APEX PLSQL REPORT REGIONS AND APPS THAT BUILD THEMSELVES

Bill Holtzman and Steve Schreck, National Air Traffic Controllers Association

Introduction

The National Air Traffic Controllers Association (NATCA) is a union organization representing about 15,000 FAA employees, including mostly air traffic controllers but also engineers, inspectors, financial and medical staff, and others. The members are spread across the United States in hundreds of locations. As a result, we have a tremendous need for internet-based productivity tools. But paying contractors for complex and highly-customized web applications is typically out of the range of our budget.

NATCA discovered Application Express in 2005 just after it was released as HTMLDB 1.6. We immediately found ApEx to be a very appealing platform because of:

- low barriers to entry,
- the ability to develop applications rapidly,
- tremendous portability,
- the ability to move up the learning curve towards more sophisticated methods,
- the scalability, ruggedness, reliability and integrity of the Oracle database,
- and most importantly the internet-based development environment.

This last feature allows us to build our applications in the tiny cracks in our time as we hold down our regular jobs as controllers. We program on our breaks at work, while sitting in airports, when our kids are taking gymnastics classes, and anywhere else that has Wi-Fi! This creates a lifestyle efficiency that has been extremely valuable. Instead of spending the same amount of time defining and explaining our business processes and writing requirements for a contractor, we are able to build the apps ourselves, translating our needs directly into code, and at the same time build valuable skills within our organization.

All of this has made it possible to automate, centralize and merge many of our business processes without the investment of millions of dollars, providing our local union representatives with productivity tools and information inexpensively. This has vastly improved the organization from the ground up, making every representative and employee more efficient and effective. It has allowed them to focus on the core aspects of their jobs: representation, negotiation, and communication.

This paper will review key techniques developed in two of our applications.

GATS – PLSQL Report Regions

One of the major applications we deployed was a grievance tracking system called GATS. Grievances are the bread-and-butter of the union and each represents a charge against the employer for either failing to adhere to federal regulation or causing some adverse action to the employee. At present there are hundreds of thousands of open grievance files in GATS.

One of the core features of GATS is the listing of grievances. These listings evolved into highly customizable presentations that enable the user to quickly and intuitively make both small-scale and large-scale updates to the data. At the heart of these listings are PLSQL-generated SQL Report Regions. This paper will review the finer aspects of these regions and the numerous methods incorporated into these regions to enhance the user experience, provide security and add capabilities.

Survey Junkie – Creating Pages on the Fly

Recently we developed a new application intended to enable non-technical users to develop and deploy surveys directly to the membership without assistance. This app had to be able to somehow develop itself, creating items and other ApEx elements at the whim of a relatively untrained user.

Our solution exploits the fundamental portability of ApEx. An Apex application export file contains code that ApEx reads to build items, regions, buttons, pages, and all of the other other elements of an application. We reviewed that code and found procedures such as **`wwv_flow_api.create_page_item`** that we could manipulate dynamically to enable the survey author user to create items. Ultimately we created code to enable the survey author user to create entire pages in the application containing all of the items, buttons, regions, and other elements to conduct their survey. This paper will review our methods in this endeavor.

PLSQL Report Regions

Figure 1 shows one of the grievance listings in GATS.

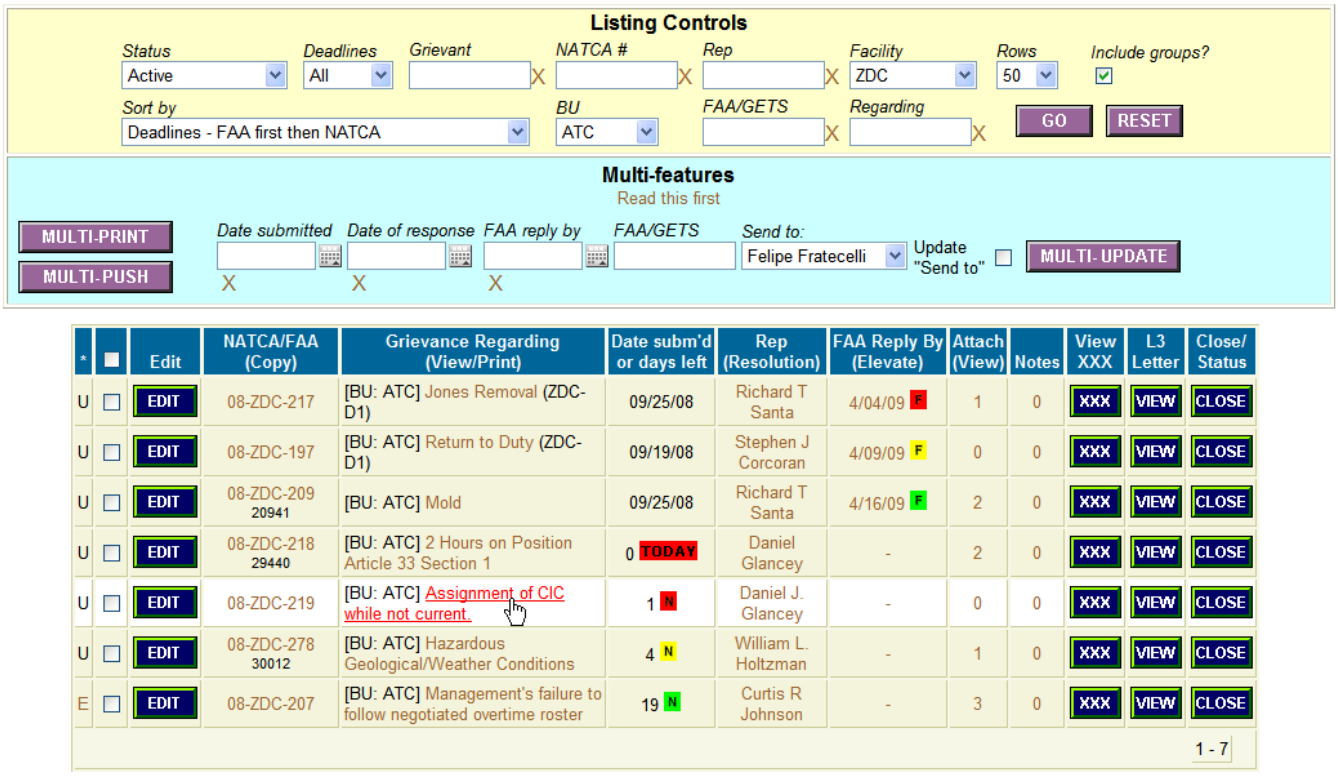

Figure 1. Typical listing of grievances

The user can customize this listing using the options above, and then can perform updates individually or in a multiple fashion using checkboxes and the fields in the Multi-features section. Almost every field in the listing contains a link to some action or page, with some of the links as javascript, others as simple column links, and others as manually coded links with manually coded parameters.

The goal of this display is to present the user with the most customizable and pertinent information in one screen that enables direct and immediate access to the most commonly used information and functions. The user therefore spends the minimum amount of time sifting through hundreds of thousands of records and scores of fields within those records and can address the immediate issue readily. Air traffic controllers adore extreme efficiency so it is necessary to squeeze maximum capability into the smallest desktop space.

Simple Column Link with Parameters

Figure 2 shows how the EDIT column in the report is defined. This is a simple column link with parameters. Note that the parameters include both a data field (#GRID#, the primary key for each grievance) as well as plain text and numbers. In this case the item :P8_RET_PAGE is a hidden item that page 8 uses to return the user to the original page, using a branch whose target is &P8_RET_PAGE..

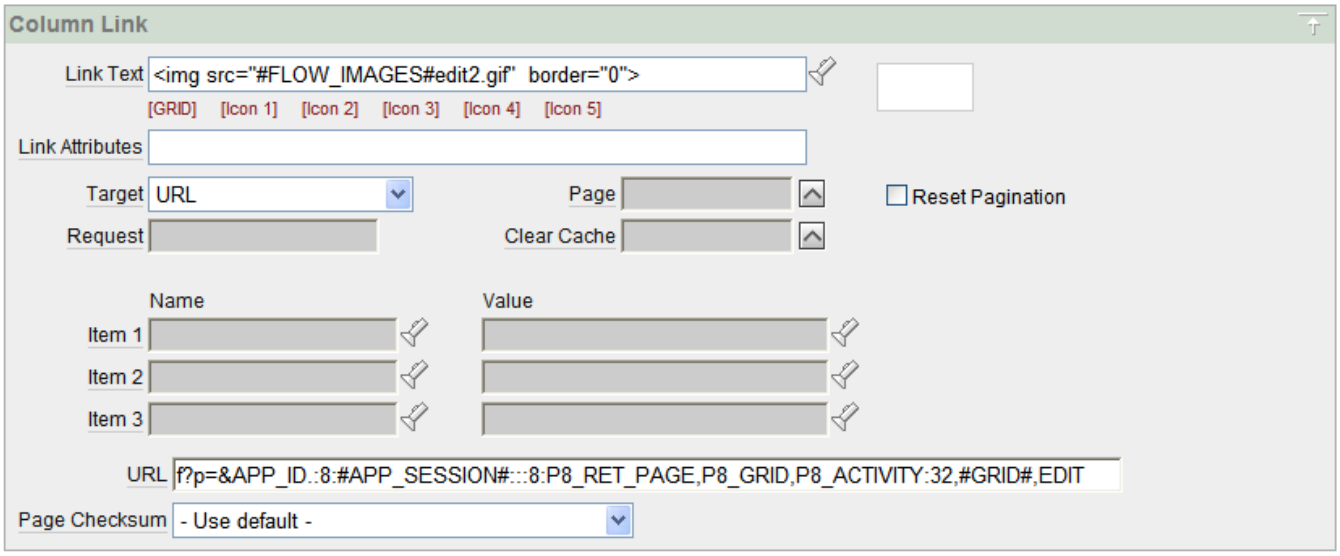

Figure 2. Simple column link for editing grievances

The column link uses the declarative tool that ApEx provides to automate the link creation. This comes in handy for the less experienced users, for rapid development, and for keeping the application simple and organized.

Manual column link with concatenation

You notice that the NATCA/FAA column in Figure 1 has in some cases two text strings, one of which is a link and the other plain text. This display cannot be created using the automated column link because that would cause all text in the column to be a link. To achieve the split effect that provides more visual clarity, the link has to be coded manually into the SQL that defines the region. The following code accomplishes this.

```
select '<a href="' || htmldb_util.prepare_URL('f?p=&APP_ID.:8:' || :APP_SESSION ||
':::8:P8_DUP_GRID,P8_RET_PAGE,P8_ARTICLE:' || g.grid || ',32,' ||g.article) || '">' || g.natca ||
'</a><br /><span style="font-size:8pt">' || g.faanum || '</span>' "NATCA/FAA" from grievance g
```

Note how the plain text is formatted through the span tag. Also note the use of the **`htmldb_util.prepare_URL`** function. This function is used to add security to the application. It appends a checksum to the end of the URL and this checksum is used by ApEx to determine if the entire link is legitimate or has been created by a user attempting to tamper with the application. This is only required because of the manual column link. Otherwise, ApEx would provide this checksum automatically via internal processes when the developer has enabled session state protection declaratively in the Shared Components section of the development environment.

Manual javascript link

The Grievance Regarding column in Figure 1 is a link to provide a pop-up window with a printable version of the grievance. Since this field also contains plain, non-linked text, the link is again written into the SQL as follows:

```
select '[BU: ' || nvl(b.buid, 'None') || '] '<a href="javascript:myPopUp(''' ||
htmldb_util.prepare_URL('f?p=&APP_ID.:9:' || :APP_SESSION || '::::P9_GRID:' || g.GRID) || ''')">' ||
g.topic || '</a>' || gr_groupid(g.grid) "Topic" from grievance g, gr_bu b where g.bu_id = b.id (+)
```

The **`htmldb_util.prepare_URL`** function is again used to append a checksum to the URL and prevent tampering. Notice also that plain text fields exist both before and after the linked string. This helps present the maximum amount of information to the user in the smallest space. The ability to render this column with both plain text and hypertext gives it a more organized look that is required to avoid overwhelming the user with information.

Composite data column

Another method for maximizing the use of space in the report is the use of conditional fields that display different types of data In the previous two examples, plain text was concatenated with hypertext. But there's no reason to limit the options of

what can be displayed. In the column labeled “Date subm’d or days left”, as shown again in Figure 3, a conditional combination of dates, numbers and images can be displayed based on the data.

To generate this display, a lengthy query is used using both decode and case functions.


```
select decode(g.status_id, 1, decode(g.date_sub_2, null, trunc(g.u_action_2) -
trunc(sysdate) || ' ' ||
case
when (g.u_action_2 - sysdate) > 7 then '<img src="/c/greenN.gif" alt="NATCA must submit
this by ' ||
to_char(g.u_action_2, 'MM/DD/YY') || '">'
when (g.u_action_2 - sysdate) > 3 then '<img src="/c/yellowN.gif" alt="NATCA must
submit this by ' ||
to_char(g.u_action_2, 'MM/DD/YY') || '">'
when (g.u_action_2 - sysdate) > 0 then '<img src="/c/redN.gif" alt="NATCA must submit
this by ' ||
to_char(g.u_action_2, 'MM/DD/YY') || '">'
when (g.u_action_2 - trunc(sysdate)) = 0 then '<img src="/c/red_today.gif" alt="NATCA
must submit this today">'
when g.u_action_2 is null and date_rec_1 is null then '<img src="/c/greenN.gif"
alt="FAA has not replied at the E1 level">'
else '<img src="/c/past.gif"  border="0" alt="NATCA failed to submit this by ' ||
to_char(g.u_action_2,
'MM/DD/YY') || '">'
end,
to_char(g.date_sub_2, 'MM/DD/YY')), 'Closed') "DATE_SUB" from grievance g
```

Figure 3. Composite data column

Use of PLSQL

It becomes apparent that nested decodes and case clauses add a great deal of complexity to the query and might be better handled by PLSQL. There are performance reasons to switch to PLSQL-generated SQL. In addition, once you introduce different types of data into a column any kind of simple sorting method will typically fail.

For example, the user might like to see the grievances sorted by any of the following criteria:

- those requiring the most immediate action first,
- grievance number,
- those requiring immediate action by the employer,
- date submitted, or
- other complicated sorting schemes.

Since the displayed columns do not neatly conform to some of the sorting schemes, it becomes necessary to write the ORDER BY clause dynamically. This is accomplished by converting the entire SQL Report Region to a PLSQL-generated SQL Report Region. The first step in this process is easy thanks to the declarative ApEx environment. It involves simply reselecting the region type as shown in Figure 4.

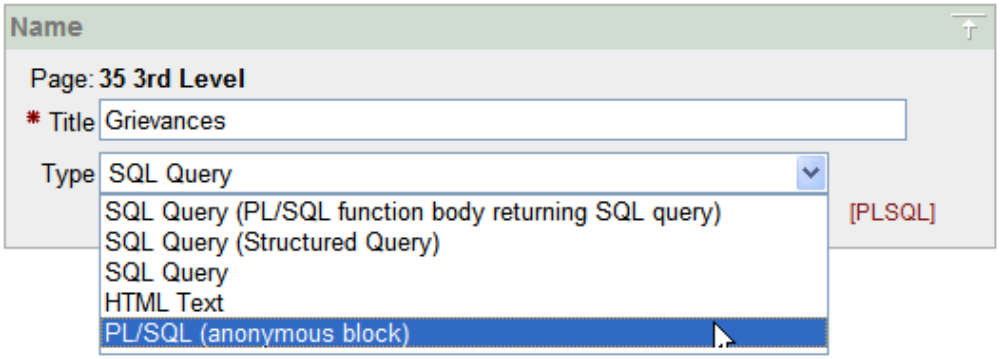

Figure 4. Converting to a PLSQL-generated SQL Report Region

Next, the code must be redesigned as PLSQL to generate the appropriate SQL. A simple example of this process is shown. Note the use of the 10g quoting syntax to clarify the string that is to be generated.

```
declare
p_sql varchar2(32767);
begin
p_sql := q'! select grid from grievance !';
return p_sql;
end;
```

In this case the simple query **`select grid from grievance`** will be generated by this PLSQL. The process of converting the SQL region source into a PLSQL block mostly involves inserting the existing query into the 10g quotes and then building the code to append various WHERE and ORDER BY clauses to the resulting SQL. In the current example the code looks like this:

```
case
when :P32_SORT = 1 then p_sql := p_sql || q'! order by trunc(g.reply_by_2), trunc(g.date_sub_2) nulls
last,
trunc(g.u_action_2) nulls last, substr(g.natca,4,3), g.natcasub !';
when :P32_SORT = 2 then p_sql := p_sql || q'! order by trunc(g.date_sub_2) nulls first,
trunc(g.u_action_2)
nulls last, trunc(g.reply_by_2), substr(g.natca,4,3), g.natcasub !';
when :P32_SORT = 3 then p_sql := p_sql || q'! order by substr(g.natca,1,2), substr(g.natca,4,3),
natcasub !';
when :P32_SORT = 4 then p_sql := p_sql || q'! order by g.faanum nulls last, substr(g.natca,1,2),
substr(g.natca,4,3), natcasub !';
when :P32_SORT = 5 then p_sql := p_sql || q'! order by g.date_sub_2 nulls first, u_action_2 !';
when :P32_SORT = 6 then p_sql := p_sql || q'! order by g.rep !';
else null;
end case;
```

By using PLSQL we are able to provide the user with sorting schemes of virtually any level of complexity.

The other benefit of using PLSQL is that the ability to selectively include or exclude various WHERE clauses gives a performance boost. In the GATS application, the user typically can choose from a large number of filtering options. Without a PLSQL-generated query, each of these options must be included in the SQL as a WHERE clause even if the criteria is all-inclusive. This clump of code looked like this:

```
where g.gr_status = 2
and g.status_id = p.id
and ((g.faanum like '%'||:P32_FAANUM||'%' and :P32_FAANUM is not null) or :P32_FAANUM is null)
and ((:P32_FACILITY_ID != 0 and g.facility_id = :P32_FACILITY_ID) or (:P32_FACILITY_ID = 0 and
(g.facility_id in (select id from gr_facility_lookup where region_id = (select y.region_id from
gr_facility_lookup y, gr_emp z where upper(z.username) = :APP_USER and z.facility_id = y.id)) or
g.facility_id = 3)))
and (g.status_id = 1 or g.close_date > sysdate - :P32_DAYS)
and ((g.date_sub_2 is not null and :P32_NF = 1) or (g.date_sub_2 is null and :P32_NF = 2) or (:P32_NF =
0))
and g.status_id != 2
and (:P32_REP is null or lower(g.rep) like '%' || lower(:P32_REP) || '%')
and (:P32_NATCA is null or upper(:P32_NATCA) = g.natca)
and (:P32_GRIEVANT is null or upper(g.grievant) like '%'||upper(:P32_GRIEVANT)||'%')
and (:P32_TOPIC is null or lower(g.topic) like '%' || lower(:P32_TOPIC) || '%')
```

By using PLSQL, these clauses can all be removed unless the user has entered some value for any of the filter fields. In almost no case will the user have entered values for more than two or three of the fields, so that every query generated by PLSQL will include significantly fewer WHERE clauses. Here is an example of the PLSQL that is used to generate the where clauses:

```
if :P32_REP is not null then
p_sql := p_sql || q'! and lower(g.rep) like '%' || lower($P32_REP) || '%' !';
end if;
if :P32_NATCA is not null then
p_sql := p_sql || q'! and g.natca like '%'||upper($P32_NATCA)||'%' !';
end if;
```

So if the user hasn't entered anything into either of these two text field page items, there will be no WHERE clause in the SQL that generates the region. Upon close inspection, you notice that a **$** has been substituted where there should be a colon. Within PLSQL the colon indicates a bind variable, so the colon must be substituted back into the SQL after the string has been constructed.

So the code to create SQL for the Topic column now looks like this:

```
p_sql := p_sql || q'! '[BU: ' || nvl(b.buid, 'None') || '] <a ref="javascript$myPopUp(''f?p=&APP_ID.$9$'
|| $APP_SESSION || '$$$$P9_GRID$' || g.GRID || ''')" &F168_PRINT.>' || g.topic || '</a>' ||
gr_groupid(g.grid) "Topic", !';
```

The **$** is replaced at the end of the PLSQL block:

```
return replace(p_sql,'$',':');
```

When the page is rendered the PLSQL runs and creates the SQL, and then the SQL is used to create the region. Since the region contains the same columns every time, the Report Attributes tab is still used to format the columns in the display, including everything from date formats to LOVs to column links.

Apps that Build Themselves

With 15,000 members spread out over 50 states and beyond, it's important to our organization to find ways to keep track of both attitudes and activities at the ground level. Surveying is a key part of our strategy to accomplish this.

In the past, we used inexpensive subscription tools like Survey Monkey. These allow the user to create surveys and then deploy via email or publication of a unique URL. The limitations of these tools are that they are not integrated with the organization's database and they exist outside of the organization's IT system. As a result, some of your internal data – names, email addresses, and the data itself – are exposed to a third party.

In addition, users are directed to a site outside of your system and they cannot use an existing login to access the system. If you make the survey public, the data integrity is limited since the identity of the user is not assured. As is always the case, having direct access to the code behind the system allows far greater control including advanced validation of user input, custom email functions to users, email notification of specific results, etc. Another advantage of bringing the system in-house is the ease with which survey authors can select recipients based on internal criteria such as organizational roles or access levels.

Development of the concept

Our original intent was to actually build surveys one by one. It didn't take long to realize that this was a labor intensive method. Having seen Survey Monkey it was clear that our road to glory was going to be an automated tool.

If you hold the survey author to a very rigid format, you can build an app that uses existing items on blank pages. For instance, you might have a page with 6 items. Two of these are radio buttons with yes-no answers, two are select items, and two are text fields. On the survey author page, you enable the author to enter the label for the yes-no questions, the label and answers for the select list items, and the label for the text items. Using substitution strings and a query for the select list answers, you can create the survey.

Of course this is a woefully inadequate method since it gives the author extremely limited capability.

In order to give the author maximum flexibility, we realized the author needed the ability to create new items. If you poke around in the application export file from one of your apps you will find PLSQL like the following:

```
wwv_flow_api.create_page_item(
  p_id                      => 6005523245719266 + wwv_flow_api.g_id_offset,
  p_flow_id                 => wwv_flow.g_flow_id,
  p_flow_step_id            => 1,
  p_name                    =>'P1_PREVIOUS_PAGE',
  p_data_type               => 'VARCHAR',
  p_accept_processing       => 'REPLACE_EXISTING',
  p_item_sequence           => 20,
  p_item_plug_id            => 72280136279022072+wwv_flow_api.g_id_offset,
  p_use_cache_before_default => 'YES',
  p_item_default_type       => 'STATIC_TEXT_WITH_SUBSTITUTIONS',
  p_source_type             => 'STATIC',
  p_display_as              => 'HIDDEN',
  etc.
```

We decided to investigate whether we could insert ApEx export code into a page process to allow the app to build new page elements at the request of users, on the fly.

Initial testing showed it could be done. We had to understand each of the parameters of the various procedures in the API. Whenever there was a question, it was a fairly simple matter to create a page item (branch, button, etc.) normally, export the page and then see how each change was translated into parameters used by the API. The following is a discussion of some of these:

p_id => 6005515200719266 + wwv_flow_api.g_id_offset,

Every page, item, branch, button, process, etc. has an ID value. Since every aspect of an ApEx application is stored in the database, one would assume that a page item is represented in the database as a row in a table, and all of the other parameters are columns. This ID value is simply the primary key in that table. For exporting and importing applications, ApEx has its own convention for assigning the value of the ID field. For our purpose, however – to simply obtain an available value to use as the ID for this row – we can pull a value from the internal ApEx sequence:

p_id => wwv_flow_id.next_val

p_flow_id => wwv_flow.g_flow_id,

Most will recognize this as the ApEx application ID. When you query the **apex_activity_log**, you will see this field and will recognize your apps by it.

p_flow_step_id => 1,

Most will also recognize this parameter as the page ID.

p_name => 'P1_PREVIOUS_PAGE',

The name of the item as entered by the developer in the page item editor.

p_data_type => 'VARCHAR',

The data type of the item seems to default universally to **VARCHAR**. We found that we could simply leave this parameter out of our process and it would default to null. This did not seem to affect our application.

p_accept_processing => 'REPLACE_EXISTING',

We take the default on this parameter.

p_item_sequence => 20,

This is simply the rendering sequence number of the item as entered by the developer in the page item editor.

p_item_plug_id => 72280136279022072+wwv_flow_api.g_id_offset,

Page plug is an ApEx name for a region. So this value must be the ID of the region selected in the page item editor. In our case, the region is constructed on the fly and we control its ID.

p_use_cache_before_default => 'YES',

This is the "Source Used" option in the source section of the page item editor. The two options are "Always, replacing any existing value in session state" and "Only when current value in session state is null". The value of "No" in this parameter corresponds the the former option, and "Yes" the latter.

p_item_default_type => 'STATIC_TEXT_WITH_SUBSTITUTIONS',

This is the value of "Default Value Type" in the page item editor. The other options are PL/SQL Function Body and PL/SQL Expression.

p_source_type => 'STATIC',

This is the value of "Source Type" in the page item editor. Other options include SQL Query, PLSQL Function Body, etc.

p_display_as => 'HIDDEN',

The "Display As" field in the page item editor. Other options include Text Field, Select List, Radio Group, etc.

The value **wwv_flow_api.g_id_offset** that appears frequently is an ApEx feature that helps to preclude over-writing any other ApEx element during an import from another schema. We have no need for it since we are operating within a schema.

These are just a few of the parameters that must be understood. By continuing in this process of defining each parameter used by the API's, we can create the knowledge base needed to program our application to build ApEx elements itself.

Once we learned we could create various elements of an ApEx page, we considered the general configuration of our app to allow for different levels of access. We wanted a configuration that specifically allowed the base user access only to those surveys they had been designated to take by author-level users.

Requirements

Our requirements were something like this:

- All users are authenticated.
- Two user levels, survey authors and survey takers. An author could also be a taker.
- Authors have the ability to individually or collectively select takers of each of their surveys from the membership database.
- Authors have a versatile method for selecting different classes of members.
- Authors can create survey questions of type yes/no, text field, text area, and multiple choice, both single and multiple answer.
- Photos can be associated with multiple choice answers.
- All users are provided access to all of the current surveys for which they have been selected.
- Some kind of result reporting must be provided, access to which could be author-only or authors and takers.

Various other requirements included automated notifications, selective email capability for authors, password change capability, the ability to opt out of surveys selectively by author, and other NATCA-specific features.

Given these requirements, especially those restricting access, it seemed apparent that the most natural way to do this was to create a new page for each survey. Onto each page the app would create regions, items, buttons, branches, and processes to accomplish the survey as defined by the user.

The only thing left at this point was to figure out how to do that!

Creating a Page

If you review an ApEx export file, you'll find code that creates pages. It takes a fair bit of tinkering to get comfortable, but this code can be modified and used in a page process in your ApEx app to generate a new page. We used it to generate new survey pages as follows.

```
wwv_flow_api.create_page(
  p_id                          => p_page,
  p_flow_id                     => wwv_flow.g_flow_id,
  p_tab_set                     => '',
  p_name                        => 'Survey: ' || p_survey_name,
  p_step_title                  => '',
  p_step_sub_title              => 'Page 1:'|| p_survey_name,
  p_step_sub_title_type         => 'TEXT_WITH_SUBSTITUTIONS',
  p_first_item                  => 'NO_FIRST_ITEM',
  p_include_apex_css_js_yn      =>'Y',
  p_help_text                   => ' ',
  p_html_page_header            => ' ',
  p_step_template               => '',
  p_required_role               => p_taker_auth + wwv_flow_api.g_id_offset,
  p_required_patch              => null + wwv_flow_api.g_id_offset,
  p_last_updated_by             => '',
  p_last_upd_yyyymmddhh24miss   => '',
  p_page_is_public_y_n          => 'N',
  p_page_comment                => '');
```

The value **p_page** is generated by the internal ApEx sequence: **wwv_flow_id.next_val.** The only other parameter of interest here is **p_required_role.** This value specifies the authorization scheme to be applied to this new page. To supply this value, we simply obtained the internal ID numbers of the authorizations from an export file and insert the appropriate value for each page as it is constructed.

Creating Regions, Items, Buttons, and Branches

Next up is a region. This code is used in our page process to create a region on the new page.

```
vId := wwv_flow_id.next_val;
p_region_id := vId;
wwv_flow_api.create_page_plug (
 p_id                          => vId,
 p_flow_id                     => wwv_flow.g_flow_id,
 p_page_id                     => p_page,
 p_plug_name                   => '<h1>'||p_survey_name||'</h1>',
 p_region_name                 =>'<h1>'||p_survey_name||'</h1>',
 p_plug_template               => 191260956775468936+ wwv_flow_api.g_id_offset,
 p_plug_display_sequence       => 10,
 p_plug_display_column         => 1,
 p_plug_display_point          => 'BEFORE_SHOW_ITEMS',
 p_plug_source                 => null,
 p_plug_source_type            => 'STATIC_TEXT',
 p_translate_title             => 'Y',
 p_plug_display_error_message  => '#SQLERRM#',

 p_plug_query_row_template     => 1,
 p_plug_query_headings_type    => 'QUERY_COLUMNS',
 p_plug_query_num_rows_type    => 'NEXT_PREVIOUS_LINKS',
 p_plug_query_row_count_max    => 500,
 p_plug_column_width           => '',
 p_plug_query_show_nulls_as    => ' - ',
 p_plug_display_condition_type => '',
 p_pagination_display_position =>'BOTTOM_RIGHT',
 p_plug_customized             =>'0',
 p_plug_caching                => 'NOT_CACHED',
 p_plug_comment=> '');
```

We maintain a table in the database containing metadata on the application, including the page numbers and names of existing survey pages. To determine the value for **p_page**, we query that table, obtaining the highest page number value, and add one.

Every survey page needs a button, and one is added using the following code.

```
vId := wwv_flow_id.next_val;
wwv_flow_api.create_page_button(
 p_id                  => vId,
 p_flow_id             => vAppId,
 p_flow_step_id        => p_page,
 p_button_sequence     => 10,
 p_button_plug_id      => p_region_id + wwv_flow_api.g_id_offset,
 p_button_name         => 'SUBMIT',
 p_button_image_alt    => 'Submit',
 p_button_position     => 'TOP_AND_BOTTOM',
 p_button_alignment    => 'LEFT',
 p_button_redirect_url => '',
 p_required_patch      => null + wwv_flow_api.g_id_offset);
```

The items are a bit more complicated. Each has different attributes assigned by the user. Those attributes are stored in the table **survey_order_questions** and are retrieved when the items are created. Meanwhile, the answers to multiple choice questions are stored in the table **survey_order_ans**. These are retrieved as well to create the radio items. The item names are standardized to conform to the format **Pxxx_Qyy**, where **xxx** is the page of the generated survey and **yy** is the question sequence number.

```
declare
cursor c1 is select question_id, question_num, question_type, question_text
from survey_order_questions where survey_id = :P6_SURVEY_ID
order by question_num;
cursor c2 is select id, answer
from survey_order_ans where question_id = p_question_id
order by id;
p_page         number(5);
p_item_name    varchar2(100);
p_seq_val      number(3,0) := 20;
```

```
begin
select max(page) into p_page                                -- get the last page used
from survey_order_list;
p_page := p_page + 1;
[code to create page, regions, etc.]

for a1 in c1 loop                                           -- item code creation begins
p_item_name := 'P' || p_page || '_Q' || p_current_qnum;     -- item names like P202_Q1, P202_Q2, etc.
p_question_id := a1.question_id;
  if a1.question_type = 1 then                              -- question_type 1 is multiple choice
    for a2 in c2 loop
    p_ans := p_ans || trim(a2.answer) || ';' || a2.id || ',';
    end loop;
    p_ans := rtrim(p_ans,',');
    vId := wwv_flow_id.next_val;
    wwv_flow_api.create_page_item(
         p_id                          => vId,
         p_flow_id                     => vAppId,
         p_flow_step_id                => p_page,
         p_name                        => p_item_name,
         p_item_sequence               => p_seq_val,
         p_item_plug_id                => p_region_id,       -- determined in region creation code
         p_use_cache_before_default    => 'NO',
         p_prompt                      => '<b>'||a1.question_num||'. '||a1.question_text||'</b>',
         p_source                      => p_ans_sql,
         p_source_type                 => 'QUERY',
         p_display_as                  => 'RADIOGROUP',
         p_lov                         => 'STATIC2:'||p_ans,
         p_lov_columns                 => 1,
         p_label_alignment             => 'ABOVE',
         p_field_alignment             => 'LEFT');
  else
  [other code for different item/question types]
  end if;
p_seq_val := p_seq_val +10;
end loop;
```

Buttons and branches are created in similar fashion. A sticking point with branches is that they need to include the value of **`SESSION`**. But if your page process includes the string **`&SESSION.`**, then the session ID will be substituted and become part of the branch. With your branch action parameter within the branch create procedure set to this:

```
p_branch_action=> 'f?p=&APP_ID.:10:&SESSION.',
```

You will see that when the branch is created, ApEx will actually run the following:

```
p_branch_action=> 'f?p=&APP_ID.:10:4676970450940099',
```

In this case, the hard-coded session ID is the value taken from the session in which the user activated the process. This situation was tricky to detect because when you examine the branch that is created on the newly-generated page, there is no indication of the presence of a hard-coded session ID. What led us to the problem was that when we accessed the page as users of the app, we would frequently experience unexpected logouts. Only by creating an export of the generated page can you see the hard-coded value instead of the substitution string.

To get the substitution string into the actual branch, a substitution character is used to prevent ApEx from making the substitution prematurely:

```
p_session varchar2(100) := '$SESSION.';
```

The character is replaced when the branch is created:

```
p_branch_action=> 'f?p=&APP_ID.:10:' || replace(p_session,'$','&'),
```

Creating the Process

The PLSQL to create the page process was created by working backwards. Survey responses are recorded in the table **`SURVEY_EMP_ANS`**, whose three columns are **`QUESTION_ID`**, **`EMP_ID`**, and **`ANS_ID`**. So the page process has to either insert a new row or update one (if the user is revisiting their answer). If the page had been created manually, the process might look like this:

```
insert into SURVEY_EMP_ANS (QUESTION_ID, EMP_ID, ANSWER) values (1234, f_emp_id(:APP_USER), :P212_Q1);
```

where **f_emp_id** is a function that retrieves the user's **EMP_ID** and item **P212_Q1** is question 1 on page 212. The tricky part here is determining what answer the user had supplied, because this process is being generated by another process when the survey is published. We wanted to generate on each survey page a single cursor that would loop through all of the questions in that survey, read the answers the user had supplied, and insert or update those answers into the table **SURVEY_EMP_ANS.**

Again working backward, we built the following code to be the core of the submit process on survey pages, where the **c1** cursor loops through the questions in the selected survey. Note that the use of a standardized item name comes in handy here.

```
for a1 in c1 loop
      l_sql := 'BEGIN  '
            || ' :l_val :=  v(''P'  ||:APP_PAGE_ID||  '_Q'||p_q_num  || '''); '
            || 'END;';
      EXECUTE IMMEDIATE l_sql USING OUT p_page_item;
      select count(*) into p_count from survey_emp_ans
      where question_id = a1.question_id and emp_id = p_empid;
      if p_count = 0 then
         insert into survey_emp_ans (question_id, emp_id, answer) values
         (a1.question_id, p_empid, p_page_item);
      else
         update survey_emp_ans set answer = p_page_item
         where question_id = a1.question_id and emp_id = p_empid;
      end if;
      p_q_num := p_q_num + 1;
   end loop;
```

The **EXECUTE IMMEDIATE** SQL that would run when the user submits the page would then looks like this:

```
BEGIN
:l_val := v('P208_Q1');
END;
```

This retrieves the user's answer. The remaining code then performs the insert or update.

The above process must be generated when the survey author publishes the survey. The publishing page therefore has to have a process to generate that process. Here it is:

```
p: = p || 'declare' || chr(10) ||
'p_survey_id number(5,0);' || chr(10) ||
'p_empid number(7,0);' || chr(10) ||
'p_page_item varchar2(4000);' || chr(10) ||
'p_q_id number(5,0);' || chr(10) ||
'p_q_num number(3,0):= 1;' || chr(10) ||
'p_max_q_num number(3,0);' || chr(10) ||
'l_sql VARCHAR2(1000);' || chr(10) ||
'p_count number(1);' || chr(10) ||
'' || chr(10) ||
'cursor c1 is' || chr(10) ||
'select question_id' || chr(10) ||
'from survey_order_questions' || chr(10) ||
'where survey_id = p_survey_id' || chr(10) ||
'order by question_num;' || chr(10) ||
'' || chr(10) ||
'begin' || chr(10) ||
'' || chr(10) ||
'p_empid := gr_empid(:APP_USER);' || chr(10) ||
'' || chr(10) ||
'select survey_id into p_survey_id' || chr(10) ||
'from survey_order_list' || chr(10);

p:=p || 'where page = :APP_PAGE_ID;' || chr(10) ||
'' || chr(10) ||
'   for a1 in c1 loop' || chr(10) ||
'' || chr(10) ||
'      l_sql := ''BEGIN  ''' || chr(10) ||
'            ||  '' :l_val :=  v(''''P''   || :APP_PAGE_ID ||   ''_Q'' || p_q_num   ||  ''''''); '' '
|| chr(10) ||
'            ||  ''END;'';' || chr(10) ||
'' || chr(10) ||
'      EXECUTE IMMEDIATE l_sql USING OUT p_page_item;' || chr(10) ||
```

```
'' || chr(10) ||
'      select count(*) into p_count from survey_emp_ans ' || chr(10) ||
'      where question_id = a1.question_id and emp_id = p_empid;' || chr(10) ||
'' || chr(10) ||
'      if p_count = 0 then' || chr(10) ||
'         insert into ';

p:=p || 'survey_emp_ans (question_id, emp_id, answer) values' || chr(10) ||
'          (a1.question_id, p_empid, p_page_item);' || chr(10) ||
'      else' || chr(10) ||
'         update survey_emp_ans set answer = p_page_item ' || chr(10) ||
'         where question_id = a1.question_id and emp_id = p_empid;' || chr(10) ||
'      end if;' || chr(10) ||
'' || chr(10) ||
'      p_q_num := p_q_num + 1;' || chr(10) ||
'   end loop;' || chr(10) ||
'end;';
```

This process is the final major piece in giving our app the ability to "build itself".

At the request of the user, our app can create new pages and new regions within those pages. Into these regions our app can insert page items containing user-defined surveys questions. It can create the button to enable survey takers to submit their answers, and it can create the process to record the data in each survey to the database. It can also create the branch that fires after the user submits their answers to the survey.

Finishing Touches

There are a number of issues that are not included here for the sake of brevity but are worth mentioning. These include the generation of a results page, a process to create validations, securing the author-side of the app from the rest of the users through branching and conditions, and the mechanics of choosing survey recipients.

One lesson we did learn – fortunately without much pain – was that if your app generates its own pages, you don't want to import the whole app into prod from dev. After we did this, we realized we had over-written some of our users' surveys. It doesn't take a rocket scientist to figure out that the pages that had been created in the prod instance would disappear! The ability to export and import individual pages in ApEx has never been of greater value for us.

Conclusions

ApEx is a powerful, versatile and convenient tool, and developers with just modest experience in SQL and PLSQL can really get going in a hurry. What we tried to show in this paper is that, for more advanced users the power of ApEx can be extended greatly through the use of the two approaches described.

In the earlier section, we explored the possibilities of PLSQL-generated Report Regions. We looked at how these regions could be elaborately customized to:

- Eliminate SQL at runtime that might hinder performance
- Allow for complex sorting
- Make it possible to generate composite report columns containing mixtures of data types and images

In the latter section, we looked at how to use ApEx' own import-export code to enable your apps to "multiply" themselves. No doubt, ApEx itself uses this code to generate your apps when you are building them. So we leveraged the legendary characteristic of ApEx - that it is "built in itself" – to build apps that can be used to build other apps. ApEx itself is an example of such an app, if that isn't too much circular thinking for you!

Both of the approaches described in this paper provide important and powerful capabilities to the ApEx developer.

You can contact the authors if you like:

- Bill Holtzman at skyworker@comcast.net or 703-403-0139
- Steve Schreck at stlsjschreck@natca.net or 636-399-4549