Simpli List Views Developer/User Manual

Table of Contents

Overview

Simpli List Views (SLV) is an AppExchange managed package. It allows better management of list views within the Salesforce ecosystem as well as providing extra functionality on top of the Salesforce OOTB capabilities.

The following are some of the added features of SLV over the OOTB list views provided by Salesforce –

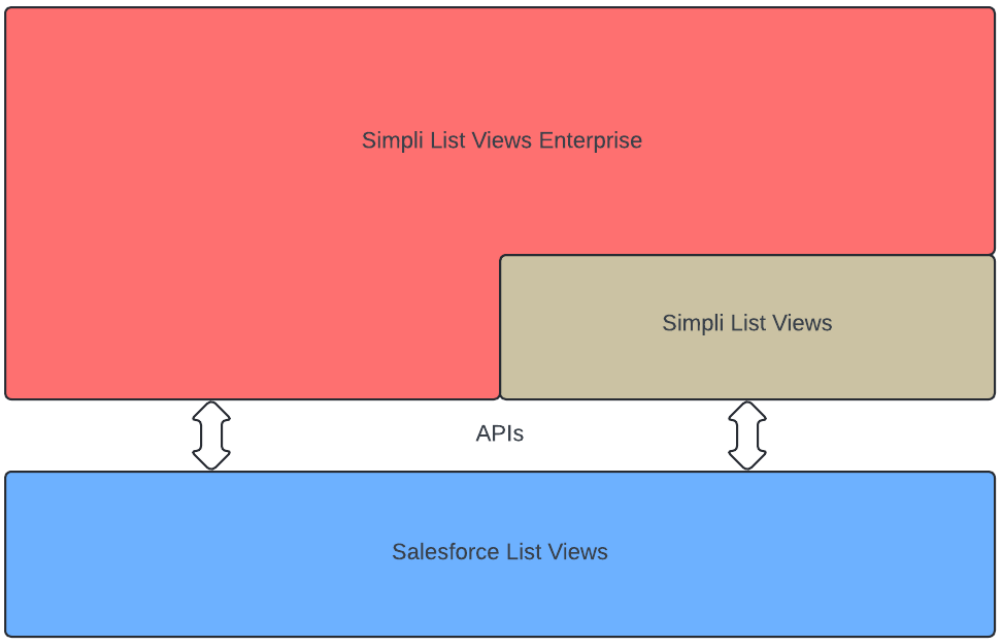
- Interactions between multiple list views.
- Adhoc additional field display
- Display list views for activities, tasks, campaigns, campaign members.
- Conditional highlighting – row or cell.
- Create custom actions including for multi-row selection
- Create list views for adhoc data outside of SObjects.Return up to 2500 rows at one time.
- Real-time multi-column sorting
- One-click data download and PDF display
- Auto refresh
- Aggregate footer row
- Individual column styling

These features as discussed in more detail further down.

Architecture Overview

This section discusses the general architecture and security of the application. It will be of interest to those developers taking advantage of the API accessibility features or need to understand the architecture for security reasons.

SLVE is an application that sits of top of SLV and Salesforce. The high level architecture could be considered as follows –



This diagram shows the architecture for an individual SFDC org. SLVE communicates both with SLV and with Salesforce using both Apex and APIs based on whether Tooling and REST API requests need to be made.

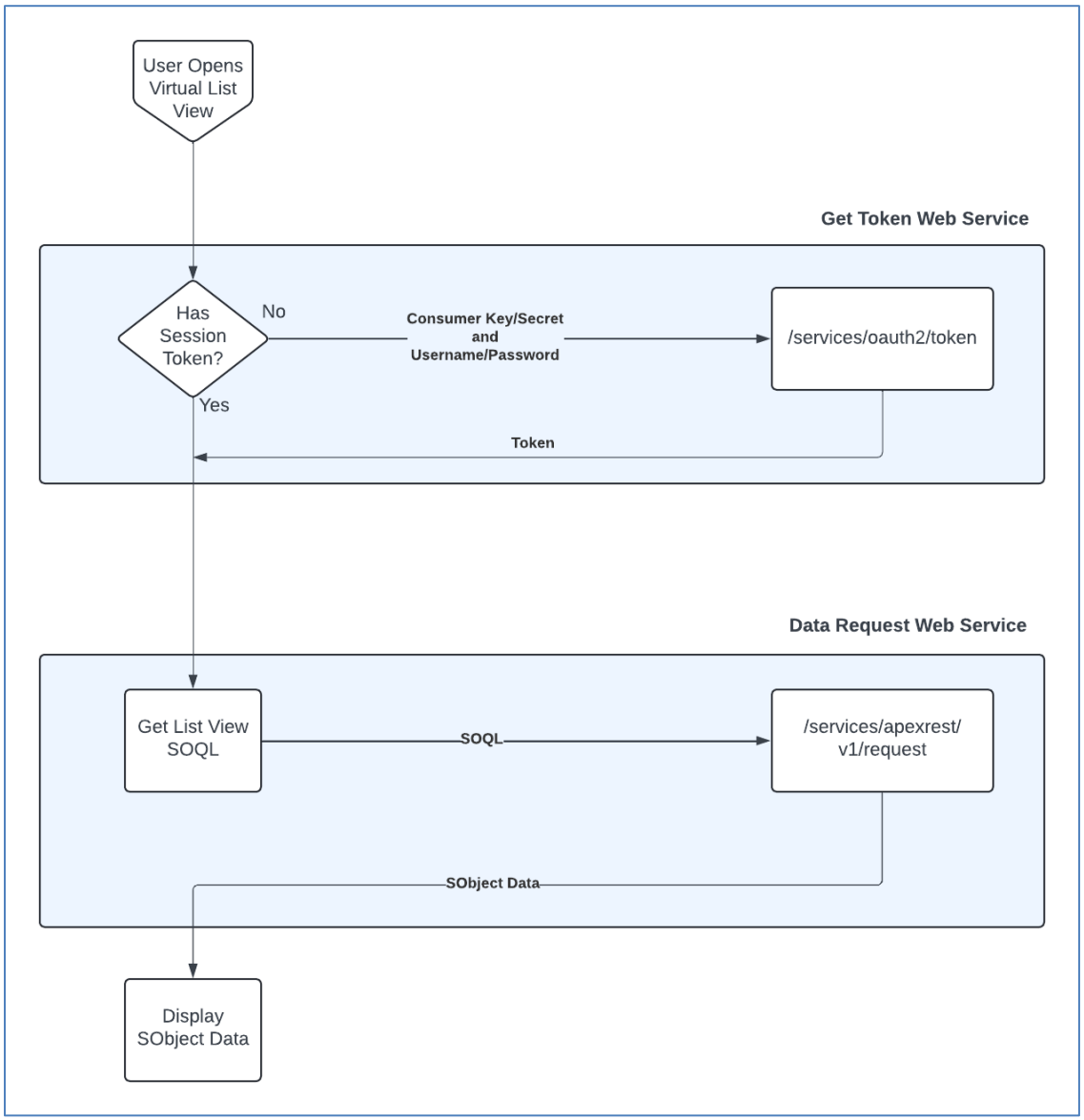
Security Overview

SLVE connects to other Salesforce instances. SLVE uses the standard named credential/auth provider/connected app pattern available through Salesforce to ensure the credentials remain secure. A description of how to create a connection is described below.

Note –

- Connection information is only used for Virtual List View functionality.
- The above process assumes that a connected app exists on the target Salesforce org.
- For connections to work correctly all remote site settings must be appropriately set on the source SFDC org.
- SLVE must be installed on both the source and target orgs before the virtual list view functionality can be used.

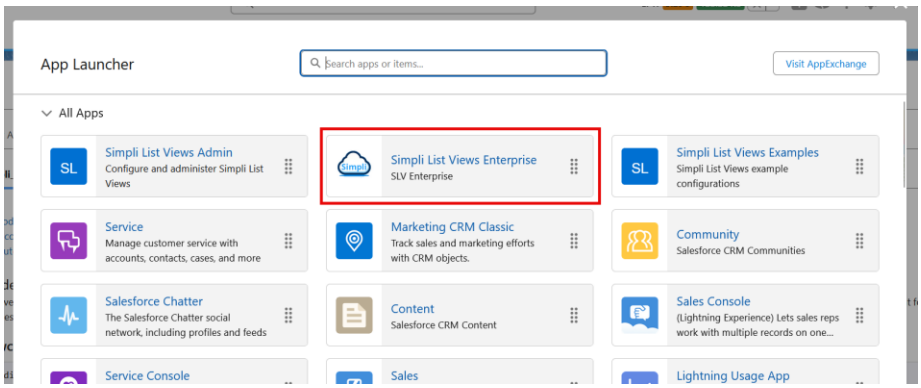
Below is a flow diagram of data retrieval from a destination Salesforce org including retrieving a token if required. The data passed during each part of the process is shown.



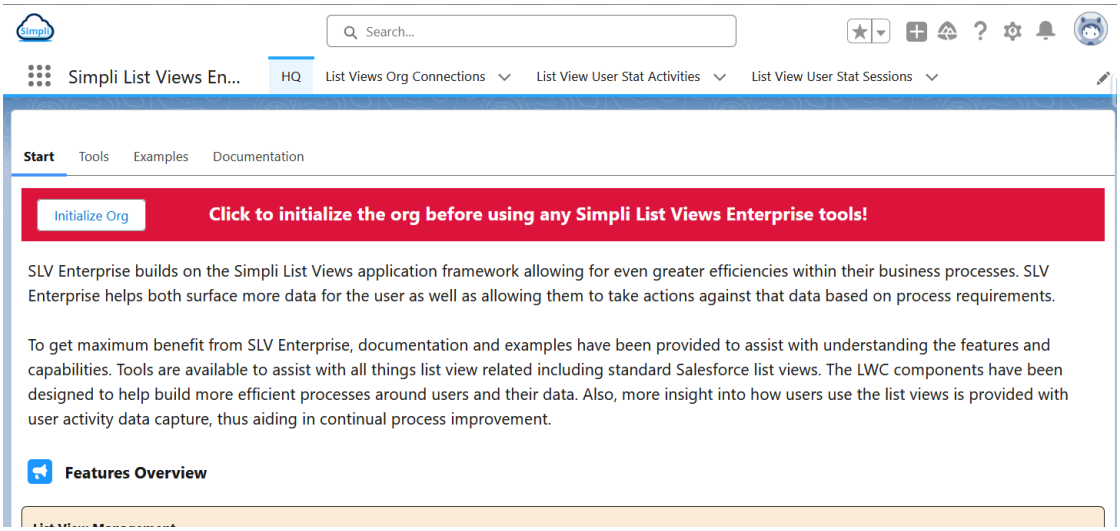
Installation

SLVE is dependent on the SLV AppExchange package. That package can be found here. SLV is a free package. Ensure to install and initialize SLV before installing SLVE. SLVE will not work and cannot be installed if the SLV package is not found.

Once SLV has been installed SLVE can be installed. After installing SLVE the admin user should go to the newly added Simpli List Views Enterprise app.



After clicking on the app the user will be taken to the HQ tab and placed at the Start subtab. A button will appear requesting the admin user initialize SLVE. This initialization should only take a few moments but is critical as tools and other list views will not function if SLVE has not been initialized.



Once initialization has completed the button will disappear. The org is now initialized and ready for development and testing of SLVE.

General Features

Simpli List Views (SLVs) are designed to take the OOTB list view to a new level. By providing extra functionality, SLVs can be incorporated into business processes, allowing for better user experience and more efficient processes.

Highlights

- Multiple list view interactions
- Campaign/Member, Task/Activity list views
- Include lookup fields (5 levels deep)
- Complex SOQL query's
- Auto-refresh at configured intervals (https://www.youtube.com/watch?v=-Ry89arO48I)
- Multiple column sorting
- Footer aggregation
- Row/Cell highlighting based on criteria
- Lists up to 2500 records
- Column styling (https://www.youtube.com/watch?v=7DFBIFzNn5k)
- Create list view actions
- Export list view data with the click of a button as CSV or PDF
- Include deleted records in list view (records show in red)
- List views for all objects on one page
- Create list views from Tooling API
- Configure list views at the org, page and list view level
- Use drop down or type-ahead list view selection.
- Developers…..think <lightning-listview>

There are many online videos discussing the above features including examples.

List View Modes

There are several modes that an SLV can be used in depending on the need.

App Page

Used when multiple object list views are to be made available. Very useful when data from multiple list views needs to be viewed quickly without going to multiple pages to find the data. Also useful for system administrators to view data from all available objects on one page.

This mode is typically used on LWC pages of type App Page where the page is formatted for a single region and the SLV component takes up the entire region. A drop down is provided where

an object can be selected. A second drop down allows for the selection of a list view. The list view is then displayed. All SLV features are available in this mode.

Single Object List View

Used when list views from only a single object are to be made available on the component. This mode is typically used on LWC pages of type App Page where several list views for the same object may be needed to fulfill a function.

Single List View

Used when only one list view is made available on the component. This mode is typically used on LWC pages of type App Page where multiple list views are used together to filter for records based on both criteria of the list views and records chosen by the user.

Related List View

Used when one list view is made available on the component. This mode is typically used on LWC pages of type Record Page where the records displayed in the list view are related to the currently selected record on the page. See the following video for more information - https://www.youtube.com/watch?v=4geKs0cqqoE

Split View

Used when the list view is displayed on one side of the screen where the Id of the selected record is passed to other components on the other side of the screen. The list can display a maximum of 4 fields. This mode is typically used on LWC pages of type App Page where the user works on multiple records of the same type in a short period of time. See the following video for more information - https://www.youtube.com/watch?v=RmDgbEpwizw

Stand Alone

Used when arbitrary data outside of a Salesforce object is to be displayed. Useful for when disparate information from multiple sources needs to be displayed to the user. This mode can only be displayed on LWC pages of type App Page. Some functionality is not available for this mode. The data and column information is provided to the component in JSON format. See the following video for more information - https://www.youtube.com/watch?v=aViCr9cn7bk

List Views Administration

There are a number of tools available for the administration of SLV.

Sandbox Data Management

When spinning up new sandboxes that are not full copies there are a few activities that need to be performed to ensure SLV works as expected. Depending on the sandbox needs SLV can be initialized just using the core list views. This means that only SLV components using core list views will function. If custom list views are needed they will need to be imported from the production org.

Core Initialization

SLV data needs to be reconstituted

Data Import

Data importing requires access to the SLV Admin app.

List View Settings

There are several settings that can be set to indicate whether common functionality is available or not. The settings are applied by precedence depending on where they are set. Precedence of disabled settings follows different precedence than enabled settings.

Disabled – global over component over list view

Enabled – listview over component over global

For example no matter how the component or list view is set, if a global setting is disabled the setting will be disabled for all list views. Also, if a global setting is enabled, one component could have the setting enabled but another component could have it disabled.

All global, component and list view settings and any notes specific to the setting can be found on the start page of the List Views Start tab on the Admin and Examples apps.

Scheduled Core List Refresh

Ddsds

List View Actions

To Do

Actions Wizard

To Do

List View Management

List View Management is one of the main features of SLVE. It allows list views to be managed from a single page rather than having to go to each object individually.

Highlights

- Standard list views can be created for any standard object allowed by Salesforce.
- Custom list views can be created for any object.
- The list view management tool has been LWC componentized for use anywhere its needed.
- Tool is configurable based on needs. e.g. if users can only create custom list views.
- A modal LWC component is provided.
- Go to HQ -> Tools -> to see and start using the tool.

Standard vs. Custom List Views

Salesforce has OOTB functionality for working with standard list views. This allows list views to be managed but only on the object page where the list views exist.

Simpli List Views adds an additional layer on top of the Salesforce standard list view. SLV enhances the standard list view functionality by including features such as aggregate totals, row highlighting, automatic refresh etc. Those list views are still considered standard. The SLV functionality could be thought of as window dressing around the standard list view. Any field or logic changes have had to go through the standard Salesforce process.

Simpli List Views also introduced the custom list view. This list view allows for the display of data using a SOQL query that the user provides. Although this is very powerful and although much of the functionality of SLV works with the custom list view there is some functionality which is not available.

The list view management tool helps handle that delineation as well as provide a one-stop-shop for managing list views within the org. The list view management tool can create/update/delete/clone both standard and custom list views. It ensures the correct information required for each kind of list view is provided and saved appropriately.

LWC And Modal Components

We recognize that users and admins might have their own applications that they use for their administration needs. The SLVE list view management tool is an LWC component that can be dragged on to any LWC page for use as needed. This allows users to place the tool on any page that makes sense and fits into their admin processing needs.

It is also available as a modal component. This is useful if the tool needs to be displayed as a modal dialog box.

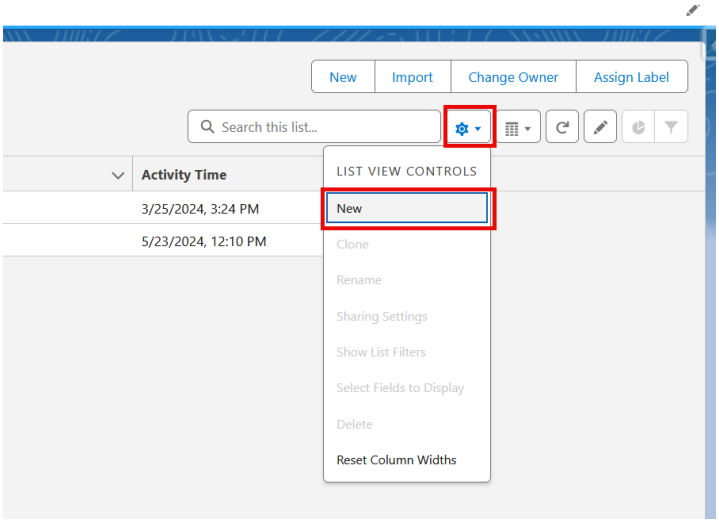
Adding Standard Objects

If trying to create or modify a standard objects list view it might be the case that the object does not appear in the drop down list of objects. In this case it is required that the object be processed before the list view management tool can make the object available for modification.

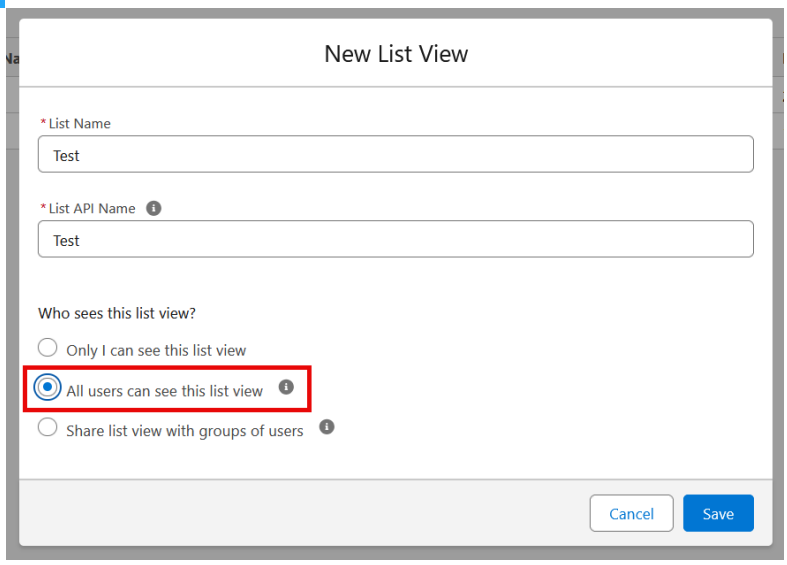
There is a tool to assist in making the object available. This can be found in HQ -> Tools -> Add Standard Object. The tool works by having the user create a list view for the object. That list view is then ingested and the object field details are then identified and stored away. This is done for all standard fields for the object.

Tool Step 1

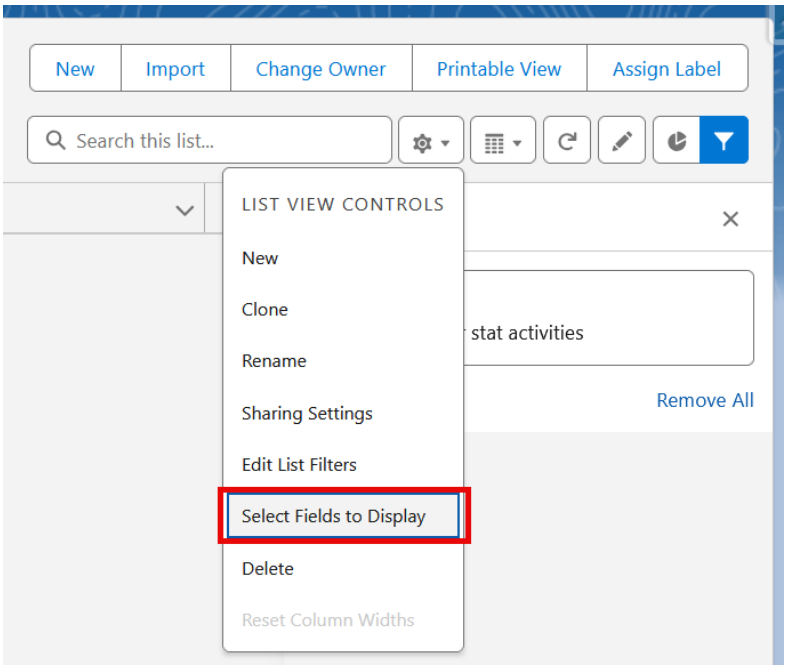
Go to the object that list views need to be created for. Create a new list view by clicking on the cog icon on the right and then clicking New.



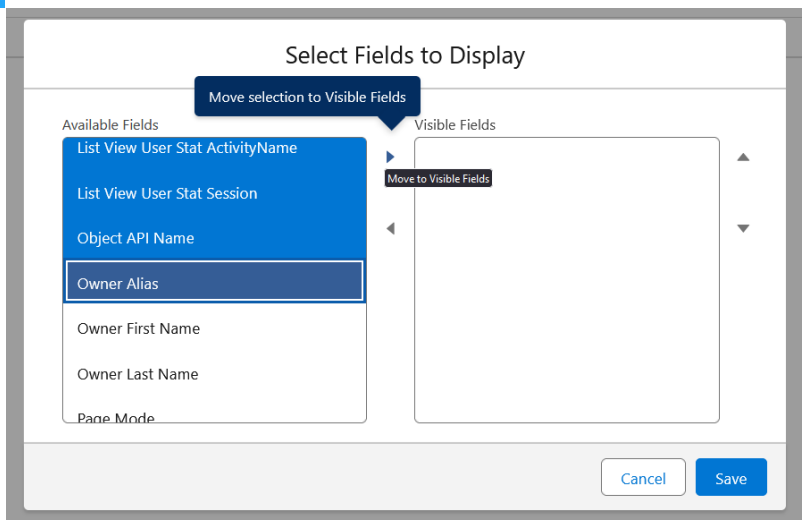
Add a name for the list view and ensure that all users can see the list view. This option is critical as private list views are not visible to SLV.



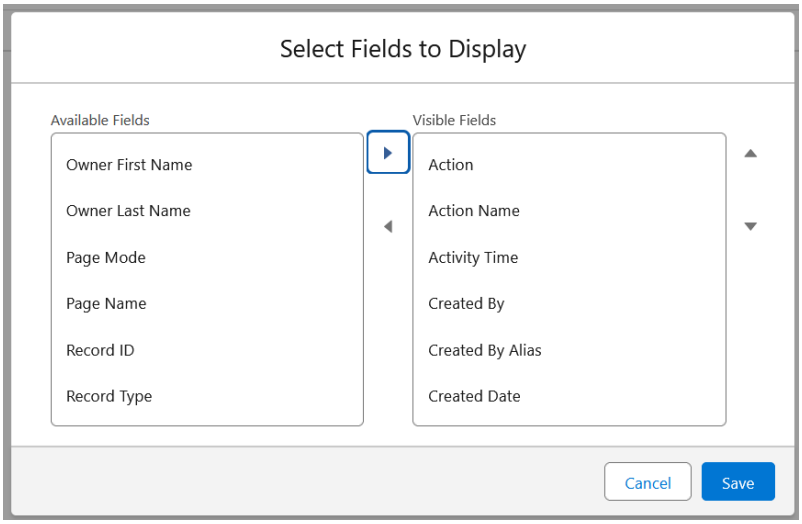
Click save. Once the list view has been created click the cog icon on the top right and select the “Select Fields to Display” option.



If there are any visible fields remove them all. Then select the first 15 fields from the left list.

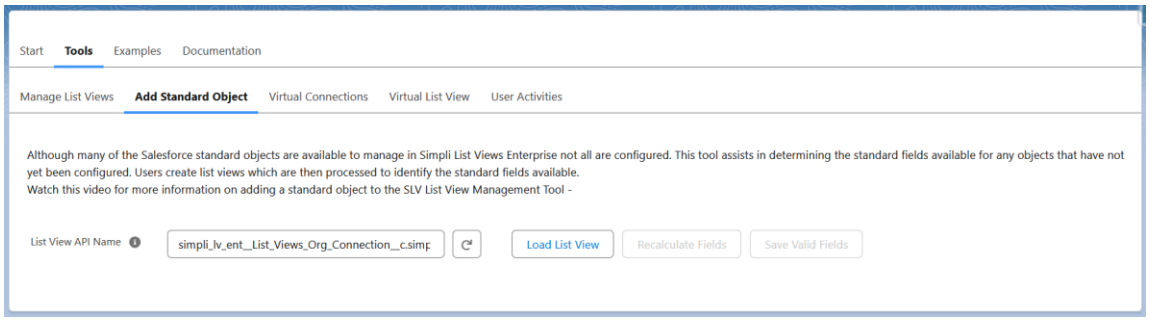


and move them to the right list and click Save.

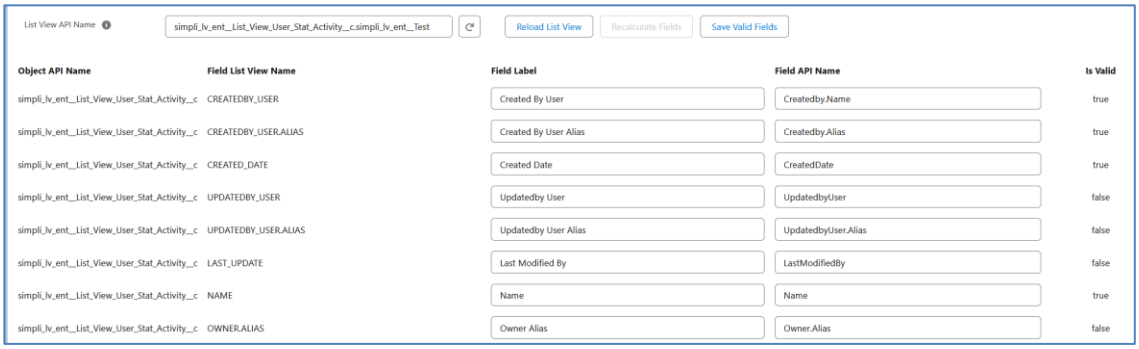


Tool Step 2

Open HQ in a new tab and go to HQ -> Tools -> Add Standard Object. the following is displayed –



Notice the list view just created will show up as the default List View API Name. This removes the need to enter the list view name manually. If the listview should be different then enter the API name including the object API name. i.e. Account.MyAccountListView. Click Load List View to display the first 15 fields. The results will look similar to the following –



Object API Name	Field List View Name	Field Label	Field API Name	Is Valid
simpli_lv_ent__List_View_User_Stat_Activity__c	CREATEDBY_USER	Created By User	Createdby.Name	true
simpli_lv_ent__List_View_User_Stat_Activity__c	CREATEDBY_USER.ALIAS	Created By User Alias	Createdby.Alias	true
simpli_lv_ent__List_View_User_Stat_Activity__c	CREATED_DATE	Created Date	CreatedDate	true
simpli_lv_ent__List_View_User_Stat_Activity__c	UPDATEDBY_USER	Updatedby User	UpdatedbyUser	false
simpli_lv_ent__List_View_User_Stat_Activity__c	UPDATEDBY_USER.ALIAS	Updatedby User Alias	UpdatedbyUser.Alias	false
simpli_lv_ent__List_View_User_Stat_Activity__c	LAST_UPDATE	Last Modified By	LastModifiedBy	false
simpli_lv_ent__List_View_User_Stat_Activity__c	NAME	Name	Name	true
simpli_lv_ent__List_View_User_Stat_Activity__c	OWNER.ALIAS	Owner Alias	Owner.Alias	false

These results show the internal Salesforce field information as well as an approximate guess as to the Salesforce API field name. Unfortunately Salesforce uses their internal field name for list views and so we need to map those fields to the external API field name. SLVE requires that the Salesforce API field name be accurate to be able to function correctly.

The admin user needs to correct the field API name and the field label. Knowledge of the Salesforce object fields is useful in determining the correct API field name. For example, in the above screenshot the field name returned is UpdatedByUser. This is not an field API name. The correct name is ModifiedBy.Name.

Once the admin user has updated the API names they can click the Recalculate Field button. The fields will then be submitted and tested. If SLVE can determine that the API name is valid the Is Valid column will change to true. The admin user can submit the fields as many times as necessary to determine the API field name.

Once all the API field names return their validity as true the admin user can click the Save Valid Fields button and the fields will be saved for later use as well as being removed from the list. Note that only fields that validate will be saved. If a field is not known or will definitely not be used it can be ignored.

Tool Step 3

Now that the first 15 fields have been saved the admin user then needs to return to the list view they created, remove all fields and add the next 15 fields and perform the same process as the first 15. This needs to happen for all fields in the object. Once this has

been completed for all fields the list view management tool will ensure the object appears as an option in the dropdown list.

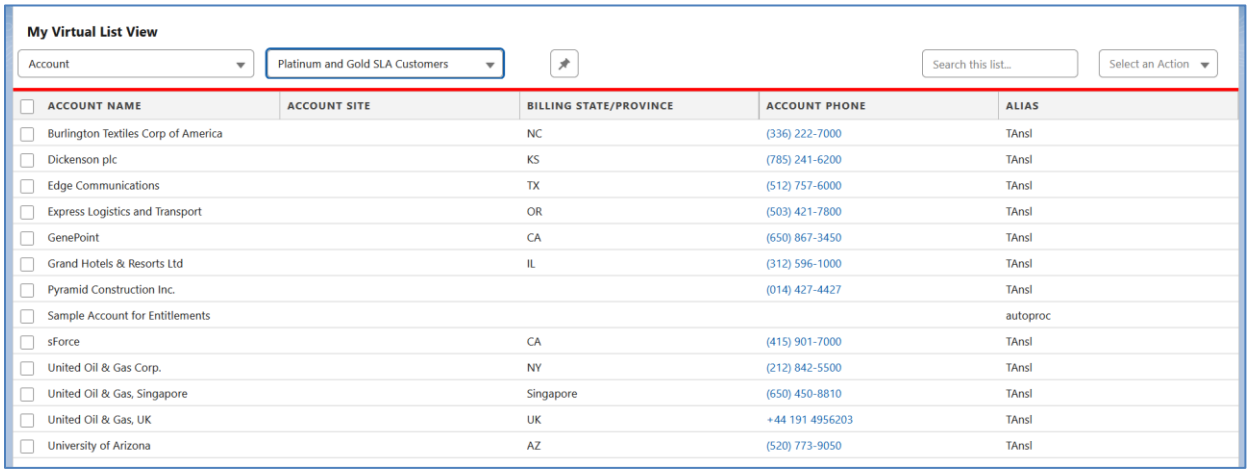
Virtual List Views

Virtual list views allow users to view and act against data residing on another Salesforce instance. This is useful when disparate information is stored across different Salesforce orgs. Actions can be implemented which execute on both local and remote data based on process needs.

Highlights

- Virtual list views are always associated with a connection.
- Connections are managed by going to the HQ -> Tools -> tab.
- Virtual list views use a separate LWC component.
- For actions to work with virtual list views SLV Enterprise must be installed in the destination org and matching actions must exist in both orgs.
- Go to HQ -> Examples -> to see examples using the LWC component.

Virtual list views look almost identical to regular list views. Much of the functionality is available including actions and pinning. Some functionality does not, including inline editing. To differentiate between virtual and regular list views a red line is displayed across the top of a virtual list view. This is to help ensure the user understands they are working with information from a different Salesforce org.



ACCOUNT NAME	ACCOUNT SITE	BILLING STATE/PROVINCE	ACCOUNT PHONE	ALIAS
Burlington Textiles Corp of America		NC	(336) 222-7000	TAnsl
Dickenson plc		KS	(785) 241-6200	TAnsl
Edge Communications		TX	(512) 757-6000	TAnsl
Express Logistics and Transport		OR	(503) 421-7800	TAnsl
GenePoint		CA	(650) 867-3450	TAnsl
Grand Hotels & Resorts Ltd		IL	(312) 596-1000	TAnsl
Pyramid Construction Inc.			(014) 427-4427	TAnsl
Sample Account for Entitlements				autoproc
sForce		CA	(415) 901-7000	TAnsl
United Oil & Gas Corp.		NY	(212) 842-5500	TAnsl
United Oil & Gas, Singapore		Singapore	(650) 450-8810	TAnsl
United Oil & Gas, UK		UK	+44 191 4956203	TAnsl
University of Arizona		AZ	(520) 773-9050	TAnsl

When first installing SLVE no connections are set up. All virtual list views must be associated with a connection. A new connection needs to be configured.

Creating A New Connection

SLVE uses named credentials, authorization providers and connected apps to create secure connections between Salesforce. These mechanisms are managed by Salesforce and used by

SLVE within the managed package. The following steps are needed to create a secure connection with another Salesforce org –

- Create connected app (target org)
- Create authorization provider (source org)
- Create named credential (source org)
- Create SLVE connection record (source org)

Create Connected App (on TARGET Salesforce org)

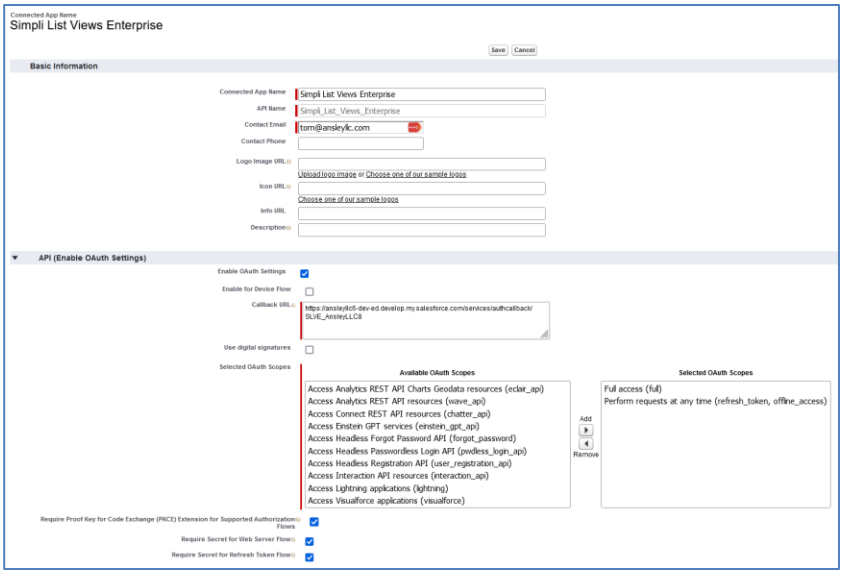
A connected app needs to be created on the target Salesforce org. Connected apps inform Salesforce that any requests using the connected app are legitimate. This is part of the OAuth2 specifications used by SLV when making callouts to other Salesforce orgs.

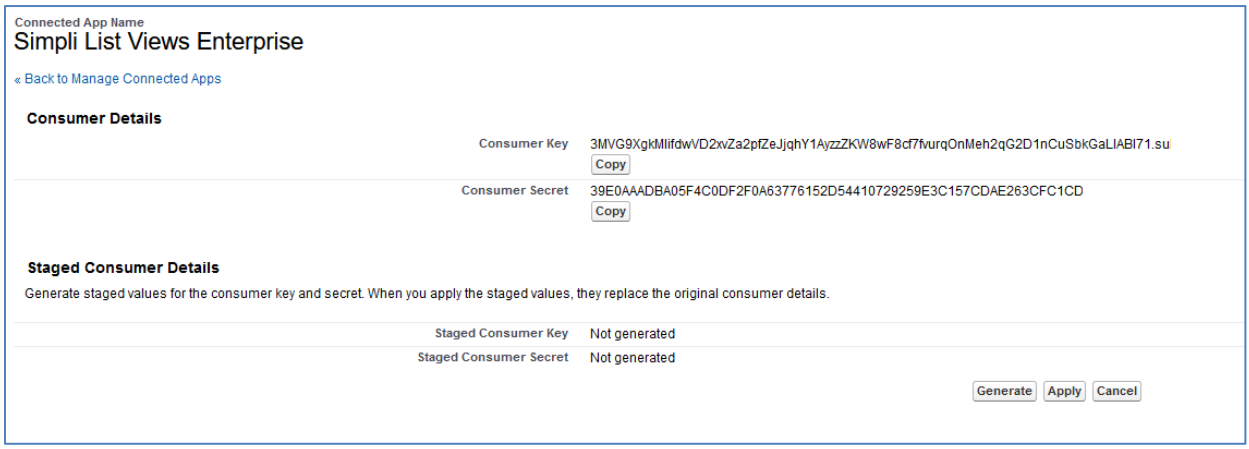
Create a new connected app by going to Setup|Build|Create|Apps|Connected Apps and entering the following information –

Connected App Name – the name helps recognize what the connected app is used for.
Contact Email – the email of the point of contact for the connected app.
Enable OAuth Settings – this checkbox must be checked.
Callback URL – enter a fake URL (i.e. https://fake.com) as this URL will be updated later based on other configuration.
Selected OAuth Scopes – Ensure the following 2 scopes are selected –
Full access (full)
Perform requests at any time (refresh_tokenm offline_access)

Click save. Once the connected app is saved the consumer key and secret, which are both uniquely generated for each connected app is needed in the next step. To retrieve those values view the newly connected app and click the **Manage Consumer Details** button. You may be asked to verify your identity. After verification the consumer key and secret are displayed. Save these values away for later.

See the below screenshots of an example connected app configuration and consumer key/secret.





Create Authorization Provider (on SOURCE Salesforce org)

Create an authorization provider by navigating to Setup|Administer|Security Controls|Auth. Providers. Click the New button and select Salesforce as the provider type. The following information needs to be provided –

Name – provide a name that identifies the auth provider.
URL Suffix – usually this is the name removing any special characters.
Consumer Key – the key provided in the previous step.
Consumer Secret – the secret provided in the previous step.
Authorize Endpoint URL – the authorization URL of the TARGET Salesforce org including the service name. The URL is usually as follows –

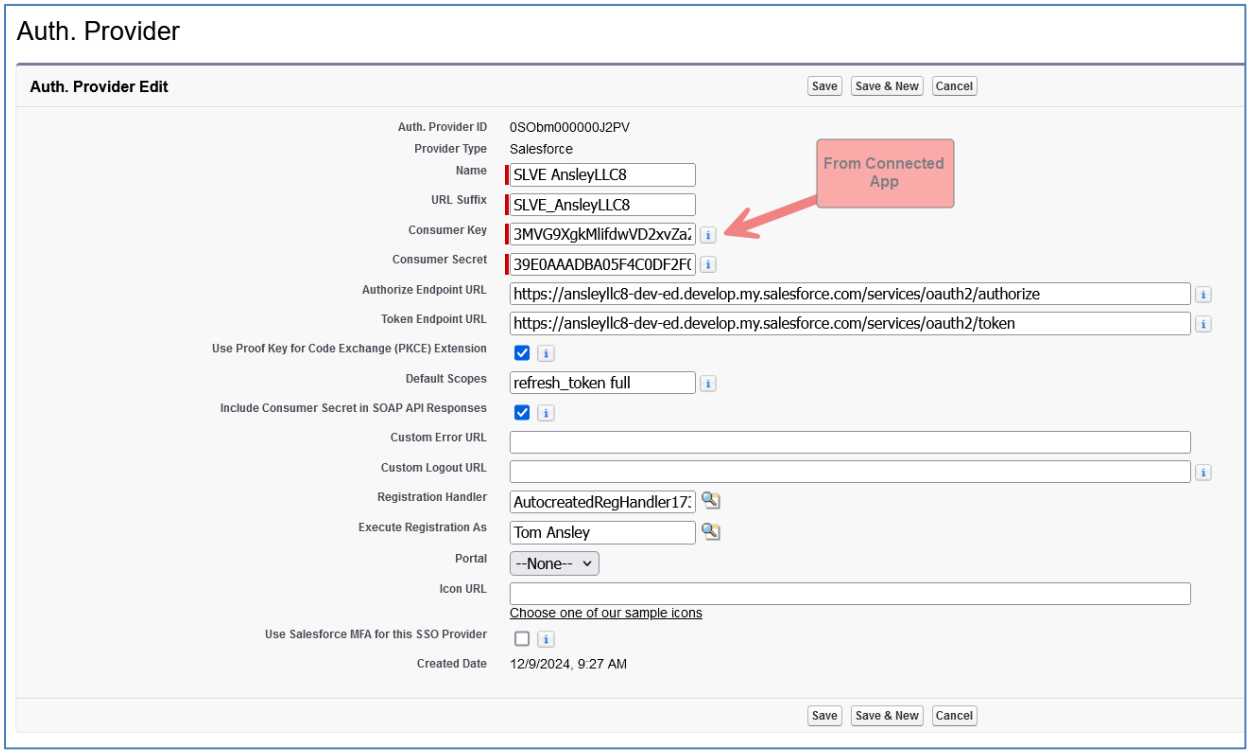
```
https://<SOURCE_SFDC_ORG>/services/oauth2/authorize
```

Token Endpoint URL – the token URL of the TARGET Salesforce org including the service name. The URL is usually as follows –

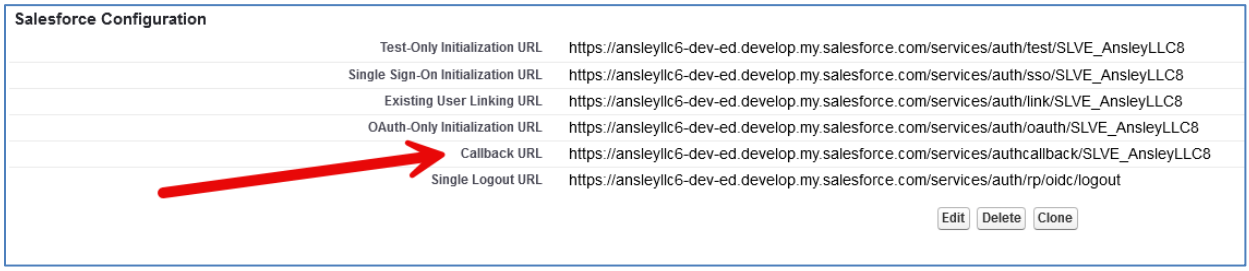
```
https://<SOURCE_SFDC_ORG>/services/oauth2/token
```

Registration Handler – click the *Automatically create a registration handler template* button which will generate an apex class.
Execute Registration As - select the user assigned to execute any processing when a user tries to login.

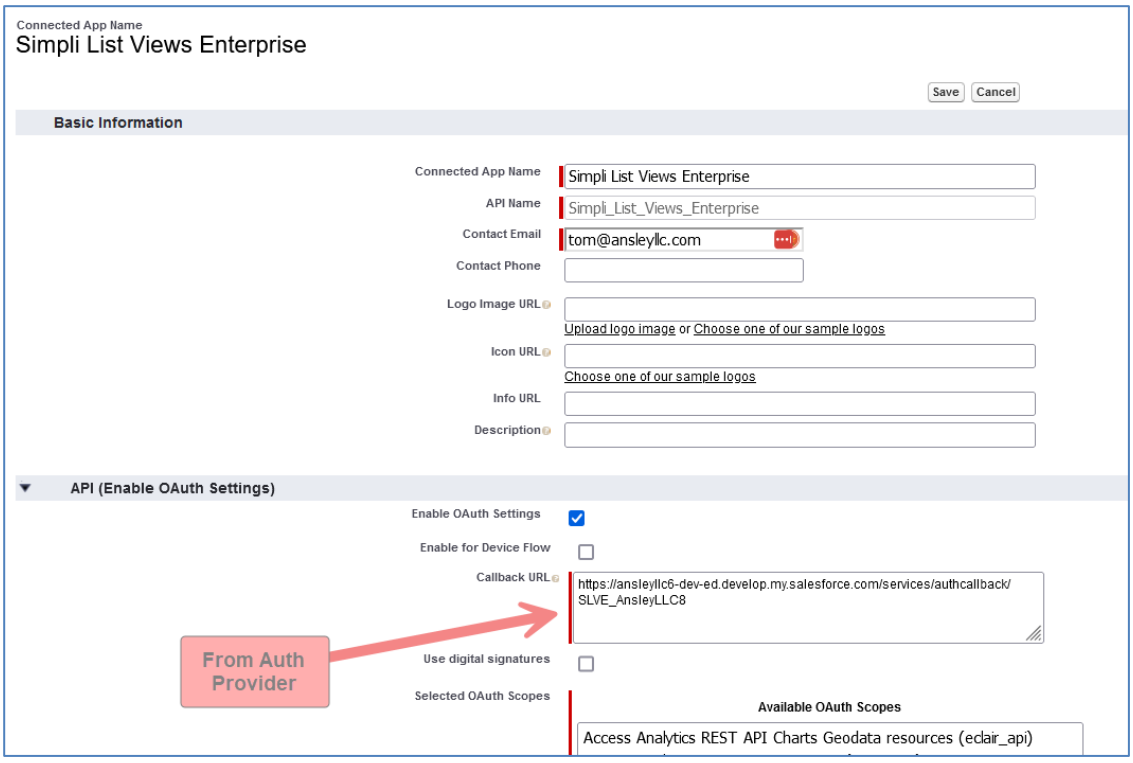
See screenshot below of an example authorization provider.



Once the Auth Provider has been created a list of URLs will be provided as shown in the screenshot below.



The callback URL must be copied into the Connected App created in the first step where the "fake" URL was used as a placeholder.



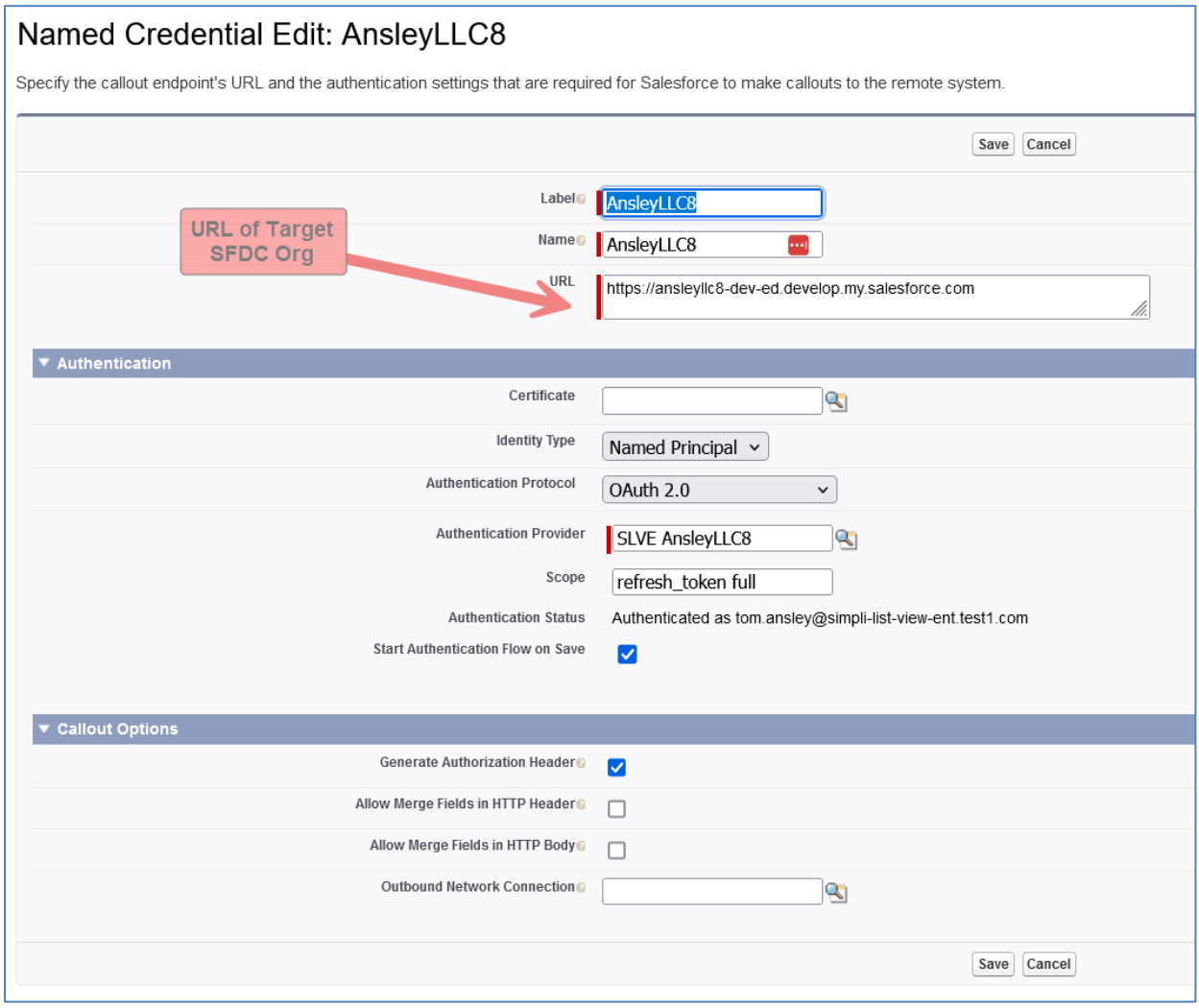
Create Named Credential (on SOURCE Salesforce org)

A named credential specifies the URL of a callout endpoint and its required authentication parameters in one definition. This simplifies the setup of authenticated callouts.

Navigate to Setup|Security |Named Credentials and click *New Legacy* button. The following fields must be populated –

Label – a memorable name perhaps including the user being used and the authentication provider.
URL – the url of the SOURCE Salesforce org.
Identity Type – Named Principal
Authentication Protocol – OAuth 2.0
Authentication Provider – choose the provider created in the above step.
Scope – refresh_token full ← use this exact text including spaces.
Start Authentication Flow On Save – check the box
Generate Authorization Header – check the box.

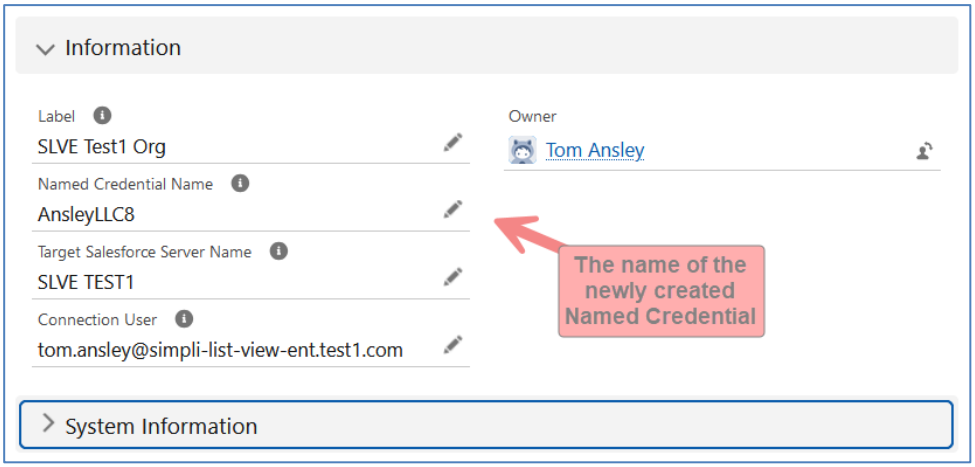
Click Save. You will be directed to the login page for authentication. Use the TARGET user/pass to authenticate.



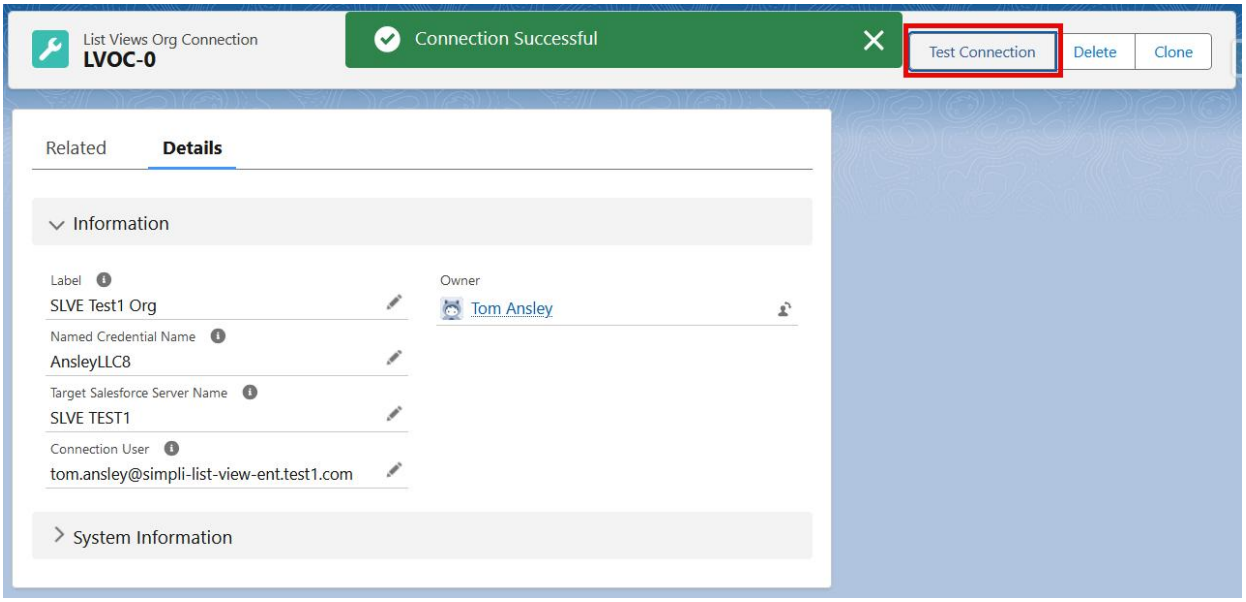
Create SLVE Connection Record (on SOURCE Salesforce org)

The connection records allow SLVE to display all connections available for use when connecting using the virtual list view. Most of the fields are for display purposes only. The following fields must be populated –

Label - The label for the connection. This label is displayed to users when viewing available connections.
Named Credential Name - The name of the Named Credential this connection is associated with.
Target Salesforce Server Name - The name of the target Salesforce org this connection will be connecting to. This field is only used for informational purposes and is not required.
Connection User - The user used to make the connection. This field is for informational purposes only and is not required.



Once the record has been saved the connection can be tested. Click on the Test Connection button on the connection record page. If everything was successful a success message should be displayed.



Virtual Actions

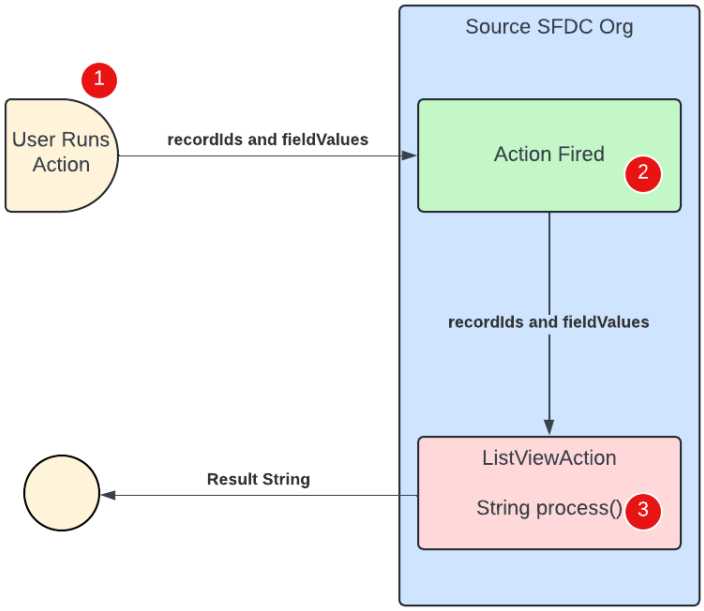
Virtual actions are a powerful tool in the SLVE suite. The capability allow for actions to be performed on another SFDC org against a set of selected records. The functionality is almost identical to the standard SLV actions that can be performed today. The only difference is that SLVE virtual actions return a result to the source org which itself can then be processed.

An example could be that an action returns a set of data records. The source org could then upsert those records. Essentially importing the records. All done from a list view!

The standard apex implementation for an action looks as follows –

```
global with sharing class ListViewActionXXX  extends ListViewAction {

    global ListViewActionXXX() {
    }

    public override String process(List<String> ids, Map<String, Object> fieldValues)
    {
        return 'Ok:This was successful';
    }

}
```

And the process flow looks as follows -

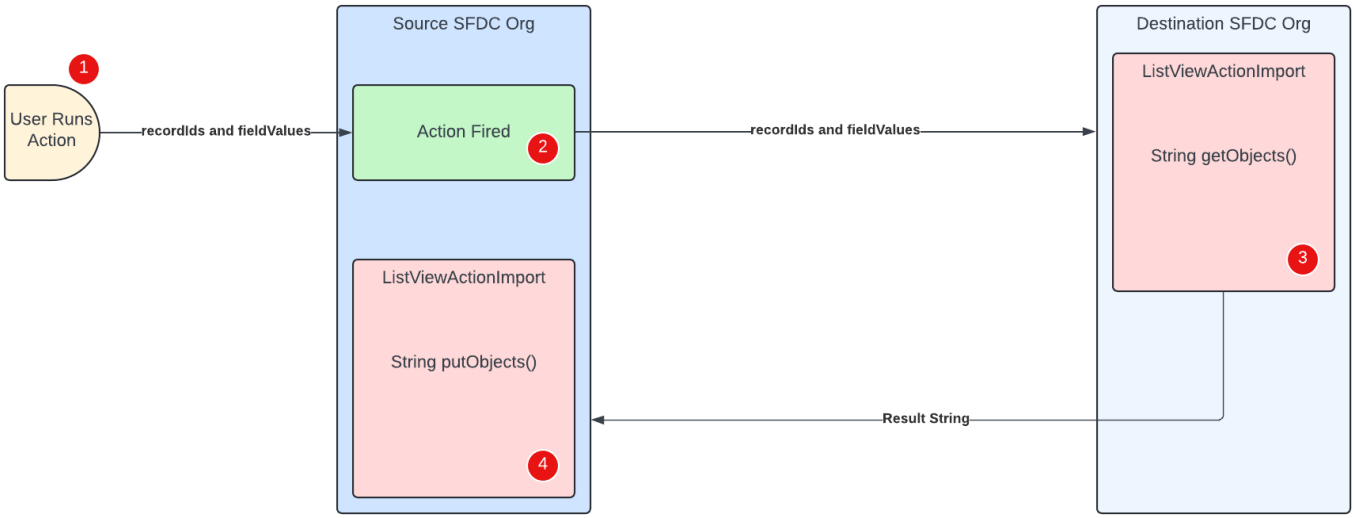


The process is simple. The action is fired. The implemented apex class which extends ListViewAction is instantiated and its process() method is fired to perform the logic. The result is a string which is returned to the UI.

The virtual action implementation is similar. It looks like the following –

```
global with sharing class ListViewActionVirtualXXX extends ListViewActionImport {

    global ListViewActionVirtualXXX() {

    }

    public override String getObjects(List<String> ids, Map<String, Object> fieldValues)
    {
        Object resp = something to be returned...;
        return JSON.serialize(resp);
    }

    public override String putObjects(String jsonString)
    {
        String response = 'Errors:';
        if (!noErrors)
                response = 'Ok:Successfully Processed';
        return response;
    }

}
```

The process flow looks as follows –



The action is fired. The recordsIds and fieldValues are passed across to the destination org. The implemented apex class which extends ListViewActionImport is instantiated and its getObjects() method is fired to perform the first set of logic. The result is a JSON string which is passed back to the source org. The implemented apex class is then reinstantiated and its putObjects() method is fired. The result is a string which is returned to the UI.

Things to note about virtual action

- The virtual actions apex class must exist on both the source and destination orgs.
- The List_View_Action__c record holding the action details including the action apex class name must exist on both source and destination orgs.
- Do not add JSON to the JSON string. i.e. do not serialize any JSON string into another JSON string. This will have unpredictable results.

Some virtual actions have been provided. These include a basic import utility. For more complex importing of records bespoke virtual actions should be implemented.

Virtual examples have also been provided and can be found by going to HQ -> Examples -> Virtual Examples

API Accessibility

SLVE opens up the entire Simpli List Views framework to developers to enhance SLV to better fit their processes. This includes both apex methods as well as LWC component events which can be handled and acted upon. This is a powerful addition to the developers toolkit allowing external code to interact with SLV components.

Highlights

- API Apex methods for processing list views.
- LWC component events for handling user interactions.
- API examples provided.

Developers should look to the documentation and examples provided in the installed package. Go to HQ -> Examples as well as HQ -> Documentation to see built out examples and code as well as full descriptions and examples for each apex method and LWC event.

Appendix

Apex APIs

ListViewHelper

getListViewObjects()

Parameters None

Response Map<String, String> - a map with the retrieved object names. Key = API name, Value = label

- An aggregated list of objects that have both standard and custom list views are returned.
- The logic uses the SLV list view data. Refresh the SLV list view data if unexpected results are returned
- This method does not use the cache but always returns a fresh compiled list of objects.

Example

```
Map<String, String> objs =
simpli_lv_ent.ListViewHelper.getListViewObjects();
for (String key: objs.keySet())
    System.debug(key + ', ' + objs.get(key));
```

getObjectListViews(String objectName)

Parameters objectName - the API name of the object that will have its list views returned.

Response Map<String, simpli_lv__List_View__c> - a map with the retrieved list views. Key = list view API name, Value = list view object

- Both standard and custom list views for the provided object is returned.
- The following logic is used to determine if the list view should be returned to the currently running user
    1. If user is sys admin
    2. If user is owner
    3. If list view has role requirement and user is in role
    4. If user is in allowed groups
    5. If user is in allowed territories
    6. Otherwise do not return list view
- This method uses the cache if available. The list views list is stored in cache per user.

Examples

```
Map<String, simpli_lv__List_View__c> listviews =
simpli_lv_ent.ListViewHelper.getObjectListView(String objectName);
for (String key: listviews.keySet())
System.debug(key + ', ' + listviews.get(key));
```

refreshSingleListView(String objectName, String listViewName)

Parameters objectName - the API name of the object that will have its list views refreshed.
listViewName - the API name of the list view that will be refreshed.

Response String - "success" indicating the result of the refresh request.

- The standard list view for the provided object name and list view name is refreshed.
- An object API name must be provided.
- A listview API name must be provided.
- This method is synchronous. The list view is updated immediately.
- If the method fails a simpli_lv.ListViewException is thrown with the message indicating the issue.
- The SLV config cache for the running user is cleared.

Examples

```
String objectName = 'Account';
String listViewName = 'AllAccounts';
simpli_lv_ent.ListViewHelper.refreshSingleListView(objectName,
listViewName);
```

refreshObjectListViews(String objectName)

Parameters objectName - the API name of the object that will have its list views refreshed.

Response String - "success" indicating the result of the refresh request.

- All standard list views for the provided object name are refreshed.
- An object API name must be provided.
- This method is synchronous. List views are updated immediately.
- If the method fails a simpli_lv.ListViewException is thrown with the message indicating the issue.
- The SLV config cache for the running user is cleared.

Examples

```
String objectName = 'Account';
simpli_lv_ent.ListViewHelper.refreshObjectListViews(objectName);
```

refreshAllListViews()

Parameters None

Response String - the batch job Id of the list view refresh batch job.

- All standard list views are refreshed.
- This method is asynchronous. The list views are processed as a batch job.
- The returned job id can be used to keep track of the status of the batch job.
- If the method fails a simpli_lv.ListViewException is thrown with the message indicating the issue.
- The SLV config cache for the running user is cleared.

Examples

```
String jobId = simpli_lv_ent.ListViewHelper.refreshAllListViews();
ApexJob job = [[SELECT Status, NumberOfErrors, JobItemsProcessed,
ApexClass.Name FROM AsyncApexJob WHERE Id = :jobId];
```

refreshChangedListViews()

Parameters None

Response String - "success" indicating the result of the refresh request.

- The last changed standard list view by the running user is refreshed.
- This method is synchronous. The list view is updated immediately.
- If the method fails a simpli_lv.ListViewException is thrown with the message indicating the issue.
- The SLV config cache for the running user is cleared.

Examples

```
simpli_lv_ent.ListViewHelper.refreshChangedListViews();
```

exportJSON()

Parameters None

Response String - a JSON string containing all custom list views, list view configs for the returned list views and all list view actions.

- Method to export all custom list views, configuration and actions to a JSON string.
- Currently no standard list views are exported. This may change in future releases.
- For more advanced filtering on entities that are exported use the below methods.
- The JSON string is compatible with the importJSON() method and can be used directly.

Examples

```
String jsonStr = simpli_lv_ent.ListViewHelper.exportJSON();
System.debug(jsonStr);
```

importJSON(String importJSON)

Parameters importJSON - the JSON string containing the entities to be imported.

Response String - a colon and semi-colon delimited string holding the success for each entity imported. See notes.

- Method to import all custom list views, configurations and actions from a provided JSON string.
- The input JSON string can be created using the exportJSON() method and can be used directly.

Examples

```
String jsonStr = simpli_lv_ent.ListViewHelper.exportJSON();
String result = simpli_lv_ent.ListViewHelper.importJSON(jsonStr);
```

```
System.debug(result);

//Example output : List Views:12:2;Configs:12:2;Actions:7:0;
//Above example output indicates the following -
// 12 list views successfully added, 2 failed with errors
// 12 list view configs successfully added, 2 failed with errors
// 7 actions successfully added, 0 failed
```

getListViewQuery(String objectName, String listViewName)

Parameters objectName - the API name of the object that the list view is created for. listViewName - the API name of the list view that the query will be pulled from.

Response String - the SOQL query for the given object and list view.

- This method pulls the base SOQL query for a given list view.
- The method works for both standard and custom list views.
- For more advanced queries which include offsets, sorting, joining and text search use the more advanced methods.

Examples

```
String objectName = 'Account';
String listViewName = 'AllAccounts';
String query = simpli_lv_ent.ListViewHelper.getListViewQuery(objectName,
listViewName);
System.debug('The query - ' + query);

//Example output : The query - SELECT NAME, SITE, BILLINGSTATE, PHONE,
TOLABEL(TYPE), OWNER.ALIAS, ID, CREATEDDATE, LASTMODIFIEDDATE,
SYSTEMMODSTAMP, OWNER.ID, OWNERID, ISDELETED FROM ACCOUNT USING SCOPE
Everything ORDER BY Name ASC NULLS LAST, Id ASC NULLS LAST LIMIT 100
```

getListViewColumns(String objectName, String listViewName)

Parameters objectName - the API name of the object that the list view is created for. listViewName - the API name of the list view that the columns data will be pulled from.

Response List<Map<String, String>> - the column data for the given object and list view.

- Method that returns column information for the provided object and list view.
- The method works for both standard and custom list views.

Examples

```
String objectName = 'Account';
String listViewName = 'AllAccounts';
List<Map<String, String>> columns =
simpli_lv_ent.ListViewHelper.getListViewColumns(objectName, listViewName);
System.debug(JSON.serialize(query));

//Example output :

[
{
```

```
"type": "string",
"columnWidth": null,
"name": "Name",
"label": "Account Name"
},
{
"type": "phone",
"columnWidth": null,
"name": "Phone",
"label": "Account Phone"
},
{
"type": "picklist",
"columnWidth": null,
"name": "Type",
"label": "Account Type"
},
{
"type": "string",
"columnWidth": null,
"name": "Owner.Alias",
"label": "Alias"
}
]
```

clearCache()

Parameters None
Response None

- Clears all cache for the entire Simpli List Views app.

Examples

```
simpli_lv_ent.ListViewHelper.clearCache());
```

ListViewConfigHelper

updateOrgWideParam(String paramName, String value)

Parameters paramName - the API name of the setting to be updated. value - the value the setting should be set to. Note the value must be a string.
Response None

- The setting API names can be found at HQ -> Documentation ->

Examples

```
simpli_lv_ent.ListViewConfigHelper.updateOrgWideParam('Debug', 'true');
//the example above turns on debugging
```

updateOrgWideParams(Map<String, String> newParams)

Parameters newParams - a map containing the API name and value pairs of the settings to be updated.

Response None

- The setting API names can be found at HQ -> Documentation ->

Examples

```
Map<String, String> params = new Map<String, String>();
params.put('Debug', 'true');
params.put('AllowAdmin', 'false');
simpli_lv_ent.ListViewConfigHelper.updateOrgWideParams(params);

//the example above turns on debugging and stops the admin cog icon from
being available to users
```

getOrgWideParam(String paramName)

Parameters paramName - the API name of the setting to be retrieved.

Response String - the value of the setting

- The setting API names can be found at HQ -> Documentation ->

Examples

```
String setting =
simpli_lv_ent.ListViewConfigHelper.getOrgWideParam('Debug');
System.debug('The setting is - ' + setting);

//the example above gets the debug setting value
```

getOrgWideParams()

Parameters None

Response Map<String, String> - a map containing the API name and value pairs of all org wide settings.Notes

- The setting API names can be found at HQ -> Documentation ->

Examples

```
Map<String, String> params =
simpli_lv_ent.ListViewConfigHelper.getOrgWideParams();

//this example gets all org wide params
```

LWC Component APIs

Code Examples

All event interactions with the **<simpli_lv-simpli-u-i-list-views>** component are performed through an API. Events are fired when the user interacts with the component. Events can also be sent to the component for processing. All event requests and notifications occur in a very similar pattern. The general HTML component looks like the following:

LWC HTML Page

```
<div>
   <simpli_lv-simpli-u-i-list-views
         mode="App Page"
         page-name="My Component"
         oneventresponse={handleEventResponse}
         use-message-channel=true>
   </simpli_lv-simpli-u-i-list-views>
</div>
```

- Events are only available in Simpli List Views Enterprise.
- The page name is required and used as an id if using events and multiple SLV components are on one page. Think of the page name as a component name. (see examples)
- oneventresponse is the handle to any events fired by the component. In this example handleEventResponse() is the function which handles the response
- It is important to set the use-message-channel property so that events are processed by the SLV component

LWC Controller

```
import { LightningElement, wire, track } from 'lwc';
export default class TestController extends LightningElement {

 //-------------------------------------------------------------------
 //USE THIS - use this method if there is a single SLV component on the
 //              page to handle OUTGOING requests to the component
    //-------------------------------------------------------------------
    handleEventRequest() {
        const slv = this.template.querySelector('simpli_lv-simpli-u-i-
list-views');
        let evt = { type: 'refreshData' };
        slv.eventRequest(evt);
    }

    //-------------------------------------------------------------------
    //OR THIS - use this method if there are multiple SLV components on
the page to
 //             handle OUTGOING requests to the component
    //-------------------------------------------------------------------
    handleEventRequestWithMultipleComponents() {
        const slvs = this.template.querySelectorAll('simpli_lv-simpli-u-i-
list-views');
```

```
        let slv = null;

        slvs.forEach(tmpslv => {
          if (tmpslv.uniqueComponentId.includes("My Component")) {
            slv = tmpslv;
          }
        });
        let evt = { type: 'refreshData' };
        slv.eventRequest(evt);
    }

    //--------------------------------------------------------
    //used to handle INCOMING events fired from the component
    //--------------------------------------------------------
    handleEventResponse(event) {
        console.log('Event Response - ' + JSON.stringify(event.detail));
    }

    }
```

- The handleEventRequest() method gets the component using a querySelector() assuming only one SLV component exists on the page.
- The handleEventRequestWithMultipleComponents() method gets the component using a querySelectorAll() assuming multiple SLV components exist on the page.
- The event request is then created. In this example we are setting the event type to "refreshData".
- Any response to an event is handled by the handleEventResponse(event) method. All data for the event being fired is stored in the event.detail property.

Incoming Events

The following incoming events are fired on user interactions. In some cases data is returned in the response. This is noted in those cases.

- **rendering** - fired when the component finishes rendering
- **standAloneRendering** - fired when the component is in Stand Alone mode and the data finishes rendering
- **objectSelected** - fired when an object is selected
- **listViewSelected** - fired when a list view is selected
- **urlClicked** - fired when a url is clicked. The URL is returned in the event response
- **processSingleListView** - fired when a single list view is reprocessed
- **processObjectListViews** - fired when an objects list views are reprocessed
- **processAllListViews** - fired when all list views are reprocessed
- **dataSaved** - fired when the user saves data

The following events are based on user interactions as well as event requests made to the component via code. The event response details for these events can be found in the corresponding event request API docs below.

- refreshData
- refreshActions
- refreshObjects
- refreshListViews
- refreshOnce
- autoRefreshOn
- autoRefreshOff
- downloadData
- downloadSelectedData
- pinListView
- unpinListView
- spinnerOn
- spinnerOff
- showAdmin
- runAction

Incoming Event Handler Example Code

```
handleEventResponse(event) {
    let resp = event.detail;
    console.log('All Event Response Details - ' + JSON.stringify(resp));
    console.log('Event Response Status - ' + resp.status);
    console.log('Event Response Type - ' + resp.type);
    console.log('Event Response Count - ' + resp.count); //if the event
deals with a list. i.e. refreshData, downloadData, refreshObjects
    console.log('Event Response Object - ' + resp.object);
    console.log('Event Response ListView - ' + resp.listView);
    console.log('Event Response URL - ' + resp.url); //if a URL was
clicked
    console.log('Event Response Data - ' + resp.data); //if data was
downloaded
}
```

Outgoing Event Requests

- refreshData
- refreshActions
- refreshObjects
- refreshListViews
- refreshOnce
- autoRefreshOn
- autoRefreshOff
- downloadData
- downloadSelectedData
- pinListView
- unpinListView
- spinnerOn

- spinnerOff
- showAdmin
- runAction
- updateSetting

refreshData

Request { type: 'refreshData' }

Response {type: "refreshData", status: "started"} - on starting refresh
{type: "refreshData", status: "finished", count: rowCount} - on finishing refresh

- Event to refresh the data currently being displayed in the list view.
- This request is fairly unique in that two events are fired by the component. Both a start and finish message

Examples

```
let evt = { type: 'refreshData' };
this.template.querySelector('simpli_lv-simpli-u-i-list-
views').eventRequest(evt);
```

refreshActions

Request { type: 'refreshActions' }

Response {type: "refreshActions", status: "finished"}

- Event to update the actions available in the action drop down.
- Typically the actions are refreshed after a data refresh.

Examples

```
let evt = { type: 'refreshActions' };
this.template.querySelector('simpli_lv-simpli-u-i-list-
views').eventRequest(evt);
```

refreshObjects

Request { type: 'refreshObjects' }

Response {type: "refreshObjects", status: "finished", count: objectCount }

- Event to update the objects available in the objects drop down

Examples

```
let evt = { type: 'refreshObjects' };
this.template.querySelector('simpli_lv-simpli-u-i-list-
views').eventRequest(evt);
```

refreshListViews

Request { type: 'refreshListViews' }

Response {type: "refreshListViews", status: "finished", count: listViewCount, object:

objectName}

- Event to update the list views in the list view drop down
- The event request uses the selected object to determine the retrieved list views.

Examples

```
let evt = { type: 'refreshListViews' };
this.template.querySelector('simpli_lv-simpli-u-i-list-
views').eventRequest(evt);
```

refreshOnce

Request { type: 'refreshOnce' }
Response {type: "refreshOnce", status: "finished"}

- Event to refresh the data. This is similar to using the refreshData event.

Examples

```
let evt = { type: 'refreshOnce' };
this.template.querySelector('simpli_lv-simpli-u-i-list-
views').eventRequest(evt);
```

autoRefreshOn

Request { type: 'autoRefreshOn' }
Response {type: "autoRefreshOn", status: "finished"}

- Event to turn auto refreshing on for the component.
- The rate of refresh used is whatever is provided in the list view config.

Examples

```
let evt = { type: 'autoRefreshOn' };
this.template.querySelector('simpli_lv-simpli-u-i-list-
views').eventRequest(evt);
```

autoRefreshOff

Request { type: 'autoRefreshOff' }
Response {type: "autoRefreshOff", status: "finished"}

- Event to turn auto refresh off for the component.

Examples

```
let evt = { type: 'autoRefreshOff' };
this.template.querySelector('simpli_lv-simpli-u-i-list-
views').eventRequest(evt);
```

downloadData

Request { type: 'downloadData' }

Response {type: "downloadData", status: "finished", data: csvData}

- Event which is similar to clicking the download data hyperlink on the list view.
- The event response contains the CSV data string.

Examples

```
let evt = { type: 'downloadData' };
this.template.querySelector('simpli_lv-simpli-u-i-list-
views').eventRequest(evt);
```

downloadSelectedData

Request { type: 'downloadSelectedData' }
Response {type: "downloadSelectedData", status: "finished", data: csvData}

- Event which is similar to clicking the selected data download hyperlink
- The event response contains the CSV data string.

Examples

```
let evt = { type: 'downloadSelectedData' };
this.template.querySelector('simpli_lv-simpli-u-i-list-
views').eventRequest(evt);
```

pinListView

Request { type: 'pinListView' }
Response {type: 'pinListView', status: 'finished', object: selectedObject, listView: selectedListView}

- Event to pin the currently viewed list view

Examples

```
let evt = { type: 'pinListView' };
this.template.querySelector('simpli_lv-simpli-u-i-list-
views').eventRequest(evt);
```

unpinListView

Request { type: 'unpinListView' }
Response {type: 'unpinListView', status: 'finished', object: selectedObject, listView: selectedListView}

- Event to unpin the currently viewed list view.

Examples

```
let evt = { type: 'unpinListView' };
this.template.querySelector('simpli_lv-simpli-u-i-list-
views').eventRequest(evt);
```

spinnerOn

Request { type: 'spinnerOn' }
Response None

- Event to turn the components spinner on.
- The spinner will continue to be displayed until it is turned off and will not allow access to the list view during this time.

Examples

```
let evt = { type: 'spinnerOn' };
this.template.querySelector('simpli_lv-simpli-u-i-list-
views').eventRequest(evt);
```

spinnerOff

Request { type: 'spinnerOff' }
Response None

- Event to turn the components spinner off.

Examples

```
let evt = { type: 'spinnerOff' };
this.template.querySelector('simpli_lv-simpli-u-i-list-
views').eventRequest(evt);
```

showAdmin

Request { type: 'showAdmin' }
Response None

- Event similar to clicking the cog icon. The list views config admin modal dialog is displayed.

Examples

```
let evt = { type: 'showAdmin' };
this.template.querySelector('simpli_lv-simpli-u-i-list-
views').eventRequest(evt);
```

runAction

Request { type: 'runAction', name: 'actionAPIName' }
Response {type: "runAction", status: "finished"}

- Event to run an action which is specified in the request data.
- No data can currently be passed in to the request other than the action name.
- If rows are selected on the list view those rows will be passed into the action.

Examples


```
let evt = { type: 'runAction' };
this.template.querySelector('simpli_lv-simpli-u-i-list-
views').eventRequest(evt);
```

updateSetting

Request { type: 'updateSetting', name: 'settingAPIName', value: 'settingValue' }
Response None

- The setting API names are specified in the HQ -> Documentation ->

Examples

```
let evt = { type: 'updateSetting' };
this.template.querySelector('simpli_lv-simpli-u-i-list-
views').eventRequest(evt);
```