Web Services for Remote Portlets Specification

Working Draft 0.85, 26 November 2002

Document identifier:

WSRP_Specification-v0.85 (Word)

Location:

http://www.oasis-open.org/committees/wsia

http://www.oasis-open.org/committees/wsrp

Editors:

Alan Kropp, Epicentric, Inc. <akropp@epicentric.com>
Carsten Leue, IBM Corporation <cleue@de.ibm.com>
Rich Thompson, IBM Corporation <richt2@us.ibm.com>

Contributors:

Chris Braun, Novell <cbraun@silverstream.com>
Jeff Broberg, Novell <jbroberg@silverstream.com>
Mark Cassidy, Netegrity <mcassidy@Netegrity.com>
Michael Freedman, Oracle Corporation <Michael.Freedman@oracle.com>
Timothy N. Jones, CrossWeave <tim@crossweave.com>
Thomas Schaeck, IBM Corporation <schaeck@de.ibm.com>
Gil Tayar, WebCollage <Gil.Tayar@webcollage.com>

Abstract:

Integration of remote content and application logic into an End-User presentation has been a task requiring significant custom programming effort. Typically, vendors of aggregating applications, such as a portal, had to write special adapters for applications and content providers to accommodate the variety of different interfaces and protocols those providers used. The goal of this specification is to enable an application designer or administrator to pick from a rich choice of compliant remote content and application providers, and integrate them with just a few mouse clicks and no programming effort.

This specification is a joint effort of two OASIS technical committees. Web Services for Interactive Applications (WSIA) and Web Services for Remote Portlets (WSRP) aim to simplify the integration effort through a standard set of web service interfaces allowing integrating applications to quickly exploit new web services as they become available. The joint authoring of these interfaces by WSRP and WSIA allows maximum reuse of user-facing, interactive web services while allowing the consuming applications to access a much richer set of standardized web services.

This joint standard layers on top of the existing web services stack, utilizing existing web services standards and will leverage emerging web service standards (such as security) as they become available. The interfaces are defined using the Web Services Description Language (WSDL).

Status:

This draft is an early version of the public spec. Various concepts continue to be debated. Points needing clarification as this evolves into the final specification are much appreciated and may be emailed to *Rich Thompson*.

If you are on the wsia@lists.oasis-open.org or wsrp@lists.oasis-open.org list for committee members, send comments there. If you are not on that list, subscribe to the wsia-comment@lists.oasis-open.org or wsrp-comment@lists.oasis-open.org list and send comments there. To subscribe, send an email message to wsia-comment-request@lists.oasis-open.org or wsrp-comment-request@lists.oasis-open.org with the word "subscribe" as the body of the message.

The errata page for this specification is at http://www.oasis-open.org/committees/wsrp/specification_v1_errata.html.

Table of Contents

1 Introduction

Both Web Services for Interactive Applications (WSIA) and Web Services for Remote Portlets (WSRP) define a web service interface for accessing and interacting with user-facing, interactive presentation-oriented web services.

This specification defines the joint WSIA/WSRP interfaces. It is based on the requirements gathered by both committees and on the concrete proposals to both committees.

Scenarios that motivate WSRP/WSIA functionality include:

- Portal servers providing portlets as user-facing web services that can be used by aggregation engines.
- Portal servers consuming user-facing web services provided by portal or non-portal Producers and integrating them into a portal framework.

However this description also applies to non-portal environments, mostly identified by the WSIA use cases[1]. For a detailed overview of Web Services, Portal Environments and the application of WSRP to these environments, please refer to the ***[WSRP Whitepaper]*** and additional documents at http://www.oasis-open.org/committees/wsrp/.

*This specification accounts for the fact that Producers (web services conforming to this specification) and Consumers (applications consuming Producers in a manner conforming to this specification) may be implemented on very different platforms, be it as a Java/***[Boyer-Moore]** *http://www.cs.utexas.edu/users/moore/best-ideas/string-searching/*

[DIME] *http://www.ietf.org/internet-drafts/draft-nielsen-dime-02.txt*

[J2EE] *based web service, a web service implemented on Microsoft's* **[JSR168]** *http://www.jcp.org/jsr/detail/168.jsp*

[.Net] platform or a portlet published directly by a portal [A100]. Special attention has been taken to ensure this platform independence.

These web services are built on standard technologies, including **[SSL/TLS]** *http://www.ietf.org/html.charters/tls-charter.html*

[URI/URL] *http://www.ietf.org/rfc/rfc2396.txt*

[WSDL] and ***[SOAP]***, and will leverage future applicable Web Service standards, such as WS-Security and WS-Policy (see section 3.1) [A102].

1.1 Motivation

Portals render and aggregate information from different sources and provide it in a compact and easily consumable form to an End-User. Typically, this information consists of markup fragments that are surrounded by a decoration that contain Portal-inserted controls (e.g. minimize and maximize buttons). The whole construct is commonly referred to as a “portlet” and the content as “markup” or “markup fragment”.

[1] http://www.oasis-open.org/committees/wsia/use_cases/index.shtml

Among typical sources of information are web services. Traditional data-oriented web services, however, require aggregating applications to provide specific presentation logic for each of these web services. Furthermore, each aggregating application communicates with each web service via its unique interface. This approach is not well suited to dynamic integration of business applications and content as a plug-and-play solution.

This specification solves this problem by introducing a user-facing web service interface that allows the inclusion of and interaction with content from a web service. Such a user-facing web service provides both application logic and presentation logic. This specification provides a common protocol and a set of interfaces for all user-facing web services. Thus, aggregating applications can easily adopt these web services by utilizing generic proxy code.

1.2 Actors

This protocol describes the conversation between Producers and Consumers on behalf of End-Users (clients of the Consumer). Producers provide user-facing web services that are able to render markup fragments and process user interaction requests. Consumers use these web services to present the generated markup to End-Users and manage the user's interaction with the markup.

1.2.1 Producer

Producers are modeled as containers of the actual content generators (e.g. portlets from the portal scenario). These content generators are called entities by this specification. The Producer provides a set of interfaces, including:

- *Self description:* A required interface that allows Consumers to find out the capabilities of the Producer and about the entities it hosts, including the metadata necessary for a Consumer to properly interact with each entity.
- *Markup:* A required interface used to render and interact with markup fragments.
- *Registration:* An optional interface used to establish a relationship between a Producer and a Consumer (e.g. for billing or book-keeping purposes).
- *Entity management:* An optional interface that grants access to the life-cycle of the hosted entities. This interface also includes *Property management,* which enables programmatic access to an entity's persistent state.

In order to allow different levels of sophistication for both the Producer and Consumer, parts of this functionality are optional. This specification contains various examples of how a Producer might implement particular functionality for varying levels of sophistication and with regards to implementing some of the optional portions of the protocol.

A particular entity is identified with an `entityHandle`. The Consumer uses `entityHandles` throughout the communication to address and interact with entities via the Producer. The entities a Producer publishes as available for all Consumers to interact with are called "Producer_Offered_Entities". Producer_Offered_Entities are pre-configured and not modifiable by Consumers.

If the Producer chooses to expose the *entity management* interface, it is allowing Consumers to clone the entities offered by the Producer and customize those cloned entities. Such a uniquely configured entity is called a "Consumer_Configured_Entity". Like Producer_Offered_Entities, an `entityHandle` is used to address Consumer_Configured_Entities. This `entityHandle` is both; 1) invariant until released, and 2) unique within and scoped by the supplied `registrationHandle`.

Besides entity management, the Producer optionally manages Consumer *registrations*. The Producer may require Consumers to register prior to discovering and interacting with entities. A registration represents a relationship (often including both technical and business aspects) between the Consumer and Producer.

1.2.2 Consumer

A Consumer is an intermediary system that communicates with user-facing web services (i.e. Producers and the entities they host) on behalf of its users. It gathers and aggregates the markup delivered by the entities and presents the aggregation to the End-User. One typical Consumer is a portal, which mediates the markup and the interaction with this markup between End-Users and user-facing web services. Another typical Consumer is an e-Commerce application that aggregates manufacturer-provided content into its own pages.

While this specification is neutral as to the markup used to represent the user interface to the End-User, we note that general performance concerns favor markup technologies that push the processing of user interface logic, such as the validation of End-User input, as far toward the user agent as possible. XForms[2] represents a markup technology that can be leveraged to address these performance concerns.

1.2.3 End-User

The main purpose of a Consumer that aggregates content from various Producers/entities is the preparation and presentation of markup to an End-User. In addition, the Consumer needs to manage the processing of interactions with that markup in order to properly correlate the interactions with the stateful environment that produced the markup. This specification defines the operations, **getMarkup()**, **processInteraction()**, and **processBlockingInteraction()**, for this purpose:

- **getMarkup()** is invoked to obtain the markup fragments from an entity. The markup returned depends on things such as the entity's current state, the user context, the markup type requested, etc.
- **performInteraction()** is invoked when an End-User interacts with the markup from the entity. This interaction may result in a state change of the entity, which often causes changes in the markup returned on a subsequent **getMarkup()** call.
- **performBlockingInteraction()** carries all the same semantics as **performInteraction()**, but in addition causes the Consumer to block both the streaming of its own markup to the End-User and the gathering of markup from other entities until this operation finishes.

1.3 Typical Process Flow

While some of the following steps are optional, the typical flow of interactions between these actors is:

[2] http://www.w3.org/TR/xforms/

1. Establishment of a relationship between the Consumer and Producer. This may involve the exchange of information regarding capabilities, security requirements or other business and/or technical aspects of the relationship.
2. Establishment of a relationship between the Consumer and End-User. This permits the Consumer to authenticate the End-User and may allow the End-User to personalize the aggregated pages presented by the Consumer.
3. Production of aggregated pages. This typically involves the Consumer defining some base level of page design (often with customized entities) and may involve further customization of those pages by the End-User.
4. Request for a page. This typically results when the End-User directs an agent (e.g. browser) to the Consumer's URL, but also occurs indirectly as a result of processing an interaction with the markup of a previous page.
5. Processing interactions. Some End-User interactions with the markup of a page will result in an invocation on the Consumer to provide some logical function. The Consumer will process this invocation to determine the Producer/entity that the interaction has targeted and the nature of the invocation on that entity that has been requested. Since the resulting invocation of that entity is likely to change its state (and may also change the state of other entities), the Consumer must also treat this as an indirect request for a page and thereby loop back to step 4.
6. Destruction of relationships. Much as new relationships are formed, at times relationships end. The protocol provides means by which the Producer and Consumer may inform each other that the relationship (or some portion of it) has ended and that related resources may be cleaned up.

1.4 Example Scenarios

This specification supports Consumers and Producers of various levels of sophistication interacting with one another. While not exhaustive, the following scenarios represent examples of the broad range of possibilities.

1.4.1 SimpleProducer

Does not support registration or persistence. May only offer one type of entity.

Examples:

- A flight schedule display that is publicly available. Neither user registration nor persistent state is required. The entity may, however, maintain interaction state using a session.
- News feed web service that allows the user to browse news topics.

1.4.2 SophisticatedProducer

Requires Consumers to register and supply the returned reference on all future invocations. Provides metadata relevant for interacting with the web service. Supports a number of entities, which publish metadata that declare the supported markup types and properties for interacting with the entity.

Example:

- Portal server that exposes portlets available through a compliant service endpoint. Each portlet may contain End-User and Consumer-specific settings and information that are persisted on the portal server.

1.4.3 SimpleConsumer

Does not persist any registration/entity information across restarting of the Consumer. Have explicit declarations for binding to and interacting with a set of Producer services.

1.4.4 SophisticatedConsumer

Supports the persistence of Producer, Consumer and End-User related data. Supports single sign on for its End-Users (may require End-User to trust Consumer with sign-on data). May support discovery of new Producers by either Administrators and/or End-Users.

Example:

- Typical portal server that access WSIA/WSRP services for content aggregated onto pages.

1.4.5 Interaction between levels of sophistication

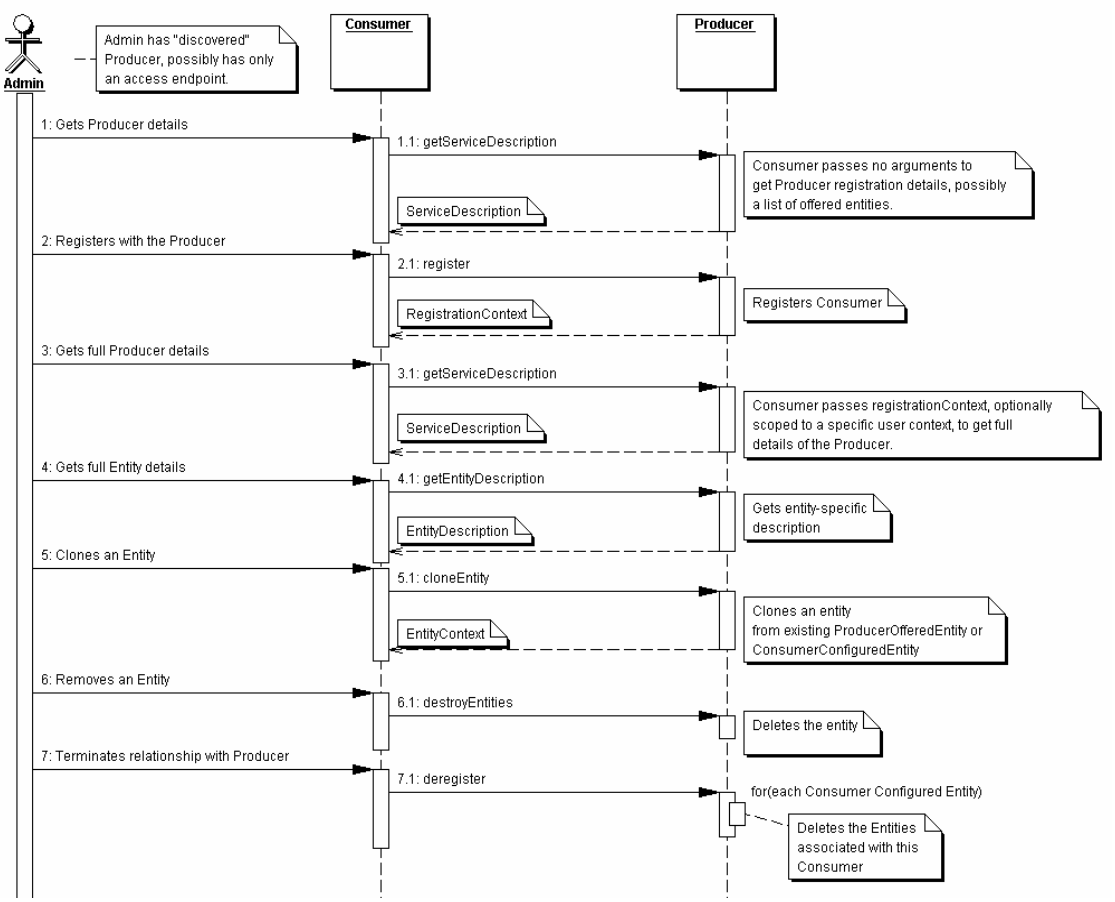
The following illustrate how the interaction between the various parties (End-Users, a Consumer and a Producer) might flow for each of the combinations of these example scenarios.

1.4.5.1 Sophisticated Consumer / Sophisticated Producer (stateful +configurable)

These interactions span the entire protocol as this Consumer/Producer pair both support and exploit as much of the protocol as possible.

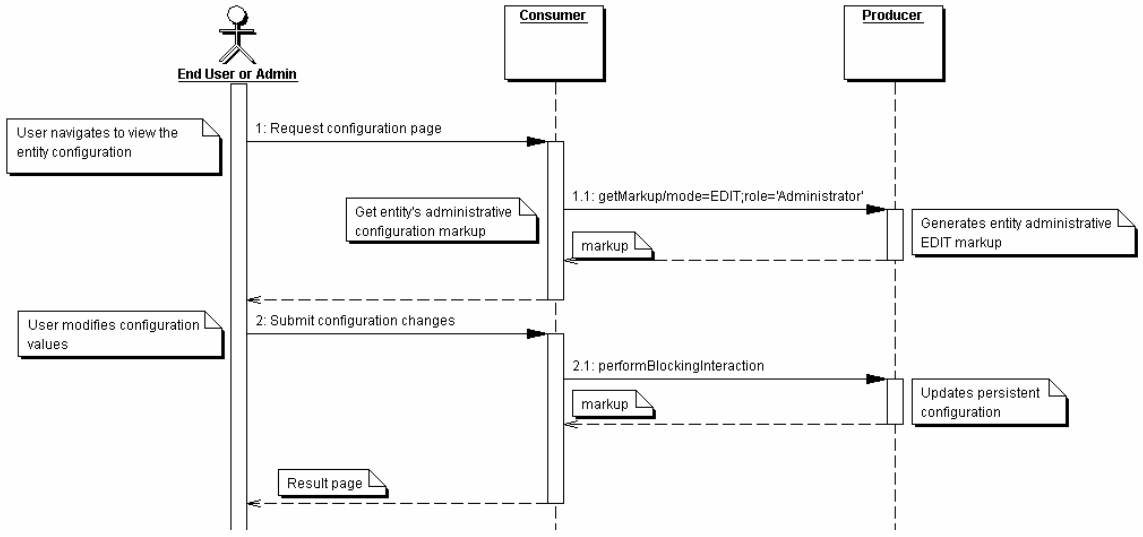
1.4.5.1.1 Administration

Administration involves both the technical and business sides of the relationship between the Consumer and Producer. In particular it involves establishing the relationship, cloning the entities the Producer offers for the Consumer to configure and eventually destroying both those configured entities and the relationship itself. The protocol has operations for each of these tasks as depicted below.

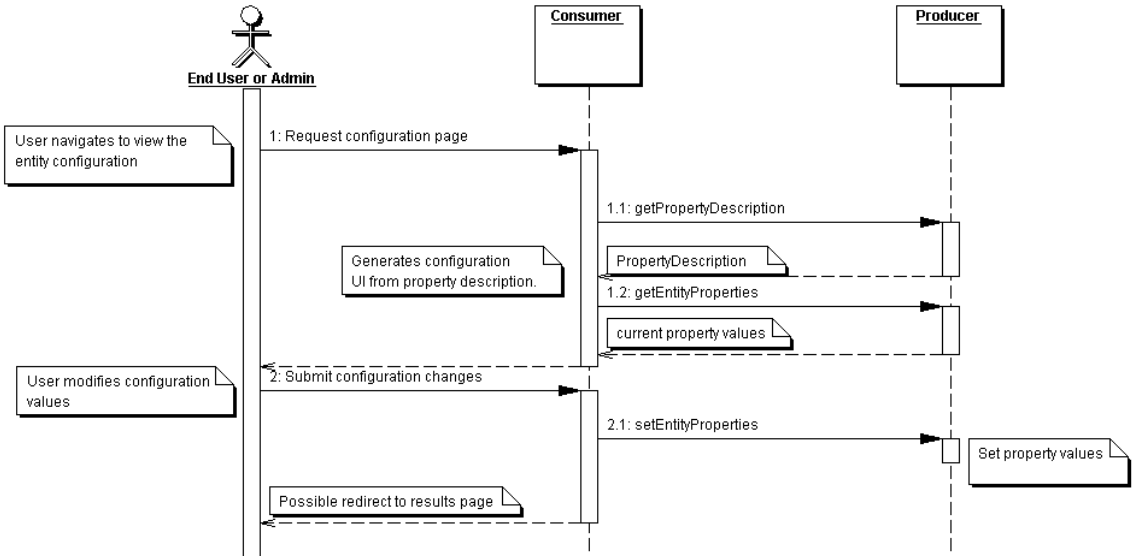


1.4.5.1.2 Configuration

Configuration can occur either using an entity-generated user interface or a consumer-generated interface. First we consider the entity-generated case.



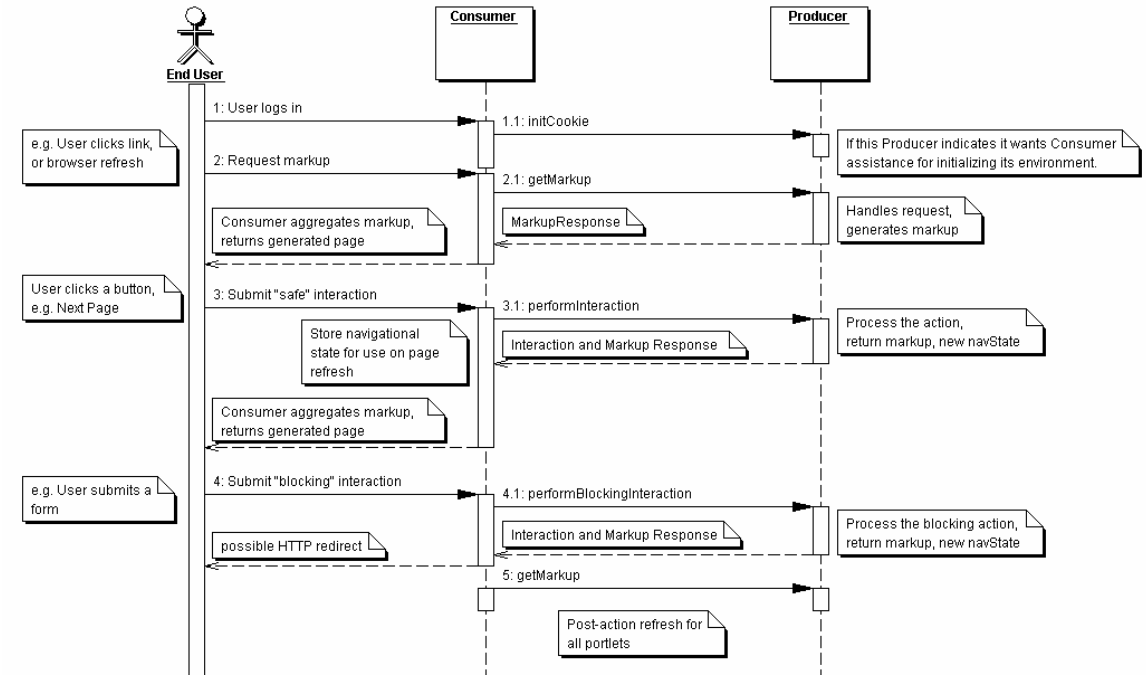
The other possibility the protocol enables is consumer-generated interfaces that interact with an entity's properties through their description and current values. Note that the Producer might not expose all configurable items as properties.



5

1.4.5.1.3 End-User Interactions

The flow of End-User interactions through the Consumer and Producer using the protocol is depicted below.

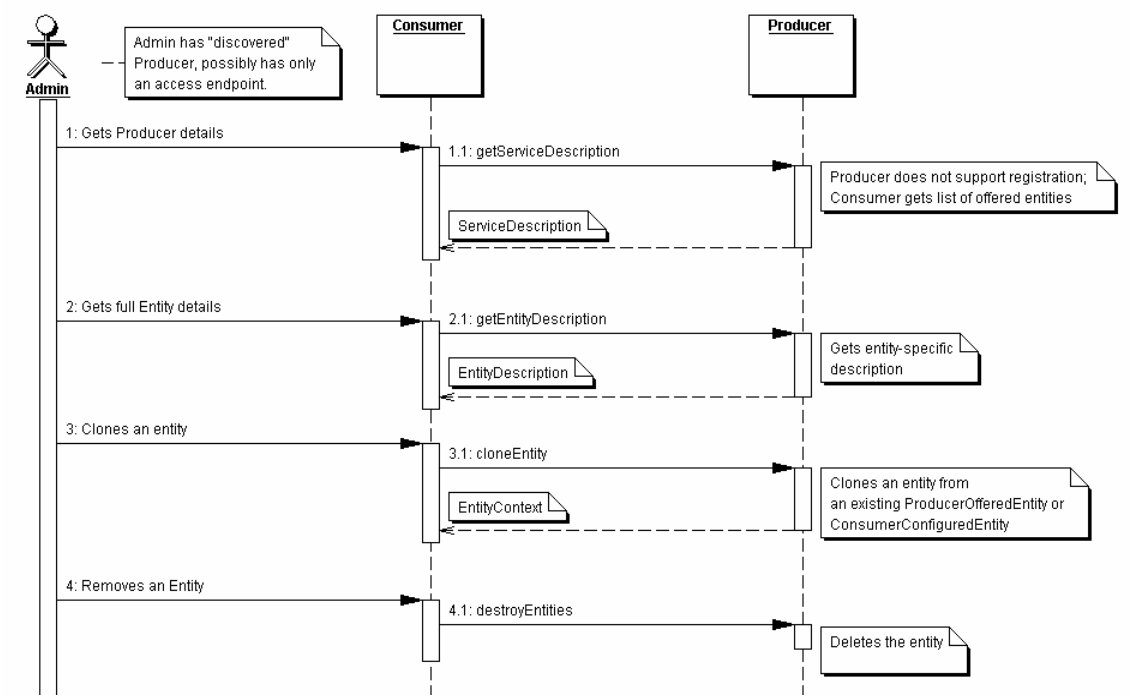


10

1.4.5.2 Sophisticated Consumer / Simple Producer (stateful)

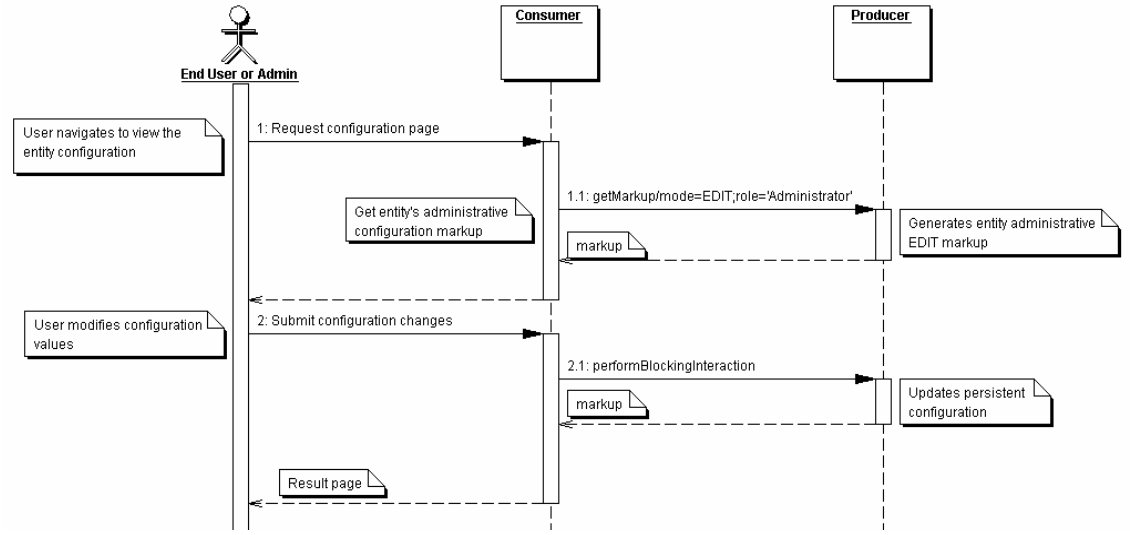
1.4.5.2.1 Administration

Since the Simple Producer example does not offer the registration interface, any aspect of registration that is required occurs outside the protocol. Other portions of the protocol remain unchanged, though invocations such as **cloneEntity()** now return their full state to the Consumer as the Producer does not offer persistence to its entities. Producers choosing this level of functionality should note the security implications of this choice and be implemented accordingly.

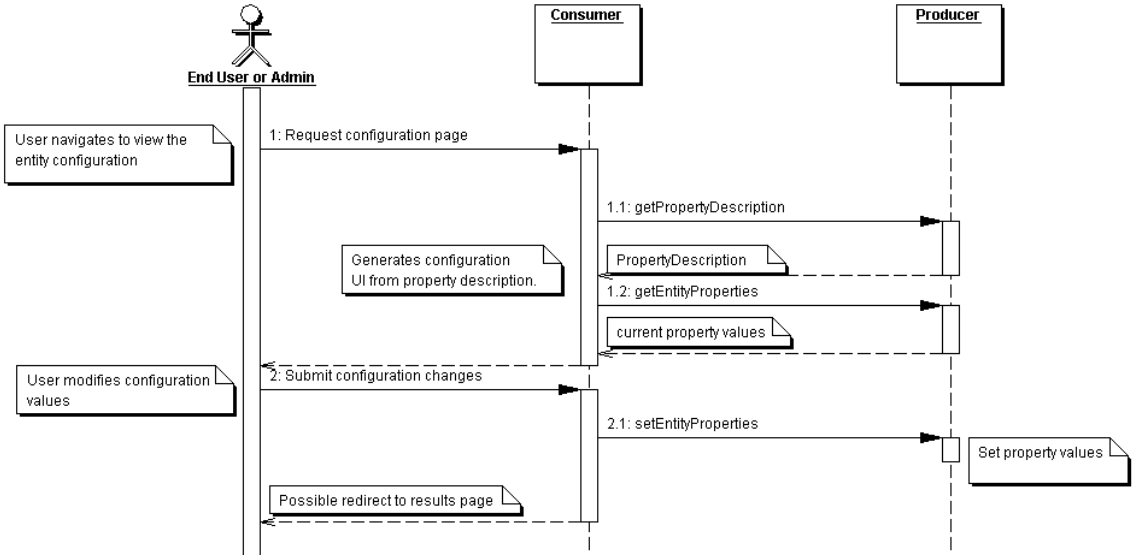


1.4.5.2.2 Configuration

Configuration can occur either using an entity-generated user interface or a consumer-generated interface. First we consider the entity-generated case.



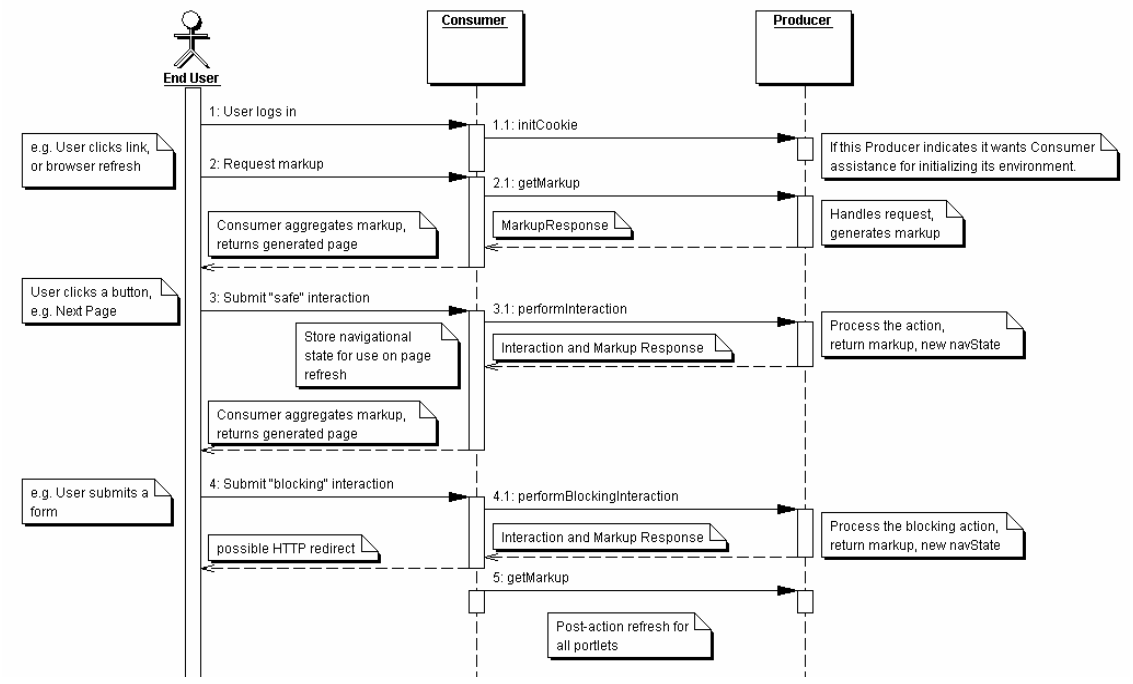
The other possibility the protocol enables is consumer-generated interfaces that interact with an entity's properties through their description and current values.



5

1.4.5.2.3 End-User Interactions

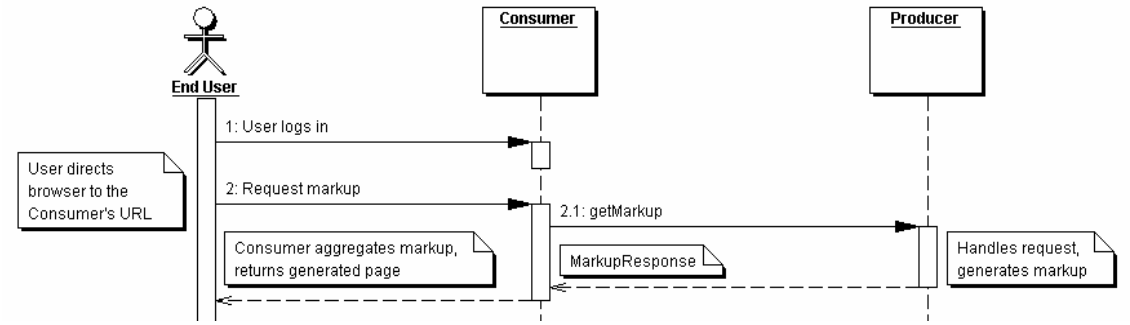
The flow of End-User interaction through the Consumer and Producer using the protocol is depicted below.



10

1.4.5.3 Simple Consumer / Simple Producer (no state)

The interactions between this pair of actors are limited to the generation of markup based on the state carried by the request for markup itself. In general, entities of this type encode everything required to generate the markup on the URL causing the invocation of **getMarkup()**. Often these entities involve only a single page, but could provide links on that page that cause the generation of a completely different markup based on the parameters passed when the link is activated. Invocations of **performInteraction()** and **performBlockingInteraction()** MAY happen in this scenario if the entity impacts some backend system as a result of the invocation, as this impact could change the markup some other entity will generate. This is depicted below:

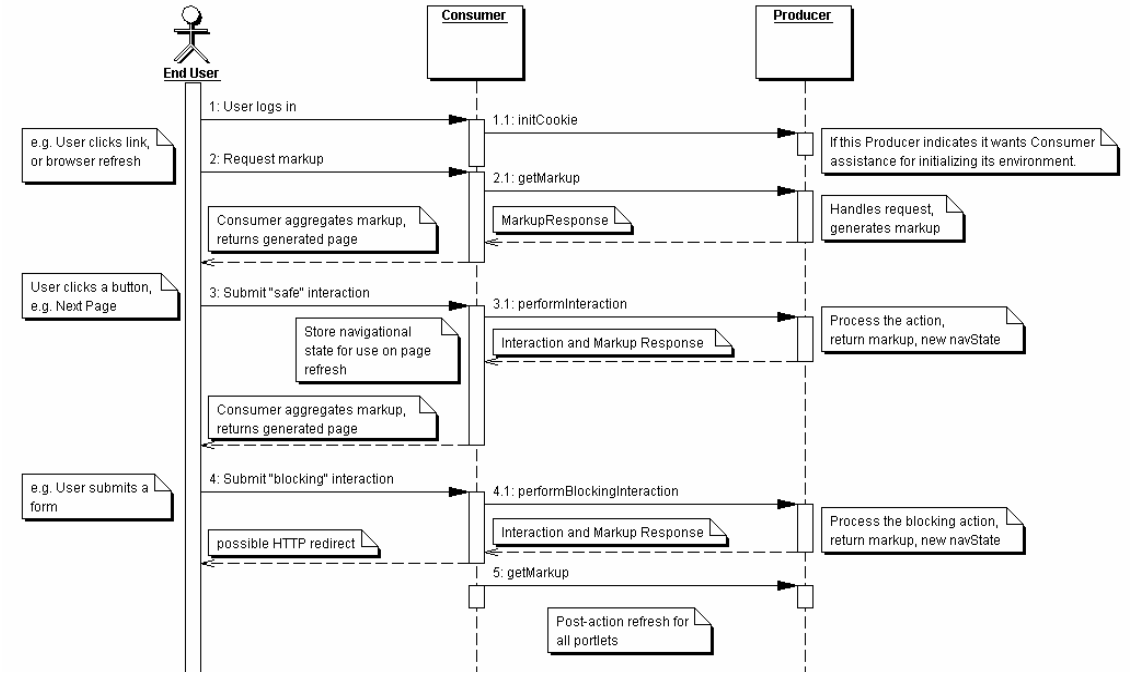


1.4.5.4 Simple Consumer / Simple Producer (with state)

In addition to the previous example, this pair of actors manage the processing of runtime state with the Producer managing that state and the Consumer supplying the means to reference that state when processing End-User interactions.

1.4.5.4.1 End-User Interactions

The flow of End-User interaction through the Consumer and Producer using the protocol is depicted below.

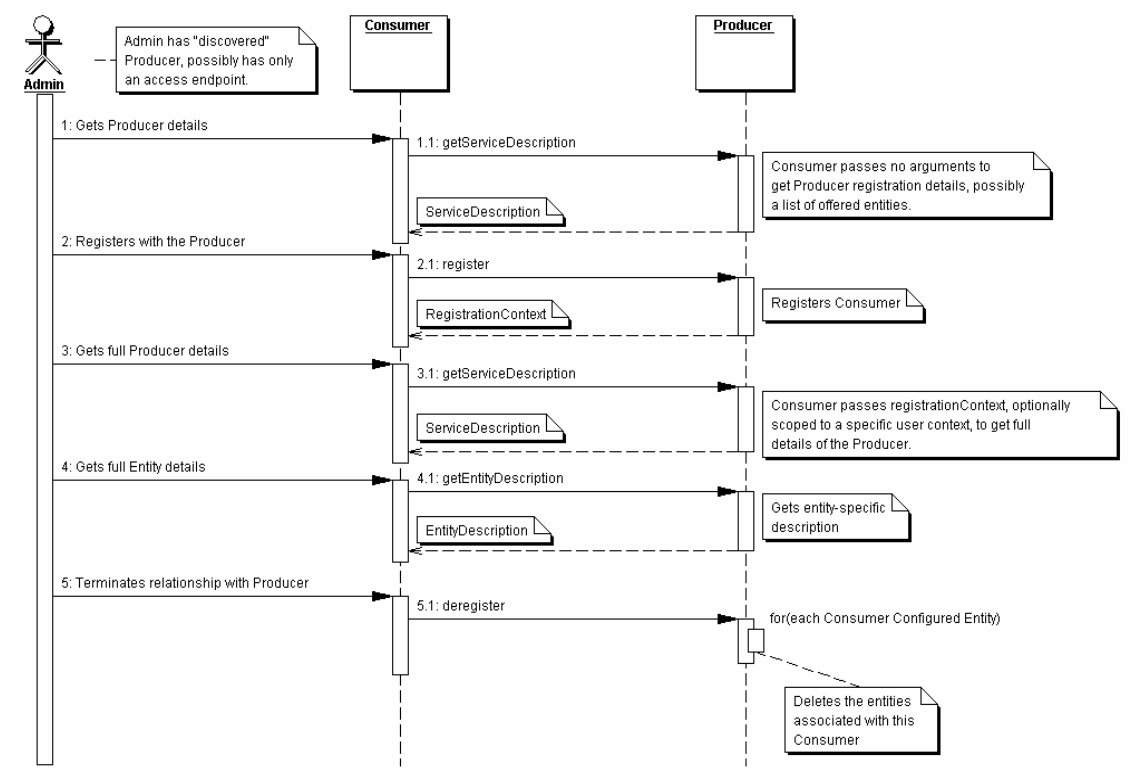


1.4.5.5 Simple Consumer / Sophisticated Producer (stateful+configurable)

While this Consumer example offers minimal services to the Producer, a sophisticated Producer is able to provide a rich set of functionality to the entities it hosts.

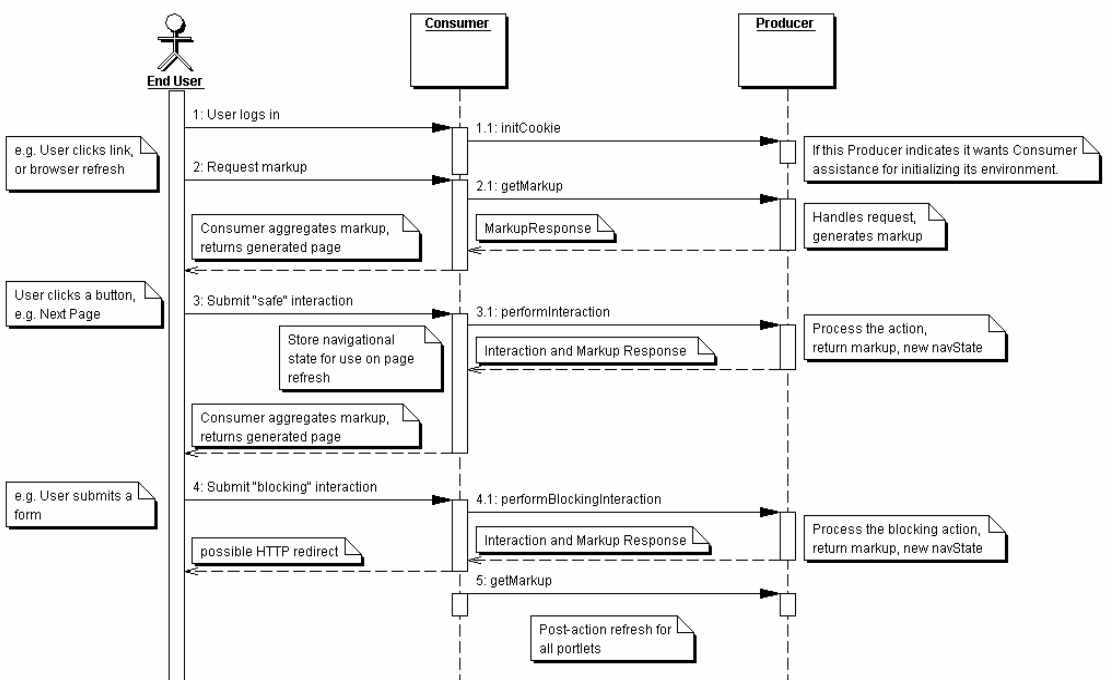
1.4.5.5.1 Administration

Since the Simple Consumer example does not persist registration information, the registration aspect of the relationship with the Producer must be created and destroyed on each set of interactions.



1.4.5.5.2 End-User Interactions

The flow of End-User interaction through the Consumer and Producer using the protocol is depicted below.



2 Terminology

The key words MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, MAY, and OPTIONAL in this document are to be interpreted as described in **[Character *Sets]*** http://www.iana.org/assignments/character-sets

[Namespaces] *http://www.w3.org/TR/REC-xml-names/*

[RFC2119].

Compliance: Mandatory – relevant to legal rules, regulations or laws. Compliancy is the act of complying with a specification and/or standard. Example: ISO 9001. IEEE defines as complying with laws and regulations.

Conformance: Not mandatory – ISO/IEC Guide 2 defines conformance or conformity as fulfillment of a product, process or service of specified requirements. Note that many times providers use “comply” to a standard to sidestep because they don’t actually “conform” to a standard. Reasons they do not “conform” often include the standard is not approved yet or that the provider does not actually meet the standard’s conformance requirements.

Cross references to the **[P3P]** *http://www.w3.org/TR/P3P/*

[Requirements] developed by both the WSIA and WSRP technical committees are designated throughout this specification by a hyperlink to the requirement contained where the requirement number is enclosed in square brackets (e.g. [A100]).

3 General Design Issues

The major design goals of this specification are simplicity, extensibility and efficiency.

3.1 Related Standards

This specification seeks to leverage both existing and emerging web service standards whenever possible. The following are particularly noted as relevant standardization efforts:

3.1.1 Existing Standards

WSDL – Defines how abstract interfaces and their concrete realizations are defined.

Schema – Defines how types are defined and associated with each other.

Namespaces – Defines how XML Namespaces are declared and used.

SOAP – Defines how to invoke remote interfaces.

SSL/TLS – Defines secure transport mechanisms.

URL – Defines URI (includes URL) syntax and encoding

Character Sets - Character set encoding

XML Digital Signatures – Defines how portions of an XML document are digitally signed.

SAML – Defines how authentication and authorization information may be exchanged.

XACML – Defines a syntax for expressing authorization rules.

P3P – Defines how a Producer/entity may publish its privacy policy so that a Consumer could enforce End-User privacy preferences.

3.1.2 Emerging Standards

XML Encryption – Defines how to encrypt/decrypt portions of an XML document.

WS-Security – Defines how document level security standards apply to SOAP messages.

RLTC – Defines a syntax for expressing authorization rules.

XCBF – Defines how to exchange biometric data.

WS-Attachments - Defines how to encapsulate a SOAP message and zero or more attachments within a DIME message.

WS-I.org - Defines profiles for use of web services standards such that interoperability is maximized.

DIME – A lightweight, binary message format that encapsulates one or more resources in a single message construct.

JSR168 – Java Community Process for standardizing a portlet API.

3.2 Data Objects

It is often necessary to pass data to operations. Typed data objects are defined as the transport mechanism wherever possible. The Schema definitions of these structures includes the <any namespace="##other"/> construct as a standard means for data extensions. Producers/entities employing these extensions SHOULD provide typing information for the extended data items [A505]. The preferred means for this typing information includes using the schema defined[3] "type" attribute to reference the correct schema on each such extension element, and use of either the Producer's WSDL (default) or a "schemaLocation" attribute as per standard schema usage to declare the details of all non-simple types. This allows Consumers to provide type checking outside of that done by typical interface layers. This specification introduces various data structures as they are needed for operations and then summarizes them all in section 11.

[3] http://www.w3.org/TR/xmlschema-1/#xsi_type

3.3 Lifecycles

"Lifecycle" is a term used to describe how items become available, are interacted with, and finally are destroyed. The two lifecycles included in this specification are:

Persistent: This lifecycle starts with an explicit operation to create the item and ends only with an explicit operation to destroy the item. Examples include the `registrationHandle` and Consumer_Configured_Entities.

Transient: This lifecycle can either start with an explicit operation OR as a side effect of some other operation [A204]. The item created is transient and no explicit operation is required to destroy it. This specification generally includes an `expires` element (a duration in seconds) whenever such an item may be created so that any resources at the Consumer related to the item may be reclaimed at an appropriate time. An example of this is session creation.

3.4 Scopes

Scope is a term used to describe when something is valid. An item often scopes both the usage and lifecycle of other items. Scopes that are referenced in this specification are:

Registration scope: This scope is initiated when a Consumer registers with a Producer and ends when the handle referring to that registration is released. As such it encompasses any entities the Consumer configures and any interactions with the entities of the Producer. From the Producer's perspective, this scope has a persistent lifecycle, as the Consumer MUST explicitly invoke **deregister()** to terminate a registration scope. This scope is referenced throughout the protocol using a `registrationHandle`. The Producer optionally exposes this scope by declaring support for the Registration portType. If the Producer exposes the Registration portType, then the Consumer MUST respect the registration requirements established by this specification.

Entity scope: This scope is initiated when an entity is cloned and as such will be encapsulated by a registration scope (which will be null if the Producer does not support registration). This scope ends when the reference to the entity is explicitly released. As such it encompasses all interactions with the entity. This scope has a persistent lifecycle and is referenced using an `entityHandle`. The Producer optionally exposes this scope by declaring support for the EntityManagement portType. If the Producer exposes the EntityMangement portType, then the Consumer MAY clone the Producer_Offered_Entities and uniquely configure them for its own use. The Consumer MAY also choose to directly use the Producer_Offered_Entities.

Session scope: This scope is initiated when an entity needs to store transient state on the Producer and is always encapsulated by the entity's scope. This scope ends when the session holding that state is released (either via an explicit operation on the Producer OR via a timeout mechanism). As such it encompasses a set of operation invocations in which the Consumer has supplied the refined entity handle (Producer-generated) that also encodes the session. This scope has a transient lifecycle and is established by the Producer returning a new `SessionContext`. The Consumer MUST respect this new scope as described in section 0.

3.5 Types of Stateful Information

Because WSIA and WSRP are connectionless protocols, the Producer must be able to return information to the Consumer, with the understanding that this information will be sent back to it [A200]. Three types of stateful information exist:

Navigational state: This is the state that allows the current page to be correctly generated, including on a page refresh. Web applications typically store this type of state in the URL so that both page refresh and bookmarked pages will generate what the End-User expects. The Producer returns this state to the Consumer as `navigationalState` such that it may satisfy these expectations of the End-User. To supply the bookmarking capability End-Users expect, the Consumer may store this state, or a reference to it, in the URL. The Consumer may also choose to not supply this functionality to its End-Users.

Transient state: This is state stored on the Producer related to a sequence of operations (for example, an e-Commerce site may store a shopping cart in its transient state). Once this type of state is generated, the Producer returns a reference to it and the Consumer must return this reference on future invocations as described in section 5.1.2. This type of state will be referred to as a **Session** (similar to an HTTP Session) and an opaque reference to one is a `sessionHandle`.

Persistent state: This is state that the Producer persists until either the Consumer or Producer explicitly discards it. This specification defines two kinds of persistent state with each referred to via a handle that MUST remain invariant once the Producer supplies it to the Consumer. This simplifies a number of issues related to Consumer processing when changes occur relative to a particular persistent state reference. These two kinds of persistent state are:

> **Consumer Registration:** Represents a relationship between a Consumer and Producer. Data that is part of the Consumer registration state impacts all invocations within the scope of the registration. The opaque reference to Consumer registration state is referred to as a `registrationHandle`**.**
>
> **Entity:** In addition to the entities a Producer offers for all Consumers to use, the ability of a Consumer to create a unique configuration of one of those entities for its own use is defined. The opaque reference to a configured entity is referred to as an `entityHandle`.

3.6 Persistence and statefulness

This specification does not mandate that either the Producer or the Consumer is stateful [A201]. In the **getMarkup()** and **performInteraction()** calls, the `navigationalState` field carries the state necessary for the entity to render the current markup to be returned to the Consumer. This enables the Consumer to reasonably support page refresh and bookmarking by the End-User. If the Producer utilizes local state, then it stores the conversational state in an implementation-dependent manner, and returns a `sessionHandle` to the Consumer for use during the lifetime of the session.

If the Consumer is operating in a stateless manner, then it may choose the way to achieve this. In the case of HTTP transport the Consumer may employ standard HTTP mechanisms (cookies or URL-rewriting) to push the navigational state or `sessionHandle` out to its client. If operating in a stateful manner, the Consumer may employ any number of persistence/caching mechanisms [A202].

The nature of the conversation between the client and the Consumer, for purposes of this section, is out of scope [A304]. This does not mean that information about the client, including user profile data, is opaque to the Producer. There are many use cases for which user identity must be conveyed to the Producer [A501][A606]. Also, a stateful Producer MUST relate its private conversational state with the `userContext` the Consumer supplies.

3.7 Sessions

In addition to any persistent data, each entity may use a runtime data area (**Session**). An entity MAY establish such a session, and return a `sessionHandle` to reference it within the context of the underlying `entityHandle` in operations such as **getMarkup()**, **performInteraction()**, and **performBlockingInteraction()**.In general, the session between a Consumer and an entity at the Producer maps to a client session with the Consumer.

3.8 Producer Mediated Sharing

Producers may implement a sharing mechanism through techniques such as a shared area within sessions for entities to use. The Producer indicates which entities share such data areas via the `groupID` parameter in the entity metadata. The Consumer MUST respect this grouping as detailed in section 5.4.

Shared data areas introduce implementation challenges in clustered environments. In such an environment, multiple concurrent requests may be routed to different cluster nodes. The Producer must ensure that entities with a common shared data area have access to the shared data even in such situations. Possible implementation choices include:

- The Producer stores the shared data in a database and accesses the same database from all cluster nodes.
- In the case of HTTP transport, a Producer can use HTTP sessions to store the shared data[4]. It must implement a mechanism that ensures only one shared HTTP session is established for each user of a set of entities accessing the shared HTTP session, even for concurrent requests.
- Requiring Consumer assistance in establishing appropriate routing information such as that detailed in section 5.4.

3.9 Information Passing Mechanisms

All information passing enabled by this specification is between exactly one Producer and one Consumer. Implementation of data sharing, including both policy and side effects, within a particular Producer service is outside the scope of this specification.

3.10 Event Handling

Event handling is explicitly not part of this version of the specification. It might be included in a future version of WSRP.

3.11 Two-step protocol

This specification attempts to account for both isolated interactions between a Consumer and a Producer, and also those interactions that may cause state changes in other entities the Consumer aggregates from the same Producer [A503]. Common causes of such shared state include use of a common backend system (e.g. database) and Producer-mediated data sharing. For these reasons, there is a “two-step” capability built into the protocol.

[4] http://www.javaworld.com/javaworld/jw-12-2000/jw-1221-servlets.html

In this two-step interaction, the Consumer first invokes either **performInteraction()** or **performBlockinInteraction()** on the entity whose markup the End-User interacted with. In the case of **performBlockingInteraction()**, the Consumer MUST block all other invocations within the context of the initiating request from the client of the Consumer until either the receipt of a response or the invocation fails (e.g. times out). The Consumer then invokes **getMarkup()** on the entities being aggregated.

Interaction semantics are well-defined across the spectrum of interaction styles supported in the protocol. In other words, the results of the Consumer invoking **performBlockingInteraction()** on an entity, regardless of whether the interaction may have side effects on other entities at the Producer, is well-defined independent of the order of **getMarkup()** invocations on the entities. Entities specifying the invocation of **performInteraction()** (i.e. non-blocking state changes) when URLs are activated from their markup MUST ensure this same determinism in the resulting aggregated markup.

3.12 Interaction Lifecycle States

This section defines the state transitions for the relationship between a Producer and a Consumer.

3.12.1 Assumptions:

In general the Producer is a web service endpoint exposing one or more entities that generate markup and handle interactions with that markup. How these entities are implemented and managed is not defined by this specification, though it is anticipated that the model of how requests are conveyed to the entities by the Producer will be strongly influenced by this specification.

3.12.2 State 0: Unknown

The Consumer has no knowledge that the Producer exists. From this state the Consumer transitions to the **Known** state via discovery; namely by learning the location of the Producer's WSDL. Examples of mechanisms for discovering this include UDDI query, WSIL declarations or other ad hoc mechanisms [A110].

3.12.3 State 1: Known

In this state the Consumer knows the location and interfaces of the Producer (i.e. the Producer's WSDL). From this state the Consumer can transition back to the **Unknown** state, but typically transitions to the **Active** state. This is the earliest state at which the Consumer MAY request a Producer to describe itself [A104]. This ability is present in all states other than **Unknown**.

3.12.4 State 2: Active

This specification is primarily concerned with what can happen when the Producer is in the **Active** state, as this is when Consumers interact with the Producer. It is possible to transition back to the **Known** state by releasing all resources related to the relationship with the Producer. The Consumer is free to perform this state transition multiple times.

3.13 Transport Issues

Since the transport layer is often used to store various pieces of information (e.g. J2EE load balancing depends on the JSessionID cookie and HTTP transport), and these pieces of information often will pertain to a `userContext` rather than the Consumer, Consumers that manage transport layer issues, such as cookies, MUST return them to the Producer only for subsequent invocations using the same `userContext`. We note that failure to properly do this management will eliminate the ability to use Producers that set `requiresInitCookie` to a value other than "none".

4 Service Description Interface

A Producer may be discovered through mechanisms such as **[P3P]** *http://www.w3.org/TR/P3P/*

[Requirements] *http://www.oasis-open.org/committees/wsia/documents/Requirements2002-09-17.html*

[RLTC] *http://www.oasis-open.org/committees/rights/*

[SAML] *https://www.oasis-open.org/committees/security/*

[UDDI] *or* **[WS-Attachments]** *http://www-106.ibm.com/developerworks/webservices/library/ws-attach.html*

[WS-I.org] *http://www.ws-i.org/*

[WSIL], which also provide information concerning the capabilities of the service. Other discovery mechanisms (e.g. emailed URL to a properly enabled browser) do not expose these capabilities. The **getServiceDescription()** operation provides a discovery mechanism-agnostic means for a Consumer to ascertain a Producer's or entity's capabilities [A110]. This interface is required of all Producers to provide a well-defined means for Consumers to ascertain the requirements to register or use the Producer.

4.1 Data Structures

The normative definitions for all data structures are contained in the WSDL referenced in section 15. For the convenience of the reader, this non-normative section uses an IDL like syntax to describe these structures, where the leading [R] indicates a field is required and [O] indicates it is optional. The operations in this section introduce the following data structures:

4.1.1 Extension

The `Extension` structure contains the payload extension mechanism for vendor and application extensions. This allows arbitrary elements from other namespaces to be sent as part of containing data structures. Each such extension MUST declare its type using the schema-defined "type" attribute[5]. We would encourage these to either be of type `xsd:string` or be explicitly typed in a WSDL file that carries the relevant type definitions so that Consumers MAY prepare the appropriate serializer/deserializer. The other option is for each message to connect the extension to a type declared in a schema using the "schemaLocation" attribute as used by schema. Consumers and Producers are NOT REQUIRED to process information supplied using these extension elements.

```
Extension
    [O] Object any[]
```

Members:

[5] http://www.w3.org/TR/xmlschema-1/#xsi_type

- `any`: A schema declaration that implementations MAY choose to extend this structure provided those extensions come from a different namespace.

4.1.2 LocalizedString Type

This `LocalizedString` structure describes both the value for a particular locale and the resource name that MAY be used to extract the value for other locales from a `ResourceList`.

```
LocalizedString
    [R] string          xmlLang
    [R] string          resourceName
    [R] string          value
```

Members:

- `xmlLang`: The locale for this supplied localized value. This is carried in the WSDL using the xml:lang attribute.
- `resourceName`: The name assigned to this localized string for dereferencing into a `ResourceList` for values from other locales.
- `value`: The value for this localized string in the declared locale.

4.1.3 ResourceList Type

This is an array of `Resource` structure, each of which carries the values for a localized resource in various locales.

```
ResourceList
    [R] Resource    resources[]
    [O] Extension   extensions[]
```

Members:

- `resources`: Each member of this array provides the localized values for a resource.
- `extensions`: A mechanism implementations MAY choose to use for extending this structure provided those extensions come from a different namespace.

4.1.4 Resource Type

The `Resource` structure carries the values for a resource in a set of locales.

```
Resource
    [R] string          resourceName
    [R] ResourceValue   values[]
    [O] Extension       extensions[]
```

Members:

- `resourceName`: The name of the resource for which this is a list of localized values.
- `values`: Each member of this array provides the value for the resource in a locale.
- `extensions`: A mechanism implementations MAY choose to use for extending this structure provided those extensions come from a different namespace.

4.1.5 ResourceValue Type

This structure provides the value of a resource for a locale.

```
ResourceValue
    [R] string      xmlLang
```

```
    [R] string       value
    [O] Extension    extensions[]
```

Members:

- `xmlLang`: The locale for this localized value. This is carried in the WSDL using the xml:lang attribute.
- `value`: The value for this localized string in the declared locale.
- `extensions`: A mechanism implementations MAY choose to use for extending this structure provided those extensions come from a different namespace.

4.1.6 RoleDescription Type

This structure is used to describe the roles a Consumer MAY assert for an End-User when interacting with the entities at the Producer. The Consumer MUST NOT assert a role for which no `RoleDescription` was part of the Producer's `ServiceDescription`. Entities MUST NOT declare support for roles that are not part of the Producer's `ServiceDescription`. Note that roles are Producer-wide and therefore are inherently shared by the Producer's entities.

```
RoleDescription
    [R] string           name
    [R] LocalizedString  description
    [O] Extension        extensions[]
```

Members:

- `name`: The name for this role. Preferred form for this name is a URI such that it is definitively namespaced.
- `description`: A localized, free form description of the role. Expected use of this field is for display at the Consumer to someone who will provide a mapping to Consumer supported roles.
- `extensions`: A mechanism implementations MAY choose to use for extending this structure provided those extensions come from a different namespace.

4.1.7 ServiceDescription

The `ServiceDescription` structure contains a set of fields that describe the offered services of the Producer.

```
ServiceDescription
    [R] EntityDescription    offeredEntities[]
    [O] RoleDescription      roleDescriptions[]
    [O] string               requiresInitCookie
    [O] boolean              requiresRegistration
    [O] ModelDescription     registrationPropertyDescription
    [O] ResourceList         resourceList
    [O] Extension            extensions[]
```

Members:

- `offeredEntities`: An array of structures (defined in Section 7.1.2) containing the metadata for the Producer_Offered_Entities.
- `roleDescriptions`: An array of role description structures as defined in Section 4.1.6. This array MUST include an entry for any role the Producer is willing to have the Consumer assert for an End-User, including if the roles named by this specification are supported.

- `requiresInitCookie`: A string (default value = "`none`") indicating whether or not the Producer requires the Consumer to assist with cookie support of the HTTP protocol. Defined values include:
  - `none`: The Producer does not need the Consumer to ever invoke **initCookie()**.
  - `perUser`: The Consumer MUST invoke **initCookie()** once per user of the Consumer, and associate the returned cookie with subsequent invocations on behalf of that user.
  - `perGroup`: The Consumer MUST invoke **initCookie()** once per unique `groupID` from the `EntityDescriptions` for the entities it is aggregating on a page for each user of the Consumer, and associate the returned cookie with subsequent invocations on behalf of that user targeting entities with identical `groupIDs`.
- `requiresRegistration`: A boolean (default value = "true") indicating whether or not the Producer requires Consumer registration. If `requiresRegistration` is set to "false" then it MUST be valid to pass null for the `registrationHandle` field to all operations with the `registrationContext` parameter.
- `registrationPropertyDescription`: Property descriptions for what may and must be supplied during registration.
- `resourceList`: This is an array of `Resource` structures, each of which carries the values for a localized resource in various locales.
- `extensions`: A mechanism implementations MAY choose to use for extending this structure provided those extensions come from a different namespace.

4.1.8 UserContext

The `UserContext` structure supplies End-User specific data to operations. Note that this does not carry user authentication type information (e.g. userID / password) as quite flexible mechanisms for communicating this information are being defined elsewhere (e.g. WS-Security (see section 3.1.2) defines how to carry User Information in a SOAP header).

```
UserContext
   [R] string      userContextID
   [O] string      producerRoles[]
   [O] UserProfile profile
   [O] Extension   extensions[]
```

Members:

- `userContextID`: A string that MAY be used as a reference to the user and that MUST remain invariant for the duration of a Consumer's registration. This key is a token that the Consumer supplies to uniquely identify the UserContext.
- `producerRoles`: An array of strings, each of which specifies an Producer-defined role which the Consumer authorizes for the End-User relative to the current operation. See the discussion of roles in section 8.4.
- `profile`: End-User profile data structure as defined in section 11.36. Note that while the `UserContext` structure is passed to many operations, only the interaction oriented operations need this optional field to be supplied.
- `extensions`: A mechanism implementations MAY choose to use for extending this structure provided those extensions come from a different namespace.

4.1.9 RegistrationState

The `RegistrationState` structure contains fields related to a particular registration of a Consumer with a Producer. It is returned by the **modifyRegistration()** operation and contains the fields of a `RegistrationContext` that allow a Producer to push the storage of state at registration scope to the Consumer.

RegistrationState	
[O] string	registrationState
[O] Extension	extensions[]

Members:

- `registrationState`: This field is used only when the Producer wants the Consumer to provide persistent storage for the state resulting from processing the registration. If this field is non-null, the Consumer MUST return this value on any subsequent calls in the context of this registration [R362].
- `extensions`: A mechanism implementations MAY choose to use for extending this structure provided those extensions come from a different namespace.

4.1.10 RegistrationContext

The `RegistrationContext` structure contains fields related to a particular registration of a Consumer with a Producer. It is returned by the **register()** operation and is a required parameter on most other operations.

RegistrationContext	
[R] string	registrationHandle
[O] string	registrationState
[O] Extension	extensions[]

Members:

- `registrationHandle`: An unique, invariant and opaque reference to the Consumer-Producer relationship. This reference is generated by either the **register()** operation [R355] or a process outside the scope of this specification. Note that Handles are restricted to a maximum length of 255 bytes.
- `registrationState`: This field is used only when the Producer wants the Consumer to provide persistent storage for the state resulting from processing the registration. If this field is non-null, the Consumer MUST return this value on any subsequent calls in the context of this registration [R362].
- `extensions`: A mechanism implementations MAY choose to use for extending this structure provided those extensions come from a different namespace.

4.1.11 desiredLocales and sendAllLocales

These two parameters are used to control what locales are used when localized strings are returned. The `desiredLocales` parameter is an array of strings, each of which specifies a single locale, whose order indicates the preference of the Consumer as to the locales values are returned for. Since localized strings use an indirection through resources to carry the set of values for different locales, the first member of this array SHOULD be used as the locale for the values returned directly in the structure. When the `sendAllLocales` boolean flag is set to "true", the Producer's response SHOULD contain values for all returned localized strings in all locales where they are available.

4.2 getServiceDescription() Operation

This operation allows a Producer to provide information about its capabilities in a context-sensitive manner (e.g. registration may be required to discover the full capabilities of a Producer) [R303].

```
serviceDescription = getServiceDescription(registrationContext, desiredLocales,
                                           sendAllLocales);
Faults: Security.AccessDenied, Security.InvalidProducerRole,
        Security.InconsistentParameters, Security.InvalidRegistration,
        Security.AuthenticationFailure, Interface.MissingParameters,
        Interface.OperationFailed
```

Producers may choose to restrict the information returned in `serviceDescription` based on the supplied registration context. The minimum information a Producer MUST return is that which declares what is required for a Consumer to register (e.g. the `registrationProperties` field) with the Producer [R300][R301][R303]. Producers may also find it useful to restrict the information returned to those portions of the service the registration context allow the Consumer to access on subsequent invocations. Note that the `registrationHandle` field of the `registrationContext` parameter is likely to be null when an unregistered Consumer invokes it. This allows the Consumer to gain access to the information required to successfully register. It is recommended that Consumers invoke **getServiceDescription()** after registering in order to receive a full description of the capabilities the Producer offers within the context of that registration. Producers MUST return a complete enough description to registered Consumers for them to properly interact with both the Producer and entities it exposes.

When generating the `ServiceDescription` response the Producer MUST use the `desiredLocales` (an array of strings) to control what locales are returned for localized strings and `sendAllLocales` (a boolean) as an indication that values for all locales are desired.

While it is possible a `ServiceDescription` will change with time (e.g. Producer deploys additional entities), Producers SHOULD return as complete a `ServiceDescription` as possible.

5 Markup Interface

As user-facing web services, one of the required portTypes a WSIA or WSRP compliant service MUST implement is the generation of markup, which is to be used to represent the current state of an entity to an End-User and the processing of interactions with that markup [A300]. This section explains both the signatures for the operations related to markup generation and processing interactions, and how the concepts of mode and window state impact the generation of the markup.

5.1 Data Structures

The normative definitions for all data structures are contained in the WSDL referenced in section 15. For the convenience of the reader, this non-normative section uses an IDL like syntax to describe these structures, where the leading [R] indicates a field is required and [O] indicates it is optional. The operations in this section introduce the following data structures:

5.1.1 SessionContext

The `SessionContext` structure contains the handle and expires information the Consumer needs to refer to the session in subsequent invocations.

```
SessionContext
    [R] string      sessionHandle
    [R] int         expires
    [O] Extension   extensions[]
```

Members:

- `sessionHandle`: An opaque string the entity defines for referencing state that is stored locally on the Producer. If the Consumer fails to return this reference on future invocations, the entity will be unable to reference this state and therefore likely not generate a markup fragment meeting the End-User's expectations. Note that `Handles` are restricted to a maximum length of 255 bytes.
- `expires`: Maximum number of seconds between invocations referencing the `sessionHandle` before the Producer will schedule releasing the related resources. A value of –1 indicates that the `sessionHandle` will never expire.
- `extensions`: A mechanism implementations MAY choose to use for extending this structure provided those extensions come from a different namespace.

5.1.2 RuntimeContext

The `RuntimeContext` structure defines a collection of fields used only in transient interactions between the Producer and Consumer.

```
RuntimeContext
    [R] string      entityInstanceID
    [O] string      sessionHandle
    [O] Extension   extensions[]
```

Members:

- `entityInstanceID`: An opaque string the Consumer MUST supply as a unique reference to this use of the entity. This reference MAY be used by the entity to properly separate data for multiple instances of the entity within any Producer-defined data sharing mechanisms.
- `sessionHandle`: An opaque string the Producer defines for referencing state stored locally on the Producer. If the Consumer fails to return this reference on future invocations, the entity will be unable to reference this state and therefore likely not generate a markup fragment meeting the End-User's expectations.
- `extensions`: A mechanism implementations MAY choose to use for extending this structure provided those extensions come from a different namespace.

5.1.3 EntityContext

The `EntityContext` structure is used as a parameter on many operations to supply the entity information that was pushed to the Consumer.

```
EntityContext
    [R] string      entityHandle
    [O] string      entityState
    [O] Extension   extensions[]
```

Members:

- `entityHandle`: An opaque and invariant handle, unique within the context of the Consumer's registration (unique within the Producer for Producers not supporting registration). Note that Handles are restricted to a maximum length of 255 bytes.
- `entityState`: An opaque string the entity uses when it depends on the Consumer to store its persistent state [A205]. If `entityState` has a non-null value, the Consumer MUST return this value on subsequent calls using the same `entityHandle`. Note that such uses MAY span various cycling of the Consumer and therefore this state MUST be persisted by the Consumer until successfully invoking **destroyEntities()** with the related `entityHandle`.
- `extensions`: A mechanism implementations MAY choose to use for extending this structure provided those extensions come from a different namespace.

5.1.4 CacheControl

The `CacheControl` structure contains a set of fields needed for the entity to manage cached markup fragments. Note that the cache key MUST always include the `MarkupParams` structure that caused the markup fragment to be generated.

```
CacheControl
    [R] int         expires
    [O] string      userScope
    [O] string      validateTag
    [O] Extension   extensions[]
```

Members:

- `expires`: Number of seconds the markup fragment referenced by this cache control entry remains valid. A value of –1 indicates that the markup fragment will never expire.
- `userScope`: A string indicating when the markup may be used by various users:
    a. "`perUser`": The markup is specific to the userContext for which it was generated. Changes to the data of the userContext MUST invalidate the markup.
    b. "`forAll`": The markup is not specific to the userContext and therefore may be supplied to all users of the Consumer.
- `validateTag`: A string the Consumer MAY use to attempt to revalidate markup once the `expires` duration elapses. This potentially eliminates the need for the entity to regenerate the markup and thereby can significantly impact the performance for the End-User.
- `extensions`: A mechanism implementations MAY choose to use for extending this structure provided those extensions come from a different namespace.

5.1.5 Templates

The `Templates` structure contains a set of fields that enable Producer URL writing. The template style format of these fields is defined in section 9.2.2.

```
Templates
    [O] string      DefaultTemplate
    [O] string      ActionTemplate
    [O] string      BlockingActionTemplate
    [O] string      RenderTemplate
    [O] string      ResourceTemplate
    [O] string      SecureDefaultTemplate
    [O] string      SecureActionTemplate
    [O] string      SecureBlockingActionTemplate
    [O] string      SecureRenderTemplate
    [O] string      SecureResourceTemplate
    [O] string      NameSpacePrefix
    [O] Extension   extensions[]
```

Members:

- `DefaultTemplate`: This template provides the default value for all of the other template fields. Note that the `SecureDefaultTemplate` field MAY provide a first-level override of this default value for the fields whose names begin with "Secure", as these frequently involve a different protocol specification.
- `ActionTemplate`: This template provides the template for URLs that will be directed to the Consumer and processed as a **performInteraction()** on the entity.
- `BlockingActionTemplate`: This template provides the template for URLs that will be directed to the Consumer and processed as a **performBlockingInteraction()** on the entity.
- `RenderTemplate`: This template provides the template for URLs that will be directed to the Consumer and processed as a **getMarkup()** on the entity.
- `ResourceTemplate`: This template provides the template for URLs that will be directed to the Consumer and processed as an HTTP GET on the named resource.
- `SecureDefaultTemplate`: This template provides the default value for all the secure template fields.
- `SecureActionTemplate`: This template provides the template for secure URLs that will be directed to the Consumer and processed as a **performInteraction()** on the entity using a secure protocol.
- `SecureBlockingActionTemplate`: This template provides the template for secure URLs that will be directed to the Consumer and processed as a **performBlockingInteraction()** on the entity using a secure protocol.
- `SecureRenderTemplate`: This template provides the template for secure URLs that will be directed to the Consumer and processed as a **getMarkup()** on the entity using a secure protocol.
- `SecureResourceTemplate`: This template provides the template for secure URLs that will be directed to the Consumer and processed as an HTTP GET over SSL/TLS on the named resource.
- `NameSpacePrefix`: This field provides a string the entity MAY use to prefix tokens that need to be unique on the aggregated page (e.g. JavaScript variables, html id attributes, etc.).
- `extensions`: A mechanism implementations MAY choose to use for extending this structure provided those extensions come from a different namespace.

5.1.6 MarkupParams

The `MarkupParams` structure contains a set of fields needed for the entity to generate markup that will enable the End-User to visualize the state of the entity.

```
MarkupParams
    [R] ClientData   clientData
    [R] boolean      secureClientCommuncations
    [R] string       userAuthentication
    [R] string       locale[]
    [R] string       markupCharacterSet
    [R] string       markupType[]
    [R] string       mode
    [R] string       windowState
    [R] string       navigationalState
    [O] Property     requestParameters[]
    [O] Templates    templates
    [O] string       validateTag
    [O] Extension    extensions[]
```

Members:

- `clientData`: A structure (defined in section 11.3) that provides information (including `userAgent`) about the client device which will render the markup.
- `secureClientCommunications`: A flag indicating whether or not the delivery channel between a client and Consumer is secure [R401]. The Consumer MUST set the `secureClientCommunications` flag as the entity MAY render different content when it knows the delivery channel is secure.
- `userAuthentication`: String indicating how the End-User was authenticated. Common values include:
    a. "`None`": No authentication was done, user information is asserted for informational purposes only.
    b. "`Password`": The End-User identified themselves using the common userid/password scenario.
    c. "`Certificate`": The End-User presented a security certificate to validate their identity.
    d. Other strings: Some authentication was done outside this limited set of possibilities.
- `locale`: An array of locales where the order in the array is the order in which the Consumer would prefer the entity generate the markup (e.g. "en-US"). Note that current practice on the Internet uses the format [2 char language code][6] "-" [2 char country code][7] as per the provided example. The Consumer SHOULD supply this information based on the setting the End-User has requested.

[6] http://lcweb.loc.gov/standards/iso639-2/langcodes.html
[7] http://www.din.de/gremien/nas/nabd/iso3166ma/codlstp1/en_listp1.html

- `markupCharacterSet`: The `characterSet`[8] (e.g. “UTF-8”, “ISO-10646-Unicode-Latin1”, etc.) the Consumer would like the entity to use for encoding the markup (i.e. the character set for the aggregated page). This encoding may be different from the character set used for the transport of the invocation from the Consumer to Producer. The Producer MUST either use this character set for the response message or properly escape any characters that would otherwise not be properly represented in the character set of the response message.

- `markupType`: An array of Mime types[9] (e.g. “text/html”, “application/xhtml+xml”, etc.) where the order in the array is the order in which the Consumer would prefer the entity generate the markup (i.e. first is most preferred, second is next preferred, etc.). In addition to these fully specified Mime types, use of “*” (indicates all Mime types are acceptable) and type/* (where type includes things such as “text”) from the HTTP definition[10] MAY be specified. Entities SHOULD generate markup in one of the specified Mime types.

- `mode`: The mode for which the entity should render its output. A set of modes is defined in this specification (see section 5.10). The Consumer SHOULD inspect the entity’s metadata to determine which of these modes the entity supports in addition to any Producer-defined modes. The Consumer MUST specify either one of the modes from the entity’s metadata or ”`normal`” (all entities are required to support this mode).

- `windowState`: The state of this entity’s virtual window relative to other entities on the aggregated page. Constants and definitions for the specification-defined states are found in section 5.11.

- `navigationalState`: This field contains the opaque navigational state for this entity either from the appropriate URL parameter (see section 9.2.1.1) or the most recently returned value for this End-User.

- `requestParameters`: Name/value pairs reflected from the query string of the activated URL. These are the query string parameters the Consumer did not consume by processing them itself. Other name value pairs (e.g. HTTP headers from the client or additional Consumer-supplied data) should be placed in the extensions array.

- `templates`: If this entity declared `doesUrlTemplateProcessing` as ”true” in its `EntityDescription`, then this field contains the templates the Consumer is supplying for that processing. If the `EntityDescription` also has templatesStoredInSession set to ”true”, then the Consumer MAY elect to only send these once for a `sessionHandle`.

- `validateTag`: This field MAY contain a `validateTag` previously supplied to the Consumer in a `MarkupContext` structure. When this field is non-null, the Consumer is indicating it has markup cached for the entity, but the `CacheControl` structure governing the use of that cached markup no longer indicates it is valid. The Consumer is supplying the `validateTag` as a means for the entity to avoid generating new markup if the cached markup can be validated. The entity sets the useCachedMarkup field in the returned `MarkupContext` to ”true” to indicate the markup referenced by the `validateTag` is still valid.

- `extensions`: A mechanism implementations MAY choose to use for extending this structure provided those extensions come from a different namespace.

[8] http://www.iana.org/assignments/character-sets
[9] http://www.isi.edu/in-notes/iana/assignments/media-types/media-types
[10] http://www.ietf.org/rfc/rfc2616.txt

5.1.7 MarkupContext

The `MarkupContext` structure contains fields relative to returning markup from various invocations.

```
MarkupContext
    [O] boolean         useCachedMarkup
    [O] string          markup
    [O] string          locale
    [O] string          markupType
    [O] boolean         requiresUrlRewriting
    [O] CacheControl    cacheControl
    [O] string          preferredTitle
    [O] Extension       extensions[]
```

Members:

- `useCachedMarkup`: A boolean used to indicate whether the markup the Consumer indicated it has cached is still valid. The default value of this field is ”false” (i.e. new markup is being returned for the Consumer’s use). When this field’s value is “true” the markup field MUST NOT be returned. If field’s value is “true”, any supplied `cacheControl` field MUST be processed as an update to the `cacheControl` originally supplied with the cached markup.
- `markup`: The markup to be used for visualizing the current state of the entity. This is a string in order to support non-XML markup (e.g. HTML). If this is encoded in a SOAP message (i.e. XML), various characters will likely need to be escaped, either by the entity or the Producer’s runtime (e.g. “<“ and “>“). The character set of the markup an entity returns MUST either match that requested in `MarkupParams` or be UTF-8. When a SOAP binding is used, the character set of the markup returned by the Producer MUST match the character set of the SOAP envelope. This field is only missing when the `useCachedMarkup` flag is “true”.
- `locale`: The locale of the returned markup. This field MUST be specified whenever markup is returned.
- `markupType`: The Mime type of the returned markup. This field MUST be specified whenever markup is returned.
- `requiresUrlRewriting`: A flag by which the entity/Producer indicates whether or not Consumer-side URL rewriting (see section 9.2.1) is required. The Consumer MUST parse the markup for URL rewriting if this flag is set to ”true”. The default value for this flag is “false”.
- `cacheControl`: Defines the caching policies for the returned markup fragment. If this field is not supplied, the Consumer MUST treat the returned markup as not cachable.
- `preferredTitle`: The title the entity would prefer to be used in any decoration of the markup.
- `extensions`: A mechanism implementations MAY choose to use for extending this structure provided those extensions come from a different namespace.

5.1.8 MarkupResponse

The `MarkupResponse` structure contains fields for returning various items in response to a **getMarkup()** invocation.

```
MarkupResponse
    [O] SessionContext     sessionContext
    [O] MarkupContext      markupContext
    [O] Extension          extensions[]
```

Members:

- `sessionContext`: This structure contains session-oriented fields that may be returned from various operations, including a new `sessionHandle` and the duration before it `expires`.
- `markupContext`: A structure carrying the returned markup and fields related to the markup.
- `extensions`: A mechanism implementations MAY choose to use for extending this structure provided those extensions come from a different namespace.

5.1.9 InteractionResponse

The `InteractionResponse` structure contains the various items **performInteraction()** can return.

```
InteractionResponse
    [R] string             navigationalState
    [O] SessionContext     sessionContext
    [O] Entitycontext      entityContext
    [O] MarkupContext      markupContext
    [O] Extension          extensions[]
```

Members:

- `navigationalState`: Opaque representation of navigational state which the entity is returning to the Consumer to indicate the navigational state to be supplied to **getMarkup()** including for page refreshes and page bookmarks. The Consumer MUST supply this value as the `navigationalState` on the subsequent invocations for this use of the entity for at least the duration of the End-User's interactions with this aggregated page. The Consumer is not required to persist the `navigationalState` for longer than this set of interactions, but MAY provide such a persistence if desired.
- `sessionContext`: This structure contains session-oriented fields that may be returned from various operations, including a new `sessionHandle` and the duration before it `expires`.
- `entityContext`: This structure is where an entity using Consumer-side persistent storage may return a change in its persistent state, provided the `entityStateChange` flag in `InteractionParams` had been set to "`OK`" or "`Clone`". When the `entityStateChange` flag had been set to "`Clone`", this may also include a new `entityHandle`. The sequence by which an entity can otherwise request changing this state is described in section 5.3.3.
- `markupContext`: Markup may be returned at the end of interaction processing as an optimization that avoids an additional remote invocation.
- `extensions`: A mechanism implementations MAY choose to use for extending this structure provided those extensions come from a different namespace.

5.1.10 UpdateResponse

The `UpdateResponse` structure contains the items normally returned by **performBlockingInteraction()**.

```
UpdateResponse
    [R] string           navigationalState
    [O] SessionContext   sessionContext
    [O] EntityContext    entityContext
    [O] string           newWindowState
    [O] string           newMode
    [O] MarkupContext    markupContext
```

Members:

- `navigationalState`: Opaque representation of navigational state which the entity is returning to the Consumer to indicate the navigational state to be supplied to **getMarkup()** including for page refreshes and page bookmarks. The Consumer MUST supply this value as the `navigationalState` on the subsequent invocations for this use of the entity. This ensures the correct state of the entity is used when processing the invocation.
- `sessionContext`: This structure contains session-oriented fields that may be returned from various operations, including a new `sessionHandle` and the duration before it `expires`.
- `entityContext`: This structure is where an entity using Consumer-side persistent storage may return a change in its persistent state, provided the `entityStateChange` flag in `InteractionParams` had been set to "`OK`" or "`Clone`". When the `entityStateChange` flag had been set to "`Clone`", this may also include a new `entityHandle`. The sequence by which an entity can otherwise request changing this state is described in section 5.3.3.
- `newWindowState`: A request from the entity to change the window state. The Consumer MAY choose to respect this request, but since the entity cannot depend on that choice it MUST NOT encode this new window state into any of its stateful settings. Rather, the entity MUST compute any such impact on stateful settings after the Consumer has actually changed the window state.
- `newMode`: A request from the entity to change the mode. The Consumer MAY choose to respect this request, but since the entity cannot depend on that choice it MUST NOT encode this new mode into any of its stateful settings. Rather, the entity MUST compute any such impact on stateful settings after the Consumer has actually changed the mode.
- `markupContext`: Markup may be returned at the end of interaction processing as an optimization that avoids an additional remote invocation.

5.1.11 BlockingInteractionResponse

The `BlockingInteractionResponse` structure contains the various items **performBlockingInteraction()** can return.

```
BlockingInteractionResponse
    [O] UpdateResponse   updateResponse
    [O] string           redirectURL
    [O] Extension        extensions[]
```

Members:

- `updateResponse`: This field captures the items returned when the entity is not directing the user to a different URL. It is mutually exclusive with the `redirectURL` field.
- `redirectURL`: As a result of processing this interaction, the entity may indicate to the Consumer that it would like the End-User to view a different URL. It is mutually exclusive with the `updateResponse` field.
- `extensions`: A mechanism implementations MAY choose to use for extending this structure provided those extensions come from a different namespace.

5.1.12 StateChange

This type is a restriction on the string type that is constrained to the values "`OK`", "`Clone`" or "`Fault`".

5.1.13 UploadContext

The `UploadContext` structure contains fields specific to uploading data to the entity.

```
UploadContext
    [R] base64Binary   uploadData
    [R] string         mimeType
    [O] Extension      extensions[]
```

Members:

- `uploadData`: A binary data blob that is being uploaded.
- `mimeType`: Mime type of what is in the `uploadData` field.
- `extensions`: A mechanism implementations MAY choose to use for extending this structure provided those extensions come from a different namespace.

5.1.14 InteractionParams

The `InteractionParams` structure contains fields specific to invoking either **performInteraction()** or **performBlockingInteraction()** operations.

```
InteractionParams
    [R] StateChange    entityStateChange
    [O] string         validNewModes[]
    [O] string         validNewWindowStates[]
    [O] UploadContext  uploadContext
    [O] Extension      extensions[]
```

Members:

- `entityStateChange`: A flag by which a Consumer indicates whether or not the processing of the interaction is allowed to return a modified `entityState`. This flag is needed; as only the Consumer knows whether or not such a state change would be acceptable. In many cases where the Consumer does not authorize the End-User to modify the persistent state of the entity in use, it may permit the Producer to clone the entity (i.e. set `entityStateChange` to "`Clone`") and return a clone of the entity in addition to any other return parameters. The full use of this flag is described in section 5.3.3.
- `validNewModes`: An array of `modes` which the Consumer is indicating as available to be requested as a `newMode` in `InteractionResponse`. It should be noted that this is no guarantee that a requested transition will be honored, as factors not easily represented may cause the Consumer to reject a requested transition. The primary reason for supplying this information is to assist the entity in preparing a user interface that does not contain links the Consumer will not honor.
- `validNewWindowStates`: An array of `windowStates` which the Consumer is indicating as available to be requested as a `newWindowState` in `InteractionResponse`. It should be noted that this is no guarantee that a requested transition will be honored, as factors not easily represented may cause the Consumer to reject a requested transition. The primary reason for supplying this information is to assist the entity in preparing a user interface that does not contain links the Consumer will not honor.
- `uploadContext`: An optional field where binary data may be uploaded.
- `extensions`: A mechanism implementations MAY choose to use for extending this structure provided those extensions come from a different namespace.

5.2 getMarkup() Operation

The Consumer requests the markup for rendering the current state of an entity by invoking:

```
markupResponse = getMarkup(registrationContext, entityContext, runtimeContext,
                           userContext, markupParams);
Faults: Security.AccessDenied, Security.InvalidProducerRole,
        Security.InconsistentParameters, Security.InvalidRegistration,
        Interface.MissingParameters, Interface.OperationFailed,
        Interface.InvalidHandle, Interface.InvalidCookie,
        Interface.UnsupportedMode, Interface.UnsupportedWindowState
        Interface.UnsupportedLocale, Interface.UnsupportedMarkuptype
```

5.2.1 Caching of markup fragments

For performance reasons the Consumer might prefer to cache markup across a series of requests. The Producer passes information about the cachability of the markup fragment in the `cacheControl` structure returned in `MarkupContext`. The Consumer can infer from this information when it may cache markup and when the cached markup needs to be invalidated and updated by a new call to **getMarkup()**.

5.2.1.1 Cachability

Whenever the `cacheControl` field of `MarkupResponse` is filled in the Consumer MAY cache the markup fragment. The Consumer MUST follow the defined invalidation policies to keep the cache up-to-date. If the `cacheControl` field is empty the Consumer MUST NOT cache the markup fragment.

5.2.1.2 Cache Invalidation

The `expires` field of the `cacheControl` provides a time duration for when the markup SHOULD be considered valid. Once this time has elapsed, counting from the point in time when the `markupContext` was returned, the Consumer SHOULD use the `validateTag` field of the `MarkupParams` structure to inquire whether the markup is still valid, as this potentially avoids having the entity regenerate the same markup. When the `MarkupParams` structure supplied for generating the markup changes, the Consumer MUST treat the cached markup as if the `expires` duration had already elapsed.

5.3 Interaction Operations

End-User interactions with the generated markup may result in invocations for the entity to respond to the interactions [A400]. In the case where the invocations may change the `navigationalState` or some data the entity is storing in a shared data area (including a database), an operation is needed to carry the semantics of this type of update. Two operations are defined for processing interactions and the state changes they may cause, one carrying the additional semantics of blocking the Consumer from both beginning the generation of the aggregated page and gathering markup from other entities on the page.

5.3.1 performInteraction() Operation

This operation does not carry the semantics of blocking the Consumer's processing:

```
interactionResponse = performInteraction(registrationContext, entityContext,
                                         runtimeContext, userContext,
                                         markupParams, interactionParams);
Faults: Security.AccessDenied, Security.InvalidProducerRole,
        Security.InconsistentParameters, Security.InvalidRegistration,
        Interface.MissingParameters, Interface.OperationFailed,
        Interface.InvalidHandle, Interface.InvalidCookie,
        Interface.UnsupportedMode, Interface.UnsupportedWindowState
        Interface.UnsupportedLocale, Interface.UnsupportedMarkuptype
        Interface.EntityStateChangeRequired
```

Since this operation potentially returns state to the Consumer for storage, this allows Consumers who wish to store this by pushing it to their client to do so before opening the stream for the aggregated page. Consumers doing this also enable End-User bookmarking of the aggregated page for later use.

5.3.2 performBlockingInteraction() Operation

This operation also carries the semantics of blocking both the Consumer beginning the generation of the aggregated page (often because the invocation MAY return a `redirectURL`) and the gathering of markup from other entities (often because shared state, including via a database, impacts the markup of other entities):

```
blockingInteractionResponse = performBlockingInteraction(registrationContext,
                                         entityContext, runtimeContext, userContext,
                                         markupParams, interactionParams);
Faults: Security.AccessDenied, Security.InvalidProducerRole,
        Security.InconsistentParameters, Security.InvalidRegistration,
        Interface.MissingParameters, Interface.OperationFailed,
        Interface.InvalidHandle, Interface.InvalidCookie,
        Interface.UnsupportedMode, Interface.UnsupportedWindowState
        Interface.UnsupportedLocale, Interface.UnsupportedMarkuptype
        Interface.EntityStateChangeRequired
```

Since this is a blocking operation, the Consumer MUST wait for the response before invoking **getMarkup()** on the entities it is aggregating. This permits any Producer-mediated sharing to proceed safely (provided it happens in a synchronous manner). Since this operation potentially returns state to the Consumer for storage, this operation also allows Consumers who wish to store this by pushing it to their client to do so before opening the stream for the aggregated page. Consumers doing this also enable End-User bookmarking of the aggregated page for later use.

5.3.3 Updating Persistent Entity State

In designing how an entity and Consumer interact in order to update the persistent state of the entity, the following items were considered:

1. Only the entity knows when such a state change is desired. While it is expected that changes to persistent state will be relatively rare, they could occur on any interaction the entity has with an End-User.
2. Only the Consumer knows whether or not a persistent state change would be safe. Reasons for this include whether the persistent state is shared among a group of users, the authorization level of the End-User to impact any shared persistent state and Consumer policies regarding whether the persistent state is modifiable.

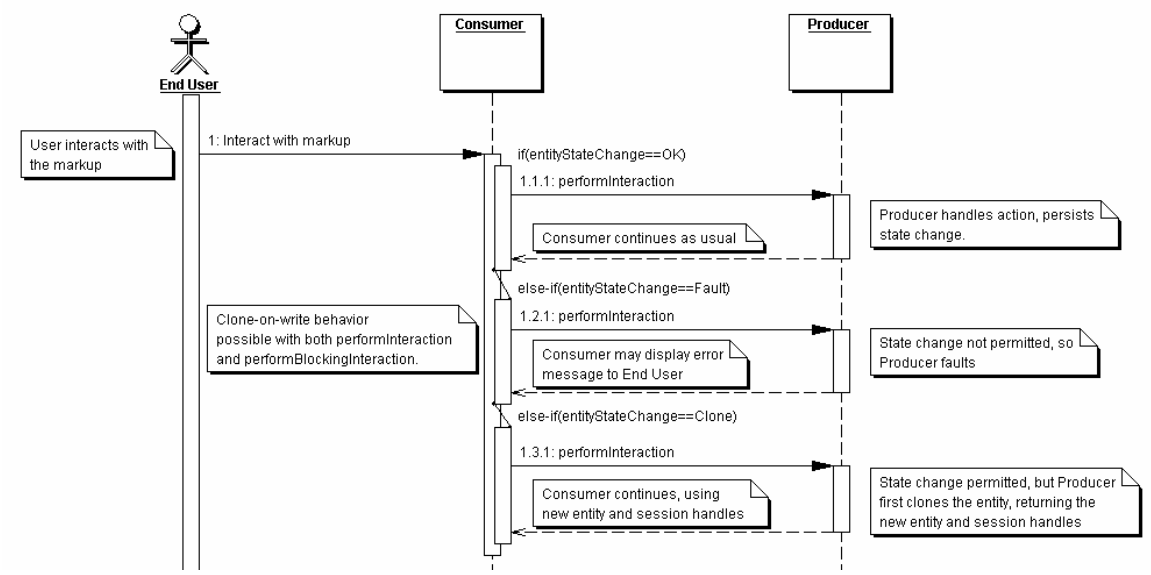
This combination requires that all persistent entity state changes happen in a manner that has Consumer approval for the change to occur, while the entity decides both when the change is required and its exact character. The Consumer indicates whether or not it is safe for the entity to modify its persistent state by setting the `entityStateChange` field in the `interactionParams` structure. If the Consumer has set the `entityStateChange` flag to "`OK`", the entity MAY modify its persistent state regardless of whether it is persisted on the Producer or Consumer.

If the Consumer has set the `entityStateChange` field to "`Clone`", persistent state changes are allowed only if the Producer first clones the entity. If the Producer does not clone the entity, processing attempts to modify persistent state MUST proceed as if the Consumer had specified "`Fault`" for `entityStateChange`. If the Producer clones the entity, processing attempts to modify persistent state on the new entity SHOULD proceed as if the Consumer had specified "`OK`" for `entityStateChange`. The Producer returns the impact of any cloning to the Consumer, regardless of whether the `entityState` is persisted on the Producer or Consumer. If the Producer returns a new `entityHandle` without returning a new `sessionHandle`, the Consumer MUST associate the current `sessionHandle` with the new `entityHandle` rather than the previous `entityHandle`.

If the Consumer has set the `entityStateChange` flag to "`Fault`", the entity MUST NOT modify its persistent state regardless of whether it is persisted on the Producer or Consumer and MUST throw a fault message if processing the interaction requires changing its persistent state. Commonly Consumer's will only set the `entityStateChange` flag to "`Fault`" for End-Users that are not authorized to clone or personalize the entity (e.g. an End-User using a guest account).

If the Producer implements access control that prevents entities from updating persistent state and an entity is unable to process the interaction without such an update, then the fault "`Interface.EntityStateChangeRequired`" MUST be thrown indicating the interaction processing failed.

This set of possibilities is depicted in the following figure:



5.4 initCookie() Operation

In general, the Producer completely manages its own environment, including items such as the initialization of cookies when using the HTTP transport. There are cases, however, when assistance from the Consumer in initializing these cookies is useful. Cookies needed to manage distribution of requests within a load balanced environment are an example of such. This operation is how the Consumer provides such assistance:

```
ReturnAny initCookie(registrationContext);
Faults: Security.AccessDenied, Security.InvalidRegistration,
    Interface.MissingParameters, Interface.OperationFailed,
    Interface.InvalidHandle
```

If the Producer's metadata has set the `requiresInitCookie` field to any value other than "`none`", then the Consumer MUST invoke **initCookie()** and supply the returned cookie according to the semantics of the value of `requiresInitCookie` as defined in section 4.1.2. If at any time the Producer throws a fault message ("`Interface.InvalidCookie`") indicating the supplied `cookie` has been invalidated at the Producer, then the Consumer MUST again invoke **initCookie()** and SHOULD reprocess the invocation that caused the fault message to be thrown.

5.5 releaseSessions() Operation

The Consumer MAY inform the Producer that it will no longer be using a set of sessions by invoking **releaseSessions()** (e.g. the Consumer is releasing resources related to the `sessionHandles`):

```
ReturnAny = releaseSessions(registrationContext, sessionHandles);
Faults: Security.AccessDenied, Interface.MissingParameters,
        Interface.OperationFailed, Interface.InvalidHandle
```

After invoking **releaseSessions()** the Consumer MUST NOT include any of the supplied `sessionHandles` on subsequent invocations.

5.7 Load Balancing

Load balancing is a part of the Producer environment that cannot easily be managed from within the protocol. Load balancing is highly dependent on mechanisms in the transport, for example the use of cookies in HTTP. In order to permit load balancing to function, regardless of the transport binding in use, the Consumer MUST manage transport level issues itself. Using HTTP as an example, if the Producer requires such support of Consumers, it MUST indicate so by setting the `requiresInitCookie` metadata to a value other than "`none`". If the Producer set `requiresInitCookie` to a value other than "`none`", the Consumer MUST ensure the cookie is properly supplied in subsequent requests for the End-User.

5.8 Consumer Transitions across Bindings

Consumers SHOULD be careful about the support supplied by the web stack with regards to multiple bindings that will be offered by many Producers. If a Producer indicates that it uses cookies, the Consumer MUST ensure that any cookies the Producer sets are available on all invocations regardless of whether the operation is in the same binding that set the cookie. Another implication of the Producer indicating it uses cookies is that the Consumer SHOULD NOT do any protocol transitions (e.g. from HTTP to HTTPS) as cookies are often managed in a manner that does not allow the information to be shared across such a transition. Switching between protocols (e.g., going from HTTP to HTTPS) will likely break transport-level load balancing on the Producer. This is because the request being routed via the new binding will almost certainly go through a different load balancer and likely a different session manager. In addition, since providing the information stored in cookies when using HTTPS for a later HTTP connection opens security issues, Consumers MUST NOT move cookies from an HTTPS service endpoint to an HTTP service endpoint. Also, since cookies asserted using HTTPS are considered a trusted portion of the message, Consumers SHOULD NOT move cookies from an HTTP to an HTTPS service endpoint.

5.9 Stateful Entity Scenarios

There are several common scenarios for entities with varying needs regarding statefulness [A202][A203]. This section explains how they map into the operational signatures above.

5.9.1 No State

This type of entity maintains no state, but encodes everything required to generate the markup on the URL causing the invocation of **getMarkup()** [A201]. Often these entities involve only a single page, but could provide links on that page that cause the generation of a completely different markup due to the parameters passed when the link is activated.

Note: Invocations of **performBlockingInteraction()** MAY happen in this scenario if the entity impacts some backend system as a result of the invocation as this impact could change the markup some other entity will generate.

The following table outlines the values for certain key parameters that support this scenario.

Method	Parameter/Field	Value	Comments
performInteraction or performBlockingInteraction	sessionContext / sessionHandle	Producer_Offered_Entity or Consumer_Configured_Entity	Always null as this entity uses no Producer-side state.
	markupParameters / navigationalState	Consumer extracts value from link.	Navigational state encoded on the URLs

			in the markup only.
	interactionResponse / navigationalState	This type of entity does not return navigationalState.	
getMarkup	sessionContext / sessionHandle	Producer_Offered_Entity or Consumer_Configured_Entity	Always null as this entity uses no Producer-side state.
	markupParameters / navigationalState	Consumer extracts value from link.	Navigational state from the URL.

5.9.2 Navigational State Only

This type of entity does not maintain state at the Producer, but does push navigational state out to the Consumer. Both to support these entities and to assist Consumers in properly supporting End-User page refreshes and bookmarks, entities are allowed to return their navigational state (`navigationalState` field) back to the Consumer. It is then the responsibility of the Consumer to retransmit the `navigationalState` to the Producer with each request [A206].

A stateless Consumer can store the `navigationalState` for all of its aggregated entities by returning them to the client, for example by encoding them in the URL. Since this implementation option requires the URL to be generated before the output stream is opened, the `navigationalState` of all entities must be known before the Consumer begins generating the output stream. In order to allow the Consumer to open the output stream before it has collected markup from all entities aggregated on the page, a **getMarkup()** invocation is not allowed to modify the `navigationalState`. Only invocations of **performInteraction()** or **performBlockingInteraction()** are allowed to modify the `navigationalState` of an entity.

The following table outlines the values for certain key parameters that support this scenario.

Method	Parameter/Field	Value	Comments
performInteraction or performBlockingInteraction	sessionContext / sessionHandle	Producer_Offered_Entity or Consumer_Configured_Entity	Always null as this entity uses no Producer-side state.
	markupParameters / navigationalState	Consumer extracts value from link or previous value.	
	interactionResponse / navigationalState	Entity may compute a changed navigationalState.	
getMarkup	sessionContext / sessionHandle	Producer_Offered_Entity or Consumer_Configured_Entity	Always null as this entity uses no Producer-side state.
	markupParameters / navigationalState	From link or previous value or from performInteraction.	

5.9.3 Local state

Entities storing state locally on the Producer establish a **Session** and return an opaque reference (a `sessionHandle`) the Consumer is then required to return on all subsequent invocations on the entity for this End-User. These entities MAY also push navigational state to the Consumer such that an End-User may bookmark some portion of the state for use in later conversations. The means by which the Consumer enables this functionality for the End-User is a Consumer implementation choice [A304].

The following table outlines the values for certain key parameters that support this scenario.

Method	Parameter/Field	Value	Comments
performInteraction or performBlockingInteraction	sessionContext / sessionHandle	Producer_Offered_Entity or Consumer_Configured_Entity or refined handle that also encodes session info	With Producer side state, the session handle offers ability to store information without impacts message size to Consumer.
	markupParameters / navigationalState	Consumer extracts value from link or previous value.	
	interactionResponse / navigationalState	Entity may compute a changed navigationalState.	
getMarkup	sessionContext / sessionHandle	Producer_Offered_Entity or Consumer_Configured_Entity or refined handle that also encodes session info	With Producer side state, the session handle offers ability to store information without impacts message size to Consumer.
	markupParameters / navigationalState	From link or previous value or from performInteraction.	

5.10 Modes

An entity should render different content and perform different activities depending on its current state, the operation (with parameters) currently being processed, and the functionality requested by the End-User. A base set of functions is defined which reflects those common for portal-portlet interactions. They are referred to as modes and should be thought of as how the Consumer is managing the interaction with the End-User. Entities may request mode changes either through parameters on a link that an End-User activates or by returning a `newMode` in a `BlockingInteractionResponse`. Whether or not such a request is honored is up to the Consumer and often will depend on access control settings for the End-User.

An entity MUST support the `view` mode and SHOULD support the `edit` and `help` modes. During **getMarkup()**, **performInteraction()**, and **performBlockingInteraction()** invocations the Consumer indicates to the entity its current mode via the `MarkupParams` data structure.

5.10.1 “view” Mode

The expected functionality for an entity in `view` mode is to render markup reflecting the current state of the entity. The `view` mode of an entity will include one or more screens that the End-User can navigate and interact with or it may consist of static content devoid of user interactions.

The behavior and the generated content of an entity in the `view` mode may depend on configuration, personalization and all forms of state.

All entities MUST support the `view` mode.

5.10.2 “edit” Mode

Within the `edit` mode, an entity should provide content and logic that let a user customize the behavior of the entity. The `edit` mode may include one or more screens which users can navigate to enter their customization data.

Typically, entities in `edit` mode will set or update entity state by making these changes persistent for the entity. How such changes impact Consumer management of entity usage by End-Users is discussed in section 5.3.3.

5.10.3 “help” Mode

When in `help` mode, an entity may provide help screens that explains the entity and its expected usage. Some entities will provide context-sensitive help based on the markup the End-User was viewing when entering this mode.

5.10.4 “preview” Mode

In preview, an entity should provide a rendering of its standard `view` mode content, as a visual sample of how this entity will appear on the End-User’s page with the current configuration. This could be useful for a Consumer that offers an advanced layout capability.

5.10.5 Custom Modes

The extensible `RegistrationData` structure provides a field for Consumers to declare additional custom modes. In addition, the extensible `EntityDescription` structure provides a field for entities to declare what modes they understand. A Consumer SHOULD NOT set a mode an entity does not understand. An entity MUST map any mode it does not understand to the `view` mode.

5.11 Window States

Window state is an indicator of the amount of page space that will be assigned to the content generated by an entity. This hint is provided by the Consumer for the entity to use when deciding how much information to render in the generated markup.

An entity MUST support the `normal` window state and SHOULD support the `minimized` and `maximized` window states.

5.11.1 “normal” Window State

The `normal` window state indicates the entity is likely sharing the aggregated page with other entities. It MAY also indicate that the target device has limited display capabilities. Therefore, an entity SHOULD restrict the size of its rendered output in this window state.

All entities MUST support the `normal` window state.

5.11.2 “minimized” Window State

When the window state is `minimized`, the entity SHOULD NOT render visible markup, but is free to include non-visible data such as javascript or hidden forms. The Producer MUST expect the **getMarkup()** operation to be invoked for the `minimized` state just as for all other window states

5.11.3 “maximized” Window State

The `maximized` window state is an indication the entity is likely the only entity being rendered in the aggregated page, or that the entity has more space compared to other entities in the aggregated page. An entity SHOULD generate richer content when its window state is `maximized`.

5.11.4 “solo” Window State

The `solo` window state is an indication the entity is the only entity being rendered in the aggregated page. An entity SHOULD generate richer content when its window state is `solo`.

5.11.5 Custom Window States

The extensible `RegistrationData` structure provides a field for Consumers to declare additional custom window states. In addition, the extensible `EntityDescription` structure contains a field for entities to declare what window states they understand. A Consumer SHOULD NOT set a window state an entity does not understand. An entity MUST map any window state it does not understand to `normal`.

---

6 Registration Interface

A Producer that supports in-band registration of Consumers exposes the optional registration portType. Regardless of whether or not the registration portType is exposed, Producers MAY offer out-of-band processes to register a Consumer. All Producer registration processes MUST result in a unique, opaque token that may be used to refer to the registration. This specification calls this token a `registrationHandle` (defined in section 4.1.10).

6.1 Data Structures

The normative definitions for all data structures are contained in the WSDL referenced in section 15. For the convenience of the reader, this non-normative section uses an IDL like syntax to describe these structures, where the leading [R] indicates a field is required and [O] indicates it is optional. The operations in this section introduce the following data structures:

6.1.1 RegistrationData

The `RegistrationData` structure provides the means for the Consumer to supply the data required for registration with a Producer as well as protocol extensions that it supports [R355][R356].

```
RegistrationData
    [R] string      consumerName
    [R] string      consumerAgent
    [O] string      consumerModes[]
    [O] string      consumerWindowStates[]
    [O] string      customUserProfileData[]
    [O] Property    registrationProperties[]
    [O] Extension   extensions[]
```

Members:

- `consumerName`: A name (preferably unique) that identifies the Consumer [R355]. An example of such a name would be the Consumer's URL.
- `consumerAgent`: Name and version of the Consumer's vendor [R356]. This string MUST start with "productName.majorVersion.minorVersion" where "productName" identifies the product the Consumer installed for its deployment, and majorVersion and minorVersion are vendor-defined indications of the version of its product. This string MAY then contain any additional characters/words the product or Consumer wish to supply.
- `consumerModes`: An array of modes the Consumer is willing to manage. This specification defines a set of constants for a base set of modes (see section 13). This array may reference both those constants and additional custom modes of the Consumer.
- `consumerWindowStates`: An array of window states the Consumer is willing to manage. This specification defines a set of constants for a base set of window states (see section 13). This array may reference both those constants and additional custom window states of the Consumer.
- `customUserProfileData`: An array of strings each of which name a userProfile extension the Consumer supports.
- `registrationProperties`: List of registration properties. The names of these properties SHOULD be from the set declared in the `registrationPropertyDescription` from the Producer's `ServiceDescription` and are not part of this specification.
- `extensions`: A mechanism implementations MAY choose to use for extending this structure provided those extensions come from a different namespace.

6.2 register() Operation

Registration describes the transition between Producer state 1 (known) and state 2 (active) as described in section 3.12 [R352]. The Consumer establishes a relationship with the Producer that will be referenced via an opaque handle in subsequent invocations the Consumer makes of the Producer within this relationship [R350]. Both the Consumer and the Producer are free to end this relationship at any time [R500]. When the Consumer chooses to end the relationship, it MUST attempt an invocation of the **deregister()** operation so that the Producer may release related resources. The Producer MAY end the registration by invalidating the registration identifier and MUST inform the Consumer of this through a fault message on the next invocation so that the Consumer may release related resources.

```
registrationContext = register(registrationData);
```

Faults: Security.AccessDenied, Security.AuthenticationFailure, Interface.MissingParameters, Interface.OperationFailed

The returned `registrationContext` is used in all subsequent invocations to reference this registration [R362]. If the Producer's metadata declares registration is not supported (i.e. `requiresRegistration` flag was set to "false"), then it MUST be valid to pass a null `registrationHandle` to operations that have a `registrationContext` parameter. If the registration fails (e.g. failed authentication), a fault message MUST be thrown indicating this to the Consumer [R363].

A Consumer MAY register itself multiple times to a Producer with potentially different settings (e.g. security settings) resulting in multiple `registrationHandles` [R351]. Different registration contexts MUST be identified by different `registrationHandles`.

6.3 modifyRegistration() Operation

This operation provides means for the Consumer to modify a relationship with a Producer [R353].

registrationState = modifyRegistration(registrationContext, registrationData);
Faults: Security.AccessDenied, Security.InconsistentParameters, Security.InvalidRegistration, Security.AuthenticationFailure, Interface.MissingParameters, Interface.OperationFailed

The supplied parameters reference a pre-existing registration and the modifications desired. If the Producer chooses to have the Consumer provide persistent storage, the change in registration state is carried in the `registrationState` field of the returned `RegistrationState` structure.

6.4 deregister() Operation

The Consumer MUST NOT consider a relationship with a Producer ended until a successful invocation of **deregister()**.

ReturnAny deregister(registrationContext);
Faults: Security.AccessDenied, Security.InvalidRegistration, Interface.OperationFailed

After this operation is invoked, all handles created within the context of the `registrationContext` become invalid [R500][R501][R503]. It is a Producer implementation choice whether this immediately aborts in-progress operations or waits until all transient resources time out. In either case, a Consumer MUST NOT attempt to use the invalidated `registrationHandle` for subsequent invocations. An attempt to use an invalidated `registrationHandle` MUST result in a fault message indicating the registration is not valid. If the **deregister()** operation fails, the Producer MUST return a fault message specifying the reason for the failure.

7 Entity Management Interface

Producers MUST expose one or more logically distinct ways of generating markup and handling interaction with that markup [A205], which this specification refers to as **Entities**. The Producer declares the entities it exposes through its description [A104]. This declaration contains a number of descriptive parameters; in particular it includes an `entityHandle` that Consumers use to refer to the so-called "Producer_Offered_Entity". These entities are pre-configured and non-modifiable by Consumers.

In addition to the Producer_Offered_Entities, a Producer MAY expose the `EntityManagement` portType and thereby allow Consumers to clone and customize the entities the Producer offers. A Consumer MAY request a unique configuration of one of these entities, either in an opaque manner (e.g. the "edit" button common on aggregated pages which invokes an entity-generated page for setting the configuration) or by using the property definitions found in the entity's metadata to configure it in an explicit manner [R600]. Such a configured entity is called a "Consumer_Configured_Entity".

7.1 Data Structures

The normative definitions for all data structures are contained in the WSDL referenced in section 15. For the convenience of the reader, this non-normative section uses an IDL like syntax to describe these structures, where the leading [R] indicates a field is required and [O] indicates it is optional. The operations in this section introduce the following data structures:

7.1.1 MarkupType Type

The `MarkupType` data structure is used to carry entity metadata that is `markupType` specific.

```
MarkupType
    [R] string      markupType
    [R] string      locales[]
    [R] string      modes[]
    [R] string      windowStates[]
    [O] Extension   extensions[]
```

Members:

- `markupType`: A mimeType supported by the entity(e.g. *HTML, XHTML, WML, VoiceXML, cHTML*) for which the remainder of this structure applies.
- locales: An array of locales for which this `markupType` is available.
- modes: The `modes` (defined in section 5.10) that are supported by the entity for this `markupType`.
- windowStates: The `windowStates` (defined in section 5.11) that are supported by the entity for this `markupType`.
- `extensions`: A mechanism implementations MAY choose to use for extending this structure provided those extensions come from a different namespace

7.1.2 EntityDescription

The `EntityDescription` structure contains a set of fields that provide the metadata to describe the entity.

```
EntityDescription
    [R] Handle          entityHandle
    [R] MarkupType      markupTypes[]
    [O] string          groupID
    [O] LocalizedString description
    [O] LocalizedString shortTitle
    [O] LocalizedString title
    [O] LocalizedString keywords[]
    [O] string[]        producerRoles
    [O] string[]        userProfileItems
    [O] string          needSecureCommunications
    [O] boolean         usesMethodGet
    [O] boolean         userContextStoredInSession
    [O] boolean         templatesStoredInSession
    [O] boolean         hasUserSpecificState
    [O] boolean         doesUrlTemplateProcessing
    [O] Extension       extensions[]
```

Members:

- `entityHandle`: The handle by which Consumers MAY refer to this Producer_Offered_Entity. Note that Handles are restricted to a maximum length of 255 bytes.
- `markupTypes`: Each member of this array specifies metadata for a single `markupType`.
- `groupID`: Identifier for the group within which the Producer places this entity or any entities derived from it via the cloning process.
- `description`: Localized descriptions of the entity. This is intended for display in selection dialogs, etc.
- `shortTitle`: Localized short title for the entity.
- `titles`: Localized title for the entity.
- `keywords`: Array of localized keywords describing the entity which can be used for search, etc.
- `producerRoles`: Array of names for the Producer's roles that the entity can manage. Note: This support MAY be provided by the Producer service on behalf of the entity. Each role declared as supported by the entity MUST have a `RoleDescription` available to the Consumer through the Producer's `ServiceDescription`. [R416]
- `userProfileItems`: An array of strings that enumerate what portions of the `UserContext` structure the entity needs to provide full functionality. For the fields this specification defines, the named profile items an entity uses MUST all come from the "Profile Name" column of the table found in section 10.
- `needSecureCommunications` Flag that indicates whether this entity requires secure communications on all hops from the End-User to the entity, for either its markup or user interactions. Possible values are:
    - "`none`" - secure client communications not needed (default value)
    - "`some`" - secure client communications needed by some markup or user interactions
    - "`all`" - secure client communications required for all markup and user interactions.

- `usesMethodGet`: A flag indicating the entity generates markup that uses method=get in an HTML form. This will require the Consumer format its URLs in a manner that keeps browsers from throwing away information (see section 9.2.4 for a description of the difficulties in using forms with method=get). The default value of this flag is "`false`".
- `userContextStoredInSession`: A flag indicating the entity will store any supplied `UserContext` in the current session. Setting this flag to "`true`" allows the Consumer to optimize when the `UserContext` is included on operation invocations. Since some data in the `UserContext` is sensitive, many Consumers will require that secure communications be used when the information is passed. Not requiring this of all invocations can result in a significant performance difference. The default value of this flag is "`false`".
- `templatesStoredInSession`: A flag indicating the entity will store any supplied `templates` in the current session. Setting this flag to "`true`" allows the Consumer to optimize when the `templates` structure is set in `MarkupParams`. Since the content of the `templates` structure can get quite large, not requiring it to be passed can result in a significant performance difference. The default value of this flag is "`false`".
- `hasUserSpecificState`: A flag indicating the entity will store persistent state specific to each End-User. For entities setting this flag to "`true`", Consumers MAY choose to clone the entity when it is placed on an aggregated page rather than waiting for the processing described in section 5.3.3. The default value of this flag is "`false`".
- `doesUrlTemplateProcessing`: A flag indicating the entity will process any `templates` supplied so as to correctly write URLs in its markup. For entities setting this flag to "`true`", Consumers MUST provide the URL writing templates. The default value of this flag is "`false`".
- `extensions`: A mechanism implementations MAY choose to use for extending this structure provided those extensions come from a different namespace.

7.1.3 DestroyFailed

The `DestroyFailed` structure contains an `entityHandle` that was not destroyed and the reason for the failure.

```
DestroyFailed
    [R] string      entityHandle
    [R] string      reason
```

Members:

- `entityHandle`: The `entityHandle` that was not destroyed.
- `reason`: A fault code from section 0 describing the reason the destroy failed.

7.1.4 DestroyEntitiesResponse

The `DestroyEntitiesResponse` structure carries an array of failed destroys.

```
DestroyEntitiesResponse
    [R] DestroyFailed   destroyFailed[]
    [O] Extension       extensions[]
```

Members:

- `destroyFailed`: An array of failures returned by destroyEntities. This is carried as a return message since not all web stacks properly handled typed information in fault messages.

- `extensions`: A mechanism implementations MAY choose to use for extending this structure provided those extensions come from a different namespace.

7.1.5 Property Type

The `Property` data structure is used to carry typed information from the Consumer to the Producer.

```
Property
    [R] string      name
    [R] string      xmlLang
    [R] string      resourceName
    [O] Object[]    value
```

Members:

- `name`: Name of the property, must not be null
- `xmlLang`: The locale for the supplied localized value. This is carried in the WSDL using the xml:lang attribute.
- `resourceName`: The name assigned to this localized string for dereferencing into a `ResourceList` (defined in section 4.1.3) for values from other locales.
- `value`: The property's value. The type information needed to properly serialize / deserialize this value is carried in the relevant `PropertyDescription`.

7.1.6 ResetProperty Type

The `ResetProperty` data structure carries the name of a Property for which the Consumer wants the value reset to the default.

```
ResetProperty
    [R] string      name
```

Members:

- `name`: Name of the property, must not be null

7.1.7 PropertyList

A `PropertyList` gathers a set of Property structures together for transmitting between the Consumer and Producer.

```
PropertyList
    [R] Property        properties[]
    [R] ResetProperty   resetProperties[]
    [O] ResourceList    resourceList
    [O] Extension       extensions[]
```

Members:

- `properties`: Each member in this array is a `Property` structure carrying information concerning one property.
- `resetProperties`: Each member in this array is a `ResetProperty` structure carrying a property to reset to its default value.
- `resourceList`: A `ResourceList` for carrying localized information for other locales.
- `extensions`: A mechanism implementations MAY choose to use for extending this structure provided those extensions come from a different namespace.

7.1.8 PropertyDescription

Each property of an entity is described using the following structure.

```
PropertyDescription
    [R] string            name
    [R] QName             type
    [O] LocalizedString   label
    [O] LocalizedString   hint
    [O] Extension         extensions[]
```

Members:

- `name`: Name of the property, must not be null.
- `type`: Type of the property. We would encourage these to either be from the set of schema-defined types or be explicitly typed in the Producer's WSDL file. This allows the Consumers to prepare the appropriate serializer/deserializer. Other possibilities for declaring these types are the schema element of a ModelDescription or the use of the schema-defined "schemaLocation" attribute.
- `label`: A short, human-readable name for the property. Intended purpose is for display in any Consumer-generated user interface for administering the entity.
- `hint`: A relatively short description of the property. Intended for display, for example, as a tooltip in any Consumer-generated user interface for editing the property.
- `extensions`: A mechanism implementations MAY choose to use for extending this structure provided those extensions come from a different namespace.

7.1.9 ModelDescription

The set of properties of an entity are described in its metadata using the following structure.

```
ModelDescription
    [R] PropertyDescription    propertyDescriptions[]
    [O] Schema                 schema
    [O] ResourceList           resourceList
    [O] Extension              extensions[]
```

Members:

- `propertyDescriptions`: Array of property descriptions, must not be null.
- `schema`: An XML schema defining any entity-specific datatypes referenced in the `propertyDescriptions`.
- `resourceList`: A `ResourceList` for carrying localized information for other locales.
- `extensions`: A mechanism implementations MAY choose to use for extending this structure provided those extensions come from a different namespace.

7.2 getEntityDescription() Operation

This operation allows a Producer to provide information about the entities it offers in a context-sensitive manner.

```
entityDescription = getEntityDescription(registrationContext, entityContext, userContext,
                                         desiredLocales, sendAllLocales);
Faults: Security.AccessDenied, Security.InvalidProducerRole,
        Security.InconsistentParameters, Security.InvalidRegistration,
        Security.AuthenticationFailure, Interface.MissingParameters,
        Interface.OperationFailed, Interface.InvalidHandle
```

Producers may choose to restrict the information returned in `entityDescription` based on the supplied registration and user contexts. Consumers may choose to alter how they interact with an entity based on the metadata contained in the returned `entityDescription`. For security reasons related to exposing the existence of something the caller is not allowed to access, it is RECOMMENDED that a Security.AccessDenied fault be generated both for the case of the supplied `entityHandle` not being a valid reference in the scope of the supplied `registrationHandle` and for the case of the user not having access to a valid reference (i.e. by definition access is denied when the entity reference is invalid).

When generating the `EntityDescription` response the Producer MUST use the `desiredLocales` to control what locales are supplied for localized strings and `sendAllLocales` to control whether values for all available locales are included.

7.3 cloneEntity() Operation

This operation allows the Consumer to request the creation of a new entity from an existing entity.

```
entityContext = cloneEntity(registrationContext, entityContext, userContext);
Faults: Security.AccessDenied, Security.InvalidProducerRole,
        Security.InconsistentParameters, Security.InvalidRegistration,
        Security.AuthenticationFailure, Interface.MissingParameters,
        Interface.OperationFailed, Interface.InvalidHandle,
        Interface.NoCloneGenerated
```

The supplied `entityContext` MUST refer to either a Producer_Offered_Entity or a previously cloned Consumer_Configured_Entity. The initial state of the new entity MUST be equivalent to the state of the entity the supplied handle references. In the case of a Consumer_Configured_Entity that pushes the entity's persistent state to the Consumer, the `entityState` field of the returned `entityContext` structure will supply that state. The new `entityHandle` MUST be scoped by the `registrationHandle` in the supplied `registrationContext` and be unique within this registration.

If a Producer chooses to push the persistent state of its entities to the Consumer, it is RECOMMENDED that the `entityHandle` encode the supplied `registrationHandle`. In this case, it is also RECOMMENDED that the `entityState` encode the `entityHandle` so that the Producer MAY do reasonable cross checks that it is receiving a consistent set of handles and state.

The returned `entityContext` contains both the `entityHandle` and `entityState` fields for use in subsequent invocations on the configured entity. No relationship between the supplied entity and the new entity is defined by this specification. The Consumer MUST attempt to release the new `entityHandle` by invoking **destroyEntities()** when it is no longer needed.

If the Producer's metadata declares registration is not supported (i.e. `requiresRegistration` flag was set to "false"), then the Consumer MUST invoke **destroyEntities()** when it would have deregistered, passing each `entityHandle` that would have been scoped by a registration.

7.4 destroyEntities() Operation

The Consumer MUST inform the Producer that a Consumer_Configured_Entity will no longer be used by invoking **destroyEntities()** and MUST NOT consider an entity to have been destroyed until **destroyEntities()** has been successfully invoked for that entity.

DestroyEntitiesResponse destroyEntities(registrationContext, entityHandles);
Faults: Security.AccessDenied, Security.InvalidProducerRole, Security.InconsistentParameters, Security.InvalidRegistration, Security.AuthenticationFailure, Interface.MissingParameters, Interface.OperationFailed

The supplied `entityHandles` is an array of type `entityHandle`, each of which the Consumer is informing the Producer it will no longer use. The Producer returns failures to destroy supplied `entityHandles` in the `DestroyEntitiesResponse` structure. It is a choice of the Producer's implementation whether the resources related to the `entityHandles` are immediately reclaimed or whether transient resources are allowed to timeout first. A Consumer MUST NOT reference any of the supplied `entityHandles` after successfully invoking **destroyEntities()** and MAY reclaim resources related to the supplied `entityHandles` (e.g. `entityState`).

7.5 setEntityProperties() Operation

The entity state in the previous operations was opaque to the Consumer (e.g. as `entityState`). In addition, means are defined by which a Producer may publish information about state in an entity-specific manner. This is enabled through **Properties** that are declared in the metadata specific to an entity. Each property declaration includes a name and type (default = `xsd:string`) [A505][A507]. This operation enables the Consumer to interact with this published portion of an entity's state.

entityContext = setEntityProperties(registrationContext, entityContext, userContext, propertyList);
Faults: Security.AccessDenied, Security.InvalidProducerRole, Security.InconsistentParameters, Security.InvalidRegistration, Security.Authenticationfailure, Interface.MissingParameters, Interface.OperationFailed, Interface.InvalidHandle

Since **setEntityProperties()** is interacting only with the published portion of the entity's state, it MUST always be safe for the entity to modify its state (i.e. equivalent to `entityStateChange` set to "OK" for a **performInteraction()** invocation). The supplied set of property changes MUST be processed together. In particular, validation SHOULD only be done considering the entire set as partial updates could easily create an internally inconsistent set of properties. The storage of the update caused by applying the set of property updates SHOULD only occur after the Producer/entity executes this validation. It should also be noted that the Producer SHOULD serialize invocations of **setEntityProperties()** for any one `entityHandle`.

7.6 getEntityProperties() Operation

This operation provides the means for the Consumer to fetch the current values of the published entity's properties. The intention is to allow a Consumer-generated user interface to display these for administrative purposes.

```
propertyList = getEntityProperties(registrationContext, entityContext, userContext,
                                   names);
Faults: Security.AccessDenied, Security.InvalidProducerRole,
        Security.InconsistentParameters, Security.InvalidRegistration,
        Security.AuthenticationFailure, Interface.MissingParameters,
        Interface.OperationFailed, Interface.InvalidHandle
```

The supplied `names` parameter is an array of strings each of which declares a property for which the Consumer is requesting its value. The returned `propertyList` declares the current values for these properties. If the Consumer passes a null `names` parameter, the Producer / entity MUST treat this as a request to enumerate the properties of the entity.

7.7 getEntityPropertyDescription() Operation

This operation allows the Consumer to discover the published properties of an entity and information (e.g. type and description) that could be useful in generating a user interface for editing the entity's configuration.

```
modelDescription = getEntityPropertyDescription(registrationContext, entityContext,
                                                userContext, desiredLocales,
                                                sendAllLocales);
Faults: Security.AccessDenied, Security.InvalidProducerRole,
        Security.InconsistentParameters, Security.InvalidRegistration,
        Security.AuthenticationFailure, Interface.MissingParameters,
        Interface.OperationFailed, Interface.InvalidHandle
```

The `modelDescription` returned is a typed `property` view onto the portion of the entity's persistent state that the user (referenced through the `userContext`) is allowed to modify. While it is possible the set of properties MAY change with time (e.g. the entity dynamically creates or destroys properties), Producers/entities SHOULD make the returned `modelDescription` as complete as possible.

When generating the `ModelDescription` response the Producer MUST use the `desiredLocales` to control which locales are supplied for localized strings and `sendAllLocales` to controls whether values for all available locales are included.

8 Security

WSIA and WSRP compliant systems will be exposed to the same security issues as other web service systems. For a representative summary of security concerns, refer to the Security and Privacy Considerations document produced by the XML-Based Security Services Oasis TC.

It is a goal within this specification to leverage standards efforts that address web services security and to avoid defining mechanisms that will be redundant with those standards efforts. These standards generally fall into two main categories: document-level mechanisms and transport-level mechanisms.

The uses of document-level security mechanisms are not covered in this version of the specification since several important standards (particularly security policy declarations) are not yet available. Producers and Consumers wishing to apply document-level security techniques are encouraged to adhere to the mechanisms defined by WS-Security, SAML, XML-Signature, XML-Encryption, and related specifications. It is anticipated that as the web services security roadmap becomes more fully specified by standards, and support for those standards becomes widely available from infrastructure components, that these will play an important role in future versions of this specification.

For this version of the specification, emphasis is placed on using transport-level security standards (e.g. SSL/TLS) to address the security issues involved in Consumers invoking Producers on behalf of End-Users. These only require that a Producer's WSDL declare bindings for an HTTPS service entry point. Consumer's can only determine that secure transport is supported by parsing the URL for the service entry point.

8.1 Authentication of Consumer

Producer authentication of a Consumer may be achieved at the transport level through the use of client certificates in conjunction with SSL/TLS. Certificate provisioning by a Producer to a Consumer happens outside the scope of this protocol, typically as part of establishing a business relationship between the Producer and Consumer.

8.2 Confidentiality & Message Integrity

SSL/TLS may be used to ensure the contents of messages are neither tampered with nor decipherable by an unauthorized third party. Consideration needs to be given to both the communication between Producer and Consumer and communication between the End-User client and the Consumer.

For Producer - Consumer communications, the Producer declares the use of SSL/TLS in the service's WSDL by declaring an HTTPS service endpoint.

For Consumer – End-User client communications, the Consumer indicates in the `MarkupParams` structure whether or not communications with the End-User are happening in a secure manner. The entity MAY choose to change behavior based on this value, for example it may generate markup that redirects the End-User to the equivalent secure page or throw a fault indicating secure client communications are required.

8.3 Access control

A Consumer MAY implement access control mechanisms that restrict which End-Users may access which entities and operations on those entities. Additionally, a Producer MAY implement access control programmatically through the use of facilities such as an authenticated user identity or Producer roles. A standard set of roles has been defined to facilitate easy mapping of common roles. A Producer's `ServiceDescription` MAY declare support for roles including both these standard roles and any additional roles it defines. A Consumer MAY map End-Users to the roles a Producer declares in any manner it chooses, including ignoring them. Producers that use roles (standard or custom) SHOULD implement appropriate default behavior in the event a Consumer does not assert any role for the End-User.

8.4 Producer Roles

A Producer optionally declares the `producerRoles` each entity is capable of supporting in the `EntityDescription` structure described in section 7.1.2. The purpose of this declaration is to indicate to the Consumer a set of Access Controls that may be asserted for invocations on the behalf of a user.

8.4.1 Role Assertions

Since roles are an optional means for the Producer and Consumer to cooperatively apply access controls that are relevant to the user, the following examines the various combinations of Producer and Consumer choices:

1. Neither Producer nor Consumer support roles. In this case the `EntityDescription` structures from the Producer will not declare any roles and the Consumer will never assert any roles in the `UserContext` structure. Any cooperative access control issues are likely dealt with by other techniques, such as the user's identity being a shared identity.
2. Both the Producer and Consumer support roles. In this case the `EntityDescription` structures from the Producer will declare roles. The Consumer will need to map its roles for the user to this set from the Producer when asserting roles in the `UserContext` structure in order to satisfy the requirement that the asserted roles come only from the Producer published roles. The Consumer controls the mechanism by which this mapping occurs.
3. Producer supports roles, but the Consumer does not. In this case the `EntityDescription` structures from the Producer declare roles, but the Consumer will never assert any roles in the `UserContext` structure. The Producer will need to default the role it uses to process invocations.
4. The Producer does not support roles, but the Consumer does. In this case the `EntityDescription` structures from the Producer will not declare any roles and the Consumer will need to map all of its roles to null. This results in the Consumer being able to process invocations normally without violating the requirement that only the roles a Producer publishes being used in the `UserContext` structure.

8.4.2 Standard Roles

To ease the mapping of End-Users to roles and to facilitate plug-and-play, the following standard role names are provided along with an abstract definition of semantics associated with each. These definitions suggest progressively restrictive levels of access to the entity and are provided as guidelines only. The specific semantics of these roles are left to each Producer's implementation.

Administrator:	This role typically grants the highest level of access to functionality of an entity.
User:	This role is typically associated with End-Users who may personalize some set of properties for an entity.
Guest:	This role is typically associated with End-Users who may view an entity on a page but not modify any of its properties

9 Markup

This section covers the issues related to entities generating markup that Consumers could safely aggregate into a page and then properly process End-User interactions [A301].

9.1 Encoding

The Consumer MUST indicate to the entity the preferred character encoding, using the `characterSet` field of the `markupParams` structure. It is up to the entity to generate markup that complies with this encoding. The entity may generate markup in the UTF-8 character set encoding if it is unable to generate the requested `characterSet`. If it is unable to generate markup in either of these character sets, the entity MUST return a fault message to the Consumer. The Producer MUST use the same character set for the XML response message as was used to generate the markup.

9.2 URL Considerations

As part of its markup, an entity will often need to create URLs that reference the entity itself. For example, when an End-User activates one of these URLs, by clicking a link or submitting a form, the result should be a new invocation targeted to the entity. This section deals with the different possibilities for how the entity can encode these URLs in its markup.

URLs embedded in a markup fragment often cannot (or should not) be direct links to the Producer for a number of reasons:

- URLs the entity writes in its markup will be invoked or accessed by the End-User operating on the client. In the general case however it is only guaranteed that the client has direct access to the Consumer; the Producer may be shielded from the client via a firewall. So the Consumer needs to intercept URLs and route them to the Producer [A103].
- The Consumer may want to intercept URLs to perform additional operations, enrich the request with context information or do some book keeping
- The client might not be able to directly invoke the Producer, e.g. if the client is a browser that cannot issue SOAP requests to the Producer but can only talk HTTP to the Consumer. In this case the Consumer must translate the request into the correct protocol.

This implies that URLs must be encoded so that the Consumer intercepts them and re-routes them to the correct entity at the Producer, including the proper context. Because the same entity can be instantiated more than once in a single page, encoded URLs will have to allow the Consumer to track the entity to which the request is targeted. The problem is that the Producer requires Consumer-dependent information to write such a link. In principle there exist two options to make the encoded URLs point back to the Consumer and consist of all necessary information for the Consumer to properly process the activation of an URL:

- The Consumer can pass information on its context to the entity. The entity exploits this information during URL encoding so the resulting URL can be passed without further modification to the client. The advantages of this technique are efficiency and exploitation of these settings, even in client-side scripting. The disadvantage is that the entity will not be able to serve static content as the content depends on Consumer runtime settings.

- The entity can use a special syntax to encode its URLs. It is then the task of the Consumer to detect URLs in the markup and modify them according to its requirements. The markup generated by the entity is now Consumer-independent, allowing the entity to exploit caching mechanisms or even to serve static content. However, the Consumer will be more complex, as it needs to parse the markup to locate and rewrite the URLs, requiring extra processing time. Consumers SHOULD seek to minimize this impact on performance by using efficient encoding and parsing mechanisms (for example, the Boyer-Moore algorithm[11]).

As there is no clear advantage to either technique, both styles of URL encoding are supported (see sections 9.2.1 and 9.2.2). This facilitates the capabilities both of the Producer and the Consumer with regards to the ability to adapt the generated markup and requires that the following semantics be followed:

1. IF an entity's metadata declares it is willing to process URL templates, then the Consumer MUST supply templates for the entity to use.
2. IF an entity is unable to completely write the URLs for its markup, it MUST set the `needsUrlRewriting` flag in `MarkupResponse` to "true".
3. If the `needsUrlRewriting` flag in `MarkupResponse` is "true", then the Consumer MUST parse the returned markup and rewrite URLs conforming to the definitions in Section 9 of this specification.

Note: In principle it would not be necessary to mark URLs in a special way. The Consumer could always analyze the markup semantically and syntactically, detect URLs and rewrite them. This approach however would be very difficult and time consuming to implement for the Consumer, for reasons including that such a rewriting algorithm would be dependent on the markup type and version. Therefore both the Consumer and the Producer URL writing scenarios are introduced for convenience.

Entities MUST adopt the following convention for including non-ASCII characters within URLs in order to comply with W3C recommendations.

1. Represent each character in UTF-8 (see [RFC2279]) as one or more bytes.
2. Escape these characters with the URI escaping mechanism (i.e., by converting each byte to %HH, where HH is the hexadecimal notation of the character value).

This procedure results in a syntactically legal URI (as defined in [RFC1738], section 2.2 or [RFC2141], section 2) that is independent of the character encoding[12] to which the document carrying the URI may have been transcoded.

[11] http://www.cs.utexas.edu/users/moore/best-ideas/string-searching/
[12] http://www.w3.org/TR/html40/charset.html - doc-char-set

9.2.1 Consumer URL Writing

All URLs the Consumer needs to write are demarcated in the markup by a token (wsrp-rewrite) both at the start (with a "?" appended to clearly delimit the start of the name/value pairs) and end (preceded by a "/" to form the end token) of the URL declaration. The Consumer will have to locate the start and end token in the markup stream and use the information between the tokens to write the URL correctly. Details on this URL writing process are Consumer-specific and out of scope for this specification. The content between this pair of tokens follows the pattern of a query string (name/value pairs separated by "&" characters) with several well-known parameter names specifying what the Consumer needs in order to both correctly write the URL and then process it when an End-User activates it. This results in an URL declaration of the form:

[3rd F2F: Change token to wsrp-rewrite for now. Revisit value of less human readable token once an implementation is available to test the impact.]

```
wsrp-rewrite?urlType&name1=value1&name2=value2 .../wsrp-rewrite
```

The Consumer is NOT REQUIRED to process URLs not conforming to this format and MAY choose to pass them unchanged, replace them with error text, do a best effort processing or invalidate the entire markup fragment. The Consumer is NOT REQUIRED to process escaped characters in parameter names, but rather MAY pass them unchanged to either the User Agent (during URL rewriting) or the Producer (during processing of an activated URL).

The following well-known parameter names (e.g. replacing "*urlType*", "name1" and "name2" above) are defined:

9.2.1.1 urlType

This parameter MUST be specified first by replacing the string "*urlType*" in the template above with one of the following values. Well-known parameter names that are valid for only one of the urlTypes are described relative to that urltype while the remainder are described later. The following values are defined for "*urlType*":

9.2.1.1.1 Action

Activation of the URL will result in an invocation of **processInteraction()** on the entity that generated the markup. Prior to invoking this method the Consumer MUST analyze the query string parameters of the URL to determine if a mode and/or window change is specified. The Consumer MUST process all mode and window state change requests invoking the operation. All query string parameters not defined by this specification MUST be passed to **processInteraction()** as `requestParameters`. In addition the Consumer MUST check for the presence of the `wsrp-navigationalState` parameter. If this parameter is present its value MUST be passed in the `navigationalState` field of the `MarkupParams` structure. If there is no such parameter, the Consumer MUST supply the current navigational state of the entity instead.

9.2.1.1.2 BlockingAction

Activation of the URL will result in an invocation of **processBlockingInteraction()** on the entity that generated the markup. Prior to invoking this method the Consumer MUST analyze the query string parameters of the URL to determine if a mode and/or window change is specified. The Consumer MUST process all mode and window state change requests invoking the operation. All query string parameters not defined by this specification MUST be passed to **processBlockingInteraction()** as `requestParameters`. In addition the Consumer MUST check for the presence of the `wsrp-navigationalState` parameter. If this parameter is present its value MUST be passed in the `navigationalState` field of the `MarkupParams` structure. If there is no such parameter, the Consumer MUST supply the current navigational state of the entity instead.

9.2.1.1.3 Render

Activation of the URL will result in an invocation of **getMarkup()**. This mechanism permits an entity's markup to contain URLs, which do not involve changes to local state, to avoid the overhead of two-step processing by directly invoking **getMarkup()**. The URL MAY specify a `wsrp-navigationalState` parameter different from the current `navigationalState` for the entity as this allows state changes resulting in different markup being rendered. The Consumer MUST pass all the URL's query string parameters not defined by this specification as `requestParameters` in the `MarkupParams` data structure.

9.2.1.1.4 Resource

Activation of the URL will result in the Consumer fetching the underlying resource, possibly in a cached manner, and returning it to the End-User. For the HTTP protocol this maps to a "get" on the underlying resource. The URL for the resource (including any query string parameters) is encoded as the value of the `wsrp-url` parameter.

9.2.1.1.4.1 wsrp-url

This parameter provides the actual URL to the resource. Note that this needs to be an absolute URL as the resource fetch will likely use HTTP GET instead of a web service invocation. Also note that since this URL will appear as a parameter value, it MUST be strictly encoded (i.e. "&", "=", "/", and "?" url-escaped) so that special URL characters do not invalidate the processing of the enclosing URL.

9.2.1.1.4.2 wsrp-rewriteResource

This boolean informs the Consumer that the resource needs to be parsed for URL rewriting. Normally this means that there are names that will be cross-referenced between the markup and this resource (e.g. javascript references). Note that this means the Consumer needs to deal with rewriting unique "namespaced" names in a set of documents, rather than treating each document individually. Consumers MAY want to process such resources in a manner that allows caching of the resulting resource by the End-User's browser. In particular, Consumers MAY process namespace rewriting by using a prefix that is unique to the user/entity pair provided any such prefix is held constant for the duration of the user's session with any one entity.

9.2.1.1.5 Namespace

This tells the Consumer that the "URL" contains a name that needs to be unique on the aggregated page (e.g. a form field's name or the name of a javascript method). While this is not technically a URL, providing this functionality in this manner limits the performance impacts of Consumer parsing to a single pass of the markup. The actual name that needs to be rewritten is encoded in the `wsrp-token` parameter. See section 9.2.4 for more details on namespace rewriting.

9.2.1.1.5.1 wsrp-token

This parameter provides the actual token that is to be namespaced.

9.2.1.2 wsrp-navigationalState

The value of this parameter defines the navigational state the Consumer MUST send to the Producer when the URL is activated. If this parameter is missing, the Consumer MUST send the current navigational state.

9.2.1.3 wsrp-mode

Activating this URL includes a request to change the `mode` parameter in `markupParams` into the mode specified as the value for this parameter name. This must be one of the modes detailed in section 5.10 or a custom mode the Consumer specified as supported during registration. The Consumer MUST process this URL request to change the mode prior to invoking operations on the entity. If the requested mode change is for an invalid or unavailable mode (e.g. policy prohibits this End-User from entering that mode), the Consumer SHOULD leave the mode unchanged.

9.2.1.4 wsrp-windowState

Activating this URL includes a request to change the `windowState` parameter in `markupParams` into the window state specified as the value for this parameter name. This must be one of the values detailed in section 5.11 or a custom window state the Consumer specified as supported during registration. The Consumer MUST process this URL request to change the window state prior to invoking operations on the entity. If the requested window state change is for an invalid or unavailable window state (e.g. policy prohibits this End-User from entering that window state), the Consumer SHOULD leave the window state unchanged.

9.2.1.5 wsrp-secureURL

A boolean indicating the resulting URL should involve secure communications between the client and Consumer, as well as between the Consumer and Producer. The default value of this boolean is "false".

9.2.1.6 URL examples

Here are some examples of what "URL"s following this format:

- Load a resource http://test.com/images/test.gif:
  - wsrp-rewrite?Resource&wsrp-url=http%3A%2F%2Ftest.com%2Fimages%2Ftest.gif/wsrp-rewrite
- Declare this token as needing namespacing:
  - wsrp-rewrite?Namespace&wsrp-token=myFunc/wsrp-rewrite
- Declare a secure interaction back to the entity:

- wsrp-rewrite?Action&wsrp-secureURL=true&wsrp-navigationalState=a8h4K5JD9&myParam=foobar/wsrp-rewrite

- Request the Consumer render the entity in a different mode and window state:
  - wsrp-rewrite?Render&wsrp-mode=help&wsrp-windowState=maximized/wsrp-rewrite

9.2.2 Producer URL Writing

Producers and entities often are willing to properly write URLs for the Consumer, as this decentralizes the preparation of the page for rendering and thereby may provide better page load performance to the End-User. At other times, entities choose to dynamically compute the URL in script on the End-User's machine just before activating the URL. To enable these functionalities, several templates are defined by which the Consumer indicates how it needs URLs formatted in order to process them properly. These all take the form of a simple template, for example:

```
http://www.Consumer.com/path/urlType_{urlType}?mode={wsrp-mode}&…
```

The definition of the content of this template is completely up to the Consumer. It may consist of zero or more replacement tokens. These tokens are enclosed in curly braces (i.e. "{" and "}") and contain the name of the parameter that should be replaced. All content outside the {} pairs MUST be treated by the Producer/entity as constants the Consumer wishes to receive when the URL is activated. The list of defined parameter names matches those in section 9.2.1, with the addition of `wsrp-requestParameters`. This parameter defines where the entity should place data items it needs in the `requestParameters` field of the `markupParams` data structure when the URL is activated. The value replacing `wsrp-requestParameters` should follow the pattern of a URL query string (properly encoded name/value pairs separated by the "&" character) and be strictly encoded (i.e. "&", "=", "/", and "?" url-escaped) as it likely will be the value of a parameter in the query string of the full URL.

This specification defines a `Templates` structure that supplies these values. The Consumer MUST supply these templates for entities specifying `doesUrlTemplatesProcessing` as "true". Entities also specifying `templatesStoredInSession` as "true" enable the Consumer to only send these until the Producer returns a `sessionHandle`. The following describe in more detail the usage of the fields from the `Templates` structure.

9.2.2.1 ActionTemplate

Activation of the URL will result in an invocation of **performInteraction()**. The Consumer SHOULD integrate placeholders for the tokens `wsrp-navigationalState`, `wsrp-entityMode` and `wsrp-windowState` in its template. The Consumer MUST apply changes in mode or state before invoking **performInteraction()**.

9.2.2.2 SecureActionTemplate

Equivalent to `ActionTemplate` using secure communications.

9.2.2.3 BlockingActionTemplate

Activation of the URL will result in an invocation of **performBlockingInteraction()**. The Consumer SHOULD integrate placeholders for the tokens `wsrp-navigationalState`, `wsrp-entityMode` and `wsrp-windowState` in its template. The Consumer MUST apply changes in mode or state before invoking **performBlockingInteraction()**.

9.2.2.4 SecureBlockingActionTemplate

Equivalent to `BlockingActionTemplate` using secure communications.

9.2.2.5 RenderTemplate

Activation of the URL will result in an invocation of **getMarkup()**. The Consumer SHOULD integrate placeholders for the tokens `wsrp-navigationalState`, `wsrp-entityMode` and `wsrp-windowState` in its template. The Consumer MUST apply changes in mode or state before invoking **getMarkup()**.

9.2.2.6 SecureRenderTemplate

Equivalent to `RenderTemplate` using secure communications.

9.2.2.7 ResourceTemplate

Activation of the URL will result in the Consumer fetching the underlying resource, possibly in a cached manner, and returning it to the End-User. For the HTTP protocol this maps to a "get" on the underlying resource. The Consumer SHOULD integrate placeholders for the token `wsrp-url` to allow the entity to place the address of the URL.

9.2.2.8 SecureResourceTemplate

Equivalent to `ResourceTemplate` using secure communications.

9.2.2.9 DefaultTemplate

A template whose value is to be used as the default value for any template whose value is not supplied. Consumers setting just this template value SHOULD also set a separate default for the secure communications templates using `SecureDefaultTemplate`, unless the default template already uses secure communications. Consumers not supplying all the other templates MUST set a value for this template.

9.2.2.10 SecureDefaultTemplate

This template provides a value that overrides the one supplied for `DefaultTemplate` for those templates whose names begin with "Secure".

9.2.2.11 NamespacePrefix

The Producer can use the content of this field as a prefix for tokens that need to be unique on the aggregated page.

9.2.3 BNF Description of URL formats

```
ConsumerURL = BeginToken UrlType NameValuePairs EndToken
BeginToken = “wsrp-rewrite?”
EndToken = “/wsrp-rewrite”
UrlType = “Action” | “BlockingAction” | “Render” | “Resource” | “Namespace”
NameValuePairs = (“&” NameValuePair)*
NameValuePair = TextPair | BooleanPair | EntityPair
TextPair = TextName “=“ Text
TextName = “wsrp-navigationState” | “wsrp-mode” | “wsrp-windowState” | “wsrp-
url” | “wsrp-token”
BooleanPair = BooleanName “=“ BooleanValue
BooleanName = “wsrp-secureURL” | “wsrp-rewriteResource”
BooleanValue = (“true” | “false”)
EntityPair = Text “=“ Text
Text = <any URL-encoded textual characters>

ProducerURLTemplate = (Text* ReplacementToken*)*
ReplacementToken = “{“ ParameterName “}”
ParameterName = TextName | BooleanName | “UrlType” | “wsrp-requestParameters”
```

9.2.4 Method=get in HTML forms

Browsers often throw away any query string from the URL indicated with the form’s action attribute when generating the URL they will activate when the form’s method is “get”. This is the simplest means for them to generate a valid query string. The difficulty this causes is that Consumer’s often prefer to store the information they will use when the URL is activated as query string parameters. As a result, entities that use HTML forms with method=get MUST specify `usesMethodGet` as “true” in their `EntityDescription`. Consumers choosing to use such entities MUST format their URLs such that URL activations are processed correctly, regardless of whether Consumer or Producer URL writing is in use.

9.3 Namespace Encoding

Aggregating multiple entities from different sources can potentially result in naming conflicts for various types of elements: named attributes, form fields, JavaScript functions and variables, etc. Such tokens must therefore be encoded to an entity-instance specific namespace [A301]. The entity does this by prefixing the name of the resource with a namespace prefix.

If namespace encoding is used for form parameters or other data the entity receives as in `markupParams`, then the Consumer MUST strip the namespace prefix from the parameter names before passing them to the Producer. This means the entity logic can be agnostic regarding namespace issues except when encoding parameters in the markup.

Similar to the case of URL rewriting, two options exist to obtain a namespace prefix.

9.3.1 Consumer Rewriting

The entity uses the static, predefined method (section 9.2.1) to denote tokens that need a namespace prefix. The Consumer parses the markup fragment to locate these tokens and replace them with a namespaced token unique in the context of the page aggregation. This namespaced token MUST be the same for occurrences of a token in the set of documents being processed while building the aggregated page. This is done using the same method as URL rewriting and is described in that section. It is expected that the length this method adds to names might make it unwieldy to content authors, but the expectation that tooling/runtime support can alleviate most of this burden led to reusing this singular technique for the Consumer parsing/rewriting the markup.

9.3.2 Producer Writing

The entity uses a namespace provided by the Consumer to prefix these tokens in its markup. The Consumer ensures this prefix is unique for the page aggregation, so the Consumer is not required to process the markup. The Consumer supplies the prefix the entity needs to use via the `NamespacePrefix` field in the `Templates` structure.

9.4 Markup Fragment Rules

Because the Consumer aggregates the markup fragments produced by entities into a single page, some rules and limitations are needed to ensure the coherence of the resulting page to be displayed to the End-User. For efficiency reasons, Consumers are not required to validate the markup fragments returned by the entity. So in order to be aggregated, the entity MUST ensure its markup conforms to the following general guidelines [A300][A302].

The disallowed tags listed below are those tags that impact other entities or may even break the entire aggregated page. Inclusion of such a tag invalidates the whole markup fragment, which the Consumer MAY replace with an error message.

9.4.1 HTML

9.4.1.1 Disallowed Tags

Since the Consumer may implement its aggregation in many ways, including using frames, some Consumers may actually support these disallowed tags. However, in order to be certified as being a cross-platform entity, an entity MUST NOT use the following tags:

base
body
frame
frameset
head

html
title

9.4.1.2 Other Tags

There are some tags that are specifically prohibited by the HTML specification from occurring outside the <head> of the document. However, browser implementations offer varying levels of support. For example, current versions of Internet Explorer and Netscape Navigator both support the style tag anywhere within the document.

It is up to the entity developer to decide when using such tags is appropriate. Here is a list of tags that fit this description:

link
meta
style

9.4.2 XHTML

9.4.2.1 Disallowed Tags

base
body
head
html
title

9.4.2.2 Other Tags

link
meta
style

9.4.3 XHTML Basic

9.4.3.1 Disallowed Tags

base
body
head
html

title

9.4.3.2 Other Tags

link
meta
style

9.5 CSS Style Definitions

One of the goals of an aggregated page is a common look-and-feel across the entities contained on that page. This not only affects the decorations around the entities, but also their content. Using a common CSS style sheet for all entities, and defining a set of standard styles, provides this common look-and-feel without requiring the entities to generate Consumer-specific markup. Entities SHOULD use these style definitions in order to participate in a uniform display of their content by various Consumers.

This section defines styles for a variety of logical units in the markup.

9.5.1 Links (Anchor)

A custom CSS class is not defined for the <a> tag. The entity should use the default classes when embedding anchor tags.

9.5.2 Fonts

The font style definitions affect the font attributes only (i.e. font face, size, color, style, etc.).

Style	Description	Example
portlet-font	Font attributes for the “normal” fragment font. Used for the display of non-accentuated information.	Normal Text
portlet-font-dim	Font attributes similar to the portlet-font but the color is lighter.	Dim Text

If an entity author wants a certain font type to be larger or smaller, they should indicate this using a relative size.

Example1: Important information

Example1: Small and dim

9.5.3 Messages

Message style definitions affect the rendering of a paragraph (i.e. alignment, borders, background color, etc.) as well as text attributes.

Style	Description	Example
portlet-msg-status	Status of the current operation.	*Progress: 80%*

portlet-msg-info	Help messages, general additional information, etc.	Info about
portlet-msg-error	Error messages.	Portlet not available
portlet-msg-alert	Warning messages.	*Timeout occurred, try again later*
portlet-msg-success	Verification of the successful completion of a task.	**Operation completed successfully**

9.5.4 Sections

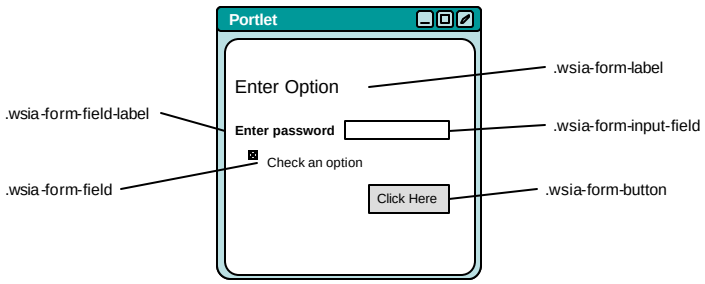
Section style definitions affect the rendering of markup sections such as table, div and span (i.e. alignment, borders, background color, etc.) as well as their text attributes.

Style	Description
portlet-section-header	Table or section header
portlet-section-body	Normal text in a table cell
portlet-section-alternate	Text in every other row in the cell
portlet-section-selected	Text in a selected cell range
portlet-section-subheader	Text of a subheading
portlet-section-footer	Table or section footnote
portlet-section-text	Text that belongs to the table but does not fall in one of the other categories (e.g. explanatory or help text that is associated with the section).

9.5.5 Forms

5 Form styles define the look-and-feel of the elements in an HTML form.

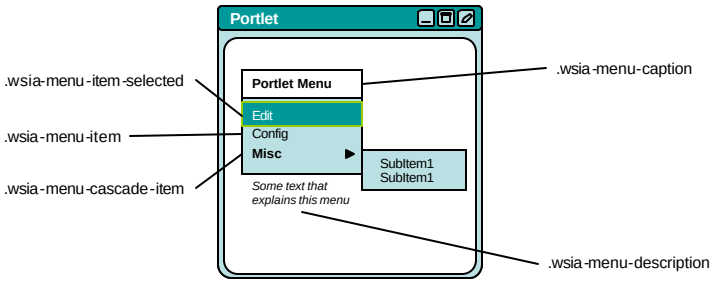
Style	Description
portlet-form-label	Text used for the descriptive label of the whole form (not the labels for fields.
portlet-form-input-field	Text of the user-input in an input field.
portlet-form-button	Text on a button
portlet-icon-label	Text that appears beside a context dependent action icon.
portlet-dlg-icon-label	Text that appears beside a “standard” icon (e.g. Ok, or Cancel)
portlet-form-field-label	Text for a separator of fields (e.g. checkboxes, etc.)
portlet-form-field	Text for a field (not input field, e.g. checkboxes, etc)



9.5.6 Menus

Menu styles define the look-and-feel of the text and background of a menu structure. This structure may be embedded in the aggregated page or may appear as a context sensitive popup menu.

Style	Description
portlet-menu	**General menu settings such as background color, margins, etc**
portlet-menu-item	Normal, unselected menu item.
portlet-menu-item-selected	Selected menu item.
portlet-menu-item-hover	Normal, unselected menu item when the mouse hovers over it.
portlet-menu-item-hover-selected	Selected menu item when the mouse hovers over it.
portlet-menu-cascade-item	Normal, unselected menu item that has sub-menus.
portlet-menu-cascade-item-selected	Selected sub-menu item that has sub-menus.
portlet-menu-description	Descriptive text for the menu (e.g. in a help context below the menu)
portlet-menu-caption	Menu caption



10 User Information

This specification provides a mechanism for entities to use End-User information as a means for personalizing behavior to the current user [A600][A606]. A standard set of user attributes has been derived from P3P User Data and is defined in Section 11. Extensibility is supported in both directions; the Consumer indicates to the Producer during registration what set of user profile extensions it supports, and an entity's metadata declares what user profile items it uses (including any extended user profile items). The following table maps the nested profile structures to profileNames:

Profile Name	Structure 1	Structure 2	Field Name
name/prefix	name		prefix
name/given	name		given
name/family	name		family
name/middle	name		middle
name/suffix	name		suffix
name/nickName	name		nickName
birthDate			birthDate
gender			gender
employerInfo/employer	employerInfo		employer
employerInfo/department	employerInfo		department
employerInfo/jobTitle	employerInfo		jobTitle
homeInfo/address/name	homeInfo	address	name
homeInfo/address/street	homeInfo	address	street
homeInfo/address/city	homeInfo	address	city
homeInfo/address/stateprov	homeInfo	address	stateprov
homeInfo/address/country	homeInfo	address	country
homeInfo/address/org	homeInfo	address	org
homeInfo/telephone	homeInfo		telephone
homeInfo/email	homeInfo		email
homeInfo/online	homeInfo		online
workInfo/address/name	workInfo	address	name
workInfo/address/street	workInfo	address	Street
workInfo/address/city	workInfo	address	City
workInfo/address/stateprov	workInfo	address	stateprov
workInfo/address/country	workInfo	address	country

workInfo/address/org	workInfo	address	org
workInfo/telephone	workInfo		telephone
workInfo/email	workInfo		email
workInfo/online	workInfo		online

Entities that need access to user information MUST declare in its metadata the specific user profile fields it needs using the names specified above.

Consumers supplying additional custom profile fields are encourage to publish a similar mapping between profileNames and the custom fields.

10.1 Passing User Information

User information MAY be passed to the Producer when a Consumer invokes certain operations. A Consumer SHOULD provide the specific fields the entity declared it needs, unless the information is not available or is restricted by policy (e.g. privacy policy).

10.2 User Identity

Mechanisms that support federation of user identity between web services systems are defined in other specifications, such as WS-Security and SAML. If a Consumer and Producer need to share a common identity for an End-User, it is recommended that compliance with these standards be the means to passing the required information.

It is anticipated that some entities will interact with one or more back-end applications that require a user identity for the End-User. If the user identity required by the back-end application is not the same as that authenticated or otherwise supplied by the Consumer, the entity MAY request the End-User to provide the necessary information (preferably using secure transport) for use with the back-end application via markup interactions (e.g. display a form that prompts for a user identity and any security tokens (such as a password) for the back-end system).

11 Data Structures

It is often necessary to pass data to operations. Wherever possible typed data object are defined as the transport mechanism for this data [A504][A505]. Extensibility elements are also provided for vendor or application-specific data extensions. Many of these data structures have been described in the sections describing the operations in which they are referenced here (with hyperlinks back to those explanations) in order to have all the structures defined in one place. This non-normative section uses an IDL like syntax for describing these structures for the convenience of the reader. The normative definitions for these structures are defined by the WSDL referenced in section 15.

Extensibility of all the data structures is defined using the schema syntax for including arbitrary content from other namespaces.

11.1 BlockingInteractionResponse Type

The `BlockingInteractionResponse` structure contains the various items **performBlockingInteraction()** can return. This data structure is defined in section 5.1.10.

11.2 CacheControl Type

The `CacheControl` structure contains a set of fields needed for the entity to manage cached markup fragments. This data structure is defined in section 5.1.4.

11.3 ClientData Type

The `MarkupParam` structure types carries information concerning the user agent and client device using this type.

```
ClientData
    [R] string      userAgent
    [O] Extension   extensions[]
```

Members:

- `userAgent`: String identifying the UserAgent of the End-User.
- `extensions`: A mechanism implementations MAY choose to use for extending this structure provided those extensions come from a different namespace.

11.4 EntityContext Type

The `EntityContext` structure is used as a parameter on many operations to supply the entity information that was pushed to the Consumer. This data structure is defined in section 5.1.3.

11.5 EntityDescription Type

The `EntityDescription` structure contains a set of fields that provide the metadata to describe the entity. This data structure is defined in section 7.1.2.

11.6 Extension Type

The `Extension` structure contains the payload extension mechanism for vendor and application extensions. This data structure is defined in section 4.1.1.

11.7 DestroyFailed Type

- The `DestroyFailed` structure contains a set of fields that provide the metadata to describe the entity and in described in section 7.1.3.

11.8 DestroyEntitiesResponse Type

- The `DestroyEntitiesResponse` structure carries an array of failed destroys and is described in section 7.1.4.

11.9 Handle Type

Handles are opaque references that are passed between the Consumer and Producer.

Handles are represented as restricted strings in the protocol. Although a string is principally unlimited in length, the length of the handle is restricted for the following reasons:

- Handles may be stored in databases and may be used for indexing.

- The Consumer will likely embed handles in client URLs.
- Comparison of handles should be efficient.

The maximum length of a handle is restricted to 255 bytes. The Consumer MAY ignore any character in a handle that falls outside this range.

```
Handle extends string (maximum length = 255)
```

11.10 InteractionParams Type

The `InteractionParams` structure contains fields specific to invoking the **performInteraction()** operation. This data structure is defined in section 5.1.14.

11.11 InteractionResponse Type

The `InteractionResponse` structure contains the various items **performInteraction()** can return. This data structure is defined in section 5.1.9.

11.12 LocalizedString Type

The `LocalizedString` structure carries the value for the string in a particular locale, which locale the value is for, and a `resourceName` for locating values for other locales. This data structure is defined in section 4.1.2.

11.13 MarkupContext Type

The `MarkupContext` structure contains a set of fields related to and including the markup an entity has generated. This data structure is defined in section 5.1.7.

11.14 MarkupParams Type

The `MarkupParams` structure contains a set of fields needed for the entity to generate markup that will enable the End-User to visualize the state of the entity. This data structure is defined in section 5.1.6.

11.15 MarkupResponse Type

The `MarkupResponse` structure contains fields for returning various items in response to a **getMarkup()** invocation. This data structure is defined in section 5.1.8.

11.16 MarkupType Type

The `MarkupType` structure contains fields for describing entity metadata that is specific to a `markupType`. This data structure is defined in section 7.1.1.

11.17 ModelDescription Type

The `ModelDescription` structure is used to collect a set of `PropertyDescriptions` and a schema they may reference for type information. This data structure is defined in section 7.1.9.

11.18 Property Type

The `Property` structure is used to transmit a new value for a particular property. This data structure is defined in section 7.1.3.

11.19 PropertyDescription Type

The properties of an entity are described in its metadata using the `PropertyDescription` structure. This data structure is defined in section 7.1.8.

11.20 PropertyList Type

A `PropertyList` gathers an array of `Property` structures and an array of `ResetProperty` structures together for transmitting between the Consumer and Producer. This data structure is defined in section 7.1.7.

11.21 RegistrationContext Type

The `RegistrationContext` structure contains fields related to a particular registration of a Consumer with a Producer. It is returned by the **register()** operation and is a required parameter on most other operations. This data structure is defined in section 4.1.10.

11.22 RegistrationState Type

The `RegistrationState` structure contains fields related to a particular registration of a Consumer with a Producer. It is returned by the **modifyRegistration()** operation and contains the fields of a `registrationContext` that allow a Producer to push the storage of state at registration scope to the Consumer. This data structure is defined in section 4.1.9.

11.23 RegistrationData Type

The `RegistrationData` structure provides the means for the Consumer to supply the data required for registration with a Producer as well as protocol extensions that it supports. This data structure is defined in section 6.1.1.

11.24 ResetProperty Type

The `ResetProperty` structure is used to transmit a request to reset a property to its default value. This data structure is defined in section 7.1.6.

11.25 Resource Type

The `Resource` structure gathers a set of `ResourceValue` structures together to describe the locale dependent values of a single resource. This data structure is defined in section 4.1.4.

11.26 ResourceList Type

The `ResourceList` structure gathers a set of `Resource` structures together. This data structure is defined in section 4.1.3.

11.27 ResourceValue Type

The `ResourceValue` structure carries the value of a resource for a particular locale. This data structure is defined in section 4.1.5.

11.28 RoleDescription Type

The `RoleDescription` structure describes a role to the Consumer. This data structure is defined in section 4.1.6.

11.29 RuntimeContext Type

The `RuntimeContext` structure is where the Consumer provides transient information to the Producer. This data structure is defined in section 5.1.2.

11.30 ServiceDescription Type

The `ServiceDescription` structure contains a set of fields that describe the offered services of the Producer. This data structure is defined in section 4.1.7.

11.31 SessionContext Type

The `SessionContext` structure carries session oriented information to the Consumer whenever a Producer initializes a new session. This data structure is defined in section 5.1.1.

11.32 StateChange Type

The `StateChange` type is a restriction on string that can only take the values “OK”, “Clone”, and “Fault”. This data structure is defined in section 5.1.12.

11.33 Templates Type

The `Templates` structure contains a set of fields that enable Producer URL writing. This data structure is defined in section 5.1.5.

11.34 UploadContext Type

The `UploadContext` structure carries fields related to data uploading to the Producer as a result of an interaction. This data structure is defined in section 5.1.13.

11.35 UserContext Type

The `UserContext` structure supplies End-User specific data to operations. Note that this does not carry user authentication type information (e.g. userID / password) as quite flexible mechanisms for communicating this information are being defined elsewhere (e.g. WS-Security (see section 3.1.2) defines how to carry User Information in a SOAP header). This data structure is defined in section 4.1.8.

11.36 User Profile Types

The `UserProfile` structure is used to carry information about the End-User. The entity uses the `userProfileItems` in its metadata to describe the fields it uses to generate markup from this set and any others the Consumer indicated were available when it registered. See section 10 for a complete description of this portion of the protocol.

UserProfile

```
    [O] UserName      name
    [O] DateTime      birthdate
    [O] string        gender
    [O] EmployerInfo  employer
    [O] LocationInfo  homeInfo
    [O] LocationInfo  workInfo
    [O] Extension     extensions[]
```

Members:

- `name`: A structure containing the various fields for the End-User's name.
- `birthdate`: The End-User's birthdate. This uses the schema-defined datatype for DateTime rather than Date as not all web stacks serialize / deserialize Date properly.
- `gender`: The End-User's gender ("M" = male, "F" = female).
- `employer`: A structure containing various fields for the End-User employer's information.
- `homeInfo`: The End-User's home location information.
- `workInfo`: The End-User's work location information.
- `extensions`: A mechanism implementations MAY choose to use for extending this structure provided those extensions come from a different namespace.

11.36.1 UserName Type

The `UserName` structure carries the detailed fields for the parts of an End-User's name.

```
UserName
    [O] string      prefix
    [O] string      given
    [O] string      family
    [O] string      middle
    [O] string      suffix
    [O] string      nickName
    [O] Extension   extensions[]
```

Members:

- `prefix`: Examples include Mr, Mrs, Ms, Dr, etc.
- `given`: The End-User's first or given name.
- `family`: The End-User's last or family name.
- `middle`: The End-User's middle name(s) or initial(s).
- `suffix`: Examples include Sr, Jr, III, etc.
- `nickName`: The End-User's preferred nick name.
- `extensions`: A mechanism implementations MAY choose to use for extending this structure provided those extensions come from a different namespace.

11.36.2 EmployerInfo Type

The `EmployerInfo` structure contains the detailed fields concerning the End-User's employer.

```
Employerinfo
    [O] string      employer
    [O] string      department
    [O] string      jobTitle
```

```
    [O] Extension   extensions[]
```

Members:

- `employer`: The name of the employer.
- `department`: The name of the department the End-User works within.
- `jobTitle`: The title of the End-User's job.
- `extensions`: A mechanism implementations MAY choose to use for extending this structure provided those extensions come from a different namespace.

11.36.3 LocationInfo Type

The `LocationInfo` structure is used to describe a location for the End-User.

```
LocationInfo
    [O] Address     address
    [O] string      telephone[]
    [O] string      email[]
    [O] string      online[]
    [O] Extension   extensions[]
```

Members:

- `address`: A structure for various fields holding portions of the postal address.
- `telephone`: An array of telephone numbers for the End-User.
- `email`: An array of email addresses for the End-User.
- `online`: An array of URIs for the End-User (usually web-sites).
- `extensions`: A mechanism implementations MAY choose to use for extending this structure provided those extensions come from a different namespace.

11.36.4 Address Type

The `Address` structure carries the detailed fields describing a particular address.

```
Address
    [O] string      name
    [O] string      street[]
    [O] string      city
    [O] string      stateprov
    [O] string      country
    [O] string      org
    [O] Extension   extensions[]
```

Members:

- `name`: The name to which items should be addressed.
- `street`: The street portion of the addess. This may involve multiple lines of an address.
- `city`: The city portion of the address.
- `stateprov`: The state or province portion of the address.
- `country`: The country portion of the address.
- `org`: Any organization needing to be specified in the address.
- `extensions`: A mechanism implementations MAY choose to use for extending this structure provided those extensions come from a different namespace.

12 Producer Roles

Administrator	This role typically grants the highest level of access to the functionality of an entity.
User	This role is typically associated with End-Users who may personalize some set of properties for an entity.
Guest	This role is typically associated with End-Users who may view an entity on a page but not modify any of its properties or settings.

13 Constants

Type	Value	Description
Mode	view	Entity is expected to render markup reflecting its current state.
Mode	edit	Entity is expected to render markup useful for End-User personalization.
Mode	help	Entity is expected to render markup useful for helping an End-User understand the entity's operation.
Mode	preview	Entity is expected to render markup representative of its configuration, as this might be useful to someone testing a page layout.
Window state	normal	The entity is sharing space with other entities and should restrict its consumption of space accordingly.
Window state	minimized	The entity, though still aggregated on the page, is expected to restrict its consumption of space to a bare minimum.
Window state	maximized	The entity is being offered significantly more that the normal share of the space available to entities on the Consumer's aggregated page.
Window state	solo	The entity is the only entity being rendered on the Consumer's aggregated page.

14 Fault Messages

In addition to generic fault messages that may be generated by the web service stacks of the Consumer and/or Producer, a variety of messages specific to this protocol are defined. WSDL defines fault codes to be strings using "." as a delimiter to scope the error codes. The following are defined for constructing a WSRP/WSIA error code within the same namespace as used for the rest of the types defined by this specification:

Top Level	Specific Code	Description
Security	AccessDenied	Policy has denied access either to the End-User, the asserted `producerRole` or the consumer's registration.
Security	InvalidProducerRole	The specified `producerRole` is not supported.
Security	InconsistentParameters	Used when a Consumer supplies an `entityHandle` that is not scoped by the supplied `registrationHandle`.
Security	InvalidRegistration	Used when a Consumer supplies a `registrationHandle/registrationState` pair that are not recognized by the Producer.
Security	AuthenticationFailure	The credentials supplied were not able to be authenticated by the Producer.
Interface	MissingParameters	Used when required parameters are missing.
Interface	OperationFailed	Normal execution of the operation failed. Check the detailed message for reasons why.
Interface	InvalidHandle	Used when the Consumer supplies an invalid `entityHandle`.
Interface	EntityStateChangeRequired	Used when an entity needs to modify its persistent state, but has been prevented from doing so.
Interface	InvalidCookie	Used only when the environment at the Producer has timed out AND the Producer needs the Consumer to invoke **initCookie()** again.
Interface	UnsupportedMode	The entity does not support generating markup for the requested mode.
Interface	UnsupportedWindowState	The entity does not support generating markup for the requested window state.
Interface	UnsupportedLocale	The entity does not support generating markup for the requested `locale`.
Interface	UnsupportedMarkupType	The entity does not support generating markup for the requested `markupType`.
Interface	NoCloneGenerated	The invocation of **cloneEntity()** did not produce a new entity.

Interface	DestroyEntitiesFailed	

15 WSDL Interface Definition

The WSDL that MUST be referenced by Producers implementing this specification are located at:

http://www.oasis-open.org/committees/wsrp/wsdl/v1/WSRP-v1-Interfaces.wsdl - Contains the messages and portType definitions for this specification.

http://www.oasis-open.org/committees/wsrp/wsdl/v1/WSRP-v1-Bindings.wsdl - Contains the standard binding definitions for this specification.

This WSDL defines the following portTypes:

1. WSRP_v1_Markup_PortType: All Producers MUST expose this portType.
2. WSRP_v1_ServiceDescription_PortType: All Producers MUST expose this portType.
3. WSRP_v1_Registration_PortType: Only Producers supporting in-band registration of Consumers need expose this portType.
4. WSRP_v1_EntityManagement_PortType: Producers supporting the entity management interface expose this portType. If this portType is not exposed, the entities of the service are not configurable by Consumers.

16 References

16.1 Normative

[Character Sets] *http://www.iana.org/assignments/character-sets*

[Namespaces] *http://www.w3.org/TR/REC-xml-names/*

[RFC2119] *S. Bradner*, Key words for use in RFCs to Indicate Requirement Levels, *http://www.ietf.org/rfc/rfc2119.txt*, *IETF RFC 2119, March 1997.*

[Schema] *http://www.w3.org/TR/xmlschema-0/*

[SOAP] *http://www.w3.org/TR/SOAP/*

[SSL/TLS] *http://www.ietf.org/html.charters/tls-charter.html*

[URI/URL] *http://www.ietf.org/rfc/rfc2396.txt*

[WSDL] *http://www.w3.org/TR/wsdl*

16.2 Non-Normative

[Boyer-Moore] *http://www.cs.utexas.edu/users/moore/best-ideas/string-searching/*

[DIME] *http://www.ietf.org/internet-drafts/draft-nielsen-dime-02.txt*

[J2EE] *http://java.sun.com/j2ee/*

[JSR168] *http://www.jcp.org/jsr/detail/168.jsp*

[.Net] *http://www.microsoft.com/net/*

[P3P] *http://www.w3.org/TR/P3P/*

[Requirements] *http://www.oasis-open.org/committees/wsia/documents/Requirements2002-09-17.html*

[RLTC] *http://www.oasis-open.org/committees/rights/*

[SAML] *https://www.oasis-open.org/committees/security/*

[UDDI] *http://www.uddi.org/specification.html*

[WS-Attachments] *http://www-106.ibm.com/developerworks/webservices/library/ws-attach.html*

[WS-I.org] *http://www.ws-i.org/*

[WSIL] *http://www-106.ibm.com/developerworks/webservices/library/ws-wsilspec.html*

[WSRP Whitepaper] *Thomas Schaeck*, Web Services for Remote Portlets (WSRP) Whitepaper, *http://www.oasis-open.org/committees/wsrp/documents/wsrp_wp_09_22_2002.pdf*, *22 September, 2002.*

[WS-Security] *http://www.oasis-open.org/committees/wss/*

[XACML] *https://www.oasis-open.org/committees/xacml/*

[XCBF] *http://www.oasis-open.org/committees/xcbf/*

[XForms] *http://www.w3.org/TR/xforms/*

[XML Digital Signatures] *http://www.w3.org/Signature/*

[XML Encryption] *http://www.w3.org/TR/xmlenc-core/*

Appendix A. Glossary

Action	A term often used elsewhere for what this specification calls “Interaction”.
Administrator	A person who installs or maintains a system (for example, a SAML-based security system) or who uses it to manage system entities, users, and/or content (as opposed to application purposes; see also End User). An administrator is typically affiliated with a particular administrative domain and may be affiliated with more than one administrative domain.
Attribute Also see “Service Attribute”	A distinct characteristic of an object. An object’s attributes are said to describe the object. Objects’ attributes are often specified in terms of their physical traits, such as size, shape, weight, and color, etc., for real-world objects. Objects in cyberspace might have attributes describing size, type of encoding, network address, etc. Salient attributes of an object is decided by the beholder.
Authentication	To confirm a system entity’s asserted principal identity with a specified, or understood, level of confidence.
Authorization	The process of determining, by evaluating applicable access control information, whether a subject is allowed to have the specified types of access to a particular resource. Usually, authorization is in the context of authentication. Once a subject is authenticated, it may be authorized to perform different types of access.
Browser	A system entity that is used by an end user to access a Web site. A browser provides a run-time environment for distributed application components on the client’s device.
Client	a system entity (not a business entity) that accesses a Web service. Contrast with Browser and Customer.
Consumer	A business entity that accesses a Web service or a Web site. Contrast with End user and Customer A business entity creating Consumer Applications
Consumer Application	A web application that uses one or more WSIA/WSRP Web Services
Credential	Data that is transferred to establish a claimed principal identity. [4]
Customer	A business entity that purchases goods or services
End-User	1. A natural person who makes use of resources for application purposes (as opposed to system management purposes; see Administrator, User). [4] 2. A person who uses a device specific Browser to access a Web site
Event	A notification that some state in the system (that you are interested in) has changed

Fragment	A piece of markup that is not part of a full document - part of aggregate - not binary, but not necessarily XML - generally a markup language - can aggregate a bunch of fragments
Identity	The unique identifier for a person, organization, resource, or service.
Sign-On	The process whereby a user presents credentials to an authentication authority, establishes a simple session, and optionally establishes a rich session.
Sign-Off	The process of presenting credentials to an authentication authority, establishing a simple session, and optionally establishing a rich session.
Party	Refers to any person who interacts with the system and/or the network the system is managing.
Portal Page	Complete document rendered by a portal
Portlet	Component that generates fragment
Producer	A business entity that hosts a Web service or a Web site One or more WSIA/WSRP web services A business entity creating, publishing and supporting WSIA/WSRP Web Services
Role	The combination of access rights available to a particular actor.
Session	A lasting interaction between system entities, often involving a user, typified by the maintenance of some state of the interaction for the duration of the interaction.
Site	An informal term for an administrative domain in geographical or DNS name sense. It may refer to a particular geographical or topological portion of an administrative domain, or it may encompass multiple administrative domains, as may be the case at an ASP site. one portal-specific example of an administrative domain, user group, etc.
System / System Entity	An active element of a computer/network system. For example, an automated process or set of processes, a subsystem, a person or group of persons that incorporates a distinct set of functionality.
Time-Out	A period of time after which some condition becomes true if some event has not occurred. For example, a session that is terminated because its state has been inactive for a specified period of time is said to “time out”.
Uniform Resource Locator (URL)	Defined as “a compact string representation for a resource available via the Internet.” URLs are a subset of URI.
User	A natural person who makes use of a system and its resources for

	any purpose. See also administrator, end user. A natural person who makes use of a system and its resources for any purpose. See also end user.
Username/User Identity	The unique identity for a user with a system
Web Service	A Web Service is a software component that is described via WSDL and is capable of being accessed via standard network protocols such as but not limited to SOAP over HTTP.
WSIA Web Service	A SOAP-compliant Web Service that adheres to one of more WSIA interfaces.
Web Site	A hosted application that can be accessed by an End user using a browser
Window States	Max, min, normal, detached
WSIA Interface	A programmatic interface defined by the WSIA committee to support the creation of Web Services that encapsulate and integrate user-facing interactive applications.
WSRP Service	Presentation oriented, interactive web services that can be aggregated by consuming applications - WSRP services can be published, found, and bound in a standard manner, describing themselves with standardized metadata
XML (Extensible Markup Language)	Extensible Markup Language, abbreviated XML, describes a class of data objects called XML documents and partially describes the behavior of computer programs which process them. XML is an application profile or restricted form of SGML, the Standard Generalized Markup Language [ISO 8879]
XML Namespace	A collection of names, identified by a URI reference, which are used in XML documents as element types and attribute names. An XML namespace is often associated with an XML schema. For example, SAML defines two schemas, and each has a unique XML namespace.

Appendix B. Acknowledgments

The following individuals were members of the WSIA committee during the development of this specification:

- Sasha Aickin, Plumtree
- Patel Ashish, France Telecom
- Stefan Beck, SAP
- Dan Bongard, Kinzan
- Kevin Brinkley, Intel
- Jeffery C. Broberg, Novell
- Rex Brooks, Individual
- Tyson Chihaya, Netegrity
- Carlos Chue, Kinzan
- Terry Cline, Peregrine Systems
- William Cox, BEA
- Suresh Damodaran, Sterling Commerce
- Alan Davies, SeeBeyond
- Jacques Durand, Fujitsu
- John Evdemon, Vitria
- Sean Fitts, CrossWeave
- Greg Giles, Cisco
- Dan Gisolfi, IBM
- Timothy N. Jones, CrossWeave
- Aditi Karandikar, France Telecom
- John Kelley, Individual
- John Kneiling, Individual
- Ravi Konuru, IBM
- Alan Kropp, Epicentric
- Michael Mahan, Nokia
- Monica Martin, Drake Certivo
- Dale Moberg, Cyclone Commerce
- Dean Moses, Epicentric
- Peter Quintas, Divine
- T.V. Raman, IBM
- Shankar Ramaswamy, IBM
- Eilon Reshef, WebCollage
- Graeme Riddell, Bowstreet
- Don Robertson, Documentum
- Royston Sellman, HP
- Sim Simeonov, Macromedia
- Davanum Srinivas, Computer Associates
- Sandra Swearingen, DoD
- Rich Thompson, IBM
- Srinivas Vadhri, Commerce One
- Vinod Viswanathan, Pointgain Corp.
- Charles Wiecha, IBM (chair)
- Chad Williams, Epicentric
- Kirk Wilson, Computer Associates
- Garland Wong, Kinzan

The following individuals were members of the WSRP committee during the development of this specification:

- Alejandro Abdelnur, Sun
- Olin Atkinson, Novell
- Sasha Aickin, Plumtree
- Jeff Broberg, Novell
- Chris Brown, Novell
- Mark Cassidy, Netegrity
- Richard Cieply, IBM
- Dave Clegg, Sybase
- Ugo Corda, SeeBeyond
- William Cox, BEA
- Michael C. Daconta, McDonald Bradley
- Ron Daniel Jr., Interwoven
- Brian Dirking, Stellent
- Angel Luis Diaz, IBM
- Gino Filicetti, Bowstreet
- Adrian Fletcher, BEA
- Michael Freedman, Oracle
- Tim Granshaw, SAP Portals
- Mike Hillerman, Peoplesoft
- Scott Huddleston, Divine
- Timothy N. Jones, CrossWeave
- Andre Kramer, Citrix
- Andreas Kuehne
- Aditi Karandika, France Telecom
- Alan Kropp, Epicentric
- Jon Klein, Reed-Elsivier
- Andreas Kuehne
- Carsten Leue, IBM
- Susan Levine, Peoplesoft
- Eric van Lydegraf, Kinzan
- Khurram Mahmood, Peoplesoft
- Lothar Merk, IBM
- Madoka Mitsuoka, Fujitsu
- Takao Mohri, Fujitsu
- Adam Nolen, Reed-Elsivier
- Petr Palas, Moravia IT
- Gregory Pavlik, HP
- Peter J Quintas, Divine
- Nigel Ratcliffe, Factiva
- Eilon Reshef, WebCollage
- Mark Rosenberg, Tibco
- Joe Rudnicki, U.S. Department Of the Navy
- Thomas Schaeck, IBM (chair)
- Robert Serr, Divine
- Davanum Srinivas, Computer Associates
- Andrew Sweet, Perficient
- David Taieb, IBM
- Yossi Tamari, SAP Portals
- Rich Thompson, IBM
- Srinivas Vadhri, CommerceOne
- Stephen A. White, SeeBeyond
- Charles Wiecha, IBM

Appendix C. Revision History

Rev	Date	By Whom	What
0.1	6/03/2002	Rich Thompson	Initial Draft
0.1.1	6/04/2002	Carsten Leue	Worked in some additional WSRP requirements
	6/05/2002	Rich Thompson	Added exemplary section to overview
	6/06/2002	Carsten Leue	Added request data to getFragment and invokeAction
0.1.2	6/06/2002	Rich Thompson	Added cloneEntities() & descriptive text
0.2	7/09/2002	Alan Kropp, Rich Thompson	Modified as per face-2-face discussions
0.21	7/10/2002	Rich Thompson	Refactored data objects
0.22	7/19/2002	Rich Thompson	Reflect discussion on email list
0.23	7/25/2002	Carsten Leue Rich Thompson	Added WSDL and included some explanations Reformat style Reflect discussion
0.3	8/01/2002	Rich Thompson	Migrate to OASIS spec template Reflect email list and concall discussions
0.31	8/08/2002	Rich Thompson Alan Kropp Chris Braun	Reflect discussion Fill out more of spec template Markup section Environment initialization section
0.32	8/10/2002	Rich Thompson Carsten Leue Chris Braun	Incorporated misc. comments/discussion Introduction section and explanation of sections Updated Markup section
0.4	8/16/2002	Rich Thompson Thomas Schaeck Alan Kropp	Rewrote Markup section, reflect discussion WSRP Use cases Cross references to requirements
0.5	8/30/2002	Rich Thompson Carsten Leue Mark Cassidy	Incorporated misc. comments/discussion UDDI, Additional data structure factoring Updated Security section
0.7	9/27/2002	Rich Thompson Carsten Leue Alan Kropp Charlie Wiecha	Reflect Sept. F2F discussion/decisions Rewrite Intro Propose caching support Propose Properties support
0.8	10/22/2002	Rich Thompson Carsten Leue Alan Kropp	Reflect decisions from weekly TC calls
0.85	10/22/2002	Rich Thompson Alan Kropp	Reflect decisions from Nov. F2F and TC calls

ToDo:

1. Section 9.2.1 – "[3rd F2F: Change token to wsrp-rewrite for now. Revisit value of less human readable token once an implementation is available to test the impact.]"
2. Revisit Glossary definitions

Appendix D. Notices

OASIS takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on OASIS's procedures with respect to rights in OASIS specifications can be found at the OASIS website. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementors or users of this specification, can be obtained from the OASIS Executive Director.

OASIS invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights which may cover technology that may be required to implement this specification. Please address the information to the OASIS Executive Director.