Adventure into PL/PSQL In Postgres

Links:
• http://www.postgresql.org/docs/8.1/static/functions-string.html
• http://www.postgresonline.com/journal/archives/58-Quick-Guide-to-writing-PLPGSQL-Functions-Part-1.html
• http://www.pgadmin.org/docs/1.6/pg/plpgsql.html
• http://wiki.postgresql.org/wiki/Return_more_than_one_row_of_data_from_PL/pgSQL_functions

PL/pgSQL – SQL Procedural Langauge

Advantages of Postgres PL/pgSQL
Creating functions and trigger procedures
Possessing control structure in the query language
Performing complex computations
Can be defined to be trusted by the server
Easy to use
Can be declared to return a set or a table or any data type they can return a single instance of.
Can be declared to return void if it has no useful return value
Can accept and return the polymorphic types anyelement and anyarray.

STRUCTURE OF PL/pgSQL
[<<label>>]
[DECLARE
	Declarations]
BEGIN
	Statements
END;

Syntax of PL/pgSQL
- Each declaration and each statement within a block is terminated by a semicolon
- A block that appears within another block must have a semicolon after END, though the final end does not require semicolon.
- All keywords and identifiers can be written in mixed upper and lower case unless double quoted.
- Two types of comments double dash (--) and block comment(/* …. */), note block comments cannot be nested. Though double slash be be inside block comments.

Eg:
```
CREATE OR REPLACE  FUNCTION somefunc() RETURNS integer AS $$
DECLARE
	quantity := 30;
BEGIN
	RAISE NOTICE ‘Quantity value is ‘, quantity; -- Quantity is 30
	quantity := 50;
	--Creating subblock
	DECLARE
		quantity integer := 80; -- Quantity inside block is 80
	BEGIN
		RAISE NOTICE  ‘Quantity inside subblock is ‘, quantity;
	END;
	RAISE NOTICE ‘After block, quantity is ‘, quantity;
END;
$$ LANGUAGE plpgsql
```

Declarations
All variables in the block must be declared in the declaration section with exception of loop variables.
PL/pgSQL variables can have any SQL data type, such as integer, varchar, and char.

Variable Declaration syntax:
Name [CONSTANT] type [NOT NULL] [{ DEFAULT | := } expression];
Default clause if given specifies the initial value assigned to variable when the block is entered, if not given the SQL null value is set.
Eg.
User_id integer;
Quantity numberic(5);
url varchar;
myrow tablename%ROWTYPE;
myfield tablename.columnname%TYPE;
arrow RECORD;
E.g. of default;
quantity integer DEFAULT 32;
url varchar := 'http://mysite.com';
user_id CONSTANT integer :=10;

Aliases for Function Paramters
Parameters to functions are named with the identifiers $1, $2, etc.. Optionally, aliases can be declared for $n parameter names for increased readability., either the alias or the numeric identifier can then be used to refer to the parameter value
There are two ways to create alias. Preferred way is giving name to parameter in the create function command. E.g.

```
CREATE FUNCTION sales_tax(subtotal real) RETURNS real AS $$
BEGIN
      RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

Or declaring alias on the declaration section
Name ALIAS FOR $n;
e.g.

```
CREATE OR REPLACE FUNCTION sales_tax(real) RETURNS real AS $$
DECLARE
      subtotal ALIAS FOR $1;
BEGIN
      RETURN subtotal * 0.06;
END;
$$
```

Copying types of variables as schema column types
Variable%TYPE
%TYPE provides the data type of a variable or table column. You can use this to declare variables that will hold database values.
User_id users.user_id%TYPE;
Advantage is the type of the variable will change with table type, so you don't have to change

Row types
Variable of composite type is called a row variable (or row-type variable) which can hold a whole row of SELECT or FOR query result, so long as that query's column set matches the declared type of variable. The individual fields of row value are accessed using the dot notation, for example rowvar.field.

Name tableName%ROWTYPE
Name composite_type_name;

Conditional structure
IF ELSE THEN
Syntax:
IF boolean-expression THEN

```
    statements
[ ELSIF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements
    ...]]
[ ELSE
    statements ]
END IF;
```

```
IF number = 0 THEN
    result := 'zero';
ELSIF number > 0 THEN
    result := 'positive';
ELSIF number < 0 THEN
    result := 'negative';
ELSE
    -- hmm, the only other possibility is that number is null
    result := 'NULL';
END IF;
IF parentid IS NULL OR parentid = ''
THEN
    RETURN fullname;
ELSE
    RETURN hp_true_filename(parentid) || '/' || fullname;
END IF;
```

EXIT condition

```
EXIT [ label ] [ WHEN expression ];
EXIT WHEN count > 0;
```

CONTINUE Condition

```
CONTINUE [ label ] [ WHEN expression ];
```

WHILE Condition

```
[ <<label>> ]
WHILE expression LOOP
    statements
END LOOP [ label ];
```

```
WHILE amount_owed > 0 AND gift_certificate_balance > 0 LOOP
    -- some computations here
END LOOP;

WHILE NOT boolean_expression LOOP
    -- some computations here
END LOOP;
```

FOR Condition

```
[ <<label>> ]
FOR name IN [ REVERSE ] expression .. expression [ BY expression ] LOOP
    statements
END LOOP [ label ];
```

```
FOR i IN 1..10 LOOP
    -- some computations here
    RAISE NOTICE 'i is %', i;
END LOOP;
```

```
FOR i IN REVERSE 10..1 LOOP
   -- some computations here
END LOOP;

FOR i IN REVERSE 10..1 BY 2 LOOP
   -- some computations here
   RAISE NOTICE 'i is %', i;
END LOOP;
```

Looping through query result

[<<label>>]
FOR target IN query LOOP
statements
END LOOP [label];

```
FOR target IN SELECT * FROM hris_values WHERE instance=singleinstance ORDER BY field_id LOOP
       hris_values_field_ids := hris_values_field_ids ||  target.field_id;
END LOOP;
```

Conditional expression
CASE

```
CASE WHEN condition THEN result
    [WHEN ...]
    [ELSE result]
END
```

Example:

```
SELECT * FROM test;
 a
---
 1
 2
 3
```

```
SELECT a,
       CASE WHEN a=1 THEN 'one'
            WHEN a=2 THEN 'two'
            ELSE 'other'
       END
   FROM test;
a | case
---+-------
 1 | one
 2 | two
 3 | other
```

Best way of avoiding Division-by-zero failure

```
SELECT ... WHERE CASE WHEN x <> 0 THEN y/x > 1.5 ELSE false END;
```

Greatest and Least

```
GREATEST(value [, ...])
LEAST(value [, ...])
```

SELECT INTO

Helps in assignment of sql result into a variable

```
CREATE OR REPLACE FUNCTION select_into() RETURNS integer AS $$
DECLARE
       -- Declarations
       var1 text;
```

```
	var2 integer;
BEGIN
	var1=E'"welcome home"';
	SELECT * INTO var2 FROM strpos(var1,'"');
	RETURN var2;
END;
$$ LANGUAGE plpgsql;

SELECT * FROM select_into();
```

TYPE CASTING

CAST(variable AS text)