DataRaptor Summer '17

December 2017

Vlocity™
INDUSTRY CLOUD APPS

salesforce strategic ISV partner

Vlocity Inc.
(844)-VLOCITY

Contents

Chapter 1 Get Started with DataRaptor

Vlocity DataRaptor is a data loading and extraction engine that simplifies importing and exporting relational data. With Vlocity DataRaptor, you can easily map denormalized data exported from legacy and product-centric systems into Vlocity's customer-centric relational schema. Fields from Interface Objects can be mapped to standard and custom objects, including defining foreign key relationships.

DataRaptor supports Lookup mappings, which map data in existing objects into new fields and Linked mappings, which map data created in previous mapping steps into new fields. DataRaptor can also use a preprocessor Apex class to manipulate data before mapping into Vlocity. Detailed error logging is available to assist administrators in resolving import issues.

How Does DataRaptor Work?

Imagine you have customer account data in a legacy system that you'd like to import into Salesforce. This data includes fields that are parts of the Account (Account Name, Record Type, Account Number…) and Contact (Last Name, First Name, Mobile Number…) objects. To import them using Vlocity DataRaptor:

1. Before import, make sure the source data is consistent with the Salesforce data model. If your data isn't compliant, you can modify the data upon import with a preprocessor Apex class.
2. Create an Interface Object or JSON.
3. Map fields from Interface Object or JSON to target object fields using in DataRaptor.
4. Load the data using any standard Salesforce tool (REST, Bulk import, or nightly CSV).
5. Check and resolve any errors.

Why Use Vlocity DataRaptor?

Vlocity DataRaptor greatly reduces the time, effort and money required to import or export data to or from Salesforce objects. For example, if you have customer account data you'd like to have available in Salesforce. Some of the data has to be written to the Contact object, while other data has to be written to the Account object. Normally, this means importing data, one object at a time. With DataRaptor, a single mapping can write across multiple Salesforce objects. DataRaptor also makes it easy to manage the data import or export process with tools that let you specify keys, manage the order in which data is written / extracted, and to create linked and lookup mapping types.

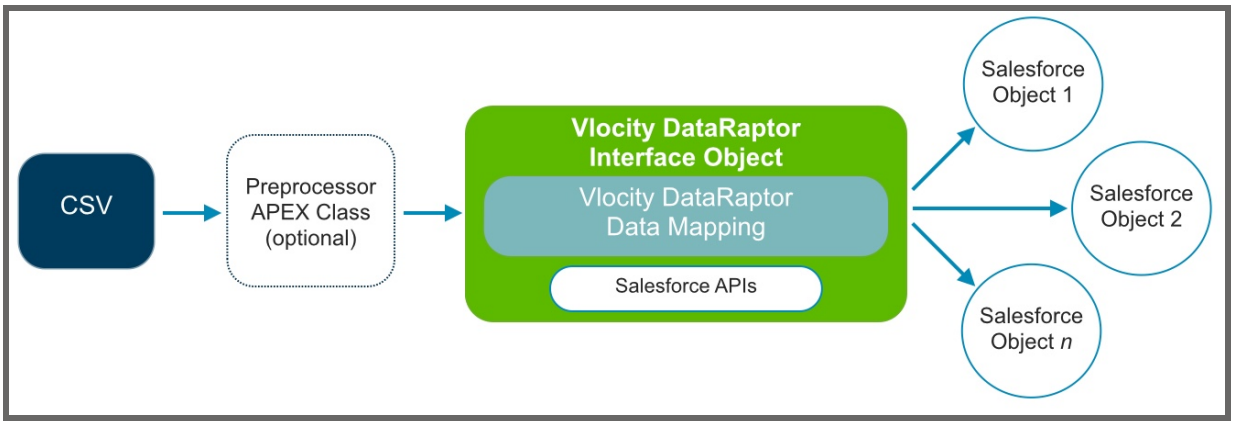
Import and Export Data

Vlocity DataRaptor can be used to both import and export data. DataRaptor can import denormalized data from legacy and product-centric systems via a staging Interface Object. Alternatively, data can be loaded as JSON data through an API. In either case, DataRaptor Data Mapping lets you perform CRUD operations against multiple standard and custom Salesforce objects in a single import. Vlocity DataRaptor Data Mapping can also be used to

export data. Whether you need data for your CRM system or for a Vlocity OmniScript, DataRaptor queries objects and returns JSON data in the format specified in the DataRaptor Interface. Exported data returns to your external system through an API, or to a Vlocity OmniScript.

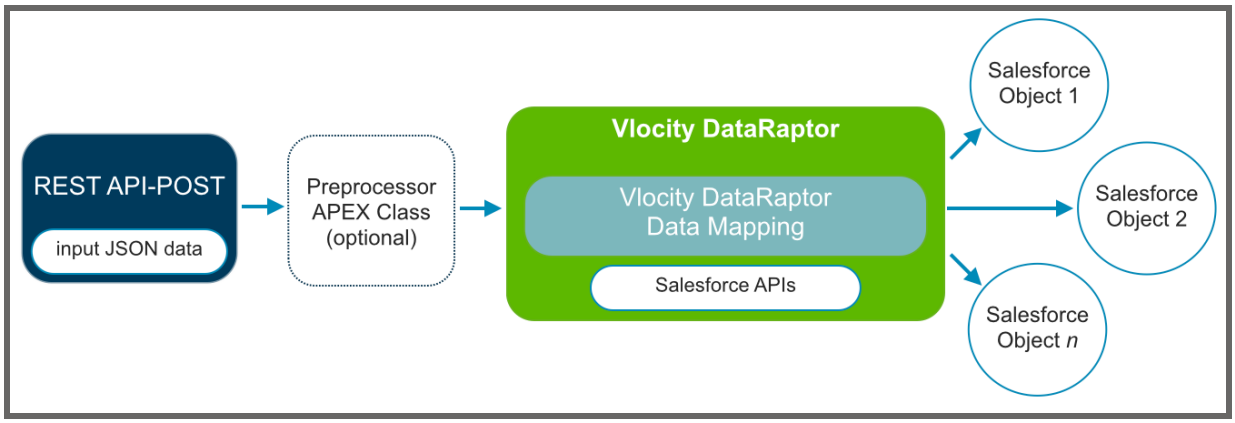
Import Data From a Flat File

When you import data via CSV, data is preprocessed by the (optional) Preprocessor Apex class. Data is then written into an Interface Object from which it is written to Salesforce and custom objects according to the data mapping.



Import Data From JSON Using a REST API

When you import data via REST API you use a sample JSON file to help you complete the mapping. Then, upload the JSON file from which data is written to Salesforce and custom objects.



Create a Preprocessor Class

If your data is not formatted correctly, you can write an Apex class that implements the IDRPreprocess interface to modify data before it is imported to the Interface Object. This Apex Interface runs `Before Insert` on each record uploaded to the Interface Object. For example, your data may include a Contact name in a single `LastName, FirstName` field. However, the standard Salesforce Contact object uses two fields—`FirstName` and `LastName`. You can create a custom Apex class to format the Contact names correctly prior to import into the Interface Object.

To create an Apex class that implements IDRPreprocess:

1. From Setup, click **Develop**, and then click **Apex Classes**, and then click **New**.
2. Write the Apex class. Your class must implement the IDRPreprocess global interface:

```
global interface IDRPreprocess
{
    // Performed once before iterating through all uploaded objects
    void bulkBeforePreprocess();

    // Performed on each object
    void preprocessObject(SObject so);

    // Performed once after iterating through all uploaded objects
    void bulkAfterPreprocess();
}
```

3. Click **Save**.

To configure DataRaptor to use the preprocessor class:

1. Go to the DataRaptor tab and select the Interface that uses the preprocessor class.
2. In the `Preprocessor Class Name` field, enter the name of the preprocessor class.
3. Click **Save**.

Use Environment Variables with DataRaptor

DataRaptor has access to a number of Environment Variables. These can be used as Default Values, Filter Values, or in Formulas.

For example, you may want to create a DataRaptor Extract JSON that only extracts cases created in the last thirty days. You can do this by creating a filter and using the Environment Variable `$Vlocity.N_DAYS_AGO:{N}`.

If you are using an Environment Variable as a Filter Value, the variable should be placed in quotation marks. Quotes are not necessary when using an Environment Variable as a Default Value. The table below details the available Environment Variables.

Variable	Description
$Vlocity.TODAY	This variable returns today's date.
$Vlocity.TOMORROW	This variable returns tomorrow's date.
$Vlocity.NOW	This variable returns the current date and time.
$Vlocity.N_DAYS_FROM_NOW:{N}	This variable returns the date N days from now, where {N} is the number of days.
$Vlocity.N_DAYS_AGO:{N}	This variable returns the date N days previous, where {N} is the number of days.
$Vlocity.NULL	This variable returns null.
$Vlocity.StandardPricebookId	This variable returns the Id of the Standard Pricebook.
$Vlocity.DocumentsFolderId	This variable returns the Documents folder ID.
$Vlocity.true or $Vlocity.TRUE	This variable returns to Boolean type true.
$Vlocity.false or $Vlocity.FALSE	This variable returns the Boolean type false.
$Vlocity.UserId	This variable returns the current user's ID.

DataRaptor XML

DataRaptor supports the use of XML as an Import and Output type.

The XML Format in DataRaptor is a Key-Value / JSON-like structure.

Note: Order is not maintained between XML → JSON → XML. No Order information in created in the JSON.

For example, the following XML code example:

```
<Root attribute1="attVal1" xmlns="http://namespace1" xmlns:a="http://namespace2">
rootValue
<Child attribute2="attval2"></Child>
<Child>childValue</Child>
</Root>
```

becomes the following code when transformed in to JSON:

```
{
"Root":{
"#text":"rootValue",
"@attribute1":"attVal1",
"#ns":"http://namespace1",
"#ns#a":"http://namespace2",
"Child":[
{
"@attribute2":"attval2"
},
{
"#text":"childValue"
}
]
}
}
```

Values inside an XML Node will always be written with the Key "#text". For example, the following code:

```
<Root>rootValue</Root>
```

Becomes:

```
{
"Root":{
"#text":"rootValue"
}
}
```

Attributes in an XML Node will have the @ symbol applied to the beginning of their Key. For example, the following code:

```
<Root attribute1="attVal1"></Root>
```

Becomes:

```
{
"Root":{
"@attribute1":"attVal1"
}
}
```

Namespaces in an XML Node will have #ns# applied to the beginning of their Key, or will be "#ns" when no more text after xmlns:

```
<Root xmlns="http://namespace1" xmlns:a="http://namespace2"></Root>
```

Becomes:

```
{
"Root":{
"#ns":"http://namespace1",
"#ns#a":"http://namespace2",
}
}
```

Since ":" is a protected character in DataRaptor, the "#" symbol is also used as a replacement character for ":" inside Keys. For example, the following code::

```
<Root:Data>data</Root:Data>
```

Becomes:

```
{
"Root#Data":{
"#text":"data",
}
}
```

When mapping a DataRaptor that will Output XML it is important to follow this exact same format.

Chapter 1 DataRaptor Transforms

External systems and OmniScripts can have different data type values. DataRaptor has a Transform layer where values can be modified through formulas and functions.

For example, you might have an external system with a string value for `Do Not Contact` equal to 'Y' or 'N'. You want to use this information to populate a checkbox in an OmniScript. Checkboxes use Booleans, so the string information can't be passed over as is. You can use the DataRaptor transform to change the data type; Do Not Contact = 'Y' can be set to *Marketing Opt Out* = 'true'.

Create DataRaptor Transforms

Each type of DataRaptor has a Transform step where Formula Mappings can be created to modify the data as part of the DataRaptor.

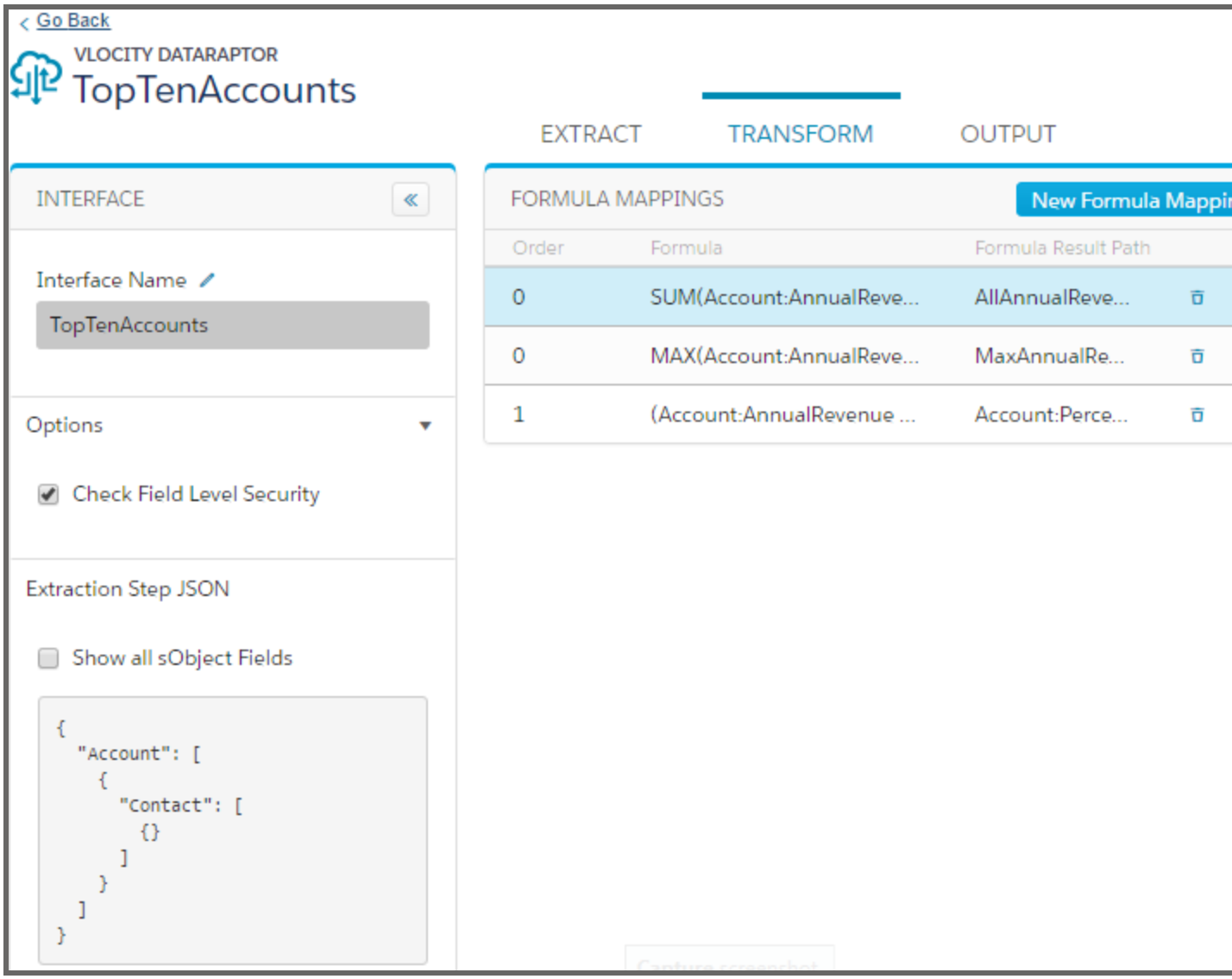
For example, we can create an Extract DataRaptor to take a contact's FirstName and LastName and create a new field with "LastName, FirstName" as the value. An Extract DataRaptor has the three following steps:

1. Extract.
2. Transform.
3. Output.

During the Extract Step, DataRaptor will lookup the associated contact.

```
{
"Id" :"ContactId",
"Contact" : {
"Id": "ContactId",
"FirstName" : "Bob",
"LastName" : "Smith" }
}
```

In the Transform Step, all the formulas for the DataRaptor will run. For an example image of the Transform step and formula mappings, see below.



- `Formula`: The formula definition itself.
- `Formula Result Path`: The path that the result will be inserted into the Extract Data. If the result should apply to each instance of the object, the naming convention should start with the object name followed by a colon. If the result path should apply to a single account, the `Formula Result Path` should start with the following naming convention `Account:`. This field must be entered
- `Order`: This field determines the order in which the formulas will run.

You can create a transform by clicking **New Formula Mappings** and entering the information needed in the Mapping Detail. For our example, we would use the following formula mappings.

Formula: **`CONCAT(Contact:LastName,", ",Contact:FirstName)`**

Result Path: **`Contact:FullName`**

The result of this function would be the string: *`Smith, Bob`*.

After the Transform Step runs, the JSON data available for mapping to output would be:

```
{
"Id" :"ContactId",
"Contact" : {
"Id": "ContactId","
"FullName" : "Smith, Bob",
"FirstName" : "Bob",
"LastName" : "Smith" }
}
```

The Output step maps the JSON shown above to the final Output. This step must be completed or else the formula will not have a values. You can create an output by clicking **New Mapping** and entering the information needed in the Mapping Detail. For our example, we would use the following formula mappings.

Extract JSON Path: **`Contact:FullName`**

Output Path: **`CustomerName`**

This mapping would result in the following ouput:

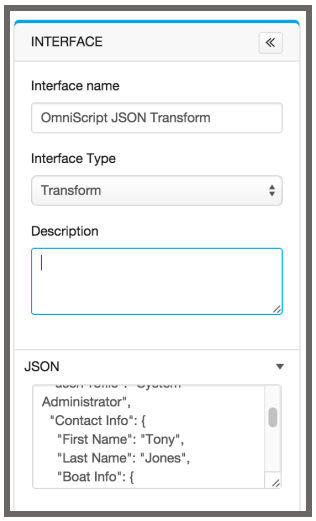
```
{
"CustomerName" : "Smith, Bob"
}
```

Create a Transform Interface

The DataRaptor transform interface allows you to transform JSON by relabeling keys, reducing attributes, and reconfiguring header attributes. The Transform service leverages Vlocity Calculations and OmniScript PDF Actions, but can be used for any situation where you need to input a JSON and transform it into another JSON.

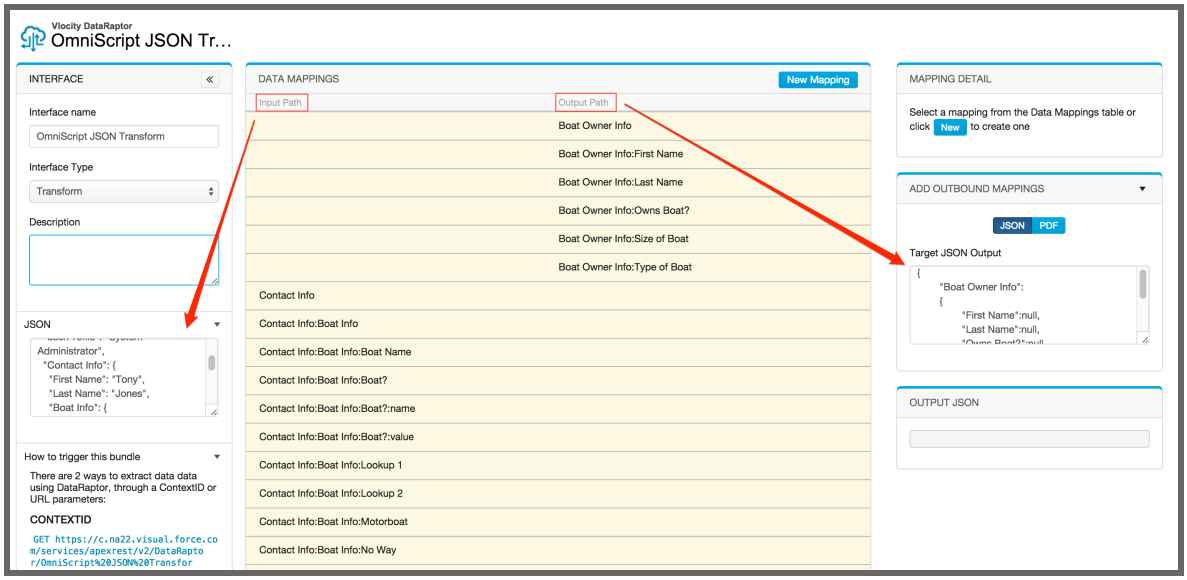
To transform an Input JSON to an Output JSON:

1. On the Vlocity DataRaptor tab, click **New**.
2. Enter an `Interface name` and select `Transform` for the `Interface Type`.
3. In the `JSON` field, paste the Input JSON.

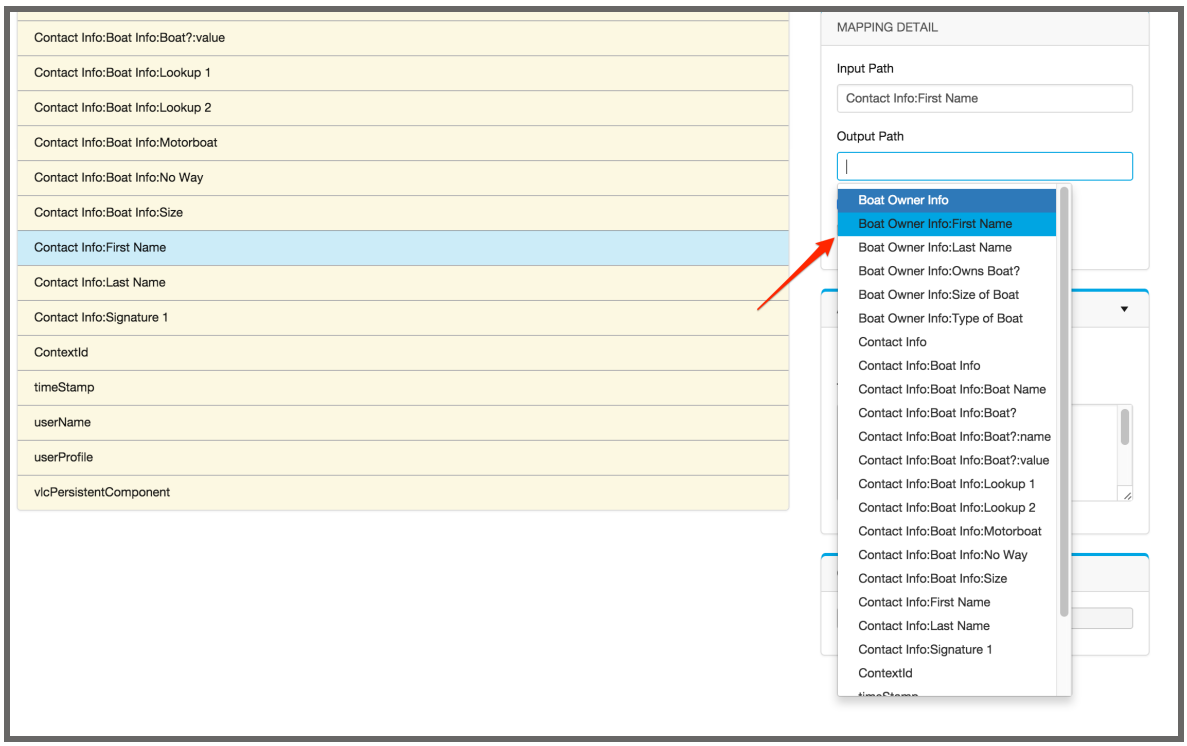


4. In the `Target JSON Output` field, paste the Target JSON structure.

   The Data Mappings table now displays every JSON node from the Input JSON on the left side, and every node from the Target JSON Output on the right side. For both the Input and Target JSON, you are not required to have a mapping for every field.



5. Click the Input JSON node to map to the Target JSON.
6. In the `Output Path`, select the path to the JSON node.



7. Repeat Step 5 until all fields are mapped.

After configuring the Transform interface, you can use the DataRaptor API to run the transform service or create a DataRaptor Transform Action in OmniScript. For more information, see **Run, Test, and Monitor DataRaptor Jobs** on page 59.

DataRaptor Transform Output Types

This page explains the supported Output Data Types available when creating a new mapping in DataRaptor. The Output Data Type is used for the for determining the final Output JSON.

DataRaptor Transform Output Data Types

Output Data Type	Description
Boolean	Converts to Boolean
Currency	Converts to Currency using the Salesforce org's Currency Code.
Double	Converts to Double
Integer	Converts to Integer
JSON	Serializse as JSON
List<Double>	Convert Double to List<Double>
List<Integer>	Converts Integer to List<Integer>
List<Map>	Converts from Map<String,Object> to List<Map<String, Object>
List<String>	Convert String to List<String>
Object	Deserializes a String as Map<String, Object> or List<Object>
String	Convert primitive to String
Multi-Select	Convert to Salesforce Multi-Select string ie: String.join(list, ';');

For lists of values transformed to a primitive type like Double, Integer, String, or Boolean, DataRaptor will select the first element in the list and cast to that type.

Formula Syntax for DataRaptor

Formulas can be used with DataRaptor transforms to update information. For example, to determine the total price of items purchased by a customer, use the following formula:

```
Formula__c = 'SUM ( Products:Price )'
FormulaResultPath__c = 'TotalPrice'
```

All unquoted tokens in the formula, including parenthesis and commas, must be separated by a space. If your variable names contain spaces or non-alphanumeric characters, enclose the variable name in double quotes and precede it with "var:" in formulas. For example, if the JSON node name is **Primary Guardian**, specify it in formulas as **var:"Primary Guardian"**.

Formulas support the following operators:

```
&& AND || OR > >= < <= = == != <> LIKE NOT LIKE + - * / % ^
```

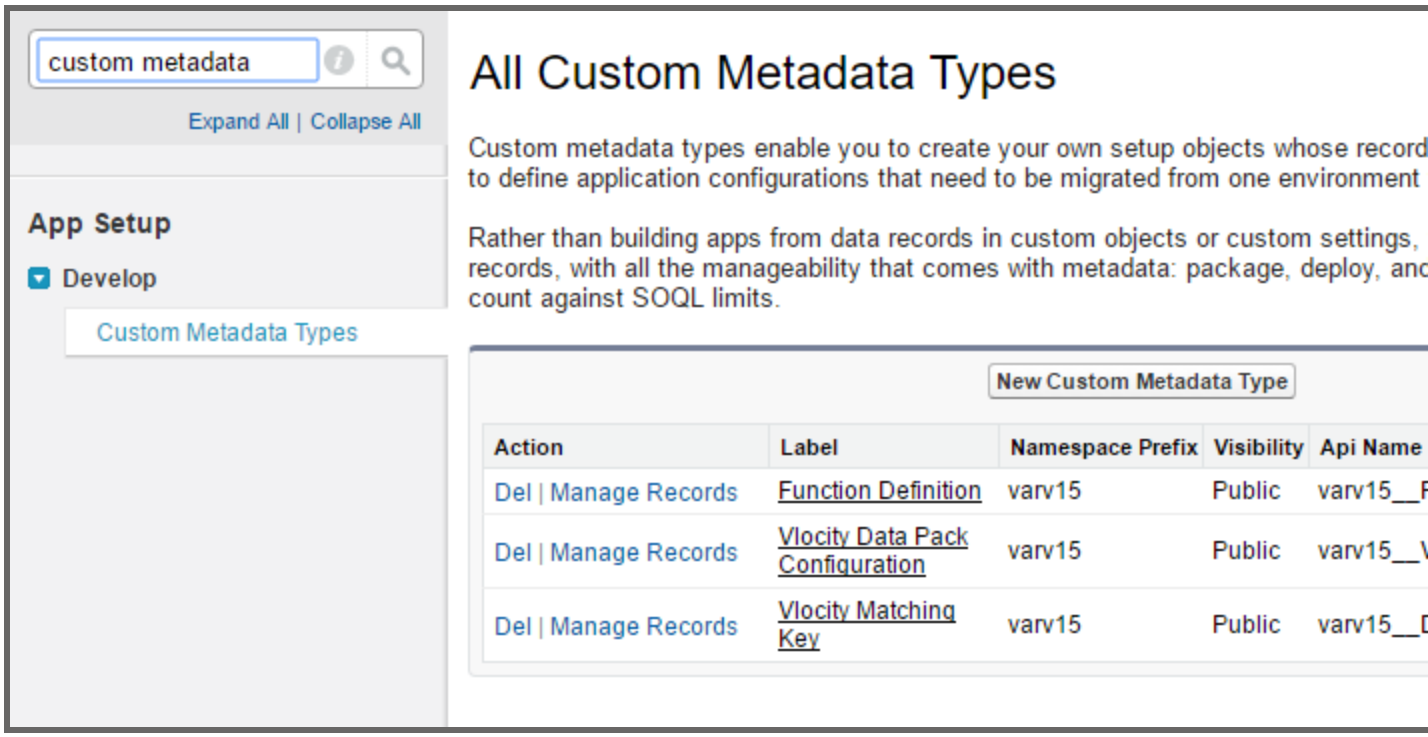
The following table lists and describes the functions that you can use in formulas

Function	Returns	Example
ABS	Absolute value of the specified number	ABS(-1) returns 1
ADDDAY	Adds the specified number of days to the specified date and returns the resulting date.	ADDDAY("1999-01-01",100) returns "1999-04-11 00:00:00"
ADDMONTH	Adds the specified number of months to the specified date and returns the resulting date.	ADDMONTH("1999-01-01",100) returns "2007-05-01 00:00:00"
ADDYEAR	Adds the specified number of years to the specified date and returns the resulting date.	ADDYEAR("1999-01-01",100) returns "2099-01-01 00:00:00"
AGE	Returns the age for the specified birthdate.	AGE(Account:Contact:Birthdate)
AGEON	For a specified birthdate, returns the age on the specified date	AGEON(Account:Contact:Birthdate,TODAY()) (same as AGE).
AVG	Average of list of values.	AVG(1,2,3,4,5,6,7,8,9,10) returns 5.5
CONCAT	Returns the concatentation of the specified strings.	CONCAT(Contact:FirstName, " ", Contact:LastName)
COUNTQUERY	Returns the number of rows that meet the WHERE clause criteria.	COUNTQUERY ("SELECT COUNT () FROM Case WHERE AccountId = '{0}'" , Id)
DATEDIFF	Returns the number of days between two specified dates. If the first date is greater than the second date, the value is returned as a negative number.	DATEDIFF("1900-01-01","2000-01-01") returns 36524 DATEDIFF("2000-01-01","1999-01-01") returns -36524 DATEDIFF(Account:Cases:CreatedDate,TODAY()) returns the number of days a case has been open

EOM	For a specified date, returns the date of the last day of the month.	EOM(Account:Cases:CreatedDate)
IF	If the condition is true, returns the first value, if false, returns the second.	IF(InputDate < "2000-01-01", "20th Century","21st Century")
MAX	Maximum value in a list of values.	MAX(1,2,3,4,5,6,7,8,9,10) returns 10
MONTH	Returns the month of the specified date as an integer.	MONTH("1999-01-11") returns 1
QUERY	Executes a SOQL query that returns a list of values. The query cannot retrieve more than one column.	QUERY ("SELECT Name FROM Account WHERE BillingState ='CA'") returns `{ "01Test": [ "ABC Company", "Kaiser Permanente", "Hoogasian Flowers" ] }`
ROUND	Rounds the specified number or expression. By default, results are rounded to two decimal places, but you can specify the desired number of decimal places for the result.	ROUND(3.1415 * 3) = 9.42 ROUND(3.1415 * 3, 0) = 9
SUM	Returns the sum of a list of values	SUM(1,2,3,4,5,6,7,8,9,10) returns 55
TODAY	Returns today's date	
YEAR	Returns the year of the specified date as an integer.	YEAR("1999-01-11") returns 1999

Create Custom Functions for Formulas

Custom functions can be defined in a Custom Metadata Object called `Function Definition`, as shown in the following figure.



A Function Definition has three fields:

1. Name: Controls how a Function is referenced in a Formula.
2. Class Name: The VlocityOpenInterface Implementation Class.
3. Method Name: The method name sent through the invoke service.

Custom Functions do not need additional syntax when used in a Formula. They are resolved using the FunctionDefinition.Name.

An example of a FunctionDefinition record is listed below:

```
FunctionDefinition__mdt.Name = SPLIT
FunctionDefinition__mdt.ClassName__c = CustomFunctionImplementation
FunctionDefinition__mdt.MethodName__c = split
```

This example shows how the function is used in a formula:

```
SPLIT ( "Smith, Bob" , "," , "LastName" , "FirstName" )
```

The following example shows the supporting code for the custom function.

```
global class CustomFunctionImplementation implements vlocity_cmt.VlocityOpenInterface
{
    /*
        inputs - arguments - List<Object> of passed in arguments
        output - result - The result of the Function being called. Supports single Object val-
ues, List<Object> or Map<String, Object>
    */
    global Boolean invokeMethod(String methodName, Map<String, Object> inputs, Map<String,
Object> output, Map<String, Object> options)
    {
        /*
            Split Returns a Map<String, Object of values. Not all functions will be able to
handle Map<String, Object> results,
            so be careful when using these functions.
            In a DataRaptor Transform step returning a Map<String, Object> from a Formula will
result in applying that Map to the
            JSON Data at the FormulaResultPath. See https://v-
locity.atlassian.net/wiki/display/RAP/Transforms
        */
        if (methodName == 'split')
        {
            List<Object> arguments = (List<Object>)inputs.get('arguments');
            output.put('result', split(arguments));
        }

        return true;
    }


    Map<String, String> split(List<Object> arguments)
    {
        Map<String, String> result = new Map<String, String>();
        String toSplit = (String)arguments[0];
        String splitter = (String)arguments[1];
        List<String> splitList = toSplit.split(splitter);

        for (Integer i = 0; i < splitList.size(); i++)
        {
            if (arguments.size() > i+2)
            {
                result.put((String)arguments[i+2], splitList[i]);
            }
            else
            {
                result.put('Split'+i, splitList[i]);
            }
        }
        return result;
    }
}
```

Chapter 1 DataRaptor Input Mappings

Vlocity DataRaptor is, at its core, a data mapping application. Each DataRaptor Mapping consists of an Interface which contains an Interface Object or JSON data and a set of Data Mappings with a Mapping Detail for each mapped item.

Unlike Salesforce data tools, the data mapping tools in DataRaptor let you insert inbound data into multiple objects with a single import. For example, trouble ticket data from a legacy system can be written across the Account, Contact, and Case objects in Salesforce in a single import.

By using the Lookup and Linked mapping types, DataRaptor moves data from an external source into Vlocity objects. It also creates references between existing objects and objects created during the mapping process.

Configure the DataRaptor Interface

An interface is a container of all elements required to load or extract data via DataRaptor. The Interface name field is the name of the DataRaptor Bundle.

Interface Type options are Load (Object) , Load (JSON) and Extract (JSON).

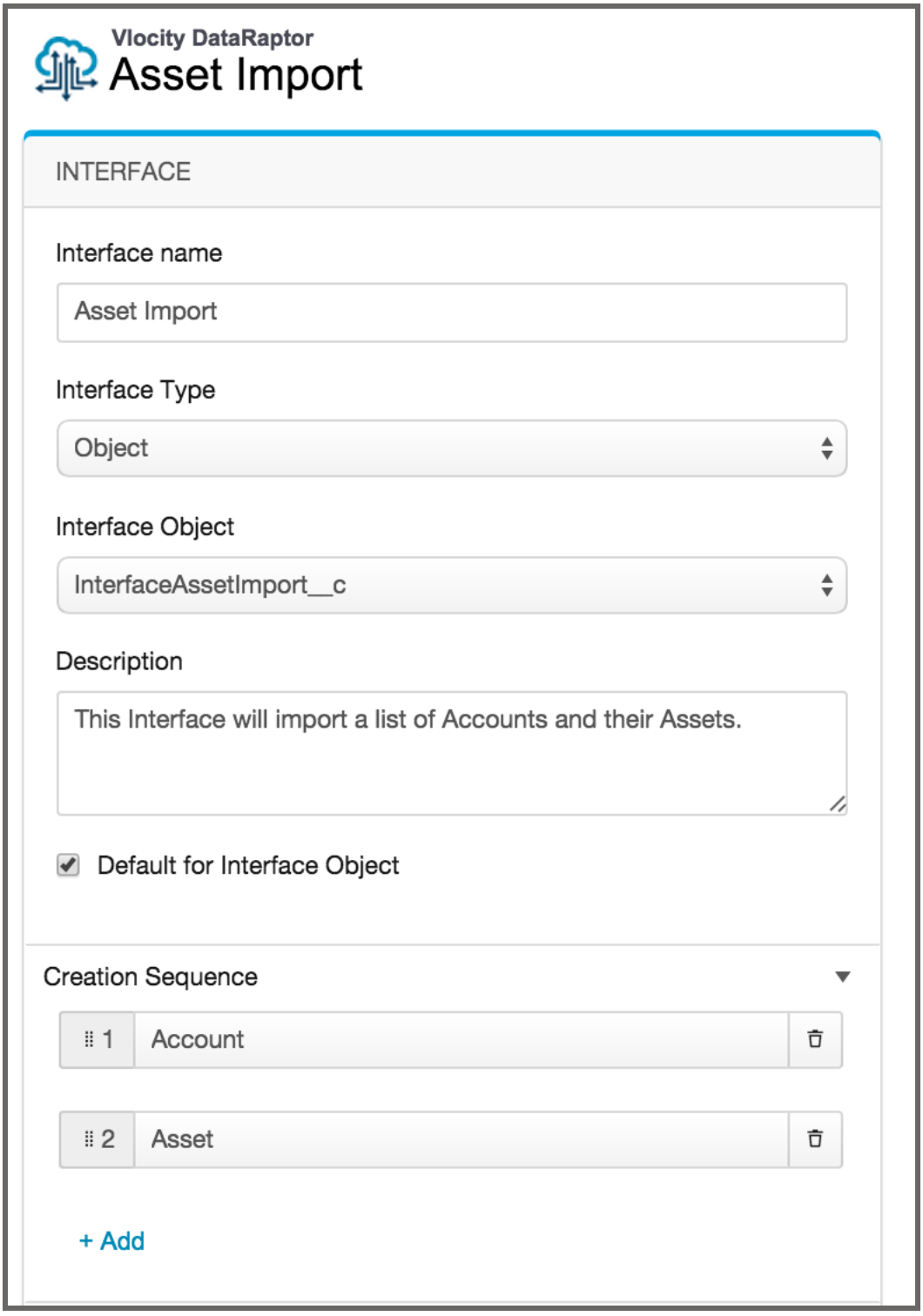
For Object-type interfaces, you need to select the Interface Object you intend to use with the Data Mapping.

The Description is just that, a field to document the details of the interface.

Default for Interface Object indicates that this is the Interface that DataRaptor will use when data is loaded into this Interface Object. There can be multiple Interfaces per Interface Object, but only one default Interface. If you want the Interface Object to use a different Interface, you can specify the Interface in the DataRaptor call.

The Creation Sequence identifies the Salesforce or Vlocity object to which data items will be written, as well as the order in which data items will be written.

Vlocity DataRaptor Asset Import



DataRaptor Map Load JSON Interfaces

The mapping process for Load (JSON) interfaces begins with a sample of the JSON you'll be importing into Salesforce.

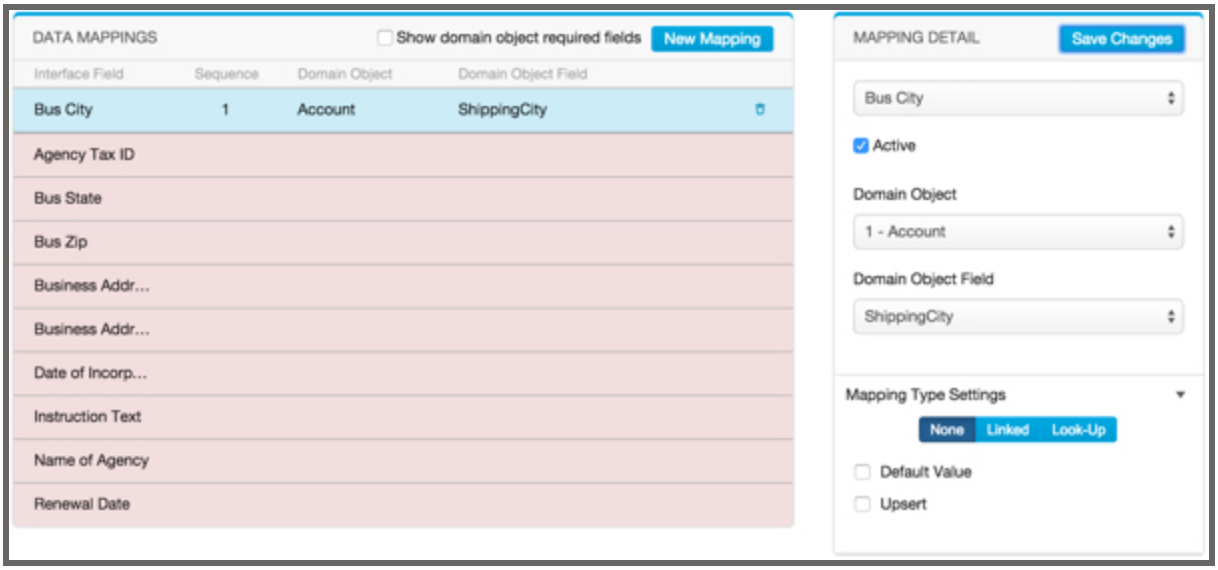
1. Paste your sample JSON in the JSON field of the Interface panel.
2. DataRaptor will pick up the field names from your JSON and prepopulate them into the Interface Field column in the Data Mappings.
3. Select an unmapped field from the Data Mappings table and complete the mapping in the Mapping Detail.

   Mapping denormalized data, data in csv for instance, requires you create an Interface Object. Fields on the Interface Object are then mapped to Salesforce and custom objects.

DataRaptor Data Mapping Types

Vlocity DataRaptor Mappings tell DataRaptor what to do with the data the data it gets, and the order in which to do it.

DataRaptor Data Mappings



In the mapping above, the Interface Field "Bus City" is the source -- whether from an Interface Object or Input JSON. Data in this field will be written in the first Sequence. Data will be written to the Domain Object "Account", to the "ShippingCity" Domain Object Field.

Vlocity DataRaptor supports three types of mapping: Standard, Lookup, and Linked.

Standard Mappings

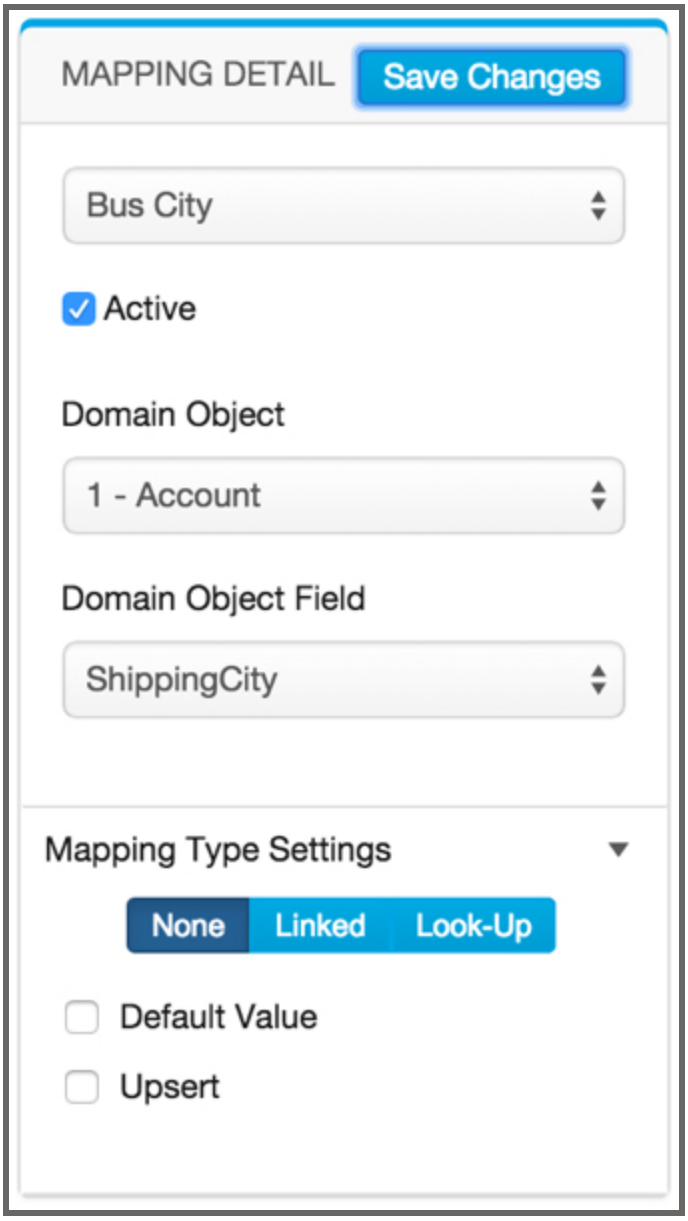
Standard Mappings map data in an Interface Field or JSON to a Domain Object Field. This is the simplest of mappings.

In the example, the data in the Bus City field is written to the ShippingCity field of the Account object. The 1 before the Domain Object indicates it will be processed with the other fields in the first Creation Sequence. If you want to update existing records, you have to check Upsert. Otherwise, Vlocity DataRaptor treats all rows as new records.

When Upsert is checked, data in this field becomes a key DataRaptor uses to determine whether to update or create new records. If multiple fields within a Creation Sequence step have Upsert checked, DataRaptor will have to match all of them to update the record. Otherwise, it creates a new record.

Note: Mapping Detail is not saved until you click **Save Changes**.

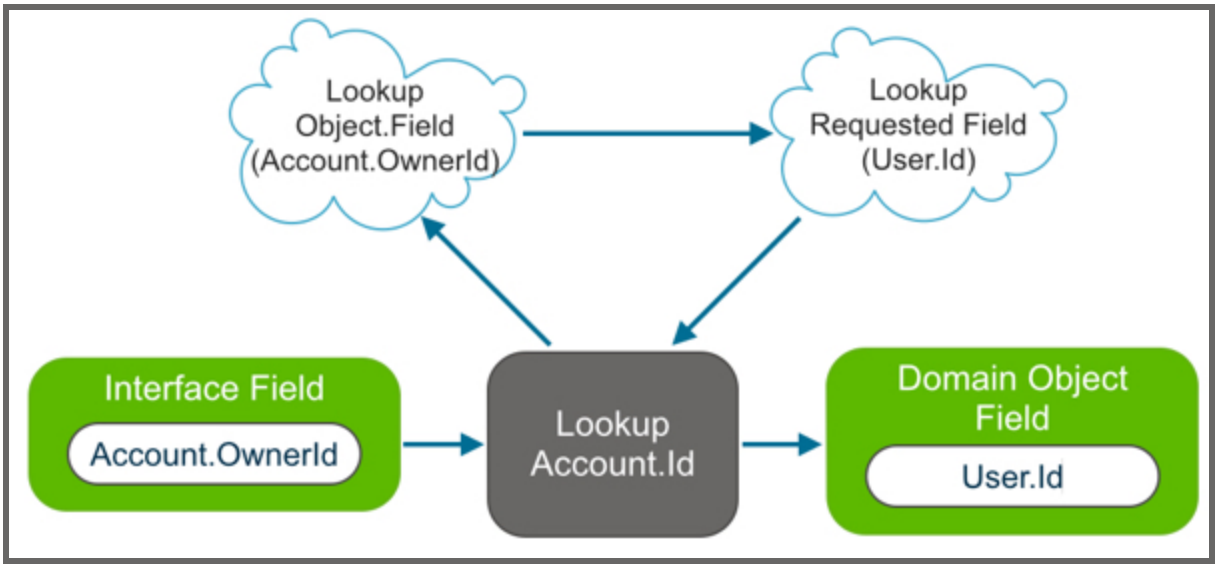
DataRaptor Data Mapping Detail Pane, Save Changes Button



Lookup Mappings

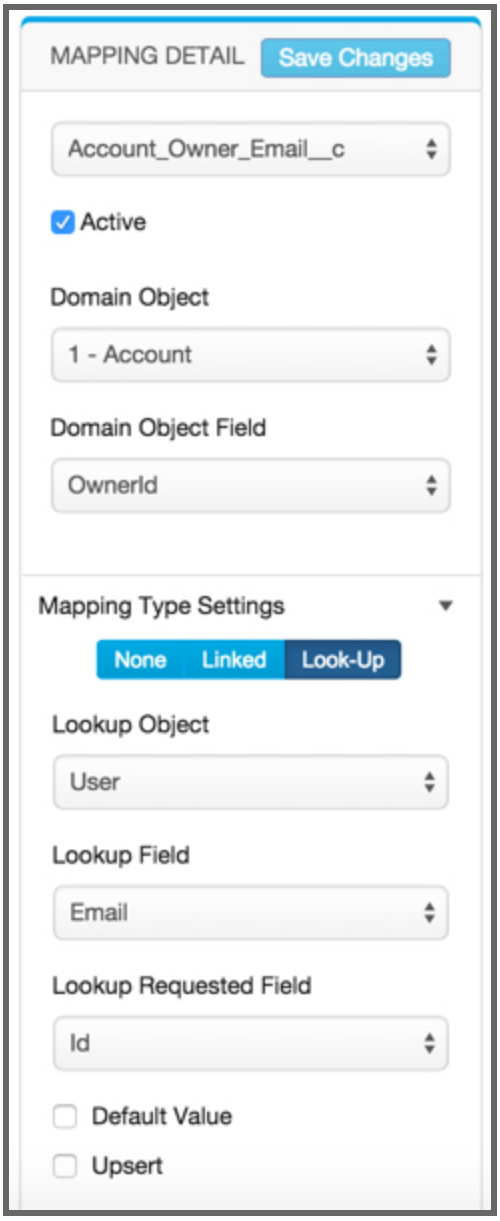
Lookup Mappings query data in SFDC Objects based on Interface Field data and write the returned data into the Domain Object Field.

Diagram



In the example, Vlocity DataRaptor looks up the record where Account_Owner_Email__c = User.Email on the User object. DataRaptor then returns the Id field of that record and writes that to the Account.OwnerId on the Domain Object.

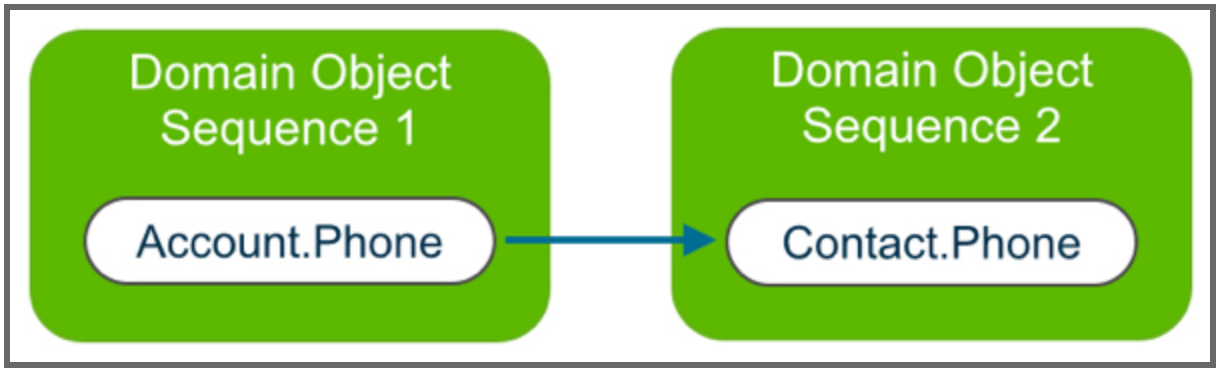
DataRaptor Mapping Detail



Linked Mappings

Linked Mappings map fields from objects created or updated in earlier Creation Sequences. For example, a phone number written to Account.Phone in Creation Sequence 1 can be written to Contact.Phone in Creation Sequence 2.

Domain Object Sequence



A Link Domain Object must be part of an earlier Creation Sequence -- one with a lower Sequence number than that of the Domain Object it links to.

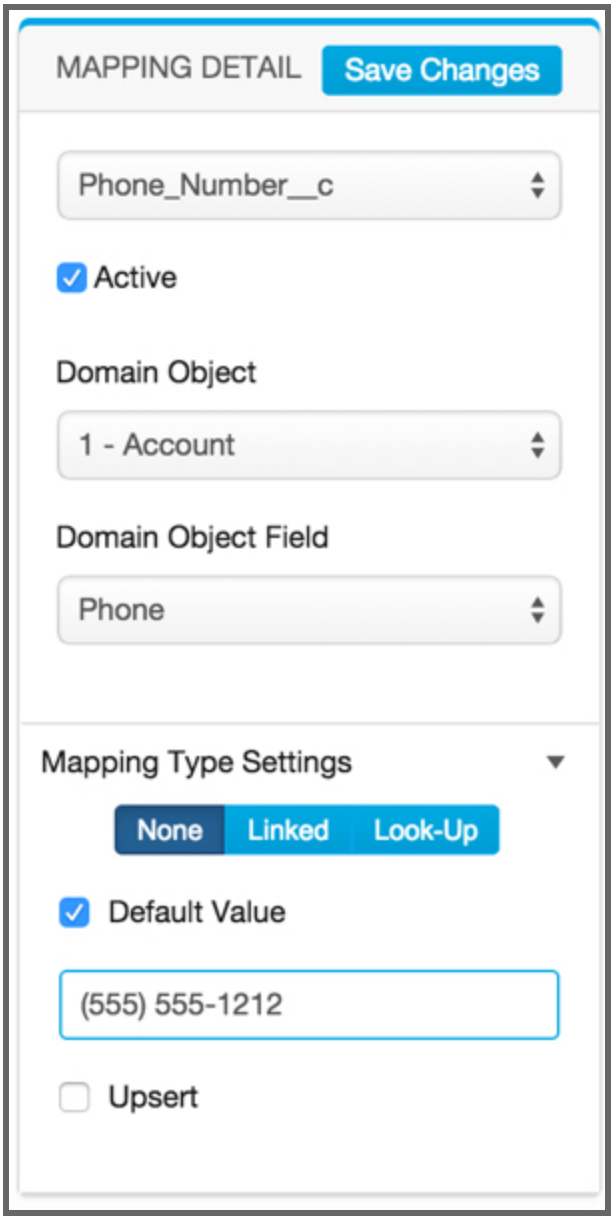
DataRaptor Data Mappings Default Values

When creating Data Mappings in Vlocity DataRaptor, you have the option of specifying a Default Value. If the Interface Field is null, the Default Value is written to the Domain Object Field. For example, if the Phone_Number__c Interface Field is null, the Default Value, (555) 555-1212, will be written to the field.

The Default Value can also overwrite any data already in the Domain Object Field with an empty string if the Interface Field is null. To do this, enter a pair of quotation marks (“”) into the Default Value field.

Note, these behaviors only apply to None mapping types. For Linked and Look-Up mappings, the Default value will be used instead of the input data whether the input contains data or is null.

DataRaptor Mapping Detail Phone_Number__c

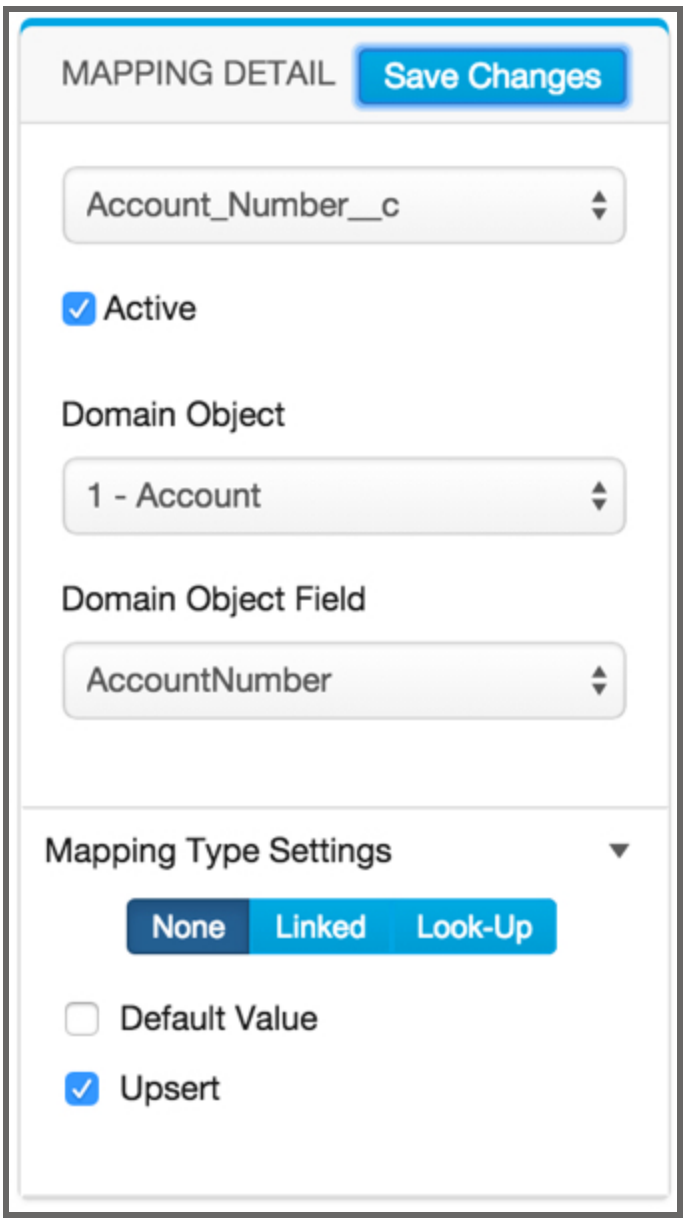


Upsert

The Upsert option identifies the fields DataRaptor will use as key(s) within a Creation Sequence. For example, in the mappings below, Account.AccountName has the Upsert option checked. When DataRaptor imports data, it will treat this field as a key and update records where there's a match. Where there is no match, DataRaptor will create new records.

Upsert mappings are marked with an icon.

DataRaptor Mapping Detail Account_Number__c



Null DataRaptor Fields During Input Mapping

When creating Load DataRaptors, you can automatically overwrite the settings of any field in Salesforce to null by checking the `Overwrite All Null Values` checkbox.

Checking this box will change the value of any fields mapped from the DataRaptor to null.

Complex Input Mappings

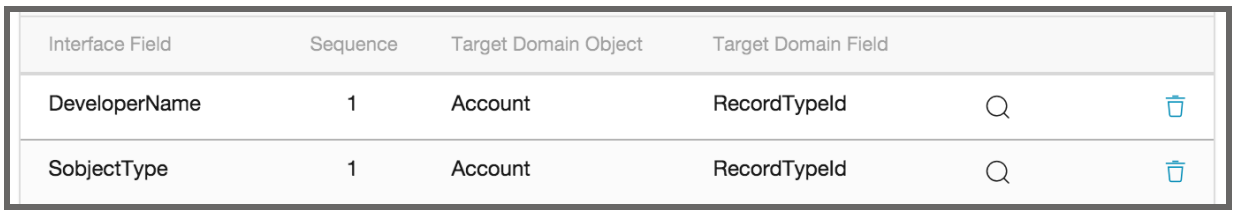
You can use DataRaptor to create complex Lookup, Linked, and standard mappings.

Joined Lookups

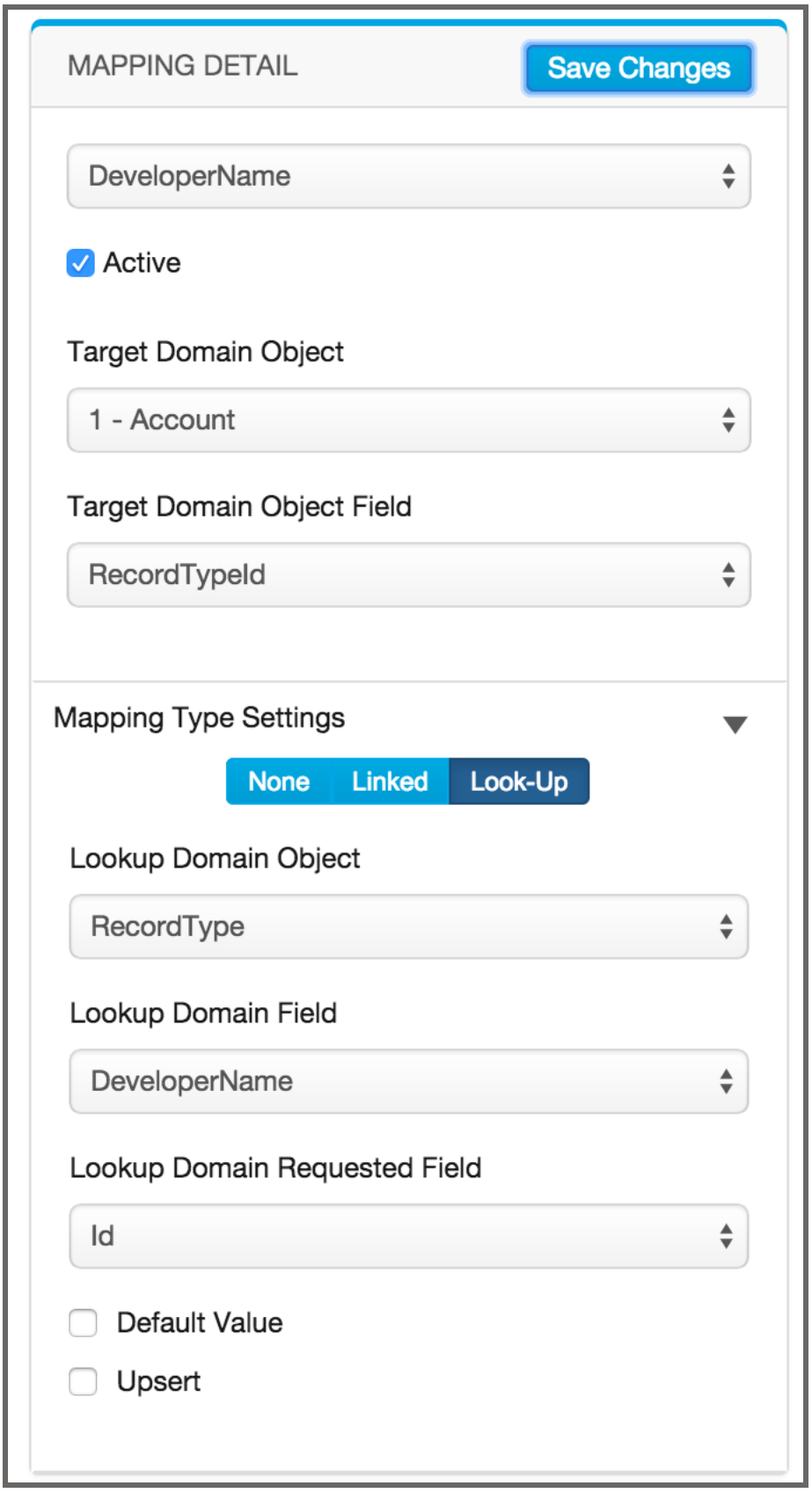
Multiple Lookup Data Mappings with the same Lookup Domain Requested Field for an object will be joined.

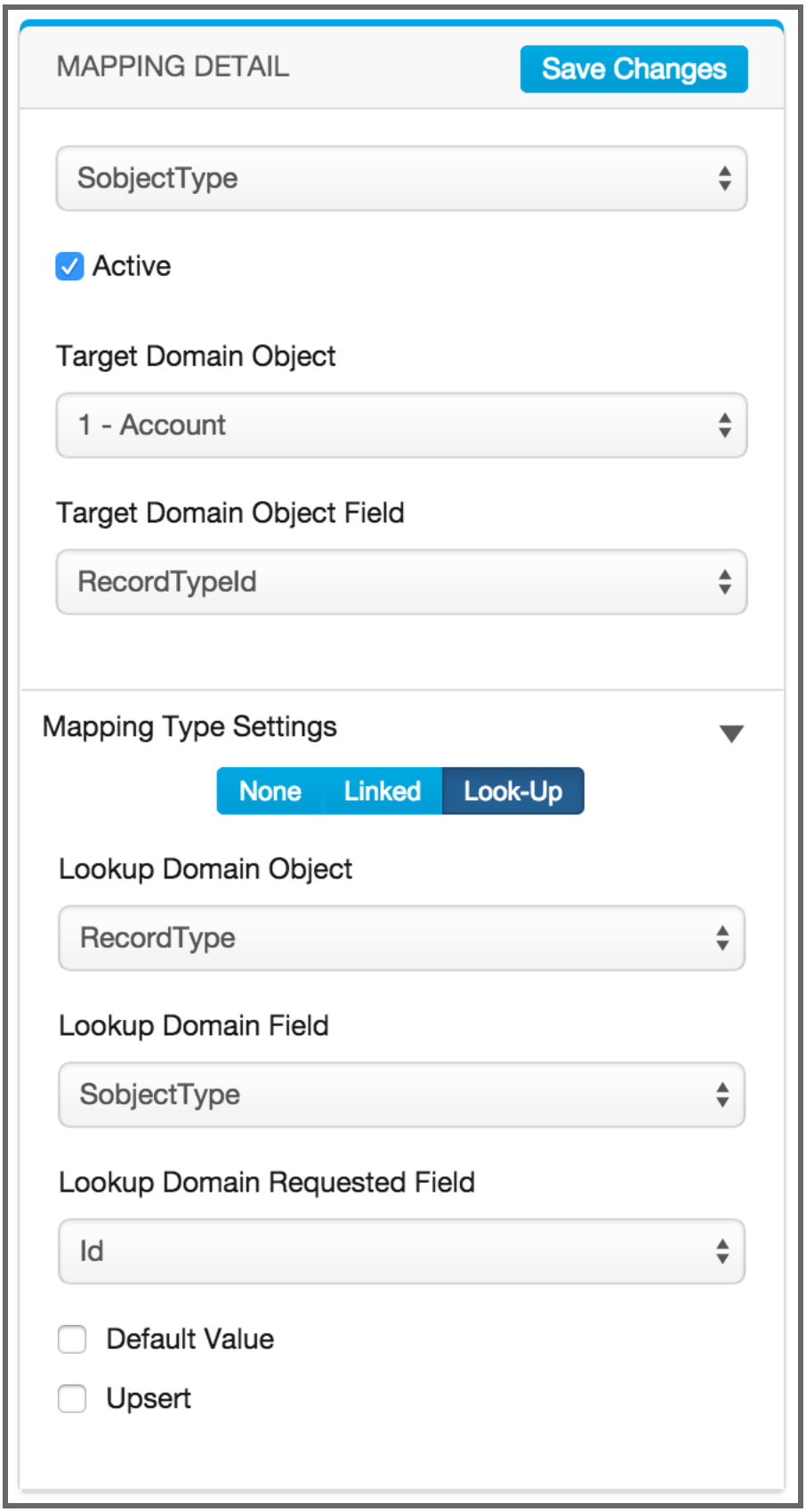
In the following example, the `RecordTypeId` field on the Accounts created in Sequence 1 is populated with `Id` from the RecordType object. Only the `Id` from RecordType that has the unique combination `RecordType.DeveloperName = DeveloperName AND RecordType.SObjectType = SobjectType` is needed.

Developer Name is like the field API name. SObjectType Default would be NS.Object__c, and so on.



Interface Field	Sequence	Target Domain Object	Target Domain Field		
DeveloperName	1	Account	RecordTypeId		
SobjectType	1	Account	RecordTypeId		

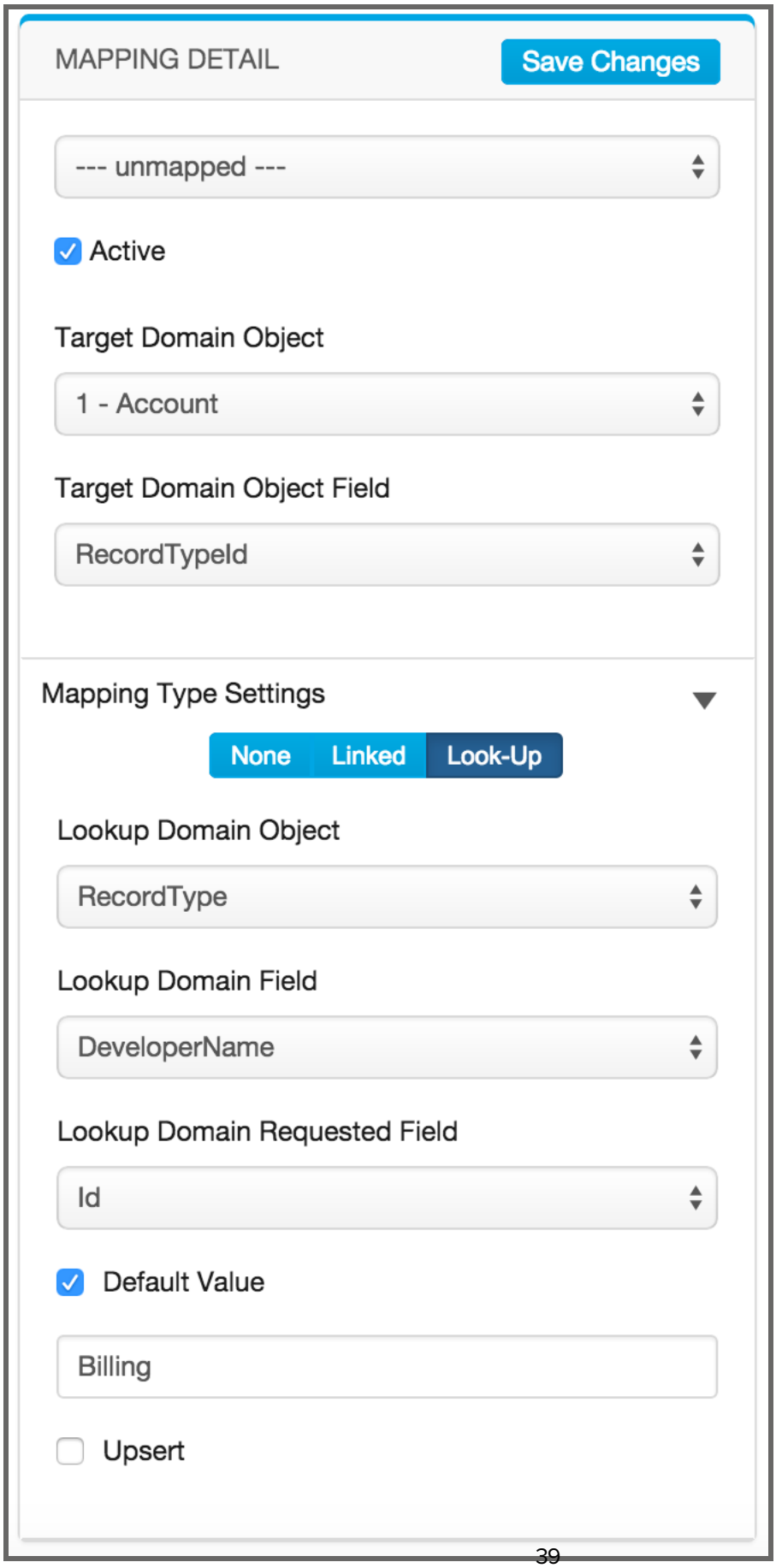

MAPPING DETAIL
Save Changes
DeveloperName
Active
Target Domain Object
1 - Account
Target Domain Object Field
RecordTypeId
Mapping Type Settings
None
Linked
Look-Up
Lookup Domain Object
RecordType
Lookup Domain Field
DeveloperName
Lookup Domain Requested Field
Id
Default Value
Upsert

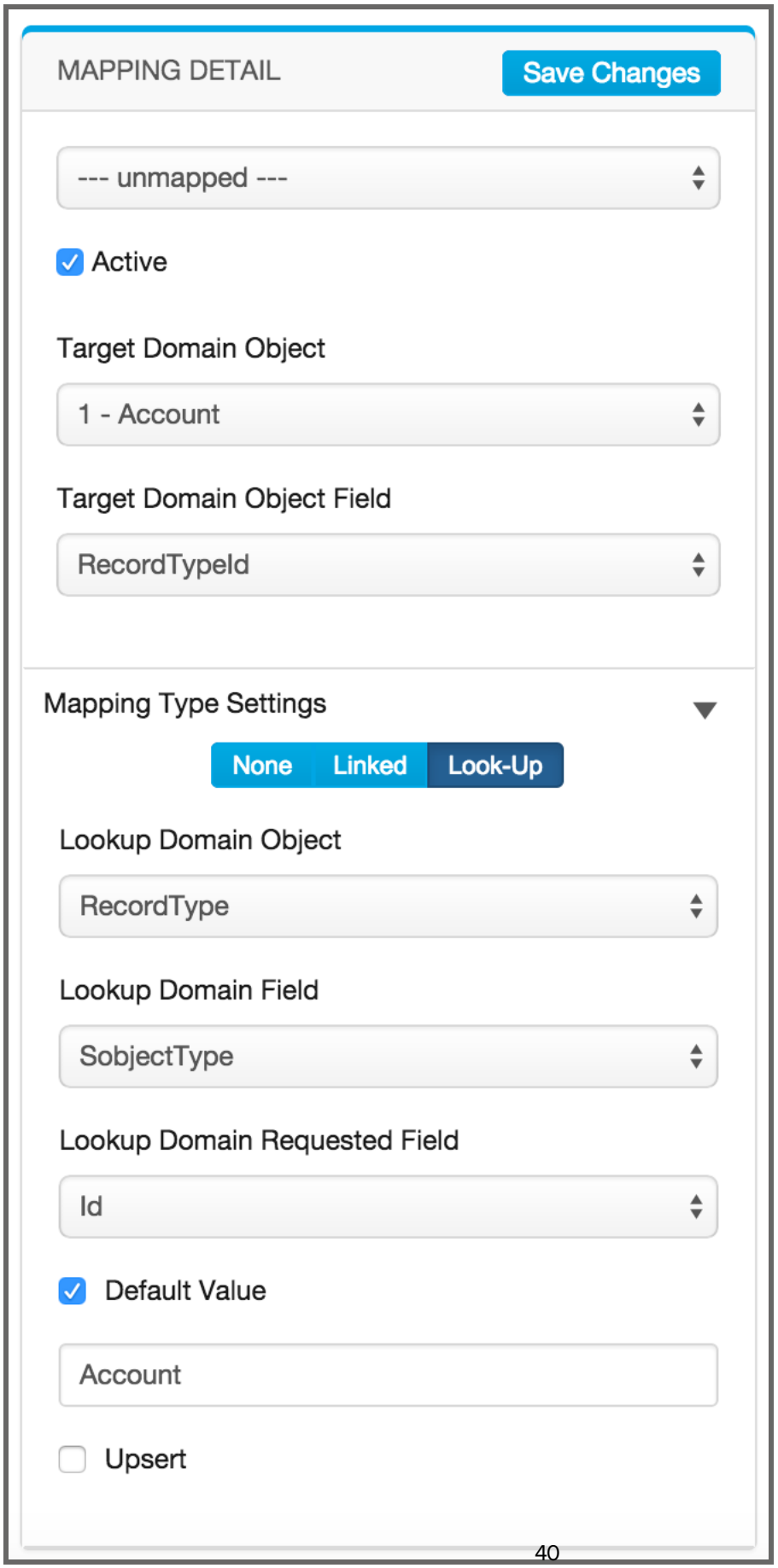

MAPPING DETAIL
Save Changes
SobjectType
Active
Target Domain Object
1 - Account
Target Domain Object Field
RecordTypeId
Mapping Type Settings
None
Linked
Look-Up
Lookup Domain Object
RecordType
Lookup Domain Field
SobjectType
Lookup Domain Requested Field
Id
Default Value
Upsert

Joined Lookups with Default Values

Instead of using an Interface field to query the Domain Object, you can “hard code” a Domain Object field with a Default Value. This is generally because there is no corresponding Interface field.

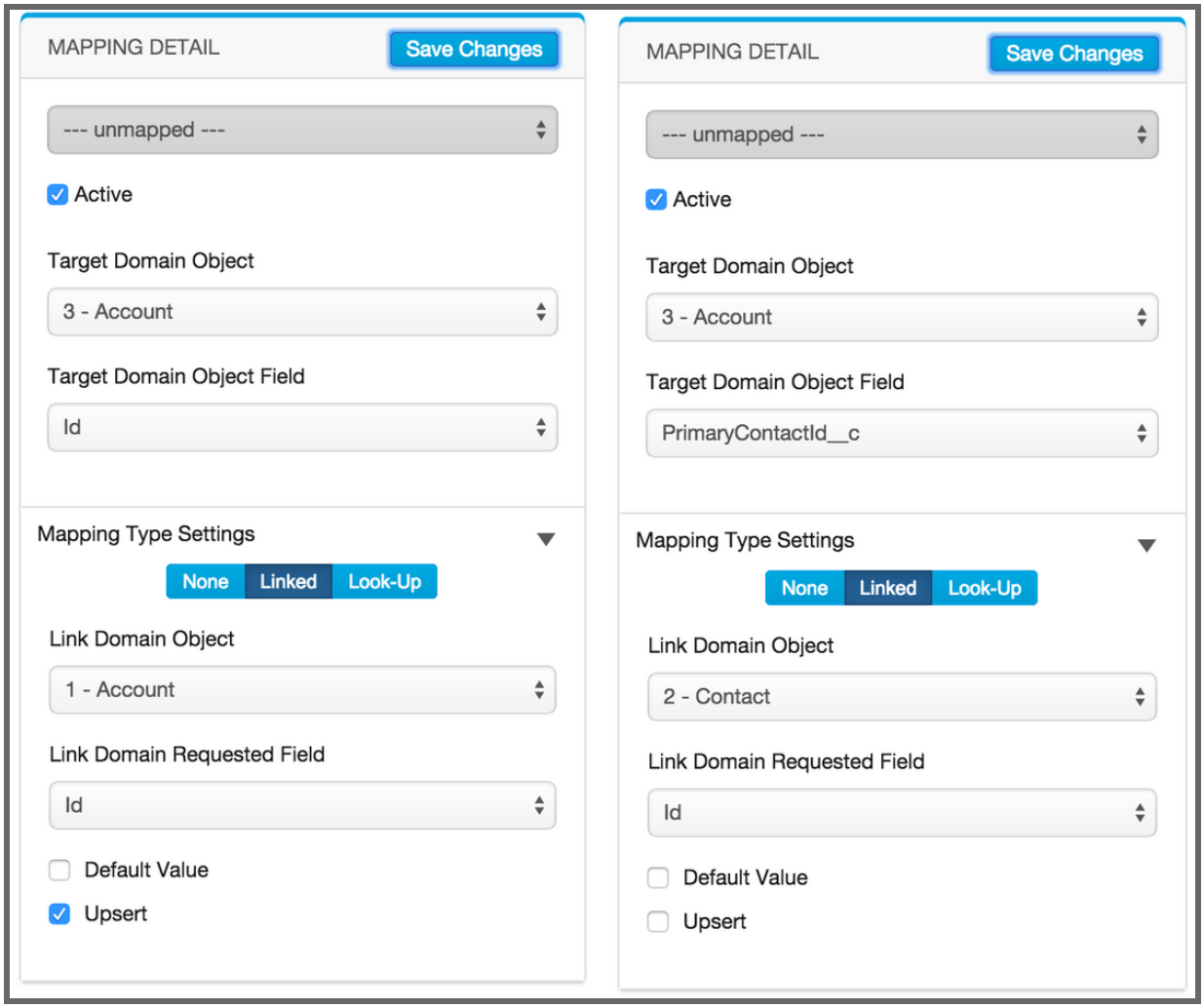
In this mapping, populate the `RecordTypeId` field on all Accounts with the Id from the RecordType object. In this case, `RecordType.DeveloperName = "Billing" AND RecordType.SObjectType = "Account"`. We only want Instead of using an Interface Field to select the Id where RecordType.DeveloperName, we use a Default Value, `Billing`. Id is then mapped in the Account.RecordTypeId Domain Object Field.


MAPPING DETAIL
Save Changes
--- unmapped ---
Active
Target Domain Object
1 - Account
Target Domain Object Field
RecordTypeId
Mapping Type Settings
None
Linked
Look-Up
Lookup Domain Object
RecordType
Lookup Domain Field
DeveloperName
Lookup Domain Requested Field
Id
Default Value
Billing
Upsert


MAPPING DETAIL
Save Changes
--- unmapped ---
Active
Target Domain Object
1 - Account
Target Domain Object Field
RecordTypeId
Mapping Type Settings
None
Linked
Look-Up
Lookup Domain Object
RecordType
Lookup Domain Field
SobjectType
Lookup Domain Requested Field
Id
Default Value
Account
Upsert

Internal Linked Mappings

In this scenario, we need to update an object created in an earlier sequence.



In this flow:

1. Sequence 1: Account is created.
2. Sequence 2: Contact is created.
3. Sequence 3: Account from Sequence 1 is updated with Contact information from Sequence 2.

To do this, create a Linked Upsert mapping which links the account from Sequence 1 to Sequence 3. Then create a second Linked mapping to put the Contact.Id field from Sequence 2 into any field on Account.

Chapter 1 Add and Map Custom Fields (Using Load Object)

Vlocity DataRaptor is preconfigured to map data into Vlocity products. If you want to import custom attributes that do not already exist in the Domain Object, you must update the DataRaptor Bundle to include those fields. You can write an Apex class that implements the IRaptorPreprocessor Interface to modify data before it is imported to the Interface Object. For more information, see "Create a Preprocessor Class" on page 9.

Add Custom Fields to Domain Objects

To add a field to a custom object:

1. Log in to Vlocity as an administrator.
2. From Setup, click **Create**, and then click **Objects**.
3. Select the object to which to add the field.
4. Scroll to Custom Fields & Relationships and click **New**.
5. Select the `Data Type` for the field and click **Next**.
6. Enter information for the field.
7. Click **Save**.

Add Custom Fields to the Interface Object

Now that you've added the custom fields to the object in Salesforce, you need to add the same field to the intermediary Interface Object.

To add new fields to the Interface Object:

1. Within the DataRaptor App, from Setup, click **Create**, and then click **Objects**.
2. Select the Interface Object to which to add fields.

3. Scroll to Custom Fields & Relationships and click **New**.
4. Add the new field.
5. Click **Save**.

Map Custom Fields

Now that both the Domain Object and Interface Object have the new custom fields, you must create a mapping between the two. Learn more about mappings in DataRaptor Mappings to determine the type of Map Item you want to use and then go to the DataRaptor tab and select a mapping.

Chapter 1 Importing Data Using an Interface Object

Data can be imported through a DataRaptor Interface Object.

This section includes the following topics:

Preparing Your Source Data

Prepare your source data to make sure it is compatible with Salesforce standards and can be written as needed across Salesforce objects.

You will also need to know how you're planning to map your source data across Salesforce objects. If data needs to be manipulated before entering the Interface Object, you can write a preprocessor Apex class that implements the IRaptorPreprocessor Interface. This Apex Interface runs before insert on each record uploaded to the Interface Object. For instance, a name field formatted "Last Name, First Name" in a CSV file can be split into the Salesforce Contact standard `firstName` and `lastName` fields, using this custom Apex class. No preprocessor class is necessary if the data is already in the correct format before import.

Creating an Interface Object in Salesforce

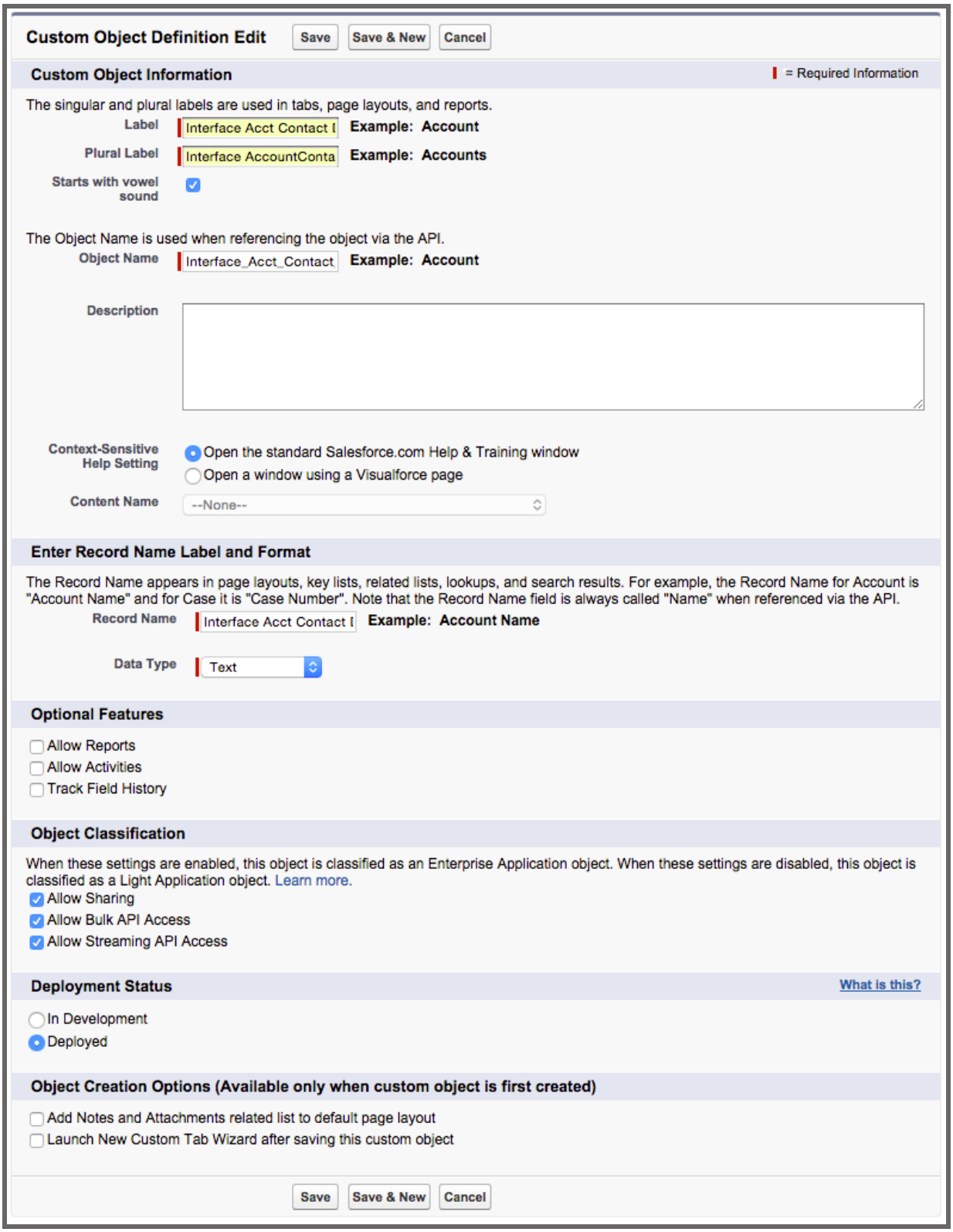
Interface Objects are staging objects that store incoming data. Once in the staging object, source data can then be mapped to Domain Objects—for example, Account, Case, Contact, Household. The Interface Object generally includes one field for every field in the source data.

In addition to creating one field for every field in the source data, all Interface Objects in the data integration module must include these four required fields:

Field Label	Field Name	Data Type	Description
DRBundleName	DRBundleName	Text Area (255)	Determines which bundle to use when an Interface Object is being used to import data.
DRProgressData	DRProgressData	Long Text Area	JSON of the objects currently created by DataRaptor during integration processing.
DRStatus	DRStatus	Text Area (255)	Integration status flag that denotes the current status of the integration (Ready, Error, Success, In Progress)
DRError	DRError	Long Text Area	Describes errors during import, including DML errors, duplicate records found for upsert, and field validation rules.

To create a new Interface Object:

1. From Setup, click **Create**, and then click **Objects**.
2. Click **New Custom Object**.
3. In the `Label` field, enter the name of the object. Vlocity recommends using a naming format similar to *`Interface ObjectName`*.
4. In the `Plural Label` field, enter the plural name of the object.
5. Click **Save**.


Custom Object Definition Edit
Save
Save & New
Cancel
Custom Object Information
= Required Information
The singular and plural labels are used in tabs, page layouts, and reports.
Label
Interface Acct Contact I
Example: Account
Plural Label
Interface AccountConta
Example: Accounts
Starts with vowel sound
The Object Name is used when referencing the object via the API.
Object Name
Interface_Acct_Contact
Example: Account
Description
Context-Sensitive Help Setting
Open the standard Salesforce.com Help & Training window
Open a window using a Visualforce page
Content Name
--None--
Enter Record Name Label and Format
The Record Name appears in page layouts, key lists, related lists, lookups, and search results. For example, the Record Name for Account is "Account Name" and for Case it is "Case Number". Note that the Record Name field is always called "Name" when referenced via the API.
Record Name
Interface Acct Contact I
Example: Account Name
Data Type
Text
Optional Features
Allow Reports
Allow Activities
Track Field History
Object Classification
When these settings are enabled, this object is classified as an Enterprise Application object. When these settings are disabled, this object is classified as a Light Application object. Learn more.
Allow Sharing
Allow Bulk API Access
Allow Streaming API Access
Deployment Status
What is this?
In Development
Deployed
Object Creation Options (Available only when custom object is first created)
Add Notes and Attachments related list to default page layout
Launch New Custom Tab Wizard after saving this custom object
Save
Save & New
Cancel

Adding Fields to Your Interface Object

After you've created your Interface Object, the next step is to prepare the interface for the data you'll be loading.

1. If you aren't there already, navigate to the Custom Object page for your Interface Object.
2. Scroll down to the Custom Fields & Relationships section.
3. Click **New** to add a new field.
4. Create one field for each field you're planning to import.

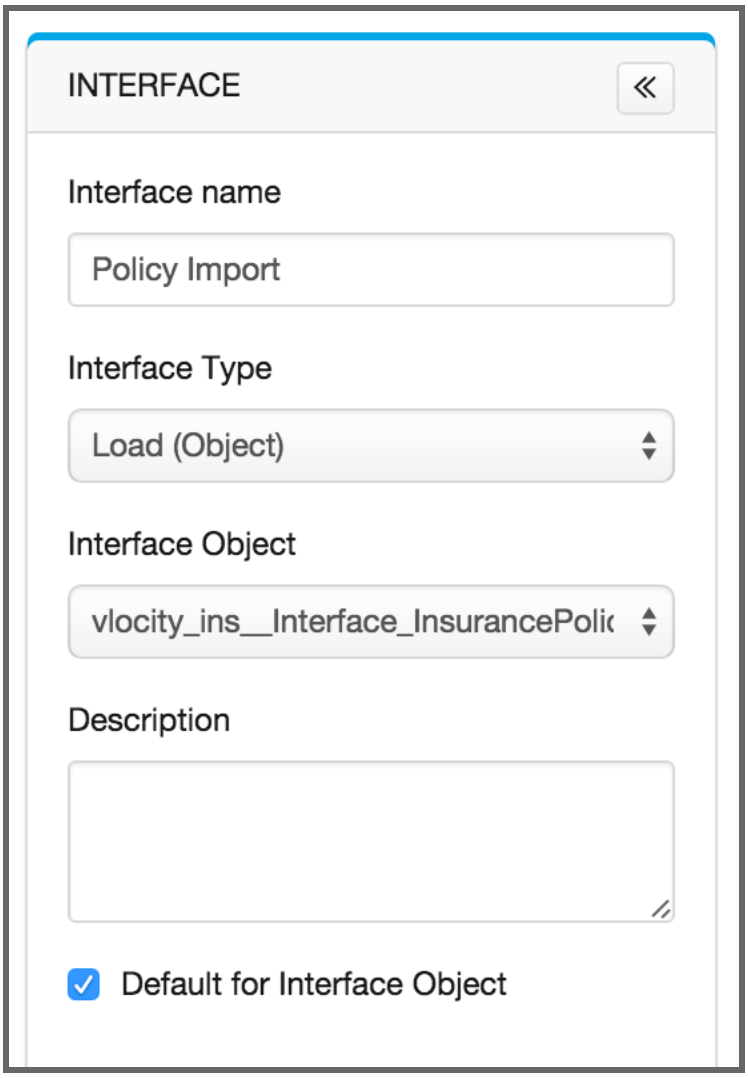
> **Note:** To avoid data loading issues, Vlocity recommends adding Text fields on the Interface Object to hold incoming data. Additionally, field names should match column headers on the incoming file.

5. Create the four required fields:
   - `DRBundleName__c`
   - `DRError__c`
   - `DRProgressData__c`
   - `DRStatus__c`

> **Note:** You can map your source data to both custom and standard objects, and to custom and standard fields on those objects. Custom objects and custom fields on standard objects should be set up before you configure the integration.

Creating a New DataRaptor Interface

After modifying the source data and creating the Interface object, the next step is to configure the new DataRaptor Interface. The interface is a container for configuration and mappings.

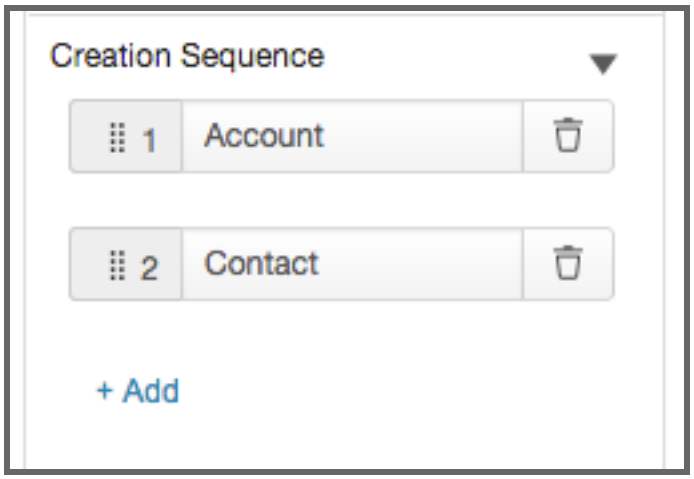


1. From the Vlocity DataRaptor tab, click **New**.
2. In the `Interface Name` field, enter a name for the Interface.
3. In the `Interface Type` field, select `Load (Object)`.
4. In the `Interface Object` field, select the Interface object that you created earlier. If the Interface Object does not appear in the list, make sure the required fields (`DRBundleName__c`, `DRError__c`, `DRProgressData__c`, or `DRStatus__c`) have been created and are configured correctly.
5. Add an optional description for the Interface.
6. Select `Default for Interface Object` if this Interface is the default for the

Interface Object. Each Interface Object can only have one default Interface.

7. Click **Save**.

Setting the Creation Sequence



The order in which DataRaptor creates or updates Domain Objects is defined in the Creation Sequence. You set the Creation Sequence when you configure the data mapping for your Interface.

For example, in this Creation Sequence DataRaptor writes data to Account fields first, and then writes to the Contact fields.

To set the Creation Sequence:

1. From the `Creation Sequence` section of the Interface, click **+Add**.
2. Select the first Domain Object in the sequence.
3. Repeat Steps 1 and 2 to build the sequence of objects to create.

Once the Creation Sequence is in place, you can begin mapping from the Interface Object to Salesforce Objects.

Creating Data Mappings From the Interface Object to Salesforce Objects

After setting the Creation Sequence, you can create the mappings in the Data Mappings table. Unmapped rows are red. All the fields you created on the Interface Object appear on the left side of the table. Click on a mapping to select it (the blue row) and to open a mapping dialogue on the right side of the screen.

Refer to Mapping Data to Salesforce for more information about types of mappings.

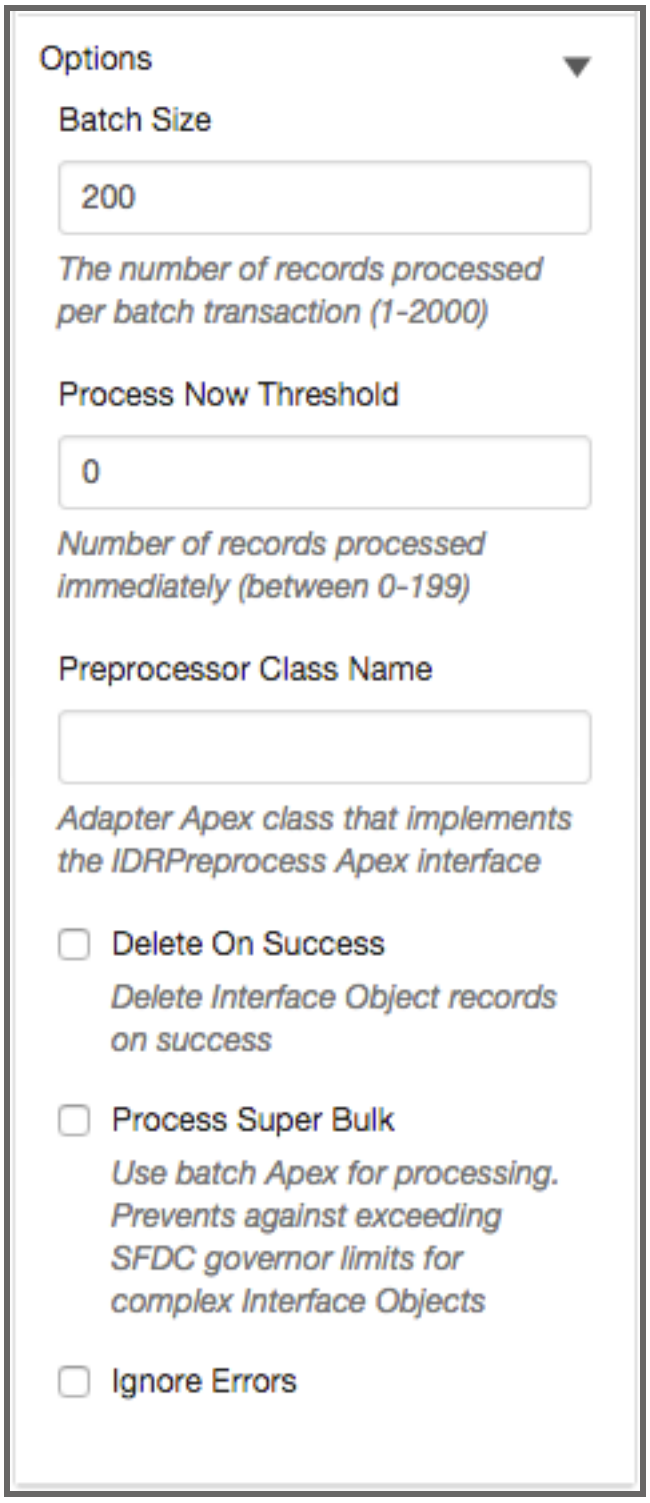
To create a new mapping:

1. Click a row in the Data Mappings table.
2. In the Mapping Detail section, select the Domain Object that you want to map the field to. Each Domain Object in the list is preceded by its Creation Sequence.
3. Select the `Domain Object Field` that the `Interface Field` should map to.
4. Adjust the Field Settings.
5. Click **Save Changes**.
6. Repeat for the remaining unmapped fields.
7. Click **Save**.

Configuring the Data Processing Settings for the Interface

After data is successfully uploaded into the Interface Object, several more DataRaptor configuration options are available. If the data is between 1-199 records, it can be processed immediately by setting the Process Immediately Threshold DataRaptor Bundle Setting. Uploads of greater than 200 records will be processed inside Batch Apex, with a number of additional settings for processing defined in the DataRaptor Bundle.

From the Options section, you can set additional processing options for the Interface:


Options
Batch Size
200
The number of records processed per batch transaction (1-2000)
Process Now Threshold
0
Number of records processed immediately (between 0-199)
Preprocessor Class Name
Adapter Apex class that implements the IDRPreprocess Apex interface
Delete On Success
Delete Interface Object records on success
Process Super Bulk
Use batch Apex for processing. Prevents against exceeding SFDC governor limits for complex Interface Objects
Ignore Errors

- `Batch Size`: The number of records processed per batch transaction.
- `Process Now Threshold`: Number of records processed immediately (between 0-199).
- `Preprocessor Class Name`: Adapter Apex class that implements the IDRPre-process Apex interface. Used for massaging imported data.
- `Delete on Success`: Deletes the Interface Object records on success.
- `Process Super Bulk`: Use batch apex for processing. Prevents against exceeding Salesforce governor limits for complex Interface Objects.
- `Ignore Errors`: Ignores all errors during processing.

Note: Clear `Delete of Success` while you're in development. Data loaded into the Interface Object will be kept, making troubleshooting easier.

Implementing a Trigger on the Interface Object

DataRaptor implements a simple Apex trigger on the Interface Object to process records upon upload. To create a new Apex Trigger:

1. From Setup, click **Build**, and then click **Create**, and then click **Object**.
2. Click the Interface Object that you created and scroll down to the Triggers section.
3. Click **New**.
4. Modify the Apex script to fit your deployment.

```
trigger Name on IO (before insert, after insert, before update, after
update)
{
NS.DRGlobal.triggerHandler(Trigger.new);
}
```

- *Name* is the name of the Trigger. This is a label and can be anything.
- *IO* is the Interface object API name—for example, `InterfacePolicy__c`.
- *NS* is the name space prefix of the installed package. The name space is for the package is found in Setup | Installed Packages.

Testting Import Data and Upload

Before bulk data upload, test your Interface by uploading a small amount of data. Use the data loader of your choice to import the data and then check your Domain Objects to see if the data mapped correctly.

Enter a simple query in the Developer Console to find any errors, for example:

```
Select DRProgressData__c, DRError__c from IO where DRStatus__c = 'Error'
```

Where *IO* = Interface object API name—for example, `InterfacePolicy__c`.

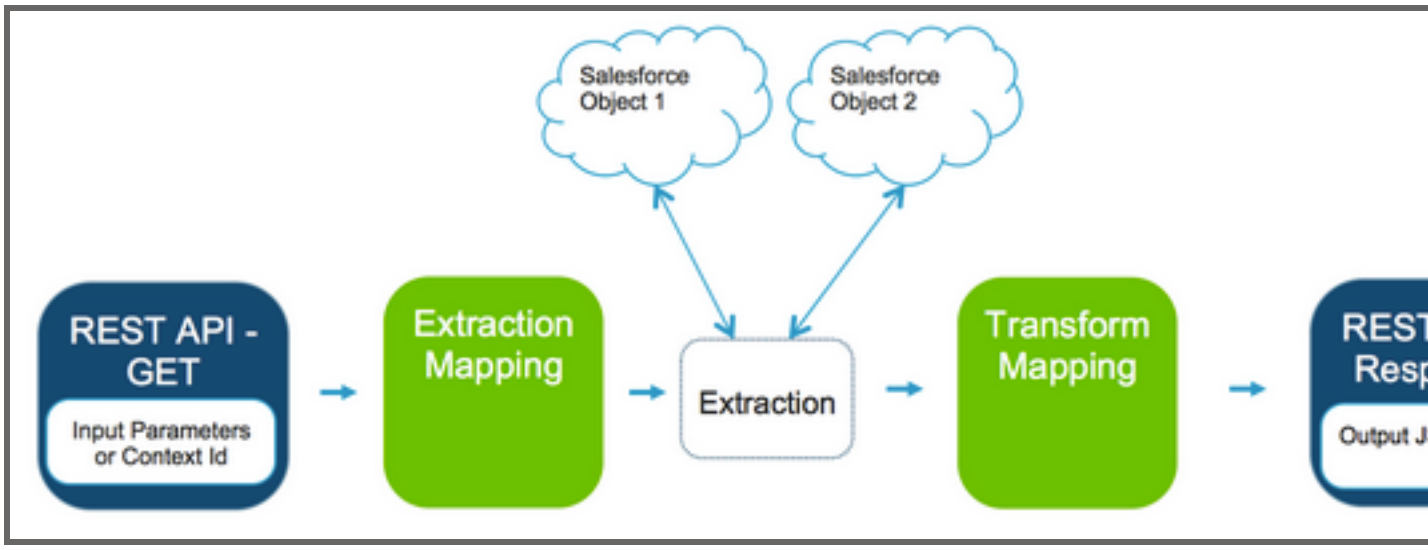
There are many third-party data loading tools available for the Salesforce1 Platform™. For loading data into the Vlocity DataRaptor engine, Vlocity has successfully tested and used the JitterbitCloud Data Loader for Salesforce.

See Running and Monitoring DataRaptor Jobs for more information about monitoring progress and checking errors.

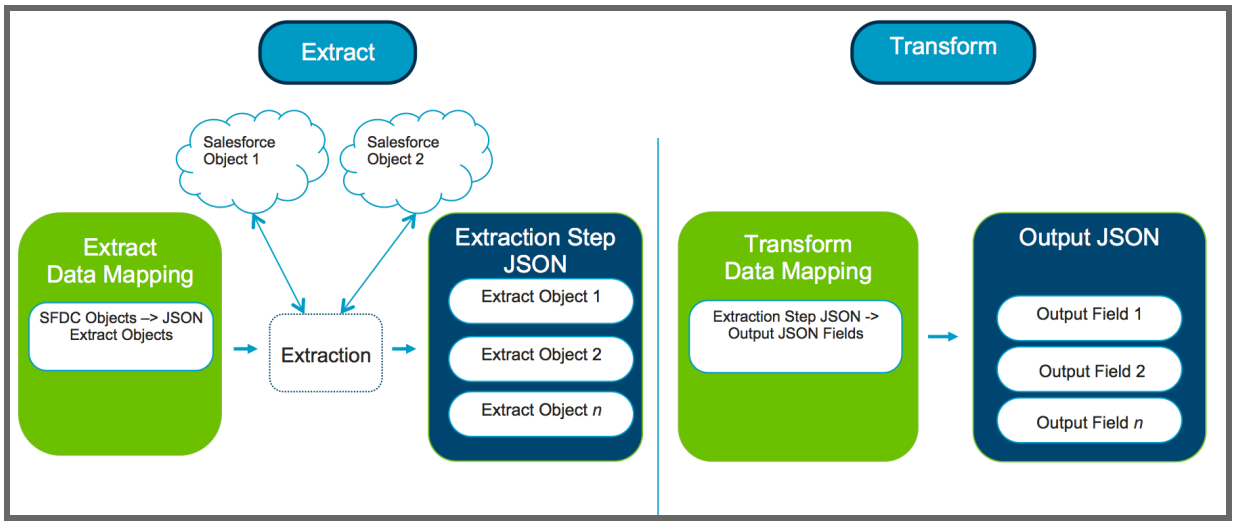
After successfully importing data, your DataRaptor Interface is ready for bulk upload. Use the data loader of your choice to proceed.

DataRaptor Extract

Using Data Raptor Extract, you can extract data by passing input parameters through a REST API into DataRaptor. DataRaptor queries Salesforce and Vlocity objects and returns intermediary extraction data in the form of a JSON blob. Transform the extraction JSON data into a user-defined output JSON and return it through the API.



Create an Extract (JSON) Interface



To create an Output Interface and extract data from objects:

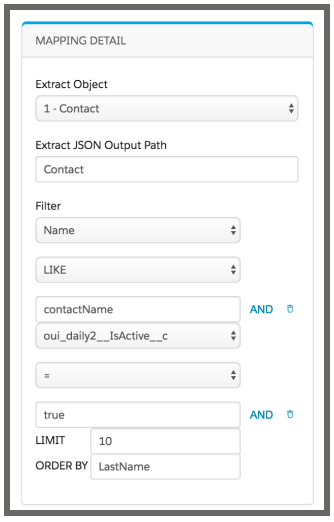
1. From the Vlocity DataRaptor tab, click **New**.
2. Enter an Interface name and select `Extract (JSON)` from the `Interface Type` field.
3. In the Extract Object Order section, click **+Add**.
4. Select the first Salesforce or Vlocity object from which to extract data.

5. In the `Extract JSON Output Path`, enter the name of the object as it should appear in the Extraction Step JSON.

   For example, AccountExtract appears in the Extraction Step JSON as:

6. The Filter defines a query that pulls data from the specified Extract Object in Salesforce and inserts it into the Extraction Step JSON at the user-defined key(Extract JSON Output Path). Choose the field on the Extract Object that you want to filter, for example LastName. In the next field, enter the filter value that will be passed in with the Input JSON (API).

7. Click **AND** to join more than one clause—for example, this query selects all fields from Contact where Name is like `contactName` AND `IsActive__c = true`. It limits the results to 10 and orders the results by LastName. `ContactName` is passed in from the input JSON of an OmniScript and the query will populate the Extract JSON with up to 10 Contacts that meet the criteria:



8. Repeat for additional Extract Objects. To reference a field from a previous Extract Object as your filter value, use the format *`Extract JSON Output Path:Field Name`*.

Use Filters with DataRaptor Extract

Filters can be used with DataRaptor Extracts to refine the type, order, and number of responses.

Choose the field on the Extract Object that you want to filter and enter the filter value. Filters can be used with fields, text values, and Environment Variables. If you are filtering by a text value or an environment variable, you must include quotation marks around the value. For example, if you use the environment variable for today's date you would enter `"$Vlocity.TODAY"` in the field.

To reference a field from a previous Extract Object as your filter value, use the format *`Extract JSON Output Path:Field Name`*. For example, `Case:Account`.

The following operators can be used with DataRaptor filters:

- =
- <>
- <
- >
- <=
- >=
- LIKE
- NOT LIKE
- INCLUDES
- EXCLUDES

INCLUDES and EXCLUDES apply only to Salesforce multi-select picklists. When using either LIKE or NOTLIKE as the operator, make sure to apply the % symbol to both sides of the value. For more information on operators, see this Salesforce documentation.

AND can be used to join multiple clauses together.

You can use the `LIMIT` option to limit the number of responses you receive. For example, if you enter `5` this would limit the number of records extracted to five. The `ORDER BY` option determines how to sort and display the information. It is a free text field and can accept any valid SQL ORDER BY clause. For more information on ORDER BY, see this Salesforce documentation.

Output Data

After you configure the Extract Objects and enter any formula mappings, the final step is to map data from the resulting Extraction Step JSON into the final Output JSON.

To transform data from the Extraction Step JSON into an Output JSON:

1. From the Output Interface, click **Output**.
2. Click **New Mapping**.
3. In the `Extract JSON Path`, select the field from the Extraction Step JSON to move to the Output JSON.
4. In the `Output Path`, enter the field name as it should appear in the Output JSON.
5. Repeat for additional Output mappings.

Chapter 1 Run, Test, and Monitor DataRaptor Jobs

Run DataRaptor

This topic explains how to run DataRaptor for each Interface Type.

To run DataRaptor:

1. Go to the DataRaptor tab and click **Start All DataRaptor Jobs**. All running jobs appear in the Batch Jobs section.
2. Click **Show Batch Jobs** to see a list of current and past jobs.

Object

Create an Apex trigger on the Interface Object and load data. See Object Input for more information.

Rest - Input

DataRaptor runs when JSON is posted through an API.

Request URL

/services/apexrest/*NS*/v2/DataRaptor/

NS is the Name space of the installed package.

Parameters

Param Name	Description	Optional	Example
bundleName	The DataRaptor Bundle Name	REQUIRED	Create%20Account
objectList	JSON Object Data to be processed	REQUIRED	
filesMap	Map of keys to base64 encoded files	OPTIONAL	
bulkUpload	Should this use Batch Apex. Default: false	OPTIONAL	

Example Request

POST

```
/services/apexrest/vlocity_cmt/v2/DataRaptor/
```

POST Data

```
{
"bundleName" : "AccountUpload",
"objectList" : {
"Agency Information": {
"Agency Name": "Vlocity",
"Agency Address": "50 Fremont",
"Agency City": "San Francisco",
"Agency State": "CA",
"Agency Zip": "94110",
}
   },
   "bulkUpload" : false
}
```

Example Result

```
{
   "createdObjectsByOrder": {
        "Open Account": {
             "1": [
                  "a10o00000022xVEAAY"                ]
        }
   },
   "createdObjectsByType": {
        "Open Account": {
             "Account": [
                  "a10o00000022xVEAAY"              ]
        }
   },
   "errors": {},
   "returnResultsData": []
}
```

Rest - Output

There are two ways to extract data using DataRaptor, through a ContextId or using URL parameters.

ContextId

Request URLRequest URL- /services/apexrest/NS/v2/DataRaptor/${BundleName}/${ContextId}

NS is the Name space of the installed package.

Parameters

Param Name	Description	Optional	Example
BundleName	The DataRaptor Bundle Name	REQUIRED	Open%20Cases
ContextId	String value passed in	REQUIRED	a10o00000022xVE

Example Request

```
GET
/services/apexrest/vlocity_cmt/v2/DataRaptor/Open%20Cases/a10o00000022xVE
```

Example Result

```
{
  "Contact": {
     "Contact Name" : "Dennis Reynolds",
     "Case Information": [
       {
           "Title": "This is the subject of Case #1"       },
       {
          "Title": "This is the subject of Case #2"       },
       {
          "Title": "This is the subject of Case #3"       }
    ]
 }
}
```

URL Parameters

Request URL

/services/apexrest/NS/v2/DataRaptor/${BundleName}/?${Param1}=${Val1}&${Param2}=${Val2}

NS is the Name space of the installed package.

Parameters

Param Name	Description	Optional	Example
BundleName	The DataRaptor Bundle Name	REQUIRED	Open%20Cases
URL Params	URL Parameters will be converted into JSON and sent through to the specified bundle	REQUIRED	

Example Request

```
GET
/services/apexrest/vlocity_cmt/v2/DataRaptor/Open%20Cases/?FirstName-
e=Dennis&LastName=Reynolds
```

Example Result

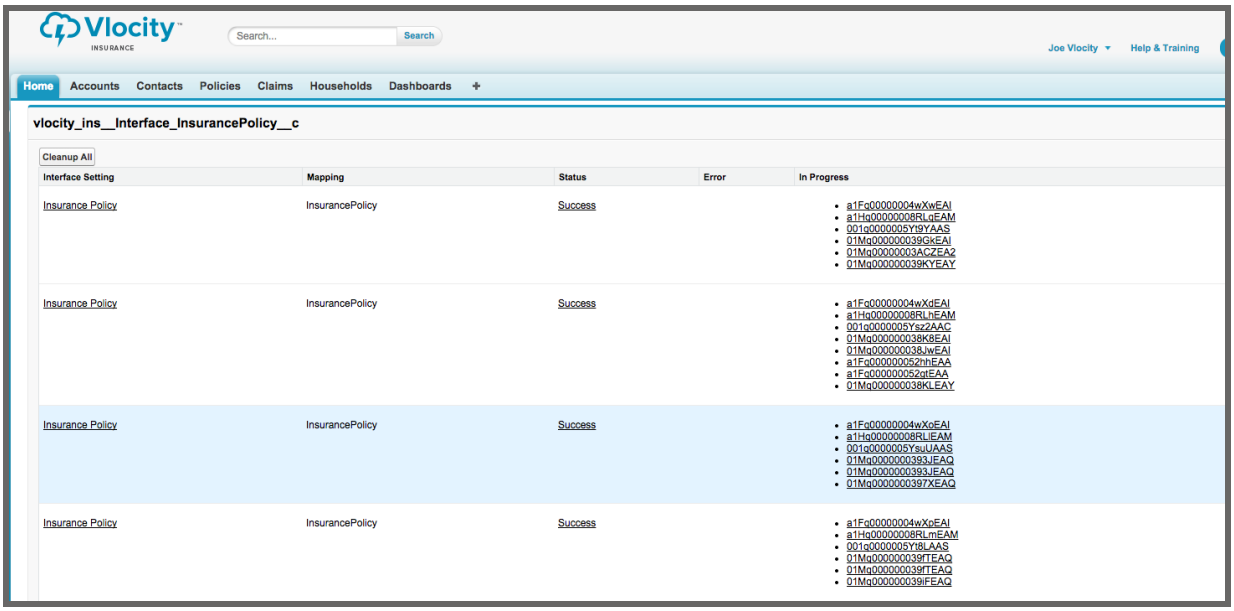
```
{
  "Contact": {
     "Contact Name" : "Dennis Reynolds",
     "Case Information": [
       {
           "Title": "This is the subject of Case #1"       },
       {
          "Title": "This is the subject of Case #2"       },
       {
          "Title": "This is the subject of Case #3"       }
    ]
 }
}
```

Monitor the Status of DataRaptor Jobs

You can monitor the progress of DataRaptor jobs directly from the Vlocity DataRaptor tab. The Progress column shows the number of records in progress, the Success column shows the number of records processed successfully, and the Errors column shows the number of records that did not process due to an error.

View Successful DataRaptor Jobs

After running a batch job, you can view verify completion by clicking **Show Success** under Actions.



View and Reprocess DataRaptor Errors

If errors occur while running a batch job, you can view them and reprocess after resolving.

To view errors:

Go to the DataRaptor tab and click **Show Errors**.

Common errors when running DataRaptor include:

- duplicate key found—Occurs when two valid entries match an upsert. The mapping must be updated to be more selective or delete duplicate record(s).
- required field missing—Occurs when either the backend system is missing required data or a field on the interface object is missing or not populated correctly.

After resolving errors, click **Reprocess Errors** under Actions to complete the batch job.

Errors for Object Type

If errors occur while running a batch job, you can view them and reprocess after resolving. From the Query Editor in the Developer Console or from https://-workbench.developerforce.com/login.php, run the following query:

- Select DRProgressData__c, DRError__c from *InterfaceObject* where DRStatus__c = 'Error'
- Replace *InterfaceObject* with the name of the Interface Object you are querying.

Errors for REST (Input and Output) Types

Any processing errors will return in the response to the API call.

Common errors when running DataRaptor include:

- duplicate key found—Occurs when two valid entries match an upsert. The mapping must be updated to be more selective or delete duplicate records.

- required field missing—Occurs when either the backend system is missing required data or a field on the interface object is missing or not populated correctly.
- duplicate found exception—Can occur when a lookup mapping returns more than one field from the Lookup Domain Object.

How To Run a DataRaptor in Batch Mode

DataRaptor can be set to run in Batch Mode. To do this you must set two properties.

- DataRaptor Bundle Properties

  Set your DataRaptor Process Now Threshold property to -1.

- DataRaptor Workers Custom Setting Property (One Time Setup)

  Create a DataRaptor Workers Custom setting with the following parameters.

To edit the DataRaptor custom settings:

1. From Setup, click **Develop** and then click **Custom Settings.**
2. Next to DataRaptor Workers, click **Manage**.
3. Click **New**.
4. Enter **A** in the `Name` field.
5. Click **Save**.

Note: You only need to create this setting once.

Test JSON with Workbench REST Explorer

DataRaptor bundles can be tested with JSON blobs with the Salesforce Workbench REST Explorer feature. This article explains the some of the parameters around a properly formatted JSON blob that can be sent via a POST action.

Parameters

Parameter Name	Description	Optional	Example
bundleName	The DataRaptor bundle name	Required	Create Account
objectList	JSON Object Data to be processed	Required	See below.
filesMap	Map of keys to base64 encoded files	Optional	See below.
bulkUpload	Should this use Batch Apex. Default: false	Optional	True/False

Resource Information

Response Format	JSON
Request URL	/services/apexrest/v2/DataRaptor/

Sample Request

```
POST
/services/apexrest/v2/DataRaptor/
POST Data
{
   "bundleName" : "AccountUpload",
   "objectList" : {
    "Agency Information": {
   "Agency Name": "Vlocity",
   "Agency Address": "50 Fremont",
   "Agency City": "San Francisco",
   "Agency State": "CA",
   "Agency Zip": "94110",
   }
   },
   "filesMap": {
   "8ed0c4a1-1f41-47b9-ab97-b7f531babc2b":
"data:text/javascript;base64,${BASE64STRING}"   },
   "bulkUpload" : false }
```

Sample Result

```
{
   "createdObjectsByOrder": {
   "Open Account": {
   "1": [
   "a10o00000022xVEAAY"   ]
   }
```

```
    },
    "createdObjectsByType": {
    "Open Account": {
    "Account": [
    "a10o00000022xVEAAY"    ]
    }
    },
    "errors": {},
    "returnResultsData": []
}
```

Using Workbench REST Explorer to Test DataRaptor Bundles

1. Log in to Workbench.
2. Navigate to the REST Explorer by going to: https://workbench.developerforce.com/restExplorer.php.
3. Select `POST`.
4. Enter **`/services/apexrest/{Enter_Package_Namespace_Here}/v2/DataRaptor/`** as the REST API service URL.
5. Enter your JSON blob in the Request Body field.

Your JSON blob should follow this format:

```
{
"bundleName" : "DataRaptor_Bundle_Name_Goes_Here",
```

```
"objectList" : {
"InterFace_Field_Name_Goes_Here": "Test_Value_Goes_Here" }
}
```

6. Click **Execute** and you will receive a Raw Response from Salesforce.

Note: Be sure to match the DataRaptor Bundle Name and Interface Field Names exactly, including capitalization.

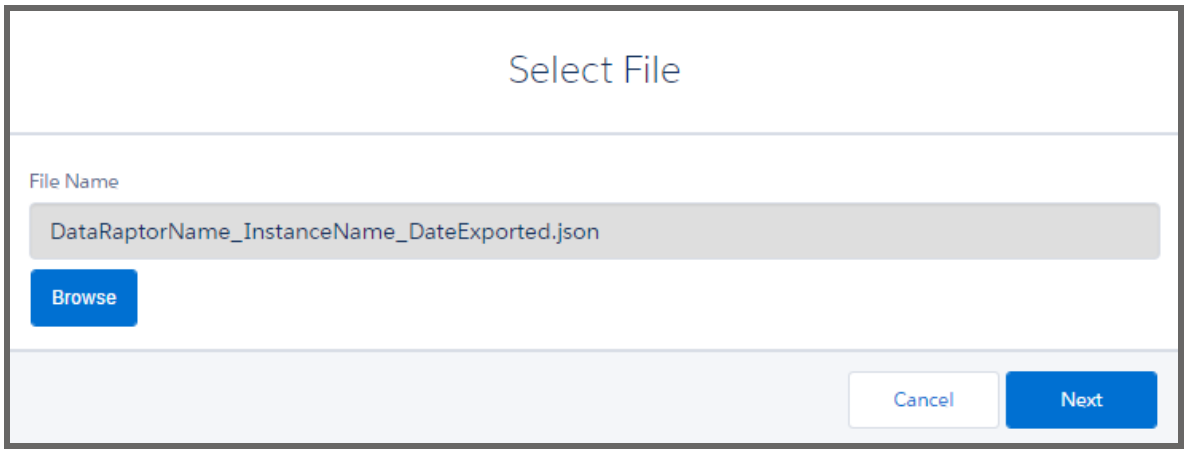
Chapter 1 Import and Export DataRaptor Interfaces

This guide covers importing and exporting DataRaptors.

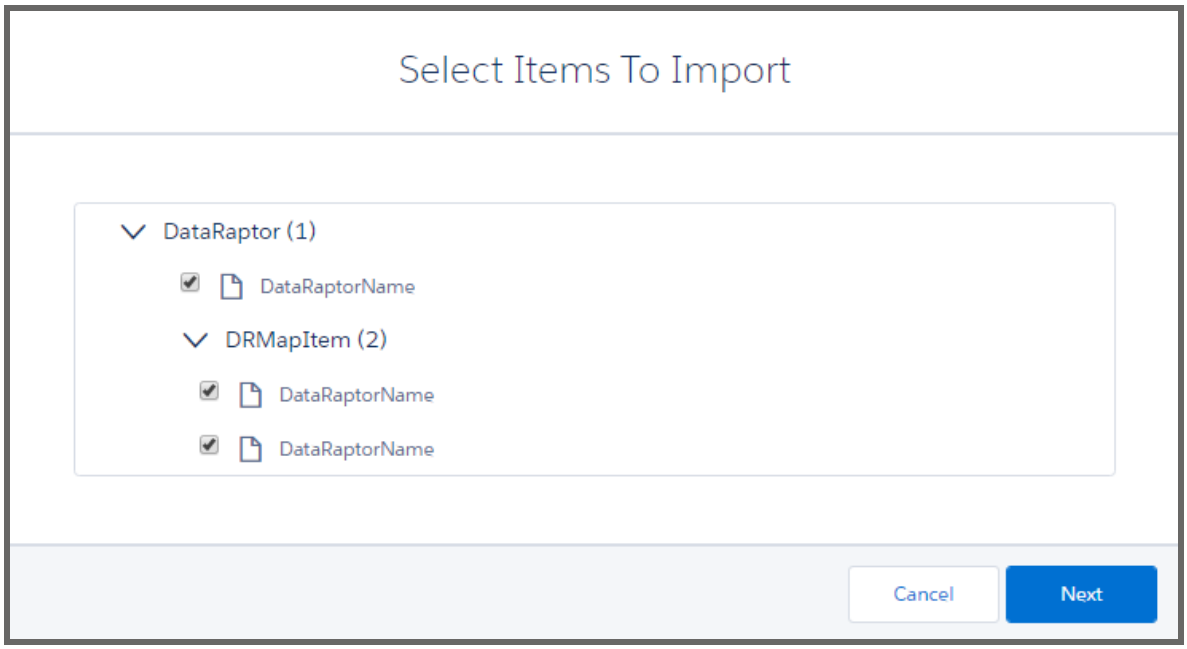
Import a DataRaptor Bundle

In Spring 2016, DataRaptor Interfaces can be imported using a the new Data Pack feature.

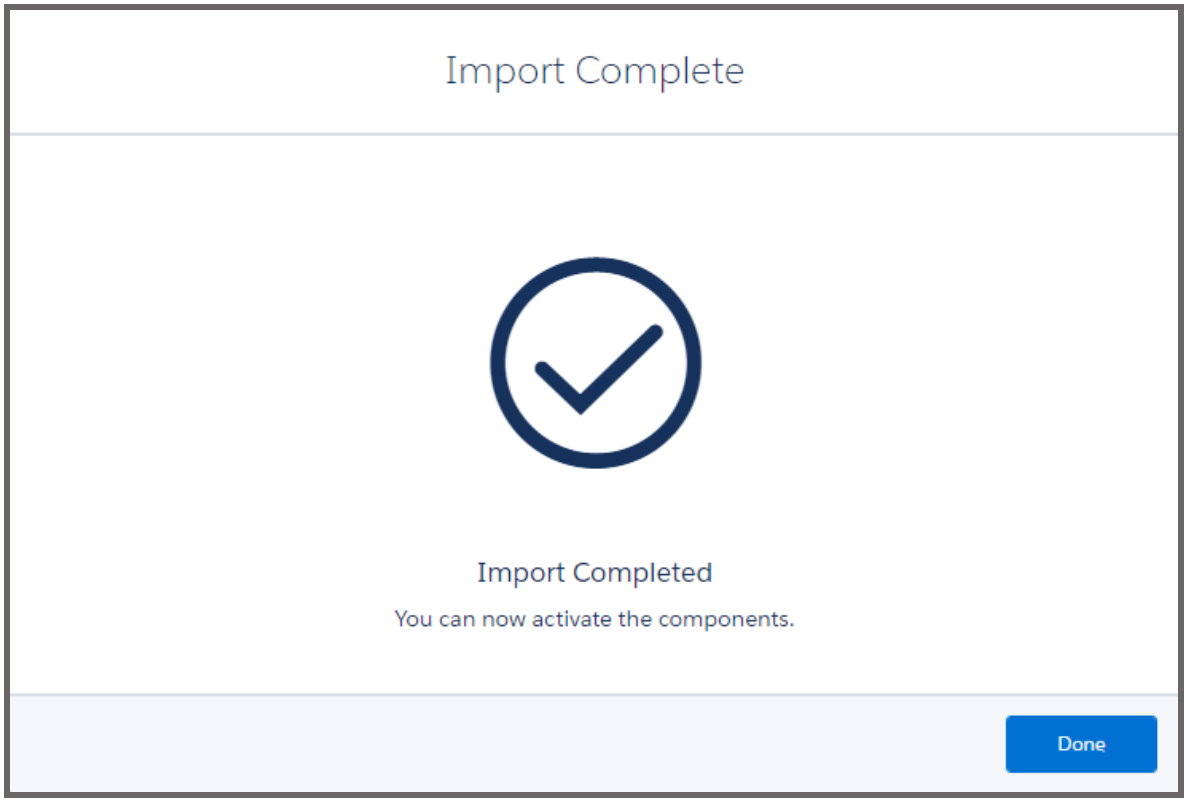
1. Go to the DataRaptor tab.
2. Click **Import**.
3. Click **Browse** and select the .JSON file that was previously exported.
4. Click **Next**.



5. Review the selected elements to import and click **Next**.



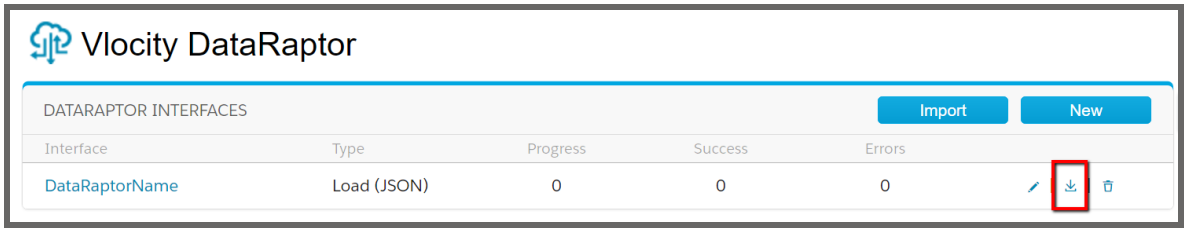
6. Wait for the import process to complete.



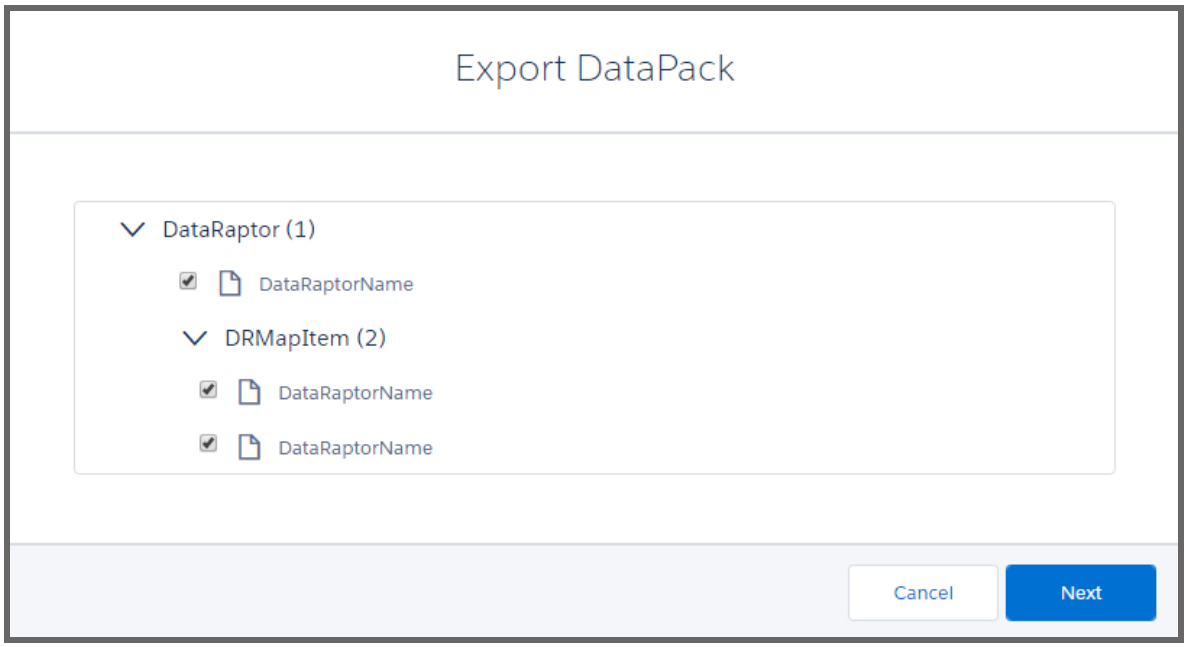
Export DataRaptors

In Spring 2016, DataRaptor Interfaces can be exported using the new Data Pack feature.

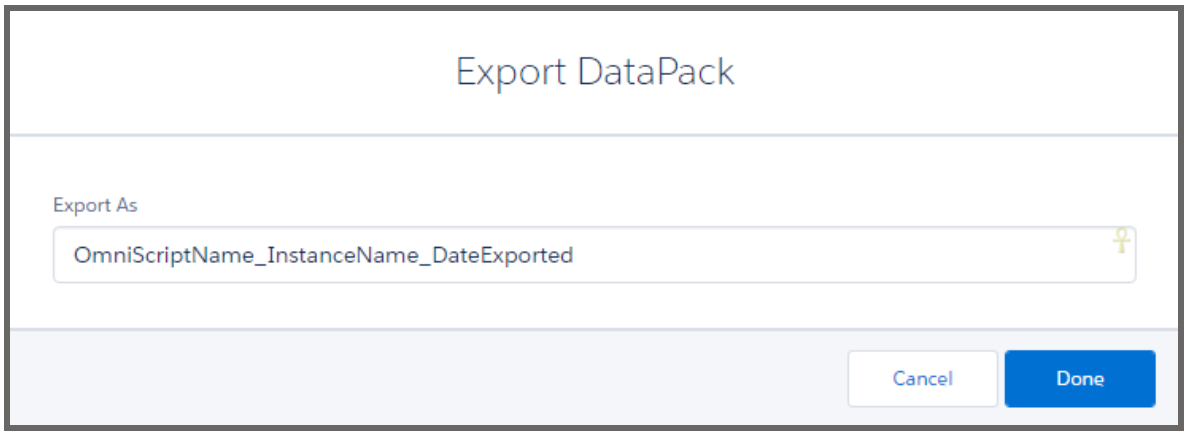
1. Go to the DataRaptor tab.
2. Identify the DataRaptor Interface to export.
3. Click the **Export** button.



4. Review the Export Elements and select all of the elements o export.
5. Click **Next**.



6. Give the export a file name and click **Done**.



7. The OmniScript will save as a .JSON file to your browser's default downloads folder.

To learn how to Import .JSON export DataRaptor interface files into another environment see **Import a DataRaptor Bundle** on page 69.

Chapter 1 Import Data Using Load JSON

To import data using the DataRaptor Load JSON process:

1. Prepare your source data to make sure it is compatible with Salesforce standards and can be written across Salesforce objects.
2. Create a sample Input JSON containing all fields to map into Vlocity. For more information, see "Create a Sample Input JSON" below.
3. Create a new DataRaptor Load (JSON) Interface in DataRaptor.
4. Set the Creation Sequence.
5. Map data from the Load (JSON) Interface to Salesforce Objects. For more information, see "Map Data From JSON to Salesforce Objects" on the next page.
6. Configure Data Processing settings for the Interface.
7. Implement a trigger on the DataRaptor Interface.
8. Import data from a source database or file into the Interface Object.
9. Check and resolve errors.
10. Prepare your source data.

Your source data must be compatible with Salesforce standards. You will also need to know how you're planning to map your source data across Salesforce objects.

To manipulate data before importing, you can write a preprocessor Apex class that implements the IRaptorPreprocessor Interface. This Apex Interface runs Before Insert on each record uploaded to the Interface Object. For instance, a name field formatted "Last Name, First Name" in a CSV file can be split into the Salesforce Contact standard firstName and lastName fields, using this custom Apex class. No preprocessor class is necessary if the data is already in the correct format before import.

Create a Sample Input JSON

DataRaptor uses a sample of the Input JSON to generate the rows in the Data Mappings table. Create a sample JSON that includes the fields from your back office system.

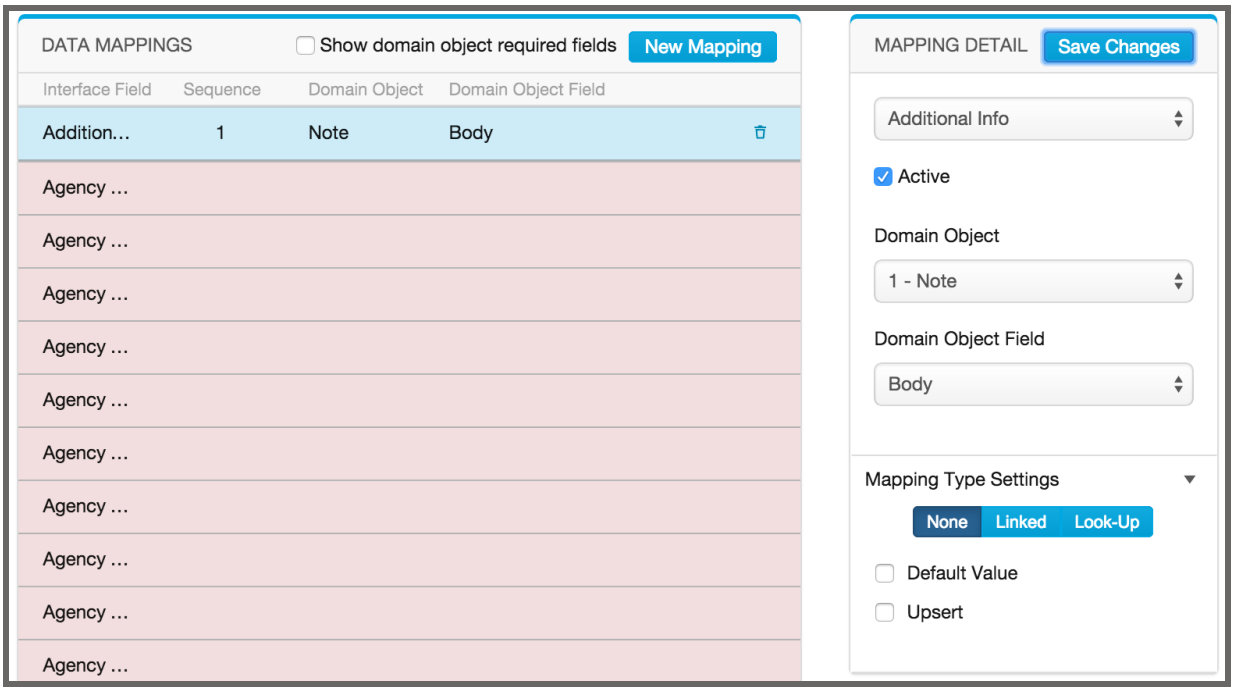
Input JSON

```
{
"Agency Information":
{
"Instruction Text":null,
"Name of Agency":"ABC Insurance",
"Date of Incorporation":null,
"Agency Tax ID":null,
"Renewal Date":null,
"Business Address Text":null,
"Business Address":null,
"Bus City":null,
"Bus State":null,
"Bus Zip":null,
"Bus Phys Address Text":null,
"Bus Phys Address":null,
"Bus Phys City":null,
"Bus Phys State":null,
"Bus Phys Zip":null,
    "Telephone":null,
"Web Page":null,
   }
}
```

Note: The field values will not affect the actual mapping at this point. The Sample Input JSON is only used to define the field and object mappings of the REST Interface.

Map Data From JSON to Salesforce Objects

After setting the Creation Sequence, you can create the mappings in the Data Mappings table. Unmapped rows are red. All the fields you describe in the sample JSON you pasted into the JSON field on the Interface dialogue appear on the left side of the table. Click on a mapping to select it (the blue row) and to open a mapping dialogue on the right side of the page.



In this mapping, the Additional Info field from the sample JSON is mapped to the Body field of the Note object.

Refer to "DataRaptor Input Mappings" on page 25 for more information on types of mappings. Configuration options and the data import process are the same for REST integrations as for Object integrations. Please refer to "Import a DataRaptor Bundle" on page 69 for more detail on these.

Chapter 1 Create or Update Records From an OmniScript Using DataRaptor

You can configure an OmniScript to map data collected from the script into custom Vlocity objects or standard Salesforce objects using DataRaptor. For example, you could create or update contact, account, or case records.

Use the **Data Raptor Post Action** element to either add a button to the script or run automatically in between steps. After you Create the OmniScript, create a DataRaptor Input Interface with the desired mappings. Return to the OmniScript and add the Interface name to the Data Raptor Post Action element. When the Action runs, it will:

- Call DataRaptor based on the provided Interface and pass in the Data JSON of the OmniScript.
- Create/update records based on the DataRaptor Interface.
- Optionally redirect to a page that shows the objects created by DataRaptor or any errors during upsert.
- Return any validation errors if the post fails. OmniScript returns the user to the field on the step that failed validation and displays the custom or standard error below that field.

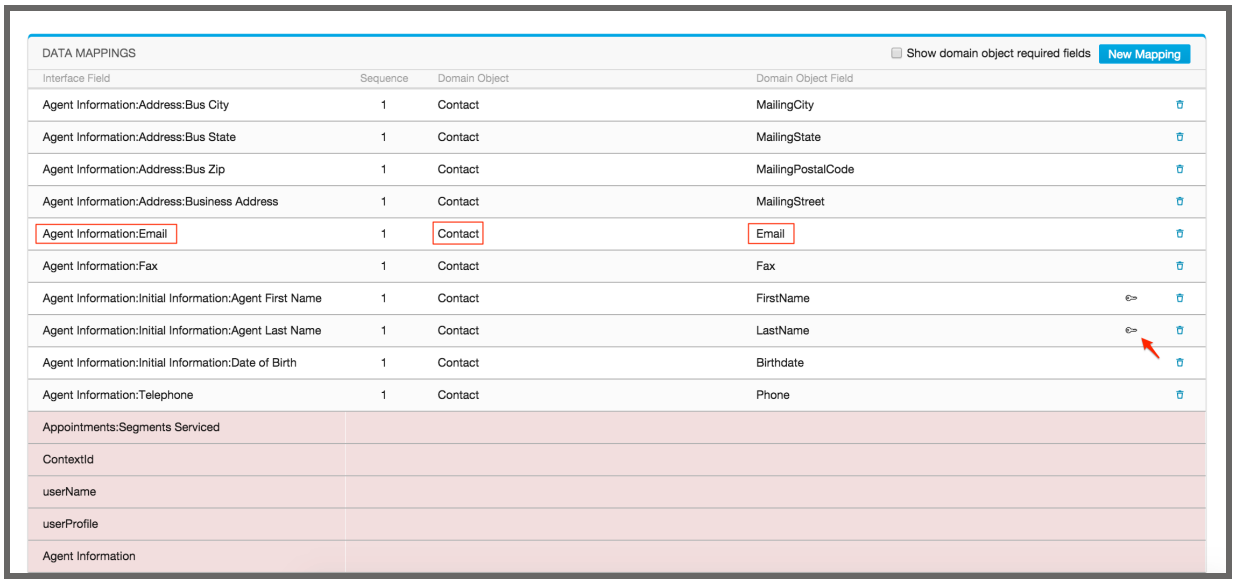
To create or update records from an OmniScript using DataRaptor:

1. Complete your OmniScript.
2. Add the **DataRaptor Post Action** to the OmniScript.
3. Click on the DataRaptor Interface dropdown and select **+Create New DataRaptor**. This will open DataRaptor in a new tab.
4. In the `Interface Name` field, enter a name for the interface.
5. In the `Interface Type` field, select `Load (JSON)`.
6. Add a description for the interface.
7. In the DataRaptor options, select the `Ignore Errors` check box. Leave all other settings unchanged. To configure the mappings, refer to the DataRaptor Input Mappings guide.

8. Test by stepping through the OmniScript and review the OmniScript Debug Console for successes or errors.
9. If records were successfully upserted, check them for accuracy. If the console shows errors, fix them and repeat the test.

In the following Interface, only pertinent Contact information is mapped to the Contact record.

When multiple Upsert Keys are used, the record must match all Upsert Keys to be considered a match. In the example pictured below, the Email element maps to Contact.Email. The mapping upserts on the first and last name of the contact. This means that if there is a match in Salesforce on the Agent first name AND last name, the record will update with the other mappings. If there is not a match for both first AND last name, DataRaptor will create a new record.



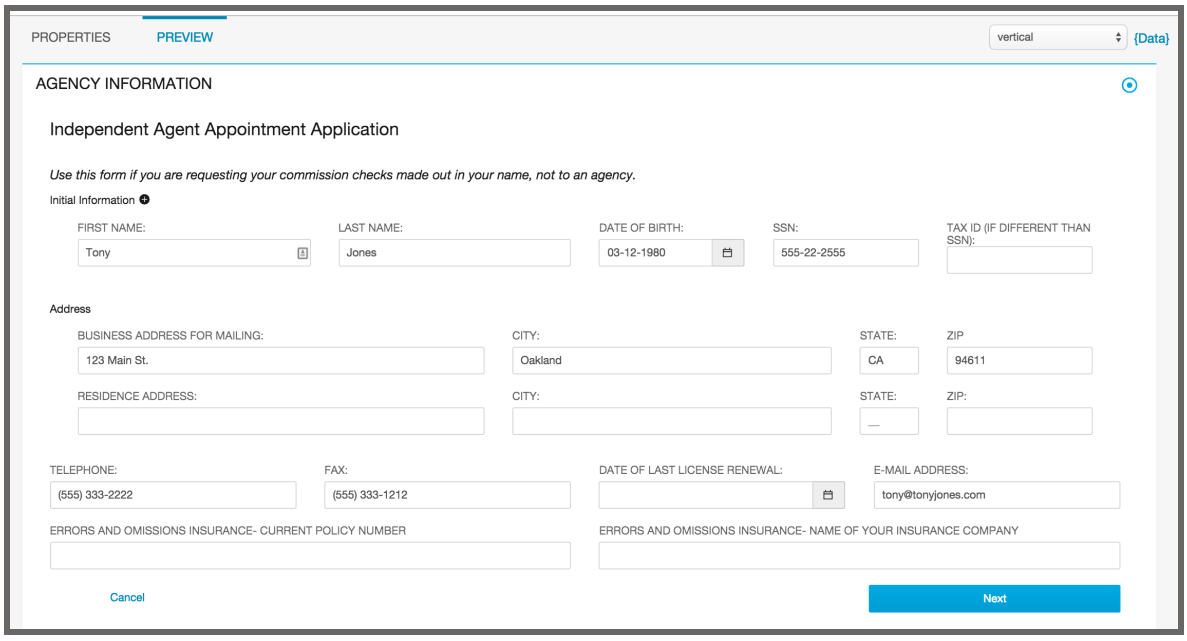
This section includes the following topics:

Test the OmniScript and DataRaptor Mappings

Now you can test the OmniScript and DataRaptor mappings.

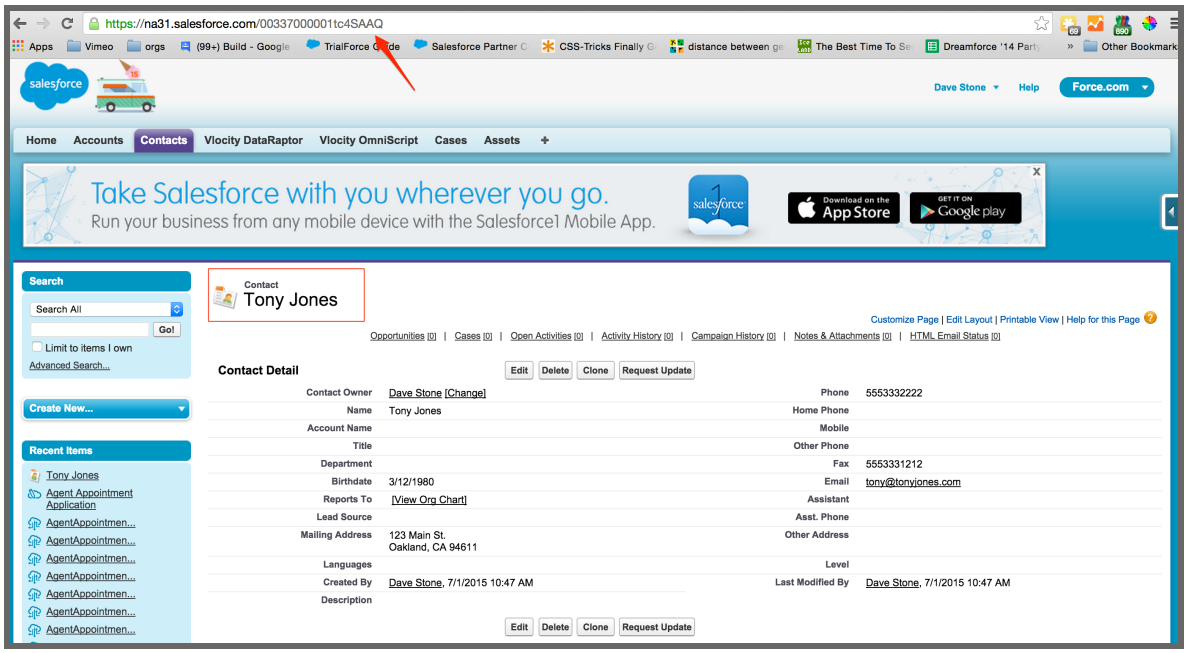
1. Either activate the OmniScript to test the form from outside the Designer, or click **Preview** to test in the Designer.

2. Fill out the form, making sure there is data in all the fields that should be mapped to Salesforce.



3. After submitting the form (or after the Data Raptor Post Action runs remotely), the redirect page opens. It displays an expandable JSON tree. Expand the `createdObjectsByType` node to see the Ids of any created or updated records. The AgentAppointmentAppPost creates one record.

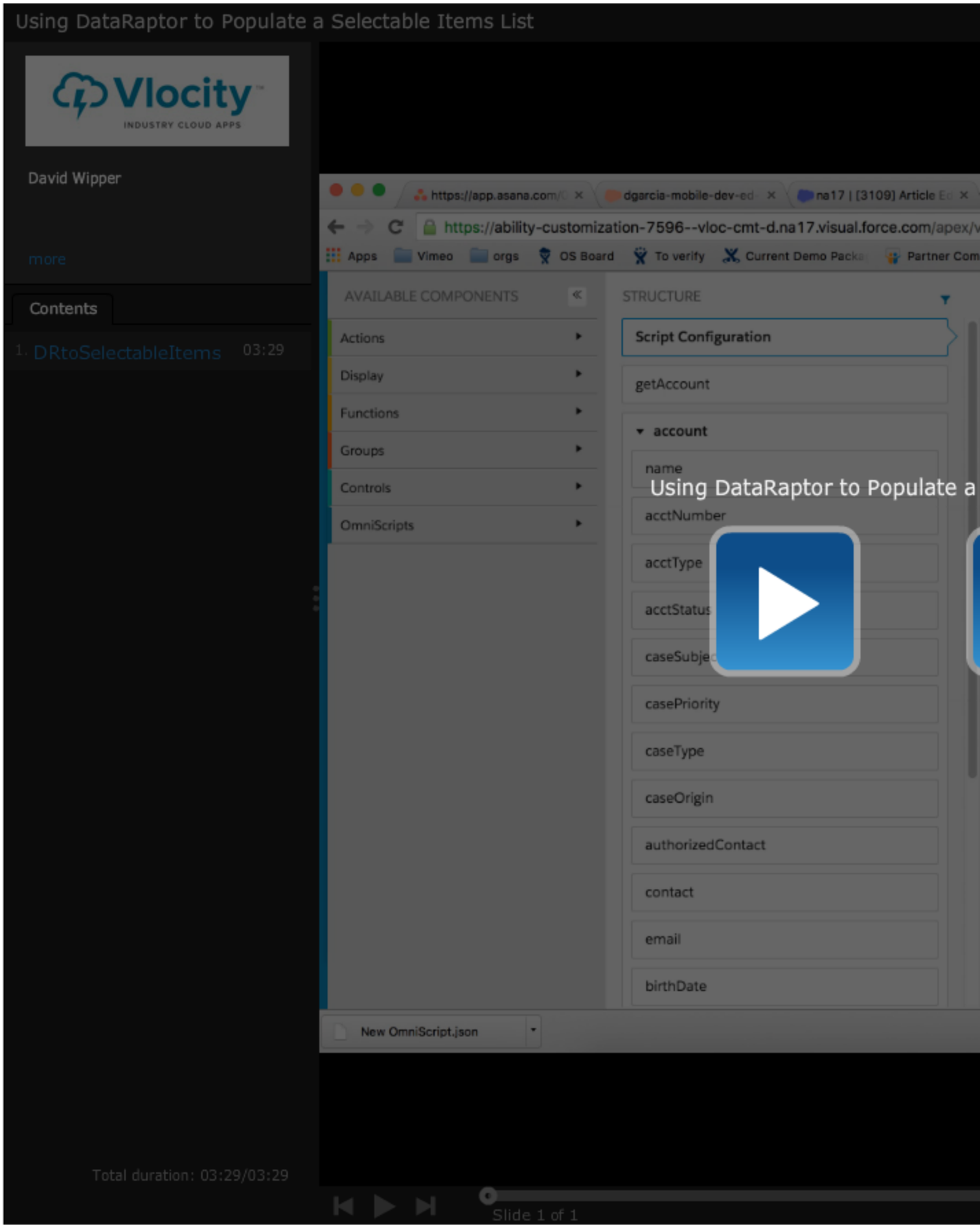
4. Copy the Id and paste it in your browser URL bar as shown below.



5. Inspect the fields on the record to confirm that the data mapped correctly.

Create and Populate a Custom Template for a Selectable Item Using DataRaptor

The video below explains how to use DataRaptor to populate a selectable items template.


Using DataRaptor to Populate a Selectable Items List
Vlocity
INDUSTRY CLOUD APPS
David Wipper
more
Contents
1. DRtoSelectableItems 03:29
Total duration: 03:29/03:29
https://ability-customization-7596--vloc-cmt-d.na17.visual.force.com/apex/v
Apps
Vimeo
orgs
OS Board
To verify
AVAILABLE COMPONENTS
Actions
Display
Functions
Groups
Controls
OmniScripts
STRUCTURE
Script Configuration
getAccount
account
name
acctNumber
acctType
acctStatus
caseSubje
casePriority
caseType
caseOrigin
authorizedContact
contact
email
birthDate
Using DataRaptor to Populate a
New OmniScript.json
Slide 1 of 1