# Installation ## NPM/Yarn NPM combined with a tool like [Browserify](http://browserify.org) or [Webpack](https://webpack.js.org) is the recommended method of installing SweetAlert. ```bash npm install sweetalert --save ``` Then, simply import it into your application: ```javascript import swal from 'sweetalert'; ``` ## CDN You can also find SweetAlert on [unpkg](https://unpkg.com/sweetalert) and [jsDelivr](https://cdn.jsdelivr.net/npm/sweetalert) and use the global `swal` variable. ```html ``` # Getting started ## Showing an alert After importing the files into your application, you can call the `swal` function (make sure it's called \*after\* the DOM has loaded!) ```js swal("Hello world!"); ``` If you pass two arguments, the first one will be the modal's title, and the second one its text. ```js swal("Here's the title!", "...and here's the text!"); ``` And with a third argument, you can add an icon to your alert! There are 4 predefined ones: `"warning"`, `"error"`, `"success"` and `"info"`. ```js swal("Good job!", "You clicked the button!", "success"); ``` ## Using options The last example can also be rewritten using an object as the only parameter: ```js swal({ title: "Good job!", text: "You clicked the button!", icon: "success", }); ``` With this format, we can specify many more options to customize our alert. For example we can change the text on the confirm button to `"Aww yiss!"`: ```js swal({ title: "Good job!", text: "You clicked the button!", icon: "success", button: "Aww yiss!", }); ``` You can even combine the first syntax with the second one, which might save you some typing: ```js swal("Good job!", "You clicked the button!", "success", { button: "Aww yiss!", }); ``` For a full list of all the available options, check out the [API docs](/docs)! ## Using promises SweetAlert uses [promises](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global\_Objects/Promise) to keep track of how the user interacts with the alert. If the user clicks the confirm button, the promise resolves to `true`. If the alert is dismissed (by clicking outside of it), the promise resolves to `null`. ```js swal("Click on either the button or outside the modal.") .then((value) => { swal(`The returned value is: ${value}`); }); ``` This comes in handy if you want to warn the user before they perform a dangerous action. We can make our alert even better by setting some more options: - `icon` can be set to the predefined `"warning"` to show a nice warning icon. - By setting `buttons` (plural) to `true`, SweetAlert will show a cancel button in addition to the default confirm button. - By setting `dangerMode` to `true`, the focus will automatically be set on the cancel button instead of the confirm button, and the confirm button will be red instead of blue to emphasize the dangerous action. ```js swal({ title: "Are you sure?", text: "Once deleted, you will not be able to recover this imaginary file!", icon: "warning", buttons: true, dangerMode: true, }) .then((willDelete) => { if (willDelete) { swal("Poof! Your imaginary file has been deleted!", { icon: "success", }); } else { swal("Your imaginary file is safe!"); } }); ``` # Advanced examples ## Customizing buttons We've already seen how we can set the text on the confirm button using `button: "Aww yiss!"`. If we also want to show and customize the \*cancel button\*, we can instead set `buttons` to an \*array of strings\*, where the first value is the cancel button's text and the second one is the confirm button's text: ```js swal("Are you sure you want to do this?", { buttons: ["Oh noez!", "Aww yiss!"], }); ``` If you want one of the buttons to just have their default text, you can set the value to `true` instead of a string: ```js swal("Are you sure you want to do this?", { buttons: ["Oh noez!", true], }); ``` So what if you need \*more\* than just a cancel and a confirm button? Don't worry, SweetAlert's got you covered! By specifying an object for `buttons`, you can set exactly as many buttons as you like, and specify the value that they resolve to when they're clicked! In the example below, we set 3 buttons: - `cancel`, which by default resolves to `null` and has a custom `"Run away!"` text. - `catch`, which will resolve to the `value` we've specified (`"catch"`) and has the custom text `"Throw Pokéball!"`. - `defeat`. Here, we specify `true` to let SweetAlert set some default configurations for the button. In this case, it will set the `text` to `"Defeat"` (capitalized) and the resolved value to `defeat`. Had we set the `cancel` button to `true`, it would still resolve to `null` as expected. ```js swal("A wild Pikachu appeared! What do you want to do?", { buttons: { cancel: "Run away!", catch: { text: "Throw Pokéball!", value: "catch", }, defeat: true, }, }) .then((value) => { switch (value) { case "defeat": swal("Pikachu fainted! You gained 500 XP!"); break; case "catch": swal("Gotcha!", "Pikachu was caught!", "success"); break; default: swal("Got away safely!"); } }); ``` You can check out all the available button options in the [docs](/docs#buttons). ## AJAX requests Since SweetAlert is promise-based, it makes sense to pair it with AJAX functions that are also promise-based. Below is an example of using `fetch` to search for artists on the iTunes API. Note that we're using `content: "input"` in order to both show an input-field \*and\* retrieve its value when the user clicks the confirm button: ```js swal({ text: 'Search for a movie. e.g. "La La Land".', content: "input", button: { text: "Search!", closeModal: false, }, }) .then(name => { if (!name) throw null; return fetch(`https://itunes.apple.com/search?term=${name}&entity=movie`); }) .then(results => { return results.json(); }) .then(json => { const movie = json.results[0]; if (!movie) { return swal("No movie was found!"); } const name = movie.trackName; const imageURL = movie.artworkUrl100; swal({ title: "Top result:", text: name, icon: imageURL, }); }) .catch(err => { if (err) { swal("Oh noes!", "The AJAX request failed!", "error"); } else { swal.stopLoading(); swal.close(); } }); ``` ## Using DOM nodes as content Sometimes, you might run into a scenario where it would be nice to use the out-of-the box functionality that SweetAlert offers, but with some custom UI that goes beyond just styling buttons and text. For that, there's the `content` option. In the previous example, we saw how we could set `content` to `"input"` to get an `` element in our modal that changes the resolved value of the confirm button based on its value. `"input"` is a predefined option that exists for convenience, but you can also set `content` to any DOM node! Let's see how we can recreate the functionality of the following modal... ```js swal("Write something here:", { content: "input", }) .then((value) => { swal(`You typed: ${value}`); }); ``` ...using a custom DOM node! We're going to use [React](https://facebook.github.io/react) here, since it's a well-known UI library that can help us understand how to create more complex SweetAlert interfaces, but you can use any library you want, as long as you can extract a DOM node from it! ```js import React, { Component } from 'react'; import ReactDOM from 'react-dom'; const DEFAULT\_INPUT\_TEXT = ""; class MyInput extends Component { constructor(props) { super(props); this.state = { text: DEFAULT\_INPUT\_TEXT, }; } changeText(e) { let text = e.target.value; this.setState({ text, }); /\* \* This will update the value that the confirm \* button resolves to: \*/ swal.setActionValue(text); } render() { return ( ) } } // We want to retrieve MyInput as a pure DOM node: let wrapper = document.createElement('div'); ReactDOM.render(, wrapper); let el = wrapper.firstChild; swal({ text: "Write something here:", content: el, buttons: { confirm: { /\* \* We need to initialize the value of the button to \* an empty string instead of "true": \*/ value: DEFAULT\_INPUT\_TEXT, }, }, }) .then((value) => { swal(`You typed: ${value}`); }); ``` This might look very complex at first, but it's actually pretty simple. All we're doing is creating an input tag as a React component. We then extract its DOM node and pass it into under the `swal` function's `content` option to render it as an unstyled element. The only code that's specific to SweetAlert is the `swal.setActionValue()` and the `swal()` call at the end. The rest is just basic React and JavaScript.  Using this technique, we can create modals with more interactive UIs, such as this one from Facebook. # Using with libraries While the method documented above for creating more advanced modal designs works, it gets quite tedious to manually create nested DOM nodes. That's why we've also made it easy to integrate your favourite template library into SweetAlert, using the [SweetAlert Transformer](https://github.com/sweetalert/transformer). ## Using React In order to use SweetAlert with JSX syntax, you need to install [SweetAlert with React](https://www.npmjs.com/package/@sweetalert/with-react). Note that you need to have both `sweetalert` and `@sweetalert/with-react` as dependencies in your `package.json`. After that, it's easy. Whenever you want to use JSX in your SweetAlert modal, simply import swal from `@sweetalert/with-react` instead of from `sweetalert`. ```js import React from 'react' import swal from '@sweetalert/with-react' swal(

# Hello world!

This is now rendered with JSX!

) ``` The JSX syntax replaces the modal's `content` option, so you can still use all of SweetAlert's other features. Here's how you could implement that Facebook modal that we saw earlier: ```js import React from 'react' import swal from '@sweetalert/with-react' const onPick = value => { swal("Thanks for your rating!", `You rated us ${value}/3`, "success") } const MoodButton = ({ rating, onClick }) => ( onClick(rating)} /> ) swal({ text: "How was your experience getting help with this issue?", buttons: { cancel: "Close", }, content: (

) }) ``` # Upgrading from 1.X SweetAlert 2.0 introduces some important breaking changes in order to make the library easier to use and more flexible. The most important change is that callback functions have been deprecated in favour of [promises](#using-promises), and that you no longer have to import any external CSS file (since the styles are now bundled in the .js-file). Below are some additional deprecated options along with their replacements: - When using a single string parameter (e.g. `swal("Hello world!")`), that parameter will be the modal's `text` instead of its `title`. - `type` and `imageUrl` have been replaced with a single `icon` option. If you're using the shorthand version (`swal("Hi", "Hello world", "warning")`) you don't have to change anything. - `customClass` is now `className`. - `imageSize` is no longer used. Instead, you should specify dimension restrictions in CSS if necessary. If you have a special use case, you can give your modal a custom class. - `showCancelButton` and `showConfirmButton` are no longer needed. Instead, you can set `buttons: true` to show both buttons, or `buttons: false` to hide all buttons. By default, only the confirm button is shown. - `confirmButtonText` and `cancelButtonText` are no longer needed. Instead, you can set `button: "foo"` to set the text on the confirm button to "foo", or `buttons: ["foo", "bar"]` to set the text on the cancel button to "foo" and the text on the confirm button to "bar". - `confirmButtonColor` is no longer used. Instead, you should specify all stylistic changes through CSS. As a useful shorthand, you can set `dangerMode: true` to make the confirm button red. Otherwise, you can specify a class in the [button object](/docs#buttons). - `closeOnConfirm` and `closeOnCancel` are no longer used. Instead, you can set the `closeModal` parameter in the [button options](/docs#buttons). - `showLoaderOnConfirm` is no longer necessary. Your button will automatically show a loding animation when its `closeModal` parameter is set to `false`. - `animation` has been deprecated. All stylistic changes can instead be applied through CSS and a custom modal class. - `type: "input"`, `inputType`, `inputValue` and `inputPlaceholder` have all been replaced with the `content` option. You can either specify `content: "input"` to get the default options, or you can customize it further using the [content object](/docs#content). - `html` is no longer used. Instead use the [content object](/docs#content). - `allowEscapeKey` is now `closeOnEsc` for clarity. - `allowClickOutside` is now `closeOnClickOutside` for clarity.