Distributing education services to personal and institutional systems using Widgets

Scott Wilson, Paul Sharples, and Dai Griffiths

University of Bolton

Abstract. One of the issues for the Personal Learning Environment is the integration of personal tools with institution-centric information and services such as timetable information, and the coordination of cohort-specific group activities. One potential solution is the use of widgets – small, single-function applications that can be used in different applications, including personal technologies. One of the challenges facing the use of widgets is the diversity of platforms, and the issues around security, privacy and control. This paper describes an approach developed as part of the EU-funded TenCompetence project to develop a system based on open standards for enabling widgets to be used in a range of personal and institutional systems.

Introduction

One of the key issues in distributed e-learning is how to enable coordination to take place across a diverse range of personal as well as institutionally-managed applications in the context of formal learning. One of the key mechanisms that has been used to date has been the sharing of RSS feeds; another approach has been the use of widgets such as Facebook applications to provide institutional information in learner-managed spaces. Widgets are useful where it makes sense to offer a user interface for a service rather than a feed, or where making such a user interface available in addition to a machine-useable web API can lower the barrier to use. Examples of useful widgets within a distributed learning context include access to institutional services such as timetabling, support services, and libraries, and also widgets that enable access to shared activities with a cohort focus, such as shared chats, voting, and forums. Examples to date include the use of Facebook applications by the Open University (Hirst, 2008) and the use of OpenSocial at the University of Cambridge (Boston, 2008).

Currently, many applications provide their own plugin mechanisms to enable third-party widgets to be incorporated by users; these include learning management systems (Moodle, Blackboard), personal blogging systems (Wordpress), social networking sites (Facebook, Elgg, Ning), and the operating system itself (Apple Dashboard, Windows Sidebar). However, each system has a different API, and widgets must be developed for each one using the native programming platform of the system.

To overcome this issue a number of initiatives to standardize widget platforms have emerged. Google Gadgets and OpenSocial is one effort to create a single widget platform. The Google Gadget platform consists of a very wide range of

interconnected javascript APIs coupled with REST services to manage and deploy widgets on different social network platforms. However, the actual implementation of the platform is very complex with a lot of internal dependencies, and so an open source server solution has been created, Apache Shindig[1], to lower the barrier to entry. Although it has many good features, the Google solution remains a proprietary solution, and there has been little engagement by Google in open standards in this area.

Another initiative is the W3C's Widget specification initiative[2]. This is an open-standards approach being developed to harmonize widgets between the Apple, Microsoft, Yahoo, Nokia and Opera platforms. While focused primarily on desktop-style widget engines, the specifications cover many of the core concerns of widget development for the web, including packaging, deployment, description, and access to APIs.

In either case, a widget is typically defined as a portable application, typically packaged in a Zip archive, and implemented using HTML, Javascript and CSS, with some deployment metadata for use by the container such as height, width, title and author information. The widget's Javascript code needs to be written to make calls to a set of standard APIs for making use of the services offered by the container. However, the two approaches differ in the package, metadata, and most importantly the API specification.

This created a dilemma for our development team, which was primarily tasked with solving issues of including tools in learning activities for the TenCompetence project[3]. In the end we decided to adopt the W3C approach, and extend the W3C specification to handle web-deployed widgets with collaboration features, rather than adopt the proprietary Google Gadgets platform. We created an open-source engine for these widgets and reference plugins for several container platforms.

For the future we hope to see some convergence between Google and W3C; alternatively we may work to mitigate the division between the approaches by working with the Apache Shindig project to support W3C specifications.

Architecture

The W3C model for widgets requires the use of a widget engine to provide the services for widgets such as persistence and external web access; in the case of web applications this engine needs also to communicate with the web application that is acting as the widget container. The components of the architecture we designed for our solution are shown in Figure 1.

[1] http://incubator.apache.org/shindig/
[2] http://www.w3.org/TR/widgets/
[3] http://www.tencompetence.org

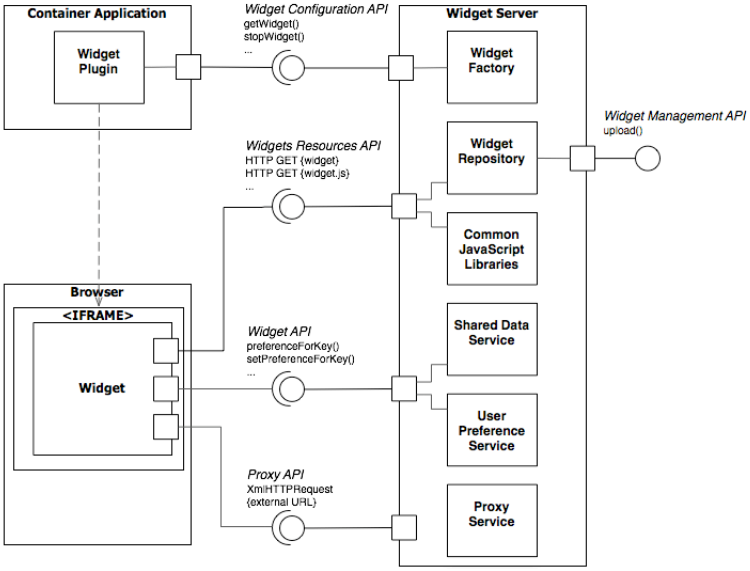

Fig. 1. Architecture of the system

Container Application

The architecture supports a wide range of potential *container applications*. Each container application must support the Widget Configuration API, and be capable of rendering a view that loads the resulting URL in a container (such as an IFrame), preferably one conforming to the dimension hints supplied in the getWidget() response.

The *widget plugin* enables a widget to be placed within the context of the container application, and is responsible for implementing the widget configuration API. The widget plugin collects the relevant configuration information via the container application's authoring system, and instantiates a widget using the getWidget() method.

Browser

Within the browser, the view rendered by the container application will typically be an IFrame that displays the widget content. The widget content is loaded from the Widget Server with the URL supplied to the plugin.

The widget is typically a small HTML file with a number of included JavaScript libraries; these include common JavaScript libraries offered by the widget server to enable widgets to communicate with the *Widget API*. Each instance of a widget (that is, a specific widget instantiated by the plugin and then displayed for a particular user) has its own unique key used in conjunction with the *Widget API* to store and retrieve

user preference information, and to access data shared across multiple instances (for example, across all chat widgets within a single course context). Widgets often require access to external services, for example to make AJAX requests for RSS feeds or to access external content; this needs to be routed via the Widget Server's *Proxy API* to avoid violating the same origin policy of the browser.

Widget Server

The Widget Server is a standalone server application that can support multiple container applications by managing and distributing widgets and offering supporting services including persistence and a URL proxy for cross-site requests.

The **Widget Factory** is responsible for instantiating and managing instances of Widgets. The Widget Factory offers the Widget Configuration API; this is used by Container Applications to instantiate widgets and to obtain configuration information including the title, URL, height and width of a widget instance.

The **Widget Repository** is responsible for managing and distributing the assets of widgets, including their HTML, CSS, images and JavaScript files. The Widget Repository offers the Widget Management API, which is used to upload and install new Widgets, and the Widget Resources API, which is simply enabling access to assets using HTTP GET.

The **Common JavaScript Libraries** are a standard set of support libraries that Widgets use to communicate with the Widget API and to enable callbacks on events. These are accessed using the Widget Resources API for access using HTTP GET.

The **Shared Data Service** persists and returns information that is shared across multiple widget instances. For example, the content of a chat conducted between several chat widget instances. The Shared Data Services is exposed using the Widget API and typically accessed via a common JavaScript library included in each widget. As well as conventional access, the Shared Data Service is configured to support Reverse AJAX, also known as Comet[4]; in this model, updates are pushed to subscribing widget instances using rapid polling. This supports widget applications such as instant messaging or online voting without the need to set up externally hosted services.

The **User Preference Service** persists and returns information persisted for a single widget instance, such as user preference settings and any other data that is unique to an individual instance of a widget. The User Preference Service is exposed using the Widget API and typically accessed via a common JavaScript library included in each widget.

The **Proxy Service** executes AJAX requests on behalf of widgets. This circumvents Same Origin Policy restrictions in the browser environment. The Proxy Service can be configured using whitelist or blacklists and should be secured for access only via widgets served by the Widget Server.

The **Widget Configuration API** is invoked by container applications to instantiate, stop, and resume widgets. The main method offered by the API is getWidget() which is called by the container with context information for the widget,

[4] see e.g. http://en.wikipedia.org/wiki/Comet_(programming)

and returns a URL generated by the server for the container to render, and the height and width in pixels of the widget instance.

The container context consists of the application identifier, the context identifier (for example, a course id) and a user handle. The latter needs to be unique, but can be entirely opaque to the widget server rather than an externally-referenced user identifier. The service then uses a nonce and MD5 hashing algorithm to return an opaque widget reference number to prevent unauthorised access to widget instances.

For example, a call to getWidget() may return a response such as:

```
<widgetdata>
<url>http://localhost:8080/wookie/wservices/www.tencomp
etence.org/widgets/WP3/natter/chat.htm?idkey=xR8OG1IFX5
8z/YVvlz910PQVtv8.eq.&amp;url=http://localhost:8080/woo
kie/dwr/interface/widget.js&amp;proxy=http://localhost:
8080/wookie/proxy</url>
<height>383</height>
<width>255</width>
<maximize>false</maximize>
</widgetdata>
```

The **Widget Resources API** enables access to the repository of widget assets, and can be implemented as standard web resource access.

The **Widget API** is the service that is invoked by widgets via the interface offered by a javascript Widget object called from within the Widget javascript code. The API provides methods for accessing the user preference service and shared data service. This API is based upon the W3C Widget Services and Events specification.

The **Proxy API** provides a mechanism for widgets to request external URLs without breaking the Same Origin Policy. For example, a typical AJAX request for an external API or RSS feed needs to be routed through the widget server Proxy API to prevent potential cross-site scripting vulnerabilities.

Implementation

Wookie server

We created an implementation of the widget server, which we call *Wookie*. Wookie is implemented as a standalone Java servlet application with a MySQL backend database. The application provides all the services identified in the architecture as simple REST-style calls, and provides an administration interface for uploading, tagging and deploying widgets. It also implements a whitelist function for the Proxy

API, which the administrator can configure. For the implementation of Comet functionality we used the Direct Web Remoting (DWR) open-source Java libraries[5].

Wordpress plugin

One container application we tested Wookie with is Wordpress. Wordpress already offers its own specific PHP-based Widget API, and it was very simple to extend this to enable Wordpress to call the Widget Configuration API and render a widget in its sidebar. This means that the widgets served by Wookie can behave in the same manner as native widgets from the viewpoint of a Wordpress user. An example is shown in Figure 2.

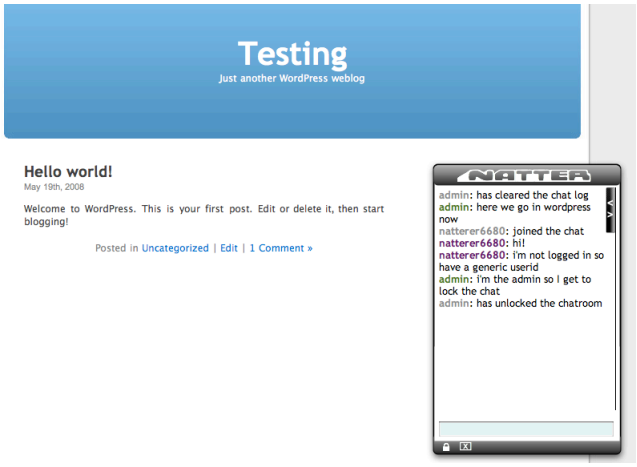

Fig. 2. Wookie widget running in Wordpress.

Moodle plugin

The second container application we developed a plugin for was the Moodle learning management system[6]. Like Wordpress, Moodle also has its own API for extensions, which Moodle calls *blocks*. The implementation of Widgets for Moodle created a new simple block type with a single configuration element for the widget type. This enables users to add Wookie widgets that then behave in the same manner as other Moodle blocks, and can be moved around the course layout as desired. An example is shown in Figure 3. Note that the "moon" and "chords" blocks are Wookie widgets

[5] http://directwebremoting.org/dwr/overview/dwr
[6] http://www.moodle.com

that have been converted from Apple Dashboard format to W3C format. The “natter” block is a Wookie widget we developed that uses the Shared Data Service.

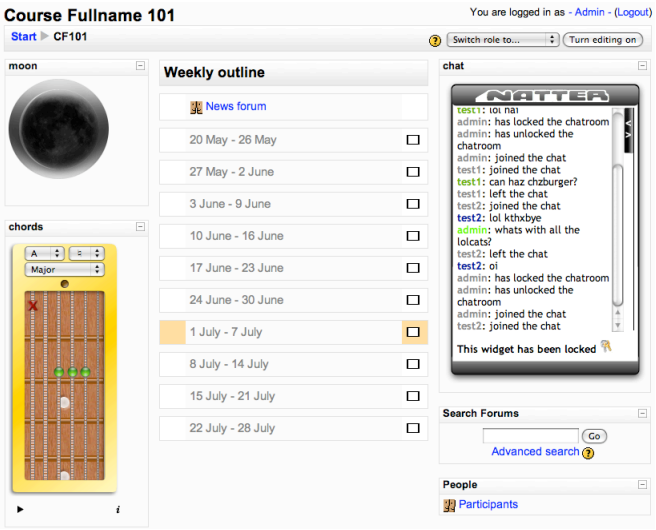

Fig. 3. Wookie widgets running in Moodle.

Collaboration Widgets

To demonstrate the capabilities of the shared data service, we created a number of collaboration widgets, including Chat, Forum, and Voting. These correspond to commonly requested features for learning designs.

Discussion

In this work we have been able to demonstrate the feasibility of extending the W3C open specification for widgets to the use of web applications, and to extend the functionality of widgets to the implementation of collaboration tools commonly employed in learning applications. By enabling such tools to be distributed in a range of containers, including personal web applications such as blogs, the potential exists to exploit this capability for offering more flexibility in the provision of e-learning.

For example, a course-cohort chat widget may be offered both through the LMS and externally through the applications in a student’s own web-based PLE. We also see a strong potential role for Wookie in enabling institutional services to be embedded in both the LMS and PLE, such as timetable, support, and tutor messaging services. We see this as a pragmatic means of enabling the co-existence of PLE and LMS approaches, easing the transition from a provision-centric model to a coordination-centric model while the capabilities of the LMS and PLE converge;

without such a convergence a shift by institutions to supporting PLE users is unlikely due to the significant differences in functionality as defined by the dominant applications (Wilson et al., 2007).

Widgets also offer an easier transition for some types of institutionally managed services to be offered through personal systems; partly at least as the benefits can be expressed in terms of institutional variety management. For example, widgets can offer a relatively low-cost method for exposing services across different institutional systems such as library, MIS portal, intranet, web content management system, and LMS.

The crucial component for the future, however, will be the integration of user-centric authorization within widget architecture. The oAuth[7] protocol offers a means for sites to establish authorization to use APIs without the transmission of user credentials. This would enable users to authorize a widget within their PLE to access data held by an institution without compromising the identity management of the institution. Currently there is considerable effort underway in the oAuth community to support widget applications, and this convergence will have a significant effect on the range of services that can be offered using widget technologies.

Future Work

For the future we intend to continue to align the architecture and implementation with the W3C specifications as they evolve towards final status, and to look into additional capabilities, such as supporting Google OpenSocial in some fashion, or integration with the Apache Shindig project, and possibly also to integrate oAuth capability. We will also continue work on supporting the use of widgets for learning designs. In the short term we intend to continue to trial the system in other contexts in order to evaluate its effectiveness, performance and usability; currently we are working with two UK-based projects involving timetable information and tutor-student messaging, and are investigating potential collaboration with several projects in other countries.

References

Boston, Ian. (2008). Sakai and OpenSocial: A Different Approach to Distributed Learning Applications. *e-Literate*. Retrieved July 4th, 2008, from http://www.mfeldstein.com/sakai-and-opensocial-a-different-approach-to-distributed-learning-applications/

Hirst, Tony. (2008). Open University Course Profiles Facebook App. *OUseful Info*. Retrieved July 4th, 2008, from http://blogs.open.ac.uk/Maths/ajh59/010855.html

Wilson, S., Liber, O., Beauvoir, P., Milligan, C., Johnson, M., & Sharples, P. (2007) Personal Learning Environments: Challenging the dominant design of educational systems. **Journal of e-Learning and Knowledge Society** (2). Giunti: Genoa.

[7] http://oauth.net