Getting SAP to talk to Google Gadgets

Applies to:

SAP ECC6. For more information, visit the ABAP homepage.

Summary

Using Google Gadgets as a lightweight desktop analytics feature.

Author: Patrick Dean

Company: Hafele

Created on: 27 April 2011

Author Bio

Patrick Dean is an ABAPer with 9 years experience, currently working for Hafele. His favorite SDN contributors are Thomas Jung and Jocelyn Dart, enjoys all things Tech and will carry on doing SAP as long as he can get away with it!

Table of Contents

Gadgets – Concepts Explained

In business, we often want to quickly look at Key Performance Indicators (KPIs) which give us a brief description of how the business is doing. A KPI for the Sales Team might be “Number of Sales Orders placed today”. The Customer Services department might want “Number of phone calls answered in the last 3 hours”, and the Purchasing Department might be interested in Stock Value.

Google Desktop Gadgets gives us a tool where we can, at a glance, see these KPIs. The Gadgets are lightweight, and sit on the desktop without taking up additional windows.

In this example, we have two Gadgets. The one on the left shows us the number of Sales Documents of different types we have done today. The one on the right shows Open Transfer Orders.

The nice thing about these gadgets is that they update themselves. Since the gadget is written in javascript, we can make use of the timer function that will repetitively go and ask SAP for some information, so the data can refresh itself without any interaction from the user.

Google Desktop Gadgets are written in XML with a Javascript. XML does the layout – where the elements (labels, logos, list items) sit in the gadget, and Javascript does the logic (getting the data, doing any calculations to it.)

As part of the standard download from Google Desktop, you get a nice little sidebar, which you can put extra gadgets into. Here I’ve got the Media Player one(with Adele’s excellent “21” album), the weather for Coventry (cold, but not raining) and a little notepad upon which to make quick notes.

To get SAP to talk to the Gadgets, you need 2 key skills, which we will introduce in this document. First up, you need to be able to get SAP to present the data. Secondly, you need to get the gadget to read the data. This is a big geeky step-by-step document, you will most likely only need to use it once or twice. (You’ll only download the gadget editor once, for instance…)

Presenting the Data from SAP

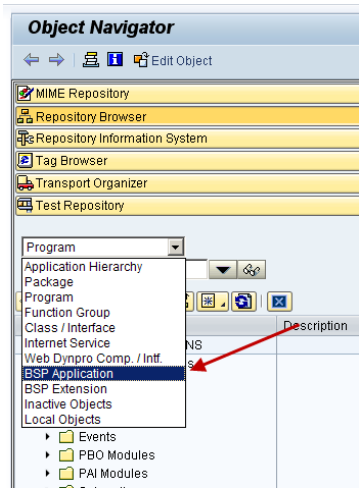
Creating a BSP

The easiest way to get SAP to serve up the data is to create a BSP (Business Server Page). BSP is kind of deprecated by SAP now, it was their interim Web technology before they came up with WD4A (Web Dynpro for ABAP). However, the good thing about BSP is that you can use it to create XML files.

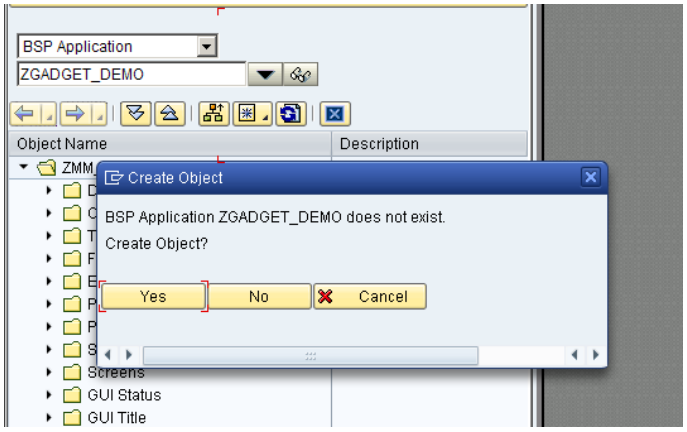
In this example we will create a really simple BSP which will just use the system time to select some text, and we'll ultimately serve that to the Gadget. In real life you could create a function module or just select directly within your BSP to get the information you're interested in.

BSPs are created in transaction SE80. Yep, that transaction SE80, that one you use to create ABAP code.

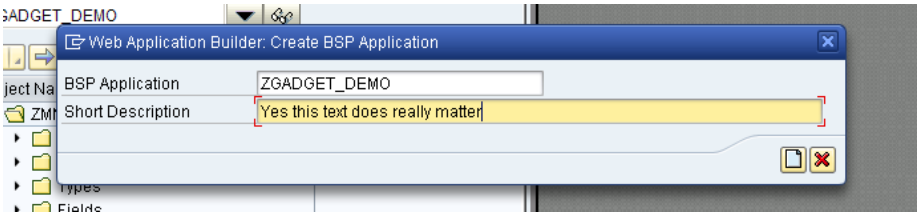
Select "BSP Application" from the dropdown menu.



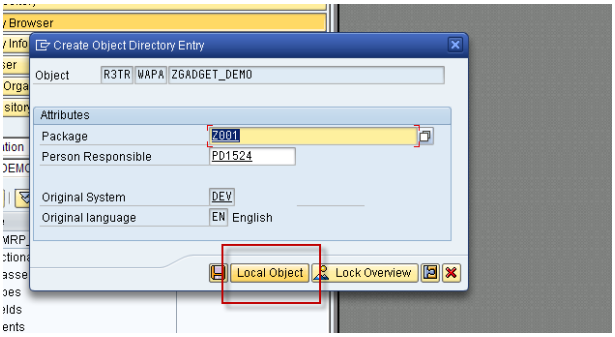
Then make a new BSP Application - I called mine ZGADGET_Demo - using forward navigation. (i.e. typing the name of something that doesn't exist yet, and have SAP create it for you…)



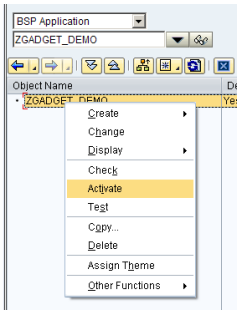
Give it a description..



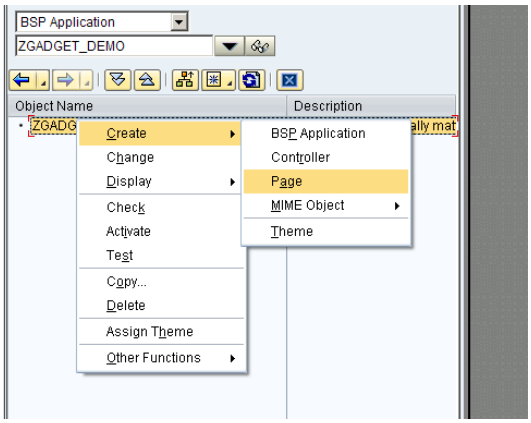
Just create as a local object for now

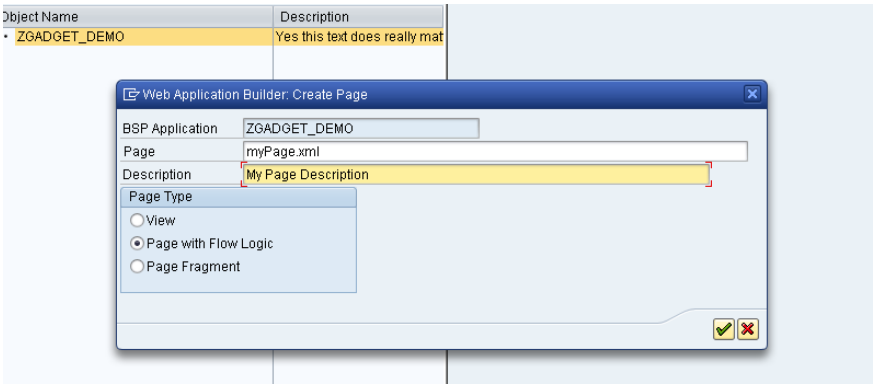


You've now got a BSP Application that can act as a container, or holder, for the XML page. Activate this, by rclicking on the object choosing "Activate". (If you don't do this now it'll struggle when you try and do it later at the same time as your XML, you have been warned… it's not terminal, but it's annoying).



Next up, you want to create the XML page itself. RClick on the Application, go to "Create", then "Page"

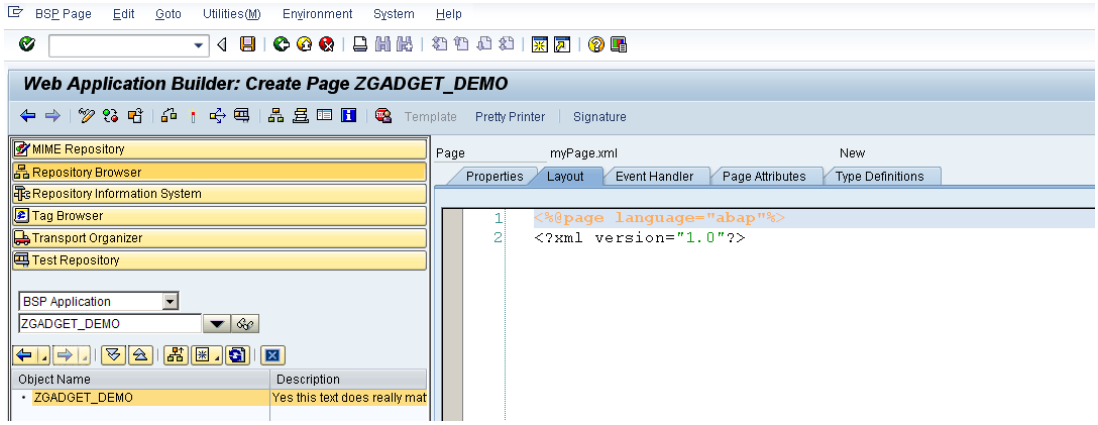




The default Page will be ".bsp". Change the extension to ".xml" and give it a name. And a description.

Leave the pagetype as "Page with flow logic".

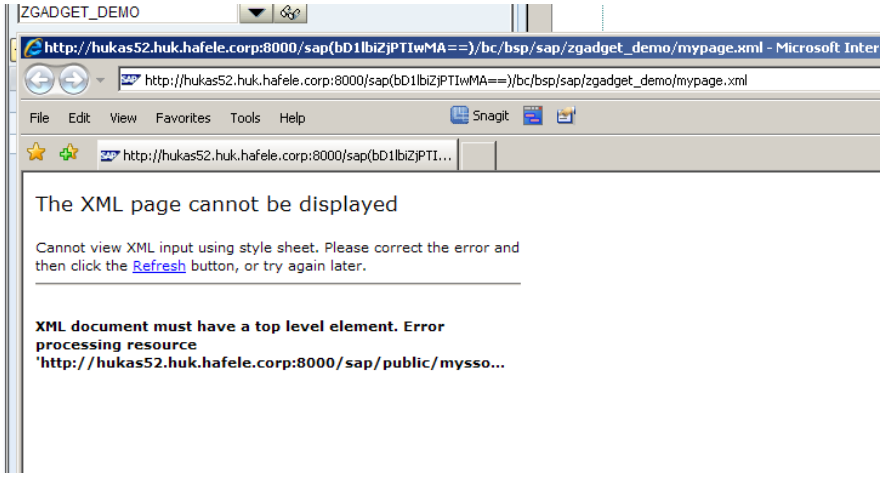
You've now got a new page you can edit and mess with.



At present the page is "New", which is no good to anyone, so activate it. or Ctrl-F3.

If you've not already activated the Application, SAP will have a problem with this, and you'll need to save the page then activate the Application, at which point it'll tell you that you're still editing the page. So, having saved the page, switch to "read only mode", activate the application, then the page. Bottom line is, it's easier if you activate the application first, before dealing with the page!

So far so good. If you tap F8 to run this now, it will try and present it in the default web browser. That's what BSPs do.



At the top we can see that this is a webpage being served by SAP. The /bc/bsp/sap bit shows us that it's a SAP BSP, the hukas52 bit at the start is the name of my SAP DEV server.

What we see here is the XML formatting error.

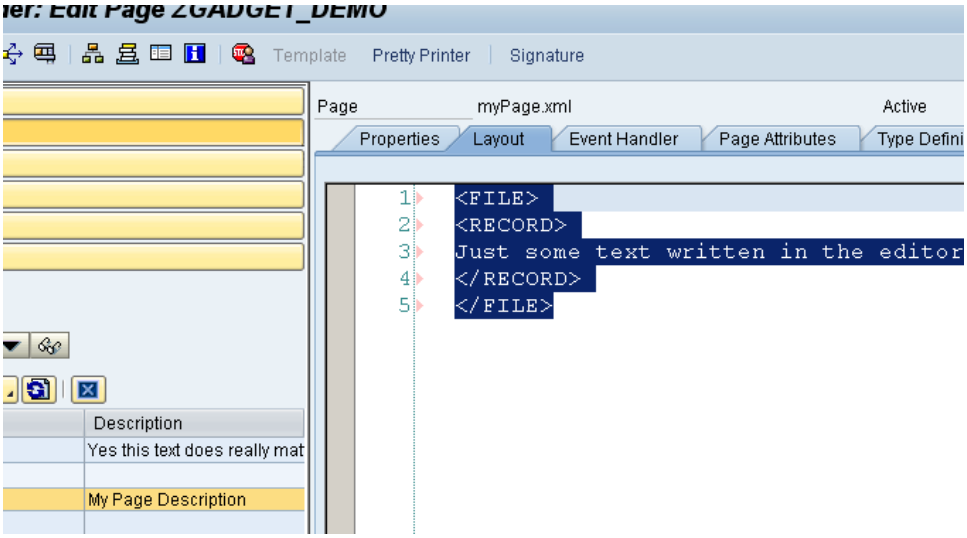
So if we go back to the SE80 Editor, get rid of the default XML stuff

```
<%@page language="abap"%>
<?xml version="1.0"?>
```

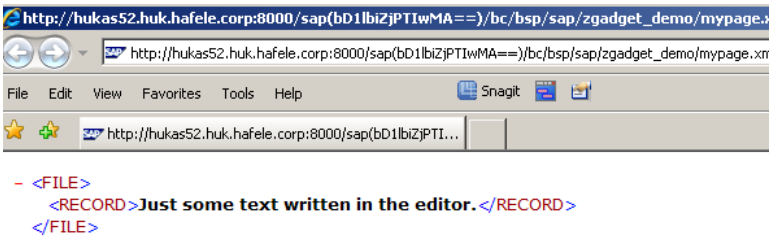
And replace it with the following XML:

```
<FILE>
<RECORD>
Just some text written in the editor.
</RECORD>
</FILE>
```

So it should now look like this: (Don't forget to save and activate!)



When you F8 this, it should show the xml in the Web Browser:



Adding ABAP to your BSP

Alright, so we've got SAP Serving up some data in XML. But it's kind of useless, because it's just hardcoded at the moment, without any ABAP logic. Fortunately BSP allows us to switch between ABAP and HTML/XML.

Using the `<%` and `%>` tags, we can open and close portions of ABAP.

In order to do this, we include the following code in the top of our BSP:

```
<%@page language="abap"%>
<%@extension name="htmlb" prefix="htmlb"%>
```

We then open some ABAP to do the logic of what data we want to present:

```
<%

data: l_message type char120.

case sy-uzeit+5.
    when 1.
      l_message = 'You reached SAP... well done!'.
    when 2.
      l_message = 'Congrats, SAP has been hit.'.
    when 3.
      l_message = 'BSP Rocks. Like Paul does.'.
    when 4.
      l_message = 'Britney Spears runs SAP.'.
    when 5.
      l_message = 'The best run businesses run SAP'.
    when others.
      l_message = 'You will see this one the most.'.
endcase.

%>
```

What this does is take the 6th character of the system time, (sy-uzeit+5) and uses it to make the local char120 variable `l_message` a different value each time. This value is going to be 0-9, and we're handling 1-5 with funny messages, and everything else with `'You will see this one the most.'`

But we still need to present this. We do this with the following code.

```
<MESSAGE>
 <TXT><%= l_message %></TXT>
</MESSAGE>
```

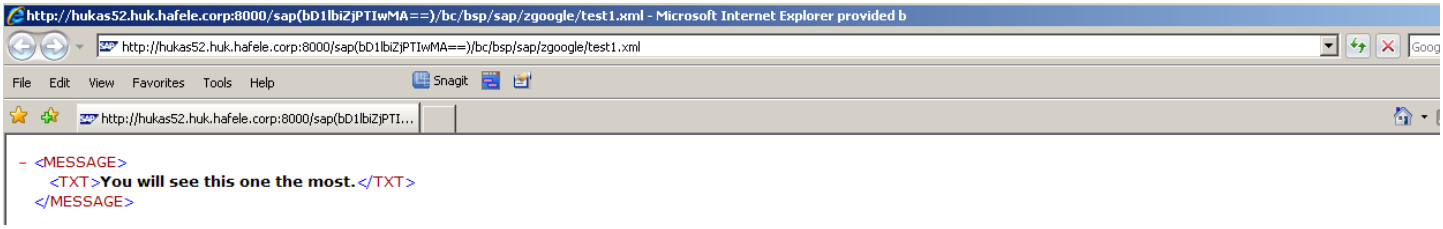
Notice we need to "escape" the ABAP variable `l_message` into the xml, using the `><%= l_message %>` notation.

Your whole code should now look like this :

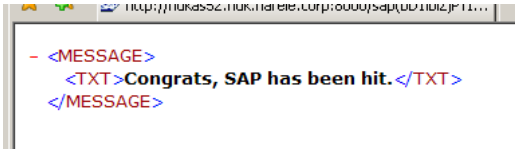
```
<%@page language="abap"%>
<%@extension name="htmlb" prefix="htmlb"%>
<%

data: l_message type char120.

case sy-uzeit+5.
    when 1.
      l_message = 'You reached SAP... well done!'.
    when 2.
      l_message = 'Congrats, SAP has been hit.'.
    when 3.
      l_message = 'BSP Rocks. Like Paul does.'.
    when 4.
      l_message = 'Britney Spears runs SAP.'.
    when 5.
      l_message = 'The best run businesses run SAP'.
    when others.
      l_message = 'You will see this one the most.'.
endcase.

%>
<MESSAGE>
 <TXT><%= l_message %></TXT>
</MESSAGE>
```

Testing the BSP

You can now run this by pressing F8,



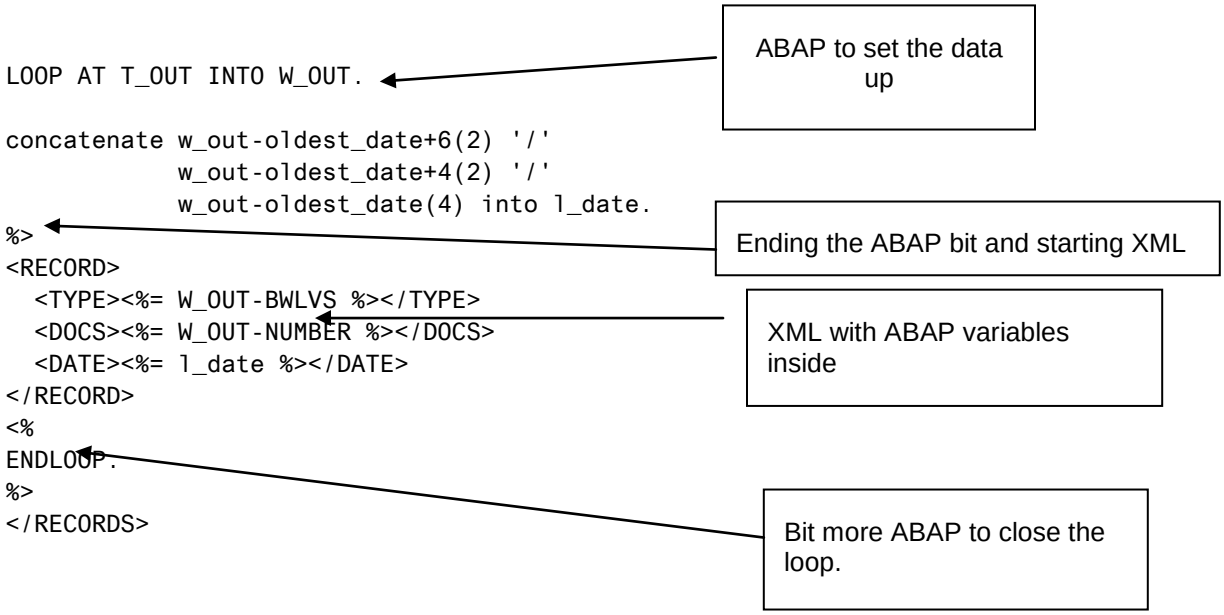
Press “Refresh” on your browser a few times,  and you should start to see the different values depending on the system time:



Take note of the http address of the XML. You will need it later to point the Gadget at the data published here.

Mine was : http://hukas52.huk.hafele.corp:8000/sap(bD1lbiZjPTIwMA==)/bc/bsp/sap/zgoogle/test1.xml

So this example is a bit trivial, but it gives you the building blocks for more complicated XMLs. One of the neat things about BSP is that you can escape-in and escape-out of ABAP several times if you need to. Here we see a loop starting in ABAP, switching to XML format, escape-in a few times for variable values, then doing some more ABAP to close the loop.



Well Formed XML

For XML to be well formed, it needs a file level encapsulating tag. So instead of :

```
<record>
   <stuff1>myval</stuff1>
   <stuff2>myotherval</stuff2>
</record>
<record>
   <stuff1>mymumsval</stuff1>
   <stuff2>mymumsotherval</stuff2>
</record>
```

You need to just add tags at the top and the bottom. Nothing special about what they're called, they just need to be there.

```
<wholefile>
<record>
   <stuff1>myval</stuff1>
   <stuff2>myotherval</stuff2>
</record>
<record>
   <stuff1>mymumsval</stuff1>
   <stuff2>mymumsotherval</stuff2>
</record>
</wholefile>
```

Okay, that's all for now from the BSP side of things. Next we move on to the Gadget Editor.

Google Gadgets – Getting Started

Google are good at documenting their own APIs, gadgets and so on. There's plenty of documentation on their website as to how the editor works and it can be picked up pretty quickly.

First of all, download the Google Desktop Gadget SDK from

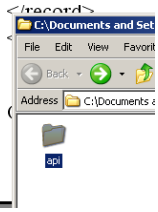
http://desktop.google.com/downloadsdksubmit (link correct at 03.03.2011).

And a load of top notch getting started documentation here:

http://code.google.com/apis/desktop/docs/Tutorials/ModifyHelloWorld/index.html (link correct at 03.03.2011).

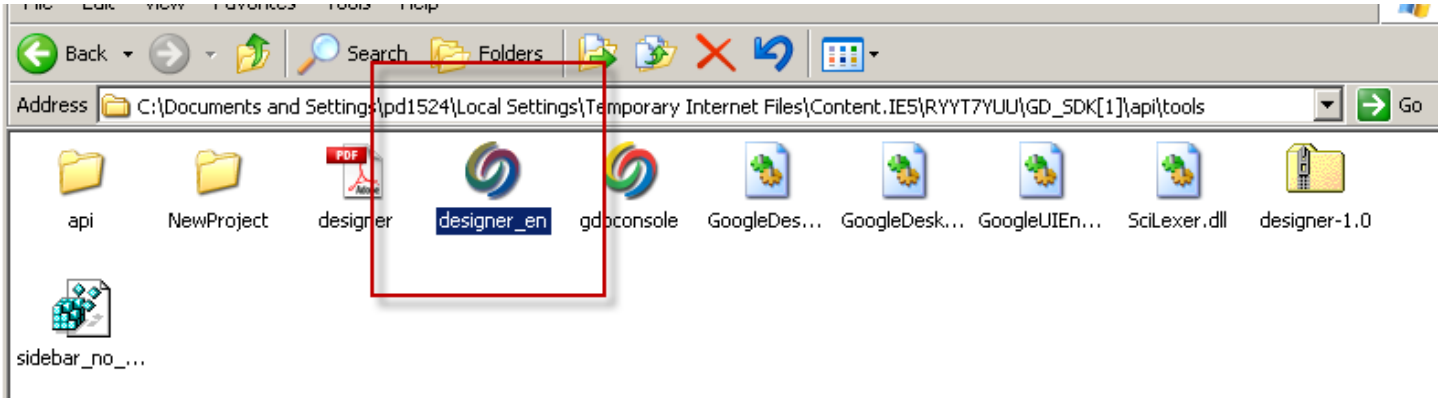
If googling for Google Gadget Editor, be careful because Google differentiate between Web Gadgets and Desktop Gadgets. In this case, we're more interested in the Desktop Version.

You end up with a folder called API:

Open this: 

then go to the Tools folder.

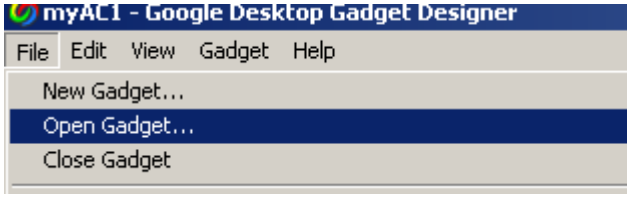
Designer_en may need unzipping. Once it's been unzipped, it should look like this:



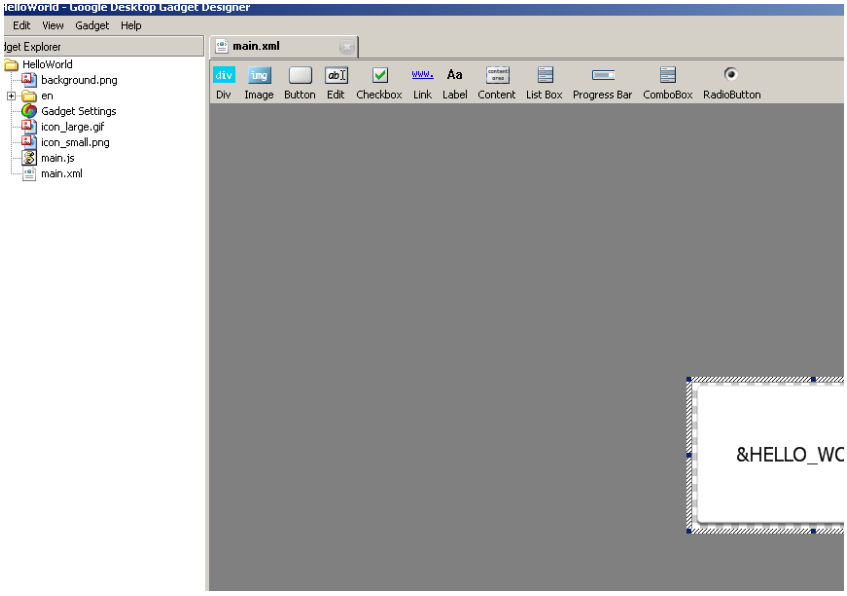
Dclick into the designer_en, and this brings you to Google's Desktop Gadget Designer.

With their SDK, Google provide a load of examples of how to do different things, these are all stored in the "API\Samples\Gadgets" folder.

To look at one in action, and how it's put together, go to "File->Open Gadget"



Find the directory you put the API folder into, open up Samples\Gadgets and there's a good old fashioned "Hello World" example.



The navigator on the left hand side allows you to navigate between different elements of your gadget. For the most part you'll be interested in **main.xml** – an xml file controlling the layout of your gadget, and **main.js**, a javascript file that handles the logic.

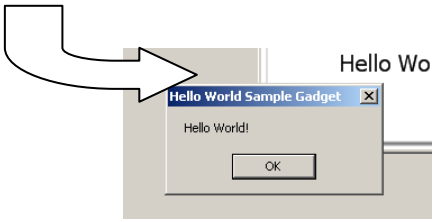
Javascript is just another programming language, with many principals and commands similar to what you should already be familiar with from ABAP. Again, there's tonnes of documentation on the web, and the objects Google have as part of their Desktop API are documented here

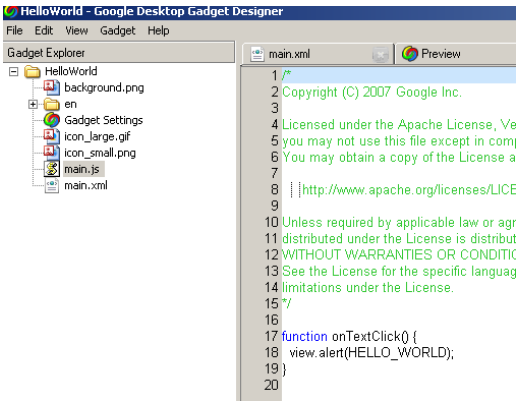
http://code.google.com/apis/desktop/docs/gadget_apiref.html (link correct at 03.03.2011)

You can run the “Hello World” example from the Google Desktop Gadget Editor by pressing F5.

This gives you a simulation of what the Gadget will look like.

And if you click on the “Hello World!” you get a popup box





In the Gadget Editor, use the Navigator to look at the contents of the main.js file

Unsurprisingly, it’s REALLY basic. There is one javascript function, that pops an alert box with the value of variable “HELLO_WORLD”.

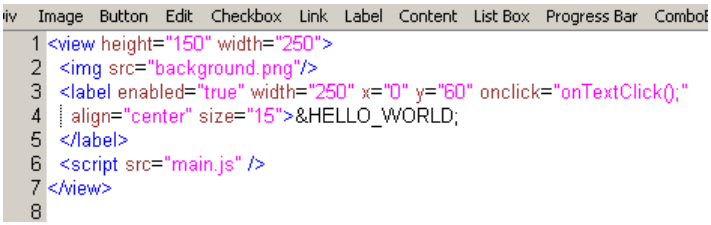
We’ve seen by testing it that it gets called when the user clicks on the “Hello World!” text, but where is this call defined?

It’s actually referred to in the main.xml file, the file that handles our layout.

main.xml can be viewed in two ways, the designer view and the source view.

These are controlled by the two buttons down at the bottom left hand side of the main gadget designer screen:

Designer view gives a graphical layout, Source view gives the XML description of that layout.



There's a lot of stuff in there, so let's break it down bit by bit.

Describes the height and width of the view in pixels

```
<view height="150" width="250">
```

Uses background.png as the background for the Gadget. Opening the file from windows shows that it's basically a white background with some blurred edges to give your gadget a nice floating shadowy effect.

```
 <img src="background.png"/>
```

The following sets a label up at x,y, coordinates 0,60. And the interesting bit is onclick="onTextClick()" This ties together the onClick event of the label, to the "onTextClick()" java function.

```
<label enabled="true" width="250" x="0" y="60" onclick="onTextClick();"
  align="center" size="15">&HELLO_WORLD;
</label>
```

This bit just points the xml at the javascript file so it knows where to get it's javascript from.

```
<script src="main.js" />
```

This bit closes the view tag.

```
</view>
```

Gadget talking to SAP

So what about getting it to talk to SAP? We do this by way of javascripts httpRequest object. The best way to explain is just to show the code and detail what all the components are doing.

```
main.xml

<view height="250" resizable="true" width="250" onopen="view_onOpen()"
>
  <img height="287" x="0" y="-5" src="stock_images\background.png"
   />
  <edit height="98" name="edit1" width="212" x="18" y="15" bold="true"
    font="Arial" size="14" wordwrap="true" multiline="true"
    readonly="true" value="text"/>
  <script src="main.js" />
</view>
```

Nothing stunning here… we set the view sizes. The view has an attribute “onopen”, that is to say, what javascript function gets called when the gadget is opened. In this case, we set it to “view_onOpen()”, so we’d expect there to be a java function of that name when we open main.js.

The other thing to note is we’ve got an edit1 field, which has been set to readonly=”true”.

We’ll use the main.js functions to change the contents of the edit1 field.

In the next section I’ll show the whole code so you can cut and paste it into the Gadgets editor, then repeat it with notes so you can understand what’s going on.

main.js – no explanation…

```
var myOut;                                          //Sets a global parameter for use
later.

function view_onOpen() {                                    //
var myTimerID = setInterval("button1_onclick()",3000 );

}

function edit1_onclick() {
}

function button1_onclick() {

var message;
var xmlMessage;

var req = new XMLHttpRequest();
req.open("GET",
"http://hukas52.huk.hafele.corp:8000/sap(bD1lbiZjPTIwMA==)/bc/bsp/sap/zgoogle/test1.x
ml", false);  // The last parameter determines whether the request is asynchronous.
req.send();
```

```
if (req.status == 200) {
xmlMessage = req.responseXML;

myOut = xmlMessage.firstChild.text;
gadget.debug.trace(myOut);
edit1.value = myOut;
edit1.opacity = 255.
FadeOut();

//gadget.debug.trace(req.responseText);
} else {
   alert(req.responseText);
//gadget.debug.trace(req.responseText);
}
}

function label1_onclick() {
}
function div1_onclick() {
}

function FadeOut() {
  anim = beginAnimation("Animate()", 255, 0, 3000);
}

// This function is called repeatedly for 2000 millisecondsbutton1_onclick()

function Animate() {
  edit1.opacity = event.value;
}
```

main.js – with explanation

```
var myOut;                                    //Sets a global parameter for use
later.
```

Here we declare the function onOpen. It creates a timer object called myTimerID which will run every 3000 ms (i.e. once every 3 seconds) This will call a function called “button1_onclick” every 3 seconds.

```
function view_onOpen() {                                //
var myTimerID = setInterval("button1_onclick()",3000 );
}
```

Here is the button1_onclick function. This gets called by the myTimerID object, as detailed above.

```
function button1_onclick() {
```

Declare some variables for use later..

```
var message;
var xmlMessage;
```

This variable is of type XMLHttpRequest(), and is at the heart of what allows Gadgets to talk to SAP.

```
var req = new XMLHttpRequest();
```

The method ‘open’ with an XML address allows the javascript to read stuff in from SAP. This is where you need to enter the http of the BSP you created in the first part of the exercise.

```
req.open("GET",
"http://hukas52.huk.hafele.corp:8000/sap(bD1lbiZjPTIwMA==)/bc/bsp/sap/zgoogle/test1.x
ml", false);  // The last parameter determines whether the request is asynchronous.
```

Sends the request to SAP to get whatever information it wants to share with us.

```
req.send();
```

```
if (req.status == 200) {                If we successfully talked to SAP
xmlMessage = req.responseXML;                   Gets the response into a big long field.

myOut = xmlMessage.firstChild.text;                     Looks at the value of the first “child”
or node in the XML.
gadget.debug.trace(myOut);              Used to output the myOut value to the debugger.
edit1.value = myOut;                    Sets the edit1 field’s value to what was in the XML field.
edit1.opacity = 255.            Sets the opacity (how dark or see-through it is on the screen) to hi
FadeOut();                      Calls the Fadeout function (makes it look a bit livelier…
                                                You can leave that bit out if you like)
    //gadget.debug.trace(req.responseText);
} else {                                        If we didn’t talk to SAP
       alert(req.responseText);                 Output an error message.
    //gadget.debug.trace(req.responseText);
    }
    }
```

These two functions just make it look a bit prettier, by taking the edit1 field and animating it from Opacity = 255 to Opacity = 0 over the course of 3000ms, this gives it a heartbeat effect, which I like, some people don’t.

```
function FadeOut() {
  anim = beginAnimation("Animate()", 255, 0, 3000);
}

function Animate() {
  edit1.opacity = event.value;
}
```

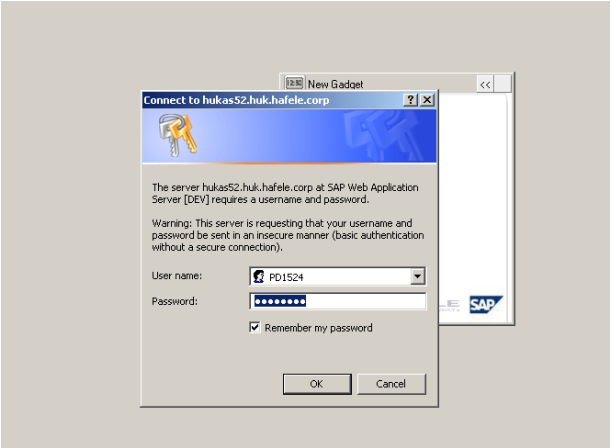
Bringing it All Together

All being well, we should now be at a point where we've created the key ingredients:

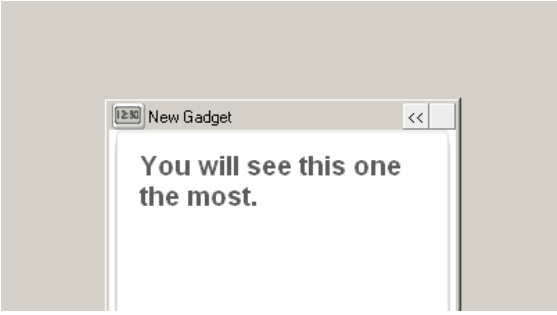
1) A BSP-xml that presents information out of SAP.
2) A Javascript that can read it
3) An xml file that handles Google Gadgets presenting of the data on the desktop.

Execute (F5) the gadget in the Google Editor to see if it worked!?

In my case, since the Gadget is trying to talk to a SAP Server, SAP wants to know a username and password to get on there. Enter your SAP credentials here.



All being well, you should see the test version of the Gadget in the Gadget Test area, heartbeating away, getting data from SAP:



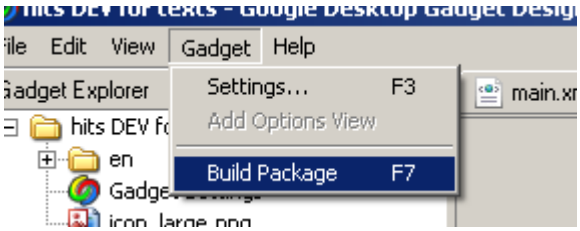
Deploying to the Desktop

Now we want to be able to deploy it on the desktop. For this we need the Google Desktop Sidebar.

This is available from http://desktop.google.com/en/GB/

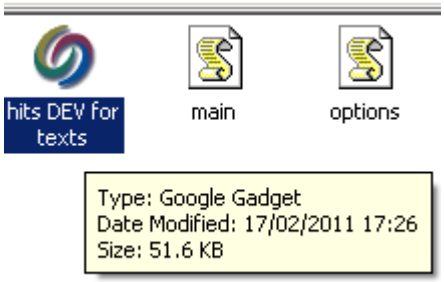
It comes with a few standard gadgets, and the option to download more. It's a lovely bit of software.

Once you've done that, you need to package the gadget you've just created. In the Gadget Editor, select "Gadget->Build Package"



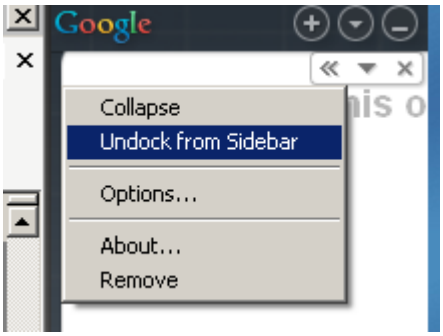
This creates a google gadgets file (.gg extension) in the same directory as you built the gadget in the first place.

Whilst running Google Desktop Sidebar, dclick on the .gg file created as part of the package. I've renamed mine, but the principal is the same:

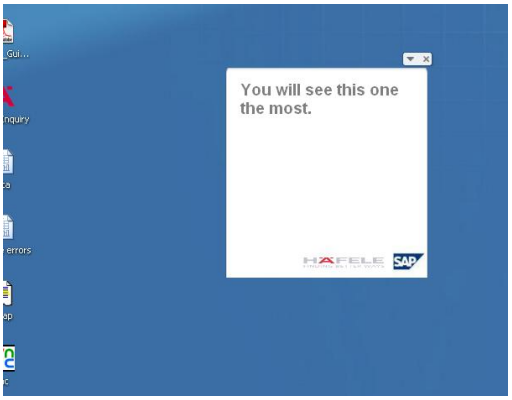


The gadget should open in the sidebar :

From here you can undock it

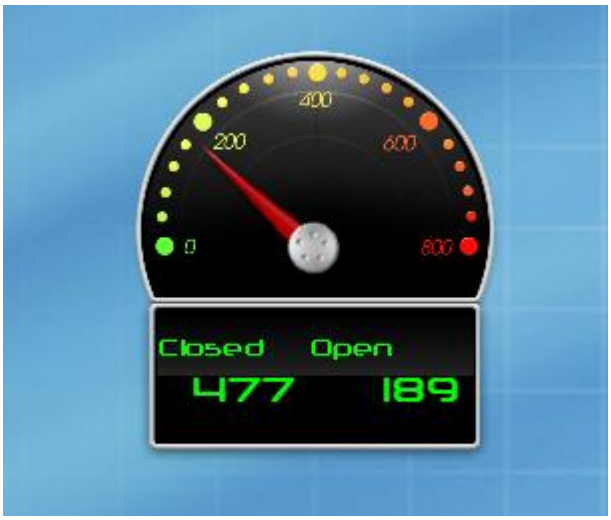


and have it floating about on your desktop :



What's Next?

The potential for these Gadgets is huge. In my experience users love the "at a glance" functionality, and the aesthetic feel of these gadgets. Using Java you can manipulate graphics too, so I had our design guys knock up this cool gauge to monitor how many closed and open TOs we've got at the moment. The needle and gauge are separate graphics items, the needle being rotated as a function of the number of Open Transfer Orders. The text at the bottom was handled by the styling of the XML in the Gadget Editor.



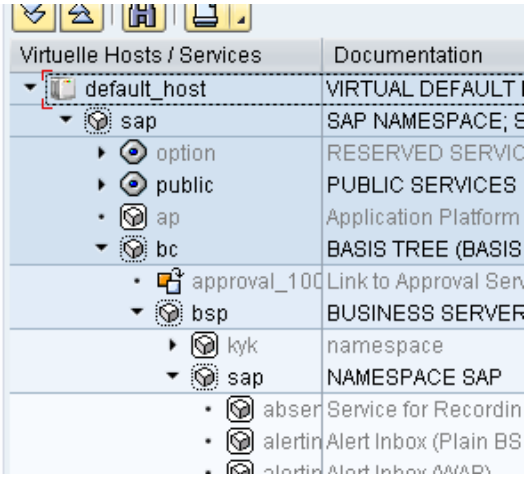
Since Gadgest use javascript, there are all sorts of funky things you can do with them – animations and timers being the most obvious, but you can also do some AJAXy/Web2.0 style things; think facebook style autocompletes and Google Maps APIs. The XML layout allows for logos/images, css skinning etc to make it all fit in with a company brand and complementing colors for your desktop… if you're that way inclined.

Now you've taken the first few steps, and are familiar with the environment, here's a few extra notes:

The BSP for real-world apps **probably should sit on your PRD server.** rather than your DEV server. Once you've transported it there, run it in PRD to check it works. If it does, then you should change the http address in your gadget, since the PRD server is different to the DEV server, the http address will be different.

If it gives you a "Service is not active" error**, then you may need to activate the service,** using SAP transaction SICF.

Execute through the first SICF screen, then follow the menupath default_host->sap->bc->bsp->sap



Then scroll down till you find the name of your service. Right click on it and activate it.

You can interact with SAP… kind of. By passing page attributes as part of your xmlHttpRequest, you can tell SAP what information you're interested in and make it return different data. Either google BSP page attributes and put 2 and 2 together… or wait for the next tutorial!

Float past the SAP Security message –

Replace

```
req.open("GET",
"http://hukas52.huk.hafele.corp:8000/sap(bD1lbiZjPTIwMA==)/bc/bsp/sap/zgoogle/test1.x
ml", false);
```

with

```
req.open("GET",
"http://hukas52.huk.hafele.corp:8000/sap(bD1lbiZjPTIwMA==)/bc/bsp/sap/zgoogle/test1.x
ml", false, "USERNAME", "PASSWORD" );
```

Since this is just "view" data, you can probably hardcode and use a generic username/password. Or your own. Up to you.

Possible Performance Implications.

The use of the timer function in javascript to automate the polling of SAP aught to be used with caution, particularly for database intensive selects. Remember that each of these "heartbeats" is re-calling SAP and asking it for data, so 100 users with 5 gadgets each updating every 10 seconds = 5000 calls in a minute. We've seen situations where lots of users want up to the second information on dozens of metrics You need to explain to them the implications!

You could get around this with SAP XI, since this can present the data without researching it out of ECC6 again. BSP is not really designed for this XML presentation, the WebService function might be more suitable.

Related Content

Dashboard Design

Thomas Jungs WebServices

WD4A + Flex

For more information, visit the ABAP homepage

Disclaimer and Liability Notice

This document may discuss sample coding or other information that does not include SAP official interfaces and therefore is not supported by SAP. Changes made based on this information are not supported and can be overwritten during an upgrade.

SAP will not be held liable for any damages caused by using or misusing the information, code or methods suggested in this document, and anyone using these methods does so at his/her own risk.

SAP offers no guarantees and assumes no responsibility or liability of any type with respect to the content of this technical article or code sample, including any liability resulting from incompatibility between the content within this document and the materials and services offered by SAP. You agree that you will not hold, or seek to hold, SAP responsible or liable with respect to the content of this document.