# Mashup-Aware Corporate Portals

Sandy Pérez and Oscar Díaz

ONEKIN Research Group, University of the Basque Country,
San Sebastián, Spain
{sandy.perez,oscar.diaz}@ehu.es

**Abstract.** Unlike other Web applications, corporate portals reckon to provide an integration space for corporate services. Mashups contribute to this goal by bringing a relevant customization technique whereby portal users can supplement portal services with their own data needs. The challenge is to find a balance between portal reliability and mashup freedom. Our approach is to split responsibilities between service providers and portal users. Providers decide on how services can be mashuped, portal users determine the supplemented content, and finally, the portal engine mediates between the two. This permits portal services to be reliably customized through user mashups. The approach is realized for *Liferay* as the portal engine, portlets as the realization of portal services, and XBL as the integration technology.

**Keywords:** portal, portlet, mashup, customization, presentation integration.

## 1 Introduction

Traditional mashuping distinguishes two scenarios w.r.t. source applications. In the first scenario, the mashup is a separate application from source applications (e.g. *Yahoo! Pipes*). In the second scenario, the mashup is an enhancement upon the source application (e.g. *MashMaker [4], MARGMASH* [3]). This normally requires the installation of a browser plugin for the mashup to be woven with the application markup. In both cases, source applications ignore they are being subject to mashup ("unaware mashuping"). Corporate portals are Web applications. Hence, unaware mashuping can also be applied to portals. However, being "unaware", this approach cannot capitalize on portal utilities (e.g. single-sign on, access control, customization, etc.). You can use *MashMaker* on portals but the role of the portal is totally passive. However, and unlike other Web applications, portals reckon to provide an integration space for corporate services. By mashuping at the back of the portal, mashuping misses the opportunity to benefit from this integration space. This paper introduces a collaborative approach where portals actively support mashuping ("mashup-aware portals").

This departs from traditional scenarios. First, and unlike *Yahoo! Pipes*-like approaches, the mashup is offered without leaving the portal. Second, and unlike *MashMaker*-like approaches, now the portal takes an active role on facilitating mashups on portal services. The implications are three-fold: (1) no additional plugin is necessary since mashup weaving is already engineered into the portal, (2) the portal "guides" users throughout the mashup process, and (3), the portal provides the context for mashups to be seamlessly integrated into portal services.

From the portal perspective, mashuping becomes an additional approach to customize portal offerings. Customization helps portal services (e.g. booking flight tickets) to be adapted to the users' roles. Both content and services can be adapted to the current user (e.g. *flightBooking* is only available for senior engineers). Personalization goes one step further by permitting users themselves to set some configuration options (e.g. the *destinationAirport* parameter is set to *"New York"* by *John Douglas*). This work introduces mashups as an additional personalization mechanism. For instance, *John Douglas* is very apprehensive to weather conditions so that he looks at the weather forecast before setting the trip date. This just applies to *Mr. Douglas*, and it is not contemplated by *flightBooking*. Hence, *Mr. Douglas* is forced to move outside the portal realm to satisfy this data need (e.g. through a *weatherForecast* widget), and to bridge himself the passing of data from the portal to the widget. By contrast, mashup-aware portals would assist *Mr. Douglas* in weaving the *weatherForecast* widget to the *flightBooking* service.

The paper addresses the challenges that this approach poses for both the portal and the providers of portal services. The approach is borne out for *Liferay*[1] as the portal engine, portlets as the service realization technology [2], and *XBL* as the binding technology [8]. The paper begins by introducing the challenges.

## 2 Introducing the Challenges

We regard portal mashups as enhancements provided *by* users but accomplished *through* the portal. This introduces a distinction among the tasks offered through the portal: **main tasks** (e.g. *flightBooking*) and **mashup tasks** (e.g. *weatherForecast*).

**Main tasks** are set by the portal administrator. They support the functional backbone of the portal. Portlets are the standardized approach for supporting these presentation-oriented services. Portlets strive to play at the front end the same role that Web services currently enjoy at the back end, namely, enablers of application assembly through reusable services. Portlets are user-facing (i.e. return markup fragments rather than data-oriented XML) and multi-step (i.e. they encapsulate a chain of steps rather than a one-shot delivering) [2]. The latter is worth noticing: service fulfilment is rarely achieved through a single shot but a *set* of steps are needed (e.g. date setting, site booking, entering billing data and so on). Each step is supported through a markup (a.k.a. a portlet fragment). Backed by the WSRP and JSR286 standards, portlets are currently supported by major portal vendors.

On the other hand, **mashup tasks** are subordinated to main tasks. Normally, they consult and provide additional data rather than updating the service state. Unlike main tasks, mashup tasks are set by portal users. We choose *widgets* as the realization technology for mashup tasks. Widgets are full-fledged client-side applications that are authored using Web standards and packaged for distribution [7].

Traditionally, portal tasks (whether being supported through portlets or widgets) are readily presented as you enter the portal page. This is based on the premise that tasks are all equal. In our scenario, this premise however, is not longer valid: mashup tasks should not be readily available but only when they are needed. Otherwise you will end up with cluttered portal pages full of widgets with no obvious purpose. The purpose of

[1] http://www.liferay.com/

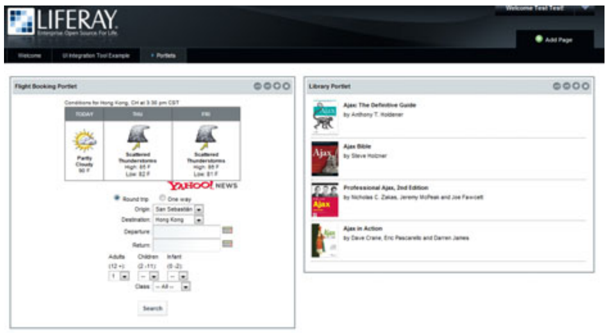

LIFERAY
Enterprise Open Source For Life
Welcome Test Test!
Add Page
Welcome
UI Integration Tool Example
Portlets
Flight Booking Portlet
Conditions for Hong Kong, CH at 3:30 pm CST
TODAY
THU
FRI
Partly Cloudy 90 F
Scattered Thunderstorms High: 85 F Low: 82 F
Scattered Thunderstorms High: 85 F Low: 81 F
YAHOO! NEWS
Round trip
One way
Origin: San Sebastián
Destination: Hong Kong
Departure
Return
Adults (12 +): 1
Children (2 -11): --
Infant (0 -2): --
Class: -- All --
Search
Library Portlet
Ajax: The Definitive Guide
by Anthony T. Holdener
Ajax Bible
by Steve Holzner
Professional Ajax, 2nd Edition
by Nicholas C. Zakas, Jeremy McPeak and Joe Fawcett
Ajax in Action
by Dave Crane, Eric Pascarello and Darren James

**Fig. 1.** Rendering of two portlets (i.e. *flightBooking* and *librarySearch*), and one widget (i.e. *weatherForecast*). The widget is user-specific and supports the selection of the flight date. Once *flightBooking* moves to the next stage, *weatherForecast* is not longer available.

a mashup task should be sought in the context of a main task. In our previous example, *weatherForecast* only makes sense when *flightBooking* reaches the point of prompting for the trip date. Once *flightBooking* moves to the next stage, *weatherForecast* is of no use. Additionally, the *weatherForecast* widget should be located the closest to the entry form for the destination airport. However, traditional side-by-side composition would assign *flightBooking* and *weatherForecast* distinct (although co-located) cells. It is then possible for the departure airport to appear at the bottom of the cell while forecast data is rendered upper on the page.

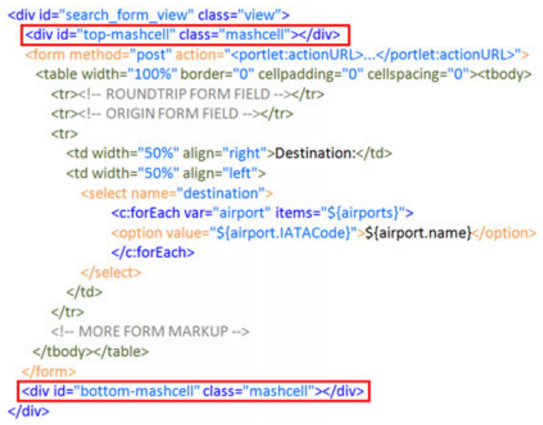
Mashup-aware portals permit main tasks to inlay mashup tasks (rather than being co-located). Figures 1 provides an example. First, *flightBooking* and *library* are presented side-by-side as realization of main portal tasks. By contrast, mashup tasks, such as *weatherForecast*, can now be inlayed within the rendering space of *flightBooking*.

This scenario introduces three actors, namely

- **the portal user**. *Once the portal is deployed*, users can require additional data to better accomplish main tasks. Akin to the DIY approach, it is up to users to find the appropriate sources for mashup tasks (e.g. widgets). *Requirement: hot-deployment of mashup tasks.*
- **the portlet provider**. Portlet providers ensure the quality of main tasks (data integrity, service throughput, etc.). Now, they are also responsible to decide how the portlet can be mashuped. Portlet designers should foresee placeholders to inlay mashup tasks on accomplishing the portlet task (e.g. on selecting the destination airport). *Requirement: placeholder specification for main tasks.*
- **the portal** (i.e. the portlet consumer). Akin to the portal-as-an-integration-space, portals should offer weaving mechanisms that permit data to seamlessly flow between main tasks and mashup tasks. *WeatherForecast* provides an example: its parameter *"location"* is to be obtained from the *flightBooking* destination airport, so that every time the airport is changed, the inlayed widget is refreshed. Requirement: *data-flow portal facilities.*

## 3 Realizing the Portlet Provider Perspective

Portlets support well-focus functionality: booking a flight seat, handling a bank transfer, and so on. On the other hand, portlets are born to be reused. The same portlet can be offered through different portals. As in any other component technology, this implies an attempt to foresee requirements for distinct potential consumers. However, traditional component development already advises that "*no design can provide information for every situation, and no designer can include personalized information for every user*" [5]. This is when mashups come into play. Mashups permit portal users to complement portlet functionality. It is most important to notice that we *do not mashup the portlet as such but the offering of this portlet through this portal.* The very same portlet can have a different mashup when offered through a distinct portal.



```
<div id="search_form_view" class="view">
  <div id="top-mashcell" class="mashcell"></div>
  <form method="post" action="<portlet:actionURL>...</portlet:actionURL>">
   <table width="100%" border="0" cellpadding="0" cellspacing="0"><tbody>
     <tr><!-- ROUNDTRIP FORM FIELD --></tr>
     <tr><!-- ORIGIN FORM FIELD --></tr>
     <tr>
       <td width="50%" align="right">Destination:</td>
       <td width="50%" align="left">
         <select name="destination">
              <c:forEach var="airport" items="${airports}">
              <option value="${airport.IATACode}">${airport.name}</option>
              </c:forEach>
         </select>
       </td>
     </tr>
     <!-- MORE FORM MARKUP -->
   </tbody></table>
  </form>
  <div id="bottom-mashcell" class="mashcell"></div>
</div>
```

**Fig. 2.** The *"search_form_view"* fragment: two *mashcells* pingpoint mashup placeholders

A similar situation arises in *XML Schema*. Schema standards are set by international bodies. Since the specificities of each sector/country can be difficult or inappropriate to be directly captured by the general schema, extension points are defined for consumers to adapt the schema to their own contexts. Likewise, portlet designers need to find a balance when supporting the portlet functionality, i.e. the portlet should be general enough to be appropriate for a large set of consumers while including "mashup placeholders" to cater for mashup specifics (hereafter referred to as *"mashcells"*). Their role is similar to the <*any*> element in XML schemas.

Mashcells do not have any presentation impact other than pinpointing where portlet markups can be extended. Implementation wise, mashcells are supported as the *CSS* class *"mashcell"*. Figure 2 shows a snippet for the *"search_form_view"* fragment for *flightBooking*. The designer decides to provide two mashcells right before and after the entry form: *"top-mashcell"* and *"bottom-mashcell"*, respectively. Portal tools can then light up these mashcells, pinpointing mashup placeholders for users to fill up with desired widgets. This moves us to the portal perspective.

## 4 Realizing the Portal Perspective

Now, portlets become "the canvas" where widgets can be placed. The design space is then set on a portlet basis. Broadly, this space comprises three dimensions (see Fig. 3):

– *what* to include (i.e. widget selection). Values stand for the widgets available at the portal. Permission can be granted for portal users to add their own widgets, hence personalizing their own data purveyors,
– *where* is to be included. Values correspond to mashcells for the portlet at hand,
– *how* is to be included. Values stand for potential "data feeds" to be obtained from the portlet markup.

Figure 3 shows the design space for the *flightBooking* portlet. This space frames the setting for deciding *what-where* (hereafter, referred to as *"composition coordinate"*), and *where-how* (referred to as *"orchestration coordinate"*). For the sample problem, the *weatherForecast* gadget[2] is to be inlayed into the *"top-mashcell"* (composition coordinate). Additionally, this gadget is to be fed after the *destination* airport entry form (orchestration coordinate). Next paragraphs introduce two requirements to be fulfilled by the implementation technology: dynamic binding and presentation-based orchestration.

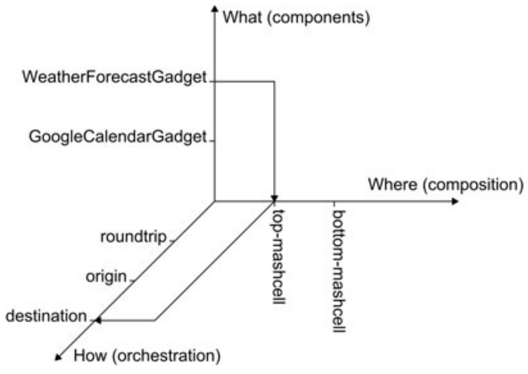


**Fig. 3.** Design space for the *flighBooking* portlet

**Dynamic binding:** coordinates can be set once the portal is already deployed. Behaving as a kind of portal preferences for decision taken, widgets and the associated coordinates can be added by portal users at any time. This hot-deployment of widgets implies the ability to dynamically define coordinates.

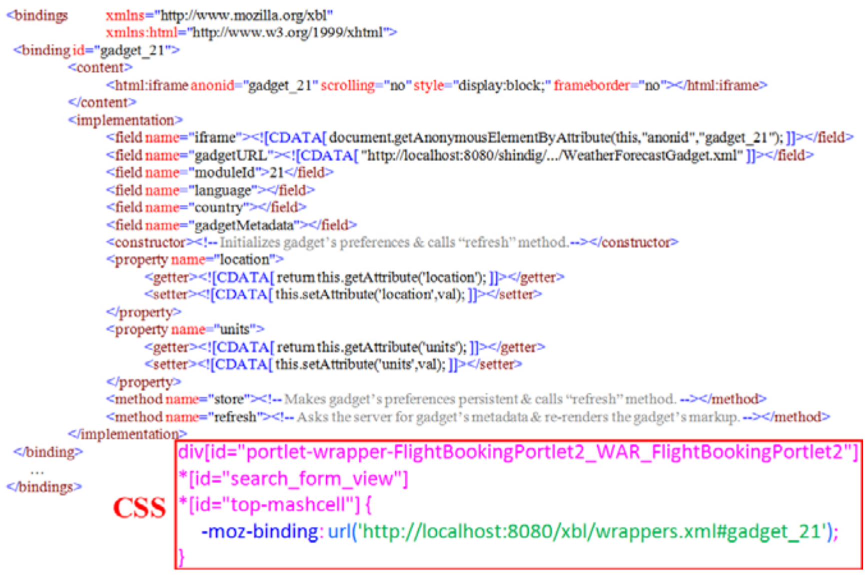
**Presentation-centred orchestration:** widget parameters can be obtained from portlet markup. Being both portlets and widgets presentation components, it is just natural to use events for this purpose[3]. The fact of *HTML* being the standard for delivering rendering through the Web makes *HTML* the *lingua franca* for portlet-widget communication. That is, the *DOM API* is used [6]. This *API* provides a set of low-level UI events (e.g. load, mouse over, etc.) to operate on low-level *UI* components (e.g. menus, button, etc.). Therefore, we rely on *DOM* events to specify the orchestration model. This forces composition to take place at the client[4]. Based on these two requirements (i.e. dynamic binding, and client-based, *DOM* event-based orchestration), *XML Binding Language (XBL)* [8] is selected.

*XBL* is a *W3C* candidate recommendation for describing bindings that can be attached to elements in other documents. It is currently supported by Firefox. The element that the binding is attached to, called the *bound element*, acquires the new behaviour specified by the binding [8]. An XBL document has <*bindings*> as a root. The <*bindings*> tag contains a collection of <*binding*> elements that describe element behaviour. Each <*binding*> can include the <*content*> tag, which is used to describe anonymous content that can be inserted around a *bound element*, an <*implementation*> tag that captures new properties and methods, and a <*handlers*> tag to account for event handlers on the *bound element*. Next, bindings can be attached to elements through *CSS*

[2] Gadgets are a concrete technology for widgets.

[3] Portlets already have a standard that permits portlets to synchronize their lifecycles through events [1]. But this leaves widgets out.

[4] Actually, portlets produce the markup at the server while widgets are scripts to be run at the client.



```
<bindings       xmlns="http://www.mozilla.org/xbl"
                xmlns:html="http://www.w3.org/1999/xhtml">
 <binding id="gadget_21">
        <content>
            <html:iframe anonid="gadget_21" scrolling="no" style="display:block;" frameborder="no"></html:iframe>
        </content>
        <implementation>
            <field name="iframe"><![CDATA[ document.getAnonymousElementByAttribute(this,"anonid","gadget_21"); ]]></field>
            <field name="gadgetURL"><![CDATA[ "http://localhost:8080/shindig/.../WeatherForecastGadget.xml" ]]></field>
            <field name="moduleId">21</field>
            <field name="language"></field>
            <field name="country"></field>
            <field name="gadgetMetadata"></field>
            <constructor><!-- Initializes gadget's preferences & calls "refresh" method.--></constructor>
            <property name="location">
                <getter><![CDATA[ return this.getAttribute('location'); ]]></getter>
                <setter><![CDATA[ this.setAttribute('location',val); ]]></setter>
            </property>
            <property name="units">
                <getter><![CDATA[ return this.getAttribute('units'); ]]></getter>
                <setter><![CDATA[ this.setAttribute('units',val); ]]></setter>
            </property>
            <method name="store"><!-- Makes gadget's preferences persistent & calls "refresh" method. --></method>
            <method name="refresh"><!-- Asks the server for gadget's metadata & re-renders the gadget's markup. --></method>
        </implementation>
 </binding>
    ...
</bindings>
```

CSS

```
div[id="portlet-wrapper-FlightBookingPortlet2_WAR_FlightBookingPortlet2"]
*[id="search_form_view"]
*[id="top-mashcell"] {
    -moz-binding: url('http://localhost:8080/xbl/wrappers.xml#gadget_21');
}
```

**Fig. 4.** *XBL* support for the composition coordinate *(WeatherForecastGadget,top-mashcell)*

using the *-moz-binding* property. Next sections look at how *XBL* can be used to realize both *composition coordinates* and *orchestration coordinates*. *XBL* files can be bound dynamically, hence, changing the coordinates based on either the profile of the current user or the portal configuration parameters.

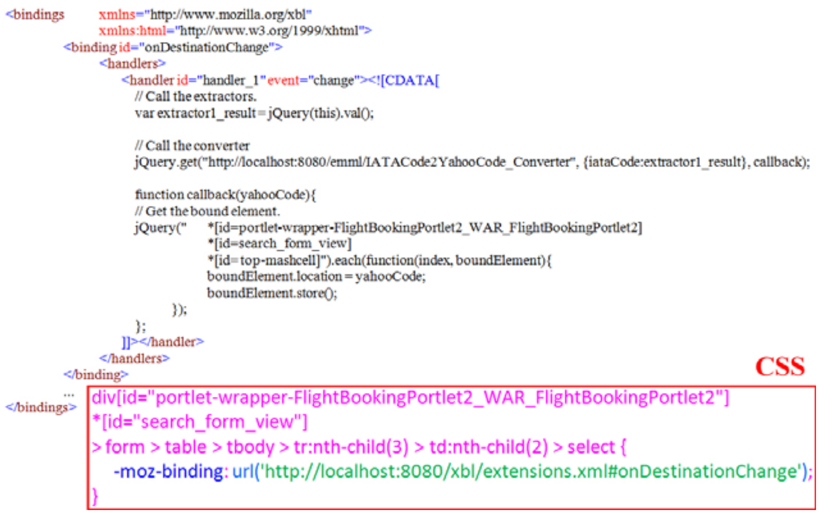
## 5 Setting Composition Coordinates

A composition coordinate *(what, where)* specifies *what* widget is to be place in *which* mashcell. *XBL* wise, the binding provides the *what* as a *<content>* element, while the *CSS* provides the *bound element*, i.e. the *where*. Figure 4 provides an example for the coordinate *(WeatherForecastGadget, top-mashcell)*.

**What.** The binding *gadget_21* describes this widget as anonymous content. Widget preferences are supported as properties. In this case, getter and setter methods are provided for the two widget preferences: *location* and *units*. This file is kept at *"localhost:8080/xbl/wrappers.xml"*.

**Where**. Widgets are placed in mashcells. The *CSS* file bounds the *gadget_21* binding to the "*top-mashcell*" to be found in the markup of the *flightBooking* portlet. This markup is shown in Figure 2: the *"top-mashcell"* div pertains to the *"search_form_view"* (notice that a portlet can deliver a set of markups) which in turn is wrapped by a portal decorator (a.k.a. portlet wrappers)[5]. The *CSS* addresses the whole portal page; hence the *CSS* style must identify a single cell within the portal page. This explains the three *ID* conditions in a raw found in the *CSS* sample.

[5] The latter identification is needed since it is possible for the same portlet to be instantiated more than once in the very same portal page. That is, they can be distinct *flightBooking portlet* instances in the very same portal page.



```
<bindings       xmlns="http://www.mozilla.org/xbl"
                xmlns:html="http://www.w3.org/1999/xhtml">
        <binding id="onDestinationChange">
                <handlers>
                    <handler id="handler_1" event="change"><![CDATA[
                      // Call the extractors.
                      var extractor1_result = jQuery(this).val();

                      // Call the converter
                      jQuery.get("http://localhost:8080/emml/IATACode2YahooCode_Converter", {iataCode:extractor1_result}, callback);

                      function callback(yahooCode){
                      // Get the bound element.
                      jQuery("      *[id=portlet-wrapper-FlightBookingPortlet2_WAR_FlightBookingPortlet2]
                                    *[id=search_form_view]
                                    *[id=top-mashcell]").each(function(index, boundElement){
                                    boundElement.location = yahooCode;
                                    boundElement.store();
                              });
                      };
                    ]]></handler>
                </handlers>
        </binding>
        ...
</bindings>
```

```
div[id="portlet-wrapper-FlightBookingPortlet2_WAR_FlightBookingPortlet2"]
*[id="search_form_view"]
> form > table > tbody > tr:nth-child(3) > td:nth-child(2) > select {
   -moz-binding: url('http://localhost:8080/xbl/extensions.xml#onDestinationChange');
}
```

**Fig. 5.** *XBL* support for the orchestration coordinate *(top-mashcell, destination)*

## 6 Setting Orchestration Coordinates

An orchestration coordinate *(where, how)* specifies how data flows from the portlet to the widget. For instance, the *weatherForecast's location* parameter is to be obtained through subscription to changes in the *destination* form field of the *flightBooking* portlet. This stands for the orchestration coordinate *(top-mashcell, destination)*. Figure 5 illustrates its XBL realization.

**How**. The binding *onDestinationChange* specifies how to extract the desired data (e.g. through a handler on the *DOM* event *"change"*). The handler proceeds along three steps: (1) calling the data extractors, (2) enacting the converters for data transformation, if required, and finally (3), localize the *bound element* where the widget resides. Data conversion is needed since data formats can differ between the provider and the consumer of data. In this case, a converter is used to map *IATA* airport identification used by the portlet to *Yahoo!* city code utilized in the gadget.

**Where**. The *CSS* file associates the previous *onDestinationChange* binding to the form field that collects the *destination* airport. The *CSS* locates the *bound element* by first identifying the portlet decorator, next the portlet view, and finally the DOM node that holds the data. At enactment time, this binding will cause changed in *destination* to be propagated to the *weatherForecast* markup.

## 7 Related Work

For the purpose of this work, mashup approaches can be classified based on the role taken by the source application: passive ("unaware mashuping") versus active ("aware mashuping"). Unaware mashuping is illustrated by *MashMaker* [4] and *MARGMASH* [3]. Both tools allow end users to augment the content of an *existing* Web application by inserting mashups' markup throughout the source application's pages. The source application is completely unaware of being mashup so that screen-scraping techniques are used "to glue" the mashup to the existing code. Browser plugins can be available to facilitate such data extraction. By contrast, corporate portals offer a more controlled environment for mashuping. Here, the applications to be mashuped (i.e. the portlets) are already integrated into the portal. Such integration implies that portlet providers need first to be certified, and this agreement can now include the existence of mashup placeholders that facilitate mashuping from portal users. The portal ensures access control, presentation guidelines, and data extraction. No plugin is required. The mashup is set through the portal as an additional utility.

## 8 Conclusions

Corporate portals achieve front-end integration for presentation-oriented Web Services (a.k.a. portlets). As any other Web application, portlets can also be subject to mashup. However, their special characteristics (i.e. reusable components being offered by third parties) make portlet mashup a combined endeavour of portlet consumers and portlet providers. We have presented an architecture where portlet providers facilitate mashup placeholders (i.e. *mashcells)* to add companion widgets. As for portlet consumers, *XBL* bindings are used to dynamically bound user mashups to *mashcells*. The approach strives to find a balance between portal reliability and mashup freedom. This architecture is borne out for *Liferay*. Next follow-on includes capitalizing on the portal utilities for leveraging mashup. Single-sign on, access control, customization mechanisms are now at our disposal to adapt mashup techniques when achieved through a corporate portal.

**Acknowledgments.** This work is co-supported by the Spanish Ministry of Education, and the European Social Fund under contract TIN2008-06507-C02-01/TIN (MODELINE). Perez enjoys a doctoral grant from the Basque Government under the "Researchers Training Program".

## References

1. JSR 286: Portlet Specification Version 2.0, `http://jcp.org/en/jsr/summary?id=286`
2. Díaz, O., Rodríguez, J.J.: Portlets as Web Components: an Introduction. Journal of Universal Computer Science, 454–472 (2004)
3. Díaz, O., Pérez, S., Paz, I.: Providing Personalized Mashups Within the Context of Existing Web Applications. In: Benatallah, B., Casati, F., Georgakopoulos, D., Bartolini, C., Sadiq, W., Godart, C. (eds.) WISE 2007. LNCS, vol. 4831, pp. 493–502. Springer, Heidelberg (2007)
4. Ennals, R.J., Garofalakis, M.N.: MashMaker: mashups for the masses. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (2007)
5. Rhodes, B.J.: Margin Notes: Building a Contextually Aware Associative Memory. In: Proceedings of the International Conference on Intelligent User Interfaces, IUI 2000 (2000)
6. W3C. Document Object Model (DOM) Level 2 Events Specification, `http://www.w3.org/TR/DOM-Level-2-Events/`
7. W3C. Widgets Family of Specifications, `http://www.w3.org/2008/webapps/wiki/WidgetSpecs`
8. W3C. XML Binding Language (XBL) 2.0 (2007), `http://www.w3.org/TR/xbl/`