Salesforce

Salesforce component

Available as of Camel 2.12

This component supports producer and consumer endpoints to communicate with Salesforce using Java DTOs.
There is a companion maven plugin Camel Salesforce Plugin that generates these DTOs (see further below).

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-salesforce</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

URI format

The URI scheme for a salesforce component is as follows

```
salesforce:topic?options
```

You can append query options to the URI in the following format, `?option=value&option=value&...`

Supported Salesforce APIs

The component supports the following Salesforce APIs

Producer endpoints can use the following APIs. Most of the APIs process one record at a time, the Query API can retrieve multiple Records.

Rest API

- getVersions - Gets supported Salesforce REST API versions
- getResources - Gets available Salesforce REST Resource endpoints
- getGlobalObjects - Gets metadata for all available SObject types
- getBasicInfo - Gets basic metadata for a specific SObject type
- getDescription - Gets comprehensive metadata for a specific SObject type
- getSObject - Gets an SObject using its Salesforce Id
- createSObject - Creates an SObject
- updateSObject - Updates an SObject using Id
- deleteSObject - Deletes an SObject using Id
- getSObjectWithId - Gets an SObject using an external (user defined) id field
- upsertSObject - Updates or inserts an SObject using an external id
- deleteSObjectWithId - Deletes an SObject using an external id
- query - Runs a Salesforce SOQL query
- queryMore - Retrieves more results (in case of large number of results) using result link returned from the 'query' API
- search - Runs a Salesforce SOSL query
- limits - fetching organization API usage limits
- recent - fetching recently viewed items
- approval - submit a record or records (batch) for approval process
- approvals - fetch a list of all approval processes
- composite-tree - create up to 200 records with parent-child relationships (up to 5 levels) in one go
- composite-batch - submit a composition of requests in batch

For example, the following producer endpoint uses the upsertSObject API, with the sObjectIdName parameter specifying 'Name' as the external id field.
The request message body should be an SObject DTO generated using the maven plugin.
The response message will either be `null` if an existing record was updated, or `CreateSObjectResult` with an id of the new record, or a list of errors while creating the new object.

```
        ...to("salesforce:upsertSObject?sObjectIdName=Name")...
```

Rest Bulk API

Producer endpoints can use the following APIs. All Job data formats, i.e. xml, csv, zip/xml, and zip/csv are supported.
The request and response have to be marshalled/unmarshalled by the route. Usually the request will be some stream source like a CSV file,
and the response may also be saved to a file to be correlated with the request.

- createJob - Creates a Salesforce Bulk Job
- getJob - Gets a Job using its Salesforce Id
- closeJob - Closes a Job
- abortJob - Aborts a Job
- createBatch - Submits a Batch within a Bulk Job
- getBatch - Gets a Batch using Id
- getAllBatches - Gets all Batches for a Bulk Job Id
- getRequest - Gets Request data (XML/CSV) for a Batch
- getResults - Gets the results of the Batch when its complete
- createBatchQuery - Creates a Batch from an SOQL query
- getQueryResultIds - Gets a list of Result Ids for a Batch Query
- getQueryResult - Gets results for a Result Id

For example, the following producer endpoint uses the createBatch API to create a Job Batch.
The in message must contain a body that can be converted into an `InputStream` (usually UTF-8 CSV or XML content from a file, etc.) and header fields 'jobId' for the Job and 'contentType' for the Job content type, which can be XML, CSV, ZIP_XML or ZIP_CSV. The put message body will contain `BatchInfo` on success, or throw a `SalesforceException` on error.

```
        ...to("salesforce:createBatchJob")..
```

Rest Streaming API

Consumer endpoints can use the following sytax for streaming endpoints to receive Salesforce notifications on create/update.

To create and subscribe to a topic

```
        from("salesforce:CamelTestTopic?
notifyForFields=ALL&notifyForOperations=ALL&sObjectName=Merchandise__c&updateTopic=true&sObjectQuery=SELECT Id,
Name FROM Merchandise__c")...
```

To subscribe to an existing topic

```
        from("salesforce:CamelTestTopic&sObjectName=Merchandise__c")...
```

Examples

Uploading a document to a ContentWorkspace

Create the ContentVersion in Java, using a Processor instance:

```
public class ContentProcessor implements Processor {
        public void process(Exchange exchange) throws Exception {
            Message message = exchange.getIn();

                ContentVersion cv = new ContentVersion();
                ContentWorkspace cw = getWorkspace(exchange);
                cv.setFirstPublishLocationId(cw.getId());
                cv.setTitle("test document");
                cv.setPathOnClient("test_doc.html");
                byte[] document = message.getBody(byte[].class);
                ObjectMapper mapper = new ObjectMapper();
                String enc = mapper.convertValue(document, String.class);
                cv.setVersionDataUrl(enc);
                message.setBody(cv);
    }

        protected ContentWorkspace getWorkSpace(Exchange exchange) {
                // Look up the content workspace somehow, maybe use enrich() to add it to a
                // header that can be extracted here
                ....
        }
}
```

Give the output from the processor to the Salesforce component:

```
        from("file:///home/camel/library")
                .to(new ContentProcessor())     // convert bytes from the file into a ContentVersion SObject
                                                                  // for the salesforce component
                .to("salesforce:createSObject");
```

Using Salesforce Limits API

With **salesforce:limits** operation you can fetch of API limits from Salesforce and then act upon that data received. The result of **salesforce:limits** operation is mapped to *org.apache.camel.component.salesforce.api.dto.Limits* class and can be used in a custom processors or expressions.

For instance, consider that you need to limit the API usage of Salesforce so that 10% of daily API requests is left for other routes. The body of output message contains an instance of *org.apache.camel.component.salesforce.api.dto.Limits* object that can be used in conjunction with Content Based Router and Spring Expression Language (SpEL) to choose when to perform queries. Notice how multiplying **1.0** with the integer value held in **body. dailyApiRequests.remaining** makes the expression evaluate as with floating point arithmetic, without it - it would end up making integral division which would result with either **0** (some API limits consumed) or **1** (no API limits consumed).

```
from("direct:querySalesforce")
    .to("salesforce:limits")
    .choice()
    .when(spel("#{1.0 * body.dailyApiRequests.remaining / body.dailyApiRequests.max < 0.1}"))
        .to("salesforce:query?...")
    .otherwise()
        .setBody(constant("Used up Salesforce API limits, leaving 10% for critical routes"))
    .endChoice()
```

Working with approvals

All the properties are named exactly the same as in the Salesforce REST API prefixed with **approval**. You can set approval properties by setting **approval PropertyName** of the Endpoint these will be used as template -- meaning that any property not present in either body or header will be taken from the Endpoint configuration. Or you can set the approval template on the Endpoint by assigning **approval** property to a reference onto a bean in the Registry

You can also provide header values using the same **approvalPropertyName** in the incoming message headers.

And finally body can contain one **AprovalRequest** or an **java.util.Iterable** of **ApprovalRequest** objects to process as a batch.

The important thing to remember is the priority of the values specified in these three mechanisms:

1. value in body takes precedence before any other
2. value in message header takes precedence before template value
3. value in template is set if no other value in header or body was given

For example to send one record for approval using values in headers use:

Given a route:

```
from("direct:example1")
        .setHeader("approval.ContextId", simple("${body['contextId']}"))
        .setHeader("approval.NextApproverIds", simple("${body['nextApproverIds']}"))
        .to("salesforce:approval?"
            + "approval.actionType=Submit"
            + "&approval.comments=this is a test"
            + "&approval.processDefinitionNameOrId=Test_Account_Process"
            + "&approval.skipEntryCriteria=true");
```

You could send a record for approval using:

```
final Map<String, String> body = new HashMap<>();
body.put("contextId", accountIds.iterator().next());
body.put("nextApproverIds", userId);

final ApprovalResult result = template.requestBody("direct:example1", body, ApprovalResult.class);
```

Using Salesforce Recent Items API

To fetch the recent items use **salesforce:recent** operation. This operation returns an **java.util.List** of **org.apache.camel.component.salesforce.api.dto.RecentItem** objects (**List<RecentItem>**) that in turn contain the **Id**, **Name** and **Attributes** (with **type** and **url** properties). You can limit the number of returned items by specifying **limit** parameter set to maximum number of records to return.

For example:

```
from("direct:fetchRecentItems")
    to("salesforce:recent")
        .split().body()
            .log("${body.name} at ${body.attributes.url}");
```

Using Salesforce Composite API to submit SObject tree

To create up to 200 records including parent-child relationships use `salesforce:composite-tree` operation. This requires an instance of `org.apache.camel.component.salesforce.api.dto.composite.SObjectTree` in the input message and returns the same tree of objects in the output message. The `org.apache.camel.component.salesforce.api.dto.AbstractSObjectBase` instances within the tree get updated with the identifier values (`Id` property) or their corresponding `org.apache.camel.component.salesforce.api.dto.composite.SObjectNode` is populated with `errors` on failure.

Note that for some records operation can succeed and for some it can fail—so you need to manually check for errors.
Easiest way to use this functionality is to use the DTOs generated by the `camel-salesforce-maven-plugin`, but you also have the option of customizing the references that identify the each object in the tree, for instance primary keys from your database.
Lets look at an example:

```
Account account = ...
Contact president = ...
Contact marketing = ...

Account anotherAccount = ...
Contact sales = ...
Asset someAsset = ...

// build the tree
SObjectTree request = new SObjectTree();
request.addObject(account).addChildren(president, marketing);
request.addObject(anotherAccount).addChild(sales).addChild(someAsset);

final SObjectTree response = template.requestBody("salesforce:composite-tree", tree, SObjectTree.class);
final Map<Boolean, List<SObjectNode>> result = response.allNodes()
                                                    .collect(Collectors.groupingBy(SObjectNode::hasErrors));

final List<SObjectNode> withErrors = result.get(true);
final List<SObjectNode> succeeded = result.get(false);

final String firstId = succeeded.get(0).getId();
```

Using Salesforce Composite API to submit multiple requests in a batch

The Composite API batch operation (`composite-batch`) allows you to accumulate multiple requests in a batch and then submit them in one go, saving the round trip cost of multiple individual requests. Each response is then received in a list of responses with the order perserved, so that the n-th requests response is in the n-th place of the response.

> *The results can vary from API to API so the result of the request is given as a `java.lang.Object`. In most cases the result will be a `java.util.Map` with string keys and values or other `java.util.Map` as value. Requests made in JSON format hold some type information (i.e. it is known what values are strings and what values are numbers), so in general those will be more type friendly.*
> *Note that the responses will vary between XML and JSON, this is due to the responses from Salesforce API being different. So be careful if you switch between formats without changing the response handling code.*

Lets look at an example:

```
final String acountId = ...
final SObjectBatch batch = new SObjectBatch("38.0");

final Account updates = new Account();
updates.setName("NewName");
batch.addUpdate("Account", accountId, updates);

final Account newAccount = new Account();
newAccount.setName("Account created from Composite batch API");
batch.addCreate(newAccount);

batch.addGet("Account", accountId, "Name", "BillingPostalCode");

batch.addDelete("Account", accountId);

final SObjectBatchResponse response = template.requestBody("salesforce:composite-batch?format=JSON", batch,
SObjectBatchResponse.class);

boolean hasErrors = response.hasErrors(); // if any of the requests has resulted in either 4xx or 5xx HTTP
status
final List<SObjectBatchResult> results = response.getResults(); // results of three operations sent in batch

final SObjectBatchResult updateResult = results.get(0); // update result
final int updateStatus = updateResult.getStatusCode(); // probably 204
final Object updateResultData = updateResult.getResult(); // probably null

final SObjectBatchResult createResult = results.get(1); // create result
@SuppressWarnings("unchecked")
final Map<String, Object> createData = (Map<String, Object>) createResult.getResult();
final String newAccountId = createData.get("id"); // id of the new account, this is for JSON, for XML it would
be createData.get("Result").get("id")

final SObjectBatchResult retrieveResult = results.get(2); // retrieve result
@SuppressWarnings("unchecked")
final Map<String, Object> retrieveData = (Map<String, Object>) retrieveResult.getResult();
final String accountName = retrieveData.get("Name"); // Name of the retrieved account, this is for JSON, for
XML it would be createData.get("Account").get("Name")
final String accountBillingPostalCode = retrieveData.get("BillingPostalCode"); // Name of the retrieved
account, this is for JSON, for XML it would be createData.get("Account").get("BillingPostalCode")

final SObjectBatchResult deleteResult = results.get(3); // delete result
final int updateStatus = deleteResult.getStatusCode(); // probably 204
final Object updateResultData = deleteResult.getResult(); // probably null
```

Camel Salesforce Maven Plugin

This Maven plugin generates DTOs for the Camel Salesforce.

Usage

The plugin configuration has the following properties.

Option	Description
clientId	Salesforce client Id for Remote API access
clientSecret	Salesforce client secret for Remote API access
userName	Salesforce account user name
password	Salesforce account password (including secret token)
version	Salesforce Rest API version, defaults to 25.0
outputDirectory	Directory where to place generated DTOs, defaults to ${project.build.directory}/generated-sources/camel-salesforce
includes	List of SObject types to include
excludes	List of SObject types to exclude

includePattern	Java RegEx for SObject types to include
excludePattern	Java RegEx for SObject types to exclude
packageName	Java package name for generated DTOs, defaults to org.apache.camel.salesforce.dto.

For obvious security reasons it is recommended that the clientId, clientSecret, userName and password fields be not set in the pom.xml. The plugin should be configured for the rest of the properties, and can be executed using the following command:

```
        mvn camel-salesforce:generate -DclientId=<clientid> -DclientSecret=<clientsecret> -DuserName=<username>
-Dpassword=<password>
```

The generated DTOs use Jackson and XStream annotations. All Salesforce field types are supported. Date and time fields are mapped to Joda DateTime, and picklist fields are mapped to generated Java Enumerations.

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started