Banner General
API Developer Guide

Release 8.2
July 2010 (Revised)

SUNGARD HIGHER EDUCATION

Trademark, Publishing Statement and Copyright Notice

Prepared by: SunGard Higher Education

4 Country View Road
Malvern, Pennsylvania 19355
United States of America

Customer Support Center Website

http://connect.sungardhe.com

Documentation Feedback

http://education.sungardhe.com/survey/documentation.html

Distribution Services E-mail Address

distserv@sungardhe.com

Revision History Log

Publication Date	Summary
June 2008	New version that supports Banner General 8.0 software.
June 2009	Corrected an error in API package name standards (1-2A6WAQ).
July 2010	Revised version to address documentation defect 1-AXZDV4.

Banner General 8.2
API Developer Guide

Contents

Chapter 5 Automated API Code Generation

Chapter 6 Creating a Form using APIs

Chapter 7 API Testing

Chapter 8 API Technical Documentation

Chapter 9 How to Call Entity APIs

1 Banner's Application Programming Interfaces (APIs)

Why APIs?

Over time, Banner® has been evolving towards a layered architecture that is accessible through Application Programming Interfaces (APIs). The evolution of the architecture has been designed to minimize the impact on your institution.

This architecture, with its APIs:

- Consolidates code, significantly reducing Banner's complexity
- Facilitates interactions between Banner and external entities

Key benefits from using APIs include:

- APIs make Banner business entities and services available to other applications that comprise your institution's digital campus.
- APIs ensure that business logic validations are performed consistently when data is manipulated.
- APIs eliminate duplicate code that had been maintained in separate applications previously.

A Banner *business entity* is the person, place, or thing that the business process operates on. It is the smallest, most granular grouping of information that a Banner application can change. It is defined by information in one form or one form block.

Each business entity that Banner operates on supports the following requirements:

- A business entity is created with valid data before it is stored in the Oracle database.
- A business entity must be able to be retrieved.
- Once created in the system, a business entity must be able to be changed while still supporting valid data in the system.
- A business entity may need to be removed from the system while preserving relationship integrity with other business entities in the system.

Integration and Interoperability

"Depending upon the maturity of applications and the IT infrastructure, higher education institutions spend anywhere from 30% to 50% of annual IT expenditures on application interfaces and integration in pursuit of application interoperability. Building and maintaining lots of separate application-to-application interface programs is costly and time-consuming." - Dave Moldoff, Sr. VP SunGard Higher Education

"For every dollar spent on enterprise applications, an additional $4 - $5 is spent on programmatic adaptation and integration of those applications into the existing enterprise environment." Gartner, October 2001

SunGard Higher Education has responded to this research by delivering a variety of APIs in each release that will help you integrate Banner with other applications on your campus. The Banner APIs can help you integrate the different approaches and technologies at your institution.

Approach	What	Type	Why
Portal Integration and Single Sign-On (SSO)	Channels and Campus Pipeline Integration Protocol (CPIP)	Interactive User	Provides access based on the user and his or her roles.
Process Automation	Workflow	Near real time interchange	Orchestrates business processes.
Web Services	Integration Solutions Web Services Banner Adapter, SOAP	Near real time interchange	Requests services, data, or changes to data.
Publish-Subscribe Messaging	Banner Gateway for OpenEAI, Java Message Service (JMS)	Near real time interchange	Synchronizes changes to enterprise data.
Method Invocation	Remote Procedure Calls, CORBA, Java RMI	Near real time interchange	Provides tight coupling (limited to core modules).
Batch/ETL	Integration Technologies Batch Integration Framework	Periodic interchange	Performs bulk data loads.
Database Replication and Sharing	Database to database at the table level	Periodic and real time interchange	Shares data when applications use the same data model.

Assumptions and Constraints

Banner APIs are implemented in Oracle PL/SQL and are deployed as *Oracle stored program units*. *Oracle stored program units* is a collective term that refers to database stored packages, procedures, and functions. Make every attempt to incorporate existing functions and procedures into packages - use standalone procedures and functions only on an exception basis.

Banner APIs provide support for Banner client applications including the Banner Administrative Client, Banner Self-Service (web, voice, and kiosk), Banner batch programs, Workflow, the Banner Gateway for OpenEAI asynchronous messaging, and Web Service gateways.

2 The Architecture of Banner Business Entity APIs

Banner Business Entity APIs define:

- A set of procedures and functions that define the common operations that every Banner Business Entity supports.
- Common types that support data encapsulation requirements so institutions can manipulate Business Entity data as a whole. Examples of these types include: record definitions, collection types, and reference cursor types.

Common Functions

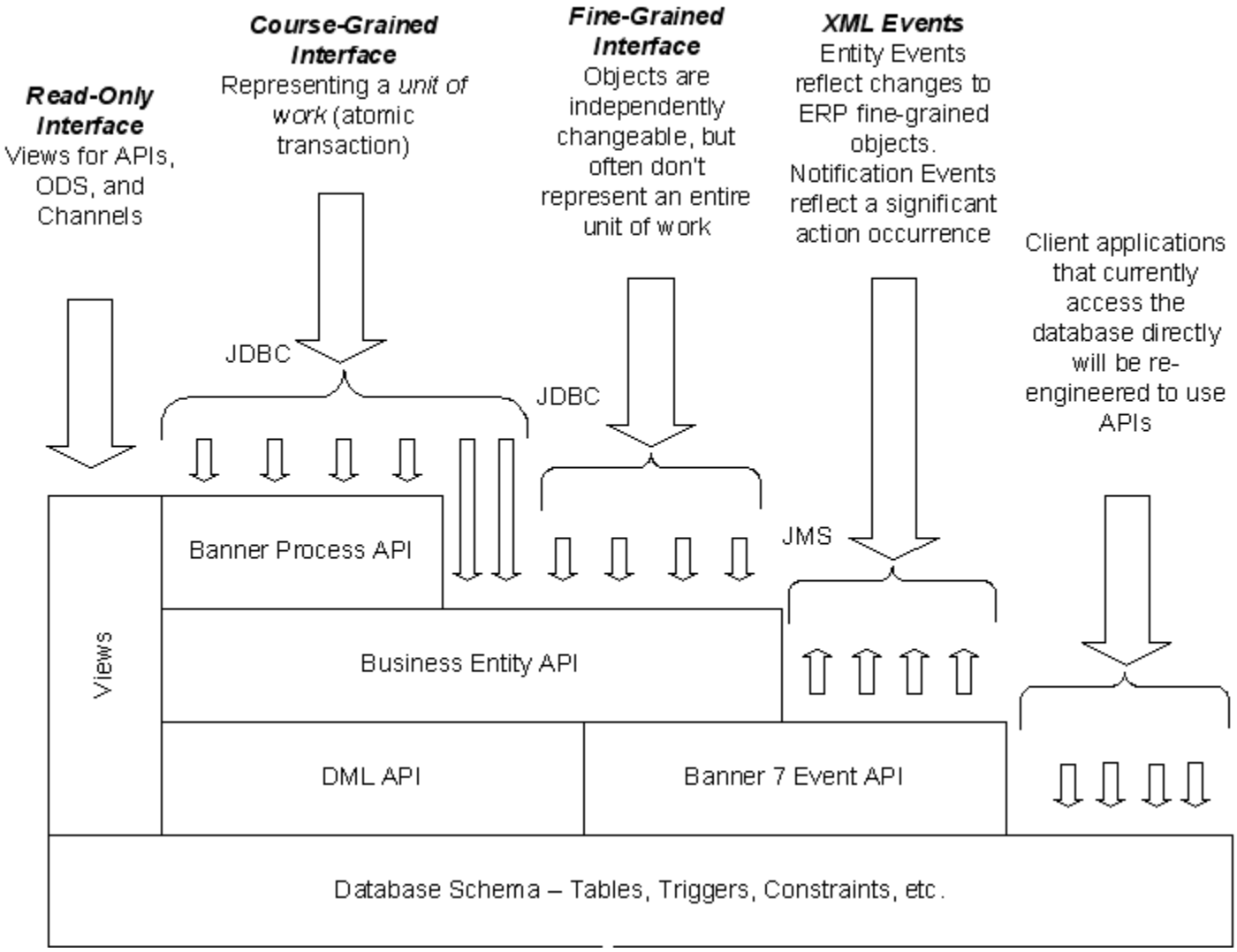
Every Banner Business Entity will support the following minimum operations. Please see the section on package structure for more detail.

Common Function	Description
`f_query_one`	Function to retrieve one business entity via primary key. It returns a reference cursor that is based on the entities record definition type.
`f_query_one_lock`	Same as `f_query_one`, except that it locks the record.
`f_query_all`	Function to retrieve one or more business entities via a key. It returns a reference cursor that is based on the entity's record definition type.
`f_exists`	Function to determine if the business entity exists. It returns VARCHAR2 *Y* if the entity exists or *N* if the entity does not exist.
`f_isequal`	Compares two variables of the APIs record type and returns *Y* if all the fields are equal, *N* otherwise. Nulls match nulls and key fields are not considered in the match.
`f_api_version`	Returns a VARCHAR2 indicating the components version number. The version number will change only when a component of the public signature (package specification) changes.

Common Function	Description
p_lock	Procedure to lock the business entity via primary key. It is used by the API to lock records before modification, and by Banner to enforce the Oracle forms pessimistic locking approach.
p_create	Procedure to create the business entity. The last parameter is of type OUT which is used to return the Oracle ROWID.
p_update	Procedure to update the business entity.
p_delete	Procedure to delete the business entity.
p_validate	Procedure to validate the business entity before it is created or updated. This procedure implements the common business rules for the entity.

API Layers

The Banner layered architecture is comprised of the following APIs:



Banner Business Process API

A Banner Business Process API is a database package that executes a business process spanning calls to multiple Banner Entity APIs in the context of a single transaction. A business process API may expose different interfaces for different purposes: an XML interface to support integrations such as Web Services, an OOT (Oracle Object Type) interface to support simple handling of complex types, or a simple type interface to support legacy applications that cannot immediately take advantage of OOTs or XML. Often, multiple interfaces are exposed for the same business logic within Banner.

Banner Business Entity API

A Banner Business Entity is the smallest, most granular grouping of information that a Banner application client can modify. Examples of business entities include course,

address, e-mail, and purchase order line item. This layer delegates to the DML API, and also interacts with the Banner Event API.

Banner Event API

This API layer was introduced in Banner 7.0 to support the publishing of events that may be of use to external applications. The Business Entity API uses this API to register changes to business entities, which are then captured within an XML event by this API. The Banner Event API exposes a Java Message Service (JMS) interface to external client applications so they can consume the Banner Events in which they are interested.

Validation Packages

This layer looks up codes and retrieve the corresponding descriptions for validation tables.

The code generator that creates the DML layer and the greater part of the Banner Business Entity API layer is described in detail in Chapter 7.

Banner Business Entity API

The Banner Business Entity API provides access to Banner Business entities and supports create, update, delete and read functions.

Each Business Entity API defines a set of methods (procedures and functions) for the common operations that the Banner Business Entity supports. It supports Create, Retrieve, Update, and Delete operations, and it validates business rules for the Entity.

Using these APIs consolidates Banner business logic within a single place and code set.

Keep in mind that Forms and other user interfaces will still need to have some of the same validation logic in order to provide a robust user interface. Therefore, there is not 100% code consolidation, but any business logic that is required for the entity regardless of the user interface must be implemented in the API.

Note

All new Banner tables will have a Business Entity API written for them. ■

DML Layer

The Data Manipulation Layer (DML) is implemented as a separate PL/SQL package and it supports Insert, Update, and Delete operations for the associated business entity. Only the Business Entity API is permitted to call the DML package for a given table. This separates the IO layer from the business logic layer.

The DML layer does not implement business rules for the business entity. It provides a lower-level internal API for business entities to easily insert, update, and delete table information.

Related Packages

Validation Packages

Validation packages are tables that constrain the domain or range of valid values for a particular field. They usually contain only a code and description.

Validation packages were created to provide specific support functions for common operations that APIs need to perform with validation tables. Validation packages verify that a code exists in an underlying table and retrieve its description.

Note

Tables that contain other attributes in addition to a code and description are considered rules tables. SunGard Higher Education has created real APIs for rules tables.

Package Structure

A Business Entity API actually is composed of three packages, the main API package, a *strings* package containing error message text, and a *rules* package containing validation and other optionally other procedures and functions.

Standard API Package Sections

Audit Trail (Spec and Body)

Audit trail requirements, etc., for PL/SQL packages have not changed.

Constants (Spec and Body)

Define all package-level constants in this area. All constants are defined in uppercase; they are the only user-defined items that are not lowercase. If the package is messaging-enabled, group the required messaging constants together, and order them alphabetically by name. Put public constants in the package specification and put constants local to the package in the package body.

Each API should define at least the following constant in the main package specification:

```
M_ENTITY_NAME          CONSTANT VARCHAR2(21) := 'EMERGENCY_CONTACT';
```

where `EMERGENCY_CONTACT` is the name of the Business Entity this package implements. It is known to messaging support.

Each API should define the following constant in the main package body to be returned by the `f_api_version` function:

```
CURRENT_RELEASE      CONSTANT PLS_INTEGER := 1;
```

- This is the version number of the Banner Business Entity this package implements. It is not the version control file version number.

Package-Level Variables (Body)

These are variables that are global to all subprograms in the package body. Use package-level variables sparingly, if at all.

User-Defined Oracle Types (Spec)

All user-defined types should be grouped together in logical order within the package specification:

- Record types
- Table types based on those record types
- Cursor variable types that return those record types

Cursors (Spec and Body)

Only cursors that need to be visible to all subprograms in the package need to be defined at this level. Public cursors can be defined in the package specification, and private cursors global to the package go in the body.

Forward Declarations (Body)

A forward declaration is the signature of a subprogram that is referenced before it is fully defined. (Subprograms defined in the package specification are a type of forward declaration.) They tell the compiler the name, parameters and return values of any subprograms so that proper syntax checking can occur even before the entire subprogram is compiled. Since public subprograms are defined before private subprograms, you may need to include forward declarations so the PL/SQL compiler does not generate an error. See the package template file for examples of forward declarations.

Public Functions (Spec and Body)

These are the functions that can be accessed by any application, the signatures of which are defined in the package specification.

Public Procedures (Spec and Body)

These are the procedures that can be accessed by any application, the signatures of which are defined in the package specification.

Private Functions (Body)

These are the functions that can only be accessed by other subprograms in this package.

Private Procedures (Body)

These are the procedures that can only be accessed by other subprograms in this package.

Standard DML Package Sections

The DML layer requires three signatures to support the business entity layer: `p_insert`, `p_update`, and `p_delete`.

p_insert

This procedure inserts a table row using a parameter of type `table%ROWTYPE`. It returns the Oracle ROWID as an OUT parameter of type VARCHAR2.

```
P_INSERT(p_rec table%ROWTYPE,
         p_rowid OUT VARCHAR2)
```

p_update

This procedure dynamically updates all parameters that do not equal the common UNSPECIFIED values defined in the `DML_COMMON` package. It accepts an optional Oracle ROWID to perform a direct access update.

```
P_UPDATE(p_rec  table%ROWTYPE,
         p_rowid VARCHAR2 DEFAULT NULL);
```

p_delete

This procedure deletes a table row. It accepts an Oracle ROWID to perform a direct table row delete.

```
P_DELETE(p_rowid VARCHAR2 DEFAULT NULL);
```

Package Structure Conventions and Programming Standards

This section describes all the sections in a package. Not all sections are always required. All packages should conform to existing coding standards.

Parameter Names

The PL/SQL standards for parameter names are:

Parameter Type	Name Format
IN parameters	`p_<name>`
OUT parameters	`p_<name>_out`
INOUT parameters	`p_<name>_inout`

This easily distinguishes local variables from passed parameters, while simplifying the indication of IN parameters (the mode for parameters is IN by default).

Package Design Concepts

Encapsulation

Encapsulation means writing programs with clearly defined interfaces that hide complex processing logic. Business processing logic is performed in private centrally located procedures, and application programs do not need to be concerned with the details of the implementation. The application programs only need to make sure their procedure calls conform to the requirements of the public signature.

Modularity

Good programming practice requires that each small program unit do one job well, and defer other tasks to other program units. The main body of a procedure or function should be only a few screens of text, at most.

Global variables are discouraged because they allow program units to change the behavior of other units in ways that are difficult to trace. All functions should have exactly one return statement, so the exit point is easy to identify. Functions should never have OUT parameters because they can produce difficult-to-trace side effects.

Scope

A procedure or function that is only referenced by one other procedure or function should be placed in the declaration section of that subprogram, and not at the package level. Only subprograms that need to be called from multiple subprograms should be defined at the package level.

The rule is that you should define every program object at the lowest possible scope.

Exception Handling

Write exception handlers only for exceptions you actually intend to handle. Avoid using `WHEN OTHERS` (it can hide errors) unless your handler either completely deals with the problem or you make it propagate the exception by raising it again. That way, you can cause exceptions to propagate up to the calling procedures so any error messages are eventually displayed.

Naming Conventions

API Package Naming Conventions

Banner code that has gone through the Banner API development process has been packaged as Oracle stored packages and named using the following conventions:

- Each package name begins with the single letter Banner product identifier. For example, *G* represents *General*, *S* represents *Student*, etc.
- The second letter indicates *B* for Business Entity APIs and *P* for Business Process APIs.
- The package name describes the business entity following Oracle package name constraints.
- Oracle package names are limited to 30 characters.
- Business entity names are separated by the underscore character (_).

Some sample Banner General API package names are:

- `GB_EMERGENCY_CONTACT`
- `GB_ADDRESS`
- `GB_TELEPHONE`
- `GB_EMAIL`

API Package File Name Conventions

API package file names will follow the standard conventions with the following additions:

- File names are limited to 21.3.
- If a file name must be abbreviated to fit the 21.3 rule, create one abbreviated string and use it consistently for the API, rules, and strings package file names.
- The first letter of the package name identifies the product (e.g., *G* for *General*, *S* for *Student*, etc.)
- The second letter identifies the module (e.g., *O for Overall*, *P* for *Packaging*, etc.)
- The third through the fifth letters are always the same: *KB_*
- The rest of the letters are the name of the API.
- The last character of the filename will be a zero (*0*) to identify the file as a package specification or a one (*1*) to identify the file as a package body.
- For strings packages, there will be an `_s` before the *0* or *1*, and for rules there will be an `_r` before the *0* or *1*.

As an example, the file names for the `GB_EMERGENCY_CONTACT` API are:

- The package specification is `gokb_emergency_cont0.sql`
- The package body is `gokb_emergency_cont1.sql`
- The strings package is `gokb_emergency_cont_s0.sql` and `gokb_emergency_cont_s1.sql`
- The rules package is `gokb_emergency_cont_r0.sql` and `gokb_emergency_cont_r1.sql`

DML Package Naming Conventions

DML package names are in the format `DML_`*tablename*, where *DML_* is the package name prefix and *tablename* is the actual Banner table name.

For example, `DML_SPREMRG` is the DML package name for the Emergency Contact Table (SPREMRG).

DML Package File Name Conventions

Package file names will follow the standard Banner package file name guidelines with the following additions.

- The package file name may contain a maximum of 16 characters.

- The fourth letter of the package name will be the value *D* to identify a table DML package.
- The fifth character of the package name will be the underscore character (_).
- The last character of the filename will be a zero (*0*) to identify the file as a package specification or a one (*1*) to identify the file as a package body.

For example:

- `GOKD_SPREMRG0.sql` is the DML package specification filename for the SPREMRG table.
- `GOKD_SPREMRG1.sql` is the DML package body filename for the SPREMRG table.

New Transaction Model

Banner APIs require a new transaction model. Application clients that call each Banner Business Entity API are now required to remove SQL COMMIT and ROLLBACK statements from their client-side code. Instead, application client programs must call `GB_COMMON.P_COMMIT` to commit the current transaction, or `GB_COMMON.P_ROLLBACK` to rollback the current transaction. This is necessary because messages are published at the end of the transaction when `P_COMMIT` is called. If application clients do not call `GB_COMMON.P_COMMIT`, messages will not be generated and published.

Likewise, if an exception is raised during the business processing, application clients must call `GB_COMMON.P_ROLLBACK` to remove all business entities that have been registered with messaging support, and to rollback the current database transaction.

Any application that calls an API, either directly or indirectly (via another package) must issue a call to either `gb_common.p_commit` or `p_rollback` at the end of the transaction. Otherwise, the data synchronization message data will be lost.

Note

There are special considerations for table triggers that call APIs because of these required COMMIT and ROLLBACK statements. A library was changed to deal with this situation for Forms. However, all other applications (e.g., C programs) that could call an API indirectly via another procedure must be changed to issue `p_commit` and `p_rollback`. This includes database triggers on tables, where the trigger could call an API even if the table itself does not have an API.

Identifying whether or not a program indirectly calls an API may take extensive research using a tool (e.g., CAST).

Business Entity Architecture

Use the following approach for layering business entity functionality:

- The Common API for each Business Entity is implemented in one Oracle package. The package is named for the Business Entity.
- The Business Entity's common business rules are implemented in a RULES package.
- Text that is displayed to the user (e.g., error messages) is implemented in a STRINGS package.

For example:

- `GB_EMERGENCY_CONTACT` is the common API package for the Emergency Contact Business Entity.
- `GB_EMERGENCY_CONTACT_RULES` contains the `P_VALIDATE` procedure and other common business rules for Emergency Contact.
- `GB_EMERGENCY_CONTACT_STR` contains text strings that are displayed to the application clients.

Application Security and Auditing

Some applications require a level of security beyond the normal login security. One example might be an application that allows Finance users to update information for one FOPAL code only. This security is:

- Defined within Banner.
- Based on the user ID.
- Unlikely to be understood by all external API applications and messaging partners.

Another aspect of this security is the ability to audit the source of the data. You need to determine what application was the source of the data and who (which person) made the change.

To accomplish this, the following database changes were made:

- A new column, `<tablename>_data_origin VARCHAR2(30)`, was added to every table for which an API is built. This field is nullable. It will be populated with the data source's name, e.g., e-Procurement, Banner, etc.
- A `<tablename>_USER_ID` field was added to every table that did not already have one.

- The `user_id` field will always default to `gb_common.f_sct_user` in the API. Currently, this function returns `sys_context('USERENV', 'SESSION_USER')`. If you are using proxy accounts to map external users to local Oracle usernames, this will return the local username instead of the logged on username, providing a more accurate audit of who created or changed the data.

The DATA_ORIGIN Constant

SunGard Higher Education has included a package global constant `DATA_ORIGIN` to `gb_common` which has the value *Banner*.

```
DATA_ORIGIN CONSTANT VARCHAR2(6) := 'Banner';
```

This is referenced in the API as `gb_common.DATA_ORIGIN`. This is for any Banner code that needs to define the default value for a `<table>_data_origin` column.

Because `gb_common.DATA_ORIGIN` is not accessible from Forms, the global Forms variable `:global.data_origin` was defined for this purpose.

`DATA_ORIGIN` is defined as:

- The column name is `<tablename>_DATA_ORIGIN`
- It can be null
- The data type is VARCHAR2
- The length is 30
- The description is *Source system that created or updated the data*

`DATA_ORIGIN` will be populated on insert and update.

Note

Create Source will be added to tables where there is a functional need to store and preserve the source of the original insert. This should only be placed on the tables to meet a specific functional requirement. It does not need to be added to all tables. ■

Note

The data origin and the create source can be displayed on forms when there is a functional need, but it is not required. ■

All Banner products will use *Banner* for `DATA_SOURCE`, including the self-service products.

Package Specification Components

This section describes the components and special features of a package that implements a Banner Business Entity. These are created by the API code generator, but must be present regardless of the method used to create the API.

Note

There are cases where additional functions and procedures make sense, so an API could have more program units than just the standard ones. Conversely, there are cases where some standard functions are not appropriate. For example, if an API exists for a table from which records are never deleted, the API might not have a `p_delete` procedure.

Standard Functions

f_api_version

Returns `CURRENT_RELEASE` in all cases. This is the code:

```
FUNCTION F_API_VERSION
   RETURN PLS_INTEGER IS
 BEGIN
   RETURN(CURRENT_RELEASE);
 END F_CURRENT_VERSION;
```

f_build_<tablename>_rec

This function will take the column parameters for the API and assign them to their respective columns in a local variable which is a record type defined to match the underlying table. This is needed since the logic to call the DML is passing records and not individual columns.

f_isequal

This function takes two records of the business entity record type as parameters, and compares them for equality. This will initially be useful during testing where an API call is made and then the data is fetched back to compare the new record against a test record to ensure the change was made correctly. It compares each field in one record with the corresponding field of the other record.

Note

If both are NULL then the fields are considered equal.

f_exists

This function returns a Y/N value to say whether or not a record specified by the key values in the parameters exists in the table. Parameters must be the primary key values of the entity. If a primary key is not available it should define a set of parameters that

can resolve as specifically as possible, with the understanding that *exists* may mean that more that one record exists for the given key.

f_query_all

This function is similar to the one above, but returns a reference cursor that will fetch multiple rows. This function returns different things depending on the requirements of the development or product group.

For example, you may need to return all addresses for a PIDM, or all addresses for a PIDM and address type. The record type returned is defined as all of the fields in the table, plus ROWID.

f_query_lock

This function locks a row, but at the same time returns the entire record that is being locked. This is used in other parts of the system where both a lock and reading the database record state before an update is required.

f_query_one

This function returns a cursor variable (reference cursor) for the entity, defined by the key values passed as parameters. The intent of this function is to return a reference cursor that will fetch exactly one row based on the primary key values of the entity. The record type returned is defined as all of the fields in the table, plus the rowid.

Note

If the table doesn't have a unique key, be sure the documentation for the function states that the row returned may not be the only row that meets the condition.

Standard Procedures

p_create

This procedure creates a record in the table using the values passed as parameters. It also performs any other business processing and validation required by the business entity.

p_delete

This procedure will delete a record, specified by the primary key values passed as parameters, or ROWID, and also perform any other associated processing required by the business logic.

p_lock

This procedure issues a lock against the Banner table.

p_overlay_validation_rec

A mechanism was developed to support the dynamic generation of update statements. This allows the application program to only specify those optional parameters that it knows about and needs to update.

Therefore, for each basic datatype (CHAR, CLOB, DATE and NUMBER) a constant was created in the `dml_common` package to define an *Unspecified* value. The value chosen for each is one that is extremely unlikely to appear as an actual value for an item of that data type.

p_update

This procedure will update a database table by calling the appropriate DML procedure that will generate a dynamic update statement. Primary key values are required, and all other values can be passed as named parameters.

Parameters

There are several specific features of the parameters passed that need to be highlighted.

Unspecified Values

Parameters that are passed to the DML `p_update` procedure with this unspecified value are excluded from the dynamic update statement.

Unspecified values are:

Constant Name	Value
`UNSPECIFIED_CLOB`	`CHAR(1)` - ASCII SOH character
`UNSPECIFIED_DATE`	`TO_DATE('10000101', 'YYYYMMDD')`
`UNSPECIFIED_NUMBER`	`1E-35` - one raised to the -35 power
`UNSPECIFIED_STRING`	`CHR(1)` - ASCII SOH character

Given a record type variable and a set of separate values, it examines each parameter and sets the corresponding record value to the value passed in, as long as it is not equal to the *Unspecified* value.

Warning

Be aware that PL/SQL will round numeric values when they are passed between procedures when the called procedure has defined the parameter with a DEFAULT such as table.column%type, or, for example, NUMBER(9,2).

Note

PL/SQL will always round the value to the most constrained type, so passing `UNSPECIFIED_NUMBER` to a procedure that has DEFAULT (9.2)

will cause the `UNSPECIFIED_NUMBER` to be rounded to 0. This can cause very odd problems, not the least of which is data corruption. ■

ROWID

In general, each Business Entity is defined by the values in the primary key of the underlying table. There are situations however where a primary key is not available:

- *Update and Delete* - Each Update and Delete procedure also has a ROWID IN parameter. Delete and Update procedures use the ROWID if present and ignore other key parameters. If ROWID is null, then the other parameters that are part of the primary key are used to specify the record to be deleted or updated.
- *Query* - For Query functions, the ROWID is returned in the record. It is not passed as a parameter, because the assumption is that if you have a ROWID already, you had some other means of querying the database to get it.
- *Create* - For Create operations, the ROWID is an OUT parameter, and the procedure passes out the ROWID value of the new record that is created. Use the RETURNING clause of the INSERT statement to obtain the ROWID.

There is a remote possibility that an application program may call an API with a ROWID that corresponds to a different row than the one defined by the key parameters. This could result in significant and hard-to-trace data corruption.

The processing overhead necessary to validate this condition within the API would be too great to be of any practical use. Please be aware of this situation and make sure that the applications you are creating or changing pass the correct parameters.

Note

In most cases, if a ROWID is passed to an API, the API will attempt to use it (rather than the key values) for locating rows in the database for update and delete operations. This is not a hard-and-fast rule, however, as it may use either the keys passed or the ROWID to locate a row. ■

Out Parameters

In addition to the ROWID parameter on a Create operation, there may be cases where a new value, such as a sequence number, is generated by a Create operation. In those cases, the generated value is returned as an OUT parameter.

The generated value is placed in the end of the parameter list, before the ROWID parameter.

Remember, OUT parameters are required, so a writeable variable must be passed in the API call to accept the results.

Handling Context Differences

A function and a procedure have been delivered in the GB_common package that provide a mechanism by which an API can be made to execute differently depending on the calling context.

Signatures

```
PROCEDURE p_set_context (
     p_package_name    VARCHAR2,
     p_context_name    VARCHAR2,
     p_context_value   VARCHAR2,
     p_stateless_ind   VARCHAR2 DEFAULT 'Y'
  );
```

The parameters are:

- p_package_name - name of the API package requiring the context information.
- p_context_name - any string agreed upon by the calling application and the API. It should have a functional meaning, and not be the name of a program, etc.
- p_context_value - any string, should have a functional meaning.
- p_stateless_ind - defaults to *Y*, meaning one call to f_get_context for this variable will both return and erase it. If you pass any other value, the value persists between API calls until the API clears it or the session ends.

Function

```
 FUNCTION f_get_context (p_package_name VARCHAR2, p_context_name
VARCHAR2)
      RETURN VARCHAR2;
```

The parameters are:

- p_package_name - Name of the API package requiring the context information.
- p_context_name - Any string agreed upon by the calling application and the API. Returns the value that was assigned in the p_set_context call.

Usage

No case conversion is done, and no verification that p_package_name is an actual package is performed.

It is recommended that all parameter values be passed as uppercase - not the parameter name in the code itself, but the parameter value. For example, p_context_name =>'EXAMPLE'.

In the future we may be able to implement a security check so that only certain users may set certain context values.

Here is a simple run in sqlplus to show how this works:

```
SQL> execute gb_common.p_set_context('GB_ADDRESS','FINANCE_ADDRESS_RULES','TRUE');
```

PL/SQL procedure successfully completed.

```
SQL> select gb_common.f_get_context('GB_ADDRESS','FINANCE_ADDRESS_RULES') from du-
al;
TRUE
SQL> select gb_common.f_get_context('GB_ADDRESS','FINANCE_ADDRESS_RULES') from du-
al;
SQL>
```

Note

In the second call for the `FINANCE_ADDRESS_RULES` context variable above, the value has been reset to NULL, so nothing is returned. ■

This is the default behavior; it is set for only *one* call of `f_get_context`.

```
SQL> execute
gb_common.p_set_context('GB_ADDRESS','FINANCE_ADDRESS_RULES','TRUE','N');
```

PL/SQL procedure successfully completed.

```
SQL> select gb_common.f_get_context('GB_ADDRESS','FINANCE_ADDRESS_RULES') from du-
al;
GB_COMMON.F_GET_CONTEXT('GB_ADDRESS','FINANCE_ADDRESS_RULES')
TRUE
SQL> select gb_common.f_get_context('GB_ADDRESS','FINANCE_ADDRESS_RULES') from du-
al;
GB_COMMON.F_GET_CONTEXT('GB_ADDRESS','FINANCE_ADDRESS_RULES')
TRUE
```

Note

This time the value is retained in the second call for the `FINANCE_ADDRESS_RULES` context variable. This is because we passed *N* for the stateless indicator. ■

Helper Packages

In addition to implementing the public subprograms defined in the specification, it may be useful to create helper packages that implement specific parts of the business logic or otherwise support the main subprograms. You can design these packages in any way that makes sense to your product, but be certain these packages do not attempt to perform CRUD on Banner Business Entities directly. The goal is to make sure the API packages do not grow too large to manage, and at the same time keep all code in modular units.

Candidates for helper packages are:

- Rules packages that can contain record validation functions, messaging registration procedures, etc.
- Strings packages that contain all literal strings (primarily error messages) that the API will return to the calling application.
- Functions that perform more complex business validation (e.g., does the PIDM have a current enrollment record for the specified term?)
- Functions that do some specific calculations or other processing
- Common cursors for doing more elaborate queries than provided by `query_one` and `query_all`

The rules for helper packages are:

- They can only be called by the API, never directly by an application program.
- They should include large or infrequently-used segments of code. This will reduce the overhead involved in loading more code into memory than is required to perform common tasks.

To avoid confusion, it is crucial that no other package contain public signatures that are identical to the main Banner Business Entity API procedures.

The names of these packages should make it clear what Business Entity they pertain to.

Rules Packages

The rules package can have any functions and procedures you require to perform the business logic for the API.

There may be cases where, due to the nature of the base tables, multiple APIs are so similar that there is little or no need for separate rules packages. In this case, you can have a set of APIs refer to a common rules package. Name them in a logical way and make sure they conform to the same API package standards.

Messaging Constants

Messaging constants are PL/SQL constants defined in the DML package that contain the names of the Business Entity and its attributes.

Strings Packages

The strings package defines API-generated exception messages. A table of messages is defined in the strings package specification and populated by the initialization code in the package body. The function `f_get_error` is defined that returns the internationalized string associated with the label.

Normal PL/SQL constants cannot be used because they are not accessible from Forms. With this arrangement, however, any application can fetch the string associated with a particular message label.

Each is defined using the Internationalization function `g$_nls.get`, which allows the strings to be easily translated into other languages.

Note

All API packages that emit literal message strings of any sort must first pass them to `g$_nls.get`. ■

There may be cases where, due to the nature of the base tables, multiple APIs are so similar that there is little or no need for separate strings packages. In this case, it is perfectly OK to have a set of APIs refer to a common strings package. Name them in a logical way and make sure they conform to the same API package standards.

Error Processing

The API throws exceptions using the standard Oracle exception handling mechanism for a variety of reasons:

- Validation failure
- Business logic failure
- Some other condition determined by the developer to warrant an exception

Exceptions are either API-generated or Oracle-generated.

Oracle exceptions are generally not handled unless the program can correct the problem and continue processing, and are allowed to propagate up to the application program. Do not use the WHEN OTHERS exception handler unless you make sure you raise any exception again that was not completely handled the first time.

Handling API Exceptions

All API exceptions are raised using `RAISE_APPLICATION_ERROR` with a predefined exception message number: `GB_COMMON.ERR_CODE` which is defined as -20100.

Generally speaking, each type of application is responsible for handling errors in its own fashion.

The calling program is responsible for issuing a `gb_common.p_rollback` call when it receives an exception from an API. The API will not automatically commit or rollback any data.

The API will validate all of the fields in the record on `p_create` and `p_update` operations.

Note

Because the API validates everything, not just the data being updated, it is possible that an API error could be returned with a validation failure on a field that was not part of the update. This could happen if existing data on the table is invalid.

All messages for failed validations are returned in a single exception.

User Exits

At the request of several clients, SunGard Higher Education has implemented a method by which you can install your own custom procedures that will be called from the APIs. This allows you to execute custom validations and other operations without having to change Banner baseline code.

The Local API User Exit Mapping Table (GUBLAPI), contains the names of your site-specific package/procedure combinations and the name of the delivered API you want them to work with. The parameters that are passed to the APIs will be passed to your packages.

You must use insert rows into this table via SQL - no Banner form supports this table at this time.

Implementation Examples

The following describes how you can implement a user exit in the GB_EMAIL API.

1. Create a package that contains a procedure with the same signature as the p_create procedure in your custom API:

```
REM
REM This is a local procedure which is mapped to the GB_CM_SOURCE_PRIORITY routine on the
REM GORCMSP table.
REM
REM This is just an example of a local routine that raises and exception
REM so you can till that it was called.
REM
create or replace procedure P_LOCAL_EMAIL_VALIDATION (
       p_pidm,
       p_emal_code,
       p_email_address,
       p_status_ind,
       p_preferred_ind,
       p_user_id,
       p_comment,
       p_disp_web_ind,
       p_data_origin) is
begin
  raise_application_error(GB_COMMON_STRINGS.ERR_CODE,
       'User Exit procedure was called.');
end P_LOCAL_VALIDATION;
/
```

2. Add the GUBLAPI entry that identifies this new package. Be sure to refer to gublapi_ins.sql:

```
REM
REM gublapi_ins.sql
REM
REM Create record in table to map a LOCAL procedure to the GB_EMAIL API.
REM
INSERT INTO GENERAL.GUBLAPI
       (GUBLAPI_API_NAME,
        GUBLAPI_LOCAL_API_NAME,
        GUBLAPI_USER_ID,
        GUBLAPI_ACTIVITY_DATE)
VALUES ('GB_EMAIL',
        'P_LOCAL_EMAIL_VALIDATION',
        USER,
        SYSDATE)
/
```

3. Recompile the API package body:

```
SQL> @gokb_email1.sql
```

4. Execute the API and see the new exception:

```
Declare
lv_rowid VARCHAR2(19);
BEGIN
 gb_email.p_create(
 p_pidm          => &pidm,
 p_emal_code     => '&Email_code',
 p_email_address=> 'test@test.com',
 p_rowid_out     => lv_rowid);
END;
/
```

An Example Run Before a User Exit is Installed

```
S4B70>Declare
  2  lv_rowid VARCHAR2(19);
  3  BEGIN
  4   gb_email.p_create(
  5   p_pidm          => &pidm,
  6   p_emal_code     => '&Email_code',
  7   p_email_address=> 'test@test.com',
  8   p_rowid_out     => lv_rowid);
  9  END;
 10  /
Enter value for pidm: 509
old   5:  p_pidm          => &pidm,
new   5:  p_pidm          => 509,
Enter value for email_code: MA
old   6:  p_emal_code     => '&Email_code',
new   6:  p_emal_code     => 'MA',

PL/SQL procedure successfully completed.
```

An Example Run After a User Exit is Installed

```
S4B70>Declare
  2  lv_rowid VARCHAR2(19);
  3  BEGIN
  4   gb_email.p_create(
  5   p_pidm          => &pidm,
  6   p_emal_code     => '&Email_code',
  7   p_email_address=> 'test@test.com',
  8   p_rowid_out     => lv_rowid);
  9  END;
 10  /
Enter value for pidm: 509
old   5:  p_pidm          => &pidm,
new   5:  p_pidm          => 509,
Enter value for email_code: MA
old   6:  p_emal_code     => '&Email_code',
new   6:  p_emal_code     => 'MA',
Declare
*
ERROR at line 1:
ORA-20100: ::User Exit procedure was called.::
ORA-06512: at "BANINST1.GB_EMAIL_RULES", line 179
ORA-06512: at "BANINST1.GB_EMAIL", line 279
ORA-06512: at line 4
```

3 Banner Event API

Publishing Entities to External Systems

The Business Services API, Business Control API, and Business Entity API layers provide access to Banner data from external client applications so that these applications can retrieve and change Banner data.

The Banner Event API allows these external client applications to be notified when Banner data is changed; the impetus comes from Banner. As we move to a true Service Oriented Architecture (SOA), the external client applications should request data when it is needed. However, it will always be necessary for Banner to notify external client applications when Banner entities are changed. For example:

- Applications that do not move to an SOA may still need Banner to synchronize their data. Because Banner is considered the system of record (the authoritative source) for most Banner Business Entities, Banner must provide a mechanism to communicate all changes to these entities to other systems within the enterprise. The Banner Gateway for OpenEAI may consume Banner events to synchronize data across applications within an OpenEAI messaging enterprise.
- Workflow may consume events to initiate workflows.
- Web Service gateways may consume events, causing a third-party web service to be triggered.

Note

The third-party web service may be used to synchronize its own database.

The Banner Business Entity API allows all client application types to change the Banner Business Entities. Since this API is the single interface in Banner to change Banner Business Entities, this API is used to register changes to each entity so they can be included in a subsequent published event.

The solution for communicating events to the external client applications differs between Banner 6.0 and Banner 7.0, but the interface used by the Banner Business Entity API to register changes with the solutions to be described in this section remains essentially the same.

Specifically, Banner 6.0 incorporated Banner Messaging Support. It served the same purpose as the Banner 7 Event API, but it used an internal event format to communicate

directly with the Banner Gateway for OpenEAI via Java Remote Method Invocation (RMI). (The Banner Gateway for OpenEAI is used with Integration for e-Procurement.)

Banner 7.0 incorporates the Banner Event API, which provides an interface to register entity changes. The Banner Event API creates *XML Events* reflecting entity changes, and publishes them to an Oracle Advanced Queuing multi-consumer queue. External application client types (like the gateway used within the Integration solution) subsequently consume these XML Events. The gateway will use these events for the purposes of creating and publishing 'synchronization messages' to the enterprise to reflect changes to the data.

In future releases the Banner Event API may also be used to provide *notification* events that are not associated to changes to Banner Business Entities. The XML Event Schema has been designed to support multiple types of events.

Banner Event API

The Banner Event API (introduced with Banner 7.0) provides a flexible, decoupled solution for communicating events to other application clients within the enterprise.

It provides procedures that are used by the Business Entity API to register changes to a Banner Business Entity so that those changes can be reflected in an XML Event.

Banner Event API Configuration

Event Support System Wide Configuration

The column `GENERAL.GUBINST.GUBINST_MESSAGE_ENABLED_IND` controls whether event production is enabled for the entire Banner system. *Y* indicates event production is *enabled* system wide. All other values indicate event production is *disabled* for the entire Banner system.

Event API Support Business Entity Configuration

The table GENERAL.GURMESG enables event production for each Banner Business Entity. Each row defines the business entity name, source indicator and enabled indicator. The source indicator will be *B* for *Baseline* data or *L* for *Customer Local* data. The enabled indicator will be *Y* for *enabled*, and all other values indicate *disabled*.

GB_EVENT

The Banner package named GB_EVENT defines the specification for the Banner Event API. The Banner Business Entity API will interact with the Banner Event API to make Banner business entity data available to external systems. Here is a high level overview:

1. After Banner business logic has successfully completed, the Banner API program will check to see if event production is enabled for this business entity via the GB_EVENT.F_ENTITY_PUBLISHABLE function.

2. If event production is enabled for this business entity, the Banner API program will register this business entity with the event API and identify the operation type that was performed via the GB_EVENT.P_REGISTER_ENTITY procedure.

3. The current state of the business entity data is captured via multiple GB_EVENT.P_ADD_PARAMETER procedure calls.

4. The Banner client program issues a GB_COMMON.P_COMMIT to publish an event containing entity information and commit the transaction.

The API routine must check to determine if this business entity is publishable:

```
FUNCTION F_ENTITY_PUBLISHABLE(
        p_SOURCE_IND              VARCHAR2,
        p_ENTITY_NAME             VARCHAR2 DEFAULT NULL)
    RETURN BOOLEAN;
```

Where P_SOURCE_IND will contain either *B* for *Baseline* or *L* for *Local Client Modifications* referenced via GB_EVENT.BASELINE_IND, and P_ENTITY_NAME is the business entity name. A Boolean value of *TRUE* will be returned if this business entity is publishable.

After a business entity has been successfully changed by the business logic, register this business entity with messaging support and identify the type of change.

```
PROCEDURE P_REGISTER_ENTITY(
       p_ENTITY_NAME           VARCHAR2,
        p_OPERATION_TYPE           NUMBER);
```

Where P_ENTITY_NAME is the business entity name and P_OPERATION_TYPE defines the type of operation performed. The constants GB_EVENT.CREATE_OPERATION, UPDATE_OPERATION, and DELETE_OPERATION should be used to identify the operation type.

Once a business entity has been registered by the P_REGISTER_ENTITY procedure, data associated with the entity can be added.

```
PROCEDURE P_ADD_PARAMETER(
       P_NAME                  VARCHAR2,
        P_VALUE                    VARCHAR2);
```

Where P_NAME is the business entity attribute name and P_VALUE is the business entity attribute value. This procedure is overloaded to support passing DATE and NUMBER types for the value.

When all business entities have been registered with the Banner Event API, the P_PUBLISH procedure will be used (by GB_COMMON.P_COMMIT) to create an XML Event reflecting the affected entities and publish it to an Oracle AQ multi-consumer queue.

If an error occurs in a Banner application and a rollback is required, then the procedure P_DISCARD must be called to remove all business entities that have been registered within the current transaction.

Special Purpose APIs

```
PROCEDURE P_DISABLE_SYNC_PUBLISHER;
PROCEDURE P_ENABLE_SYNC_PUBLISHER;
```

The message gateway used within Integration must not publish a synchronization messaging in response to an XML Event resulting from entity changes made due to consumption of an OpenEAI SYNC message type. The P_DISABLE_SYNC_PUBLISHER procedure will ensure an XML Event is flagged so it will not be used for synchronization messaging by the gateway. After the Banner Gateway processes the SYNC message it will call the P_ENABLE_SYNC_PUBLISHER procedure so events are no longer flagged to prevent synchronization publication.

```
PROCEDURE P_SET_BULK_SYNC_CODE
```

This procedure enables bulk sync operating mode for event publishing and specifies a bulk sync code to be used in the dispatched events that can be used to associate the events with a specific bulk synchronization task (such as initializing a new application added to the enterprise). The events published when in this mode, will be published to a special 'Bulk Sync' multi-consumer Oracle AQ destination.

```
PROCEDURE P_CLEAR_BULK_SYNC_CODE
```

This procedure disables bulk sync operating mode for event publishing and clears the bulk sync code.

Publishing Banner Business Entities

You must develop Banner Business Entity API procedures for an entity so changes to it can be reflected in Banner events.

Client application requirements are more stringent when changes to entities must be communicated to other systems (e.g., through the Banner Gateway for OpenEAI). This means that it is possible to create an API and gradually change client applications to call it, but you cannot have a fully integrated messaging system until all applications call the API.

A portion of the Banner System will be considered enabled for entity publishing when:

1. The Banner Business Entity API has been applied to a Banner business entity.

2. The Business Entity package has changed the `P_CREATE`, `P_UPDATE`, and `P_DELETE` procedures to register the business entity information with Messaging Support.

To include a change to a Banner Business Entity in a Banner event, the Banner Event API must have the following information:

1. The Business Entity must be defined via an entry in the GURMESG table. This table controls which business entities are enabled for 'event' functionality.

2. The operation type that altered the Business Entity, e.g., create, update or delete.

3. The current state of the Business Entity data. It is identified by the Business Entity attribute names and values.

The Business Entity name and entity attribute names should be defined as constants in the Business Entity Rules package.

The following is an example of these constant definitions for the `EMERGENCY_CONTACT` business entity.

```
-- Constants
M_ENTITY_NAME        CONSTANT VARCHAR2(17) := 'EMERGENCY_CONTACT';
M_PIDM               CONSTANT VARCHAR2(4)  := 'SPREMRG_PIDM';
M_PRIORITY           CONSTANT VARCHAR2(8)  := 'SPREMRG_PRIORITY';
M_LAST_NAME          CONSTANT VARCHAR2(9)  := 'SPREMRG_LAST_NAME';
M_FIRST_NAME         CONSTANT VARCHAR2(10) := 'SPREMRG_FIRST_NAME';
M_MI                 CONSTANT VARCHAR2(2)  := 'SPREMRG_MI';
M_STREET_LINE1       CONSTANT VARCHAR2(12) := 'SPREMRG_STREET_LINE1';
M_STREET_LINE2       CONSTANT VARCHAR2(12) := 'SPREMRG_STREET_LINE1';
M_STREET_LINE3       CONSTANT VARCHAR2(12) := 'SPREMRG_STREET_LINE1';
M_CITY               CONSTANT VARCHAR2(4)  := 'SPREMRG_CITY';
M_STAT_CODE          CONSTANT VARCHAR2(9)  := 'SPREMRG_STAT_CODE';
M_NATN_CODE          CONSTANT VARCHAR2(9)  := 'SPREMRG_NATN_CODE';
M_ZIP                CONSTANT VARCHAR2(3)  := 'SPREMRG_ZIP';
M_PHONE_AREA         CONSTANT VARCHAR2(10) := 'SPREMRG_PHONE_AREA';
M_PHONE_NUMBER       CONSTANT VARCHAR2(12) := 'SPREMRG_PHONE_NUMBER';
M_PHONE_EXT          CONSTANT VARCHAR2(9)  := 'SPREMRG_PHONE_EXT';
M_RELT_CODE          CONSTANT VARCHAR2(9)  := 'SPREMRG_RELT_CODE';
M_ATYP_CODE            CONSTANT VARCHAR2(9)  := 'SPREMRG_ATYP_CODE';
```

Example Code Snippet for GB_EMERGENCY_CONTACT.P_CREATE

Here is an example of how the Emergency Contact business entity P_CREATE API will interact with the Banner Event API to make this data available within an event message.

```
-- NOTE:  Banner Event API logic is only called after a successful DML operation
    -- Check if this business entity is publishable via an event.
IF (gb_event.f_entity_publishable( gb_event.baseline_ind,
         gb_emergency_contact_rules.M_ENTITY_NAME )) THEN
        -- Register this business entity with the Event API.
        gb_emergency_contact_rules.p_register_entity(
                     gb_messaging.CREATE_OPERATION,
                     p_pidm,
                     p_priority,
                     p_last_name,
                     p_first_name,
                     p_mi,
                     p_street_line1,
                     p_street_line2,
                     p_street_line3,
                     p_city,
                     p_stat_code,
                     p_natn_code,
                     p_zip,
                     p_phone_area,
                     p_phone_number,
                     p_phone_ext,
                     p_relt_code,
                     p_atyp_code);
    END IF;
```

After the Emergency Contact P_CREATE procedure successfully executes the business logic and DML, the procedure checks to see if the business entity is publishable through an event via the GB_EVENT.F_ENTITY_PUBLISHABLE function. This function takes a baseline/local indicator where all Banner delivered code will be considered baseline using the GB_EVENT.BASELINE_IND constant, and where the second parameter is the business entity name.

The actual process of interacting with the messaging support APIs is contained in the business entities RULES package. This package contains common business rules and code that is shared by the entity.

An example of the GB_EMERGENCY_CONTACT_RULES.P_REGISTER_ENTITY procedure is provided below:

```
gb_event.p_register_entity(
gb_emergency_contact_rules.M_ENTITY_NAME,p_operation_type );
gb_event.p_add_parameter( M_PIDM, p_pidm );
gb_event.p_add_parameter( M_PRIORITY, p_priority );
gb_event.p_add_parameter( M_LAST_NAME, p_last_name );
gb_event.p_add_parameter( M_FIRST_NAME, p_first_name );
gb_event.p_add_parameter( M_MI, p_mi );
gb_event.p_add_parameter( M_STREET_LINE1, p_street_line1 );
gb_event.p_add_parameter( M_STREET_LINE2, p_street_line2 );
gb_event.p_add_parameter( M_STREET_LINE3, p_street_line3 );
gb_event.p_add_parameter( M_CITY, p_city );
gb_event.p_add_parameter( M_STAT_CODE, p_stat_code );
gb_event.p_add_parameter( M_NATN_CODE, p_natn_code );
gb_event.p_add_parameter( M_ZIP, p_zip );
gb_event.p_add_parameter( M_PHONE_AREA, p_phone_area );
gb_event.p_add_parameter( M_PHONE_NUMBER, p_phone_number );
gb_event.p_add_parameter( M_PHONE_EXT, p_phone_ext );
gb_event.p_add_parameter( M_RELT_CODE, p_relt_code );
gb_event.p_add_parameter( M_ATYP_CODE, p_atyp_code );
```

In the above example, the entity named EMERGENCY_CONTACT, defined via the constant GB_EMERGENCY_CONTACT_RULES.M_ENTITY_NAME, is registered with the Banner Event API via the GB_EVENT.P_REGISTER_ENTITY procedure call. The operation that was performed is identified by the P_OPERATION_TYPE parameter. In this case, the value for the operation type is GB_EVENT.CREATE_OPERATION.

After the entity has been registered with messaging support, subsequent GB_EVENT.P_ADD_PARAMETER procedure calls are made to identify the current state of the data. This data will be used by the Banner Event API when the application client program executes the GB_COMMON.P_COMMIT procedure to end this transaction. The data then will be incorporated into an event and published via Oracle Advanced Queuing so an application client can consume it.

Example Code Snippet for P_Delete

```
-- NOTE:  Banner Messaging Support logic is only called after a successful DML operation
    -- Check if events are enabled/licensed for this business entity.
IF (gb_event.f_entity_publishable( gb_messaging.BASELINE_IND,
                        gb_emergency_contact_rules.M_ENTITY_NAME ))
    THEN
        -- Register this business entity with messaging support.
        gb_emergency_contact_rules.p_register_entity(
          gb_event.DELETE_OPERATION, p_pidm, p_priority );
    END IF;
```

After a successful delete operation is performed, the P_DELETE procedure checks to determine if messaging is enabled for this business entity. If it is enabled, the GB_EMERGENCY_CONTACT_RULES.P_REGISTER_ENTITY procedure is called to perform the actual registration process. In this example, the value GB_EVENT.DELETE_OPERATION is passed to identify a delete operation was performed.

Below is the code for the GB_EMERGENCY_CONTACT_RULES.P_REGISTER_ENTITY procedure:

```
gb_event.p_register_entity(
      gb_emergency_contact_rules.M_ENTITY_NAME,
      P_OPERATION_TYPE );
gb_event.p_add_parameter( M_PIDM, p_PiDM );
gb_event.p_add_parameter( M_PRIORITY, p_priority );
```

In the example above, an `EMERGENCY_CONTACT` entity is registered with the Event API, identifying that a Delete operation was performed.

Subsequent `P_ADD_PARAMETER` procedure calls are made to show the current state of the data. For a Delete operation, only primary key values need to be specified.

4 Developing a Banner Business Entity API

Tools you can use to create an API for a local table are provided in the API Toolkit available from the Customer Support Center. This toolkit is not support by SunGard Higher Education, but may be used to implement APIs using the same methodology and architecture as Banner Entity APIs.

The basic steps required to create an API within the Business Entity API layer are:

1. Define a business entity that requires an API.
2. Determine what tables are involved.
3. Generate and compile the DML layer package. Please refer to the *Code Generation* section for more information.
4. Generate the API layer package. Please refer to the *Code Generation* section for more information.
5. Review and edit the API package (see the checklist).
6. Create validation packages as required using validation package generator `gurmpk_val.sql`.
7. Split into separate files using `split.pl`.
8. Compile API packages using the generated build script.
9. Write and run initial test script to test basic functionality.
10. Add business logic code to the API.
11. Modify the create, update and delete procedures, and any other rules, strings, codes or packages that are required.
12. Modify test scripts to include bad data, and test all validation and business logic functionality.
13. Modify applications to call the new API.

Functional and Technical Requirements Template

The following is an example of the Functional and Technical Requirements document for an API. This document is essential for capturing the functional requirements for a Business Entity, and it is used by the developer to add code to the API to implement the business logic required. In this example the API is GB_CERTIFICATIONS.

Use this as a template to document the edits required by your new API.

The following sections list the tables and columns associated with the GB_CERTIFICATIONS API. These are the requirements for data in the columns within the tables that make up the Employee Certifications in Banner. Each section includes field edits and row edits for the table. Additionally included are listings of objects that reference the table and differences in the processing logic between products that need to be addressed before developing a GB_CERTIFICATIONS API.

Example: PPRCERT

Column Name	Description	Data Type	Field Edits
PPRCERT_PIDM	PIDM: Internal ID number.	NUMBER(8)	Not null, PIDM is valid, user has authority.
PPRCERT_CERT_CODE	CERTIFICATION CODE: 4-character code identifying the certification	VARCHAR2(4)	Not null, valid value in PTRCERT
PPRCERT_CERT_DATE	CERTIFICATION DATE: The date the certification was obtained. The Date Req field on the Certification Code Rule Form (PTRCERT) controls whether this field is required.	DATE	
PPRCERT_NEXT_CERT_DATE	NEXT CERTIFICATION DATE: The date on which the employee is scheduled to be re-certified.	DATE	

Column Name	Description	Data Type	Field Edits
PPRCERT_EXPIRE_DATE	CERTIFICATION EXPIRATION DATE: The date on which the certification will expire.	DATE	
PPRCERT_CERT_NO	CERTIFICATION NUMBER: The number of the certification, if applicable.	VARCHAR2 (20)	
PPRCERT_ACTIVITY_DATE	ACTIVITY DATE: Date of last activity (insert or update) on this record.	DATE	Not null.
PPRCERT_LCSV_CODE	LICENSE/ CERTIFICATION STATUS: The status of the Certification.	VARCHAR2(4)	Valid value in PTVLCSV.
PPRCERT_STAT_CODE	STATE CODE: The code identifying the state/ province of the Certification.	VARCHAR2(3)	Valid value in STVSTAT.
PPRCERT_NATN_CODE	NATION CODE: The code identifying the nation/ country of the Certification.	VARCHAR2(5)	Valid value in STVNATN.
PPRCERT_COMMENTS	COMMENTS: Stores any additional comments.	VARCHAR2 (2000)	

Required Row-level Edits

Insert

PPRCERT_CERT_DATE is required if PTRCERT_DATE_REQD_IND = Y for PPRCERT_CERT_CODE.

If PPRCERT_NEXT_CERT_DATE is populated, then it must be >= PPRCERT_CERT_DATE which is now required.

If PPRCERT_EXPIRE_DATE is populated, then it must be >= PPRCERT_CERT_DATE which is now required.

If both are populated, then PPRCERT_EXPIRE_DATE must be >= PPRCERT_NEXT_CERT_DATE.

Update

PPRCERT_CERT_DATE is required if PTRCERT_DATE_REQD_IND = Y for PPRCERT_CERT_CODE.

If PPRCERT_NEXT_CERT_DATE is populated, then it must be >= PPRCERT_CERT_DATE which is now required.

If PPRCERT_EXPIRE_DATE is populated, then it must be >= PPRCERT_CERT_DATE which is now required.

If both are populated, then PPRCERT_EXPIRE_DATE must be >= PPRCERT_NEXT_CERT_DATE.

Delete

Rows in PPRENDS (Endorsements) are also to be deleted based on PIDM, CERT_CODE. There is no constraint so the order of events is not dictated, but child records should be deleted first using F_QUERY_ALL and P_DELETE.

Security Considerations

The user requesting the change must have appropriate authority within HR Security to make changes for the salary level, employee class, etc. for the job of the person associated with the PIDM. Currently handled by form library trigger POQRPLS.P$_DO_SECURITY.

Creating HTML Documentation

One of the modern standards for Java documentation is Javadoc, a program that extracts information from Java source code to give technical users information about how to use a Java program.

This documentation is extracted from the Java source code by a public domain program that identifies text marked by special tags, extracts it, and formats it into html documents with an index. This allows all Java documentation to share a consistent look that developers have become accustomed to. All Java programmers are familiar with the look of this style of documentation, as it is used by Sun to document all the Java APIs.

No such public standard exists for PL/SQL, but the standards developed for Java documentation have been incorporated into a similar system for PL/SQL called PL/Doc.

The code for extracting this information is available freely on the web at www.sourceforge.net.

The API generators create documentation using the column and table comments stored in the database. This information is written into the package body source code using these tags.

Manually Correct the Generated Documentation

The API code generator extracts column comments from the database, attempts to clean up common spelling errors and redundant phrases, and eliminates the uppercase leading string up to the first semicolon in the column comment.

Standards for the Sections

PL/Doc Document Tags

The PL/Doc tags we use are a subset of the Javadoc tags, and provide information that describes what packages contain, what procedures and functions do, what the parameters for a procedure or function call are, and what data is returned by functions.

The tags supported are:

- `@headcom` - header comments for a package
- Procedure comments - which are delimited by strict comment tags `/**` and `*/`
- `@param` - parameter tags giving the name and description of parameters
- `@return` - describes the return value of a function

The PL/SQL doc tags appear in a package spec only; they are not recognized or supported for package bodies. The idea is that we are documenting the public interface to the API, and package specifications represent that public interface. This does not replace, but augments, the normal inline documentation a programmer is expected to put in the code.

Viewing API Documentation

Use any of the open source PLDoc generators to generate the HTML documentation. See www.sourceforge.com, or you may use a PL/SQL tool such as PLSql Developer.

Error Message Format

API error messages need to meet the requirements of a variety of audiences. These include Forms end users, developers, technical support staff and integration partners. Because of this, you need to strike a balance between including simple English messages directed at the functional user, and specific technical information for the technician.

API error messages are initially derived from several sources, including the column comments defined in the database. The API generator creates other messages for conditions such as violation of referential integrity in the database or missing data for mandatory fields as defined in the base table. Be sure to review the column comments in the database before generating an API, making sure that the format of the comment includes an English-like column description separated from the rest of the description by a colon. This English-like description can be extracted by the API generator into the validation messages for that column.

Standards for column comments are documented in the General Technical Reference Manual.

In some cases we want to provide both the Entity Name and the API package name. In those cases we will follow the Entity Name with the API name in parentheses.

Example of Edit	Standard Format	Example
Missing required value	Mandatory <description of column> for the <API package name> API is missing.	*Mandatory PIDM for the pb_honor_award API is missing.*
Invalid value for column	<description of column> for the <API package name> API is not valid.	*Hierarchy Type for the gb_cm_source API is not valid.*
Cannot find row - update	Cannot find record using primary or unique key for the <API package name> API.	*Cannot find record using primary or unique key for the pb_honor_award API.*
Cannot find row - delete	Cannot delete <Entity Name>(<API package name>) because the record does not exist.	*Cannot delete Prior College Major (gb_pcol_major) because the record does not exist.*
Record already exists	Cannot create <Entity Name>(<API package name>) record because the record already exists.	*Cannot create Prior College Concentration (gb_pcol_concentration) because the record already exists.*
Validation of all mandatory indicator columns (e.g., SORDEGR_PRIMARY_IND)	<description of column> is not valid for < Entity Name> (<API package name>). Valid values are 'N' and 'Y'.	*Primary Indicator is not valid for Prior College Degree (gb_pcol_degree). Valid values are 'N' and 'Y'.*

Example of Edit	Standard Format	Example
Required parent record not found	<description of column> is not valid for < Entity Name> (<API package name>). Valid values are 'N' and 'Y'.	*Required Source Background Institution record not found for the gb_pcol_degree API.*
Child record exists	Cannot delete < Entity Name> (<API package name>), because <child Entity name> detail record exists.	*Cannot delete Prior College Degree (gb_pcol_degree), because Prior College Concentration detail record exits.*
Associated Banner product is not installed	<Item name> for <API Entity Name> not allowed when Banner <product name> is not installed.	*STRS Assignment entry for Job Detail not allowed when Banner Payroll is not installed.*

Returning Runtime Data with Error Messages

If you wanted to show a runtime value in an error message, you would define the error message in the `gb_myapi_strings` package and pass it through `g$_nls.get` so that it would get translated correctly:

```
error_tab('INACTIVE_DETAIL_CODE')
     := g$_nls.get ('x', 'SQL','Detail Code is inactive: ');
```

Then add the runtime value to the error message when you validate the value.

An example of the incorrect use of runtime data with error message would be:

```
error_message := gb_myapi_strings.f_get_error
('INACTIVE_DETAIL_CODE')||lv_runtime_variable;
```

Which should result in the message:

```
'Detail Code is inactive: Z'
```

The problem with the example above is that the value has been placed at the end of the string. This is not translatable in all languages - word order can be very different in other languages.

In order to accommodate the correct translation of strings and placement of runtime values, the `g$_nls.get` function allows you to use placeholders in a string. The values passed in are substituted at runtime. The translator only needs to move the placeholders to the correct locations in the translated string. Placeholders have the format *%01%*, *%02%* etc.

So, the correct way to do this is to create an error message with a substitution variable that will hold the runtime data.

```
error_tab('INACTIVE_DETAIL_CODE')
      := g$_nls.get ('x', 'SQL','Detail Code%01% is inactive.',' %01%');
```

Then, wrap the get error message call in a call to `gb_common_strings.f_get_error_with_value`:

```
error_message :=
      gb_common_strings.f_get_error_with_value(
      'GB_MYAPI', 'INACTIVE_DETAIL_CODE', lv_program_variable));
```

Which results in the message:

```
'Detail Code Z is inactive.'
```

assuming at runtime `lv_program_variable` has a value of *Z*. If no value is passed, the placeholder is simply removed from the resulting string.

Standards for Individual Package Sections

Package-Level Documentation

The package documentation should give the reader enough information about the contents of the package to determine if they have chosen the correct package for their project. Describe what business entity this API is implementing, and anything unusual about it. Be sure to enter a functional description of what the API does. It should be technical; functional information will be provided elsewhere. For example:

> *If the API is passed a ROWID, the nametype can be changed. If the ROWID is not passed, . . .*

Be sure to include the technical behavior of the API, including all parameters that are passed to it and what could happen when each is passed. Be sure to indicate which parameters are required.

The text can be formatted using html tags. Use `<BR>` to force new lines, etc.

The text should be in complete sentences, with a subject and a predicate, and should use correct punctuation.

All your text is placed between the `/**` and `* @headcom` markers. You can have as many lines as you like, and you can include simple html tags. Use `<BR>` and `<P>` to break lines and paragraphs. Every line must start with an asterisk and a space.

The first line of this section is the descriptive title of the package. Subsequent paragraphs give more detail about how it works.

This is an example of the package documentation:

```
CREATE OR REPLACE PACKAGE gb_address AS
/**
* Common Business interface for the ADDRESS API (gb_address).
*
* A person or company may have many addresses of different types, and even
* multiple addresses of the same type differentiated by a sequence number.
* The status indicator provides a way to disable an address without deleting it.
* Active addresses have a status indicator of null, inactive ones are set to 'I'.
* Different combinations of fields are mandatory depending on what else
* is defined. For instance, if a ZIP code is passed in, State and Nation
* parameters are required.
* <br>
* There are two types of processing rules implemented in this API.
* The default processing rules require that the time period defined
* by the start and end dates does not overlap the time period of any other
* active address of the
* same type for a given person. This is the default. Null start or end
* dates are considered the beginning and end of time respectively.
* In addition,  a person or company may have multiple inactive addresses
* of the same type with overlapping dates.
* <br>
* The p_seqno INOUT returns the generated sequence number when
* a new address is created. This must be passed in as NULL.
* <br>
* The other type of processing rule is enabled by the p_finance_rules
* parameter. If this is Y, then the API does no date checking at all.
* It allows a person or company to have multiple, active address of
* the same type, regardless of dates. Note that this can lead
* to unusual behavior if a person has addresses
* that were created using both sets of rules.
* In addition, the p_finance_rules parameter allows the calling application
* to specify a value for p_seqno_inout on create, rather than automatically
* generating it.
* <br>
* See special notes in p_update for updating the status indicator of ADDRESS.
*/
```

This is how it might be rendered in html:

Common Business interface for the ADDRESS API (gb_address).

A person or company may have many addresses of different types, and even multiple addresses of the same type differentiated by a sequence number.
The status indicator provides a way to disable an address without deleting it.
Active addresses have a status indicator of null, inactive ones are set to 'I'.
Different combinations of fields are mandatory depending on what else is defined. For instance, if a ZIP code is passed in, State and Nation parameters are required.

There are two types of processing rules implemented in this API.
The default processing rules require that the time period defined by the start and end dates does not overlap the time period of any other active address of the same type for a given person. This is the default. Null start or end dates are considered the beginning and end of time respectively.
In addition, a person or company may have multiple inactive addresses of the same type with overlapping dates.

The p_seqno INOUT returns the generated sequence number when a new address is created. This must be passed in as NULL.

The other type of processing rule is enabled by the p_finance_rules parameter. If this is Y, then the API does no date checking at all.
It allows a person or company to have multiple, active address of the same type, regardless of dates. Note that this can lead to unusual behavior if a person has addresses that were created using both sets of rules.
In addition, the p_finance_rules parameter allows the calling application to specify a value for p_seqno_inout on create, rather than automatically generating it.

See special notes in p_update for updating the status indicator of ADDRESS.

Function/Procedure-Level Documentation

Before each public program unit (in the package spec), add a block of documentation in the following format (note the `/** */` comment start and end markers). This again is delimited by strict comment markers `/**` and `*/` and this block must immediately precede the function or procedure definition. The first lines are always the description of the procedure or function. The procedure and function documentation should give them everything they need to know to call the subprogram correctly, and to understand its affects on the data and any returned values.

Parameter descriptions and return values follow.

The format of the `@param` tag is:

```
*<space>@param<space><parameter name><space><parameter description>
```

It would appear in the document as:

```
/*<newline>
*<space><One line description of the procedure or function>
*<space><Any number of lines of descriptive text, each starting with *
followed by a space>
*<space>@param1<space><parameter name><space><parameter description>
<space>[<datatype>][Required][Key]
*<space>@param2<space><parameter name><space><parameter description>
<space>[<datatype>][Required][Key] etc
*<space><@Return><Return value> (for functions only, not procedures)
*/
```

Note

Objects in brackets ([]) are optional. ■

Where:

- <data type> is:

    - The data type of the column in the base table that will store the actual value (e.g., VARCHAR2(20), DATE, NUMBER(9,2), etc.), or
    - The PL/SQL data type
- The word *Required* indicates that the parameter is mandatory.
- The word *Key* appears for parameters that are parts of the primary key. They are required, and cannot be updated.

```
p_pidm     Internal identification number of the person. NUMBER(8) Required Key
p_atyp_code Type of address associated with the person. VARCHAR2(2) Required Key
```

The following is a completed example:

```
/** Updates a record.
* There are two separate types of address processing rules that may be applied, and which
* may cause unexpected results if misunderstood.
*<p>
* Normal default address rules stipulate that for a given person and address type, no
* two active addresses can have overlapping dates. That is, for a given date, only one
* address has a selected date fall between its start and end dates. Null start date
* means the beginning of time, and null to date means the end of time. Also, every new
* address gets a new sequence number for that type, regardless of status.
* <p>
* In finance, address dates are completely ignored, and a person (vendor) may have up
* to 99 addresses of a particular type.
* <p>
* In order to accommodate both rules, the p_finance_rules parameter is provided. If
* passed a 'Y', then no address checking is done. If an inactive address is
* reactivated by changing status to NULL, default address checking is reapplied
* and the update will fail if another address overlaps the dates. If finance
* rules are applied, no date checking is done.
* <p>
* Because status indicator is part of the unique key, and updates to it are allowed,
* special processing is required. The rowid is used to compare the value passed
* with the value in the database. If they are different, the row identified by
* rowid is considered the target row, and the status indicator passed is used
* to update the database. In this case the key values of p_pidm, p_atyp_code,
* and p_seqno are ignored for purposes of identifying the target row for update.
*
* @param p_pidm Internal identification number of person. NUMBER(8) Required Key
* @param p_atyp_code Type of address associated with person. VARCHAR2(2) Required Key
* @param p_seqno Internal sequence number for addresses of a particular address type.
Number(2) Required Key. NUMBER(2) Required Key
* @param p_from_date Effective start date of address associated with person. DATE
* @param p_to_date Effective end date of address associated with person. DATE
* @param p_street_line1 First line of the address associated with person. VARCHAR2(30)
* @param p_street_line2 Second line of the address associated with person. VARCHAR2(30)
* @param p_street_line3 Third line of the address associated with person. VARCHAR2(30)
* @param p_city City associated with the address of person. VARCHAR2(20) Required
* @param p_stat_code State associated with the address of person. VARCHAR2(3)
* @param p_zip Zip code associated with the address of person. VARCHAR2(10)
* @param p_cnty_code County associated with the address of person. VARCHAR2(5)
* @param p_natn_code Nation/Country associated with the address of person. VARCHAR2(5)
* @param p_status_ind Identifies if address information is inactive. Valid values are
NULL-Active,  I - Inactive.
* @param p_user The Id for the User who create or update the record. VARCHAR2(30)
* @param p_asrc_code Address source code. VARCHAR2(4)
* @param p_delivery_point This field is used to designate the delivery point for mail
as established by the Postal Service. NUMBER(2)
* @param p_correction_digit The Correction Digit field is defined by the Postal Service
and is used to determine the digits used for delivery related to the ZIP code. NUMBER(1)
* @param p_carrier_route The addresses to which a carrier delivers mail. In common usage,
carrier route includes city routes, rural routes, highway contract routes, post office box
sections, and general delivery units. VARCHAR2(4)
* @param p_gst_tax_id Goods and Services Tax Identification of vendor at this address.
VARCHAR2(15)
* @param p_reviewed_ind Reviewed Indicator, Y indicates the address has been reviewed.
VARCHAR2(1)
* @param p_reviewed_user Reviewed User, indicates the user who reviewed the address.
VARCHAR2(30)
* @param p_data_origin Source system that generated the data. VARCHAR2(30)
* @param p_rowid_out Database rowid of record created. VARCHAR(18) Required
* @param p_finance_rules If set to 'Y' then no date checking is done, and sequence number
can be passed in. Default is 'N'.
* @param p_rowid Database rowid of record to update. VARCHAR(18)
*/
  PROCEDURE p_update (
    p_pidm               spraddr.spraddr_pidm%TYPE,
```

etc…

This is how it might be rendered in HTML.

For more information on PL/Doc see http://sourceforge.net/projects/pldoc/.

Rules Package

The rules package should explain the contents and list any special validation procedures that have been added. The procedure documentation for `p_validate` should list the items validated and what values are required to pass validation. You can use html to format simple lists if you want:

```
/**
* Validates all the data in the record.
*
* The following edits are performed. All failed edit messages are appended
* and issued in a single application exception.
* <UL>
* <LI>p_id,p_last_name, p_entity_ind cannot be null.</LI>
* <LI>If p_entity = 'P' then first name cannot be null.</LI>
* <LI>If p_entity = 'C' then first must be null.</LI>
* <LI>p_ntyp_code must exist in gtvntyp.</LI>
* <LI>p_change_ind must be NULL, 'I' or 'N'.</LI>
* <LI>p_entity_ind must be NULL, 'P' or 'C'.</LI>
* </UL>
* @param p_id Identification number used to access person on-line. VARCHAR2(9) Required Key
* @param p_last_name Last name of person. VARCHAR2(60) Required Key
* @param p_first_name First name of person. VARCHAR2(15) Key
* @param p_change_ind I - ID change, N - name change. VARCHAR2(1) Key
* @param p_entity_ind P - person, C - non-person. VARCHAR2(1)
* @param p_ntyp_code Type associated with a person's name. VARCHAR2(4) Key
*/  PROCEDURE p_validate (
```

etc.…

This might be rendered:

Strings Package

Special consideration must be given to documenting API error messages. There must be an accessible explanation of each error and the causes and actions required by the user to address each.

This section describes another utility script that helps format documentation of API error messages. Use this after you have created your API and completed the definition, review and testing of all your API error messages.

Because the documentation is generated from the specification only, we need a way to describe them in the generated API technical documentation. The solution is to place an HTML table containing the strings and descriptions in the package specification.

There is a new utility script in the API Developer Toolkit called `gurmkst_doc.sql` that will generate a text file called `<strings package name>.txt` containing the HTML table with the text of the API error message strings defined in an API strings package.

It assumes that the package was defined according to the Banner API standards, and that it contains a public pl/sql table called `error_tab`, and an `f_get_error` function.

1. Run the script from SQLplus:

```
SQL> @gurmkst_doc
```

Note

When prompted, enter the name of the STRINGS package, *not* the API Package name.

2. The output file will be named `<stringspackage>.txt`.

3. Open the file in your favorite text editor, copy all the text, and paste it into the strings package SPECIFICATION.

4. The text should be pasted into the line following the `CREATE OR REPLACE` command.

5. For each error message, add descriptive text to places marked *Put your description here*. Describe what causes this error, and what the user should do to correct the problem. You may add as much text as needed. Sometimes your description will be a simple rewording of the error message, which is OK.

6. Recompile, then generate the documentation using the PL/Doc plug-in for PLSQL Developer and review.

The parameters are:

```
Strings Package name..: Name of the API strings package
(gb_address_strings).
```

It is very important that the documentation embedded in each API is complete, accurate, and consistent between other APIs. Because this documentation will be extracted from the code and presented to your vendors, it must be in everyday English.

- Every package specification must have a one-line short description, plus descriptive text describing what the package contains. This is delimited with the `@headcom` tag.
- Every public procedure and function must have a one-line short description, a paragraph describing in some detail what the package does and how to use it, and descriptions for each parameter. See the generated code or the PL/Doc description below for details.
- If it is a function it must have an `@return` tag to describe the return value.

Editing the Error Messages

The code generator creates error message for conditions it anticipates, such as missing required data. However, these default messages include in the actual name of the database columns.

The strings package body will contain entries like this:

```
error_tab('MISSING_PIDM')
     := g$_nls.get ('x', 'SQL',
                          'Missing mandatory value for SPRTELE_PIDM');
```

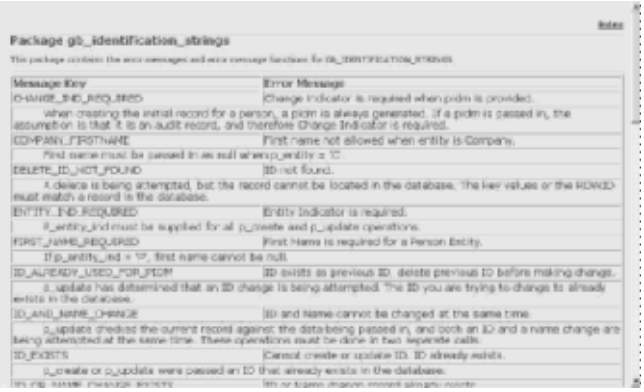
The developer needs to edit these default messages to make them more user-friendly. Messages created by the developer for errors not generated must be checked accuracy and consistency as well.

Sample Error Message Documentation

Sample documentation:

```
CREATE OR REPLACE PACKAGE gb_address_strings AS
/**
* Error messages and error message functions for GB_ADDRESS_STRINGS.
*
* <BR><TABLE border=1 CLASS="STRINGS_TABLE">
* <TR><TD CLASS=LIST_TITLE><B>Message Key</B></TD><TD CLASS=LIST_TITLE><B>
* Error Message</B></TD></TR>
* <TR><TD>ACTIVE_ADDR_EXISTS</TD><TD>Active address already exists for this
*  type</TD></TR>
* <TR><TD colspan=2 style="text-indent: 3em">You can only have one active
*  address of a particular type. Use a different address type, or inactivate
*  the other address of the same type.</TD></TR>
* <TR><TD>ADDRESS_EXISTS</TD><TD>Address of this type exists</TD></TR>
* <TR><TD colspan=2 style="text-indent: 3em">There is already an address
*  record that is active for the time between this FROM and TO date.
*  Select dates that do not overlap the existing address.</TD></TR>
* <TR><TD>ADDRESS_TYPE_REQUIRED</TD><TD>Address type is required</TD></TR>
* <TR><TD colspan=2 style="text-indent: 3em">Address Type is mandatory,
* use a value from the STVATYP table.</TD></TR>
* <TR><TD>CITY_REQUIRED</TD><TD>City is required</TD></TR>
* <TR><TD colspan=2 style="text-indent: 3em">City cannot be null, specify
*  a City.</TD></TR>
* <TR><TD>INVALID_ADDRESS_TYPE</TD><TD>Invalid Address type</TD></TR>
* <TR><TD colspan=2 style="text-indent: 3em">The Address Type is not
*  found in the STVATYP table. Please specify an address found in STVATYP.</TD></TR>
* <TR><TD>INVALID_ADDRESS_TYPE_SEQN</TD><TD>Address Type and Sequence
*  does not exist for this entity</TD></TR>
```

The corresponding html rendering:



Package gb_identification_strings

Message Key	Error Message
CHANGE_IND_REQ_IMED	Change Indicator is required when pidm is provided.
when creating the initial record for a person, a pidm is always generated. If a pidm is passed in, the assumption is that it is an audit record, and therefore Change Indicator is required.	
COMPANY_FIRSTNAME	First name not allowed when entity is Company.
First name must be passed in as null when p_entity = 'C'	
DELETE_ID_NOT_FOUND	ID not found.
A delete is being attempted, but the record cannot be located in the database. The key values or the ROWID must match a record in the database.	
ENTITY_IND_REQUIRED	Entity Indicator is required.
If_entity_ind must be supplied for all p_create and p_update operations.	
FIRST_NAME_REQUIRED	First Name is required for a Person Entity.
If p_entity_ind = 'P', first name cannot be null.	
ID_ALREADY_USED_FOR_PIDM	ID exists as previous ID. delete previous ID before making change.
p_update has determined that an ID change is being attempted. The ID you are trying to change to already exists in the database.	
ID_AND_NAME_CHANGE	ID and Name cannot be changed at the same time.
p_update checked the current record against the data being passed in, and both an ID and a name change are being attempted at the same time. These operations must be done in two seperate calls.	
ID_EXISTS	Cannot create or update ID. ID already exists.
p_create or p_update were passed an ID that already exists in the database.	

Validation Packages

Make sure you look at validation package documentation the same way. Add descriptions to *parameters*, *Data Type*, *Required* and *Key*, if needed. There is often little to say about a validation package or its parameters.

Large Object (LOB) Data Types

LOB data types are used to store very large quantities of data. The API generator can handle Character Large Objects (CLOBs) which are a subtype of LOB, but CLOBs are too large to pass efficiently in a message. If the API is for a table that contains a CLOB, do the following:

1. Edit the `p_register_entity` procedure. Remove the CLOB parameter.
2. Delete the `gb_event.p_add_parameter` call for that CLOB parameter.
3. Edit the call to `p_register_entity` and remove the CLOB from the call.

Keyless Tables

Make every effort to ensure that a table has a Primary Key before attempting to build an API for it. Expect that the development time you'll need to create the API for a keyless table will be much longer than if the table has a proper Primary Key. The Primary Key should have the following characteristics:

- All values are mandatory
- None are updateable

If this is not the case, make sure that there is at least a Unique index, and that none of the fields of the index allow nulls. If they do, refer to *Updateable Keys* later in this chapter.

If the API generator encounters a table with no unique identifier, then it makes ROWID a mandatory parameter and assumes all values are updateable. It also generates an `f_query_one` function that cannot guarantee that only one row will be returned by the cursor.

The `p_delete` procedure and `f_exists` function will each take ROWID as their only parameter.

Updateable Keys

Some current applications break the fundamental RDBMS design rule that keys are never allowed to be updated. If this is the case, there are several workarounds, but they are all manual and none of them are optimal.

The best option when the application requires that a key field be updated is to alter the `p_update` to perform a `p_delete` followed by a `p_create`. For example:

```
EXAMPLE p_update logic:
--
-- Check to see if they sent a rowid, and if the email address or email
-- code are different than what was passed. If this is the case,
-- assume they are updating these, and delete the existing row and
-- add a new one.
    IF ((lv_validation_rec.r_emal_code != p_emal_code
         OR lv_validation_rec.r_email_address != p_email_address)
         AND p_rowid IS NOT NULL) THEN
      p_delete(p_pidm          => lv_validation_rec.r_pidm,
               p_emal_code     => lv_validation_rec.r_emal_code,
               p_email_address => lv_validation_rec.r_email_address,
               p_rowid         => lv_validation_rec.r_internal_record_id);
--
-- Merge the existing data with the bits they passed in.
-- You have to do this here and not before because you need
-- the original values of validation rec, not the overlaid ones,
-- to decide if you should do the delete/create.
--
      p_overlay_validation_rec(lv_validation_rec,
                               p_pidm,
                               p_emal_code,
                               p_email_address,
                               p_status_ind,
                               p_preferred_ind,
                               p_user_id,
                               p_comment,
                               p_disp_web_ind,
                               p_data_origin);

      p_create(p_pidm          => lv_validation_rec.r_pidm,
               p_emal_code     => lv_validation_rec.r_emal_code,
               p_email_address => lv_validation_rec.r_email_address,
               p_status_ind    => lv_validation_rec.r_status_ind,
               p_preferred_ind => lv_validation_rec.r_preferred_ind,
               p_user_id       => lv_validation_rec.r_user_id,
               p_comment       => lv_validation_rec.r_comment,
               p_disp_web_ind  => lv_validation_rec.r_disp_web_ind,
               p_data_origin   => lv_validation_rec.r_data_origin,
               p_rowid_out     => lv_rowid_out);
    ELSE
-- We're not 'updating' any key values, so do a normal update.
..
```

This is the better approach because it is clearer, requires less manual coding, and means that in the future a proper Primary Key can be implemented without immediately requiring updates to the API.

If you can't use this approach for some reason, and you must update the key values directly, then you will have to do much more manual coding:

1. In `p_update`, test to see if ROWID is passed in.
   - If it is, then fetch the row with `f_query_one` using that ROWID, and compare the key values in the database row with the key values passed in. Explicitly call `gb_event.p_add_parameter` for those key values that changed, since they will not be in `p_register_entity`.
   - If it isn't, process the update as usual.
2. In the DML, perform a similar test to see if ROWID is passed in.
   - If it is, add `dml_common.p_add_set_item` calls for each one, so the new value is registered to messaging.
   - If it isn't, no change to the DML is necessary.

Implementing Changes to Existing Business Entity APIs

Changing the signature of an Entity API means adding, removing, modifying, or *changing the order* of any parameters to any public function, procedure, cursor, or type definition.

Note

Since only PL/SQL applications can use positional notation in making database procedure calls, in the order of parameters in a public signature should never be changed. All new parameters must be added at the end of the existing parameter list.

You must maintain *backward compatibility* whenever you add any parameters to a procedure, function, cursor, or you add any fields to a record type. Otherwise, there cannot be any existing or planned integration to the API.

You can achieve backward compatibility by procedure overloading if:

1. All the new parameters are mandatory.

2. The new data signature contains different data type elements than the existing one does.

Note

If procedure overloading is not possible, create a new program unit with a new name, implement as much of the new functionality as possible in it, and delegate the rest of the functionality back to the existing program unit.

Procedure overloading allows you to maintain the required backward compatibility for the current and previous API signature.

The following is an example of procedure overloading:

1. The procedure `PROC1` needs to be modified to accept an additional optional parameter. The existing signature is:

```
PROCEDURE PROC1 (p_parm1 VARCHAR2, p_user_id VARCHAR2, p_internal_record_id VARCHAR2)
IS
BEGIN
--- your original code
END;
```

2. You should create a new procedure with the same name and add the new parameters. Be sure to preserve the order of existing parameters. Copy all the logic in the original procedure to the new one, and add any logic required for the new parameter.

```
PROCEDURE proc1 (p_parm1 VARCHAR2, p_user_id VARCHAR2, p_internal_record_id VARCHAR2,
p_new_parameter VARCHAR2)
IS
BEGIN
-- your original code
-- plus code that handles p_new_parameter.
-- Note that even though p_new_parameter is mandatory, it
-- can be passed as null.
END;
```

3. Call the new procedure from the old one, passing the value required to make the call into the new parameter. This allows existing applications to continue to function without having to supply the new required parameter.

```
PROCEDURE proc1 (p_parm1 VARCHAR2, p_user_id VARCHAR2,
        p_internal_record_id VARCHAR2) IS
lv_local_default VARCHAR2(20):=NULL; --or whatever makes sense as a default value
BEGIN
  P_proc1 ( p_parm1, p_user_id, p_internal_record_id, lv_local_default);
END;
```

In this way, the new procedure exposes the new signature, and the old procedure delegates processing to the new procedure.

Oracle will determine which signature is called based on the number of parameters and their data types.

Note

Do not change the parameter order of the original signature. ■

API Versioning

The PL/SQL API version number is obtained from the `F_API_VERSION` function. This value is used to track changes for the API signatures, and should be a simple increasing integer, e.g., 1, 2, 3.

The API version number will not match the PL/SQL package release number identified in the audit trail:

- The *package release number* is used to track changes to the package when functional changes are made.
- The *API version number* only changes when a procedure or function signature change is necessary. It allows an application to determine dynamically if it can make the call.

The API version number is incremented only when the public signature changes. Other changes to the API may have an effect on the integrated applications, but these changes will not require you to change the version number.

The Banner Entity API Version is designed to track public Create, Retrieve, Update, and Delete (CRUD) operations, so only changes to the following will necessitate incrementing the API version number:

- `p_create`
- any of the `f_query` functions
- `p_update`
- `p_delete`

Although, technically, `p_validate` is public, each destructive CUD operation always calls `p_validate` to ensure that the business rules are enforced, making it a helper routine. Do not change the API version number if only the signature of `p_validate` or other helper routines has changed.

5 Automated API Code Generation

Much of the code to create both the DML and API packages can be automated. In general, the code generators retrieve information from the data dictionary to create PL/SQL packages for each API. This information could be the data type, constraints, table and column comments, etc.

RAW columns are explicitly ignored, as these are generally obsolete. Any column named `<table>_activity_date` is excluded from public signatures, and the API forces the value to be the SYSDATE.

Because the current version of Oracle allows `ALTER TABLE DROP COLUMN x`, these obsolete columns should be evaluated and scheduled for deletion if possible.

The APIs require that any table that does not have a `USER_ID` and `DATA_ORIGIN` column have those added before the generator is run. Both must be VARCHAR2(30) and allowed to be null to minimize impact on clients.

Some existing tables have a `USER` field; do not rename them `USER_ID`.

CLOB Objects

The API code generator does support the CLOB data type. The main difference is that all CLOB columns are defaulted to `EMPTY_CLOB()`, and the `f_isequal` function uses the `dbms_lob.compare` function rather than the = operator.

Currently the CLOB data type is not supported by most messaging environments, so you may need to remove the CLOB column from the list of `dml_common.p_add_set_item` calls in the DML package, depending on the message being defined. You may also need to make the `lv_set_clause` variable in the DML package much larger - since it is a VARCHAR2 data type, it may not be able to hold all the CLOB data. This limitation is being researched.

DML Package Generation

The DML (Data Manipulation Language) layer of the Banner API performs the low-level Update, Insert, and Delete operations on tables. There is one DML package per Banner

table, and no other application, including an API, is allowed to execute Update, Insert, and Delete commands against a table for which an API exists.

The API layer sits on top of the DML layer, and calls it when it needs to perform any of these operations. The API layer is permitted to query tables directly, as are other applications. The API should never directly update, insert or delete records in *any* table. If the API needs to change data in another table for which there is no API, then at least a DML package should be generated for that other table, and the API should call the DML package to perform these operations.

The DML packages can be generated entirely automatically.

How to Generate a DML Package

The file `gurmkpk_dml.sql` is a PL/SQL script which inserts text into `general.guraddl`. After all the text is created, it is spooled to a file containing both the DML package spec and body. Use the `split.pl` script to break the generated file into the spec and body files, or simply do this with a text editor.

The input parameters needed to generate a DML package are:

- Table Name
- Owner Name
- Product Name
- Version Number
- Developer Initials

Other than splitting the file into appropriately named package spec and body files, you do not need to make any other changes to the package.

The DML Layer and FGAC

It is possible that you could have access through Oracle's Fine-Grained Access Control (FGAC) to select records, but not update or delete them. Oracle will not raise an exception in this case, but the DML layer will give you one (or more) of the following error messages:

```
Update Failed. Exactly one row must be updated.
Delete Failed. Exactly one row must be deleted.
```

The DML layer has been designed to raise an exception when the `SQL%ROWCOUNT` does not equal 1 after a delete or update.

API Package Generation

Generating an API package will require more manual intervention than the DML packages, as this is where you will insert the custom business logic.

How to Generate an API Package

The script `gurmkpk_api.sql` can be run from any Oracle account with access to the Data Dictionary. It is designed to allow you to generate multiple packages that share common rules, strings, and codes packages. Input parameters are:

- Table Name - Name of the table to generate an API for
- Owner name - Owner of the tables
- Product name - Banner product (e.g., Student, Payroll)
- Package name - Full package name (e.g., `gb_prior_college`, `sb_catalog`)
- File name - Output file name, which is used to override generated file name
- Version Number - Version Control release number for the Audit Trail
- Developer Initials - Your initials for the Audit Trail

Checklist - After You Run the API Generator

Note

The generated output is in a single file. It is often easier to edit this file and split it into the separate files later.

Ensure that Names are Correct

The generator manipulates the input parameters to create the file names, package names, etc. If the result cannot fit into the size required (16 characters for file names, 30 for Oracle object names), the generator simply truncates the name. Be sure to review the package names, file names, and string constants, and make any needed changes manually.

Update Documentation

Review the documentation section and add descriptive text as needed. Also, the parameter documentation in the package spec is generated in the PL/Doc format for automated html documentation generation. The descriptions of parameters are taken from the column comments in the database. They are filtered somewhat, but you should review all of them

carefully. Search for the comments marked `@param`. These comments will be visible to your messaging partners, so make sure they are correct.

Review Message Strings

In the strings package, make sure all messages are clear, user-friendly, and adhere to standards.

`f_query_all`

The `f_query_all` function, as generated, is identical to the `f_query_one` function. You must determine how the `f_query_all` function should work and change the WHERE clause appropriately. Usually this involves removing one or more parameters from the WHERE clause, or leaving them but making them optional using NVL. For example:

```
WHERE SORDEGR_DEGC_CODE = NVL(p_degc_code, SORDEGR_DEGC_CODE)
```

How you handle this depends on whether the table in question has a parent table, either logically or via declared constraints. If it is does, `f_query_all` should take *only the key fields of the parent table* as its parameters - `f_query_all` returns all child rows that belong to the parent table. For example, if:

- The parent table has a primary key (PK) of *pidm* and *address_type*

  and

- The child table (the one you're creating the API for) has a PK of *pidm*, *address_type*, and *sequence*

  and

- The child table has a foreign key (FK) constraint from its *pidm* and *address_type* to the PK of the parent table

You would make `f_query_all` have only *pidm* and *address_type* as parameters. You would drop the sequence number parameter created for the function by the code generator.

Disabled Constraints

The generator implements validation checks based on all constraints in the database, even disabled ones. The disabled constraints were a way of documenting rules imposed by application programs, but they cannot be implemented as enabled full-database constraints for logistical reasons. The validation checks generated, therefore, are a way of encouraging the consolidation of code that exists in the application programs but should actually be in the API. Verify that all these checks are valid for your Business Entity.

Parent-Child Relationships

Child APIs

A parent-child relationship is normally one in which there is a foreign key constraint from a child table to a parent table which is composed of some subset of the columns of the child table's primary key.

The API generator attempts to detect this situation accurately, but you may have to do some editing. The generator assumes that any foreign key composed of one or more columns that are also part of that table's primary or unique key defines a parent-child relationship.

If the generator thinks it has found a parent-child relationship it will attempt to look at the GURMESG table to see if it can locate an API for the parent table, and generate a call to its `f_exists` function. If it cannot locate an API for the parent, it creates an `f_parent_exists` function and calls it to test for the existence of a parent record.

Parent APIs

If this table is the parent of another, a section of documentation is created that lists the names of the constraints involved. No executable code is generated. Look for the string -- *Do checks for child of <owner>.<tablename>*.

Use this information plus what you know of the application to determine if you need to add any code to, for instance, cascade the delete to the children, or prevent the delete if children exist. To implement a cascade delete, you can call each child API `f_query_all`, and fetch each child record in a loop, and call that child API `p_delete` for each returned row. Otherwise, you can simply fetch once, and if any rows are returned from the child API, raise an application error.

Note

Remove this section of the documentation when you decide what to do. ■

PIDM Checks

Since SPRIDEN has no primary key, there cannot be foreign key constraints against it for PIDM columns. Therefore, the generator simply looks to see if a column is named *PIDM*. If it is, a call is generated to `gb_common.f_pidm_exists` to validate it. Make sure this call is appropriate.

Out Parameters

Some `p_create` procedures will generate a sequence number as well as a ROWID when the record is created. These sequence number parameters will be listed in the normal

order, and will be generated as IN parameters. If you create a record that assigns a sequence number, or something like it, then move the parameter from its generated place in the list to the end of the parameter list, but before the ROWID. Make it an OUT parameter. This way it can be returned to the calling program when the new record is created.

There may be cases where the calling program should determine what the sequence number is. In that event, make it an INOUT parameter, and test to see if a value is passed in before assigning a new one. Remember to use `_OUT` on the end of out parameter names, and `_INOUT` on the end of INOUT parameters.

Splitting the Generated Output

You can now split the file into its separate components. See `split.pl`, or, if you don't have Perl, check the compiled executable version of the script. You can pipe the generated file into the script:

```
perl split.pl<generated_file.sql
```

Or, if you use the exe version of the script:

```
split<generated_file.sql
```

When You Add Code

Once these checks are finished and you have compiled the generated code, test the basic functionality by writing a simple PL/SQL program that calls the `p_create`, `p_update`, and `p_delete` procedures of the API. This ensures that you have a working basic API. Then you can start adding your business logic from the forms and other applications. You may be adding new procedures or functions. These are things to keep in mind as you do:

Make Sure Public Functions Return Y/N Rather Than Boolean

The BOOLEAN data type has a limited usefulness, as it is not supported outside of PL/SQL. If you create functions that return logical TRUE/FALSE values, have them return *Y* or *N* instead. This means they can be used in many other places, such as views and check boxes.

Validation Package Generation

The API generator will create code to validate the data that makes up foreign key relationships in the table. This code will be comprised of calls to validation packages that may not exist yet. These validation packages can be generated by another code generator

script, `gurmkpk_val.sql`. The input parameters are the same as the ones for the DML generator, only the table name is expected to be a validation table. The script generates a mini API for the validation table, containing a public cursor and two functions.

For purposes of the API development process, a validation table is defined as a table that contains exactly three columns, code, description and activity date. Any table with more than these columns should probably not have a validation package created, but rather have a regular API generated for them.

Public signatures of validation table API code:

Name	Type	Description
`<tablename>_c` Example: `stvterm_c`	Cursor	Returns a `<tablename>%ROWTYPE` given a primary key value
`f_code_exists`	Function	Returns *Y* or *N*, depending on value of primary key passed in
`f_get_description`	Function	Returns `<tablename>_desc` field from table

Note

If the table does not have an `<tablename>_desc` column, then `f_get_description` will have to be manually changed to return another appropriate column.

Note

No dynamic SQL is used, so if you want to create a function that is more flexible in what it returns, you may add those functions to this package.

Note

It is not anticipated that validation tables will have full APIs created for them.

6 Creating a Form using APIs

When you are creating new forms, remember that you will be calling APIs to perform insert, update, and delete operations rather than using a great deal of SQL in your form. No code that exists in the APIs should be duplicated in the forms, unless you need to provide that functionality before the commit is performed.

Step 1 Preparation

Make sure you understand the functional and technical specifications for the new form. Get a list of the tables with which the new form must interact. Check to see if any APIs exist for those tables:

1. To determine if a table has an associated API, execute this command:

   ```
   select gb_common.f_get_api_name('BANINST1','<tablename>') from dual;
   ```

   For example:

   ```
   select gb_common.f_get_api_name('BANINST1','SPRADDR') from dual;
   GB_COMMON.F_GET_API_NAME('BANINST1','SPRADDR')
   ------------------------------------------------------------------------
   GB_ADDRESS
   ```

2. To find the API itself, run the following:

   ```
   SELECT REFERENCED_OWNER "Owner",
            SUBSTR(NAME,1,30) "Name",
            REFERENCED_TYPE "Type"
          FROM ALL_DEPENDENCIES
          WHERE OWNER = 'BANINST1'
   AND REFERENCED_NAME = 'DML_TableName'
   ```

   Three rows are typically returned:

   2.1. The DML package name itself

   2.2. The API name

   2.3. The Rules package for that API

Read the html documentation embedded in the API and review the CAST diagrams to get a better understanding of the API.

Validation Packages

Validation packages verify that a code exists in an underlying table and retrieve the corresponding description.

If the form you are creating needs to validate items for the users, search for *XB_TableName*, where *X* represents the product in the standard Banner naming conventions (*G* for *General*, *S* for *Student*, etc.).

Note

All validation packages for Common tables are owned by Banner General.

Step 2 Create Your Form

It may be helpful to clone an existing form that is similar to the one you are creating. Be sure to follow all the User Interface guidelines in the *UI Methodology Handbook* so your form will match the existing Banner forms.

1. Run `gurmkpk_trg.sql` for all tables with which your form will interact. This script is described in detail in another section of this document.

2. Edit the generated code, adding the appropriate names you want to use for each block on your form.

3. If the block interacts with one, and only one table, you must include each of the following triggers, if the corresponding action is permitted in the block:

   3.1. `ON-INSERT`

   3.2. `ON-UPDATE`

   3.3. `ON-DELETE`

   Run the `gurmkpk_trg.sql` script again for each table used in each block. If the table is used in more than one block, it does not need to be run again. You can reuse the code in each block.

4. If the block does not interact with one table, call the needed procedures as appropriate.

   For example, the Address block of the SPAIDEN form includes telephone fields. The Address block was mapped to the SPRADDR table via the Address API (`gb_address`), and the triggers `ON-INSERT`, `ON-UPDATE`, and `ON-DELETE` were created for it. To support the telephone fields, a `POST-INSERT` trigger was added to call the Telephone API (`gb_telephone`):

```
IF :PHONE_NUMBER IS NOT NULL  THEN
 GB_TELEPHONE.P_CREATE( p_PIDM         =>:PIDM,
                        p_TELE_CODE    =>:TELE_CODE,
                        p_PHONE_AREA   =>:PHONE_AREA,
                        p_PHONE_NUMBER =>:PHONE_NUMBER,
                        p_PHONE_EXT    =>:PHONE_EXT,
                        p_STATUS_IND   =>:SPRADDR_STATUS_IND,
                        p_ATYP_CODE    =>:SPRADDR_ATYP_CODE,
                        p_ADDR_SEQNO   =>:SPRADDR_SEQNO,
                        p_PRIMARY_IND  =>'Y',
                        p_DATA_ORIGIN  =>:GLOBAL.DATA_ORIGIN,
                        p_USER_ID      =>USER,
                        p_SEQNO_OUT    =>:HOLD_SEQNO,
                        p_ROWID_OUT    =>lv_rowid);
    END IF ;
```

If this call to `gb_telephone.p_create` will be needed elsewhere on the form, you can create an `INSERT_SPRTELE` trigger with the following:

```
GB_TELEPHONE.P_CREATE( p_PIDM         =>:PIDM,
                       p_TELE_CODE    =>:TELE_CODE,
                       p_PHONE_AREA   =>:PHONE_AREA,
                       p_PHONE_NUMBER =>:PHONE_NUMBER,
                       p_PHONE_EXT    =>:PHONE_EXT,
                       p_STATUS_IND   =>:SPRADDR_STATUS_IND,
                       p_ATYP_CODE    =>:SPRADDR_ATYP_CODE,
                       p_ADDR_SEQNO   =>:SPRADDR_SEQNO,
                       p_PRIMARY_IND  =>'Y',
                       p_DATA_ORIGIN  =>:GLOBAL.DATA_ORIGIN,
                       p_USER_ID      =>USER,
                       p_SEQNO_OUT    =>:HOLD_SEQNO,
                       p_ROWID_OUT    =>lv_rowid);
```

The `POST-INSERT` trigger would have the following:

```
IF :PHONE_NUMBER IS NOT NULL  THEN
    EXECUTE_TRIGGER( 'INSERT_SPRTELE' ) ;
    G$_CHECK_FAILURE ;
   END IF ;
```

5. Write additional functions and procedures as necessary if your form requires additional validations that are not included in the API.

Note

You may want to add these to the API so other forms can take advantage of them. ■

Be sure to trap exceptions:

```
EXCEPTION
  WHEN OTHERS THEN
    G$_DISPLAY_ERR_MSG(SQLERRM);
    RAISE FORM_TRIGGER_FAILURE;
```

6. If you want to include field-level edits, look for a `when-validate-item` trigger in your cloned form. If there is, see if there is an existing Validation Package that performs the edits and use the `f_code_exists` function from that Validation Package.

Let's assume, for example, that you wanted to validate the Nation field on your custom form against the codes that exist in the STVNATN table. Previously you would have relied on the `when-validate-item` trigger to see if the code existed and to display an error message if the users did not enter it. Now you can use the `gb_stvnatn` validation package instead:

```
IF GB_STVNATN.f_code_exists( :SPRADDR_NATN_CODE ) = 'N' THEN
G$_CHECK_VALUE (FALSE,
   G$_NLS.Get('x','FORM','*ERROR* <message> Press list for valid values'),  TRUE);
END IF;
```

Where <message> is the text of the message you want to be presented to the user, e.g., *Invalid Nation Code*. The form-specific string *Press list for valid values* should remain in the form. Strings like these are coded within the form, and should be enclosed in the `G$_NLS.GET` call so they can be translated.

7. Determine if the code should exist in pre- or post- triggers for inserting, updating, or deleting data:

7.1. If data will be inserted, updated or deleted in another table to support the same operations in the base block (or is part of other logic that does), add code in the pre- or post- triggers.

For example, the Address tab of the SPAIDEN form includes address fields from the SPRADDR table and telephone fields from the SPRTELE table. As described previously, the post-insert, post-update, and post-delete triggers should call the `GB_TELEPHONE` API, which would do the actual inserts, updates, and deletes.

7.2. If you use pre-insert and pre-update triggers to perform some of the editing done by the APIs, thereby giving the users immediate feedback, be sure to see if the API has the specific public functions to perform the checks.

For example, one of the row-level edits on the GOBCMUS table specifies that `GOBCMUS_CMSC_CODE` is required if `GOBCMUS_CMSC_OPTION_IND` = *N*. In the `GB_CM_USER_SETUP_RULES` package, you must add the `f_validate_cmsc_cde_option` function so you can use it in the pre-insert trigger for the GOBCMUS block on the GOACMUS form. The function is also used in the `p_validate` procedure of the `GB_CM_USER_SETUP` API, so the code is always executed the same way.

```
BEGIN
IF Gb_Cm_User_Setup_Rules.f_validate_cmsc_cde_option
(:GOBCMUS.GOBCMUS_CMSC_OPTION_IND,:GOBCMUS.GOBCMUS_CMSC_CODE) = 'N' THEN
message(Gb_Cm_User_Setup_Str.f_get_error('MISSING_CMSC_CODE'),ACKNOWLEDGE);
  RAISE_FORM_TRIGGER_FAILURE;
  END IF;
  EXECUTE_TRIGGER( 'UPDATE_ACTIVITY_DATE' ) ;
  G$_CHECK_FAILURE ;
END;
```

If you want to send this constant feedback and still use common code, you'd have to change the `p_validate` procedure in the API to use a number of detailed functions that perform the various edits.

Note

Be careful when you break the `p_validate` procedure into sub-procedures (some of which are available publicly from the API). It may be helpful to determine if the sub-procedures would need to be called independently from each other in multiple places. If they won't be used that way, keep them in the `p_validate` procedure. ■

8. The following are suggestions for queries:

   - Use API calls if you're using a `POST_QUERY` trigger to fetch information in addition to the information obtained by the form's execute query.
   - Add an `F_GET_`*XXX* function (where *XXX* is the name of the item to be retrieved) to the API if you need to use a `POST_QUERY` trigger to display a field from a table that has a complete API. Don't use an `F_QUERY_ONE` function; it would return a ref cursor and would require additional coding changes to the form.

     For example, if you needed to display the current name type for a PIDM from the SPRIDEN table, and that was the only field you needed, you could add the function `F_GET_CURRENT_NAME(PIDM)` to the API.

     This is useful if you need to fetch a single field or a subset of single fields. You can group them into a single, logical query and use it for other forms and clients.

9. Use the messages in the strings packages for all the messages you want to display to the end users.

   For example, if you have a field-level edit to validate the city entered by the user, use `message(GB_ADDRESS_STRINGS.f_get_error('CITY REQUIRED'))`. instead of composing an error message in the form. This method is simpler, more consistent, and will have already been translated if you have installed the international language support.

Note

You should code form-specific error messages in the form itself. ■

10. Whenever you plan to write SQL to perform a validation or calculation, make sure that a function does not already exist that performs that function. If it doesn't, you may want to add it to the APIs or packages. As code snippets are added to the APIs and packages, you will have less need to add program units to the form.

In general, you should look at the API and see if there is anything there that you can use. You can add any extra common functions that make sense to you and call them from the form.

7 API Testing

API Testing Framework

SunGard Higher Education is using an open source PL/SQL test harness called UtPLSQL (http://sourceforge.net/projects/utplsql/). In this environment you create a package that contains one procedure for each of your test cases. Each procedure calls the API and returns either a success or failure message. For each test you call an `ASSERT` function which you pass the test case and your expected result. The `ASSERT` function will execute the test and return true if the results obtained match what you expected.

In addition, our goal is that each test series will eventually create all the prerequisite data required for the tests, and delete this data when complete. The test framework has setup and teardown procedures that you define to handle this.

Building these test cases can be very time consuming, so plan accordingly.

Include at least one test for Delete and Update that passes in the ROWID. This will verify the ROWID logic in each API.

Utility Scripts

There are two utility scripts that can help you make sure that error messages have been appropriately tested:

1. `gurmkpk_obs.sql` - lists the error messages defined in the API strings package that are never used by the API. If an error message is found by this script, either remove it from the strings package or add logic to the API so the message will be returned under a specific condition.

Note

Some messages may be returned that are used by a different API.

2. `gurmkpk_ut.sql` - lists the error messages for an API that were not included in the corresponding Unit Test package in the database. If an error message is found by this script, add the missing test cases to the Unit Test package.

8 API Technical Documentation

HTML Documentation

One of the modern standards for Java documentation is Javadoc, a program that extracts information from Java source code to give technical users information about how to use a Java program.

This documentation is extracted from the Java source code by a public domain program that identifies text marked by special tags, extracts it, and formats it into html documents with an index. This allows all Java documentation to share a consistent look that developers have become accustomed to. All Java programmers are familiar with the look of this style of documentation, as it is used by Sun to document all the Java APIs.

No such public standard exists for PL/SQL, but the standards developed for Java documentation have been incorporated into a similar system for PL/SQL called PL/Doc.

The code for extracting this information is available freely on the web at www.sourceforge.net.

The API generators create documentation using the column and table comments stored in the database. This information is written into the strings package body source code using these tags.

The API technical documentation is designed for technical developers who are writing applications that need to call the APIs.

Package-Level Documentation

The package documentation gives the reader enough information about the contents of the package to determine if they have chosen the correct package for their project. It describes what business entity this API is implementing, and anything unusual about it. It explains the technical behavior of the API, including all parameters that are passed to it and what could happen when each is passed, and it indicates which parameters are required.

The text can be formatted using html tags. Use `<BR>` to force new lines, etc.

The text should be in complete sentences, with a subject and a predicate, and should use correct punctuation.

All your text is placed between the `/**` and `* @headcom` markers. You can have as many lines as you like, and you can include simple html tags. Use `<BR>` and `<P>` to break lines and paragraphs. Every line must start with an asterisk and a space.

The first line of this section is the descriptive title of the package. Subsequent paragraphs give more detail about how it works.

Function/Procedure-Level Documentation

Each public program unit has a block of documentation that contains the description of the procedure or function, and provides a parameter list. The procedure and function documentation gives developers everything they need to know to call the subprogram correctly, and to understand its affects on the data and any returned values.

Rules Packages

The rules package documentation explains the package contents and lists any special validation functionality.

Strings Packages

Strings package documentation explains each error and its causes, and describes the actions required by the user to address each.

9 How to Call Entity APIs

After an API has been created for a table, all applications must be changed to call the API rather than modifying the table directly. You will need to change your custom forms, Web and PL/SQL applications, and batch processes to interact with Banner APIs. The information in this section will also guide you in developing new applications to call entity APIs.

Note that there are special considerations when you are adding a column to a table and that table is used by APIs.

The calling application is responsible for handling these API-generated error messages. The following section describes:

- The format of the error messages
- How they are generated
- Support routines that can be used to format and display them

Handling Errors

Error Message Format

Each message is separated from the others by a delimiter, defined as `GB_COMMON.ERR_DELIMITER ::`. Once the processing is complete, the API calls `RAISE_APPLICATION_ERROR`, and Oracle appends its error information onto the end of the API-generated exception message.

Example error message:

```
ID is required::Last Name is required::ENTITY_IND is required::First Name is required
for a Person Entity::Invalid Name Type Code::Invalid Change Indicator::Invalid Entity
Indicator.::
ORA-06512: at "BANINST1.GB_IDENTIFICATION_RULES", line 128
ORA-06512: at "BANINST1.GB_IDENTIFICATION", line 901
ORA-06512: at "BANINST1.GB_IDENTIFICATION", line 392
ORA-06512: at line 1
```

Support Functions

A set of functions has been provided to aid in parsing the API-generated messages. Keep in mind that the API will report multiple issues in one message, and these should be parsed apart to appear legible to the user.

gb_common Functions

Function	Description
F_ERR_MSG_ADD_DELIM	Given an error message, adds the standard error delimiter (GB_COMMON_STRINGS.ERR_DELIMITER) to the front and back.
F_ERR_MSG_REMOVE_DELIM	This replaces the delimiter with a string of spaces for those applications that need the whole error string in a more readable form. It replaces GB_COMMON_STRINGS.ERR_DELIMITER with GB_COMMON_STRINGS.ERR_SEPARATOR
F_ERR_MSG_REMOVE_DELIM_TBL	Returns a PL/SQL table of VARCHAR2 (GB_COMMON.MSGTAB type) containing one message per record.

gb_common_strings Functions

Function	Description
F_GET_ERROR_WITH_VALUE	Allows you to add up to three program variable values to an error message, while still facilitating the translation of the strings into other languages.
F_APPEND_ERROR	Appends an error message string to an existing string separated by GB_COMMON_STRINGS.ERR_DELIMITER

Implementing User Exits

It is possible for institutions to install custom procedures that will be called from the APIs; they can execute custom validations and other operations without being required to modify Banner baseline code. The call to the institution's code will be compiled into the baseline API package via a table and some extra code in the API package script.

If a record is present in the GUBLAPI table for an API (identified by GUBLAPI_API_NAME = M_ENTITY_NAME in the API), then extra code is compiled into the p_create and p_update procedures of the API to call the procedure defined in

GUBLAPI_LOCAL_API_NAME. All the parameters that were passed to the API procedure are passed to the user's local procedure.

Note

There is no user exit for p_delete.

Forms Application Modifications

As the Banner applications forms have been redesigned, the intent has been to keep the rich user interface and the existing functionality by implementing all the data validation and data manipulation functionality via the APIs, while continuing to execute the validation built into the form at run time. You will want to do the same with your custom forms.

Note

This means that some validations will be performed twice. Not all code will be consolidated into the API.

When evaluating each block for modifications, you will need to identify all of the supporting validation tables used by the items in the block. For each of the validation tables used, a supporting validation package should have been created during the development of the API. This will allow the existing routines that were performing simple validation of the code and retrieval of a corresponding description to be replaced with calls to the validation package. Additionally, all Pre- and Post-DML logic should be evaluated to determine whether it can be included into database API package.

Implementation

Purpose

The current implementation of Banner forms uses the built-in functionality of the Oracle Developer tool kit. This means that all database manipulation is provided and handled automatically. To use the APIs, this database manipulation will have to be programmatically performed. Changing to a database API design also introduces the need to provide improved error messages and handling.

Modified Referenced Objects

There are several types of changes that are required for the forms:

- The first type of change is being handled via several new triggers that are being referenced into the forms via an existing property class. This will be true for most of the forms, but not all of them.

- The second type is to add the needed data manipulation triggers to each existing data block (those that currently support DML processing).
- The third type is to identify all applicable code validation and description lookups, to see which ones can be replaced by calls to the validation packages.
- Another requirement is to review and identify the required data validation logic for inclusion into the API.

All forms currently reference the G$_FORM_CLASS property class. The new triggers (ON-ROLLBACK, POST-FORMS-COMMIT, ON_ROLLBACK_TRG and POST_FORMS_COMMIT_TRG) have been added to make them message-aware. These new triggers will be automatically added to a form simply by generating the respective fmx.

A few forms currently have implemented logic in the POST-FORMS-COMMIT and ON-ROLLBACK triggers. In those forms, the existing triggers must be renamed. If a new trigger is added via the property class and one already exists locally, the local version will still exist and function as designed. These few cases will need to be identified and changed.

New Objects

A new method has been created to handle errors raised by the APIs. Error messages that are propagated back to the forms can be up to 510 characters in length. To be able to display these lengthy messages, a new routine (G$_DISPLAY_ERR_MSG) was created in the GOQRPLS library and a new form (GURERRM) was built. The exception handler within the new form triggers will call the new routine, passing the error message text, which will invoke the new form to display the message text. This new form will display as a modal window on top of the calling form.

When the form encounters an API-specific error and displays the corresponding message in the GURERRM form, p_rollback is called automatically to rollback the transaction and discard any pending messaging data.

Developer Responsibilities

Verify POST-FORMS-COMMIT Trigger

You will need to open the form and verify if the POST-FORMS-COMMIT trigger is referenced via the G$_FORM_CLASS or if it exists locally in the form. If the trigger does exist locally, you will need to rename this trigger to POST_FORMS_COMMIT_TRG.

Verify ON-ROLLBACK Trigger

Another new trigger was also added to the G$_FORM_CLASS. An ON-ROLLBACK trigger was created; you need to verify if this trigger is being referenced or if it exists locally. If the trigger does exist locally, you will need to rename this trigger to ON_ROLLBACK_TRG.

After making these changes, save, close, then reopen the form. This should show the new version of the `ON-ROLLBACK` and `POST-FORMS-COMMIT` triggers.

The `POST-FORMS-COMMIT` trigger is needed to execute the `gb_common.P_COMMIT`, and the `ON-ROLLBACK` trigger is needed to execute the `gb_common.P_ROLLBACK`.

Trigger Generation

The data manipulation modifications (DML) that need to be added to all database table blocks will require several new triggers to be added. To reduce the amount of time and effort to produce these new triggers, you can run the `gurmkpk_trg.sql` script via SQL*Plus to generate most of the code for each of them.

Note

This process also assumes that the corresponding database API has been already created. ■

The code that is generated will be based on the current table structure, and any primary keys or unique keys on that table. The code will have to be evaluated to ensure its authenticity and accuracy. You should verify the following:

1. The `block.column` name matches the values in the form.
2. The name of the API is correct.
3. All of the table columns exist in the block. If there are columns on the table that are not present in the data block, those entries should be deleted from the generated code.

New Triggers

The new triggers required are:

- `ON-INSERT`
- `ON-UPDATE`
- `ON-DELETE`

If the block does not currently support any of these DML functions, the corresponding new trigger does not need to be added. For example, if the block does not support the Delete function, `ON-DELETE` is not required.

These new triggers replace the default data manipulation logic provided by the Oracle Developer tool kit. When the Oracle Developer normal processing finds one of these new triggers, it transfers control to the new trigger and allows it to process the data manipulation logic.

The default design is to use *named parameter notation*. This means that only the columns which are present in the block are passed: all others are not accounted for and don't have to be. All required columns will be passed to the Update and Insert routines. For the Delete

routine, only the minimum required columns to uniquely identify the appropriate record must be passed.

If the block commit logic is attempting to update other tables, review the existing APIs to see if these updates can be implemented via another API call (the preferred method). Typically this means that a null value should be passed for the ROWID parameter.

Pre- and Post-DML Triggers

You will need to evaluate any and all of these existing triggers to see if you can move this logic to the database API. In many cases, the logic can be moved and thus removed from the form.

Audit Trails

You will need to add the appropriate audit trail explaining these changes. The `gurmkpk_trg.sql` script will create a default audit trail that should be the starting point for the complete entry that is to be updated and completed by the developer.

The end goal is that there will be an ever-increasing library of APIs that the developer can use. These routines will need to be documented and made available to all developers.

Code Consolidation Example

The following is an example to help clarify what code should be removed from a form and what code should be changed or left alone.

First, we assume that during the development of the API for the SOBSBGI table, this form and others were reviewed and validations like this were identified. In addition, if the table has foreign key constraints defined, then the API generator will also create code to validate this column in the table.

Generated code in the validation package of the API for SOBSBGI:

```
IF (p_stat_code IS NOT NULL AND sb_stvstat.f_code_exists(p_stat_code) = 'N') THEN
         error_message := error_message
                  || gb_common.F_ERR_MSG_ADD_DELIM(
            gb_source_background_str.f_get_error('INVALID_STAT_CODE'));
END IF;
```

The developer creating the API will not be able compile the API until the Validation Package for the STVSTAT table is created, which will provide the `f_code_exists` function *and* the `f_get_description` function.

So, the API is done, the Validation Package is done, and attention turns to Forms:

The SOAPCOL form provides a `POST-CHANGE` trigger on `SOBSBGI_STATE_CODE` field:

```
BEGIN
   G$_CHECK_QUERY_MODE ;
--
   DECLARE
      CURSOR PTI_CURSOR IS
         SELECT STVSTAT.STVSTAT_DESC
         FROM   STVSTAT
         WHERE  :SOBSBGI_STAT_CODE = STVSTAT.STVSTAT_CODE ;
   BEGIN
      OPEN PTI_CURSOR ;
      FETCH PTI_CURSOR INTO :STVSTAT_DESC ;
      G$_CHECK_VALUE (PTI_CURSOR%NOTFOUND, NULL, TRUE);
   END ;
--
   :GLOBAL.QUERY_MODE := 'O' ;
EXCEPTION
   WHEN FORM_TRIGGER_FAILURE THEN
      :GLOBAL.QUERY_MODE := 'O' ;
      RAISE FORM_TRIGGER_FAILURE ;
END ;
```

This form must keep this trigger so that users will be notified that they have entered an invalid State code when they try to leave the field.

The developer then changes the trigger to call the Validation Package for STVSTAT, *the same one that the sobsbgi API calls*. Since we need description in the form, we call the `f_get_description` function rather than `f_exists`:

```
BEGIN
   G$_CHECK_QUERY_MODE ;
--
   DECLARE
   lv_desc VARCHAR2(30);
   BEGIN
     lv_desc := gb_stvstat.f_get_description(:SOBSBGI_STAT_CODE);
     IF (lv_desc IS NULL) THEN
       :GLOBAL.QUERY_MODE := 'O' ;
       RAISE FORM_TRIGGER_FAILURE ;
     END IF;
     :STVSTAT_DESC := lv_desc;
   END ;
   :GLOBAL.QUERY_MODE := 'O' ;
END ;
```

Now, we also found in our original analysis, that there is a `POST-DELETE` trigger on the block that calls a Form trigger `CHECK_FINAID`. That trigger calls another trigger called `FINAID_RECON`. The `FINAID_RECON` trigger has code that deals with updating stuff for Financial Aid, and all that logic was moved to the API.

Therefore, the `POST-DELETE`, `CHECK_FINAID`, and `FINAID_RECON` triggers all can be deleted from the form. The API has this logic, so it does not need to be in the form.

Web and PL/SQL Application Modifications

Since Web applications are already written in PL/SQL, they can be grouped together with any other PL/SQL packages/procedures/functions currently in use for the purposes of this document.

The overall requirements for converting existing packages to call the API are the same as for any other client application:

- Call the API rather than directly accessing database tables. Never do direct insert/update/delete on a database table that has an API defined for it. Instead, convert Insert/Update/Delete operations into calls to the appropriate API. Convert Select operations to calls to the appropriate API Query function and, if needed, define new ones to be added to the API.
- Handle API-generated exceptions. Be aware that the API will generate exceptions to validation failures, and the format of the message may require special handling to produce a useful message for the user.
- Perform Commit and Rollback operations using the API. The `gb_common.p_commit` and `gb_common.p_rollback` procedures must be used in place of explicit COMMIT or ROLLBACK commands. Never rely on Oracle to commit by default when a program exits. For web applications, the transaction ends when the procedure or function exits. Therefore, any programs that relied on this default behavior must now include calls to `gb_common.p_commit` or `gb_common.p_rollback` before returning control to the user.

  For example:

```
EXCEPTION
      WHEN others THEN
        gb_common.p_rollback;
       if SQLCODE = gb_common_strings.err_code then
         twbkfrmt.p_storeapimessages(SQLERRM);
         p_selectmtypupdate (mtyp);
       end if;
     WHEN OTHERS THEN
        gb_common.p_rollback;
        RAISE;
```

- Perform field-level validation using the API. The API performs record validation. It takes an entire input record and validates all the fields or combinations of fields. Many PL/SQL applications, particularly Web applications, need to perform field validation so they can report problems with specific fields to the user. This is still done, but be aware that the rules imposed by the application field validation logic must match those enforced by the API validation.
- Become familiar with cursor variables. Few current PL/SQL procedures make use of cursor variables at present. The API query operations return cursor variables, so you will need to make yourself familiar with them by reading the appropriate Oracle documentation.

Examples

BWGKOADR does the following insert into SPRTELE:

```
 INSERT INTO sprtele
             (
                sprtele_pidm,
                sprtele_tele_code,
                sprtele_seqno,
                sprtele_activity_date,
                sprtele_phone_area,
                sprtele_phone_number,
                sprtele_phone_ext,
                sprtele_intl_access,
                sprtele_atyp_code,
                sprtele_addr_seqno,
                sprtele_primary_ind,
                sprtele_unlist_ind
             )
      VALUES (
         pidm,
         teles (i),
         pseqno,
         SYSDATE,
         areas (i),
         phones (i),
         exts (i),
         accss (i),
         atyp,
         aseqno,
         NULL,
         unl_tab (i));
```

This would be converted to call the API:

```
DECLARE
lv_rowid gb_common.internal_record_id_type;
lv_seqno PLS_INTEGER;
BEGIN
gb_telephone.p_create (
   p_pidm         => pidm,
   p_tele_code    => teles (i),
   p_phone_area   => areas (i),
   p_phone_number => phones (i),
   p_phone_ext    => exts (i),
   p_atyp_code    => atyp,
   p_addr_seqno   => aseqno,
   p_PRIMARY_IND  => null,
   p_UNLIST_IND   => unl_tab (i),
   p_INTL_ACCESS  => accss (i),
   p_data_origin  => gb_common.DATA_ORIGIN,
   p_user_id      => gb_common.f_sct_user,
   p_seqno_out    => lv_seqno,
   p_rowid_out    => lv_rowid
  );
gb_common.p_commit;
END;
```

Note

`p_rowid` and `p_seqno` are OUT parameters, and are therefore mandatory. Determine this by looking at the public signature for the procedure: ■

```
PROCEDURE P_CREATE(
        p_PIDM                  SPRTELE.SPRTELE_PIDM%TYPE,
        p_TELE_CODE             SPRTELE.SPRTELE_TELE_CODE%TYPE,
        p_PHONE_AREA            SPRTELE.SPRTELE_PHONE_AREA%TYPE,
        p_PHONE_NUMBER          SPRTELE.SPRTELE_PHONE_NUMBER%TYPE,
        p_PHONE_EXT             SPRTELE.SPRTELE_PHONE_EXT%TYPE,
        p_STATUS_IND            SPRTELE.SPRTELE_STATUS_IND%TYPE,
        p_ATYP_CODE             SPRTELE.SPRTELE_ATYP_CODE%TYPE,
        p_ADDR_SEQNO            SPRTELE.SPRTELE_ADDR_SEQNO%TYPE,
        p_PRIMARY_IND           SPRTELE.SPRTELE_PRIMARY_IND%TYPE,
        p_UNLIST_IND            SPRTELE.SPRTELE_UNLIST_IND%TYPE,
        p_COMMENT               SPRTELE.SPRTELE_COMMENT%TYPE,
        p_INTL_ACCESS           SPRTELE.SPRTELE_INTL_ACCESS%TYPE,
        p_SEQNO                 OUT SPRTELE.SPRTELE_SEQNO%TYPE,
         p_ROWID                   OUT VARCHAR2) IS
```

Using positional notation, you would only have to pass the parameters that are required, plus the ones you want to update. Not all API signatures require all parameters to be passed.

Also, note that the explicit commit was changed to `gb_common.p_commit;`.

Batch Application Modifications

Batch programs, like any other application program, must be changed to call APIs to perform create, retrieve, update, and delete operations instead of doing direct Data Manipulation Language (DML) operations on the base tables. In many cases, this requires restructuring the logic in the program.

This section describes the changes that will be needed to have them call the Banner API.

Batch Program Process

The following is a simplified summary of the steps required and issues that may need to be addressed.

Functional Design Phase

- Identify any batch programs that need to be changed in the API design phase.
- Review the program function and structure, and determine the level of effort required to make the changes.
- Determine if this program could/should be rewritten entirely, perhaps as a PL/SQL package with a simple C wrapper that calls it.

Technical Design Phase

- Determine how much of the code can be removed and put into database stored procedures.
- Of the remaining code, identify how much of it should be restructured to call the API and handle API-generated exceptions.
- If the program is simply validating data for later processing, remember that the API exposes its validation routines. These should be called to verify the data before the main program is run to perform the insert/update.
- Determine at what points the API will be called, and how the call will be made (regular cursor fetch and EXEC SQL procedure call loop, or anonymous PL/SQL block). Consider what should happen in the event of an API exception. The goal is to structure the program so you can report API errors to the user and then continue processing records.
- Ascertain if this is a program that updates data records after each are processed. Can it commit data as it goes along, record which records were processed, and be restarted at any time? If so, there are special considerations with calling the API and handling errors that are described in more detail later in this section.
- Determine if this program commits only at the end, or on program exit. Is everything rolled back when an Oracle error is encountered? Decide what should happen when an API error is encountered.

Coding

- The API will generate validation error messages. The program has to handle these appropriately in addition to handling Oracle errors. The API exceptions will be thrown before any database changes are attempted.
- Anonymous PL/SQL blocks are a very handy tool to use in C programs. However, if an exception is raised, the C program can only find out about it when the block exits. So, while they seem to be an elegant way to consolidate iterative processes, only use them in cases where current program logic requires a complete rollback and exit when an API exception occurs (same as an Oracle error).
- If you need to report API exceptions and then continue, create a normal cursor loop (see examples) with calls to the API in the loop, and `WHENEVER SQLERROR DO <function>` statements to handle errors.
- Make sure every program commit and exit point (for both normal and error exits), contains a call to `gb_common.p_commit` or `gb_common.p_rollback`. This is required for messaging support.
- If the program reports API errors and continues processing, modify the control report to show how many records were processed correctly and how many records had exceptions.

- Never call commit or rollback explicitly, and never rely on the default Oracle behavior of commit on exit to get your data committed. This has been the default behavior in many C programs, so make sure to add this call before any exit2os.
- Review other non-API requirements for batch programs and make sure this one complies. It must:
  - Conform to coding standards.
  - Be able to be run from job submission.
  - Be able to be restarted without causing damage.
  - Have an Audit/Update option.

Changes to the POSTORA Macro

The existing POSTORA error check will remain unchanged. C programs converted to call the APIs will by default exit on any API error or any Oracle error. For the C program to handle API errors differently, a new macro `POSTORA_API` will be provided to return the API generated error messages in a struct. The struct will contain three pointers:

1. A pointer to the unmodified API error returned from the raised exception.
2. A pointer to the count representing the total number of API errors encountered in the last call.
3. A pointer to an array of strings, representing the individual parsed error messages with delimiters removed.

Therefore, the only thing you have to do differently if you need to process API generated exceptions differently than Oracle generated exceptions is to test to see if `POSTORA_API` is not null, and then examine the structure to find the error messages. Display or log them, then take any error recovery action required.

If `POSTORA_API` finds an Oracle-generated error, it will rollback and exit as it does now. If it finds an API-generated error, it will first attempt to loop through and print all of the API error messages, then it will rollback and exit. There will be times when an API error shouldn't cause a failure and will need to be coded specifically to handle those situations.

When calling the API, the `POSTORA_API` call should be coded to test for a return value. For example:

```
if ( POSTORA_API )
  {
    /* This will print the entire error string */
    printf ("The API generated error string is %s\n", xxxx);

    /* This simple loop will print out all the API generated error messages individually */
    for (j=0; j< xxxx; j++)
      printf("Message %d = %s\n",j, xxxx[j]);
  }
```

Call the API to Perform Inserts, Updates and Deletes

A logical unit of work is a set of database transactions that are either committed as a whole or rolled back as a whole. It is the work done between commits or rollbacks.

Statements that perform a select and insert in one operation may use an anonymous PL/SQL block with a loop to perform the same work. Remember, this statement is all or nothing. If there are any API exceptions or Oracle errors, the logic must call `gb_common.p_rollback`.

Here are some examples of ways to convert a program to call the API. We start with what is probably the most common construct from the converted RPT programs, a function containing a cursor, a call to the report function, and a body function that performs the processing.

There are several ways to change this:

1. Replace the insert with an API call
2. Restructure it all into one C function
3. Use an anonymous PL/SQL block.

Examples

Replace an Insert with an API Call

The original function containing driving cursor:

```
static int sel_recr(int mode)
{
#ifdef SCT_DEBUG
    printf("sel_recr\n");
    fflush(stdout);
#endif

  EXEC SQL DECLARE cursor_029 CURSOR FOR
       SELECT RCTRECR_PIDM,
              RCTRECR_TERM_CODE,
              RCTRECR_DEPT_CODE,
              RCTRECR_LEVL_CODE,
              RCTRECR_DEGC_CODE,
              RCTRECR_MAJR_CODE,
              RCTRECR_RSTA_CODE
         FROM RCTRECR
        WHERE RCTRECR_PIDM IN
                 (SELECT ROTIDEN_PIDM
                    FROM ROTIDEN
                   WHERE ROTIDEN_INFC_CODE = :tape_type
                     AND ROTIDEN_STAT_IND  = :rec_type_ind
                     AND NVL(ROTIDEN_DELETE_FLAG,'N') = 'N'
                     AND ROTIDEN_AIDY_CODE = :aid_year)
          AND RCTRECR_INFC_CODE = :tape_type
          AND RCTRECR_AIDY_CODE = :aid_year;
… OPEN … FETCH… ETC…
```

The original report function call to drive the body function:

```
report(sel_recr,recrbody,recrhead,NULL);
```

The original body function that processes data from the cursor:

```
static void recrbody(void)
{
#ifdef SCT_DEBUG
    printf("recrbody\n");
    fflush(stdout);
#endif
  EXEC SQL
        INSERT INTO SRBRECR
               (SRBRECR_PIDM, SRBRECR_TERM_CODE,
                SRBRECR_DEPT_CODE, SRBRECR_LEVL_CODE, SRBRECR_DEGC_CODE,
                SRBRECR_MAJR_CODE, SRBRECR_RSTA_CODE, SRBRECR_ADD_DATE,
                SRBRECR_ACTIVITY_DATE, SRBRECR_TERM_CODE_CTLG_1)
        VALUES (:tmp_pidm,
                :tmp_term_code,
                :tmp_dept_code,
                :tmp_levl_code,
                :tmp_degc_code,
                :tmp_majr_code,
                :tmp_rsta_code,
                 SYSDATE,
                 SYSDATE,
                :tmp_term_code);
  POSTORA;
}
```

To replace the insert with API call (Option 1), replace the EXEC SQL Insert part of the body function with API call:

```
static void recrbody(void)
{
#ifdef SCT_DEBUG
    printf("recrbody\n");
    fflush(stdout);
#endif
  EXEC SQL EXECUTE
          gb_recruit.p_create
                (:tmp_pidm,
                :tmp_term_code,
                :tmp_dept_code,
                :tmp_levl_code,
                :tmp_degc_code,
                :tmp_majr_code,
                :tmp_rsta_code,
                 SYSDATE,
                 SYSDATE,
                :tmp_term_code);
if ( POSTORA_API )
  {
   …processes errors here …
  }

  END-EXEC;
}
```

Restructure It All into One C Function

This example shows how you would replace the report function and cursor from a converted C program to a single function. This is also the structure you would create if you were coding a new program.

The error handling code is removed in the example for simplicity. More detailed examples of error handling follow.

```
static void ins_spriden_n(void)
{
#ifdef SCT_DEBUG
   printf("ins_spriden_n\n");
   fflush(stdout);
#endif
   EXEC SQL DECLARE  sel_rotiden CURSOR FOR
      SELECT rotiden_pidm, rotiden_id, rotiden_last_name, rotiden_first_name,
             rotiden_mi, rotiden_entity_ind
        FROM rotiden
       WHERE rotiden_stat_ind = :rec_type_ind
         AND NVL (rotiden_delete_flag, 'N') = 'N'
         AND rotiden_infc_code = :tape_type
         AND rotiden_aidy_code = :aid_year;
      POSTORA;
 EXEC SQL OPEN sel_rotiden;
for (;;) {
    EXEC SQL FETCH sel_rotiden INTO
     rotiden_pidm:Ind_01,
     rotiden_id:Ind_02,
     rotiden_last_name:Ind_03,
     rotiden_first_name:Ind_04,
     rotiden_mi:Ind_05,
     rotiden_entity_ind:Ind_06;
POSTORA;
   EXEC SQL EXECUTE
      BEGIN
         gb_identification.p_create_id
         (:rotiden_id,          /*  P_ID          VARCHAR2   IN              */
          :rotiden_last_name,   /*  P_LAST_NAME   VARCHAR2   IN              */
          :rotiden_first_name,  /*  P_FIRST_NAME  VARCHAR2   IN              */
          :rotiden_mi,          /*  P_MI          VARCHAR2   IN              */
          NULL,                 /*  P_CHANGE_IND  VARCHAR2   IN              */
          :rotiden_entity_ind,  /*  P_ENTITY_IND  VARCHAR2   IN              */
          USER,                 /*  P_USER        VARCHAR2   IN     DEFAULT  */
          :rptname,             /*  P_ORIGIN      VARCHAR2   IN              */
          NULL,                 /*  P_NTYP_CODE   VARCHAR2   IN              */
          :rotiden_pidm,        /*  P_PIDM        NUMBER(8)  IN/OUT          */
          :rotiden_rowid);      /*  P_ROWID       VARCHAR2   OUT             */
      END;
  if (POSTORA_API) {
/* error checking would go here */
}
   END-EXEC;
     printf("p_create_id done, for pidm %s, new rowid=%s\n",rotpers_pidm,
rotpers_rowid);
```

Anonymous PL/SQL Blocks

An anonymous PL/SQL block is a section of code passed to the database by the C program for execution. There are many performance advantages of using them However, the program can only handle exceptions raised during the execution of the block after the block exits. This may be a problem if the block processes multiple API calls - the C program cannot report the error and then continue executing the block.

Do not use anonymous PL/SQL blocks when you require the program to deal with individual record exceptions and then continue. Once the block exits with an API error, you cannot resume processing with the next available record.

Do use them when you can perform the operations as a single transaction that must complete successfully or rollback entirely, and when a single API or Oracle generated error will force a rollback and exit.

The cursor and body functions can be combined with a call to the appropriate API in a single function. It may be something like:

```
static void insert_recruit(void)
{
EXEC SQL EXECUTE
DECLARE
   CURSOR select_rctrecr_c IS

/* Cursor to select temp recruit records */

        SELECT RCTRECR_PIDM,
               RCTRECR_TERM_CODE,
               RCTRECR_DEPT_CODE,
               RCTRECR_LEVL_CODE,
               RCTRECR_DEGC_CODE,
               RCTRECR_MAJR_CODE,
               RCTRECR_RSTA_CODE
          FROM RCTRECR
         WHERE RCTRECR_PIDM IN
                  (SELECT ROTIDEN_PIDM
                     FROM ROTIDEN
                    WHERE ROTIDEN_INFC_CODE = :tape_type
                      AND ROTIDEN_STAT_IND  = :rec_type_ind
                      AND NVL(ROTIDEN_DELETE_FLAG,'N') = 'N'
                      AND ROTIDEN_AIDY_CODE = :aid_year)
           AND RCTRECR_INFC_CODE = :tape_type
           AND RCTRECR_AIDY_CODE = :aid_year;
        BEGIN
         FOR r_rec IN select_rctrecr_c LOOP
   /*
    Call fictitious new API procedure
   */
          gb__recruiting.p_create_recruit
                  (r_rec.RCTRECR_PIDM,
                   r_rec.RCTRECR_TERM_CODE,
                   r_rec.RCTRECR_DEPT_CODE,
                   r_rec.RCTRECR_LEVL_CODE,
                   r_rec.RCTRECR_DEGC_CODE,
                   r_rec.RCTRECR_MAJR_CODE,
                   r_rec.RCTRECR_RSTA_CODE,
                   SYSDATE,
                   SYSDATE);
         END LOOP;
       END;
   END-EXEC;
  POSTORA; -- This will catch any error, API or Oracle, and exit.
}
```

The cursor may be defined inline, at the function level as here, or at the program level in the event it can be used by other functions.

Note

With anonymous PL/SQL blocks, you have to plan carefully to address the issues of when to commit/rollback, and what to do when the block throws an exception.

Examples

Insert into <destination> AS select <columns,…> from <source table>;

Sometimes the program performs a direct load of data from one table to another.

```
  EXEC SQL
   INSERT INTO spriden
            (spriden_pidm, spriden_id, spriden_last_name, spriden_first_name,
             spriden_mi, spriden_entity_ind, spriden_activity_date,
             spriden_user, spriden_origin)
   SELECT rotiden_pidm, rotiden_id, rotiden_last_name, rotiden_first_name,
          rotiden_mi, rotiden_entity_ind, SYSDATE, USER, :rptname
     FROM rotiden
    WHERE rotiden_stat_ind = :rec_type_ind
      AND NVL (rotiden_delete_flag, 'N') = 'N'
      AND rotiden_infc_code = :tape_type
      AND rotiden_aidy_code = :aid_year;
  POSTORA;
}
```

Replace with Anonymous PL/SQL block

API Example using inline cursor FOR loop:

```
EXEC SQL EXECUTE
   BEGIN
      FOR r_rec IN
          SELECT *
            FROM rotiden
           WHERE ROTIDEN_STAT_IND = :rec_type_ind
             AND NVL(ROTIDEN_DELETE_FLAG,'N') = 'N'
             AND ROTIDEN_INFC_CODE = :tape_type
             AND ROTIDEN_AIDY_CODE = :aid_year LOOP

           gb_identification.p_create_id(
                       r_rec.ROTIDEN_ID,
                       r_rec.ROTIDEN_LAST_NAME,
                       r_rec.ROTIDEN_FIRST_NAME,
                       r_rec.ROTIDEN_MI,
                       NULL,
                       r_rec.ROTIDEN_ENTITY_IND,
                       USER,
                       :rptname,
                       NULL,
                       r_rec.ROTIDEN_PIDM);

      END LOOP;
   END;
   END-EXEC;
          POSTORA;
```

Generic Example

This example shows the use of the following:

- `PRAGMA INIT` to define and catch API exceptions within the block.
- Count how many records were processed correctly and how many had API exceptions.
- Inline rather than separate cursor definition.
- Show how PL/SQL can save data to program variables.

- Commit every *n* records.

```
void create_recruit() {
int total_records;  -- program variable to hold total record count
int total_errs;     -- Number of API errors encountered
EXEC SQL EXECUTE
DECLARE
   commit_count   PLS_INTEGER := 20; -- How many records for each commit
   record_count   PLS_INTEGER := 0; -- Total count of records processed
   err_count      PLS_INTEGER := 0; -- Number of API errors
   api_error      EXCEPTION; -- To catch API generated exceptions
   PRAGMA EXCEPTION_INIT (api_error, -20100);
BEGIN
   FOR r_rec IN
      (SELECT rctrecr_pidm, -- No need to declare cursor separately
              rctrecr_term_code,
              rctrecr_dept_code,
              rctrecr_levl_code,
              rctrecr_degc_code,
              rctrecr_majr_code,
              rctrecr_rsta_code
         FROM rctrecr
        WHERE rctrecr_pidm IN (
                 SELECT rotiden_pidm
                   FROM rotiden
                  WHERE rotiden_infc_code = :tape_type
                    AND rotiden_stat_ind = :rec_type_ind
                    AND NVL (rotiden_delete_flag, 'N') = 'N'
                    AND rotiden_aidy_code = :aid_year)
          AND rctrecr_infc_code = :tape_type
          AND rctrecr_aidy_code = :aid_year) LOOP
      /*
       Call fictitious new API procedure
      */
      BEGIN --Note the extra BEGIN
         gb__recruiting.p_create_recruit (r_rec.rctrecr_pidm,
                                          r_rec.rctrecr_term_code,
                                          r_rec.rctrecr_dept_code,
                                          r_rec.rctrecr_levl_code,
                                          r_rec.rctrecr_degc_code,
                                          r_rec.rctrecr_majr_code,
                                          r_rec.rctrecr_rsta_code,
                                          SYSDATE,
                                          SYSDATE);
         record_count := record_count + 1;                        -- success
                                       -- If you had updates to do, do them here
      EXCEPTION
         WHEN api_error THEN
            err_count := err_count + 1; -- You only get here if API error
            /* If you want to do anything else to flag the error, do it here */
      END; -- And END, to control scope of the exception
           --At this point, even though an exception occurred,
           --you don't exit the loop, you keep processing

      total_records := total_records + i; -- Add to total

      IF (MOD (record_count, commit_count) = 0) THEN -- Commit every 20
         gb_common.p_commit; -- do the message publish and commit.
      END IF;
   END LOOP;
   :total_records := record_count;
   :total_errs := err_count;
END; /* anonymous block */
END-EXEC;
printf ("%d records processed, %d records skipped \n",
        total_records, total_errs);
}
/* This code could be encapsulated in another function */
```

Remember, you can define BEGIN-END blocks anywhere you want, and any such block can have an exception handler.

The API generates an application error with an error code of -20100 so the pragma statement associates the error code to a named exception.

Committing Records Automatically in Logically-Sized Groups

Most batch programs commit only after all records are processed, so if there is an error all data is automatically rolled back. It is a good idea for you to change your institution's custom programs to commit as they go, although it is not required at present.

To minimize the amount of system resources required to store rollback information and cache message data, batch programs should commit more frequently.

The following is a discussion of the considerations involved in making programs commit records in groups.

Implications

A program designed to commit as it goes, rather than to only commit at the end of the run, has to handle some issues other programs do not:

1. It must detect if a previous run has left behind any unprocessed data, and process it.
2. It must be able to commit (`gb_common.p_commit`) records in some logical grouping, and report any errors before proceeding to the next group.
3. The control report needs to have new data added to it, namely, the number of records read, the number processed and the number rejected.

Program Structure

It is up to the designer to determine if the program exits on the first API error, or continues, displaying useful error messages as it goes. If time permits and it makes sense to the business requirements of the program, having the program continue provides the advantage that all work that can be done is done, and all errors that are encountered are reported on the first run. The user can address all the remaining issues with the data at one time and have a successful completion on the second try.

The program attempts to process records in groups of 20, and commits after each set. If an API error is encountered, it is reported, and the program continues processing with the next set.

Note

API validation errors are generated before the DML operation is attempted. Therefore, API exceptions leave nothing to roll back. All that is required before processing continues is that the message be displayed. ■

Performance Issues

The optimal number of records to commit at one time appears to be approximately 20. This number should be defined as a program constant so that it can be adjusted easily, as needed. Obviously, if an error occurs, an appropriate message should be displayed, and the program will either continue or exit.

Possible Control Report Changes

You may have to change the control report for the batch process to show how many input records were processed successfully, how many had errors, how many were skipped because they were part of a set with an error, etc.

Restartability

A program may exit after some, but not all, of the data is committed. When it restarts it must behave appropriately.

This means that the program must either:

- Mark source records as they are processed, and change the main selects so they are not retrieved again once the update/insert is committed

Or:

- Use some other mechanism to remember the state of the program at the time it was forced to exit, reread the input data up to the point, and start processing from there forward

Examples

Cursor

```
cursor selrecr is
 select x
    into temp_x
  from rctrecr a
where a.pidm in (select pidm from tapeload_table);
```

Report

Then

```
report(sel_recr,recrbody,recrhead,NULL);
```

`recrbody` does the following:

```
Insert into SRBRECR(a)
values(temp_x);
```

New Cursor

The new cursor makes this program restartable.

```
cursor selrecr is
select x
   into temp_x
from rctrecr a
where a.pidm in
   (select pidm from tapeload_table
    MINUS
    Select pidm
    from SRBRECR);  prevents records successfully inserted already from
being selected again.
```

A different example using NOT IN (caution- this may perform poorly):

```
   FOR r_rec IN (SELECT *
                 FROM rotiden
                 WHERE rotiden_stat_ind = :rec_type_ind
                   AND NVL (rotiden_delete_flag, 'N') = 'N'
                   AND rotiden_infc_code = :tape_type
                   AND rotiden_aidy_code = :aid_year)
                   AND rotiden_pidm NOT IN (SELECT rotiden_pidm FROM rotiden)
    LOOP
```

Validation and Audit/Update Option

If the program performs updates, inserts or deletes, it should have an audit option so the user can see any potential error messages and take corrective action before the updates are committed.

The API will have public validation procedures that will return information describing what data may need to be changed for the Create or Update operation to succeed.

Validation will be repeated when the API operation is called to ensure no intervening changes induce validation failures.

One way to set up auditing might be:

```
if ( compare(run_mode,"A",EQS) ) /* We're in Audit mode */
   {EXEC SQL EXECUTE
   BEGIN
     gb_thing.p_validate(a,b,c);
   END;
   -- add exception and error handling here
   END-EXEC;}
else
   {EXEC SQL EXECUTE          /* Do the real thing */
   BEGIN
     gb_thing.p_create(a,b,c);
   END;
   -- add exception and error handling here
   END-EXEC;}
```

Call GB_COMMON.p_commit or p_rollback

Batch application programs must always call the API common `p_commit` or `p_rollback` procedure rather than calling COMMIT or ROLLBACK explicitly.

This ensures that any message processing is completed at the appropriate time, and performs the commit for you.

In addition, client applications must never rely on the Oracle default behavior of committing work upon program exit. `p_commit` or `p_rollback` should always be called before the program exits.

Adding a Column to a Table

Note

Please refer to the “Implementing Changes to Existing Business Entity APIs” section in the “Banner Business Entity APIs” chapter for more information about what constitutes a signature change for an API and how to implement the new items in the API.

There are special considerations when you are adding a column to a table and that table is used by APIs:

1. After you define the column and document its functional requirements, create the Unit Test cases in the API test document. The test cases must be defined, even if the column isn’t part of the API yet.

2. Add any new error messages to the API, as necessary. Update the strings documentation in the strings package specification (_STR), too.

3. The API will have to have a new version number, unless the column can be populated without being added to the API public signature. Increment the `CURRENT_RELEASE` constant by 1.

4. Add the column to the table.

5. Rerun the API and DML generators.

6. Look for the new column in the generated code, and merge it into the existing API. This process can be simple or complex, depending on how many changes have been made to the API.

7. Test your changes.

A Code Samples

PL/SQL

Normal p_create Call

```
-- A very simple PLSQL block showing how to call an API.
-- In this case, the gb_identification API will create a
-- Spriden record.
SET SERVEROUTPUT ON                    -- 00
DECLARE                                -- 01
  gv_id varchar2(12):='GENERATED';     -- 02
  gv_pidm1 number;                     -- 03
  gv_rowid1 varchar2(18);              -- 04
BEGIN                                  -- 05
  gb_identification.p_create(          -- 06
    p_id_inout      => gv_id,          -- 07
    p_last_name     => 'Doe',          -- 08
    p_first_name    => 'Jon',          -- 09
    p_change_ind    => NULL,           -- 10
    p_entity_ind    => 'P',            -- 11
    p_origin        => 'Banner',       -- 12
    p_pidm_inout    => gv_pidm1,       -- 13
    p_rowid_out     => gv_rowid1);     -- 14
  DBMS_OUTPUT.PUT_LINE('ID: '||gv_id||' Pidm :'||gv_pidm1);      -- 15
  EXCEPTION WHEN OTHERS THEN                                     -- 16
    IF SQLCODE = gb_common_strings.ERR_CODE THEN                 -- 17
      DBMS_OUTPUT.PUT_LINE('We had an API Error:'||sqlerrm);     -- 18
    ELSE                                                         -- 19
      RAISE;                                                     -- 20
    END IF;                                                      -- 21
END;
/

Notes:
Line  Comment
----- ------------------------------------------------------------
00    So we can see the output from the dbms_output statements
01    This sample is an anonymous PL/Sql block, so we start with a DECLARE
02    Since ID is an IN/OUT parameter, we have to pass a variable rather than a
literal. If we pass a null, the API will assign an ID and return it in this vari-
able.
03    Same with PIDM, we can pass one, but in this case we want the API to assign
it.
04    The Rowid of the newly created row is always returned
05    Start the block
06    Call the API p_create procedure
07    The parameter name has _inout on the end, to remind us that it needs a vari-
able passed, and cannot accept a literal value.
08    The API technical documentation shows what datatype each parmamter is, in
this case a varchar2, and that it's required
09    First name is not required, but we create one anyway
10    Again, the technical documentation explains what valid values you can send.
```

```
11    We're creating a person record
12    We'll say it's a Banner application that is creating the record
13    We pass a variable in to catch the generated pidm. Note the '_inout' on the
parameter name.
14    We pass a variable in to catch the generated rowid.
15    We print out the values returned after the call
16    Catch any exceptions
17    We test to see if it's an API generated error
18    If it is, we just print out the error that the API returned. Normally you
would do something more than this
19
20    It is *absolutely critical* if you use WHEN OTHERS, to re-raise any excep-
tion you do not handle.
21
```

p_create Call Generating an API Exception

```
SET SERVEROUTPUT ON                    -- 00
DECLARE                                -- 01
  gv_id varchar2(12):='GENERATED';                  -- 02
  gv_pidm1 number;                     -- 03
  gv_rowid1 varchar2(18);              -- 04
BEGIN                                  -- 05
  gb_identification.p_create(          -- 06
    p_id_inout      => gv_id,          -- 07
    p_last_name     => 'Doe',          -- 08
    p_first_name    => 'Jon',          -- 09
    p_change_ind    => NULL,           -- 10
    p_entity_ind    => 'D',            -- 11
    p_origin        => 'Banner',       -- 12
    p_pidm_inout    => gv_pidm1,       -- 13
    p_rowid_out     => gv_rowid1);     -- 14
  DBMS_OUTPUT.PUT_LINE('ID: '||gv_id||' Pidm :'||gv_pidm1);      -- 15
  EXCEPTION WHEN OTHERS THEN                                     -- 16
    IF SQLCODE = gb_common_strings.ERR_CODE THEN                 -- 17
      DBMS_OUTPUT.PUT_LINE('We had an API Error.');
      DBMS_OUTPUT.PUT_LINE(' SQLCODE='||SQLCODE);
      DBMS_OUTPUT.PUT_LINE(' and the message is:'||sqlerrm);     -- 18
    ELSE                                                         -- 19
      RAISE;                                                     -- 20
    END IF;                                                      -- 21
END;
/
Line  Comment
----- ----------------------------------------------------------------
11    'D' is not a valid value. So we expect to get an error. This is the output:

We had an API Error.
SQLCODE=-20100
and the message is:ORA-20100: ::Invalid Entity Indicator.::

PL/SQL procedure successfully completed.
```

Normal p_delete Call

```
--
-- This example shows how to delete a record using an API
-- and it's primary key values.
--
DECLARE
lv_pidm NUMBER:=509;
BEGIN
  gb_bio.p_delete(p_pidm  => lv_pidm);
  EXCEPTION WHEN OTHERS THEN
    IF SQLCODE = gb_common_strings.ERR_CODE THEN
      DBMS_OUTPUT.PUT_LINE('We had an API Error.');
      DBMS_OUTPUT.PUT_LINE(' SQLCODE='||SQLCODE);
      DBMS_OUTPUT.PUT_LINE(' and the message is:'||sqlerrm);
    ELSE
      RAISE;
    END IF;
END;
/

PL/SQL procedure successfully completed.
```

p_delete Call Raising an Exception

```
--
-- This example shows how to delete a record using an API
-- and it's primary key values.
--
DECLARE
lv_pidm NUMBER:=123456;
BEGIN
  gb_bio.p_delete(p_pidm  => lv_pidm);
  EXCEPTION WHEN OTHERS THEN
    IF SQLCODE = gb_common_strings.ERR_CODE THEN
      DBMS_OUTPUT.PUT_LINE('We had an API Error.');
      DBMS_OUTPUT.PUT_LINE(' SQLCODE='||SQLCODE);
      DBMS_OUTPUT.PUT_LINE(' and the message is:'||sqlerrm);
    ELSE
      RAISE;
    END IF;
END;
/
We had an API Error.
SQLCODE=-20100
and the message is:ORA-20100: ::Cannot delete, record does not exist.::

PL/SQL procedure successfully completed.
```

p_delete Showing How to Use rowid

```
--
-- This example shows how to delete a record using an API
-- and it's primary key values.
--
DECLARE
lv_pidm NUMBER:=123456; -- this is a bogus pidm in this example
lv_rowid VARCHAR2(19):='AAAIEeAADAAAGB+AAB';
BEGIN
  gb_bio.p_delete(
    p_pidm  => lv_pidm,  -- Note that when rowid is passed, the key value parame-
ters, though required, are ignored
    p_rowid => lv_rowid);
  EXCEPTION WHEN OTHERS THEN
    IF SQLCODE = gb_common_strings.ERR_CODE THEN
      DBMS_OUTPUT.PUT_LINE('We had an API Error.');
      DBMS_OUTPUT.PUT_LINE(' SQLCODE='||SQLCODE);
      DBMS_OUTPUT.PUT_LINE(' and the message is:'||sqlerrm);
    ELSE
      RAISE;
    END IF;
END;
/

PL/SQL procedure successfully completed.
The reason it succeeded was because when ROWID is passed, the API ignores the PK
values passed, even though they are mandatory.
```

Revalidating Existing Data Using the Banner APIs

The Banner Business Entity APIs consolidate business logic that often resided in several different places. Now, all Create, Delete, Update operations on a table happen through a single package. The rules imposed by the new API ensure consistency for all Banner applications. However, it is possible that the tables already have data in them that may not meet the validation requirements of the new API.

Every time a record is updated, a copy of the result row is created and validated before the update is attempted. This means that even if the application updates column A of a record, columns A through Z all must pass the validation tests before the update is made. If column B was created before the APIs where in place, and had bad data in it, updates to that record will fail. This is true even though the application program is not attempting up update column B.

In order to make sure that all the data in a table could pass the validation checks in an API, a special program needs to be written for each. This is an outline of what that program needs to do.

This test calls the rules package `p_validate` procedure for every row in a table and records any API errors.

```
/*

This procedure is an example of how you might implement a data
cleaning tool that would examine the data already in a table,
to see if it passes at least the minimum API validations.
It also shows different ways to handle API exceptions.
Specifically, it shows how to obtain error messages and format them.
It shows how to tell API errors apart from other exceptions which must be
re-raised.

These are just examples. You can use these as a guideline.

In this example, the SPRADDR table data is validated with the gb_address API.

The data returned can be customized, but in this example, the ROWID of the row
with the error is returned along with the API error message.

Be aware that there is often business logic in the p_create and p_update
procedures of an API that validates other data in addition to that
validated by p_validate.

If you are running in 10gR2, you can go back to using DBMS_OUTPUT rather
than guraddl, as the output buffer limits on DBMS_OUTPUT have been lifted.

Here is some sample output

This is the raw error=ORA-20100: ::Invalid PIDM::::Zip is required if state is
provided::

This is with all delimiters removed=Invalid PIDM    Zip is required if state is
provided

*/
set serveroutput on
DECLARE
  lv_seq                 PLS_INTEGER := 0;
  lv_session    CONSTANT VARCHAR2(30) := userenv('SESSIONID');
  LV_MAX_ERRORS CONSTANT PLS_INTEGER  := 100; -- Stop after this many errors
  lv_err                 VARCHAR2(2000); -- The API error
  lv_temp                VARCHAR2(2000); -- temp buffer to hold modified error message
  lv_address_rec         gb_address.address_rec;
  lv_address_ref         gb_address.address_ref;
  lv_rowid               gb_common.internal_record_id_type;
  lv_message_table       gb_common.msgtab;

  CURSOR rowid_c IS
    SELECT ROWID FROM spraddr;

  PROCEDURE p_print(p_text IN VARCHAR2) IS
  BEGIN
      INSERT INTO guraddl VALUES (userenv('SESSIONID'), genutil.incr, p_text);
  END p_print;

BEGIN
  DELETE FROM guraddl WHERE guraddl_user = lv_session;
  COMMIT;

-- Start selecting ROWIDs
  FOR rowid_rec IN rowid_c LOOP
```

```
-- Use the f_query_by_rowid to get the whole row
    lv_address_ref := gb_address.f_query_by_rowid(rowid_rec.ROWID);
    FETCH lv_address_ref
      INTO lv_address_rec;
    CLOSE lv_address_ref;
-- Pass the data to the p_validate procedure and trap any exception
    BEGIN
      gb_address_rules.p_validate(p_pidm             => lv_address_rec.r_pidm,
                                  p_atyp_code        => lv_address_rec.r_atyp_code,
                                  p_seqno            => lv_address_rec.r_seqno,
                                  p_street_line1     => lv_address_rec.r_street_line1,
                                  p_city             => lv_address_rec.r_city,
                                  p_stat_code        => lv_address_rec.r_stat_code,
                                  p_zip              => lv_address_rec.r_zip,
                                  p_cnty_code        => lv_address_rec.r_cnty_code,
                                  p_natn_code        => lv_address_rec.r_natn_code,
                                  p_status_ind       => lv_address_rec.r_status_ind,
                                  p_asrc_code        => lv_address_rec.r_asrc_code,
                                  p_delivery_point   =>
lv_address_rec.r_delivery_point,
                                  p_correction_digit =>
lv_address_rec.r_correction_digit,
                                  p_reviewed_ind     => lv_address_rec.r_reviewed_ind,
                                  p_reviewed_user    =>
lv_address_rec.r_reviewed_user);

    EXCEPTION
      WHEN OTHERS THEN
        IF SQLCODE = -20100 THEN -- It's an API error
          lv_seq := lv_seq + 1;

          lv_err := SQLERRM; -- capture original error message
          p_print( 'This is the raw error='||lv_err);

          lv_temp := gb_common.f_err_msg_remove_delim(SQLERRM);
          p_print( 'This is with all delimiters removed='||lv_temp);

          -- Now pass the error to the routine that removes delimiters, removes the
          -- Oracle call stack, and returns a plsql table with one error message per
          -- entry.  Print each one out.
          lv_message_table :=
gb_common.f_err_msg_remove_delim_tbl(p_error_message => lv_err);
          FOR i in lv_message_table.FIRST..lv_message_table.LAST LOOP
             p_print(  'Messagetable('||i||')'||lv_message_table(i));
          END LOOP;
          -- Now, return the rowid that had the problem, and
          -- just strip the Oracle call stack from the front of the string.
          lv_temp := substr(lv_address_rec.r_internal_record_id || ':' ||
                            lv_err,
                            1,
                            2000);
         p_print( 'Raw error with call stack removed showing rowid='||lv_temp);
       ELSE -- It's some other exception, just reraise it.
         p_print( 'Some other Exception :'||SQLCODE);
         RAISE;  -- Make sure you re-raise it!!
       END IF;
    END;
-- Exit if we hit the limit
    IF lv_seq >= LV_MAX_ERRORS THEN
      EXIT;
    END IF;
  END LOOP;
END;
/
```

```
set linesize 100
column guraddl_command format a80 heading 'Errors' wrap word
select guraddl_command
 from guraddl
where guraddl_user = userenv('SESSIONID')
order by guraddl_seq

spool Validator.lis
/

-- Clean up
DELETE guraddl
WHERE guraddl_user = TO_CHAR (USERENV ('sessionid'));
commit;
```

Here is a sample output of one of the results.

```
This is the raw error=ORA-20100: ::Invalid PIDM.::::Zip is required if state is
provided.::

This is with all delimiters removed=Invalid PIDM.    Zip is required if state is
provided.

Messagetable(1)Invalid PIDM.
Messagetable(2)Zip is required if state is provided.
Raw error with call stack removed showing rowid=AAAIEhAAMAAAHbaAAZ:ORA-20100:
::Invalid PIDM.::::Zip is required if state is provided.::
```

Java

```
/*****************************************************************************
 $Source$
 $Revision$

 © 2006 SunGard Higher Education Corporation.  All Rights Reserved.

 CONFIDENTIAL BUSINESS INFORMATION

 THIS PROGRAM IS PROPRIETARY INFORMATION OF SYSTEMS &amp;amp; COMPUTER TECHNOLOGY
 CORPORATION AND IS NOT TO BE COPIED, REPRODUCED, LENT, OR DISPOSED OF,
 NOR USED FOR ANY PURPOSE OTHER THAN THAT WHICH IT IS SPECIFICALLY PROVIDED
 WITHOUT THE WRITTEN PERMISSION OF THE SAID COMPANY
*****************************************************************************/
package test;

import com.sct.messaging.bif.BatchResourceHolder;
import com.sct.messaging.bif.banner.BannerUtils;

import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Types;

/**
 * Example class to call the Banner Email entity API
 * This example uses the Banner Batch Integration Framework library
 * located in BANNER_HOME/general/java/gurjbif.jar
 * This example program simply creates an email record, then deletes it.
 * @author Dan Sterling
 *
 * @version 1.0
 * Jun 27, 2006    DPS    Initial Creation
 **/

public class EmailExample {


    //------------------------------ Data Members ------------------------------


    // Change to a valid pidm
    private static String pidm      = "12345";
    private static String emailCode = "HOME";
    private static String emailAddr = "somebody@myuniversity.edu";


    // ------------------------------ Constructors ------------------------------


    /**
     *
     */
```

```
    public EmailExample() {
        super();
    }


    // ------------------------------ Public Methods --------------------------


    /**
     * Main method
     * @param args username, password, jdbcUrl
     */
    public static void main(String[] args) {

        // Command line arguments require Oracle username, password, and jdbcUrl
        if (args.length < 3) {
            System.err.println("Usage: java test.EmailExample username password jd-
bcurl");;
            System.exit(1);
        }

        try {
            // Initialize BatchResources, this will connect to the Oracle database
            BatchResourceHolder.initialize( args[0], args[1], args[2], "ExampleE-
mail" );
        } catch (SQLException sqle) {
            System.err.println(sqle.getMessage());
            sqle.printStackTrace();
            System.exit( 1 );
        }

        // Get an establish database connection
        Connection conn = BatchResourceHolder.getConnection();

        // Create an Email entity
        try {
            createEmail( conn );
            BannerUtils.performCommit( conn );
        } catch (SQLException sqle) {
            BannerUtils.performRollback( conn );
            System.err.println("Error creating email");
            sqle.printStackTrace();
        }
        // Delete the Email entity
        try {
            deleteEmail( conn );
            BannerUtils.performCommit( conn );
        } catch (SQLException sqle) {
            BannerUtils.performRollback( conn );
            System.err.println("Error deleting email");
            sqle.printStackTrace();
        }

    }


    // ------------------------------ Private Methods --------------------------


    private static void createEmail( Connection conn ) throws SQLException{
        String createCall = "{ call gb_email.p_create(?,?,?,?,?,?,?,?,?,?) }";
        CallableStatement cstmt = conn.prepareCall( createCall );
```

```
        // Bind the parameters
        cstmt.setString( 1, pidm );                     // pidm
        cstmt.setString( 2, emailCode );                // emal_code
        cstmt.setString( 3, emailAddr );                // email_address
        cstmt.setString( 4, "N" );                      // status_ind
        cstmt.setString( 5, "N" );                      // preferred_ind
        cstmt.setString( 6, conn.getMetaData().getUserName() ); // user_id
        cstmt.setString( 7, "Example email create" );   // comment
        cstmt.setString( 8, "N" );                      // disp_web_ind
        cstmt.setString( 9, "Banner" );                 // data_orgin
        cstmt.registerOutParameter( 10, Types.VARCHAR ); // rowid output parameter

        // execute the statement
        cstmt.execute();
    }

    private static void deleteEmail( Connection conn ) throws SQLException {
        String deleteCall = "{ call gb_email.p_delete(?,?,?,?) }";
        CallableStatement cstmt = conn.prepareCall( deleteCall );

        // Bind the parameters
        cstmt.setString( 1 , pidm );                    // pidm
        cstmt.setString( 2, emailCode );                // emal_code
        cstmt.setString( 3, emailAddr );                // email_address
        cstmt.setString( 4, null );                     // rowid, if known

        // execute the statement
        cstmt.execute();
    }
}
```

C

```
/*
This sample program simply calls the gb_email API to create an
email record
*/
#include "guarpfe.h"
EXEC SQL INCLUDE guaorac.c;
EXEC SQL BEGIN DECLARE SECTION;
  static NUMSTR  pidm      = "";
  static CHAR5   emailcode = "";
  static CHAR27  emailaddr = "";
  static CHAR20  rowid_out = "";
int j=0;
EXEC SQL END DECLARE SECTION;

main(int argc,char *argv[])
{
  if(argc > 1)
    strcpy(user_pass, argv[1]);
  getxnam(*argv);
  login();
  strcpy(pidm, "509");
  strcpy(emailcode, "HOME");
  strcpy(emailaddr, "somebody@myuniversity.com");
  EXEC SQL EXECUTE
    DECLARE
      row_id VARCHAR2(18);
    BEGIN
      gb_email.p_create(
        p_pidm           => :pidm,
        p_emal_code      => :emailcode,
        p_email_address  => :emailaddr,
        p_rowid_out      => :rowid_out);
    END;
  END-EXEC;
  printf("New record rowid is %s\n",rowid_out);
/* After every API call, we test POSTORA_API to see if we got an error.
If we got an Oracle error, POSTORA_API will terminate the program.
If we got an API error, the transaction is rolled back, the api_error_list
structure is populated with the error messages, and control returns
to our program so we can determine if we want to handle the error and
continue.
*/
```

```
 if (POSTORA_API)
   {
      if (api_error_count > 0) {
        printf("    API generated errors:\n ");
        for(j = 0; j < api_error_count; j++)
          printf("%s ",  api_error_list[j]);
        printf("\n");
        fflush(stdout) ; }
  }
  printf("this is the end\n");
  exit2os(EXIT_SUCCESS);
  return EXIT_SUCCESS;
}

The first time you run this program, the output is
New record rowid is AAAHoGAATAAACjYABR
this is the end

the second time you run this program, the output is
New record rowid is
    API generated errors:
 Cannot create, record already exists.
this is the end
```

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W