Parser Plug-in
should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described in this document remains at the sole discretion of Oracle.

This document in any form, software or printed matter, contains proprietary information that is the exclusive property of Oracle. Your access to and use of this confidential material is subject to the terms and conditions of your Oracle Master Agreement, Oracle License and Services Agreement, Oracle PartnerNetwork Agreement, Oracle distribution agreement, or other license agreement which has been executed by you and Oracle and with which you agree to comply. This document and information contained herein may not be disclosed, copied, reproduced, or distributed to anyone outside Oracle without prior written consent of Oracle. This document is not part of your license agreement nor can it be incorporated into any contractual agreement with Oracle or its subsidiaries or affiliates.

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Sample Code

Oracle may provide sample code in SuiteAnswers, the Help Center, User Guides, or elsewhere through help links. All such sample code is provided “as is” and “as available”, for use only with an authorized NetSuite Service account, and is made available as a SuiteCloud Technology subject to the SuiteCloud Terms of Service at www.netsuite.com/tos.

Oracle may modify or remove sample code at any time without notice.

No Excessive Use of the Service

As the Service is a multi-tenant service offering on shared databases, Customer may not use the Service in excess of limits or thresholds that Oracle considers commercially reasonable for the Service. If Oracle reasonably concludes that a Customer's use is excessive and/or will cause immediate or ongoing performance issues for one or more of Oracle's other customers, Oracle may slow down or throttle Customer's excess use until such time that Customer's use stays within reasonable limits. If Customer's particular usage pattern requires a higher limit or threshold, then the Customer should procure a subscription to the Service that accommodates a higher limit and/or threshold that more effectively aligns with the Customer's actual usage pattern.

Beta Features

Oracle may make available to Customer certain features that are labeled “beta” that are not yet generally available. To use such features, Customer acknowledges and agrees that such beta features are subject to the terms and conditions accepted by Customer upon activation of the feature, or in the absence of such terms, subject to the limitations for the feature described in the User Guide and as follows: The beta feature is a prototype or beta version only and is not error or bug free and Customer agrees that it will use the beta feature carefully and will not use it in any way which might result in any loss, corruption or unauthorized access of or to its or any third party's property or information. Customer must promptly report to Oracle any defects, errors or other problems in beta features to support@netsuite.com or other designated contact for the specific beta feature. Oracle cannot guarantee the continued availability of such beta features and may substantially modify or cease providing such beta features without entitling Customer to any refund, credit, or other compensation. Oracle makes no representations or warranties regarding functionality or use of beta features and Oracle shall have no liability for any lost data, incomplete data, re-run time, inaccurate input, work delay, lost profits or adverse effect on the performance of the Service resulting from the use of beta features. Oracle’s standard service levels, warranties and related commitments regarding the Service shall not apply to beta features and they may not be fully supported by Oracle's customer support. These limitations and exclusions shall apply until the date that Oracle at its sole option makes a beta feature generally available to its customers and partners as part of the Service without a “beta” label.
Send Us Your Feedback

We'd like to hear your feedback on this document.

Answering the following questions will help us improve our help content:

- Did you find the information you needed? If not, what was missing?
- Did you find any errors?
- Is the information clear?
- Are the examples correct?
- Do you need more examples?
- What did you like most about this document?

Click [here](#) to send us your comments. If possible, please provide a page number or section title to identify the content you're describing.

To report software issues, contact NetSuite Customer Support.
Table of Contents

Parser Plug-in Overview ................................................................. 1
- Developing a Parser Plug-in ......................................................... 2
  - Enabling Features for the Parser Plug-in .................................. 2
  - Creating the Parser Plug-in Script File ..................................... 2
  - Testing the Parser Plug-in ....................................................... 2
- Administering the Parser Plug-in ............................................... 5
  - Enabling Features for the Parser Plug-in .................................. 5
  - Adding the Plug-in Implementation ......................................... 6
  - Activating the Parser Plug-in .................................................. 6
Parser Plug-in Interface Definition .................................................. 8
- parseBankStatement ..................................................................... 8
  - BankStatementParserInput ...................................................... 10
  - BankStatementParserOutput .................................................. 12
- getStandardTransactionCodes ..................................................... 24
  - StandardTransactionCodeOutput ............................................ 25
Parser Plug-in Overview

Use the Parser Plug-in to create and upload your own parsers for different bank and credit card statement formats and store them in NetSuite. Parser Plug-ins translate the information from file formats that are not supported by NetSuite's default parsers. The Parser Plug-ins then turn the data into account statements or transactions that NetSuite can understand and use. This enables you to import other file formats and then use downstream matching and reconciliation features.

The Parser Plug-in can be used for both manual import and automatic import of bank data:

For manual import:

- NetSuite administrators associate a Parser Plug-in to a format profile for the appropriate financial institution record. For details, see the help topic Creating Format Profiles
- General financial users import a bank statement from the Statement File Import page, using the settings configured in the format profile. They can manually upload the file, or they can forgo a file upload by using the Bank Connectivity Plug-in. For details, see the help topic Manual Bank Data Import.

For automatic import, administrators must configure a format profile that uses the Parser Plug-in in conjunction with the Financial Institution Connectivity Plug-in. These two components enable them to directly connect to a financial institution and automate the retrieval of bank data. For details, see the help topics Automatic Bank Data Import and Creating Format Profiles. For information on the Financial Institution Connectivity Plug-in, see the help topic Financial Institution Connectivity Plug-in Interface Overview.

The following diagram shows the workflow for the Parser Plug-in:

The Parser Plug-in is supported by SuiteCloud Development Framework (SDF). SDF allows you to customize your NetSuite account from an integrated development environment (IDE) on your local computer. For details, see the help topic SuiteCloud Development Framework Overview.

<table>
<thead>
<tr>
<th>NetSuite Role</th>
<th>Topic Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Developer</td>
<td>Developing a Parser Plug-in</td>
</tr>
</tbody>
</table>
Developing a Parser Plug-in

To develop a Parser Plug-in, complete the following steps:

- Enabling Features for the Parser Plug-in
- Creating the Parser Plug-in Script File
- Testing the Parser Plug-in

Enabling Features for the Parser Plug-in

To begin development or administration for a Parser Plug-in, make sure that the Server SuiteScript feature is enabled on the development account.

To enable features for the plug-in:

1. Choose Setup > Company > Enable Features.
2. On the SuiteCloud subtab, make sure Server SuiteScript is checked. If necessary, check the box and agree to the Terms of Service.
3. Click Save.

Creating the Parser Plug-in Script File

You must implement each Parser Plug-in interface function in a Javascript file (with a .js extension). You can use the SuiteCloud IDE, another Javascript IDE, or a text editor to create the plug-in implementation script file.

This plug-in uses SuiteScript 2.0. Begin your implementation script file with this comment:

```*/
* @NApiVersion 2.0
* @NScriptType bankStatementParserPlugin
*/```
Developing a Parser Plug-in

The following is a parser sample:

```javascript
/**
 * @NApiVersion 2.0
 * @NScriptType bankStatementParserPlugin
 */
define(['N/file', 'N/log'],
function(file, log) {
    return {
        parseBankStatement: function (context) {
            var accountStatement = context.output.createNewAccountStatement();
            accountStatement.accountMappingKey = defaultAccountMappingKey;
            log.debug(
                title: 'Adding a new account statement',
                details: accountStatement
            );
            var accountStatementId = context.output.addAccountStatement({"parsedAccountStatement":accountStatement});
            log.debug(
                title: 'New account statement ID',
                details: accountStatementId
            );
            var statementFile = context.input.file;
            var statementLineIterator = statementFile.lines.iterator();
            statementLineIterator.next();
            statementLineIterator.each(function (line) {
                log.debug(
                    title: 'Read a line from the statement',
                    details: line.value
                );
            });
            var formatProfileId = context.input.formatProfileId;
            log.debug(
                title: 'Format Profile Id',
                details: formatProfileId
            );
            var partsOfCSVLine = line.value.split(',');
            var transaction = context.output.createNewTransaction();
            transaction.accountStatementId = accountStatementId;
            var rawDate = partsOfCSVLine[0];
            transaction.date = rawDate.substring(6, 10) + '-' + rawDate.substring(0, 2) + '-' + rawDate.substring(3, 5);
            transaction.amount = partsOfCSVLine[4];
            transaction.transactionMappingKey = partsOfCSVLine[3];
            transaction.transactionNumber = partsOfCSVLine[2];
            transaction.payee = partsOfCSVLine[1];
            transaction.currency = "USD";
            transaction.memo = partsOfCSVLine[5];
            transaction.customerRawId = partsOfCSVLine[6];
            transaction.customerName = partsOfCSVLine[7];
        }
    }
});
```

Parser Plug-in
Developing a Parser Plug-in

transaction.invoices = partsOfCSVLine[8].split(',');

log.debug(
    title: 'Adding a new transaction',
    details: transaction
);

context.output.addTransaction({"parsedTransaction":transaction});
return true;
});

getStandardTransactionCodes: function (context) {
    var tranTypes = ['ACH', 'CHECK', 'CREDIT', 'DEBIT', 'DEPOSIT', 'FEE', 'INTEREST', 'PAYMENT',
                    'TRANSFER', 'OTHER'];
    for (var i = 0; i < tranTypes.length; ++i)
    {
        var standardTransactionCode = context.output.createNewStandardTransactionCode();
        standardTransactionCode.tranCode = tranTypes[i];
        standardTransactionCode.tranType = tranTypes[i];

        log.debug(
            title: 'Adding a new standard transaction code',
            details: standardTransactionCode
        );
    }
};

To surface user-facing error messages that are readable, you must raise error messages as simple strings in the plug-in script. Currently, throwing error objects generates errors as JSON strings to users, which are not easily readable. For examples of raising an error message as a simple string, see the sample strings below:

throw('The file could not be imported because the currency code is not valid.');

var translateableErrorMessage = translation.get({collection: 'my_errors', key: 'CURRENCY_ERROR'}());
throw(translateableErrorMessage);

In the second example, translation.get is a SuiteScript API that can be implemented by plug-in authors if they wish to translate strings. For more information about translation, see the help topic N/translation Module.

Your plug-in script should contain all logic required for your implementation. The following table describes the functions that you must implement in the plug-in script file. For details about the required functions and their objects, see Parser Plug-in Interface Definition.

The following table describes the interface functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parseBankStatement</td>
<td>Parse a streamed bank statement.</td>
</tr>
</tbody>
</table>
Developing a Parser Plug-in

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BankStatementParserContext</td>
<td>This function can stream bank statement file content and obtain default settings for the file. This function adds the parsed transactions and account statements to NetSuite. BankStatementParserContext is the context for this function.</td>
</tr>
<tr>
<td>getStandardTransactionCodes</td>
<td>Store the standard transaction code mapping in NetSuite. This function contains information that maps the transaction code from the parsed transactions to the corresponding NetSuite transaction types. StandardTransactionCodeContext is the context for this function.</td>
</tr>
</tbody>
</table>

**Rules and Guidelines for Creating a Plug-in Implementation Script File**

Use the following rules and guidelines when creating the plug-in implementation script file:

- The plug-in script file can have any name, as long as it has a .js extension and contains an implementation of each of the interface functions.
- If you want to create utility files with helper functions to use with the plug-in implementation script file, you can include those files when you create the plug-in implementation in NetSuite.
- Due to limitations with how strings are handled in NetSuite, you cannot use the JavaScript `case` statement in the plug-in implementation script file.
- The plug-in allows up to 1000 usage units.
- There are special character restrictions set out by the import file specifications. For example, according to the OFX specification, the characters "<", ">", and "&" must be escaped using the entities "&lt;", "&gt;", and "&amp;", respectively. Otherwise, the parser will fail during the import process. If an uploaded file violates the file format specification for its file type, the parser will throw an error.
- If the character limit is exceeded in a field, the value will be truncated.

**Testing the Parser Plug-in**

To enable the plug-in for testing, see Adding the Plug-in Implementation.

**Administering the Parser Plug-in**

To install and set up the Parser Plug-in, complete the following steps:

- Enabling Features for the Parser Plug-in
- Adding the Plug-in Implementation
- Activating the Parser Plug-in

**Enabling Features for the Parser Plug-in**

See Enabling Features for the Parser Plug-in.
Adding the Plug-in Implementation

After creating the plug-in script file, upload and implement the file in NetSuite.

**To add the plug-in implementation:**

2. For the Script File, click the plus icon.
3. Click Choose File, then select the JavaScript file.
4. Click Save.
5. Click Create Plug-in Implementation.
6. On the Plug-in Implementation page, enter the following information:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>User-friendly name for the implementation. This name appears in the plug-in implementation list.</td>
</tr>
<tr>
<td>ID</td>
<td>Internal ID for the implementation for use in scripting. If you do not provide an ID, NetSuite will provide one for you after you click Save. As a developer, when you write code that uses this implementation, you will reference the implementation using this ID.</td>
</tr>
<tr>
<td>Status</td>
<td>Current status for the implementation. Choose Testing to have the implementation accessible to the owner of the implementation. Choose Released to have the implementation accessible in a production environment</td>
</tr>
<tr>
<td>Log Level</td>
<td>Logging level you want for the script. Select Debug, Audit, Error, or Emergency. The log information appears on the Execution Log subtab for the implementation after you create it. Go to Customization &gt; Plug-ins &gt; Plug-in Implementations, select your implementation, and then click the Execution Log subtab.</td>
</tr>
<tr>
<td>Execute As Role</td>
<td>Role that the script runs as. The Execute As Role field provides role-based granularity in terms of the permissions and restrictions of the executing script. The Current Role value indicates that the script executes with the permissions of the currently logged-in NetSuite user. If you created a custom role, be sure to test using that role. You can then bundle the custom role to distribute it with the plug-in implementation.</td>
</tr>
<tr>
<td>Description</td>
<td>Optional description of the implementation.</td>
</tr>
<tr>
<td>Owner</td>
<td>User account that owns the implementation. Default is the name of the current user</td>
</tr>
<tr>
<td>Inactive</td>
<td>Indicates that the plug-in implementation does not run in the account. Inactivate a plug-in implementation, for example, to temporarily disable it for testing purposes.</td>
</tr>
</tbody>
</table>

7. On the Unhandled Errors subtab, select which individual(s) will be notified if script errors occur. By default the Notify Script Owner box is checked.

   To enter multiple email addresses in the Notify Emails box, separate email addresses with a semicolon.

8. Click Save.

Activating the Parser Plug-in

After you install the plug-in bundle, you can activate the plug-in. The developer must give you the name of the plug-in implementation if it is different from the name of the SuiteApp.
To activate the plug-in:

1. To activate the plug-in implementation, go to Customization > Plug-ins > Manage Plug-ins.
2. Under Bank Statement Parser, check the box next to the name of your plug-in implementation.
3. Click Save.

You can access the list of implementations by going to Customization > Plug-ins > Plug-in Implementations.
Parser Plug-in Interface Definition

This chapter describes the functions, objects, and methods for the Parser Plug-in.

Plug-in Interface Functions

The plug-in interface includes the following functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parseBankStatement(BankStatementParserContext)</td>
<td>Parse a streamed bank statement. This function can stream bank statement file content and obtain default settings for the file. This function adds the parsed transactions and account statements to NetSuite. When NetSuite requests the bank statement content, the plug-in implements this function, which parses and sends the translated transaction data to NetSuite. BankStatementParserContext is the context for this function.</td>
</tr>
<tr>
<td>getStandardTransactionCodes(StandardTransactionCodeContext)</td>
<td>Store the standard transaction code mapping to NetSuite. This function contains information that maps the transaction code from the parsed transactions to the corresponding NetSuite transaction types. When NetSuite requests the standard transaction code mapping, the plug-in implements this function, which sends the list back to NetSuite. StandardTransactionCodeContext is the context for this function.</td>
</tr>
</tbody>
</table>

Context Objects

The Parser Plug-in interface functions use the following context objects.

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
</table>

parseBankStatement

**Function Declaration**

```c
void parseBankStatement(BankStatementParserContext bankStatementParserContext)
```

**Type**

Interface function

**Description**

Parse a streamed bank statement. This function can stream bank statement file content and obtain default settings for the file. This function adds the parsed transactions and account statements to NetSuite.
When NetSuite requests the bank statement content, the plug-in implements this function, which parses and sends the translated transaction data to NetSuite.

**Returns**
void

**Input Parameters**
bankStatementParserContext

**Example**

```javascript
parseBankStatement: function (context) {

    var accountStatement = context.output.createNewAccountStatement();
    accountStatement.accountMappingKey = defaultAccountMappingKey;
    log.debug(
        title: 'Adding a new account statement',
        details: accountStatement
    );

    var accountStatementId =
    context.output.addAccountStatement({'parsedAccountStatement':accountStatement});
    log.debug(
        title: 'New account statement ID',
        details: accountStatementId
    );

    var statementFile = context.input.file;

    var statementLineIterator = statementFile.lines.iterator();
    statementLineIterator.next();
    statementLineIterator.each(function (line) {
        log.debug(
            title: 'Read a line from the statement',
            details: line.value
        );

        var partsOfCSVLine = line.value.split(',');
        var transaction = context.output.createNewTransaction();
        transaction.accountStatementId = accountStatementId;
        var rawDate = partsOfCSVLine[0];
        transaction.date = rawDate.substring(6, 10) + '-' + rawDate.substring(0, 2) + '-' +
        rawDate.substring(3, 5);
        transaction.amount = partsOfCSVLine[4];
        transaction.transactionMappingKey = partsOfCSVLine[3];
        transaction.transactionNumber = partsOfCSVLine[2];
        transaction.payee = partsOfCSVLine[1];
        transaction.currency = "USD";
        transaction.memo = partsOfCSVLine[5];
        transaction.customerRawId = partsOfCSVLine[6];
        transaction.customerName = partsOfCSVLine[7];
        transaction.invoices = partsOfCSVLine[8].split(',’);

        log.debug(
            title: 'Adding a new transaction',
            details: transaction
        );
    });
```
context.output.addTransaction({'parsedTransaction':transaction});
return true;
});

BankStatementParserInput

<table>
<thead>
<tr>
<th>Type</th>
<th>Interface input object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Input object for <code>parseBankStatement</code>. When NetSuite requests the bank statement content, this object obtains the bank statement file content and default settings from the file.</td>
</tr>
</tbody>
</table>
| Methods               | ■ getFile
■ getFormatProfileId |
| Parent Object(s)      | N/A                    |
| Child Object(s)       | N/A                    |

**getFile**

**Function Declaration**

```javascript
nlobjFileInterface getFile()
```

**Type**

Object method

**Description**

This method gets the imported file content to the parser plug-in.

**Returns**

nlobjFileInterface - A file object containing the bank statement content. You can also use a file API to stream large files.

For more information on streaming large files, see the help topic N/file Module.

**Input Parameters**

None

**Parent object**

BankStatementParserInput

**Example**

```javascript
var statementFile = context.input.file;

var statementLineIterator = statementFile.lines.iterator();
statementLineIterator.next();
statementLineIterator.each(function (line) {
    log.debug({
        title: 'Read a line from the statement',
        details: line.value
    });
});
```

**getFormatProfileId**

**Function Declaration**

```javascript
int getFormatProfileId()
```
<table>
<thead>
<tr>
<th>Type</th>
<th>Object method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>This method passes the ID of the format profile, which the user selected at the time of import. To get related configurations to parse the bank or credit card statement, the script can use this ID combined with SuiteScript. These configurations could include date format, currency precision, and other user-defined configurations.</td>
</tr>
<tr>
<td>Returns</td>
<td>int – the ID of the format profile, which the user selected at the time of import.</td>
</tr>
<tr>
<td>Input Parameters</td>
<td>None</td>
</tr>
<tr>
<td>Parent object</td>
<td>BankStatementParserInput</td>
</tr>
</tbody>
</table>

**Example**

The following example is a generic implementation of the getFormatProfileId method:

```javascript
var formatProfileId = context.input.formatProfileId;
log.debug(
    title: 'Format Profile Id',
    details: formatProfileId
);
```

The following example uses the getFormatProfileId method to get parser configuration values from a format profile record:

```javascript
function getParserConfigs(context) {
    var ftid = context.input.formatProfileId;
    var ppid = 'customscript_bsp_parser_bai2';
    var configs = {};

    search.create({
        type : 'customrecord_bsp_configuration_value',
        filters : [
            ['custrecord_bsp_config_ftid', search.Operator.IS, ftid],
            'and',
            ['custrecord_bsp_config_ppid', search.Operator.IS, ppid]
        ],
        columns : [
            'name',
            'custrecord_bsp_config_value'
        ]
    }).run().each(function(result) {
        var name = result.getValue('name');
        var value = result.getValue('custrecord_bsp_config_value');

        configs[name] = {
            id : result.id,
            value : value
        };

        return true;
    });

    return configs;
}
```
In the preceding example, the custom record **customrecord_bsp_configuration_value** has a foreign key called **custrecord_bsp_config_ftid** that connects it to the format profile. A User Event script is deployed to display these records in the Configuration subtab on the Format Profile page, where users set their values.

The function **getParserConfigs** is triggered when the import process starts. When triggered, the function locates the configurations necessary for parsing the imported statement, according to the specified file format. In the preceding example, **customscript_bsp_parser_bai2** stands for BAI2.

### BankStatementParserOutput

<table>
<thead>
<tr>
<th>Type</th>
<th>Interface output object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Output object for <code>parseBankStatement</code>. When NetSuite requests the bank statement content, this object parses and sends the translated transaction data to NetSuite.</td>
</tr>
</tbody>
</table>
| Methods    | ■ `createNewAccountStatement`  
            ■ `addAccountStatement`  
            ■ `createNewTransaction`  
            ■ `addTransaction` |
| Parent Object(s) | N/A |
| Child Object(s) | ■ `ParsedAccountStatement`  
                 ■ `ParsedTransaction` |

### `createNewAccountStatement`

<table>
<thead>
<tr>
<th>Function Declaration</th>
<th>ParsedAccountStatement <code>createNewAccountStatement()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Object method</td>
</tr>
</tbody>
</table>
| Description          | Create a new ParsedAccountStatement object to enable the Parser Plug-in to set property values.  
                        This method does not persist a new account statement in NetSuite. |
| Returns              | ParsedAccountStatement                                |
| Input Parameters     | None                                                 |
| Parent object        | BankStatementParserOutput                             |

**Example**

```javascript
var accountStatement = context.output.createNewAccountStatement();
accountStatement.accountMappingKey = defaultAccountMappingKey;
log.debug({
    title: 'Adding a new account statement',
    details: accountStatement
});
```
**addAccountStatement**

**Function Declaration**

```java
int addAccountStatement(ParsedAccountStatement parsedAccountStatement)
```

**Type**
Object method

**Description**
The plug-in calls this method to persist an account statement record in NetSuite, which enables the plug-in to later associate transactions to the account.

**Returns**
`int` - the record ID of the account statement.

**Input Parameters**
`ParsedAccountStatement` - the object containing information to create the account statement record.

**Parent object**
`BankStatementParserOutput`

**Example**

```javascript
var accountStatementId =
context.output.addAccountStatement({'parsedAccountStatement':accountStatement});
log.debug(
    {title: 'New account statement ID',
     details: accountStatementId
    });
```

**createNewTransaction**

**Function Declaration**

```java
ParsedTransaction createNewTransaction()
```

**Type**
Object method

**Description**
Create a new object handler to enable the Parser Plug-in to set property values.
This method does not persist a new transaction in NetSuite.

**Returns**
`ParsedTransaction`

**Input Parameters**
None

**Parent object**
`BankStatementParserOutput`

**Example**

```javascript
var partsOfCSVLine = line.value.split(',');
var transaction = context.output.createNewTransaction();
transaction.accountStatementId = accountStatementId;
var rawDate = partsOfCSVLine[0];
transaction.date = rawDate.substring(6, 10) + '-' + rawDate.substring(0, 2) + '-' + rawDate.substring(3, 5);
transaction.amount = partsOfCSVLine[4];
transaction.transactionMappingKey = partsOfCSVLine[3];
transaction.transactionNumber = partsOfCSVLine[2];
transaction.payee = partsOfCSVLine[1];
transaction.currency = "USD";
transaction.memo = partsOfCSVLine[5];
```
transaction.customerRawId = partsOfCSVLine[6];
transaction.customerName = partsOfCSVLine[7];
transaction.invoices = partsOfCSVLine[8].split(',');

addTransaction

Function Declaration
void addTransaction(ParsedTransaction parsedTransaction)

Type
Object method

Description
The plug-in calls this method to persist the new transaction in NetSuite.

Returns
void

Input Parameters
ParsedTransaction – the new transaction to be persisted to NetSuite.

Parent object
BankStatementParserOutput

Example

context.output.addTransaction({"parsedTransaction":transaction});
return true;
)

ParsedAccountStatement

Type
Child Object

Description
Definition of a parsed account statement. createNewAccountStatement uses this object.

Methods
■ getAccountMappingKey
■ setAccountMappingKey
■ getStatementDate
■ setStatementDate
■ getOpeningBalance
■ setOpeningBalance
■ getClosingBalance
■ setClosingBalance

Parent Object(s)
BankStatementParserOutput

Child Object(s)
N/A

getAccountMappingKey

Function Declaration
string getAccountMappingKey()

Type
Object method

Description
This method gets the account mapping key, to associate the parsed account statements
with a selected NetSuite account.
**setAccountMappingKey**

**Function Declaration**

```csharp
void setAccountMappingKey(string accountMappingKey)
```

**Type**

Object method

**Description**

This method sets the account mapping key. The account mapping key must map to a valid NetSuite account ID. The mapping can be set up in the format profile or in the Online Banking Data Upload page.

This is a required field.

**Returns**

void

**Input Parameters**

accountMappingKey – The value is used to map the account statements to a NetSuite account.

**Parent object**

ParsedAccountStatement

---

**getStatementDate**

**Function Declaration**

```csharp
string getStatementDate()
```

**Type**

Object method

**Description**

This method gets the statement date.

**Returns**

string

**Input Parameters**

None

**Parent object**

ParsedAccountStatement

---

**setStatementDate**

**Function Declaration**

```csharp
void setStatementDate(string statementDate)
```

**Type**

Object method

**Description**

This method sets the account statement date. The date must be in the format YYYY-MM-DD. For example, "2018-05-28".

**Returns**

void

**Input Parameters**

statementDate – the statement date

**Parent object**

ParsedAccountStatement

---

**getOpeningBalance**

**Function Declaration**

```csharp
string getOpeningBalance()
```
Type | Object method  
---|---  
**Description** | This method gets the opening balance for the statement.  
**Returns** | string  
**Input Parameters** | None  
**Parent object** | ParsedAccountStatement  

**setOpeningBalance**

**Function Declaration** | void setOpeningBalance(string openingBalance)  
**Type** | Object method  
**Description** | This method sets the opening balance for the statement.  
**Returns** | void  
**Input Parameters** | openingBalance  
**Parent object** | ParsedAccountStatement  

**getClosingBalance**

**Function Declaration** | string getClosingBalance()  
**Type** | Object method  
**Description** | This method gets the closing balance for the statement.  
**Returns** | string  
**Input Parameters** | None  
**Parent object** | ParsedAccountStatement  

**setClosingBalance**

**Function Declaration** | void setClosingBalance(string closingBalance)  
**Type** | Object method  
**Description** | This method sets the closing balance for the statement.  
**Returns** | void  
**Input Parameters** | closingBalance  
**Parent object** | ParsedAccountStatement  

**ParsedTransaction**

**Type** | Child Object  
**Description** | Definition of a parsed transaction.
createNewTransaction uses this object.

Methods

- getAccountStatementId
- setAccountStatementId
- getDate
- setDate
- getAmount
- setAmount
- getTransactionMappingKey
- setTransactionMappingKey
- getTransactionNumber
- setTransactionNumber
- getPayee
- setPayee
- getCurrency
- setCurrency
- getMemo
- setMemo
- getFinancialInstitutionId
- setFinancialInstitutionId
- getTransactionStatus
- setTransactionStatus
- getCustomerRawId
- setCustomerRawId
- getCustomerName
- setCustomerName
- getInvoices
- setInvoices

Parent Object(s)  BankStatementParserOutput
Child Object(s)  N/A

getAccountStatementId

Function Declaration  int getAccountStatementId()
Type  Object method
Description  This method gets the account statement ID.
Returns  int
Input Parameters  None
Parent object  ParsedTransaction

setAccountStatementId

Function Declaration  void setAccountStatementId(integer accountStatementId)
### parseBankStatement

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Returns</th>
<th>Input Parameters</th>
<th>Parent object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object method</td>
<td>This method sets the account statement ID, to which the transaction is associated. This record ID must belong to a valid account statement record, which NetSuite creates during the bank import process.</td>
<td>void</td>
<td>accountStatementId – the record ID of the account statement.</td>
<td>ParsedTransaction</td>
</tr>
</tbody>
</table>

### getDate

**Function Declaration**

```
string getDate()
```

**Type**

Object method

**Description**

This method gets the date for the transaction.

**Returns**

string

**Input Parameters**

None

**Parent object**

ParsedTransaction

### setDate

**Function Declaration**

```
void setDate(string date)
```

**Type**

Object method

**Description**

This method sets the date for the transaction. The date must be in the format YYYY-MM-DD. For example, "2018-05-28".

This is a required field.

**Returns**

void

**Input Parameters**

date – the date for the transaction.

**Parent object**

ParsedTransaction

### getAmount

**Function Declaration**

```
string getAmount()
```

**Type**

Object method

**Description**

This method gets the amount for the transaction.

In bank and credit card statements, amounts must be provided in decimal formats only, with no commas. For example, "1 000 000.00".

Negative numbers must be represented with a minus (-) sign. For example, "-100".

**Returns**

string
setAmount

Function Declaration
void setAmount(string amount)

Type
Object method

Description
This method sets the amount for the transaction.
This is a required field.

Returns
void

Input Parameters
amount – the amount for the transaction.

Parent object
ParsedTransaction

getTransactionMappingKey

Function Declaration
string getTransactionMappingKey()

Type
Object method

Description
This method gets the transaction mapping key.

Returns
string

Input Parameters
None

Parent object
ParsedTransaction

setTransactionMappingKey

Function Declaration
void setTransactionMappingKey(string transactionMappingKey)

Type
Object method

Description
This method sets the transaction mapping key.
This transaction mapping key must map to a valid NetSuite transaction type. The mapping can be set up in the format profile or in getStandardTransactionCodes.
This is a required field.

Returns
void

Input Parameters
transactionMappingKey – the mapping key.

Parent object
ParsedTransaction

getTransactionNumber

Function Declaration
string getTransactionNumber()
<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Returns</th>
<th>Input Parameters</th>
<th>Parent object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object method</td>
<td>This method gets the transaction number.</td>
<td>string</td>
<td>None</td>
<td>ParsedTransaction</td>
</tr>
</tbody>
</table>

**setTransactionNumber**

*Function Declaration*: void setTransactionNumber(string transactionNumber)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Returns</th>
<th>Input Parameters</th>
<th>Parent object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object method</td>
<td>This method sets the transaction number.</td>
<td>void</td>
<td>transactionNumber</td>
<td>ParsedTransaction</td>
</tr>
</tbody>
</table>

**getPayee**

*Function Declaration*: string getPayee()

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Returns</th>
<th>Input Parameters</th>
<th>Parent object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object method</td>
<td>This method gets the payee for the transaction.</td>
<td>string</td>
<td>None</td>
<td>ParsedTransaction</td>
</tr>
</tbody>
</table>

**setPayee**

*Function Declaration*: void setPayee(string payee)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Returns</th>
<th>Input Parameters</th>
<th>Parent object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object method</td>
<td>This method sets the payee for the transaction.</td>
<td>void</td>
<td>payee</td>
<td>ParsedTransaction</td>
</tr>
</tbody>
</table>

**getCurrency**

*Function Declaration*: string getCurrency()
Type | Object method
---|---
Description | This method gets the currency code for the transaction.
Returns | string
Input Parameters | None
Parent object | ParsedTransaction

**setCurrency**

**Function Declaration** | void setCurrency(string currency)
**Type** | Object method
**Description** | This method sets the currency code for the transaction.
The currency must be one of the standard currency codes.
**Returns** | void
**Input Parameters** | currency – the currency code.
**Parent object** | ParsedTransaction

**getMemo**

**Function Declaration** | string getMemo()
**Type** | Object method
**Description** | This method gets the memo for the transaction, if applicable.
**Returns** | string
**Input Parameters** | None
**Parent object** | ParsedTransaction

**setMemo**

**Function Declaration** | void setMemo(string memo)
**Type** | Object method
**Description** | This method sets the memo for the transaction, if applicable.
**Returns** | void
**Input Parameters** | memo
**Parent object** | ParsedTransaction

**getFinancialInstitutionId**

**Function Declaration** | string getFinancialInstitutionId()
setFinancialInstitutionId

Function Declaration  void setFinancialInstitutionId(string financialInstitutionId)

Type  Object method
Description  This method sets the Financial Institution ID for the transaction.
Returns  void
Input Parameters  financialInstitutionId
Parent object  ParsedTransaction

getTransactionStatus

Function Declaration  string getTransactionStatus()

Type  Object method
Description  This method gets the transaction status.
Returns  string
Input Parameters  None
Parent object  ParsedTransaction

setTransactionStatus

Function Declaration  void setTransactionStatus(string transactionStatus)

Type  Object method
Description  This method sets the transaction status.
Returns  void
Input Parameters  transactionStatus
Parent object  ParsedTransaction

getCustomerRawId

Function Declaration  string getCustomerRawId()
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Returns</th>
<th>Input Parameters</th>
<th>Parent object</th>
</tr>
</thead>
<tbody>
<tr>
<td>setCustomerRawId</td>
<td>This method sets the customer ID for the transaction.</td>
<td>void</td>
<td>customerRawId</td>
<td>ParsedTransaction</td>
</tr>
<tr>
<td>getCustomerName</td>
<td>This method gets the customer name for the transaction.</td>
<td>string</td>
<td>None</td>
<td>ParsedTransaction</td>
</tr>
<tr>
<td>setCustomerName</td>
<td>This method sets the customer name for the transaction.</td>
<td>void</td>
<td>customerName</td>
<td>ParsedTransaction</td>
</tr>
<tr>
<td>getInvoices</td>
<td></td>
<td>string[]</td>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>
setInvoices

Function Declaration: void setInvoices(string[] invoices)

Type: Object method

Description: This method sets the invoices associated with the transaction.

Returns: void

Input Parameters: invoices

Parent object: ParsedTransaction

getStandardTransactionCodes


Type: Interface function

Description: Store the standard transaction code mapping to NetSuite. This function contains information that maps the transaction code from the parsed transactions to the corresponding NetSuite transaction types.

Returns: void

Input Parameters: StandardTransactionCodeContext

Example

```javascript
getStandardTransactionCodes: function (context) {
    var tranTypes = ['ACH', 'CHECK', 'CREDIT', 'DEBIT', 'DEPOSIT', 'FEE', 'INTEREST', 'PAYMENT', 'TRANSFER', 'OTHER'];
    for (var i = 0; i < tranTypes.length; ++i) {
        var standardTransactionCode = context.output.createNewStandardTransactionCode();
        standardTransactionCode.tranCode = tranTypes[i];
        standardTransactionCode.tranType = tranTypes[i];

        log.debug({
            title: 'Adding a new standard transaction code',
            details: standardTransactionCode
        });
    }
}
```
StandardTransactionCodeOutput

**Type**
Interface output object

**Description**
Output object for `getStandardTransactionCodes`
When NetSuite requests the standard transaction codes, this object sends the list of
standard transaction code mappings to NetSuite.

**Methods**
- `createNewStandardTransactionCode`
- `addStandardTransactionCode`

**Parent Object(s)**
N/A

**Child Object(s)**
StandardTransactionCode

---

**createNewStandardTransactionCode**

**Function Declaration**

**Type**
Object method

**Description**
The plug-in calls this method to create a new `StandardTransactionCode` object to enable
the Parser Plug-in to set property values.

This method does not persist a new account statement in NetSuite.

**Returns**
`StandardTransactionCode`

**Input Parameters**
None

**Parent object**
StandardTransactionCodeOutput

**Example**
```javascript
var standardTransactionCode = context.output.createNewStandardTransactionCode();
standardTransactionCode.tranCode = tranTypes[i];
standardTransactionCode.tranType = tranTypes[i];
```

---

**addStandardTransactionCode**

**Function Declaration**

**Type**
Object method

**Description**
The plug-in calls this method to pass a standard transaction code mapping to NetSuite, which enables the plug-in to match the transaction code of the parsed transaction to the correct NetSuite transaction type, and correct Credit or Debit type.

**Returns**
void

**Input Parameters**
- `standardTransactionCode` – the object containing the mapping of standard bank transaction code to NetSuite transaction type.

**Parent object**
StandardTransactionCodeOutput
getStandardTransactionCodes

Example

```java
```

**StandardTransactionCode**

<table>
<thead>
<tr>
<th>Type</th>
<th>Child Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Definition of a standard transaction code. <code>createNewStandardTransactionCode</code> uses this object.</td>
</tr>
</tbody>
</table>

**Methods**

- `getTranCode`
- `setTranCode`
- `getTranType`
- `setTranType`
- `getCreditDebitType`
- `setCreditDebitType`
- `getDescription`
- `setDescription`

**Parent object**

- `StandardTransactionCodeOutput`

**Child Object(s)**

- N/A

**getTranCode**

<table>
<thead>
<tr>
<th>Function Declaration</th>
<th>string getTranCode()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Object method</td>
</tr>
<tr>
<td>Description</td>
<td>This method gets the raw transaction code from the parsed transaction.</td>
</tr>
<tr>
<td>Returns</td>
<td>string</td>
</tr>
<tr>
<td>Input Parameters</td>
<td>None</td>
</tr>
<tr>
<td>Parent object</td>
<td><code>StandardTransactionCode</code></td>
</tr>
</tbody>
</table>

**setTranCode**

<table>
<thead>
<tr>
<th>Function Declaration</th>
<th>void setTranCode(string tranCode)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Object method</td>
</tr>
<tr>
<td>Description</td>
<td>This method sets the raw transaction code for the transaction code mapping. This is a required field.</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
<tr>
<td>Input Parameters</td>
<td>tranCode – raw transaction code.</td>
</tr>
<tr>
<td>Parent object</td>
<td><code>StandardTransactionCode</code></td>
</tr>
</tbody>
</table>
### getTranType

**Function Declaration**

```
string getTranType()
```

**Type**

Object method

**Description**

This method gets the transaction type from the parsed transaction.

**Returns**

string

**Input Parameters**

None

**Parent object**

StandardTransactionCode

### setTranType

**Function Declaration**

```
void setTranType(string tranType)
```

**Type**

Object method

**Description**

This method sets the NetSuite transaction type, to which the transaction code maps. This is a required field.

**Returns**

void

**Input Parameters**

tranType – Transaction type.

**Parent object**

StandardTransactionCode

### getCreditDebitType

**Function Declaration**

```
string getCreditDebitType()
```

**Type**

Object method

**Description**

This method gets the type of either Credit or Debit from the parsed transaction.

**Returns**

string

**Input Parameters**

None

**Parent object**

StandardTransactionCode

### setCreditDebitType

**Function Declaration**

```
string getCreditDebitType()
```

**Type**

Object method

**Description**

This method sets the type to Credit or Debit.

BAI2 and other similar file formats use this method to determine if the amount should be positive or negative.

**Returns**

string

**Input Parameters**

creditDebitType – the value is either Credit or Debit.
<table>
<thead>
<tr>
<th>Function Declaration</th>
<th>string getDescription()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Object method</td>
</tr>
<tr>
<td>Description</td>
<td>This method gets the description from the parsed transaction.</td>
</tr>
<tr>
<td>Returns</td>
<td>string</td>
</tr>
<tr>
<td>Input Parameters</td>
<td>None</td>
</tr>
<tr>
<td>Parent object</td>
<td>StandardTransactionCode</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function Declaration</th>
<th>void setDescription(string description)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Object method</td>
</tr>
<tr>
<td>Description</td>
<td>This method sets the description from the parsed transaction.</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
<tr>
<td>Input Parameters</td>
<td>description</td>
</tr>
<tr>
<td>Parent object</td>
<td>StandardTransactionCode</td>
</tr>
</tbody>
</table>