# Dialog Control, SSJS and Refreshing an Area of the Page

You can reference the original article submitted by Paul Withers

## Quick Summary

A common requirement is to close an Extension Library dialog control and partially refresh an area of the page. The natural assumption would be to perform a partial refresh and set the refreshId. But that comes from a misunderstanding of how the dialog is closed and the refresh triggered.

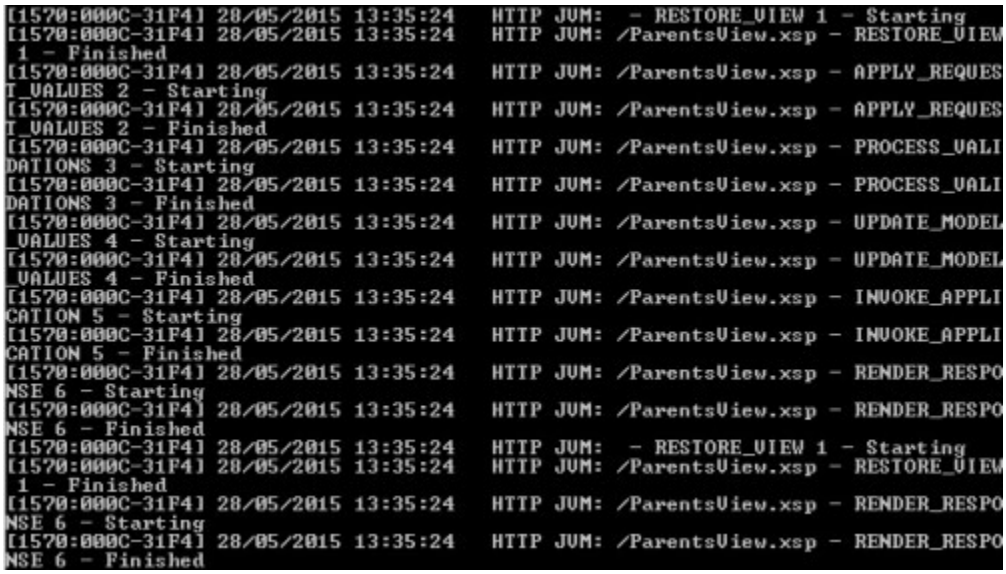
## Detailed Overview

If closing the dialog via CSJS, a second parameter is added for the client-side ID of the area to refresh. This, understandably, triggers a `XSP.partialRefreshGet()` to refresh the relevant area of the page. No data is posted back to the server, no updates from the dialog are processed, hence a `XSP.partialRefreshGet` is sufficient.

In most scenarios, the dialog will be closed via SSJS, because some data wants to be posted from the dialog to the server, in order to influence the refresh area. For example, we may add a button to the row in e.g. a Data View, which runs SSJS / Java to set a viewScope variable with the current entry's NoteID and load the dialog. Because we want to act on the right document from the dialog and after the dialog has been closed, we need to serialise a pointer to the relevant document on the server side, for which viewScope is the natural place. The dialog then prompts for some kind of data entry or action, which runs SSJS / Java and uses that viewScope variable to identify which document to process. Afterwards, if processing is successful, the code will clear the viewScope variable, close the dialog and refresh the view.

That is the key description - "close the dialog and refresh the view".

This all becomes apparent when using FireBug or a Phase Listener on an example in the Extension Library demo database. The Phase Listener shows the process goes through all six phases, then through phases 1 (Restore View) and 6 (Render Response).




Investigating the source code for the UIDialog's SSJS methods makes everything apparent. The `show()` method runs the following:

**show() method**

```
Action pendingAction = new Action(context,this,ACTION_SHOW_DIALOG,p);
ExtLibUtil.postScript(context, pendingAction.generateClientScript());
```

The `hide()` method runs similar code:

**hide() method**

```
Action pendingAction = new Action(context,this,ACTION_HIDE_DIALOG,refreshId,refreshParams);
ExtLibUtil.postScript(context, pendingAction.generateClientScript());
```

The key in both is that, after the SSJS runs, it calls ExtLibUtil.postScript() to write CSJS back to the browser and trigger it. So it's posting **Client-Side JavaScript** back to the browser. Investigating the **Action** class (an inner class within the UIDialog class) shows it's passing CSJS to call `XSP.openDialog()` and `XSP.closeDialog()`. This directs us to the code in **Dialog.js**. Looking at that, we see both trigger an `XSP.partialRefreshGet`. So the SSJS is returning CSJS which is triggering a partial refresh.

This explains the extra logging of Restore View and Render Response phases. It also explains why the updates to the view are displayed when closing an Extension Library dialog. The partial refresh coded in your XPage makes updates and returns CSJS, which calls a `XSP.partialRefreshGet` to reload the view entries for the current page (NotesXspViewEntries cannot be serialised, so every refresh has to get the view entries again) and update the HTML on the page with the *changed* details.

## Related Articles

- Dialog Control, SSJS and Refreshing an Area of the Page