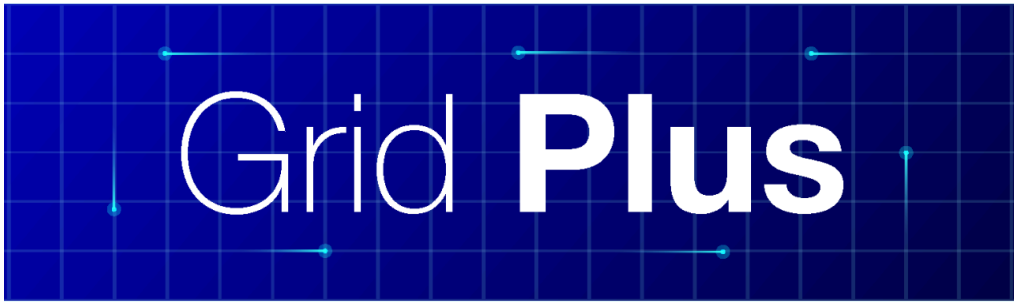


Editable Grid Component Plug-In

Configuration Documentation

Alex King
Technology Strategy Engineer

Version 3.1.3

appian

Table of Contents

Overview

This is an editable grid component that allows Appian users to easily display up to 5,000 rows of editable record data in a single component, without needing pagination or experiencing lag. End users can quickly manipulate single items of data or large batches by way of multicell selection. End users can also add to their record by adding data into empty rows either in the middle or at the bottom of the grid.

Features

1. **Flexible presentation of record data**
    a. Resize columns
    b. Drag & drop columns
    c. Display row headers
    d. Add borders around selected cells
    e. Hide or show columns
    f. Freeze or unfreeze columns
    g. Define column widths for each column
    h. Display links to record views
2. **Export record data from grid**: download to local storage a CSV file of your grid's current data view
3. **Sorting and filtering**: rapid, server-side, multi-column
4. **Data formatting**: currency, percentages, date, dropdowns, checkboxes, etc.
5. **Edit records**: alter individual or many cell values to update existing records
6. **Create records**: create new records of a designer-specified record type
7. **Delete records**: remove rows of data to delete records from the data fabric
8. **Excel-like behavior**
    a. Range cell selection with mouse select and drag
    b. Copy, cut, and paste
    c. Keyboard manipulation of and navigation between data
    d. Intuitive context menus with complex functionality (insert new row, cut, copy, add border)
9. **User based security:** restrict editing by user and group with SAIL expressions
10. **Column level data types** (Auto-Wrapping Text, Numeric, Date, Dropdown, Checkbox)
    a. Specialized filtering, editing, and validation ability by type
11. **Internationalization**: Automatically applies language updates based on the locale configured in Appian's internationalization settings (Supported locales include: arAR, zhCN, nlNL, enUS, frFR, deDE, itIT, jaJP, koKR, plPL, ptBR, ruRU, and esMX)

Component Configuration - Overview

Note:
In order to display record data, a column configuration object must be defined for each field to display. More information can be found in the following sections on how to define a column configuration object within the **columnConfigs** parameter.

If **columnConfigs** is defined and not **rows**, an empty grid will appear using the defined columns and primaryKeyField as a schema with which to create Data Fabric backed records.

To create, delete, or update the records displayed, the following parameters must be defined:

- **primaryKeyFieldName**
- **changeDataValue** and **changeDataSaveInto** (can be set to the same Appian object)
- **deleteDataValue** and **deleteDataSaveInto** (can be set to the same Appian object)

To query and display related record fields, the relationship between the primary and related record types must be defined in the column configuration object of each related record field. See **Component Configuration with Related Record** for more information. When displaying related records, ensure that the first item in your data does not have null values in the related record fields to display.

Component Configuration - Parameters

height (optional) - Determines the component and layout's height. Valid values: "SHORT", "MEDIUM", "TALL", "AUTO" (default).

rowsValue/rowsSaveInto (see above note) - Array of data items that have the following format:

```
Unset
{
        { fieldName: value, fieldName2: value, ...},
        { fieldName: value, fieldName2: value, ...}
}
```

This format is automatically followed when using a!queryRecordType() to display data.

primaryKeyFieldName (see above note) - String value of the primary key of the record to display and modify. This can be done using Appian's tostring() function or simply writing the name as text.

Using the tostring method:

```
Unset
primaryKeyFieldName: tostring(recordType!<recordTypeName>.<primaryKeyFieldName>),
```

Using a text value:

```
Unset
primaryKeyFieldName: "primaryKeyFieldName"
```

columnConfigs (required) - Array of objects representing each field of your data. Each object is constructed from a function plug-in matching the field's data type. The colConfig options include the following types:

- textColConfig
- numericColConfig
- dropdownColConfig
- dateColConfig
- checkboxColConfig

The parameters of each type vary slightly. See Column Configuration Functions for further configuration options. Each type has a parameter for a custom validator. See Column Validators.

changeDataSaveInto/changeDataValue (see above note) - Appian variable to track changes made to the grid data. When writing changes to records, you will be looping over the contents of this variable.

- The same object should be used for both parameters.
- This value will track only the values of the primaryKey field and the modified field for each record changed.

Using a local variable, local!changes:

```
Unset
a!localVariable{
  local!changes: {},
  editableGridComponentField(
    ...
    changeDataSaveInto: local!changes,
    changeDataValue: local!changes,
    ...
  )
}
```

deleteDataSaveInto/deleteDataValue (see above note) - Appian variable to track deletions made to the grid data. When writing deletions to records, you will be looping over the contents of this variable.

- The same object should be used for both parameters.
- This variable will track the IDs of each deleted record, relying on the 'primaryKeyFieldName' parameter to retrieve the primary key identifier.

Using a local variable, local!changes:

```
Unset
a!localVariable{
  local!deletions: {},
  editableGridComponentField(
    ...
    deleteDataSaveInto: local!deletions,
    deleteDataValue: local!deletions,
    ...
  )
}
```

readOnly (optional) - Boolean value to set the grid as readOnly (True) or editable (False). Default to false.

Can be defined with a SAIL expression to define permission groups. For example:

```
Unset
readOnly: a!isUserMemberOfGroup(
                username: touser(loggedInUser()),
                groups: { 51, 49 }
            ),
```

rowHeaders (optional) - Boolean value determining whether to display row headers or not. Default to true.

exportConfig (optional) - Object to define how data should be exported using the exportconfig() function. If a value is set, an export data button will appear above the grid. If set to null or not defined, no button will appear and the grid's behavior will be unchanged.

hiddenFields (optional) - List of fields to hide on grid render. This value can be set by providing the field names as text values or by wrapping the field reference in the tostring() function

showPrimaryKey (optional) - Boolean value determining whether to display the record's primary key or not. Default to false.

style (optional) - List to define accent and validation colors in the grid. The list accepts the following key-value pairs:

- theme: "DARK"/"LIGHT"/null
- accentColor: "AUTO"/<any valid HEX>/null
- validationColor: "AUTO"/<any valid HEX>/null

If the theme is left unset or set to null, the default theme of "LIGHT" will be used.

If accentColor is set to "AUTO" or null, the accent color used by the site containing the grid will be used. This value will determine the color of any links, checkboxes, or date pickers in the grid.

If validationColor is set to "AUTO" or null, a generic validation color of #ffbeba will be used. This value will determine the background color of cells that have been validated and found their conditions to be false.

Component Configuration with Related Records

When displaying record data in this grid, the fields queried by record relationships must be specifically configured. There are also several **important considerations** when working with related record data:

1. As the first row of your data shapes the component's understanding of the overall data, if your first queried record has a null value for a relationship, the following records will not display related record fields.
2. Each column configuration object **must** specify the relationshipName connecting the related record field to the primary queried record type. If this is not set, no value will be shown in the column.
3. Related fields in the current version of this component can only be displayed as **read only**.

The relationship name can be found in the data model of your queried record type. It is also the relationship name used to access the field when accessing the field from the queried record type.

For example: In the Product and Order example below, we are querying the record Order and including the field 'productName.' The query field might be defined as: 'recordType!Order.productRelationship.productName' where productRelationship is the relationship name between the two record types.

The configuration for this field is written as follows:

```
Unset
{
   textcolconfig(
      field: "relatedFieldName",
      linkConfig: null,
      title: "Field Name",
      relationshipName: "relationshipName",
      validator: null,
      readOnly: true,
      colWidth: null
    ),
}
```

E.g. displaying a product name that relates to each Order record:

Unset

```
{
   textcolconfig(
      field: "productName",
      linkConfig: null,
      title: "Product Name",
      relationshipName: "productRelationship",
      validator: null,
      readOnly: true,
      colWidth: null
    ),
}
```

Column Configuration Functions

Each field queried must be defined by a column configuration function. This function determines how data is **displayed**, **edited**, and **validated**. It also determines behavior when sorting or filtering on each column. The column configuration can be set by the Appian designer in the columnConfigs parameter using any of the following functions.

```
Unset
{
	textcolconfig(
		field: "id",
		linkConfig: null,
		title: "ID",
		relationshipName: null,
		validator: null,
		readOnly: false,
		colWidth: null
	),
	datecolconfig(
		field: "dateofbirth",
		dateFormat: "YYYY-MM-DD",
		correctFormat: false,
		title: "DOB",
		relationshipName: "directorRel",
		validator: null,
		readOnly: null,
		colWidth: null
	),
	...

}
```

textcolconfig (default cell type) - displays data as auto-wrapping text. Has the following required parameters:

- field - data field corresponding to the text column configuration. Must match the key field name in the data of your rows input. Can not be set to null.
- linkConfig - defines a link to a record view. Links must be configured with the recordlinkconfig() function. This will automatically make this column readOnly. Defaults to no link if set to null.
- title - column header title. Defaults to field name if set to null.

- relationshipName - name of relationship between the primary and related record. If the field is queried by a related record, this value must be set. Otherwise, this value can be set to null.
- validator - custom validator logic using the 'validator' function. Defaults to generic text validator if set to null.
- readOnly - whether data in the column is read only or not. Defaults to false if set to null.
- colWidth - initial width of the column on load. Valid values include AUTO, null, or an integer between 10 and 600. Defaults to AUTO if set to null.

numericcolconfig - displays numeric cell types and highlights non numeric data in red. Has the following required parameters:

- field - data field corresponding to the numeric column configuration. Must match the key field name in the data of your rows input. Can not be set to null.
- linkConfig - defines a link to a record view. Links must be configured with the recordlinkconfig() function. This will automatically make this column readOnly. Defaults to no link if set to null.
- format - display format of data in each cell. Valid options include "$0,0.00", "0,0", "0.0%", "0.00" for currency, large numbers, percentages, and decimals respectively. Defaults to no formatting if set to null.
- title - column header title. Defaults to field name if set to null.
- relationshipName - name of relationship between the primary and related record. If the field is queried by a related record, this value must be set. Otherwise, this value can be set to null.
- validator - custom validator logic using the 'validator' function. Defaults to generic text validator if set to null.
- readOnly - whether data in the column is read only or not. Defaults to false if set to null.
- colWidth - initial width of the column on load. Valid values include AUTO, null, or an integer between 10 and 600. Defaults to AUTO if set to null.

Example:

```
Unset
numericcolconfig(
      field: "advertisingRevenue",
      linkConfig: null,
      format: "$0,0.00",
      title: "Advertising Revenue",
      relationshipName: null,
      validator: null,
      readOnly: false,
      colWidth: null
```

```
    ),
```

datecolconfig - displays string input as a Date and uses a Date picker to edit each value. Has the following required parameters:

- field - data field corresponding to the Date column configuration. Must match the key field name in the data of your rows input. Can not be set to null.
- dateFormat - date display format for each cell. Examples of valid options include "MM-DD-YYYY", "MM/DD/YYYY", "MM/DD/YY", "MM/DD". Defaults to "MM-DD-YYYY" if set to null.
- correctFormat - whether to correct inputted format to match date format. Defaults to false if set to null.
- title - column header title. Defaults to field name if set to null.
- relationshipName - name of relationship between the primary and related record. If the field is queried by a related record, this value must be set. Otherwise, this value can be set to null.
- validator - custom validator logic using the 'validator' function. Defaults to generic text validator if set to null.
- readOnly - whether data in the column is read only or not. Defaults to false if set to null.
- dateFormat - string value. The format of your date's day, month, and year values. Default to 'DD/MM/YYYY'
- colWidth - initial width of the column on load. Valid values include AUTO, null, or an integer between 10 and 600. Defaults to AUTO if set to null.

```
Unset
 datecolconfig(
     field: "dateofbirth",
     dateFormat: "YYYY-MM-DD",
     correctFormat: false,
     title: "DOB",
     relationshipName: "directorRel",
     validator: null,
     readOnly: null
   ),
```

checkboxcolconfig - displays and allows manipulation of boolean values as checked or unchecked boxes. Value can be changed using a mouse click, 'Space' bar, or 'Enter' key. Multiple

cells can be changed simultaneously by selecting a range and pressing ‘space’ or ‘enter’. Has the following required parameters:

- field - data field corresponding to the checkbox column configuration. Must match the key field name in the data of your rows input. Can not be set to null.
- label - label in each cell. Defaults to no label if set to null.
- labelPosition - where to display the label on each cell. Valid options include "BEFORE" and "AFTER". Defaults to after label if set to null.
- checkedTemplate- value displayed as a checked box. Defaults to a boolean True if set to null.
- uncheckedTemplate - value displayed as an unchecked box. Defaults to a boolean False if set to null.
- title - column header title. Defaults to field name if set to null.
- relationshipName - name of relationship between the primary and related record. If the field is queried by a related record, this value must be set. Otherwise, this value can be set to null.
- validator - custom validator logic using the 'validator' function. Defaults to generic text validator if set to null.
- readOnly - whether data in the column is read only or not. Defaults to false if set to null.
- colWidth - initial width of the column on load. Valid values include AUTO, null, or an integer between 10 and 600. Defaults to AUTO if set to null.

```
Unset

checkboxcolconfig(
      field: "availableGlobally",
      label: "Is global?",
      labelPosition: "BEFORE",
      checkedTemplate: "true",
      uncheckedTemplate: "false",
      title: "Available Globally?",
      relationshipName: null,
      validator: null,
      readOnly: false,
      colWidth: 150
    ),
```

dropdowncolconfig - displays a list of options for user’s selection. Saved as a string value. Has the following required parameters:

- field - data field corresponding to the dropdown column configuration. Must match the key field name in the data of your rows input. Can not be set to null.

- source - List of options to display in dropdown. Value must be a non-null value of type list. Defaults cell to text input behavior if set to an empty list.
- strict - Whether to highlight input not present in the given source list. Defaults to false if set to null.
- filter - Whether to filter dropdown options as user types in the cell. Defaults to true if set to null.
- title - column header title. Defaults to field name if set to null.
- relationshipName - name of relationship between the primary and related record. If the field is queried by a related record, this value must be set. Otherwise, this value can be set to null.
- validator - custom validator logic using the 'validator' function. Defaults to generic text validator if set to null.
- readOnly - whether data in the column is read only or not. Defaults to false if set to null.
- colWidth - initial width of the column on load. Valid values include AUTO, null, or an integer between 10 and 600. Defaults to AUTO if set to null.

```
Unset
local!genres: {
    "Comedy",
    "Horror",
    "Documentary",
    "Drama",
    "Romance",
    "Thriller"
  },
dropdowncolconfig(
      field: "genre",
      source: local!genres,
      strict: true,
      filter: false,
      title: "Genre",
      relationshipName: null,
      validator: null,
      readOnly: false,
      colWidth: 200
    ),
```

Column Validator Function

The validator function plug-in means that custom validation logic can be applied to the grid data on a per column basis. Each column configuration function has a 'validator' parameter where this function can be used. This function has the following parameters:

- name - unique name to be assigned to your custom validator. Required value, no default.
- operator - operator for your validator logic. Valid operators include lessThan, greaterThan, equals, notEquals, regex, isTrue, isFalse, isNullOrEmpty, isNotNullOrEmpty, contains, and notContains. Required value, no default.
- value - value to compare cell value to using operator. Required for all operators with the exception of isTrue, isFalse, isNullOrEmpty, or isNotNullOrEmpty.

Certain operators are better suited to evaluate certain data types. See the below chart for which operators are permitted for each column configuration type:

	Checkbox	Numeric	Text	Date	Dropdown
lessThan	❌	✅	❌	✅	❌
greaterThan	❌	✅	❌	✅	❌
equals	✅	✅	✅	✅	✅
notEquals	✅	✅	✅	✅	✅
regex	❌	❌	✅	✅	✅
isTrue	✅	❌	❌	❌	❌
isFalse	✅	❌	❌	❌	❌
isNullOrEmpty	✅	✅	✅	✅	✅
isNotNullOrEmpty	✅	✅	✅	✅	✅
contains	❌	❌	✅	❌	✅
notContains	❌	❌	✅	❌	✅

When querying related record data, it is important to note which fields are from 1 to Many vs. Many to 1 relationships. A field queried from a 1 to Many (1:N) relationship is an array type and is validated differently. Supported operators on these fields are:

- isNull, isNotNull, contains, notContains, equals and notEquals

Example using a validator on text and checkbox columns:

```
Unset
{
   textcolconfig(
      field: "email",
      title: "Email",
      relationshipName: null,
      validator: validator(
        name: "verifyAppianEmail",
        operator: "regex",
        value: "@appian\.com"
      ),
      readOnly: false
   ),
   checkboxcolconfig(
      field: "availableGlobally",
      label: null,
      labelPosition: null,
      checkedTemplate: null,
      uncheckedTemplate: null,
      title: null,
      relationshipName: null,
      validator: validator(
        name: "testing",
        operator: "isTrue",
      ),
      readOnly: null
   ),

}
```

Export Configuration Function Plug-In

The exportconfig() function is made available through Version 3.0 of the Grid+ Helper Function Plug-In. This function provides guidance to configuring your component parameter 'exportConfig'. The following parameters are available to configure your export behavior:

- fileName (Text) - Name of the exported csv file. Use '[YYYY]-[MM]-[DD]' in the text value to include the current date in the file name. This name does not include the file's csv extension. Defaults to DataExport_YYYY-MM-DD if set to null.
- columnDelimiter (Text) - Value of your column delimiter. Defaults to ',' if set to null.
- rowDelimiter (Text) - Value of your row delimiter. Defaults to '\r\n' if set to null.
- columnHeaders (Boolean) - Whether columnHeader names should be included in export. Defaults to false.
- exportHiddenColumns (Boolean) - Whether to include hidden columns in exported file. Defaults to false.
- exportRange (Text) - Range of rows and columns to export. Defined by the exportrange function. Structure of range is [startRowIndex, startFieldName, endRowIndex, endFieldName]. Defaults to all rows and fields if null or undefined.
    - startRow (Number)
    - startFieldName (Text)
    - endRow (Number)
    - endFieldName (Text)
- buttonLabel (Text) - Optional text value to display with download icon in export button. Defaults to no text if set to null.

Example of a basic export button configuration:

```
Unset
gridPlusLayout(
       ...
       exportConfig: exportconfig(
         fileName: "GridPlusExample",
         columnDelimiter: ",,",
         rowDelimiter: null,
         columnHeaders: false,
         exportHiddenColumns: true,
         exportRange: exportrange(
           startRow: 1,
           startFieldName: "id",
           endRow: 50,
           endFieldName: "title"
         ),
         buttonLabel: "Download Report"
       )
```

```
        ...
)
```

Record Link Configuration Function Plug-In

The recordlinkconfig() function is uploaded with the Grid+ Helper Plug-In. This function defines the values needed to configure a column of Grid+ to display links to record views. Recordlinkconfig has the following parameters:

- recordTypeXML (Text) - The record type to link to. Defined by a record reference wrapped in a toxml() SAIL function. Ensure the resulting value contains the UUID of the recordType reference. Required, no default value.
- identifierFieldName (Text) - The name of the identifier field of the record. This is typically the primary key field. Required, no default value.
- dashboard (Text) - The URL stub for the record view that the link will open. Go to the record type and click Views to find the URL stub for each record view. Default is "summary" if set to null.

Note:

1. When setting the value for the 'identifierFieldName', ensure that this field is both present in your the list assigned to rowsValue and is defined in columnConfigs.
2. Configuring the linkConfig parameter in a column means that it will always be made readOnly.
3. If setting a record link for a 1:N related record field that displays multiple values per cell, ensure that the identifier field specified in identifierFieldName also represents the 1:N relationship with an array of values.

Example of a portion of a grid's columnConfigs parameter where 'id' and 'title' fields are defined. The linkConfig in the 'title' field is defined with reference to the 'id' field. This means that for each row of the grid, the link created in the title field will be formed from the value in the 'id' field. This works because 'id' is the primary key of the given record in the recordTypeXML parameter.

```
Unset
columnConfigs: {
        textcolconfig(
             field: "id",
             linkConfig: null,
             title: "ID",
             relationshipName: null,
```

```
            validator: null,
            readOnly: null,
            colWidth: 45
          ),
      textcolconfig(
            field: "title",
            linkConfig: recordlinkconfig(
                  recordTypeXML:toxml(<recordType!EG Show>),
                  identifierFieldName: "id",
                  dashboard: "summary"),
            title: "Title",
            relationshipName: null,
            validator:null,
            readOnly: false,
            colWidth: 200
       ),
      ...
}
```

Creating, Updating, and Writing to Records

As changes are made to the data displayed in the grid, they are updated in the list references assigned to the changeData and deleteData parameters. In this section, find instructions on casting the contents of these variables to records and saving them to the data fabric. **It is highly encouraged to follow the recipe below to ensure that changes are securely applied to records that only a given user has access to.**

Creating and Updating Records Recipe

Given the necessary parameters have been defined in the component (primaryKeyFieldName, changeDataValue/changeDataSaveInto for saving), add the following SAIL logic to the saveInto parameter of your form's button widget.

1. Prior to defining an expression that applies changes to a record, define an expression rule that extracts the non-readonly columns in the ColumnConfig. This is to ensure that only column configurations with a null or false value for the **readOnly** field can be edited and saved by the user.
    a. Shown below is a sample expression rule. In the sample application, this is referred to as **GA_GetNonReadOnlyColumns.**

       Rule Inputs:

- **configs** (List of Text): representing a list of column configurations

```
Unset
a!localVariables(
  local!list: ri!configs,
  a!forEach(
    local!list,
    if(
      a!isNullOrEmpty(a!fromJson(fv!item)["readOnly"]),
      a!fromJson(fv!item)["data"],
      {}
    )
  )
)
```

2. Cast each change object to a record and save in a new list, e.g. local!newRecordList. To securely apply these record changes, define an expression rule with the following rule inputs and expression contents:

   Rule Inputs:
   - **recordTypeCast** (Any Type): record type constructor in the form of the record type reference with parentheses. Example with the record type "EG Business": recordType!<EG Business>
   - **newData** (Map): variable assigned to changeData
   - **allFields** (Any Type): List of all record field references displayed in the grid
   - **primaryKey** (Text): String for the name of the primary key of the specified record type. For instance, if the record "EG Business" has a primary key titled "id", then this will be the value of the rule input.
   - **nonReadOnlyCols** (Text): List of columns that are editable (non-readonly)

Sample Expression Rule (can be used for both creates/updates and deletions)
At a high level, this Appian expression rule processes incoming a list of changes or deletions (ri!newData) by determining whether each record should be created or updated based on the presence of a primary key (ri!primaryKey) and the record type (ri!recordTypeCast). It maps field names to their corresponding UUIDs, filters for editable fields, and retrieves existing records (via queryByIdentifier) when necessary. Finally, it updates or creates records accordingly and returns only valid processed records.

Unset

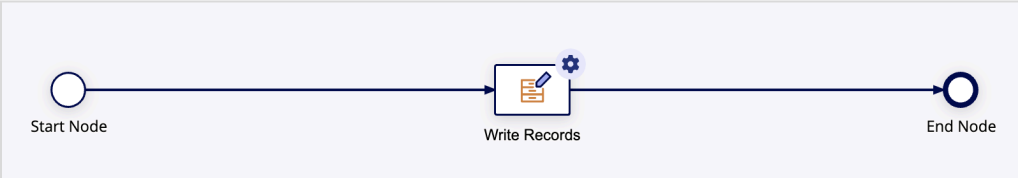
```
a!localVariables(
  local!allFields: ri!allFields,
  local!allFieldsStrings: a!forEach(
    items: local!allFields,
    expression: tostring(fv!item)
  ),
  /* check fields that are editable and accessible to the user */
  local!editableFields: intersection(ri!nonReadOnlyCols, local!allFieldsStrings),
  /* Create a mapping of field names to UUIDs for easy lookup */
  local!fieldNameToUuidMap: a!update(
    a!map(),
    a!forEach(
      items: local!allFields,
      expression: touniformstring(fv!item)
    ),
    local!allFields
  ),
  local!processedRecords: a!forEach(
    items: ri!newData,
    expression: a!localVariables(
      local!currentRow: fv!item,
      local!primaryKeyValue: fv!item[ri!primaryKey],
      local!editableKeys: a!forEach(
        items: a!keys(fv!item),
        expression: if(
          contains(local!editableFields, fv!item),
          fv!item,
          {}
        )
      ),
      local!mappedFieldUuids: a!forEach(
        local!editableKeys,
        /* Retrieve field UUIDs for each editable key in the row */
        index(local!fieldNameToUuidMap, fv!item)
      ),
      if(
        /* Determine whether to create a new record or update an existing one based on primary
key presence */
        a!isNullOrEmpty(local!primaryKeyValue),
        cast(ri!recordTypeCast, fv!item),
        /* Case where the primary key is present (update scenario) */
        a!localVariables(
          local!existingRecord: a!queryRecordByIdentifier(
            recordType: ri!recordTypeCast,
            identifier: local!primaryKeyValue,
            fields: local!mappedFieldUuids /* Only fetch necessary fields */
          ),
          if(
            a!isNotNullOrEmpty(local!existingRecord),
            a!update(
              data: local!existingRecord,
              index: local!mappedFieldUuids,
              value: a!forEach(
```

```
                items: local!editableKeys,
                expression: index(local!currentRow, fv!item)
              )
            ),
            null()
          )
        )
      )
    )
  ),
  reject(a!isNullOrEmpty, local!processedRecords)
)
```

*In the sample application, an expression rule titled **GA_GetShowRecordsFromChanges** includes the expression above. This is applicable for creates/updates and deletions. **Please reference how this rule is defined and called in the GA_INT_GridPlus_Update interface for more context.** This is also demonstrated in the image below. Be sure to note the use of the **GA_GetNonReadOnlyColumns()** expression rule, which was discussed earlier.

```
a!save(
  local!changedRecordList,
  rule!GA_GetShowRecordsFromChanges(
    recordTypeCast: recordType!GA Show,
    newData: local!changes,
    allFields: local!fields,
    primaryKey: "id",
    nonReadOnlyCols: rule!GA_GetNonReadOnlyColumns(rule!GA_Show_ColConfigs_Related())
  )
)
```

3. Write the contents of the new list to the appropriate record database by setting up a **process model** with a **Write Records** node. Images of this model and its process variables are shown below for reference. In the sample application, this process model is titled "GA Create Update Shows":



Name	Type	Value	Parameter?	Required?	Multiple?	Hidden?	
records	GA Show (Record Type)	(No Value)	Yes	No	Yes	No	✕

   a. This process model should include a process variable titled as "records", which is of type List of <Record Type>, which is also a process parameter.

4. To successfully initiate the process model, leverage the **a!startProcess()** function, passing in the appropriate process parameters (in this case, the new list of records). Shown below is example SAIL code for this operation:

```
Unset
        a!startProcess(
                          processModel: cons!GACreateUpdateShowsModel,
                          processParameters: {
                            records: local!changedRecordList
                          },
                          onSuccess: {
                            a!save(local!changes, {})
                          },
                          onError: a!save(local!errors, "Unable to update records")
                        )
```

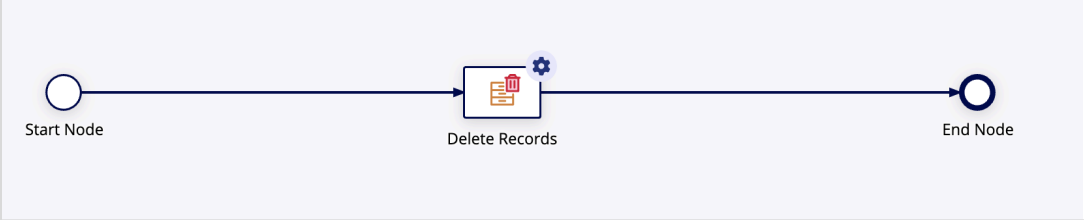
Examples of this logic can be found in the **GA_INT_GridPlus_Update** interface of the sample application. **To ensure that the logic for applying record changes and initiating process models works correctly, follow the expression defined for the GA_INT_GridPlus_Update interface.**

Deleting Records Recipe

Deleting records can be accomplished through the pop up context menu within the grid's interface. After selecting the rows to delete, control click to show the menu, and select 'Remove Rows.'

Once the primaryKeyFieldName and deleteDataSaveInto/deleteDataValue have been properly set, follow these general steps to register the changes in the data fabric. See above for more detailed guidance and expression rules to cast from a Javascript object to record data.

1. Cast and store the changes in the deleteData reference to records using the above expression or similar logic.
2. Set up a process model with a **Delete Records** node. Images of this model and its process variables are shown below for reference. In the sample application, this process model is titled "GA Delete Shows":



Name	Type	Value	Parameter?	Required?	Multiple?	Hidden?	
records	GA Show (Record Type)	(No Value)	Yes	No	Yes	No	✕

   a. This process model should include a process variable titled **"records"**, which is of type List of <Record Type> and a process variable.
   b. For the **Delete Records** node, set the value of **Records** field in the Node Inputs section to be pv!records
3. To successfully initiate this process model, use the **a!startProcess()** function, passing in the appropriate list of deleted records as a process parameter. Shown below is example SAIL code for this operation:

```
Unset
        a!startProcess(
                          processModel: cons!GADeleteShowsModel,
                          processParameters: {
                            records: local!deletedRecordsList
                          },
                          onSuccess: {
                            a!save(local!deletions, {})
                          },
                          onError: a!save(local!errors2, "Unable to delete records")

                        )
```

Installation

From the Appian AppMarket, download the component zip called Grid+ Component Plug-In and the associated Java servlet called Grid+ Helper. The Grid+ Component holds the Appian component called gridPlusLayout.

The Java servlet will allow you to use the column configuration functions when defining your grid.

Sample App Setup

1. Install the Grid+ Component and Helper Plug-In from the AppMarket or the admin console.
2. Download and unzip the package called 'Grid+ Sample App' from the component AppMarket listing.
3. From the Appian Designer tab, import the sample application, Grid+ App, into your site.
4. From the Cloud Database tab, import the 3 SQL files into your database.

    a. This will create the table structures for GA_SHOW, GA_AWARD, and GA_DIRECTOR. The tables will not yet have data.
5. Go to the Grid+ Demo site. From the first screen 'Populate Records,' select the three records in the dropdown to view new sample data for each. Press 'Load Sample Data' to create the records.
    a. If the records are successfully created, a message will appear below the grid.
6. Navigate to the 'Update Show Record' tab. This tab displays a query of Show data, including several fields from the related records Director and Awards.
7. Try out the features mentioned previously in this interface, including creating, editing, and deleting data.

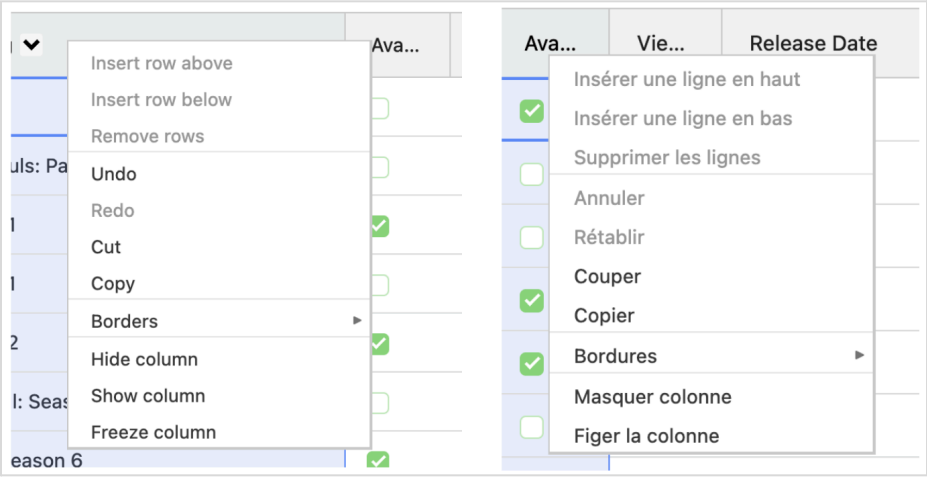
Security

If displaying data queried from an Appian record query, the viewing security defined in the relevant record type persists. If a user can view but not edit data, the grid component's editability can be defined by the readOnly parameter. **To ensure that changes (creates, updates, deletes) are applied to the appropriate records, be sure to utilize the expression rules defined in the Creating, Updating, and Writing To Records section.**

Internationalization

The Grid+ Component now supports internationalization for operation labels. Internationalization will be automatically applied based on the locale defined in the Admin Console's Internationalization settings.

Supported locales: ar-AR (Arabic), zh-CN (Chinese), nl-NL (Dutch), en-US (English), fr-FR (French), de-DE (German), it-IT (Italian), ja-JP (Japanese), ko-KR (Korean), pl-PL (Polish), pt-BR (Portugese), ru-RU (Russian), and es-MX (Spanish)



Recommended Use Cases

Grid+ allows you to provide enhanced functionality for your users. Use Grid+ if you want your users to be able to:

Import and display data from a spreadsheet

- Import spreadsheet data and display it in Grid+
- Validate entries based on custom validation criteria for each column
- Edit imported data with Excel-like abilities
- Save spreadsheet data as newly created record types in seconds

Customize the display of large data sets

- View up to 5,000 records at a time to display in Grid+
- Configure column-level data types to specify cell formatting, manipulation, and validation
- Sort and filter data in each column
- Drag and drop, resize, hide, or freeze columns to adjust data display

Bulk create and delete record data

- Create new records directly from the grid
    - Enter new data into empty row at the bottom of the grid
    - Use the context menu to create a new row at any point within the grid
- Delete records from data fabric by selecting rows and using 'Remove rows' in the context menu (command + click for MacOS)
- Select a range of cells to manipulate
    - Drag and drop range selection
    - Keyboard shortcuts seen in Excel (command + arrow keys for MacOS)
- Modify selected cells
    - Empty by pressing the delete key
    - Fill with values by the 'paste' keyboard shortcut
    - Drag values from other cells by their bottom right cell corner

Performance Considerations and Component Limitations

- The number of records displayed at a time is limited to 5000. If you would like to sort through and manipulate more than 5k at once, it is recommended to leverage Appian's querying logic in the interface to minimize the query size before manipulation.
- For text fields, limit the character input to prevent excessive size and potential memory issues.
- All fields queried and defined in the columnConfig parameter must have unique field names. For example, the grid can not handle querying a Show field with the name 'name' and a related Award field also with the name 'name.'
- Editing of related record fields is not supported.

- After changing component configuration values, a refresh of the grid is required to load the changes.
- Supported data types include those defined by the column configuration functions. Complex cell types which are supported by Appian's native grid components but not supported by Grid+ include:
    - Barcode
    - Dates of Appian date type**
    - Date & Time**
    - Dropdowns with differing values and labels
    - Encrypted Text
    - File Upload
    - Image
    - Multiple Dropdown
    - Pickers
    - Progress Bar
    - Radio Buttons
    - Rich Text Display
    - More information on the supported contents of a row in an Editable Grid Component can be found here

  **A workaround exists to present Appian date types using the todate() and tostring() SAIL functions.