Message Storage and API Format

v1.0.0

SYMPHONY

Message Storage and API Format

Messages are currently stored in markdown with entities in several additional fields as JSON objects. There is a need to provide a way to mark up messages to include formatting and custom entities through the Agent API and for various reasons we are unhappy with the current storage format.

An HTML style of markup for formatting (bold, italic, numbered and unnumbered lists etc.) seems appropriate and the Agent API defines MessageML as a markup to express some of this formatting and a limited set of entities. Processing XML on the client in JavaScript is unnatural at the very least and this does not seem like a good format for the storage of embedded entities within messages. An HTML like approach to markup of formatting does, on the other hand, seem attractive.

In order to ensure that we define good structures for marking up messages both at rest (stored in the database) and in motion (over the wire on an API) it is necessary to articulate the various actors involved in the creation and rendering of messages including complex embedded entities.

Actor	Description
Message Author	The process which creates a message, which may have an opinion of how the message should be presented (e.g. by marking parts of the message as bold or italic or as a paragraph break) and which may create embedded entities expressed in a structure it defines.
Entity Renderer	A process which knows about one or more entity formats and how to render them.

When a message author expresses formatting for a message this should be done in a semantic rather than a presentational way. For example, the message author may say that there is a paragraph break in a message but it is up to the renderer to decide precisely how that should be rendered, e.g. whether the space between the end of one paragraph and the next should be 2 pixels or 15 pixels. In an ideal world we would want to mark up messages to indicate emphasis in a presentationally neutral way for example by referring to a word being "emphasized" as opposed to being "bold" or "italic".

However, the existing GUI message editor provides buttons labelled as bold and italic and users are familiar with these terms as ways of editing a message. We will therefore treat bold and italic as semantic formatting, even though these terms express a presentational concept.

In one use case the message creator and the custom entity renderer are developed by the same person, but this is not necessarily so. For example, Atlassian might define a JIRA webhook and an entity format to represent a JIRA ticket together with a renderer which knows how to present a JIRA ticket entity as a pretty table with familiar icons for the current state of the ticket. IBM produce the Jazz range of SDLC tools which includes its own ticketing system and they might similarly produce an entity format for a Jazz ticket and a corresponding renderer.

Now it could be that IBM produce a version of their renderer which can display a JIRA ticket in a way which is convenient for users who are familiar with Jazz but who have to also use JIRA. A message containing a JIRA entity could be displayed using the JIRA renderer for some users, but by the Jazz rendered for other users who have that renderer installed.

Alternatively the Jazz webhook might choose to provide a main entity format which expresses all the detail of a Jazz ticket with a fall back entity in the JIRA entity format so that when the message is viewed by a user who only has the JIRA renderer installed that they get a better experience than they would otherwise.

Similar cases exist in the financial industry domain where multiple instrument IDs or ticker symbols may refer to the same or closely related entities.

Proposed Format

We propose to divide the markup of messages into two areas of concern. PresentationML is a markup language to express semantic presentation of messages in a domain agnostic way. PresentationML allows the message author to describe how the message should be presented in a semantic way by marking up text to be presented as

- Bold
- Italic
- Paragraph Break
- Line Break
- Numbered List
- Unnumbered list
- Clickable URL
- Simple html style tables.

No specific guarantee will be made about the graphical presentation of these elements which may vary from device to device and from user to user.

All valid PresentationML will be valid HTML and valid XML without modification. This means that API consumers will be able to parse PresentationML content with an XML parser and the UI will be able to display it as HTML with a CSS style sheet.

MessageML is a superset of PresentationML which also provides an in-line expression of arbitrary embedded entities. All valid MessageML will be valid XML which means that API consumers can treat MessageML content as XML and parse it with an XML parser. Messages transmitted over APIs will be in MessageML format.

All custom entities will be required to provide a default rendering expressed either as PresentationML or plain ASCII. In the event that no renderer for a particular custom entity can be found then this default rendering will be used. A renderer to present PresentationML as plain ASCII will be provided for use by content export and other consumers requiring plain text.

To enable the UI to process messages efficiently messages stored in the database will be represented in StorageML which consists of a single string of PresentationML and a list of entities represented as a JSON array of entity structures. Each entity structure may be either a single JSON object or an array of

objects and a start and end offset in the PresentationML string representing the part of the PresentationML which should be replaced with the rendered entity. In the case where an array is provided the elements of this array represent alternate versions for the same entity in descending order of preference. For example using the example above the first element could be a Jazz format entity representing a ticket and the second element might be a JIRA format entity representing the main aspects of the same ticket. In this case the UI would first try to find a renderer for the Jazz format, if none is found then it would look for a renderer for the JIRA format and if none is found for that either then the PresentationML version of the entity would be displayed.

Conversion of messages between MessageML and StorageML is straight forward in Java and can be done on the server for delivery via the Agent API and similar use cases. The formats will be defined such that all valid StorageML can be represented as valid MessageML and vice versa, and conversions defined such that any message in one format which is converted to the other and then converted back again will result in the same markup.

Examples

The following examples are illustrative and do not represent a complete specification of the formats:

API Format	**Storage Format**	
MessageML	**PresentationML**	**JSON**
Hello World	Hello World	
Plain Bold Bold+Italic Bold Italic	Plain Bold Bold+Italic Bold Italic	

New Task <entity>
<format type="com.atlassian.jira" id="27" name="develop code" host="http://symphony.jira.com/getTicket?id=27"/>
Jira:27
</entity>
	New Task
	Jira:27
		{
		"indexStart": "17",
		"indexEnd": "23",
		"entity": [
		[{
		"type": "com.atlassian.jira",
		"id": "27",
		"name": "develop code",
		"host":
		"http://symphony.jira.com/getTicket?id=27"
		}]
		]
		}

PresentationML

The following tags are defined for PresentationML, tag names are case sensitive:

Tag	Description	Can Be Nested Within
<PresentationML>document</PresentationML>	A PresentationML document.	
text	Paragraph, contains text to be rendered as a single block of text.	<PresentationML> <td><th>
text	Bold, contains text to be rendered in an emphasized way, by presentation as a bold font if possible.	<PresentationML> <td><th>
text	Italic, contains text to be rendered in a (secondary) emphasized way, by presentation as an italic font if possible.	<PresentationML> <td><th>
List Item 1 List Item 1	Unordered list, should be rendered as bullet points if possible.	<PresentationML> <td><th>

List Item 1 List Item 1	Ordered list, should be rendered as a numbered list possible.	<PresentationML> <p><b><i> <td><th>
<table> <tr><th>Col 1</th><th>Col 2</th></tr> <tr><td>Cell 1,1</td><td>Cel 2,1</td></tr> </table>	Simple table.	<PresentationML> <p>
<a url="http://some.url">text<a/> <a url="http://some.url"/>	Hyperlink.	<PresentationML> <p><b><i> <td><th>
<chime/>	An audible chime message. A message MAY contain a single <chime/> tag, in which case it MUST contain no other content. Conversely a message which contains any other content MUST NOT contain any <chime/> tag. This is a limitation of the current implementation which may or may not change in the future.	<PresentationML>

Named Character References

MessageML special characters can be escaped using HTML named character references. < > and & should be used to escape < > and &. The set of supported entities is as defined in http://www.w3.org/TR/html5/syntax.html#named-character-references

MessageML

The following tags are defined for MessageML, tag names are case sensitive:

Tag	Description	Can Be Nested Within
<MessageML>document</MessageML>	A MessageML document.	
<hash tag="label"/>	A hashtag, equivalent to entering #label in the GUI.	<MessageML> <p><b><i><td><th>
<cash tag="label"/>	A cashtag, equivalent to entering $label in the GUI.	<MessageML> <p><b><i><td><th>

<entity> <format type="type.id1" attr1="value1" attr2="value2"/> <format type="type.id2" attr3="value3" attr4="value4"> <any><tags with="any attributes"></tags></any> </format> PresentationML </entity>	A custom entity with one or more entity formats and a default rendering expressed in PresentationML. Following the opening <entity> tag there MUST be one or more <format> tags. Each <format> tag MUST have a type attribute which contains a format identifier in Java package name format. Only the owner of the equivalent domain in DNS should define that ID. Each format tag MAY then have any number of additional attributes, but each attribute name MUST be unique within that tag. Each format tag MAY contain an arbitrary set of sub-tags, each of which MAY have zero or more attributes and further sub tags. Only tags may be contained within a <format> tag, text content outside of an attribute MUST NOT appear within a <format> tag. Format tags are listed in descending order of preference, when displaying an entity the formats will be checked in order and the first format for which a valid renderer exists will be used and all other formats will be ignored. In the event that multiple renderers exist for the same format then the presentation engine will select its preferred renderer according to its own rules. Finally after all <format> tags there SHOULD be a sequence of valid PresentationML tags which represent the default presentation in the event that no appropriate renderer for any provided format can be found.	<MessageML> <p><b><i><td><th>

<errors> <error>Some error message</error> <error>Another error message</error> </errors>	A structure to allow for the reporting of errors in transformations. These tags should only be produced as a result of generating MessageML via transformation from another format. When generating MessageML as part of a read operation, the generating system MAY include a single <errors> section to indicate problems which occurred during processing. When sending MessageML as part of a write request the sending system SHOULD NOT generate an <errors> section. If such a section is received by the server fulfilling a write operation then the operation MUST fail.	<MessageML>

Additionally all PresentationML tags except for <PresentationML></PresentationML> are also valid MessageML tags, and have the same meaning as defined above. This means that all PresentationML tags can be nested within <MessageML>