com/hc/en-us)
- Write for Us (/write-for-us/)
- Advertise (/advertise/)

Connect

(https://www.facebook.com/sitepoint) (http://twitter.com/sitepointdotcom) (/versioning/)
(https://www.sitepoint.com/feed/) (https://plus.google.com/+sitepoint)

© 2000 – 2018 SitePoint Pty. Ltd.