Introduction to Runtime User’s Guide

Version 3.4

Introduction to Runtime

This document gives an overview of runtime classes that you would use/encounter. You would interact with these indirectly using formula code or using the platform specific language (such as Java). The API or formula functions provided to access the classes in all these languages are quite similar. Hence in this document, we have discussed these classes in general and where applicable we have demonstrated the functionality using formula code. The details provided here should apply, in spirit, to any of the platform specific language and to the formula language. If you want details about how to work with the classes discussed here in a particular language refer to the API.

Class/Entity	**Description**
Data Object	Data Object is a hierarchical object representation of a message.
Raw Message	Raw message is an abstraction for an unprocessed message.
Service Objects	Service objects are the runtime representation of the core entities that you define using the Designer.
Context Object	Helper objects used to access deployed service objects and to perform an operation with it.
Transform Exception	Exception raised while processing (parsing, validation serialization etc) a message.

DataObject

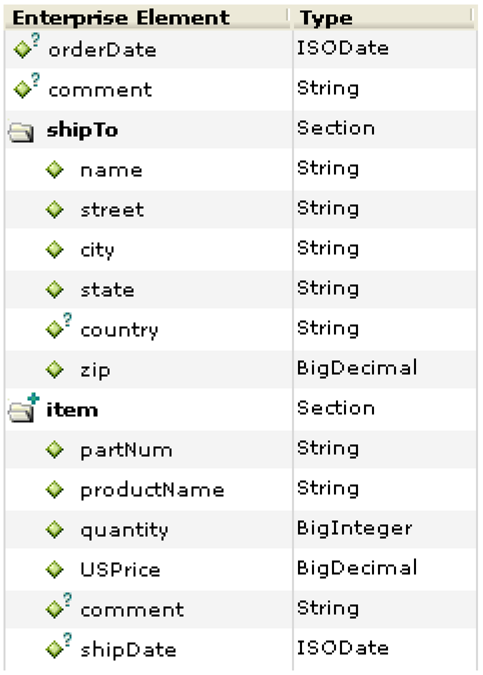
DataObject is a generic object that encapsulates a hierarchical data structure. It is structured object representation of a message and its sub-elements.

DataObject contains named fields and sections. Typically, these fields and sections (structure) are defined using Designer, and the DataObject is the runtime representation of this structure. During code generation phase, the structure defined is converted to concrete platform specific class(es). This class implements the DataObject interface, which provides a generic way of accessing the data structure. The generated class also has getters and setters, which let you access its fields and sections directly. You would interact with a DataObject using formula code or using the platform specific language (such as Java). The API or functions provided to access the DataObject in all these languages are quite similar.

Fields in the object represent primitive scalar value (like int, string etc) while section is a collection of sub-elements, which themselves are DataObjects.

At runtime, DataObject has support for the following,

It allows you to get/set fields it contains, either by name or by index.
It provides meta-information about the object itself.



Enterprise Element	Type
orderDate	ISODate
comment	String
shipTo	Section
name	String
street	String
city	String
state	String
country	String
zip	BigDecimal
item	Section
partNum	String
productName	String
quantity	BigInteger
USPrice	BigDecimal
comment	String
shipDate	ISODate

See Also:

Field/Section Access
Null Value
Sections
Meta Data

Field/Section Access

Fields and sections are the primary contents of the DataObject. Hence the main purpose of the DataObject interface is to let you access its fields and sections. As mentioned earlier, fields are scalars values, while sections are collection of a composite object.

See Also:

Accessing a Field in Formula

Accessing in Platform Specific Code

Accessing a Field in Formula

There are two ways of accessing the fields/sections in a DataObject. The one you use depends on whether the type (structure) of the DataObject is known. If the type is known, you can directly access its fields, using the *object.field* notation in formula or using getter/setter methods in platform specific code. If the type is not known, you need to use the generic interface; that is, get/set the field's value by its name or index.

If the type of the object is known (common scenario), the fields are available as properties which you can access using the dot notation by using its name. For example to access the field orderDate in a DataObject 'obj', use,

```
obj.orderDate
```

Care should be taken when the field you access can potentially have a null value (say if it is optional). If it has a null value, accessing the field will result in a runtime exception. Hence in such cases, it is important to guard field access with a isNull() check. That is, before you access the field check that it has a non-null value; if not, handle it as per your requirement.

```
If(isNotNull(obj.orderDate), obj.orderDate, today())
```

Note that, in many cases the DataObject is implicitly available. For instance, in the mapping screen, the source is implicitly available; hence it need not be referred by name. You can access its fields directly without any prefix.

```
orderDate
```

Here the orderDate refers to field orderDate in the implicitly available object.

The **obj.orderdate** syntax is typically used within a message flow, where number of such objects may be declared. In other cases, such as in mapping, validation and preprocessing, the DataObject 'obj' is implicitly available; hence you can directly access its fields.

If the type of the object is not known, you use the formula functions defined in 'DataObject' category to access its fields. This kind of scenario typically occurs in case of "Dynamic flows" or when you are accessing an object whose structure is not defined/available in cartridge.

```
Get<TypeName>(obj, fieldNameOrIndex)
```

Returns the value of the field indicated by fieldNameOrIndex from the given object. The return type corresponds to the type name.

For example, to access a field of type int, you would use

```
GetInt(obj, “myField”)
```

If the underlying field cannot be converted to the corresponding type by identity conversion or widening conversion, a runtime exception is raised.

`Set<TypeName>(obj, fieldNameOrIndex, value)`

Sets the value of the field indicated by fieldNameOrIndex in the given object. The value should be of the corresponding type.

For example to set a field of type int, you would use

```
SetInt(obj, “myField”, 10)
```

If the value specified cannot be converted to the actual type of the field by identity conversion or widening conversion, a runtime exception is raised.

Formula functions (such secSelectValues) are also available to access the value of a deeply nested field. Refer to formula functions category “DataObject” and “Transform” for more details.

See Also:

Accessing in Platform Specific Code
Field/Section Access

Accessing in Platform Specific Code

As in formula, the way you access a field depends on whether the type of the object is known or not. As you would remember, the data structures defined in Designer gets converted to platform specific classes. If the type is known you need to cast the object to appropriate generated class. The generated class provides getter/setter method to access the fields.

```
Order order = (Order) internalMessage.createObject();
Date date = Order.getOrderdate();
```

Like in formula, you should be careful about accessing fields, which can potentially have a null value. The getter method throws a runtime exception if the field’s value is null. To avoid this, you need to first check whether it has null value using the isNull() method.

Unlike in formula, it is very common to use generic field access, even if you know what the type of object you are dealing with. The main reason is to avoid explicitly using the generated classes in the client code. If this is the case, or if you do not know the type of the object, you can always resort to generic access. The DataObject interface provides number of method to generically get/set a field's value.

The fields can be referred either by their name of by their index. The name of field is the name specified in the Designer while defining the structure. If name contains spaces or special characters you can either use the name as it is (with special characters) or use a mangled representation of the name. The index is the position (zero based) of the field in the enclosing structure.

Fields of a DataObject are statically typed. That is, their types are known at Design time. Hence the runtime representation (Dataobject) enforces type safe access/modification of field values. For instance, attempt to set the value of field with incorrect type will result in a runtime exception.

See also:

Accessing a Field in Formula
Field/Section Access

Null Value

Fields support a concept of "null". Field whose value is not explicitly set is in null state. Initially when the object is constructed all its fields have a null value. Once a value is set, it changes to a non-null state. It is also possible to explicitly reset the value of the field to null, from a valid value.

In Designer, fields can be defined to be optional or mandatory. If a field is defined as optional, then a "null" value for it is legal and valid. A mandatory field can also temporarily have a null value, for instance when the enclosing object is constructed. But before the object is put to use (for instance serialized), the value of all mandatory fields must be set.

To check whether a field has a null value, you can use the isNull and isNotNull functions.

```
if(isNotNull(obj.orderDate)) {
     if(obj.orderDate > today()) {
          //reject the order
     }
}
```

If the type of the object is not known, you can dynamically check whether the field has null value;

```
if(isNotNull(obj, "orderDate")) {
     if(GetDate(obj, "orderDate") > today()) {
          //reject the order
     }
}
```

`IsNull(field)`

Returns ‘true’ if the given field has null value. It should be noted that the IsNull() function should be used only with fields. If you want check whether a section is empty (no elements) use "Sec.$size == 0" or one of the SecExists functions. The IsNull() function applied on a section would always return true since the section itself is never null but it can be without elements.

`IsNull(obj, fieldNameOrIndex)`

Returns true if the field indicated by fieldNameOrIndex has a null value.

The isNotNull variants are exact opposite of isNull functions.

In platform specific code, you can use the methods isNull functions available in DataObject interface to check for null valued fields.

Note:

> When the field you access can potentially have a null value (say if it is optional), it is important to guard field access with a isNotNull() check; otherwise it will lead to a null value exception. That is, before you access the field, ensure that that it has a non-null value.
>
> Sections do not support a concept of null. Sections, as mentioned earlier represent a collection of DataObjects. A section in a DataObject is never null; it can be empty. During construction of the enclosing DataObject, the section is initialized to an empty collection.

See Also:

DataObject

Sections

Section is a collection of sub-elements. As mentioned before, there are two ways of accessing the sections in a DataObject. The one you use depends on whether the type (structure) of the DataObject is known. If the type is known you can directly access its fields, using the *object.field* notation in formula or using getter/setter methods in platform specific code. If the type is not known you need to use the generic interface; that is, get the section collection by its name or index.

If the type of the object is known (common scenario), the fields are available as properties which you can access using the dot notation by using its name.

```
def sec = obj.Item;
```

If the type of the object is not known, you use the formula functions defined in 'DataObject' category to access its sections.

```
def sec = getSection(obj, "item");
```

Like fields, sections can be optional or mandatory. A mandatory section should have at least one child element. Since section represents a collection, if it is marked as repeating, it can have more that one child element. The cardinality of a section is indicated by two properties, minOccurs and maxOccurs. These two properties provide a bound for the number of child elements in the section.

See Also:

Accessing Elements of a Section

Accessing Elements of a Section

In the formula code you can access the elements of a section using an array like syntax. To get the **n**th element of the section, use

```
sec[n] //returns the nth element of the section
```

The number of elements in the section can be obtained using the $size property or the secCount function.

```
sec.$size  //returns the number of elements in the section
```

See Also:

Sections
Meta data

Meta Data

The meta-information about a DataObject is made available at runtime using the class DataObjectMetaInfo. The DataObjectMetaInfo class can be used to get information about the types and properties of the fields and sections in a DataObject.

```
DataObject obj = …
DataObjectMetaInfo  metaInfo = obj. getMetaInfo();
int count = metaInfo.getFieldCount();
```

Similarly there are formula functions in the “DataObject” category to access the meta-information about an object. These include functions such as GetFieldName(), GetFieldType() etc.

See Also:

DataObject

Service Objects

Service objects are the runtime representation of the core entities that you define using the Designer. In Designer you define a Message (External/Internal), Mapping and a Message flow. These entities are represented at runtime by service interfaces of the same name.

The following service objects defined in Designer are available at runtime

- Message
  - a. External Message
  - b. Internal Message
- Message Mapping
- Message Flow

During code generation the Design time entities are converted to platform specific code. The generated service objects can be accessed at runtime using the LookupContext.

These service objects can be obtained by name using the LookupContext. The name you use the same as the one you specified in the Designer. Though all entities are available, the recommended approach is to use the message flow as façade for other components. That is, the client program should interact only with the Message flow component; the message flow itself would invoke or make use of other components listed above.

Note:

There is no way to use/invoke the service elements directly from formula. You can use the entities defined in cartridge from message flows.

See Also:

Message
Mapping
Message Flow

Message

Message represents a message processor service object that can be used to parse, validate and serialize a message. This interface has methods that apply to the raw and the object representation of the message.

parse - converts raw message to message object
serialize- converts message object to raw form
validate - validates a message object
create - creates a uninitialized message object.

Each concrete implementation of this interface typically is capable of parsing one type of message (for e.g. New Order FIX message).
This interface has two variants of the methods for performing an operation. The former performs the operation and throws an exception in case of errors. The latter, which has a suffix '2' (as in parse2), accumulates (cascades) the exceptions and returns a Result object. The Result object is a pair containing the output of the method and the accumulated exceptions. While the former, returns an output or throws an exception, the latter, depending on the severity of the exception encountered during processing, may return an output, exception or both.

There are two major classifications of messages, internal and external. There is an interface each to represent the two types of the messages,

InternalMessage
ExternalMessage

See Also:

Service Objects

Mapping

This MessageMapping interface is used for mapping a source message to a destination message. The mapping definition should have been defined using the Designer and deployed in this runtime environment. The primary method in this interface is the map(DataObject, DataObject, TransformContext) method, which maps a source message object to a destination message object.

Message Mapping comes in four different flavors,

External to Internal
External to External
Internal to External
Internal to Internal

Like other service objects, you can get access to a MessageMapping object by looking up using LookupContext, LookupContext.lookupMessageMapping(String). The preferred way of using a MessageMapping is through a message flow and not directly from client applications.

The MessageMapping is responsible for

Mapping the source message object to destination message object

See Also:

Service Objects

Message Flow

MessageFlow interface represents a executable content designed using the 'Message Flow' entity in Designer. Message flow is made of activities, which are executed in the sequence specified in the Designer. The message flow can take one or more parameters as input and produces one or more outputs. The primary method in this interface is the **`run(Object[],TransformContext)`** method. This method executes the message flow with the specified parameters and returns the output(s). Note that, it supports multiple input and parameters and returns multiple output values. The input/output parameters and there types are defined when you design the Message flow using the Designer. The values passed to the run method should conform to the types specified at design time.

There are two variants of this method,

`run(Object[],TransformContext)` - returns the output and throws an exception in case of failure.

`run2(Object[],TransformContext)` - returns the output and any exceptions as a Result object. The output generated is a snapshot of output variables in the message flow at the time of the exception. This method is useful in a 'Repair' like application where incorrect input should not treated as fatal.

Both these method support number of ways in which arguments can be passed.

> as an array of objects - values in the array correspond to the parameters specified during Message flow design. The length of the array must be same the number of parameters. Null values are allowed as long as it is gracefully handled in the Message flow that you design. Passing wrong types will lead to ClassCastException.
>
> as a **`DataObject`** - Use **`createInputDataObject()`** to create a DataObject representing the input parameters.The fields in this object are same as variables that were marked as 'INPUT' while defining the message flow in Designer. Set the fields in this object using setField("XXX", value) methods. Using run() with a DataObject has better type safety compared to Object[] since the incorrectly typed values will be caught while constructing the DataObject.
>
> as a InputSource - This is convenience method that can be used if your message flow takes only a raw message (byte[]) as input parameter.

Though Message flow objects are stateless, it may cache other service elements that it uses. Hence it may be expensive to create a new instance of the MessageFlow. It is recommended that you reuse the MessageFlow objects for further processing.

See Also:

Service Objects

Context Object

There are two important context objects, which you would encounter when you are writing applications that interact with cartridge entities. These are LookupContext and TransformContext. The former is use for looking up service objects while the later is used while invoking the methods in the service objects.

LookupContext

The LookupContext provides access to other service objects executing in the runtime environment. Using the lookup context you can access other generated components such as message flow, mappings etc.

Obtaining Lookup Context:

The ‘LookupContext’ context can be obtained from the LookupContextFactory as shown below.

```
 LookupContext lcxt = LookupContextFactory.getLookupContext();
```

The Lookup Context that is returned will differ depending on the environment in which it is obtained. That is why instead of instead of instantiating the Lookup Context we use LookupContextFactory to obtain it.

Methods in Lookup Context:

The following are the methods that are most commonly used during transformation. These are not the complete set of methods available.

`lookupMessage(String name)`	Looks up a message and returns it. The name to be looked up should be the name of an external message or a internal message defined in Designer.
`lookupExternalMessage(String name)`	Looks up an external message and returns it. The name to be looked up should be the name of the external message defined in Designer.
`lookupInternalMessage(String name)`	Look up a Internal message (formerly Business Transaction) and returns it. The name to be looked up should be the name of the Internal message defined in Designer.
`lookupMessageFlow(String name)`	Looks up Message flow and returns it. The name to be looked up should be the name of a message flow defined in Designer.
`lookupMessageMaping(String name)`	Looks up Message Mapping and returns it. The name to be looked up should be the name of a mapping defined in Designer.
`lookupDataSource(String name)`	Looks up a data source. This is mostly used internally and is not relevant for clients

See Also:

Context Object

TransformContext

It defines the context in which the current transformation occurs. This context object contains a set of properties (name-value pairs) related to the current transformation.

This object is passed to all the components (message flow, messages etc.) that take part in processing.

Methods in Transform Context:

The following are the methods that are most commonly used during transformation. These are not the complete set of methods available.

`setProperty(String name, Object value)`	This method can be used set any property related to the current transformation.
`getContextProperty(String name)`	Returns the value of the specified property

Note that TransformContext is an interface. You can use the class TransformContextImpl, class to create an instance.

You can use the function **`getContextProperty()`** to access properties in the TransformContext object. This formula function can be used in formula code for mapping, validation, message flow. Specifically, it is not available in places where a Transformation context does not exist, such as in a function definition.

The context property can typically be used to control the behavior of an entity from outside. For instance, in case of mapping the only object that is available for mapping is the input object. If for some reason you want to control the mapping from outside (call point) we can pass an option (property) using the transform context. In the mapping rules you can lookup this property's value and perform the mapping in a slightly different manner. It is important to note that the activity must honor (recognize and respond to) the property; otherwise it has no impact.

These properties should be treated as additional configuration options and should not be used to exchange data values. Since the properties need to be passed by the caller, the entity should always have a default value which will be used if a property is not available in the context.

Note that, some activities (like Swift Parse), support some predefined properties. These properties are documented elsewhere in the help.

Setting the values in Context Property:

There are number of ways to set a property in the TransformContext.

1. Many activities in a message flow have an “Other options” tab which lets you set the properties in the Transform Context. The values set using the tab will only be visible to the activity that is invoked and not in the rest of the message flow.

2. Use the setContextProperty function in a custom activity to set/update a context property. Properties set using this function are available till the message flow completes and is passed to any activity that is subsequently invoked.

3. The caller of the message flow usually creates the Transform Context object. The caller can be the hand written client or another message flow. In all cases there is a way to set the context property before invoking the flow. Properties set by the caller are available through out the flow and is passed to all activities that are invoked.

4. An activity such as mapping which can use formula code can itself set the context property. These properties are available only till the activity terminates; they are not visible to the invoking flow.

Usage:

Use the transform context to pass properties from the client to the message flow. Based on the property value the flow can behave differently. This is particularly applicable for generic clients, since a hand written client can pass values as additional arguments to the flow.

Use the ‘Other options tab’ of an activity in the message flow to pass additional information to the activity. This additional information can make the activity behave differently or provide additional data that the activity may need. For example, if a value is known only at runtime (through other means), but is not part of the input object, this value can be passed as a property to the mapping activity. During mapping you can fetch this value from the context and set as an output field’s value. Since the ‘other Options’ supports formula in value column, you can easily pass dynamically synthesized values to an activity.

See Also:

Context Object

Raw Message

Raw message is an abstraction for an unprocessed message. Apart from the content or the body of the message, it supports features such as properties and attachments that are commonly supported in other message representations.

Provides a single, unified message API

Provides an API suitable for creating messages that match the format used by message providers

In its simplest form raw message just contains the body, which can either be a text or a stream of bytes. Complex messages may contain typed properties and attachments.

Raw Messages are composed of the following parts:

Body – This represents the content or the payload of the message. Two types of message body are supported, text and a stream of uninterpreted bytes

Properties - Each message contains a built-in facility for supporting application defined property values. Properties are name value pairs. Properties provide an efficient mechanism for supporting application defined message filtering (impl in future).

Attachments, which themselves are named raw messages.

See Also:

Message Body
Message Properties
Message Attachments
Using Raw Message in Message Flows
Message Conversion
Support for Batching

Message Body

Raw messages support two types of message body:

Text - a message containing text or String.

Bytes - a stream of uninterpreted bytes. This message type is for literally encoding a body to match an existing message format.

You can access the content of a message from the Designer using formula functions.

Function	**Description**
ToRawMessage(Binary Or String)	Converts the given string or binary value to Raw Message. This is only of the possible ways of creating a RawMessage. Typically, the RawMessage is produced

	as output by the Serialize activity.
ToText(rawMessage)	Returns the text representation of the message
ToBinary(rawMessage)	Returns the binary (bytes) representation of the message
NewMessage()	Creates an empty in-memory raw message

See Also:

Raw Message

Message Properties

Every raw message has an associated property set. The property set is a collection of name-value pairs. The property names in the set are unique. The property values are typed; most basic Designer types are supported (see table below).

Message properties support the following conversion table. The marked cases are supported. The unmarked cases will fail. The String to primitive conversions may throw a runtime exception if the primitives are not convertible to String.

A value written as the row type can be read as the column type.

	boolean	int	long	float	double	String	DateTime
boolean	X					X	
int		X	X			X	
long			X			X	
float				X	X	X	
double					X	X	
String	X	X	X	X	X	X	X
DateTime						X	X

You can access the properties of a message from the Designer using formula functions.

Function	**Description**
`GetProperty(message, propertyName, defaultValueOpt)`	Gets the property indicated by the specified name. If the specified property is not present, the given default value is returned.
`SetProperty(message, propertyName, propertyValue)`	Sets the property indicated by the specified name
`HasProperty(message, propertyName)`	Returns 'true' if there is a property with the specified name
`RemoveProperty(message, propertyName)`	Removes the specified property.
`ClearProperties(message)`	Clears all the properties.
`GetPropertyNames(message)`	Returns a list of available properties

`GetProperties(message, propertyPrefix)`	Returns a Property Map, which contains all properties that have the specified prefix. All the functions discussed here can be applied to this Property Map.
	
	The following code clears all properties that start with "jms.".
	
	def jmsProps = raw.GetProperties("jms");
	jmsProps.ClearProperties();
	
	Note that the prefix specified should not contain the '.' character.

Identical functions are available in the platform specific language (Java) as well.

Though the property set is essentially flat, for convenience it can be viewed in a hierarchical manner. This is useful because the property set of a raw message contains properties obtained from various sources. For instance, in case of JMS Message, the JMS message properties are mapped to properties in the RawMessage with a prefix “jms”. Hence if the actual JMS property name is “appSource”, in the raw message the property name is “jms.appSource”. Similarly the JMS header fields are again represented as properties but with a different prefix “jmsHeader”. To access only the JMS properties,

```
def jmsProps = mes.getProperties("jms");
def appSrc = jmsProps.getProperty("appSource", "");
```

This is equivalent to,

```
def appSrc = mes.getProperties("jms.appSource "");
```

The former is convenient if you are working with number of properties.

The ability to partition the property set into different namespaces is very useful. Because of this, it can contain message properties, message headers and application specific properties (say for routing) without interfering with each other. You can access/modify each property group without disturbing others.

See Also:

Raw Message

Message Attachments

Some type of messages, such as Email, SOAP etc support attachments. Designer represents each message attachment as another RawMessage and associates it with the original message.

Function	**Description**
GetAttachment(message, attachmentName)	Gets the attachment indicated by the specified name from a Raw Message
SetAttachment(message, attachmentName, attachedMessage)	Attaches a message indicated by the specified name to a Raw Message. The attachment should be a Raw Message.
HasAttachment(message, attachmentName)	Returns 'true' if there is an attachment with the specified name in the Raw Message.

Example:

The following code snippet creates a raw message and adds an attachment to it.

```
def mes = ToRawMessage("MESSAGE CONTENT");
def attachment = ToRawMessage("ATTACHED MESSAGE CONTENT");
mes.SetAttachment("myAttachment", attachment);
return mes;
```

See Also:

Raw Message

Using Raw Message in Message Flows

Variables in the message flow can be of type RawMessage. Compared to simple Binary type, which is usually used to represent the message content, RawMessage type has the following advantages.

- You can get/set message properties
- You can work with message attachments

Note:

RawMessage type represents a message along with its attributes. The Binary type represents uninterpreted stream of bytes. Just like other types, you can have Binary typed fields. It is not necessary the Binary type be always used for representing messages. Unlike Binary, a RawMessage always represents an entire message. For this reason, fields of type “RawMessage” in format plugins are not supported (does not make sense?). So the only place where a RawMessage type makes sense is in message flows, where you directly handle incoming and outgoing messages.

The activities that accept Binary as input also accept a RawMessage. For example, the Parse activity accepts a variable of type RawMessage as the source. Similarly, activities that produce Binary typed output can be made to produce a RawMessage as output. In both cases, just define the variables of type ‘RawMessage’ and use them the Parse/Serialize activities.

Along with this, there are also formula functions to convert RawMessage to and from Binary type. Also, for input parameters of the message, there is an implicit conversion from Binary to RawMessage and vice versa. That is, if the input parameter of a flow is of type RawMessage, but the actual value passed is of type ‘Binary’ it is automatically converted to a RawMessage.

It is not necessary to migrate from Binary type to RawMessage to represent an unparsed message. If you do not have any use for message properties and attachments you can continue to use Binary types to represent a message. This addition is fully backwards compatible. It just provides more features; features that are supported in many message types (such as EMail).

See Also:

Raw Message

Message Conversion

When a message is received from a source (provider), it first is converted to (or wrapped with) a RawMessage. You can access the payload, properties and the attachments in the original message using the RawMessage API. You can access this API in the native platform using the interface RawMessage or using formula functions available in the "Message Functions" category.

In some cases the provider specific message representation may support more or different set of features, which are not directly available in case of RawMessage. For instance, JMSMessage supports a set of predefined headers fields. Since the concept of header fields is not available in case of RawMessage they are represented as message properties with a specific prefix (to distinguish it from other message properties). The general idea is to represent all the attributes of a provider specific message in a raw message, so that they can be accessed using RawMessage API. Often, the property set of the RawMessage is used to represent other attributes which do not have a direct representation. For example, in case of an EMAIL message, the subject is represented using the property "mail.subject".

When a raw message is sent out, again it needs to be converted to an external message representation (such as a Mail message). The payload, properties and attachments in the RawMessage are converted to a provider specific message representation. The conversion is the opposite of the conversion process discussed above.

Make sure that you understand the conversion between a raw message and a provider specific message. For each type of message, the conversion to and from a raw message is discussed in detail elsewhere in the help.

See Also:

Raw Message

Support for Batching

The main difference between normal messages and a batched message on the input side is that the batch messages are huge in size. All messages are abstracted by the type “RawMessage”. Note that while “Binary” type represents a concrete byte array, RawMessage is implementation dependent. The length of “Binary” is restricted by the amount of physical memory, while RawMessage can be backed by a file or some other source (depending on the implementation); without loading the entire message into memory. For this reason, it is imperative that you use RawMessages (and not Binary) when working with batches.

There are number of formula functions which apply to RawMessage all of which operate efficiently on a huge message. These functions have same name and semantics as functions that operate on Binary and String data types. These functions are

Mid(rawIn, index, length).

This function returns a chunk of data from the raw message. The returned chunk is an array of bytes (Binary type). There two more variants of this function, left and right, which also allow you to access parts of the message. As long the accessed chunk is reasonable in size (< 1 MB) these functions work well even with huge batches. For instance, if the RawMessage size if 300MB and you are accessing chunks of 1755 bytes (per record), it is safe to use this function. This function is useful for breaking up a batched message with fixed length records into chunks.

At(rawIn, index)

The At function returns the byte (as int) at the specified index in the message. This is equivalent to using array access operator rawIn[index].

Again, this is reasonably efficient even for huge messages. If you are using this function to access to the entire file, byte at a time, there will be slight degradation in performance, since you would be accessing one byte at a time. Typically, this is useful for searching for a delimiter in a batch; in such cases, use the findFirst function instead. If possible avoid using At function to access the entire file. Having said that, the degradation in performance of accessing a huge file byte at a time is in most cases acceptable.

findFirst(rawIn, "\r\n", index)

The findFirst function searches for the specified characters or byte in the rawMessage starting at the specified location. It returns the index at which the sequence is found or -1 if there is no match. The function is useful for breaking up a batched message with delimited records. Once you get a match, you can extract the chunk using Mid function.

These RawMessage functions can be used when the batch needs to be broken up into chunks manually; that is when the records are delimited or of fixed length. The format specific parser in these cases operates only on reasonably sized chunks.

Refer to formula functions in the Message category for more details on how to work with RawMessage in formula. Refer to the API documentation the RawMessage interface and its implementations for details about using RawMessage from client code.

See Also:

Raw Message

Transform Exception

It is the exception raised while processing (parsing, validation serialization etc) a message. Like many other object discussed in this document, this object available both in formula code and as part runtime API.

TransformException is represented as a first class Normalized Object. Because of this, you can create/access it fields like any other Normalized Object. Using the Designer, you can define mappings to and from a TransformException. In the message flow you can catch exceptions thrown by the activities and take appropriate corrective actions.
Each Transform exception object provides several pieces of information:

- A string describing the error. This is used as the Java Exception message, available via the method getMessage.
- An error code and an internal error code
- Severity of the error
- Name of the field in which error occurred (if applicable) or null.
- Id of the field in which error occurred (if applicable) or null.

Note that a TransformException can represent a collection of exceptions. In such cases the above apply to the values for the first exception. Methods are available to access other elements of the collection.

See Also:

Introduction to Runtime