

User Manual

Aptean Food and Beverage ERP

Interface Management

Product Documentation



Contents

[Release Notes](#)

[2.0.53779.0](#)

[1.10.48950.0](#)

[1.9.46836.0](#)

[Introduction](#)

[Interface Management](#)

[Partner extensibility](#)

[Interface Management Helper functions](#)



Release Notes

Article • 4/3/2023 • 1 min read

To view what's new in each release of Interface Management extension, select the specific release version.

Date	Release Version	Description
July 2022	2.0.53779.0	Stability release
May 2022	1.10.48950.0	Stability release
April 2022	1.9.46836.0	Stability release



2.0.53779.0

Release Note • 4/3/2023 • 1 min read

Enhancements / Modifications

Added functionality to seamlessly migrate from the Interface Management app to the Integration Layer app.



1.10.48950.0

Release Note • 4/3/2023 • 1 min read

Enhancements / Modifications

Minor technical changes were made to keep our apps in line with the *updated* coding guidelines for Business Central.



1.9.46836.0

Release Note • 4/3/2023 • 1 min read

Enhancements / Modifications

Minor technical changes were made to keep our apps in line with the *updated* coding guidelines for Business Central.



Introduction

Article • 4/3/2023 • 1 min read

The Apteian Interface Management (IFM) extension connects installed Apteian apps making them able to interact with each other without any dependency. Of course, apps that use Interface management do have a dependency on this app.

By using this app it's possible to share data and open function usage between Apteian apps and hereby integrate food processes without dependency. Apps can be developed and maintained independently of each other. When an app isn't installed the interface with this app will simply not respond and not throw any errors.

As a result, customers are flexible in selecting Apteian apps they would like to use and install and/or upgrade them separately as long as there are no breaking changes.



Interface Management

Article • 4/3/2023 • 5 min read

Scenario

From the Aptean Mobile Warehouse app we need the possibility to register packaging transactions in case the customer uses the Aptean packaging app. The integration in this case, without creating dependency, is handled via the Aptean Interface Management app.

High level overview

Note

Interface Management only consist of generic (help) functions related to the interface process. All module specific code related to the interface process is part of the specific modules.

1. Packaging entries are created via scanning and saved into a temporary table available in the Aptean Mobile Warehouse app.
2. The packaging entries (only available / known by the Aptean Mobile Warehouse app) are packed (serialized to JSON via the Interface Management helper function) and send via generic functions available in the Interface Management app on basis of the interface contract. It's possible serialize to JSON automatically or define the schema (per field (type) + how to handle) yourself.
3. The package is received and unpacked (deserialized) in a temporary table (Only available/known by the Aptean Mobile Warehouse app) by the Packaging app, again using generic functions available in the Interface Management app on basis of the interface contract. When fields aren't present in the defined schema they will not be parsed i.e. skipped.
4. The App Integration Code triggers the defined procedure in the Packaging app, in this case register the Packaging entries on basis of the received record set in the actual Packaging entries table(only available / known by the Packaging app). When all goes well Packaging will set Handled to true. When errors occur in this final step it's necessary to also backport the messages to the Aptean Mobile Warehouse app. Again this is possible by using the generic functions available in the Interface Management app on basis of its specific interface contract.

Process in more detail

Interface Management will only respond to known Interface Implementations. The instances of subscribers are managed by a SingleInstance codeunit. Interface Mgt. Each individual app requires a pair of codeunits:



1. Contains the logic (EventSubscriberInstance = Manual)
2. To register itself as an implementation, to bind an instance of codeunit 1 and pass that instance to Interface Mgt. when required.

Flow

1. The Interface Management app knows all interface implementations by calling the interface implementations with a discovery event.
2. The Aptean Mobile Warehouse app asks Interface Management to provide available implementations.
3. The Aptean Mobile Warehouse app sets the following parameter values and asks Interface Management to raise the event:
4. Interface Code, PACInterfaceCode, Implementation Code, PACImplementationCode, App Integration Code, SendPackTransFunction.
5. Interface Management runs the codeunit set on registration of the interface implementation. This codeunit is responsible for binding an instance of the codeunit that contains the logic in the Aptean Mobile Warehouse app and passing that instance back to Interface Management which stores it in a VAR and keeps it in scope long enough to respond to the event in the following step
6. Interface Management calls the OnInterfaceEvent event with the VAR set above, the one bound codeunit in scope within the Packaging app responds to the event.
7. After reading the event parameters the Packaging app processes the received data on basis of the interface implementation procedure and if defined sets response data as requested (handled=true or false in case errors occur).
8. The Aptean Mobile Warehouse app gets and reads the response if requested.



Partner extensibility

Article • 4/3/2023 • 1 min read

For partners it's possible to add their own Interface Implementations via subscription to the Integration Events available in the Interface Management app. Via the OnRegisterInterface event it's possible to add an Interface Implementation and via the OnInterfaceEvent it's possible to define how the data/function should be handled.

Considerations:

1. Data should be representable as JSON.
2. There is no way to pass complex types with their state or global variables set.
3. Be alert on schema and type definition, the compiler is unable to check the field definition and relation because there's no dependency.
4. Remember that codeunit parameters are VAR which is useful in at least two ways:
 - The codeunit that is called can set values in the record and they will be passed back to the calling codeunit.
 - You can pass a set of records (temporary records or filtered set). The calling codeunit is then able to just REPEAT...UNTIL over the set rather than extracting the result from a string.



Interface Management Helper functions

Article • 4/3/2023 • 3 min read

App Integration Data FDW

SetIntegrationData

Procedure SetIntegrationData(Description: Text; Value: Variant)

This function is used to set information on Interface Management so that another module will pick it up later. It is used to send any filters or other information when attempting to call a function from another module.

GetIntegrationDataText

Procedure GetIntegrationDataText(Description: Text; Index: Integer): Text

This function is also present in versions for other data types. It is used to retrieve information stored in Interface Management, which enables modules to receive data from the module they've previously communicated with, like function returns.

Json Helper FDW

Serialize

Procedure Serialize(RecordVariant: Variant; FieldNameFilter: Text) JsonText: Text

This function serves as a generic serialization of a given record to JSON. If the desired JSON file needs to have all fields from the table, this is the function to use.

AddVariable

Procedure AddVariable(var JObject: JsonObject; VariableName: Text; VariableValue: Variant);

This function takes a JObject and adds a new field to the record being serialized. The VariableName is the name of the field and the VariableValue is its value.



AddRecord

Procedure AddRecord(var JObject: JsonObject; RecordVariant: Variant; FieldNameFilter: Text);

This function serializes a full record into a JObject. The containing fields can be filtered using the FieldNameFilter parameter so that only the fields who meet the filtering criteria get added to the JObject.

AddRecords

Procedure AddRecords(var JArray: JsonArray; RecordVariant: Variant; FieldNameFilter: Text);

This function adds to a JArray whatever records meet the filtering criteria set on the RecordVariant itself. Once again, the fields can be filtered using the FieldNameFilter.

RemoveNonAlphaNumericCharacters

Procedure RemoveNonAlphaNumericCharacters(OldValue: Text) NewValue: Text;

This function removed non-alphanumeric characters from a text. This is used to search for and write fields on a JSON because it's good practice to not use these characters in JSON fields.

GetRecord

Procedure GetRecord(JObject: JsonObject; var RecordVariant: Variant; FieldNameFilter: Text)

This function deserializes a single record, given a JObject containing the JSON token pertaining to that serialized record.

GetFieldValue

Procedure GetFieldValue(JObject: JsonObject; FieldName: Text): JsonValue

This function is used to get a certain field from a JObject using only its name to retrieve it. This function already removes the non-alphanumeric characters when searching for the field.

GetFieldValueAsObject

Procedure GetFieldValueAsObject(JObject: JsonObject; FieldName: Text): JsonObject



This function gets a JObject from inside another JObject. This is useful for nested JSON objects, in which the value for a field in the object is an object itself.

GetGenericJsonValue

Procedure GetGenericJsonValue(JObject: JsonObject; VariableName: Text): JsonValue

This function does the same as GetFieldValue without taking care of non-alphanumeric characters when searching for the field.

GetGenericJsonObject

Procedure GetGenericJsonObject(JObject: JsonObject; VariableName: Text): JsonObject

This function does the same as GetFieldValueAsObject without taking care of non-alphanumeric characters when searching for the field.

GetGenericJsonToken

Local procedure GetGenericJsonToken(JObject: JsonObject; VariableName: Text): JsonToken

This function does the same as the previous two functions but returns a JsonToken instead.

GetJsonValue

Procedure GetJsonValue(JObject: JsonObject; Name: Text; var TextValue: Text)

This function, which is present in many versions concerning different data types, will retrieve from the JObject a value from the field with the given Name. This function doesn't account for non-alphanumeric characters.

Deserialize

Procedure Deserialize(JsonText: Text; RecordVariant: Variant; FieldNameFilter: Text)

This function serves to fully deserialize a JSON text into a set of records from a given table. The function will attempt to insert the records upon deserializing them from the JSON text.

