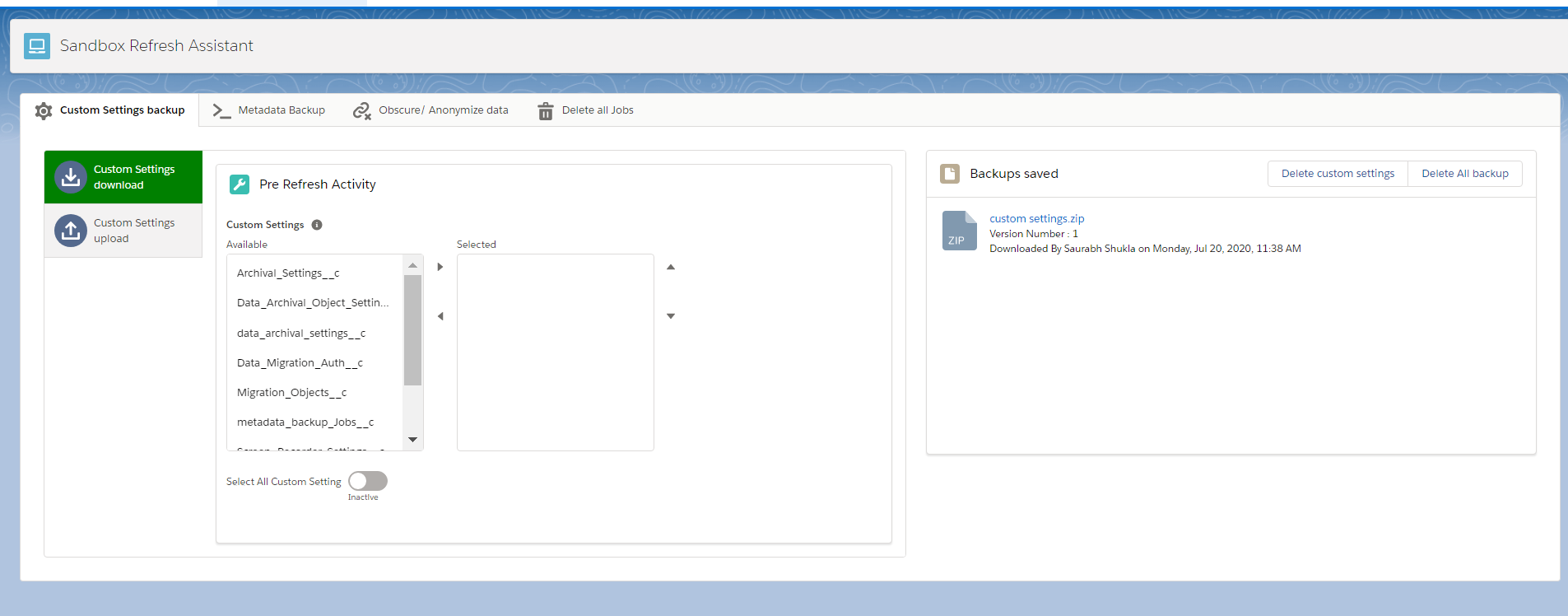
**Sandbox Refresh Assistant**

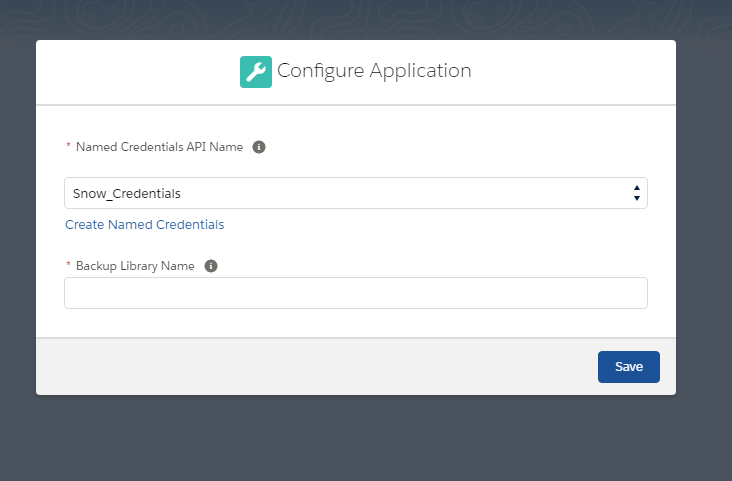
* When a sandbox refresh happens in salesforce, we will have to do some pre refresh and post refresh activities.
* These activities include taking backup of metadata, custom setting records, anonymizing customer data after refresh, data loading custom setting records and aborting all scheduled jobs.
* All these activities are time consuming and have a high risk of data loss if done manually

Ex: custom setting records need to be queried individually and stored in a local disk and data loaded back again from the local disk. This can take a significant amount of time for a support resource, also can have issues while comparing the existing data with the backup.



* We Built an application that allows an admin to take backup of all the custom setting records inside a zip file and push it into production where it can be safe until the refresh happens where it will come back to the sandbox.
* The admin can use this zip file to update all the custom setting records directly or can also perform a conflict with the existing settings and merge/ edit the backup before loading the data.
* This also includes a tab to take backup for all metadata and deploy it to the sandbox post refresh.
* A data anonymizing tool is included which will allow users to select the object and fields that contains personal information and fill those with junk. Upto 5 selections are saved as user preferences for future use.
* The application has a tab to abort all the running jobs in sandbox to avoid any automated field updates and emails to be triggered.

**Initial Configuration Components:**



This is shown when the app is opened for the first time to configure the named credentials that will be used for connecting to target org and to specify the backup library API name from the target org.

**Components used:**

**Lightning components:**

getAuthComponent – shows the popup for configuring the application.

AuthSuccessEvent- called when authentication is successful and will be handled by sandboxRefreshAppComponent.

**Objects:**

Data\_Migration\_Auth\_\_c custom setting:

Created during initial setup after selecting the named credential to be used for integration.

|  |  |
| --- | --- |
| **Field Name** | **Purpose** |
| Name | To hold the unique value. Should be ‘refresh’ |
| Named\_credentials\_\_c | To hold the named credentials api name to make callouts. This should be the named credential that connects and authenticates to the target org. |

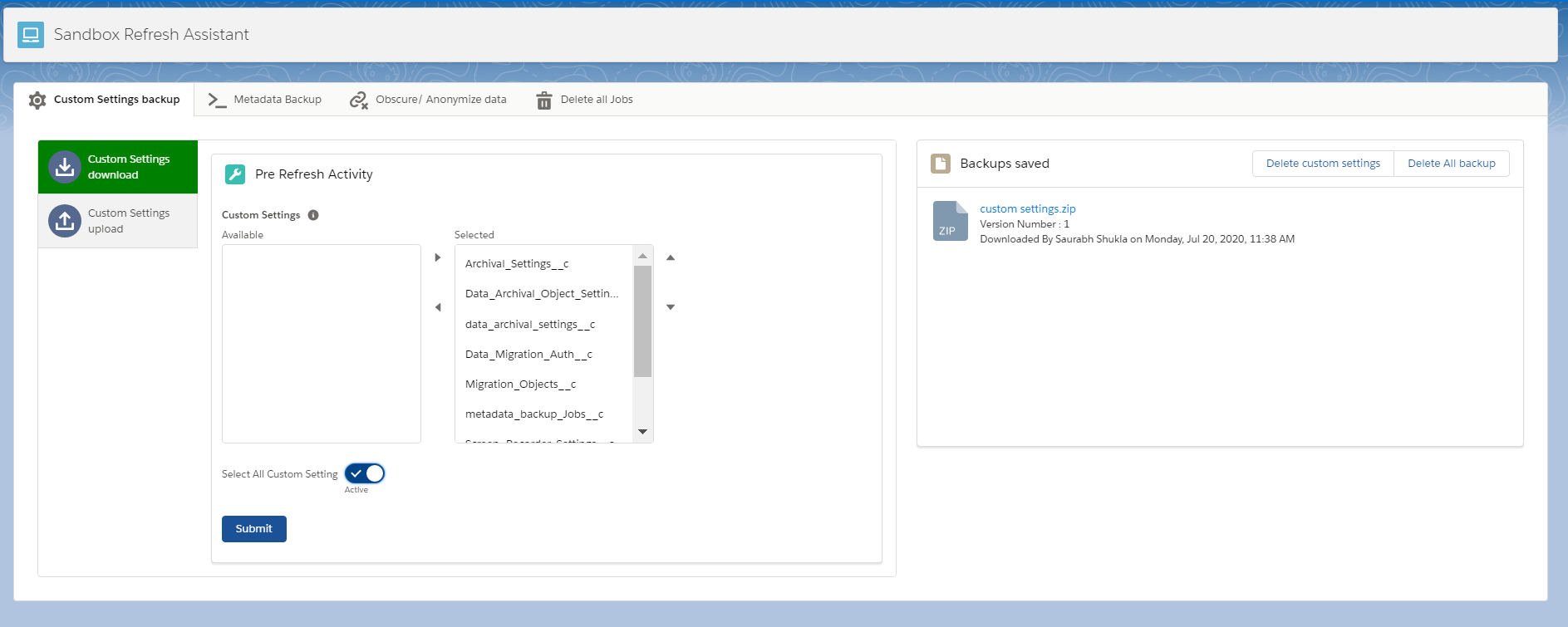
**Classes:**

mainApplicationController – Used during the initial application setup to get the named credentials in the org so the users can select the desired one and will be saved in the Data\_Migration\_Auth\_\_c custom setting record with the record Name ‘**refresh’.** Also checks if the current org is prod or sandbox.

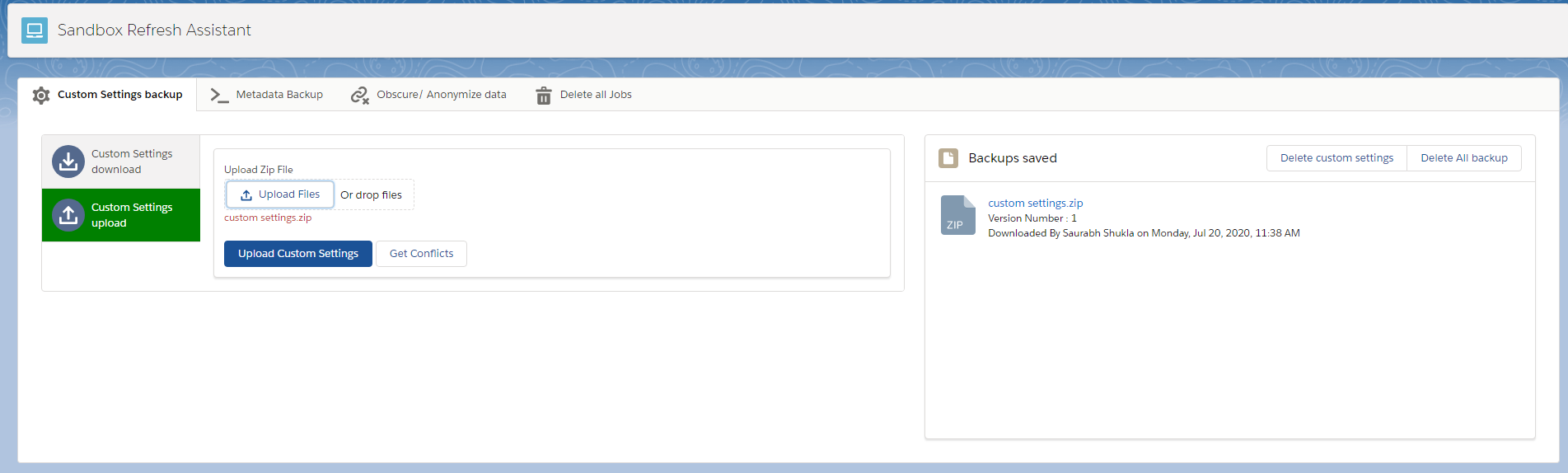
**Tab 1 – Custom Setting Backup**

In this **pre refresh activity**, users are allowed to take a backup of all custom settings records into a zip file containing the records of each custom setting object in a CSV. When users submit the form after selecting the custom setting objects the screen will show a message indicating that a backup job is in progress. At this point users can’t do anything in the ‘Custom settings download’ tab. Below is the example of zip file we get after taking backup in the ‘backup saved’ section which users can download.

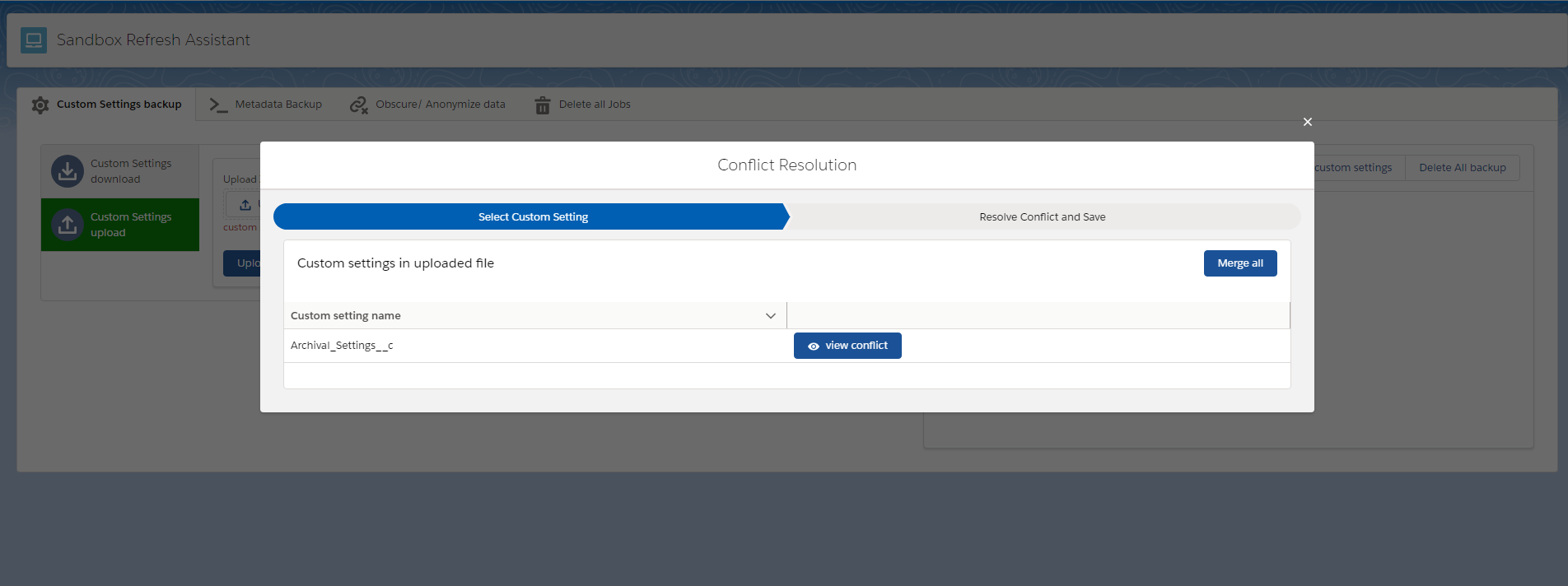




In this **post refresh activity**, tab name is ‘custom settings upload’ where users can upload the zip file and can perform a bulk update of all records in the custom settings inside the zip file.



There is another option that allows users to perform a conflict resolution where users can open each csv file inside the zip and save the required records individually instead of updating everything.



**Components used:**

customSettingBackupComponent – hosts the tabs for custom setting download, upload and the backups saved section.

customSettingDownloadComponent- submits the job for taking backup of selected custom setting object records.

customSettingUploadComponent- used for submitting the job to update the custom setting records in the zip file containing the CSV files for each custom setting object. It also provides a button to the user to perform a comparison of existing and new values for the custom setting records and save them individually.

customSettinglist- shows an overlay which shows a path. First item in path will display the datatable with list of all files contained inside the uploaded zip file. The files inside the zip files should be CSV and have the file name same as the custom object api name.

The next will show the customSettingCompareRecords component which opens the CSV file inside the zip and displays records in the table.

customSettingCompareRecords –opens the CSV file and displays records in the table using pagination.

tableCellComponent – displays each table cell based on the field type and also if it is editable. Fields for existing record values are read only and for the values from CSV are editable.

customSettingSaved – This shows the backups saved section and displays the backup files from the target org and provides a download link in the app. Users can also delete the custom settings only or delete every backup related to this org from the target.

If the file size Is exceeded, then fileSizeEvent is triggered which will be handled by other components to hide the job submission functionality.

reloadBackupEvent- is triggered from any component when backup files need to be reloaded.

sandboxRefreshAppComponent – listens to the authSuccessEvent and displays the tool if success. It also hides the tool if the org is a production org.

mainAppHeader – displays the App header with the title, icon and theme in a standalone SandboxRefreshAssistant App.

**Classes:**

customSettingBkpController –

controller class for custom settings backup section.

Methods:

deleteSavedData- makes a callout to /services/apexrest/saveZipFile/ with DELETE method to delete the backup files from target org based on the ‘objects’ parameter. If this is ‘custom settings’ then only custom setting backup are deleted. If it is ‘metadata’ then metadata backups are deleted. And if it is ‘all’ then every backup file is deleted from target org.

We send a parameter called orgName which will be the current sandbox url which will be used to delete only the backups that were created from this sandbox.

isJobRunning – checks if the readZipQueueable or createZipQueueable jobs are running. If it returns false, then the sections are hidden to avoid submitting more jobs.

CSList – gets a list of all custom settings in the org to display in the dual list box.

submitJob – submits createZipQueueable job to get the backup of all selected custom setting object records.

getSaved – makes a callout to /services/apexrest/saveZipFile/ with GET method which returns the file and its downloadable links from target org to be displayed in backups section based on the ‘objects’ parameter. If this is ‘custom settings’ then only custom setting backup are returned. If it is ‘metadata’ then metadata backups are returned.

We send a parameter called orgName which will be the current sandbox url which will be used to get only the backups that were created from this sandbox.

saveFile – used during the custom settings upload which will be called from lightning component to upload the zip file and enqueues the readZipQueueable class to update the records from the CSV file contained in the zip file.

getFileNames – gets all file names contained in the uploaded zip file.

mergeAllSettings – called from the conflict resolution component which basically merges all the records from the zip file into the org except for the ones that were already merged.

customSettingFileWrapper- wrapper class for file name and merge status for the files while conflict resolution

returnBackup – wrapper class for returning the backup file links from target org to the lightning component.

createZipQueueable –

creates a zip file of all the records for the custom setting objects selected. It will first remove the first element from the selected object list, queries the records, creates a comma seperated string with record values and creates a zip file using the **Zippex** utility class. The CSV file is now added to this zip file using addFile method.

The remaining custom setting objects are now passed to another instance of createZipQueueable (chaining). When all objects are processed, we make a callout to the target org to ‘services/apexrest/saveZipFile’ to save the file in the target org.

readZipQueueable –

accepts the zipfile and the file names inside the zip file. It will first remove the first element from the the list of file names and then gets the CSV file from the zip file.

We then upsert the records based on the custom setting Name field since it is unique**. Id is not being used for update only Name field is used.** If the record size is more than 10000, records are inserted/ updated from sobjectDMLBatch class else it is inserted/ updated directly.

sobjectDMLBatch-

Used for inserting / updating custom setting records if the count is more than 10000.

ConflictResolutionController-

Used for getting the records from the zip file using the file name which should be the custom setting api name.

Methods-

getObjectFields – returns a list of fieldWrapper which consists of fieldname, field type and if the field is updateable. All this information is got from Schema and returned after adding it into wrapper.

getConflicts- we send the zipFile in base64Format and the objectAPI name of the custom setting. The CSV file is got from the zip file and the same logic as readZipQueueable is used to get the records from the zip file and return them to the lightning component as a wrapper containing the current org record and the record from the backup file.

saveMergedRecords – after merging the records from the table in the lightning component, we save the changed records in this method following the same logic as readZipQueueable.

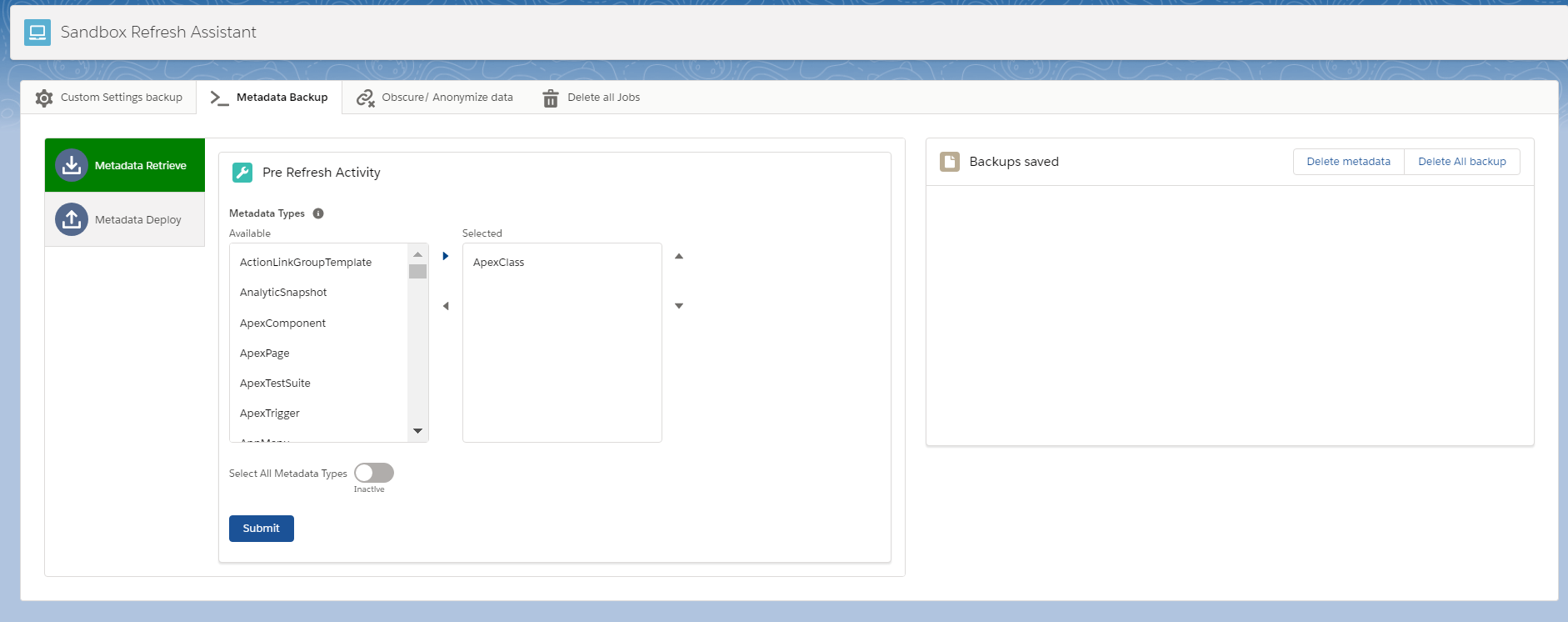
**Zippex, Puff and HexUtil** are utility classes for creating zip files. Go to the below link in github for more information the open source apex code regarding methods, variables etc.

<https://github.com/pdalcol/Zippex>

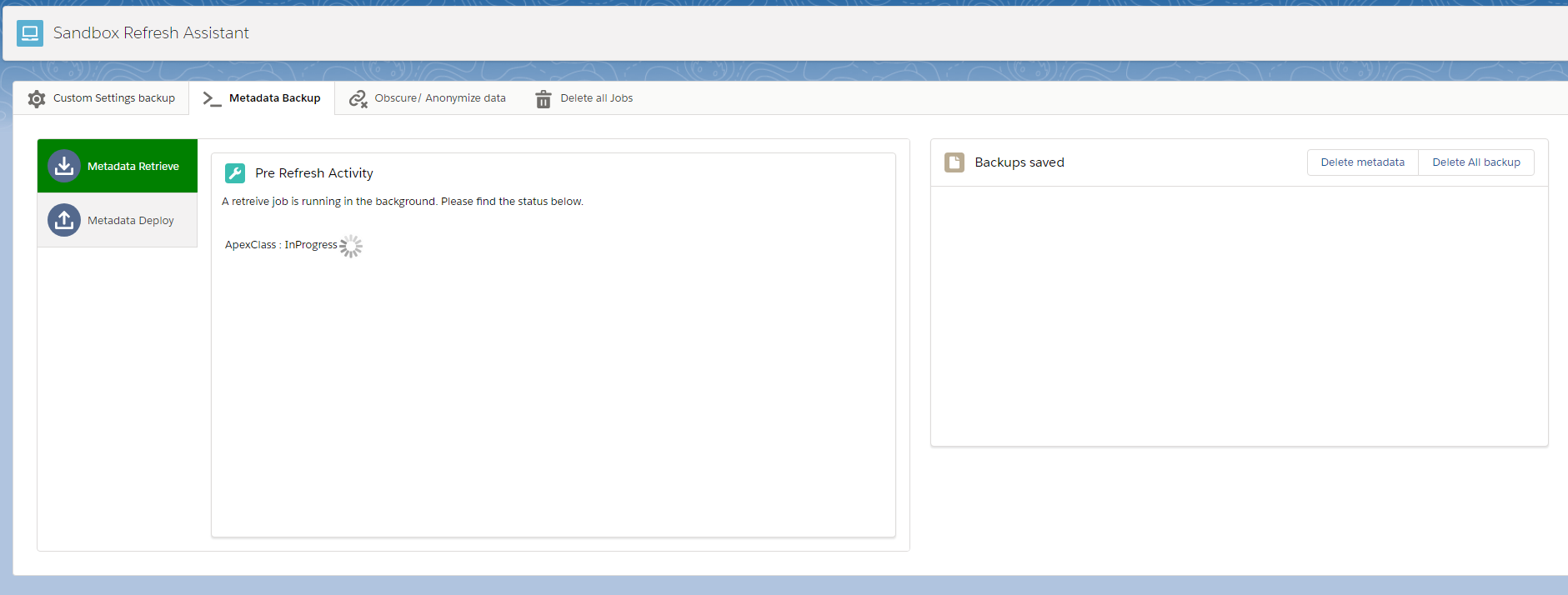
**Tab 2 – Metadata Backup**

In this **pre refresh activity**, users are allowed to take a backup of all the metadata from the org and store the backup in the target org in the form of a zip file in a content library. The backup files stored in target are shown in the backups saved section. This section has 2 vertical tabs which are for download and for deployment.

In the download tab users can select the metadata object to download and submit the job.

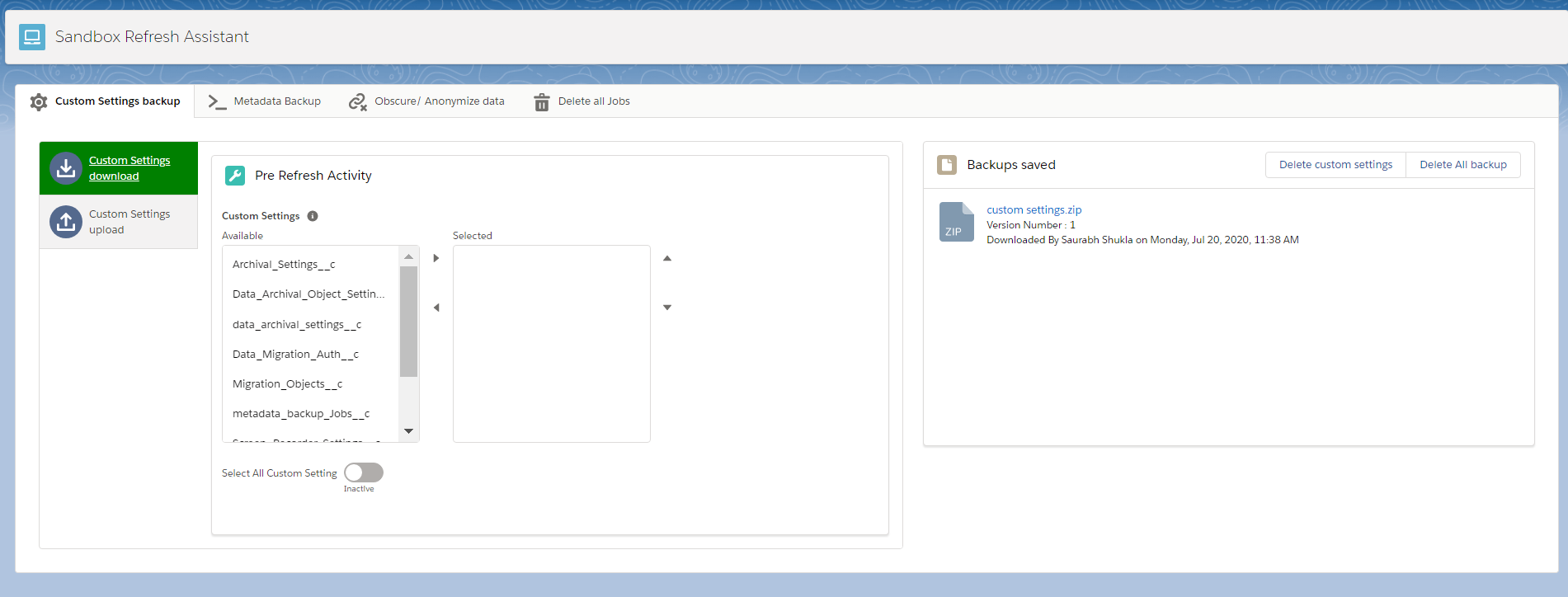


Once submitted we get the live status of retrieval as shown below.

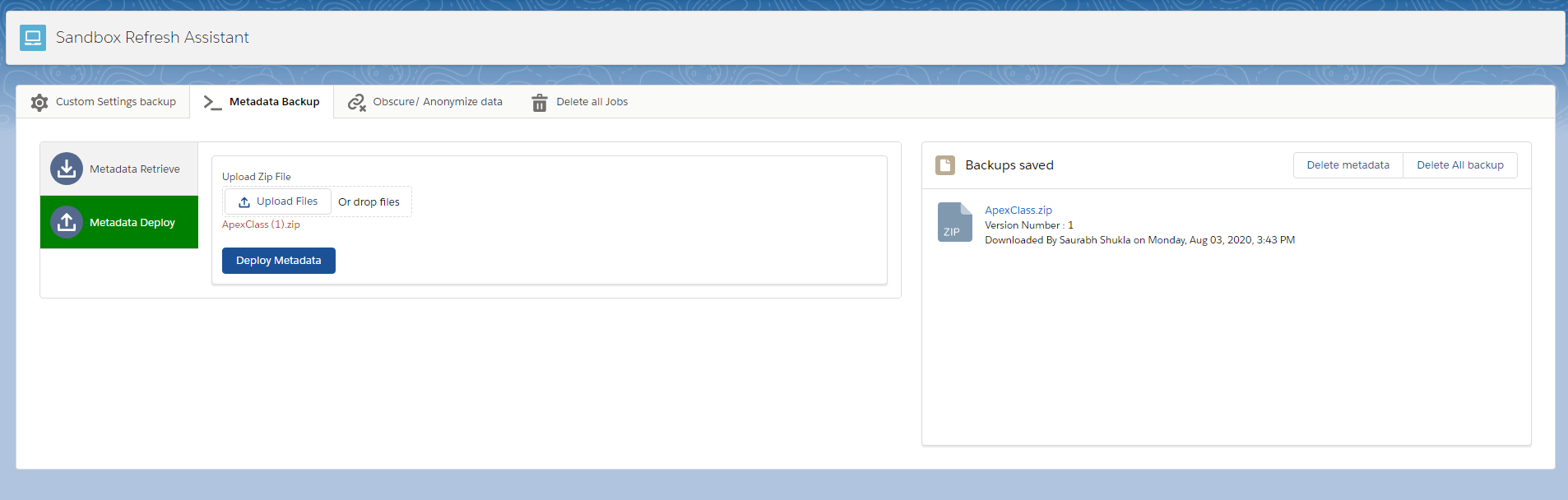


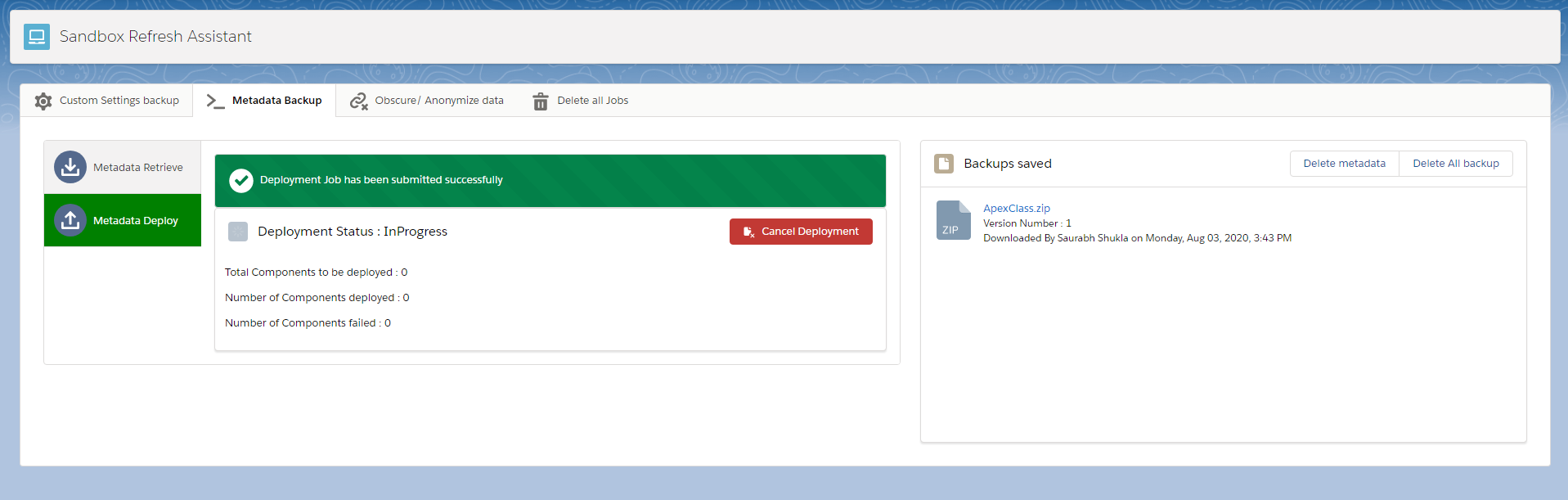
After the job is successful, we can see the zip file for ApexClass in the backups saved section. Unlike custom setting backup, the files are stored seperated for each metadata object.

Through this we can take backup for all metadata objects including custom labels and custom metadata records.



In this **post refresh activity**, we can deploy the components by uploading the zip file in the deploy metadata section. Which shows the live status of deployment and allows for deployment cancellation as well.





The Backups saved section allows users to delete only metadata backup or delete all the backup for the current sandbox from the target org. (deleteSavedData and getSaved methods from customSettingBkpController are used).

**Components used:**

**Lightning components:**

metadataBackupComponent – contains the metadata backup and deploy vertical tabs along with backup saved section.

metadataDownloadComponent – This component allows user to select the metadata objects and submits the metadata Retrieve job from metadata API to get the zip file for each of the objects. It also shows the live status of the retrieve jobs are already in progress.

MetadataDeployComponent – This component allows users to upload the zip file that was created by download component which has the package.xml and component files. Once the job for deployment in metadata API is submitted it shows the deployment status along with cancel deployment button.

**VF Page:**

SessionIdPage – used to get the session Id in lightning context since userInfo object doesn’t provide the session Id when called in lightning. Used to use the metadata API to deploy components, retrieve, check deployment status, check retrieve status and cancel deployment.

**Objects:**

metadata\_backup\_Jobs\_\_c custom setting:

These records are created for tracking the metadata backup and retrieve jobs and deleted after the jobs are completed.

|  |  |
| --- | --- |
| **Field Name** | **Purpose** |
| Name | Metadata Object Name |
| job\_type\_\_c | Deploy or retrieve depending on the action |
| Job\_Id\_\_c | Metadata retrieve or deploy job Id |
| Boolean\_Field\_Test\_\_c | Used in test class for creating test records for covering all field types logic in test classes. |
| date\_Field\_Test\_\_c | Used in test class for creating test records for covering all field types logic in test classes. |
| datetime\_field\_test\_\_c | Used in test class for creating test records for covering all field types logic in test classes. |
| number\_field\_Test\_\_c | Used in test class for creating test records for covering all field types logic in test classes. |

**Classes:**

MetadataHandlerClass –

Used for getting the session Id from SessionIdPage VF page and also to send emails.

MetadataService –

Metadata API class generated from WSDL to do all metadata related activities.

MetadataBackupController –

Main controller for Metadata Backup section to deploy and retrieve metadata records and create the backup files in target org as a zip file.

Methods:

getMetadataJobIds – This method returns a list of jobWrapper which has the job status and metadata job Id. This is done after retrieving metadata\_backup\_Jobs\_\_c records. If there are records, then the sections are hidden

checkJobRetrieve – This method accepts a metadata\_backup\_Jobs\_\_c record and returns a jobwrapper object.

For each metadata object type this method is called repeatedly from the lightning component using **promises** and **promise.all()** function until all jobs are Completed.

This method calls the checkRetrieveStatus method from metadata API to get the retrieve status for any job Id.

If status is pending, then we return a status of ‘Progress’.

If status is Succeeded, we delete the metadata\_backup\_job\_\_c record for that metadata object, get the retrieved zip file and send it to target org by calling ‘services/apexrest/saveZipFile’ api. Library name, org name, file name, are passed in the header. File name is the metadata object name.

If status is failed, we delete the metadata\_backup\_job\_\_c record.

submitRetreiveJob – calls the metadata retrieve job by passing the metadata object name and creates metadata\_backup\_job\_\_c setting records for each object with job type as ‘retrieve’. This returns all components for that metadata type.

saveFile – calls the metadata deploy job by passing the base64 encoded zip file to the method and creates metadata\_backup\_Jobs\_\_c setting records with job type as ‘deploy’. Only one record is created for deployment, unlike retrieve job where a retrieve job is created for each metadata object.

createService – creates the metadataService.MetadataPort object to call any metadata API methods.

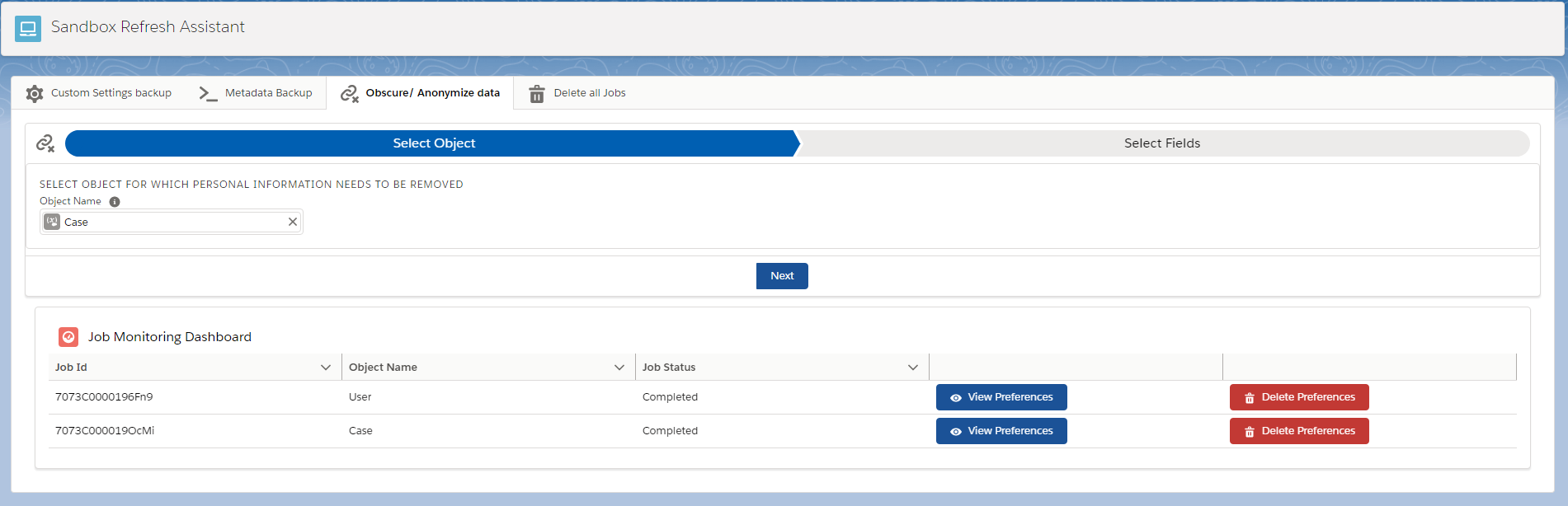
cancelDeployment – calls the metadata cancel method by passing the deploy job Id which cancels the deployment.

checkJobDeploy – This method is called repeatedly to get the status of deployment and returns the jobWrapper object with status, no of components failed, no of components deployed and no of total components.

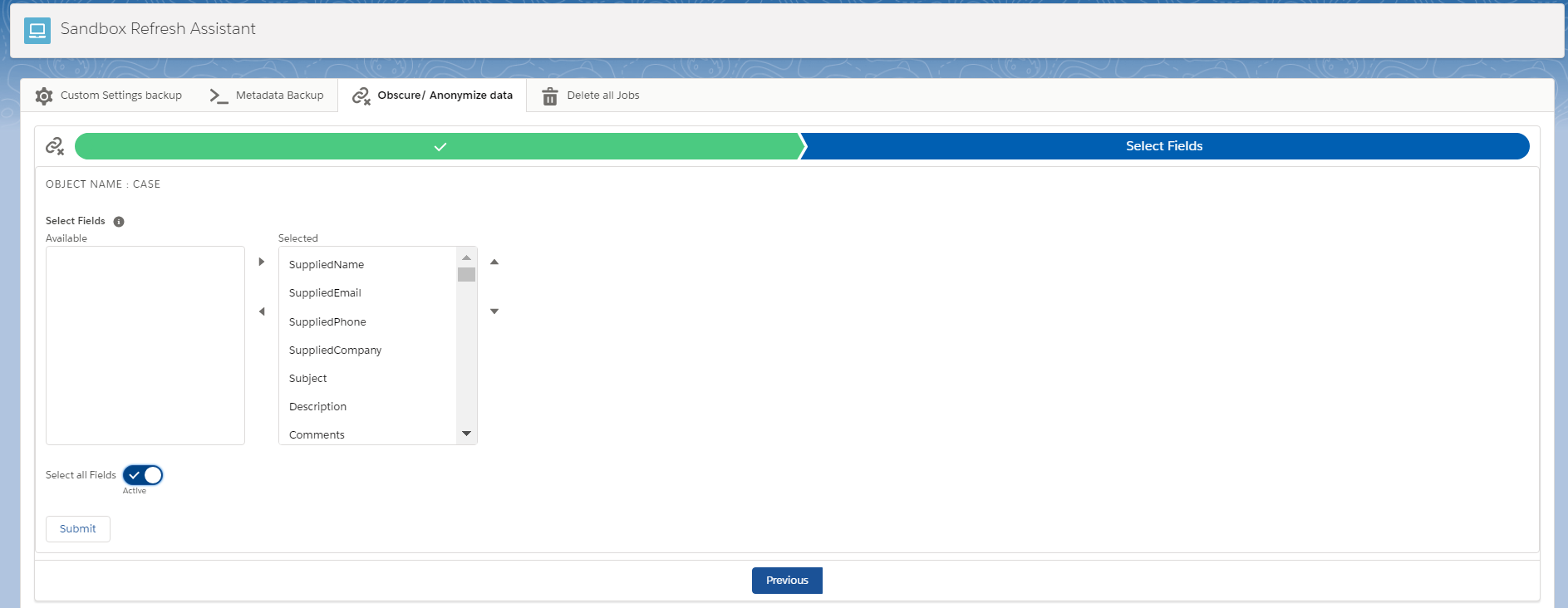
**Tab 3 – Obscure/ Anonymize data**

This is a **post refresh activity**, and allows users to obscure the customer related data from all sobject in the org.

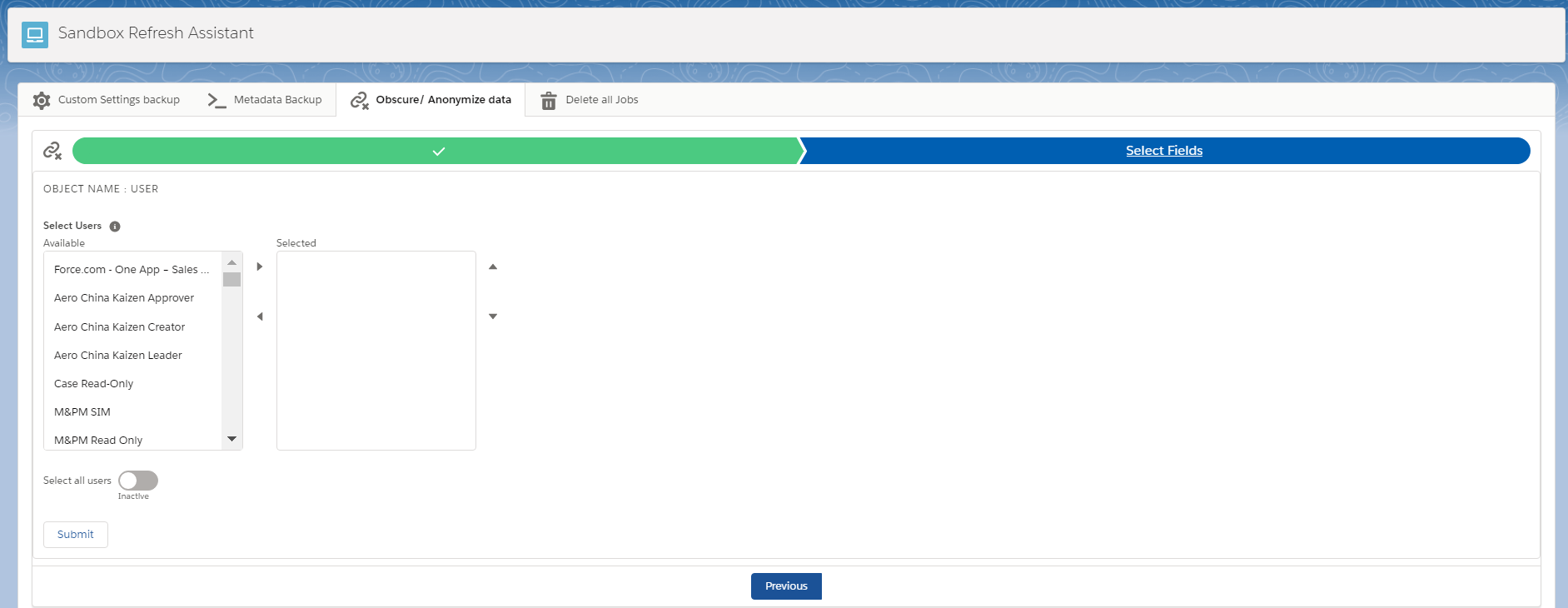
We first ask the users to select the object from which the records need to be obscured.



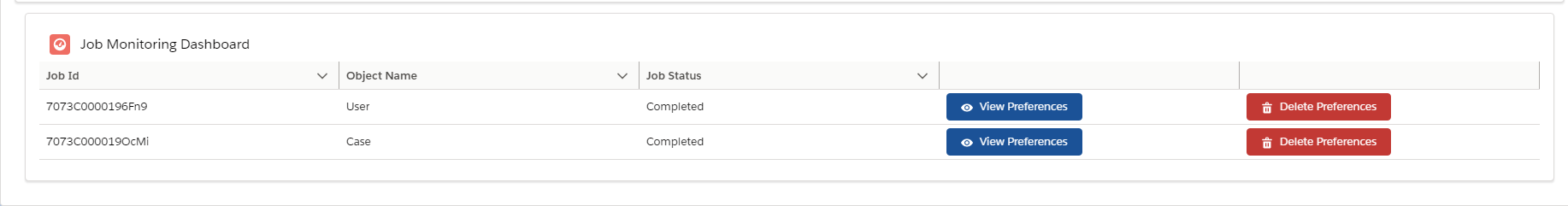
Then the user needs to select the fields where the data has to be obscured. Only text, email, number fields are allowed to be obscured.



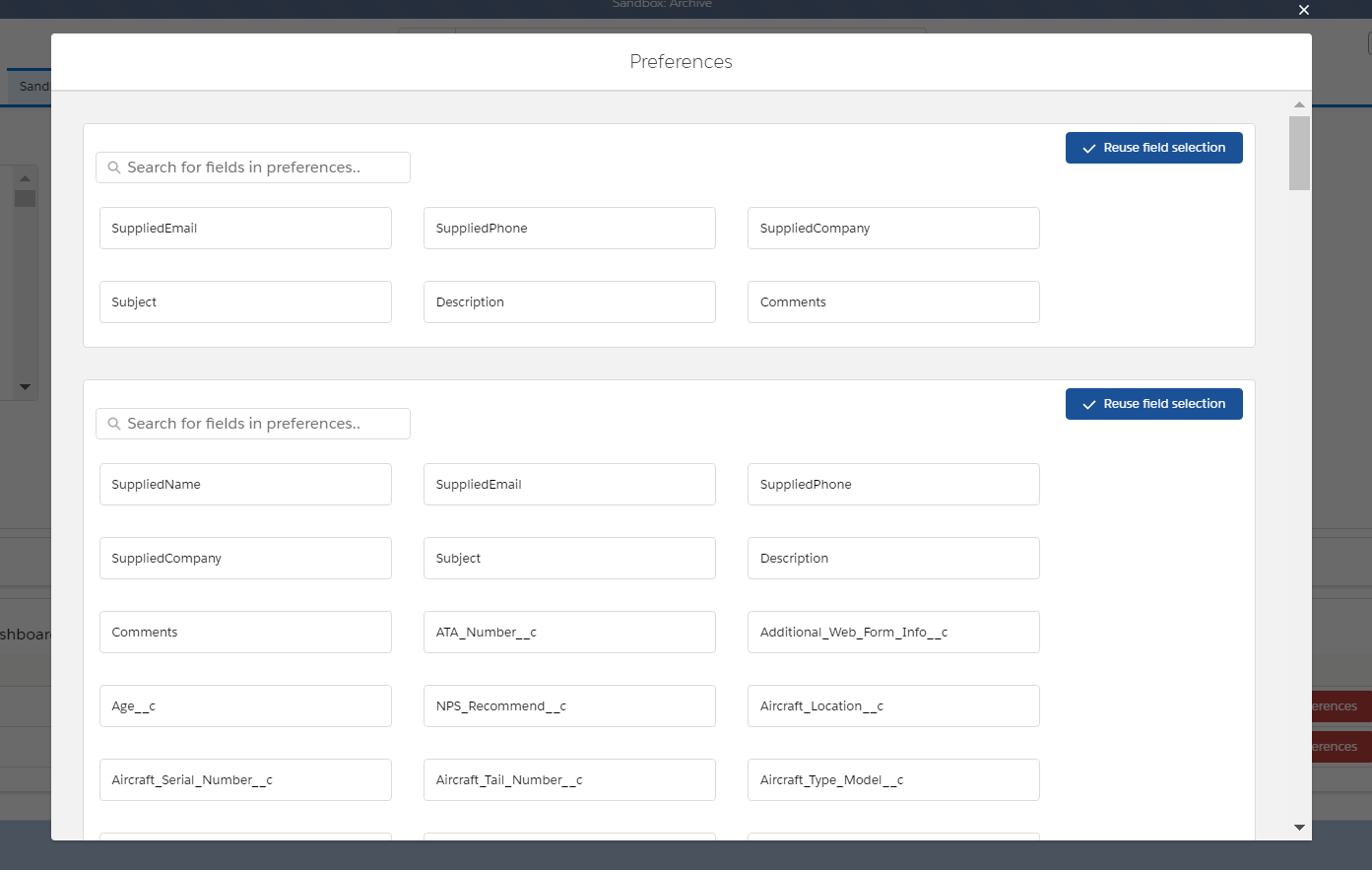
If the object selected is ‘**User**’ then the user has to select the profiles of the user records which need to be obscured.

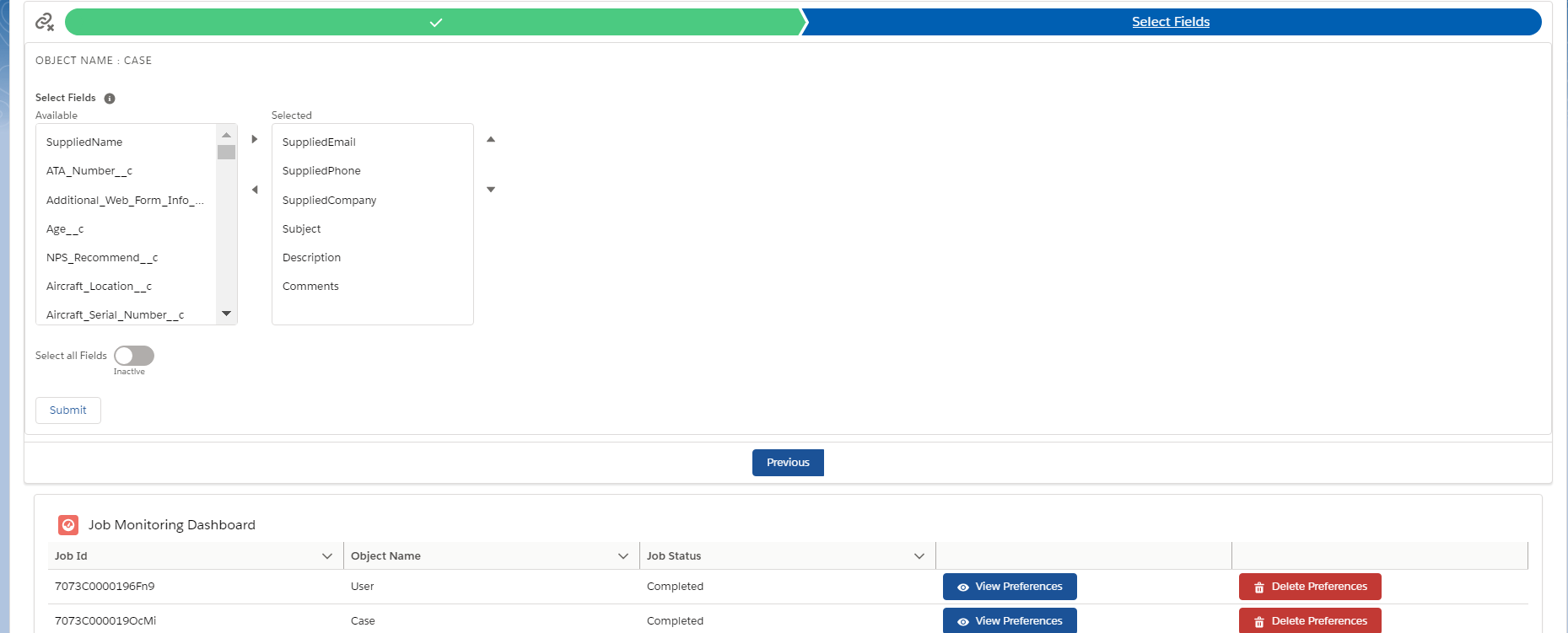


Once submitted, all the selected fields of the records of the selected object will be replaced with junk randomized values. Along with this there will be user preferences automatically created using the selection which will have the object and fields selected so that it can be reused in future. The job monitoring dashboard will show the job status along with the object names and job Id.



When users click on ‘View preferences’, there will be a modal box opened which will show all the field preferences for that object. Users can search for fields in the preferences and click on ‘Reuse Field selection’ button to auto fill the object and fields. Maximum of 5 preferences can be created for each object after which the tool asks the user to delete the preferences for that selected object using ‘Delete Preferences’ button.





**Components used:**

**Lightning components:**

personalInfoComponent – hosts the lightning:path with the object selection search box and field display component.

fieldSelectionComponent – hosts the dual list box to select user profiles when User is selected as object and object fields when any other object is selected.

searchComponent – used for searching objects

searchResults- displays the matched results as user types object names

clearValueEvent – triggered when the object selection is cleared from the field

oSelectedValueEvent – triggered when an object is selected

jobDashboardComponent - displays the live status of jobs from the custom metadata with view and delete preferences button, where the job preferences (field and object selection) for each object are stored.

fieldListModal - displays an overlay when ‘View preferences’ button is clicked. This will contain the component that displays all the field preferences for any object.

fieldDisplayComponent – displays each preference in a layout along with ‘Reuse Preferences’ button.

**Objects:**

Refresh\_setting\_met\_\_mdt custom metadata:

Each record stored contains the job id and job status for each object that has been submitted for obscure data. These records are displayed on JobDashboardComponent with view and delete preferences button

|  |  |
| --- | --- |
| **Field Name** | **Purpose** |
| DeveloperName | Object api name |
| Job\_Id\_\_c | Job Id for the job submitted for the object |
| Status\_\_c | Status of the job submitted for the object |

fieldList\_\_mdt custom metadata:

These are the field preferences for each object and has a metadata relationship with Refresh\_setting\_met\_\_mdt. Each Refresh\_setting\_met\_\_mdt record can have a maximum of 5 fieldList\_\_mdt records.

|  |  |
| --- | --- |
| **Field Name** | **Purpose** |
| DeveloperName | Data time stamp prefixed with flds\_\_ is used to maintain uniqueness |
| fieldList\_\_c | List of field selection seperated by comma |
| Refresh\_settings\_\_c | Metadata relationship field to Refresh\_setting\_met\_\_mdt |

**Classes:**

customMetadataService –

Used to access the custom metadata in the test classes. custom metadata can’t be queried from test classes and it will return an empty list if we call a method that is querying a custom metadata.

To avoid this we have created a class that has a private Map<string,List<sobject>> and a method called getQuery.

The getQuery method is called with the query in the controller.

Ex: **customMetadataService.getQuery('SELECT id, MasterLabel,DeveloperName**

**FROM Refresh\_setting\_met\_\_mdt** **‘);**

This method will actually query the records when not in Test Context and returns a list of sobjects.

When we call from test class it will return the value contained in the map. So, in the test class we need to do the below:

**customMetadataService.bigObjectQueryMap.put('SELECT id, MasterLabel,DeveloperName**

**FROM Refresh\_setting\_met\_\_mdt ‘, metadataList );**

Where metadatalist is the list of Refresh\_setting\_met\_\_mdt record instances that we want to be returned in the controller when the query executes.

personalInfoDeletionController –

Main controller for the obscure/ anonymize data tab.

Methods:

getObjectFields – gets all the fields of a selected object. If the object is ‘User’ then it returns a list of profiles.

checkJobRunning- checks if there are any jobs running by querying the Refresh\_setting\_met\_\_mdt custom metadata.

deleteMetadata – deletes all the child fieldList\_\_mdt records, for the Refresh\_setting\_met\_\_mdt record id passed to the method. Deletion is done through metadata API

checkJobs – returns a list of Refresh\_setting\_met\_\_mdt records along with the child fieldList\_\_mdt records to be displayed in the job Dashboard.

submitJob – This method accepts a list of fields/ profiles and the object api name.

1. It checks if there are more than 5 records of fieldList\_\_mdt for the selected object. If yes, an exception is returned asking the user to delete the fieldList\_\_mdt metadata records.
2. If not, we submit the clearPersonalInfoBatch job, create/update a refresh\_setting\_met\_\_mdt record for the selected object with the job status and job Id and deploy the refresh\_setting\_met\_\_mdt and attach a callback personalInfoDeletionCallback to it which accepts the object name and field Value where the fieldList\_\_mdt record is created with the field values placed in the ‘fieldList\_\_c’ field and linked to the parent refresh\_setting\_met\_\_mdt record.

personalInfoDeletionCallback –

Used to create the fieldList\_\_mdt records in the deployment callback, after the Refresh\_setting\_met\_\_mdt record is created.

clearPersonalInfoBatch –

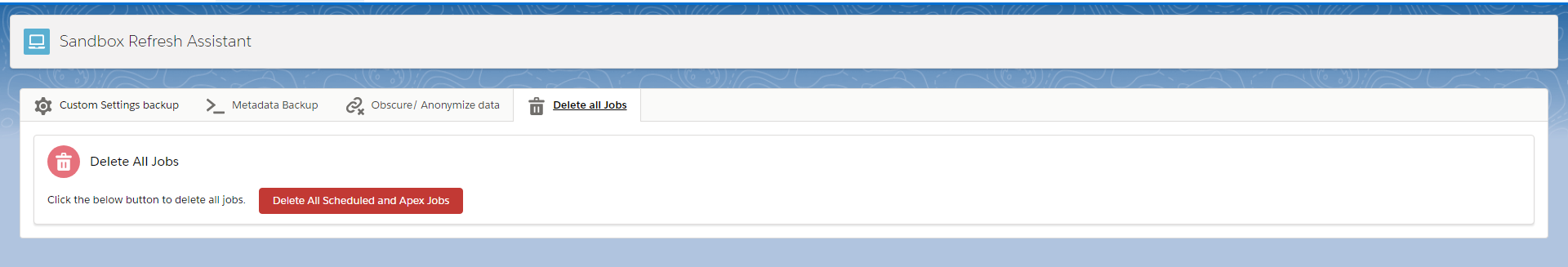
This is the batch class that is responsible for removing the customer personal information from the selected sobject. It also updates the job status during job failure or job success in the refresh\_setting\_met\_\_mdt metadata record.

The personal information is replaced with the current datetimestamp and works only on string, number, email, textarea fields.

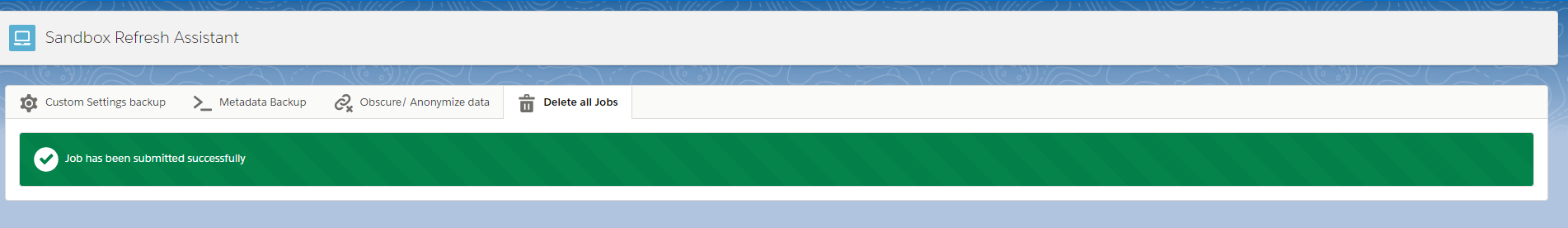
**Tab 4 – Delete all jobs**

This is a **post refresh activity** and this tab consists of a button that allows the user to delete all the scheduled jobs and apex jobs that are still running in the newly refreshed sandbox to avoid manual activity.

The button is hidden once clicked and a job will be submitted to delete all scheduled jobs.



After submitting the job, we get the below message:



**Components used:**

**Lightning components:**

deleteJobComponent – displays a button to delete all the scheduled jobs and apex jobs and hides the button after submission.

**Classes:**

deleteJobController –

This is the main controller and submits the job for deletion and also checks if there are any existing deletion jobs.

Methods:

checkJobRunning – This returns true if there is a job already running for deleting the scheduled jobs.

deleteJobMethod – executes a job for deleting ‘AsyncApexJob’ and ‘CronTrigger’ by passing this parameter to deleteScheduledJobs class.

deleteScheduledJobs –

Batch class for deleting all scheduled and apex jobs. The jobs that are related to the application are skipped using the custom label **jobs\_to\_skip.** This will avoid any metadata backup, custom setting backup jobs to fail.

We query all the jobs based on the parameter passed and abort them.

**VF page application component:**

Has the same functionalities as the lightning version except for custom setting conflict resolution and field preference functionality for obscuring/ anonymizing customer data.

**Class** - sandboxRefreshVFController

**VF Page** – sandboxRefreshVFApplication

**Test Classes:**

|  |
| --- |
| personalInfoDeletionTest |
| MetadataBackupControllerTest |
| BatchJobsTest |
| customSettingBackupTest |
| MetadataServiceTest |
| mainApplicationControllerTest |
| ConflictResolutionControllerTest |
| ZippexTests |
| metadataCalloutMock |
| customSettingBackupTestMock |
| MetadataBackupTestMock |

Test Suite **sandboxRefreshSuite** can be run which covers all test classes.

**Target Org Components:**

Since the backup files need to be stored in a different org, we need to have a webservice in that org to get, create and delete the files. File versioning is provided which stores a new version each time the backup is taken.

**Components used:**

**Field and record type:**

ContentVersion object:

We create a field called **‘org\_name\_\_c’** where the base url of the source org is stored so while retrieving or deleting the files we can do it on the base url (sent in the header) of the org which calls the web service.

The record type ‘**refresh\_backup’** is used to differentiate the files created from this application from any other application since we wont be able to do it using the library name as files cant be filtered by library name if there are multiple versions.

**Classes:**

zipFileServiceClass –

Web service class to get / create/ delete backup files from the target org.

Methods:

doGet – called with the rest resource ‘/saveZipFile/\*’ with GET method. Returns a wrapper object based on the **orgName, isMetadata** Parameter from the header with:

1. All the backup files stored in the org.
2. A boolean indicating whether the file size for the org has reached the limit.
3. The downloadable link base url.

doDelete – called with the rest resource ‘/saveZipFile/\*’ with DELETE method. Deletes the backup files based on the orgName, objects Parameter from the header.

doPost – called with the rest resource ‘/saveZipFile/\*’ with POST method. Creates the backup files based on orgName, filename, libraryName from the header. If the same filename exists for the same org and library name, we create a new version of the file else we create a new file.

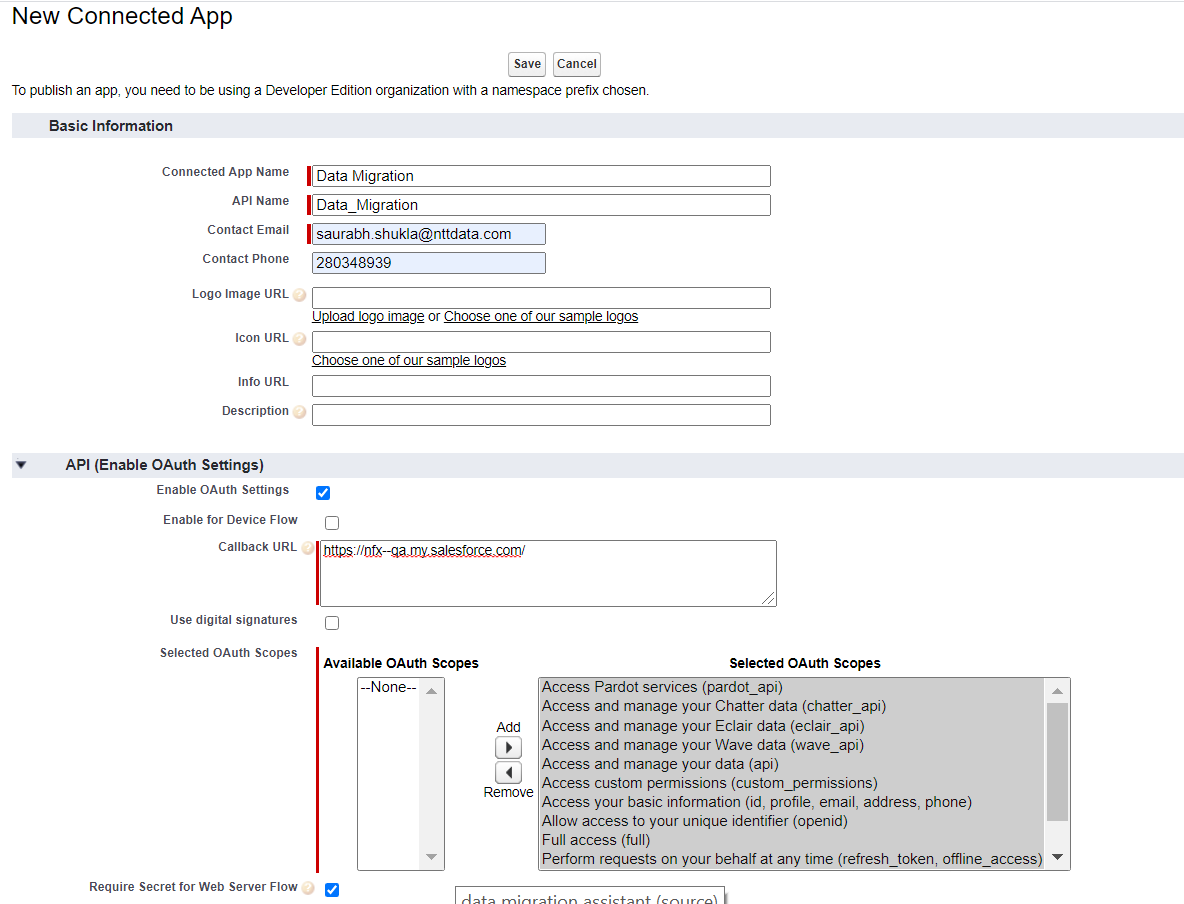
getSize- It returns a Boolean indicating whether the size of files belonging to ‘refresh\_backup’ record type has exceeded the size limit specified in the custom label **backup\_library\_size\_limit**.

**Test Class:**

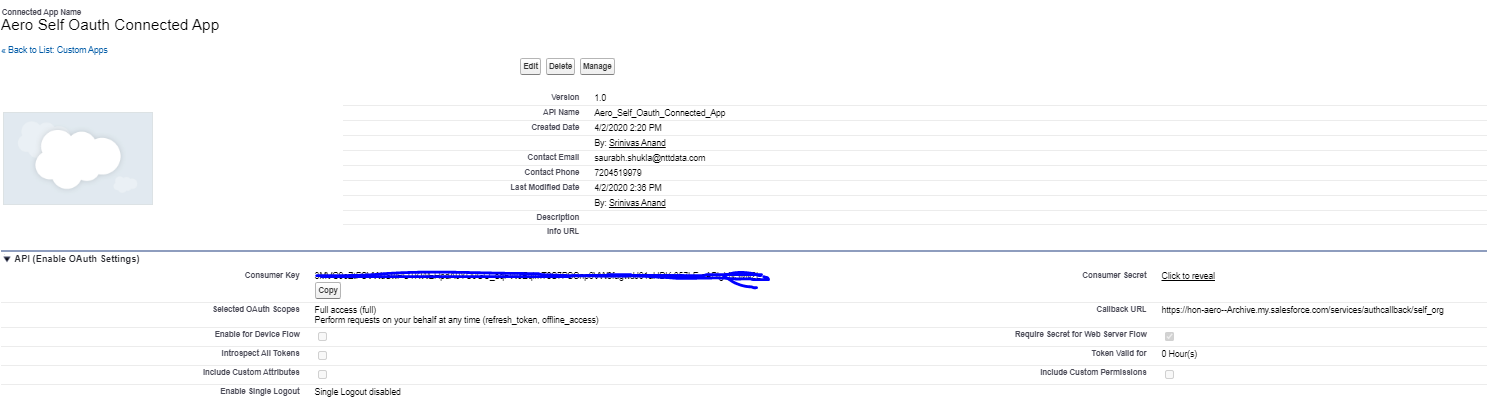
zipFileServiceClassTest

**Connected App setup:**

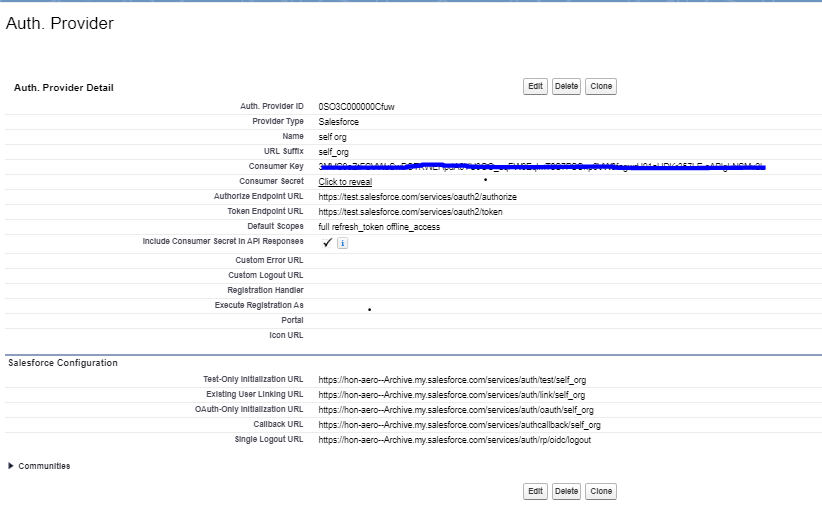
1. Create a connected app in the **target org** with oauth enabled and below settings. Callback url can be the source org for now and select all oauth scopes.



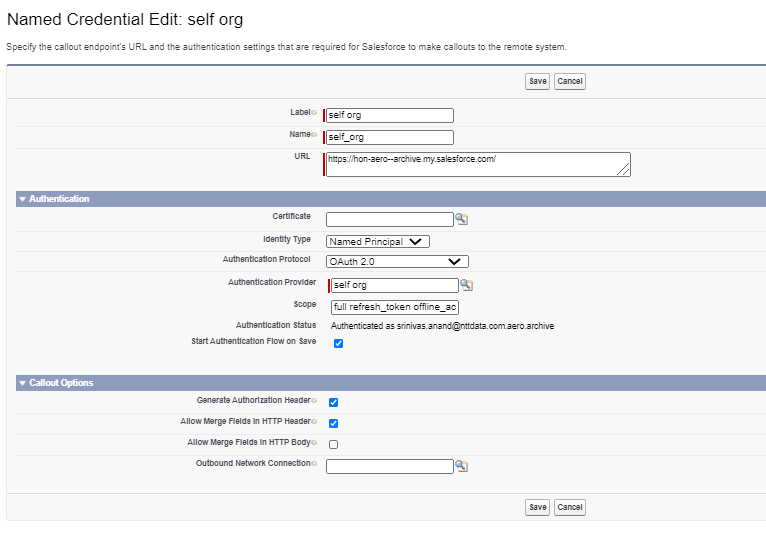
1. Save the app.



1. Create a auth. Provider in the **source org** with salesforce settings. Copy the client Id and client secret from the above created connected app and paste it in the auth provider.



1. Save the auth. Provider and you will find a callback url in the related list. Copy and paste the callback url in the connected app callback url.
2. Create a named credential in the **Source Org** use the Named principal authentication and oauth 2.0 option.
3. Lookup the auth provider that was created and use the same in the named credential.
4. Allow merge fields in the Http header, generate authorization header, start authentication flow on save should be checked.



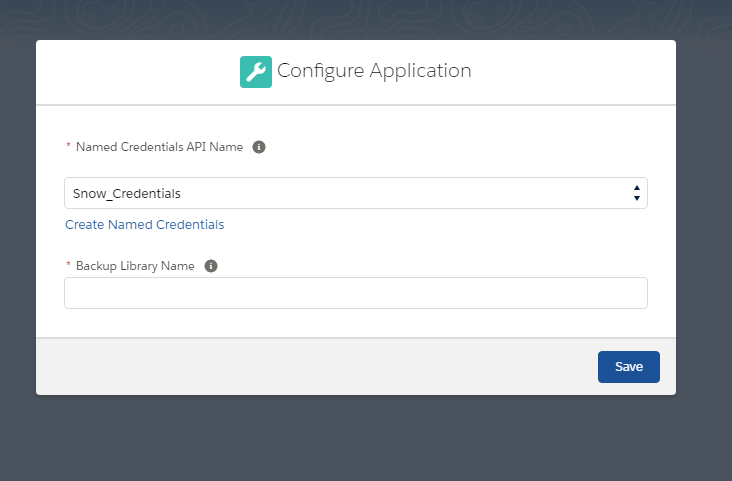
1. Authorize the named credential on Save.

**Post Installation Steps Target Org:**

1. Go to the custom label **backup\_library\_size\_limit** and provide the maximum size in Bytes that should be dedicated for storing backup files from all orgs.
2. Create a library to be used as storage for backup files and give it a unique name.

**Post Installation Steps Source Org:**

1. Open the application ‘Sandbox Refresh Assistant and you will see the popup below:



1. It will show the list of named credentials in your org. Select the one that was created previously.
2. Enter the backup library API name from the target org and Save.