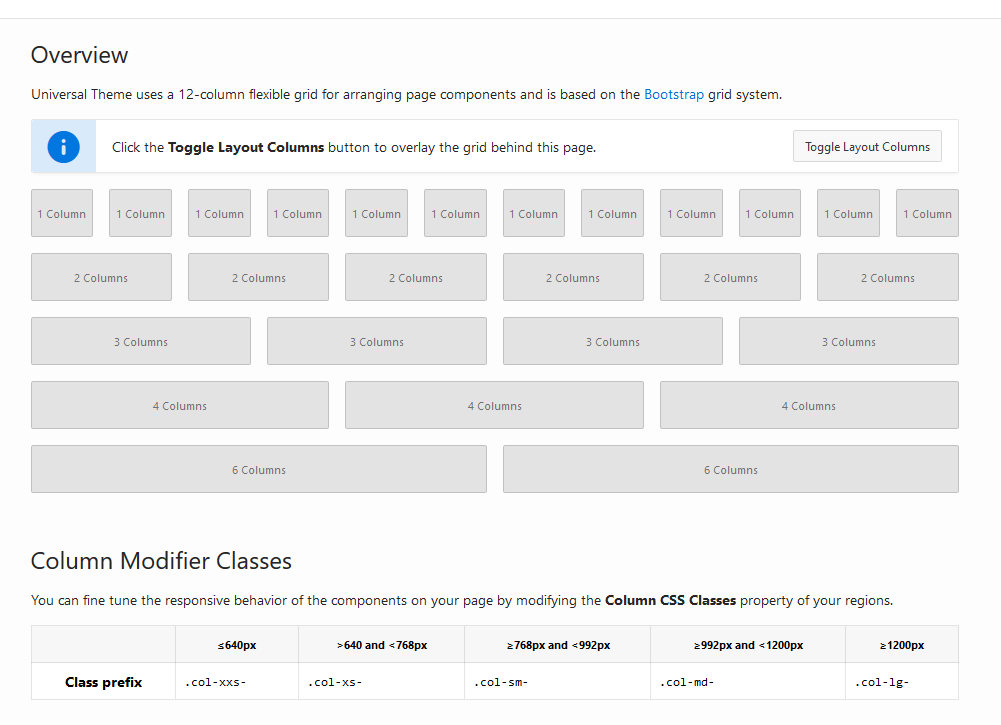
The set of possible region positions is determined by the page’s *template*.

The regions in a position are organized in a grid; each region is assigned to a row and column of that grid.

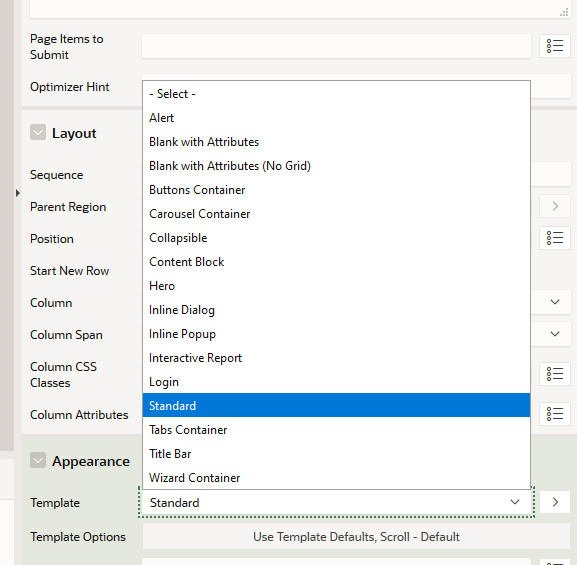
APEX uses the values of the Column and Column Span properties (shown in Figure 3-12) to determine the offset and width of each region in a row.

<https://apex.oracle.com/pls/apex/apex_pm/r/ut/grid-layout>



The Column property specifies the offset of the region within its row. APEX divides the width of a position into 12 “grid points”; the value of Column is a number from 1 to 12, denoting one of those points. A value of 1 specifies that the region begins at the far left of the position; a value of 7 specifies that the region begins in the position’s center. The Column Span property specifies the width of the column. Its value is also a number from 1 to 12, denoting the width in grid points.

The Standard template, which is the most common, displays the region’s title in a shaded border across the top. Many of the other templates have special purposes, such as Alert, Inline Dialog, Login, Title Bar, and Wizard Container. Some of these positions (namely, After Header, Body 1, Body 2, Body 3, Footer, and Before Footer) are *legacy* positions.



In Shared Components, An application file can be referenced from a specific application only, whereas a workspace file can be accessed by any application in the workspace. To display your image file using an HTML <img> tag, you must specify its location on the server. However, APEX will not tell you where it stored your image files; instead, it provides you with built-in variables. There are two variables, APP\_IMAGES and WORKSPACE\_IMAGES, whose values hold the path to your application images and workspace images on the server. You use substitution strings to access the value of these variables.

**<img src="&APP\_IMAGES.employees.jpg">**

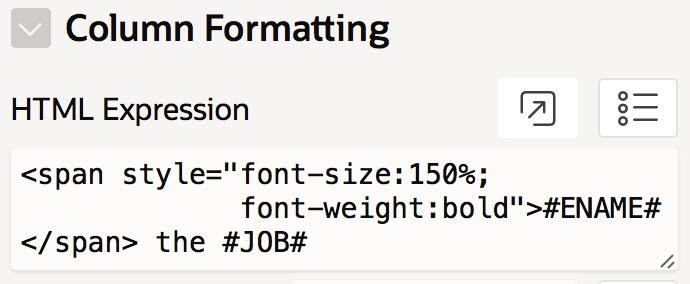
If a report’s source query has an order by clause, the records are sorted in that order and cannot be altered by the user. This correspondence between the result value (the department number) and its display value (the department name) is called a *list of values*.

 If you are not familiar with the Oracle format mask syntax, you can get a reasonable understanding by examining the various masks. For example, 9 denotes an optional digit, 0 a required digit, G a thousands’ separator, D the decimal point, and so on.

The mask for HIREDATE was created similarly. An examination of the various date format masks shows that DD denotes the day, Mon the three-character abbreviation of the month, and YYYY the four-digit year.

**Column Formatting**

Columns of type Plain Text have a section called Column Formatting, which has a property named HTML Expression. This property allows you to format a column value by wrapping it in an HTML expression.



**<span style="font-size:150%;**

**font-weight:bold">#ENAME#</span> the #JOB#**

It is possible to extract the year from within an SQL query. The idea is to have the SQL source query compute a new column, whose value is the hire year. The following query does the job:

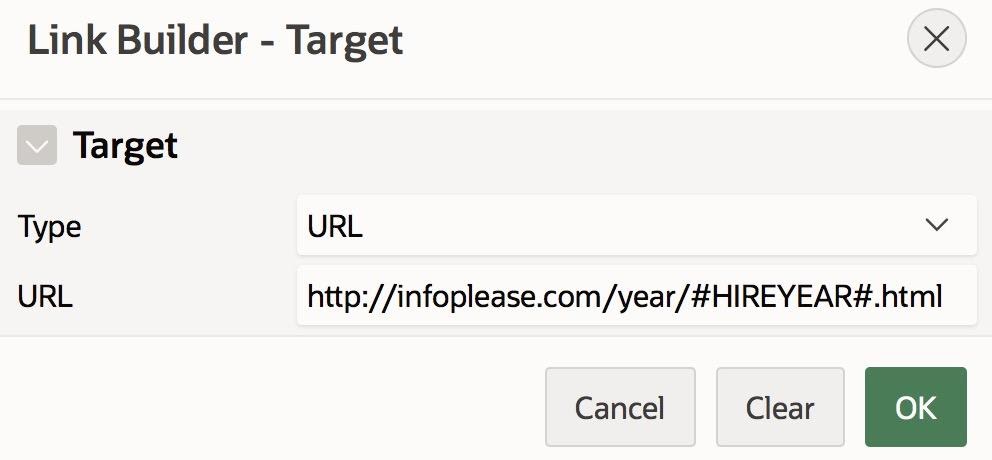
select e.\*, extract(year from e.HireDate) as HireYear

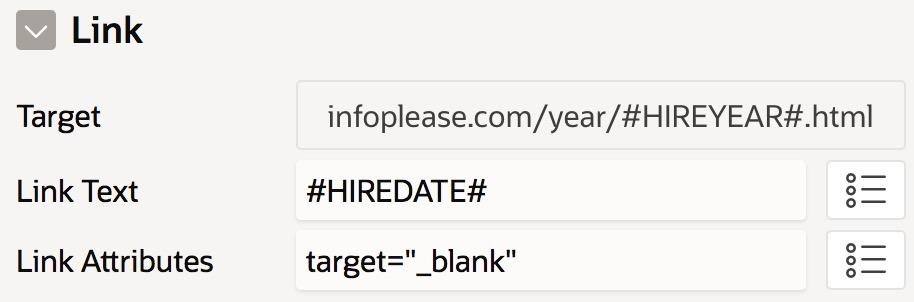
from EMP e

A consequence of using this new source query is that HIREYEAR will now be a column of the report.

<a href="http://www.infoplease.com/year/#HIREYEAR#.html"

     target="\_blank">#HIREDATE#</a>





For example, the following expression makes all salary values red:

<span style="color:red">#SAL#</span>

But how do you write an expression to make only some of the values red? Because HTML by itself cannot perform calculations, you must again rely on SQL to do it for you. The idea is to have the SQL source query compute another new column, called SalColor, whose value is the color of the salary value. Here is the desired query (new code is in bold):

select e.\*, extract(year from e.HireDate) as HireYear,

**case when e.Sal>2500 then 'red' else 'black' end as SalColor**

from EMP e

**using =>** <span style="color:#SALCOLOR#">#SAL#</span>

You can apply the same technique to the OFFSITE column. In the formatted report, an Offsite value shows an image of a green check mark if its value is **Y** and a red check mark otherwise. Assume that these two images have been saved to APEX as the application files checkgreen.jpg and checkred.jpg. Then a value of **Y** should be formatted as the following HTML expression:

<img src="&APP\_IMAGES.checkgreen.jpg">

The value of **N** should be formatted as follows:

<img src="&APP\_IMAGES.checkred.jpg">

The way to distinguish between these two file names is to extend the SQL source query so that it computes a new column, named OffsiteImage, whose value is either 'checkgreen.jpg' or 'checkred.jpg'. The SQL query now looks like this (new code is in bold):

**select e.EmpNo, e.EName, e.Job, e.HireDate, e.Sal, e.DeptNo, e.Offsite,**

**extract(year from e.HireDate) as HireYear ,**

**case when e.Sal>2500 then 'red' else 'black' end as SalColor,**

**case when e.Offsite = 'Y' then 'checkgreen.jpg'**

**else 'checkred.jpg' end as OffsiteImage**

**from EMP e**

The rule of thumb is that the source query should specify *what* you want displayed, whereas the column properties should specify *how* you want them displayed.

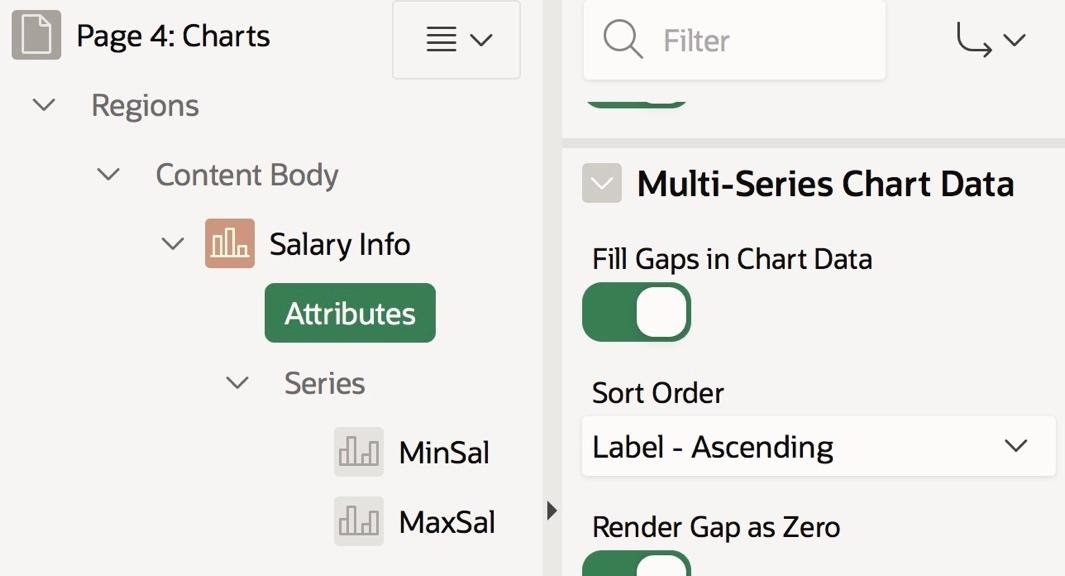
Every chart needs a *source*, which is an expression that specifies the chart’s data points. A common source is an SQL query. If the chart has N series, then the query will have N+1 columns—one column contains the labels, and the other columns contain the values for each series.

select Job, min(Sal) as MinSal, max(Sal) as MaxSal

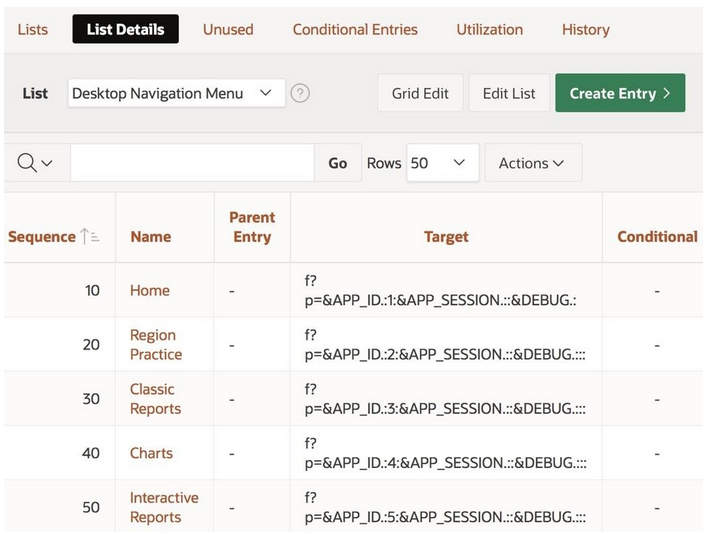
from EMP

group by Job

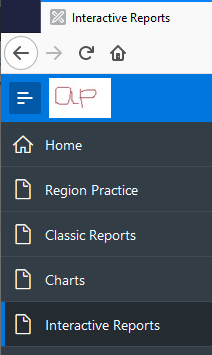
order by MinSal

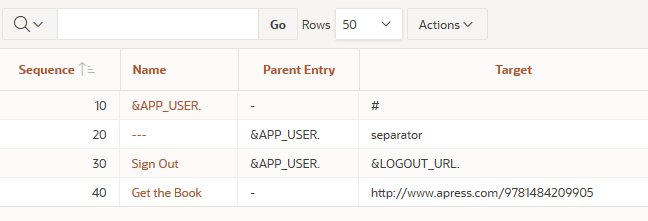


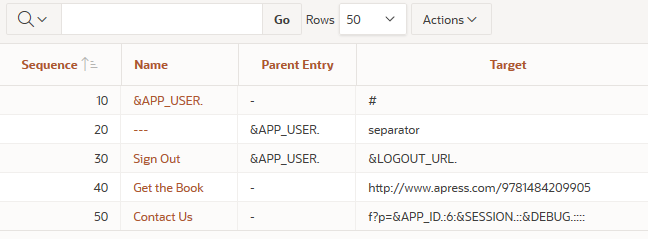
You have already encountered three navigational features in your use of APEX: links, breadcrumbs, and the navigation menu

****

**Cooresponds 1 to 1 with your applications list references**

** from  to**

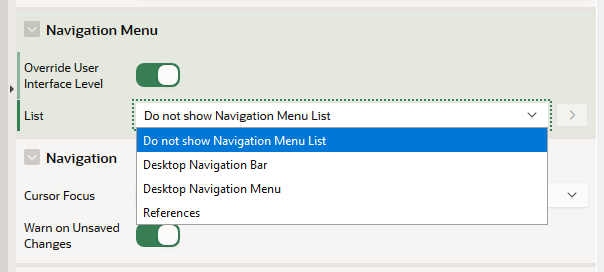
****

****

To change the way a list is displayed, you must change its template.

 References Using Cards, References Using Links, References as a Navigation Bar, and Navigation Menu.

But what if you want the menu hidden on just a single page? The answer is to use the Navigation Menu properties of the page, as shown in Figure 4-19.

****

**That works great !!**

You saw earlier how to hide the navigation menu on all pages of an application by setting the global property Navigation Menu List to null. But what if you want the menu hidden on just a single page? The answer is to use the Navigation Menu properties of the page, as shown in Figure 4-19.

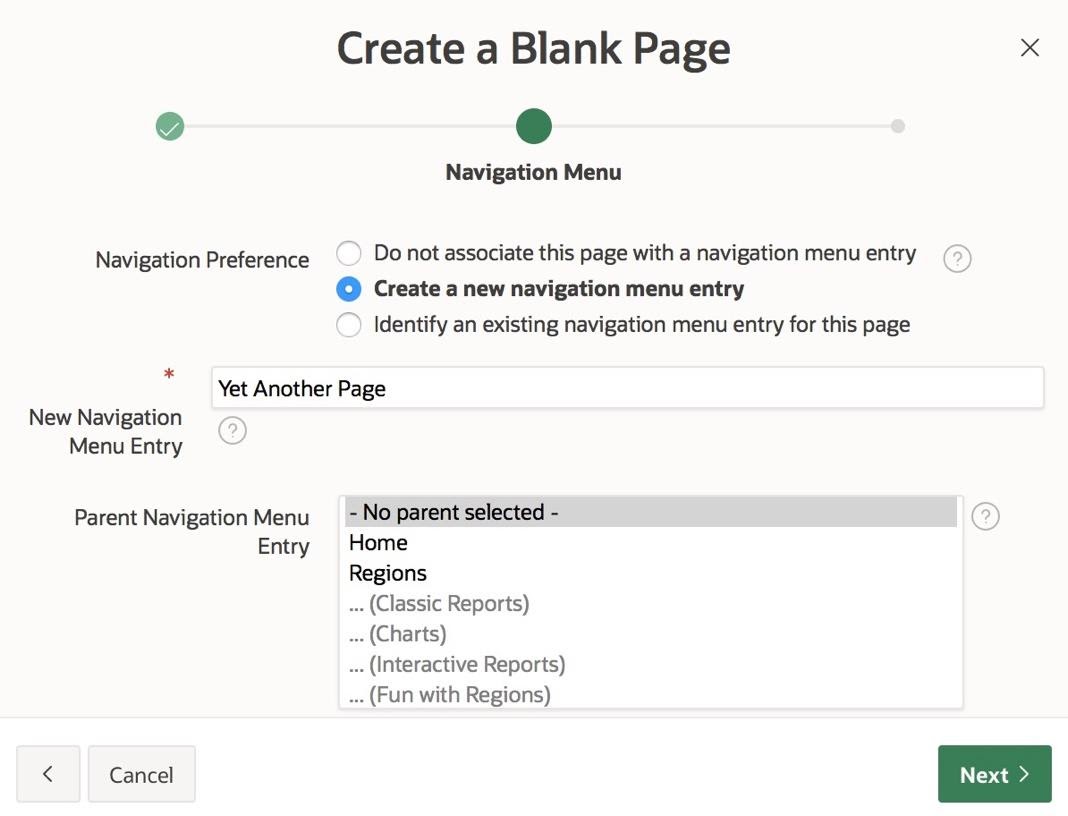
Storing links in a list and displaying them in a list region is an important technique; it is far superior to manually formatting the links in HTML, as you did on the References region of the home page. One reason is that the template of a list region takes responsibility for formatting the list entries;

a list region separates the content (the source list) from the formatting (the list template). Writing HTML code to implement a list is not only tedious, but it also intermixes content and formatting. Any decision to change the format of the entries would require you to rewrite the HTML.

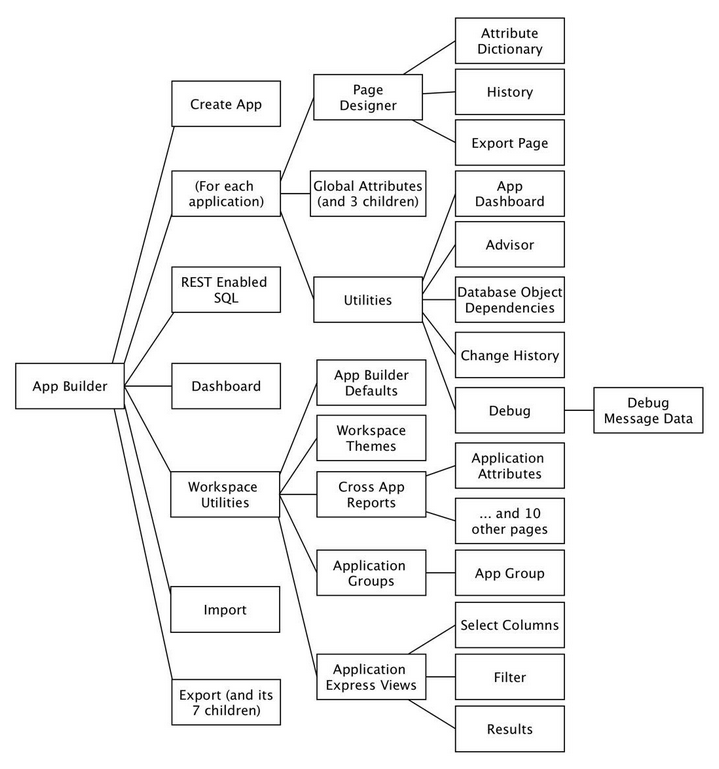
In a hierarchical list, each list entry can have a *parent* entry. A list entry without a parent defines a hierarchy and is called the *root* of that hierarchy. A hierarchy can have several levels.

This book shall follow the convention that a root having a single child will have that child be its target, whereas a root having several children will have a null target.

In particular, the selectable entries are the ones that either are already parents or are “potential parents” in the sense that they have no target. Most of the time, the parent you want will be selectable. But if you want your new entry to be a child of one of the grayed-out entries, you must do so via the list manager.



A more recent design idea is to keep the hierarchical organization but display all menu entries at once, in what is called a *mega menu*. Go to the application’s Navigation Menu section and set the Position property to Top and the List Template property to Top Navigation Mega Menu. Typically, the set of all pages in an application is also hierarchically structured. An explicit rendering of this page hierarchy is called a *site map*.



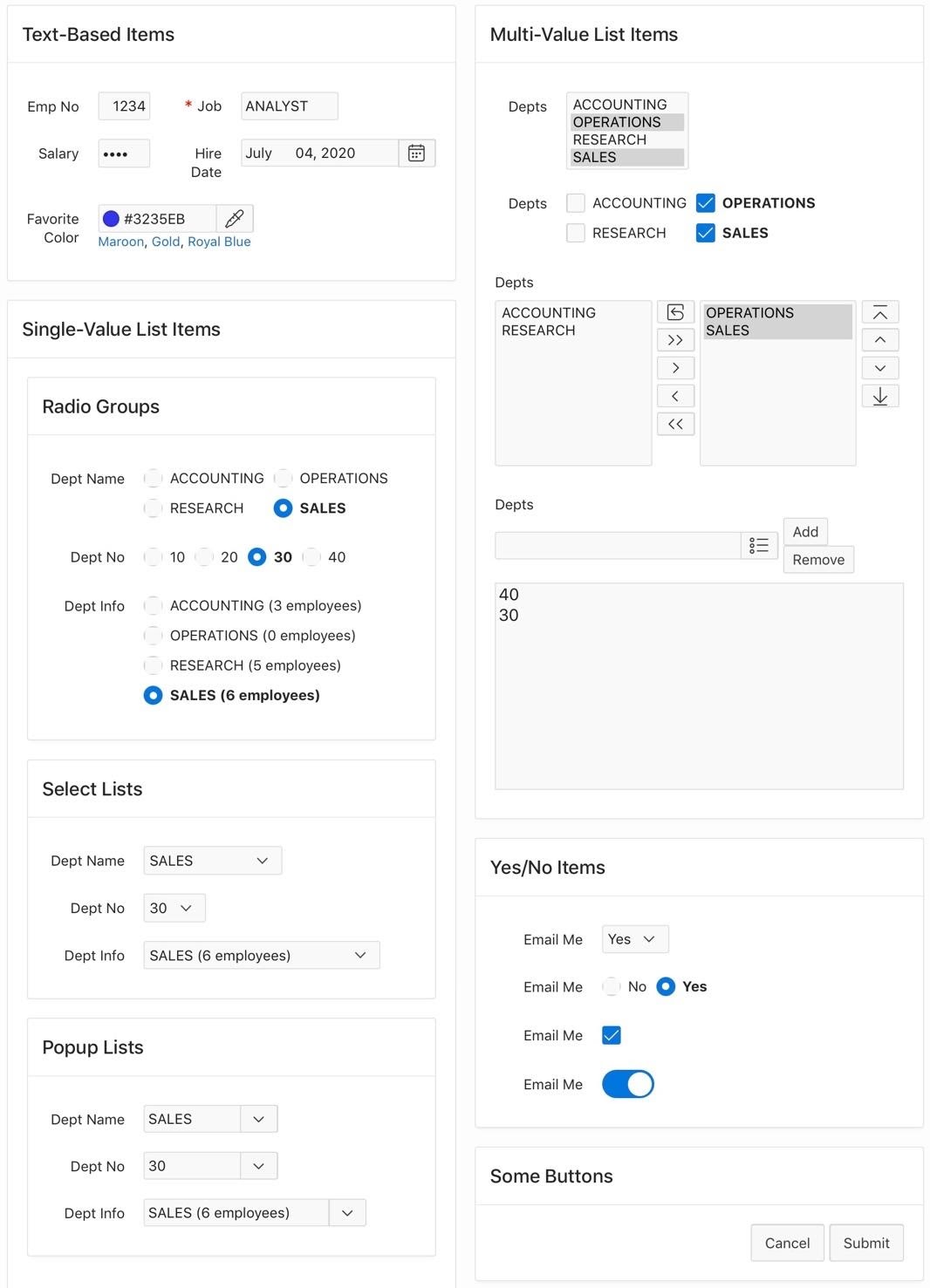
Items and buttons, WAYS TO GET THINGS BACK TO THE SERVER.

THERE are 22 kinds of items, 3 distinct types

***text-based*** items let users specify a value by typing it,

***list-based*** items let users specify a value by choosing it from a list, and

***display-based*** items prevent users from specifying values.



Because items are page components, each item will have a corresponding node in its page’s rendering tree.

By default, the maximum length is blank, denoting no maximum. A good design rule is to define a maximum length for all text-based items. In Chapter 13 you will see how an item with no maximum length (or an unreasonably large one) can contribute to serious security loopholes.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **REGION** | **ITEM\_NAME** | **LABEL** | **DISPLAY\_AS** | **SEQof** | **ITEM\_SOURCE** | **LOV\_DEFINITION** | **GRID\_C**  **OLUMN\_**  **SPAN** | **BEGINS\_ON**  **\_NEW\_**  **ROW** |
| Text-Based Items | P8\_EMPNO | Emp No | Number Field | 10 | Y | select EmpNo from EMP where Job = 'PRESIDENT' | 4 | Yes |
| Text-Based Items | P8\_JOB | Job | Text Field with autocomplete | 20 | N | STATIC:CLERK,ANALYST,SALESMAN,MANAGER,INTERN | - | No |
| Text-Based Items | P8\_SALARY | Salary | Password | 30 | N | - | 4 | Yes |
| Text-Based Items | P8\_HIREDATE | Hire Date | Date Picker | 40 | N | - | - | No |
| Text-Based Items | P8\_COLOR | Favorite Color | Color Picker | 50 | N | - | 7 | Yes |
| Radio Groups | P8\_RADIO\_NAME | Dept Name | Radio Group | 10 | N | select DName, DeptNo from DEPT order by DName | - | Yes |
| Radio Groups | P8\_RADIO\_NO | Dept No | Radio Group | 20 | N | select DeptNo as DisplayVal, DeptNo as ResultVal from DEPT order by DisplayVal | - | Yes |
| Radio Groups | P8\_RADIO\_INFO | Dept Info | Radio Group | 30 | N | select d.DName || ' (' || count(e.EmpNo) || ' employees)' as DisplayVal, d.DeptNo as ResultVal from DEPT d left join EMP e on d.DeptNo = e.DeptNo group by d.DeptNo, d.DName order by DisplayVal | - | Yes |
| Select Lists | P8\_SELECT\_NAME | Dept Name | Select List | 10 | N | select DName, DeptNo from DEPT order by DName | - | Yes |
| Select Lists | P8\_SELECT\_NO | Dept No | Select List | 20 | N | select DeptNo as DisplayVal, DeptNo as ResultVal from DEPT order by DisplayVal | - | Yes |
| Select Lists | P8\_SELECT\_INFO | Dept Info | Select List | 30 | N | select d.DName || ' (' || count(e.EmpNo) || ' employees)' as DisplayVal, d.DeptNo as ResultVal from DEPT d left join EMP e on d.DeptNo = e.DeptNo group by d.DeptNo, d.DName order by DisplayVal | - | Yes |
| Popup Lists | P8\_POPUP\_NAME | Dept Name | Popup LOV | 10 | N | select DName, DeptNo from DEPT order by DName | - | Yes |
| Popup Lists | P8\_POPUP\_NO | Dept No | Popup LOV | 20 | N | select DeptNo as DisplayVal, DeptNo as ResultVal from DEPT order by DisplayVal | - | Yes |
| Popup Lists | P8\_POPUP\_INFO | Dept Info | Popup LOV | 30 | N | select d.DName || ' (' || count(e.EmpNo) || ' employees)' as DisplayVal, d.DeptNo as ResultVal from DEPT d left join EMP e on d.DeptNo = e.DeptNo group by d.DeptNo, d.DName order by DisplayVal | - | Yes |
| Multi-Value List Items | P8\_M\_SELECT | Depts | Select List | 20 | N | select DName, DeptNo from DEPT order by DName | - | Yes |
| Multi-Value List Items | P8\_CHECKBOX | Depts | Checkbox | 30 | N | select DName, DeptNo from DEPT order by DName | - | Yes |
| Multi-Value List Items | P8\_SHUTTLE | Depts | Shuttle | 40 | N | select DName, DeptNo from DEPT order by DName | - | Yes |
| Multi-Value List Items | P8\_LIST | Depts | List Manager | 50 | N | select DName, DeptNo from DEPT order by DName | - | Yes |
| Yes/No Items | P8\_YN\_SELECT | Email Me | Select List | 10 | N | STATIC2:No,Yes | - | Yes |
| Yes/No Items | P8\_YN\_RADIO | Email Me | Radio Group | 20 | N | STATIC2:No;No,Yes;Yes | - | Yes |
| Yes/No Items | P8\_YN\_CHECKBOX | Email Me | Checkbox | 30 | N | STATIC: ;Yes | - | Yes |
| Yes/No Items | P8\_YN\_SWITCH | Email Me | Switch | 40 | N | - | - | Yes |

If you want your application to know about the values of the items on a page, you must first upload those values to the server. This operation is called *submit*. For redirect, you specify the target page as part of the redirect action. For submit, you specify the target page via a separate *branch* component—if no branches are specified, the current page is reloaded. The primary difference between submit and redirect is that submit uploads the page’s item values to the server, whereas redirect does not.

 The Redirect and Set Value option saves the value of that item in the session state and then redirects to the same page. That is, it behaves like submit, but for just one item instead of all items on the page.

 If you want to reference the variable from within SQL code, prepend a colon in front of the item name. Such a reference is called a *bind variable*.

 If you want to reference an item from within an HTML expression, prepend an ampersand in front of the item name and append a period at its end.

For example, consider the item named P8\_EMPNO on the Item Sampler page. To refer to it as a substitution string, write “&P8\_EMPNO.”. To refer to it as a bind variable, write “:P8\_EMPNO”.

------- new rpt ----

You can improve the functionality of this region by allowing a user to select a null value for P9\_JOB or P9\_DEPTNO. The intent is that a null value for P9\_JOB (or P9\_DEPTNO) should mean that the query will not use that item for filtering. For example, if both items were null, the report would display all employees. To enable this functionality, revise the source query to explicitly test for null item values, as follows:

select EmpNo, EName, Job, Sal, DeptNo

from EMP

where (:P9\_JOB is null  or  Job = :P9\_JOB)

and   (:P9\_DEPTNO is null  or  DeptNo = :P9\_DEPTNO)

order by EName

select EmpNo, EName, Job, Sal, DeptNo /\* for a checklist item \*/

from EMP

where InStr( ':' || :P10\_DEPTNO || ':',    ':' || DeptNo || ':' )  >  0

order by EName

**select Job, count(\*) as EmpCount**

**from EMP**

**where (:P11\_MIN\_SAL is null or :P11\_MIN\_SAL <= Sal)**

**and (:P11\_MAX\_SAL is null or :P11\_MAX\_SAL >= Sal)**

**group by Job**

**order by EmpCount**

**select \* from (**

**select d.DName, d.Loc, count(e.EmpNo) as EmpCount, d.DeptNo**

**from EMP e right join DEPT d**

**on e.DeptNo = d.DeptNo**

**group by d.DeptNo, d.DName, d.Loc**

**order by DName**

**)**

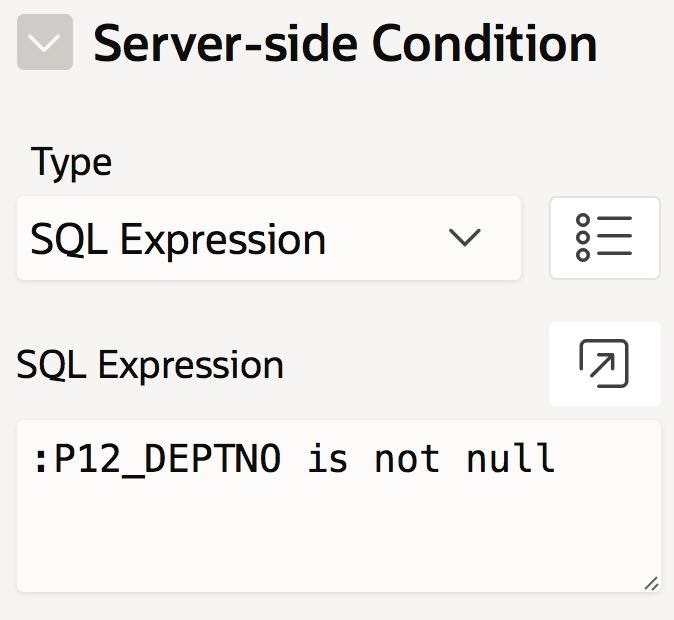
**select EmpNo, EName, Job, Sal**

**from EMP**

**where DeptNo = :P12\_DEPTNO**

**order by EName**

|  |  |
| --- | --- |
|  |  |

 **using sql**

The Processing section has four nodes, one for each type of activity that can occur during a submit action. This activity occurs in four stages:

* During the *computation* stage, the server computes additional session state values in preparation for the main processing activity. Such computations are occasionally useful, and are briefly discussed in Chapter 8.
* During the *validation* stage, the server runs validation checks to ensure that the submit action has been called with appropriate data. The server will abort further processing if it detects a problem. Validation is the subject of Chapter 9.
* During the *processing* stage, the server executes the processes that have been associated with the submit request. The definition and use of these processes is the subject of this chapter and Chapter 8.
* During the *branching* stage, the server decides which page to return to the browser. Branching is the subject of Chapter 10.

APEX calls these stages *execution points*, and gives different a name to two of the stages. In particular, the computation stage is called *After Submit*, and the branching stage is called *After Processing*.

**Computations (aka after submit) , Validation, Processing, Branching (aka after processing)**

APEX calls these stages *execution points*, and gives different a name to two of the stages. In particular, the computation stage is called *After Submit*, and the branching stage is called *After Processing*.

###    Point

Select at what point in page rendering or processing this process is executed.

Available options include:

New Session = Executes when a new session is generated before any pages have been rendered.

Before Header = Rendering: Executes before the **page** renders (and not during a submit action).

After Header = Rendering: Executes after the **page** header renders (and not during a submit action).

Is stored in the rendering page , not processing page so its shown on the rendering page only including processes.

Before Regions = Rendering: Executes before the regions render.

After Regions = Rendering: Executes after the regions render.

Before Footer = Rendering: Executes before the footer renders.

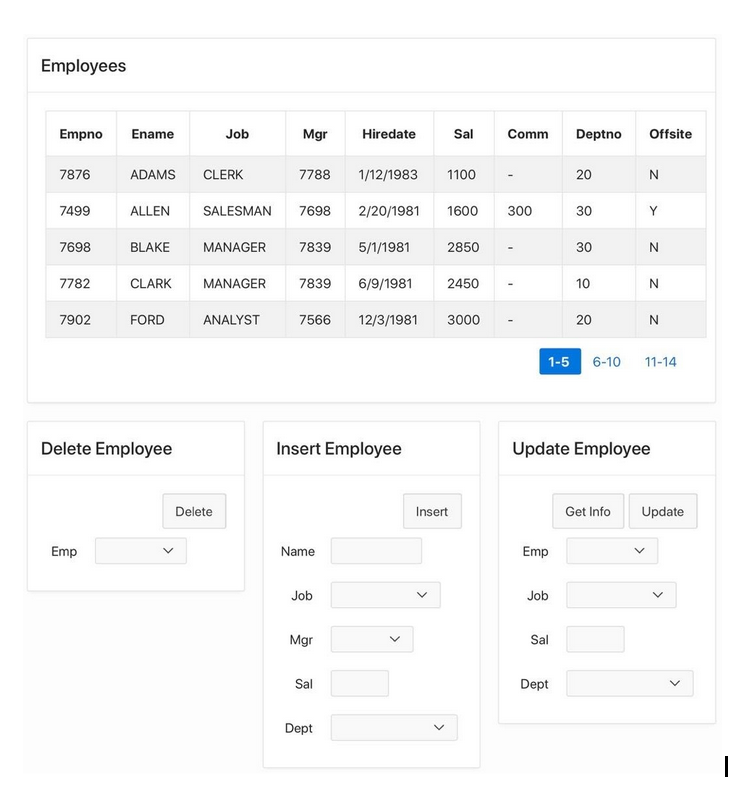
After Footer = Rendering: Executes after the page renders.

After Submit = Processing: Executes after the page is submitted, but **before** computations, validations, and page processes performed.

Processing = Processing: Executes after validations and computations performed, when page processing performed.

Ajax Callback = Executes when called by apex.server JavaScipt APIs.

The default process type is PL/SQL Code

****

**EMPNO NUMBER(4,0)**

**ENAME VARCHAR2(10)**

**JOB VARCHAR2(9)**

**MGR NUMBER(4,0)**

**HIREDATE DATE**

**SAL NUMBER(7,2) ----------**

**COMM NUMBER(7,2)**

**DEPTNO NUMBER(2,0)**

**OFFSITE VARCHAR2(1)**

**P17\_INSERT\_**

**:P17\_INSERT\_ENAME,**

**:P17\_INSERT\_JOB, ----------------------------**

**:P17\_INSERT\_MANAGER, ----------------------**

**:P17\_INSERT\_SALARY, -------------------**

**:P17\_INSERT\_DEPT, ----------------------**

**:P17\_UPDATE\_ENAME,**

**:P17\_UPDATE\_SALARY,**

**:P17\_UPDATE\_JOB,**

**:P17\_UPDATE\_MANAGER,**

**:P17\_UPDATE\_DEPT,**

The values of P17\_UPDATE\_JOB are defined by the query:

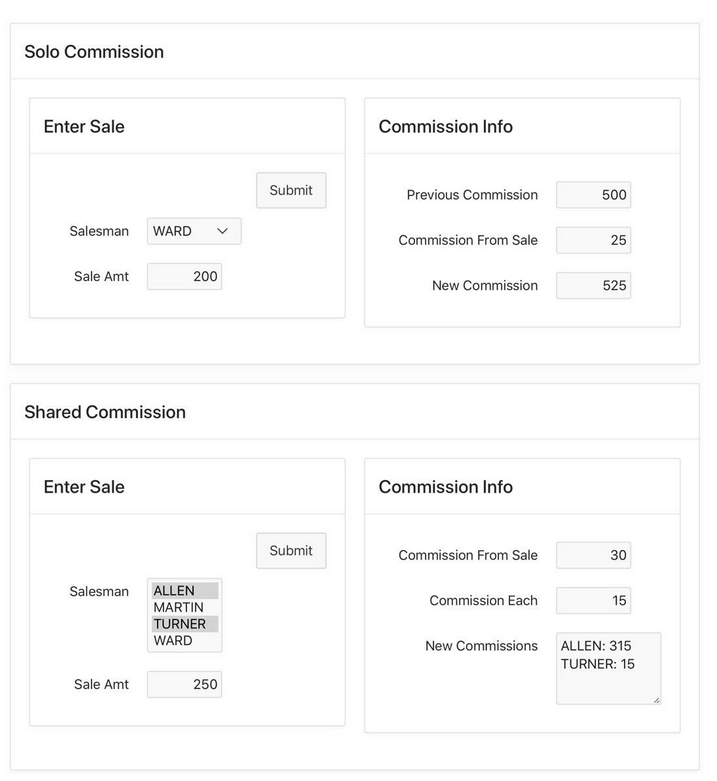
select distinct Job as DisplayVal, Job as ResultVal from EMP;

The values of P17\_UPDATE\_MANAGER are defined by the query:

select EName, EmpNo from EMP where Job = 'MANAGER'

The values of P17\_UPDATE\_DEPT are defined by the query:

select DName, DeptNo from DEPT

****

The rule used by APEX is to commit the current transaction (and begin a new one) each time one of the following three events occurs: a page has finished rendering, a submit request has completed, or a process that modified the session state has completed. Roughly speaking, this means that if there is one associated process per button, each button click corresponds to a transaction.

Oracle automatically checks for potential conflicts when a transaction tries to commit, and aborts the transaction if it detects one.

Now for the main issue: How can the UpdateEmp process determine that no other transaction has updated the record it is about to update? The simplest way is for GetEmpInfo to store a copy of each value it retrieved from the database. Then UpdateEmp can re-retrieve those values from the database, compare them with the stored copies, and if the values are identical, perform the update.

begin

    -- First retrieve the data.

    select Job, Sal, DeptNo

    into :P19\_UPDATE\_JOB, :P19\_UPDATE\_SALARY, :P19\_UPDATE\_DEPT

    from EMP

    where EmpNo = :P19\_UPDATE\_EMPNO;

    -- Then use hidden items to save a copy of the data.

    :P19\_ORIGINAL\_JOB  := :P19\_UPDATE\_JOB;

    :P19\_ORIGINAL\_SALARY  := :P19\_UPDATE\_SALARY;

    :P19\_ORIGINAL\_DEPT := :P19\_UPDATE\_DEPT;

end;

**/\* storing two versions of that data \*/**

It does so by re-reading the values from the database and comparing them against the original values in the hidden items.

declare

    v\_newjob  varchar2(9);

    v\_newsal  number(7,2);

    v\_newdept number(4,0);

begin

    -- First re-read the data.

    select Job, Sal, DeptNo

    into v\_newjob, v\_newsal, v\_newdept

    from EMP

    where EmpNo = :P19\_UPDATE\_EMPNO;

    -- Then compare it with the original data.

    if :P19\_ORIGINAL\_JOB = v\_newjob and

       :P19\_ORIGINAL\_SALARY  = v\_newsal and

       :P19\_ORIGINAL\_DEPT = v\_newdept

    then

         -- The record hasn't changed, so update it.

         update EMP

         set Job    = :P19\_UPDATE\_JOB,

             Sal    = :P19\_UPDATE\_SALARY,

             DeptNo = :P19\_UPDATE\_DEPT

         where EmpNo = :P19\_UPDATE\_EMPNO;

    else

         -- The record has changed, so abort.

         raise\_application\_error(-20000,

                      'The record is out of date. Get it again.');

    end if;

end;

## Using a Hash Function

Although this revised code works, the technique does not scale well. The need to save and compare all the original values is tedious and will become more so as the number of saved values increases. An easier approach is to combine all the original values into a single value; you can then save and compare that one value. The preferred technique is to use a ***hash function*, which transforms a collection of values into a single, fixed-length value,** called a *checksum*. A hash function also obfuscates its input, which makes it useful for encoding user passwords—as will be seen in Chapter 13.

APEX has a built-in hash function, named apex\_util.get\_hash. The input to this function is a collection that contains the values you want hashed. You should use the APEX built-in type

**apex\_t\_varchar2 to hold this collection.**

For an illusrative example, consider the following code:

**v\_vals   apex\_t\_varchar2 := apex\_t\_varchar2('CLERK', 20);**

**v\_result varchar2(1000)  := apex\_util.get\_hash(v\_vals);**

*Second Revision of the GetEmpInfo Process of Listing 7-4*

declare

    v\_valuesToHash apex\_t\_varchar2;

begin

    -- First retrieve the data.

    select Job, Sal, DeptNo

    into :P19\_UPDATE\_JOB,

         :P19\_UPDATE\_SALARY,

         :P19\_UPDATE\_DEPT

    from EMP

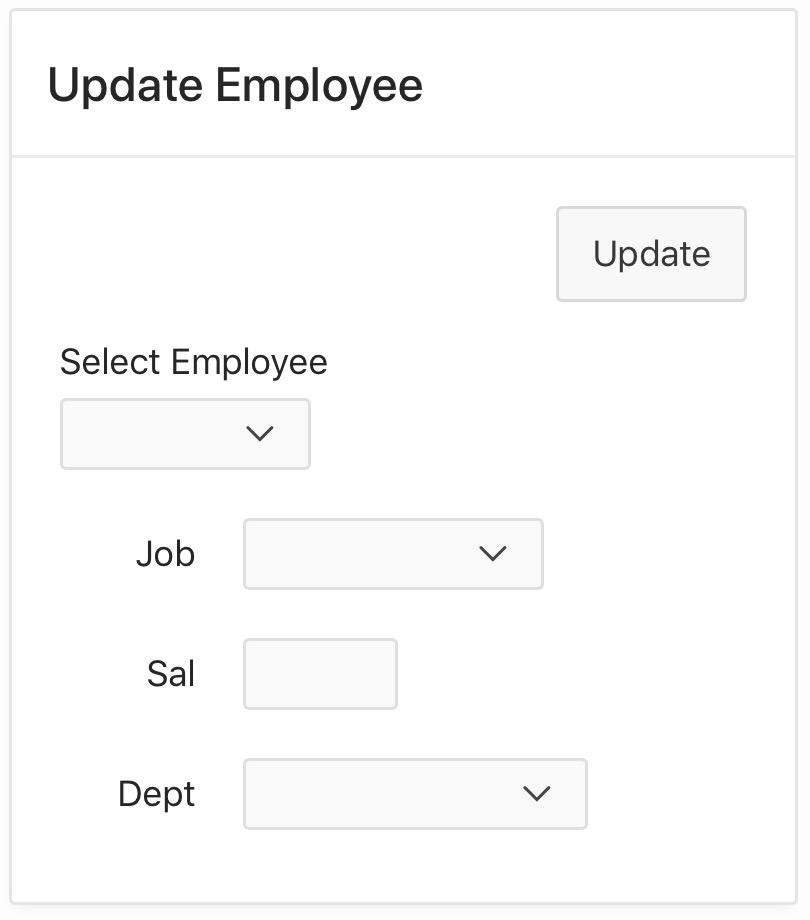
    where EmpNo = :P19\_UPDATE\_EMPNO;

    -- Then save the hash of these values.

**v\_valuesToHash := apex\_t\_varchar2(:P19\_UPDATE\_JOB,  :P19\_UPDATE\_SALARY,  :P19\_UPDATE\_DEPT);**

    :P19\_HASH := apex\_util.get\_hash(v\_valuesToHash);

end;



. Note that its SQL expression makes use of the APEX variable **REQUEST, which holds the name of the item that performed the submit action**. The expression references the variable using bind variable notation because it is an SQL expression. The expression compares the variable value to the string ‘P19\_UPDATE\_EMPNO’ (without a colon) because the expression is interested in the *name* of that item, not the *value* it holds.

Another situation that cannot make use of the When Button Pressed property is when a process is applicable to more than one button.

Another situation that cannot make use of the When Button Pressed property is when a process is applicable to more than one button. This section contains two examples related to the Revised Employee Data Entry page.

create table EMPLOG(Request varchar2(20),

                    UserName varchar2(20),

                    RequestDate Date);

Let LogChanges be the name of the process that inserts records into EMPLOG. You should create this process in the Revised Employee Data Entry page, giving it the PL/SQL code of Listing 7-13.

*Listing 7-13. PL/SQL Code for the LogChanges Process*

begin

    insert into EMPLOG (Request, UserName, RequestDate)

    values (:REQUEST, :APP\_USER, sysdate);

end;

his process needs to execute each time the Insert, Delete, or Update button is clicked, but not when the GetInfo button is clicked. Clearly, the When Button Pressed property is of no use here because it lets you choose only a single button.

The solution is to set the server-side condition of the process to be an SQL expression, as shown in Figure 7-18.

This expression ensures that the process will fire only when a button named Insert, Delete, or Update is clicked. Because each of these buttons also has its own dedicated process, clicking any of these buttons will cause two processes to execute. The processes will execute in order of their sequence number.

Lets look at send email during a suspicious update.

declare

  v\_action varchar2(40);

begin

  if :REQUEST = 'Delete' then

    v\_action := ' deleted the record for emp '  || :P19\_DELETE\_EMPNO;

  else

    v\_action := ' modified the record for emp ' || :P19\_UPDATE\_EMPNO;

  end if;

  apex\_mail.send(

     p\_from => 'EmployeeDemo@MyCompany.com',

     p\_to     => 'dba@MyCompany.com',

     p\_subj => 'Suspicious changes to EMP',

     p\_body => :APP\_USER || v\_action);

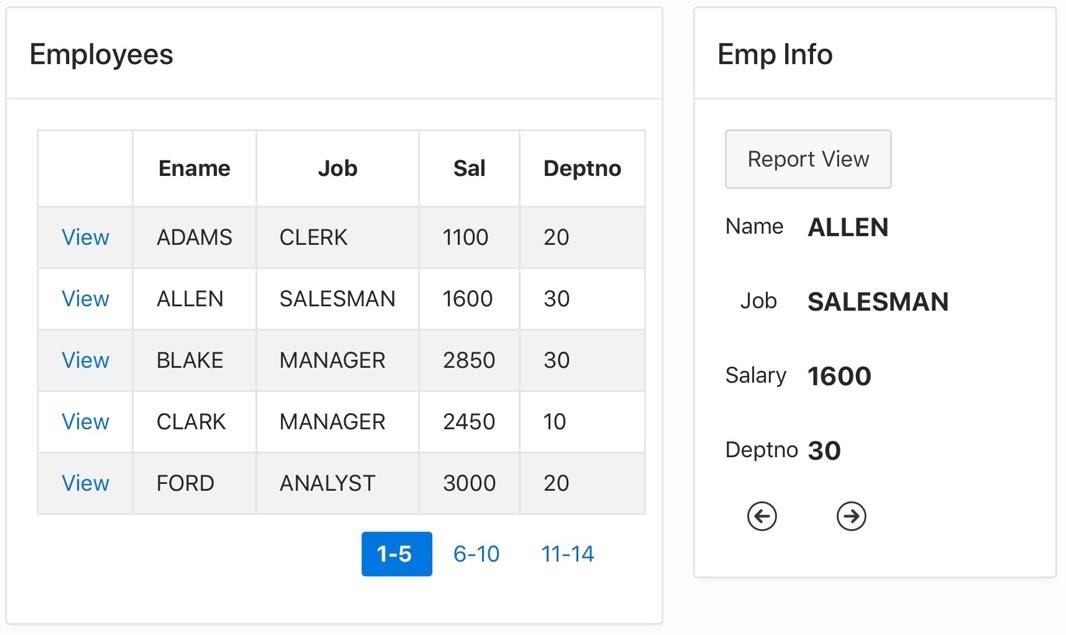
**end;**

In serverside conditions, use sql express for the below:

(:REQUEST = 'DELETE' or :REQUEST = 'UPDATE') and   
(  
extract(HOUR from current\_timestamp) > 16   
or   
extract(HOUR from current\_timestamp) < 9   
)

The next topic is to consider processes that execute during page rendering. This situation often arises when a page uses redirection instead of submit, such as when linking from a row of a report.

Although Figure 7-20 shows the Employees and Emp Info regions together, they actually are never displayed this way. The Employees region is displayed only when there is no selected record, and the Emp Info region is displayed only when there is a selected record.



**Executing a Process Without a Submit**

links perform a redirect so What you therefore need is for the process to execute after the redirection—that is, during page rendering.

Note that the parent of the FetchRow node is named After Header, which denotes a point during page rendering—in particular, the point after the server has generated the page’s HTML header code but before it generates the page contents. This is the default execution point for a page-rendering process.

 it should execute its SQL statement only when P20\_EMPNO is non-null. That is the purpose of the if-statement in Listing 7-15.

## Moving Between Regions

At this point in your application, you should verify that clicking a link on the Employees report will correctly place the values for the selected employee into the items of Emp Info. The next issue is to work out a plan for how the regions can “take turns” rendering themselves.

The key idea is to realize that a non-null value for P20\_EMPNO indicates that a row has been selected. So to “un-select” a row, it follows that you should put a null value into P20\_EMPNO. The two regions can thus use the value of that item to determine their conditional rendering—The Employees region should be visible when the item value is null, and the Emp Info region should be visible when the value is not null. In other words, the value of P20\_EMPNO determines whether the page is in *report mode* or *single-row mode*.

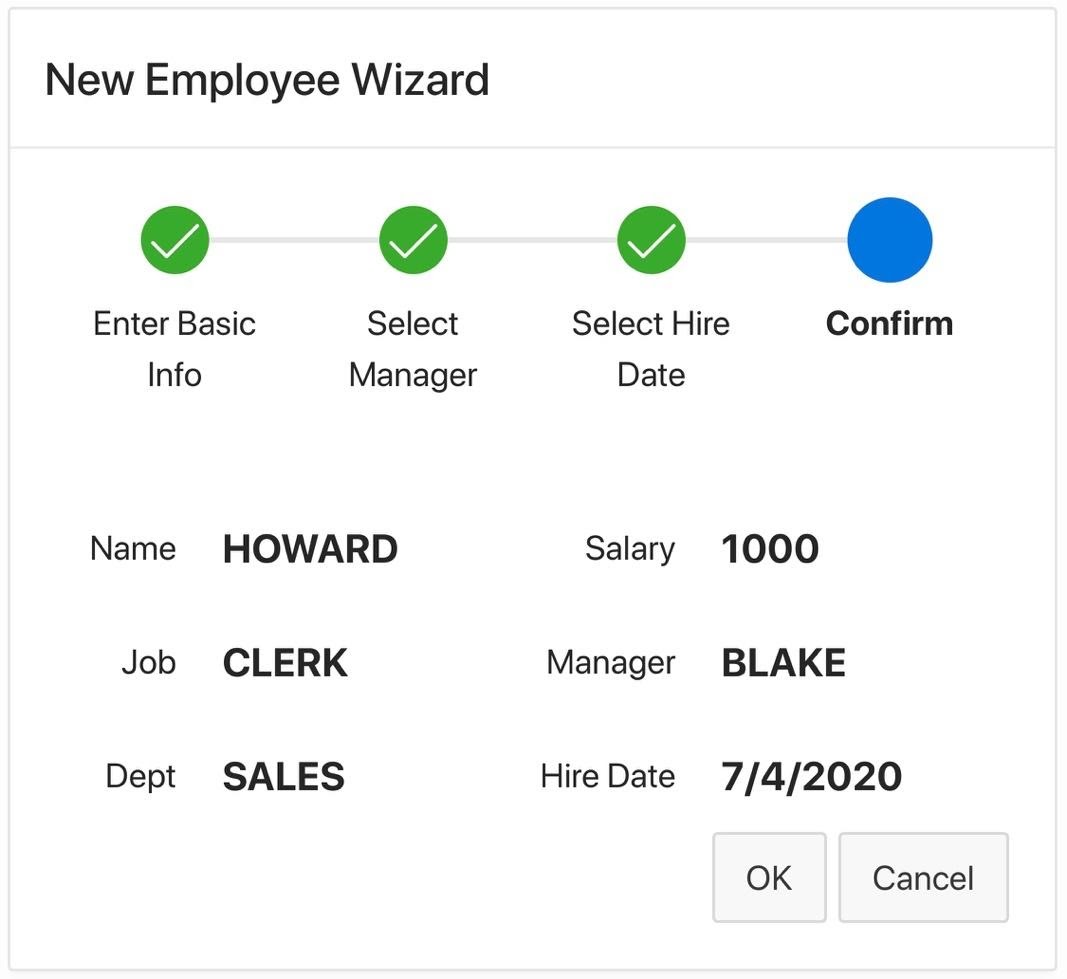
**Ch9===================**

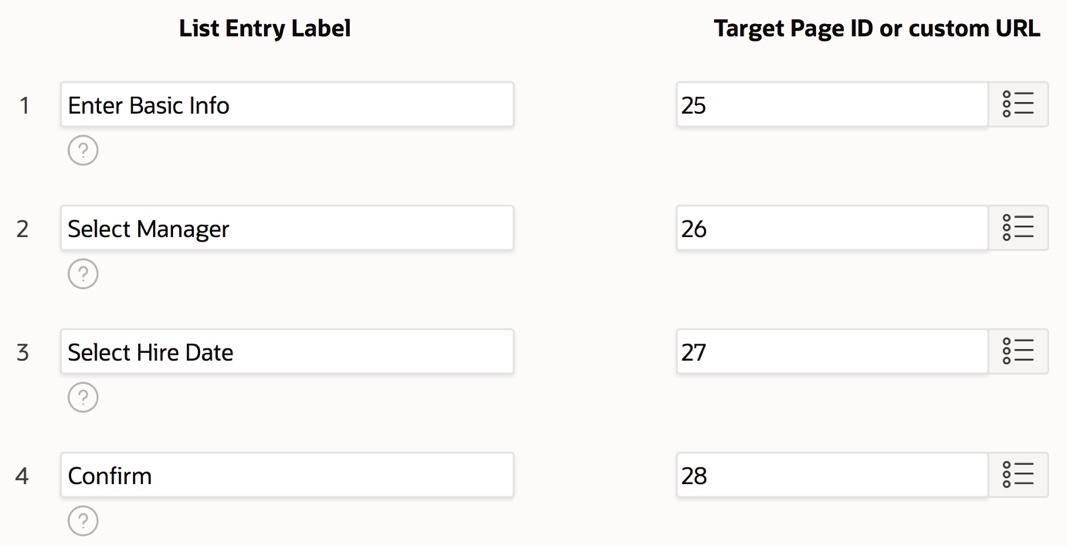
When the APEX server receives a submit request from a browser, it performs the validations and processes associated with that request. Its final task is to choose a target page to send to the browser. By default, APEX chooses the page that was submitted, but it is possible to specify another page by creating a *branch*.

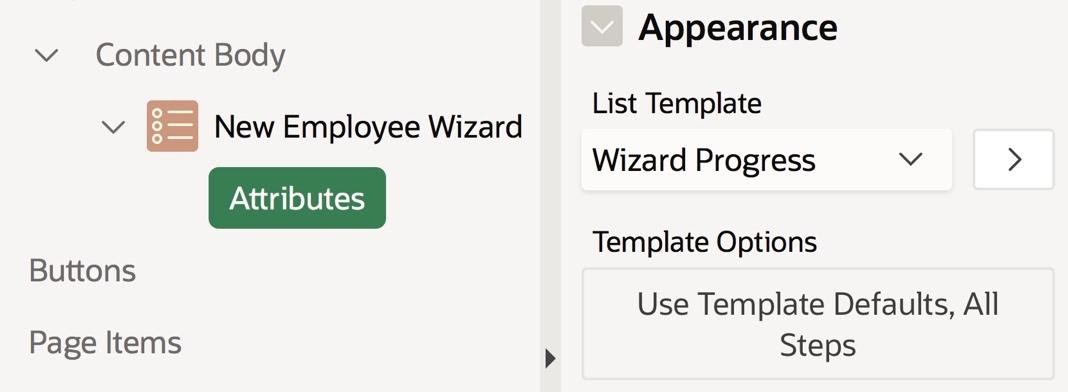
A common application design technique is to use separate pages for input and output.

In order to get page 22 to branch to page 23, you need to create a *branch* object. A branch object is similar to a process, in the sense that it performs an action in response to a submit.

In order to get page 22 to branch to page 23, you need to create a *branch* object. A branch object is similar to a process, in the sense that it performs an action in response to a submit. The difference is that a process executes code, whereas a branch redirects to a page.







This chapter looked at branch objects and gave some examples of their use. Branches execute during a submit operation, after the validations and processes. A branch has two important components: its target and its condition.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* CH10\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*