Salesforce CPQ Plugins

Version 64.0, Summer '25

Last updated: June 20, 2025

CONTENTS

CHAPTER 1 Salesforce CPQ Plugins

In this chapter ...

Salesforce CPQ plugins let you add customized functionality to features within the Salesforce CPQ package.

EDITIONS

Available in: All Salesforce CPQ Editions

Javascript Quote Calculator Plugin

Add extra functionality to the quote line editor in Salesforce CPQ with custom JavaScript code. Seven available methods allow you to change how calculations are performed and manage page-level security such as field visibility.

JavaScript code is saved in Salesforce CPQ as custom scripts.

EDITIONS

Available in: Salesforce CPQ Winter '16 and later

Quote Calculator Plugin Guidelines

Consider these key guidelines when planning scripts for the Javascript Quote Calculator Plugin.

Quote Calculator Plugin Methods

The Quote Calculator Plugin can reference these seven methods. You can export any, all, or none of them to achieve your desired behavior.

Calculating True End Date and Subscription Term

Use JavaScript to make a Quote Line Calculator plugin that calculates values and stores maximum values for the custom quote line fields True Effective End Date and True Effective Term.

Custom Package Total Calculation

The sample JavaScript script can be used in the Quote Line Calculator to calculate the total price for all components in a quote line and then store that value in a custom field.

Find Lookup Records

Use this sample JavaScript script in the Quote Line Calculator to query records within the plugin and to set each quote line's Description field using fields from those records.

Insert Records

The sample JavaScript script can be used in the Quote Line Calculator to insert records.

Javascript Page Security Plugin

Use Javascript functions to control field visibility and editability on your CPQ quotes.

Legacy Page Security Plugin (Apex)

The Salesforce CPQ Apex page security plugins let developers control field-level visibility or data entry mode in Salesforce CPQ VisualForce pages.

Guidelines for Heroku in Quote Calculator Plugins

Salesforce CPQ quote calculator plugins call Heroku to perform asynchronous calculations. When you write a quote calculator plugin, review important guidelines for working with the Heroku service.

Quote Calculator Plugin Guidelines

Consider these key guidelines when planning scripts for the Javascript Quote Calculator Plugin.

EDITIONS

Available in: Salesforce CPQ Winter '16 and later

Promises

A Promise is a built-in JavaScript object that allows for asynchronous programming in the browser. Promises let you delay a certain action until another one has completed. Promises support a `.then(success, failure)` method, where success is a function called when the promise resolves successfully, and failure is a function called when the promise is rejected. If you want to do any asynchronous programming in the plugin, such as a server callout, you must return a promise that resolves once that action is completed. This guarantees that calculation steps occur in the proper order. If a method doesn't require asynchronous behavior, you can return a promise that resolves immediately as `return`

`Promise.resolve();`. Promises can resolve to a value, which passes as a parameter to the `.then()` callbacks. You can use this fact in your own code, but remember that the promises that these methods return don't need to resolve to a value. Always directly modify the quote and line models provided in the parameters.

QuoteModel and QuoteLineModel Types

The JavaScript calculator represents Quote__c and QuoteLine__c objects as QuoteModel and QuoteLineModel objects respectively. You can access the underlying SObject through the `.record` property on both objects, which lets you reference fields by using their API name. For example, you can reference a custom field SBQQ__MyCustomField__c on a given QuoteLineModel by accessing the attribute record ["SBQQ__MyCustomField__c"]. You can also reference fields on related records. For example, if you want to reference the field MyField__c on an account associated with a quote, access the record ["Account__r"]["MyField__c"].

External fields aren't loaded by default. To use an external field, such as one from an opportunity or account, create a custom quote formula field that pulls in the value of the desired field. Then include the custom quote field in your custom script. You can also reference this external field in a price action formula to preload and then include it in your custom script.

Salesforce Field Types

You can change records stored in JavaScript Object Notation, or JSON. These records are serialized from your org. Number, Text, and Boolean fields are all stored without any conversion, but you can convert any other type. For example, dates are represented as strings of the format "YYYY-MM-DD." If you reference or change a field containing a date, you have to preserve that format.

JSForce

JSForce is a third-party library that provides a unified way to perform queries, execute Apex REST calls, use the Metadata API, or make HTTP requests remotely. Methods access jsforce through the optional parameter `conn`.

Note:

- The JSQCP is an ES6 module. It is transpiled via Babel and module-scoped by default. You can use any elements of the ES6 language or syntax. However, the plugin must be able to run in both browser and node environments. Global browser variables such as window may not be available.
- With plugins, callouts (requests) to non-Salesforce endpoints aren't supported for asynchronous calculations. For example, `requestGet` fails in asynchronous calculations.

Field Availability

Javascript Quote Calculator plugins don't support custom fields on consumption rates and consumption schedules.

Character Limits

You can't increase the maximum character limit of a custom script in Javascript Quote Calculator Plugin.

Quote Calculator Plugin Methods

The Quote Calculator Plugin can reference these seven methods. You can export any, all, or none of them to achieve your desired behavior.

EDITIONS

Available in: Salesforce CPQ Winter '16 and later

API Version Management

In general, Salesforce CPQ uses Salesforce API that's one version behind the newest Salesforce API. For example, Salesforce Summer '21 uses Salesforce API version 52.0, so Salesforce CPQ Summer '21 uses Salesforce API version 51.0.

If you need to reference entities or fields Salesforce API version that's newer than what you're using in Salesforce CPQ, use the JSforce property assignment `conn.version='x';`, replacing `x` with the version that you want to use. For example, the following method shows how to overwrite the default API version to version 52.0.

```
export function onInit(quote, conn) {
   conn.version = '52.0';
   return conn.query("SELECT Name FROM ConsumptionSchedule")
      .then(function (results) {
         console.log(results);
         return Promise.resolve();
      })
}
```

onInit

Param	Type	Description
{QuoteLineModel[]}	quoteLineModels	An array containing Javascript representations of all lines in a quote.

The calculator calls this method before formula fields are evaluated. Returns `{promise}`.

```
export function onInit(quoteLineModels) {
return Promise.resolve();
};
```

onBeforeCalculate

Param	Type	Description
{QuoteModel}	quoteModel	Javascript representation of the quote you're evaluating
(QuoteLineModel[]}	quoteLineModels	An array containing Javascript representations of all lines in the quote

The calculator calls this method before calculation begins, but after formula fields have been evaluated. Returns `{promise}`.

```
export function onBeforeCalculate(quoteModel, quoteLineModels) {
return Promise.resolve();
};
```

onBeforePriceRules

Param	Type	Description
{QuoteModel}	quoteModel	Javascript representation of the quote you're evaluating
(QuoteLineModel[]}	quoteLineModels	An array containing Javascript representations of all lines in the quote

The calculator calls this method before it evaluates price rules. Returns `{promise}`.

```
export function onBeforePriceRules(quoteModel, quoteLineModels) {
return Promise.resolve();
};
```

onAfterPriceRules

Param	Type	Description
{QuoteModel}	quoteModel	Javascript representation of the quote you're evaluating
(QuoteLineModel[]}	quoteLineModels	An array containing Javascript representations of all lines in the quote

The calculator calls this method after it evaluates price rules. Returns `{promise}`.

```
export function onAfterPriceRules(quoteModel, quoteLineModels) {
return Promise.resolve();
};
```

onAfterCalculate

Param	Type	Description
{QuoteModel}	quoteModel	Javascript representation of the quote you're evaluating
(QuoteLineModel[]}	quoteLineModels	An array containing Javascript representations of all lines in the quote

The calculator calls this method after it completes a calculation, but before re-evaluating formula fields. Returns `{promise}`

```
export function onAfterCalculate(quoteModel, quoteLineModels) {
return Promise.resolve();
};
```

isFieldVisible

Note: This method can't be used to alter data.

Param	Type	Description
{FieldName}	String	Name of the field that will be hidden or made visible
(QuoteLineModelRecord}	quoteLineModelRecord	Javascript representation of the SObject record of line you're evaluating

The calculator calls this method after it completes a calculation. Returns `{Boolean}`

```
export function isFieldVisible(fieldName, quoteLineModelRecord) {

if (fieldName == 'SBQQ__Description__c') {
return false;
}

return true;

};
```

isFieldEditable

Note: This method can't be used to alter data.

Param	Type	Description
{FieldName}	String	Name of the field that will be made read-only or editable
(QuoteLineModelRecord}	quoteLineModelRecord	Javascript representation of the SObject record of line you're evaluating

The calculator calls this method after it completes a calculation. Returns `{Boolean}`

```
export function isFieldEditable(fieldName, quoteLineModelRecord) {

if (fieldName == 'SBQQ__Description__c') {
return false;
}

return true;

};
```

Calculating True End Date and Subscription Term

Use JavaScript to make a Quote Line Calculator plugin that calculates values and stores maximum values for the custom quote line fields True Effective End Date and True Effective Term.

EDITIONS

Available in: Salesforce CPQ Winter '16 and later

1. On the quote line object, create the following custom fields.
   a. A date field with the API name True_Effective_End_Date__c
   b. A number field with the API name True_Effective_Term__c
2. Create a custom script record with a name of your choosing.
   a. In the Quote Line Fields field, add *True_Effective_End_Date__c* and *True_Effective_Term__c*.
   b. In the Code field, provide Javascript code that exports all of the methods that the calculator looks for and documents their parameters and return types. Save your custom script and add its name to the Quote Calculator Plugin field in the Plugins tab of Salesforce CPQ package settings. We've provided a sample custom script below.

```
export function onAfterCalculate(quote, lineModels) {

    var maxEffectiveEndDate = null;
    var maxEffectiveTerm = 0;
    if (lineModels != null) {
        lineModels.forEach(function (line) {
            var trueEndDate = calculateEndDate(quote, line);
            var trueTerm = getEffectiveSubscriptionTerm(quote, line);
            if (maxEffectiveEndDate == null || (maxEffectiveEndDate < trueEndDate))
{
                maxEffectiveEndDate = trueEndDate;
            }
            if (maxEffectiveTerm < trueTerm) {
                maxEffectiveTerm = trueTerm;
            }
            line.record["True_Effective_End_Date__c"] = toApexDate(trueEndDate);
            line.record["True_Effective_Term__c"] = trueTerm;
        });
       quote.record["True_Effective_End_Date__c"] = toApexDate(maxEffectiveEndDate);

        quote.record["True_Effective_Term__c"] = maxEffectiveTerm;
    }
    return Promise.resolve()
}

function calculateEndDate(quote, line) {
    var sd = new Date(line.record["SBQQ__EffectiveStartDate__c"]);
    var ed = new Date(line.record["SBQQ__EffectiveEndDate__c"]);
    if (sd != null && ed != null ) {
        ed = sd;
       ed.setUTCMonth(ed.getUTCMonth() + getEffectiveSubscriptionTerm(quote, line));

        ed.setUTCDate(ed.getUTCDate() - 1);
    }
    return ed;
}

function getEffectiveSubscriptionTerm(quote, line) {
```

```
   if (line.record["SBQQ__EffectiveStartDate__c"] != null){
        var sd = new Date(line.record["SBQQ__EffectiveStartDate__c"]);
   }
    if (line.record["SBQQ__EffectiveEndDate__c"] != null){
       var ed = new Date(line.record["SBQQ__EffectiveEndDate__c"]);
   }
    if (sd != null && ed != null ) {
        ed.setUTCDate(ed.getUTCDate() + 1);
        return monthsBetween(sd, ed);
    } else if (line.SubscriptionTerm__c != null) {
        return line.SubscriptionTerm__c;
    } else if (quote.SubscriptionTerm__c != null) {
        return quote.SubscriptionTerm__c;
    } else {
        return line.DefaultSubscriptionTerm__c;
    }
}

/**
 * Takes a JS Date object and turns it into a string of the type 'YYYY-MM-DD', which
 is what Apex is expecting.
 * @param {Date} date The date to be stringified
 * @returns {string}
 */
function toApexDate(/*Date*/ date) {
    if (date == null) {
        return null;
    }
    // Get the ISO formatted date string.
    // This will be formatted: YYYY-MM-DDTHH:mm:ss.sssZ
    var dateIso = date.toISOString();

    // Replace everything after the T with an empty string
    return dateIso.replace(new RegExp('[Tt].*'), "");
}

function monthsBetween(/*Date*/ startDate, /*Date*/ endDate) {
    if(startDate != null && endDate != null ){
    // If the start date is actually after the end date, reverse the arguments and
multiply the result by -1
    if (startDate  > endDate) {
        return -1 * this.monthsBetween(endDate, startDate);
    }
    var result = 0;
    // Add the difference in years * 12
    result += ((endDate.getUTCFullYear() - startDate.getUTCFullYear()) * 12);
    // Add the difference in months. Note: If startDate was later in the year than
endDate, this value will be
    // subtracted.
    result += (endDate.getUTCMonth() - startDate.getUTCMonth());
    return result;
    }
    return 0;
}
```

Custom Package Total Calculation

The sample JavaScript script can be used in the Quote Line Calculator to calculate the total price for all components in a quote line and then store that value in a custom field.

This sample JavaScript code exports all of the methods that the calculator looks for, and documents their parameters and return types.

Note: The sample script assumes the Salesforce admin created a custom field `Component Custom Total` on the Quote Line object.

EDITIONS

Available in: Salesforce CPQ Winter '16 and later

Javascript

```
export function onInit(lines) {
    if (lines != null) {
        lines.forEach(function (line) {
            line.record["Component_Custom_Total__c"] = 0;
        });
    }
};

export function onAfterCalculate(quoteModel, quoteLines) {
    if (quoteLines != null) {
        quoteLines.forEach(function (line) {
            var parent = line.parentItem;
            if (parent != null) {
                var pComponentCustomTotal = parent.record["Component_Custom_Total__c"] ||
 0;
                var cListPrice = line.ProratedListPrice__c || 0;
                var cQuantity = line.Quantity__c == null ? 1 : line.Quantity__c;
                var cPriorQuantity = line.PriorQuantity__c || 0;
                var cPricingMethod = line.PricingMethod__c == null ? "List" :
line.PricingMethod__c;
                var cDiscountScheduleType = line.DiscountScheduleType__c || '';
                var cRenewal = line.Renewal__c || false;
                var cExisting = line.Existing__c || false;
                var cSubscriptionPricing = line.SubscriptionPricing__c || '';

                var cTotalPrice = getTotal(cListPrice, cQuantity, cPriorQuantity,
cPricingMethod, cDiscountScheduleType, cRenewal, cExisting, cSubscriptionPricing,
cListPrice);
                pComponentCustomTotal += cTotalPrice;

                parent.record["Component_Custom_Total__c"] = pComponentCustomTotal;
            }
        });
    }
};

function getTotal(price, qty, priorQty, pMethod, dsType, isRen, isExist, subPricing,
listPrice) {
    if ((isRen === true) && (isExist === false) && (priorQty == null)) {
       // Personal note: In onAfterCalculate, we specifically make sure that priorQuantity
 can't be null.
```

```
        // So isn't this loop pointless?
        return 0;
    } else {
       return price * getEffectiveQuantity(qty, priorQty, pMethod, dsType, isRen, isExist,
 subPricing, listPrice);
    }
}

function getEffectiveQuantity(qty, priorQty, pMethod, dsType, isRen, exists, subPricing,
listPrice) {
    var delta = qty - priorQty;

    if (pMethod == 'Block' && delta == 0) {
        return 0;
    } else if (pMethod == 'Block') {
        return 1;
    } else if (dsType == 'Slab' && (delta == 0 || (qty == 0 && isRen == true))) {
        return 0;
    } else if (dsType == 'Slab') {
        return 1;
    } else if (exists == true && subPricing == '' && delta < 0) {
        return 0;
    } else if (exists == true && subPricing == 'Percent Of Total' && listPrice != 0 &&
delta >= 0) {
        return qty;
    } else if (exists == true) {
        return delta;
    } else {
        return qty;
    }
}
```

Find Lookup Records

Use this sample JavaScript script in the Quote Line Calculator to query records within the plugin and to set each quote line's Description field using fields from those records.

Each version of these JavaScript code samples exports all of the methods that the calculator looks for, and documents their parameters and return types.

Use `conn.query` before you use loops that require lookup information so that you can avoid JSON errors with resolving loops.

EDITIONS

Available in: Salesforce CPQ Winter '16 and later

Javascript

```
export function onAfterCalculate(quote, lines, conn) {
 if (lines.length > 0) {
  var productCodes = [];
  lines.forEach(function(line) {
   if (line.record['SBQQ__ProductCode__c']) {
    productCodes.push(line.record['SBQQ__ProductCode__c']);
   }
```

```
  });
  if (productCodes.length) {
   var codeList = "('" + productCodes.join("', '") + "')";
   /*
    * conn.query() returns a Promise that resolves when the query completes.
    */
   return conn.query('SELECT Id, SBQQ__Category__c, SBQQ__Value__c FROM SBQQ__LookupData__c
 WHERE SBQQ__Category__C IN ' + codeList)
    .then(function(results) {
     /*
      * conn.query()'s Promise resolves to an object with three attributes:
      * - totalSize: an integer indicating how many records were returned
      * - done: a boolean indicating whether the query has completed
      * - records: a list of all records returned
      */
     if (results.totalSize) {
      var valuesByCategory = {};
      results.records.forEach(function(record) {
       valuesByCategory[record.SBQQ__Category__c] = record.SBQQ__Value__c;
      });
      lines.forEach(function(line) {
       if (line.record['SBQQ__ProductCode__c']) {
        line.record['SBQQ__Description__c'] =
valuesByCategory[line.record['SBQQ__ProductCode__c']] || '';
       }
      });
     }
    });
  }
 }
 return Promise.resolve();
}
```

Javascript Using Method-Chaining

This plugin uses method-chaining style to construct the query, which is useful when you want to dynamically construct your queries.

```
/**
 * Created on 9/27/16.
 */
export function onAfterCalculate(quote, lines, conn) {
 if (lines.length) {
  var codes = [];
  lines.forEach(function(line) {
   var code = line.record['SBQQ__ProductCode__c'];
   if (code) {
    codes.push(code);
   }
  });
  if (codes.length) {
   var conditions = {
    SBQQ__Category__c: {$in: codes}
   };
   var fields = ['Id', 'Name', 'SBQQ__Category__c', 'SBQQ__Value__c'];
```

```
   /*
    * Queries can also be constructed in a method-chaining style.
    */
   return conn.sobject('SBQQ__LookupData__c')
    .find(conditions, fields)
    .execute(function(err, records) {
     if (err) {
      return Promise.reject(err);
     } else {
      var valuesByCategory = {};
      records.forEach(function(record) {
       valuesByCategory[record.SBQQ__Category__c] = record.SBQQ__Value__c;
      });
      lines.forEach(function(line) {
       if (line.record['SBQQ__ProductCode__c']) {
        line.record['SBQQ__Description__c'] =
valuesByCategory[line.record['SBQQ__ProductCode__c']] || '';
       }
      });
     }
    });
  }
 }
 return Promise.resolve();
}
```

Insert Records

The sample JavaScript script can be used in the Quote Line Calculator to insert records.

This sample JavaSciprt code exports all of the methods that the calculator looks for, and documents their parameters and return types.

EDITIONS

Available in: Salesforce CPQ Winter '16 and later

```
export function onAfterCalculate(quote, lines, conn) {
 if (lines.length) {
  var codes = [];
  lines.forEach(function(line) {
   var code = line.record['SBQQ__ProductCode__c'];
   if (code) {
    codes.push(code);
   }
  });
  if (codes.length) {
   var conditions = {
    SBQQ__Category__c: {$in: codes}
   };
   var fields = ['Id', 'Name', 'SBQQ__Category__c', 'SBQQ__Value__c'];
   return conn.sobject('SBQQ__LookupData__c')
    .find(conditions, fields)
    .execute(function(err, records) {
     console.log(records);
```

```
    if (err) {
     return Promise.reject(err);
    } else {
     var valuesByCategory = {};
     records.forEach(function(record) {
      valuesByCategory[record.SBQQ__Category__c] = record.SBQQ__Value__c;
     });
     var newRecords = [];
     lines.forEach(function(line) {
      var code = line.record['SBQQ__ProductCode__c'];
      var desc = line.record['SBQQ__Description__c'];
      if (code && desc && !valuesByCategory[code]) {
       newRecords.push({
        SBQQ__Category__c: code,
        SBQQ__Value__c: line.record['SBQQ__Description__c']
       });
      }
     });
     if (newRecords.length) {
      return conn.sobject('SBQQ__LookupData__c')
       .create(newRecords, function(err, ret) {
        console.log(ret);
       });
     }
    }
   });
  }
 }
 return Promise.resolve();
}
```

Javascript Page Security Plugin

Use Javascript functions to control field visibility and editability on your CPQ quotes.

EDITIONS

Available in: Salesforce CPQ Summer '15 and later

The Javascript Page Security plugin supports four functions. The functions `isFieldVisible` and `isFieldEditable` are available starting in Salesforce CPQ Summer '15 and control quote line field visibility and editability. The functions `isFieldVisibleForObject` and `isFieldEditableForObject` are available starting in Salesforce CPQ Summer '19 and can control field visibility and editability for both quote fields and quote line fields. When a method using one of these functions returns False, Salesforce CPQ locks or hides the chosen fields. The fields are unchanged if the method returns Null or True.

Because `isFieldVisibleForObject` and `isFieldEditableForObject` can accept a quote or quote line, we recommend naming your object parameter `quoteOrLine`.

Note:

- Salesforce CPQ prioritizes field-level security over page security plugins. If a field is read-only and an editability function for that field returns True, the field remains read-only.
- Use the page security plugin only for hiding, showing, and adjusting the editability of fields. If you want change field values, use the Javascript Quote Calculator Plugin.

- The quote line editor shows blank empty spaces for quote line drawer fields hidden by the page security plugin. To remove these spaces, go to Salesforce CPQ line editor package settings and select **Enable Compact Mode**.

To create a page security plugin, define your code in a custom script record and then reference that record's name in the Quote Calculator Plugin field within Salesforce CPQ Plugin package settings. If you're already using a quote calculator plugin in that field, you can add your page security plugin code to the calculator plugin's custom script record.

Table 1: isFieldVisible (Summer '15)

Parameter	Type	Definition
`fieldname`	string	If `isFieldVisible` returns False, this quote line field is hidden.
`line`	SObject	The quote line object
`conn`	Object	Methods access jsforce through the optional parameter `conn`.

Table 2: isFieldEditable (Summer '15)

Parameter	Type	Definition
`fieldname`	string	If `isFieldEditable` returns False, this quote line field is locked from edits.
`line`	SObject	The quote line object
`conn`	Object	Methods access jsforce through the optional parameter `conn`.

Table 3: isFieldVisibleForObject (Summer '19)

Parameter	Type	Definition
`fieldName`	String	A field on the quote or quote line. If `isFieldVisibleForObject` returns False, this field is hidden.
`quoteOrLine`	SObject	The object containing the field that you're evaluating to determine whether `fieldName` is visible. Can be a quote or a quote line.
`conn`	Object	Methods access jsforce through the optional parameter `conn`.
`objectName`	String	The object that contains `fieldName`. If `quoteOrLine` is evaluating a quote, use *`Quote__c`*. If `quoteOrLine` is evaluating a quote line, use *`QuoteLine__c`*. Leave this parameter undefined to target the same field on the quote and the quote line.

Table 4: isFieldEditableForObject (Summer '19)

Parameter	Type	Definition
`fieldName`	String	A field on the quote or quote line. If `isFieldEditableForObject` returns False, this field is locked from edits.
`quoteOrLine`	SObject	The object containing the field that you're evaluating to determine whether `fieldName` is editable. Can be a quote or a quote line.
`conn`	Object	Methods access jsforce through the optional parameter `conn`.
`objectName`	String	The object that contains `fieldName`. If `quoteOrLine` is evaluating a quote, use *`Quote__c`*. If `quoteOrLine` is evaluating a quote line, use *`QuoteLine__c`*. Leave this parameter undefined to target the same field on the quote and the quote line.

We strongly recommend that users on Salesforce CPQ Summer '19 and later use the new functions given their improved flexibility. If your plugin uses pre-Summer '19 functions with `isFieldEditableForObject` or `isFieldVisibleForObject` functions that use the `line` parameter, Salesforce CPQ ignores the new functions and uses the old functions instead.

To specify whether your changes apply to a field on the quote or on the quote line, use an `if` statement for your `objectName` in the `isFieldVisibleForObject` or `isFieldEditableForObject` code block. For example, in the following code segment, we're targeting the Markup Rate field on the quote.

```
export function isFieldEditableForObject(fieldName, quoteOrLine, conn, objectName){
  if (objectName === 'Quote__c' && fieldName === 'SBQQ__MarkupRate__c')
```

However, the following code segment targets Markup Rate on both the quote and the quote line.

```
export function isFieldEditableForObject(fieldName, quoteOrLine, conn, objectName){
  if fieldName === 'SBQQ__MarkupRate__c'
```

Example: In this example, if a quote's Customer Discount is greater than 10%, we lock the quote's Markup Rate field from edits.

```
export function isFieldEditableForObject(fieldName, quoteOrLine, conn, objectName){
  if (objectName === 'Quote__c' && fieldName === 'SBQQ__MarkupRate__c') {
     if (quoteOrLine.SBQQ__CustomerDiscount__c > 10) {
       return false;
  }
 }
}
```

Example: In this example, if a quote line's Distributor Discount is greater than 10%, we hide the quote line's Markup Rate field.

```
export function isFieldVisibleForObject(fieldName, quoteOrline, conn, objectName){
  if (objectName === 'QuoteLine__c' && fieldName === 'SBQQ__MarkupRate__c') {
```

```
    if (quoteOrLine.SBQQ__CustomerDiscount__c > 10) {
      return false;
   }
 }
}
```

Example: One function can also evaluate and act on quote and quote line fields at the same time, including twin fields. In this example, if a quote's Customer Discount is greater than 10%, we lock the quote's Markup Rate field from edits. If the quote line's Distributor Discount is greater than 10%, we hide the quote line's Markup Rate field.

Example:

```
export function isFieldEditableForObject(fieldName, quoteOrLine, conn, objectName){
  if (objectName === 'Quote__c' && fieldName === 'SBQQ__MarkupRate__c') {
    if (quoteOrLine.SBQQ__CustomerDiscount__c > 10) {
      return false;
  }
}

if (objectName === 'QuoteLine__c' && fieldName === 'SBQQ__MarkupRate__c') {
  if (quoteOrLine.SBQQ__DistributorDiscount__c > 10) {

      return false;
  }
 }
}
```

Legacy Page Security Plugin (Apex)

The Salesforce CPQ Apex page security plugins let developers control field-level visibility or data entry mode in Salesforce CPQ VisualForce pages.

EDITIONS

Available in: All Salesforce CPQ Editions

Note: Salesforce CPQ has deprecated support for Apex page security plugins. Review Javascript Page Security Plugin for information on the currently-supported version.

The Legacy Page Security Plugin handles two types of use cases.

Show or hide fields on each quote line

For example, you're selling training classes and you want to capture how many students are participating in the class. Use the page security plugin to hide a student number field.

Make quote line fields read-only or writable

For example, you allow your users to specify the subscription term on each quote line, but you have some products that can only be quoted on a 12-month basis. Use the page security plugin to make the Subscription Term field read-only for such products, while keeping it writable for the other products.

To use the Legacy Page Security Plugin, first create an Apex class. Then enter the Apex class name in the Legacy Page Security Plugin setting in the Salesforce CPQ package settings. You can call only one Apex class at a time in the Legacy Page Security Plugin.

Example:

```
global class MyPageSecurityPlugin implements SBQQ.PageSecurityPlugin2 {
    public Boolean isFieldEditable(String pageName, Schema.SObjectField field) {
        return null;
```

```
    }

   public Boolean isFieldEditable(String pageName, Schema.SObjectField field, SObject
 record) {
        return null;
    }

    public Boolean isFieldVisible(String pageName, Schema.SObjectField field) {
        return null;
    }

    public Boolean isFieldVisible(String pageName, Schema.SObjectField field, SObject
 record) {
        if ((pageName == 'EditLines') && (record instanceof SBQQ__QuoteLine__c)) {
            SBQQ__QuoteLine__c line = (SBQQ__QuoteLine__c)record;
            if ((line.SBQQ__Bundle__c == true) && (field !=
SBQQ__QuoteLine__c.SBQQ__ProductName__c)) {
                return false;
            }
        }
        return null;
    }
}
```

Guidelines for Heroku in Quote Calculator Plugins

Salesforce CPQ quote calculator plugins call Heroku to perform asynchronous calculations. When you write a quote calculator plugin, review important guidelines for working with the Heroku service.

- Quote calculator plugins perform synchronous calculations in the quote line editor UI, within a standard web browser with all expected platform and browser information available. However, asynchronous calculations occur within a Heroku application outside of the web browser. If your plugin must reference the state of the platform running the calculation, make sure to account for whether the quote line editor or Heroku is handling the calculation.
- If your plugin makes callouts to an endpoint that you own, make sure that both the local Salesforce host and your external Heroku host can access the endpoint URI.
- The total time for a calculation plus the time for a callout to Heroku from your system can't be longer than 30 seconds. Otherwise, Heroku will terminate the calculation.

Product Search Plugin

The Salesforce CPQ product search plugin is an interface that you can implement to customize product search results in the Product Search page and the Guided Selling page. The plugin methods vary slightly between Product Search and Guided Selling implementations.

EDITIONS

Available in: All Salesforce CPQ Editions

Product Search Plugin - Product Search Interface

Use implemented SBQQ.ProductSearchPlugin methods to further filter a product search on the Product Search page after users enter their own search queries.

SBQQ.ProductSearchPlugin - Guided Selling Interface

Use implemented SBQQ.ProductSearchPlugin methods to further filter a prompt on the Guided Selling UI.

Product Search Plugin - Product Search Interface

Use implemented SBQQ.ProductSearchPlugin methods to further filter a product search on the Product Search page after users enter their own search queries.

Namespace

SBQQ

Usage

For example, in Product Search, you could configure the plugin to return all search results in descending order from the most recent Last Ordered Date. If a sales rep enters "Tablets," the search results show tablets starting from the most recent Last Ordered Date field value. Users can then further filter through the Product Search filter panel, if necessary.

Product search plugins can use only a subset of CPQ quote fields by default. If you can't pass a field to your product search, or if it passes as null, you must instead pull it with a SOQL query using the ID passed with the quote model.

Note: Date fields are returned as strings in the yyyy-mm-dd format.

Method Order of Execution

Salesforce CPQ uses the following order to execute the SBQQ.ProductSearchPlugin implemented methods for a Product Search.

```
//**The constructor is optional**//
* 1.0 Constructor()
 * 2.0 FOREACH(Search Field){
 *   2.1 isFilterHidden()
 *   2.1 getFilterDefaultValue()
 * }
 * 3.0 isSearchCustom (CUSTOM vs ENHANCED)
 * IF(isCustom){
 *  4.0 search()
 * }
 * ELSE{
 *   4.0 getAdditionalSearchFilters()
 * }
```

1. The Constructor can be called first, but it's not required for implementation.
2. Salesforce CPQ calls the following two methods for each search input.
   - isFilterHidden: Determines whether to hide the search input from the quote line editor
   - getFilterDefaultValue: Sets the field value for the initial search
3. Salesforce CPQ calls isSearchCustom to determine whether you're using Custom or Enhanced searching.
4. If isSearchCustom returned True, Salesforce CPQ calls search(). This method gives you full control of the search query - you'll build the Select Clause and Where Clause manually, then build and perform the query.
5. If isSearchCustom returned False, Salesforce CPQ calls getAdditionalSearchFilters. This method appends a WHERE clause to the existing SOQL query.

SBQQ.ProductSearchPlugin Product Search Methods

SBQQ.Product Search Plugin - Product Search Example Implementation

SBQQ.ProductSearchPlugin Product Search Methods

The following are methods for a Product Search implementation of `SBQQ.ProductSearchPlugin`.

getAdditionalSearchFilters(quote, fieldValuesMap)
Appends a WHERE clause to the SOQL query used for the product search, so that you can further refine a user's search input. Salesforce CPQ calls this method only when `isSearchCustom` returns FALSE.

getFilterDefaultValue(quote, fieldName)
Determines the value for the initial search. Salesforce CPQ calls this implemented method for each input field.

isFilterHidden(quote, fieldName)
Determines the visibility of a filter in the UI. Return `True` to hide the filter and `False` to let users see the filter. Salesforce CPQ calls this implemented method for each search input field.

isSearchCustom(quote, fieldValuesMap)
Called after `isFilterHidden` and `getFilterDefaulValue`. Returns `True` if the plugin uses custom searching or `False` if the plugin uses enhanced searching.

search(quote, fieldValuesMap)
Overrides the entire user search input. Salesforce CPQ calls this method only when `isSearchCustom` returns TRUE.

`getAdditionalSearchFilters(quote, fieldValuesMap)`

Appends a WHERE clause to the SOQL query used for the product search, so that you can further refine a user's search input. Salesforce CPQ calls this method only when `isSearchCustom` returns FALSE.

Signature

```
global String getAdditionalSearchFilters(SObject quote, Map<String,Object>
fieldValuesMap)
```

Parameters

`quote`
Type: SObject

The current quote.

`fieldValuesMap`
Type: Map<String,Object>

A map of the search criteria. The Key is a Product2 API name and the value is the desired search value.

Return Value

Type: String

The additional search filter, as a SOQL query formatted as a string. For example, `'AND Product2.Inventory_Level__c > 3'`

Example

In this method, we take an existing filter for products in a Hardware family, and append an additional filter for inventory levels greater than 3.

```
global String getAdditionalSearchFilters(SObject quote, Map<String, Object> fieldValuesMap)
 {
  String additionalFilter = NULL;
  if(fieldValuesMap.get('Family') == 'Hardware') {
    additionalFilter = 'AND Product2.Inventory_Level__c > 3';
  }
  return additionalFilter;
}
```

`getFilterDefaultValue(quote, fieldName)`

Determines the value for the initial search. Salesforce CPQ calls this implemented method for each input field.

Signature

```
global String getFilterDefaultValue(SObject quote, String fieldName)
```

Parameters

`quote`
Type: SObject

The current quote.

`fieldName`
Type: String

Quote field used in evaluations to determine the method's output.

Return Value

Type: String

Returns the string value used as the filter's initial search input.

Example

In this example, the method sets the product family filter's value to Service if the quote has a type of Quote.

```
global String getFilterDefaultValue(SObject quote, String fieldName) {
  // This would set Product Family filter to Service if Quote Type is Quote
  return (fieldName == 'Family' && quote.SBQQ__Type__c. == 'Quote') ? 'Service' : null;
}
```

`isFilterHidden(quote, fieldName)`

Determines the visibility of a filter in the UI. Return `True` to hide the filter and `False` to let users see the filter. Salesforce CPQ calls this implemented method for each search input field.

Signature

```
global Boolean isFilterHidden(SObject quote, String fieldName)
```

Parameters

quote
Type: SObject

The quote object containing the field used to determine whether the method returns true or false.

fieldName
Type: String

The field tested to determine whether the method returns true or false.

Return Value

Type: Boolean

Return `True` to hide the filter and `False` to let users see the filter.

Example

This example shows a method that returns `True` and hides the filter if the quote's status has a value of `Approved`.

```
global Boolean isFilterHidden(SObject quote, String fieldName) {
  // This would hide Product Code filter if Quote Status is Approved
  return fieldName == 'ProductCode' && quote.SBQQ__Status__c. == 'Approved';
}
```

`isSearchCustom(quote, fieldValuesMap)`

Called after `isFilterHidden` and `getFilterDefaulValue`. Returns `True` if the plugin uses custom searching or `False` if the plugin uses enhanced searching.

Signature

```
global Boolean isSearchCustom(SObject quote, Map<String,Object> fieldValuesMap)
```

Parameters

quote
Type: SObject

The current quote.

fieldValuesMap
Type: Map<String,Object>

A map of the search criteria. The map key is a Product2 API name and the value is the desired search value.

Return Value

Type: Boolean

Return `True` to use custom searching or `False` to use enhanced searching.

If you use custom searching, Salesforce CPQ calls `search` for search execution. The `search()` method lets you build the SOQL query's SELECT and WHERE clauses manually before you perform your query.

If you use enhanced searching, Salesforce CPQ calls `getAdditionalSearchFilters` for search execution. The `getAdditionalSearchFilters` method appends a WHERE clause to the existing SOQL query.

Example

In this example, we want a method that returns `True` if the original search criteria defined and used a Search field for sorting.

```
global Boolean isSearchCustom(SObject quote, Map<String,Object> fieldValuesMap) {
  // This would use CUSTOM mode if a Search field for sorting was defined and used
  return fieldValuesMap.get('Sort_By__c') != '';
}
```

search(quote, fieldValuesMap)

Overrides the entire user search input. Salesforce CPQ calls this method only when `isSearchCustom` returns TRUE.

Signature

```
global List<PricebookEntry> search(SObject quote, Map<String,Object> fieldValuesMap)
```

Parameters

quote
Type: SObject

The current quote.

fieldValuesMap
Type: Map<String,Object>

A map of the search criteria. The map key is a Product2 API name and the value is the desired search value. Contains only keys for non-null values.

Return Value

Type: List<PricebookEntry>

Example

This example builds and returns a list of price book entries.

```
global List<PricebookEntry> search(SObject quote, Map<String,Object> fieldValuesMap){
  // Get all possible filter fields from the search filter field set
  List<Schema.FieldSetMember> searchFilterFieldSetFields =
SObjectType.Product2.FieldSets.SBQQ__SearchFilters.getFields();
  // Get all possible fields from the search result field set
  List<Schema.FieldSetMember> searchResultFieldSetFields =
SObjectType.Product2.FieldSets.SBQQ__SearchResults.getFields();
  // Build the Select string
  String selectClause = 'SELECT ';
  for(Schema.FieldSetMember field : searchResultFieldSetFields) {
    selectClause += 'Product2.' + field.getFieldPath() + ', ';
```

```
  }
  selectClause += 'Id, UnitPrice, Pricebook2Id, Product2Id, Product2.Id';
  // Build the Where clause
  String whereClause = '';
  for(Schema.FieldSetMember field : searchFilterFieldSetFields) {
    if(!fieldValuesMap.containsKey(field.getFieldPath())) {
      continue;
    }
    if(field.getType() == Schema.DisplayType.String || field.getType() ==
Schema.DisplayType.Picklist) {
      whereClause += 'Product2.' + field.getFieldPath() + ' LIKE \'%' +
fieldValuesMap.get(field.getFieldPath()) + '%\' AND ';
    }
  }
  whereClause += 'Pricebook2Id = \'' + quote.get('SBQQ__Pricebook__c') + '\'';
  // Build the query
  String query = selectClause + ' FROM PricebookEntry WHERE ' + whereClause;
  // Perform the query
  List<PricebookEntry> pbes = new List<PricebookEntry>();
  pbes = Database.query(query);
  return pbes;
}
```

SBQQ.Product Search Plugin - Product Search Example Implementation

This is an example implementation of the `SBQQ.ProductSearchPlugin` interface.

```
global class ExampleProductSearchPlugin implements SBQQ.ProductSearchPlugin{

/**Constructor. Not required for implementation**/
global ExampleProductSearchPlugin(){
}

/**Product Search Methods**/
// if isSearchCustom returns True, the plugin uses search(), otherwise it uses
getAdditionalSearchFilters()
global Boolean isSearchCustom(SObject quote, Map<String,Object> fieldValuesMap){ return
true; }
global Boolean isFilterHidden(SObject quote, String fieldName){ return false; }
global String getFilterDefaultValue(SObject quote, String fieldName){ return NULL; }
global String getAdditionalSearchFilters(SObject quote, Map<String,Object> fieldValuesMap){
 return NULL; }
global List<PricebookEntry> search(SObject quote, Map<String,Object> fieldValuesMap){
return NULL; }
```

SBQQ.ProductSearchPlugin - Guided Selling Interface

Use implemented SBQQ.ProductSearchPlugin methods to further filter a prompt on the Guided Selling UI.

Namespace

SBQQ

Usage

You can configure the Product Search plugin to filter a guided selling prompt based on certain parameters when users enter a value. For example, in a guided selling prompt, you could configure the plugin to return all search results in descending order from the most recent Last Ordered Date. When the user chooses their input, the products returned in the search results are shown starting from the most recent Last Ordered Date field value.

Product search plugins can use only a subset of CPQ quote fields by default. If you can't pass a field to your guided selling input, or if it passes as null, you must retrieve it with a SOQL query.

Note: Date fields are returned as strings in the format yyyy-mm-dd.

Order of Execution

For a Guided Selling prompt, Salesforce CPQ executes the implemented SBQQ.ProductSearchPlugin methods in the following order.

```
//**The constructor is optional**//
* 1.0 Constructor()
 * 2.0 FOREACH(Search Field){
 *   2.1 isInputHidden()
 *   2.1 getInputDefaultValue()
 * }
 * 3.0 isSuggestCustom (CUSTOM vs ENHANCED)
 * IF(isCustom){
 *  4.0 suggest()
 * }
 * ELSE{
 *   4.0 getAdditionalSuggestFilters()
 * }
```

1. The constructor can be called first, but it's not required.
2. Salesforce CPQ calls the following two methods for each guided selling input.
   - `isInputHidden`: Determines whether to hide the guided selling input from the guided selling prompt
   - `getInputDefaultValue`: Sets the field value for the initial input
3. Salesforce CPQ calls `isSuggestCustom` to determine whether you're using custom or enhanced searching.
4. If `isInputCustom` returned TRUE, Salesforce CPQ calls `suggest`. This method gives you full control of the search query - you'll build the SELECT Clause and WHERE Clause manually, then build and perform the query.
5. If `isInputCustom` returned FALSE, Salesforce CPQ calls `getAdditionalSuggestFilters`. This method appends a WHERE clause to the existing SOQL query.

SBQQ.ProductSearchPlugin GuidedSelling Methods

SBQQ.ProductSearchPlugin - Guided Selling Example Implementation

SBQQ.ProductSearchPlugin GuidedSelling Methods

The following are methods for a guided selling implementation of `SBQQ.ProductSearchPlugin`.

getAdditionalSuggestFilters(quote, fieldValuesMap)
Appends a WHERE clause to the SOQL query used for a guided selling prompt. Salesforce CPQ Calls this method only when `isSuggestCustom` returns FALSE.

getInputDefaultValue(quote, fieldName)
Determines the input for the initial guided selling prompt.

isInputHidden(quote, fieldName)
Determines the visibility of an input in the Guided Selling UI. Return `True` to hide the input and `False` to let users see the input. Salesforce CPQ calls this method for each input.

isSuggestCustom(quote, fieldValuesMap)
Called after `isInputHidden` and `getInputDefaulValue`. Returns `True` for Salesforce CPQ to use Custom searching or `False` for Salesforce CPQ to use Enhanced searching.

suggest(quote, fieldValuesMap)
Overrides the user suggestion input. Salesforce CPQ calls this method only when `isSuggestCustom` returns TRUE.

`getAdditionalSuggestFilters(quote, fieldValuesMap)`

Appends a WHERE clause to the SOQL query used for a guided selling prompt. Salesforce CPQ Calls this method only when `isSuggestCustom` returns FALSE.

Signature

```
global String getAdditionalSuggestFilters(SObject quote, Map<String,Object>
fieldValuesMap)
```

Parameters

`quote`
Type: SObject

The current quote.

`fieldValuesMap`
Type: Map<String,Object>

A map of the guided selling suggestion criteria. The Key is a Product2 API name and the value is the desired suggest value.

Return Value

Type: String

The additional suggestion filter, as a WHERE clause. For example, `'AND Product2.Inventory_Level__c > 3'`

Example

```
    global String getAdditionalSuggestFilters(SObject quote, Map<String, Object>
inputValuesMap) {
```

```
        System.debug('METHOD CALLED: getAdditionalSuggestFilters');
        /**Adds an inventory check in an input = 'Yes' for an urgent shipment**/
        String additionalFilter = NULL;
        String isUrgent = 'No'
        if(inputValuesMap.containsKey('Urgent Shipment')){
            isUrgent = (String) inputValuesMap.get('Urgent Shipment');
        }
        if(isUrgent == 'Yes'){
            additionalFilter = 'AND Product2.Inventory_Level__c > 3';
        }
        return additionalFilter;
}
```

`getInputDefaultValue(quote, fieldName)`

Determines the input for the initial guided selling prompt.

Signature

```
global String getInputDefaultValue(SObject quote, String fieldName)
```

Parameters

quote
Type: SObject

The current quote.

fieldName
Type: String

Quote field used in evaluations to determine the method's output.

Return Value

Type: String

Returns the string value used as the guided selling prompt's initial input.

`isInputHidden(quote, fieldName)`

Determines the visibility of an input in the Guided Selling UI. Return `True` to hide the input and `False` to let users see the input. Salesforce CPQ calls this method for each input.

Signature

```
global Boolean isInputHidden(SObject quote, String fieldName)
```

Parameters

quote
Type: SObject

The current quote.

fieldName
Type: String

The field tested to determine whether the method returns true or false.

Return Value

Type: Boolean

Return `True` to hide the input and `False` to let users see the input.

Example

This example hides an input called "Urgent Shipment" on Fridays.

```
global Boolean isInputHidden(SObject quote, String input){
  /**Hides an input called 'Urgent Shipment' on Fridays**/
  return input == 'Urgent Shipment' && Datetime.now().format('F') == 5;
  }
```

isSuggestCustom(quote, fieldValuesMap)

Called after `isInputHidden` and `getInputDefaulValue`. Returns `True` for Salesforce CPQ to use Custom searching or `False` for Salesforce CPQ to use Enhanced searching.

Signature

```
global boolean isSuggestCustom(SObject quote, Map<String,Object> fieldValuesMap)
```

Parameters

quote
Type: SObject

The current quote.

fieldValuesMap
Type: Map<String,Object>

A map of the suggestion criteria. The map key is a Product2 API name and the value is the desired search value.

Return Value

Type: Boolean

Return `True` to use Custom searching or `False` to use Enhanced searching.

If you use Custom searching, Salesforce CPQ calls `search` for search execution. The `search()` method lets you build the SOQL query's SELECT and WHERE clauses manually before you perform your query.

If you use Enhanced searching, Salesforce CPQ calls `getAdditionalSearchFilters` for search execution. The `getAdditionalSearchFilters` method appends a WHERE clause to the existing SOQL query.

suggest(quote, fieldValuesMap)

Overrides the user suggestion input. Salesforce CPQ calls this method only when `isSuggestCustom` returns TRUE.

Signature

```
global List<PricebookEntry> suggest(SObject quote, Map<String,Object> fieldValuesMap)
```

Parameters

quote
Type: SObject

The current quote.

fieldValuesMap
Type: Map<String,Object>

A map of the search criteria. The map key is a Product2 API name and the value is the desired search value. Contains only keys for non-null values.

Return Value

Type: List<PricebookEntry>

Example

```
    /**
     * When Using Guided Selling in CUSTOM mode, Over-Ride entire search
     * Product2 Fields in the Search Results Field Set should be Set.
     */
    global List<PricebookEntry> suggest(SObject quote, Map<String, Object> inputValuesMap)
 {
        System.debug('METHOD CALLED: suggest');
        //GET ALL POSSIBLE FIELDS FROM THE SEARCH RESULTS FIELD SET
        List<Schema.FieldSetMember> searchResultFieldSetFields =
SObjectType.Product2.fieldSets.SBQQ__SearchResults.getFields();

        //BUILD THE SELECT STRING
        String selectClause = 'SELECT ';

        for (Schema.FieldSetMember field : searchResultFieldSetFields) {
            selectClause += 'Product2.' + field.getFieldPath() + ', ';
        }
        selectClause += 'Id, UnitPrice, Pricebook2Id, Product2Id, Product2.Id';

        //BUILD THE WHERE CLAUSE
        String whereClause = '';

        whereClause += 'Pricebook2Id = \'' + quote.get('SBQQ__Pricebook__c') + '\'';

        //BUILD THE QUERY
        String query = selectClause + ' FROM PricebookEntry WHERE ' + whereClause;

        //DO THE QUERY
        List<PricebookEntry> pbes = new List<PricebookEntry>();
        pbes = Database.query(query);

        return pbes;
```

```
    }
}
```

SBQQ.ProductSearchPlugin - Guided Selling Example Implementation

This is an example implementation of the `System.ProductSearchPlugin_GuidedSelling` interface.

```
global class ExampleProductSearchPlugin implements SBQQ.ProductSearchPlugin{

/**Constructor. Not required for implementation**/
global ExampleProductSearchPlugin(){
}

/**Guided Selling Methods**/
// if isSuggestCustom returns True, the plugin uses suggest(), otherwise it uses
getAdditionalSuggestFilters()
global Boolean isSuggestCustom(SObject quote, Map<String,Object> inputValuesMap){ return
true; }
global Boolean isInputHidden(SObject quote, String input){ return false; }
global String getInputDefaultValue(SObject quote, String input){ return NULL; }
global List<PriceBookEntry> suggest(SObject quote, Map<String,Object> fieldValuesMap){
return null; }
global String getAdditionalSuggestFilters(SObject quote, Map<String,Object> inputValuesMap){
 return null; }
```

Recommended Products Plugin

Use the Recommended Products plugin to recommend related products based on the existing products on a quote.

The Recommended Products plugin lets customers use their own product recommendation service in Salesforce CPQ, together with product search, search filters, favorites, and guided selling.

With the Recommended Products plugin, sales reps can see a list of recommended products while they are creating their quote. By making relevant products easier to find, the quoting process is more efficient, and sales reps can sell more products.

EDITIONS

Available in: Salesforce CPQ Winter '21 and later

Sample use cases of the Recommended Products plugin include:

- You can help sales reps close more deals by enabling them to create quotes with the ideal mix of products that anticipate their customers' needs.
- Sales reps can upsell more products by adding recommended products that are frequently sold with the products on their quotes.

Complete these steps to implement the Recommended Products plugin:

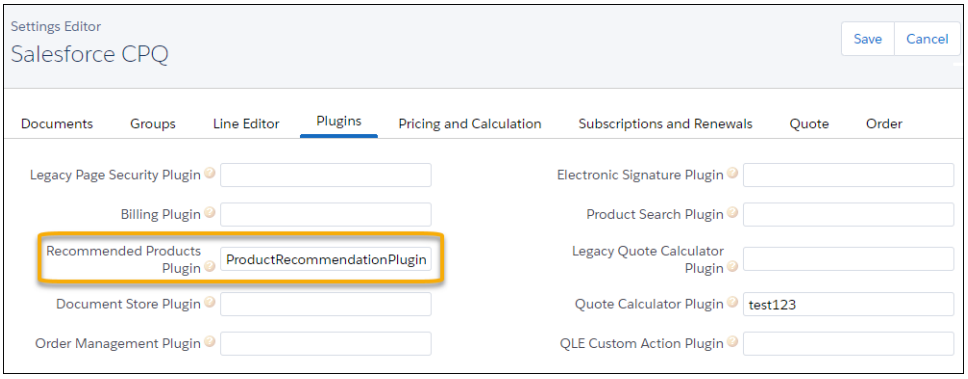
- On the **Plugins** tab in the **Settings Editor**, enable the Recommended Products Plugin.
- To add the **Add Recommendations** button to the quote line editor, activate the Add Recommendations custom action record. See Custom Actions.

- To fully enable the Recommended Products feature, you must provide your own implementation of the plugin interface. You can use your recommendation engine or a third-party service. Your plugin interface must implement the recommend() method in the global ProductRecommendationPlugin interface:

```
global interface ProductRecommendationPlugin {
    PricebookEntry[] recommend(SObject quote, List<SObject> quoteLines);
}
```

This method takes quote and quote lines as arguments, and it outputs an ordered list of recommended price book entries. If at runtime the plugin input is missing a field required for your implementation, add it to the `ReferencedFields` field set on the corresponding object.

Add your plugin's class name on the Plugins tab in **Settings**.

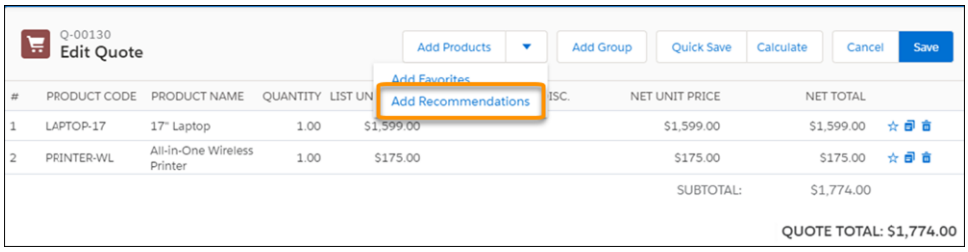


Note:

- A maximum of 2,000 price book entries are shown on the Recommended Products page, similar to the limit on the Product Lookup page. Your implementation class can return up to your top 2,000 recommendations. If you have more, you can use a recommendation score to sort and choose your top 2,000 products.
- The Recommended Products plugin doesn't support the Large Quote Threshold setting.
- Salesforce CPQ prioritizes field-level security over Recommended Product plugins. If your plugin includes fields that your users don't have permission to view, those fields aren't displayed on the Recommended Products page.
- The Product Recommendation page uses the same field set as the Product Lookup page.

Walkthrough

1. Add products to a quote and click the **Add Recommendations** custom action.



2. Salesforce calls your plugin implementation class and obtains an ordered list of PricebookEntry sObjects. These sObjects are then displayed on the Recommended Products page.

Methods

recommend()

Description

Returns the product recommendations.

Parameters

Param	Type	Description
quote	SBQQ__Quote__c	Current quote object
quoteLines	List<SBQQ__QuoteLine__c>	Current quote lines of the quote.

Return Values

PricebookEntry[ ] : Ordered list of PricebookEntry SObjects.

Error Scenarios

If your plugin throws an exception, the error shows on the Recommendations Lookup page. To show that the plugin implementation throws the error, the error message shows “Plugin Error:" as a prefix followed by the message from the exception.

Sample Implementation

When you implement the Recommended Products plugin, you can create your own recommendation engine or use a third-party recommendation service. Store the product recommendations in your org in a custom object. Then, query the recommendations in the custom object from in the recommend() method of your plugin.

In this example, the name of the custom object that stores recommendations is ProductRecommendation__c.

```
// Create your own Custom Object to store your product recommendations
ProductRecommendation__c {
    Id Product2Id__c,
    Id RecommendedProduct2Id__c,
}
```

```
global class ProductRecommendationPluginJH implements SBQQ.ProductRecommendationPlugin{
global PricebookEntry[] recommend(SObject quote, List<SObject> quoteLines) {
    System.debug('ProductRecommendationPluginJH');
    // Get the price book Id of the quote
```

```
    Id pricebookId = (Id)quote.get('SBQQ__PriceBookId__c');
    String quoteCurrency=(String)quote.get('CurrencyIsoCode');
    // Get Ids of all products in the quote
    Id[] productIdsInQuote = new Id[0];
    for (SObject quoteLine : quoteLines) {
        Id productId = (Id)quoteLine.get('SBQQ__Product__c');
        productIdsInQuote.add(productId);
    }

    // Query the recommendation custom object records of all products in quote.
   ProductRecommendation__c[] recommendations = [
        SELECT RecommendedProduct2Id__c
        FROM ProductRecommendation__c
        WHERE Product2Id__c IN :productIdsInQuote];

    // Get Ids of all recommended products
    Id[] recommendedProductIds = new Id[0];
    for(ProductRecommendation__c recommendation : recommendations) {
        recommendedProductIds.add(recommendation.RecommendedProduct2Id__c);
    }
    System.debug('>>>>>>>>>>>'+recommendedProductIds);
    // Query the price book entries of the above recommended products
    PricebookEntry[] priceBookEntries = [
       SELECT Id, UnitPrice, Pricebook2Id, Product2Id, Product2.Name, Product2.ProductCode

        FROM PricebookEntry
        WHERE Product2Id IN :recommendedProductIds AND Pricebook2Id = :pricebookId AND
CurrencyIsoCode=:quoteCurrency];

    return priceBookEntries;
}
}
```

External Configurator Plugins

Enable sales reps to create quotes that incorporate your product's unique attributes, bundle configuration, and other information. A CPQ external configurator replaces the CPQ product configurator for the specified products, while still allowing you to use other Salesforce CPQ features such as price calculations and product rules.

EDITIONS

Available in: Salesforce CPQ Winter '16 and later

You can develop your configurator in a Visualforce page or host it in an external web application. Salesforce sends the product information payload to your external configurator, where you can create and modify its attributes, configure bundles, and perform other tasks specific to your organization's needs. Then, you send the updated payload back to Salesforce CPQ so it can build a quote line for the product with the attributes you've configured. If the sales rep reconfigures the product, the payload is sent back to your custom configurator.

Set Up an External Configurator to Launch from a Custom Action

Create a custom action that launches a non-Salesforce CPQ configurator.

Create an External Configurator

Host your configurator in Salesforce using Visualforce pages, or in an external web application such as Heroku. Use the easyXDM library to transfer data between your configurator and Salesforce CPQ.

Configure Salesforce CPQ to Use the External Configurator

Configure the Salesforce CPQ package to launch your custom configurator from the quote line editor. Indicate which products are configured externally by setting the product's **Externally Configurable** field to `true`.

Set Up an External Configurator to Launch from a Custom Action

Create a custom action that launches a non-Salesforce CPQ configurator.

You may have to add the following layouts and values.

- Add the Page and URL Target fields to the custom action page layout.
- Add the Popup value to the custom action's URL Target field.
- Add a label that represents your external configurator's name to the custom action's Label field.

1. From your Custom Actions tab, click **New**.
2. From the Label field, choose GIS.
3. From the Page field, choose Product Configurator.
4. From the URL Target field, choose Popup.
5. In the URL field, add a URL for a custom website that uses a secure https protocol.
6. Save your changes.

Important: We strongly recommend that you choose Dialog Window for the value of URL Target. Replace Window causes users to lose all their work when the external configurator loads.

EDITIONS

Available in: Salesforce CPQ Winter '16 and later

Create an External Configurator

Host your configurator in Salesforce using Visualforce pages, or in an external web application such as Heroku. Use the easyXDM library to transfer data between your configurator and Salesforce CPQ.

1. The easiest way to include the easyDM library in your web application is to include the CDN library (hosted by CloudFlare).

```
<!-- easyXDM.min.js compiled and minified JavaScript to communicate with Salesforce
CPQ-->
<script type="text/javascript"
src="https://cdnjs.cloudflare.com/ajax/libs/easyXDM/2.4.20/easyXDM.min.js"
crossorigin="anonymous">
</script>
```

You can also download the library then upload it to your org as a static resource.

2. Initialize the easyXDM library. Use this library to send information between your configurator and Salesforce CPQ. This example uses the variable `configObj` to store the information sent from Salesforce CPQ.

```
// Initialize the EasyXDM connection to Salesforce CPQ
var rpc = new easyXDM.Rpc({}, {

      //method defined in Salesforce CPQ
      remote: {
          postMessage: {}
      },
```

EDITIONS

Available in: Salesforce CPQ Winter '18 and later

```
    // method for receiving configuration JSON from Salesforce CPQ
    local: {
      postMessage: function (message) { // parse the incoming information, for example:

            var configObj = JSON.parse(message);
      }
    }
  });
```

3. Perform the configuration, such as enforcing rules or adding new product attributes that will appear in the quote line editor. For example, set the value of a configuration option called "Special Code" to 12345. This example assumes that you've parsed the incoming JSON into the variable `configObj`.

```
configObj.product.optionConfigurations["Special Code"] = '12345';
```

4. Send the configuration information back to Salesforce CPQ as a JSON string.

```
rpc.postMessage(JSON.stringify(configObj));
```

Configure Product Bundles

Use the `optionConfigurations` parameter to create product bundles. You can nest bundles up to four levels deep, including the top-level product.

External Configurator Parameters

Salesforce CPQ passes configuration information to your custom configurator in JSON format. Modify the and return the information to Salesforce CPQ.

External Configurator Example

This example shows how to initialize the easyXML library and create **Send** and **Cancel** buttons. It then displays the configuration data sent from Salesforce CPQ.

Configure Product Bundles

Use the `optionConfigurations` parameter to create product bundles. You can nest bundles up to four levels deep, including the top-level product.

EDITIONS

Available in: Salesforce CPQ Winter '16 and later

Enable Nested Bundles

1. From Setup, enter `Installed Packages`, and then select Installed Packages.
2. Find the Salesforce CPQ package and click **Configure**.
3. Navigate to the **Additional Settings** tab and select **Nested Bundles for External Configurator.**.
4. Click **Save**.

Note: This setting can't be disabled.

Create Nested Bundles

Use the `optionConfigurations` parameter to define a nested product in a bundle.

Example: **Create a Work Anywhere Software Bundle**

For example, suppose that a sales rep quotes a Work Anywhere software product. The Work Anywhere product can include VPN access as a nested option. The VPN access can include the Ultra High-Speed option, and the Ultra HighSpeed option can include the Ad Blocker:

```
Work Anywhere
    VPN Access
        Ultra High Speed
           Ad Blocker
```

Use the following payload to configure the Work Anywhere bundle .

```
{
  "quote": {},
  "product": {
    "configuredProductId": "<Product2 Id>"  // ID of the WorkAnywhere Product,
    "lineItemId": null,
    "lineKey": null,
    "configurationAttributes": {
      "SBQQ__UnitPrice__c": null,
      "attributes": {
        "type": "SBQQ__ProductOption__c"
      }
    },
    "optionConfigurations": {
      "Other Options": [
        {
          "optionId": "<SBQQ__ProductOption__c Id>",
          "selected": true,
          "ProductName": "VPN Access",
          "Quantity": 1,
          "configurationData": {},
          "readOnly": {},
          "optionConfigurations": {
            "Other Options": [
              {
                "optionId": "<SBQQ__ProductOption__c Id>",
                "selected": true,
                "ProductName": "Ultra High Speed",
                "Quantity": 1,
                "configurationData": {},
                "readOnly": {},
                "optionConfigurations": {
                  "Other Options": [
                    {
                      "optionId": "<SBQQ__ProductOption__c Id>",
                      "selected": true,
                      "ProductName": "Ad Blocker",
                      "Quantity": 1,
                      "configurationData": {},
                      "readOnly": {}
                    }
                  ]
```

```
                }
              }
            ]
          }
        }
      ]
    },
    "configurationData": {}
  },
  "products": [],
  "readOnly": {},
  "redirect": {
    "save": true
  }
}
```

Considerations for Nested Bundles

Consider the following when configuring nested bundles with the external configurator.

- You can configure up to four levels of nested bundles, including the top-level product. If you reconfigure a bundle containing five levels, only four levels are sent to the external configurator. Deselect the fifth level by deselecting the top-level product.
- We don't support the auto property with bundles, so users can't return to the Salesforce CPQ configurator and continue configuring the product. Instead, users are redirected back to the quote line editor.
- Default configurations aren't supported. When returning the payload, you must explicitly select each nested option, even for reconfiguration payloads.
- Nested bundles are assumed configured. Nested bundles with the product's Configuration Type set to `Required` are considered configured if they're part of the payload.
- Min/Max options aren't supported with the external configurator. Min/Max options are directed to the Salesforce CPQ configurator. Attempting to set min/max options with the external configurator can result in errors.
- Duplicate dynamic options are supported. That is, you can add the same option to a bundle multiple times.

External Configurator Parameters

Salesforce CPQ passes configuration information to your custom configurator in JSON format. Modify the and return the information to Salesforce CPQ.

The following parameters are required unless otherwise indicated.

EDITIONS

Available in: Salesforce CPQ Winter '16 and later

quote
Object. The `SBQQ__Quote__c` record.

product
Object. The product being configured.

configuredProductId
String. Read-only. The ID of the Product2 record.

lineItemId
String. Read-only. The ID of the corresponding quote line. Populated on reconfigure when the quote line is saved.

lineKey
Number. Read-only. Salesforce CPQ uses this field to identify the corresponding quote line for this product.

configurationAttributes
Object. Required, but can be empty. If you use configuration attributes, this parameter contains the attribute field values.

optionConfigurations
Object. Indicates the options for a nested bundle.

optionId
String. Required for static options. The ID of the SBQQ__ProductOption__c record.

productId
String. Required for dynamic options. The ID of the SBQQ__Product2__c record. Available in API version 57.0 and later.

selected
Boolean. Required for static options but optional for dynamic options. `true` if the product option is selected; otherwise `false`.

ProductName
String. Read-only. Name of the product.

Quantity
Number. The line item's quantity.

configurationData
Object. Required, but can be empty. Use this parameter to set editable SBQQ__ProductOption__c fields, which can be used with rules or twin field mapping.

readOnly
Object. The quote line that corresponds to the selected option. Salesforce CPQ populates this field on reconfigure requests.

index
Number. Required when the SBQQ__ProductFeature__c record's Option Selection Method field is `Add`. When you add the same product to a feature multiple time, use this parameter to uniquely identify each instance of the same product.

optionConfigurations
Object. Available when Nested Bundles for External Configurator is enabled. Available in API version 56.0 and later. Use this object to include options for a nested bundle

configurationData
Object. Required, but can be empty. Field - value pair that sets twin field values on the quote line for the product being configured.

products
Array. Optional. Use this parameter to clone the product that is being configured.

readOnly
Object. The quote line that corresponds to the product being configured. Salesforce CPQ populates this field on reconfigure requests.

redirect
Object. Contains properties that specify the save and redirect behavior.

save
Boolean. To save the configuration, set this parameter to `true`. To cancel the configuration, set this value to `false`.

auto
Boolean. To redirect the user to the quote line editor, set this value to `true`. To redirect the user to the CPQ Configurator, set this value to `false`. This parameter isn't available when `Nested Bundles for External Configurator` is enabled.

External Configurator Example

This example shows how to initialize the easyXML library and create **Send** and **Cancel** buttons. It then displays the configuration data sent from Salesforce CPQ.

EDITIONS

Available in: Salesforce CPQ Winter '16 and later

```
<apex:page doctype="html-5.0" standardstylesheets="false" showHeader="false">
 <html>
    <head>
<!-- easyXDM.min.js compiled and minified JavaScript to communicate with Salesforce CPQ-->
<script type="text/javascript"
src="https://cdnjs.cloudflare.com/ajax/libs/easyXDM/2.4.20/easyXDM.min.js"
crossorigin="anonymous">
</script>
    </head>
    <body>
        <div>
            <button onclick="broadcastSend()">Send</button>
           <button onclick="broadcastCancel()">Cancel (Customer's Cancel Button)</button>

            <br></br>
          <textarea id="output" type="text" style="width:1400px; height:700px"></textarea>

        </div>
        <script type="text/javascript">
            // Set up the EasyXDM connection to Salesforce CPQ
            var rpc = new easyXDM.Rpc({},{
                remote: {
                    postMessage: {}
                },
                local: {
                    //Method that receives the configuration information.
                    postMessage: function(message) {
                        // Display the JSON received from Salesforce CPQ.
                        document.getElementById('output').value =
JSON.stringify(JSON.parse(message), null, 4);
                    }
                }
            });
            function broadcastSend() {
                //Return the configuration information to Salesforce CPQ
                rpc.postMessage(document.getElementById('output').value);
            }
            function broadcastCancel() {
                rpc.postMessage(null);
            }
        </script>
    </body>
</html>
</apex:page>
```

Configure Salesforce CPQ to Use the External Configurator

EDITIONS

Available in: Salesforce CPQ Winter '18 and later

Configure the Salesforce CPQ package to launch your custom configurator from the quote line editor. Indicate which products are configured externally by setting the product's **Externally Configurable** field to `true`.

1. From Setup, in the Quick Find box, *`Installed Packages`*, and then select **Installed Packages**.
2. Find the Salesforce CPQ package and click **Configure**.
3. Select the **Additional Settings** tab.
4. In the External Configurator URL field, enter the URL for your external configurator.

   To get the URL of a Visualforce page, click `preview`. You can use either an absolute or a relative URL. To use the custom configurator with Experience Cloud, you must use a relative URL, because the URL format of Salesforce Lightning and externally-hosted web applications are different.
5. Optional: On the Additional Settings page, you can also select the **Third Party Configurator** field. When active, Salesforce CPQ launches the external configurator so it takes up your entire screen. Any action that closes the external configurator, such as clicking cancel or save, redirects to the page that launched the external configurator.
6. Find the products to configure with the external configurator.
   a. Select the **Externally Configurable** field on each product record.
   b. Make sure each product's **Configuration Type** field is set to `Required`.

      The external configurator launches when a user clicks the wrench next to a configurable bundle, or when the user adds a product where a configuration event is required. If you set the configuration type to `Allowed`, the external configurator launches only when a sales rep selects the wrench icon next to a configurable bundle.

Legacy Quote Calculator Plugin

EDITIONS

Available in: All Salesforce CPQ Editions

Use Apex code to perform calculations within the CPQ quote line editor.

Important: As of Winter '17, Salesforce CPQ isn't developing new features for the Legacy Quote Calculator Plugin. Salesforce CPQ continues to support admin-related configuration cases for legacy calculator features. Salesforce Customer Support responds to bugs only for regressions from existing legacy calculator features. For current quote calculator plugin documentation, review Javascript Quote Calculator Plugin.

To use the Legacy Page Security Plugin, first create an Apex class. Then enter the Apex class name in the Legacy Page Security Plugin setting in the Salesforce CPQ package settings. You can call only one Apex class at a time in the Legacy Page Security Plugin.

Calculating True End Date and Subscription Term

The sample JavaScript script can be used in the Quote Line Calculator to calculate values and store maximum values for the custom quote line fields True Effective End Date and True Effective Term.

Custom Package Total Calculation

The sample Apex class calculates the total price for all components in a quote line and then stores that value in a custom field.

Find Lookup Records

The sample Apex class can be used in the Legacy Quote Line Calculator to query records within the plugin and use fields from those records to set each quote line's Description field.

Calculating True End Date and Subscription Term

The sample JavaScript script can be used in the Quote Line Calculator to calculate values and store maximum values for the custom quote line fields True Effective End Date and True Effective Term.

EDITIONS

Available in: Salesforce CPQ Winter '16 and later

Note: Salesforce CPQ no longer provides support for Legacy Quote Calculator plugins. Check out the Javascript Quote Calculator Plugin for support and improved features.

Note: The sample script assumes the Salesforce admin created custom fields `True Effective End Date` and `True Effective Term` on the Quote Line object.

Example:

```
global class QCPWinter16Legacy2 implements SBQQ.QuoteCalculatorPlugin,
SBQQ.QuoteCalculatorPlugin2 {

/* This QCP examples calculates and stores the effective end date on each quote line,
 as well as the effective term.
   It also stores the max(effective end date) and max(effective term) on the Quote
object
*/


    /* NOTE: the getReferencedFields method is no longer required if you use the
ReferencedFields field set on
 the Quote Line object.
             This field set must be created as it's not a managed one.
       NOTE: if you need to access Quote fields, you can create the ReferencedFields
field set on the Quote object as well.
       NOTE: if you do not use the getReferencedFields method, you can remove
SBQQ.QuoteCalculatorPlugin2 from the class declaration.
    */
    global Set<String> getReferencedFields() {
        return new Set<String>{
           /* Note: add fields using the following format - Only add fields referenced


              by the plugin and not in the Line Editor field set on the Quote Line
object
            String.valueOf(SBQQ__QuoteLine__c.My_Field_API_Name__c)
            */
            String.valueOf(SBQQ__QuoteLine__c.True_Effective_End_Date__c),
            String.valueOf(SBQQ__QuoteLine__c.True_Effective_Term__c),

            String.valueOf(SBQQ__Quote__c.True_Effective_End_Date__c),
            String.valueOf(SBQQ__Quote__c.True_Effective_Term__c),

            String.valueOf(SBQQ__QuoteLine__c.SBQQ__EffectiveStartDate__c),
            String.valueOf(SBQQ__QuoteLine__c.SBQQ__EffectiveEndDate__c),
            String.valueOf(SBQQ__QuoteLine__c.SBQQ__SubscriptionTerm__c),
            String.valueOf(SBQQ__QuoteLine__c.SBQQ__DefaultSubscriptionTerm__c),

            String.valueOf(SBQQ__Quote__c.SBQQ__SubscriptionTerm__c)
        };
    }
```

```
    global void onBeforePriceRules(SObject quote, SObject[] lines) {
    }

    global void onAfterPriceRules(SObject quote, SObject[] lines) {
    }

    global void onBeforeCalculate(SObject quote, SObject[] lines) {
    }

    global void onAfterCalculate(SObject quote, SObject[] lines) {
        Date maxEffectiveEndDate = null;
        Decimal maxEffectiveTerm = 0;
        for(SObject line : lines) {
            Date trueEndDate = calculateEndDate(quote, line);
            Decimal trueTerm = getEffectiveSubscriptionTerm(quote, line);
            if(maxEffectiveEndDate == null || maxEffectiveEndDate < trueEndDate) {
                maxEffectiveEndDate = trueEndDate;
            }
            if(maxEffectiveTerm < trueTerm) {
                maxEffectiveTerm = trueTerm;
            }
            line.put(SBQQ__QuoteLine__c.True_Effective_End_Date__c, trueEndDate);
            line.put(SBQQ__QuoteLine__c.True_Effective_Term__c, trueTerm);
        }
        quote.put(SBQQ__Quote__c.True_Effective_End_Date__c, maxEffectiveEndDate);
        quote.put(SBQQ__Quote__c.True_Effective_Term__c, maxEffectiveTerm);
    }

    global void onInit(SObject[] lines) {
    }

    private Date calculateEndDate(SObject quote, SObject line) {
        Date startDate =
(Date)line.get(String.valueOf(SBQQ__QuoteLine__c.SBQQ__EffectiveStartDate__c));
        Date endDate =
(Date)line.get(String.valueOf(SBQQ__QuoteLine__c.SBQQ__EffectiveEndDate__c));
        if ((startDate != null) && (endDate == null)) {
           /* Note: we are assuming that Subscription Term Unit is Month in the package
 settings */
            endDate = startDate.addMonths(getEffectiveSubscriptionTerm(quote,
line).intValue()).addDays(-1);
           /* Note: we are assuming that Subscription Term Unit is Day in the package
 settings */
//          endDate = startDate.addDays(getEffectiveSubscriptionTerm(line).intValue()
 - 1);
        }
        return endDate;
    }

    private Decimal getEffectiveSubscriptionTerm(SObject quote, SObject line) {
        Decimal lineTerm = null;
        Date startDate =
(Date)line.get(String.valueOf(SBQQ__QuoteLine__c.SBQQ__EffectiveStartDate__c));
```

```
        Date endDate =
(Date)line.get(String.valueOf(SBQQ__QuoteLine__c.SBQQ__EffectiveEndDate__c));
        if ((startDate != null) && (endDate != null)) {
           /* Note: we are assuming that Subscription Term Unit is Month in the package
 settings */
            lineTerm = startDate.monthsBetween(endDate.addDays(1));
           /* Note: we are assuming that Subscription Term Unit is Day in the package
 settings */
//            lineTerm = startDate.daysBetween(endDate.addDays(1));
        } else {
            lineTerm =
(Decimal)line.get(String.valueOf(SBQQ__QuoteLine__c.SBQQ__SubscriptionTerm__c));
            if (lineTerm == null) {
                lineterm =
(Decimal)quote.get(String.valueOf(SBQQ__Quote__c.SBQQ__SubscriptionTerm__c));
                if (lineTerm == null) {
                    return
(Decimal)line.get(String.valueOf(SBQQ__QuoteLine__c.SBQQ__DefaultSubscriptionTerm__c));

                }
            }
        }
        return lineTerm;
    }

}
```

Custom Package Total Calculation

The sample Apex class calculates the total price for all components in a quote line and then stores that value in a custom field.

EDITIONS

Available in: Salesforce CPQ Winter '16 and later

Note: Salesforce CPQ no longer provides support for Legacy Quote Calculator plugins. Check out the Javascript Quote Calculator Plugin for support and improved features.

Note: The sample script assumes the Salesforce admin created a custom field `Component Custom Total` on the Quote Line object.

Example:

```
global class QCPWinter16Legacy implements SBQQ.QuoteCalculatorPlugin,
SBQQ.QuoteCalculatorPlugin2 {

    /* NOTE: the getReferencedFields method is no longer required if you use the
ReferencedFields field set on
 the Quote Line object.
            This field set must be created as it's not a managed one.
       NOTE: if you need to access Quote fields, you can create the ReferencedFields
field set on the Quote object as well.
       NOTE: if you do not use the getReferencedFields method, you can remove
SBQQ.QuoteCalculatorPlugin2 from the class declaration.
    */
    global Set<String> getReferencedFields() {
```

```
        return new Set<String>{
           /* Note: add fields using the following format - Only add fields referenced


               by the plugin and not in the Line Editor field set on the Quote Line
object
            String.valueOf(SBQQ__QuoteLine__c.My_Field_API_Name__c)
            */
            String.valueOf(SBQQ__QuoteLine__c.Component_Custom_Total__c),

            String.valueOf(SBQQ__QuoteLine__c.SBQQ__ProratedListPrice__c),
            String.valueOf(SBQQ__QuoteLine__c.SBQQ__PriorQuantity__c),
            String.valueOf(SBQQ__QuoteLine__c.SBQQ__PricingMethod__c),
            String.valueOf(SBQQ__QuoteLine__c.SBQQ__DiscountScheduleType__c),
            String.valueOf(SBQQ__QuoteLine__c.SBQQ__Renewal__c),
            String.valueOf(SBQQ__QuoteLine__c.SBQQ__Existing__c),
            String.valueOf(SBQQ__QuoteLine__c.SBQQ__SubscriptionPricing__c)
        };
    }

    global void onBeforePriceRules(SObject quote, SObject[] lines) {
    }

    global void onAfterPriceRules(SObject quote, SObject[] lines) {
    }

    global void onBeforeCalculate(SObject quote, SObject[] lines) {
    }

    global void onAfterCalculate(SObject quote, SObject[] lines) {
        for(SObject line : lines) {
            SObject parent =
line.getSObject(SBQQ__QuoteLine__c.SBQQ__RequiredBy__c.getDescribe().getRelationshipName());

            if(parent != null) {
              Decimal pComponentCustomTotal =
(Decimal)parent.get(String.valueOf(SBQQ__QuoteLine__c.Component_Custom_Total__c));

              Decimal cListPrice =
(Decimal)line.get(String.valueOf(SBQQ__QuoteLine__c.SBQQ__ProratedListPrice__c));
              Decimal cQuantity =
(Decimal)line.get(String.valueOf(SBQQ__QuoteLine__c.SBQQ__Quantity__c));
              Decimal cPriorQuantity =
(Decimal)line.get(String.valueOf(SBQQ__QuoteLine__c.SBQQ__PriorQuantity__c));
              String cPricingMethod =
(String)line.get(String.valueOf(SBQQ__QuoteLine__c.SBQQ__PricingMethod__c));
              String cDiscountScheduleType =
(String)line.get(String.valueOf(SBQQ__QuoteLine__c.SBQQ__DiscountScheduleType__c));
              Boolean cRenewal =
(Boolean)line.get(String.valueOf(SBQQ__QuoteLine__c.SBQQ__Renewal__c));
              Boolean cExisting =
(Boolean)line.get(String.valueOf(SBQQ__QuoteLine__c.SBQQ__Existing__c));
              String cSubscriptionPricing =
(String)line.get(String.valueOf(SBQQ__QuoteLine__c.SBQQ__SubscriptionPricing__c));
```

```
                pComponentCustomTotal = (pComponentCustomTotal == null) ? 0 :
pComponentCustomTotal;

                cListPrice = (cListPrice == null) ? 0 : cListPrice;
                cQuantity = (cQuantity == null) ? 1 : cQuantity;
                cPriorQuantity = (cPriorQuantity == null) ? 0 : cPriorQuantity;
                cPricingMethod = (cPricingMethod == null) ? 'List' : cPricingMethod;
                cDiscountSCheduleType = (cDiscountSCheduleType == null) ? '' :
cDiscountSCheduleType;
                cRenewal = (cRenewal == null) ? false : cRenewal;
                cExisting = (cExisting == null) ? false : cExisting;
                cSubscriptionPricing = (cSubscriptionPricing == null) ? '' :
cSubscriptionPricing;

                Decimal cTotalPrice = getTotal(cListPrice, cQuantity, cPriorQuantity,
cPricingMethod, cDiscountScheduleType, cRenewal, cExisting, cSubscriptionPricing,
cListPrice);
                pComponentCustomTotal += cTotalPrice;

                parent.put(SBQQ__QuoteLine__c.Component_Custom_Total__c,
pComponentCustomTotal);

            }
        }
    }

    global void onInit(SObject[] lines) {
        for(SObject line : lines) {
            line.put(SBQQ__QuoteLine__c.Component_Custom_Total__c, 0);
        }
    }

    private Decimal getTotal(Decimal price, Decimal quantity, Decimal priorQuantity,
String pricingMethod, String discountScheduleType, Boolean renewal, Boolean existing,
 String subscriptionPricing, Decimal ListPrice) {
        price = (price == null) ? 0 : price;
        renewal = (renewal == null) ? false : renewal;
        existing = (existing == null) ? false : existing;

        if(renewal == true && existing == false && priorQuantity == null) {
            return 0;
        } else {
           return price * getEffectiveQuantity(quantity, priorQuantity, pricingMethod,
 discountScheduleType, renewal, existing, subscriptionPricing, listPrice);
        }

    }

    private Decimal getEffectiveQuantity(Decimal quantity, Decimal priorQuantity,
String pricingMethod, String discountScheduleType, Boolean renewal, Boolean existing,
 String subscriptionPricing, Decimal ListPrice) {
        Decimal result = 0;
        Decimal deltaQuantity = 0;
```

```
        quantity = (quantity == null) ? 0 : quantity;
        priorQuantity = (priorQuantity == null) ? 0 : priorQuantity;
        pricingMethod = (pricingMethod == null) ? '' : pricingMethod;
        discountScheduleType = (discountScheduleType == null) ? '' :
discountScheduleType;
       subscriptionPricing = (subscriptionPricing == null) ? '' : subscriptionPricing;

        renewal = (renewal == null) ? false : renewal;
        existing = (existing == null) ? false : existing;
        listPrice = (listPrice == null) ? 0 : listPrice;

        deltaQuantity = quantity - priorQuantity;

        if(pricingMethod == 'Block' && deltaQuantity == 0) {
            result = 0;
        } else {
            if(pricingMethod == 'Block') {
                result = 1;
            } else {
                if(discountScheduleType == 'Slab' && (deltaQuantity == 0 || (quantity
 == 0 && renewal == true))) {
                    result = 0;
                } else {
                    if(discountScheduleType == 'Slab') {
                        result = 1;
                    } else {
                     if(existing == true && subscriptionPricing == '' && deltaQuantity
 < 0) {
                            result = 0;
                        } else {
                            if(existing == true && subscriptionPricing == 'Percent Of
 Total' && listPrice != 0 && deltaQuantity >= 0) {
                                result = quantity;
                            } else {
                                if(existing == true) {
                                    result = deltaQuantity;
                                } else {
                                    result = quantity;
                                }
                            }
                        }
                    }
                }
            }
        }

        return result;
    }

}
```

Find Lookup Records

The sample Apex class can be used in the Legacy Quote Line Calculator to query records within the plugin and use fields from those records to set each quote line's Description field.

EDITIONS

Available in: Salesforce CPQ Winter '16 and later

Note: Salesforce CPQ no longer provides support for Legacy Quote Calculator plugins. Check out the Javascript Quote Calculator Plugin for support and improved features.

Example:

```
global class QCPForFindingLookupRecords implements
SBQQ.QuoteCalculatorPlugin, SBQQ.QuoteCalculatorPlugin2 {
 global set<String> getReferencedFields() {
 return new Set<String> {
 String.valueOf(SBQQ__QuoteLine__c.SBQQ__ProductCode__c),
 String.valueOf(SBQQ__QuoteLine__c.SBQQ__Description__c)
 };
 }

 global void onInit(SObject[] lines) {}

 global void onBeforeCalculate(SObject quote, SObject[] lines)
 {}

 global void onBeforePriceRules(SObject quote, SObject[] lines)
 {}

 global void onAfterPriceRules(SObject quote, SObject[] lines)
 {}

 global void onAfterCalculate(SObject quote, SObject[] lines)
 {
 if (!lines.isEmpty()) {
 String[] productCodes = new String[0];
 for (SObject line : lines) {
 String productCode =
(String)line.get(String.valueOf(SBQQ__QuoteLine__c.SBQQ__ProductCode__c));

 if (productCode != null && !productCode.isWhitespace()) {
 productCodes.add(productCode);
 }
 }
 SBQQ__LookupData__c[] ds = [SELECT Id, SBQQ__Category__c,
SBQQ__Value__c FROM SBQQ__LookupData__c WHERE SBQQ__Category__c
 IN :productCodes];
 if (!ds.isEmpty()) {
 Map<String,String> valuesByCategory = new
Map<String,String>();
 for (SBQQ__LookupData__c d : ds) {
 valuesByCategory.put(d.SBQQ__Category__c, d.SBQQ__Value__c);

 }
 for (SObject line : lines) {
 String productCode =
(String)line.get(String.valueOf(SBQQ__QuoteLine__c.SBQQ__ProductCode__c));
```

```
 if (productCode != null && !productCode.isWhitespace()) {
 line.put(String.valueOf(SBQQ__QuoteLine__c.SBQQ__Description__c),
valuesByCategory.get(productCode));
 }
 }
 }
 }
 }
}
```

Product Configuration Initializer for Guided Selling

The product configuration initializer uses a custom user-provided APEX page to select options and set field values based on the results of guided selling prompts. It works only for standard product option fields and not for configuration attributes or custom product option fields.

A product configuration initializer consists of a Visualforce controller and Visualforce page. To use the initializer on all of your org's quote processes, go to the Product Configuration Initializer field in Salesforce CPQ line editor package settings and enter `c__` followed by the Visualforce page's name. To use an initializer on a specific quote process, go to the quote process's Product Configuration Initializer field and enter `c__` followed by the Visualforce page's name. Product configuration initializers on quote process records override the package-level product configuration initializer.

Example: Sample Visualforce controller:

```
public with sharing class LF_ProductInitializerController {
    public Product2[] products {get; set;}
    public Boolean skip {get; set;}
    Map<String,SBQQ__ProductOption__c> optionsByCode = new
Map<String,SBQQ__ProductOption__c>();

    public LF_ProductInitializerController() {
        // Set "skip" to true to bypass the configuration page, or to false to on the
 config page after the initializer has completed
        skip = true;

        // Retrieve product (bundle)
        String pidsStr = ApexPages.currentPage().getParameters().get('pids');
        String[] pids = pidsStr.split(',');
        products = [SELECT Id, Family, (SELECT SBQQ__OptionalSKU__r.ProductCode,
SBQQ__Quantity__c, SBQQ__Selected__c FROM SBQQ__Options__r) FROM Product2 WHERE Id IN
 :pids];
        for (SBQQ__ProductOption__c opt : products[0].SBQQ__Options__r) {
            optionsByCode.put(opt.SBQQ__OptionalSKU__r.ProductCode, opt);
        }

        String  myInput1 = ApexPages.currentPage().getParameters().get('Process Input
```

```
1');
      Decimal myInput2 = toInteger(ApexPages.currentPage().getParameters().get('Process
Input 2'));

        // Perform any logic you want here

        // Then select options in the bundle, for example:
        if (myInput1 == 'ABC') {
            selectOption('MyProductOption1', myInput2);
        } else {
            selectOption('MyProductOption2', 1)
        }
    }

    private Decimal toInteger(String value) {
        return String.isBlank(value) ? 0 : Decimal.valueOf(value);
    }

    private void selectOption(String code, Decimal qty) {
        optionsByCode.get(code).SBQQ__Selected__c = (qty > 0);
        optionsByCode.get(code).SBQQ__Quantity__c = qty;
    }
}
```

Example: Sample Visualforce page:

```
<apex:page controller="LF_ProductInitializerController" contentType="text/xml"
showHeader="false" sidebar="false">
    <products skipConfiguration="{!skip}">
    <apex:repeat var="product" value="{!products}">
        <product id="{!product.Id}">
            <apex:repeat var="opt" value="{!product.SBQQ__Options__r}">
                <option id="{!opt.Id}"
                    selected="{!opt.SBQQ__Selected__c}"
                    quantity="{!ROUND(opt.SBQQ__Quantity__c, 0)}"/>
            </apex:repeat>
        </product>
    </apex:repeat>
    </products>
</apex:page>
```

Product Search Executor for Guided Selling

A Product Search Executor plugin filters the results of a guided selling prompt after a sales rep's input. It consists of a Visualforce controller and Visualforce page, which you can associate with a specific quote process within a guided selling configuration. It's useful if you want to add an extra level of guided selling product filtering beyond what sales reps can control.

To use an executor with a quote process, go to the quote process's Product Search Executor field and enter `c__` followed by the Visualforce page's name.

Example: Your organization sells laptop workstations. A guided selling prompt lets sales reps filter your product catalog by memory, screen size, and type of processor. You could also make a simple product search executor plugin that further filters their results by active products and products that aren't bundle components.

Here's a sample APEX controller for your plugin.

```
public with sharing class TWProductSearchController {
    public Product2[] products {get; set;}
    public String error {get; set;}
    public TWProductSearchController() {
       products = [SELECT Id FROM Product2 WHERE IsActive = true AND SBQQ__Component__c
 = false];
    }
}
```

And here's a sample Visualforce page.

```
<apex:page controller="TWProductSearchController" contentType="text/xml"
showHeader="false" sidebar="false">
    <products error="{!error}">
    <apex:repeat var="product" value="{!products}">
        <product id="{!product.Id}"/>
    </apex:repeat>
    </products>
</apex:page>
```

Document Store Plugin

Use a CPQ document store plugin to store your quote documents as custom objects or in third-party integrations.

EDITIONS

Available in: All Salesforce CPQ Editions

Table 5: storeDocument Method

Param	Type	Description
quote	SObject (SBQQ__Quote__c)	Quote record information from the Salesforce CPQ quote.
document	SObject (SBQQ__QuoteDocument__c)	The quote document record from Salesforce CPQ.
content	Blob	Represents the actual PDF or Word file contents.

Example: Here's a sample document store plugin.

```
public class TestDocumentStorePlugin implements SBQQ.DocumentStorePlugin {

   public void storeDocument(SObject quote, SObject document, Blob content) {
       // Custom document saving logic goes here.
```

```
    }

    // Reserved for future use
    public Boolean isQuoteDocumentSaved() {
        return true;
    }

    // Reserved for future use
    public SObject[] listDocuments(SObject quote) {
        return null;
    }

}
```

Custom Action Plugin

A custom action plugin lets you run code before or after custom actions in Salesforce CPQ. Currently, custom action plugins support only cloning actions.

A custom action plugin can call either the `onBeforeCloneLine` method or the `onAfterCloneLine` method so that you can evaluate and modify a quote line before or immediately after the cloning process. These methods accept the following parameters.

Table 6: Parameters for onBeforeCloneLine and onAfterCloneLine

Parameter	Type	Definition
quote	QuoteModel	A representation of the quote object.
clonedLines	Object	Properties: **clonedLines** Available with onAfterCloneLine. An array of new QuoteLineModels created from the clone action. When using onBeforeCloneLine, this property is undefined. **originalLines** Available with onBeforeCloneLine and onAfterCloneLine. An array of QuoteLineModels for the original quote lines that the user is cloning. You can use the cloneLines parameter to change fields on the old and new quote lines.
conn	Object	A jsforce connection.

To create a custom action plugin, create a custom script record and enter your code in the Code field. Then, go to Salesforce CPQ package settings an open the plugins tab. Enter the name of your custom script in the Custom Action Plugin field and save your changes.

Here's a basic template for the plugin, without any additional code. You can use onBeforeCloneLine or onAfterCloneLine as needed.

```
export function onBeforeCloneLine(quote, clonedLines) {
   return Promise.resolve();
}
```

Salesforce CPQ Electronic Signature Plugin

An electronic signature plugin lets developers add electronic signature functionality to their orgs. This is useful for organizations who wish to streamline processes involving signatures, such as finalizing purchases and contracts.

EDITIONS

Available in: All Salesforce CPQ Editions

Example:

```
global virtual interface ElectronicSignaturePlugin {
 void send(QuoteDocument__c[] documents);

 void updateStatus(QuoteDocument__c[] documents);

 void revoke(QuoteDocument__c[] documents);

 String getSendButtonLabel();
}


global interface ElectronicSignaturePlugin2 extends
ElectronicSignaturePlugin {
 Boolean isSendButtonEnabled();
}
```

INDEX