Common Container

Overview

Proposal of

The common container is a lightweight gadget-and-container framework.

Motivations

For container clients, this is aimed to:

- simplify container and gadget integration model.
- provide near-zero barrier to entry to become a gadget container.
- directly fix problems in all containers, instead of per-container basis.
- reduce changes and differences across containers.

For the Shindig-based team, this is aimed to:

- place a localized control of gadget-and-container interaction.
- clearly separates gadget-and-container interaction from container-specific logic.
- allow security, latency, and functionality changes simultaneously in both gadget and container.

Features

Some features are, but not limited to:

- versioned JS loading.
- navigation of gadgets (pop-in and -out on page).
- latency timing broadcasting.
- hooks for gadget+container RPC functionalities.
- pre-loading of gadgets.
- security token refresh.
- user-preference.

APIs

All non-private methods and members can be used publicly.
However, a container client should restrict its usage from the context of the container instance, as much as possible.Specifically, restrict your usage to below --

From shindig.container.Container">From shindig.container.Container

```
// to instantiate a container
var container = new shindig.container.Container(...)

// to instantiate a gadget site
var site = container.newGadgetSite(...)

// to preload / unload (un-preload) gadgets
container.preloadGadget(...)
container.preloadGadgets(...)
container.unloadGadget(...)
container.unloadGadgets(...)

// to navigate / close gadget sites
container.navigateGadget(...)
container.closeGadget(...)

// to get gadget metatadata
container.getGadgetMetadata(...)

// to register gadgets.rpc service in container
container.rpcRegister(...)

// useful constants
shindig.container.ContainerConfig.*
shindig.container.RenderParam.*
shindig.container.ViewParam.*
```

From shindig.container.GadgetSite

```
// to manipulate a gadget site
site.setHeight(...)
site.setWidth(...)
// to have container call site via gadgets.rpc.
site.rpcCall(...)
```

From shindig.container.GadgetHolder">From shindig.container.GadgetHolder

```
// none
```

Examples / How to?

Render your first helloworld gadget

```
var elem = document.getElementById("blah");                      // a DOM element to insert a gadget iframe
into.
var gadget = "http://www.labpixies.com/campaigns/todo/todo.xml";  // a URL to the gadget XML .
var container = new shindig.container.Container();
var site = container.newGadgetSite(elem);
container.navigateGadget(site, gadget, {}, {});
```

Render a gadget with container-specific logic

Specify &container= in the common container script inlined in your container page --

```
<script src="/gadgets/js/container.js?c=1&container=[container]"></script>
```

Optionally, one can specify --

&debug=1 -- for pretty-print of JS. Useful for debugging. Internal IP only.
&onload=XXX -- for a callback function. Useful for dynamic script injection. Allowed function: [a-zA-Z0-9_]+.
&jsload=1 -- for lazy-loading versioned JS. Useful to reduce latencies.

Force container to render gadgets in debug and force-fetch (ie: skip cache) mode

You can pretty-print JS output of the resulting render and force Shindig to fetch gadget spec by below. This will be enabled for all gadgets in the container.

```
var config = {};
config['renderDebug'] = true;
var container = shindig.container.Container(config);
```

Allow container to render gadgets in test mode

Gadgets have <Content view="test:default">...</Content>. This is a test script that can be ran to test the corresponding view (in this case, default). The content in this testable-view content will be renderer, the specified script will be executed and its results can be used by an automated test harness. To render this corresponding testable view along with the desired view --

```
var config = {};
config['renderTest'] = true;
var container = shindig.container.Container(config);
```

Select rendering view of the gadget

A gadget can have 1+ views (ex: home, canvas, profile, default), but only one can be rendered at a time.

```
var renderParams = { view: 'home' };
container.navigateGadget(site, gadget, {}, renderParams);
```

Set initial iframe width and height of the gadget to render

This will set the gadget iframe width and height. Dynamic-height calls to adjust height can change this initial height.

```
var renderParams = { width: 100, height: 200 };
container.navigateGadget(site, gadget, {}, renderParams);
```

Set class name of iframe for styling

To pop-in iframe with a custom CSS style.

```
var renderParams = { class: 'style' };
container.navigateGadget(site, gadget, {}, renderParams);
```

Set user preferences of gadget to render

User preferences can be container-specified. They can be of type string, number and an array. If the specified user preferences are not recognized by the gadget (ie: not specified in the gadget specified), they will simply be ignored. Internally, this will not affect the cache-ability of gadgets metadata, ie: subsequent gadget render request using different user preferences will not call another metadata fetch.

```
var userPrefsObject = {
  str: 'abc',
  num: 123,
  arr: [ 'one', 'two', 'three' ]
};
var renderParams = { userPrefs: userPrefsObject };
container.navigateGadget(site, gadget, {}, renderParams);
```

Pre-load gadgets before rendering

Pre-loading of gadgets will reduce latency. This is done by priming the gadget metadata cache in the client, with (possibly) all known gadgets to be rendered in the container page. After which, switching from one gadget to another, is simply from the cache. The common container supports this by calling --

```
var container = shindig.container.Container();
var gadget1 = "http://www.labpixies.com/campaigns/todo/todo.xml";
var gadget2 = "http://www.google.com/ig/modules/youtube_igoogle/v2/youtube.xml";
container.preloadGadgets([gadget1 gadget2]);
container.preloadGadget(gadget1);
```

Refresh security token

Security token is an authentication mechanism for gadget rendering, used to get people data, container validation, etc. For security reasons, they have a limited life-time, after which it ceases to work (and your gadget will likely break). CC supports refreshing of tokens by means of requesting for new ones, and pushing them to the gadget who needs, via gadgets.rpc. CC will do in the background (start upon a navigateTo(), preloadGaget(), preloadGadgets(), stop when all gadgets are closed). As a user, you should not need to do anything, but you can see it in action in Firebug.

What's new?">What's new?

2010/12/08: Allow gadget render in cajole mode

This can now be requested –
container wide, via config['renderCajole'] = true
per gadget request, via renderParams['cajole'] = true

2010/12/05: Correct gadget metadata caching

Correct client-side caching behavior. Previously, CC cached gadgets iframes and metadatas indefinitely locally in the browser client. This is a problem for container pages that long-lived (ie: AJAX applications will not see new gadget changes unless reloaded). Now, cache is busted according to what the server says (currently, fixed to shindig.cache.xml.refreshInterval), with some adjustments to differences between server/client absolute times. Cache will not be busted for navigated and preloaded gadgets. For simplicity, gadgets need to be closed (unnavigated) and/or unloaded (if preloaded) for its cache to be possibly evicted.

2010/11/01: Broadcast CSI latency times

Provides latency timing and reporting back to the container, upon a gadget navigation. This is facilitated by requiring the container to register a callback function, ie:

```
config['navigateCallback'] = function(data) { ... }
```

CC will synchronously call the specified callback function upon a gadget navigation. As part of the data, CC will return a JSON map of –

id : ID of the calling gadget site, ie: gadget site that performs the navigated.
url : Gadget URL (of gadget site with id) navigated to.
xrt : XHR time.

2010/10/05: Lazy-load and cache container.js

Add support for lazy-loading versioned JS. Specify &jsload= and &onload= (required upon presence of &jsload=) in /gadgets/js/container.js?jsload=1&onload=XXX. Shindig will return a very slim JS (with default browser cache) to dynamically load a corresponding versioned &v= JS in /gadgets/js/container.js?&v=YYY. Either one of two things will happen –

- If first encounter, this will do another fetch to Shindig for the full common container JS. This will have a long browser expiry time, 1 year.

- Otherwise, content will be retrieved from browser cache.

In both cases, each will require (at minimum) 1 HTTP fetch for the lighweight loader JS.

2010/09/30: Onload notification of container.js

Add support for callback upon JS download complete. Specify optional &onload= in /gadgets/js/container.js&onload=XXX, where XXX is globally-defined fu
nction window[XXX] = function()

Unknown macro: { ... }

.

Announcement email

> In Google, we've been developing a lightweight gadget-and-container
> framework, called "common container", to 1) simplify container and gadget
> integration model, and 2) provide near-zero barrier to entry to become a
> gadget container. The framework is a combination of JS (which container
> clients script source) and API RPC endpoints (which provides the JS with
> runtime metadata/information). Many of the features have been implemented,
> and are currently being productionized. Several Google containers are busy
> prototyping (to integrate with common container) with some success and speed.
> Meanwhile, there has been an independent community effort to modernize the
> Shindig container. It has a similar/same set of goals. In the spirit of
> re-use, we would like to bring non-Google-specific work that we've done in
> Google to Shindig. Engineers at Google and Paul Lindner have met, discussed
> and agreed on the motivations and feature sets. Roughly speaking, they are as
> follow –
> 1) JS is served via the gadgets server, as a container feature, ie:
> /gadgets/js/shindig.container?c=1. This will leverage gadget server
> compilation of JS and management of library dependencies through
> transitive-closure.
> 2) Standard JS API to get metadata to render a gadget, ie:
> /api/rpc?method=gadgets.get.
> - API is implemented using an RPC call similar to other APIs (people,
> activities, etc) in Shindig.
> - RPC will return a URL template and gadget metadata, sufficient to generate
> iframe render URL. Also, the response can be cached by the clientto evaluate
> URL template with data for subsequent/similar requests (ie: different user
> preferences) without requiring an HTTP fetch.
> - Also need JS API to refresh a security (/OAuth) token, ie: /api/rpc?
> method=tokens.get. Gadgets requiring tokens, both short- and long-lived,
> relies on the container to have these primed in a timely fashion.
> 3) Standard JS API to render/navigate gadget.
> - Base API renders into an HTML element, ie: container.navigateGadget(string,
> Element)
> - Ideally support double buffering for switching gadgets views, ie:
> gadget.views.requestNavigateTo().
> - Need a standard "ready-to-draw" RPC callback, when the gadget has finished
> initial rendering.
> - Respects preferred width, height and other gadget-specified metadata used
> in rendering.
> - Need rendering "styles" to show gadget in a modal dialog, show hovering
> next to an HTML element, etc.
> - Need concept of a "primary" gadget on the page. Navigating to this gadget
> should fire a callback on the parent page to support navigation in the
> browser bar.
> 4) All RPC APIs (gadgets, people, activities) should be callable form the
> container page.
> - Likely implementation is an iframe on the domain of the gadget host. Web
> site calls gadgets.rpc() to iframe and iframe makes XHR to gadget host.
> - Gadget render iframe URLs can be enhanced for latency-optimizing features,
> ie: container can be an RPC hub for gadgets on the page.
> - RPC responses can be obtained from pre-loaded/cache metadata for latency
> improvements.
> - Need to flesh out a standard API to redirect to a login page from the 3rd
> party site.
> What's next? Discuss. We would very much like to hear your thoughts. In the
> next few days, we will prepare common container code base for public viewing,
> and upon no strong objections, we will send an initial code drop for review,
> where which we can discuss further in greater technical details.