Extended Documentation

1. Introduction

1.1. Execution Flow

Extended code is executed sequentially from top to bottom. The value returned is the result of the last line of code executed. All preceding lines of code are executed, but their results are not part of the final return value. If the final line is part of a statement, like a forEach loop, the result of that entire statement will be the return value. It is crucial to be mindful of this behavior to ensure the intended value is returned.

Example:
The following code would return 8, as it is the result of the last line.

```
2+2
3+3
4+4
```

1.2. Comments

Comments are used to add explanatory notes within the code that are ignored during execution. They are particularly useful for documenting complex code or specific functions to make it understandable for others.

- **Single-line comments:** Start with a double forward slash //.
  ```
  // This is a single-line comment
  ```

- **Multi-line comments:** Enclosed between /* and */.
  ```
  /*
   * This is a multi-line comment.
   * It can span across several lines.
   * It's useful for explaining complex logic.
   */
  ```

2. Variables

Variables store values, such as the result of an expression, and are identified by a unique name. This name is then used to access the stored value in subsequent code.

2.1. Declaration and Assignment

- **Syntax:** <variable name> := <value OR expression>
- The := operator assigns the value on the right to the variable on the left. This is different from the equality operator =.
- Variables must be assigned a value upon declaration.
- Variables can be referenced by their name or by wrapping the name in ${}. For example, var is the same as ${var}.

Examples:

```
myNum := 5 // Assigns the integer 5 to myNum
myStr := "Hello World" // Assigns the string "Hello World" to myStr
total := 1 + 2 // Evaluates the expression and assigns the result (3) to total
```

Once assigned, variables can be used in other expressions:

```
apples := 10
oranges := 15
totalFruits := apples + oranges // totalFruits is now 25
```

2.2. Variable Scope

A variable's scope extends throughout the entire expression. This means a variable created within a conditional block (like an IF-THEN-ELSE statement) is accessible outside of that block.

Example:

```
var := 10
IF var > 5 THEN
  var2 := "I'M STILL ACCESSIBLE"
ENDIF
var2 // "I'M STILL ACCESSIBLE"
```

3. Conditional Logic: IF-THEN-ELSE

Conditional statements allow the code to execute different blocks of code based on whether a condition evaluates to True or False.

- **Structure**:
  ```
  IF <condition> THEN
    // Code to execute if condition is true
  ELSE
    // Code to execute if condition is false
  ENDIF
  ```
- **Best Practice**: It is highly recommended to **always include the ELSE clause**, even if it is empty (e.g., ELSE ""). Omitting the ELSE clause can sometimes lead to unexpected errors or behaviors, so including it ensures that the statement is always complete and explicit.
- **Negation (NOT)**: Extended does not have a universal NOT operator for inverting boolean expressions directly. To check for a false condition, you must explicitly compare the boolean value to false or use the inequality operator !=.
  ```
  myBool := false

  // Correct way to check for a false condition
  IF myBool = false THEN
    // ...
  ENDIF

  // This will NOT work
  // IF NOT myBool THEN ... ENDIF
  ```

4. Extended Objects and Locations

Extended code can be written in various objects and locations within the system. These are categorized as primary and secondary locations.

4.1. Primary Locations

These are objects that require Extended code to function.

- **Extended Function**: An Extended expression that is evaluated only at its specific location in the Model and cannot be called from elsewhere. It can, however, call an Extended Expression.
- **Extended Expression**: A reusable piece of Extended code that can be called from multiple locations. These are stored in the Configuration Studio's Expression area. When called, the expression is evaluated within the context of where it was called from, unless a different context is specified using the calculate() function.
    - **Calling an Extended Expression**:
        - Using the expression property: t.<expressionID>.expression
        - Wrapping the ID in curly braces: {<expressionID>}
- **Extended Table**: An object that generates a table from Extended code. The code must evaluate to either a Table or a List.
- **Extended Property**: A property that can be added to an object and contains an Extended expression that returns a value. The return data type must be specified during configuration.
- **Extended Action**: An "Action" in the Workflow area of Configuration Studio that supports transactional Extended. It executes when its parent Event rule runs, using the objects returned by the rule as its context.
- **Extended Target**: An object in the Transformer area that serves as an endpoint for a Transformer job and supports transactional Extended. It allows writing Extended code that uses cell data from the Transformer job.

4.2. Secondary Locations

These are locations where Extended code is optional and is used to enhance functionality. This is often enabled via an "Advanced Mode" checkbox.

- **Charts**: Most charts have an Extended expression that can resolve to a table, which can be used as an alternative to the Table reference property.
- **Dataset Tables**: An Extended expression can be used to query a list of objects for an axis.
- **Action Button**: When using the "Add" functionality, an Extended expression is required to determine the destination for the new object.

5. Style Guide

A recommended style guide ensures consistency and readability of Extended code.

5.1. Indentation

- **Hanging Indent**: Subsequent lines are indented. Use a tab or 4 spaces.
```
SELECT Kpi FROM this.object
    WHERE <condition1>
    AND <condition2>
```

- **Vertically Aligned Indentation**: Subsequent lines are aligned with an opening delimiter or dot notation.
```
myList.filter(<condition1>)
```

```
        .filter(<condition2>)
        .as(<expression>)
```

5.2. Blank Lines

Use blank lines to logically separate and group sections of code.

5.3. Whitespace

- Use a space to separate operators from their operands (e.g., var := "Hello", 5 + 5).
- Use a space after a comma when delimiting values (e.g., LIST(1, 2, 3)).

5.4. Naming Conventions

- **Variables**: Use camelCase and make names descriptive. Avoid reserved words.
- **Extended Expressions**: Use descriptive camelCase IDs.
- **Properties**: Use descriptive camelCase IDs and avoid conflicts with system properties.

5.5. String Quotation Marks

Use either single (') or double (") quotes for strings, but be consistent.

5.6. Comments

Always add comments to longer or more complex Extended code. A multi-line comment at the beginning should explain the overall purpose, with smaller comments for specific code blocks.

5.7. Queries

- Write keywords SELECT, FROM, and WHERE in uppercase.
- For long queries, break them onto new lines with hanging indents for each condition.

6. Data Types

6.1. Number

Numbers can be integers or decimals (using a period . as the separator). Thousands separators are not supported.

- **Numeric Operators**: + (addition), - (subtraction), * (multiplication), / (division), % (modulo).
- **Operator Precedence**:
  1. () (parentheses)
  2. /, %, * (evaluated left-to-right)
  3. +, -
- **Special Values**:
  - NA: Not Available, indicates the absence of a number value.
  - NaN: Not a Number, indicates an invalid number, such as the result of division by zero.
- **Numeric Functions**:
  - abs(value): Returns the absolute value.
    ```
    abs(-13) // 13
    myNumber := -500
    abs(myNumber) // 500
    ```
  - cbrt(value): Returns the cube root.
    ```
    cbrt(729) // 9
    myNumber := 125
    cbrt(myNumber) // 5
    ```

- ceil(value): Rounds up to the nearest integer.
  ```
  ceil(13.3) // 14
  ceil(0.5) // 1
  ```

- floor(value): Rounds down to the nearest integer.
  ```
  floor(13.3) // 13
  floor(0.5) // 0
  ```

- pow(base, exponent) **[5.2.5.1+]**: Calculates exponents.
  ```
  pow(5, 2) // 25
  pow(4, 2) // 16
  ```

- round(value): Rounds to the nearest integer (0.5 rounds up).
  ```
  round(13.3)  // 13
  round(0.5) // 1
  ```

- sqrt(value): Returns the square root.
  ```
  sqrt(81) // 9
  myNumber := 225
  sqrt(myNumber) // 15
  ```

6.2. Traditional Functions

Functions from the Function Calculator can be used in Extended and will yield a numeric value. The ${} syntax is not required but will be evaluated correctly if present.

```
// Calculates value of Node with ID 100, Node Type defaults to Actual
[100]

// Calculates value of Node with ID 100 for Node Type Actual
A[100]

// Calculates value of Node with ID 100 for the current org + children
AGG('A[100]', *this)

// Calculates the value of the node in the nodeReference property
// for the organizations in the orgRollup property of the object
AGG('A[${this.object.nodeReference.id}]', ${this.object.orgRollup})

// Same as above, without the ${} syntax
AGG('A[this.object.nodeReference.id]', this.object.orgRollup)
```

6.3. String

Strings can be enclosed in single or double quotes.

- **Escaping Characters**: Use a backslash \ to escape characters with special meaning (e.g., 'Extended\'s functionality').
- **Wildcard Character**: The asterisk * is a wildcard that matches any number of unknown characters. It can be used at the start, middle, or end of a string for partial matching.
- **Concatenation**: The + operator joins strings.
- **String Comparison**: The equality operator = is case-sensitive for normal strings but case-insensitive when using wildcards.
- **String Methods**:
  - string.indexOf(substring, [startIndex]): Returns the index of the first occurrence of a substring. Returns MISSING if not found.
    ```
    modelID := "ORG007_KP005_TABL001"
    undIndex := modelID.indexOf("_") // 6
    undIndex2 := modelID.indexOf("_", undIndex + 1) // 12
    ```
  - string.size(): Returns the length of the string.
    ```
    "hello".size() // 5
    ```
  - string.strip(): Removes extra leading, trailing, and inline spaces.
    ```
    myString := "  There are too many    spaces in this string  "
    myString.strip() // "There are too many spaces in this string"
    ```
  - string.substring(startIndex, [endIndex]): Returns a fragment of the string.
    ```
    "abcdefg".substring(3)    // "defg"
    "abcdefg".substring(3, 4) // "d"
    ```

6.4. Boolean

Represents True or False values.

- **Logical Operators**: AND, OR. The NOT operator is limited and can only be used to invert CONTAINS and IN operators. For general boolean negation, see Section 3 on Conditional Logic.
- **Operator Precedence**:
  1. () (parentheses)
  2. AND
  3. OR
- **Relational Operators**: >, <, >=, <=, =, != (or <>), CONTAINS, IN.
- **Special Boolean Properties**:
  - visible: True if the object is visible.
  - inheritVisible: True if the object and all its parents are visible.
  - inScope: True if the object's lifetime is within the current period.
  - inheritScope: True if the object and all its parents' lifetimes are within the current period.
  - available: True if the object is both visible and inScope.
  - inheritAvailable: True if the object and all of its parents are visible and inScope.

6.5. Date

Dates follow the ISO 8601 standard YYYY-MM-DD. They are stored at the millisecond level.

- **Date Keywords**:
  - TODAY: The current server date and time.
  - BOP: The start date of the current context period.

  - EOP: The end date of the current context period.
  - BOY: The start of the calendar year that the current context period corresponds to.
  - EOY: The end of the calendar year that the current context period corresponds to.
- **Time Adjustments**: Dates can be adjusted using traditional modifiers (e.g., BOP - 1M, EOP - 1Y).
- **Date Functions**:
  - date(string): Converts a string to a date.
    ```
    date("04-07-2018") // Jul 4, 2018 00:00:00
    date("17/05/2018") // May 17, 2018 00:00:00
    ```

6.6. Object

Represents Corporater BMP objects (e.g., KPIs, Scorecards). Properties of the object can be accessed.

- **Referencing**: Objects are referenced by their ID space and ID (e.g., o.123, t.KPI001).
- **Object Equality**: Objects are equal if they have the same ID and ID space.
- **Object Methods**:
  - ancestor(type) **[5.2.0.1+]**: Returns the first ancestor object of a matching type.
    ```
    // Equivalent to t.123.scorecard
    t.123.ancestor(Scorecard)
    // Gets the ancestor indicator list
    t.123.ancestor(IndicatorList)
    ```
  - children([type1], [type2], ...) **[5.2.0.1+]**: Returns a list of first-level child objects, optionally filtered by type.
    ```
    // Works the same as .children without parameters
    this.object.children()
    // Equivalent to t.123.children.filter(className = 'BarChart')
    t.123.children(BarChart)
    // Returns both BarChart and LineChart children
    t.123.children(BarChart, LineChart)
    ```
  - descendants([type1], [type2], ...) **[5.2.0.1+]**: Returns a list of all child objects from all levels, optionally filtered by type.
    ```
    // Gets a list of all descendants
    this.object.descendants()
    // Same as o.100.kpis
    o.100.descendants(Kpi)
    // Returns a list of all barcharts and line charts in organisation 100 and sub-organisations
    o.100.descendants(BarChart, LineChart)
    ```
  - generate([includeId]): Outputs the transactional code to replicate the object(s). includeId is a boolean.
    ```
    t.123.generate() // Code to create object with ID 123 and any descendants
    t.123.generate(true) // Same, but includes the IDs
    ```
  - rref([property], [type], [startDate], [endDate]) **[5.2.0.1+]**: Performs a reverse lookup of objects that reference the current object.

```
myKpi := t.KP001
// Returns a list of the Action Plans with an affects property that contains myKpi
myKpi.rref(affects, ActionPlan)
```

- tree([childExp], [boolExp]) **[5.2.0.1+]**: Creates a nested list structure (treelist) for tree tables. The childRequest is an expression that resolves to a list of children, and the same request is called on nested children downwards. The default is children. collapse is a boolean expression to indicate if an object should be collapsed.

```
// Create a tree list of all organisations
root.organisation.tree(organisations)

// Create a tree list of organisations containing "Department" and their child Scorecards
root.organisation.tree(children.filter(name = '*Department*' OR className = 'Scorecard'))

// Create a treelist of managers and their subordinates (custom property)
managers := SELECT User where membership contains g.managers
managers.tree(subordinates)

// Use rref to build a tree based on a reverse reference
managers.tree(self.rref(managers))

// Use an IF statement to resolve multi-property trees
g.managers.tree(
  IF self.manager THEN
    members
  ELSE
    endusers
  ENDIF
)
```

- url([...]) **[5.3.0.0+]**: Creates a clickable link. The link can be to the object itself or an external URL. It can be displayed as the object's icon, custom text, a custom icon, or a combination. The functionality is determined by the order of the arguments provided.
  Argument Order and Functionality
  The function of each argument is determined by its position and data type:
  1. **Object Reference (objectReference | FileResource)**: An object reference or a reference to an SVG file in Resources (r.myFileResourceID). This will be displayed as an icon. It is best practice to make this the first argument if used.
  2. **First String (displayText)**: The text to be displayed as the clickable link. If this is the only argument, no icon will be shown.
  3. **Second String (tooltipText)**: Text that appears as a tooltip when the user hovers over the link.
  4. **Third String (externalURL)**: An external URL to link to. If this is provided, the link will point here instead of the object the method is chained to.

Syntax<object>.url()
// Result: A link to the object, displayed as the object's default icon.

<object>.url([displayText])
// Result: A link to the object, displayed as displayText with no icon.

<object>.url([objectReference], [displayText])
// Result: A link to the object, displayed as the icon of objectReference and displayText side-by-side.

<object>.url([FileResource], [displayText])
// Result: A link to the object, displayed as a custom icon from FileResource and displayText side-by-side.

<object>.url([displayText], [tooltipText])
// Result: A link to the object, displayed as displayText with a tooltip and no icon.

<object>.url([displayText], [tooltipText], [externalURL])
// Result: A link to externalURL, displayed as displayText with a tooltip and no icon.

<object>.url([FileResource], [displayText], [tooltipText], [externalURL])
// Result: A link to externalURL, displayed with a custom icon, displayText, and a tooltip.
Examples// A clickable KPI icon linking to KPI001
t.KPI001.url()

// Clickable "My KPI" text linking to KPI001
t.KPI001.url("My KPI")

// Clickable "Click Here" text linking to the parent of KPI001, with tooltip "Parent Objective"
t.KPI001.parent.url("Click Here", "Parent Objective")

// Clickable Strategic Objective icon + "Click Here" text linking to the parent of KPI001
t.KPI001.parent.url(self, "Click Here")

// Clickable "External Dashboard" text linking to an external site with a tooltip
t.KPI001.url("External Dashboard", "BI Dashboard", "https://company.com/dashboards/KPI001")

// Clickable custom icon from Resources + "Dashboard" text linking to an external site with a tooltip
t.KPI001.url(r.TABLEAUICON, "Dashboard", "Tableau", "https://company.com/dashboards/KPI001")

6.7. List

An ordered collection of elements of any data type.

- **Creation**: LIST(item1, item2, ...)
- **List Operators**:
  - CONTAINS: Checks if a list contains an item.
  - IN **[5.3.0.0+]**: Checks if an item is in a list.
  - NOT: Inverts CONTAINS and IN.
- **List Methods**:
  - as(property): Returns a new list with the values of the specified property for each object in the original list.
    basketballKpis := list(t.KP001, t.KP002, t.KP003)
    basketballKpis.as(responsible) // [Michael Jordan, Larry Bird, Hakeem Olajuwon]

  - avg(): Returns the average of numeric values in the list. Non-numeric values are excluded. Returns NA if no numbers are present.
    myList := LIST(50, 100)

myList.avg() // 75

- distinct(): Returns a list with duplicate elements removed.
  animals := LIST("dogs", "cats", "tigers", "dogs")
  animals.distinct() // ["dogs", "cats", "tigers"]

- filter(condition): Returns a list of elements that meet the condition.
  myKpis := LIST(t.200, t.201, t.202) // KPIs: Revenue, Upsell Revenue, Net Income
  myKpis.filter(name = "*Revenue*") // [Revenue, Upsell Revenue]

- first([n]): Returns the first element or a list of the first n elements.
  myBands := LIST("Dire Straits", "Rainbow", "Deep Purple")
  myBands.first() // "Dire Straits"
  myBands.first(2) // ["Dire Straits", "Rainbow"]

- groupBy(property): Groups items by a property and allows for aggregate calculations.
  list := SELECT ceProcedure
  list.groupBy(name).count()
  list.groupBy(name).sum(weight)

- item(n): Returns the element at index n.
  myBands := LIST("Dire Straits", "Rainbow", "Deep Purple")
  myBands.item(1) // "Rainbow"

- join(separator): Concatenates all items into a string, separated by the separator.
  myList := LIST("ABC", "DEF", "GHI")
  myList.join("") // "ABCDEFGHI"

- last([n]): Returns the last element or a list of the last n elements.
  myBands := LIST("Dire Straits", "Rainbow", "Deep Purple")
  myBands.last() // "Deep Purple"
  myBands.last(2) // ["Rainbow", "Deep Purple"]

- map(keyExp, [valueExp]): Converts a list into a map, allowing for grouping and counting.
  // Count all KPIs by status
  allKpis := SELECT Kpi
  allKpis.map(statusClassification).count().table('name','amount')

- max(): Returns the largest numeric value in the list. Non-numeric values are excluded. Returns NA if no numbers are present.
  myList := LIST(55, 1, 24)
  myList.max() // 55

- merge(list2): Returns a new list containing all distinct elements from both lists.
  fruits1 := LIST("nectarines", "bananas", "grapes")
  fruits2 := LIST("raspberries", "pears", "mangos")
  fruits1.merge(fruits2) // ["nectarines", "bananas", "grapes", "raspberries", "pears", "mangos"]

- min(): Returns the smallest numeric value in the list. Non-numeric values are excluded. Returns NA if no numbers are present.
  myList := LIST(55, 1, 24)
  myList.min() // 1

- remove(item): Removes all instances of an item or a list of items.
  myList := list(1, 3, 5, 1, 1, 1, 7)
  myList.remove(1) // [3, 5, 7]

- reverse(): Returns a reversed version of the list.
  numberList := LIST(1, 100, 55, 3, 89)
  numberList.reverse() // [89, 3, 55, 100, 1]

- size(): Returns the number of elements in the list.
  myFruits := LIST("Nectarines", "Bananas", "Grapes")
  myFruits.size() // 3

- sort([expression]): Returns a sorted version of the list (ascending).
  numberList := LIST(1, 100, 55, 3, 89)
  numberList.sort() // [1, 3, 55, 89, 100]

- sortReverse([expression]): Returns a sorted version of the list (descending).
  numberList := LIST(1, 100, 55, 3, 89)
  numberList.sortReverse() // [100, 89, 55, 3, 1]

- sum(): Returns the sum of numeric values in the list. Non-numeric values are excluded. Returns NA if no numbers are present.
  myList := LIST(10, 15, 50)
  myList.sum() // 75

- union(list2): Returns a list containing all elements from both lists, including duplicates.
  list1 := LIST("abc", "abc")
  list2 := LIST("abc", "def")
  list1.union(list2) // ["abc", "abc", "abc", "def"]

- tree(...) **[5.2.0.1+]**: Creates a treelist (same as the object method).

6.8. Map

A data structure of key-value pairs.

- **Creation**: MAP(key1;value1, key2;value2, ...)
- **Conversion from List**: list.map(keyExp, [valueExp])
- **Map Methods**:
  - sum(): Sums the numeric values in the map.
  - max(): Finds the max of the numeric values.
  - min(): Finds the min of the numeric values.
  - count(): Counts the values.
  - distinct(): Removes duplicate values from the value part.
  - table([keyHeader], [valueHeader]): Creates a table from the map.
  - get(key): Returns the value for a given key.

6.9. Special Values

- **MISSING**: Represents the absence of a value.
  - isMissing(): Returns true if the expression's value is MISSING.
  - whenMissing(defaultValue): Provides a default value if the expression's value is MISSING.
  - **Note**: While isMissing() exists, the preferred method for checking for a missing value is direct comparison:
    ```
    // Preferred method
    myKpi.responsible.first() = MISSING
    myKpi.responsible.first() != MISSING

    // Less preferred
    myKpi.responsible.first().isMissing()
    ```

- **Choice**: A special data type for system properties with a fixed set of choices (e.g., priority, statusClassification). Equality is checked against the choice name (e.g., this.object.statusClassification = RED). **Limitation**: In Reporter, you must compare the .name property of the choice to a string (e.g., this.object.statusClassification.name = "RED").
- Visibility: The visibility property returns the actual choice value for visibility settings, whereas .visible returns a boolean.

Choice Value	String Name
VISIBLE	VISIBLE
NOVISIBLE	NOVISIBLE
ADMINVISIBLEONLY	ADMINVISIBLEONLY
VISIBLEASPARENTONLY	VISIBLEASPARENTONLY

7. General Functions

These functions can be used without being chained to a data type instance.

- LIST(...): Creates a list.
  ```
  myList := LIST(1, 2, 3, 4, 5)
  ```

- md(markdownString): Converts a string containing Markdown into rich text / HTML. Supports ${} syntax for variables and expressions.
  ```
  md("Important Info: <span style=""color:blue"">Year 1</span>")
  ```

- num(value): Converts a value to a number.
  ```
  myString := "12345"
  myNumber := num(myString) // 12345
  ```

- output(expression): Gets the literal "formula" text for a Function or Expression without evaluating it.
  // Get the expression text from an Extended Expression with ID 'myExpression'
  output(t.myExpression.expression)

- priority(probability, consequence, [RiskSetting]) **[5.3.5.0+]**: Calculates and returns a Priority data type.
  // Priority value for top-right corner of a Risk Chart using default settings
  priority(5, 5) // "RED" with default settings

- str(value): Converts a value to a string.
  myNumber := 9000
  myString := str(myNumber) // "9000"

- **Numeric Functions**: abs(), cbrt(), ceil(), floor(), pow(), round(), sqrt().

8. Token Expressions

Token expressions use dot notation to dynamically access properties and methods from objects.

- **Syntax**: <object>.<propertyID>, <object>.<refPropertyID>.<propertyID>
- **Context Tokens**: this.object, this.organisation, this.user, this.start, this.end, this.bop, this.eop. The keyword self refers to the context object within table rows or calculate() functions.
- **List-returning Tokens**: children, descendants, organisations, scorecards, kpis, tasks, speedos, contextFiles, files, etc., return lists of objects.

9. Context

Context refers to the current environment (selected object, date, user).

- **Keywords**: this, self.
- **calculate() function**: Applies a specific context to an expression. It can be chained to an object to set the object context, or to a list to iterate over each item.
  // calculates the actual YTD value of the currently selected context object
  calculate(actual, BOY, EOP)

  // calculates the target for object with ID "IND_001" for the current time context
  t.IND_001.calculate(target)

  // returns a list of the responsible property value for each KPI in the list
  myKpis := LIST(t.KPI001, t.KPI002, t.KPI003)
  responsibleUsers := myKpis.calculate(responsible)

10. Querying

Extended supports querying to extract lists of objects.

- **Syntax**: SELECT <type(s)> [FROM <root(s)>] [WHERE <condition(s)>]
- **Keywords**:
  - SELECT: Specifies the object type(s) to retrieve.

    - FROM: (Optional) Specifies the root object(s) to search from. Defaults to the entire model.
    - WHERE: (Optional) Filters the results based on one or more conditions.
- **Performance Tips**:
    - Use the smallest possible root.
    - Place faster-evaluating and more restrictive conditions first.
    - Write the property as the first operand in comparisons (e.g., WHERE numberProperty = 5).
    - Prefer SELECT over multiple .filter() calls for initial data retrieval.
- **Model Roots**: The FROM clause can use specific model roots to narrow the search space, such as root.accessProfile, root.actionPlan, root.organisation, root.user, etc.
- **Examples**:

```
// Returns a list of all the KPIs in the model
SELECT Kpi

// Returns a list of all the Perspectives under the context Organisation or its children
SELECT Perspective FROM this.organisation

// Returns a list of all organisations with an orgType property of "Franchise"
SELECT Organisation WHERE orgType = "Franchise"

// Returns a list of Kpis under the context organisation that have a red status
// and the context user in their responsible property
SELECT Kpi FROM this.organisation
  WHERE responsible CONTAINS this.user
  AND statusClassification = RED
```

11. Tables

Extended supports creating and manipulating tables.

- **Creation**:
    - list.table([prop1], [prop2], ...): Creates a table from a list of objects.
    - createtable([header1], [header2], ...): Creates an empty table with specified column headers.
- **Table Methods**:
    - addColumn(header, expression): Adds a new column.

```
myKpis.table(id, name)
  .addColumn("Perspective", parent.parent.name)
```

    - addTimeColumns(value, periodType, startDate, endDate, columnName): Adds multiple columns for a range of time periods.

```
myKpis.table(name)
  .addTimeColumns(actual, M, BOY, EOP, this.bop.long)
```

    - addRow(obj, [val1], [val2], ...): Adds a new row. **Limitation**: Reusing the same context object in multiple rows is not supported.

```
myTable := createtable("ID", "Name", "Number of Children")
myTable.addRow(t.CORPORATE_FINANCE, id, name,
t.CORPORATE_FINANCE.children.size())
```

    - align(LEFT | RIGHT | CENTER): Sets alignment for a table, column, or row.

  - collapse(): Hides child rows by default.
  - decimals(n): Sets the number of decimal places.
  - formattype(type): Changes the formatting of numeric values (e.g., PERCENTAGE, THOUSANDS, DATE, DURATION).
  - hidden() **[5.3.1.0+]**: Hides a column by default.
    ```
    kpis.table(id, name)
       .addColumn(actual).hidden()
       .addColumn(budget).hidden()
    ```
  - indent(n): Indents a row.
  - postfix(text): Adds a postfix to values.
  - prefix(text): Adds a prefix to values.
  - readonly() **[5.1.8.0+]**: Makes a table, column, or row read-only.
  - style(style1, ...): Applies styles like bold, italics, wrapped, full, truncated, separator **[5.2.0.0+]**. wrapped, full, truncated are mutually exclusive.
  - url(link): Makes the values in a table or column clickable links. The link must be a URL string or an object reference.
    ```
    // Add column with link to an external site
    myTable.addColumn("External Link", "Sharepoint").url("https://www.sharepoint.com")

    // Add value column with link to an object
    myTable.addColumn("Actual", actual).url(t.CORPORATE_KPI001)
    ```

12. Transactional Methods

These methods cause changes in the system and can be run from the Extended Interface, Extended Action, or Extended Target.

- add(objectType, [prop1 := val1, ...]): Adds a new object. **Limitation**: Cannot use templates from TemplateCategory or Defaults.
  ```
  // Adds a Scorecard to the Organization with id 137
  o.137.add(Scorecard)
  // Adds a KPI and sets its name and responsible user
  t.6938.add(Kpi, name := "Revenue", responsible := u.ahernandez)
  ```

- affixLink(templateObject): Links a model object to a template object.
  ```
  t.8345.affixLink(t.432)
  ```

- change(prop1 := val1, ...): Changes properties of an object or list of objects.
  ```
  t.KP001.change(name := "Revenue")
  ```

- clear([prop1], ...): Clears property values or InputView variables.
  - **Clearing Object Properties**:
    ```
    myKpi.clear(description) // Clears the description property
    ```
  - **Clearing Input View Variables**: Can clear all variables or specific ones for session-based or object-based input views.

```
// Clears all session variables for the page
clear()
// Clears a specific session variable
clear(testStr)
// Clears all variables for the first input view on an object
this.object.inputView.clear()
// Clears a specific variable key for all input views on an object
this.object.inputViews.clear(testStr)
```

- copy(object, [prop1 := val1, ...]): Copies an object.
  t.870.copy(t.499, id := "MYNEWID")

- delete(): Deletes an object or list of objects.
  t.9683.delete()

- error(message): Triggers an error and aborts the script.
  ```
  IF kp.name = "Revenue" THEN
     error("Revenue Kpi was found")
  ENDIF
  ```

- generate([includeId]): Outputs the script to recreate an object.
- link(templateObject): Copies a template object, preserving the link.
  t.2389.link(t.500)

- move(destinationObject): Moves an object to a new parent. **Limitation**: Cannot move between models.
  t.KP001.move(t.KPI_FOLDER)

- moveAfter(destinationObject): Moves an object after another object.
  t.KP004.moveAfter(t.KP003)

- moveBefore(destinationObject): Moves an object before another object.
  t.KP003.moveBefore(t.KP004)

- notify(subject, [body], [recipients], [category]): Sends a system notification. If no recipients are specified, it defaults to the System Administrator.
  ```
  // Simple notification
  notify("Project Approval", "PROJ100 has been approved!", g.PPM, "Projects")

  // Notification with a URL link to a page
  subject := 'Initiative due date approaching'
  body := this.object.name + ' requires your attention. ' + t.myPageActions.url()
  recipient := this.object.responsible
  category := 'Due date approaching'
  notify(subject, body, recipient, category)
  ```

- reset([prop1], ...): Resets overridden properties of a linked object to their template values.
  t.67203.reset(name)

- sendmail(subject, body, recipients): Sends an email.
  sendmail("KPI Update", "Please enter comments", "bruce@wayneenterprises.com")

- start(): Executes a runnable object (e.g., Event Rule, Transformer Job).
  t.81037.start()

- unlink() **[5.1.10.0+]**: Removes the link from a linked object.
  t.5492.unlink()

13. Advanced Topics

13.1. Markdown Formatting

The md() method can be used to convert a string containing Markdown into rich text (HTML). This is useful for dynamic rendering in labels, descriptions, and extended properties.

- **Syntax**: md(<markdownString>)
- **Features**:
  - Supports ${} syntax for variables and expressions inside the string.
  - Can use inline HTML for styling (e.g., <span style="color:blue">).
- **Limitations**:
  - Header sizes (# H1) may render inconsistently in BMP and may require CSS adjustments.
  - Bullet lists (* Item) may not render correctly without CSS updates.
  - Strikethrough, emojis, and tables are not currently supported.
- **Recommendation**: Use for enhancing clarity. Avoid overuse in dense areas like tables. When using in an Extended Property, set the return type to NONE to ensure proper rendering.

13.2. Custom HTML Formatting

It is possible to inject custom HTML into fields with rich text capabilities (like Labels or Text Elements) to create customized and more visually appealing designs. This is an advanced, undocumented practice.

- **Advantages**:
  - **Advanced Styling**: Apply custom colors, backgrounds, and complex layouts.
  - **Dynamic Content**: Combine HTML with Extended logic to generate content dynamically.
  - **Enhanced UX**: Create more intuitive and engaging interfaces.
- **Disadvantages & Limitations**:
  - **Undocumented**: Not officially supported and may break in future updates.
  - **Maintenance**: Can be complex to read and maintain.
  - **Tag Stripping**: Some HTML tags, like <button>, will be stripped by the system.
  - **Styling Differences**: border-radius works on Labels but not on Text Elements.

Modular HTML Formatting

To manage complexity, a modular approach is recommended. This involves defining styles, building content blocks, and assembling them into a final string.

Step 1: Define Styling with Variables
Centralize CSS styles in variables for easy updates and consistency. A style is a simple string of CSS properties.

- **Formalized Examples**:

```
// Basic styling for a container
vContainerStyle := 'padding: 10px; border: 1px solid #e0e0e0; border-radius: 3px; background-color: #FFFFFF;'

// Styling for a header text
vHeaderStyle := 'font-size: large; font-weight: bold; color: #05164d;'

// Styling for standard paragraph text
vTextStyle := 'font-size: small; color: #000000;'

// Styling for list items
vListItemStyle := 'font-size: small; margin: 1px 0 1px 20px;'
```

Step 2: Build HTML Content with Extended Code
Construct HTML strings by concatenating static HTML tags with dynamic Extended code, such as object properties.

- **Example**:

```
// Create a title using the object's name
vTitleHtml := '<div style="' + vHeaderStyle + '">' + this.object.name + '</div>'

// Create a paragraph with the object's description
vDescriptionHtml := '<p style="' + vTextStyle + '">' + this.object.description.whenMissing('No description provided.') + '</p>'
```

Step 3: Use Conditional Logic (Optional)
Use IF-THEN-ELSE statements to conditionally include or exclude entire HTML blocks based on data.

- **Example**:

```
vStatusHtml := ''
IF this.object.statusClassification = RED THEN
  vStatusHtml := '<div style="font-weight: bold; color: red;">Status: Requires Attention</div>'
ELSE
  vStatusHtml := '<div style="color: green;">Status: On Track</div>'
ENDIF
```

Step 4: Construct the Final HTML String
Combine the individual HTML components into a single string. Use str() to ensure the final output is correctly interpreted as a string.

- **Example**:

```
vFinalHtml := '<div style="' + vContainerStyle + '">'
        + vTitleHtml
        + vDescriptionHtml
        + vStatusHtml
        + '</div>'
str(vFinalHtml)
```

Using genedit() for Text Elements

You can use the change() method (often found via .genedit()) to inject HTML into the text and longText properties of a Text Element.

```
// Define HTML content for the main view and the "show more" section
vMainText := '<div style=...><h6...>Information</h6>...</div>'
vShowMore := '<div style=...><p>The Information that you see here is very relevant</p></div>'

// Update the TextElement with the defined content
t.7561881.change(
  name := 'Information',
  text := vMainText,
  longText := vShowMore
)
```

Creating Reusable Style Functions

For consistency across multiple elements, create a reusable function (an Extended Expression) that defines a set of CSS style variables. This function can accept parameters to customize styles.

```
// Expression 'getStyle_Questionnaire_Info'
// Use input colors or apply defaults
vHeaderColor := vColorPrimary.whenMissing("#05164D")
vTextColor := vColorText.whenMissing("#000000")

// Define and output style variables
vStylePadding := "10px"
vStyleMainTitle := "color: " + vHeaderColor + "; font-size: large; border-bottom: 1px solid " +
vHeaderColor + ";"
// ... other styles
```

Then, call this expression to apply styles before building your HTML string:

```
// Set a custom color for this specific use case
vColorPrimary := "#F57C00"

// Call expression to define styles
{getStyle_Questionnaire_Info}

// Define HTML Content using the style variables
vString := "<div style=\"" + vStylePadding + "\">"
      + "<div style=\"" + vStyleMainTitle + "\">Identifying High-Risk AI Systems</div>"
      // ... more HTML
      + "</div>"
str(vString)
```

13.3. Bulk Reordering of Objects

Transactional methods like moveBefore can be used in loops to reorder multiple objects at once, which is difficult to do manually in Configuration Studio.

```
// Example: Move a specific dashboard to the top of the list for multiple orgs

// 1. Select the organisations to affect
myOrgs := SELECT Organisation FROM o.myOrgs

// 2. Identify the template object to be moved
item1 := t.riskDashboard

// 3. Loop through each organisation
myOrgs.forEach(myOrg:
  // 4. Find the destination (the first existing scorecard to move the item before)
  myDestinationItem := myOrg.children.filter(className = 'Scorecard').first()
  // 5. Find the actual object in the model that needs to be moved
  object := myOrg.children.filter(linkedTo = item1).first()
  // 6. Execute the move
  object.moveBefore(myDestinationItem)
)
```

14. Reference

14.1. Operators

Type	**Operators**
Numeric	+, -, *, /, %
Logical	AND, OR, NOT
Relational	>, <, >=, <=, =, !=, <>, CONTAINS, IN

14.2. ID Spaces

ID spaces define the application area of an object. The get() method can be used on an ID space to retrieve an object by its ID string (e.g., o.get("100")).

ID	**Scope**
ap	ACCESSPROFILE
C	CLASSCONFIG
d	DEFAULTS
g	GROUP
k	CUSTOM_PROPERTY

n	NODE
ndi	NODEDATAIMPORT
nt	NODETYPE
o	ORGANISATION
p	CUSTOM_PERIOD
r	EXTERNALRESOURCE
rt	ROOT
t	ACTIONPLAN, SHAREDWEB, TEMPLATECATEGORY
u	USER
ceven	CEVENDOR
cetas	CETASK
cecom	CECOMMENT
ceinc	CEINCIDENT
cepro	CEPROCEDURE
cepol	CEPOLICY
cecme	CECONTROLMEASURE
ceiss	CEISSUE
ceass	CEASSET
ceser	CESERVICE
cecot	CECONTRACT
ceprj	CEPROJECT
cereg	CEREGULATION
cecor	CECOMPLIANCEREQUIREMENT
ceind	CEINDICATOR
ceatt	CEATTACHMENT
ceras	CERISKASSESSMENT
acpol	ACCESSPOLICY
role	ROLE
ceprd	CEPRODUCT
sa	SERVICEACCOUNT

cepsc	CEPRESCREENING
ceprv	CEPRIVACY
cewfl	CEWORKFLOW
cedis	CEDISTRIBUTION
ceinq	CEINQUIRY
ceqst	CEQUESTIONNAIRE
cedpi	CEDPIA
cetia	CETIA
ceasa	CEASSURANCEACTIVITY
fas	FORMSANDSURVEYS