Infor ES InContext Modeler Development Guide

Release 10.7.x

Publication Information

Release: Infor LN 10.7.x
Publication Date: January 21, 2019
Document code: ln_10.7.x_lndevincontextdg__en-us

Contents

About this guide

This guide describes the concept of InContext messages and the way to generate and adapt InContext models.

Intended audience

This document is intended for Infor LN developers and technical consultants.

Related documents

You can find the documents in the product documentation section of the Infor Support Portal, as described in "Contacting Infor".

- *Infor LN Studio Application Development Guide*

Contacting Infor

If you have questions about Infor products, go to Infor Concierge at https://concierge.infor.com/ and create a support incident.

If we update this document after the product release, we will post the new version on the Infor Support Portal. To access documentation, select **Search > Browse Documentation**. We recommend that you check this portal periodically for updated documentation.

If you have comments about Infor documentation, contact documentation@infor.com.

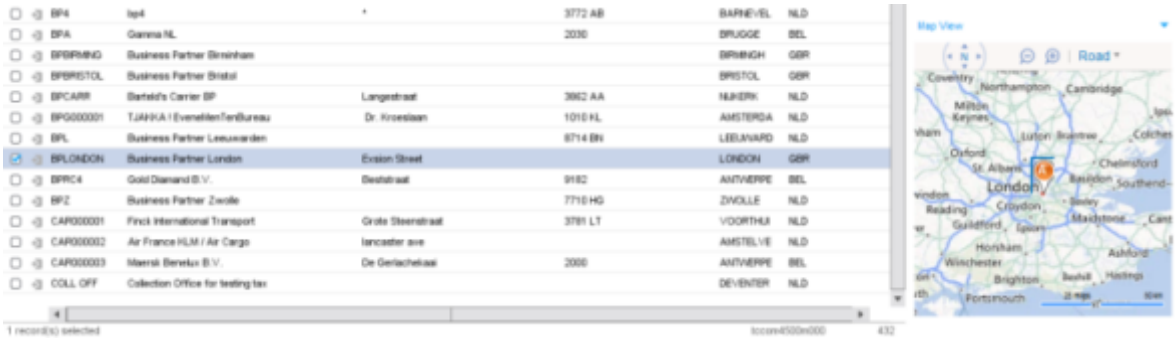
Chapter 1: Introduction

With the InContext Modeler you can create integration between Infor LN and several Infor Ming.le web parts. This integration is based on so-called context messages. In those context messages the characteristics of the Infor LN objects, which have focus in the UI, are sent to the web parts. Those web parts have functionality to display the data that is related to the Infor LN objects.

Integration examples:

- An address of a Business Partner (BP). When you look up this BP, the Map web part will show his location.
- A Sales order can have many documents attached. This also applies to the BP for which the Sales Order applies. The Content Assistant web part shows both the documents attached to the Sales Order and the BP.

This screenshot shows an example of the integration between an LN session and the Map web part:



Different web parts may be interested in different context message types. The Map web part, for example, is interested in a message that contains the address information. A Package Tracker web part is interested in a tracking number. The Content Assistant web part is interested in the id-fields of the current object.

The InContext Modeler is used to specify which context messages must be sent by a session and how the content of those messages must be filled. No changes are required in the source code of the standard session. This implies that you even can define your own context message types for own developed web parts and get this running without the need of customizing the standard sessions.

This guide will help you to use the InContext Modeler for making (small) changes to existing models, but also for generating completely new models for your own developed tables and sessions. The overall development procedure is described in Development procedure on page 13.

In the development process for InContext messages, there are two models. The first one is the InContext Reference Model (IRM). This model contains the relations between tables and context message types.

This model is generated based on the tables in the LN Data Dictionary and the defined context message types. For more information on the IRM, see InContext Reference Model (IRM) on page 8. The second model is the Incontext Implementation Model (IIM). It contains models for tables and sessions and generated 3GL libraries for the sessions. How the IIM can be generated is described in Incontext Implementation Model on page 9.

After the IIM is generated, the generated models in the IIM can be changed. Changes can be made by editing the models in Infor LN Studio. After regeneration of the InContext Library, the changes are active. The process of changing the models can be found in Model editing on page 17.

The runtime architecture is described in Runtime architecture.

Note:

Infor LN 10.2.1 has implemented a standard InContext model. It sends the “inforBusinessContext” context message, which is picked up by the Content Assistant web part. Future releases of Infor LN will contain additional context messages.

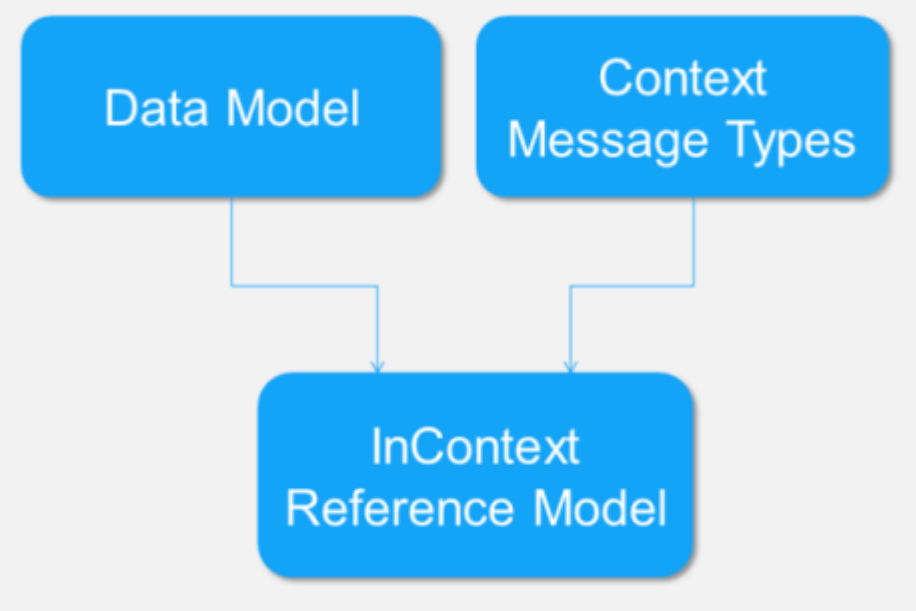
Infor Ming.le 10.2 or later is required to run the Content Assistant web part.

Chapter 2: InContext Reference Model (IRM)

The IRM is a model of tables and references that can be generated from the existing table definitions and other data. The table definitions contain the so-called hard references between tables. They are stored into a new simple accessible model.

In this model also a basic link between tables and context messages is registered. This is based on a configuration file that describes the available context messages and field types (domains) which are connected to certain context messages. For example the domain that describes the latitude of a GPS location will lead to a map integration. For more information about this file see Context message type definition on page 28.

An IRM covers an Application (a set of Packages that belong together and follow the same versioning). The PMC Base VRC is used as the notion of this set of packages. Multiple model versions can be generated, stored, compared and used to generate the InContext Implementation Model (IIM).



Chapter 3: InContext Implementation Model (IIM)

The IIM contains the link between the software components (tables and sessions) and the context messages that must be generated. Per table the applicable context message types are registered and which fields must be used to construct the message. Sessions by default inherit the implementation of the main table and this can be extended at session level. Also sessions without a main table can send context messages.

This model can be generated initially from the IRM. Modifications are made on table level and session level in Infor LN Studio. In the IIM hooks are available to influence the default behavior.

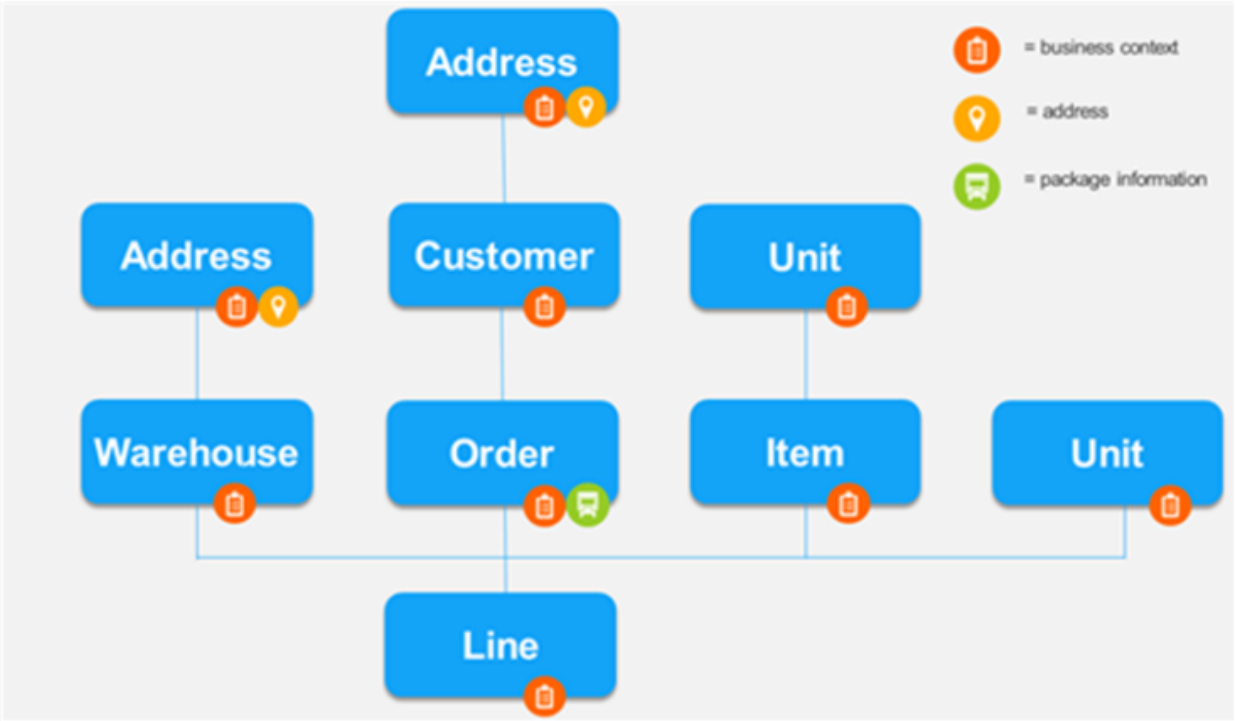
Initial generation

The initial generation can be done in these ways:

- Generation with full inheritance. All applicable context messages will be generated for a table and inherited by the tables that have a reference to the current table. Afterward the not wanted references and context messages can be hidden in the model editing step. Use this method only for applications with small data models; otherwise the models are unmanageable.
- Generation without inheritance. The table models will get the context messages that are applicable for the current table. The references to other tables are generated as “hidden”, and there is no inheritance of the context messages of referenced tables. When editing the model the context messages to inherit can be added by unhiding the table reference to the table of which the context messages must be inherited.

Inheritance of context messages

The diagram contains an example data model and context message types to explain the inheritance of context messages:



If the model is generated with full inheritance, the context messages on Line level listed in this table can occur:

Entity	Context Messages
Line	Business Context
Warehouse	Business Context
Address (1)	Business Context, Address
Order	Business Context, Package Information
Customer	Business Context
Address (2)	Business Context, Address
Item	Business Context
Unit (1)	Business Context
Unit (2)	Business Context

When looking at Line, the Business Context message is sent for nine entities, the Address message for two entities and the Package Information for one entity. This makes the information in the web parts too superfluous to be of any help to the end user.

There are two ways to reduce the number of context messages:

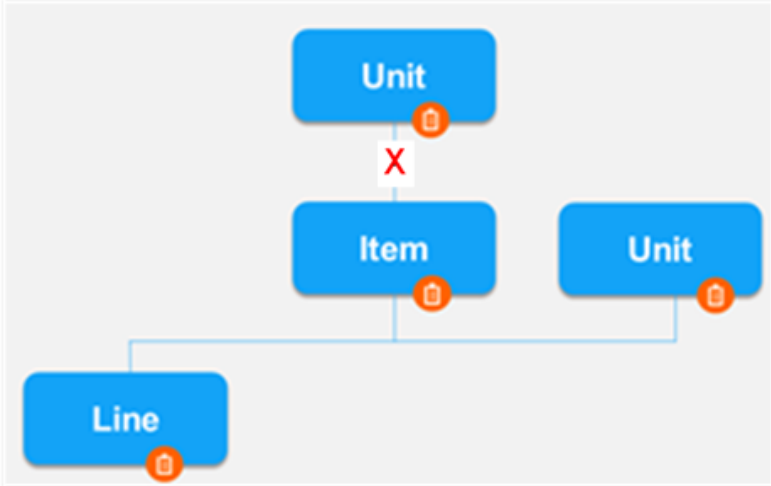
1. Bottom up.
2. Top down.

Bottom up reduction

If the implementation model was generated without inheritance, this method is chosen by definition. No context messages are inherited from the referenced tables. To explain the bottom up reduction we assume the model was generated with full inheritance.

Use the bottom up reduction when you do not want to inherit anything from a referenced table and none of the tables that refer to the current table needs a context message of a referenced table.

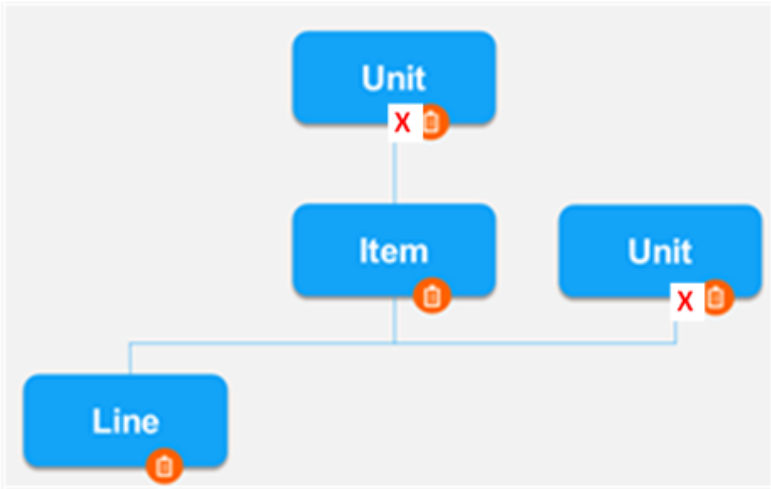
In this example the reference from Item to Unit is changed to "hidden". The Item does not inherit the messages related to Unit (on the top), the Line will not inherit it either. The Line will still inherit the Unit that is on the right side of the diagram.



Top down reduction

If a referenced table is not hidden, the context messages of the referenced table are inherited only if these context messages are not hidden for the children. Top down reduction can be used to hide context messages even if the reference to the table is not hidden.

In this example the context message for the Unit is hidden for all children. With one change the inheritance of this context message is stopped.



Code generation

Many events within Infor LN will result in sending context messages to the Infor Ming.le web parts. To reduce the performance impact the IIM is compiled to object code. For each session that can send context messages in certain events (selection of a record, change of a field, getting focus) a library is generated. This library can quickly compose the correct context message at runtime, without reading the model from the database. Those generated libraries will be part of the standard product, and can be regenerated by customers if they change the IIM.

The library generation uses this information to compose the libraries:

- Session model context messages and mappings, including the inherited ones
- Session model hooks
- Main table model generic hooks (if session has a main table)
- Referenced tables generic hooks (only for tables of which context messages are inherited)
- Message template from the Context Messages Configuration file

The code of the generated library is the same as the session code, suffixed by "`i`". For example, for session `tcmcs0145m000` the InContext Library is `tcmcs0145m000i`.

Chapter 4: Development procedure

The development procedure depends on what you want to achieve. These scenarios are described:

- Building an initial model.
- Rebuilding after Reference Model changes.
- Changing a table or session model.

Building an initial model

Run these procedures to build an initial model:

1. Creating Context Message Types configuration file.
2. Creating an InContext Reference Model.
3. Generating Implementation Model.

Note: An initial model is created already for Infor LN 10.2.1.

Creating context message types configuration file

To create a context message types configuration file:

1. Start session **Additional Files (ttadv2570m000)**.
2. Change your Current Package VRC to the VRC of the “ta” package that is configured in your Package Combination.
3. Create an Additional File with this information:
   - Package: “ta”
   - Module: “gen”
   - Additional File: “contextmessages.xml”
4. Alternatively, copy the already existing Additional File to your own VRC.
5. Add the context message type definitions to the file and save the file.

   For the format of this file and examples, see Context message type definitions on page 28.

Creating an InContext Reference Model (IRM)

To create an IRM:

1. Select **Tools > Application Development > InContext Modeling**.
2. Start session **InContext Reference Models (tticm1500m000)**
3. Add a new record in this session.

   For more information on the fields in this session click the session's online help.

   Note: A PMC Base VRC is used to identify all packages and versions that belong together.

   The generation of the Reference Model processes the table definitions of the Export VRC defined for this Base VRC. See the PMC Distributor chapters in the *Infor Enterprise Server - Administration Guide*.
4. Select the **(Re)generate Reference Model** option.
5. In session **(Re)generate Reference Model (tticm1200m000)** specify the range of tables which you want include in the reference model.

   Note: Not until all tables are included in the reference model, the reference model is not valid and you can't generate the implementation model.
6. Select the **Validate Reference Model** checkbox.

   This option validates whether the reference model is complete.

Generating InContext Implementation Model (IIM)

To generate an IIM:

1. Select **Tools > Application Development > InContext Modeling**.
2. Start session **InContext Reference Models (tticm1500m000)**.
3. Select the **(Re)generate Implementation Model** option.
4. In session **(Re)generate Implementation Model (tticm1210m000)** specify the ranges and options for generation.

   A report shows the results of the generation step. For more information on the fields in this session see the session's online help.

Rebuilding after reference model changes

Run these procedures to rebuild the InContext Implementation Model after the Reference Model was changed:

1. Creating a new IRM.
2. Comparing the models.
3. Regenerating the Implementation Model.

Creating a new reference model

See Creating an InContext Reference Model (IRM) on page 14.

Comparing the models

To compare the models:

1. Select the previous model and new model generated earlier.
2. Click **Compare Reference Models**.

Regenerating the IIM

To regenerate the implementation model:

1. Select the **(Re)generate Implementation Model** option.
2. In session "**(Re)generate Implementation Model" (tticm1210m000)** specify the ranges and options for generation.

   For more information on the fields in this session see the session's online help and/or InContext Implementation Model (IIM) on page 9

   To reduce the elapsed time of this session you can run it multiple times for small ranges, based on the differences that were found in the previous step.

Changing a table or session model

Run these procedures to change the Implementation Model for a table or session:

1. Editing the model in Infor LN Studio.
2. Refactor the dependent models

Editing the model in Infor LN Studio

Note: see the *Infor LN Studio Application Development Guide* for more information.

To edit the model:

1. Start Infor LN Studio.
2. Specify the Reference Model in the Application properties window
3. Create a new Activity

4. Open this Activity in the Activity Explorer
5. Retrieve the model from the Infor ES server.

   Note:

   InContext models are stored in the Infor ES data dictionary as Additional Files. For table models the Additional File has a name that starts with the table name and has an extension "tabicim", for example tcibd001.tabicim. For session models the Additional File has a name that starts with the session name and has an extension "sessicim", for example tcibd0501m000.sessicim.

   The InContext models can also be retrieved through a hyperlink in the Table Editor and Session Editor.
6. Checkout the model.
7. Change the Model with the Table Model Editor or Session Model Editor.
8. Session models only: Regenerate the InContext Library.

Refactor the dependent models

This step applies to table models only.

To refactor the models:

1. Click the Dependent Context Models hyperlink in the Table Model Editor.
2. The Dependent Context Models View shows the found models.
3. Checkout those models.
4. Refresh those models.
5. Regenerate the InContext Libraries for those models
6. Check in those models.

Chapter 5: Model editing

Table and Session Models can be updated with Infor LN Studio. The generation of the InContext libraries can also be done from LN Studio. Use the model editors in case you want to change one of these parts of a model:

- Table References (Table models only)
- Drillback session (Table models only)
- Generic hooks (Include hook, Declaration hook, Function hook)
- Context Messages
- Mappings
- Context Message hooks (Condition hook, Before Mappings hook, After Mappings hook)

After changing a table model you must refactor the dependent models.

For more information about the procedures in LN Studio see the *Infor LN Studio Application Development Guide*

Table references

With the Table InContext Model editor you can add, remove, change, hide and unhide references to other tables. Those Table References result in extra entities that are considered in the context messages to be sent.

Adding a table reference

To add a table reference:

1. Add a table in the Referenced Tables grid.

   Note that this table must have an InContext model itself. If it does not have a model, click the Table model icon to create this model.
2. Add the field mapping to link the referenced table to the current table. These are the possible reference types:
   - **Refers**: the generated query uses the "refers to" clause to read the referenced data.

- **Where**: the generated query uses the "and" clause to read the referenced data. Use this type only if you are sure the referenced data exists; if the referenced data does not exist, no data will be read and the context message may be empty.
- **Nested Query**: a nested "select" statement is generated. Use this type to read referenced data if the reference data may not exist.
- **Hook**: write your own code in the TableRead hook. Use this type if you cannot read the referenced data with field values you have already available. Example: You need to read data from another table that has the same key as the current table, but in that key also a constant is present, which defines the object type. For example, you have sales orders and warehouse orders, but the warehouse order key contains also the order type.

  The hook must contain a complete SQL statement that retrieves the data of the referenced table. If this referenced table has other references itself, those references are processed automatically based on their reference types.

  Example of a TableRead hook:

```
select whinh001.*
from   whinh001
where  whinh001.order = :tdsls432.order
and    whinh001.type = whtype.sales
as set with 1 rows
selectdo
       break
endselect
```

3 Refresh the Context Messages.

Removing a table reference

To remove a table reference:

1 Select the table reference in the grid.

2 Right-click and remove the row.

3 Refresh the Context Messages.

Note: You can only remove a manually added table reference. Table references that come from the InContext Reference Model cannot be removed. Use the "hide" function instead.

Changing a table reference

To change a table reference:

1 Select the table reference in the grid.

2 Change the Reference Type, Mandatory flag or the field mappings.

For more information on the Reference Type, see "Adding a Table Reference" on page 17

3 Refresh the Context Messages.

Hiding or unhide a table reference

To hide on unhide a table reference:

1. Select or clear the Hidden checkbox.
2. Refresh the Context Messages.

Drillback session

Context messages can contain drillback URLs. When a web part receives a context message with a drillback URL, this URL can be displayed in the web part. When the user clicks this URL, the session mentioned in the URL is started in Infor LN.

For each table this drillback session must be specified in the model. It must be a session that has the current table as a main table.

The session code, mode and index can be specified directly or, as of version ES 10.3, defined by hooks. To add a new hook click the Hook link in the editor. To delete a hook clear the hook code.

A drillback can have these hooks:

- Session hook
- Mode hook
- Index hook

Session hook

Use this hook to specify the session code. This hook must return a String.

Example:

```
if tdpcg030.maty <> tdpcg.maty.sobook then
return ("tdpcg0130m010")
else
return ("tdpcg0130m020")
endif
```

Mode hook

Use this hook to select the mode to open the session.

This hook must return: SINGLE_OCC or MULTI_OCC.

Example:

```
return (SINGLE_OCC)
```

Index hook

Use this hook to select the index the session starts with.

This hook must return a Long.

Example:

```
if tdpcg030.maty <> tdpcg.maty.sobook then
return (1)
else
return (2)
endif
```

If the session code is specified using a hook, the index must also be specified by a hook. If it is left empty default 1 is used.

Generic hooks

An InContext model has these generic hooks:

- Include hook
- Declaration hook
- Function hook

Use these hooks to specify 3GL code, which is needed in the defined mapping hooks for the context message mappings.

Note: The generic hooks for the current table and the ones of the referenced tables are concatenated during library generation. Ensure that declaration of variables makes them unique for each table model. The same applies to functions in the Function hook. Make them unique as well. Prefix each variable and function with a string that at least includes the table code. For example "`tm.tdsls400.my.own.variable`" or "`tm.tdsls400.calculate.diff.amnt()`". Session models also have the generic hooks.

Include hook

Use this hook to include standard functions or links to libraries, which will be used in the mappings of the context messages.

Example:

```
#include "itcmcs2000"
#pragma used dll otdslsdll0001
```

Declaration hook

Use this hook to declare additional tables and variables, which will be used in the mappings of the context messages. Note that all tables that are in the referenced tables list are implicitly declared.

Example:

```
table ttisfc001 |* Production Orders
domain tcamnt tm.ticst001.amount
domain tcitem tm.ticst001.item
```

Function hook

Use this hook to code functions, which are used in the mappings of the context messages. Note that functions that are used in a direct mapping of type "Function" must return a value. Functions used in hooks such as the Condition Hook can be of type "void".

Example:

```
function tm.tisfc010.read.production.order(domain tcpdno i.pdno)
{
select tisfc001.*
from   tisfc001
where  tisfc001.pdno = :i.pdno
as set with 1 rows
selectdo
       break
endselect
}
```

```
function string tm.tisfc010.get.item.desc(domain tcitem i.item)
{
select tcibd001.dsca
from   tcibd001
where  tcibd001.item = :i.item
as set with 1 rows
selectdo
       break
selectempty
       tcibd001.dsca = "???"
endselect
```

```
return(tcibd001.dsca)
}
```

Context messages

With the Table InContext Model editor and Session you can add, remove, change, hide and unhide context messages. By default all unhidden messages of the referenced tables are inherited. Changing of Context Messages is handled below in the paragraphs about Mapping and Context Message hooks.

Add a context message

To add a Context Message:

1. Add a context message in the Context Messages grid. Note that you can add only context message types that have been defined in the Context Messages Configuration File. In a table model you can add a context message type only once for the current table. In session models you can add the same type multiple times.
2. Complete the mappings. If needed add one or more context message hooks. See below.

Remove a context message

To remove a Context Message:

1. Select the context message in the grid.
2. Right-click and remove the row.

Note: You can only remove a context message that has been added manually. Context Messages that come from the InContext Reference Model cannot be removed. Use the “hide” function instead.

Hiding / unhiding a context message

To hide or unhide a Context Message in a table model:

Change the value in the Hidden column. You can use these values:

- No: The context message is not hidden. It will be shown for the current table and will also be available for table and session models that have a reference to the current table.
- Current level: The context message will not be shown for the current table, but is available for table models that have a reference to the current table.
- Child level: The context message will be shown for the current table, but is not available for table models that have a reference to the current table.

- All: The context message will not be shown for the current table, and is not available for table models that have a reference to the current table.

For Context Messages in a session model, select/clear the Hidden checkbox to hide/unhide.

Mappings

The Mappings are used to map real values to the placeholders in the Template of the context messages. Mappings cannot be added or removed, the only action you can do on mappings is to change the mapped values. You can use these types of mappings:

- Field
- Hook
- Function
- None

Mapping type field

Use this mapping type if you want to map the placeholder to a field of the current table. You can also use fields that are read in the Before Mappings Hook. Moreover, you can specify constants; note that constants must be strings, including the quotes.

Examples:

Target	Source
ic_tablename	“tisfc001”
ic_carrier	“UPS”
ic_phone	“31342428888”
ic_primekey	tiipd001.item
ic_name	tm.tisfc001.customer.name

Mapping type hook

If you select this mapping type, you can create the source code in the hook editor. This hook must return a string value.

Example:

```
return(tisfc001.pdno & "/" & strip$(tisfc001.mitm))
```

Mapping type function

Use this mapping type if you want the get the result value of a function directly. You can pass arguments to those functions.

Examples:

Target	Source
ic_company	get.compnr()
ic_screenid	icm.get.screenid()
Desc	get.item.description(tisfc001.mitm) **Note:** add this function in the Function hook, or if this function is included in a DLL, you must link this DLL with a “#pragma used dll” in the Include hook.

For the Infor Business Context message a helper functions are available:

Function	Description
icm.getScreenId()	Function returns the screen id of the current session. This is “ln.<session code>”
icm.getLogicalId()	Function returns the logical ID of the environment as used in ION
icm.getContextId	Function returns a unique identification for the context message, used for debugging.
icm.getInternalAccountingEnti-ty(<tablecode>)	Function returns the physical company of the passed table, preceded by "infor.ln.".
icm.getReadOnly("tablecode")	Function returns “true” if the passed table is the maintable of the session, otherwise it returns “false”.
icm.getDrillbackURL("tablecode")	Function returns the drillback url of the passed table.

Mapping type none

Use this mapping type if the element must not be created in the context message.

Context message hooks

A Context Message can have these hooks:

- Condition hook

- Before mappings hook
- After mappings hook

Note: The context messages belonging to the current model have their own implementation of those hooks. For inherited context messages also the hooks are inherited. However, you can override those hooks in the current model.

Condition hook

Use this hook to send a context message depending on a certain condition. This hook must return a Boolean.

Example

```
return(tisfc001.plid <> tisfc001.sfpl)
```

When the Planner and Shopfloor Planner are equal, this context message is not sent.

Before mappings hook

Use this hook to read additional data to construct the context message. The same can be achieved by using the Mapping Type "Function". In a Before Mappings hook you can read multiple values in one go or write more complex conditions. You cannot return a value.

Example

```
on case tcibd001.kitm
case tckitm.purchase:
        tm.tcibd001.description = "Purchased Item: " & tcibd001.desc
        tm.tcibd001.readonly = "false"
        break
case tckitm.manufacture:
        tm.tcibd001.description = "Manufactured Item: " & tcibd001.desc
        tm.tcibd001.readonly = "false"
        break
default:
        tm.tcibd001.description = tcibd001.desc
        tm.tcibd001.readonly = "true"
        break
endcase
```

This hook determines multiple values that can be used in the mappings.

After mappings hook

Use this hook to modify the generated message after the mapping has been done. This may be necessary if you need to deviate your message from the message that is generated based on the Template. For more information about the template see Template on page 32 .

In this hook, the variable 'i.message.node' is available, which is the XML node that contains the built up message. Note that if there is a repeating part in the message, the After Mappings hook is called after building up an occurrence in the repeating part.

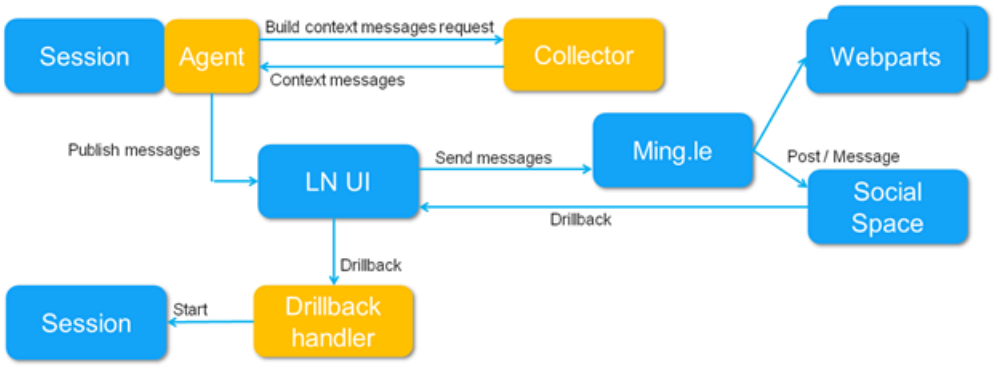
In this example a modeled drillback session, which is placed in the drillback URL by function icm.getDrillbackURL, is replaced by another session in a special case.

```
long   ret
long   enums
long   enum.node
long   drillback.node
string drillback(500)
string drillback.session(20)

if tdpcg030.maty <> tdpcg.maty.sobook then
      return
endif

enums = xmlFindMatch("?<drillbackURL>", i.message.node)
if enums = 0 then
      return
endif
enum.node = xmlGetFirstChild(enums)
while enum.node <> 0
      drillback.node = lval(xmlData$(enum.node))
      drillback = xmlData$(drillback.node)
      drillback = str.replace$( drillback,
                               "Session=tdpcg0130m010",
                               "Session=tdpcg0130m020")
      ret = xmlRewriteDataElement(
                                xmlGetParent(drillback.node),
                                "drillbackURL",
                                drillback)
      enum.node = xmlGetRightSibling(enum.node)
endwhile

ret = xmlDelete(enums)
```

This hook determines multiple values that can be used in the mappings.

Chapter 6: Runtime architecture

The diagram shows the runtime architecture for sending context messages and handling drillbacks from other Infor Ming.le parts, such as SocialSpace:



The Agent, Collector and Drillback Handler are components, for which the code is generated in the InContext Library.

Agent
This part handles the events that occur in the session. For example, if a record is selected the Agent receives a signal from the session. The Agent communicates with the Collector to receive the inContext message.

Collector
The Collector creates the XML that must be send to Infor Ming.le. Note that the real message that is sent to Workspace is in JSON format. The conversion is done by LN UI.

Drillback Handler
A context message can contain a drillback URL. The function `icm.getDrillbackURL()` constructs the URL that can start a session with the correct filter. The Drillback handler processes the URL query and activates the session with the correct data.

Debugging

You can among other libraries, debug the generated InContext Libraries with Infor LN Studio. The restriction is that you must run the session in Infor Ming.le, otherwise the InContext Library functions

are not called. To debug a session in Infor Ming.le, you must select **Debug and Profile 4GL** in the **Options** menu of LN UI. Infor Ming.le and LN must be running on your client.

Logging

Set the environment variable ICM_LOG_LEVEL to trace the runtime building up of context messages. These levels are available:

Level	Description
1	Errors are logged. This is the default if the variable is not set.
2	Warnings are logged
3	Information is logged
4	Debug logging. This is the most extensive logging and results in large log files,

The log files are created in `$BSE/log`. The filename is: `<username>_icm.log`.

To activate the logging, select **Active trace mode** from the **Options** menu in LN UI. The sent messages are displayed in the Java console.

Context message type definitions

XSD

The file with context message type definitions (the Additional File tagencontextmessages.xml) must meet this schema:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDe
fault="qualified">
    <xs:element name="ContextMessages">
      <xs:complexType>
          <xs:sequence>
            <xs:element name="ContextMessage" type="ContextMessageType"
minOccurs="0" maxOccurs="unbounded"/>
          </xs:sequence>
       </xs:complexType>
    </xs:element>
      <xs:complexType name="ContextMessageType">
          <xs:sequence>
            <xs:element ref="MessageIdentifier"/>
```

```
            <xs:element ref="MaxSelect"/>
           <xs:element name="Template" type="TemplateType" minOccurs="2"
 maxOccurs="2"/>
            <xs:element name="AppliesTo" type="AppliesToType"/>
          <xs:element name="Mappings" type="MappingsType" minOccurs="0"/>

            <xs:element ref="IncludeHook" minOccurs="0"/>
            <xs:element ref="DeclarationHook" minOccurs="0"/>
            <xs:element ref="FunctionHook" minOccurs="0"/>
            <xs:element ref="ConditionHook" minOccurs="0"/>
            <xs:element ref="BeforeMappingHook" minOccurs="0"/>
            <xs:element ref="AfterMappingHook" minOccurs="0"/>
          </xs:sequence>
          <xs:attribute name="Id" type="xs:string" use="required"/>
          <xs:attribute name="name" type="xs:string" use="required"/>
      </xs:complexType>
      <xs:complexType name="TemplateType">
          <xs:sequence>
            <xs:element name="_Block_" type="_Block_Type" minOccurs="1"
maxOccurs="2"/>
          </xs:sequence>
          <xs:attribute name="condition" use="required">
              <xs:simpleType>
                 <xs:restriction base="xs:string">
                    <xs:enumeration value="gen:data"/>
                    <xs:enumeration value="gen:empty"/>
                 </xs:restriction>
              </xs:simpleType>
          </xs:attribute>
      </xs:complexType>
      <xs:complexType name="_Block_Type">
          <xs:sequence>
             <xs:any minOccurs="0"/>
          </xs:sequence>
          <xs:attribute name="condition">
              <xs:simpleType>
                 <xs:restriction base="xs:string">
                    <xs:enumeration value="gen:noRepeat"/>
                 </xs:restriction>
              </xs:simpleType>
          </xs:attribute>
      </xs:complexType>
      <xs:complexType name="AppliesToType">
          <xs:sequence>
            <xs:element name="Table" type="TableType" maxOccurs="unbound
ed"/>
          </xs:sequence>
      </xs:complexType>
      <xs:complexType name="TableType">
          <xs:sequence>
            <xs:element ref="Filter" minOccurs="0"/>
            <xs:element name="Domain" type="DomainType" minOccurs="0"/>
          </xs:sequence>
      </xs:complexType>
      <xs:complexType name="DomainType">
          <xs:sequence>
```

```
            <xs:element ref="Filter" minOccurs="0"/>
            <xs:element name="TableField" type="TableFieldType" minOc
curs="0"/>
          </xs:sequence>
      </xs:complexType>
      <xs:complexType name="TableFieldType">
          <xs:sequence>
            <xs:element ref="Filter" minOccurs="0"/>
          </xs:sequence>
      </xs:complexType>
      <xs:complexType name="MappingsType">
          <xs:sequence>
            <xs:element name="Mapping" type="MappingType" minOccurs="0"
maxOccurs="unbounded"/>
          </xs:sequence>
      </xs:complexType>
      <xs:complexType name="MappingType">
          <xs:sequence>
            <xs:element ref="Source"/>
            <xs:element ref="Target"/>
          </xs:sequence>
          <xs:attribute name="type" use="required">
              <xs:simpleType>
                 <xs:restriction base="xs:string">
                    <xs:enumeration value="domain"/>
                    <xs:enumeration value="field"/>
                    <xs:enumeration value="function"/>
                    <xs:enumeration value="hook"/>
                 </xs:restriction>
              </xs:simpleType>
          </xs:attribute>
      </xs:complexType>
    <xs:element name="AfterMappingHook" type="xs:string"/>
    <xs:element name="BeforeMappingHook" type="xs:string"/>
    <xs:element name="ConditionHook" type="xs:string"/>
    <xs:element name="DeclarationHook" type="xs:string"/>
    <xs:element name="Filter" type="xs:string"/>
    <xs:element name="FunctionHook" type="xs:string"/>
    <xs:element name="IncludeHook" type="xs:string"/>
    <xs:element name="MaxSelect" type="xs:int"/>
    <xs:element name="MessageIdentifier" type="xs:string"/>
    <xs:element name="Source" type="xs:string"/>
    <xs:element name="Target" type="xs:string"/>
</xs:schema>
```

Context messages

The highest level element in the context message type definitions is <ContextMessages>. It can have ContextMessage sub elements only.

Example:

```
<ContextMessages>
   <ContextMessage Id="org.company.map.message" name="Map">
            <...>
   </ContextMessage>
   <ContextMessage Id="org.company.uom" name="Units">
            <...>
   </ContextMessage>
</ContextMessages>
```

For each context message that is supported, a ContextMessage element must be present. A ContextMessage element has these attributes:

Attribute	Description
Id	Identification of the message to make it unique during modeling. It is recommended to start it with your company's namespace, for example "com.mycompany" or "ge.myfirma".
Name	The description of this message type.

A ContextMessage element has these sub elements:

Element	Description
MessageIdentifier	The identification of the message during runtime. The web parts subscribe to this identifier.
MaxSelect	The maximum number of selected records that can be processed for this context message. The value to specify here depends on the data the web part can process and whether it makes sense to see the context of multiple selected records. Example: For a web part that shows linked documents of a selected object, it makes no sense to show documents of multiple selected objects. For a web part that shows a location on a map, it can be handy to show the locations of all selected objects.
Template	The template for the XML that forms the runtime context message. These template elements must be present: • The template for a filled message. • The template for an empty message. This message is sent to clear the web part. For the content of the Template element, see below.
AppliesTo	This element describes for which tables, domains and/or table fields the message applies. This is used during the generation of the InContext Reference Model to link the context message types to tables. For the content of the AppliesTo element, see below.

Element	Description
Mappings	This element describes how the placeholders in the templates must be filled with real values when the context message is built up at run-time.
IncludeHook, DeclarationHook, FunctionHook, ConditionHook, BeforeMappingHook, AfterMappingHook	Those hooks contain the source code that must be included in the corresponding hooks of the context messages in the generated models.

Example:

```
<ContextMessage Id="org.company.map.message" name="Map">
   <MessageIdentifier>companyMapMessage</MessageIdentifier>
   <MaxSelect>1</MaxSelect>
   <Template condition=”gen:data”>
     <...>
   </Template>
   <AppliesTo>
     <...>
   </AppliesTo>
   <Mappings>
     <...>
   </Mappings>
   <DeclarationHook>
table ttcmcs080   |* Carriers/LSP
   </DeclarationHook>
   <ConditionHook>
select tcmcs080.scac
from   tcmcs080
where  tcmcs080.cfrw = :#table#.carr
as set with 1 rows
selectdo
    return(tcmcs080.scac = "UPSN")
endselect
return(false)
   </ConditionHook>
</ContextMessage>
```

Template

For each context message two Template elements must be present. A Template element has this attribute:

Attribute	Description
condition	The runtime condition that applies for sending a context message according this template. These values are allowed: • `gen:data` – use this value for the template that must be used for sending data. • `gen:empty` – use this value for the template that must be used to send an empty message (to clear the web part).

A Template element has this sub element:

Element	Description
`_Block_`	A block of XML elements. At least one block must be present. This block will be repeated with data for each context message of the same context message type that must be sent. For example if multiple records are selected or if the same context message type must be sent for a referenced table. An optional block can be added for XML elements that only must occur once in the context message. For the content of the `_Block_` element, see the example.

Example:

```
<Template condition="gen:data">
  <_Block_>
    <...>
  </_Block_>
</Template>
```

Block

For each Template one or two _Block_ elements must be present. A _Block_ element has this attribute:

Attribute	Description
condition	This attribute must have the value “gen:norepeat” if the elements in the block apply to all different instances that make up the context message.

A Template element has this sub element:

Element	Description
Any	A number of XML elements. Those elements can have 1 level of sub elements. Within the elements, placeholders can be used. Placeholders are identified by "#" around a string, for example `#tablename#`. In the Mappings element those placeholders are mapped to a real value. Elements that have a sub level and are in a block that does not have the "`gen:norepeat`" condition, can have these attributes: • `INFOR_CONTEXT_ARRAY="true"` – use this attribute if the XML to JSON conversion (see …) must create a JSON array although there is only one occurrence. • `post_process="some.function"` – use this attribute to specify a function (from the FunctionHook or a linked library) that must be processed after the element is created. You cannot pass additional arguments to this function. One argument is passed: the node that just was created at runtime.

Example:

```
<Template condition="gen:data">
  <_Block_ condition="gen:norepeat">
    <screenid>#screen.id#</screenid>
  </_Block_>
  <_Block_>
    <entities INFOR_CONTEXT_ARRAY="true" post_process="post.process.enti
ties">
      <entityType>#tablename#</entityType>
      <key>#key#</entityType>
    </entities>
   </_Block_>
</Template>
```

AppliesTo

For each context message one AppliesTo element must be present. An AppliesTo element has these sub elements:

Element	Description
Table Filter Domain Filter TableField Filter	The Table element describes the tables to which the context message applies. The Filter restricts the tables. The Domain element describes the domain that must be present in the table to satisfy the criteria for the context message. The Filter restricts the domains. The TableField element describes the field that must be present in the table to satisfy the criteria for the context message. **Note:** when restricting elements are absent, the table matches the criteria. There can be multiple Table, Domain and TableField elements. Between these elements the OR operator applies. For the table filter wildcards can be used at the end of the table name.

Examples:

This context message applies to all tables.

```
<AppliesTo>
  <Table/>
</AppliesTo>
```

This context message applies to all tables of package "wh" and module "inh".

```
<AppliesTo>
  <Table>
    <Filter>whinh*</Filter>
  </Table>
</AppliesTo>
```

This context message applies to all tables of "whinh" that have a field with domain "tcrefa".

```
<AppliesTo>
  <Table>
    <Filter>whinh*</Filter>
    <Domain>
      <Filter>tcrefa</Filter>
    </Domain>
   </Table>
</AppliesTo>
```

This context message applies to all tables of "whinh" that have a field with domain "tcrefa" and field name "cpro"

```
<AppliesTo>
  <Table>
    <Filter>whinh*</Filter>
    <Domain>
      <Filter>tcrefa</Filter>
      <TableField>
        <Filter>cpro</Filter>
      </TableField>
    </Domain>
  </Table>
</AppliesTo>
```

This context message applies to all tables that have a field with domain "tcglat".

```
<AppliesTo>
  <Table>
    <Filter>*</Filter>
    <Domain>
      <Filter>tcglat</Filter>
    </Domain>
  </Table>
</AppliesTo>
```

This context message applies to all tables in modules "whinh" and "whwmd" that have a field with domain "tcglat".

```
<AppliesTo>
  <Table>
    <Filter>whinh*</Filter>
    <Domain>
      <Filter>tcglat</Filter>
    </Domain>
 </Table>
 <Table>
    <Filter>whwmd*</Filter>
    <Domain>
      <Filter>tcglat</Filter>
    </Domain>
 </Table>
</AppliesTo>
```

This context message applies to all tables that have a field with domain "tcglat" or "tcglon".

```
<AppliesTo>
  <Table>
    <Filter>*</Filter>
    <Domain>
      <Filter>tcglat</Filter>
    </Domain>
    <Domain>
      <Filter>tcglon</Filter>
    </Domain>
  </Table>
</AppliesTo>
```

Mappings

For each context message one Mappings element must be present. A Mappings element has this sub element:

Element	Description
Mapping	A Mapping element describes the source which must be used to fill a placeholder in the template. For the content of the Mapping element, see the example.

Example:

```
<Mappings">
            <Mapping>
          <…>
           </Mapping>
             <Mapping>
          <…>
```

```
            </Mapping>
        < /Mappings>
```

For each mappings element within a context message Mapping elements can be present. A Mapping element has this attribute:

Attribute	Description
type	How to process the Source element to fill the Target placeholder: **1** field – the placeholder will be replaced directly with the value of the field. **2** domain – the placeholder will be replaced directly with the value of the (first) field that is linked to this domain. **3** function – the placeholder will be replaced with the return value of the function. **4** hook – the placeholder will be replaced with the result value of the hook; a function wrapper will be created and this acts the same as a mapping of type “function”.

A Template element has these sub elements:

Element	Description
Source	Depending on the “type” attribute, the source is a field, a domain, a function or a hook.Within Source, you can use these placeholders: • #table#: this will be replaced by the table code for which the implementation model is generated. • #key<nn>#, where <nn> is numbered from 01 to 15: this will be replaced by the nth field of the primary key of the table • #instance#: this will be replaced by a fieldname of a field that is named “desc”, “dsca” or “name” or “nama”.
Target	The placeholder as used in the Template

Examples:

```
<Mapping type="field">
<Source>#table#.#key05#</Source>
<Target>ic_key05</Target>
< /Mapping>
<Mapping type="field">
<Source>#table#.#instance#</Source>
<Target>ic_instancename</Target>
< /Mapping>
<Mapping type="domain">
<Source>tcrefa</Source>
<Target>ic_trackingnumber</Target>
< /Mapping>
<Mapping type="function">
<Source>icm.get.screenid()</Source>
<Target>ic_screenid</Target>
< /Mapping>
<Mapping type="hook">
<Source>return(“ERPLN_” & icm.get.screenid())</Source>
```

```
<Target>ic_screenid</Target>
< /Mapping>
```