Salesforce mini how to: Open a SubTab in Console from Lightning Web Component

Desislav Michev | **23 Nov 2020**

These days UI development in Salesforce prefers using Lightning Web Components (LWC) as they are faster and more modern than Lightning Components (aura). Unfortunately, they are still not a complete replacement for Lightning Components. According to this official page for example, a Workspace API is not supported. This API is used for managing the tabs in Console. The Console itself is highly used by the sales teams or other teams that use SoftPhone inside Salesforce. So it seems that we have a problem, the solutions to which are not that complicated.

Solution #1

The solution is to create a wrapper Lightning Component for your LWC and implement all the tabs functionality inside that component as it is proposed by Salesforce itself. By sending custom events from LWC to the wrapper Lightning Component you can use Workspace API indirectly.

This approach could be useful in case your LWC is invoked by Action Button for example. As you already know, the action button could not execute LWC directly, but can execute Lightning Component instead. This component is your wrapper and listener for the custom events.

You can read what the official Salesforce documentation says on sending and handling custom events.

Solution #2

On the opposite side of the first solution, in case your LWC is placed directly in the page via App Builder, then you have no Lightning Component wrapper. The solution then is to recreate what Salesforce does internally to work with the tabs inside Console.

To demonstrate these two approaches, we will create components which will allow us to open Account Record from Contact View in a subtab.

Implementation

Solution #1

For this solution we will first need a LWC and then Lightning Component which will perform the opening of a record in a new subtab.

exampleLWC.html

```
<template>
   <p>This is example LWC component for Solution #1</p>
   <p>
       <lightning-button disabled={openAccountButtonDisabled} title="Open Account" label="Open Account" onclick={onOpenAccountClick}></lightning-button>
   </p>
</template>
```

exampleLWC.js

```
import { LightningElement, api, wire } from 'lwc';
import { getRecord } from 'lightning/uiRecordApi';

export default class ExampleLwc extends LightningElement {
 @api
 recordId;

 accountId;
 openAccountButtonDisabled = true;

 @wire(getRecord, { recordId: '$recordId', fields: ['Contact.AccountId']})
 getRecordAccount({ data }) {
   if (data) {
     this.accountId = data.fields.AccountId.value;
     this.openAccountButtonDisabled = false;
   }
 }

 onOpenAccountClick() {
   this.openSubTab(this.accountId);
 }

 openSubTab(recordId) {
   this.dispatchEvent(new CustomEvent('subtab', {
     detail: {
       recordId
     }
   }));
 }
}
```

LWCConsoleWrapper.cmp

```
<aura:component description="LWCConsoleWrapper" implements="force:hasRecordId, force:lightningQuickAction">
   <aura:handler name="init" value="{! this }" action="{! c.onInit }"/>

   <lightning:workspaceAPI aura:id="workspaceAPI" />
   <c:exampleLWC recordId="{! v.recordId }" onsubtab="{! c.handleOpenSubTab }" />
</aura:component>
```

And its controller - LWCConsoleWrapperController.js

```
({
 onInit: function (component, event, helper) {

 },
 handleOpenSubTab: function (component, event, helper) {
   const workspaceAPI = component.find('workspaceAPI');
   const recordId = event.getParam('recordId');

   if (!recordId) {
     return;
   }

   workspaceAPI.isConsoleNavigation().then(isConsole => {
     if (isConsole) {
       workspaceAPI.getFocusedTabInfo()
         .then(
           result => {
             workspaceAPI.openSubtab({
               parentTabId: result.tabId,
               recordId,
               focus: true
             }).then(
               tabId => {
                 console.log("Solution #1 - SubTab ID: ", tabId);
               }
             );
           }
         );
     }
   });
 }
});
```

As you can see it is pretty simple - LWC is executed via Lightning Component, which in our example was executed by an Action Button in Contact View page. Clicking the “Open Account” button will emit CustomEvent with Account RecordId as a payload - method “openSubTab”. Our Lightning Component has a registered event handler and executes a function - “handleOpenSubTab” when the event occurs.

Inside our Lightning Component we have full access to the WorkspaceAPI, so we can easily open a new subtab with the desired information. In our case it is the Contact’s Account.

SUBSCRIBE TO OUR NEWSLETTER

Name

Email

SUBSCRIBE

Solution #2

In this solution we use WorkspaceAPI directly from our LWC. This is accomplished by a simple CustomEvent with specific payload, sent to the “window”. It is that simple.

examplePureLWC.html

```
<template>
   <lightning-card>
       <div class="slds-p-around_medium">
           <p>This is example LWC component for Solution #2</p>
           <p>
               <lightning-button disabled={openAccountButtonDisabled} title="Open Account" label="Open Account" onclick={onOpenAccountClick}></lightning-
           </p>
       </div>
   </lightning-card>
</template>
```

examplePureLWC.js

```
import { api, LightningElement, wire } from 'lwc';
import { getRecord } from 'lightning/uiRecordApi';

export default class ExamplePureLwc extends LightningElement {
 @api
 recordId;

 accountId;
 openAccountButtonDisabled = true;

 @wire(getRecord, { recordId: '$recordId', fields: ['Contact.AccountId']})
 getRecordAccount({ data }) {
   if (data) {
     this.accountId = data.fields.AccountId.value;
     this.openAccountButtonDisabled = false;
   }
 }

 onOpenAccountClick() {
   this.invokeWorkspaceAPI('isConsoleNavigation').then(isConsole => {
     if (isConsole) {
       this.invokeWorkspaceAPI('getFocusedTabInfo').then(focusedTab => {
         this.invokeWorkspaceAPI('openSubtab', {
           parentTabId: focusedTab.tabId,
           recordId: this.accountId,
           focus: true
         }).then(tabId => {
           console.log("Solution #2 - SubTab ID: ", tabId);
         });
       });
     }
   });
```

```
  }

  invokeWorkspaceAPI(methodName, methodArgs) {
    return new Promise((resolve, reject) => {
      const apiEvent = new CustomEvent("internalapievent", {
        bubbles: true,
        composed: true,
        cancelable: false,
        detail: {
          category: "workspaceAPI",
          methodName: methodName,
          methodArgs: methodArgs,
          callback: (err, response) => {
            if (err) {
                return reject(err);
            } else {
                return resolve(response);
            }
          }
        }
      });

      window.dispatchEvent(apiEvent);
    });
  }
}
```

All you need to do is copy or recreate the “invokeWorkspaceAPI” method and start using it. For ”methodArgs” you should refer to the WorkspaceAPI official documentation. Keep in mind that the method returns JS Promise.