Interactive Web Services (IWS)

1. Introduction

This document provides a **draft** specification for Interactive Web Services (IWS) — a standard set of interfaces based on Web Services standards such as SOAP, WSDL and UDDI — that allows companies to encapsulate and share interactive Web applications. It is published as a draft proposal for the OASIS WSCM (Web Services Component Model) Technical Committee (TC).

Why Web Services?

Web Services are emerging as the leading technology for encapsulating business processes and sharing them with business partners. They provide the foundation for reducing the integration cost between businesses and for creating new business relationships.

Web Services are emerging at a time when more companies are realizing they must share and integrate their Web applications with business partners. Several years of online experience have led companies to recognize that the high cost of business-to-business integration impedes the establishment of business relationships. Web Services provide the technology that enables companies to package applications and business processes for integration with multiple business partners in a cost-effective way.

Why Interactive Web Services?

Today, Web Services enable businesses to encapsulate *business logic* and share it with their partners, thus enabling integration of business logic from different companies. This is done by "calling" functions using established protocol stacks (such as the HTTP-XML-SOAP stack) to transfer the function arguments, and getting the result back.

This method of encapsulation of business logic is analogous to an application calling a function exposed in a shared dynamically linked library (DLL) – one application exposing its functionality via functions in the DLL, and another application using standard calling conventions to call the function and use it.

The Web Service standards stack today does not offer a standard method of encapsulating *customer experience*, i.e. enabling the encapsulation of the user interface (UI) and integrating it in a "container" UI.

Why share customer experience?

Today's Web Services standards facilitate exposing applications' business logic, and suggest that Web Services consumers write a new presentation logic on top of it. This method has several limitations:

1. A business has usually spent many resources on creating this UI. Re-creating this UI in another place is a waste of resources and is actually the problem Web Services have been trying to solve. A UI is not "just the HTML". It is a whole layer dealing with issues like state management, error handling, client-side scripting, etc… Re-creating this in another place is a non-trivial problem.
2. Businesses often invest heavily in their applications' *customer experience* and *brand* associated with the products or services it sells, and would like to maintain this experience across its partner network. Specifically, one cannot be sure that the UI is created to a business' exact specification. UI design is a subtle thing, and can easily (or maliciously) be wrongly re-created by the container.
3. Encapsulating a UI using a set of functions usually means exposing the whole business "object model", which is usually complex and consists of a long set of functions. This need can often be eliminated by exposing a coarser interface that exposes only the user interface.

The Interactive Web Services (IWS) model addresses those issues by defining a model enabling an application to encapsulate its UI, and by defining how a container can use this model to integrate the Web Service in its container UI.

IWS is "inspired" by the various existing "desktop" models for UI components – ActiveX, JavaBeans, OpenDoc, … - and tries not only to define the protocol, but also to re-create the success of "drag-and-drop" visual programming that have made those models a success.

Example Use Case

The specification overview uses the following scenario to explain IWS.

A provider of travelers checks would like to extend its travelers checks business to travel related sites, bringing their online services closer to the prospective customer and offering them in the relevant context. Already a profitable business, the provider is looking to place its service in any online site in which it makes sense to buy travelers checks. For example, a customer making travel reservations through a travel site would be able to buy travelers checks as well.

The provider has already developed e-commerce applications making its travelers checks available for purchase from its own Web site.

The provider's business objectives are:

- Get closer to customers, let them buy travelers cheques where they are, in the right context
- Increase revenue through a syndicated e-commerce application
- Avoid competition with partners
- Keep users on the partner's site and avoid sending them back to the vendor's site
- Package the application as an interactive Web service (reusable)
- Be cost effective to implement and maintain both for the vendor and its partners
- Be transparent to current and future vendor applications.

2. IWS Design Goals

This section outlines some of the design goals behind this specification.

1. Supports encapsulation of any interactive multi-step application as an Interactive Web Service. In particular:
    a. Support any type of application workflow, whether represented formally (using languages such as WSFL) or encapsulated within a computer program (such as a set of JSP files). In particular, support any programming model and development environment for the UI of the IWS.
2. Allows the container to add value to the integrated Web Service. In particular:
    a. Allows the container to customize the presentation, yet allow the provider of the Web Service to control the business logic behind those changes.
    b. Enable the container to control and intervene in the workflow of the IWS, and in particular to exchange data with the Web Service throughout the interaction with the user.
3. Provide easy means for the container to incorporate an IWS (with a “drag-and-drop ease”). In particular:
    a. Support any container HTML environment. IWS defines a model but does not mandate a specific environment or development model by the container.
    b. Do not mandate custom programming for an individual IWS. IWS requires (simple) code at the container application, but no new code is required per new Web Service.
    c. Do not require the container to “understand” the page-to-page flow of the IWS.
    d. Support and encourage visual programming capabilities by the container application.
4. Provide a *practical* model. In particular:
    a. Support encapsulation of existing applications. Although the IWS model does not define a mechanism for transparently converting existing applications into IWS, it is design to support automated or semi-automated publication of applications as IWS.
    b. Support loose coupling between an IWS and an IWS container. Changes to the application are by default transparent to the container application, and do not explicitly require changes by the container.
    c. Build on previous UI encapsulation models and concepts such as *properties* and *event*.
    d. Define a *minimal* and *simple* protocol.
    e. Use standard protocols and protocol stacks. In particular, leverage SOAP, WSDL and UDDI as a transport, description and publishing standards. Assume the existence — and leverage — security standards.
5. Provide an efficient model from a performance perspective. In particular, provide coarse-grained interfaces to support integration of IWS across different companies.

3. IWS Specification Overview

Document Scope

This document defines, in precise terms, the Schema of IWS messages[1], and defines, in precise terms, the semantic of each part of the IWS message. It also covers meta-data, how it is specified, and how tools can use it to get information about the IWS message.

The IWS specification is a model for enabling a **container** to embed an **Interactive Web Service** containing many interacting **presentation** pages (served by an **Interactive Web Service Server**), into **container pages** (served by the **container**), so that a **user agent** shows both as one unit.

> In our example use case, the *Interactive Web Service* is a Web Service enabling the purchase of travelers checks. The *presentation pages* are the step-by-step wizard-like pages that enable the user to input the information needed to purchase them. The travel site is the *container* that inserts the Web Service into one of its travel purchase pages, which is shown in the browser (the *user agent*) as a seamless unit.

An Interactive Web Service exposes one or more IWS messages through which the container communicates with it. This overview presents the protocol, but does it piecemeal wise, not explaining all the parameters at once, but rather representing them one by one, in accordance with an explanation of the functionality they enable. Each section in this overview describes some functionality, and describes along with it the parameters of the IWS message, and the behavior of the container and IWS.

Presentation

An IWS Message is a SOAP message (usually, but not necessarily, transported over HTTP) sent by the container to an Interactive Web Service Server for the purpose requesting an action from the IWS, and action which results in the "presentation" (typically HTML, but can also be XML, which the container transforms into HTML via an XSL) of a page of the Interactive Web Service. This is analogous to standard Web applications in which actions like navigation or form submission result in another page being shown to the user.

This is the essence of an IWS message – send an **`action`**, receive the **`presentation`**, and embed it in some **container** presentation. So an IWS message has the form (in pseudo-code) of -

```
presentation = IWSMessage(action)
```

> In our example use case, the travel site sends a SOAP message using a SOAP client, requesting the "start" action – the entry point of the application. The resultant HTML is inserted into the container page.

A container can request any action it wants from the IWS, whenever it wants – receiving the resultant presentation. But an Interactive Web Service is not just a group of non-correlated actions. The real "driver" of these actions should be the user, who performs those actions while *interacting* with the page.

Interaction

To enable the user to interact with the IWS, and to still preserve the embedding in the container pages, all interactions of the Interactive Web Service *pass through* a **`controller`**. When the user interacts with a page and requests a server side action – by submitting a form or clicking a link – the

[1] For the semantics of and IWS message see below.

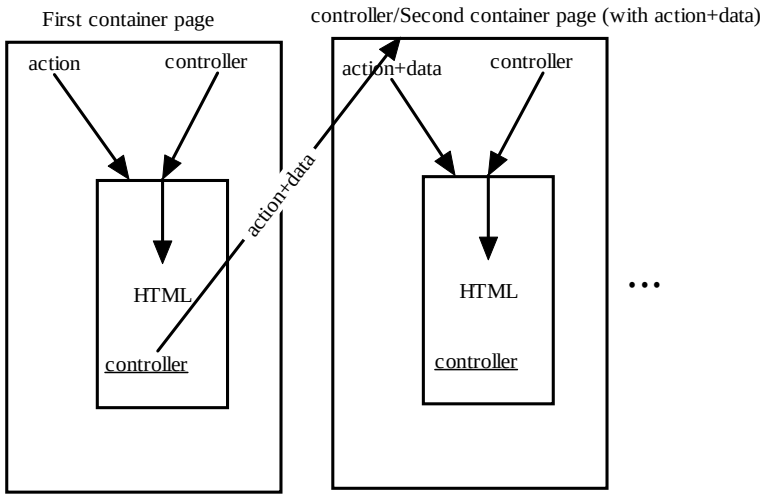
`action` and its **`data`** are sent to the controller (provided by the container application). The controller subsequently requests the IWS to perform the action and presents the resulting presentation to the user (usually embedded in the container application presentation). Further interactions with the page repeat the process.

To ensure that this happens, the Interactive Web Service points all the links and the forms in the **`presentation`** to the **`controller`**. This ensures that when the user clicks on a link or submits a form, the **`controller`** receives the **`action`** and the associated **`data`** so that it can request the IWS to perform the action. The **`action`** is usually transferred as a query parameter to the controller, and the **`actionData`** is received by the controller as "form data" in a POST to the **`controller`** URL. The controller is typically oblivious to the content of the **`action`** and the **`actionData`**.

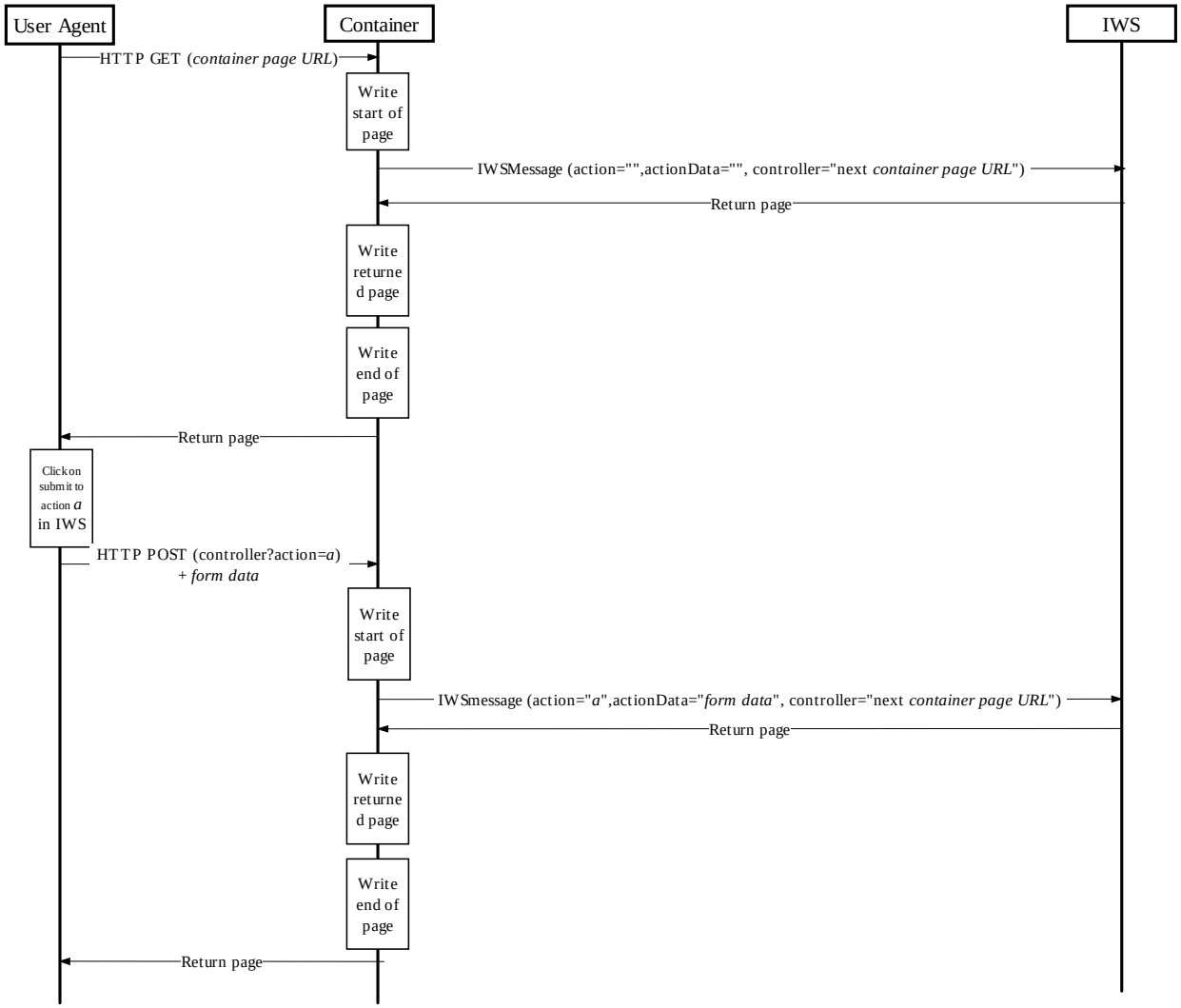
So an IWS message has the form (in pseudo-code) of -

```
presentation = IWSMessage(action, actionData, controller)
```

The flow in time looks like –



And in a sequence diagram –



In our example use case, the travel site can use the URL of the current page it is displaying to the user as the **`controller`**, so that all interactions would return to the same page that the user is in. This page ensures that the **`action`** received via the **`controller`** is transferred when re-sending the SOAP message.

Componentization

Interactive Web Services are components – building blocks for an interactive application. As such, they should have the attributes of components as the software industry has defined them for the past decades – **properties**, **events**, and **methods**. These entities of components enable a standard structure that helps build programming tools that are component-enabled. These programming tools read the component meta-data that describes these entities (see "Component Meta-data" below) and show GUI that enables input of entity values, thus enabling drag-and-drop construction of applications.

As methods are already inherent in Web Services (Web Services *are* technically a group of methods, called operations), Interactive Web Services augment the model with properties and events.

Properties in Interactive Web Services can enable customization of presentation and business logic. Using values set by the container, an IWS can change the presentation, and even the business logic of an IWS.

Events in interactive Web Services enable the IWS to communicate back with the container, *before* returning the page back, in order to send data or receive data. This enables the Web Service, for example, to retrieve user information from the container, and embed that information in the presentation.

As interaction with Interactive Web Services is usually over the Internet, it would be impractical to send a message on each setting of a property. Instead, the container sends all properties and event handlers on each method, in accordance with the coarse-grained philosophy of Web Services. Also, to be able to return values from an IWS message (for example, to "read" a property value), property values can be returned in the IWSMessage response..

The two component attributes – properties and events – are named **entities**.

So an IWS message has the form (in pseudo-code) of -

```
(presentation, properties) = IWSMessage(action, actionData, controller, properties,
events)

where…
properties = array of <name, value> pairs
events = array of <name, handlerUri> pairs
```

In our example use case, the travel site can adapt the Web Service to its requirements by defining the width of the presentation using the `Width` property, and can customize its look and feel to conform with its own by replacing the "next step" image presented at the top of every page with its own image with the `NextStepImage` property.

Also, the travel site can "auto-fill" user information in the Web Service by handling the `GetUserInformation` event and returning a SOAP response with the information it has about the user.

The Web Service also returns a `PageType` property which defines which step the user is in thus enabling the travel site to present different container pages (technically, by redirecting to different container URLs) based on the physical size of the Web Service page.

Final IWS Message Form

In the interest of performance, **`presentation`** should be the *last* parameter of the response, so that the container can output it directly while serving the container page. On the reverse direction, the same goes for **`actionData`**.

So an IWS message has the form (in pseudo-code) of -

```
(properties, presentation) = IWSMessage(action, controller, properties, events,
actionData)

where…
properties = array of <name, value> pairs
events = array of <name, handler> pairs
```

Component Meta-Data

To enable programming tools to use Interactive Web Services as they would use components, a component meta-model is defined.

First of all, the IWS must publish an XML Schema, and define the properties and events of the IWS (Schema's can be embedded in WSDL-s, so a WSDL is also OK). This schema is based upon the `IWS.xsd` which is presented below, and is very strict to enable tools to avoid rigorous parsing and understanding the precise semantics of Schema.

The programming tools read the XML Schema, and read the Schema annotation embedded in each property and event to determine various meta-data information.

The metadata information defined by IWS is –

1. Entity Type – given as a Schema type.
2. Entity Category – a URL defining the category. IWS defines some pre-defined categories.
3. Entity Category title – in case the tool does not recognize the Category URL, a title giving the human readable name can be used.
4. Event Schema – in the case of events, defines the schema for the event handler request and response messages. (In WSDL format; exact method TBD).

This meta-data enables the container tool to show a box with the list of properties and events, enabling the developer to set their values, and creates the code that calls the IWS message automatically.

> In our example use case, the travel site easily implements the above by importing the WSDL of the Web Service into its IWS-enabled LogicSphere v10 development environment, dropping the Web Service onto the HTML designer, using the property editor to customize the Web Service, and automatically generating the event handler functions that receive the event and can respond with the appropriate data.

State

Interactive Web Services usually have *state*. Because IWS pages are *served* (as HTML pages are), there must exist a mechanism for preserving state between interactions of the pages.

In normal Web pages, three mechanisms exist for that – the URL, hidden fields in forms, and cookies.

Preserving state in the URL means that all links between pages send the state in the URLs of the links themselves. This is a good thing because then *bookmarking* pages saves the state along with the bookmark. But, URL-s can save only a limited amount of information, and the programming of such a model is not really very easy.

Saving state in hidden fields means that all interactions between pages are must be form submissions where a hidden field is used to pass data between the forms. URL-s and form submits are not *natural* for state saving, as they were not built for such.

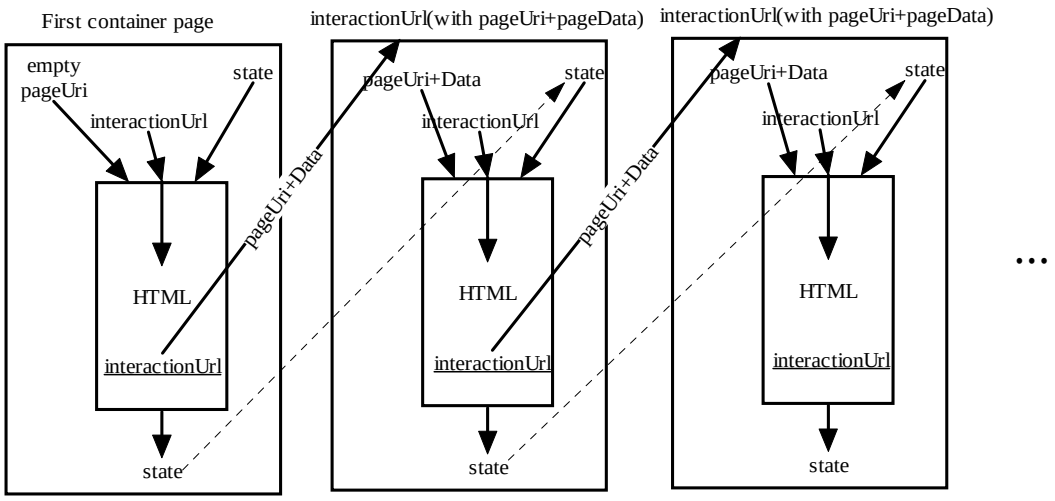
Cookies, on the other hand, were built expressly for this purpose. A cookie is a piece of data sent by the server to the user agent in order for the user agent to save it and present it back to the server on the next request. So, one page sends a cookie with the state, and the next page receives the state via the cookie. Cookies are usually grouped into two – session cookies, which are kept for the duration of a user-agent session, and persistent cookies, which are also preserved between sessions.

Interactive Web Services save state in ways that parallel the normal Web pages ways. The IWS can save state in the **`action`** and in the **`actionData`**. Because the IWS generates the **`action`** and

`actionData` for their next pages, state can be sent from one page to another, similarly to URL and hidden fields in forms.

To enable more structured state saving (a-la cookies), IWS defines two "standard" properties that can be passed and returned from an IWS state – **`SessionState`** and **`PersistentState`**. They are usually sent and received as standard properties.

And the diagram -



In our example use case, the vendor's multi-step buying process needs to use ***`SessionState`*** to save the user input from page to page, state which it passes in each response and gets back in the next request.

Implementing an IWS

An IWS can be easily implemented using standard Web development tools. The description below illustrates standard Java technologies as an example.

1. A servlet receives the SOAP message, and interprets it (using JAXM, for example), creating a JavaBean holding the information. The `action` indicates which JSP page to forward the request (with the bean) to. A simple convention can be used, where a URI in the form `action:`*`id`* forwards the request to */`*`id`*`.jsp`. The `SessionState` can be appended to the forwarded URL in the form of "*/`*`id`*`.jsp;jsessionid=`*`Sessionstate`*" so that JSP state support can be utilized transparently.
2. The JSP returns the SOAP response (using the JAXM tag library, for example), embedding the presentation in the `<presentation>` tag of the SOAP message, while doing the following –
    a. All links are rewritten to point to the `controller` received in the message, appending the correct `action` needed.
    b. Properties in the servlet-generated `JavaBean` are used to customize the presentation (using `<jsp:getProperty>` for example).

4. Detailed Specification

Abstract

An Interactive Web Service is a SOAP Web Service containing IWS Messages. An IWS Message is a SOAP message sent by a container to an Interactive Web Service for the purpose of performing an action and receiving the resultant presentation. This presentation can be embedded in the container presentation, and, when shown by a user agent, can request the controller to perform other actions – specifically navigating to, and interacting with, other pages of the Interactive Web Service - while still being shown in the user agent as contained in container pages' presentations, and without the need for the container page to be explicitly aware of the navigations and interactions between the Interactive Web Service pages.

An Interactive Web Service's Schema can contain meta-data for the service's "properties" and "events" so that programming tools can enable drag-and-drop programmability of Interactive Web Services.

The schema for an IWS message and response is given in Appendix A. An IWS message is not required to have an XML Schema, but is required to conform to it.

Terminology

- **Interactive Web Service**
  A SOAP Service containing Interactive Web Service Messages.
- **IWS**
  Acronym for Interactive Web Service. Is also the name of this specification.
- **IWS Message**
  A message in an IWS conforming to the IWS Schema of the element "`MessageBase`", and returning a response conforming to "`MessageResponseBase`".
- **Interactive Web Service Server**
  A server serving IWS messages.
- **Container**
  A Web application that consumes/subscribes/uses IWS.
- **action**
  An action that can be performed by the IWS, resulting in a presentation.
- **presentation**
  The result of an action is a presentation, which is usually embedded in the container
- **controller**
  A container-given URL, which receives all user's requests for actions and can request the IWS to perform those actions.
- **User Agent**
  A software that enables viewing page presentations, and interacting with them. This is usually a browser.
- **Property**
  A part of the component model of IWS, enabling customization of IWS and the returning of value to container
- **Event**
  A part of the component model of IWS, enabling sending and getting of data from container to IWS.

- **Entity**
  A property or event.

Naming Conventions

Types are in Pascal notation (initial caps) - `MessageBase`.

Elements are in Smalltalk notation (initial caps except for first word) – `actionData.`

Properties and events are in Pascal Notation[2], even though they appear as elements in XML.

Acronyms conform to notation and are not all uppercase – `IwsMessage`.

Namespace URI-s (and "typing URI-s) are all lowercase and in the form `http://type.domain.com/year/month/name` – `schemas.xmliws.org/2002/01/iws`.

"`xsd:`" refers to the namespace `http://www.w3.org/2001/XMLSchema`.

"`iws:`" refers to the namespace `http://schemas.xmliws.org/2002/01/iws`.

All URI parameters end in either `Uri` or `Url`, where a `Url`-ending parameter by convention means that the URI is interpretable by a User Agent, whereas a `Uri`-ending parameter means that the URI is an "abstract" namespacing-type URI.

IWS Message

The IWS message can be sent as any SOAP message is. The IWS MAY refuse to accept an insecure connection (e.g. HTTP) and return a `soap:Server.Iws.SecurityMismatch` to indicate this, and to allow the container to resend the message securely (e.g. HTTPS). In this case the message MUST be accepted by the IWS as if the first one was not sent.

Below is a list of all elements an IWS message MUST have.

`action`

Data that identifies an action of the IWS. The IWS Client SHOULD NOT process the **`data`** and should treat it as opaque, unless otherwise indicated by the IWS.

The IWS SHOULD enable empty `action`, indicating that the container requests a default (initial) page for the Web Service.

The IWS MUST accept any **`action`** produced by the IWS itself in the **`action`** parameter found in the IWS Client URL, up to a reasonable amount of time.

The IWS SHOULD gracefully handle **`action`**-s provided after a long period of time. It is assumed that in some cases, the **`action`** is part of a longer URL that may be bookmarked by the user. Thus, if the **`action`** cannot be processed (because, for example, the session embedded in the URI expired), the IWS is expected to return HTML that indicates it.

The IWS MAY accept **`action`** other than an empty one and ones produced by it.

In some implementations, the action is a simple HTTP URL and indicates the URL of a page of some underlying Web application that "supplies" the HTML and the application logic of the IWS.

`actionData`

OPTIONAL data that is sent along with the **`action`**.

The `actionData` MAY conform to the `application/x-www-form-url-encoded` mime type, but other forms of encoding are possible.

---

[2] To conform to conventions in existing component models.

actionData is usually data submitted to the **controller**.

actionData MUST be sent by the container along with the **action** it received in its **controller** if it received a form submit to its **controller** .

controller
A URL that defines where all requests for actions by the end user reaches.

The URL MUST be interpretable by the user agent. The user agent MUST receive requests for the `controller` after the string "`(action)`" is replaced (by the IWS) by an `action` that can be used in another IWS message to the IWS.

In most implementation, the IWS re-writes all links in the presentation returned by this message to point to `controller`, and replaces `(action)` with the URI of the page the link would have linked to.

The IWS MAY "upgrade" the security of a specific instance (in a link) of the `controller` if the data that are sent by that link should be sent securely (e.g., from http to https). Note that the IWS may not recognize all `controller-s` schemes, or that scheme may not have a secure "sibling" scheme. In this case, it may fail or put the `controller` as it is.

The container MUST respond to requests to the `controller`. The `controller` MUST accept "link" interactions and form submit interactions, UNLESS the Provider specifies that the IWS includes only one set of interactions.

The IWS MAY accept an empty `controller`. The IWS MAY implement this as navigating to a page external to the IWS subscriber, or it may infer the `controller` by other means (see `ContextUri`, for example).

properties [in]
A structure consisting of an array of `<name>value</name>` pairs containing values for properties. Each IWS SHOULD define a Schema element for this structure and use this element in the schema for the IWS messages in the IWS.

The **properties** element MAY have a different name (or namespace) as is it different from one IWS message to another. The name "**properties**" is a placeholder for these structures.

The IWS may use IWS pre-defined properties.

For any property of type "`iws:ContainerUrl`" (which is a subtype of `xsd:anyURI`) scanning and replacement of `(action)` in the value SHOULD be done before processing of the value.
Any property's "name" SHOULD conform to NMTOKEN restrictions, as defined in the XML standard.

The IWS SHOULD ignore properties it does not recognize.

The IWS specification does not define the semantics of the property values. In particular, it does not define whether setting properties is "persistent" or not, or even whether it is the same for all the properties of an IWS. This is IMPLEMENTATION-DEFINED.

events
A structure consisting of an array of `<name>handler</name>` pairs containing handlers for events.

The **events** element MAY have a different name (or namespace) as is it different from IWS message to another. The name "**events**" is a placeholder for these structures.

The **handler**-s MUST be interpretable by the IWS. It is assumed that the handlers are WSDL elements that describe SOAP messages, in a way to be defined. In this case, the events are SOAP

messages sent by the IWS "when the event happens". However, event handlers MAY also be standards URLs, and it is assumed that the IWS invokes them as needed.

IWS MUST send the events in the context of the message request, before or while returning the response, and data returned from them MAY be used to generate the response.

IWS Message Response

Below is a list of all elements an IWS message response MUST have.

`presentation`

The presentation resulting from `action`. The IWS specification DOES NOT define what the presentation language should be.

The IWS specification defines restrictions that an IWS SHOULD conform to – see Presentation Restrictions.

`properties [out]`

A structure consisting of an array of `<name>value</name>` pairs containing values for properties. Each IWS SHOULD define a Schema element for this structure and use this element in the schema for the IWS messages in the IWS.

The IWS container SHOULD ignore properties it does not recognize.

IWS SOAP Faults

All standard IWS SOAP faults are under the `soap:Client.IWS` or `soap:Client.IWS`. An IWS MAY define its own faults.

`soap:Server.Iws.SecurityMismatch`: If the SOAP message was sent insecurely (e.g. HTTP), but the request must be sent securely (e.g. HTTPS), as the data is sensitive, the IWS can return this fault. The container MAY resend the message in a secure way, and the IWS MUST accept it.

Pre-defined Entities

The pre-defined entities are NOT *reserved*, and an entity having that name DOES NOT imply that it has the same semantics, *unless* meta-data attribute `iwsSemantics` is true.

Pre-defined Properties

- **`ContextUri [in]`**
  An implementation-defined URI that identifies a "context" file. A context file includes static values of properties and static handler URL-s of events (a.k.a., a Property Sheet), and possibly other information regarding the handling of the Web Service inside the container page (e.g. *in a context*).
- **`SessionState [in,out]`**
  The session state is state information that SHOULD be saved by the container (**`SessionState`** is also returned as output) between container pages, and SHOULD NOT persist beyond a user's session. If the IWS returns a **`SessionState`** value, and does not receive it subsequent requests, its behavior is undefined (IWS-specific).
  In some implementation, IWS clients save this information in a session cookie of the container page.
- **`PersistentState [in,out]`**
  The persistent state is state information that SHOULD be saved by the container (**`PersistentState`** is also returned as output) between container pages, and SHOULD persist beyond a user's session. If the IWS returns a **`PersistentState`** value, and does not receive it in subsequent requests, its behavior is undefined (IWS-specific).

In some implementation, containers save this information in a persistent cookie of the container page.

- **`UserAgent [in]`**
  Enables the IWS to target its presentation to different user agents (including different devices). If the IWS expects this property, the IWS Client SHOULD set this property to the user agent which expects the data.
  In some implementations, containers set this value to the value of the User-Agent HTTP header they received in the request for a container page.
- **`IfModifiedSince [in]`**
  A date telling the IWS to return a 304 status code (with a `soap:Server.Iws.NotModified` fault) in case the output was not modified since this date. This enables caching of the page by the container.
  In case container implements caching of IWS pages, output properties SHOULD also be cached if they are processed by the container application.
- **`PageType [out]`**
  A value indicating the "type" of page returned. This can enable the container to redirect to another container page that should display IWS pages of that *type*.
- **`Expires [out]`**
  with semantics similar to the HTTP Expires header, this property specifies when the presentation expires, and enables caching of the page until that date.
  In case container implements caching of IWS pages, output properties SHOULD also be cached if they are processed by the container application.
- **`LastModified [out]`**
  with semantics similar to the HTTP Last-Modified header, this property specifies when the presentation was last modified, and enables caching of the page using If-Modified-Since.
  In case container implements caching of IWS pages, output properties SHOULD also be cached.

Pre-defined Events
[TBD]

Schema Specification
The schema for the IWS message and response is given in Appendix A. An example IWS schema is also given in Appendix B. An example message from the example IWS is given in Appendix D.

An IWS message DOES NOT have to provide a schema, but it is RECOMMENDED. The schema of an IWS Message SHOULD be used by programming tools to determine the component meta-data of that IWS. The target namespace is `"http://schemas.xmliws.org/2002/01/iws"`.
In case a schema is defined for the IWS message (or IWS message response), the following MUST be true.

- `Iws:[Respone]Message`
  MUST the base for any Schema of a compliant IWS message [response].
- The defined **`properties`** structure MUST be an extension of `iws:Properties` (or an extension of a type that is an extension of that type), and its `substitionGroup` MUST be `iws:properties`.
- Same for **events**.

Component Meta-data in Schema

- All [response] message schemas SHOULD include an `<iws:Iws[Response]MessageInfo>` appinfo annotation, to enable programming tools to determine which messages are IWS messages.
- Each ***entities*** structure defined by an IWS message SHOULD include an `iws:`*`Entities`*`sInfo` element in its annotation of the element and also include the `iws:`*`Entity`*`Info` (see below) on each property - to help programming tools understand the component meta-data.

`Entity`*`Info`*

- **`category`** attribute
  a URI defining the category of the entity. See "Pre-defined Categories" for a list of categories pre-defined by IWS.
- **`categoryTitle`** attribute
  in case a programming tool does not recognize the tool, category title is the human readable name of the category.
- **`iwsSemantics`** attribute
  true if one of standard IWS entities.

The *`Entity`*`Info`-s in the schema MUST be in the order they are to be passed in the ***`entities`*** structure, so that programming tools need not be read and understand the whole Schema. There are two *`Entity`*`Info`-s – `PropertyInfo` and `EventInfo` both of which extend the base `EntityInfo`

`PropertyInfo`

- **in** element
  The OPTIONAL element indicates whether the property MAY be sent in the request. Default=true.
- **out** element
  The OPTIONAL element indicates whether the property MAY be sent in the request. Default=false.

`EventInfo`

- **eventSchema** element
  The OPTIONAL element includes the schema for the message sent by the event.
- **eventResponseSchema** element
  The OPTIONAL element that includes the schema for the response returned from the event.

Using Meta-Data

A programming tool can use the pre-defined "`appinfo`-s" to read IWS message meta-data, without the need to read and understand the XML schema. In addition, the meta-data enables the programming tool to read information that is not present in the Schema itself.

First, the programming tool must search for `xsd:appinfo` (`xsd` is a shortcut for the Schema namespace). Embedded in it should be an element of the `iws` namespace defining which element it is – it can be:

- `Message[response]Info` – The element above is an IWS Message [response]. To get the name, search for the first XML element above it with a "name" or "ref" attribute.
- `Properties/EventsInfo` – the element above is a properties/… structure of the IWS message. This is needed by the programming tool, to determine the name of the element (which

may be different from message to message). To get the name, search for the first XML element above it with a “name” or “ref” attribute.

- `Property/EventInfo` - the element above is a property/…. The programming tool can determine the category and category title of the property from the attributes. To get the name, search for the first XML element above it with a “name” or “ref” attribute. To get the type of the entity, look for the type attribute of the XML element with the name.

Pre-defined Categories

- General
- Presentation
- Interaction
- User Agent
- State
- HTTP

Presentation Restrictions

1. The Presentation SHOULD be embeddable. The IWS MAY provide a restricted IWS presentation that is not embeddable in any location – for example, “never inside an absolutely positioned layer”, or “always inside a table”. However, the IWS specification does not define any syntactic mechanism to define such restrictions.
2. The returned HTML SHOULD be well-behaved. Meaning: it SHOULD NOT temper with any containing data, such as form input; any content in the presentation SHOULD NOT interfere with content outside (for example, in HTML, do not defined a style on paragraphs- `P {...}`, as it may override the container’s style); and others…