DPSP™ – Dynamic PL/SQL™ Server Pages

User's Guide and Reference

Version 2.3.3

Document ID: 008-200002-021
Last Revision: July 3, 2003

COPYRIGHT INFORMATION AND ACKNOWLEDGEMENTS

Table of Contents

INTRODUCTION

WHAT IS DPSP AND WHY WE DECIDED TO CREATE IT?

Dynamic PSP™, or DPSP™, is the PL/SQL™ Server Pages interpreter and compiler designed for Oracle8*i*™/9*i* RDBMS and Oracle® HTTP Server (OHS)/Oracle9*i*™ Application Server (9iAS), a simple yet very powerful server-side scripting solution for Oracle. It is installed into Oracle8*i*/9*i* RDBMS as several PL/SQL packages and Java™ classes and is instantly available after that, provided that OWA (Oracle Web Applications Toolkit) packages are already installed (if Oracle's `mod_plsql` module is used as the gateway) and target schema is published via 9iAS or Oracle HTTP Server. DPSP units are created, edited and managed via web-based interface requiring no new software installation for developers beyond any HTML4.0/DHTML-compliant browser (Microsoft Internet Explorer 5.5 or later or Netscape 6.02 or later recommended). Optional Java-based Unit Editor is also provided with enhanced user interface.

Although Oracle has a term 'PSP' with the same meaning of 'PL/SQL Server Pages', it differs seriously from what you will read about in this document. After a bit of playing with Oracle PSP we decided that it's way too limited in functionality we needed, so we decided to create our own server-side extension to Oracle RDBMS/AS and devised it *Dynamic PSP* (DPSP) as this name best describes what we intended to achieve but sets us aside from Oracle's PSP approach. We believe Dynamic PSP is a viable alternative to Oracle PSP and JSP as well, because it doesn't require experienced Oracle PL/SQL programmers to learn Java and JSP and doesn't limit them in the way they can get things done using their favorite language.

DPSP shares common syntax with Oracle PSP, but extends it with some helpful features, like a number of predefined functions, dynamic execution (unlike Oracle's one-time compilation approach), unlimited number of parameters to any DPSP unit (in PSP, the number of parameters is fixed and is defined at compile time), 100% web-based development environment, and more, while preserving syntax and common functionality of PSP. Version 2 extends the Dynamic PSP capabilities by adding built-in access control and versioning, caching subsystem, introduces new DPSP Registry subsystem, which is similar to Windows Registry and is used by DPSP modules for configuration data persistent storage, and more.

WHAT IS IN THIS DOCUMENT?

This document is your primary source of information about Dynamic PSP Version 2. It provides information on Dynamic PSP Version 2 (DPSP2) internal architecture, syntax, APIs and interfaces; built-in features, functions and procedures; and contains numerous examples. The document is divided into several chapters:

Chapter I – DPSP2 Internal Architecture and Configuration

provides background information on DPSP2 Internal Architecture, gives insight on DPSP2 kernel installation, configuration and DPSP2 projects configuration and explains how the whole thing works.

Chapter II – DPSP2 Syntax and Built-in Functions and Procedures

describes the DPSP2 syntax and built-in functions and procedures. Each syntactical construction and function or procedure is accompanied with sample code demonstrating its use.

Chapter III – DPSP2 Standard API Reference

describes standard APIs (Application Programming Interfaces) provided by the kernel and standard modules. It includes standard types declarations, utility packages descriptions, and sample code.

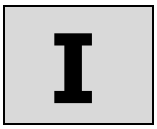
Chapter IV – DPSP2 Hooks and Cache

describes DPSP2 hook (trigger) mechanisms and built-in caching and cache control mechanisms.

Chapter V – DPSP2 Unit Commander

describes the web-based Development Interface of DPSP2 which is called 'Unit Commander'. All areas of the interface are described in detail and all basic operations that can be performed in the Unit Commander are outlined.

Other documents you need to familiarize yourself with are product README file, which includes important information about product installation and required Oracle patches; and release notes accompanying each release. Release notes contain the change log for the product and can be used for quick reference on new features and important fixes for each release.



CHAPTER I. DPSP2 INTERNAL ARCHITECTURE AND CONFIGURATION.

This chapter describes the Dynamic PSP Version 2 internal architecture and Kernel configuration.

DPSP2 KERNEL, OPTIONAL MODULES AND PROJECTS.

Dynamic PSP Version 2 is modular by design. Modules are 'plugged into' the system and add certain functionality to it. Dynamic PSP modules are actually PL/SQL packages, which are installed into the Dynamic PSP kernel schema and registered with Dynamic PSP Registry. There are mandatory and optional modules. *DPSP Registry* and *Kernel* modules are always required, as well as *FLAT preprocessor* module. DPSP Registry provides hierarchical persistent storage facility similar to Windows Registry for all other modules, including the Kernel. The Kernel module is the central piece of the system – you can consider it as a hub to which all other modules are connected. The Kernel defines global DPSP2 types and APIs, manages inter-module and inter-project communications, sets up runtime environment for modules and DPSP units, and processes requests. Preprocessor modules are responsible for unit source code translation and executable code generation. The DPSP2 Kernel is installed into a separate Oracle schema and is shared among all DPSP2 projects. Each *DPSP2 project* resides in its own Oracle schema and has access only to shared Kernel services and its own data and database objects. Special *Agent* module is installed into each project schema, which is used by the Kernel to process external requests. All public packages of the Kernel are specified with `DEFINER` rights, which mean that the Kernel code can access only objects in the Kernel schema and `PUBLIC` objects. The project Agent is also specified with `DEFINER` rights and can only manipulate the database objects for which the owner has appropriate privileges. Source code for all units of all projects is stored in the Kernel schema, and only Kernel schema owner has access to it. Other project-specific database objects are located in the project schema. `PUBLIC SYNONYM`s are created for public Kernel objects, like stored procedures, packages and views, which allows project schemas to use Kernel services. Built-in access control system controls access to the Development Interface.

DYNAMIC PSP VERSION 2 AND OWA TOOLKIT.

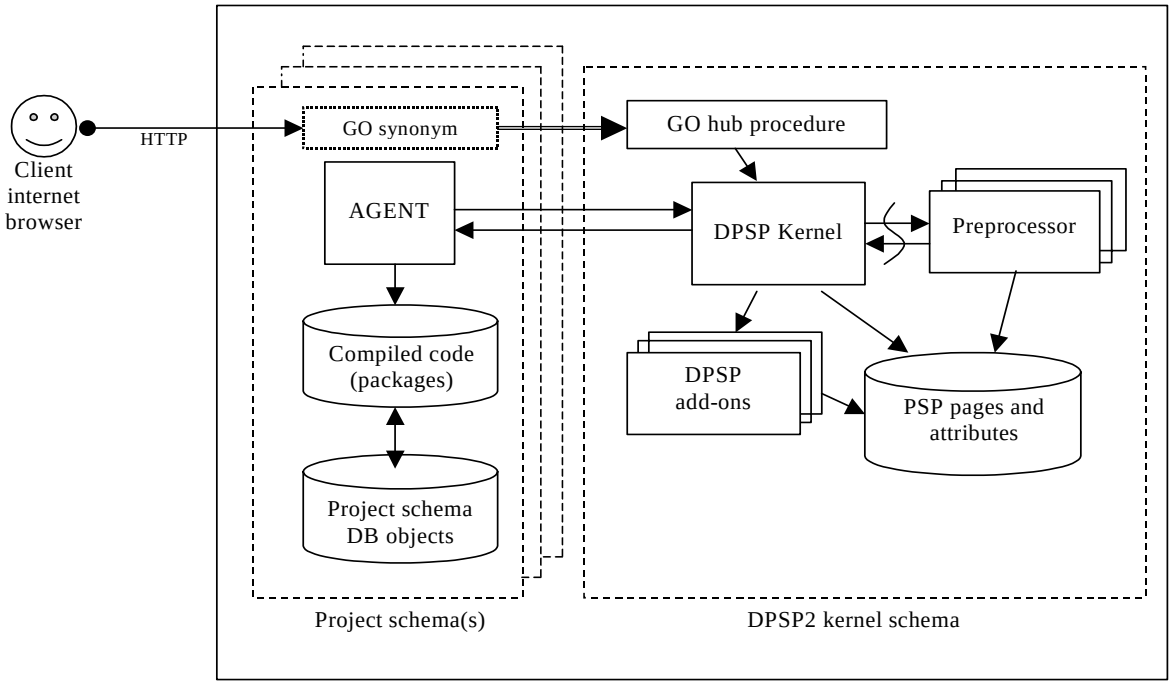
OWA stands for Oracle Web Applications [Toolkit] and is a set of PL/SQL packages installed in the RDBMS, which are used to communicate with the PL/SQL gateway module (mod_plsql). Without OWA packages, mod_plsql will be unable to communicate with the RDBMS and your classic web applications and Oracle PSP pages will not function.

Dynamic PSP Version 2 was designed so that it does not rely on OWA Toolkit presence. This allows using alternate gateways for accessing DPSP2 units. One such gateway, JOPA Gateway Servlet (Java servlet for Oracle PL/SQL Applications), is developed by N-Networks and can be used to provide access to DPSP applications from any server that supports Java servlets 2.0 or later (Apache with JServ or Tomcat, for example). Due to this design, it is imperative that you do not call OWA packages from your DPSP units unless you absolutely need to. More than that, most OWA calls will not produce expected results in DPSP2 environment. For example, output produced by `HTP` and `HTF` packages is simply ignored and will not be sent to the client (although it can be embedded into the Dynamic PSP content with special API exposed by the Kernel), as well as most `OWA_COOKIE` and `OWA_UTIL` calls. If the gateway used to access Dynamic PSP is mod_plsql, some OWA calls will still produce correct results, but will add unnecessary overhead. For example, `OWA_COOKIE.GET()` call will return requested cookie, but will cause unnecessary cookie parsing which was already done by the Dynamic PSP Kernel. Same is with `OWA_UTIL.GET_CGI_ENV()` function. Dynamic PSP Version 2 built-in procedures, functions and additional APIs provide the same functionality as OWA packages, and, in some cases, even extend and supersede it.

This said, we should note that if mod_plsql is the gateway used to access Dynamic PSP applications, OWA packages still must be installed, otherwise mod_plsql will be unable to communicate with your applications and the DPSP Kernel. You should understand that although Dynamic PSP does not need OWA Toolkit to function, mod_plsql still needs it. OWA Toolkit is also needed for your legacy applications built using it to function.

DYNAMIC PSP VERSION 2 CALL FLOW.

When the Kernel receives an HTTP request, it parses the call parameters, creates runtime environment, calls corresponding preprocessor to translate the source code if necessary, sends the translated code to the project Agent and then passes control to the Agent. Project Agent then executes the translated code in the project schema as anonymous PL/SQL block, or calls compiled version of the unit if present and actual, and feeds the result back to the Kernel, which then passes it over to the Oracle PL/SQL gateway. If the currently executed unit requests another unit to be executed, the Kernel processes this request similarly, but skips runtime environment preparation step. If called unit is compiled, preprocessing step is skipped and Agent calls compiled version of the unit instead. The following diagram represents sample call flow:



The `GO` hub procedure is in the Kernel schema and has `PUBLIC SYNONYM`. This procedure is called by the PL/SQL gateway (mod_plsql or JOPA). Although the request is initiated with privileges of the calling user (project schema owner), the `GO` procedure is running with the Kernel schema owner privileges and has access to the Kernel schema objects. The procedure processes parameters it received and passes them to the Kernel, which then creates the runtime environment and stores the call parameters and environment variables for further access. The Kernel figures out which unit is called, retrieves its attributes and determines if it should call a preprocessor to translate the unit source code, and calls the corresponding preprocessor if needed. If there are `BeforeMain` and/or `BeforeExec` hooks defined, the Kernel invokes all hook routines in the DPSP2 Kernel schema context. The Kernel then invokes the Agent in the project schema and either passes translated (preprocessed) code to it for execution, or requests to execute compiled version of the unit, which is stored in the project schema as PL/SQL package. The unit is executed, accessing project schema objects and other units as needed (units run with project schema owner (definer) privileges or invoker privileges, depending on the corresponding unit attribute), and then the Kernel invokes any defined `AfterExec` and/or `AfterMain` hook routines and passes the final result to the PL/SQL gateway. If the main unit invokes any other units through exec() calls, `BeforeExec` and `AfterExec` hook routines (if any) are invoked in the DPSP2 Kernel schema context before and after unit execution.

UNIT EXECUTION MODES.

Dynamic PSP units may be executed dynamically or statically. When a unit is executed dynamically, each time it is called the preprocessor retrieves the unit source code, processes it and translates to an anonymous PL/SQL block, which is then executed by the Agent. In this mode all changes you made to the unit source code are taking effect immediately. Static mode means that the unit source code is processed once and PL/SQL package is created from it, which is then compiled on the server and becomes immune to subsequent source code changes until it is recompiled or its processing mode is changed. Ideally, all debugged production units should be static as there is no preprocessing and parsing overhead when they are executed. On Oracle9*i*, when you compile the unit for static execution you may also choose to compile resulting PL/SQL package natively further improving its performance (note that if your unit is SQL-intensive then there unlikely to be any performance gain from native compilation.)

The `PROCESSING_MODE` unit attribute affects the way the unit may be processed. In mode 0 (No processing) all calls to the unit are disabled. In mode 1 (Interpreted only) the unit may only be executed dynamically regardless if there is a valid compiled version of it. In mode 2 (Compiled only) the unit may only be executed statically, that is, there should be compiled version of the unit available or execution will fail. Mode 3 (Interpreted & Compiled) combines modes 1 and 2: static version is considered for execution first and is executed if exists and valid, and current version in unit repository is executed dynamically if there is no valid compiled version of the unit.

INSTALLING DYNAMIC PSP.

Dynamic PSP is installed into a separate Oracle schema, called Kernel schema. Dynamic PSP also allows you to create any number of "project schemas", one schema for each specific application project. All project schemas are served by single DPSP Kernel installed into the Kernel schema.

The Kernel schema user is usually named 'DPSP2'. You may choose a different name during the installation though. Choose a strong password for the DPSP2 user (mixed alpha and numeric characters, at least 8 symbols) because this account is essential and should be well guarded. This user should be granted a minimum set of privileges. `CREATE SESSION`, `CREATE PROCEDURE`, `CREATE TABLE`, `CREATE VIEW`, `CREATE SEQUENCE`, `CREATE SYNONYM`, `CREATE TRIGGER`, `CREATE TYPE`, `ALTER SESSION` and `QUERY REWRITE` is an acceptable privilege set. `CREATE PUBLIC SYNONYM` and `DROP PUBLIC SYNONYM` may also be granted to the DPSP2 user to allow DPSP2 system to publish DPSP units globally. We recommend creating a new tablespace and then create DPSP user with default tablespace being the newly created tablespace. Do not forget to grant acceptable quota to the new user for its default and temporary tablespaces, otherwise the user will be unable to store any data in the database.

Experienced DBA can create the Kernel schema manually, according to the requirements above. Or you may choose to use one of supplied SQL*Plus scripts that will prompt for all details needed to create the new user for the Kernel schema and then will create new user and all needed objects.

For minimum release:

```
> sqlplus <dba_account>/<password>[@TNSAlias] @create_kernel.sql
```

For standard release:

```
> sqlplus <dba_account>/<password>[@TNSAlias] @create_std.sql
```

With these scripts, the Kernel schema is created and populated with DPSP2 database objects: scripts will create all packages, tables and views in the DPSP2 Kernel schema. Installation log will be written to the corresponding `.LST` file; review it after installation to see if there were any error messages during the creation of database objects. Minimum release will only create minimal set of objects to support Dynamic PSP (runtime support), while standard release will also install modules included in standard edition.

If there were no errors during installation, you have DPSP2 successfully installed in the database.

For additional information about required Oracle database and Oracle HTTP Server patches, please review corresponding section in README.txt file accompanying the product.

CREATING AND ENABLING ORACLE SCHEMAS FOR USE WITH DYNAMIC PSP.

Before you can invoke Dynamic PSP units from certain Oracle schema (or under certain Oracle user privileges), this schema (user) should be made known to the Dynamic PSP Kernel. Several SQL*Plus scripts are provided for this purpose. These scripts are in the root of the product distribution directory. To create new schema and automatically enable it for use with DPSP, DBA should invoke `create_project.sql` script:

```
> sqlplus <system>/<manager>[@TNSAlias] @create_project.sql
```

The script will prompt for new user name, password and various other parameters and then will create new user, assign it proper privileges and create all database objects (packages and synonyms) needed to enable new schema for DPSP.

To enable an existing schema for use with DPSP, the schema owner can use `enable_project.sql` script:

```
> sqlplus <user>/<passw>[@TNSAlias] @enable_project.sql
```

This script will create database objects (packages and synonyms) in the `<user>` schema that are needed to enable DPSP support for it. Please note that the calling user should have at least `CREATE PROCEDURE` and `CREATE SYNONYM` privileges for script to complete.

Finally, to remove DPSP support from certain schema, the schema owner can use `drop_project.sql` script:

```
> sqlplus <dpsp_project>/<passw>[@TNSAlias] @drop_project.sql
```

This script will remove all objects created with `enable_project.sql` while leaving other objects that may exist in the schema intact.

CREATING AND CONFIGURING DYNAMIC PSP DADS.

To publish a Dynamic PSP project via mod_plsql, a Database Access Descriptor (DAD) should be created for it. Create new DAD (database access descriptor) for each DPSP2 project schema and for the DPSP2 Kernel schema. DAD for DPSP2 Kernel schema will be your primary access point to the Development Interface, and project DADs will provide access to particular projects. Note that Dynamic PSP Version 2 features built-in access control system so you generally do not need to take extra actions to protect the Development Interface from unauthorized access. You may want to require secure (SSL) connection for the Development Interface though. Please see our online FAQ at http://www.dpsp-yes.com/dpsp/prod/!go?id_=nfaq for details on configuring Oracle HTTP Server powered by Apache (OHS) to require SSL for Development Interface connections.

To create the DPSP2 DAD, do the following:

1. Open PL/SQL Gateway Administration page (usually accessible through `http://server/pls/ANYDAD/`**`admin_/gateway.htm`** URL).
2. Click on Gateway **Database Access Descriptor Settings**
3. Click on **Add Default (blank configuration)**
   *** Do **NOT** create Portal 3.x or WebDB configuration, they are not compatible with Dynamic PSP and will not function properly. ***
4. Fill the following fields and leave others at their default settings:

   Database Access Descriptor Name = new DAD name
   Oracle schema name = Dynamic PSP project schema name
   Oracle user name = Dynamic PSP project owner name
   Oracle user password = password for Dynamic PSP project owner
   Oracle connect string = Net8 alias of the instance this DAD connects to

   Make sure **Package/Session Management Type** is set to "`Stateless (Reset Package State)`".
   You can enable or disable connection pooling as you see fit (we recommend leaving it enabled for maximum performance.)

If you intend to upload or download files in this project, you will need to create the document table according to the mod_plsql documentation (documentation is accessible by clicking Help icon in the upper right corner of each PL/SQL Gateway Administration interface screen), section 1.7.1 (as of documentation for Release 1 (v1.0.2.2) Part Number A90099-01). Then you need to specify the name of this table in Document Table field of DAD configuration. You may also create your own document access procedure. Read the mod_plsql documentation to find out how to create one, and specify its name in Document Access Procedure field. Also, specify Document Access Path prefix that will invoke the document access procedure to process the request.

Dynamic PSP provides built-in support for file upload and download through mod_plsql and JOPA Gateway servlet. If you want to make use of this functionality, you can configure corresponding DAD parameters as follows:

Document Table = `<kernel_schema>.NN$T_DOWNLOAD`
Document Access Procedure = `<kernel_schema>.NN$PSP_OWA.DOWNLOAD`
Document Access Path = any name of your choice

(where `<kernel_schema>` is the name of the Dynamic PSP Kernel schema.) Once you set up these parameters, binary files will be uploaded into the `NN$T_DOWNLOAD` table, which resides in the Kernel schema and is exposed to `PUBLIC`, and may be downloaded by specifying the URL similar to the following:

http://server/pls/DAD/doc_access_prefix/FileName.ext

where `doc_access_prefix` is the Document Access Path value you entered on the DAD configuration page, and FileName.ext is the value of the `NAME` column in the document table, which was assigned to the file when it was uploaded.
You can also create your own document table and specify it for DPSP2 project DAD, and Dynamic PSP will use it instead of its default global document table for file storage and retrieval in this particular project. The only requirement for this table is that its layout should match the one described in mod_plsql documentation (that is, column names and types should match those specified in mod_plsql documentation.) However, do not drop the `NN$T_DONWLOAD` table as it is required by some components of Dynamic PSP.

DYNAMIC PSP KERNEL CONFIGURATION.

Dynamic PSP Version 2 Kernel configuration parameters, as well as configuration parameters for other DPSP2 modules, are stored in the DPSP2 Registry. DPSP2 Registry operates on *keys* and *items*. Keys are similar to file system folders, while items are similar to files in these folders and contain actual values. DPSP2 Kernel configuration hive is located under the `NN$PSP/Kernel` key. You can use the DPSP2 Registry API, web-based Registry Editor in the Development Interface, or Java-based Registry Editor (Regedit) to view and modify Kernel configuration. For more information on the DPSP2 Registry, refer to the **DPSP2 Registry User's Guide**. Following is the list of Kernel configuration keys, items and their meaning:

NN$PSP/Kernel

This key holds global Kernel configuration parameters:

`ErrorStyle` — This parameter controls the way the Kernel reports internal errors. It can have one of the two values: `HTTP` or `Internal`. `HTTP` will force the Kernel to return an HTTP error code and corresponding error page if an error is encountered. For example, `404` and `403` (`"Not Found"` and `"Forbidden"`, respectively) codes will be returned if the requested unit is not found or current user does not have sufficient privileges to perform requested operation on the unit. Note that in this mode the gateway will not signal any error to the HTTP server and the error page generated by the Kernel will be returned to the client with no further processing on the server.

`Internal` instructs the Kernel to return `200 OK` response along with its own error page except for the case when called unit could not be found/resolved, in which case an exception is raised and propagated to the gateway. The returned page may contain debugging information and unit source code, depending on the debug mode set for the unit. When an inexistent unit is called, the Kernel will raise `ORA-06508` exception, which will be intercepted and processed in the gateway: `404 Not Found` error will be triggered and sent to the HTTP server, which in turn will generate the error page according to its configuration (you may override the default error page with your own page or script using `ErrorDocument` Apache directive.)

Default mode is `HTTP`.

`GMT_DIFF` — If set, this parameter sets server time offset in hours from GMT (Greenwich Mean Time), also known as UTC (Universal Coordinated Time). Value should be in range –13..12. This item overrides `TIMEZONE` setting if it exists. If not set, `TIMEZONE` setting is used to calculate GMT dates. Affects cookie expiration time generation.

`TIMEZONE` — If set, this parameter specifies the time zone server is in. Should be valid time zone specification recognized by Oracle. For the list of supported time zones, consult Oracle server documentation. Defaults to `'GMT'`. Affects cookie expiration time generation.

`SessionLife` — Session timeout (period of inactivity) in days. May accept fractional values. For example, value of 0,5 means 12 hours.

`ContextLife` — Session context timeout in days. When this period expires, all context entries for the session whose age exceeds this value are removed from context. May accept fractional values. For example, value of 0,5 means 12 hours.

`Schema` — Designates Dynamic PSP Kernel schema. It is set automatically during the Kernel installation and should not be edited unless explicitly instructed to do so by customer support.

NN$PSP/Kernel/License

This key holds License information for the DPSP2 instance. **DO NOT MODIFY THIS KEY OR YOU WILL RISK LOSING YOUR LICENSE INFORMATION**, which will render the instance inoperable.

NN$PSP/Kernel/Purger

This key holds the Kernel Context Purger job parameters:

MSG_LOG_DAYS	Days to keep error messages in error log (default = 7)
SESSION_DAYS	Days to keep sessions in session context storage (default = 7)
UNIT_ARCHIVE_DAYS	Days to keep archived copies of DPSP2 units (default = 30)
UNIT_TRACE_DAYS	Days to keep unit trace statistics (default = 30)
CACHE_DAYS	Days to keep cached units content in internal cache (default = 3)

Purger job automatically deletes information from Kernel internal tables when it becomes too old according to these settings. Purger job is an Oracle job automatically created during the Dynamic PSP 2 Kernel installation, which is scheduled to run once a day at 1:00am. For this job to run, JOB_QUEUE_PROCESSES Oracle initialization parameter should be set to a value greater than 0 to enable background job processes (SNP*n*) to pick up and process jobs. This parameter may be altered with ALTER SYSTEM command without the need to shutdown and restart the database.

NN$PSP/Kernel/References

This key holds mappings of public DPSP2 module names and their internal names. These mappings are used to integrate additional modules into the runtime system. Do not change anything in this key unless instructed to do so by customer support.

NN$PSP/Dev

This hive controls DPSP2 Development Interface. It is used internally and should not be edited unless instructed to do so by customer support.

NN$PSP/Kernel/Hooks

This hive is used to declare DPSP2 Kernel hooks. For more information on declaring and implementing kernel hooks, see Chapter IV.

EXECUTING DPSP UNITS.

Each DPSP unit can be executed from the web unless its properties expressly prohibit web access (that is, unless unit is designed to be called from other units only, not directly). DPSP units can represent dynamic pages or reusable parts of a page, forms, form input handlers, etc. To execute a DPSP unit you need to make an HTTP request from any browser to the DPSP server.

URL format for Dynamic PSP call is as follows:

[<proto>]server[:<port>]/<DPSP_schema>/!go[?<parameter list>] *

where

<proto>	is an optional protocol specifier (for example, http://)
<port>	is an optional server port number (default is 80)
<DPSP_schema>	is the DPSP schema access prefix (as per Oracle PL/SQL Gateway specification, usually pls/<Database Access Descriptor>)
<parameter list>	is the list of call parameters, URL-encoded (that is, constituted of name=value pairs delimited by & (ampersand) character).

You can pass unlimited number of named parameters to a DPSP2 unit. If you specify several values for one parameter name (e.g. `'param=value1&param=value2'`), they can be retrieved in the unit as a PL/SQL table (array). **!go** is mandatory member of a DPSP2 unit call URL – it is the DPSP2 hub procedure which initially accepts all parameters, processes them and then calls the DPSP2 Kernel to further process the request. Exclamation sign in front of `GO` tells the mod_plsql gateway that the hub procedure follows 'flexible parameter passing' conventions. If you designate a DPSP2 unit as `FORM` handler, be sure to specify form attribute `ACTION="!go"`.

Some special parameters are recognized and processed by the `GO` procedure and are not passed to the called unit, while others are considered unit call arguments and are passed to the called unit for processing. Special parameters are:

id_ Specifies called unit ID. You can also use unit logical name attribute in place of unit ID (`id_=<unit name>`), but you should take care about unit name uniqueness within the project in this case – the system does not enforce uniqueness of attributes. The system will fail to resolve the name to ID if several units in the project have the same name attribute value.

op Specifies requested operation, which can be one of the following:
- **exec** - execute the unit specified by `ln` or `id_` parameter
- **edit** - edit unit source code (requires authentication)
- **view** - view generated PL/SQL code for the unit (requires authentication)
- **ucom** - invoke Unit Commander Development Interface (requires authentication)

This parameter is optional and defaults to `exec` if omitted.

ctx Specifies context generated by `WrapCtx` function, which will be retrieved from the server-side persistent storage, expanded into a parameter list and appended to the list of other parameters. Other standard parameters, including invoked unit ID and/or logical name and operation may be encoded to a context, so `ctx` may be the only parameter passed to the `GO` procedure.

ln Specifies unit logical name (can be used in conjunction or instead of unit ID parameter `id_`). This parameter is processed by the Resolver module if it is installed; otherwise minimal standard resolver is used, which tries to locate the unit with name attribute being equal to parameter value. `ln` parameter, if specified, takes precedence over `id_` parameter. If logical names resolver is unable to resolve its value, and `id_` is specified, `id_` is used to locate the called unit.

* Although this is mandatory call layout for DPSP, one can use Apache's mod_rewrite module for hiding this layout. For example, mod_rewrite may be used to translate URLs like /dpsp_schema/unit_id to /dpsp_schema/!go?id_=unit_id automatically.

debug Specifies debug level for the called unit and all units it will call during request servicing. This flag also forces interpreted execution of all units involved in servicing the request. Values for this parameter may be **0,1,2** and **3**, with 0 turning debugging off and 3 turning on the most verbose debug output. When specified, this parameter overrides individual settings of DEBUG_LEVEL unit attribute for affected units.
This parameter is only honored when calling user is authenticated through Development Interface. Ordinary anonymous users will have this parameter ignored. This behavior prevents ordinary users from seeing debug information they normally shouldn't see.

You should not assign your own parameters any of the standard parameter names (`id_`, `ctx`, `op`, `ln` or `debug`), as their values will be interpreted by the hub procedure and will not be passed to the unit.

DYNAMIC PSP SESSIONS AND SECURITY.

Dynamic PSP sessions are the means for maintaining logically contiguous uninterrupted sessions in an inherently discrete environment of the HTTP protocol, where the session state between request-response sequences is not automatically maintained. Dynamic PSP sessions are created and maintained automatically by the Kernel. Dynamic PSP sessions are designed so that they can work in both stateless and stateful modes, though for performance reasons we do not recommend using stateful mode for your Oracle web applications as they can quickly exhaust your Oracle server resources because there is no easy way for detecting and releasing closed connections. In stateless mode, Oracle connection is not dedicated for single logical session, but is reused to service new requests as they come in. In this mode, it is up to the application to maintain important session state information between HTTP requests. HTTP protocol provides one simple mechanism for state management, called HTTP Cookies, which allows storing state information on client between calls and sending it back to server with each new request so that the server can 'recall' its state. Dynamic PSP also provides server-side mechanism for preserving session state by creating and maintaining sessions (which you may treat as logical connection identifiers) and providing API for storing and retrieving server-side variables bound to these sessions, which are naturally called "session variables". This approach allows minimizing network traffic needed to pass session state data between the client and the server by sending only relatively short session identifier in an HTTP cookie and storing the rest of the session state information on the server, and protects session state data from sniffing and modification, as it is never passed over the network. Note that Dynamic PSP sessions are defined on transport level and generally should not be used for authentication and user identification. Session identifiers are mere tokens that allow Dynamic PSP to maintain contiguous logical connections between the browser (client) and the server in discrete physical connections environment of the HTTP.

When the `GO` hub procedure is invoked for the first time in an HTTP session (that is, in a new browser window), the Dynamic PSP Kernel generates a globally unique DPSP2 session identifier and sends requesting client the session cookie named '**`nn-psp-sessid`**' with the value of this session identifier. To prevent session hijacking, the Kernel always verifies request source IP address and only considers session identifier sent by client (browser) in a cookie valid if the IP address of the client is the same as that for which session identifier was initially generated; otherwise it automatically starts new session (although this protection may not work for proxied requests which all look originating from the same IP address – the address of the proxy server). All session data, excluding server-side contexts (see `WrapCtx` built-in function), which are session-independent, is bound to this session identifier and is stored on the server for configurable amount of time (default is 7 days). The browser automatically disposes the session cookie when closed, but the session data is kept on the server for configurable amount of time for debugging purposes. However, the session cookie is considered expired by the Kernel after 24 hours of inactivity within this HTTP session (for example, when you leave the browser window open but do not perform any activity in it for 24 hours or more).

Built-in authentication mechanism, which controls access to the Unit Commander, Unit Editor, Generated PL/SQL Code Viewer and other components of Development Interface, uses session variables to identify the user invoking any of these facilities in addition to user name and password, and session variables are bound to the session identifier. Since the HTTP traffic, including cookies and authentication information, can be intercepted by malicious party and then used to impersonate authenticated users by replaying the request with this information included in request header, it is recommended that sensitive operations, like unit editing, be performed over secure SSL connection, for example, by using Apache `<LocationMatch>` directive (text below is wrapped for readability, actually `<LocationMatch>` should occupy single line):

```
<LocationMatch
"^(*.)![Gg][Oo](*.)[Oo][Pp]=([Uu][Cc][Oo][Mm]|[Ee][Dd][Ii][Tt]|[Vv][Ii][Ee][Ww])(*.)$">
<IfDefine SSL>
```



```
 SSLRequireSSL # enforce SSL connection for such requests if SSL is supported
</IfDefine>
</LocationMatch>
```

which will catch all special requests to the hub procedure, and require secure connection for such requests, if available. Please note that because mod_plsql's URL parser is case-insensitive, we had to extend the regular expression to catch all keywords in any [mixed] case, since, for example, `!go` and `!GO` are both valid invocations of the hub procedure and mod_plsql makes no difference between them.

II CHAPTER II. DPSP2 SYNTAX AND BUILT-IN FUNCTIONS AND PROCEDURES.

This chapter describes the Dynamic PSP Version 2 Syntax and Built-in functions and procedures.

ORACLE PSP PAGES AND PACKAGES VS. DPSP UNITS.

Oracle PSP and Dynamic PSP both employ Oracle PL/SQL as server-side programming language for retrieving or modifying data in the database, generating page content and do a lot of other things on server before returning the page to the requesting client. There are significant differences between the two though.

In Oracle PSP you write a dynamic page as a text file and then use a special utility called `loadpsp` that parses your page, creates a stored procedure source code from it and loads/compiles this stored procedure into Oracle RDBMS. Each external parameter you define in PSP page becomes a parameter of the stored procedure. This limits number, names and types of external parameters for this particular PSP page to those you explicitly declare in the PSP file. Each time a change in functionality of the page is required, you need to edit the source file (.psp file) and reload the page into Oracle, which in turn triggers code regeneration and recompilation. Any syntax error during this process terminates loading and you need to fix it before PSP page can be loaded into Oracle. If the page is syntactically correct, but references inexistent database objects, it will fail compilation and the page will be invalidated. You will need to fix the page code and load it again before it will come back to life.

In contrast, each DPSP code block is stored in special table in the database, has *unique numeric identifier* (ID), and is parsed and compiled *dynamically* when accessed, or compiled into a PL/SQL package. These blocks are called *DPSP units*. A DPSP unit can represent complete page or a logical block (subroutine) that can be called from other DPSP units allowing you to effectively partition and reuse your code. DPSP units can be called from each other by ID or name, and can accept unlimited number of external named parameters. DPSP units are dynamically parsed and compiled and all changes you made to them become instantly available as soon as you save edited unit. You can also compile a unit into a PL/SQL package to speed up its execution by removing parsing overhead. Any syntax errors will be reported when unit is executed for the first time or compiled into a PL/SQL package. DPSP units can be treated as procedures that output directly to the client or as functions that return result in a string that can then be modified before returning it to client. Both forms are automatically available with no changes to the unit code.

When new unit is created, it is assigned a unique numeric ID, and a name. Although name of the unit can be changed later on, ID cannot be changed and will always identify the same unit once it is created.

Negative unit IDs are reserved for system units and generally should not be used for your custom units.

DPSP units can be compiled into PL/SQL packages, for example, when you want to protect them from further modification or source code assessment (in this case, you can extract unit package source after compilation and obfuscate it using WRAP utility). When a DPSP unit is debugged and complete, you set several attributes to enable its compilation and compile it into a package. When a unit is executed, the DPSP interpreter checks if a package exists and executes it if found and the package is valid, otherwise it retrieves the unit source code and interprets it. Compiling DPSP units into packages speeds up their execution by removing parsing overhead. Since Oracle9*i*, PL/SQL packages can be compiled natively (that is, into native platform-specific shared libraries) and native compilation improves execution speed even further, especially for static content and compute-intensive code.

GENERIC DPSP PAGE OUTLINE AND DPSP TAGS.

Generic DPSP page includes standard static HTML text plus dynamic content, which can consist of data retrieved from the database or submitted by a visitor through a form, blocks of HTML that are displayed conditionally, etc.

Special tags are used to deliver the dynamic content. These tags start with **<%** character sequence and end with **%>** sequence, as with most server pages languages. Sometimes you may want to output these character sequences as is and don't want them to be treated as PSP tag start/stop marks. In this case you need to 'escape' them: **<\%** and/or **%\>** sequences will be output as literal <% and %> character sequences. This is useful, for example, when you want to output some sample PSP code from your page.

Definitions of DPSP tags are as follows:

<%-- comment --%>

This tag surrounds Dynamic PSP comment. Everything enclosed within this tag, including other DPSP tags, is considered comment and ignored.

<% PL/SQL statement; … %>

This tag surrounds a block or fraction of PL/SQL code. All such fractions are combined into an anonymous block or compiled into a PL/SQL package (the way a unit is processed is determined by its attributes). The code should follow all PL/SQL requirements, restrictions and syntax. You can access any standard and custom Oracle objects (procedures, tables, views, etc.) in these blocks as well as some predefined DPSP functions. Oracle database security controls access to all Oracle objects, like tables, views or packages, the same way as for any other SQL or PL/SQL code, that is DPSP unit gains only rights of the database user that owns or executes it (depending on AUTH_ID attribute value for particular unit).

Example:

```
<% for r in (select * from all_users) loop %>
  any HTML/DPSP to put in a loop here
<% end loop; %>

<% if someVariable = someValue then %>
  This will only be seen if someVariable = someValue
<% else %>
  This will only be seen if someVariable <> someValue
<% end if; %>
```

In the first example, we do a **SELECT** from **ALL_USERS** view and the calling user should have a SELECT permission on that view. The second example demonstrates conditional display of static HTML. Note that each complete PL/SQL statement should be terminated with semicolon (for example, if … else … end if;).

<%= PL/SQL expression %>

This tag surrounds an expression. The result of the expression replaces the tag in resulting page. Note that the expression should NOT be terminated with semicolon, unlike PL/SQL statements in other types of DPSP tags. Any valid PL/SQL expression is allowed in this tag.

Example:

```
Today is <%= TO_CHAR(SYSDATE, 'Dy, DD Mon, YYYY', 'NLS_Date_Language=American') %>.
```

<%! PL/SQL declaration; … %>

This tag surrounds a global type, variable or subroutine (procedure or function) definition block. You can combine several definitions in one block. Generally, you should follow conventions for Oracle DECLARE block of an anonymous PL/SQL block. The tag can be issued anywhere in the DPSP unit as many times as you want, provided that you do not redeclare the same variable, type or subroutine. All declarations will be combined into a single DECLARE block before compilation/execution. You should not issue **DECLARE** keyword within this element. All variables, types and subroutines declared within these tags become GLOBAL, that is they will be visible throughout the whole unit, unlike inline, or local **DECLARE**s which are visible only within a subsequent **BEGIN..END;** block.

Example:

```
<%-- global declarations --%>
<%!
  TYPE myArrayType IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;

  MyArray myArrayType;
  Var1 VarChar2(30);
  Var2 Number := 0;

procedure printDate(d date)
is
begin
 prn(d, 'DD Mon, YYYY', 'nls_date_language=American');
end;

%>
<%-- local declaration --%>
<%
declare
  someVar number;
begin
  some code – someVar is visible here;
end;
%>
<%-- someVar will not be visible here --%>
Current date is <% printDate(sysdate); %>.
```

(See Example 3 above for example of usage of all these tags in a complete dynamic page.)

With PPSP (Procedural PSP) preprocessor, this tag can also be used to declare local variables in procedures.

<%@ PSP preprocessor directive %>

This tag is used by DPSP and PPSP (Procedural PSP) preprocessors to include code of other units into currently processed unit, define procedures and main unit body. DPSP and PPSP recognize the following directives:

`include [{file=|unit=}]["]{<unit_id>|<unit_name>}["]`

This directive instructs preprocessor to include the code from referenced unit when parsing the code. For compatibility with Oracle PSP syntax, `file="<unit_id>|<unit_name>"` syntax is accepted, but is equivalent to `unit="<unit_id>|<unit_name>"` syntax. Double quotes around the unit id or name are optional; if they are omitted, unit name is parsed until first word separator (usually space). You may also omit `file=`/`unit=` and specify unit without these prefixes. Unlike `exec()` call at runtime, which dynamically executes referenced unit and embeds the result of execution into the output stream, this directive includes the referenced unit code at parse time, as if it was copied and pasted directly. Thus, if you compile a unit, which includes another unit, and then change the included unit, these changes will not be incorporated into the importing unit, you will need to recompile the including unit to embed changed code. This directive was included for compatibility with Oracle PSP. Directive is recognized by both DPSP and PPSP preprocessors.

Example:

```
<%-- include the 'included unit' code --%>
```



```
<%@include unit="included unit"%>

<%-- include the code of unit 10001 --%>
<%@include 10001%>
```

`procedure <procedure_name>[(parameter spec [, parameter spec[, …]])]`

This directive defines an internal procedure (that is, a procedure which is visible only to the unit) with some optional parameters. You can define as much procedures as you want. Order of declarations is only important if you want to call one defined procedure from the other – in this case the called procedure must be declared before calling procedure. Global variables declared with <%! %> tag outside the procedure body are visible within these procedures, but variables declared in procedure definition are only visible within procedure body. Directive is recognized by PPSP preprocessor only.

You can also declare procedures and functions in <%! %> declarations block as described above.

`main`

If `procedure` directive is used, `main` directive defines the main unit body. It is mandatory to issue <%@main%> to identify main body if `procedure` directive is used anywhere in the unit code. You can issue this directive before or after `procedure` directives, preprocessor will put the code in right order automatically. Directive is recognized by PPSP preprocessor only.

Example:

```
<%! num integer := 0; %>
<%@procedure onerow(nm in varchar2)%>
 <%! localVar integer := 0; -- this variable will only be visible in onerow()%>
  <% num := num + 1; %>
  <tr>
    <td align="center"> <%=to_char(num)%> </td>
    <td align="right"> <%=nm%> <%='&'%>nbsp; </td>
    <td> "<%=owa_util.get_cgi_env(nm)%>" </td>
  </tr>
<%@main%>
<table border="1" cellpadding="1" cellspacing="1" cols="3"
       style="font-size:10pt" width="100%">
  <caption align="left" style="color:blue;font-size:12pt;font-weight:bold">
    CGI environment variables:
  </caption>
<% -- do the printing using our procedure
  onerow('GATEWAY_INTERFACE');
  onerow('REMOTE_HOST');
  onerow('REMOTE_ADDR');
  . . .
%>
</table>
```

Please note that Oracle PSP uses <%@ %> tag for slightly different purposes – for defining page parameters and various other attributes. Most Oracle PSP '@' directives are currently ignored by DPSP and PPSP preprocessors.

DPSP TAG SIZE LIMITS.

It is important to note that current implementation of Dynamic PSP applies size limit of 32K (32765) bytes for contiguous DPSP tags. That is, every single DPSP tag should not exceed 32K in size, or an error will occur when unit is executed or compiled. This limit does not apply to FLAT preprocessor, which does not process source code. DPSP and PPSP preprocessors are affected by this limit though. To overcome this limit, you should break your PSP code into chunks that do not exceed 32K in size.

STANDARD DPSP2 TYPES.

The Dynamic PSP 2 Kernel defines several data types and constants, which are used throughout the system and should be used in DPSP2 units to pass parameters and receive results from API calls and other units. All DPSP2 types are defined in **NN$TP** package. These types and constants are:

```
NEWLINE_CHAR      CONSTANT CHAR(1) := CHR(10);

TYPE VARCHAR_ARRAY IS TABLE OF VARCHAR2(32767) INDEX BY BINARY_INTEGER;
TYPE NUMBER_ARRAY  IS TABLE OF NUMBER          INDEX BY BINARY_INTEGER;
TYPE DATE_ARRAY    IS TABLE OF DATE            INDEX BY BINARY_INTEGER;
TYPE R_PAIR        IS RECORD (NAME VARCHAR2(32767), VALUE VARCHAR2(32767));
TYPE PAIR_ARRAY    IS TABLE OF R_PAIR          INDEX BY BINARY_INTEGER;
TYPE CURSORTYPE    IS REF CURSOR;

VARCHAR_ARRAY_EMPTY      VARCHAR_ARRAY;
NUMBER_ARRAY_EMPTY       NUMBER_ARRAY;
DATE_ARRAY_EMPTY         DATE_ARRAY;
PAIR_ARRAY_EMPTY         PAIR_ARRAY;
```

NEWLINE_CHAR constant is a shortcut to ASCII character 10, which is new line character. Array types are used to pass arrays of strings, numbers, dates or name-value pair records. Since you can't pass NULL values for array parameters, _EMPTY constants should be used if you need to pass empty array. CURSORTYPE is generic weak cursor you can use to pass or receive any result sets for subsequent fetching and processing. These types are used in the Kernel API as well as some built-in functions and procedures. To avoid 'incompatible type' errors, custom types you define in your own packages should be SUBTYPEs of the types defined in NN$TP if you intend to use your custom types in calls to the DPSP2 Kernel or built-in functions/procedures.

The DPSP2 Kernel also defines two global SQL collection (nested table) types:

```
NN$VARCHAR_ARRAY IS TABLE OF VARCHAR2(32767);

NN$NUMBER_ARRAY IS TABLE OF NUMBER;
```

These types are used internally by the Kernel and should not be dropped. Currently no published APIs use these types.

PUBLIC DPSP2 VIEWS.

NN$V_USER_ERROR_LOG

Definition:

```
NN$V_USER_ERROR_LOG (TIME_STAMP, UNIT_ID, ERR_TEXT, ERR_MSG)
```

Purpose:

This view lists errors logged with ERRLOG() built-in function. `TIME_STAMP` is error time stamp, `UNIT_ID` is ID of the unit which logged the error, `ERR_TEXT` and `ERR_MSG` are content of corresponding ERRLOG() arguments. This view shows only errors logged under privileges of current Oracle user and allows to delete such errors by deleting from the view. Unhandled exceptions thrown in DPSP units are also automatically logged in this view. Development Interface features Unit Error Viewer module that provides simple interface for viewing unit errors registered in this view.

BUILT-IN FUNCTIONS, PROCEDURES AND VARIABLES.

Each DPSP2 unit has several DPSP2 functions and procedures readily available. These functions and procedures allow retrieving call parameters and environment variables, invoking other DPSP2 units, facilitating unit debugging and profiling, and various other tasks. Below are specifications and descriptions of all built-in functions and procedures along with invocation examples.

CURRENT_ID

This is a constant of type `NUMBER`, which holds current unit ID. You can use this constant to quickly reference the unit, for example, when generating hypertext links to the unit itself. This constant holds the same value as the one that will be returned with `env('THIS_ID')` call. This constant is globally accessible within the unit.

Example:

```
<form action="!go">
<!-- unit itself will process the form -->
<input type="hidden" name="id_" value="<%= current_id %>">
```


EXEC

Executes (calls) a DPSP unit (page). Called unit's output is included into result document in position where call is made; it is not available to caller for additional processing.

Definition:

```
PROCEDURE Exec( ID NUMBER, ARGS VARCHAR2 DEFAULT NULL );
PROCEDURE Exec( LN VARCHAR2, ARGS VARCHAR2 DEFAULT NULL );
```

ID is ID of the DPSP unit.

LN is logical name of the DPSP unit.

ARGS is a string of named parameters (arguments) to be passed to the unit in URL-encoded format (that is, each parameter is specified by **name=value** pair and parameters are separated by ampersand "**&**"). You can pass arrays as parameters by specifying them as **'name=value1&name=value2&...'**. Caller unit parameters are also available to the called unit (inherited from caller unit) unless overridden by **ARGS** parameter list.

Example:

```
<% exec('object', 'param1=value1&param2=value2'); %>
```

The above example will execute DPSP unit identified by name 'object' passing it two parameters and will include result of execution into the resulting document.

EXECF

Calls a DPSP unit and returns result of its execution in a string array rather than including it into result document.

Definition:

```
FUNCTION ExecF( ID NUMBER, ARGS VARCHAR2 DEFAULT NULL )
  RETURN NN$TP.VARCHAR_ARRAY;
FUNCTION ExecF( LN VARCHAR2, ARGS VARCHAR2 DEFAULT NULL )
  RETURN NN$TP.VARCHAR_ARRAY;
```

These functions are the same as **EXEC** procedures, but they return result of their execution in a string array rather than embedding them in the caller's output stream. Called unit inherits caller's parameters unless they are overridden in **ARGS** parameter list.

Example:

```
<%! Res NN$TP.VARCHAR_ARRAY; %>
<%
Res := execf('an object', 'param1=value1&param2=value2');
prn(Res);
%>
```

The above example will execute 'an object', receive execution result into `Res`, and then just output it into result document with array-oriented version of the **PRN** procedure without any processing, effectively emulating **EXEC** procedure. You might want to do some processing though before printing, like replacing or removing some text within `Res`.

EXECFL

EXECFL function is similar to **EXECF** function, but returns CLOB (Character LOB) as result. **L** in function name refers to LOB.

Definition:

```
FUNCTION ExecFL( ID NUMBER, ARGS VARCHAR2 DEFAULT NULL )
  RETURN CLOB;
FUNCTION ExecFL( LN VARCHAR2, ARGS VARCHAR2 DEFAULT NULL )
  RETURN CLOB;
```

Example:

```
<%! Res CLOB; %>
<%
Res := execfl('an xml object', 'param1=value1&param2=value2');
prn( ProcessXML(Res, execfl('stylesheet.xslt')) );
%>
```

In this case we execute 'an xml object' unit, which returns XML document that you want to translate using some XSLT stylesheet defined in another DPSP unit. It is assumed that you have a function, ProcessXML, which takes a CLOB with XML document and a CLOB with XSLT stylesheet to apply to the document as parameters and returns result of the transformation as NN$TP.VARCHAR_ARRAY. Resulting array is then output to the client with array-oriented version of the **PRN** procedure. To find out more about XML/XSLT support in Oracle8*i* and Oracle9*i*, please consult Oracle XDK (XML Development Kit) documentation on Oracle Technology Network (http://technet.oracle.com).

PARAM

Returns value of a named parameter passed to the Unit. Units invoked from other units inherit caller's parameters unless the caller explicitly redefined them. Parameter list of the caller unit is restored upon return from called unit.

Definition:

```
FUNCTION Param(NAME   IN VARCHAR2,
               DEF    IN VARCHAR2 DEFAULT NULL,
               MAXLEN IN PLS_INTEGER DEFAULT 32765) RETURN VARCHAR2;
```

NAME is name of the unit parameter

DEF is value to be returned if value for the specified parameter is not available (parameter was not passed to the unit)

MAXLEN is maximum length of return value. Function will return substring of original value passed to the unit from first character up to this length (use this parameter to avoid buffer overflows and 'value too large' errors.)

If you intend to access a parameter value more than once in your unit, it is recommended (though not necessary) to declare a variable and assign it parameter's value once at the beginning and then reference the variable instead – this will improve unit execution time by eliminating unnecessary repetitive function calls and parameter searches.

Examples:

```
<%! Var_Param_Name varchar2(50) := param('param_name',null,50); %>

<% if upper(param('param_name')) = 'CONDITION_VALUE' then %>
 condition is true
<% else %>
 condition is false
<% end if; %>
```


APARAM

Returns array of values of named array parameter. Units invoked from other units inherit caller's parameters unless the caller explicitly redefined them. Parameter list of the caller unit is restored upon return from called unit.

Definition:

```
FUNCTION AParam(NAME VARCHAR2) RETURN NN$TP.VARCHAR_ARRAY;
```

NAME is name of the array parameter.

If value for the parameter is not available, NN$TP.VARCHAR_ARRAY_EMPTY is returned.

If you intend to access a parameter value more than once in your unit, it is recommended (though not necessary) to declare a variable and assign it parameter's value once at the beginning and then reference the variable instead – this will improve unit execution time by eliminating unnecessary repetitive function calls and parameter searches.

Example:

```
<%! PA NN$TP.VARCHAR_ARRAY := aparam('param_name'); %>
<% if PA.count > 0 then
    for i in PA.First..PA.Last loop %>
     <%= PA(i) %><br>
<%  end loop;
   end if; %>
```

This example will print all values of array parameter, separating each with
 tag (note, that you can achieve the same result by <% prn(PA, '
'); %>
 - you will only need to explicitly print
 tag once after the PRN call, because PRN does not append the delimiter to the last element.

ENV

Retrieves HTTP Server (CGI) or DPSP2 session environment variable.

Definition:

```
FUNCTION env( VAR_NAME VARCHAR2 ) RETURN VARCHAR2;
```

VAR_NAME is name of environment variable you want to retrieve (case-insensitive).

Some standard variables:

DOC_ACCESS_PATH	prefix for accessing files in the document table, for example 'files'. The gateway automatically recognizes requests to …/DAD/<doc_access_path>/… as requests for content stored in the document table.
DOCUMENT_TABLE	Name of the document table as specified in the current DAD.
HTTP_USER_AGENT	User-Agent string (for identifying browser name and version)
REMOTE_USER	remote user name (that is, user currently logged in on client system – do not mix this with Dynamic PSP users)
SERVER_PORT	HTTP Server port number
SERVER_NAME	HTTP Server name
REQUEST_METHOD	HTTP Request method ('POST', 'GET', 'HEAD', etc.)
REMOTE_HOST	Remote host name (may be empty string)
REMOTE_ADDR	Remote host IP address (always defined)
SERVER_PROTOCOL	Server protocol name and version (for example, 'HTTP/1.1')
HTTP_PRAGMA	HTTP pragmas
HTTP_HOST	HTTP Host (virtual host name accessed, as specified in request, not necessarily the same as SERVER_NAME)
HTTP_ACCEPT	HTTP Accept string (comma-delimited list of preferred content types)
HTTP_ACCEPT_ENCODING	HTTP Accept-Encoding string (preferred content encodings, comma-delimited)
HTTP_ACCEPT_LANGUAGE	HTTP Accept-Language string (preferred content languages, comma-delimited)
AUTHORIZATION, HTTP_AUTHORIZATION	Authorization information (if any), value depends on authorization method.
HTTP_COOKIE	Cookie string (semicolon-delimited list of all cookies passed in by client)
REQUEST_PROTOCOL	Request protocol string
CALLER_ID	Identifier of DPSP unit that called this unit
CALLER_NAME	Name of DPSP unit that called this unit
THIS_ID	Identifier of current DPSP unit (same value as Current_Id)
THIS_NAME	Name of current DPSP unit (value of the name attribute of the unit)

DPSP_CURRENT_USER	Name of the user currently logged into the Development Environment
VERSION	DPSP version string (full)
SHORT_VERSION	DPSP version string (short)
DPSP_SESSION_ID	Current Dynamic PSP Session identifier
INNER_ID	Inner unit identifier for outer units called through FRAME_PAGE attribute.

Please note that the above list is not complete and there may be more environment variables passed in by the gateway. Consult mod_plsql documentation for complete list of mod_plsql CGI environment variables and their meanings, and instructions on how to set your own CGI environment variables to be passed to your application.

Most of these variables are provided by the gateway (mod_plsql or servlet) and are retrieved from CGI environment set up by the web server. Any CGI variable available to the gateway, which is not explicitly disabled in the gateway configuration, can be retrieved using this function.

CALLER_x and **THIS_x** variables are standard DPSP2 variables used for tracing execution of units and other various things. These variables are set up by the DPSP Kernel upon unit execution. You can use them to reference the unit itself and its caller without prior knowledge or the need for hard coding of unit name or ID attributes (for example, when generating HTTP anchors).

DPSP_SESSION_ID identifies current Dynamic PSP session, which is always created and maintained automatically by the Kernel. Server-side session variables are bound to this identifier. You may use this environment variable to identify Dynamic PSP session and pass its identifier to your legacy OWA code so that it can connect to the DPSP session and set or retrieve session variables.

INNER_ID identifies the inner unit for outer units called through FRAME_PAGE attribute. Combined, this variable and FRAME_PAGE attribute allow for complex automatic unit call stacking.

It is important that you **do not** use OWA_UTIL.GET_CGI_ENV function for retrieving CGI environment variables. The Dynamic PSP Kernel automatically retrieves and prepares CGI environment for you. The Dynamic PSP Kernel does not depend on OWA packages and only calls them internally when it needs to deliver the final content to the client and detects that mod_plsql is in use. OWA packages may not even exist on your system.

Examples:

```
<% if env('caller_id') is not null then
 -- generate a call trace
 debug('Called from "' || env('caller_name') || '"<br>');
end if; %>

<%-- using HTTP environment for conditional code execution --%>
<% if upper(env('HTTP_USER_AGENT')) like '%MSIE 5%' then %>
Internet Explorer 5.x-specific HTML here
<% else %>
Other HTML here (for IE4, Netscape or Opera)
<% end if; %>


<a href="!go?id_=<%= env('THIS_ID') %>">This is myself.</a>

<%-- an example of a FRAME_PAGE unit --%>
<SCRIPT type="text/javascript">
// my common functions
</SCRIPT>
<%-- call the inner unit that triggered me --%>
<% exec(to_number(env('INNER_ID'))); %>
```


PRINT, PRN

Print any text or HTML into result document from DPSP code.

Definition:

```
PROCEDURE Print(TEXT VARCHAR2);
PROCEDURE Print(TEXT NN$TP.VARCHAR_ARRAY, DELIMITER VARCHAR2 DEFAULT NULL);

PROCEDURE PRN(TEXT VARCHAR2);
PROCEDURE PRN(TEXT NN$TP.VARCHAR_ARRAY, DELIMITER VARCHAR2 DEFAULT NULL);
```

TEXT is either a string or an array of strings

DELIMITER is a string to insert between each array element as they are printed

There are two versions of both **PRINT** and **PRN** functions. One of them accepts a string and prints it to the result document. Second version accepts array of strings and an optional delimiter string and prints array elements optionally separating them with specified delimiter. Second version may be useful for printing delimited lists or just big chunks of text, which does not fit into PL/SQL limit of 32KB for VARCHAR2 type. **PRINT** puts a line break (ASCII code 10 character) after printed text while **PRN** doesn't. Array version of **PRINT** puts a line break after each array element unless the delimiter is explicitly specified, in which case line break is only put after the last element. Note that NULL array elements will also have delimiter appended.

It is important that you **do not** use HTP and HTF OWA packages in your DPSP code, as their output will be ignored and will not make it to the client. The Dynamic PSP Kernel does not depend on OWA packages and only calls them internally when it needs to deliver the final content to the client and detects that mod_plsql is in use.

Examples:

```
<% print('HTML New Line<BR>'); %>

<%! aList NN$TP.VARCHAR_ARRAY := aparam('listvals'); %>
List: <% prn(aList, ', '); %><br>
```


ERRLOG

Puts a message into error log table. Current Oracle user name, DPSP unit ID and time stamp will be automatically added to the error message. Logged errors are then visible via DPSP$V_USER_ERROR_LOG view and in Unit Error Log viewer. Any exceptions raised in DPSP unit, which are not caught in the unit code, are handled automatically and logged in the view as well.

Definition:

```
PROCEDURE ErrLog(ERR VARCHAR2, MSG VARCHAR2 DEFAULT NULL);
```

ERR is Oracle error string or any other error description

MSG is an optional message to be appended to ERROR – specify any relevant information here or leave this parameter blank.

Logged errors may be viewed by querying the NN$V_USER_ERRORS_LOG view or using the Unit Error Log viewer in Unit Commander.

Example:

```
<%
 exception when others then
  errlog(sqlerrm, 'This happened here when trying to do that.');
%>
```


DEBUG

Prints debugging information into output stream. Depending on DEBUG_LEVEL attribute value for the unit, Debug procedure will output the message as is (if level > 1), as HTML comment (if level = 1), or will silently exit (level = 0).

Definition:

```
PROCEDURE Debug(TEXT VARCHAR2 DEFAULT NULL);
```

TEXT is a string to be written into output stream if debug level is greater than 0 for current DPSP unit.

Example:

```
<% debug('Trying to execute the code…'); %>
```


WRAPCTX

This function is intended to obfuscate the URL-encoded parameter list, called context. It takes URL-encoded parameter list as argument and returns a globally unique string identifying the stored server-side context, which can later be used to retrieve the parameter list from internal server-side storage. Second optional argument can specify parent context (already encoded using this function) to combine with this context. When decoded, a context with parent will receive parameters encoded to this context itself, and to its parent (and parent's parent, etc.)

Definition:

```
FUNCTION WrapCtx (ARGS VARCHAR2, PARENT VARCHAR2 DEFAULT NULL) RETURN VARCHAR2;
```

ARGS parameter specifies the URL-encoded parameter list to store in persistent storage area.

PARENT is optional parent context identifier (which is the output of `WrapCtx` for parent context).

This function generates globally unique string of 50 characters, and stores **ARGS** and **PARENT** values in internal context table along with this value. To decode the generated context, `GO` hub procedure should be invoked with `ctx=<wrapctx output>` parameter. The hub procedure will automatically retrieve the specified context and reconstruct the parameter list from it. Context parameter list may include other standard parameters, like `id_` or `op`, so `ctx` may be the only unit call parameter. Generated contexts are stored for duration of DPSP2 session (which is configurable in DPSP2 Registry, `NN$PSP/Kernel/SessionLife` item).

Example:

```
<%! ctx varchar2(50); %>
<% ctx := wrapCtx('id_=100&param1=value'); %>
<a href="!go?ctx=<%= ctx %>">
 Obfuscated call #1
</a>
<a href="!go?ctx=<%= wrapCtx('param2=value',ctx) %>">
 Obfuscated call #2, inherits 'id_=100&param1=value' from first context
</a>
```


XTIME

Returns time elapsed from beginning of current unit's code execution (for profiling purposes) in seconds, in string format '9990D99'.

Definition:

```
FUNCTION XTime RETURN VARCHAR2;
```

This function accepts no parameters.

Example:

```
Time elapsed: <%= xtime %> sec.
```


SETCOOKIE

Use this procedure to set any cookies you want to send to the requesting client. This procedure can be called anywhere in the code, all cookies set using this procedure are combined and sent to the requesting client when unit processing is finished. You can retrieve the cookies sent back by client by using `getCookie` function.

Definition:

```
PROCEDURE setCookie( NAME VARCHAR2, VAL VARCHAR2,
                     MAXAGE NUMBER DEFAULT NULL, PATH VARCHAR2 DEFAULT NULL,
                     DOM VARCHAR2 DEFAULT NULL, COMM VARCHAR2 DEFAULT NULL,
                     SECURE BOOLEAN DEFAULT NULL);
PROCEDURE setCookie( NAME VARCHAR2, VAL VARCHAR2,
                     EXPIRE DATE DEFAULT NULL, PATH VARCHAR2 DEFAULT NULL,
                     DOM VARCHAR2 DEFAULT NULL, COMM VARCHAR2 DEFAULT NULL,
                     SECURE BOOLEAN DEFAULT NULL);
```

The procedure is offered in two overloaded forms, one accepting numeric `Max-Age` value as **MAXAGE** parameter, which is cookie life duration in seconds, while second accepts cookie expiration date as **EXPIRE** parameter (`DATE`). Other parameters are as follows:

NAME - cookie name

VAL - cookie value

PATH - optional path this cookie is valid for

DOM - optional domain name this cookie is valid for

COMM - optional cookie comment

SECURE - optional flag specifying that this cookie should only be sent over secure channel

It is important that you **do not** use `OWA_COOKIE` OWA package to send cookies, as the cookies set using this package will not be actually sent to the client. Use only DPSP `setCookie` procedures to set cookies from DPSP units. The Dynamic PSP Kernel does not depend on OWA packages and only calls them internally when it needs to deliver the final content to the client and detects that mod_plsql is in use.

Important note: mod_plsql does NOT send RFC-2109 cookies, it only supports old Netscape proposal and filters out all cookie attributes it does not recognize. In particular, `Max-Age` and `Comment` attributes are filtered out. To work around this behavior, `setCookie` emulates `OWA_COOKIE` method for sending cookies if it detects that mod_plsql is the gateway used to access DPSP.

Example:

```
<%
if registered_user then
 setCookie('session', session_id, sysdate+7); -- session will expire 7 days later
else
 setCookie('session', session_id, 3600); -- session will expire 1 hour later
end if; %>
```


GETCOOKIE

Use this function to retrieve cookies sent by the client (browser) along with request. You can also use this function to retrieve cookie attributes, like Domain, Path and Port, which you may have specified when setting the cookie.

Definition:

```
FUNCTION getCookie( NAME VARCHAR2,
                    INST INTEGER DEFAULT 1,
                    ATTR VARCHAR2 DEFAULT NULL ) RETURN VARCHAR2;
```

where **NAME** is case-insensitive cookie name, **INST** is optional cookie instance (there may be several cookie values with the same name passed in, for example if current domain matches two or more Domain attributes for the cookie), which defaults to first instance; and **ATTR** is optional case-insensitive cookie attribute name you want to retrieve with `NULL` identifying cookie value itself. If you specify **ATTR** as `'$Domain'`, `'$Path'` or `'$Port'`, Domain, Path or Port attribute respectively will be returned instead of cookie value. `'$Version'` attribute will always return value of `'0'` or `'1'` as these are the only standard versions defined in HTTP State Management Mechanism specifications (RFC-2109 and then RFC-2965), with `'0'` being an indication that cookie attributes are not passed in at all. Note that cookie attributes are not passed in by mod_plsql and will only be accessible if the DPSP unit is accessed through JOPA servlet.

You can still use `OWA_COOKIE` package to retrieve cookies if DPSP is accessed through mod_plsql, but this will add an overhead for double parsing the cookies – the DPSP Kernel will parse them automatically for you, and if you call any `OWA_COOKIE` routine, it will parse the cookies again. Additionally, if the gateway is not mod_plsql, `OWA_COOKIE` will not be able to retrieve any cookies. We recommend using only DPSP interfaces for setting and retrieving cookies.

Examples:

```
<%! session_id  varchar2(20) := getCookie('session', 1);
    sess_domain varchar2(100) := lower(getCookie('session', 1, '$domain')); %>
<% if getCookie('session', 1, '$version') != '0' then %>
<%-- we have cookie attributes passed in --%>
<% if sess_domain = 'mysite.com' then %>
  Welcome to mysite.com
<% elsif sess_domain = 'www.mysite.com' then %>
  Welcome to www.mysite.com
<% end if; %>
<% end if; %>
```


SETHEADER

This procedure sets an HTTP header to be returned to requester as part of response. This procedure can be called anywhere in the unit code.

Definition:

```
PROCEDURE setHeader( NAME VARCHAR2, DATA VARCHAR2, ATTR VARCHAR2 DEFAULT NULL );
```

NAME is HTTP header name, **DATA** is header value and **ATTR** is optional header attributes. You can output any standard HTTP header. There is one special header, which is processed by mod_plsql and not sent to client: `Status`, which is not part of HTTP standard and is used by mod_plsql and JOPA to receive HTTP response status value from the PL/SQL code.

If you set the same header twice or more times, this will result in this header being duplicated in output. Exceptions are `Status` and `Content-Type`, which will be replaced with new values. `Content-Length` header cannot be set using this procedure and will be ignored.

Example:

```
<%! user_name varchar2(2000) := null;
    password  varchar2(2000) := null;
    tmps      varchar2(4000);
    i         number; %>
<%
 tmps := env('AUTHORIZATION');
 if tmps is null then
  <%-- some mod_plsql versions pass HTTP_AUTHORIZATION variable instead --%>
  tmps := env('HTTP_AUTHORIZATION');
 end if;
 if tmps is not null then -- decode the username:password pair (Base-64 encoded)
  tmps := nn$psp_utl.decodeBase64( substr(tmps, instr(tmps,'Basic ') + 6) );
  i := instr(tmps, ':');
  user_name := substr(tmps, 1, i - 1);
  password  := substr(tmps, i + 1);
 end if;
 if user_name is null or password is null
    or (user_name != 'auth' or password != 'pass') then
   setHeader('Status','401 Please introduce yourself');
   setHeader('WWW-Authenticate','Basic realm="Secure Area"'); %>
Access to this page requires proper authentication, sorry.
<% return;
 else %>Welcome to the secure area, <%= user_name %>!<%end if;%>
```

This example will trigger login dialog to be displayed by the browser and check user credentials before displaying welcome message. If no credentials are given or after browser-predetermined number of failed login attempts, access denial message will be displayed.

REDIRECT

This procedure allows redirecting the requester to another location to obtain the result. Current content is discarded when unit execution completes, unless written to some persistent storage (table).

Definition:

```
PROCEDURE Redirect( URL VARCHAR2, FGET BOOLEAN DEFAULT FALSE );
```

URL is new location the browser should access to retrieve the result. Procedure clears current response buffer and returns 302 response code with `URL` as new location. Note that the client will access the new location using the same method it used to access original resource. For example, if the method was `POST`, then client will also `POST` to the new location. If this is not what you need, you will need to set **FGET** to `True` which will force the procedure to return 303 response (Force GET). Some older browsers do not understand 303 response code and act on 302 code the way new browsers act on 303 code. Default 302 response should be sufficient for most uses unless you are sure the browser can understand 303 code (most modern browsers, including MS IE 5.0 and above, Netscape 6.0 and above, and Opera 6 do).

Example:

```
<%
   insert into document_table(name,
                              content_type,
                              mime_type,
                              blob_content,
                              last_modified)
   values ('somefile.ext', 'BLOB', 'application/octet-stream', lob_document, sysdate);
   commit;
   if ( env('HTTP_AGENT') like '%MSIE%' ) then
     redirect('docs/somefile.ext', TRUE);
   else
     redirect('docs/somefile.ext'); -- assuming that /docs is document access prefix
   end if;
%>
```

This example will store some LOB document in the document table and then redirect client browser to the document access location. It will optionally issue 303 response code if the client browser is Microsoft Internet Explorer and 302 code otherwise.

III CHAPTER III. DPSP2 STANDARD API REFERENCE.

In addition to built-in functions and procedures, the DPSP2 Kernel also exposes API for manipulating the Kernel settings, DPSP2 units, etc. This chapter describes the API functions and procedures in detail.

NN$PSP_RT – RUN-TIME API PACKAGE.

NN$PSP_RT is DPSP2 runtime API package, which exposes several procedures and functions used for flow control and HTTP output. Most of the procedures and functions have their built-in shortcut counterparts, but some are only available in **NN$PSP_RT** package.

HTTP OUTPUT CONTROL PROCEDURES AND FUNCTIONS.

```
procedure setStatus( nCode   in  number
                   , sName   in  varchar2  default null );
```

`setStatus` procedure sets the HTTP response status code and reason phrase. **nCode** is status code, and **sName** is reason phrase which will be returned to client. Refer to HTTP/1.0 and HTTP/1.1 specifications for details on response status codes and their meaning. Some of the most used codes:

200 OK	Request successfully completed
202 Accepted	Request was accepted, but requested operation did not yet complete
204 No Content	Request successfully completed, but no additional information/response is available
301 Moved Permanently	Requested resource is available at different location, client should use new location for all subsequent requests for this resource
302 Found	Requested resource is found at different location (AKA Moved Temporarily), client should repeat request from returned location
303 See Other	Requested operation is complete, additional information is available at different location which should be requested using GET method
401 Unauthorized	Access to requested resource requires proper user credentials
403 Forbidden	Access to requested resource is forbidden
404 Not Found	Requested resource was not found
500 Internal error	Server encountered internal error while processing the request.

```
procedure redirect( sURL      in varchar2
                  , bForceGET in Boolean default false);
```

This procedure is the same as built-in `Redirect` procedure. It clears current response buffer and returns 302 response code with **sURL** as new location. Note that the client will access the new location using the same method it used to access original resource. For example, if the method was `POST`, then client will also `POST` to the new location. If this is not what you need, you will need to set **bForceGET** to True which will force the procedure to return 303 response. Some older browsers do not understand this response code and act on 302 code the way new HTTP 1.1 browsers act on 303 code. Default 302 response should be sufficient for most uses.

```
procedure setHeader( sName   in  varchar2, sData   in  varchar2
                   , sAttr   in  varchar2  default null);
```


This procedure is the same as built-in setHeader procedure. It adds the **sName** HTTP header with **sData** value and optional **sAttr** attributes to the response. For valid HTTP headers, values, attributes and their meaning, refer to HTTP/1.0 and HTTP/1.1 specifications. One special header, **Status**, which is not part of HTTP standard, is recognized by mod_plsql and DPSP2 Gateway Java Servlet (JOPA) and is used to set HTTP response status code and reason phrase. setStatus() procedure use is preferred over this header though.

If you set the same header twice or more times, this will result in this header being duplicated in output. Exceptions are `Status` and `Content-Type`, which will be replaced with new values. `Content-Length` header cannot be set using this procedure and will be ignored.

```
procedure setCookie( sName    in  varchar2, sValue   in  varchar2
                   , nMaxAge  in  number    default null
                   , sPath    in  varchar2  default null
                   , sDomain  in  varchar2  default null
                   , sComment in  varchar2  default null
                   , bSecure  in  boolean   default null);
procedure setCookie( sName    in  varchar2, sValue   in  varchar2
                   , dExpire  in  date      default null
                   , sPath    in  varchar2  default null
                   , sDomain  in  varchar2  default null
                   , sComment in  varchar2  default null
                   , bSecure  in  boolean   default null);
```

`setCookie` procedure is identical to the built-in setCookie procedure and comes in two overloaded versions, one accepting numeric **nMaxAge** maximum cookie age in seconds before it is discarded, and the other accepting **dExpire** DATE which is desired cookie expiration date and time. **sName** is cookie name and **sValue** is cookie value. Other parameters are optional: **sPath** is path the cookie is for, **sDomain** is domain name the cookie is for, **sComment** is cookie comment and **bSecure** is the flag indicating that cookie should only be sent over secure channel.

```
function getCookie( sName in varchar2
                  , iInst in integer   default 1
                  , sAttr in varchar2  default NULL ) return varchar2;
```

`getCookie` function is identical to the built-in getCookie function and provides exactly the same functionality. Refer to its built-in counterpart definition for complete description and examples of use.

```
procedure unsetHeader( sName in varchar2 );
```

This procedure removes **sName** header from HTTP headers. Certain headers will be ignored by this call, namely `Set-Cookie`, `Status` and `Content-Type`, as removing them will leave the system in uncertain state, or, in case of `Set-Cookie`, more cookies could be removed than actually needed. Use this procedure with caution as it may result in improper output if you unset some important header.

```
procedure resetContent;
```

This procedure clears all generated content from internal buffer. Headers are left intact. Use this procedure with extreme caution – it is highly destructive and discards all work you may have performed so far. Under normal conditions, you will never need to call this procedure.

```
procedure resetHeaders;
```


This procedure discards all HTTP headers you may have generated. Content is left intact. Use this procedure with extreme caution – it is destructive and discards all headers you may have set, including those you don't want to discard. Under normal conditions, you will never need to call this procedure.

```
procedure init( sMIMEType  in  varchar2  default 'text/html' );
```

This procedure is identical to the following call sequence: resetHeaders; resetContent; setHeader('Content-type',sMIMEType); and clears all internal buffers, effectively resulting in a state the DPSP was at the beginning of request processing, and optionally setting new MIME type for content that will be generated later. Use this procedure with extreme caution – it is highly destructive and discards all work you may have performed so far. Under normal conditions, you will never need to call this procedure.

SERVER-SIDE CONTEXT SUPPORT FUNCTIONS AND PROCEDURES

Functions and procedures in this group allow creating and retrieving server-side contexts for DPSP2 units. Server-side context is a set of parameters stored on server, which are identified by unique context identifier. Instead of creating hypertext links with all parameters listed in HREF, you can create server-side context with all parameters and then use the context identifier in the HREF attribute of the link.

```
function wrapContext( sArgs varchar2, sParent varchar2 default null ) return varchar2;
```

This function is the same as built-in WrapCtx function – it takes **sArgs** arguments string (URL-encoded), optional **sParent** context identifier to be merged with arguments string, puts arguments string into persistent storage area, and returns context identifier for the **sArgs** string. Context identifier then can be issued in a hypertext link as parameter to GO hub procedure: !go?**ctx=<wrapContext return value>**, and hub procedure will automatically load the arguments from stored arguments string and optional parent context into unit environment before executing the unit.

FLOW CONTROL FUNCTIONS AND PROCEDURES.

Functions and procedures in this group allow DPSP2 units to invoke other units and either embed their execution result into their own output stream, or receive it into a variable for further processing.

```
procedure exec( id_ number, args varchar2, usr varchar2);
procedure exec( ln varchar2, args varchar2, usr varchar2);
```

These procedures are the same as built-in EXEC procedures, but add **usr** parameter, which identifies the Oracle user in whose context the unit should be executed. You will rarely need to call these procedures; it is recommended that you use only their built-in counterparts.

```
function execf( id_ number, args varchar2, usr varchar2) return NN$TP.VARCHAR_ARRAY;
function execf( ln varchar2, args varchar2, usr varchar2) return NN$TP.VARCHAR_ARRAY;
function execfl( id_ number, args varchar2, usr varchar2) return CLOB;
function execfl( ln varchar2, args varchar2, usr varchar2) return CLOB;
```

These functions are the same as built-in EXECF and EXECFL functions, but add **usr** parameter, which identifies the Oracle user in whose context the unit should be executed. You will rarely need to call these functions; it is recommended that you use only their built-in counterparts.

SERVER-SIDE SESSION-PERSISTENT VARIABLES SUPPORT

This group of procedures and functions is designed for easy session state storage and retrieval. Procedures and functions in this group allow storing and retrieving named variables in the server-side storage area maintained by the Kernel. Stored data expires and is purged from the storage area after 24 hours of session inactivity.

`setSession`*XXX* functions allow to store named values in persistent storage. Functions return 0 if operation was unsuccessful, and non-zero value otherwise. **`sName`** argument is stored value's name. Value names are *case-sensitive*. ***x*`Data`** is value to store. You can store character (`setSessionString`), numeric (`setSessionNumber`), date (`setSessionDate`), and array (`setSessionStringArray`, `setSessionNumberArray` and `setSessionDateArray`) data. Value names are limited to 64 characters in size. Character data (including single character array element) is limited to 4000 single-byte characters in size (SQL limit for `VARCHAR2` data type).

```
function setSessionString(sName varchar2,
                          sData varchar2) return number;

function setSessionNumber(sName varchar2,
                          nData number) return number;

function setSessionDate(sName varchar2, dData date) return number;

function setSessionStringArray(sName varchar2,
                               aData NN$TP.VARCHAR_ARRAY) return number;

function setSessionNumberArray(sName varchar2,
                               aData NN$TP.NUMBER_ARRAY) return number;

function setSessionDateArray(sName varchar2,
                             aData NN$TP.DATE_ARRAY) return number;
```

`getSession`*XXX* functions allow to retrieve named values from persistent storage. **`sName`** argument is stored value name. Value names are *case-sensitive*. You can retrieve character, numeric, date and various array data stored using `setSession`*XXX* functions. If you attempt to retrieve non-existent value of scalar type, functions will return `NULL` unless you provide default value (***x*`Def`**) to return in such case. For array types, empty array will be returned and you cannot specify default.

```
function getSessionString(sName varchar2,
                          sDef varchar2 default null) return varchar2;

function getSessionNumber(sName varchar2,
                          nDef number default null) return number;

function getSessionDate(sName varchar2, dDef date default null) return date;

function getSessionStringArray(sName varchar2) return NN$TP.VARCHAR_ARRAY;

function getSessionNumberArray(sName varchar2) return NN$TP.NUMBER_ARRAY;

function getSessionDateArray(sName varchar2) return NN$TP.DATE_ARRAY;
```

Values stored and retrieved using these functions are bound to the Dynamic PSP session. Values stored in one session cannot be retrieved in another session, even if their names match. The Dynamic PSP Kernel tracks sessions and sets and retrieves correct values for current session automatically.

Examples:

```
<%-- store a string --%>
<! dummy number;
   myVar varchar2(4000); %>
...
<% dummy := NN$PSP_RT.setSessionString('MyVar', myVar); %>

<%-- read stored value, it will only be accessible to the DPSP session that created
     it, other sessions will see their own data only. --%>
<% myVar := NN$PSP_RT.getSessionString('MyVar'); %>
```


NN$PSP_UTL – UTILITIES PACKAGE

NN$PSP_UTL exposes some useful utility functions and procedures. The Kernel also uses some of these functions internally.

BASE64 ENCODING/DECODING FUNCTIONS

Functions in this group provide an interface to Base64 encoding algorithm, which is used to encode arbitrary data to special form that is guaranteed to consist of known subset of ASCII characters. Base64 encoding allows avoiding data corruption due to character set translations, because this known subset presents in all character sets and is not translated.

The following functions are exposed in this group:

```
function encodeBase64( sData varchar2 ) return varchar2;
```

This function encodes the input **sData** string using Base64 encoding and returns encoded representation. Since Base64 encoding produces 4 output characters for every 3 input characters, maximum length of **sData** is 24K, which will expand into PL/SQL limit of 32K for VARCHAR2 type in encoded representation. Function will throw VALUE_ERROR exception if passed data is longer than 24K, because longer strings will overflow the output buffer and will result in truncated output. This function can be used in SQL.

```
function decodeBase64( sEncoded varchar2 ) return varchar2;
```

This function decodes the input **sEncoded** string encoded using Base64 encoding and returns decoded representation. This function does not verify that **sEncoded** is indeed encoded using Base64 algorithm, so passing unencoded data to it will produce unpredictable results. This function can be used in SQL.

CRYPTOGRAPHIC HASH FUNCTIONS

Cryptographic hash functions are based on `com.nnetworks.CryptoHash` Java2 class, which is automatically loaded when Dynamic PSP 2 is installed. If Java VM is not properly initialized in Oracle, these functions will not work. Cryptographic hashes are hashes that proven to be cryptographically strong, that is chance of getting the same hash value for two different input strings is negligible. Two widely accepted cryptographic hash algorithms are MD5 and SHA. SHA is considered more secure than MD5 due to greater hash value length (160 bits against 128 bits of MD5). Hashes are often used to obfuscate passwords, because it is almost impossible to reconstruct the hashed data by its hash value. Instead of storing the password in clear text, its hash is computed and stored. When one needs to check a password for validity, he computes the hash of given password and compares it to stored hash. If hashes match, the password is correct. However, even knowing the hash value of a password one cannot reconstruct the password itself. To further obfuscate the password, you can 'salt' the original password by adding some pseudo-random characters to it before computing the hash. This way the hash for password itself and hash for salted password will not match, which diminishes possibility of brute force attacks on password hashes without knowing the salting algorithm used. These hashes are also a good way to generate encryption keys from passwords for symmetric block ciphers, especially for those using block size matching hash size.

The following functions are exposed in this group:

```
function getHashLength( sMethod varchar2 ) return number;
```

This function returns length of hash in hexadecimal characters produced by **sMethod** algorithm. Supported methods are 'MD5' and 'SHA'. Function will return 0 for any other value of **sMethod**, or in case of Java exception (unlikely to happen). This function can be used in SQL. Function returns 32 for 'MD5' and 40 for 'SHA'.

```
function getMD5(sHashThis varchar2) return varchar2;
```


This function returns hash for **sHashThis** string computed using MD5 (Rivest's Message Digest 5) algorithm as hexadecimal string. Function will return NULL in case of unhandled Java exception. This function can be used in SQL.

```
function getSHA(sHashThis varchar2) return varchar2;
```

This function returns hash for **sHashThis** string computed using SHA (Secure Hash Algorithm) algorithm as hexadecimal string. Function will return NULL in case of unhandled Java exception. This function can be used in SQL.

To convert returned hashes to original RAW hashes, use standard HEXTORAW function.

Note that DBMS_OBFUSCATION_TOOLKIT standard package shipped with Oracle8i Release 3 (8.1.7) exposes several overloaded procedures and functions implementing MD5 hash, but these only return 16-byte raw hash as either RAW(16) or typecasted VARCHAR2(16), so you should use RAWTOHEX function on RAW(16) result to receive the same output as that of getMD5(). Also, MD5 functions are overloaded in a manner, which prevents them from being used in SQL. The DBMS_OBFUSCATION_TOOLKIT package does not expose any procedures or functions to produce more secure SHA hashes. getSHA() function closes this gap.

NN$PSP_OWA – OWA TOOLKIT INTERACTION API.

NN$PSP_OWA is the package responsible for interaction with Oracle Web Applications Toolkit packages (OWA). The package provides API for initializing the DPSP2 Kernel from OWA and passing DPSP2 call results back to OWA. OWA is the medium used by Oracle PL/SQL Gateway (mod_plsql) to communicate with the Oracle PL/SQL web application. If you have legacy web applications written using OWA, you can use NN$PSP_OWA to extend these applications with DPSP capabilities and easily mix old OWA and new DPSP code as your applications evolve.

DYNAMIC PSP KERNEL INITIALIZATION, CALL AND CONTENT CONTROL SUBPROGRAMS.

This group of subprograms is responsible for retrieving call environment from OWA and initializing the DPSP2 Kernel. You can call these functions from your legacy OWA code to initialize the DPSP2 Kernel and synchronize its environment with your legacy OWA code environment.

```
procedure initFromOWA;
```

This procedure copies current call environment from OWA and initializes the DPSP2 Kernel so that it can process requests in OWA call context. Use this procedure if you need to call DPSP2 APIs from your legacy OWA code (for example, to connect to the corresponding DPSP2 session and set or retrieve DPSP2 session variables from your legacy code.) This procedure does not pass any parameters you might have received from the URL to the DPSP2 Kernel and it does not restore an active Dynamic PSP session (see enterDPSPSession() below for routine that may be used to reestablish DPSP sessions from legacy OWA code.)

```
procedure initFromOWA(name_array   IN NN$TP.VARCHAR_ARRAY
                     ,value_array  IN NN$TP.VARCHAR_ARRAY);
```

This procedure is similar to initFromOWA(), but it also allows to pass call parameters to the DPSP2 Kernel. The procedure takes two arrays of values – array or parameter names and corresponding array of parameter values. You should ensure that both arrays have exactly the same number of elements, or the call will fail.

```
procedure go( name_array  IN NN$TP.VARCHAR_ARRAY := NN$TP.VARCHAR_ARRAY_empty
             ,value_array IN NN$TP.VARCHAR_ARRAY := NN$TP.VARCHAR_ARRAY_empty);
```

This procedure performs the same action as the GO hub procedure. It takes two arrays of values – array of parameter names and corresponding array of parameter values, identifies and decodes special arguments (see DPSP call format description for special arguments names and their purposes), initializes the DPSP2 Kernel and performs requested operation, and finally communicates the execution result back to OWA so that mod_plsql can pick it up and deliver to the client.

```
procedure flushToOWA;
```

This procedure flushes current DPSP2 Kernel response to the OWA. This procedure resets OWA state, copies DPSP response to OWA buffers and resets the DPSP response buffers. All content generated by previous calls to HTP/HTF packages is destroyed and is replaced with current DPSP response. **WARNING! THIS CALL WILL DESTROY YOUR CURRENT OWA CONTENT AND REPLACE IT WITH DPSP-GENERATED CONTENT!**

```
procedure appendContentToOWA( bResetResponse Boolean := TRUE );
```

This procedure appends DPSP-generated content to current OWA content. Procedure optionally resets DPSP response buffers if bResetResponse is True (default), so that subsequent calls will not append the same content again. This call will not copy any HTTP headers or cookies issued in the DPSP code, it will only copy document body. Unlike flushToOWA(), this procedure does not destroy current OWA content, but rather appends to it allowing you to mix legacy code output with DPSP output.

DYNAMIC PSP SESSION INTERACTION SUBPROGRAMS.

This group of subprograms allows your legacy code to connect to Dynamic PSP logical sessions and set or retrieve DPSP session variables from your legacy OWA code.

```
procedure enterDPSPSession;
```

This procedure allows your legacy OWA code to connect to an active DPSP session. The procedure automatically initializes the Kernel by calling `initFromOWA()` and retrieves the active session identifier. After connecting to the session, you can use `NN$PSP_RT` routines to access and set server-side session variables and perform other DPSP session-specific tasks.

```
procedure enterDPSPSession(name_array  IN  NN$TP.VARCHAR_ARRAY
                             default NN$TP.VARCHAR_ARRAY_empty
                          ,value_array IN  NN$TP.VARCHAR_ARRAY
                             default NN$TP.VARCHAR_ARRAY_empty);
```

This version of the procedure is similar to the previous version, but in addition allows passing call parameters to the Kernel. Internally, this function calls overloaded version of `initFromOWA(name_array, value_array)` to initialize the Kernel. You should ensure that `name_array` and `value_array` arguments contain the same number of elements, or the call will fail.

```
procedure enterDPSPSession(sSessID  IN  VARCHAR2);
```

This procedure allows your legacy OWA code to connect to an arbitrary DPSP session identified by `sSessID`. After connecting to the session, you can use `NN$PSP_RT` routines to access and set server-side session variables and perform other DPSP session-specific tasks. The function automatically initializes DPSP Kernel by calling `initFromOWA()`. `sSessID` can be retrieved from **`nn-psp-sessid`** cookie or with built-in `env('PSP_SESSION_ID')` call (this last alternative is only available for DPSP units.) Application developers may also pass the session identifier as parameter to their legacy code.

```
procedure enterDPSPSession(sSessID     IN  VARCHAR2
                          ,name_array  IN  NN$TP.VARCHAR_ARRAY
                             DEFAULT NN$TP.VARCHAR_ARRAY_empty
                          ,value_array IN  NN$TP.VARCHAR_ARRAY
                             DEFAULT NN$TP.VARCHAR_ARRAY_empty);
```

This version of the procedure is similar to the previous version, but also allows you to pass call parameters so that the Kernel can be initialized with them. Internally, this function calls overloaded version of `initFromOWA(name_array, value_array)` to initialize the Kernel. You should ensure that `name_array` and `value_array` arguments contain the same number of elements, or the call will fail.

OWA-GENERATED CONTENT IMPORT SUBPROGRAMS.

This group of subprograms provides a way to import the content generated by legacy code written for OWA into Dynamic PSP. They allow to either copy and embed the content generated by a call to legacy code into Dynamic PSP output stream, or to receive it into variable for further processing.

```
procedure getOWA( sHeaders OUT VARCHAR2, sContent OUT VARCHAR2);
```

This procedure retrieves HTTP headers and first 32K of the current OWA-generated content from OWA response buffer into `sHeaders` and `sContent` variables and destroys the OWA response buffer. Receiving variables MUST

have 32K capacity, otherwise an exception may be thrown if returned value does not fit into the receiving variable. If content buffered in the OWA response buffer is larger than 32K, the remainder is ignored and disposed.

```
procedure getOWA( asHeaders OUT NN$TP.VARCHAR_ARRAY,
                  asContent OUT NN$TP.VARCHAR_ARRAY);
```

This version is similar to the previous, but it returns HTTP headers and HTTP content as arrays. Each HTTP header is returned as separate array element.

```
procedure getOWA( sHeaders OUT VARCHAR2, clobContent OUT CLOB);
```

Yet another version of the `getOWA` procedure, which returns HTTP headers as string and HTTP content as temporary `CLOB` with `DBMS_LOB.SESSION` duration. The returned `CLOB` should be destroyed with a call to `DBMS_LOB.freeTemporary()` when no longer needed.

```
procedure copyOWA;
```

This procedure copies current OWA response into the Dynamic PSP response buffer and destroys the OWA response buffer. The procedure detects and separates HTTP headers in OWA response from HTTP content, and copies them into appropriate Dynamic PSP response buffer sections.

```
function getOWAContentString RETURN VARCHAR2;
```

This function retrieves up to 32K of the OWA response buffer and returns it as a string. The OWA response buffer is automatically destroyed. A call to this function will cause any HTTP headers in the OWA response buffer to be lost.

```
function getOWAContentArray RETURN NN$TP.VARCHAR_ARRAY;
```

This function retrieves OWA response buffer content and returns it as an array. The OWA response buffer is automatically destroyed. A call to this function will cause any HTTP headers in the OWA response buffer to be lost.

```
function getOWAContentClob RETURN CLOB;
```

This function retrieves OWA response buffer content and returns it as a temporary `CLOB` with `DBMS_LOB.SESSION` duration. The OWA response buffer is automatically destroyed. A call to this function will cause any HTTP headers in the OWA response buffer to be lost. The returned `CLOB` should be destroyed with a call to `DBMS_LOB.freeTemporary()` when no longer needed.

```
function getOWAHeadersString RETURN VARCHAR2;
```

This function retrieves HTTP headers from OWA response buffer and returns them as a string. The OWA response buffer is automatically destroyed. A call to this function will cause HTTP content in the OWA response buffer to be lost.

```
function getOWAHeadersArray RETURN NN$TP.VARCHAR_ARRAY;
```

This function retrieves HTTP headers from OWA response buffer and returns them as an array. The OWA response buffer is automatically destroyed. A call to this function will cause HTTP content in the OWA response buffer to be lost.

Examples:

```
<%
   my_owa_procedure(param1, param2); -- call to legacy OWA-based routine
   NN$PSP_OWA.copyOWA; -- copy and embed the result of the legacy routine call,
                       -- including HTTP headers
%>

<%! owaContent VARCHAR2(32767); %>
<%
   my_owa_procedure(param1, param2); -- call to legacy OWA-based routine
   owaContent := NN$PSP_OWA.getOWAContentString; -- retrieve OWA content into variable
%>
<%-- print the content, excluding HTTP headers --%>
<% = owaContent %>
```

SECURITY CONSIDERATIONS.

You should understand that Dynamic PSP sessions and their identifiers should not be used for authentication and user identification. DPSP sessions are transport-related tokens that allow the Kernel to maintain continuous logical connection state between discrete HTTP calls.

It is generally not necessary to protect the NN$PSP_OWA API from web access, because these subroutines alone will not provide the attacker with any means for retrieving server-side session data when invoked from the web. The API simply initializes the Dynamic PSP Kernel call environment from the OWA environment, or copies content from the OWA subsystem to the DPSP Kernel, and does not perform any specific actions that may grant the attacker any extra access rights. The **go** procedure published in this API is practically the same as the DPSP hub procedure, and as such does not provide any backdoors into the system as well.

IV CHAPTER IV. DPSP2 HOOKS AND CACHE.

This chapter describes the Dynamic PSP Kernel Hooks mechanism, and caching and cache control mechanisms.

DYNAMIC PSP KERNEL HOOKS.

The Dynamic PSP Kernel provides a mechanism for intercepting and pre-processing or post-processing certain Kernel events via so-called hooks. Hooks are PL/SQL routines that are called automatically by the Kernel when triggering event occurs. Best analogy for Kernel hook is a DML trigger, which is automatically called before or after some database DML statement. DPSP2 developers may use hooks to alter the Kernel environment before or after the hooked event, or perform any other action, like logging or access control.

For example, developers may want to save some context before executing a unit and restore it afterwards. Previously, context saves and restores had to be coded directly in each unit. With `BeforeExec` and `AfterExec` hooks, these operations can be implemented once for all units.

KERNEL HOOK TYPES.

Currently, 4 hook types are defined:

`BeforeMain` Triggered before main request unit is executed. Main request unit is the unit initially called to process the request.

`AfterMain` Triggered just after the main unit was executed but before returning request processing result.

`BeforeExec` Triggered before a unit is executed. Applies to main unit and any units called from it using exec() calls.

`AfterExec` Triggered just after a unit was executed but before returning control to the caller. Applies to main unit and any units called from it using exec() calls.

All hook routines are executed in the DPSP2 Kernel context, and thus should be visible to the Kernel. If a hook is implemented as a stored procedure, the Kernel owner user should be granted `EXECUTE` on this procedure, or the procedure should be created in the Kernel schema. The Kernel calls hook routines in the following order:

```
BeforeMain
 BeforeExec
  <main unit>
    BeforeExec
      <exec()>
    AfterExec
  </main unit>
 AfterExec
AfterMain
```

IMPLEMENTING KERNEL HOOKS.

A Kernel hook is a PL/SQL block, which implements some actions to be performed when triggering event occurs. All hooks are automatically wrapped into an anonymous PL/SQL code block defined as follows:

```
DECLARE
  ID_   NUMBER := :unit_id;
  DONE  NUMBER := 0;
BEGIN
 <Kernel hook code>
 :stop := DONE;
END;
```


The block defines two variables, `ID_`, which receives unit identifier for which the hook was invoked, and `DONE`, which is used to tell the Kernel to stop processing the request further if the hook code sets it to non-zero value. The code between `BEGIN` and `END` may perform any actions and call any procedures. Length of the hook code should not exceed 2000 bytes. Hook code may intercept and process exceptions, or may ignore them. The Kernel silently ignores all unhandled exceptions thrown in hook routines.

`DONE` flag is particularly helpful when your hook code determines that the unit for which the hook was invoked should not be executed. Setting this flag to a non-zero value will cause the Kernel to stop further processing of the request and return current content to the caller. This may come in handy, for example, when your hook checks some conditions before executing a unit, and only allows the Kernel to proceed with the call if the check was successful, otherwise the hook code may set `DONE` to a non-zero value to stop the processing of the request. In this case, the Kernel will return whatever content was generated before and during the hook code execution and will neither execute any further hook routines nor the unit itself.

REGISTERING KERNEL HOOKS.

Kernel hooks are registered through the DPSP2 Registry, under `NN$PSP/Kernel/Hooks` hive. Four keys are defined under this hive, one for each hook type. To register a hook, new string item should be created under corresponding hook type key, and hook code should be placed in this item. Hooks may receive arbitrary names; the only requirement is that names should be unique under each key. If several different hooks are registered for one hook type, they are invoked sequentially, in no particular order. Developers should not rely on any particular order of execution of hook routines.

KERNEL HOOK EXAMPLE.

Assume that you want to implement a global access control and authentication mechanism for your units. You could implement it by calling an authentication check routine in each unit right at the beginning, or you can use `BeforeMain` hook to do the same. First of all, you create a login page where you will check for user authentication. Let's use standard HTTP basic authorization capabilities for this:

```
<%!
   sAuth     varchar2(200) := substr(env('AUTHORIZATION'),1,200);
   sUsername varchar2(100);
   sPassword varchar2(100);
   i         integer;
   sCurrent  varchar2(100);
%>
<% -- decode AUTHORIZATION header
   if (sAuth is NULL) then
    -- some versions of mod_plsql pass in HTTP_AUTHORIZATION instead
    sAuth := substr(env('HTTP_AUTHORIZATION'),1,200);
   end if;
   if (sAuth is not NULL) then
    i := instr(sAuth, 'Basic ');
    if (i > 0) then
      sAuth := NN$PSP_UTL.decodeBase64(substr(sAuth, i+6));
      i := instr(sAuth, ':');
      if (i > 0) then
        sUsername := substr(sAuth, 1, (i-1));
        sPassword := substr(sAuth, (i+1));
      end if;
    end if;
   end if;
   begin
     sCurrent := NN$PSP_RT.getSessionString('CURRENT_USER');
     if (sCurrent is NULL) then
      -- verify user credentials
      select username, password into sUsername, sPassword
```



```
        from my_user_table
       where username = UPPER(sUsername) and password = sPassword;
      -- and save authenticated user name for the session
      NN$PSP_RT.setSessionString('CURRENT_USER',sUsername);
     else
       -- verify user credentials, additionally ensuring that this is the same
       -- user who was originally logged in
       select username, password into sUsername, sPassword
        from my_user_table
       where username = UPPER(sUsername) and password = sPassword
             and username = sCurrent;
     end if;
     -- everything passed ok, can silently return
     return;
   exception
     when no_data_found then
      -- reset session variable for authenticated user
      NN$PSP_RT.setSessionString('CURRENT_USER',NULL);
      -- output authorization request page with 401 status code
      setHeader('Status','401 Authorization required');
      setHeader('WWW-Authenticate','Basic realm="My protected site"');
%><html>
<head><title>Authorization required</title></head>
<body><H3>Authorization required to access this resource</H3></body>
</html>
<% end; -- exception handling block %>
```

The above unit will check if HTTP `Authorization` header is present, decode it, and check if passed in credentials are valid. If the check passes, `CURRENT_USER` session variable receives logged in user name, otherwise it is cleared and authentication request page is returned. Assume we saved the unit as `AuthHook`, and set its `AUTH_ID` attribute to 'CURRENT_USER' so that it executes in context of Oracle session user. Now we will call this unit from `BeforeMain` hook as follows:

```
NN$PSP_RT.exec('AuthHook',NULL, sys_context('USERENV', 'SESSION_USER'));
if (NN$PSP_RT.getSessionString('CURRENT_USER') is NULL) then
  -- authentication failed
  done := 1;
end if;
```

The code above will execute our unit in context of Oracle session user and verify if `CURRENT_USER` session variable is set. If it is not, the code assumes that the authentication failed and stops further processing since the unit already printed an authorization request page. To register the above code as the Kernel hook, open `NN$PSP/Kernel/Hooks/BeforeMain` key and create new string item in it, say `AuthCheck`. Save the code as this item value. From now on, the above code will be automatically executed before any main unit is processed.

Note that since all kernel hooks are executed in the Kernel context, they are global by default, that is, they will affect all your DPSP-enabled schemas. If you want to restrict the hook to some particular schema, the following check may be added in the beginning of the hook code:

```
if (sys_context('USERENV', 'SESSION_USER') = 'MYSCHEMA') then
  <my hook code>
end if;
```

The above code will check if the Oracle session was initiated by `MYSCHEMA` owner and will exit for any other user. Also note that since we are executing the unit in context of Oracle session user and the unit depends on MY_USER_TABLE table availability in that schema, unit will not execute against any schema that does not have such table or view, or its layout is different from expected, and Kernel will silently ignore the failure. Still, adding relevant schema verification early in the chain will reduce the amount of work to be done by the Kernel, otherwise the Kernel will have to parse

and prepare the unit for execution and attempt to execute it in some Oracle user's context only to find out that it is not valid in that context.

DYNAMIC PSP 2 CACHING AND CACHE CONTROL.

Dynamic PSP 2 supports client-side and server-side caching to improve applications performance and reduce network traffic and server loads. Caching may be configured on per-unit basis, that is, each unit may have its own caching settings. DPSP2 developers may specify if they want to cache output of some units, where they want it to be cached, and when cached content should be refreshed, via unit attributes and cache control API.

Client-side caching relies on the use of `E-Tag`, `Last-Modified` and `If-Modified-Since` HTTP headers. When a unit is to be cached on the client, DPSP Kernel automatically sets `E-Tag` and `Last-Modified` headers for the unit content when response is sent to the client. When the client re-requests the unit, it sets `If-Modified-Since` and `E-Tag` headers to notify the server that it only wants to receive new content if it was modified since the timestamp specified in the `If-Modified-Since` header. The server then verifies if the content was changed since specified timestamp, and either responds with short message including `304 Not Modified` status code, or re-executes the unit and returns full content with new `Last-Modified` timestamp. If answered with `304` code, the client simply retrieves locally cached copy of the content and displays it, otherwise it retrieves new content, stores it in local cache and displays it. Client-side caching helps to reduce both network traffic and server load. The HTTP client should be able to cache content locally, and should support requests with timestamps.

Server-side caching does not rely on client-side cache and keeps track of cached content internally. For cached units, server stores unit-generated content in an internal table and when it detects that the content was not modified it quickly retrieves the cached copy and returns it to client instead of re-executing the unit, which may take substantially longer, especially if the unit executes some complex queries to generate the content. Server-side caching helps to reduce server load, but it does not reduce network traffic between the client and the server. The HTTP client should not be aware that any caching is taking place.

Currently, only top-level units (units that are invoked initially to service a request) may be cached. Content generated by all embedded unit calls is cached along with the main unit content as single entity. Units for which caching is enabled must be *deterministic*, that is they must produce exactly the same content when exactly the same parameters are supplied to the unit, and they must not depend on volatile data, like system time. In particular, this means that all units called from cached unit must also be deterministic and are called with the same parameters for the same main unit parameter set. Cached units may depend on non-volatile data stored in the database though, including any aggregates derived from that data. In this case, developers must implement state change flags covering such data and set `CHANGE_FLAGS` unit attribute for the cached unit to include all flags covering all data the unit may depend on to allow caching subsystem to detect data changes and refresh unit content when appropriate.

To prevent server-side cache from uncontrolled growth, Purger job cleans the cache periodically deleting all cached content older than configurable amount of time. `NN$PSP/Kernel/Purger/CACHE_DAYS` registry item controls for how long the cached content may stay in the cache.

CACHE REFRESH CONTROL – STATE CHANGE FLAGS AND CHANGE_FLAGS UNIT ATTRIBUTE.

Dynamic PSP 2 provides a mechanism for controlling cache content refreshes. Developers may define certain named flags, called *state change flags*, and then indicate that certain unit depends on any combination of these flags and should be refreshed whenever the timestamp of any of the state change flags is altered. The flag state is changed through the `NN$PSP_CACHE.markFlag()` call, which may be performed, for example, in an after statement trigger on a table the unit depends on. When data in the table is changed and the trigger is fired, the trigger invokes `markFlag()` and updates the timestamp of certain flags. When the transaction commits, new flag timestamps are committed as well, and the caching subsystem will re-execute units that depend on flagged data the next time they are invoked and cache the updated content. State change flags affect both client-side and server-side cache.

Flags the unit depends on are specified in the unit `CHANGE_FLAGS` attribute as a list of comma-separated case-insensitive flag names. It is up to developers to define flags, assign them to units and update them using `markFlag()` call when the flagged data is changed. In no event should the flag timestamp be changed from the depending unit itself, or its content will never be refreshed, as it will never be called again (chicken and egg problem.)

CACHE CONTROL API – NN$PSP_CACHE.

Dynamic PSP 2 provides an API for controlling caching via NN$PSP_CACHE package. Summary of the package subprograms intended for DPSP2 developers' use is presented below.

```
procedure markFlag ( sFlagName   in  varchar2
                   , dTimeStamp  in  date   := sysdate
                   );
```

markFlag procedure should be used to update timestamps of the state change flags. sFlagName is case-insensitive state change flag name, and dTimeStamp is the new flag timestamp, which defaults to current system time. Caching subsystem verifies if a cached unit needs refresh by comparing cached content timestamp with all stage change flags timestamps the unit depends on, and re-executes the unit if any of the flags are more recent than the cached content. If a flag identified by sFlagName does not exist, it is automatically created. If a non-existent flag is specified in unit's CHANGE_FLAGS attribute, it will also be created automatically and its timestamp will be set to system time when the unit is first executed.

The procedure does not commit and may be used in triggers – changes to the flag will only be committed with the rest of transaction, or will be rolled back if the transaction rolls back.

For example, unit #100 depends on data in PERSONS table. Developer defines Persons flag and an after statement trigger on PERSONS that will update the flag:

```
CREATE OR REPLACE TRIGGER TRG$S_AIUD$PERSONS
AFTER INSERT OR UPDATE OR DELETE ON PERSONS
BEGIN
 NN$PSP_CACHE.markFlag('Persons');
END;
/
```

Developer then sets CHANGE_FLAGS for unit #100 to 'Persons' and CACHE_MODE to 3 (both client-side and server-side caching.) The Dynamic PSP 2 Kernel will now verify the flag timestamp and compare it to the cached content timestamp, and will re-execute the unit when the flag timestamp becomes more recent than the cached content timestamp, which will happen whenever the data in PERSONS table changes thanks to the after statement trigger we defined on that table.

V CHAPTER V. DPSP2 UNIT COMMANDER.

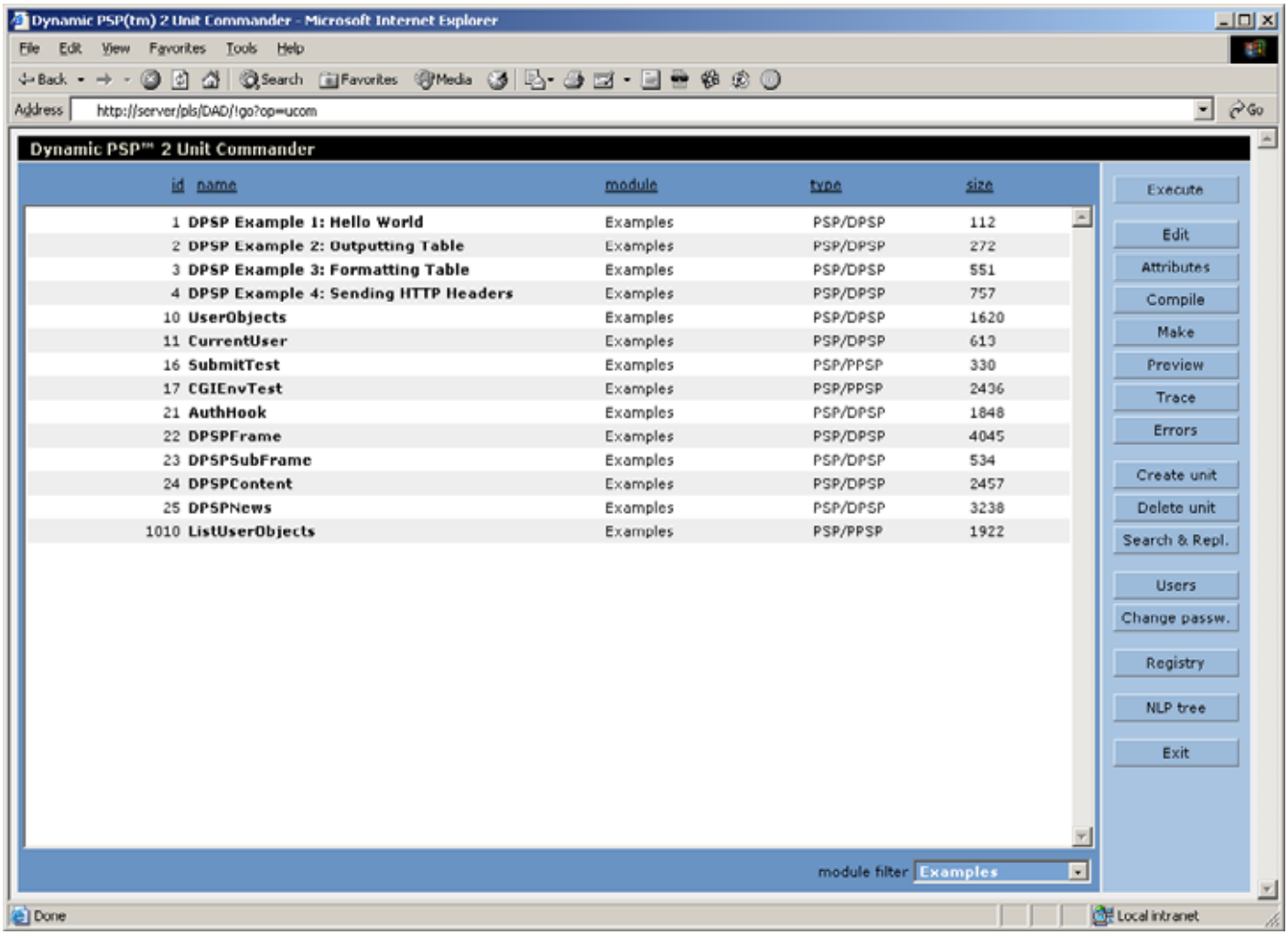
This chapter describes the new Unit Commander Development Interface introduced in Dynamic PSP Version 2. The Unit Commander is Web-based Development Interface, which allows you to manage your DPSP2 projects, create, view, edit and compile DPSP2 units, manage unit attributes, manage internal user accounts and security settings, and more.

INVOKING THE UNIT COMMANDER.

The Unit Commander is invoked through DPSP2 Project DAD (Database Access Descriptor) by specifying `op=ucom` parameter to the `GO` hub procedure:

`http://server/pls/DAD/!go?op=ucom`

The authorization dialog will appear and you will need to login into the DPSP2 Development Environment. When DPSP2 is initially installed, one default login with administrative privileges is automatically created: user "`dpsp`" with password "`dpsp`" (user names and passwords in Dynamic PSP are **case-sensitive**). Since this account provides administrative access to the system, **we recommend changing the password for this account immediately** after the Kernel is installed. After you provide correct credentials, Unit Commander screen will be displayed:



BUILT-IN ACCESS CONTROL SYSTEM.

Dynamic PSP2 features built-in access control system, which is used to control access to the Development Interface functions, as well as to DPSP2 units themselves. All DPSP2 user accounts are managed through Unit Commander. Each account is assigned a set of privileges, which control what particular user is allowed to do in the system. Each unit is also assigned a set of privileges, which identify the set of permissions required to access the unit. When particular DPSP2 user attempts to execute a unit, the Kernel checks if unit's required permissions list is not empty, and checks if user has permissions required by unit's permission mask before granting access (executing) the unit. Empty permission mask for particular unit allows anonymous access to the unit, that is, anyone can execute it. Since the Development Interface itself is built of several DPSP2 units, this mechanism provides for effective access control to them.

Note, that although the built-in access control system might seem useful for controlling access to the system by regular website users, it is not very well suited for this purpose: you cannot have separate user database for each DPSP2 project, there is no self-register capability that can be used by site visitors to register themselves, privileges set is fixed and cannot be changed, etc. Implementing such control system for your particular projects is up to you. Built-in access control system is only used for controlling access to the DPSP2 Development Interface, and we do not recommend using it for other purposes.

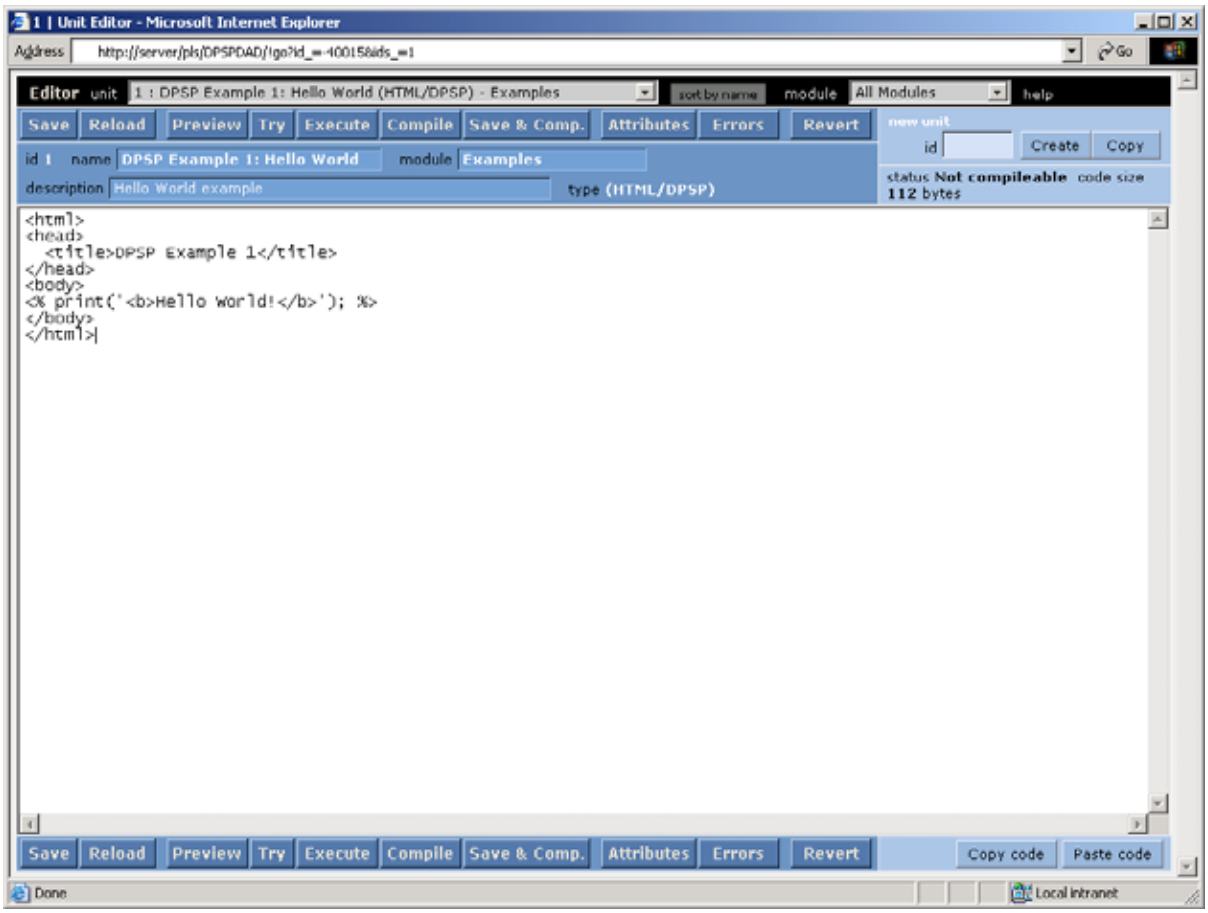
The Unit Commander provides an interface for managing built-in access control system, user accounts and their privileges, which will be described in detail later in this chapter.

UNIT COMMANDER FUNCTIONS AND CONTROLS.

The Unit Commander Development Interface is split into several areas. Main area lists DPSP2 units in current Project schema you are connected to (depending on the DAD you access the Unit Commander through), optionally filtered by logical module. Toolbar at the right side of the screen provides access to unit editor, execution and debugging facilities, user management interface and DPSP Registry. When you click on unit row in unit list, this row is highlighted and toolbar buttons are set to operate with selected unit. Unit manipulation buttons include **Execute**, **Edit**, **Attributes**, **Compile**, **Make**, **Preview**, **Errors**, and **Trace**. Access to all unit manipulation functions is controlled by built-in access control mechanism. To perform an action on a unit, select the unit in the unit list by clicking on it. The row will be highlighted, and toolbar buttons will now perform actions on the selected unit.

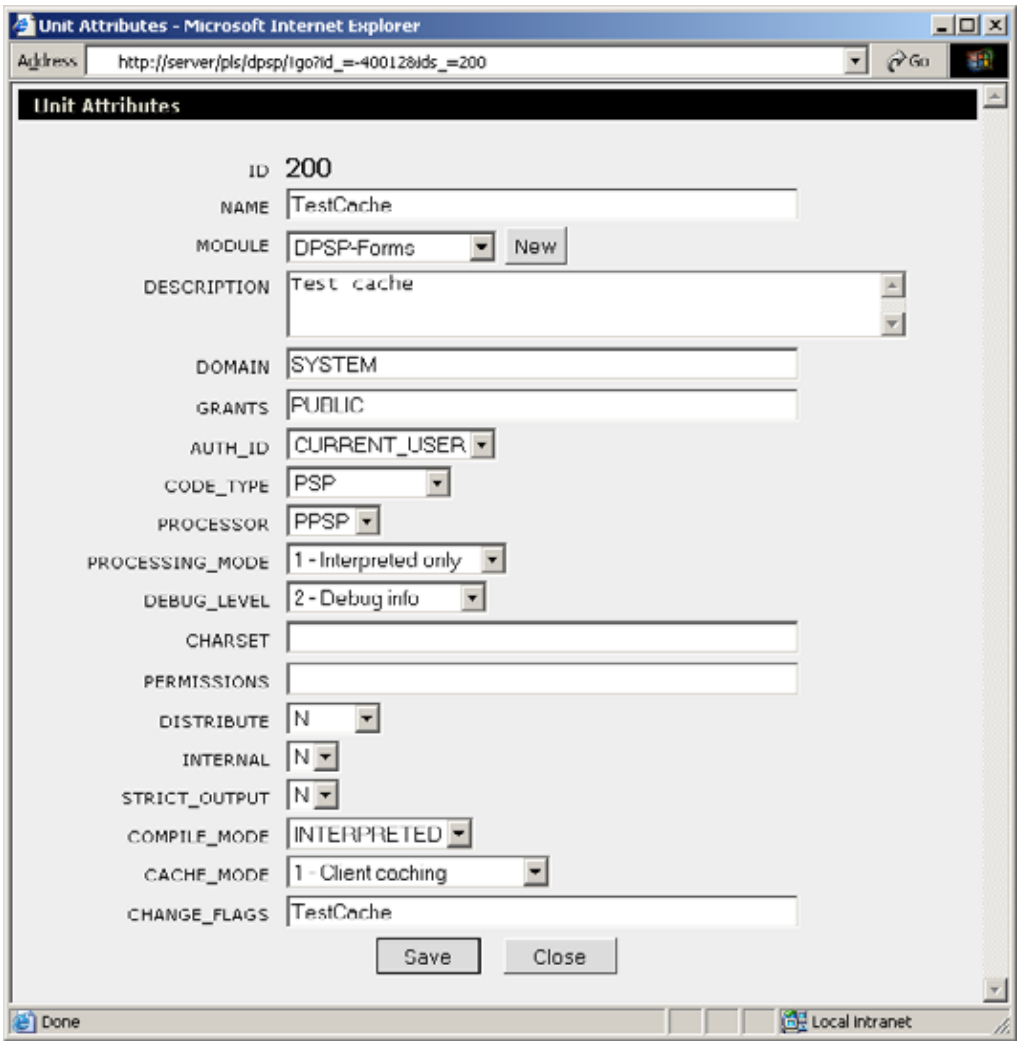
Execute button executes currently selected unit in new window.

Editor button opens new editor window and loads selected unit into editor:



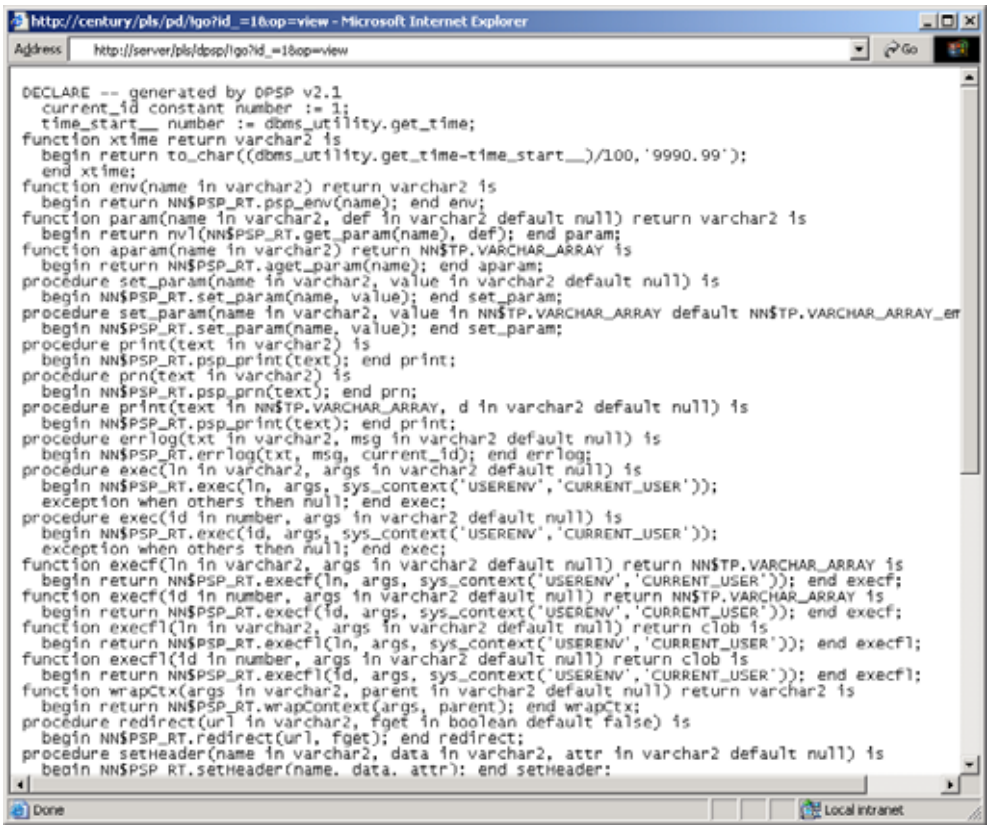
The Editor window provides tools to edit, save, compile and execute units, select different unit for editing, execute current unit, create new unit or copy current unit to new one. This functionality will be discussed in more detail later in this chapter.

Attributes button opens Unit Attributes editor window:



Unit Attributes editor allows to set all unit attributes for selected unit. Unit attributes, their meanings and possible values will be discussed in more detail later in this chapter.

Preview button opens a window with source code of processed unit:

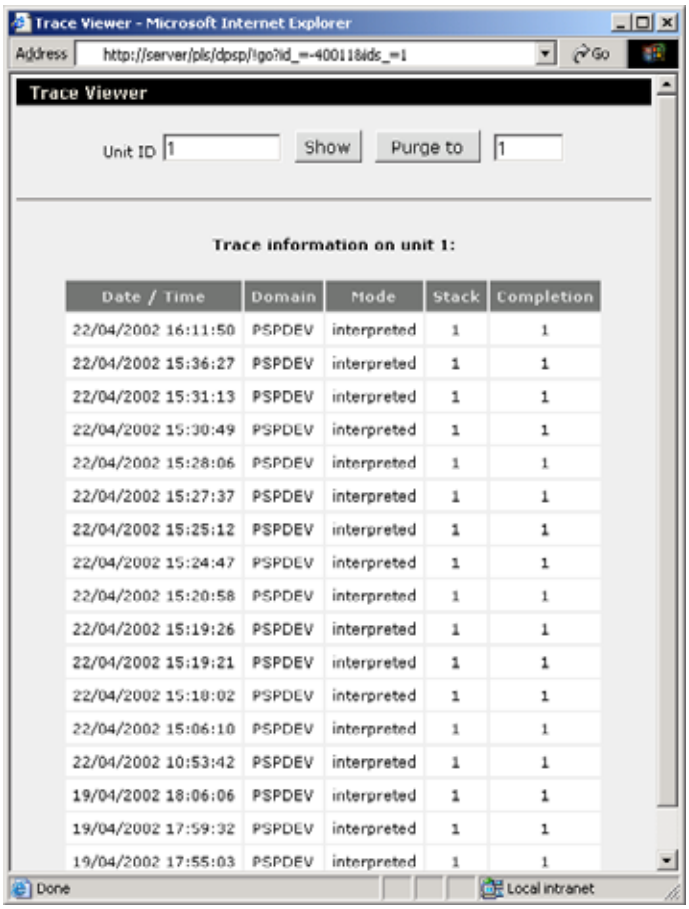


This processed PL/SQL source code is what will be actually executed when the unit execution request is received. Note that this view presents current unit code. If there is also compiled version of the code, that compiled version may be executed instead depending on `PROCESSING_MODE` unit attribute value.

Compile button starts unit compilation process. Compiler feedback window is displayed and compilation status is reported in this window. If unit compilation is unsuccessful due to syntax errors in source, unit source code with highlighted line that contains error will be displayed. When unit is compiled, chosen preprocessor parses the source code and translates it to PL/SQL package, which is then compiled by Oracle PL/SQL compiler.

Make button causes recompilation of all units in current project, which were compiled at least once and their source code was altered after the last compilation. Thus, this button will not compile any units that can be compiled but never were. It will only attempt to synchronize compiled versions of units with current unit source code.

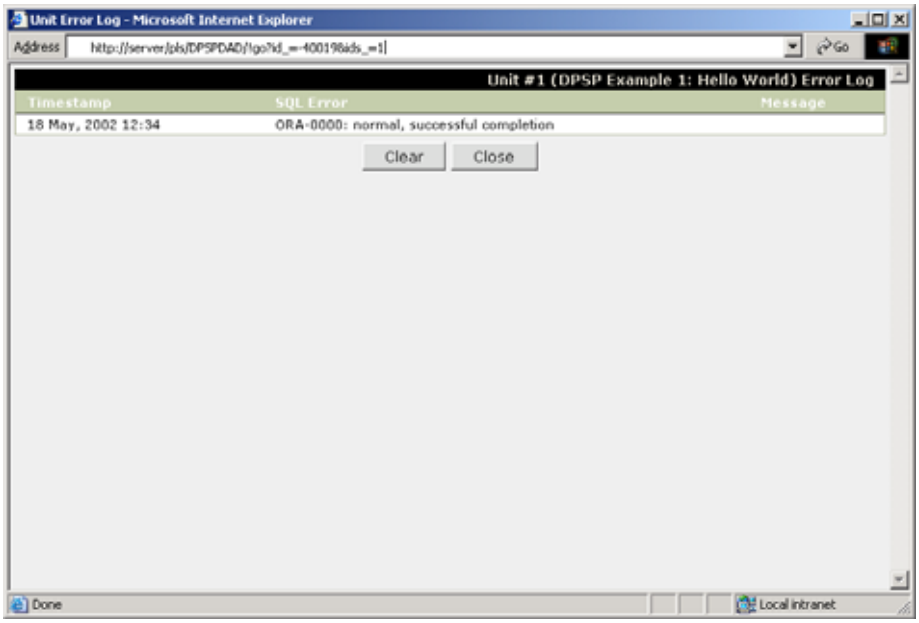
Trace button opens unit execution tracer window:



Tracer displays the log of all executions of the unit, with the following information for each call:

Time stamp of the execution, domain in which the unit was executed (Oracle schema name), mode of execution (interpreted or compiled), level on call stack (1 = top of stack, this was main unit for the call), and completion status (1 = successful, -1 = unsuccessful).

Errors button opens unit runtime error log:



In this log, all runtime errors and messages logged using the `errlog()` procedure are displayed. **Clear** button clears the error log, and **Close** button closes the window.

Two unit lifecycle buttons are grouped together below unit manipulation buttons. **Create unit** and **Delete unit** buttons allow to create new blank unit or delete currently selected unit respectively. **WARNING: deleting a unit is an IRREVERSIBLE operation.** If you delete a unit, you will not be able to revert this operation as the unit itself and all of its backup copies will be deleted from the system. Use this button with extreme caution.

Search & Repl. (Search and Replace) Button opens the Search and Replace facility window. Here you can perform system-wide or module-wide *case-sensitive* searches and selectively replace found occurrences with new values.

Users button brings in internal user management interface. This interface is described in detail later in this chapter.

Change passw. (Change password) button allows you to change your password for Development Environment. All other edits to your account are performed through user management interface and may require administrative privileges.

Unit manager button attempts to launch Java-based Unit Manager if it is installed. This Unit Manager is an optional replacement for the web-based Development Environment and requires direct connection through JDBC to the Oracle database where the Dynamic PSP Kernel resides.

Registry button opens simple web-based DPSP2 Registry browser and editor.

As other DPSP2 modules are installed and registered, additional buttons may appear in the Unit Commander toolbar (for example, screenshot of the Unit Commander in the beginning of this chapter shows **NLP Tree** button, which was added by the NLP subsystem). Functionality of these additional buttons is described in documentation for these additional modules.

UNIT ATTRIBUTES.

Each unit has a set of attributes assigned to it. These attributes determine how the unit is executed, published, edited, etc. To edit unit attributes, you use Unit Attributes Editor. Below is the list of unit attributes, their meaning and possible values:

`ID` — This is the unique numeric unit identifier. This attribute is assigned when unit is created, and cannot be changed afterwards.

`NAME` — This is Unit Logical Name. This attribute can be used for unit identification instead of Unit ID, but in this case it must be unique across the project, otherwise the resolver will be unable to resolve the name to Unit ID. Uniqueness of this attribute is not enforced by the Kernel.

`MODULE` — Logical Module the unit belongs to. This attribute may be used to logically group units. Its value has no effect on unit execution or scope. It is only used in unit list filters.

`DESCRIPTION` — Human-readable description of the unit. This attribute is informational.

`DOMAIN` — Domain attribute defines the scope of visibility for the unit. This attribute is very important, and can take the following values:

- `SYSTEM` — This unit belongs to Kernel. Do not set this value for your custom units.
- `<PROJECT SCHEMA>` — This unit belongs to specific project. `<PROJECT SCHEMA>` should be the name of the Oracle schema (user), which owns the project data.

`GRANTS` — This attribute defines comma-separated list of projects (Oracle users), which are allowed to execute the unit besides the project to which the unit belongs. `PUBLIC` value allows access to the unit for all projects on the instance.

`AUTH_ID` — This attribute defines the rights, with which the unit is compiled and/or executed. Can take the following values:

- `CURRENT_USER` — Unit is executed with privileges of the caller.
- `DEFINER` — Unit is executed with privileges of its creator (owner).

This attribute affects the unit in both interpreted and compiled mode. In interpreted mode, the Kernel will automatically pass the execution request to correct agent depending on this attribute, and the value of this attribute will be specified in package header when unit is compiled. Oracle will then automatically control the rights, with which the unit is executed, depending on this attribute.

DPSP Kernel's default for this attribute is `CURRENT_USER`.

`CODE_TYPE` — This attribute defines the code type of the unit. Can take the following values:

- `HTML` — Unit is plain HTML
- `CSS` — Unit is Cascading Style Sheet
- `JAVASCRIPT` — Unit is JavaScript
- `XML` — Unit is XML (□review□le Markup Language) document
- `XSL` — Unit is XSL (□review□le Style Language) stylesheet
- `PSP` — Unit is PSP code
- `PLAIN` — Unit is plain text
- `WML` — Unit is WML (Wireless Markup Language) document

This attribute affects the MIME type returned for the page produced by the unit. For `CSS` the

MIME type is 'text/css', for XML and XSL the MIME type is 'text/xml', for JAVASCRIPT the MIME type is 'text/javascript', for PLAIN the MIME type is 'text/plain', for WML the MIME type is 'text/vnd.wap.wml', and for all other types it is 'text/html'.

PROCESSOR — This attribute defines which processor will be used for processing unit code. Possible values:

FLAT	Unit will be processed with FLAT processor (output as is)
DPSP	Unit will be processed with Dynamic PSP processor (basic DPSP syntax supported)
PPSP	Unit will be processed with Procedural PSP processor (Procedural PSP extensions supported)
OPSP	Unit will be processed with Objective PSP processor (Objective PSP syntax supported)

PROCESSING_MODE — This attribute defines possible processing mode for the unit. Possible values:

0 - No processing	Unit will not be processed (all calls to the unit disabled, regardless its security settings and visibility.)
1 – Interpreted only	Unit must be executed dynamically, no compilation possible.
2 – Compiled only	Unit must be compiled, and only valid actual compiled version will be executed, otherwise the unit will not be executed at all.
3 - Can be compiled	Unit can be compiled, if no compiled version found or it is invalid, unit will be executed dynamically ignoring compiled version; otherwise compiled version is executed.

DEBUG_LEVEL — Defines debug information output level for the unit. This attribute controls the amount of information returned by preprocessors on unsuccessful unit execution/compilation. Level of 0 disables debug output; level 3 produces the most verbose debugging output, including highlighted source code for the unit; levels 1 and 2 are intermediate levels. Level 0 also disables output produced using DEBUG() built-in procedure. We recommend setting this attribute to 0 for production units. This attribute may be overridden at runtime by specifying debug=<level> parameter to GO hub procedure, which will force <level> debug level for the unit and all units it will call.

When DEBUG_LEVEL is > 0, execution trace is printed in HTML comments at the end of resulting HTML page. Trace includes statistics for the unit itself and all other units it called during request servicing. Currently this feature is only available when CODE_TYPE for the unit is HTML, no trace is printed for other document types.

CHARSET — Defines IANA character set returned as part of HTTP response for the unit (do not mix with Oracle character sets). Can take the following values:

DEFAULT	Browser-default character set will be applied; the Kernel will not return any specific character set.
<IANA character set name>	Specified character set will be set for the document. Character set should be IANA-registered character set, not Oracle character set.

Page developer is responsible for providing character data in declared character set, the Dynamic PSP Kernel will not perform any conversion from database character set to declared character set if they differ.

PERMISSIONS — List of permission codes, which control the right to execute the unit. Each code is one letter, and final list is a string of concatenated codes. If this list is not empty, the Kernel will only execute the unit for users, which have at least the same set of permissions. Valid permission

codes are:

- X eXecute permission, allows to execute units
- V preView permission, allows to preview unit source code after processing
- E Edit permission, allows to edit units
- D Delete permission, allows to delete units
- C Create permission, allows to create units
- A Admin permission, allows to administer system

For example, if a unit has permissions mask 'XVE', only users granted at least X, V, and E permissions will be able to execute the unit. This allows restricting access to units by their function and permissions mask. For example, editor units have their permission mask set to 'E', which means that only users that have Edit permission can execute the editor.

You do not need to set this attribute for each and every unit. Set this attribute only if you want to protect the unit, otherwise leave it blank.

DISTRIBUTE This is internal flag, and should be set to N for all custom units.

INTERNAL This flag affects visibility of the unit. Can have the following values:

- N Unit can be accessed (executed) from the Web.
- Y Unit is internal and can only be executed using built-in execXX() procedures.

STRICT_OUTPUT This flag affects the way characters between consecutive PSP code blocks are treated when processing source code. Can have the following values:

- N Non-essential characters are removed from the output. Thus, if two PSP code blocks are separated by a space character, this space character is omitted in output and blocks are combined. This is the default.
- Y All characters between PSP blocks are retained in the output.

COMPILE_MODE Defines unit compilation mode. Can have the following values:

NATIVE	Unit is natively compiled
INTERPRETED	Unit is compiled for interpreted execution (default)

This attribute affects unit compilation on Oracle9*i* and later only, it does not have any effect on Oracle8*i* as it does not support native PL/SQL compilation, and should be left at its default value on 8*i*. For more information on native PL/SQL compilation, please refer to Oracle 9*i* documentation.

CACHE_MODE Defines unit output caching mode. Can have the following values:

0 - No Caching	No caching. Unit will always be executed to produce current content.
1 - Client Caching	Client-side caching. Client is responsible for caching the content generated by the unit and informing the server which version of the content it has. Server will either return 'Not modified' response, or re-execute the unit if it detects that client-side cached copy is outdated.
2 – Server Caching	Server-side caching. Server will cache content internally and will serve cached content if it was not changed according to its internal flags instead of executing the unit again.
3 – Client + Server Caching	Both client-side and server-side caching (note that client-side caching will come into play before server-side, thus if a unit-generated content is already cached on the client, request will not be served from server-side cache if server detects that the client copy is recent.)

For details on Dynamic PSP 2 cache mechanisms, please see Chapter IV.

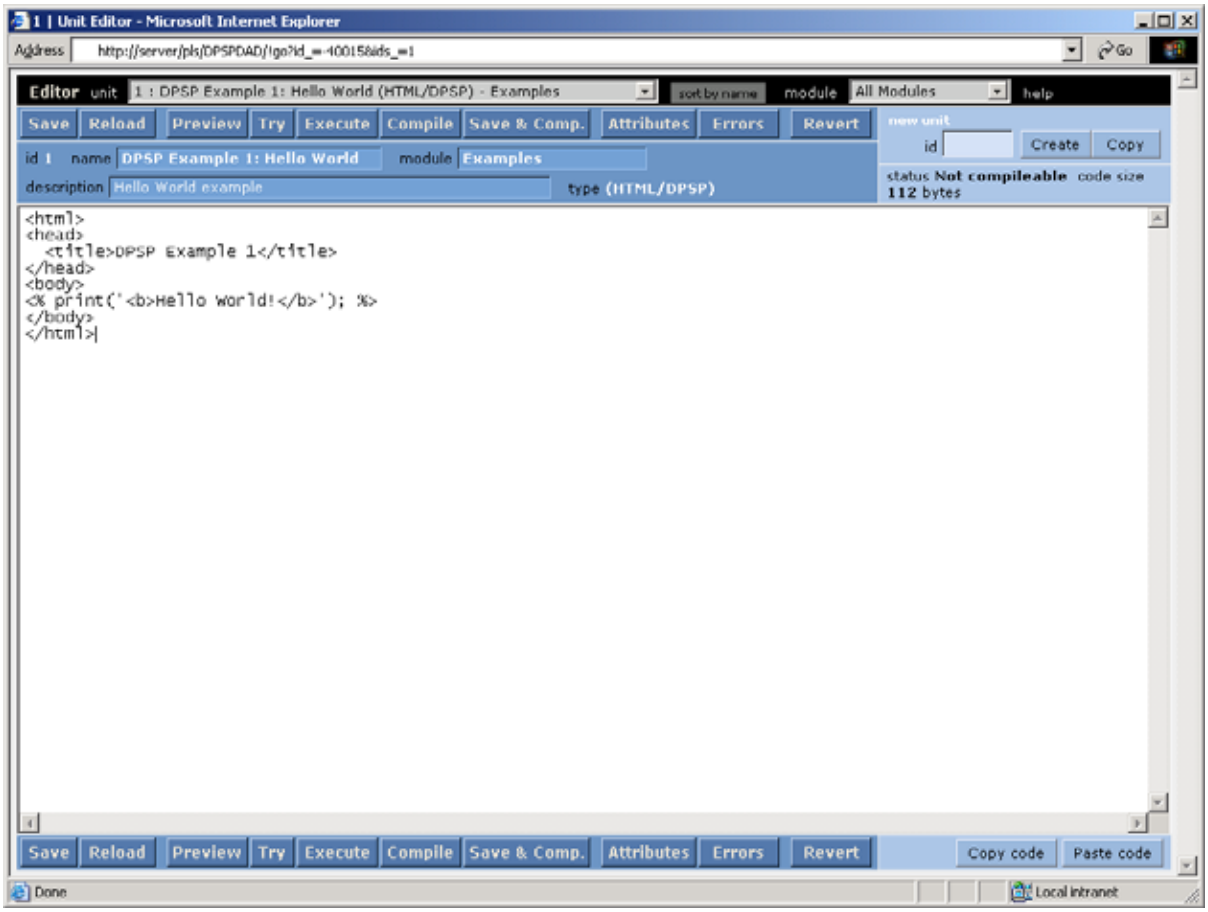
CHANGE_FLAGS — Defines stage change flags the unit depends on. Receives comma-separated list of state change flags, which affect unit content and may cause it to be re-executed to refresh cache. For details on state change flags, please see Chapter IV.

FRAME_PAGE — Defines the unit to be automatically invoked before processing current unit if it is called directly (not through `exec()` call.) The outer unit may use `INNER_ID` environment variable to identify the unit for which it is invoked, and embed the inner unit into its own output where appropriate with a call to **`exec(to_number(env('INNER_ID')))`**. This automatic outer unit invocation may be used, for example, to include common scripts and styles or common content, like page header, footer or sidebar, automatically, without the need to include them manually into each affected page. One advantage of this approach is that you can easily change outer content without the need to revisit each inner unit to fix links or common code.

When a unit with non-`NULL` `FRAME_PAGE` is called directly (through the `GO` hub procedure), the unit identified by the `FRAME_PAGE` is invoked first. This unit may generate some common content and then it should allow the inner unit to proceed by executing it with **`exec(to_number(env('INNER_ID')))`** call (may be more than once).

Frame pages may be chained (that is, an outer unit serving as frame page for some inner unit may in turn have its own outer unit) allowing for complex call layering and content embedding.

UNIT EDITOR.



The Unit Editor is where you edit your DPSP units. It is activated by highlighting a unit in Unit Commander and clicking on **Editor** button. The Unit Editor provides the following controls:

Unit selector dropdown, unit selector sort order selector button, module filter dropdown, top and bottom button bars, name, module and description attribute editors, new unit creation controls, main editor area, and clipboard control buttons. These controls are described in detail below.

UNIT SELECTOR DROPDOWN.

This dropdown allows to switch between edited units quickly. This dropdown lists units that satisfy module filter condition, sorted by ID or name. Selecting different unit from the dropdown will switch editor to the selected unit. If current unit was modified, you will be prompted if you are sure that you want discard all changes and switch to another unit.

UNIT SELECTOR SORT BUTTON.

This button will change sort order in unit selector dropdown to ID or name. Units are sorted by ID by default, but you may want to sort them by their logical names. Clicking on the sort order selector button will change sort between these two criteria.

MODULE FILTER DROPDOWN.

This dropdown allows you to filter unit selector dropdown and only display units for a selected module. Value of 'All Modules' disables the filter.

BUTTON BARS.

Button bars are located at the top and at the bottom of editor screen. Both button bars provide the same set of controls:

Save button saves current unit code into unit code repository.

Reload button refreshes the unit code from database, discarding all changes made to the code. WARNING: this button will cause all unsaved changes to the code to be lost!

Preview button brings up source code preview window.

Try button attempts to execute current version of the unit dynamically ignoring any compiled version if it exists. Use this button to test unit changes before compiling the unit.

Execute button attempts to execute currently saved copy of the code (please note, that this button DOES NOT SAVE the code in the editor, it attempts to execute last saved copy in the database). If there is valid compiled version of the unit, this compiled version is executed; otherwise current saved copy is executed dynamically.

Compile button attempts to compile currently saved copy of the code (please note, that this button DOES NOT SAVE the code in the editor, it attempts to compile last saved copy in the database). Compilation status window will pop up displaying the compilation progress. If compilation was successful, the compilation status window will automatically close 3 seconds later, otherwise it will persist to allow you to assess compilation errors. When unit is compiled, chosen preprocessor parses the source code and translates it to PL/SQL package, which is then compiled by Oracle PL/SQL compiler.

Save and Compile button saves current unit and attempts to compile it. Compilation status window will pop up displaying the compilation progress. If compilation was successful, the compilation status window will automatically close 3 seconds later, otherwise it will persist to allow you to assess compilation errors.

Attributes button brings up the Unit Attribute Editor window.

Errors button brings up the Unit Error Log window.

Revert button discards current version of the unit and restores previous version from internal archive. Each time you save the code, previous version is archived in the internal archive table and this button allows restoring last previous backup copy.

NAME, MODULE AND DESCRIPTION ATTRIBUTE EDITORS.

These editors allow you to change Name, Module and Description attributes right from the Editor window, without the need to call the Unit Attribute Editor.

NEW UNIT CREATION CONTROLS.

These controls allow you to create new unit, either empty, or by copying current unit to the new one. Specify the ID of the new unit and then click either **Create** or **Copy** button, to create empty unit or copy current unit into new unit respectively.

MAIN EDITOR AREA.

This is where you edit your code. All major text editor functions are supported, including cutting and pasting, searching, etc. Full functionality of the editor depends on your browser capabilities. We recommend using Microsoft Internet Explorer 5.5 or later for the most feature-rich in-place editor.

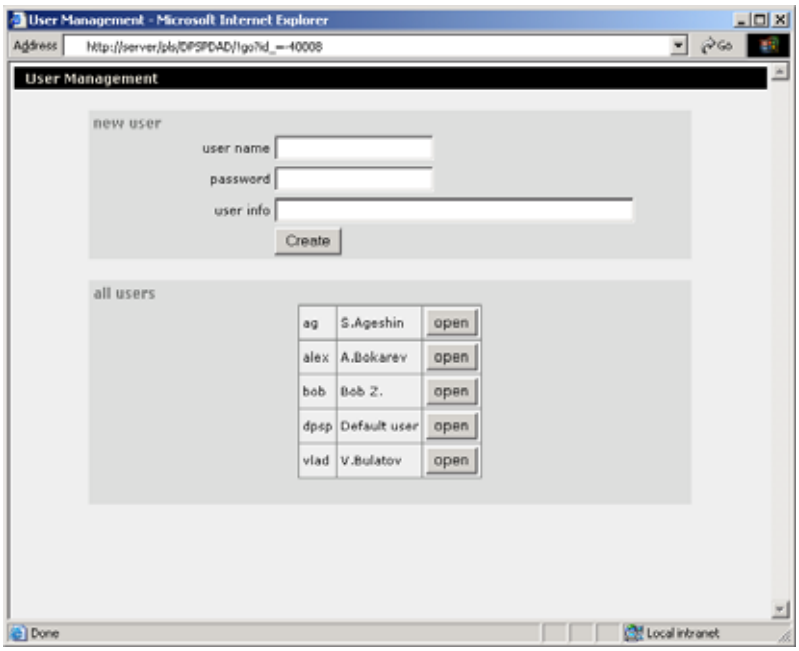
Maximum size of text the main editor area can hold is around 300 kilobytes.

CLIPBOARD CONTROLS.

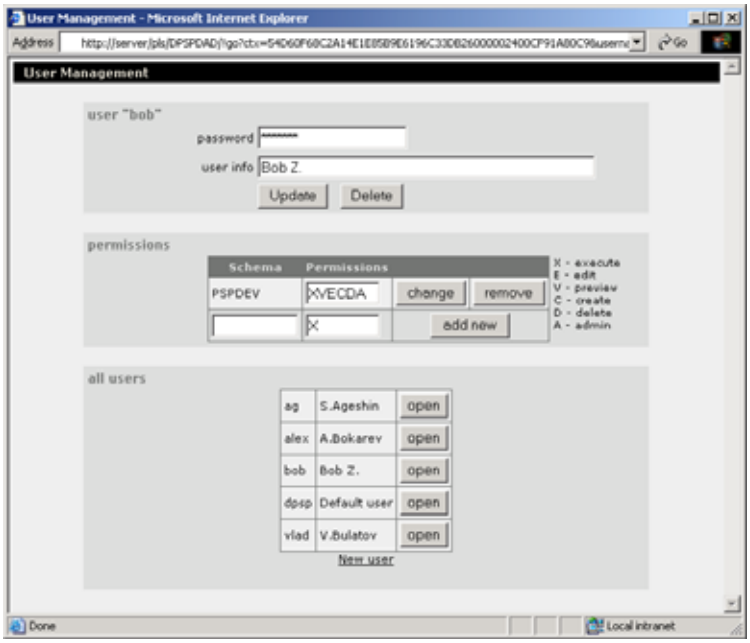
These controls allow you to copy current content of the editor area into clipboard, or paste clipboard contents over current content. This may be useful, if you are using another editor for edits (for example, if you need syntax highlighting). In this case you first copy the whole code into the clipboard using **Copy code** button, paste and edit it in your favorite editor, copy it into clipboard again, paste into the DPSP editor using **Paste code** button, and then save it using **Save** button.

BUILT-IN USER MANAGEMENT.

Dynamic PSP features built-in access control mechanism and user database. Unit Commander provides an interface for managing Dynamic PSP users. This interface is activated by clicking on **Users** button in Unit Commander toolbar. Please note that this interface can only be activated in context of DPSP Kernel (that is, through the DAD to Kernel schema), it is unavailable from project schemas. Also note that Dynamic PSP user names and passwords are **case-sensitive**, that is, when logging in the user should enter his name and password in exactly the same case as they were when the user account was created. The Kernel verifies current Oracle schema to be the Kernel schema, checks current DPSP user permissions, and based on these, denies access to the interface or brings up the User Management screen:



User Management screen allows to create new user accounts or edit credentials and permissions for existing users. To create a new user account, fill in **user name**, **password** and **user info** (usually full user name) fields and click **Create** button. To edit existing user account, click on **open** button next to user name. This will bring up a screen, similar to the following:



On this screen, you can change user's password and info, and assign user permissions for different projects (schemas). User permissions are encoded as string of concatenated permission codes. Each code is one letter designating permission type. The following permission codes are recognized: X – e**X**ecute, E – **E**dit, V – pre**V**iew, C – **C**reate, D – **D**elete, and A – **A**dministration. Resulting permissions mask is used by the Kernel to determine if the user is allowed to execute particular unit in particular project, by intersecting the unit PERMISSIONS attribute with user's permission mask and ensuring that the user's permission mask is at least as powerful as the unit's required permissions mask set by the PERMISSIONS attribute. For example, if user bob has permission mask of XE, and unit 100 PERMISSIONS attribute is X, then bob will be able to execute the unit. If the unit 100 permissions mask is changed to XVE, bob will be denied access because his permission mask does not include V permission. Admin (A) permission controls ability to administer other users in the system. Since user administration units are located in the Kernel schema, and are not published elsewhere, A permission is effective only when set for the Kernel domain (schema). If you set it for other projects, it will have no effect unless your own units in the project have A permission in their permission masks. Even in this case, this permission will not allow the user to administer other users, because the user database is only maintained in the Kernel schema, it will only allow users with admin permission to execute custom units protected with A permission code in other projects.

To grant a user access to the Development Interface for particular project, administrator (user with admin (A) permission on the Kernel schema) should expressly grant him/her access to the project by adding permissions mask for that project to the user's list of permission masks. Administrator does this by typing in the name of the project schema in the Schema field, assigning a permission mask for this schema (usually XE, or XCE, so that user will be able to execute and edit, or execute, create and edit project units), and clicking on **add new** button. To remove a permission mask for particular project, click on **delete** button next to the project schema name, and to change a permission mask, edit it and then click on **change** button to save changes.

