Helpers and Gadgets within Sakai

New Concept

- This is an idea for discussion, please feel free to edit, comment, etc.

Why is this a good idea?

During the Newport Beach Sakai Conference (Dec 07) it became apparent to me (JRN) that we are on the verge of being able to offer a different sort of Sakai experience. People were asking could we do 'X' or 'Y' and we were hearing the answer 'Yes' and getting excited. So what was 'X' and 'Y' and how could we deliver that vision?

A take on the 'X' and 'Y' Vision:

An environment in which Sakai Services would provide functional fragments into pages with appropriate permissions for appropriate roles and contexts. Together with a page composition tool based on the concepts of the drag and drop reorderer/lightbox tool being developed as part of Fluid. The page composition tool can be manipulated by the site admin (optionally within template controls for site to site consistency), or the instructor, or the student in a configurable way (in other words page edit rights are configurable). The OOTB Saki experience would then be merely a set of default templates for how the page fragments might commonly be assembled into functional course sites or project sites. A Sakai 'site type' would be a set of templates.

One set of pages would be profile information about the individual student or academic and would certainly include 'professional /academic profile' as a primary communication goal. Integration points with Social Software would also be valuable. Grade summaries, submission deadlines (for students) and progress towards personal goals might also be here, but the key point is that the institution would be able configure pages to pull in information from Sakai Services to meet local requirements.

Another set of pages could be student portfolios - in the sense of portfolio as a tool of educational development - in which the student composes a series of pages, or page collections, that are shared with
different audiences.

Another set of pages would be the course sequence; expressed in terms of time, course themes or other organisations of knowledge /work. In this set of pages, assignments, resources, timetable information, etc. would be embedded in the page and reveal key performance indicators such as the status of an assignment (how many submitted/graded for the instructor, countdown to submission date or number of peers submitting for the student)

Another set of pages would be for the research group, expressing and tracking the data collection workflow, measuring dissemination (connected to individual profiles), etc.

And another set of pages would be for the club or society - shared widely with the institution or general public.

Navigating the page sets would be conceptually similar to navigating web sites, but would be much better at offering flexible access to different audience groups.

The concept as I imagine it has a lot of similarity with the Wikipedia description of Mashups Here is the key paragraph:

> *It is possible to replicate the functionality of a JSR 168 portal entirely using mashup technology, but many mashup capabilities cannot be replicated using portal technology. However, the portal model has been around longer and has seen greater investment and product research. The technology is therefore more standardised and mature. In 2-3 years, increasing maturity and standardisation of mashup technology may propel it beyond portal technology in popularity. Alternatively, the two technologies may converge, with newer versions of current portal products resembling mashup servers in terms of flexibility while still supporting legacy portlet applications.*

It it the convergence scenario that seems most attractive for Sakai.

As one of those who said 'yes' a lot at Newport Beach, Antranig has set out some ideas about how such a vision could be realised:

Helpers and "Gadgets"

This page describes requirements and proposals for a related set of functionality that we desire to support in Sakai, for the embedding and aggregation of user interface elements, both across Sakai itself and outside it. Many of the strategies have technical relationships that do not correspond to user-visible features, and correspondingly features which appear similar to users could be implemented by a variety of schemes. To bring some order to this space, this page will first try to itemise various use cases and their value, before considering the merits of specific technical solutions.

These elements are variously called, Helpers, Gadgets, Widgets, Page Fragments, Components, etc. and this document tries to bring their functionality under a common set of requirements and terminology. It begins with setting the context with historical helpers.

Some early discussion on these issues took place at the Newport Beach conference under the head of "cross-tool integration"

Historical Helpers in Sakai

The first case to consider is the historical role of helpers in Sakai, which will also help to clean up terminology. A "helper" is a Sakai-specific concept, which from the user's point of view involves being taken from a link in one "tool" to a sequence one or more pages taken from a different "tool", which completely replace the browser pane or frame until a subtask is completed, at the end of which the user is returned to the original tool. This is supported by a quite complex API within Sakai that in theory is capable of a more contextual aggregation that will be mentioned below - but this system has never been fully standardised in terms of its usage of URL space and session state, and does not naturally generalise to aggregation of content outside Sakai. Later on in this discussion, the historical helpers system will be placed in the more general context of server-side aggregation, as opposed to the more client-driven approach we will be looking for.

A section of the Sakai (2.2.x vintage) documentation on "Sakai Tools" explains some of the requirements on a Helper, which is reproduced here in Historical Helpers.

Distinctions between helpers and gadgets/widgets - user experience and technology

In terms of technology support, the issues raised by helpers and widgets/fragments are very similar. There is a difference in user experience, typically in favour of widgets, since in many cases the user is saved an unnecessary context switch. However, for some bulky/lengthy workflows, the traditional "helper" experience may well still be appropriate. One important technological distinction is that a helper has a well-defined "end" - a point in its workflow where navigation is transferred back to a page controlled wholly by the host tool. For a gadget/widget, there may be no well-defined endpoint to the workflow - whilst some widgets may provide an affordance to be explicitly dismissed, many may simply be "dismissed" by user navigation away from the hosting page. For this reason it is important that widgets be lightweight or ideally zero in terms of their allocation of server (session) resources lest these resources accumulate on the server during navigation.

Changing industry focus towards the client side

Server-side aggregation, with examples such as JSR-168 portlets and Sakai helpers, has been the dominant strategy in the industry, largely since client-side facilities were either immature or underappreciated. Also, accessibility considerations until recently have recommended that user interfaces avoid uses of AJAX techniques, and be able to function with Javascript turned off. However, this is no longer a strong recommendation, and many AJAX-based techniques are considered increasingly accessible as well as reliable. Therefore a focus of the requirements in this document over more historical techniques will be "taming" and codifying what we expect of client-side aggregation, in itself, as well as its relation to server-side aggregation.

Requirements

Detailed requirements following in this section. These are a mixture of user, deployer and developer-focused requirements, with no attempt here at prioritisation or assignment of a specific solution strategy.

1a. Inclusion of simple "gadgets" on the client-side, both across Sakai and from outside it.

Webapps written in Sakai could include short "pages" which are suitable to be embedded in other webapps. Similarly, "mashup"-like services deployed around the 'net (usually from high-resourced and stable providers such as Google, Facebook etc.) may export "gadgets" suitable for embedding. These exports may also come from more local providers "on campus" such as SIS systems, content repositories, other CLEs, etc.

1b. Inclusion of page fragments which behave as an autonomous URL environment

Links and forms (<a> and <form>) etc., within the fragment or "gadget" environment should preserve their local semantics - that is, they should cause navigation within the gadget rather than causing top-level navigation. This behaviour should be transparent to the code implementing the gadget - it should not have to behave differently whether it is hosted at top level or embedded in another page.

1c. Inclusion of more complex widgets, themselves involving Javascript or including further widgets

Widgets may themselves have their own lifecycle or inclusion structure which needs to be respected. As pages grow more complex, and more dynamic widgets (e.g. date widgets, drag and drop controls, etc.) are used on pages, more care is needed to ensure that page initialisation is both reliable and straightforward.

2a. Providing server-side aggregation as a direct and transparent replacement for an instance of client-side aggregation

Many or indeed all of the "widgets" on a page may actually be hosted from the same server. In this case it provides a better user experience as well as enormously reducing server load, to allow the aggregation operation to happen on the server for the initial page load, even if the "widgets" continue after this to be AJAX-driven and refresh themselves individually. This shifting of the burden to the server should be as straightforward as possible to the developer, and should provide essentially the same rendered markup as if the widgets had been fetched individually. Note that this is distinct from a JSR-168 style server aggregation which produces "flat" markup with links rewritten to target the top-level page.

2b. Providing an option to degrade to flat (JSR-168 style) markup for environments where Javascript is not recommended/turned off

As a further increase in flexibility over the previous requirement, to provide a straightforward option to developers to allow server-side aggregation to drop back to a flat markup style, and to render widgets and pages using a traditional JSR-168 portal style.

3. Provide a "clean", reliable and stable URL space for addressing page fragments for inclusion both within Sakai and elsewhere

Requirements for "cool", "clean" or "short" URLs have already been presented as top-level requirements for Sakai as a whole at Sakai Technical Goals. Exactly the same requirements are placed on helpers, which suggests that hosting them within the existing Sakai tool /webapp structure may be inadequate. In addition, we require these URLs to be stable with respect to factorisation of the underlying code and module structures by developers. A particular "resource" or fragment should continue to be available at a particular fixed and simple URL even as functionality is reallocated between webapps.

This URL space should also make it easy to apply a relevant authentication policy for the gadgets - that is, to switch between a completely unauthenticated model suitable for public access widgets, or to lock down access to particular helpers only to users with a particular role within a certain scope (whether this scope is a traditional Sakai Site or some other suitable domain).

4a. Specify a straightforward and reliable scheme for passing information between clients and implementors of gadgets/helpers

Where possible, information passed out to a helper should be contained in the URL. However, some information is too bulky to be sent in this way, and information may need to be returned back from the helper to the caller. In these cases a use of session state is probably unavoidable. A protocol needs to be drawn up whereby multiple instances of the same widget on behalf of the same user do not collide in their storage.

4b. Specify a preferred architecture for coordinated form submissions across clients and implementors of gadgets/helpers

An advanced requirement which arose during the December 2007 Newport conference (cross-tool integration) was handling the case where a form submission is intended to span material from both a client and a helper. The paradigmatic example is an Assignment, which must include a UI section rendering a "Gradable" item, which would be hosted from a helper contained within the Gradebook. This creates issues not only at the level of markup, and responsibility for controls, but also at the service level, where "partial" failures of submission need to be handled robustly, providing relevant user feedback and not corrupting the state of the interface.

Relating technologies to requirements

In this section we will consider various technologies available in Sakai, and the extent to which they address the above requirements. (Initial version of this document was prepared by Antranig Basman and will naturally concentrate on the technologies I am experienced with - please flesh out this section as well as requirements above based on your own knowledge)

Markup-oriented technologies and conventions

Requirement 1a. is very easy to meet with only one or two lines of Javascript, and this scheme can be seen widely used around the web. To the extent that Sakai exports simple markup blocks that can participate in this scheme, it can enter the "general democracy" of mashups and gadgets that are now growing up. For example, the following page on AHAH microformats shows two simple Javascript functions `ahah` and `ahahDone` that are suitable for embedding markup. This is suitable for code written in any server-side presentation technology. Note that this page also contains a block `execJS` which is a partial answer to requirement 1c.

Requirement 1b. is also tolerably easy to meet, with only a little extra work. The standard RSF sample "RSFAjaxAHAH" contains a Javascript file ahah.js which performs the work of fetching and decorating a markup block via AHAH in such a way that every link and form submission is operated by an AJAX/AHAH fetch of the block itself rather than the top-level page. This code is not dependent on the use of any particular client-side or server-side technology and so is generally applicable to this requirement - however it does depend on the standard RSF client-side file `rsf.js`. This dependency could be removed by refactoring tolerably easily.

Requirement 1c. requires much more widespread agreement on markup standards, as well as more involved client-side Javascript. The most important issue is cooperation on the semantics of Javascript initialisation, which in many client-side frameworks is performed by a form of contribution to a document "onload". Without mandating a single client-side framework throughout Sakai, the use of "onloads" fights against this requirement - a more suitable approach is to use "init blocks" embedded within the markup itself, which is the approach expected by the AHAH microformats page referenced above, as well as the approach explained for building RSF components on the RSF wiki.

However the use of init blocks is not sufficient in itself to meet 1c. Any external JS files referenced by code within the AJAX block will themselves need to be fetched and parsed. The only really practicable method is for these themselves to be written into the AHAH markup via an embedded `<script>` block, this time held at the head of the block, rather than at the tail. However, for efficient and safe processing, these blocks would need to be guarded against multiple inclusion. The Dojo framework features one solution to this problem `dojo.require`, `dojo.provide`, but is coupled to the use of the Dojo framework itself, which would be an unpopular choice to mandate across all of Sakai. To meet this in a portable way across Sakai would probably be one of the upcoming goals of the Fluid Project.

Server-side technologies and conventions

A considerable portion of server-side support relates to the provision of an appropriate URL space for helpers/gadgets, together with the appropriate semantics. The current options for URLs within Sakai are documented at URLs within Sakai. In terms of helper/gadget support, the relevant requirements are that the URLs be (considering requirement 3. above)

- "Cleanliness" - URLs should be no longer than necessary to specify the information needed to locate them, and should be as readable as possible
- Abstraction - URL should not couple to a particular physical deployment location or logical build unit within Sakai
- Portability - URL space should not bias towards the use of particular server-side technologies or frameworks
- RESTful semantics - URLs should, in isolation, satisfy the normal browser semantics for GET/POST/PUT etc. of a resource held at that location
- Appropriate authorisation - The URL space should not interfere with the ability to easily apply an appropriate authorisation policy, whether public access, or fine-grained authorisation on a per-resource basis.

Sakai's "out of the box" tool URLs do not deliver very well on most of these points (see URLs within Sakai). Tool URLs, whilst portable, are longer than necessary through including a tool placement PID, and are coupled to a tool location. Also, tool URLs are typically authorised to a particular site/context which is a decision hard to un-bias. RESTful semantics are up to the implementor, but historically many tools have been developed in Sakai without regard to these.

A few schemes have more recently grown up in Sakai to address this issue, of which we will consider two [Add explanation here for any others you are aware of].

Entity Broker

The Entity Broker is a refinement of the historical Sakai entity system. It provides a scheme whereby each entity in Sakai may be assigned a well-defined top-level URL space, which is available for dispatch both internally and externally. EntityBroker URLs deliver well on the 5 points above - they are short and abstract (externally beginning /direct/prefix/id), can be implemented in any presentation technology with as RESTful semantics as is desired by the user (will service all HTTP methods) and do not bias any authorisation decisions - these are performed within the implementing code.

As well as the desirable URL space properties, packaging "gadgets" or helpers as EntityBroker-dispatched units delivers on several of the goals mentioned above. For requirement 2a., since the dispatch interface is available at top-level within Sakai, it is easy to perform a given helper dispatch on the server-side, even as a result of an on-the-fly decision, providing the same markup and semantics as would be delivered via a client-side solution. Similarly, should requirement 2b. be considered desirable, a similar dispatch could be performed to an "entity world" as a JSR-168 URL environment producing flat markup rather than Javascript-driven markup [this functionality not yet implemented].

Entity Broker URL spaces (helpers) also have a natural "structural" interpretation within Sakai, since Entities are the fundamental structuring unit of Sakai's internal data and event model. Thus as well as being "user-facing gadgets" the markup units that are rendered can be direct representations of elements of Sakai's user-level model, such as assignments, evaluations, gradebook items, etc.

Portal Handlers

The Portal infrastructure in Sakai 2.4 has been refactored to allow individual Portal Handlers to take on responsibility for sections of the top level /portal URL space. These also deliver on a good many of the URL-oriented requirements for requirement 3. - for example, URLs can be short and meaningful, the handlers may be placed in any deployed webapp, and authorisation may be appropriately delegated.

A PortalHandler gains full control over request dispatching for GET and POST requests that it expresses control over. PortalHandlers may be registered by either a fixed "prefix" (an initial URL segemtn) or may also form part of a "chaining" model whereby a number of handlers are queried for a particular request until one returns that it is responsible.

A specialised toolkit with special status within the portal is a PortalRenderEngine, a lightweight "render-only" which uses a "Spring MVC" -like idiom to produce markup from a PortalRenderContext, essentially a container for a structured HashMap. This model implies very rapid rendering times for views which may be rendered extremely frequently (part of the portal itself or its top-level views) - its current implementation is used from the SkinnableCharonPortal to render portal panels using a Velocity implementation. Only "request-oriented" frameworks such as JSPs, Velocity or IKAT (RSF renderer) are suitable for implementations of a PortalRenderEngine, full-cycle frameworks such as JSF, Wicket or full RSF may not be used. Uses of a PortalRenderEngine is an option for PortalHandlers, it is not required. The PortalHandler is currently limited in its REST idiom to servicing HTTP GET and POST requests.

Portal Handlers make requirements 2 harder to meet - dispatch is performed internally within the portal and cannot be performed on demand by arbitrary tool code. They are therefore most suitable for lightweight, rapid development of portal-level functionality. It might be easy to extend the dispatch model of the portal so that either the space of handlers can be mapped as singleton Entities, or else to meet some of the above requirements directly.

Information passing to and from helpers

Requirements 4a. and 4b. together present challenges that do not naturally relate to a particular choice of client-side or server-side technology, but are cross-cutting concerns for Sakai architecture and helpers. As stated in the requirements, 4a. should be minimised by good use of a RESTful URL space, but in more complex cases of submission of form data against a persistent data model, some data will naturally have to leak out into storage in a more or less persistent form.

In traditional web application technology, the most appropriate form of this semi-persistent data is within the HTTP Session. 4a and part of 4b amount to drawing up a suitable protocol for safe and reliable use of Session state, in a way that does not bias client- or server-side code into a possibly colliding or inconsistent use of state.

Coordinating session usage for multiple helper activation (Requirement 4a.)

The traditional Sakai helper model relies on particular, "well-known" keys within the Sakai "Tool Session" - for example `Tool.HELPER_DONE_URL` is a well-known key representing the URL to which the helper should issue a redirect when its work is finished. This implies to start with that the dispatch has already been localised to a particular Sakai tool. Secondly, it presupposes that a single instance of a particular helper is active at a time, and in some cases, that a single helper is active at all.

A proposal to generalise this use of session state is to adopt a parallel scheme, making use of the global HTTP Session (or a well-known one, for example `"sakai"`), and to allocate EL-addressible "scopes" within this session organised on a per-helper (gadget) basis. Underneath each of these scope keys will be sub-keys corresponding to GUIDs of some kind allocated per instance of the helper. For example, within the global session, the top-level key "assignment" would access an object (probably a java.util.Map) which when traversed for a particular GUID key, would return a Java Object holding the state for a particular invocation of the assignment helper or gadget. If the gadget is being dispatched via the EntityBroker or PortalHandler system, a good convention is to make this top-level key agree with the entity prefix or handler prefix.

The space of GUIDs should if possible be "freely addressed", that is, a request for a particular GUID should case storage to be automatically allocated on access rather than requiring to be pre-allocated. This reduces the requirement for time-based coordination between helpers and clients of helpers.

Coordinated submissions across multiple helpers (Requirement 4b.)

Some discussion of this complex use case appears on the cross-tool integration page. Over and above the session usage requirements in the previous suggestion, handling this use cases requires further agreement on the EL-traversible material held in Session, as well as more refinement to the Sakai transaction model to ensure that persistent state is not corrupted by partially successful submissions. One use of a further nominated global session key might be to be a repository for rendered binding errors resulting from a partially successful submission within a helper. This case will require a good deal of further discussion and elaboration with particular use cases.

Demonstrations and code samples

An AHAH-based (markup-oriented) "helper" concept using RSF and the EntityBroker is demonstrated and written up in Writing and using SuperHelpers within Sakai.