The Object Exporter task is a plugin class that writes SailPoint objects to XML files. The files can be placed into date stamped folders, and later can be zipped using the Zip Folder task.

Step-by-step guide

Creating a TaskDefinition to export all objects in a class:

- Open the Tasks page
- Select New Task
- Scroll down to find and select: Plugin Object Exporter
- A new TaskDefinition will be created with default values. Name the task according to your corporate guidelines.
- Provide a base folder for the exported files to go into. The system creates the Class folders, so it's typical that the last folder in the path is /config but that is not required.
- Typically the Remove IDs and Add CDATA checkboxes are checked.
- Specify a list of classes to export. You can also use default or leave blank, but that is not recommended.
- Specify a regex of object names to export, if you do not want them all. Best practice is to prefix all custom objects with a single prefix and use that in this field.
- Typically the Strip metadata boxes are checked but this is user defined.
- You can specify a create date to limit data exported.
- Specify the naming format, see the tool tip for examples.
- If you are using a reverse.target.properties file in the format needed for IIQDA, specify it here.
- If you have a folder of exported AuditConfig, UIConfig, Configuration, ObjectConfig, and Dictionary objects from an uncustomized system (of the same version/patch) you can reference the folder here and merge files will be created instead of full exports.

Repeat as needed. Typically a Sequential Task runner is used to sequentially run the tasks. The files will show in the folder when done.

The Plugin Zip Folder task can also be created to zip files, it is self-explanatory. See its user guide.

Examples:

Example 1: Naming custom created objects in certain classes using a prefix reflecting the customer, such as ACME.

If you do this, the extracts are much easier because you can set up just a few export tasks:

ACME Export Applications
ACME Export Configurations
ACME Export ACME

There the ACME Export Applications will have the following class names (all on one line):

Application,DynamicScope,Custom,UIConfig,AuditConfig,
CorrelationConfig,Workgroup,GroupDefinition

This works because typically these class objects are not normally prefaced with the client prefix, because there are no OOTB object of these types, except for DynamicScope.

ACME Export Configurations will have the following class names:

Configuration,ObjectConfig,TaskSchedule,ServiceDefinition

(the last two are optional)

And the following Regex filter:

^(System|Identity|Bundle|ManagedAttribute|Link|Connector).*

Putting the prefix of whichever ones of the customized objects into this list (above is an example of creating a "start with one of these" regex expressions).

And then finally the ACME Export ACME will have class names:

Rule,TaskDefinition,Workflow,Form,TaskSchedule,EmailTemplate,
PasswordPolicy,QuickLink,IdentityTrigger

And the regex of ACME.*

Or you can construct a start with one of these filters, including ACME.

This method reduced the number of export tasks to 3.

Example 2: Client has already messed up the names so you have no choice but to export everything.

Construct the following tasks (try to use a Prefix to keep your sanity)\

ACME Export All Managed Objects
ACME Export Application
ACME Export AuditConfig
ACME Export Bundle
ACME Export Configuration
ACME Export CorrelationConfig
ACME Export Custom
ACME Export DynamicScope
ACME Export EmailTemplate
ACME Export Form
ACME Export GroupDefinition
ACME Export IdentityTrigger
ACME Export ObjectConfig
ACME Export QuickLink
ACME Export Rule
ACME Export Scope (optional)
ACME Export TaskDefinition
ACME Export TaskSchedule (optional)
ACME Export UIConfig
ACME Export Workflow
ACME Export Workgroup

And have the first one be a Sequential task that runs all of the others.

Appendix A: Use of the reverse.target.properties file

Just as the IIQDA uses a reverse.target.properties file, the Object Exporter also can.

For example, if you were to have a Custom object that looks like this:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE Custom PUBLIC "sailpoint.dtd" "sailpoint.dtd">
<Custom name="ACME Static Data">
  <Attributes>
    <Map>
      <entry key="AD_PREFERRED_SERVER" value="%%AD_PREFERRED_SERVER%%"/>
      <entry key="DOMAIN_DC" value="%%DOMAIN_DC%%"/>
      <entry key="ENV" value="%%CUSTOM_ENV%%"/>
      <entry key="UPN_SUFFIX" value="%%UPN_SUFFIX%%"/>
    </Map>
  </Attributes>
</Custom>
```

And a dev.target.properties file reading:

```
%%AD_PREFERRED_SERVER%%=addc1.acmetest.net
%%DOMAIN_DC%%=dc=acmetest,dc=net
%%CUSTOM_ENV%%=DEV
%%UPN_SUFFIX%%=acmetest.net
```

When this is deployed, the active data is stored in the database. You will need to pull this data back and convert it back to the tokens (left side of the target file)

Remember that this only works for XML, do not put tokens in Java code.

The reverse.target.properties file looks like this:

```
/Custom[@name\='ACME\ Static\
Data']/Attributes/Map/entry[@key\='ENV']/@value=%%CUSTOM_ENV%%
/Custom[@name\='ACME\ Static\
Data']/Attributes/Map/entry[@key\='AD_PREFERRED_SERVER']/@value=%%AD_PREFER
RED_SERVER%%
/Custom[@name\='ACME\ Static\
Data']/Attributes/Map/entry[@key\='DOMAIN_DC']/@value=%%DOMAIN_DC%%
/Custom[@name\='ACME\ Static\
Data']/Attributes/Map/entry[@key\='UPN_SUFFIX']/@value=%%UPN_SUFFIX%%
```

(This is 4 lines, ignore word wrap)

The reverse tokenization converts whatever is in the value of the element into the token, no matter which environment it came from.

Appendix B: Use of base files to compute import XML

Import XML files overlay the key value pairs into the XML without removing all of the others. This is done using the following:

A Full System Configuration XML file starts with:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE Configuration PUBLIC "sailpoint.dtd" "sailpoint.dtd">
<Configuration name="SystemConfiguration">
  <Attributes>
    <Map>
```

This file normally replaces the entire object in memory, and for SystemConfiguration, this means you have to be very careful, and you have to store the entire object in Git.

An import XML will start like this:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE sailpoint PUBLIC "sailpoint.dtd" "sailpoint.dtd">
<sailpoint>
<ImportAction name="merge">
<Configuration name="SystemConfiguration">
  <Attributes>
    <Map>
```

For this file, you will only put the key-value pairs that are different from the default. Using an import XML means less data are stored in the file so it's easier to track in Git. For UIConfig you have to be careful as many of the values are a Map. If you add values to a Map that works fine but removing values from a Map using an Import XML requires a different approach.

For example to overwrite just the identityViewAttributes you would just do:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE sailpoint PUBLIC "sailpoint.dtd" "sailpoint.dtd">
<sailpoint>
  <ImportAction name="merge">
    <UIConfig name="UIConfig">
      <Attributes>
        <Map>
          <entry key="identityViewAttributes" value="name,firstname,lastname,email,
identityStatus,manager,type,softwareVersion,administrator,phone,state,employeeId"/>
        </Map>
      </Attributes>
    </UIConfig>
  </ImportAction>
</sailpoint>
```

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE sailpoint PUBLIC "sailpoint.dtd" "sailpoint.dtd">
<sailpoint>
  <ImportAction name="merge">
    <UIConfig name="UIConfig">
      <Attributes>
        <Map>
          <entry key="identityTableColumns" value="DELETE"/>
        </Map>
      </Attributes>
    </UIConfig>
  </ImportAction>
  <ImportAction name="merge">
    <UIConfig name="UIConfig">
      <Attributes>
        <Map>
      <entry key="identityTableColumns">
        <value>
          <List>
            <ColumnConfig dataIndex="id" fieldOnly="true" groupProperty="id" property="id"
sortProperty="id" stateId="id"/>
            <ColumnConfig dataIndex="name" groupProperty="name"
headerKey="idents_grid_hdr_name" hideable="true" property="name" sortProperty="name"
sortable="true" stateId="name"/>
            <ColumnConfig dataIndex="firstname" groupProperty="firstname"
headerKey="idents_grid_hdr_first_name" hideable="true" property="firstname"
sortProperty="firstname" sortable="true" stateId="firstname"/>
            <ColumnConfig dataIndex="lastname" groupProperty="lastname"
headerKey="idents_grid_hdr_last_name" hideable="true" property="lastname"
sortProperty="lastname" sortable="true" stateId="lastname"/>
            <ColumnConfig dataIndex="manager-displayName"
groupProperty="manager.displayName" headerKey="idents_grid_hdr_manager"
hideable="true" property="manager.displayName" sortProperty="manager.displayName"
sortable="true" stateId="manager-displayName"/>
            <ColumnConfig dataIndex="assignedRoleSummary"
groupProperty="assignedRoleSummary" headerKey="idents_grid_hdr_assigned_roles"
hideable="true" property="assignedRoleSummary" sortProperty="assignedRoleSummary"
stateId="assignedRoleSummary"/>
            <ColumnConfig dataIndex="bundleSummary" groupProperty="bundleSummary"
headerKey="idents_grid_hdr_detected_roles" hideable="true" property="bundleSummary"
sortProperty="bundleSummary" stateId="bundleSummary"/>
            <ColumnConfig dataIndex="scorecard-compositeScore"
groupProperty="scorecard.compositeScore" headerKey="idents_grid_hdr_composite_score"
hideable="true" property="scorecard.compositeScore"
renderer="SailPoint.Define.Grid.Identity.renderScore"
sortProperty="scorecard.compositeScore" sortable="true" stateId="scorecard-
compositeScore"/>
```

```
            <ColumnConfig dataIndex="lastRefresh" dateStyle="short"
groupProperty="lastRefresh" headerKey="idents_grid_hdr_last_refresh" hideable="true"
property="lastRefresh" sortProperty="lastRefresh" sortable="true" stateId="lastRefresh"/>
            <ColumnConfig dataIndex="type" groupProperty="type"
headerKey="idents_grid_hdr_type" hideable="true" property="type" sortProperty="type"
sortable="true" stateId="type"/>
            <ColumnConfig dataIndex="softwareVersion" groupProperty="softwareVersion"
headerKey="idents_grid_hdr_software_version" hidden="true" hideable="true"
property="softwareVersion" sortProperty="softwareVersion" sortable="true"
stateId="softwareVersion"/>
            <ColumnConfig dataIndex="administrator-displayName"
groupProperty="administrator.displayName" headerKey="idents_grid_hdr_administrator"
hidden="true" hideable="true" property="administrator.displayName"
sortProperty="administrator.displayName" sortable="true" stateId="administrator-
displayName"/>
            <ColumnConfig dataIndex="managerStatus" fieldOnly="true"
groupProperty="managerStatus" property="managerStatus" sortProperty="managerStatus"
stateId="managerStatus"/>
          </List>
        </value>
      </entry>
        </Map>
      </Attributes>
    </UIConfig>
  </ImportAction>
</sailpoint>
```

In order to generate the import files, you will need to create a folder of the base exports. You have to generate base exports on a brand new build. You will need to export the following objects:

AuditConfig
Configuration
Dictionary
ObjectConfig
UIConfig

And reference the containing folder.

Appendix C: Using the ignore files

The ignore files feature allows you to prevent the export of any file by placing a copy into a file structure and referencing it in the ignore files field.

The most obvious use of this feature is to export ALL of the objects from a base build and then reference those objects. This can allow you to use any naming standard you want in your customizations without the risk of exporting the OOTB objects.

The converse of that is that if you modify the OOTB objects, the changes won't be exported. While this might seem an issue, it can be used to export the OOTB objects and compare with the original values.