WRSP Primer

Working Draft 0.3, 16 December 2002

Document identifier:

WSRP_Primer_0.3 (Word)

Location:

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

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

Editors:

Gil Tayar, WebCollage <gil.tayar@webcollage.com>

Abstract:

This document's purpose is threefold:

- Approach the spec from a more tutorial point of view by giving examples of all SOAP messages and by giving a step by step understanding of a Consumer and Producer. For conciseness sake, only the body of the SOAP message is given. Also, the data in the XML which is part of the example is in italics, while the information that is required and must be a part of a Producer or Consumer that implements the scenario is in a regular style.
- Describe Consumer and Producer scenarios.
- Describe what the Consumer and Producer are required to do in order to implement a successful WSRP implementation.

Comment: Woefully inadequate.

Status:

This draft is an early version. Various concepts continue to be debated. Points needing clarification as this evolves into the final specification are much appreciated and may be emailed to Gil Tayar.

If you are on the wsia-wsrp@lists.oasis-open.org, 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/requirements_v1_errata.html.

Table Of Contents

Comment: Woefully inadequate.

1 Introduction

This document's purpose is threefold:

- Approach the spec from a more tutorial point of view by giving examples of all SOAP messages and by giving a step by step understanding of a Consumer and Producer. For conciseness sake, only the body of the SOAP message is given. Also, the data in the XML which is part of the example is in italics, while the information that is required and must be a part of a Producer or Consumer that implements the scenario is in a regular style.
- Describe Consumer and Producer scenarios.
- Describe what the Consumer and Producer are required to do in order to implement a successful WSRP implementation.

The document consists mainly of a list of scenarios. Each scenario is described, and its sub-sections describe what the Producer or Consumer need (or can) do to implement the scenario. If a sentence or paragraph are a *requirement* from the spec, the requirement is highlighted in this format **[requirement]**. [I will cross-reference the requirements to the spec when the spec is a bit more stabilized].

This document also includes a section that is an introduction to WSRP. This section assumes no knowledge of WSRP and introduces the ideas and "actors" that govern the scenarios and, ultimately, the spec, and introduces the reader to the main concepts behind WSRP. Some of the other concepts are introduced when introducing and explaining the scenarios.

The scenarios themselves are not meant to be full, but rather to be modular scenarios which real implementer can mix and match to create their own scenarios. Because of their modularity they tend to be minimal.

Most scenarios are based on two basic scenarios – the Minimal Producer scenario and the Minimal Consumer scenario, which enables their description to include only the *changes*.

1.1 Introduction to WSRP

Web Services for Remote Portlets (WSRP) is a specification, based on SOAP, which defines SOAP operations that enable a Web Service to return an HTML fragment that can be embedded in an HTML page.

Moreover, an End User that navigates to the HTML page can click on a link (that was included in the Web Service's HTML fragment) that will navigate the user to another HTML page, where the end user will see another HTML fragment the Web Service returned.

Thus the user's perception is of a small Web application embedded inside another HTML application. As HTML does not support this kind of functionality, a standards body rose to define the protocol between the "container" HTML page and the Web Service that returns the HTML fragment, a protocol that enables this supposed embedding.

We have seen three "actors" in this scenario.

- **Producer**: The Web Service that return the HTML fragments and acts as a "mini" application.
- **Consumer**: The HTML page (or application) that embeds the Producer "mini" application.
- **End user**: The end user, who via the browser, sees the two combined applications.

There is also another term to discuss, which is:

- **Portlet Entity**: a Producer Web Service can implement multiple **Portlets** via one Web Service. For example, a "stock quote" Portlet and a "weather" Portlet. Each Portlet is exposed via one or more Portlet Entities, which are customizations of the basic Portlet, and which the Producer provides – these are named **Producer Offered Portlet Entities**. Usually, for each Portlet, there is only one Portlet Entity that the Producer offers. The Consumer can create more customizations of these Portlet Entities to create **Consumer Configured Portlet Entities**.

To differentiate between the two types of entities, the text will use the word "**Portlet**" for Producer Offered Entities, and the word "**Portlet Entity**" exactly like it is used in the specification - for Portlet Entities that are both Producer Offered an Consumer Configured.

Note that the above discusses browsers, HTML, and Web applications. WSRP is intended to be more specific than that, and also enables things like embedding multiple WML/WAP applications into cell phones, or embedding Voice applications (using VoiceML) inside normal Web applications.

That is why the specification speaks of "markup" and not HTML, and discusses the "user agent" and not just a "browser". To simplify things, this tutorial will assume the simplest case – the markup is a plain HTML fragment, the "user agent" is a browser, and the Consumer is showing a typical web application to the end user.

WSRP is also more than that. It's mission is to be the protocol that binds portal servers and their portlets. A portlet, in essence, is a mini-application embedded inside another application, something which, as we have discussed, is what WSRP is all about.

But a portlet inside a portal needs more than an ability to embed itself inside a portal. It also wants to enable the portal or the end user to configure the portal. For example, our stock quote portlet would like to be configured to display on certain stocks. In WSRP parlance, this means that it wants to enter "**edit mode**" where it can display the configuration UI which configures itself. Or maybe it would like the portal to display its own UI which enables the list of "**properties**" to be configured.

Likewise, a portal would like to display the portlet HTML in certain **Window states**: "minimized", "maximized", etc…

[Discuss roles (if they survive!) …]

And finally, the Producer would like all portal applications to register themselves, and the Consumer would like to programmatically understand what entities this Producer is offering, and what customization properties they have.

To this end, the WSRP protocol is divided into four distinct "interfaces":

- **Markup**: an interface which includes operations that enable the embedding of the Producer HTML inside the Consumer, enables the interaction of the End user with the Producer HTML, while still staying embedded in the Consumer application, and includes support for modes and window states. This interface must be implemented by the Producer.
- **Service Description:** an interface which includes an operation that enable the Consumer to query the Producer about its entities. This interface must be implemented by the Producer.
- **Portlet Entity Management**: an interface which includes operations that enable the Consumer to customize entities, and even create entities of their own, entities named **Consumer Configured Entities** in the specification. This interface does not have to be implemented by the Producer.
- **Registration**: an interface which includes operations that enable the Consumer to register itself with the Producer. This interface does not have to be implemented by the Producer.

The following sections will briefly discuss the four interfaces:

1.1.1 The Markup Interface

This interface is the most important interface, but unfortunately is the most difficult to understand as it includes operations, which, if used in *a certain way*, will give the end user a perception of a portlet embedded within another application.

The difficulty with understanding this interface is not only in understanding the operations, but more in understanding how to choreograph between themselves, and between the end user interactions.

But before tackling the choreography, let's tackle the operations themselves:

- **`getMarkup:`** this operation is invoked by the Consumer in order to get the "current" HTML (we will discuss what "current" means momentarily), which will be embedded in the Consumer page.
- **`performInteraction/performBlockingInteraction:`** these operation is invoked by the Consumer after the end user interacted with the Producer HTML in the combined Consumer page (e.g. clicked on a link, or submitted a form in the Producer HTML). How interactions that originated from Producer HTML arrive at the Consumer application is a question dealt with in End User Interactions.
- **`initCookie`**: this minor operation may be invoked by the Consumer. This operation helps Producers which use HTTP cookies in their operations, but this Primer will only cursively discuss this operation and its uses, and does not use its capabilities.

1.1.1.1 End User Interactions

In the previous section, we understood that `performInteraction` and `performBlockingInteraction` are invoked by the Consumer whenever the end user interacts with the Producer HTML in the combined Consumer page, where interaction in HTML is a click on a link, or a form submit.

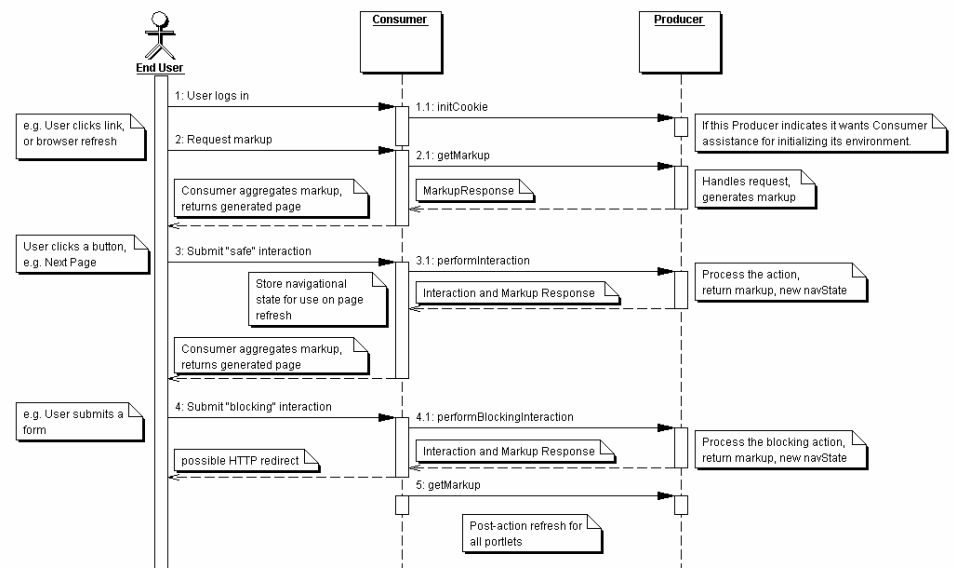
This means that all links and form submissions in the Producer HTML fragment point to the Consumer application. The specification and this tutorial refer to these links (links in the Producer which point to the Consumer in order to perform an interaction) as **interaction URLs**.

The Producer usually passes information to the Consumer in the interaction URLs. This information is referred to in the specification and the tutorial as **interaction parameters**. These parameters indicate to the Consumer information that needs to be passed in the subsequent `performInteraction` and `performBlockingInteraction` (and likewise indicates which of these two operations to call, or even whether to call these operation and not just call the next page's `getMarkup`).

The Producer knows what the interaction URL is, and where to insert the interaction parameters using two methods:

- **Producer URL-writing**: In this method, the Consumer passes a template of the interaction URL to the Producer. The producer uses placeholders in the template to insert the interaction parameters and inserts the resultant URLs in the HTML it returns.
- **Consumer URL-writing**: In this method, the Producer inserts the interaction parameters in the HTML between two placeholders. The Consumer must then search for these placeholders, read the interaction parameters, and replace this with a correct interaction URL.

A typical flow between our three actors, would essentially look like this:



[need to simplify this picture]

One of the most important interaction parameters is `navigationalState`. This parameter is important because it is the equivalent to a URL in a normal Web application. Just as giving a browser a URL will return HTML that this URL references, thus giving `getMarkup` a `navigationalState` will return an HTML fragment that this `navigationalState` references. And just as a link makes the browser change it's current URL and request HTML from the new URL, thus a link which passes the `navigationalState` interaction parameter to the Consumer makes the Consumer change the `navigationalState` for the portlet, and request HTML for the new `navigationalState` using `getMarkup`.

1.1.1.2 Sessions

And just like WSRP has the equivalent of the URL, so WSRP has the equivalent of Web **sessions**. In the Markup interface operations, all operations can return a new session. This session should be preserved by the Consumer and sent to all invocations of the Markup interface operations, usually for the duration of the end users session.

1.1.2 The Service Description Interface

The Service description interface includes just one simple operation:

- `getServiceDescription`: this operation returns a description of the Producer, and the list of entities this Producer supports. This information is also called the Producer meta-data and the Portlet meta-data. For example, it returns information about whether `initCookie` needs to be called for this Producer, or which modes a Portlet supports.

1.1.3 The Portlet Entity Management Interface

The Portlet Entity Management Interface enables the Consumer to customize a Portlet Entity. Customization is done by setting Portlet Entity **Properties**. This interface also enables the Consumer to create more entities by **cloning** the ones defined by the Producer (named **Producer Offered Entities**) to create **Consumer Configured Entities**.

The interface includes the following operations:

- **`getPortletEntityPropertyDescription`**: returns the list of properties available for a Portlet Entity, including their names, types, and human readable titles.
- **`setPortletEntityProperties`**: sets the property values of entities, in effect customizing them.
- **`getPortletEntityProperties`**: gets the property values of entities.
- **`clonePortletEntity`**: clones an entity to create a new Consumer Configured Portlet Entity.
- **`destroyPortletEntities`**: destroys a previously created Consumer Configured Portlet Entity.
- **`getPortletEntityDescription`**: returns a description of a whole Portlet Entity.

1.1.4 The Registration Interface

The Registration Interface enables the Consumer to register itself at the Producer, and to receive a registration handle which must be used in all subsequent operations. Note that there will be Producers that require registration, Producers that make registration optional, and Producers that do not have the registration interface.

The interface includes the following operations:

The interface includes the following operations:

- **`register`**: register the Consumer at the Producer. The Consumer passes information about itself, and is returned a `registrationContext` which should be used in subsequent operations.
- **`deregister`**: the inverse operation, which enables the Consumer to end its relationship with the Producer.
- **`modifyRegistration`**: enables the Consumer to notify the Producer of changes in the data the Consumer sent in the previous `register` operation.

1.2 Conventions used in this text

[Discuss white space in the examples]

[Discuss style conventions and what they mean]

---

2 Minimal Producer

In this scenario, the Producer consists of one Portlet, which shows just one single-HTML-page Producer with no links, in the locale `en`. This is practically the smallest Producer one can generate and which conforms with all the requirements from the specification.

2.1 Implementation Summary

In order to successfully implement WSRP, the Producer exposes a SOAP endpoint which implements certain operations. It implement `getServiceDescription` to enable a Consumer to query information about which Portlets it has and about the meta-data, and it implements the `getMarkup` operation that returns the HTML.

> in the example, the Portlet's handle is "`theOnlyPortlet`". This simple portlet return a "Hello, World" HTML.

The Producer implements the following operations **[the Producer MUST implement them]**:

- `getServiceDescription`: enables a Consumer to query information about w hich Portlets the Producer has and about the meta-data of the Portlets.
- `getMarkup`: returns the HTML with the "Hello, world".
- `performInteraction`: because the HTML returned by `getMarkup` contains no links, the Consumer should never invoke this method. Thus. the implementation of this operation can be an empty implementation which fails.
- `performBlockingInteraction`: because the HTML returned by `getMarkup` contains no links, the Consumer should never invoke this method. Thus. the implementation of this operation can be an empty implementation which fails.
- `initCookie:` the implementation can be an empty implementation which returns "void", as the service description returns false in the field `requiresInitCookie`.

2.2 `GetServiceDescription` Logic

The Producer ignores `desiredLocales`, `sendAllLocales`, fields which enable the Producer to return the information in multiple locales.

The Producer also ignores `registrationContext`, which enables the Producer to authenticate the Consumer, or maybe return a different list of Portlets, depending on who is requesting the information.

> For example, the Producer returns the following XML:

```
<getServiceDescriptionResponse
  xmlns="http://www.oasis-open.org/committees/wsrp/v1/wsdl/types">
  <offeredEntities> [while not required, is essential for the Consumer
to send meta-data to consumer]
     <portletEntityHandle>theOnlyPortlet</portletEntityHandle>
[required]
     <markupTypes> [required]
        <markupType>text/html</markupType> [required]
        <locales>en</locales> [required]
        <modes>view</modes> [required]
        <windowStates>normal</windowStates> [required]
     </markupTypes>
  </offeredEntities>
  <requiredRegistration>false</requiredRegistration>
</getServiceDescriptionResponse>
```

2.3 `getMarkup` Logic

The Producer ignores all the parameters sent by the Consumer, for the following reasons:

- `registrationContext`: no registration needed.

- `portletEntityContext`, including:
  - `portletEntityHandle`: The producer only supports one Portlet, and assumes the Consumer sent the correct handle **[it is not a requirement for the Producer to check this]**.
  - `portletEntityState`: there is no persistent state for this one Portlet. (this Primer will not discuss this field. See the specification for more information)
- `runtimeContext`, including:
  - `portletEntityInstanceID`: The producer does not need a unique ID. (this Primer will not discuss this field. See the specification for more information)
  - `sessionHandle`: the Producer does not need session support.
  - `userContext`: the Producer does not deal with users.
- `markupParams`, including:
  - `markupCharacterSet`: the Producer returns the allowed UTF-8 character set. A minimal Producer should always return UTF-8, as all Consumers must support this character set. **[which is a requirement for the Consumer]**.
  - `mode`: the Producer only supported mode is "view", and assumes that the Consumer sent that mode **[which is a requirement for the Consumer]**.
  - `windowState`: the Producer only supported windows state is "normal", and assumes that the Consumer sent that window state **[which is a requirement for the Consumer]**.
  - `navigationalState`: Because the Producer has only one page, there is no meaning to `navigationalState`.

For example, the Producer returns the following XML:

```
<getMarkupResponse
  xmlns="http://www.oasis-open.org/committees/wsrp/v1/wsdl/types">
  <markupContext>
     <markup>
        <![CDATA[
           <div class="portlet-font"><p>Hello, world!</p></div>
        ]]>
     </markup>
     <locale>en</locale>
     <markupType>text/html</markupType>
  </markupContext>
</getMarkupResponse>
```

Note the use of the class "`portlet-font`". This class enables the portlet to conform to the Consumer's look and feel.

3 Producer Portlet with More Than One Page – Producer URL Writing

This scenario is based on the Minimal Producer scenario, and enhances it by making the Portlet have two pages with links between one another. The Producer chooses to use Producer URL when writing its Portlet.

3.1 Implementation Summary

The Producer implements the same operations as the base scenario (i.e. only `getServiceDescription` and `getMarkup`).

This time, though, the `getMarkup` can return one of two pages. The Producer knows which HTML to return based on the `navigationalState` sent to it by the Consumer, `navigationsalState` which in turn is sent to the Consumer via interaction parameters.

> In the example, the Producer decides that the `navigationalState` for the first page is simply the string "1" and for the second page it is the string "2". This example also assumes the Consumer URL templates are the ones in templates in the Minimal Consumer scenario.

The Producer does not need `performInteraction` because in the interaction URLs, it directs the Consumer to directly invoke `getMarkup` in the next page and to bypass `performInteraction/performBlockingInteraction`.

This scenario implements interaction URLs using Producer URL writing.

3.2 `getServiceDescription` Logic

Exactly like the base scenario, except that the Portlet's `doesUrlTemplateProcessing` needs to be `true`, as it uses Producer URL writing.

> For example, the Producer returns the following XML:

```
<getServiceDescriptionResponse
  xmlns="http://www.oasis-open.org/committees/wsrp/v1/wsdl/types">
  <offeredEntities>
     <portletEntityHandle>theOnlyPortlet</portletEntityHandle>
     <markupTypes>
        <markupType>text/html</markupType>
        <locales>en</locales>
        <modes>view</modes>
        <windowStates>normal</windowStates>
     </markupTypes>
     <doesUrlTemplateProcessing>true</doesUrlTemplateProcessing>
  </offeredEntities>
</getServiceDescriptionResponse>
```

3.3 `getMarkup` Logic

As `getMarkup` now needs to return two pages, it needs to receive this information. The Producer sends this information to itself in the `navigationalState` using the interaction parameter `wsrp-navigationalState`.

The Producer ignores the same parameters as in the base scenario, except for `navigationalState`. Based on the `navigationalState` it will know which page to display.

> For example, the Producer decides that the `navigationalState` for the first page is simply the string "1" and for the second page it is the string "2".
>
> If the `navigationalState` sent by the Consumer is "1", the following XML will be returned:

```
<getMarkupResponse
  xmlns="http://www.oasis-open.org/committees/wsrp/v1/wsdl/types">
  <markupContext>
     <markup>
```

```
         <![CDATA[
            <div class="portlet-font">
               <p>Hello, world! This is the first page!</p>
               <a href="
http://consumer.com/containerpage?ut=Render&ns=2&m=view&ws=normal&res="
>
                  click here for the second page
               </a>
            </div>
         ]]>
      </markup>
      <locale>en</locale>
      <markupType>text/html</markupType>
   </markupContext>
</getMarkupResponse>
```

Note that the link for the second page uses the URL templates given by the Consumer (and assumed to be the Consumer URL templates are the ones in templates in the Minimal Consumer scenario).

If the `navigationalState` sent by the Consumer is "2", the following XML will be returned:

```
<getMarkupResponse
  xmlns="http://www.oasis-open.org/committees/wsrp/v1/wsdl/types">
  <markupContext>
      <markup>
         <![CDATA[
            <div class="portlet-font">
               <p>Hello, world! This is the second page page!</p>
               <a href="
http://consumer.com/containerpage?ut=Render&ns=1&m=view&ws=normal&res="
>
                  click here for the first page
               </a>
            </div>
         ]]>
      </markup>
      <locale>en</locale>
      <markupType>text/html</markupType>
   </markupContext>
</getMarkupResponse>
```

4 Producer Portlet with More Than One Page – Consumer URL Writing

This scenario is based on the Minimal Producer scenario, and enhances it by making the Portlet have two pages with links between one another. The Producer chooses to use Consumer URL when writing its Portlet.

4.1 Implementation Summary

The Producer implements the same operations as the base scenario (i.e. only `getServiceDescription` and `getMarkup`).

This time, though, the `getMarkup` can return one of two pages. The Producer knows which HTML to return based on the `navigationalState` sent to it by the Consumer, `navigationsalState` which in turn is sent to the Consumer via interaction parameters.

In the example, the Producer decides that the `navigationalState` for the first page is simply the string “1” and for the second page it is the string “2”.

The Producer does not need `performInteraction` because in the interaction URLs, it directs the Consumer to directly invoke `getMarkup` in the next page and to bypass `performInteraction/performBlockingInteraction`.

This scenario implements interaction URLs using Consumer URL writing.

4.2 `getServiceDescription` Logic

Exactly like the base scenario.

4.3 `getMarkup` Logic

As `getMarkup` now needs to return two pages, it needs to receive this information. The Producer sends this information to itself in the `navigationalState` using the interaction parameter `wsrp-navigationalState`.

The Producer ignores the same parameters as in the base scenario, except for `navigationalState`. Based on the `navigationalState` it will know which page to display.

For example, the Producer decides that the `navigationalState` for the first page is simply the string “1” and for the second page it is the string “2”.

If the `navigationalState` sent by the Consumer is “1”, the following XML will be returned:

```
<getMarkupResponse
  xmlns="http://www.oasis-open.org/committees/wsrp/v1/wsdl/types">
  <markupContext>
     <markup>
        <![CDATA[
           <div class="portlet-font">
              <p>Hello, world! This is the first page!</p>
              <a href="wsrp-rewrite?Render&wsrp-
navigationalState=2&wsrp-mode=view&wsrp-windowState=normal/wsrp-
rewrite”>
                 click here for the second page
              </a>
           </div>
        ]]>
     </markup>
     <locale>en</locale>
     <markupType>text/html</markupType>
     <requiresUrlRewriting>true</requiresUrlRewriting>
  </markupContext>
</getMarkupResponse>
```

Note that the link for the second page uses the standard Consumer URL writing syntax, and that `requiresUrlRewriting` is set to `true` to indicate that the Consumer needs to rewrite URLs.

If the `navigationalState` sent by the Consumer is “2”, the following XML will be returned:

```
<getMarkupResponse
  xmlns="http://www.oasis-open.org/committees/wsrp/v1/wsdl/types">
  <markupContext>
     <markup>
        <![CDATA[
           <div class="portlet-font">
```

```
                <p>Hello, world! This is the second page page!</p>
                <a href="wsrp-rewrite?Render&wsrp-
navigationalState=1&wsrp-mode=view&wsrp-windowState=normal/wsrp-
rewrite">
                   click here for the first page
                </a>
             </div>
          ]]>
      </markup>
      <locale>en</locale>
      <markupType>text/html</markupType>
      <requiresUrlRewriting>true</requiresUrlRewriting>
   </markupContext>
</getMarkupResponse>
```

5 Minimal Consumer

In this scenario, a Consumer wants to embed a specific Portlet of a specific Producer. The Consumer knows the Producer endpoints. The Consumer wants to embed this entity in locale `en`.

> In the example, the Portlet used is the `theOnePortlet` Portlet from either Producer Portlet with More Than One Page – Producer URL Writing or Producer Portlet with More Than One Page – Consumer URL Writing.

The Consumer does not know anything about this Portlet's metadata, and wants to support it no matter what metadata values the service description or Portlet description have.

5.1 Implementation Summary

As opposed to the Producer, the Consumer does not need to *implement* a Web Service. Rather, it *uses* one, and specifically, a WSRP Web Service. As such, the Consumer does not need to read any WSDL (as the interface to all WSRP Web Services is a common one) – it just needs the URL of the Web Service endpoint.

Once it knows that, it goes through three phases:

- **Producer initialization**: the Consumer invokes `getServiceDescription` to obtain information about the Web Service and about the Portlets it wants to use. If the Web Service `requiresRegistration`, then `register` is also performed by the Consumer. This is a one-shot operation invoked whenever the Consumer wants to initiate a relationship with the Producer.
- **End User initialization**: Some Producers, implementing SOAP over HTTP, use cookies. Moreover, for performance and load balancing reasons, they would like these cookies to be created in the context of the End user session. To this end, WSRP has the `initCookie` operation, which is invoked by the Consumer at the beginning of an End user session.
- **Page composition and interaction**, comprised of:
  - **First page composition**: In the first page, only `getMarkup` of the portlet is called.
  - **Next page composition**: The Consumer received a request for the "next page" via an interaction URL of the Producer, and must invoke one of the interaction operations (or skip it and go directly to `getMarkup`). Of course, any subsequent interaction is also a "Next Page" interaction.

- **Producer termination**: whenever the Consumer wants to terminate its relationship with the Producer, it invokes `deregister`.

The following sections describes the flow of invocations and processing from the point of view of the Consumer. This is the largest scenario of them all, and must be read with care, especially First Page Composition Flow and Next Page Composition Flow , which are difficult as they involve user interaction.

5.2 Producer-Initialization Flow

One time only, whenever the Consumer decides to use the Producer's Portlet, the Consumer invokes the `getServiceDescription` operations to read the following flags:

- `requiresRegistration` & `registrationPropertyDescription`
- `requiresInitCookie`
- `offeredPortletEntity[portletEntityHandle="`*`theOnlyPortlet`*`"]/`
  - `markupTypes[markupType="`*`text/html`*`"]`: to check whether HTML is supported.
  - `markupTypes[markupType="`*`text/html`*`"]/locales[.~="`*`en`*`"]`: to check whether the "en" local is supported.
  - `needSecureCommunication`: to check what type of communication needs to be established, HTTP or HTTPS.
  - `usesMethodGet`: See below.
  - `doesUrlTemplateProcessing`: to check whether Producer needs Consumer URL writing, or whether Consumer URL templates need to be passed to it.

If `requiresRegistration` is true, the Consumer registers at the Producer, using the `register` operation **[MUST]**.

For example, the Consumer sends the following XML:

```
<register
  xmlns="http://www.oasis-open.org/committees/wsrp/v1/wsdl/types">
  <consumerName>aConsumer</consumerName> [required]
  <consumerAgent>homegrownXML.1.0</consumerAgent> [required. Required
format of agent]
</register>
```

If the operation fails, the Consumer ends processing **[MUST]**. Otherwise the Consumer stores the `registrationContext` returned from the operation for later incorporation into the other operations. **[MUST]**

See Producer that Supports Registration for an example of a response to this operation.

If registration was required, the Consumer invokes `getServiceDescription` again (securely if `needSecureCommunication` is "`all`") with the new `registrationContext` to get the description of the service that fits the new `registrationHandle`. This may be a different view of the Web Service.

5.3 End-User-Initialization Flow

If `requiresInitCookie` is "`perUser`" or "`perGroup`", and the Consumer and Producer are communicating via HTTP/HTTPS, the Consumer invokes the `initCookie` operation once for each end user, and stores the returned cookies (returned in the `Set-Cookie` headers) for later incorporation into the other operations from the same end-user **[MUST]**. (see End-User-Initialization Flow in the Consumer with Two Entities From Same Producer scenario to understand the difference between "`perUser`" and "`perGroup`")

For example, the Consumer sends the following XML:

```
<initCookie
  xmlns="http://www.oasis-open.org/committees/wsrp/v1/wsdl/types">
  <registrationContext>
     the context returned from the register operation, or nothing if no
registration
  </registrationContext> [required if requiresRegistration is true[1]]
</initCookie>
```

If the operation fails, end processing **[MUST]**. Otherwise continue as usual.

5.4 First Page Composition Flow

To compose the markup of the first page of the Consumer, the Consumer retrieves the first page's markup using the `getMarkup` operation.

For example, the Consumer sends the following XML:

```
<getMarkup
  xmlns="http://www.oasis-open.org/committees/wsrp/v1/wsdl/types">
  <registrationContext>
     the context returned from the register operation, or nothing if no
registration
  </registrationContext> [required if requiresRegistration is true]
  <portletEntityContext>
     <portletEntityHandle>theOnlyPortlet</portletEntityHandle>
[required]
     <portletEntityState></portletEntityState> [required][2]
  </portletEntityContext>
  <runtimeContext />
  <userContext>
     <userContextID /> [required but can be empty]
  </userContext>
  <markupParams>
     <clientData>
        <userAgent>
           Mozilla/4.5 (Macintosh; U; PPC)
        </userAgent> [required]
     </clientData>
     <secureClientCommunications>
        false [see below]
     </secureClientCommunications> [required]
     <userAuthentication>false</userAuthentication> [required[3]]
     <locale>en</locale> [required]
     <markupCharacterSet>UTF-8</markupCharacterSet> [required]
```

[1] What it no `requiresRegistration` is false? Should the element be empty, or just not be t here?
[2] There is nothing in the spec that says what entity state to send before invocation of an entity management operation.
[3] How does the Consumer know what to put here for the first page?

```
      <markupType>text/html</markupType> [required]
      <mode>view</mode> [required]
      <windowState>normal</windowState> [required]
      <navigationalState>???[4]</navigationalState> [required?[5]]
      <templates>
         [see below]
      </templates> [required if doesUrlTemplateProcessing is true or
namespacing is required]
  </markupParams>
</getMarkup>
```

5.4.1 secureClientCommunications

If `needSecureCommunication` is "`all`", then the Consumer must receive the markup via a secure connection (e.g. use SSL when using HTTP) [6], and if sending it back to the End User, must send it back via a secure connection. Note that to send the markup securely back to the End user, the original request for the Consumer page must have been HTTPS.

5.4.2 templates

If `doesUrlTemplateProcessing` is true, the Consumer supplies templates to enable Producer URL-writing **[MUST]**.

If the Consumer wants to avoid the Producer "impinging" on markup Ids and JavaScript names, it should also send a unique `NameSpacePrefix` [7].

For example, the Consumer sends the following XML:

```
  <templates>
      <DefaultTemplate>
         http://consumer.com/containerpage?ut={urlType}&ns={wsrp-
navigational-state}&m={wsrp-mode}&ws={wsrp-windowState}&res={wsrp-url}[8]
      </DefaultTemplate> [required only if not all the other non-secure
templates are defined]
      <SecureDefaultTemplate>
         https://consumer.com/containerpage?ut={urlType}&ns={wsrp-
navigational-state}&m={wsrp-mode}&ws={wsrp-windowState}&res={wsrp-url}
      </SecureDefaultTemplate> [required only if not all the other
secure templates are defined]
      <NameSpacePrefix>FJH1</NameSpacePrefix>
  </templates>
```

---

[4] What is the information I need to put in for the first page?

[5] Ambiguous requirement: Required by the spec, but not required by the WSDL.

[6] Can a SOAP endpoint have a secure *and* a non-secure endpoint? If not, how do we solve this?

[7] There is no SHOULD or MUST in the spec about this.

[8] The consumer is not really obligated to add the three above parameters to the template. Thus, the consumer is not obligated in the next page to send them to the `getMarkup/perform*Interaction` operations. I think a MUST should be added in the form: "if the Consumer wishes to preserve the flow of the entity application, it MUST use these parameters in the template, and MUST pass those parameters in the next invocation of the `getMarkup/perform*Interaction` operations." If this sentence will *not* be there, then even if the Consumer does all the MUST-s, we won't have a working Producer which embeds its UI flow inside the Consumer! This type of sentence should probably occur in a lot more places.

5.4.3 Processing the `markupResponse`

Processing the markup consists of three phases –

- Processing the session returned by the Producer.
- Doing Consumer URL writing
- Inserting the returned HTML into the Consumer page.

5.4.3.1 Processing the Producer Session

If the `markupResponse` contains a `sessionContext`, then the Consumer stores this information so that later markup interface operations to this Portlet send it. **[Although this is not a MUST, failure to do so may in subsequent operations "likely not generate a markup fragment meeting End User requirements" (section 5.1.1 in the v0.85 spec)[9]]**. In general, the session between a Consumer and a Portlet at the Producer maps to a client session with the Consumer.

5.4.3.2 Consumer URL Writing

The Consumer processes two fields – `markup` and `requiresUrlRewriting`[10] **[If the Consumers wants to use `markup` it MUST NOT ignore `requiresUrlRewriting`]**.

If `requiresUrlRewriting` is true, the Consumer rewrites the Markup according to the algorithm in section 9.2.1 of the v0.85 spec. **[MUST]**

> For example, if the markup included the string *"wsrp-rewrite?Render&wsrp-navigationalState=2&wsrp-mode=view&wsrp-windowState=normal/wsrp-rewrite"* (see getMarkup in Producer Portlet with More Than One Page – Consumer URL Writing for this string in the proper context), then the Consumer would replace it with the following URL:
>
> *http://consumer.com/containerpage?ut=Render&ns=2&m=view&ws=normal*

5.4.3.3 Inserting the HTML into the Consumer page

After processing the markup, the Consumer inserts it into the Consumer page, allowing for the fact that they may have different character sets.

5.4.4 Consideration on `usesMethodGet`

If `usesMethodGet` is `true`, and the Consumer wishes to support such a Producer, the interaction URL-s resulting from the templates or the Consumer-URL rewrites and that are embedded in an HTML `<form method="get">`'s action attribute[11] must take into consideration that most browsers strip the query part from the URL **[MUST]**. Two practical ways of doing this:

- All interaction URL-s embedded in the HTML will contain no query part, but rather embed the interaction parameters as part of the path.

[9] I think this should be a MUST: "if the Consumer wishes to preserve the flow of the entity application, it MUST preserve the `sessionContext` and send it in subsequent invocations"

[10] Are we uppercasing acronyms or not? In other words, is it `requires`**`Url`**`Rewriting` or `requires`**`URL`**`Rewriting`.

[11] This is not defined as a MUST in the v0.85 spec. I think it should be.

For example, the templates of such URL-s will look like the following (note the replacement of "?" by ";"):

```
<DefaultTemplate>
  http://consumer.com/containerpage;ut={urlType}&ns={wsrp-navigational-
state}&m={wsrp-mode}&ws={wsrp-windowState}&res={wsrp-url}
</DefaultTemplate>
```

Another method of passing the information, is as a path:

```
<DefaultTemplate>
  http://consumer.com/containerpage/ut={urlType}&ns={wsrp-navigational-
state}&m={wsrp-mode}&ws={wsrp-windowState}&res={wsrp-url}
</DefaultTemplate>
```

Both are poorly supported by application servers. Choose the one which fits you best.

- The Consumer parses the HTML, remove the URL parameters in the URL-s of `<form method="get">`'s action attributes, and replace them with hidden fields with the corresponding name and values of the removed URL parameters.

5.4.5 Processing "Resource" requests

The Producer may have generated markup that instructs the End user agent to send "Resource" requests to the Consumer. If the Producer used Producer URL-writing, then the Producer did so by inserting the `RenderTemplate` or `SecureRenderTemplate` into the markup, and if the Producer used Consumer URL-writing, then the Producer did so by using a `urlType` with a value of "`Resource`".

The Consumer should[12], upon receiving a request to the "Resource" URL (usually an HTTP GET), return the resource defined by requesting the resource defined in `wsrp-url` and returning it, just like an HTTP reverse proxy (a.k.a. HTTP gateway) would.

5.5 Next Page Composition Flow

Although not required, the Consumer typically writes all "interaction URL-s" in the markup so that they link back to the Consumer, while passing back the "interaction parameters" (e.g. `wsrp-navigationalState`, `wsrp-url`) specified by the Producer. The flow which composes the markup of the next page of the Consumer is similar to the flow of the first page, except that the Consumer processes the interaction parameters passed in the interaction URL:

- The Consumer processes the `urlType` in order to determine whether to invoke `performBlockingInteraction` before returning any markup to the end user is needed, or whether to invoke `performInteraction` before invoking `getMarkup` is needed. This is described in detail in Processing the urlType.
- The Consumer processes the `wsrp-mode` and `wsrp-windowState` interaction parameters to determine whether a mode and/or window state change is requested by the Producer. The Consumer usually allows these requests unless it has an overriding reason not to (e.g. access control). The Consumer passes the new mode and window state to the invocations of `getMarkup`, `performInteraction`, `performBlockingInteraction` for this next page.
- The Consumer processes the `wsrp-navigationalState` interaction parameter. The Consumer passes its value to the invocations of `getMarkup`, and `performInteraction/performBlockingInteraction` for this next page.

[12] I believe this should be a MUST: "if the Consumer wants the markup to look good, the Consumer MUST…"

5.5.1 Processing the `urlType`

The `urlType` enables the Producer to indicate to the Consumer which operations are to be invoked on the next page request. Note that instead of passing back the `urlType` interaction parameter in the interaction URL, the Consumer can choose to use different URL-s altogether. In Producer URL-writing, this is accomplished by giving different URL-s in the various templates, and in Consumer URL-writing this is accomplished by writing different URL-s depending on the value of the `urlType`. As these methods are operationally identical to passing the `urlType` interaction parameter, this document will continue to refer to "the value of `urlType`" even though in some cases the `urlType` is not transferred.

Depending on the value of `urlType`, the Consumer does the following[13]:

- `BlockingAction`: The Consumer invokes `performBlockingInteraction`. The Consumer invokes this operation before returning any markup to the end user and before invoking `getMarkup`.
- `Action`: The Consumer invokes `performInteraction`. The Consumer invokes the operation before invoking `getMarkup`.
- `Render`: The Consumer invokes `getMarkup` as usual.

5.5.2 Invoking `performBlockingInteraction`

Invoking `performBlockingInteraction` is similar to invoking `getMarkup`. The same `registrationContext`, `portletEntityContext`, `runtimeContext`, `userContext`, and `markupParams` are passed to it, allowing for the fact that new window state, mode, and navigational state may be passed, as described above. Additionally, this operation requires an additional parameter – `interactionParams`.

For example, the Consumer sends the following `InteractionParams` when it receives a POST to its interaction URL-s:

```
<interactionParams>
  <portletEntityStateChange>Fault</portletEntityStateChange> [required]
  <validNewModes>view</validNewModes> [14]
  <validNewWindowStates>normal</validNewWindowStates>
  <uploadContext>[see below]
     <uploadData>name=Gil+Tayar&age=18</uploadData>
     <mimeType>application/x-www-form-urlencoded</uploadData>
  </uploadContext>
</interactionParams >
```

5.5.2.1 `portletEntityStateChange`

A minimal Consumer will set the `portletEntityStateChange` field to "Fault" to disable the ability of the Producer to change its state, and handling this state change is not a minimal requirement.

[13] Is the Consumer allowed to do otherwise? E.g., to invoke `performInteraction` on a `Render urlType`? The spec does not disallow it. It think it should.

[14] This field is optional, yet the semantics of what it means *not* to have this field are not defined.

If the user agent reached the interaction URL with data (e.g. with an HTTP POST), the Consumer should send this data to the Producer, while indicating the mime type of the data. **[Although this is not a MUST, failure to send this data when user agent sends it to the Consumer may result in not generating markup fragments meeting End User requirements[15]]**

5.5.2.2 Processing the `performBlockingInteraction` response

The Consumer processes `redirectURL`. If this field exists in the response, it indicates that the Producer would like the Consumer to redirect the end user to the URL defined in `redirectURL`. The Consumer should honor this request[16]. If `redirectURL` exists, all other fields are ignored **[MUST]**.

If no `redirectUrl` field exists, the `updateResponse` field is processed:

- `navigationalState`: this field indicates that the Producer wishes to (again) change it's `navigationalState`. The Consumer stores this information so that future invocations of `getMarkup` for this page should use this value **[MUST]**. A good way of doing this is to store the information in the Consumer URL, so that if the end user bookmarks this URL, it will return the Producer to the correct state. Storing the information in the URL necessitates the Consumer to redirect the user agent back to a Consumer URL which includes the new navigational state.
- `sessionContext`: the Consumer stores this information so that later markup interface operations to this Portlet send it.
- `portletEntityContext`: this field will only appear if `portletEntityStateChange` is "OK" or "Clone"[17], so a minimal Consumer can safely ignore this field.
- `newWindowState/newMode`: these fields indicate that the Producer wishes to change its window state and/or mode. If the Consumer honors this request, then the Consumer stores this information so that future invocations of `getMarkup` for this page should use this value **[MUST]**. A good way of doing this is to store the information in the Consumer URL, so that if the end user bookmarks this URL, it will return the Producer to the correct state. Storing the information in the URL necessitates the Consumer to redirect the user agent back to a Consumer URL which includes the new window state and mode.
- `markupContext`: the Producer can choose to return markup with this operation as an optimization. The Consumer can use this `markupContext` instead of invoking `getMarkup` afterwards, or it can choose to ignore this markup and invoke `getMarkup` again instead.

5.5.3 Invoking `performInteraction`

The Consumer invokes `performInteraction` the same way it invokes `performBlockingInteraction`, and processes the response in the same way, except for the fact that the response does not include `redirectURL`, `navigationalState`, `newWindowState`, `newMode`. Because these are not included, the Consumer can invoke this operation after markup has been returned to the End user, as the Consumer need not because of this operation.

---

[15] This is not in the spec, but I believe it should be.
[16] This should be a SHOULD, no?
[17] The wording in the spec says "MUST", but not explicitly.

5.5.4 Continuing with the markup

Invoking `getMarkup` is optional if the Producer returned markup in `performInteraction` or `performBlockingInteraction`. Otherwise the `getMarkup` occurs just as defined above, with the addition of the correct window state, mode, `navigationalState`, and `sessionContext`.

5.6 Producer-Termination Flow

If the Consumer invoked the register operation at the beginning of its relationship with the Producer, then the Consumer invokes the `deregister` operator at the end of its relationship with the Producer **[MUST]**, sending the `registrationContext` it received when it registered.

The relationship is considered ended when the invocation is successful. **[MUST]** This means that the Consumer continues to invoke the deregister operation until successful.

---

6 Producer with More Than One Portlet

This scenario is based on the Minimal Producer scenario. In this scenario, the Producer exposes more than one Portlet. The scenario will describe only the changes from the base scenario.

6.1 Implementation Summary

The Producer still implements the same operations as the base operation. This time, though, the `getServiceDescription` returns two Portlet Entity descriptions. Also, `getMarkup` checks the `portletEntityHandle` to determine what markup to return.

> The example exposes two entities – `theFirstPortlet` and `theSecondPortlet`.

6.2 `getServiceDescription` Logic

The `getServiceDescription` operation will now returning the description of the two entities.

> For example, the Producer returns the following XML:

```
<getServiceDescriptionResponse
  xmlns="http://www.oasis-open.org/committees/wsrp/v1/wsdl/types">
  <offeredEntities
     <portletEntityHandle>theFirstPortlet</portletEntityHandle
     <markupTypes>
        <markupType>text/html</markupType>
        <locales>en</locales>
        <modes>view</modes>
        <windowStates>normal</windowStates>
     </markupTypes>
  </offeredEntities>
  <offeredEntities>
     <portletEntityHandle>theSecondPortlet</portletEntityHandle>
     <markupTypes>
        <markupType>text/html</markupType>
        <locales>en</locales>
        <modes>view</modes>
```

```
        <windowStates>normal</windowStates>
      </markupTypes>
  </offeredEntities>
</getServiceDescriptionResponse>
```

6.3 `getMarkup` Logic

In this scenario, the Producer does not ignore the `portletEntityContext`, instead it looks at the `portletEntityHandle` to determine which markup to return.

For example, if the `portletEntityHandle` is `theFirstPortlet`, the Producer returns the following XML:

```
<getMarkupResponse
  xmlns="http://www.oasis-open.org/committees/wsrp/v1/wsdl/types">
  <markupContext>
     <markup>
        <![CDATA[
           <div class="portlet-font"><p>Hello, world!</p></div>
        ]]>
     </markup>
     <locale>en</locale>
     <markupType>text/html</markupType>
  </markupContext>
</getMarkupResponse>
```

while if the `portletEntityHandle` is `theSecondPortlet`, it returns the following XML:

```
<getMarkupResponse
  xmlns="http://www.oasis-open.org/committees/wsrp/v1/wsdl/types">
  <markupContext>
     <markup>
        <![CDATA[
           <div class="portlet-font"><p>Goodbye, world!</p></div>
        ]]>
     </markup>
     <locale>en</locale>
     <markupType>text/html</markupType>
  </markupContext>
</getMarkupResponse>
```

7 Consumer with Two Entities From Same Producer

This scenario is based on the Minimal Consumer scenario and the Producer with More Than One Portlet scenario. In this scenario, the Consumer embeds the two entities from the Producer.

Just like in Minimal Consumer, the Consumer does not assume anything about the Producer's or entities' meta-data.

The scenario will describe only the changes from the base scenario.

7.1 Implementation Summary

The Consumer flow is very similar to the one in the Minimal Consumer scenario, with two basic additions:

- The End User Initialization flow, where `initCookie` may need to be called, has to now take care of the "`perGroup`" flag in the Service description. In the Minimal Consumer scenario, the `initCookie` was called once per Portlet. In this scenario, because there is more than one entity, the call has to be called once per group of Portlets, as defined by the Producer.
- When receiving a request to the interaction URL, the Consumer must set up a mechanism whereby it can differentiate interactions coming from the first portlet from those coming from the second portlet.

7.2 Producer-Initialization Flow

The same initialization is done, except that now the per-Portlet meta data is read for the two entities it intends to embed.

Note that invoking `register` is per-Producer, and not per-Portlet. Thus, in this scenario, `register` is invoked only once.

7.3 End User Initialization Flow

If `requiresInitCookie` is "`perUser`" and the Consumer and Producer are communicating via HTTP, or if `requiresInitCookie` is "`perGroup`" and the `groupID` of all the entities it wishes to embed is the same `groupID`, then the Consumer invokes the `initCookies` exactly like in the base scenario.

If `requiresInitCookie` is "`perGroup`", but the `groupID` of the two entities is different, then the Consumer invokes the `initCookie` operation twice (once for each group) for each end user, and stores the returned cookies (returned in the `Set-Cookie` headers) for later incorporation into the other operations from the same end-user and same group **[MUST]**.

7.4 First Page Composition Flow

Like in the base scenario, the Consumer invokes `getMarkup` to retrieve the markup it wishes to incorporate into it's page. This time, it invokes `getMarkup` twice – once for each Portlet on the page. Note that the `getMarkups` can be invoked in parallel.

For example, the Consumer sends the following XML for the first `getMarkup`:

```
<getMarkup
  xmlns="http://www.oasis-open.org/committees/wsrp/v1/wsdl/types">
  <registrationContext>
     the context returned from the register operation, or nothing if no
registration
  </registrationContext>
  <portletEntityContext>
     <portletEntityHandle>theFirstPortlet</portletEntityHandle>
     <portletEntityState />
  </portletEntityContext>
  <runtimeContext />
  <userContext>
     <userContextID />
  </userContext>
```

```
  <markupParams>
     <clientData>
        <userAgent>
           Mozilla/4.5 (Macintosh; U; PPC)
        </userAgent>
     </clientData>
     <secureClientCommunications>
        false [see below]
     </secureClientCommunications>
     <userAuthentication>false</userAuthentication>
     <locale>en</locale>
     <markupCharacterSet>UTF-8</markupCharacterSet>
     <markupType>text/html</markupType>
     <mode>view</mode>
     <windowState>normal</windowState>
     <navigationalState>???</navigationalState>
     <templates>
        [see below]
     </templates>
  </markupParams>
</getMarkup>
```

The second `getMarkup` is identical except for the `portletEntityHandle` which is `theSecondPortlet`.

7.4.1 secureClientCommunications

If `needSecureCommunication` of a Portlet is "`all`", then the Consumer must receive the markup via a secure connection (e.g. use SSL when using HTTP), and if sending it back to the End User, must send it back via a secure connection. Note that in the case of two entities, when sending the combined markup back to the end user, it is sufficient for *one* of the entities to declare that it `needSecureCommunication` for the returned markup to be returned securely.

7.4.2 templates

If `doesUrlTemplateProcessing` is `true`, the Consumer supplies templates to enable Producer URL-writing **[MUST]**. To enable the Consumer to differentiate between these interaction URL-s afterwards, the templates are usually different and indicate the invoker of the interaction. The `NameSpacePrefix` should also be different.

For example, the Consumer sends the following XML to `theFirstPortlet`:

```
  <templates>
     <DefaultTemplate>
        http://consumer.com/containerpage?ut={urlType}&ns={wsrp-
navigational-state}&m={wsrp-mode}&ws={wsrp-windowState}&res={wsrp-
url}&eh=theFirstPortlet
     </DefaultTemplate>
     <SecureDefaultTemplate>
        https://consumer.com/containerpage?ut={urlType}&ns={wsrp-
navigational-state}&m={wsrp-mode}&ws={wsrp-windowState}&res={wsrp-
url}&eh=theFirstPortlet
     </SecureDefaultTemplate>
     <NameSpacePrefix>FJH1</NameSpacePrefix>
  </templates>
```

while it sends the following XML to `theSecondPortlet` (the only difference is in the eh URL parameter and the `NameSpacePrefix`):

```
  <templates>
     <DefaultTemplate>
        http://consumer.com/containerpage?ut={urlType}&ns={wsrp-
navigational-state}&m={wsrp-mode}&ws={wsrp-windowState}&res={wsrp-
url}&eh=theSecondPortlet
     </DefaultTemplate>
     <SecureDefaultTemplate>
        https://consumer.com/containerpage?ut={urlType}&ns={wsrp-
navigational-state}&m={wsrp-mode}&ws={wsrp-windowState}&res={wsrp-
url}&eh=theSecondPortlet
     </SecureDefaultTemplate>
     <NameSpacePrefix>FJH2</NameSpacePrefix>
  </templates>
```

Note that the mechanism used in the example above, whereby the differentiation is done by the type of the Portlet Entity, will not work if embedding two Portlets of the same type. A more general mechanism would use a unique id chosen by the Consumer and which differentiates between the two "instances" of the Portlet.

7.4.3 Processing the `markupResponse`

Processing the `markupResponse` is similar to the one in the base scenario, except that the processing is done twice.

7.4.4 Consideration on `usesMethodGet`

The processing is exactly the same as in the base scenario, except that the processing is done once per Portlet on the page.

7.4.5 Processing "Resource" requests

The processing is exactly the same as in the base scenario.

7.5 Next Page Composition Flow

Processing the next page is similar to the base scenario, except for the following differences:

- The interaction URL will indicate which Portlet on the page the end user interacted with. As described above, a typical way to do this is to embed a Portlet identifier in the interaction URL.
- When invoking the `performBlockingInteraction`, `performInteraction`, and/or `getMarkup`, the appropriate `portletEntityHandle` is sent.
- The Consumer will invoke `performBlockingInteraction` or `performInteraction` only on the Portlet the user interacted with[18].

[18] This should be a MUST.

- For the other entities (and optionally for the Portlet the user interacted with, if the `performInteraction`/`performBlockingInteraction` did not return markup) the Consumer invokes their `getMarkup` as defined in the base scenario **[SHOULD]**. The `getMarkup`-s can be invoked in parallel if the Consumer wants to, except for the Portlet the user interacted with, whose `getMarkup` is invoked after the `performInteraction`/`performBlockingInteraction` **[MUST[19]]**.
- If the interaction URL's `urlType` indicates to the Consumer to invoke `performBlockingInteraction`, then the Consumer invokes it *before* invoking the `getMarkup`-s **[MUST]**.

7.5.1 Invoking `performBlockingInteraction`

Invoking the `performBlockingInteraction` is similar to the base scenario, except that the Consumer must send the `portletEntityHandle` of the Portlet which performed the interaction, and pass the correct `sessionContext` of the Portlet.

7.5.1.1 Processing the `performBlockingInteraction` response

Processing the response is a bit different than the base scenario, so it will be reconstructed here:

The Consumer processes `redirectURL` like in the base scenario.

If no `redirectUrl` field exists, the `updateResponse` field is processed:

- `navigationalState`: this field indicates that the Portlet wishes to (again) change it's `navigationalState`. The Consumer stores this information so that future invocations of `getMarkup` for this page and Portlet should use this value **[MUST]**. Storing this information in the URL (as discussed in the base scenario) is still a good way to do this, although it should store the `navigationalState` in a URL parameter that is specific to that Portlet so that the `navigationalState` of each Portlet is independent. This approach will not scale to more than two or three entities, so a different approach is needed if the Consumer wishes to enable simultaneous navigation of more than three entities. An alternative approach would be to store this information in the Consumer's end user session.
- `sessionContext`: the Consumer stores this information per Portlet on the page and per user so that later markup interface operations to this Portlet send it.
- `newWindowState/newMode`: these fields indicate that the Portlet on the page wishes to change its window state and/or mode. The Consumer stores this information so that future invocations of `getMarkup` for this page and Portlet should use this value **[MUST]**. Storing this information in the URL (as discussed in the base scenario) is still a good way to do this, although it should store the information in a URL parameter that is specific to that Portlet on the page so that the information for each Portlet on the page is independent. This approach will not scale to more than two or three entities, so a different approach is needed if the Consumer wishes to enable simultaneous navigation of more than three entities. An alternative approach would be to store this information in the Consumer's end user session.
- `markupContext`: the Producer can choose to return markup with this operation. The Consumer can use this `markupContext` instead of invoking `getMarkup` afterwards, or it can choose to ignore this markup and invoke `getMarkup` again instead.

[19] This is not a MUST, but it MUST be!

7.5.2 Invoking `performInteraction`

Invoking the `performBlockingInteraction` is similar to the base scenario, except that the Consumer must send the `portletEntityHandle` of the Portlet on the page which performed the interaction, and pass the correct `sessionContext` of the Portlet on the page.

This operation can also be invoked in parallel to the `getMarkup` of the other Portlet on the page (the Portlet that did not interact with the end user), although not in parallel with the `getMarkup` of the interacting Portlet.

7.5.3 Continuing with the markup

The Consumer invokes `getMarkup` for the first and second Portlet, passing the correct `portletEntityHandle`, `sessionContext`, `navigationalState`, `windowState`, and mode. These can be invoked in parallel with each other.

Handling the `getMarkup` of the interacting Portlet is optional if the Producer returned markup in `performInteraction` or `performBlockingInteraction`.

7.6 Producer-Termination Flow

This is exactly like the process in the base scenario.

---

8 Producer Portlet with POST and Session

This scenario is based on the Minimal Producer scenario, except that the end user can POST information in the first page, which the Producer processes to show a second page.

8.1 Implementation Summary

Just like in Producer Portlet with More Than One Page – Producer URL Writing, the Producer needs to differentiate between `getMarkup` for the first page, and `getMarkup` for the second page. And just like in that scenario, the tool for that is the `navigationalState` field in `getMarkup` and the interaction parameter `wsrp-navigationalState`.

Unlike that scenario, the second page's `getMarkup` needs information – information that was posted from the first page. This information can be stored in two places –

- The Producer session. This makes the amount of information theoretically limitless, but makes the Producer stateful and session-based.
- the `navigationalState`. This enables the Producer to remain stateless, but limits the amount of information that can be stored.

This scenario will use the first method. A similar scenario, Producer Portlet with POST & Redirect, will use the second method.

> In the example, the first page queries for name and age of the user, which it posts to the second page, which displays them.

The Producer implements the following operations **[the Producer MUST implement them]**:

- `getServiceDescription`
- `getMarkup`

- `performInteraction`: because getting the POST information can only be done using `performInteraction` and `performBlockingInteraction` (via the `uploadData` field), the Producer chooses to use `performInteraction` because it doesn't restrict the Consumer as much as `performBlockingInteraction`, and the Producer does not need the additional capabilities of `performBlockingInteraction`.
- `performBlockingInteraction`: the implementation can be an empty implementation which fails.
- `initCookie:` the implementation can be an empty implementation which returns "void".

Note that using `performInteraction` should be done with care. Using it means that it may be invoked in parallel with other `getMarkup`s, and specifically with `getMarkup` from other Portlets in the same Producer. If this can happen, care must be taken the parallel invoking of `performInteraction` and `getMarkup` will be OK.

8.2 `getServiceDescription` Logic

Exactly like the base scenario, except that the Portlet's `doesUrlTemplateProcessing` needs to be `true`, as the Producer implements Producer URL Writing.

For example, the Producer returns the following XML:

```
<getServiceDescriptionResponse
  xmlns="http://www.oasis-open.org/committees/wsrp/v1/wsdl/types">
  <offeredEntities>
     <portletEntityHandle>theOnlyPortlet</Portlet EntityHandle>
     <markupTypes>
        <markupType>text/html</markupType>
        <locales>en</locales>
        <modes>view</modes>
        <windowStates>normal</windowStates>
     </markupTypes>
     <doesUrlTemplateProcessing>true</doesUrlTemplateProcessing>
  </offeredEntities>
</getServiceDescriptionResponse>
```

8.3 `getMarkup` Logic

The Producer ignores the same parameters as in the base scenario, except for `navigationalState` and `sessionHandle`. Based on the `navigationalState` it will know which page to display. The `sessionHandle` will point to the Producer session which holds the name and age inputted in the first page (see performInteraction to understand how the information got into the session).

For example, the Producer decides that the `navigationalState` for the first page is simply the string "1" and for the second "result" page it is the string "1result". This example also assumes the Consumer URL templates are the ones in templates defined in the Minimal Consumer scenario.

If the `navigationalState` sent by the Consumer is "1", the following XML will be returned:

```
<getMarkupResponse
  xmlns="http://www.oasis-open.org/committees/wsrp/v1/wsdl/types">
  <markupContext>
     <markup>
        <![CDATA[
           <p>Hello, world! This is the first page!</p>
```

```
            <form method="POST" action="
http://consumer.com/containerpage?ut=Action&ns=2&m=view&ws=normal&res="
>
            <div class="portlet-form???[20]">Enter your name:</div>
            <input class="form-input-field"
                                type="input" id="FJH1_Name"></input>
            <div class="portlet-form???">Enter your age:</div>
            <input class="form-input-field"
                                type="input" id="FJH1_Age"></input>
            </form>
        ]]>
    </markup>
    <locale>en</locale>
    <markupType>text/html</markupType>
  </markupContext>
</getMarkupResponse>
```

If the `navigationalState` sent by the Consumer is "1result", the following XML will be returned (assuming the information in the Producer was that the `name` is `Gil Tayar` and the age is `18`[21]:

```
<getMarkupResponse
  xmlns="http://www.oasis-open.org/committees/wsrp/v1/wsdl/types">
  <markupContext>
    <markup>
      <![CDATA[
        <div class="portlet-font">
          <p>Hello, Gil Tayar! What an age 18 is!</p>
        </div>
      ]]>
    </markup>
    <locale>en</locale>
    <markupType>text/html</markupType>
  </markupContext>
</getMarkupResponse>
```

Note the fact that even if this Portlet had used Consumer URL writing, the `requiresUrlRewriting` in the second XML would still have been `false`, because there are no links in that page.

8.4 `performInteraction` Logic

Because the first page includes a `<form method=post>`, and that POST data reaches the Consumer (through the mechanism of URL writing), the Producer can receive it only by implementing `performInteraction` or `performBlockingInteraction`. This Producer chooses to use `performInteraction` because it doesn't restrict the Consumer as much as `performBlockingInteraction`, and the Producer does not need the additional capabilities of `performBlockingInteraction` (e.g. changing `navigationalState` because of the interaction).

The Producer ignores the same parameters as in the base scenario's `getMarkup`. Note that it also ignores `sessionHandle` because it does not need to read information from the session in `performInteraction`, only to write information to it. Reading the information from the session is done in getMarkup.

[20] I'm not sure what the class for an input field label is.

[21] One can dream…

The Producer ignores most of the fields in `interactionParams` too, except for the `uploadContext` which contains the POST-ed data.

For an example of such an `uploadContext`, see Invoking performBlockingInteraction in Minimal Consumer. Assuming such an `uploadContext`, the Producer returns the following XML (assuming it doesn't return markup):

```
<performInteractionResponse
  xmlns="http://www.oasis-open.org/committees/wsrp/v1/wsdl/types">
  <sessionContext>
      <sessionHandle>
         a handle to the Producer session which includes the name and
age sent in the uploadData. This is producer-implementation-dependent.
      </sessionHandle>
      <expires>3600</expires>
  </sessionContext>
</performInteractionResponse>
```

9 Producer Portlet with POST & Redirect

This scenario is based on Producer Portlet with POST, except for the fact that it stores the information in the `navigationalState` and not in the session (see getMarkup in Producer Portlet with POST for a description of the differences between the two methods).

9.1 Implementation Summary

This necessitates `performInteraction` to change the `navigationalState`, but because the this operation cannot change the `navigationalState`, the Producer uses `performBlockingInteraction`. This is very similar to existing Web applications redirecting as a result of a POST. In fact, if the Producer returns new `navigationalState` from a `performBlockingInteraction` operation, some Consumers will redirect the user agent to reflect the change in the `navigationalState` (see Processing the performBlockingInteraction response in Next Page Composition Flow of the Minimal Consumer scenario).

The Producer implements the following operations **[the Producer MUST implement them]**:

- `getServiceDescription`
- `getMarkup`
- `performInteraction`: the implementation can be an empty implementation which fails.
- `performBlockingInteraction`: because getting the POST information and returning a new `navigationalState` as a result can only be done using `performBlockingInteraction` the Producer chooses to implement this operation.
- `initCookie`: the implementation can be an empty implementation which returns "void".

9.2 `getServiceDescription` Logic

Exactly like the base scenario, except that the Portlet's `doesUrlTemplateProcessing` needs to be `true`.

For example, the Producer returns the following XML:

```
<getServiceDescriptionResponse
```

```
  xmlns="http://www.oasis-open.org/committees/wsrp/v1/wsdl/types">
  <offeredEntities>
     <portletEntityHandle>theOnlyPortlet</portletEntityHandle>
     <markupTypes>
        <markupType>text/html</markupType>
        <locales>en</locales>
        <modes>view</modes>
        <windowStates>normal</windowStates>
     </markupTypes>
     <doesUrlTemplateProcessing>true</doesUrlTemplateProcessing>
  </offeredEntities>
</getServiceDescriptionResponse>
```

9.3 `getMarkup` Logic

Just like in Producer Portlet with More Than One Page – Producer URL Writing, the Producer needs to differentiate between `getMarkup` for the first page, and `getMarkup` for the second page. And just like in that scenario, the tool for that is the `navigationalState` field in `getMarkup` and the interaction parameter `wsrp-navigationalState`.

Unlike that scenario, the second page's `getMarkup` needs information – information that was posted from the first page. As discussed above, This information will be stored by `performBlockInteraction` in the `navigationalState`.

The Producer ignores the same parameters as in the base scenario, except for `navigationalState`. Based on the `navigationalState` it will know which page to display and what information to display in it.

For example, the Producer decides that the `navigationalState` for the first page is simply the string "1" and for the second "result" page it is the string "`1result?name=`*name*`&age=`*age*". This example also assumes the Consumer URL templates are the ones in templates.

If the `navigationalState` sent by the Consumer is "1", the following XML will be returned:

```
<getMarkupResponse
  xmlns="http://www.oasis-open.org/committees/wsrp/v1/wsdl/types">
  <markupContext>
     <markup>
        <![CDATA[
           <p>Hello, world! This is the first page!</p>
           <form method="POST" action="
http://consumer.com/containerpage?ut=BlockingAction&ns=2&m=view&ws=norm
al&res=">
           <div class="portlet-form???[22]">Enter your name:</div>
           <input class="form-input-field"
                          type="input" id="FJH1_Name"></input>
           <div class="portlet-form???">Enter your age:</div>
           <input class="form-input-field"
                          type="input" id="FJH1_Age"></input>
           </form>
        ]]>
     </markup>
     <locale>en</locale>
     <markupType>text/html</markupType>
  </markupContext>
</getMarkupResponse>
```

[22] I'm not sure what the class for an input field label is.

If the `navigationalState` sent by the Consumer is "`1result?name=Gil+Tayar&age=18`[23]", the following XML will be returned:

```
<getMarkupResponse
  xmlns="http://www.oasis-open.org/committees/wsrp/v1/wsdl/types">
  <markupContext>
     <markup>
        <![CDATA[
           <div class="portlet-font">
              <p>Hello, Gil Tayar! What an age 18 is!</p>
              </a>
           </div>
        ]]>
     </markup>
     <locale>en</locale>
     <markupType>text/html</markupType>
  </markupContext>
</getMarkupResponse>
```

Note the fact that even if this Portlet had used Consumer URL writing, the `requiresUrlRewriting` in the second XML would still have been `false`, because there are no links in that page.

9.4 `performBlockingInteraction` Logic

Because the first page includes a `<form method=post>`, and that POST data reaches the Consumer (through the mechanism of URL writing), the Producer can receive it only by implementing `performInteraction` or `performBlockingInteraction`. because getting the POST information and returning a new `navigationalState` as a result can only be done using `performBlockingInteraction` the Producer chooses to implement this operation.

The Producer ignores the same parameters as in the base scenario's `getMarkup`. Note that it also ignores `sessionHandle` because it does not need to read information from the session in `performInteraction`, only to write information to it. Reading the information from the session is done in getMarkup.

The Producer ignores most of the information in `interactionParams`, except for the `uploadContext` which contains the POST-ed data.

For an example of such an `uploadContext`, see Invoking performBlockingInteraction in Minimal Consumer. Assuming such an `uploadContext`, the Producer returns the following XML (assuming it doesn't return markup):

```
<performBlockingInteractionResponse
  xmlns="http://www.oasis-open.org/committees/wsrp/v1/wsdl/types">
  <updateResponse>
     <navigationalState>
        1result?name=Gil+Tayar&age=18
     </navigationalState>
  </updateResponse>
</performBlockingInteractionResponse>
```

[23] One can dream…

10 Producer that Includes Resources to be Proxied

This scenario is based on Minimal Producer, but the single HTML page returned includes an image, which the Consumer has to proxy (as described in Processing "Resource" requests in the Minimal Consumer scenario). The Producer uses Producer URL writing.

10.1 Implementation Summary

The Producer implements the same operations as in the basic scenario. The only difference is that in the markup returned, it uses Resource writing – the Consumer should have put the `{ws-url}` interaction parameter in the interaction URL. The Producer just needs to change t to point to the resource in question.

10.2 `getServiceDescription` Logic

The `getServiceDescription` implementation is similar to the basic scenario's, except that `doesUrlTemplateProcessing` is true.

For example, the Producer returns the following XML:

```
<getServiceDescriptionResponse
  xmlns="http://www.oasis-open.org/committees/wsrp/v1/wsdl/types">
  <offeredEntities> [while not required, is essential for the Consumer
to send meta-data to consumer]
     <portletEntityHandle>theOnlyPortlet</portletEntityHandle>
[required]
     <markupTypes> [required]
        <markupType>text/html</markupType> [required]
        <locales>en</locales> [required]
        <modes>view</modes> [required]
        <windowStates>normal</windowStates> [required]
     </markupTypes>
     <doesUrlTemplateProcessing>true</doesUrlTemplateProcessing>
  </offeredEntities>
</getServiceDescriptionResponse>
```

10.3 `getMarkup` Logic

The Producer ignores the same parameters and fields as in the basic scenario. The only difference is in the markup returned.

For example, the Producer returns the following XML (assuming the Consumer sends the templates defined in templates in the Minimal Consumer scenario):

```
<getMarkupResponse
  xmlns="http://www.oasis-open.org/committees/wsrp/v1/wsdl/types">
  <markupContext>
     <markup>
        <![CDATA[
           <div class="portlet-font"><p>Hello, world!</p></div>
           <img src="
http://consumer.com/containerpage?ut=Resource&ns=&m=&ws=&res=http://pro
ducer.com/images/sayhello.jpg"></img>
        ]]>
     </markup>
     <locale>en</locale>
```

```
      <markupType>text/html</markupType>
      <requiresUrlRewriting>true</requiresUrlRewriting>
  </markupContext>
</getMarkupResponse>
```

11 Producer that Uses More Modes

12 Consumer that Supports More Modes

13 Producer that Uses More Window States

14 Consumer that Supports More Window States

15 Producer that Uses Registration

16 Producer that Supports Consumer Configured Entities

17 Consumer that Uses Consumer Configured Entities