

Converting JavaScript Buttons for **Lightning Experience**



TABLE OF CONTENTS

Executive Summary	1
Discovery	2
Simple Solutions	2
Declarative Solutions	3
Advanced Solutions	6
Example: Warn the User	6
• Step 1: Apex Class	6
• Step 2: Lightning Component	7
• Step 3: Lightning Component Controller	9
• Step 4: Create the Button	11
• Step 5: Add to Page Layout	12
• Step 6: Test and Deploy	12
Example: Mass Delete from a List View	13
• Step 1: Apex Class / Controller Extension	13
• Step 2: Visualforce Page	14
• Step 3: Create the Button	15
• Step 4: Add to Layout	16
• Step 5: Test and Deploy	16
Other Scenarios for Advanced Solutions	17
Implementation Tips	17
Additional Resources	18

Most new features are being released exclusively for Lightning Experience and are not available in Salesforce Classic. With each new release, incentives to switch to Lightning Experience are piling up.



EXECUTIVE SUMMARY

Lightning Experience, Salesforce.com's redesigned user interface for their CRM applications and platform, does not support JavaScript buttons. In the old interface (now called Salesforce Classic), JavaScript buttons are very flexible and can be developed directly in a production environment. For many existing Salesforce customers, it was quick and easy to develop JavaScript buttons to assist in a variety of business processes, but now a heavy reliance on JavaScript buttons can be an impediment to adopting Lightning Experience.

There are good reasons to adapt the features from JavaScript buttons into Lightning-friendly alternatives. Salesforce did not incorporate JavaScript buttons into Lightning Experience for security reasons. JavaScript's flexibility and ability to access the Salesforce database and browser features, which make it a powerful developer tool, also expose vulnerabilities for hackers to capture confidential data or inject malicious code via cross site scripting attacks.

Lightning Experience was more limited than Salesforce Classic in many ways back in the first release (Winter '16), but after multiple releases, Lightning Experience's functionality has caught up with Salesforce Classic. Almost all new features are being released exclusively for Lightning Experience and are not available in Salesforce Classic. With each new release, incentives to switch to Lightning Experience are piling up.

If you are still using Salesforce Classic and JavaScript buttons, it will require some development energy to create alternative solutions that will work in Lightning, but it is possible to create a Lightning-friendly replacement in virtually any given scenario. If your users are clamoring to switch to Lightning Experience or if you think they could benefit from some of the new features, JavaScript buttons need not hold you back.

This white paper provides some strategies for replacing your JavaScript buttons with Lightning-friendly alternatives and walks through some simple and advanced example solutions.



Discovery

The first step in providing alternatives for your JavaScript buttons is to inventory all of the JavaScript buttons in your Salesforce org. You can go about this in two ways.

First, you can run the Lightning Experience Readiness evaluation. You can launch the evaluation from the Lightning Experience section in the Force.com Setup menu in your Salesforce org. An automated process will check all of the features you are using in your org that are not supported or may not operate as expected in Lightning Experience and you will receive a complete report via email once the process has finished checking everything in your org. The report should highlight every JavaScript button you have in your org under the “Custom Buttons and Links – JavaScript” section and will also provide details about the page layouts, number of profiles, and number of users for each button and a basic recommendation about how to replace the button.

Second, you can take a manual inventory of all of your JavaScript buttons by taking a look at the Buttons, Links, and Actions section of each standard and custom object in use in your Salesforce org and identify any that use OnClick JavaScript as the Content Source.

If you make a manual inventory of your JavaScript buttons, you should still take into consideration which page layouts use each JavaScript button and whether those page layouts are assigned to profiles for users who would be using Lightning Experience. When you are looking at the Detail page for a JavaScript button, you can click on the “Where is this used?” button to see which page layouts include it.

When planning alternatives to your JavaScript buttons, it’s also helpful to find out from your users whether or not they use each button, how often they use each, and how critical each button is to their work. This information can help you prioritize your JavaScript alternative release strategy.

Simple Solutions

Look for the simplest solutions first. If a JavaScript button isn’t included in any page layouts or the List View layout for the relevant object, you don’t need to replace it with an alternative.

If the button is used in a layout, but you’ve found out from your users that they never use it, you can move to Lightning Experience without providing an alternative for the button. If it does happen that some of your users need it, they can always switch back to Classic while you develop an alternative.

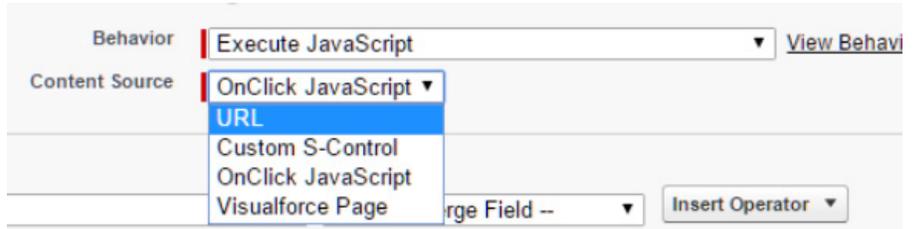
Some JavaScript buttons were developed many Salesforce releases ago. If your button is a few years old, there’s a chance that Salesforce has rolled out a new feature since then that can replace the function of your JavaScript button. Search Salesforce’s success site for potential out-of-the-box alternatives to your JavaScript button. It’s not very likely that you’ll find a new Salesforce feature to replace your button, but it’s worth a look because of the ease of implementation.

Here’s an example of a JavaScript button that could be replaced by a standard Salesforce feature. Say you have a JavaScript button on your Opportunity record that will open up a popup window with a search on a news aggregator for relevant stories about the Account to which the Opportunity belongs. In Lightning Experience, you can enable the News feature which will show news stories related to the associated Account record and Industry right on the Opportunity page.

Declarative Solutions

If you need to replace a JavaScript button that's important to your users, look into whether a declarative solution will work before evaluating a programmatic solution. Declarative solutions that can replace a JavaScript button including Custom URL Buttons, VisualForce Page Buttons, and Quick Actions.

Custom URL buttons are suitable substitutes for JavaScript buttons that direct to a page on an external website or that navigate to a different record in Salesforce. For internal Salesforce links, use the URLFOR function and \$Action functions or Relative Salesforce URLs (e.g. /{!Contact.Id}) rather than hard-coded URLs (e.g. https://na11.salesforce.com/{!Contact.ID}) to keep your users from getting directed out of Lightning Experience. You can update a JavaScript button to a Custom URL button by updating the Content Source for your button from OnClick JavaScript to URL, but make sure to test in a Sandbox first before you do this in your Production environment.



If your JavaScript button directs to a VisualForce page, you may be able to make a very easy update to your button to make it compatible with Lightning Experience. If your VisualForce page uses the Standard Controller for the object where the button resides, you can update your existing button to use the Visualforce Page as the Content Source, rather than OnClick JavaScript, and select the correct Visualforce Page from the dropdown list of available pages. Again, when making updates to an existing button, always test in a Sandbox first before making the change directly in your Production environment.

If your JavaScript button directs to a VisualForce page that uses a Custom Controller, you may want to look into making that page available in a Tab and re-training your users to access it through the Tab in Lightning Experience.

A very common use for JavaScript buttons is to populate a new record with values based on the values in the record where a user clicks the button. Let's say, for example, that you have a custom object in your Salesforce org named "Project" and in your business process, when you win an Opportunity, a new associated Project record needs to be created. The new Project record needs to be linked back to the Opportunity record and will almost always share information with the Opportunity such as the Account, Primary Campaign Source, and Type. To make the new Project creation process easier for your team, you've built a JavaScript button on Opportunities to pull up a new Project record that automatically links back to the won Opportunity and also pre-populates fields on the new Project with information from the Opportunity like the Account, Primary Campaign Source, and Type.

Opportunity Actions New Action

A screenshot of the 'Enter Action Information' form in Salesforce. The form is titled 'Opportunity Actions' and 'New Action'. It contains the following fields: 'Object Name' (Opportunity), 'Action Type' (Create a Record), 'Target Object' (Project), 'Standard Label Type' (--None--), 'Label' (New Project), 'Name' (New_Project), 'Description' (empty), 'Create Feed Item' (checkbox), 'Success Message' (empty), and 'Icon' (Change Icon). There are 'Save' and 'Cancel' buttons at the top and bottom of the form.

This is a great use case for a Quick Action.

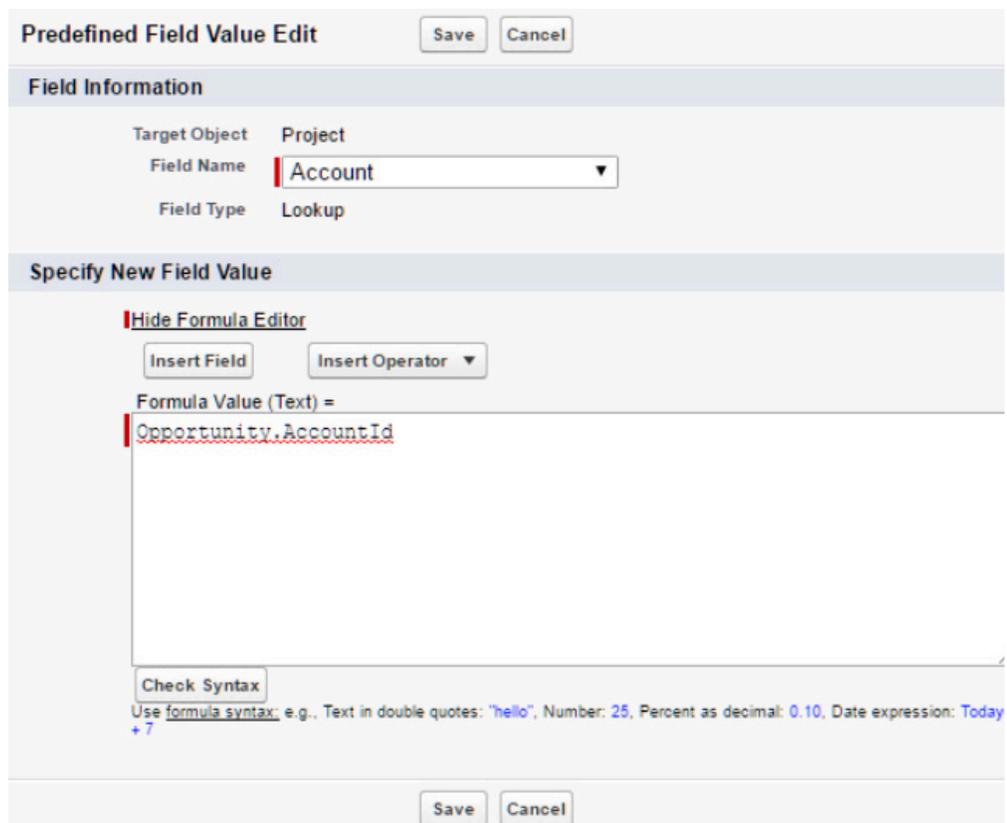
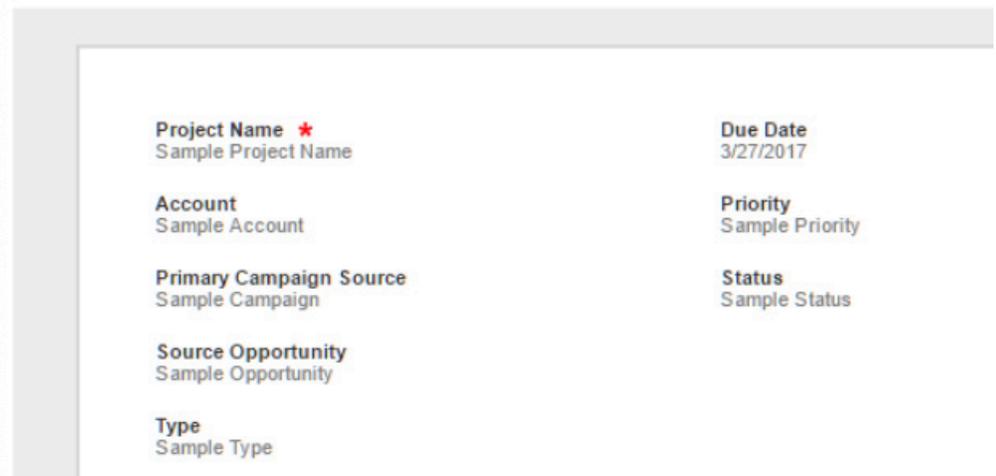
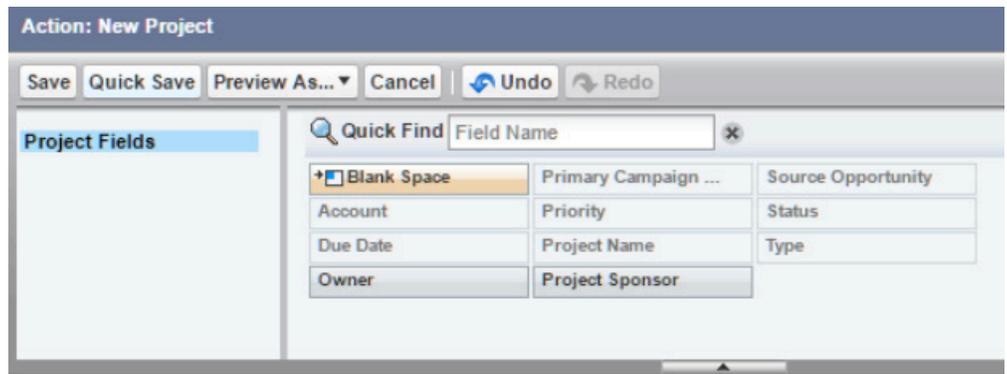
To create a Quick Action for this scenario, create a new Quick Action for Opportunities. Choose the "Create a Record" Action Type and choose "Project" as the Target Object.



Next you'll want to add, to the page layout for the action, any fields that the user should fill out when creating a new Project record, as well as any fields that should be pre-populated from information on the Opportunity.

After you have set up the page layout, you can set up Predefined Field Values.

For example, here is how a Predefined Field Value to pre-populate the Account field with the same Account as the Opportunity would look:



New Project

* Project Name <input type="text"/>	Due Date <input type="text" value=""/>
Account <div style="border: 1px solid #ccc; padding: 2px; display: flex; align-items: center;"> United Oil & Gas Corp. ✕</div>	Priority <input type="text" value="--None--"/>
Primary Campaign Source <input type="text" value="Search Campaigns"/>	Status <input type="text" value="--None--"/>
Source Opportunity United Oil Emergency Generators	
Type <input type="text" value="--None--"/>	

Now your new Project record will be automatically populated with the correct Account record when a user clicks on the New Project button on Opportunities (you'll need to make the Quick Action available in the page layout to make the button available).

If you need more advanced pre-population, for example, if you need to pre-populate the Primary Contact from the Opportunity into the Project Sponsor field on a new Project, you may need to look into a more advanced programmatic solution.

Advanced Solutions

In many cases, a simple declarative solution will not suffice as a replacement for a JavaScript button. Since JavaScript buttons are powerful and relatively easy to develop, many of them execute more complex functions than a simple quick action or URL can handle. Replacing the functionality of some JavaScript buttons for compatibility with Lightning Experience requires a programmatic solution.

Some examples of JavaScript buttons that will require replacement with a programmatic solution in Lightning Experience are buttons that provide popups or feedback to the user based on inputs or data state, buttons that operate on multiple records in a list view, or buttons that make a callout to a third-party service.

In one of these more complex scenarios, there are two types of programmatic solutions that you can use to replace a JavaScript button: Lightning Actions, or Visualforce. Salesforce recommends using Lightning Actions where possible, but you'll need to use Visualforce to replace a JavaScript button that performs a mass action on records from a List View.

A Lightning Action is simply a Quick Action that calls a Lightning Component, so one advantage of using a Lightning Action rather than a Visualforce page is that once the work is complete, the Lightning Component can be re-used in other Lightning pages and components. On the other hand, if you are looking to do a quick switch over to Lightning and you have a lot of classic Visualforce development experience on your team, calling a Visualforce page from a Quick Action is a viable option, especially now that the Lightning Design System is available to provide users a seamless transition between Visualforce pages and the rest of Lightning Experience.

Example: Warn the User

Here is a scenario where a Lightning Action can replace your JavaScript button in a way that a Quick Action can't. You have a powerful tool, a JavaScript button that opts out all Contacts at an Account from receiving email. Users need to be able to use this button to manage email marketing subscriptions properly and keep up a good reputation for the organization, but at the same time, this isn't a button users should be able to click and execute on accident. For this reason, the JavaScript includes a warning so the user must click through before the update is executed.

You can replace this JavaScript button with a Lightning Action, i.e. a Quick Action that calls a Lightning Component. We'll walk through the steps of how you would construct this replacement button. Before you can create the button, you'll need to create a Lightning Component with a client-side controller and a server-side controller to support it.

Step 1: Apex Class

You will need an Apex class to handle the server-side logic of the opt out operation, that is, actually updating the opt out field on the Contacts in the database. The Apex Class is the server-side controller for your Lightning Component.

The example class below contains two methods that can be used by a Lightning Component since they are Aura Enabled. The first method (getAccName) will allow us to display the Account Name in our Lightning Component (only the Account ID is available by default). The second method (optOutContacts) takes the Account ID as a parameter, queries the Contacts related to the Account based on the ID, and updates the HasOptedOutOfEmail field on all of those Contact records.



```
public class optOutController {

    //Method to retrieve the Account Name.
    @AuraEnabled
    public static String getAccName(ID accid){
        return [SELECT ID, Name FROM Account WHERE ID = :accid LIMIT 1][0].Name;
    }

    //Method to opt out the Contacts at the Account with the specified ID parameter.
    @AuraEnabled
    public static void optOutContacts(ID accid){
        List<Contact> contactsToOptOut = new List<Contact>(
            [SELECT ID, AccountId, HasOptedOutOfEmail
             FROM Contact
             WHERE AccountId = :accid]);
        for(Contact c : contactsToOptOut){
            c.HasOptedOutOfEmail = TRUE;
        }
        update contactsToOptOut;
    }
}
```

Step 2: Lightning Component

After the Apex Class has been created, we can refer to it in a Lightning Component. Using the Developer Console, create a new Lightning Component. The Component in the example below will display a helpful warning prompt to the user along with a warning icon, allow the user to activate the Email Opt Out method via a button, and, at the bottom, will display a message to the user describing the status of whether the Email Opt Out method executed successfully.

When creating a Lightning Component that needs to call server-side logic, you need to reference your Apex class as the controller in the beginning aura:component tag. To make a Component usable in a Lightning Action, add force:lightningQuickAction, and to make it retrieve the record ID from the record it's called from, add force:hasRecordId.

The styling in this example is quite simple, but you can add some fancier styling that matches the rest of Lightning Experience by tapping into the Lightning Design System stylesheet that is automatically available to your Lightning Component (<https://www.lightningdesignsystem.com/>).



```
<aura:component controller="optOutController"
implements="force:lightningQuickAction,force:hasRecordId">
  <aura:attribute name="recordId" type="Id" /><!-- Makes the record ID
available for use in the component. -->
  <aura:attribute name="accname" type="String" /><!--Attribute to hold the
Account Name. Will be populated by the doInit action in the handler. -->
  <aura:attribute name="message" type="String" default=" " /><!-- Attribute to
hold the success/error message after attempting to opt out the contacts. -->
  <aura:handler name="init" value="{!this}" action="{!c.doInit}" /><!-- Will call
the doInit method when the Component loads to populate the Account Name (accname)
attribute -->

  <div>
    <lightning:icon iconName="utility:warning" variant="warning" /><!-- Warning
icon from the Lightning Design System to help the user recognize this as a warning
prompt. -->
  </div>
  <br/>
  <p> <!-- Helpful text for the user. The Account Name will be populated
from the controller. -->
    Are you sure you want to opt out all Contacts at <ui:outputText
value="{!v.accname}"/> from receiving emails?
    This action cannot be undone.
  </p>
  <br/>
  <div>
    <!-- This is the button that will allow the user to Opt Out the Contacts at
the Account. It calls a function in the Component Controller which in turn calls
the function in the Apex Controller -->
    <ui:button label="Opt Out All Contacts"
      press="{!c.submitOptOut}" />
  </div>
  <br/>
  <p>
    <!-- The success or error message after clicking the Opt Out All Contacts
button will display here. This message is always rendered because it starts out as
just a space and doesn't appear to the user, but you could also add some
conditional rendering to show different things to the user depending on the state
such as a loading icon before the message is received from the Apex callout. -->
    <ui:outputText value="{!v.message}"/>
  </p>
</aura:component>
```

Step 3: Lightning Component Controller

To make your Lightning Component responsive to user input and communicate with your Apex Class, you'll need to create a client-side Controller for your Lightning Component. You can very easily create your Component Controller by clicking on the CONTROLLER button in the right panel of the Developer Console when you are focused on the tab for your Lightning Component.

The Lightning Component Controller has, by far, the most lines of code of any of the code pieces of our example opt out feature. This is because it's the go-between and has to pass information between the Lightning Component and the Apex Class.

In this example Lightning Component Controller, there are two functions. The first one, doInit, is meant to be executed when the Component is loaded. It will call up the getAccName method from the Apex Class to get the Account Name based on the Account ID from the Lightning Component. The second one, submitOptOut, will request the submitOptOut method from the Apex Class be executed using the Account ID from the Lightning Component. This should update the Email Opt Out checkbox on all the Contacts at the Account, and if it does, the client-side controller will set the message to be displayed in the Lightning Component to a success message. If there happens to be an error, the client-side controller will set the message to an error message. The submitOptOut function will also disable the "Opt Out All Contacts" button so the user can't press it again.

```

({
    //Function called on initialization of the component.
    doInit : function(component, event) {
        var action = component.get("c.getAccName");//Sets up a call to the getAccName
method from the Apex Class.
        action.setParams({
            "accid": component.get("v.recordId");//Sets up the Account record
ID from the Lightning Component as the accid Parameter when calling the
getAccName method from the Apex Class.
        });

        action.setCallback(this, function(response){//Set up the logic of what
will happen when this client-side controller hears back from the Apex Controller.
            var state = response.getState();//Get the state (SUCCESS or
ERROR) from the response from the Apex Controller.

            if (component.isValid() && state === "SUCCESS") {//If the Apex
Controller sends back a successful result...
                component.set("v.accname", response.getReturnValue());//The
return value should be a string containing the Account Name. Set this to the
accname attribute in the Lightning Component so that it can be displayed to
the user.
            }
            else {
                console.log("Failed with state: " + state); //Will log the state
in the console for a little help with debugging if there is a failure.
            }
        });
    }
});

```



```
    $A.enqueueAction(action); //Sends the call to the Apex Class to get the
Account Name.
  },

  //Function called when the button in the component is clicked/pressed.
  submitOptOut: function(component, event) {
    var action = component.get("c.optOutContacts"); //Sets up a call to
the optOutContacts method in the Apex Class.
    action.setParams({
      "accid": component.get("v.recordId") //Sets up the Account ID from
the Lightning Component as accid parameter in the call to the optOutContacts
method. });

    action.setCallback(this, function(response) { //Set up the logic of what
will happen when this client-side controller hears back from the Apex Controller.
      var state = response.getState(); //Get the state (SUCCESS or ERROR)
from the response from the Apex Controller. Different text will be pushed back
to the message attribute in the Lightning Component to be displayed to the
user depending on what happens.
      if (state === "SUCCESS") {
        component.set("v.message", "All Contacts have been successfully
opted out.");
      }
      else if (state === "INCOMPLETE") {
        component.set("v.message", "The operation may not have completed.
Please try again.");
      }
      else if (state === "ERROR") {
        component.set("v.message", "An error was encountered.
Please contact your administrator.");
        var errors = response.getError();
        if (errors) {
          if (errors[0] && errors[0].message)
            { console.log("Error message: " +
errors[0].message);
          }
        } else {
          console.log("Unknown error");
          component.set("v.message", "An error was encountered.
Please contact your administrator.");
        }
      }
    });

    $A.enqueueAction(action); //Send the call to the Apex controller to
update the Contacts at the Account to email opt out status.

    var btn = event.getSource(); //Find the button that was pressed to call
this function.
    btn.set("v.disabled", true); //Disable the button from being pressed.
  }
})
```

Step 4: Create the Button

After you have created the Lightning Component and its controllers, you can create a Lightning Action to access it.

In the Lightning Experience setup menu, navigate to the Object Manager, then go to the Account object. In the Buttons, Links, and Actions section, click on the New Action button.

Set up your Lightning Action similarly to the screenshot below. For the Action Type, select "Lightning Component". Then select the Lightning Component you created. If your Lightning Component does not appear in the picklist, go back to the code for your Lightning Component and make sure you've added this tag:

```
implements="force:lightningQuickAction,force:hasRecordId" in your opening aura:component
```

Enter a height that will fit the content of your component and a Label that will be displayed on the button itself and as the heading at the top of the Lightning Component when it is opened.

Account Actions

[Help for this Page](#)

New Action

Enter Action Information

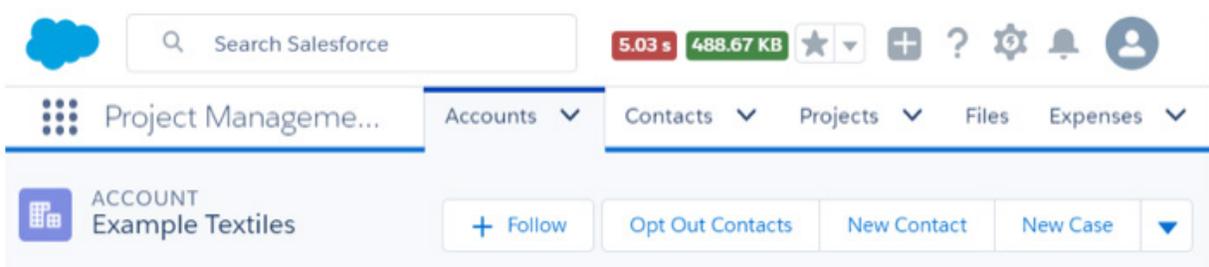
Object Name	Account <small>i</small>
Action Type	Lightning Component <small>i</small>
Lightning Component	c.optOut <small>i</small>
Height	250px <small>i</small>
Standard Label Type	--None-- <small>i</small>
Label	Opt Out Contacts
Name	Opt_Out_Contacts <small>i</small>
Description	<input type="text"/> <small>i</small>
Icon	Change Icon

Step 5: Add to Page Layout

After you've saved the button, you can add it to the Account page layout. On the Object Manager page for Account, navigate to the Page Layouts section. Click the Page Layout where you'd like to add your new Opt Out button. Select Salesforce1 & Lightning Actions from the UI element selector, find your new custom Opt Out button, and drag it into the Salesforce1 and Lightning Experience Actions section below, then Save.

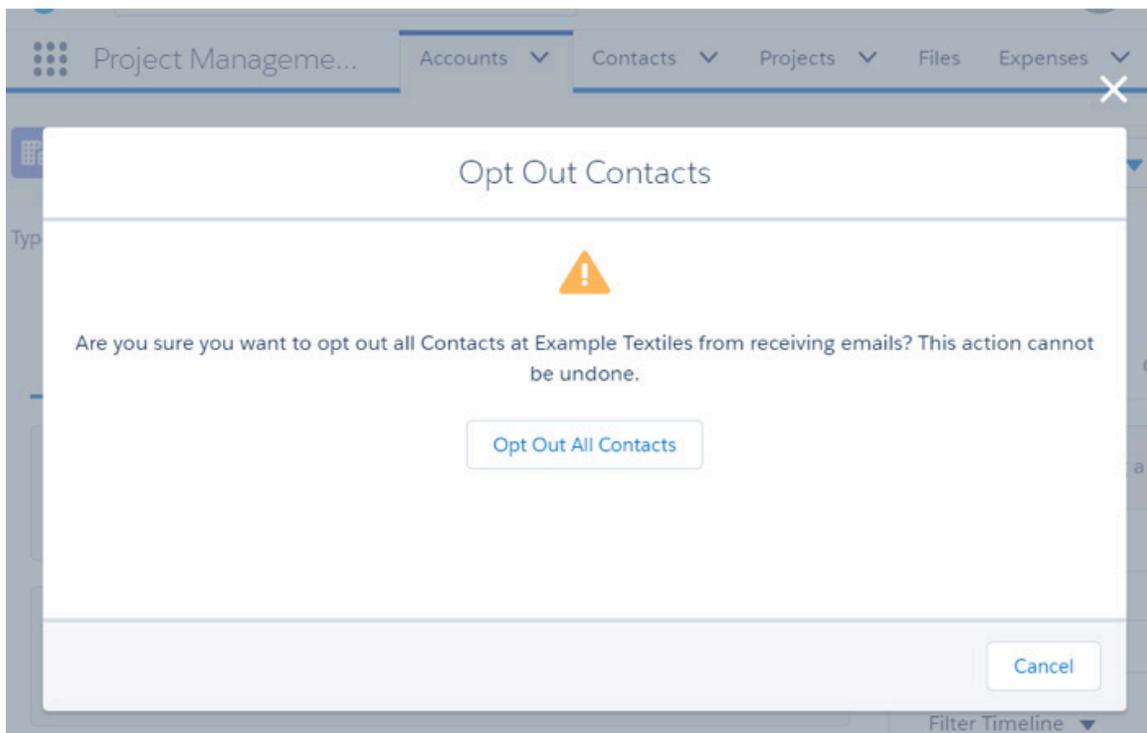
Note that since you'll be adding the button to the Salesforce1 and Lightning Experience actions section, it will only appear for users in the Lightning Experience. There is no need to create a separate page layout for your users in Lightning Experience to make sure users see the appropriate button for their UI setting.

Now your new custom button should appear if you navigate to an Account record in Lightning Experience. Depending on where in the order you placed the button, it may appear when you look at the Account button bar, or you may need to click the downward arrow to access it.



Step 6: Test and Deploy

Now that you've added the Opt Out Contacts custom button to the Account page layout, you can click on it to see what it looks like. Here is what you'll see when clicking on our example button.





You can click the Opt Out All Contacts button to test and confirm that your Contacts do get opted out and that the success message appears. Since this solution includes an Apex Class to handle the server-side logic, you'll also need to do some scripted testing in an Apex Test Class in order to deploy to production.

Once you've created the test class and verified that everything works as expected, deploy your Lightning Component Bundle, Apex Class, Apex Test Class, Custom Button, and Page Layout via your deployment tool of choice.

Example: Mass Delete from a List View

The Mass Delete button is one of the more common uses of a JavaScript button used in a list view, especially for custom objects, which don't have a mass-delete tool available in Force.com setup. Mass Delete allows a user to delete selected records from a list view. Since Mass Delete is a button used in a list view, it is necessary to create a Visualforce button to replace the JavaScript button since Lightning Actions are not yet available for List Views.

Here is an example of how to create a Mass Delete button that works in Lightning Experience for a custom object named "Project". Since this requires Apex code development, you'll need to build out the solution first in a Sandbox before deploying to your Production instance.

Step 1: Apex Class / Controller Extension

To create your Visualforce button, you will first need to create an Apex Class that will function as an extension of the standard set controller. This controller extension class needs to be created to handle deleting the selected records from a list view.

Here is a very basic version of the controller extension code for mass deletion of selected items from a list view. You could also add a property to show the number of selected records to be deleted or a preview of the records to be deleted and handling to show the user whether the deletion request succeeded or failed. If you look closely at this class, you will notice it could be used as a generic set controller extension for any object (although if you added a property to show a preview of the records to be deleted, it would become object-specific).

```
public with sharing class MassDeleteProjects {
    //Construct a standard set controller as controller extension.
    public ApexPages.StandardSetController controller;

    public MassDeleteProjects(ApexPages.StandardSetController constructor) {
        controller = constructor;
    }

    //Page Reference method to delete the selected records and then go back to
    the previous page.
    public pageReference deleteProjects() {
        delete controller.getSelected();
        return controller.cancel();
    }
}
```

Step 2: Visualforce Page

Next you will need to create a Visualforce page to be accessed by your button. You will need to use the standard controller for your custom object (in this case `Project__c`) and reference the Apex Class you created as the controller extension. You will also need to define a `recordSetVar` to make it accessible from a list button rather than a detail page button.

You could just call the method from the controller to delete the records as an initialization action on the page so that it doesn't show up at all to the user, but since the method deletes multiple records, it's a good idea to have a user confirmation step. It does not require much more development work to add the user confirmation step since it's already necessary to build a Visualforce page.

Now that the Lightning Design System is available for Visualforce pages, take advantage of it to make your deletion confirmation page a seamless part of the Lightning Experience UI rather than sticking out as

an obvious custom UI element with Classic styling.

You can find resources for everything you need to know about the Lightning Design System at <https://www.lightningdesignsystem.com/>, Salesforce's official website for the Lightning CSS framework. This site can help you find the right styles for each component of your Visualforce page (or Lightning component or mobile app).

Here is what the code for a basic deletion confirmation page looks like. It contains a header, some helpful warning text for the user, a button to call up the `deleteProjects` method from our custom controller extension, and a cancel button to use the cancel feature from the standard controller to go back to the previous page.

You could also add some additional elements to this page to correspond to additional elements in the controller extension, such as a count of the selected records, a preview of the selected records, or markup to display success or error messages.

```
<apex:page standardController="Project__c" extensions="MassDeleteProjects"
recordSetVar="proj">
  <apex:slds /><!-- Implement the Salesforce Lightning Design System -->
  <div class="slds-scope" style="text-align:center">
    <div class="slds-modal__header">
      Confirm Delete
    </div>
    <div class="slds-modal__content slds-p-around--medium">
      <p>
        Are you sure you want to delete the selected Project records?
        You may not be able to retrieve these records from the recycle
        bin after choosing to delete them.
      </p>
    </div>
    <apex:form >
      <apex:commandLink action="{!deleteProjects}" id="del">
        <button class="slds-button slds-button--neutral">
          Delete Projects
        </button>
      </apex:commandLink>
      <apex:commandLink action="{!cancel}" id="goback">
        <button class="slds-button slds-button--neutral">
          Cancel
        </button>
      </apex:commandLink>
    </apex:form>
  </div>
</apex:page>
```

Step 3: Create the Button

Once the Visualforce page is set up, you can create a button to access it. You will need to create a Custom Button or Link rather than an Action in order to use your Visualforce page.

In the Lightning Experience setup menu, you'll need to navigate to the Object Manager, then go to the object you're setting up a mass delete button for (in this example, Project). In the Buttons, Links, and Actions section, click on the New Custom Button or Link button.

Set up your button similarly to the screenshot below. First, give your button a user-friendly label. Then select List Button as the display type so that the button can display from a list view. Make sure the Display Checkboxes box is checked since the Mass Delete feature is dependent on the user being able to select items from a list. For the Content Source, select Visualforce Page and from the Content picklist, select the Visualforce Page created above. If your Visualforce Page does not appear in the Content picklist, go back to the code for your Visualforce Page and make sure that the standardController refers to your custom object correctly and that the recordSetVar is defined in the opening apex:page tag.

Custom Button or Link Edit Save Quick Save Preview Cancel

Label: Delete Selected

Name: Delete_Selected i

Description:

Display Type: Detail Page Link [View example](#)
 Detail Page Button [View example](#)
 List Button [View example](#)
 Display Checkboxes (for Multi-Record Selection)

Behavior: Display in existing window with sidebar View Behavior Options

Content Source: Visualforce Page

Content
MassDeleteProjects [MassDeleteProjects]

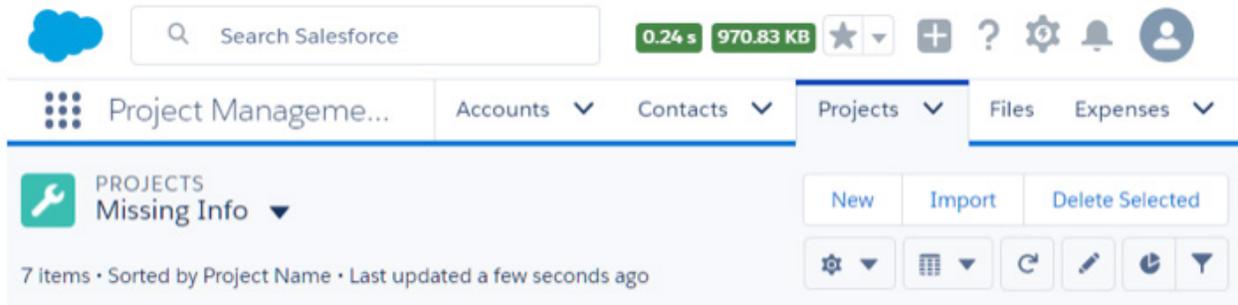
Save Quick Save Preview Cancel

Quick Tips

- [Getting Started](#)
- [Sample Buttons & Links](#)
- [Operators & Functions](#)

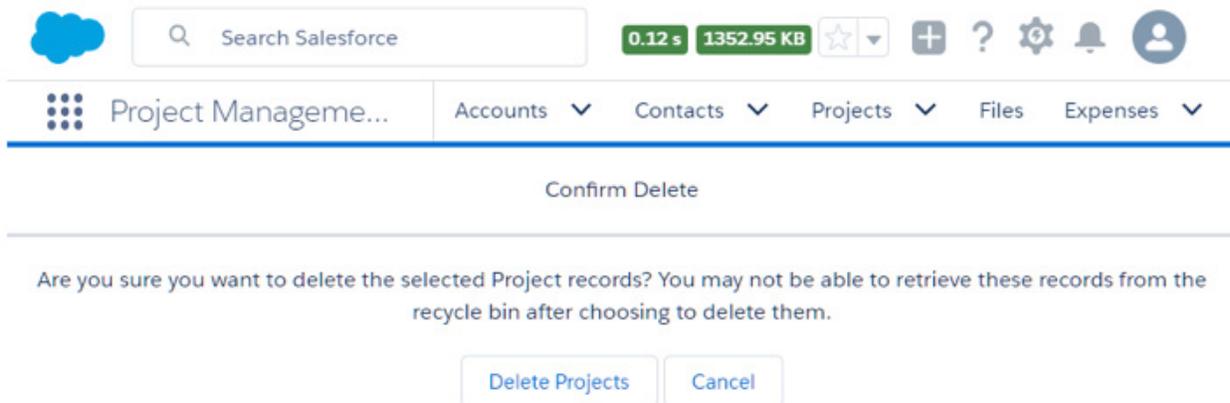
Step 4: Add to Layout

After you've saved the button, you can add it to the list view layout. On the Object Manager page for Project (or your custom object), navigate to the Search Layouts section. Click the arrow beside the List View layout to select Edit. Move your button from Available Buttons to Selected Buttons and Save. Now your custom button should appear if you navigate to a list view for your custom object. In the screenshot below, you can see the Delete Selected button at the far right of the navigation for a list view in the Projects tab.



Step 5: Test and Deploy

Now you can test out the button for yourself. Pull up a list view of your custom object, select a few disposable records, and click your new custom button. Here is what our example deletion confirmation page looks like.



You can click the Delete Projects button (or the equivalent button that you have created) to test and confirm that it deletes the records, but since you've built an Apex Class to handle the record deletion, you'll also need to do some scripted testing in an Apex Test Class in order to deploy to production.

Once you've created the test class and verified that everything works as expected, deploy your Apex Class, Visualforce Page, Apex Test Class, Custom Button, and Search Layout via your deployment tool of choice.



Other Scenarios for Advanced Solutions

These two examples are only a small sample of the ways that JavaScript buttons can be replaced with alternative methods that work in Lightning Experience.

Another common use for JavaScript buttons is to perform record validation and displaying feedback or error messages based on user input when saving a record. Depending on the exact scenario, this type of JavaScript button can be replaced by an Apex Trigger or Validation Rule, which will take effect in either Classic or Lightning Experience. You could also replace a button of this kind with a Lightning Action that handles the validation and retrieves messages for the user via a Lightning Component Controller.

Sometimes a scenario where it seems at first like a coded solution would be needed to replace a JavaScript can be satisfied by a point-and-click solution instead. For example, if you're using JavaScript to run validation and provide feedback messages to remind users to fill in a few specific fields at specific stages in a business process, you might be able to adopt the Path feature to help users focus on those specific fields in the appropriate phases. Another JavaScript button that could be replaced with a point-and-click solution is a button that replaces values on a Contact record from the Contact's associated Account record. This is useful when a Contact moves to a different Account and needs contact information updated accordingly, but user validation is still required to make sure all the details are correct. This wouldn't need to be replaced with a Trigger, Process Flow, or Lightning Action, but could be replaced by a humble Quick Action with pre-populated values from the new Account record.

Another common type of JavaScript button is one that makes API calls to another system. This type of button will always require a programmatic replacement, but a Lightning Action should always be able to cover it (unless it's a list view button, in which case a Visualforce button can replace it). The Lightning Action can call a Lightning Component that can handle user input and display any success or error messages with a Component Controller that handles sending the input to an Apex Controller that sends to and receives content from the external API and populating any feedback messages back into the Component.

Implementation Tips

A key difference between working with JavaScript buttons and programmatic solutions that are compatible with the Lightning Experience is the sandbox. Although it's a good practice to test out a JavaScript button in a sandbox first before deploying it to Production, it's not required, and it's frighteningly easy to add buttons that don't work, or worse, harm your data (no doubt one of the reasons Salesforce is phasing this solution out). On the other hand, programmatic solutions will need to be developed in a Sandbox first and all Apex code will need to have at least 75% code coverage from Apex test classes in order to be deployed to a Production environment.

Take advantage of the testing requirement in the Sandbox to write good Apex tests that cover every scenario you can think of with plenty of system.asserts as well as testing out your buttons through the UI to make sure you've worked out all the bugs before deploying to production. This can save you a lot of complaints or support tickets from your users down the line.

When you do roll out Lightning Experience to your users, you can roll out to a smaller group of users first who can help spot any additional bugs or snags before your roll out Lightning Experience to all your users. As you roll out Lightning Experience make sure your users know how to use the switcher so they can go back to Classic to use a critical JavaScript button if a replacement still has a bug. This is a great safety-net to use, especially if you are new to developing Lightning Components or Apex Classes.



ADDITIONAL RESOURCES

If you need assistance with a Lightning migration, contact us at salesforcesuccess@bayforce.com. We are always here to assist you with your Salesforce projects.

Self-Guided Training on Salesforce Trailhead

Lightning Experience Rollout: https://trailhead.salesforce.com/modules/lex_migration_rollout

Lightning Alternatives to JavaScript Buttons: https://trailhead.salesforce.com/modules/lex_javascript_button_migration

Lightning Components Basics: https://trailhead.salesforce.com/modules/lex_dev_lc_basics

Lightning Design System: https://trailhead.salesforce.com/modules/lightning_design_system

Salesforce Developer Documentation

Lightning Components Developer Guide: https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/intro_framework.htm

Lightning Design System: <https://www.lightningdesignsystem.com/>

Apex Developer Guide: https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_dev_guide.htm