Feature retired in Kentico 11

Salesforce integration in Kentico 11 uses Salesforce SOAP API version 23.0, which is deprecated and will be retired in 2023. For more information, see the Salesforce Platform API Versions 21.0 through 30.0 Retirement article.

If you wish to use Salesforce integration, you need to upgrade to Kentico Xperience 13.

In addition to the built-in replication process that automatically converts Kentico contacts into Salesforce leads, Kentico also provides integration with the Force.com API. The integration allows you to develop custom logic for manipulating data inside your Salesforce organization (query, create, update and delete).

This page introduces the Force.com integration API and presents basic examples to get you started.

The following content assumes you already have knowledge of the Force.com API and Salesforce Object Query Language (SOQL). SOQL is the object language developed for querying data on the Force.com platform. Refer to the official Force.com documentation for more information on the Force.com API and SOQL.

Kentico integrates the API using the Partner version of the web service, which is weakly typed. This means that the integration API does not use classes such as Lead or Company. Only general classes are available: *Object* and *Field*. For example, to create a new contact in your Salesforce organization, you need to create a new Object and set its properties so that it represents a contact. The flexibility of this form of integration reduces the reliance on the Force.com data model.

Note: There are several basic naming differences between the default Force.com API and the integration API in Kentico:

Salesforce	Kentico
Object	Entity
Object Type	Model
Object Field	Entity Attribute
Object Field Type	Entity Attribute Model

Salesforce API requirements and limitations

You can only use the integration API if your Salesforce organization has the API feature enabled. This feature is enabled by default for the following editions:

- Unlimited
- Enterprise
- Developer
- Some Professional Edition organizations may have the API enabled

Force.com also enforces a limit on the number of API calls your organization can make during a 24 hour window. If your organization exceeds the limit, all calls result in an error until the number of calls over the last 24 hours drops below the limit. The maximum number of allowed calls for your organization is determined by the edition you are using.

Reference requirement

The Salesforce integration API requires a reference to the **CMS.SalesForce** namespace.

Add the following *using* directive to all files that access the integration API:

```
using CMS.SalesForce;
```

Establishing a session with the Salesforce organization

Before you can start working with the data, you need to establish a session with your Salesforce organization.

1. Use the following code to create a session for your Salesforce organization based on your Salesforce integration settings.

```
// Provides a Salesforce organization session
ISessionProvider sessionProvider = new ConfigurationSessionProvider();
Session session = sessionProvider.CreateSession();
```

2. Create a client to access the organization's data using the session.

```
// Creates a client to access Salesforce organization's data
SalesForceClient client = new SalesForceClient(session);
```

You can then use the client to perform specific operations.

> All examples in the sections below use the **client** variable for this purpose.

Querying data

1. Describe the entity you will be working with.
   - The following examples work with Salesforce *Contact* entities, but you can use the same approach for other Salesforce objects such as Leads or Accounts.

```
// Describes the Salesforce Contact entity
EntityModel model = client.DescribeEntity("Contact");

// Displays basic information about the entity's attributes
foreach (EntityAttributeModel attributeModel in model.AttributeModels)
{
        Console.WriteLine("{0} is an attribute labeled {1} of type {2}",
attributeModel.Name, attributeModel.Label, attributeModel.Type);
}
```

2. Load the attributes of the entity using SOQL.

```
// Executes a query using SOQL
SelectEntitiesResult result = client.SelectEntities("SELECT Id, Name, MobilePhone
FROM Contact", model);

// Displays how many contacts were found
Console.WriteLine("{0} contacts were found", result.TotalEntityCount);

// Displays basic information about each of the contacts
foreach (Entity contact in result.Entities)
{
    Console.WriteLine("Contact {0} with name {1} and phone number {2}", contact.
Id, contact["Name"], contact["MobilePhone"]);
}
```

Creating objects

1. Create a new contact.

```
// Describes the Contact entity
EntityModel model = client.DescribeEntity("Contact");

// Creates a new contact entity
Entity customContact = model.CreateEntity();

customContact["LastName"] = "Jones";
customContact["Description"] = "Always has his wallet on him";

Entity[] customContacts = new Entity[] { customContact };
CreateEntityResult[] results = client.CreateEntities(customContacts);
```

2. Use the following code to check whether the creation was successful or not.

```
// Checks if the contact was successfully created in Salesforce
foreach (CreateEntityResult createResult in results)
{
    if (createResult.IsSuccess)
    {
        Console.WriteLine("A contact with id {0} was inserted.", createResult.
EntityId);
    }
    else
    {
        Console.WriteLine("A contact could not be inserted.");
        foreach (Error error in createResult.Errors)
        {
            Console.WriteLine("An error {0} has occurred: {1}", error.StatusCode,
error.Message);
        }
    }
}
```

Updating existing objects

To update an object, you first need to either prepare a new entity, or load an existing one using SOQL.

Custom Salesforce field required

The following examples use a custom external ID attribute for Salesforce contacts, so you first need to define the custom field in Salesforce.

1. Log in to Salesforce.
2. Navigate to **Setup -> Customize -> Contacts -> Fields**.
3. Click **New** in the **Contact Custom Fields & Relationships** section.
4. Choose **Text** as the data type and click **Next**.
5. In the **Enter the details** step, fill in the following values:
   - **Field label**: KenticoContactID
   - **Length**: 32
   - **Field Name**: KenticoContactID
   - **External ID**: enabled

1. Update a contact and assign an external ID value.

```
// Describes the Contact entity
EntityModel model = client.DescribeEntity("Contact");

// Updates an existing contact
Entity updateContact = model.CreateEntity();

updateContact["Description"] = "Always has his satchel on him";
updateContact["KenticoContactID"] = "D900172AABCE11E1BC6924C56188709B";

Entity[] updateContacts = new Entity[] { updateContact };
UpdateEntityResult[] updateResults = client.UpdateEntities(updateContacts);
```

2. Use the following code to check whether the update was successful or not.

```
// Checks if the contact was successfully updated
foreach (UpdateEntityResult updateResult in updateResults)
{
    if (updateResult.IsSuccess)
    {
        Console.WriteLine("The contact with id {0} was updated.", updateResult.
EntityId);
    }
    else
    {
        Console.WriteLine("The contact could not be updated.");
        foreach (Error error in updateResult.Errors)
        {
            Console.WriteLine("An error {0} has occurred: {1}", error.StatusCode,
error.Message);
        }
    }
}
```

Upserting objects

The *Upsert* operation uses an external ID to identify already existing objects and then decides whether to *update* the object or *create* a new one:

- If the external ID does not exist, a new record is *created*.
- When the external ID matches an existing record, the given record is *updated*.
- If multiple external ID matches are found, the upsert operation reports an error.

Note: It is generally recommended to use upsert instead of directly creating entities if you plan on specifying an External ID attribute. This allows you to avoid creating duplicate records.

1. Create a new entity using the *upsert* call.
    - If you leave the value of *KenticoContactID* the same as in the previous example, the existing contact will be *updated*.

```
// Describes the Contact entity
EntityModel model = client.DescribeEntity("Contact");

// Creates a new contact or updates an existing one
Entity upsertContact = model.CreateEntity();

upsertContact["Description"] = "Is a professor";
upsertContact["KenticoContactID"] = "D900172AABCE11E1BC6924C56188709B";

Entity[] upsertContacts = new Entity[] { upsertContact };
UpsertEntityResult[] upsertResults = client.UpsertEntities(upsertContacts,
"KenticoContactID");
```

2. Use the following code to check for the results of the upsert call.

```
// Checks the results of the upsert call
foreach (UpsertEntityResult upsertResult in upsertResults)
{
    if (upsertResult.IsSuccess)
    {
        if (upsertResult.IsUpdate)
        {
            Console.WriteLine("The contact with id {0} was updated.",
upsertResult.EntityId);
        }
        else
        {
            Console.WriteLine("A new contact with id {0} was inserted.",
upsertResult.EntityId);
        }
    }
    else
    {
        Console.WriteLine("The contact could not be created nor updated.");
        foreach (Error error in upsertResult.Errors)
        {
            Console.WriteLine("An error {0} has occurred: {1}", error.StatusCode,
error.Message);
        }
    }
}
```

Deleting objects

The delete call takes an array of entity IDs as a parameter. The ID is generated by Salesforce and is unique for each object.

1. Use an upsert call to create several new contacts that you will later delete.

```
// Describes the Contact entity
EntityModel model = client.DescribeEntity("Contact");

// Creates new contacts
Entity customContact1 = model.CreateEntity();
Entity customContact2 = model.CreateEntity();
Entity customContact3 = model.CreateEntity();

customContact1["LastName"] = "Drake";
customContact1["Description"] = "Is a hunter";
customContact1["KenticoContactID"] = "6C2L16B8W0ZXSU5SYPYRVE08QSKOR7F6";
customContact2["LastName"] = "Croft";
customContact2["Description"] = "Is an archeologist";
customContact2["KenticoContactID"] = "N5XX8Z42ISTVYGPC98AH81T1RHVOSESR";
customContact3["LastName"] = "Solo";
customContact3["Description"] = "Is a pilot";
customContact3["KenticoContactID"] = "N34VY7D5K677GKG8JOGDIA9UHVBHHVSL";

Entity[] newCustomContacts = new Entity[] { customContact1, customContact2,
customContact3 };
UpsertEntityResult[] createResults = client.UpsertEntities(newCustomContacts,
"KenticoContactID");
```

2. Use the following example to check whether the creation was successful or not.

```
// Checks the results of the upsert call
foreach (UpsertEntityResult upsertResult in createResults)
{
    if (upsertResult.IsSuccess)
    {
        if (upsertResult.IsUpdate)
        {
            Console.WriteLine("The contact with id {0} was updated.",
upsertResult.EntityId);
        }
        else
        {
            Console.WriteLine("A new contact with id {0} was inserted.",
upsertResult.EntityId);
        }
    }
    else
    {
        Console.WriteLine("The contact could not be created nor updated.");
        foreach (Error error in upsertResult.Errors)
        {
            Console.WriteLine("An error {0} has occurred: {1}", error.StatusCode,
error.Message);
        }
    }
}
```

3. Select the new contacts using a SOQL query and list their IDs.

   Note: Force.com returns an 18 character version of object IDs in API calls.

```
// Executes a query using SOQL
SelectEntitiesResult queryResult = client.SelectEntities("SELECT Id, Name FROM
Contact WHERE description LIKE 'Is a%'", model);

// Displays how many contacts have been found
Console.WriteLine("{0} contacts were found", queryResult.TotalEntityCount);

// Displays basic information about each of the contacts
foreach (Entity contact in queryResult.Entities)
{
    Console.WriteLine("Contact {0} with name {1}", contact.Id, contact["Name"]);
}
```

4. Delete all of the contacts returned by the query.
   - Identify the contacts using an array of ID values.
   - Make sure you fill in the IDs returned by the previous selection query.

```
// Deletes contacts
String[] contactIDs = new String[] { "a0BA0000000L2ZCMA0", "a0BA0000000L2ZCMA1",
"a0BA0000000L2ZCMA2" };
DeleteEntityResult[] deleteResults = client.DeleteEntities(contactIDs);
```

5. Check whether the deletion was successful.

```
// Checks if the contact was successfully deleted
foreach (DeleteEntityResult deleteResult in deleteResults)
{
    if (deleteResult.IsSuccess)
    {
        Console.WriteLine("A contact with id {0} was deleted.", deleteResult.
EntityId);
    }
    else
    {
        Console.WriteLine("A contact could not be deleted.");
        foreach (Error error in deleteResult.Errors)
        {
            Console.WriteLine("An error {0} has occurred: {1}", error.StatusCode,
error.Message);
        }
    }
}
```

Loading deleted objects

After you delete an object in Salesforce, its *IsDeleted* attribute is set to *True*. You can't select deleted objects unless you enable the IncludeDeleted option for the query.

1. Change the client options so that queries include deleted objects .

```
// Changes the client settings
client.Options.IncludeDeleted = true;
```

2. Select the deleted contacts using a SOQL query.

```
// Describes the Contact entity
EntityModel model = client.DescribeEntity("Contact");

// Executes a query using SOQL

SelectEntitiesResult includeDeletedQueryResult = client.SelectEntities("SELECT
Id, Name FROM Contact WHERE description LIKE 'Is a%'", model);

// Displays how many contacts were found
Console.WriteLine("{0} contacts were found", includeDeletedQueryResult.
TotalEntityCount);

// Displays the basic information of each contact
foreach (Entity contact in includeDeletedQueryResult.Entities)
{
    Console.WriteLine("Contact {0} with name {1}", contact.Id, contact["Name"]);
}

// Reverts the client settings to ignore deleted contacts again
client.Options.IncludeDeleted = false;
```

Call options

You can adjust Salesforce client calls through additional options. The options are similar to the SOAP headers available in the Force.com API.

Call option	Type	Description
TransactionEnabled	Bool	Specifies whether the command operations run in a transaction.
AttributeTruncationEnabled	Bool	Specifies whether truncation is allowed for string values.
FeedTrackingEnabled	Bool	Specifies whether the changes made by the call are tracked in feeds.
MruUpdateEnabled	Bool	Specifies whether the call updates the list of most recently used items in Salesforce.
IncludeDeleted	Bool	Specifies whether SOQL queries include deleted entries.
ClientName	String	String identifier for the client.
DefaultNamespace	String	String that identifies a developer namespace prefix.
CultureName	String	Specifies the language of the returned labels.
BatchSize	Int	Specifies the maximum number of entities returned by one query call.