pgAdmin 4 Documentation

Release 1.5

The pgAdmin Development Team

Jul 25, 2019

CONTENTS

Welcome to pgAdmin 4. pgAdmin is the leading Open Source management tool for Postgres, the world's most advanced Open Source database. pgAdmin 4 is designed to meet the needs of both novice and experienced Postgres users alike, providing a powerful graphical interface that simplifies the creation, maintenance and use of database objects.

Contents:

CHAPTER

ONE

GETTING STARTED

Pre-compiled and configured installation packages for pgAdmin 4 are available for a number of desktop environments; we recommend using an installer whenever possible. A standard installation using the pgAdmin installer is a server deployment.

In a Server Deployment, the pgAdmin application is deployed behind a webserver or with the WSGI interface. If you install pgAdmin in server mode, you will be prompted to provide a role name and pgAdmin password when you initially connect to pgAdmin. The first role registered with pgAdmin will be an administrative user; the administrative role can use the pgAdmin *User Management* dialog to create and manage additional pgAdmin user accounts. When a user authenticates with pgAdmin, the pgAdmin tree control displays the server definitions associated with that login role.

Contents:

1.1 Server Deployment

pgAdmin may be deployed as a web application by configuring the app to run in server mode and then deploying it either behind a webserver running as a reverse proxy, or using the WSGI interface.

The following instructions demonstrate how pgAdmin may be run as a WSGI application under `Apache HTTP`, using `mod_wsgi`.

1.1.1 Requirements

Important: Some components of pgAdmin require the ability to maintain affinity between client sessions and a specific database connection (for example, the Query Tool in which the user might run a BEGIN command followed by a number of DML SQL statements, and then a COMMIT). pgAdmin has been designed with built-in connection management to handle this, however it requires that only a single Python process is used because it is not easily possible to maintain affinity between a client session and one of multiple WSGI worker processes.

On Windows systems, the Apache HTTP server uses a single process, multi-threaded architecture. WSGI applications run in `embedded` mode, which means that only a single process will be present on this platform in all cases.

On Unix systems, the Apache HTTP server typically uses a multi-process, single threaded architecture (this is dependent on the `MPM` that is chosen at compile time). If `embedded` mode is chosen for the WSGI application, then there will be one Python environment for each Apache process, each with it's own connection manager which will lead to loss of connection affinity. Therefore one should use `mod_wsgi`'s `daemon` mode, configured to use a single process. This will launch a single instance of the WSGI application which is utilised by all the Apache worker processes.

Whilst it is true that this is a potential performance bottleneck, in reality pgAdmin is not a web application that's ever likely to see heavy traffic unlike a busy website, so in practice should not be an issue.

Future versions of pgAdmin may introduce a shared connection manager process to overcome this limitation, however that is a significant amount of work for little practical gain.

1.1.2 Configuration

In order to configure pgAdmin to run in server mode, it is first necessary to configure the Python code to run in multi-user mode, and then to configure the web server to find and execute the code.

Note that there are multiple configuration files that are read at startup by pgAdmin. These are as follows:

- `config.py`: This is the main configuration file, and should not be modified. It can be used as a reference for configuration settings, that may be overridden in one of the following files.
- `config_distro.py`: This file is read after `config.py` and is intended for packagers to change any settings that are required for their pgAdmin distribution. This may typically include certain paths and file locations.
- `config_local.py`: This file is read after `config_distro.py` and is intended for end users to change any default or packaging specific settings that they may wish to adjust to meet local preferences or standards.

Python

In order to configure the Python code, follow these steps:

1. Create a `config_local.py` file alongside the existing `config.py` file.
2. Edit `config_local.py` and add the following setting:

```
SERVER_MODE = True
```

3. In most cases, the default file locations are setup for running in desktop mode. Add settings similar to the following to `config_local.py` to use paths suitable for server mode.

   NOTE: You must ensure the directories specified are writeable by the user that the web server processes will be running as, e.g. apache or www-data.

```
LOG_FILE = '/var/log/pgadmin4/pgadmin4.log'
SQLITE_PATH = '/var/lib/pgadmin4/pgadmin4.db'
SESSION_DB_PATH = '/var/lib/pgadmin4/sessions'
STORAGE_DIR = '/var/lib/pgadmin4/storage'
```

4. Run the following command to create the configuration database:

```
# python setup.py
```

5. Change the ownership of the configuration database to the user that the web server processes will run as, for example, assuming that the web server runs as user www-data in group www-data, and that the SQLite path is `/var/lib/pgadmin4/pgadmin4.db`:

```
# chown www-data:www-data /var/lib/pgadmin4/pgadmin4.db
```

Apache HTTPD Configuration (Windows)

Once Apache HTTP has been configured to support `mod_wsgi`, the pgAdmin application may be configured similarly to the example below:

```
<VirtualHost *>
    ServerName pgadmin.example.com
    WSGIScriptAlias / "C:\Program Files\pgAdmin4\web\pgAdmin4.wsgi"
    <Directory "C:\Program Files\pgAdmin4\web">
            Order deny,allow
            Allow from all
    </Directory>
</VirtualHost>
```

Apache HTTPD Configuration (Linux/Unix)

Once Apache HTTP has been configured to support `mod_wsgi`, the pgAdmin application may be configured similarly to the example below:

```
<VirtualHost *>
    ServerName pgadmin.example.com

    WSGIDaemonProcess pgadmin processes=1 threads=25
    WSGIScriptAlias / /opt/pgAdmin4/web/pgAdmin4.wsgi

    <Directory /opt/pgAdmin4/web>
        WSGIProcessGroup pgadmin
        WSGIApplicationGroup %{GLOBAL}
        Order deny,allow
        Allow from all
    </Directory>
</VirtualHost>
```

Note: If you're using Apache HTTPD 2.4 or later, replace the lines:

```
Order deny,allow
Allow from all
```

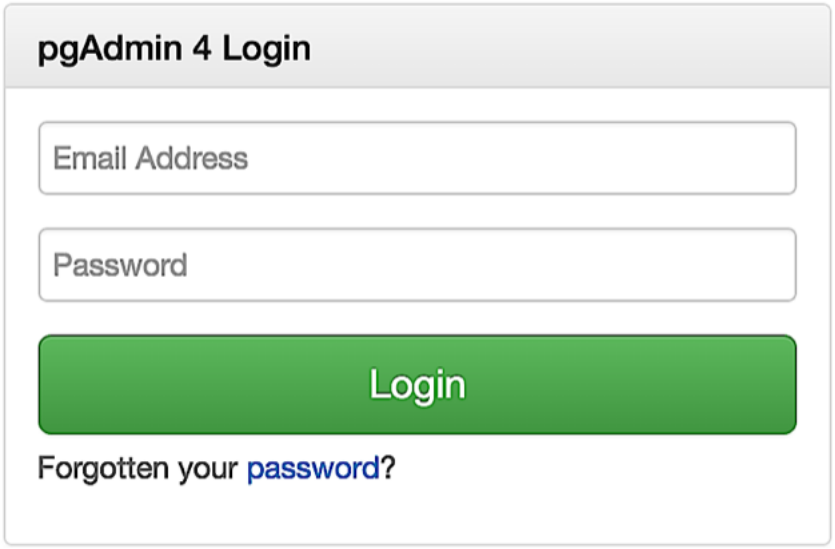
with:

```
Require all granted
```

Adjust as needed to suit your access control requirements.

1.2 The pgAdmin Login Dialog

Use the *pgAdmin Login* dialog to log in to pgAdmin:

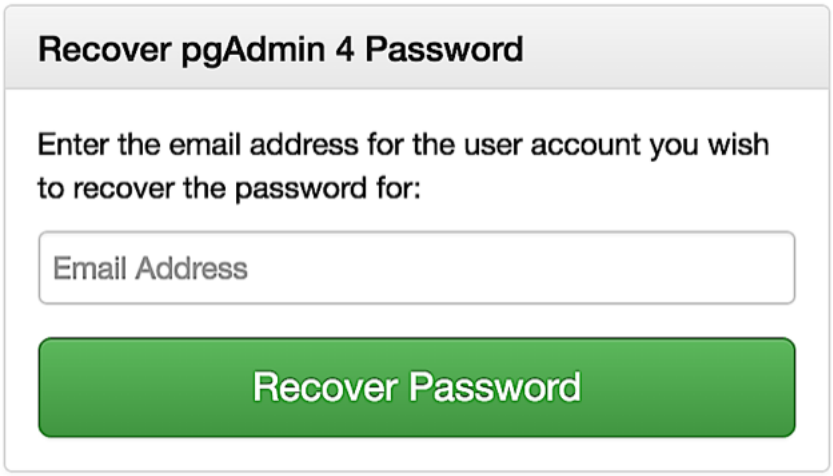


Use the fields in the *pgAdmin Login* dialog to authenticate your connection:

1. Provide the email address associated with your account in the *Email Address* field.
2. Provide your password in the *Password* field.
3. Click the *Login* button to securely log into pgAdmin.

Recovering a Lost Password

If you cannot supply your password, click the *Forgotten your password?* button to launch a password recovery utility.



1. Provide the email address associated with your account in the *Email Address* field.
2. Click the *Recover Password* button to initiate recovery. An email, with directions on how to reset a password, will be sent to the address entered in the *Email Address* field.

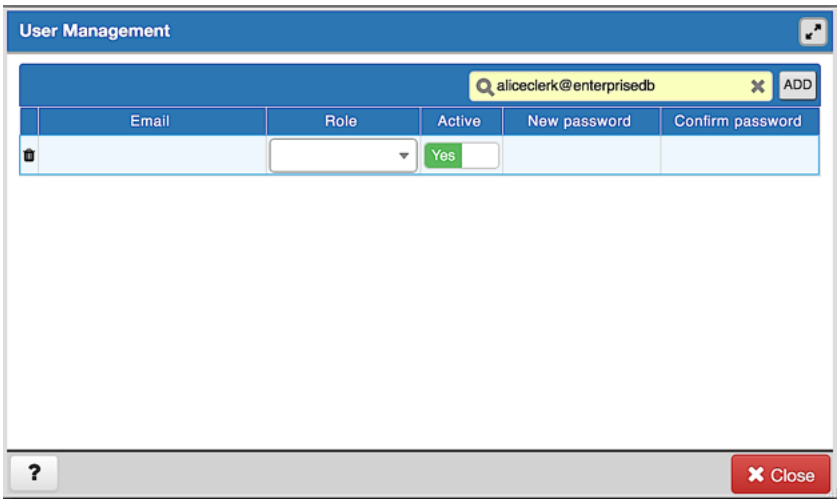
If you have forgotten the email associated with your account, please contact your administrator.

1.3 The User Management Dialog

When invoking pgAdmin in desktop mode, a password is randomly generated, and then ignored. If you install pgAdmin in server mode, you will be prompted for an administrator email and password for the pgAdmin client.

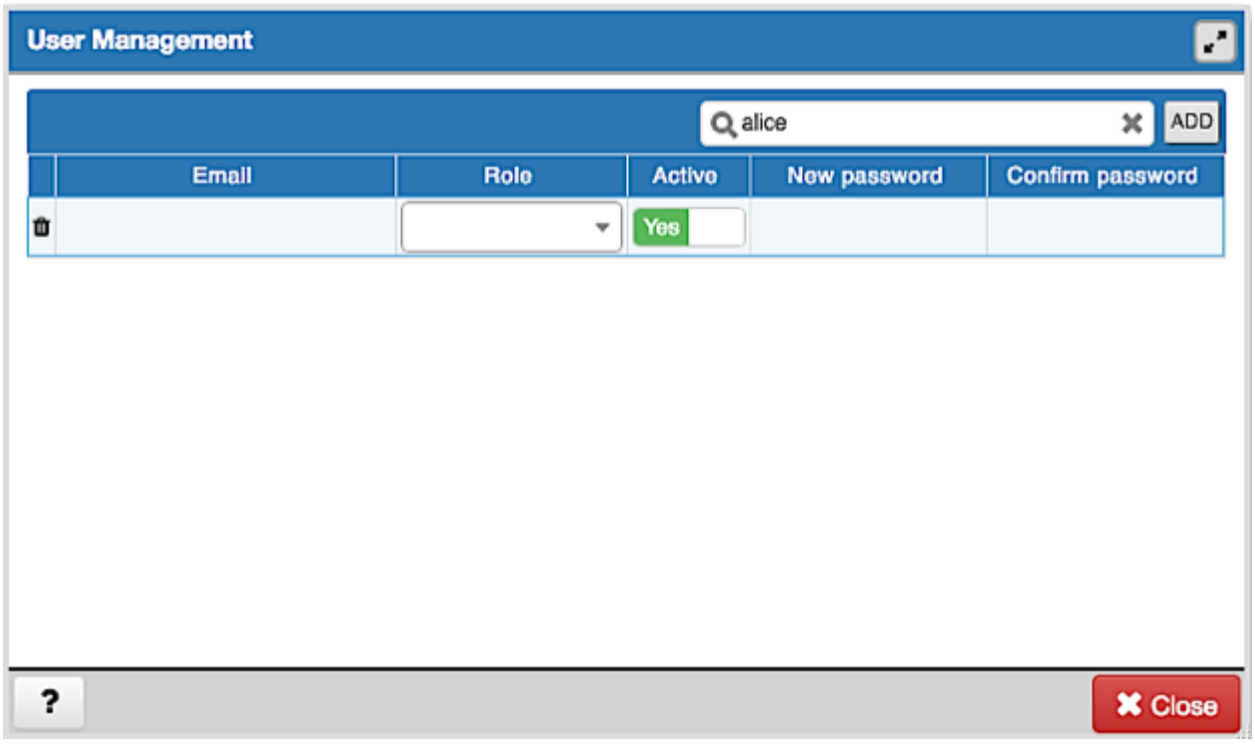
When you authenticate with pgAdmin, the server definitions associated with that login role are made available in the tree control. An administrative user can use the *User Management* dialog to

- add or delete pgAdmin roles
- assign privileges
- manage the password associated with a role



Use the *Filter by email* search field to find a user; enter a user's email address to find a user. If the user exists, the *User Management* table will display the user's current information.

To add a user, click *Add* to add new role.



Provide information about the new pgAdmin role in the row:

- Click in the *Email* field, and provide an email address for the user; this address will be used to recover the

password associated with the role should the password be lost.

- Use the drop-down listbox next to *Role* to select whether a user is an *Administrator* or a *User*.
  - Select *Administrator* if the user will have administrative privileges within the pgAdmin client.
  - Select *User* to create a non-administrative user account.
- Move the *Active* switch to the *No* position if the account is not currently active; the default is *Yes*. Use this switch to disable account activity without deleting an account.
- Use the *New password* field to provide the password associated with the user specified in the *Email* field.
- Re-enter the password in the *Confirm password* field.

To discard a user, and revoke access to pgAdmin, click the trash icon to the left of the row and confirm deletion in the *Delete user?* dialog.

Users with the *Administrator* role are able to add, edit and remove pgAdmin users, but otherwise have the same capabilities as those with the *User* role.

- Click the *Help* button (?) to access online help.
- Click the *Close* button to save work. You will be prompted to return to the dialog if your selections cannot be saved.

In a Desktop Deployment, the pgAdmin application is configured to use the desktop runtime environment to host and display the program on a supported platform. Typically, users will install a pre-built package to run pgAdmin in desktop mode, but a manual desktop deployment can be installed and though it is more difficult to setup, it may be useful for developers interested in understanding how pgAdmin works.

Contents:

1.4 Desktop Deployment

pgAdmin may be deployed as a desktop application by configuring the application to run in desktop mode and then utilising the desktop runtime to host and display the program on a supported Windows, Mac OS X or Linux installation.

Note: Pre-compiled and configured installation packages are available for a number of platforms. These packages should be used by end-users whereever possible - the following information is useful for the maintainers of those packages and users interested in understanding how pgAdmin works.

1.4.1 Configuration

In order to configure pgAdmin to run in desktop mode, it is first necessary to configure the Python code to run in single-user mode, and then to configure the runtime to find and execute the code.

Note that there are multiple configuration files that are read at startup by pgAdmin. These are as follows:

- `config.py`: This is the main configuration file, and should not be modified. It can be used as a reference for configuration settings, that may be overridden in one of the following files.
- `config_distro.py`: This file is read after `config.py` and is intended for packagers to change any settings that are required for their pgAdmin distribution. This may typically include certain paths and file locations.
- `config_local.py`: This file is read after `config_distro.py` and is intended for end users to change any default or packaging specific settings that they may wish to adjust to meet local preferences or standards.

Python

In order to configure the Python code, follow these steps:

1. Ensure that any existing configuration database (`pgadmin4.db`) is moved out of the way in the `web/` directory containing the pgAdmin Python code.
2. Create a `config_local.py` file alongside the existing `config.py` file.
3. Edit `config_local.py` and add the following setting:

```
SERVER_MODE = False
```

4. Run the following command to create the configuration database:

```
$ python setup.py
```

Alternatively, you can simply run `pgAdmin4.py` at this point or aat a later time, and `pgadmin4.db` will be created automatically at first run.

Runtime

When executed, the runtime will automatically try to execute the pgAdmin Python application. If execution fails, it will prompt you to adjust the Python Path to include the directories containing the pgAdmin code as well as any additional Python dependencies. You can enter a list of paths by separating them with a semi-colon character, for example:

```
/Users/dpage/.virtualenvs/pgadmin4/lib/python2.7/site-packages/;/Users/dpage/python-
↪libs/
```

The configuration settings are stored using the QSettings class in Qt, which will use an INI file on Unix systems, a plist file on Mac OS X, and the registry on Windows. The Python Path setting is stored in the `PythonPath` key.

The pgAdmin 4 client features a highly-customizable display that features drag-and-drop panels that you can arrange to make the best use of your desktop environment.

The tree control provides an elegant overview of the managed servers, and the objects that reside on each server. Right-click on a node within the tree control to access context-sensitive menus that provide quick access to management tasks for the selected object.

The tabbed browser provide quick access to statistical information about each object in the tree control, and pgAdmin tools and utilities (such as the Query tool and the debugger). pgAdmin opens additional feature tabs each time you access the extended functionality offered by pgAdmin tools; you can open, close, and re-arrange feature tabs as needed.

Use the *Preferences* dialog to customize the content and colors of the pgAdmin display. To open the *Preferences* dialog, select *Preferences* from the *File* menu.

Help buttons in the lower-left corner of each dialog will open the online help for the dialog. You can access additional Postgres help by navigating through the *Help* menu, and selecting the name of the resource that you wish to open.

Contents:

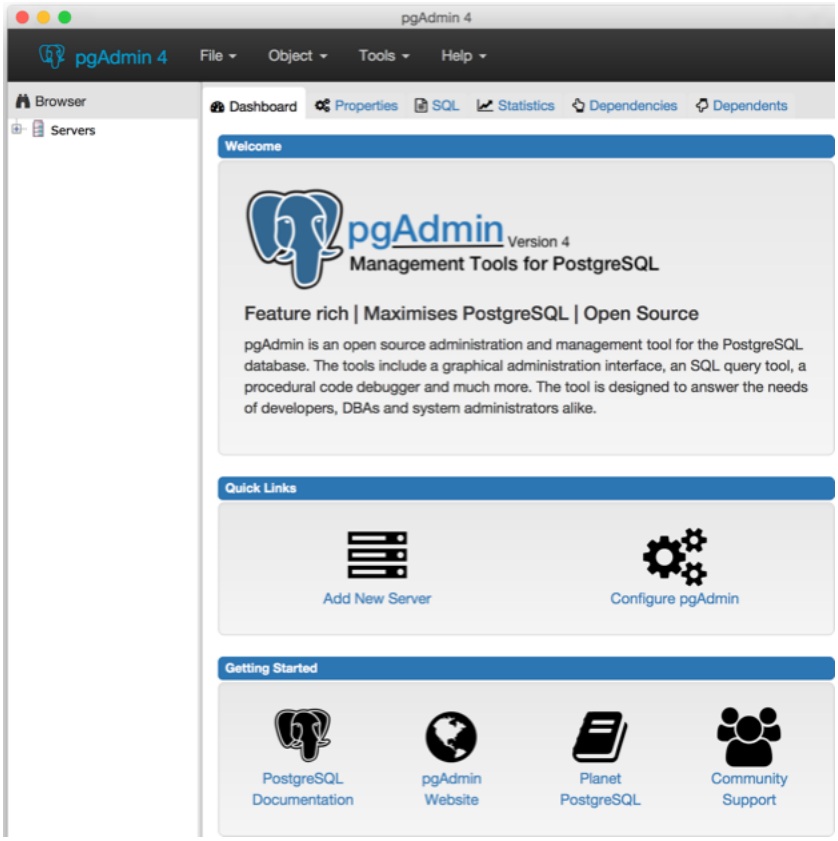
1.5 The pgAdmin 4 Client

pgAdmin 4 supports all PostgreSQL features, from writing simple SQL queries to developing complex databases. It is designed to query an active database (in real-time), allowing you to stay current with modifications and implementations.

Features of pgAdmin 4 include:

- auto-detection and support for objects discovered at run-time
- a live SQL query tool with direct data editing
- support for administrative queries
- a syntax-highlighting SQL editor
- redesigned graphical interfaces
- powerful management dialogs and tools for common tasks
- responsive, context-sensitive behavior
- supportive error messages
- helpful hints
- online help and information about using pgAdmin dialogs and tools.

When pgAdmin opens, the interface features a menu bar and a window divided into two panes: the *Browser* tree control in the left pane, and a tabbed browser in the right pane.



Select an icon from the *Quick Links* panel on the *Dashboard* tab to:

- Click the *Add New Server* button to open the *Create - Server dialog* to add a new server definition.
- Click the *Configure pgAdmin* button to open the *Preferences dialog* to customize your pgAdmin client.

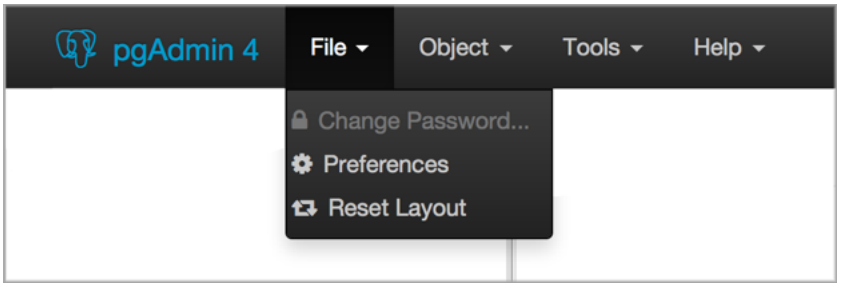
Links in the *Getting Started* panel open a new browser tab that provide useful information for Postgres users:

- Click the *PostgreSQL Documentation* link to navigate to the *Documentation* page for the PostgreSQL open-source project; once at the project site, you can review the manuals for the currently supported versions of the PostgreSQL server.
- Click the *pgAdmin Website* link to navigate to the pgAdmin project website. The pgAdmin site features news about recent pgAdmin releases and other project information.
- Click the *Planet PostgreSQL* link to navigate to the blog aggregator for Postgres related blogs.
- Click the *Community Support* link to navigate to the *Community* page at the PostgreSQL open-source project site; this page provides information about obtaining support for PostgreSQL features.

1.6 The pgAdmin Menu Bar

The pgAdmin menu bar provides drop-down menus for access to options, commands, and utilities. The menu bar displays the following selections: *File*, *Object*, Tools*, and *Help*. Selections may be grayed out which indicates they are disabled for the object currently selected in the *pgAdmin* tree control.

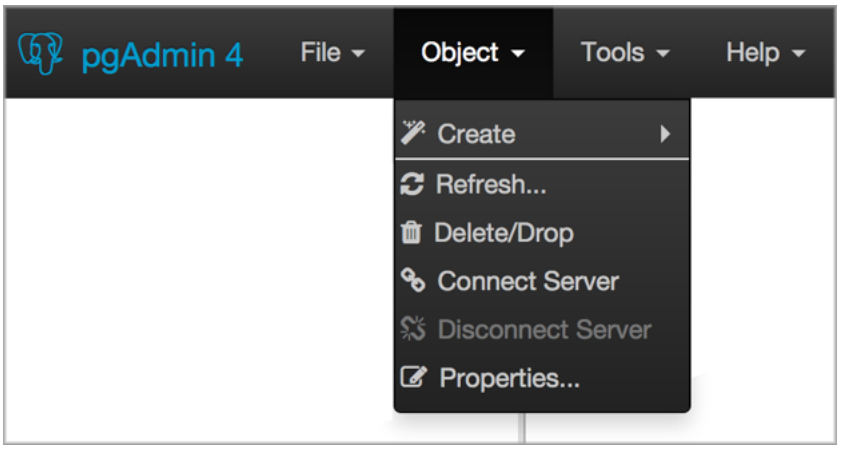
The File Menu



Use the *File* menu to access the following options:

Option	Action
Change Password...	Click to open the *Change Password...* dialog to change your password.
Preferences	Click to open the *Preferences* dialog to to customize your pgAdmin settings.
Reset Layout	If you have modified the workspace, click to restore the default layout.

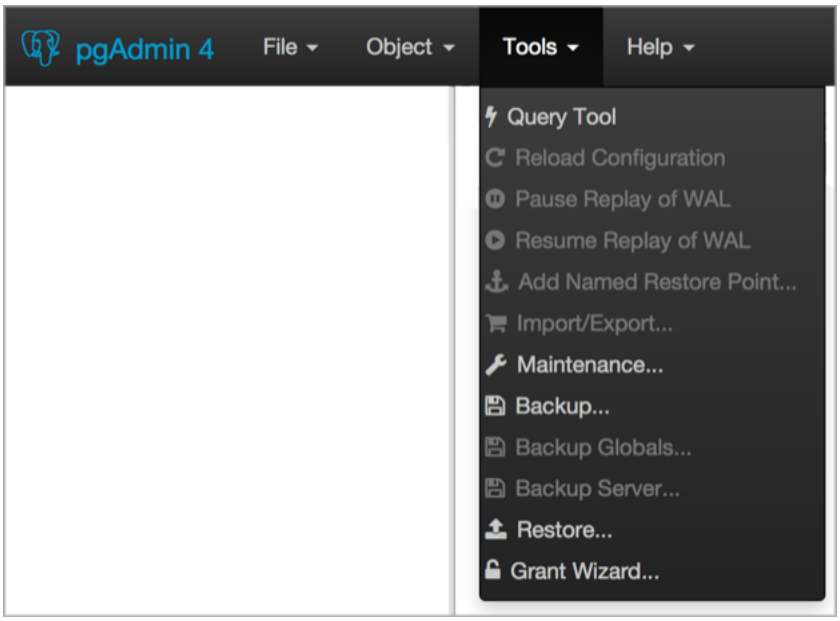
The Object Menu



The *Object* menu is context-sensitive. Use the *Object* menu to access the following options (in alphabetical order):

Option	Action
Connect Server...	Click to open the *Connect to Server* dialog to establish a connection with a server.
Create	Click *Create* to access a context menu that provides context-sensitive selections. Your selection opens a *Create* dialog for creating a new object.
Delete/Drop	Click to delete the currently selected object from the server.
Disconnect Server...	Click to refresh the currently selected object.
Drop Cascade	Click to delete the currently selected object and all dependent objects from the server.
Properties...	Click to review or modify the currently selected object's properties.
Refresh...	Click to refresh the currently selected object.
Scripts	Click to open the *Query tool* to edit or view the selected script from the flyout menu.
Trigger(s)	Click to *Disable* or *Enable* trigger(s) for the currently selected table. Options are displayed on the flyout menu.
Truncate	Click to remove all rows from a table (*Truncate*) or to remove all rows from a table and its child tables (*Truncate Cascade*). Options are displayed on the flyout menu.
View Data	Click to access a context menu that provides several options for viewing data (see below).

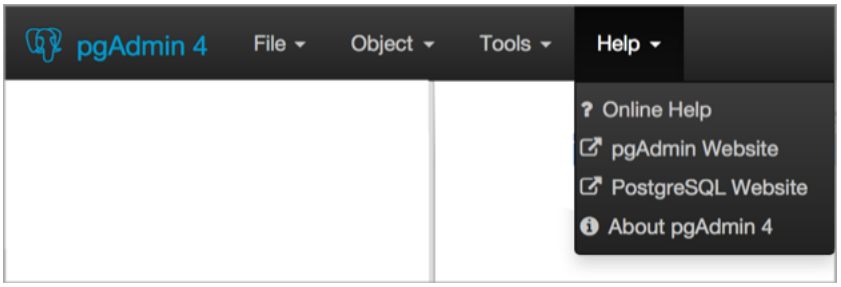
The Tools Menu



Use the *Tools* menu to access the following options (in alphabetical order):

Option	Action
Add named restore point	Click to open the *Add named restore point...* dialog to take a point-in-time snapshot of the current server state.
Backup...	Click to open the *Backup...* dialog to backup database objects.
Backup Globals...	Click to open the *Backup Globals...* dialog to backup cluster objects.
Backup Server...	Click to open the *Backup Server...* dialog to backup a server.
Grant Wizard...	Click to access the *Grant Wizard* tool.
Import/Export...	Click to open the *Import/Export data...* dialog to import or export data from a table.
Maintenance...	Click to open the *Maintenance...* dialog to VACUUM, ANALYZE, REINDEX, or CLUSTER.
Pause replay of WAL	Click to pause the replay of the WAL log.
Query tool	Click to open the *Query tool* for the currently selected object.
Reload Configuration...	Click to update configuration files without restarting the server.
Restore...	Click to access the *Restore* dialog to restore database files from a backup.
Resume replay of WAL	Click to resume the replay of the WAL log.

The Help Menu



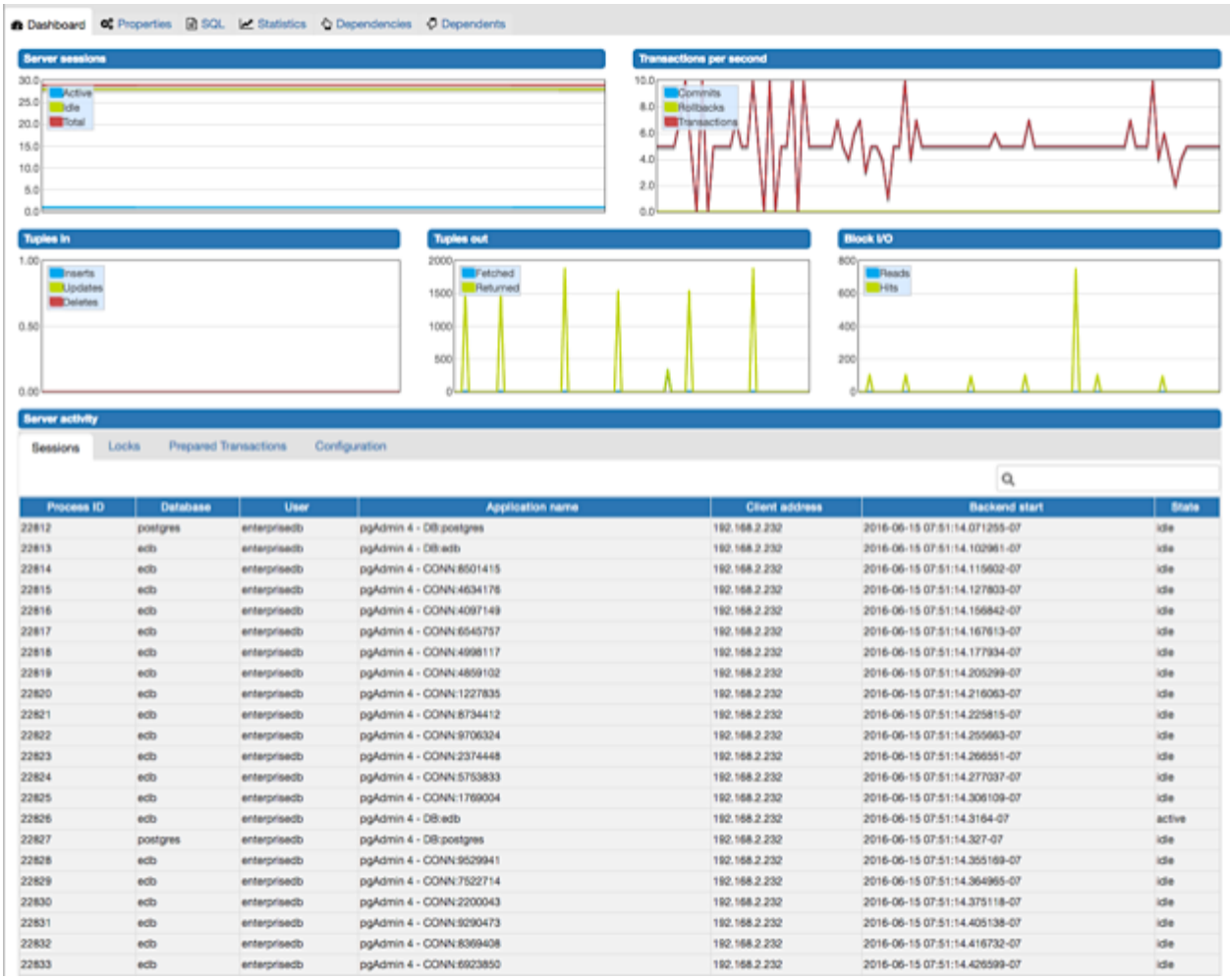
Use the options on the *Help* menu to access online help documents, or to review information about the pgAdmin installation (in alphabetical order):

Option	Action
About pgAdmin 4	Click to open a window where you will find information about pgAdmin; this includes the current version and the current user.
Online Help	Click to open documentation support for using pgAdmin utilities, tools and dialogs. Navigate (in the newly opened tab?) help documents in the left browser pane or use the search bar to specify a topic.
pgAdmin Website	Click to open the *pgAdmin.org* website in a browser window.
PostgreSQL Website	Click to access the PostgreSQL core documentation hosted at the PostgreSQL site. The site also offers guides, tutorials, and resources.

1.7 The pgAdmin Tabbed Browser

The right pane of the *pgAdmin* window features a collection of tabs that display information about the object currently selected in the *pgAdmin* tree control in the left window.

Permanent tabs are named *Dashboard, *Properties*, *SQL*, *Statistics*, *Dependencies* and *Dependents*; each tab may be repositioned as a floating window. Select a tab to access information about the highlighted object in the tree control.

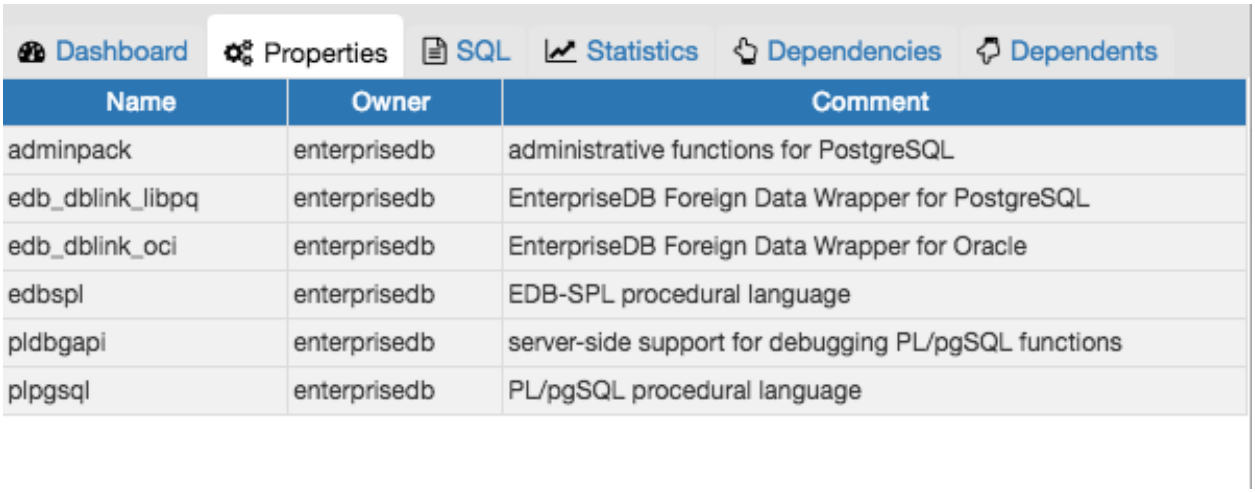


The *Dashboard* tab provides a graphical analysis of the usage statistics for the selected server or database:

- The *Server sessions* or *Database sessions* graph displays the interactions with the server or database.
- The *Transactions per second* graph displays the commits, rollbacks, and total transactions per second that are taking place on the server or database.
- The *Tuples In* graph displays the number of tuples inserted, updated, and deleted on the server or database.
- The *Tuples out* graph displays the number of tuples fetched and returned from the server or database.
- The *Block I/O* graph displays the number of blocks read from the filesystem or fetched from the buffer cache (but not the operating system's file system cache) for the server or database.

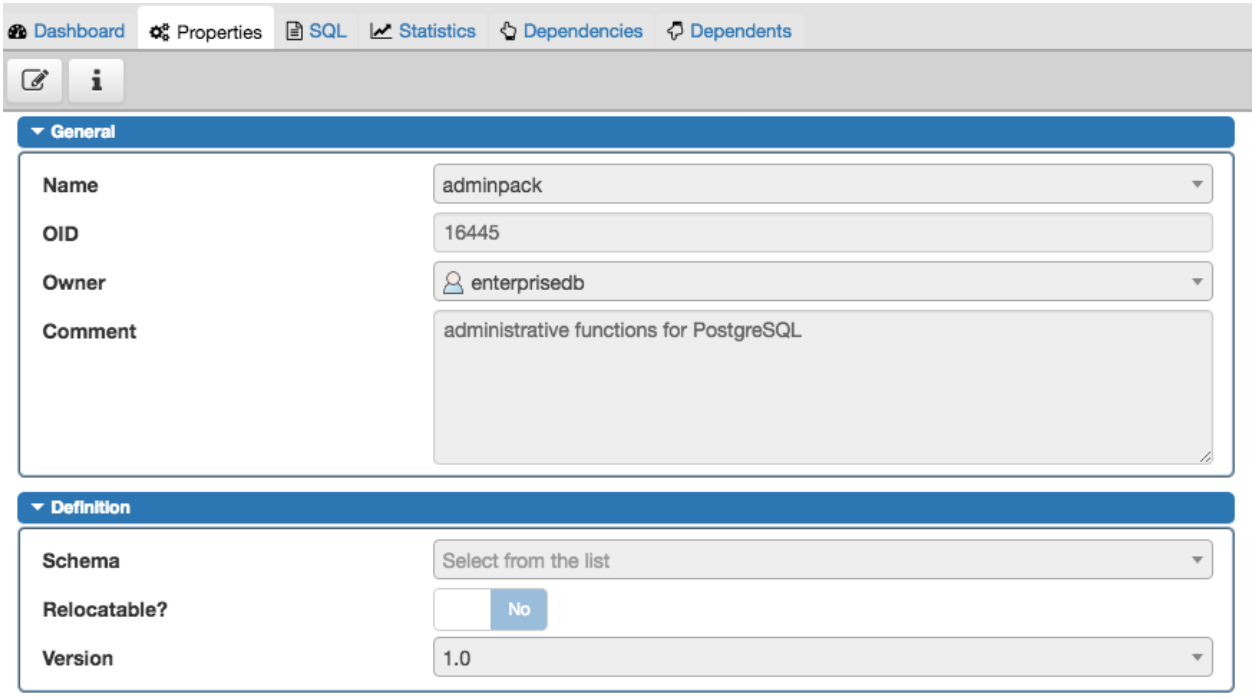
The *Server activity* panel displays information about sessions, locks, prepared transactions and configuration. The information is presented in context-sensitive tables.

Click the *Properties* tab to continue.

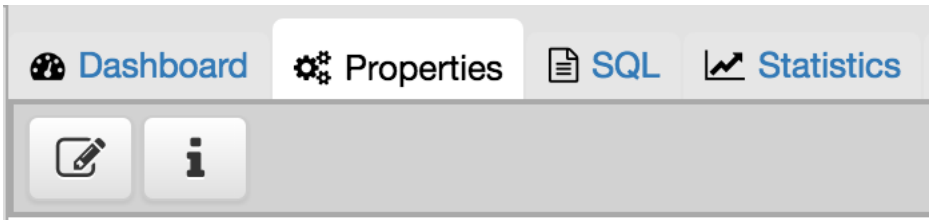


Review properties on expandable windows specific to the *Object* selected. If multiple boxes are displayed, you can click the arrow to the left on the blue bar at the top of each box:

- Point the arrow to the right to contract the box.
- Point the arrow down to expand the window.



Click the *Edit* icon in the toolbar under the browser tabs to launch a dialog.



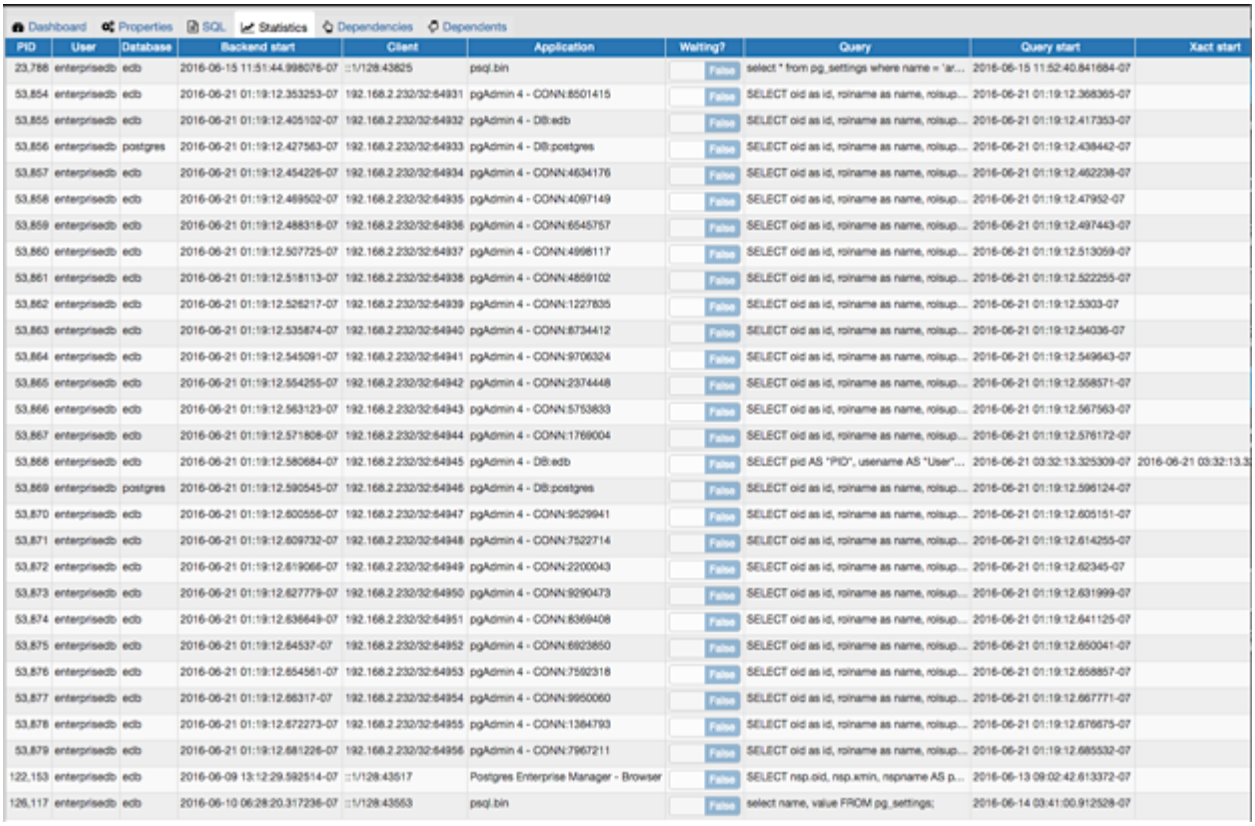
If you change properties in the opened dialog, save your work. The *Properties* tab updates to show recent modifica-

tions.

Click the *SQL* tab to continue.

The SQL pane on the *SQL* tab contains an SQL script that creates the highlighted object, and if applicable, a (commented out) SQL statement that will *DROP* the selected object. You can copy the SQL statements to an editor of your choice using cut & paste shortcuts.

Click the *Statistics* tab to continue.

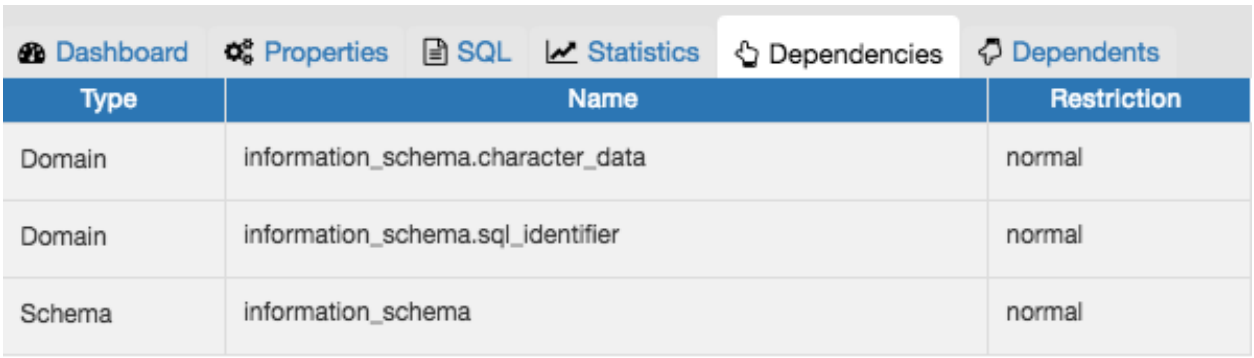


PID	User	Database	Backend start	Client	Application	Waiting?	Query	Query start	Xact start
23,788	enterprisedb	edb	2016-06-15 11:51:44.998076-07	::1/128:43825	psql.bin	False	select * from pg_settings where name = 'ar...	2016-06-15 11:52:40.841684-07	
53,854	enterprisedb	edb	2016-06-21 01:19:12.353253-07	192.168.2.232/32:64931	pgAdmin 4 - CONN:8501415	False	SELECT oid as id, rolname as name, rolsup...	2016-06-21 01:19:12.368365-07	
53,855	enterprisedb	edb	2016-06-21 01:19:12.405102-07	192.168.2.232/32:64932	pgAdmin 4 - DB:edb	False	SELECT oid as id, rolname as name, rolsup...	2016-06-21 01:19:12.417353-07	
53,856	enterprisedb	postgres	2016-06-21 01:19:12.427563-07	192.168.2.232/32:64933	pgAdmin 4 - DB:postgres	False	SELECT oid as id, rolname as name, rolsup...	2016-06-21 01:19:12.438442-07	
53,857	enterprisedb	edb	2016-06-21 01:19:12.454226-07	192.168.2.232/32:64934	pgAdmin 4 - CONN:4634176	False	SELECT oid as id, rolname as name, rolsup...	2016-06-21 01:19:12.462238-07	
53,858	enterprisedb	edb	2016-06-21 01:19:12.469502-07	192.168.2.232/32:64935	pgAdmin 4 - CONN:4097149	False	SELECT oid as id, rolname as name, rolsup...	2016-06-21 01:19:12.47952-07	
53,859	enterprisedb	edb	2016-06-21 01:19:12.488318-07	192.168.2.232/32:64936	pgAdmin 4 - CONN:6545757	False	SELECT oid as id, rolname as name, rolsup...	2016-06-21 01:19:12.497443-07	
53,860	enterprisedb	edb	2016-06-21 01:19:12.507725-07	192.168.2.232/32:64937	pgAdmin 4 - CONN:4998117	False	SELECT oid as id, rolname as name, rolsup...	2016-06-21 01:19:12.513059-07	
53,861	enterprisedb	edb	2016-06-21 01:19:12.518113-07	192.168.2.232/32:64938	pgAdmin 4 - CONN:4859102	False	SELECT oid as id, rolname as name, rolsup...	2016-06-21 01:19:12.522255-07	
53,862	enterprisedb	edb	2016-06-21 01:19:12.526217-07	192.168.2.232/32:64939	pgAdmin 4 - CONN:1227835	False	SELECT oid as id, rolname as name, rolsup...	2016-06-21 01:19:12.5303-07	
53,863	enterprisedb	edb	2016-06-21 01:19:12.535874-07	192.168.2.232/32:64940	pgAdmin 4 - CONN:8734412	False	SELECT oid as id, rolname as name, rolsup...	2016-06-21 01:19:12.54036-07	
53,864	enterprisedb	edb	2016-06-21 01:19:12.545091-07	192.168.2.232/32:64941	pgAdmin 4 - CONN:9706324	False	SELECT oid as id, rolname as name, rolsup...	2016-06-21 01:19:12.549643-07	
53,865	enterprisedb	edb	2016-06-21 01:19:12.554255-07	192.168.2.232/32:64942	pgAdmin 4 - CONN:2374448	False	SELECT oid as id, rolname as name, rolsup...	2016-06-21 01:19:12.558571-07	
53,866	enterprisedb	edb	2016-06-21 01:19:12.563123-07	192.168.2.232/32:64943	pgAdmin 4 - CONN:5753833	False	SELECT oid as id, rolname as name, rolsup...	2016-06-21 01:19:12.567563-07	
53,867	enterprisedb	edb	2016-06-21 01:19:12.571808-07	192.168.2.232/32:64944	pgAdmin 4 - CONN:1769004	False	SELECT oid as id, rolname as name, rolsup...	2016-06-21 01:19:12.576172-07	
53,868	enterprisedb	edb	2016-06-21 01:19:12.580684-07	192.168.2.232/32:64945	pgAdmin 4 - DB:edb	False	SELECT pid AS "PID", username AS "User"...	2016-06-21 03:32:13.325309-07	2016-06-21 03:32:13.3
53,869	enterprisedb	postgres	2016-06-21 01:19:12.590545-07	192.168.2.232/32:64946	pgAdmin 4 - DB:postgres	False	SELECT oid as id, rolname as name, rolsup...	2016-06-21 01:19:12.596124-07	
53,870	enterprisedb	edb	2016-06-21 01:19:12.600556-07	192.168.2.232/32:64947	pgAdmin 4 - CONN:9529941	False	SELECT oid as id, rolname as name, rolsup...	2016-06-21 01:19:12.605151-07	
53,871	enterprisedb	edb	2016-06-21 01:19:12.609732-07	192.168.2.232/32:64948	pgAdmin 4 - CONN:7522714	False	SELECT oid as id, rolname as name, rolsup...	2016-06-21 01:19:12.614255-07	
53,872	enterprisedb	edb	2016-06-21 01:19:12.619066-07	192.168.2.232/32:64949	pgAdmin 4 - CONN:2200043	False	SELECT oid as id, rolname as name, rolsup...	2016-06-21 01:19:12.62345-07	
53,873	enterprisedb	edb	2016-06-21 01:19:12.627779-07	192.168.2.232/32:64950	pgAdmin 4 - CONN:9290473	False	SELECT oid as id, rolname as name, rolsup...	2016-06-21 01:19:12.631999-07	
53,874	enterprisedb	edb	2016-06-21 01:19:12.636649-07	192.168.2.232/32:64951	pgAdmin 4 - CONN:8369408	False	SELECT oid as id, rolname as name, rolsup...	2016-06-21 01:19:12.641125-07	
53,875	enterprisedb	edb	2016-06-21 01:19:12.64537-07	192.168.2.232/32:64952	pgAdmin 4 - CONN:6923850	False	SELECT oid as id, rolname as name, rolsup...	2016-06-21 01:19:12.650041-07	
53,876	enterprisedb	edb	2016-06-21 01:19:12.654561-07	192.168.2.232/32:64953	pgAdmin 4 - CONN:7592318	False	SELECT oid as id, rolname as name, rolsup...	2016-06-21 01:19:12.658857-07	
53,877	enterprisedb	edb	2016-06-21 01:19:12.66317-07	192.168.2.232/32:64954	pgAdmin 4 - CONN:9950060	False	SELECT oid as id, rolname as name, rolsup...	2016-06-21 01:19:12.667771-07	
53,878	enterprisedb	edb	2016-06-21 01:19:12.672273-07	192.168.2.232/32:64955	pgAdmin 4 - CONN:1384793	False	SELECT oid as id, rolname as name, rolsup...	2016-06-21 01:19:12.676675-07	
53,879	enterprisedb	edb	2016-06-21 01:19:12.681226-07	192.168.2.232/32:64956	pgAdmin 4 - CONN:7967211	False	SELECT oid as id, rolname as name, rolsup...	2016-06-21 01:19:12.685532-07	
122,153	enterprisedb	edb	2016-06-09 13:12:29.592514-07	::1/128:43517	Postgres Enterprise Manager - Browser	False	SELECT nsp.oid, nsp.xmin, nspname AS p...	2016-06-13 09:02:42.613372-07	
126,117	enterprisedb	edb	2016-06-10 06:28:20.317236-07	::1/128:43553	psql.bin	False	select name, value FROM pg_settings;	2016-06-14 03:41:00.912528-07	

The *Statistics* tab displays the statistics gathered for each object on the tree control; the statistics displayed in the table vary by the type of object that is selected. Click a column heading to sort the table by the data displayed in the column; click again to reverse the sort order. The following table lists some of the statistics that are available:

Panel	Description
PID	The process ID associated with the row.
User	The name of the user that owns the object.
Database	displays the database name.
Backends	displays the number of current connections to the database.
Backend start	The start time of the backend process.
Xact Committed	displays the number of transactions committed to the database within the last week.
Xact Rolled Back	displays the number of transactions rolled back within the last week.
Blocks Read	displays the number of blocks read from memory (in megabytes) within the last week.
Blocks Hit	displays the number of blocks hit in the cache (in megabytes) within the last week.
Tuples Returned	displays the number of tuples returned within the last week.
Tuples Fetched	displays the number of tuples fetched within the last week.
Tuples Inserted	displays the number of tuples inserted into the database within the last week.
Tuples Updated	displays the number of tuples updated in the database within the last week.
Tuples Deleted	displays the number of tuples deleted from the database within the last week.
Last statistics reset	displays the time of the last statistics reset for the database.
Tablespace conflicts	displays the number of queries canceled because of recovery conflict with dropped tablespaces in database.
Lock conflicts	displays the number of queries canceled because of recovery conflict with locks in database.
Snapshot conflicts	displays the number of queries canceled because of recovery conflict with old snapshots in database.
Bufferpin conflicts	displays the number of queries canceled because of recovery conflict with pinned buffers in database.
Temporary files	displays the total number of temporary files, including those used by the statistics collector.
Size of temporary files	displays the size of the temporary files.
Deadlocks	displays the number of queries canceled because of a recovery conflict with deadlocks in database.
Block read time	displays the number of milliseconds required to read the blocks read.
Block write time	displays the number of milliseconds required to write the blocks read.
Size	displays the size (in megabytes) of the selected database.

Click the *Dependencies* tab to continue.

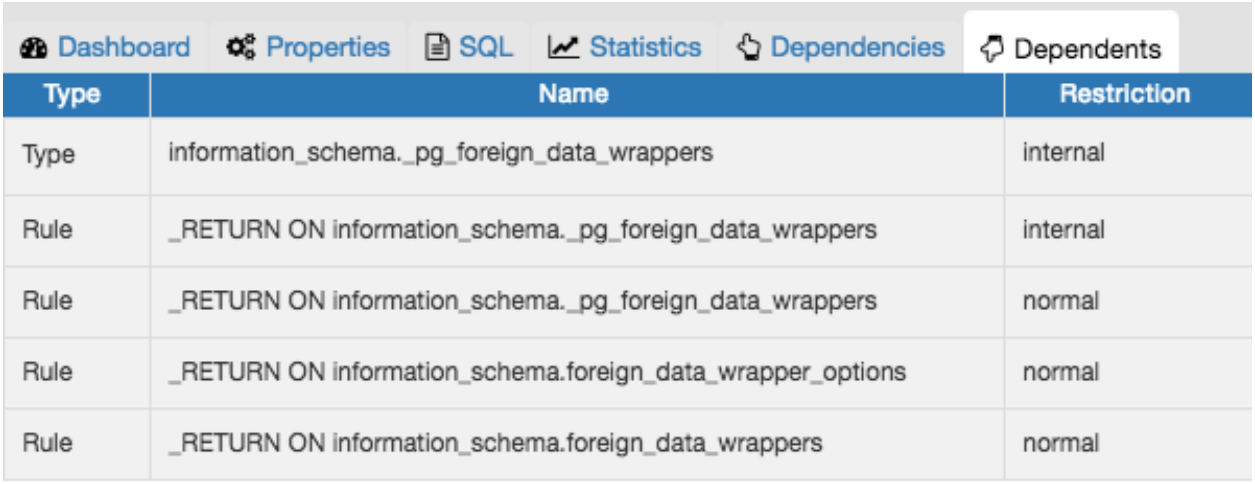


The *Dependencies* tab displays the objects on which the currently selected object depends. If a dependency is dropped, the object currently selected in the pgAdmin tree control will be affected. To ensure the integrity of the entire database structure, the database server makes sure that you do not accidentally drop objects that other objects depend on; you must use DROP CASCADE to remove an object with a dependency.

The *Dependencies* table displays the following information:

- The *Type* field specifies the parent object type.
- The *Name* field specifies the identifying name of the parent object.
- **The *Restriction* field describes the dependency relationship between the currently selected object and the parent.**
  - If the field is *auto*, the selected object can be dropped separately from the parent object, and will be dropped if the parent object is dropped.
  - If the field is *internal*, the selected object was created during the creation of the parent object, and will be dropped if the parent object is dropped.
  - If the field is *normal*, the selected object can be dropped without dropping the parent object.
  - If the field is *blank*, the selected object is required by the system, and cannot be dropped.

Click the *Dependents* tab to continue.

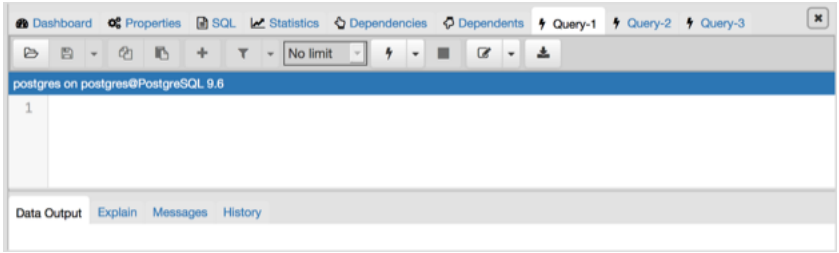


The *Dependents* tab displays a table of objects that depend on the object currently selected in the *pgAdmin* browser. A dependent object can be dropped without affecting the object currently selected in the *pgAdmin* tree control.

- The *Type* field specifies the dependent object type.
- The *Name* field specifies the identifying name for the dependent object.
- The *Database* field specifies the database in which the object resides.

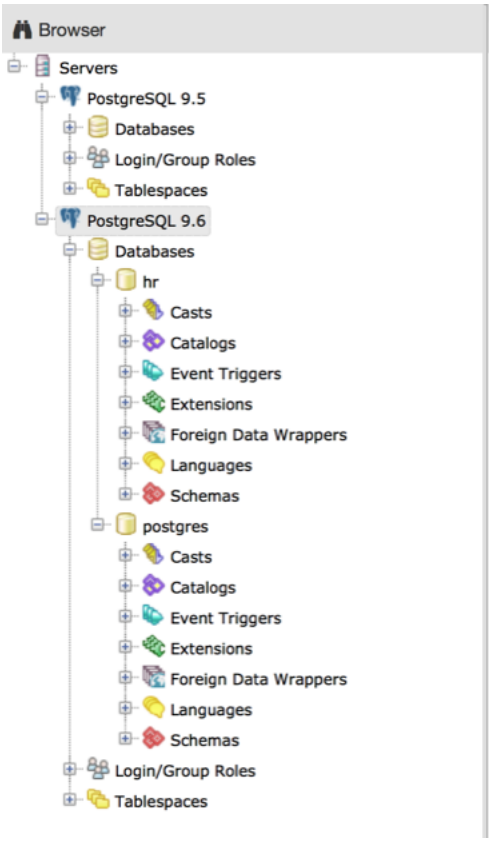
Feature Tabs

Additional *feature tabs* will open in the *pgAdmin* tabbed browser when you access the extended functionality offered by pgAdmin tools. For example, if you select the *Query tool* from *Tools* in the menu bar, pgAdmin will open the Query tool on a tab labeled *Query-1*. These feature tabs are not permanent and you can close them when you are finished using the tool. Like permanent tabs, these tabs may be repositioned.



1.8 The pgAdmin Tree Control

The left pane of the main window displays a tree control (the *pgAdmin* tree control) that provides access to the objects that reside on a server.



You can expand nodes in the tree control to view the database objects that reside on a selected server. The tree control expands to display a hierarchical view:

- Use the plus sign (+) to the left of a node to expand a segment of the tree control.
- Click the minus sign (-) to the left of a node to close that node.

Access context-sensitive menus by right-clicking on a node of the tree control to perform common tasks. Menus display options that include one or more of the following selections (options appear in alphabetical order):

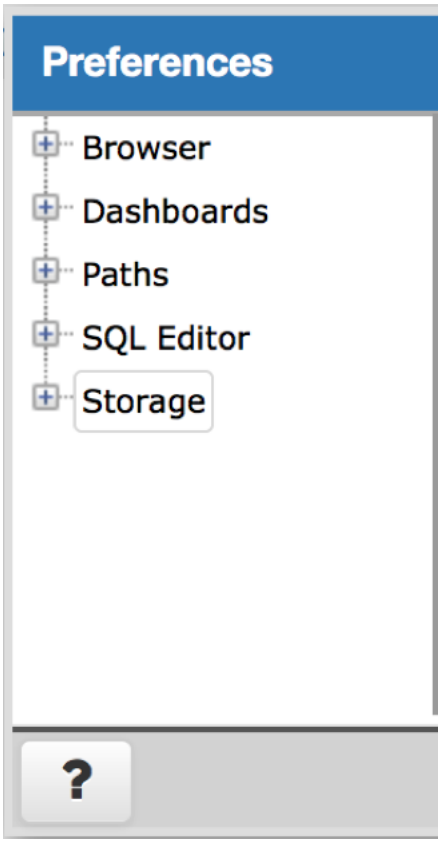
Option	Action
Add named restore point	Click to create and enter the name of a restore point.
Backup...	Click to open the *Backup...* dialog to backup database objects.
Backup Globals...	Click to open the *Backup Globals...* dialog to backup cluster objects.
Backup Server...	Click to open the *Backup Server...* dialog to backup a server.
Connect Server...	Click to open the *Connect to Server* dialog to establish a connection with a server.
Create	Click to access a context menu that provides context-sensitive selections. Your selection opens a *Create* dialog for creating a new object.
CREATE Script	Click to open the *Query tool* to edit or view the CREATE script.
Debugging	Click through to open the *Debug* tool or to select *Set breakpoint* to stop or pause a script execution.
Delete/Drop	Click to delete the currently selected object from the server.
Disconnect Database...	Click to terminate a database connection.
Disconnect Server...	Click to refresh the currently selected object.
Drop Cascade	Click to delete the currently selected object and all dependent objects from the server.
Debugging	Click to access the *Debugger* tool.
Grant Wizard	Click to access the *Grant Wizard* tool.
Maintenance...	Click to open the *Maintenance...* dialog to VACUUM, ANALYZE, REINDEX, or CLUSTER.
Properties...	Click to review or modify the currently selected object's properties.
Refresh...	Click to refresh the currently selected object.
Reload Configuration...	Click to update configuration files without restarting the server.
Restore...	Click to access the *Restore* dialog to restore database files from a backup.
View Data	Use the *View Data* option to access the data stored in a selected table with the *Data Output* tab of the *Query Tool*.

The context-sensitive menus associated with *Tables* and nested *Table* nodes provides additional display options (options appear in alphabetical order):

Option	Action
Import/Export...	Click open the *Import/Export...* dialog to import data to or export data from the selected table.
Reset Statistics	Click to reset statistics for the selected table.
Scripts	Click to open the *Query tool* to edit or view the selected script from the flyout menu.
Truncate	Click to remove all rows from a table.
Truncate Cascade	Click to remove all rows from a table and its child tables.
View First 100 Rows	Click to access a data grid that displays the first 100 rows of the selected table.
View Last 100 Rows	Click to access a data grid that displays the last 100 rows of the selected table.
View All Rows	Click to access a a data grid that displays all rows of the selected table.
View Filtered Rows...	Click to access the *Data Filter* popup to apply a filter to a set of data.

1.9 pgAdmin Preferences

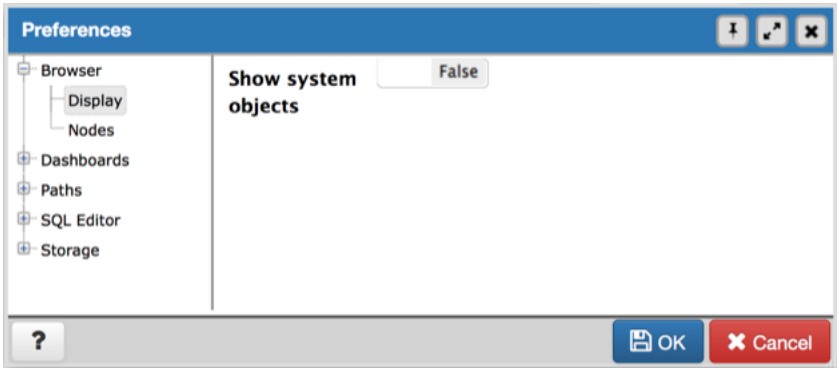
pgAdmin 4 has a selection of configuration options (*Preferences*) that you can use to customize your pgAdmin client. To open the *Preferences* dialog, select *Preferences* from the *File* menu.



The left pane of the *Preferences* dialog displays a tree control; each node of the tree control provides access to options that are related to the selected node.

- Use the plus sign (+) to the left of a node to expand a segment of the tree control.
- Click the minus sign (-) to the left of a node to close that node.

Expand the **Browser** node of the tree control to personalize your workspace.

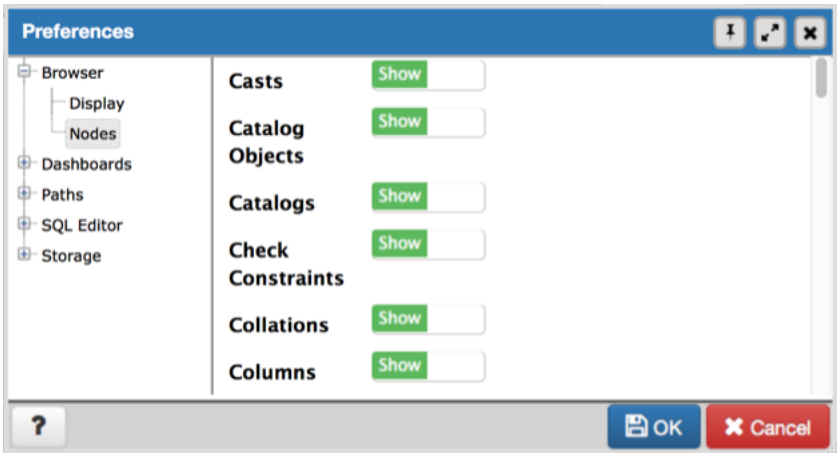


Use the options on the *Display* dialog to specify general display preferences:

Move the *Show system objects* switch to the *True* position to display system objects in the *pgAdmin* tree control. This

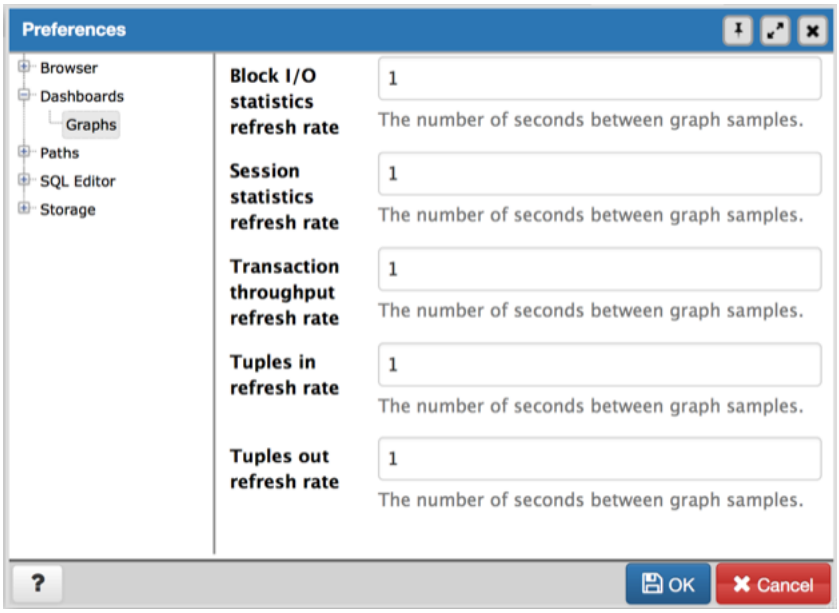
option instructs pgAdmin to display objects such as system schemas (e.g. pg_temp*) and system columns (e.g. xmin, ctid) in the tree control.

Use the options on the *Nodes* dialog to select the object types that will be displayed in the *pgAdmin* tree control.



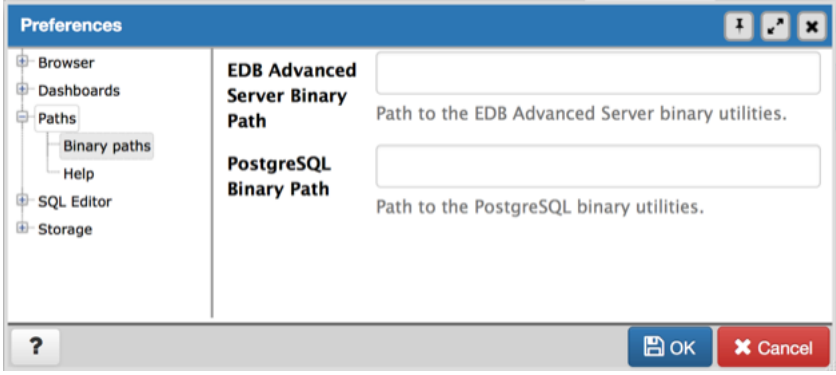
The right pane of the *Preferences* dialog displays a list of database objects. Slide the switch located next to each object to *Show* or *Hide* the database object. When querying system catalogs, you can reduce the number of object types displayed to increase speed.

Expand the **Dashboards** node to specify your graphing preferences.



Use the options on the *Graphs* dialog specify a refresh rate for statistics, transaction throughput and tuples. The rate you specify will affect a corresponding graph on the *Dashboard* tab of the *pgAdmin* tabbed browser.

Expand the **Paths** node to specify the locations of supporting files.

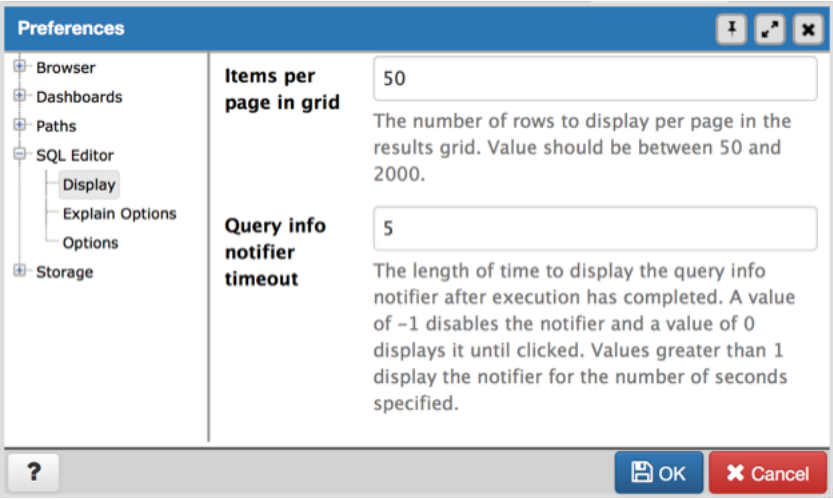


Use the fields in the *Binary paths* node to specify paths to the PostgreSQL binary utilities and EnterpriseDB Postgres Advanced Server binary utilities.

Use the *Help* dialog to customize links to support documentation.

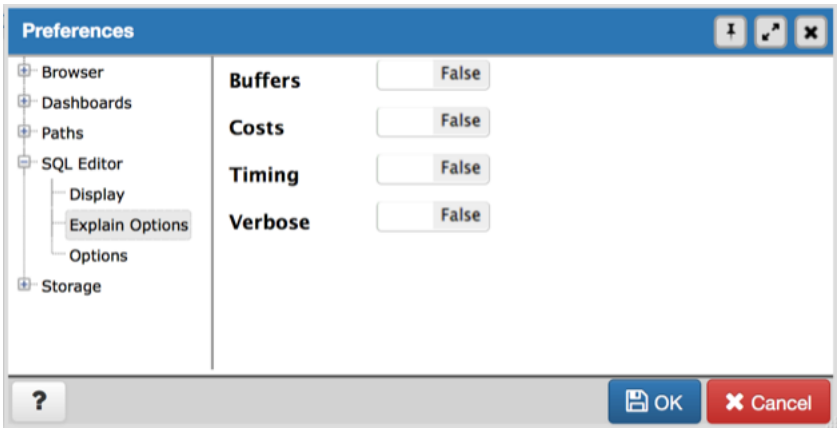
- Use the *EDB Advanced Server Help Path* to find a link path for EnterpriseDB Postgres Advanced Server documentation on the company website. This link is editable: substitute the applicable PostgreSQL version number for *$VERSION$*, or provide an alternate link path.
- Use the *PostgreSQL Help Path* to find a link path to the current set of PostgreSQL core documentation. This link is editable: substitute the applicable PostgreSQL version number for *$VERSION$*, or provide an alternate link path.

Expand the **SQL Editor** node to specify your preferences for the SQL Editor tool.



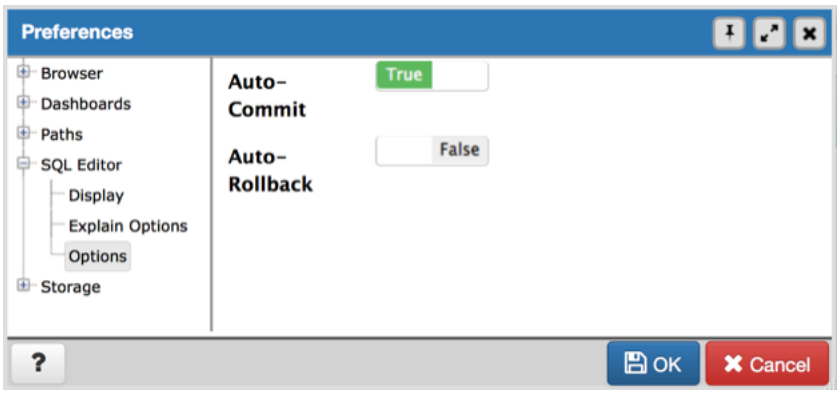
Use the *Display* dialog to specify your preferences for the SQL Editor display.

- Use the *Query info notifier timeout* to control the behaviour of the notifier that is displayed when query execution completes. A value of *-1* will disable the notifier, and a value of 0 will display it until clicked. If a positive value above zero is specified, the notifier will be displayed for the specified number of seconds. The default is *5*.



Use the options on the *Explain Options* dialog to specify the level of detail included in a graphical EXPLAIN.

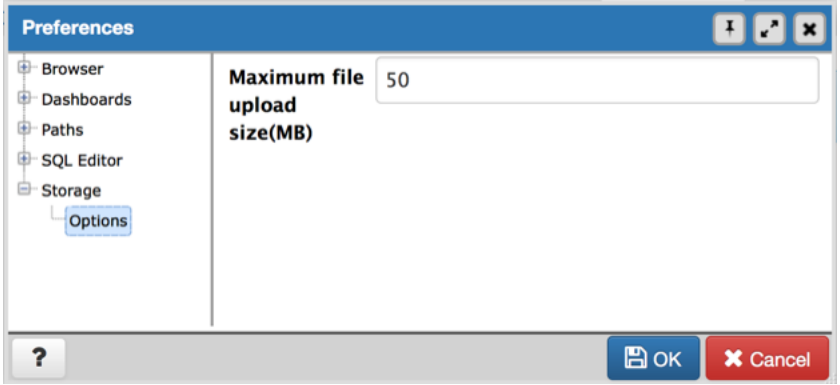
- Move the *Buffers* switch to the *True* position to include information on buffer usage.
- Move the *Costs* switch to the *True* position to include information on the estimated startup and total cost of each plan, as well as the estimated number of rows and the estimated width of each row.
- Move the *Timing* switch to the *True* position to include actual startup time and time spent in each node in the output.
- Move the *Verbose* switch to the *True* position to display additional information regarding the plan.



Use the options in the *Options* dialog to manage modifications to a SQL statement.

- Move the *Auto-Commit* switch to the *True* position to commit a SQL statement upon completion.
- Move the *Auto-Rollback* switch to the *True* to rollback a SQL statement to the beginning of the statement or to a prior rollback.

Expand the **Storage** node to specify a maximum file size for uploads.



Use the *Maximum file upload size(MB)* in the *Options* node of the **Storage** node to specify the maximum file size for an upload.

1.10 Keyboard Shortcuts

Keyboard shortcuts are provided in pgAdmin to allow easy access to specific functions.

Desktop Runtime

When running in the Desktop Runtime, the following keyboard shortcuts are available:

Shortcut (Windows/Linux)	Shortcut (Mac)	Function
Alt+Shift+A	Option+Shift+A	Display the runtime's About box
Alt+Shift+P	Option+Shift+U	Open the runtime preferences dialogue
Alt+Shift+U	Option+Shift+U	Open an arbitrary URL
Ctrl+Q	Cmd+Q	Quit
Ctrl+Plus	Cmd+Plus	Zoom in
Ctrl+Minus	Cmd+Minus	Zoom out

SQL Editors

When using the syntax-highlighting SQL editors, the following shortcuts are available:

Shortcut (Windows/Linux)	Shortcut (Mac)	Function
Alt+Left	Option+Left	Move to the beginning of the line
Alt+Right	Option+Right	Move to the end of the line
Ctrl+Alt+Left	Cmd+Option+Left	Move left one word
Ctrl+Alt+Right	Cmd+Option+Right	Move right one word
Ctrl+A	Cmd+A	Select all
Ctrl+C	Cmd+C	Copy selected text to the clipboard
Ctrl+R	Cmd+R	Redo last edit un-done
Ctrl+V	Cmd+V	Paste text from the clipboard
Ctrl+Z	Cmd+Z	Undo last edit
Ctrl+Plus	Cmd+Plus	Zoom in
Ctrl+Minus	Cmd+Minus	Zoom out

Query Tool

When using the Query Tool, the following shortcuts are available:

Shortcut (Windows/Linux)	Shortcut (Mac)	Function
F5	F5	Execute query
F7	F7	EXPLAIN query
Shift+F7	Shift+F7	EXPLAIN ANALYZE query
F8	F8	Execute query to CSV file
Alt+G	Alt+G	Jump (to line:column)
Ctrl+Space	Ctrl+Space	Auto-complete
Ctrl+F	Cmd+F	Find
Ctrl+G	Cmd+G	Find next
Shift+Ctrl+G	Shift+Cmd+G	Find previous
Shift+Ctrl+F	Shift+Cmd+F	Replace
Shift+Ctrl+R	Shift+Cmd+Option+F	Replace all

Before using pgAdmin to manage objects that reside on a server, you must define a connection to the server; for more information please see *Connecting to a Server*:

Contents:

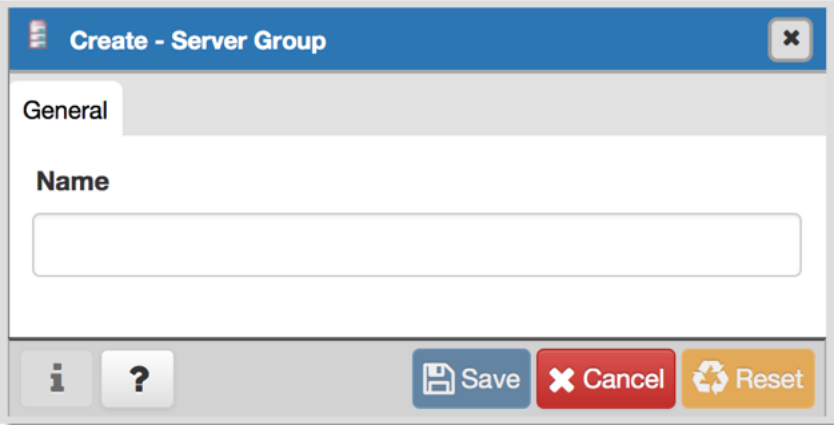
1.11 Connecting to a Server

Before you can use the pgAdmin client to manage the objects that reside on your Postgres server, you must define a connection to the server. You can (optionally) use the *Server Group* dialog to create server groups to organize the server connections within the tree control for easier management. To open the *Server Group* dialog, right-click on the *Servers* node of the tree control, and select *Server Group* from the *Create* menu.

Contents:

1.11.1 The Server Group Dialog

Use the *Server Group* dialog to add a new server group. Assign servers to server groups to simplify management of multiple servers. Server groups are displayed as part of the *pgAdmin* tree control.



Use the *Name* field on the *Server Group* dialog to specify a name that will identify the server group in the *pgAdmin* tree control.

- Click the *Save* button to save work.
- Click the *Cancel* button to exit without saving work.

- Click the *Reset* button to restore configuration parameters.

To create server connections in a server group, right click on the named server group and select the *Create* option to open the *Create - Server* dialog.

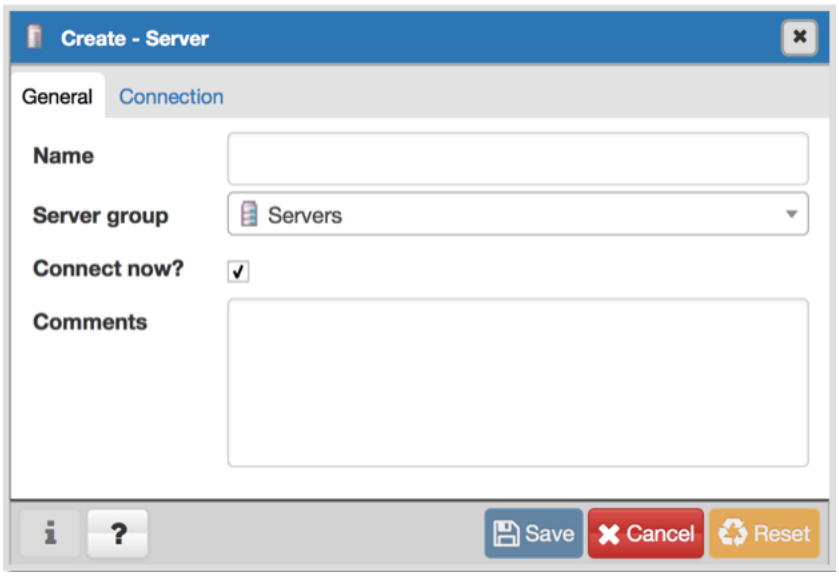
Use the fields on the *Server* dialog to define the connection properties for each new server that you wish to manage with pgAdmin. To open the *Server* dialog, right-click on the *Servers* node of the tree control, and select *Server* from the *Create* menu.

Contents:

1.11.2 The Server Dialog

Use the *Server* dialog to describe a connection to a server. Note: you must ensure the pg_hba.conf file of the server from which you are connecting allows connections from the host of the client.

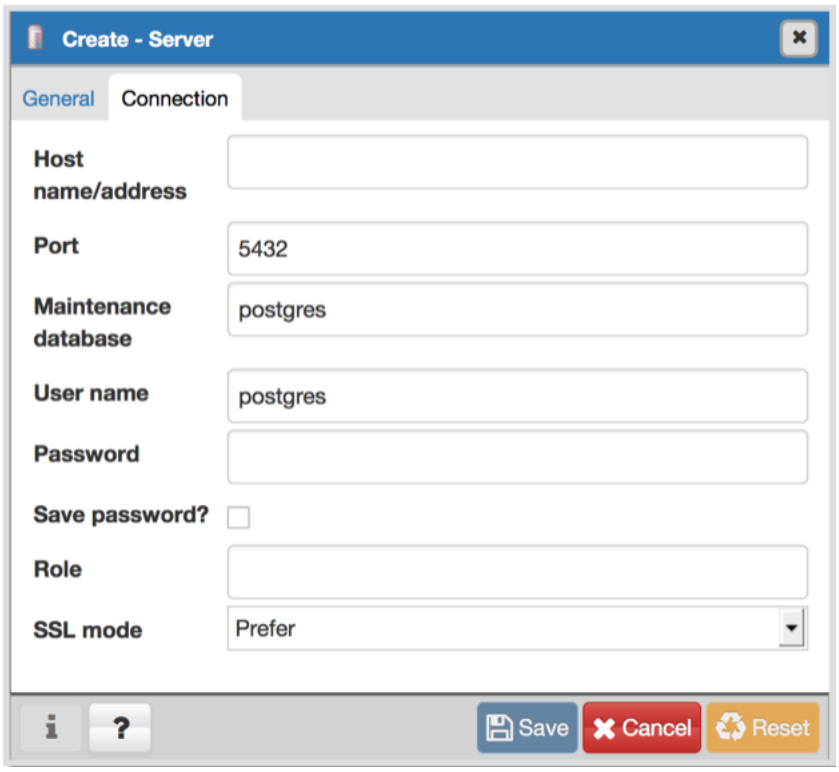
The *Server* dialog organizes the connection of a server through the following dialog tabs: *General*, and *Connection*.



Use the fields in the *General* tab to identify the server:

- Use the *Name* field to add a descriptive name for the server; the name specified will be displayed in the *pgAdmin* tree control of the client.
- Use the drop-down list box in the *Server group* field to specify the *pgAdmin* tree control parent node for the server.
- Uncheck the checkbox next to *Connect now?* to instruct pgAdmin not to attempt a connection upon completion of the dialog. The default enables connection.
- Provide a comment about the server in the *Comments* field.

Click the *Connection* tab to continue.



Use the fields in the *Connection* tab to configure a connection:

- Specify the IP address of the server host, or the fully qualified domain name in the *Host name/address* field. On Unix based systems, the address field may be left blank to use the default PostgreSQL Unix Domain Socket on the local machine, or may be set to an alternate path containing a PostgreSQL socket. If you enter a path, the path must begin with a "/".
- Enter the listener port number of the server host in the *Port* field. The default is *5432*.
- Use the *Maintenance database* field to specify the name of the initial database to which the client will connect. If you will be using pgAgent or adminpack objects, the pgAgent schema and adminpack objects should be installed on that database.
- Use the *User name* field to specify the name of a role that will be used when authenticating with the server.
- Use the *Password* field to provide a password that will be supplied when authenticating with the server.
- Check the box next to *Save password* to instruct pgAdmin to save the password for future use.
- Use the *Role* field to specify the name of a role that has privileges that will be conveyed to the client after authentication with the server. This selection allows you to connect as one role, and then assume the permissions of this specified role after the connection is established. Note that the connecting role must be a member of the role specified.
- Use the drop-down list box in the SSL field to select the type of SSL connection the server should use. For more information about using SSL encryption, see Section 31.18 of the Postgres documentation:

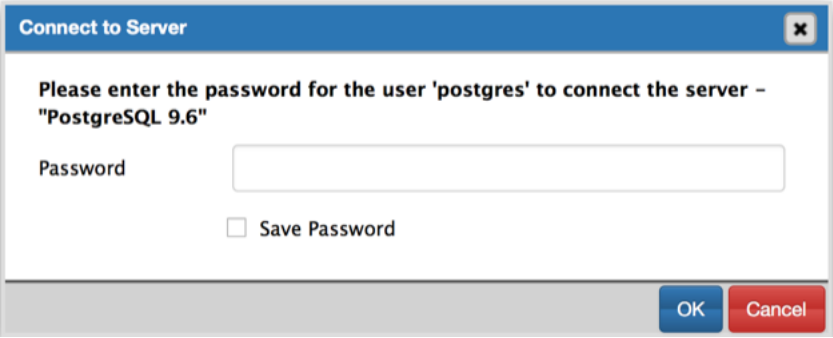
  http://www.postgresql.org/docs/9.5/static/libpq-ssl.html
- Click the *Save* button to save work.
- Click the *Cancel* button to exit without saving work.
- Click the *Reset* button to restore configuration parameters.

After defining a server connection, right-click on the server name, and select *Connect to server* to authenticate with the server, and start using pgAdmin to manage objects that reside on the server.

Contents:

1.11.3 Connect to server

Use the *Connect to Server* dialog to authenticate with a defined server and access the objects stored on the server through the pgAdmin tree control. To access the dialog, right click on the server name in the *pgAdmin* tree control, and select *Connect Server...* from the context menu.



Provide authentication information for the selected server:

- Use the *Password* field to provide the password of the user that is associated with the defined server.
- Check the box next to *Save Password* to instruct the server to save the password for future connections; if you save the password, you will not be prompted when reconnecting to the database server with this server definition.

The pgAdmin client displays a message in a green status bar in the lower right corner when the server connects successfully.

If you receive an error message while attempting a connection, verify that your network is allowing the pgAdmin host and the host of the database server to communicate. For detailed information about a specific error message, please see the *Connection Error* help page.

To review or modify connection details, right-click on the name of the server, and select *Properties...* from the context menu.

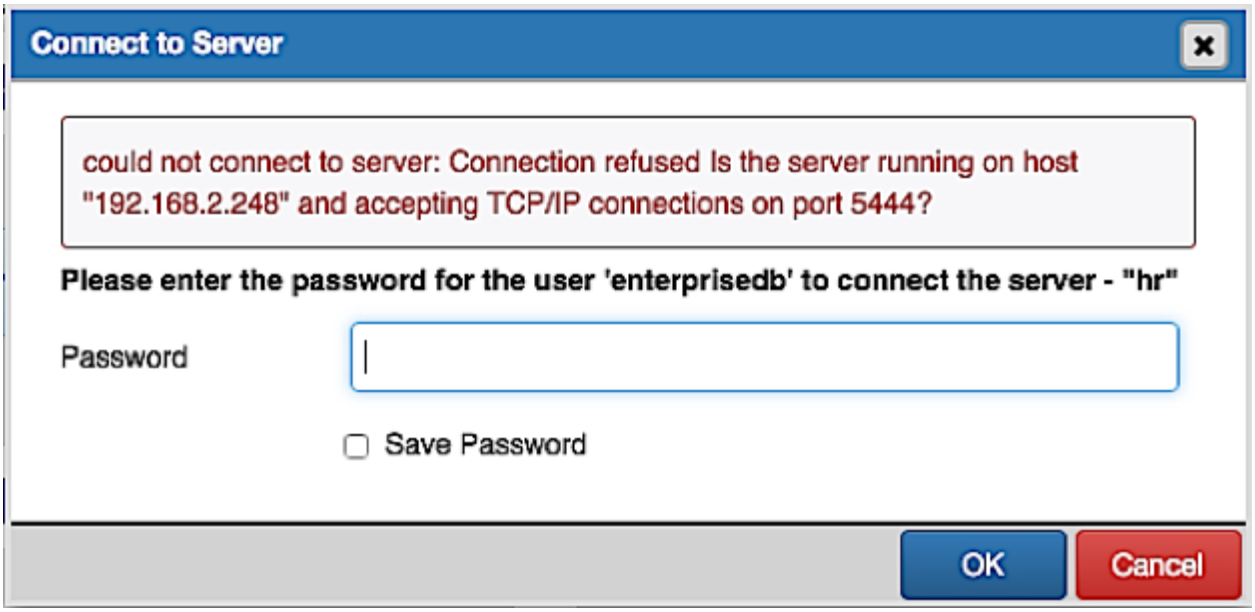
1.11.4 Connection error

When connecting to a PostgreSQL server, you may get an error message. If you encounter an error message, please review the message carefully; each error message attempts to incorporate the information you'll need to resolve the problem. For more details about specific errors, please locate the error message in the list below:

Connection to the server has been lost

This error message indicates that the connection attempt has taken longer than the specified threshold; there may be a problem with the connection properties provided on the *Server* dialog, network connectivity issues, or the server may not be running.

could not connect to Server: Connection refused



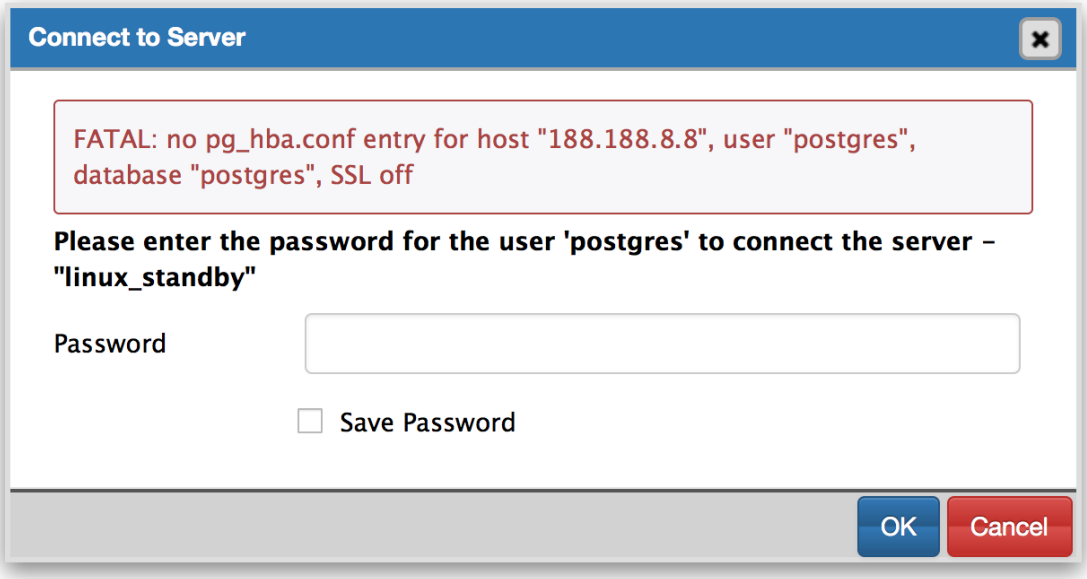
If pgAdmin displays this message, there are two possible reasons for this:

- the database server isn't running - simply start it.
- the server isn't configured to accept TCP/IP requests on the address shown.

For security reasons, a PostgreSQL server "out of the box" doesn't listen on TCP/IP ports. Instead, it must be enabled to listen for TCP/IP requests. This can be done by adding **tcpip = true** to the postgresql.conf file for Versions 7.3.x and 7.4.x, or **listen_addresses='*'** for Version 8.0.x and above; this will make the server accept connections on any IP interface.

For further information, please refer to the PostgreSQL documentation about runtime configuration.

FATAL: no pg_hba.conf entry



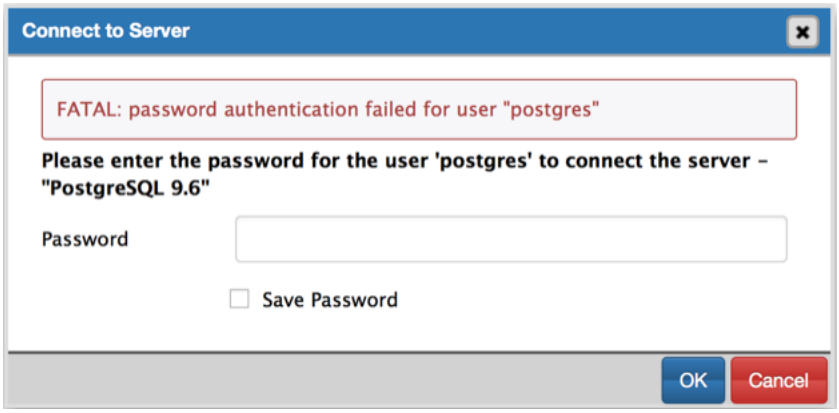
If pgAdmin displays this message when connecting, your server can be contacted correctly over the network, but is not configured to accept your connection. Your client has not been detected as a legal user for the database.

To connect to a server, the pg_hba.conf file on the database server must be configured to accept connections from the host of the pgAdmin client. Modify the pg_hba.conf file on the database server host, and add an entry in the form:

- **host template1 postgres 192.168.0.0/24 md5** for an IPV4 network
- **host template1 postgres ::ffff:192.168.0.0/120 md5** for an IPV6 network

For more information, please refer to the PostgreSQL documentation about client authentication.

FATAL: password authentication failed



- The *password authentication failed for user* error message indicates there may be a problem with the password you entered. Retry the password to confirm you entered it correctly. If the error message returns, make sure that you have the correct password, that you are authorized to access the server, and that the access has been correctly configured in the server's postgresql.conf configuration file.

CHAPTER

TWO

MANAGING CLUSTER LEVEL OBJECTS

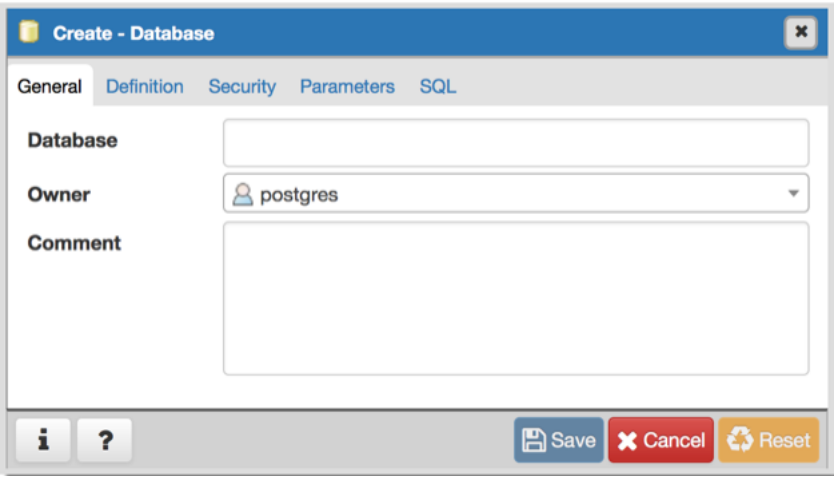
Some object definitions reside at the cluster level; pgAdmin 4 provides dialogs that allow you to create these objects, manage them, and control their relationships to each other. To access a dialog that allows you to create a database object, right-click on the object type in the pgAdmin tree control, and select the *Create* option for that object. For example, to create a new database, right-click on the *Databases* node, and select *Create Database...*

Contents:

2.1 The Database Dialog

Use the *Database* dialog to define or modify a database. To create a database, you must be a database superuser or have the CREATE privilege.

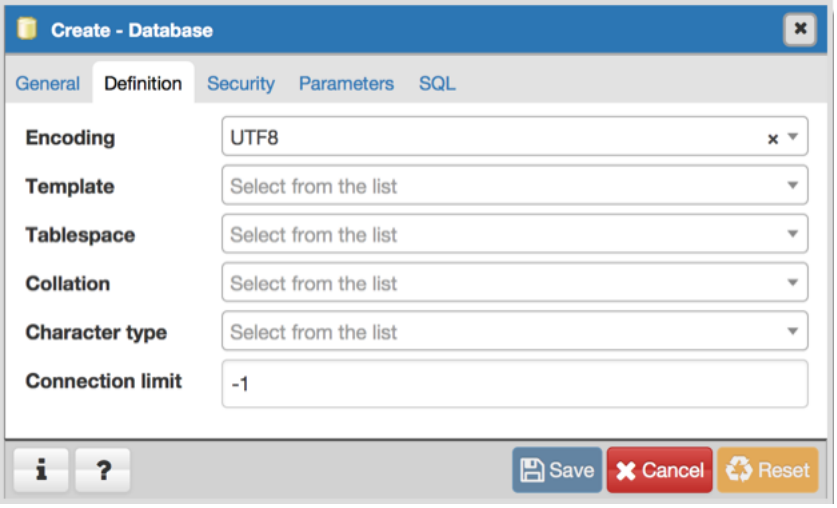
The *Database* dialog organizes the development of a database through the following dialog tabs: *General*, *Definition*, *Security*, and *Parameters*. The *SQL* tab displays the SQL code generated by dialog selections.



Use the fields in the *General* tab to identify the database:

- Use the *Database* field to add a descriptive name for the database. The name will be displayed in the *pgAdmin* tree control.
- Select the owner of the database from the drop-down listbox in the *Owner* field.
- Store notes about the database in the *Comment* field.

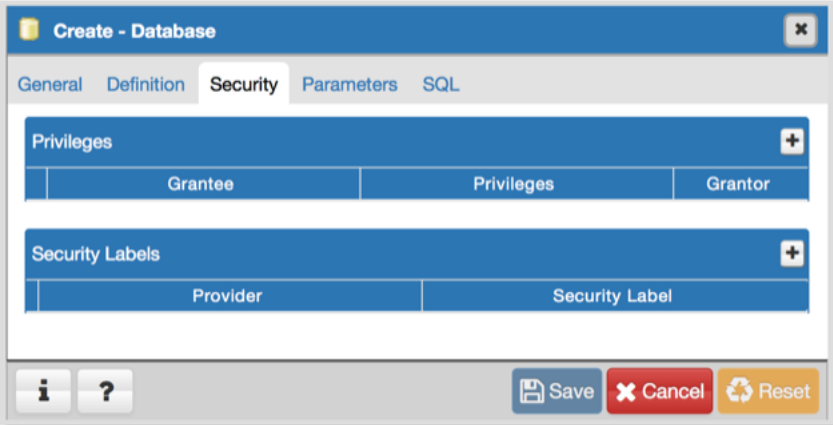
Click the *Definition* tab to continue.



Use the *Definition* tab to set properties for the database:

- Select a character set from the drop-down listbox in the *Encoding* field. The default is *UTF8*.
- Select a template from the drop-down listbox in the *Template* field. If you do not specify a template, the database will use template1.
- Select a tablespace from the drop-down listbox in the *Tablespace* field. The selected tablespace will be the default tablespace used to contain database objects.
- Select the collation order from the drop-down listbox in the *Collation* field.
- Select the character classification from the drop-down listbox in the *Character Type* field. This affects the categorization of characters, e.g. lower, upper and digit. The default, or a blank field, uses the character classification of the template database.
- Specify a connection limit in the *Connection Limit* field to configure the maximum number of connection requests. The default value (*-1*) allows unlimited connections to the database.

Click the *Security* tab to continue.



Use the *Security* tab to assign privileges and define security labels.

Use the *Privileges* panel to assign privileges to a role. Click the *Add* icon (+) to set privileges for database objects:

- Select the name of the role from the drop-down listbox in the *Grantee* field.
- Click inside the *Privileges* field. Check the boxes to the left of one or more privileges to grant the selected privilege to the specified user.

- Select the name of the role from the drop-down listbox in the *Grantor* field. The default grantor is the owner of the database.

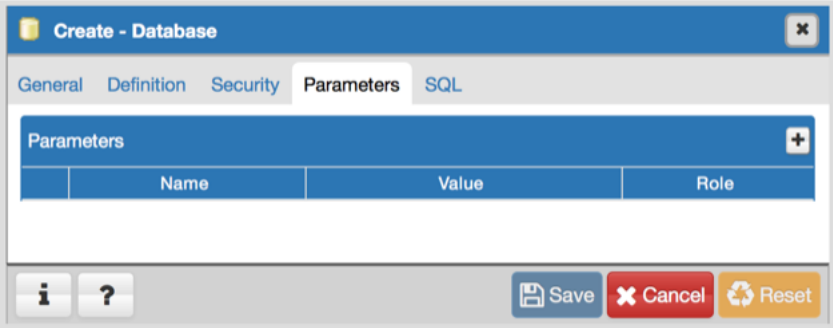
Click add to set additional privileges; to discard a privilege, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Use the *Security Labels* panel to define security labels applied to the database. Click the *Add* icon (+) to add each security label selection:

- Specify a security label provider in the *Provider* field. The named provider must be loaded and must consent to the proposed labeling operation.
- Specify a a security label in the *Security Label* field. The meaning of a given label is at the discretion of the label provider. PostgreSQL places no restrictions on whether or how a label provider must interpret security labels; it merely provides a mechanism for storing them.

To discard a security label, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Click the *Parameters* tab to continue.



Use the *Parameters* tab to set parameters for the database. Click the *Add* icon (+) to add each parameter:

- Use the drop-down listbox in the *Name* field to select a parameter.
- Use the *Value* field to set a value for the parameter.
- Use the drop-down listbox next to *Role* to select a role to which the parameter setting specified will apply.

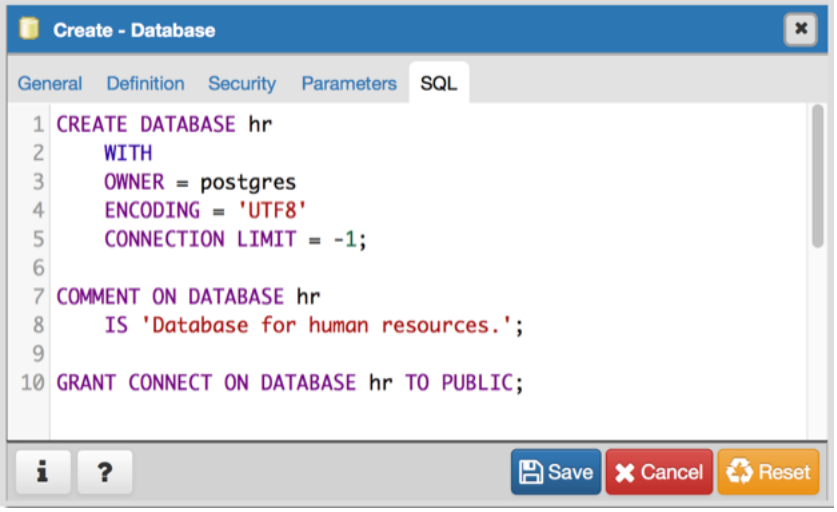
Follow these steps to add additional parameter value definitions; to discard a parameter, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Click the *SQL* tab to continue.

Your entries in the *Database* dialog generate a SQL command (see an example below). Use the *SQL* tab for review; revisit or switch tabs to make any changes to the SQL command.

Example

The following is an example of the sql command generated by user selections in the *Database* dialog:



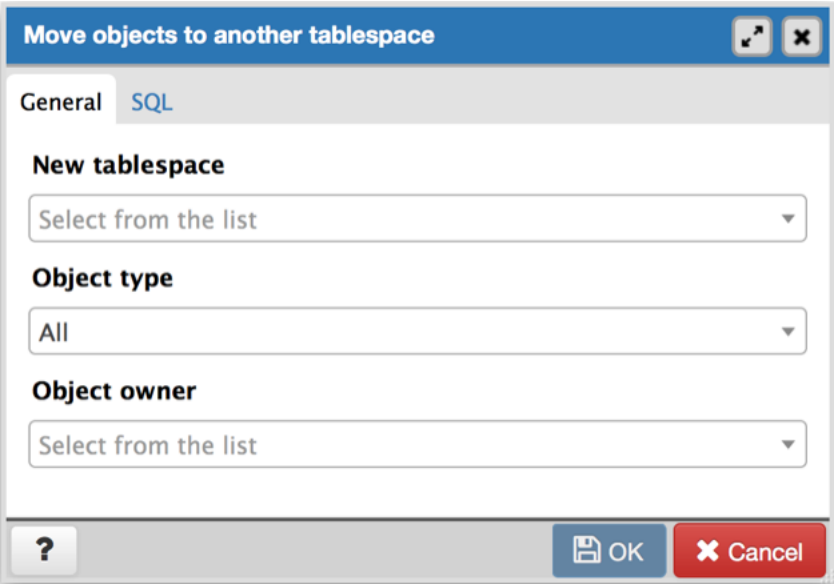
The example creates a database named *hr* that is owned by *postgres*. It allows unlimited connections, and is available to all authenticated users.

- Click the *Info* button (i) to access online help. View context-sensitive help in the *Tabbed browser*, where a new tab displays the PostgreSQL core documentation.
- Click the *Save* button to save work.
- Click the *Cancel* button to exit without saving work.
- Click the *Reset* button to restore configuration parameters.

2.2 The Move Objects Dialog

Use the *Move Objects* dialog to to move database objects from one tablespace to another tablespace.

The *Move Objects* dialog organizes the movement of database objects with the *General* tab; the *SQL* tab displays the SQL code generated by dialog selections.



Use the fields in the *General* tab to identify the items that will be moved and the tablespace to which they will be moved:

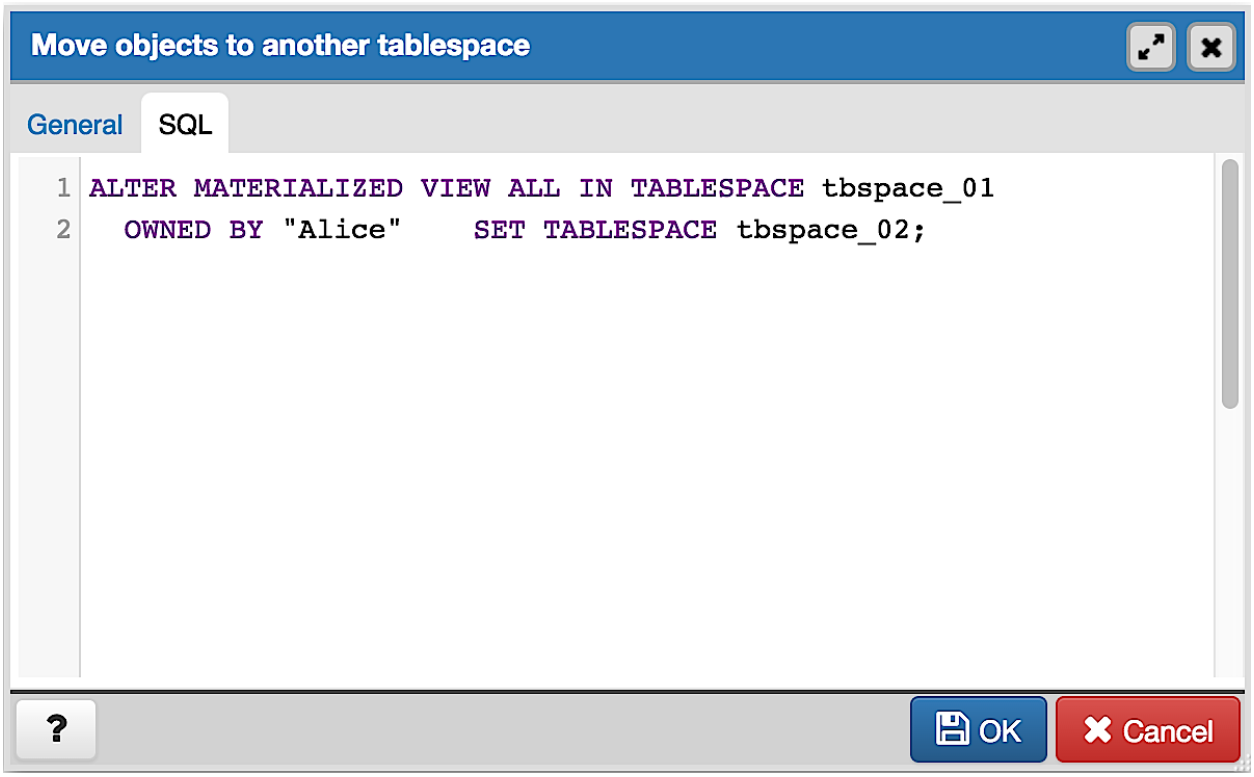
- Use the *New tablespace* drop-down listbox to select a pre-existing tablespace to which the object will be moved. (To create a tablespace, use the *Tablespace* dialog; access the dialog by right clicking *Tablespaces* in the *pgAdmin* tree control and selecting *Create Tablespace...* from the context-menu.)
- Use the *Object type* drop-down listbox to select from the following:
  - Select *All* to move all tables, indexes, and materialized views from the current tablespace (currently selected in the *pgAdmin* tree control) to the new tablespace.
  - Select *Tables* to move tables from the current tablespace to the new tablespace.
  - Select *Indexes* to move indexes from the current tablespace to the new tablespace.
  - Select *Materialized views* to move materialized views from the current tablespace to the new tablespace.
- Use the *Object owner* drop-down listbox to select the role that owns the objects selected in the *Object type* field. This field is optional.

Click the *SQL* tab to continue.

Your entries in the *Move Objects* dialog generate a SQL command (see an example below). Use the *SQL* tab for review; revisit the *General* tab to modify the SQL command.

Example

The following is an example of the sql command generated by user selections in the *Move Objects* dialog:



The example shown demonstrates moving materialized views owned by Alice from tablespace *tbspace_01* to *tbspace_02*.

- Click the *Help* button (?) to access online help.
- Click the *OK* button to save work.

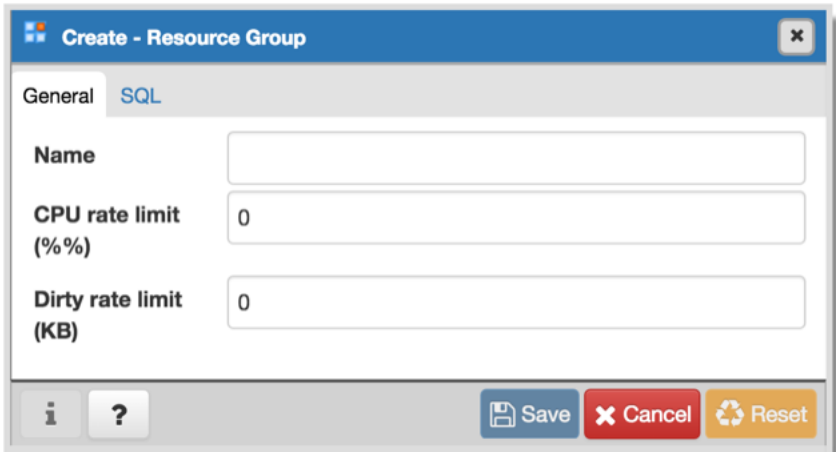
- Click the *Cancel* button to exit without saving work.

2.3 The Resource Group Dialog

Use the *Resource Group* dialog to create a resource group and set values for its resources. A resource group is a named, global group on which various resource usage limits can be defined. The resource group is accessible from all databases in the cluster. To use the *Resource Group* dialog, you must have superuser privileges. Please note that resource groups are supported when connected to EDB Postgres Advanced Server; for more information about using resource groups, please see the EDB Postgres Advanced Server Guide, available at:

http://www.enterprisedb.com/

The *Resource Group* dialog organizes the development of a resource group through the *General* dialog tab. The *SQL* tab displays the SQL code generated by dialog selections.



Use the fields in the *General* tab to specify resource group parameters:

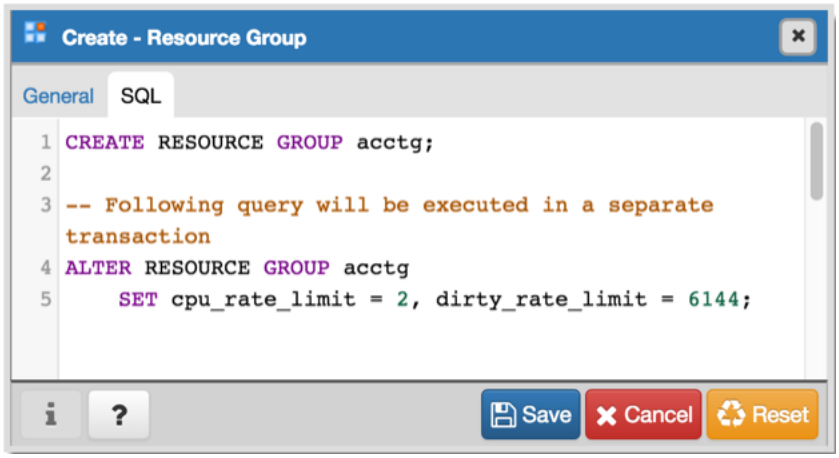
- Use the *Group Name* field to add a descriptive name for the resource group. This name will be displayed in the *pgAdmin* tree control.
- Use the *CPU Rate Limit (%)* field to set the value of the CPU rate limit resource type assigned to the resource group. The valid range for a CPU rate limit is from 0 to 1.67772e+07. The default value is 0.
- Use the *Dirty Rate Limit (KB)* field to set the value of the dirty rate limit resource type assigned to the resource group. The valid range for a dirty rate limit is from 0 to 1.67772e+07. The default value is 0.

Click the *SQL* tab to continue.

Your entries in the *Resource Group* dialog generate a SQL command (see an example below). Use the *SQL* tab for review; revisit the *General* tab to make any changes to the SQL command.

Example

The following is an example of the sql command generated by selections made in the *Resource Group* dialog:



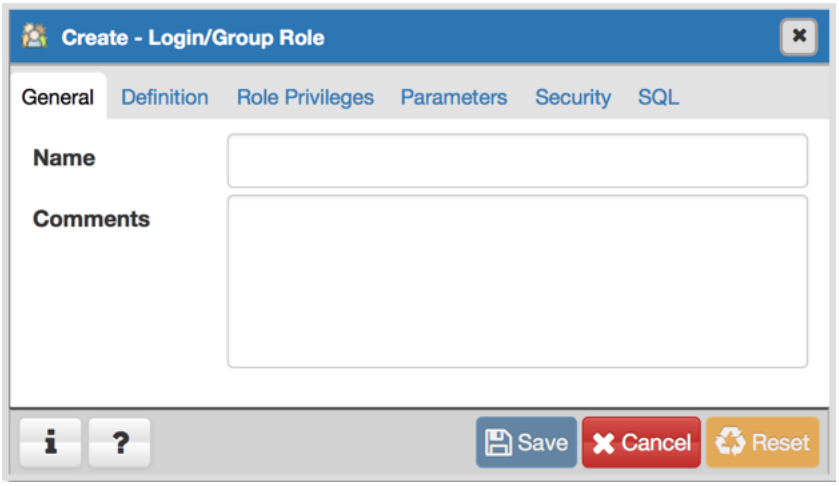
The example creates a resource group named *acctg* that sets *cpu_rate_limit* to *2*, and *dirty_rate_limit* to *6144*.

- Click the *Info* button (i) to access online help. View context-sensitive help in the *Tabbed browser*, where a new tab displays the PostgreSQL core documentation.
- Click the *Save* button to save work.
- Click the *Cancel* button to exit without saving work.
- Click the *Reset* button to restore configuration parameters.

2.4 The Login/Group Role Dialog

Use the *Login/Group Role* dialog to define a role. A role may be an individual user (with or without login privileges) or a group of users. Note that roles defined at the cluster level are shared by all databases in the cluster.

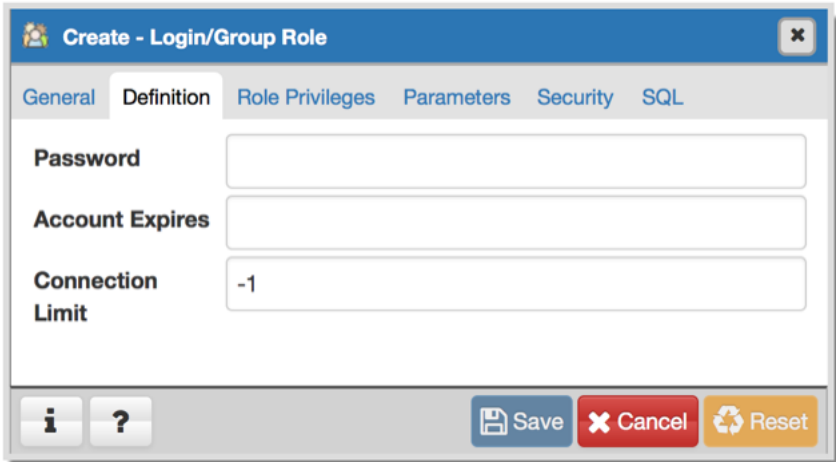
The dialog *Login/Group Role* organizes the creation of roles through the following dialog tabs: *General*, *Definition*, *Role Privileges*, *Parameters*, and *Security*. The *SQL* tab displays the SQL code generated by dialog selections.



Use the fields in the *General* tab to identify the role.

- Use the *Name* field to provide the name of the role. The name will be displayed in the *pgAdmin* tree control.
- Provide a note about the role in the *Comments* field.

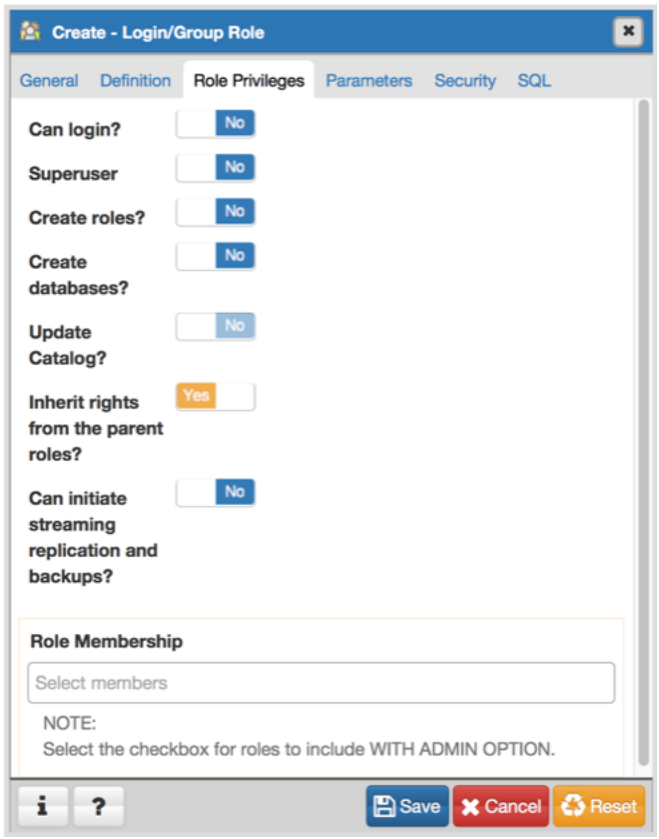
Click the *Definition* tab to continue.



Use the *Definition* tab to set a password and configure connection rules:

- Provide a password that will be associated with the role in the *Password* field.
- Provide an expiration date for the password in the *Account Expires* field (the role does not expire). Provide the date in a mm/dd/yyyy format. The expiration date is not enforced when a user logs in with a non-password-based authentication method.
- If the role is a login role, specify how many concurrent connections the role can make in the *Connection Limit* field. The default value (*-1*) allows unlimited connections.

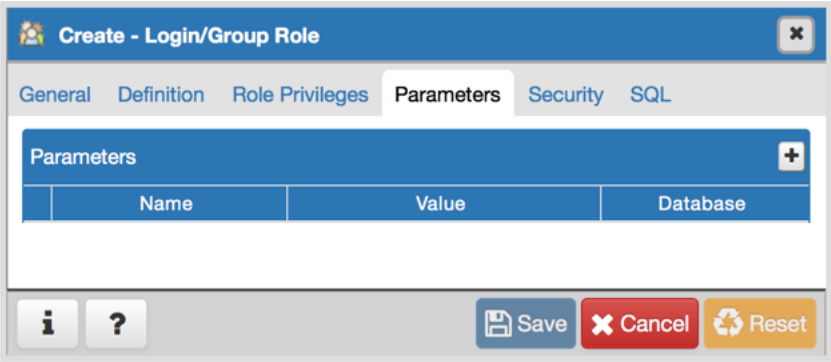
Click the *Role Privileges* tab to continue.



Use the *Role Privileges* tab to grant privileges to the role.

- Move the *Can login?* switch to the *Yes* position if the role has login privileges. The default value is *No*.
- Move the *Superuser* switch to the *Yes* position if the role is a superuser within the database. The default value is *No*.
- Move the *Create roles?* switch to the *Yes* position to specify whether a role is permitted to create roles. A role with this privilege can alter and drop roles. The default value is *No*.
- Move the *Create databases* switch to the *Yes* position to control whether a role can create databases. The default value is *No*.
- The *Update catalog?* switch is disabled until the role is given superuser privileges. Move the *Update catalogs?* switch to the *No* position to control whether a role can update catalogs. The default value is *Yes* when the *Superuser* switch is in the *Yes* position.
- Move the *Inherit rights from the parent roles?* switch to the *No* position if a role does not inherit privileges. The default value is *Yes*.
- Move the *Can initiate streaming replication and backups?* switch to the *Yes* position to control whether a role can initiate streaming replication or put the system in and out of backup mode. The default value is *No*.
- Specify members of the role in the *Role Membership* field. Click inside the *Role Membership* field to select role names from a drop down list. Confirm each selection by checking the checkbox to the right of the role name; delete a selection by clicking the *x* to the left of the role name. Membership conveys the privileges granted to a role to each of its members.

Click the *Parameters* tab to continue.

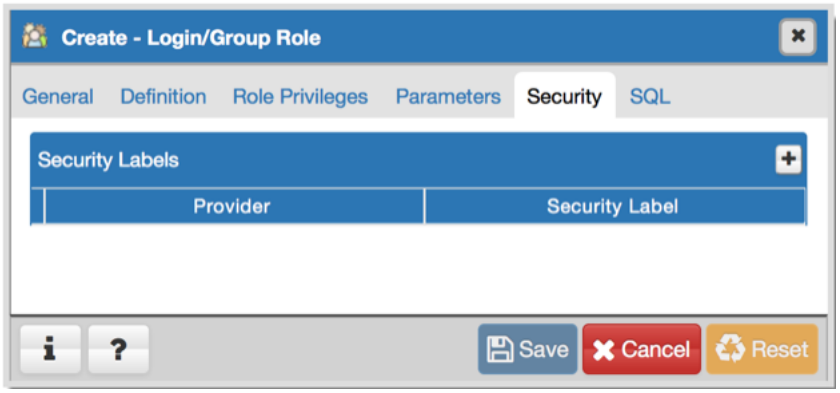


Use the fields on the *Parameters* tab to set session defaults for a selected configuration parameter when the role is connected to a specified database. This tab invokes the ALTER ROLE... SET configuration_parameter syntax. Click the *Add* icon (+) to create a parameter.

- Use the drop-down listbox in the *Name* field to select a parameter.
- Use the *Value* field to specify a value for the parameter.
- Use the drop-down listbox in the *Database* field to select a database.

Click the *Add* icon (+) to specify each additional parameter; to discard a parameter, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Click the *Security* tab to continue.



Use the *Security* tab to define security labels applied to the role. Click the *Add* icon (+) to add each security label selection.

- Specify a security label provider in the *Provider* field. The named provider must be loaded and must consent to the proposed labeling operation.
- Specify a a security label in the *Security Label* field. The meaning of a given label is at the discretion of the label provider. PostgreSQL places no restrictions on whether or how a label provider must interpret security labels; it merely provides a mechanism for storing them.

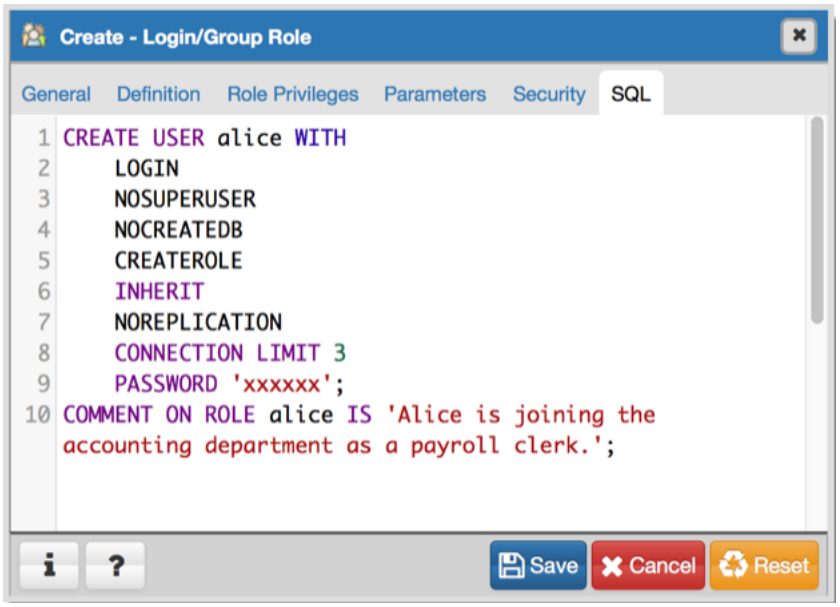
To discard a security label, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Click the *SQL* tab to continue.

Your entries in the *Login/Group Role* dialog generate a SQL command (see an example below). Use the *SQL* tab for review; revisit or switch tabs to make any changes to the SQL command.

Example

The following is an example of the sql command generated by user selections in the *Login/Group Role* dialog:



The example creates a login role named *alice* with *CREATE ROLE* privileges; the role is limited to 3 connections to the server at any given time.

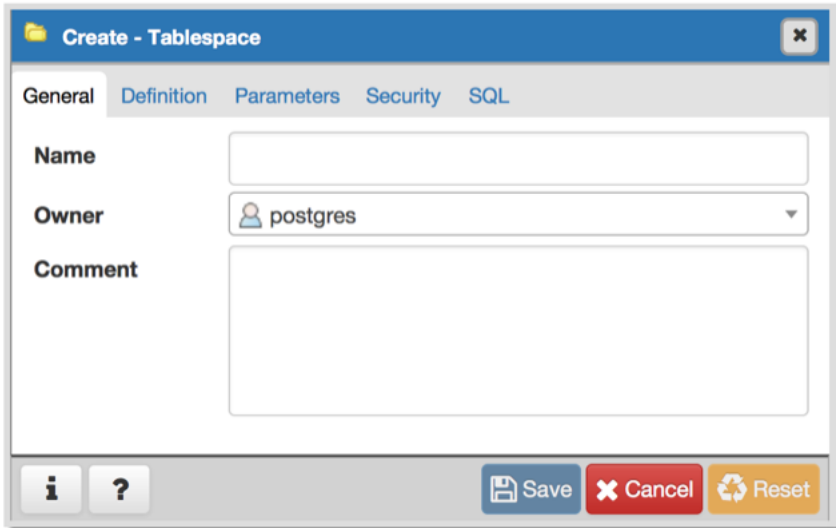
- Click the *Info* button (i) to access online help. View context-sensitive help in the *Tabbed browser*, where a new tab displays the PostgreSQL core documentation.

- Click the *Save* button to save work.
- Click the *Cancel* button to exit without saving work.
- Click the *Reset* button to restore configuration parameters.

2.5 The Tablespace Dialog

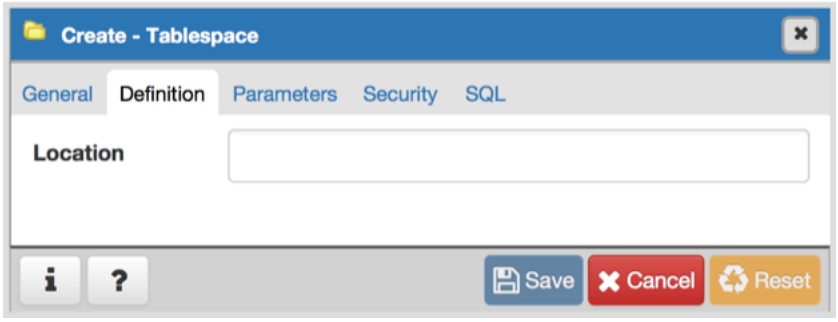
Use The *Tablespace* dialog to define a tablespace. A tablespace allows superusers to define an alternative location on the file system where the data files containing database objects (such as tables and indexes) reside. Tablespaces are only supported on systems that support symbolic links. Note that a tablespace cannot be used independently of the cluster in which it is defined.

The *Tablespace* dialog organizes the definition of a tablespace through the following tabs: *General*, *Definition*, *Parameters*, and *Security*. The *SQL* tab displays the SQL code generated by dialog selections.



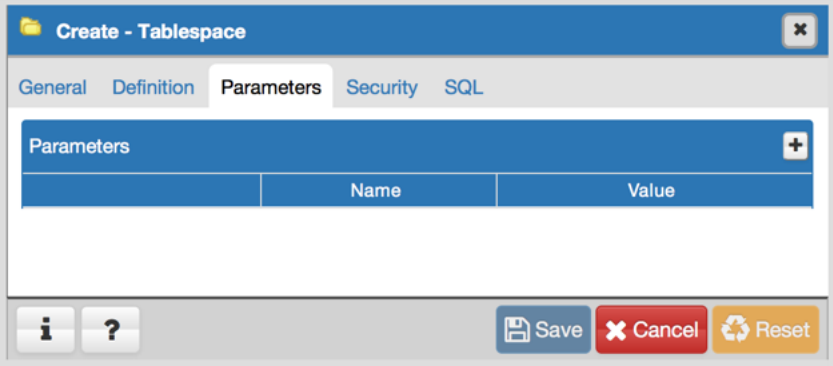
- Use the *Name* field to identify the tablespace with a descriptive name. The name cannot begin with pg_; these names are reserved for system tablespaces.
- Select the owner of the tablespace from the drop-down listbox in the *Owner* field.
- Store notes about the tablespace in the *Comment* field.

Click the *Definition* tab to continue.



- Use the *Location* field to specify an absolute path to a directory that will contain the tablespace.

Click the *Parameters* tab to continue.

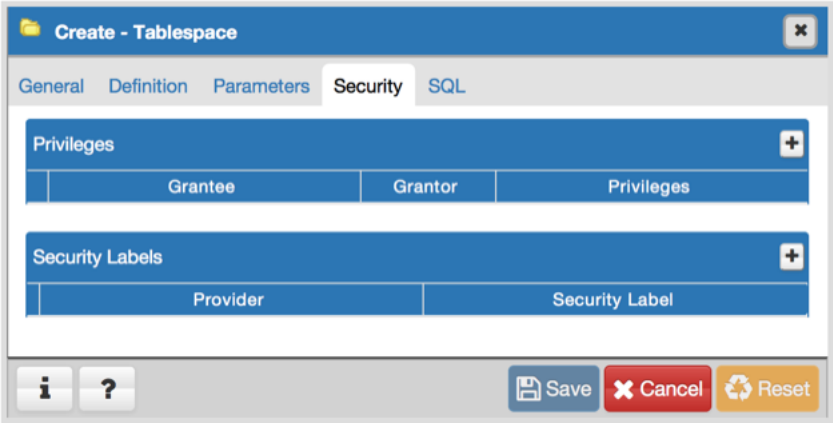


Use the *Parameters* tab to set parameters for the tablespace. Click the *Add* icon (+) to add a row to the table below.

- Use the drop-down listbox next to *Name* to select a parameter.
- Use the *Value* field to set a value for the parameter.

Click the *Add* icon (+) to specify each additional parameter; to discard a parameter, click the trash icon to the left of the row and confirm deletion in the *Delete Row* dialog.

Click the *Security* tab to continue.



Use the *Security* tab to assign privileges and define security labels for the tablespace.

Use the *Privileges* panel to assign security privileges. Click the *Add* icon (+) to assign a set of privileges:

- Select the name of the role from the drop-down listbox in the *Grantee* field.
- Select the name of the role from the drop-down listbox in the *Grantor* field. The default grantor is the owner of the owner of the tablespace.
- Click inside the *Privileges* field. Check the boxes to the left of one or more privileges to grant the selected privileges to the specified user.

Click the *Add* icon to assign additional sets of privileges; to discard a privilege, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Use the *Security Labels* panel to define security labels applied to the tablespace. Click the *Add* icon (+) to add each security label selection:

- Specify a security label provider in the *Provider* field. The named provider must be loaded and must consent to the proposed labeling operation.
- Specify a a security label in the *Security Label* field. The meaning of a given label is at the discretion of the label provider. PostgreSQL places no restrictions on whether or how a label provider must interpret security labels; it

merely provides a mechanism for storing them.

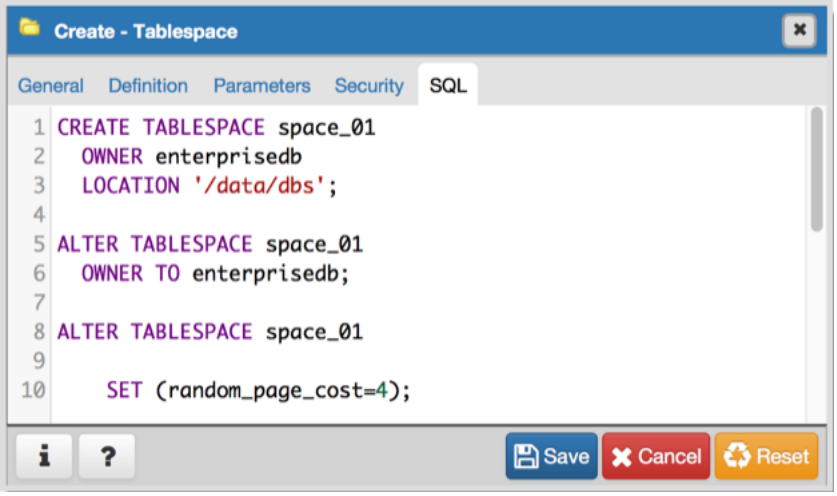
To discard a security label, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Click the *SQL* tab to continue.

Your entries in the *Tablespace* dialog generate a SQL command (see an example below). Use the *SQL* tab for review; revisit or switch tabs to make any changes to the SQL command.

Example

The following is an example of the sql command generated by user selections in the *Tablespace* dialog:



The example shown demonstrates creating a tablespace named *space_01*. It has a *random_page_cost* value equal to *4*.

- Click the *Info* button (i) to access online help. View context-sensitive help in the *Tabbed browser*, where a new tab displays the PostgreSQL core documentation.
- Click the *Save* button to save work.
- Click the *Cancel* button to exit without saving work.
- Click the *Reset* button to restore configuration parameters.

CHAPTER

THREE

MANAGING DATABASE OBJECTS

pgAdmin 4 provides simple but powerful dialogs that you can use to design and create database objects. Each dialog contains a series of tabs that you use to describe the object that will be created by the dialog; the SQL tab displays the SQL command that the server will execute when creating the object.

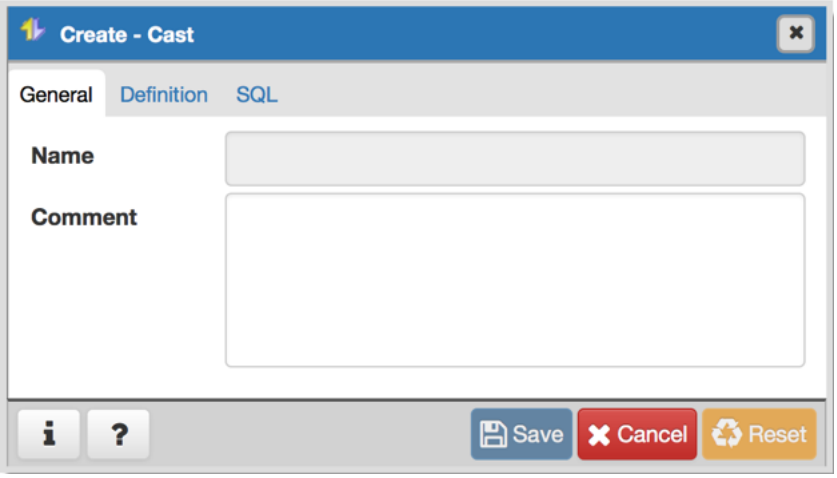
To access a dialog that allows you to create a database object, right-click on the object type in the pgAdmin tree control, and select the *Create* option for that object. For example, to create a new cast, right-click on the *Casts* node, and select *Create Cast...*

Contents:

3.1 The Cast Dialog

Use the *Cast* dialog to define a cast. A cast specifies how to convert a value from one data type to another.

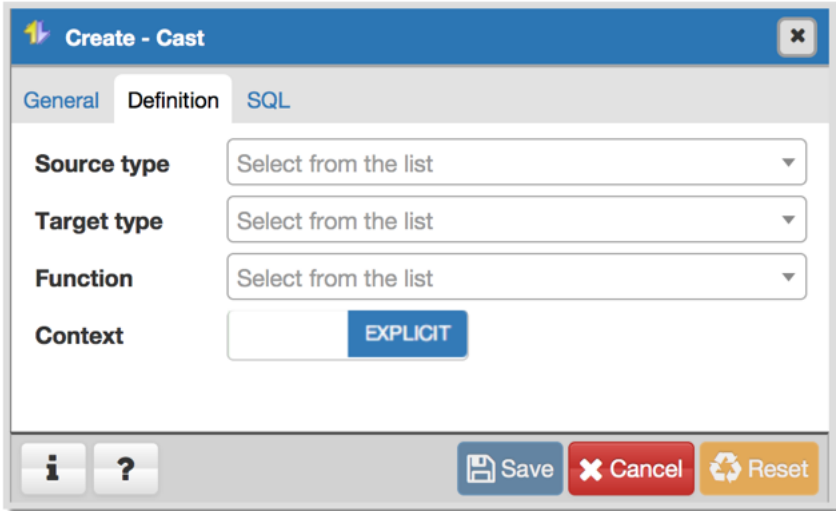
The *Cast* dialog organizes the development of a cast through the following dialog tabs: *General* and *Definition*. The *SQL* tab displays the SQL code generated by dialog selections.



Use the fields in the *General* tab to identify the cast:

- The *Name* field is disabled. The name that will be displayed in the *pgAdmin* tree control is the *Source* type concatenated with the *Target* type, and is generated automatically when you make selections on the *Cast* dialog *Definition* tab.
- Store notes about the cast in the *Comment* field.

Click the *Definition* tab to continue.



Use the fields in the *Definition* tab to define parameters:

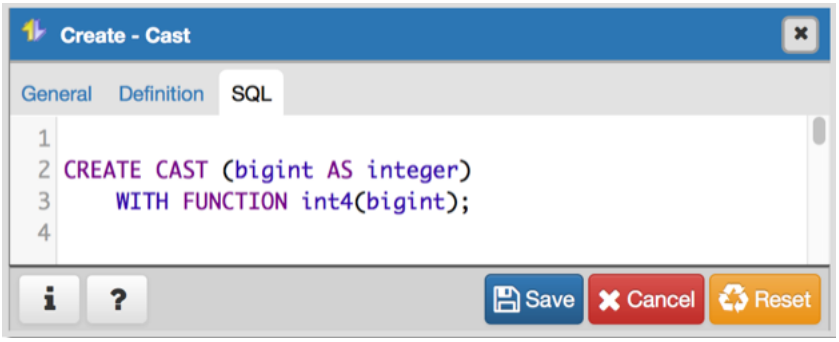
- Use the drop-down listbox next to *Source type* to select the name of the source data type of the cast.
- Use the drop-down listbox next to *Target type* to select the name of the target data type of the cast.
- Use the drop-down listbox next to *Function* to select the function used to perform the cast. The function's result data type must match the target type of the cast.
- Move the *Context* switch to the *Implicit* position if the cast is implicit. By default, a cast can be invoked only by an explicit cast request. If the cast is marked *Implicit* then it can be invoked implicitly in any context, whether by assignment or internally in an expression.

Click the *SQL* tab to continue.

Your entries in the *Cast* dialog generate a SQL command (see an example below). Use the *SQL* tab for review; revisit or switch tabs to make any changes to the SQL command.

Example

The following is an example of the sql command generated by user selections in the *Cast* dialog:



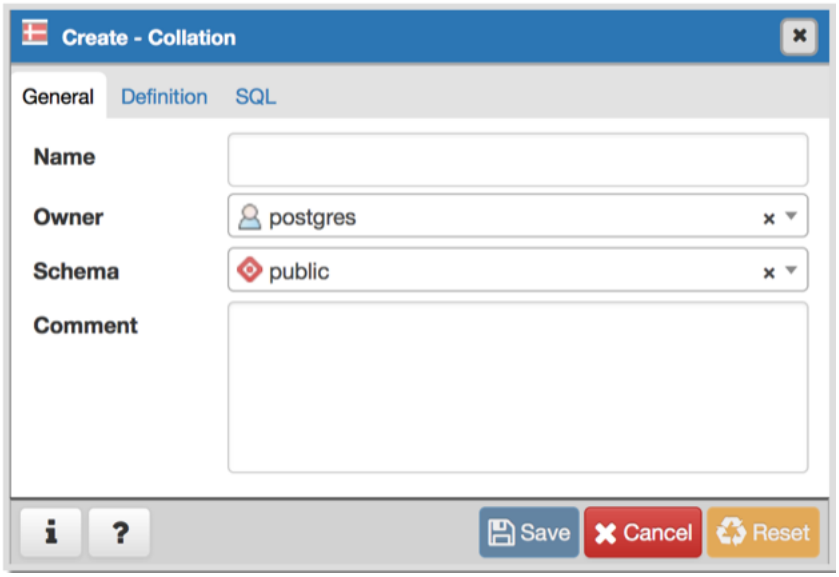
The cast uses a function named *int4(bigint)* to convert a biginteger data type to an integer.

- Click the *Info* button (i) to access online help. View context-sensitive help in the *Tabbed browser*, where a new tab displays the PostgreSQL core documentation.
- Click the *Save* button to save work.
- Click the *Cancel* button to exit without saving work.
- Click the *Reset* button to restore configuration parameters.

3.2 The Collation Dialog

Use the *Collation* dialog to define a collation. A collation is an SQL schema object that maps a SQL name to operating system locales. To create a collation, you must have a CREATE privilege on the destination schema.

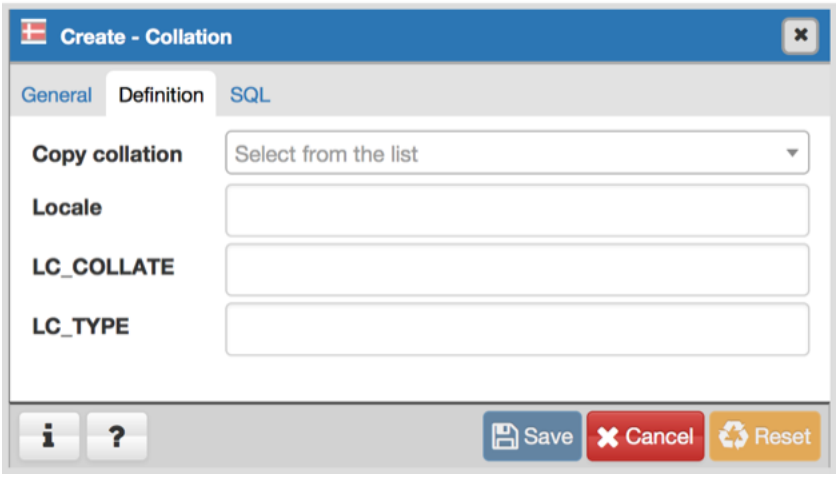
The *Collation* dialog organizes the development of a collation through the following dialog tabs: *General* and *Definition*. The *SQL* tab displays the SQL code generated by dialog selections.



Use the fields in the *General* tab to identify the collation:

- Use the *Name* field to provide a name for the collation. The collation name must be unique within a schema. The name will be displayed in the *pgAdmin* tree control.
- Select the name of the owner from the drop-down listbox in the *Owner* field.
- Select the name of the schema in which the collation will reside from the drop-down listbox in the *Schema* field.
- Store notes about the collation in the *Comment* field.

Click the *Definition* tab to continue.



Use the fields in the *Definition* tab to specify the operating system locale settings:

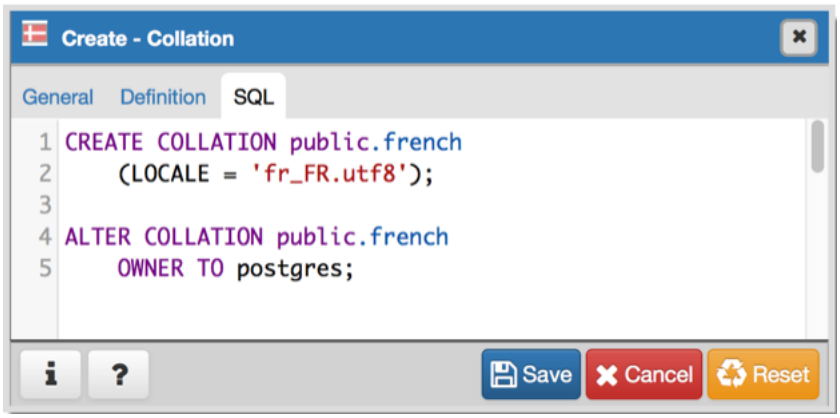
- Use the drop-down listbox next to *Copy collation* to select the name of an existing collation to copy. The new collation will have the same properties as the existing one, but will be an independent object. If you choose to copy an existing collation, you cannot modify the collation properties displayed on this tab.
- Use the *Locale* field to specify a locale; a locale specifies language and language formatting characteristics. If you specify this, you cannot specify either of the following parameters. To view a list of locales supported by your Linux system use the command *locale -a*.
- Use the *LC_COLLATE* field to specify a locale with specified string sort order. The locale must be applicable to the current database encoding. (See CREATE DATABASE for details.)
- Use the *LC_CTYPE* field to specify a locale with specified character classification. The locale must be applicable to the current database encoding. (See CREATE DATABASE for details.)

Click the *SQL* tab to continue.

Your entries in the *Collation* dialog generate a SQL command (see an example below). Use the *SQL* tab for review; revisit or switch tabs to make any changes to the SQL command.

Example

The following is an example of the sql command generated by user selections in the *Collation* dialog:



The example shown demonstrates creating a collation named *french* that uses the rules specified for the locale, *fr_FR.utf8. The collation is owned by *postgres*.

- Click the *Info* button (i) to access online help. View context-sensitive help in the *Tabbed browser*, where a new tab displays the PostgreSQL core documentation. For more information about setting a locale, see Chapter 22.1 Locale Support of the PostgreSQL core documentation:

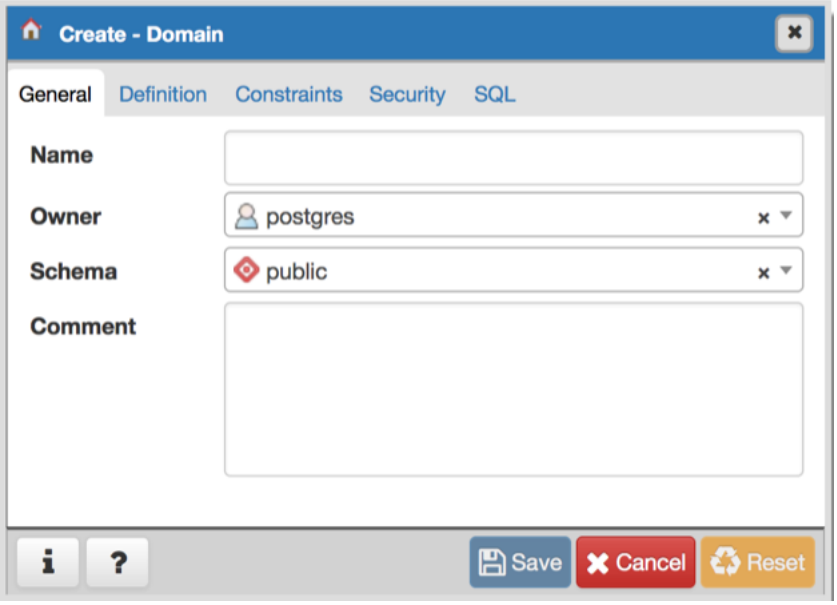
  http://www.postgresql.org/docs/9.5/static/locale.html

- Click the *Save* button to save work.
- Click the *Cancel* button to exit without saving work.
- Click the *Reset* button to restore configuration parameters.

3.3 The Domain Dialog

Use the *Domain* dialog to define a domain. A domain is a data type definition that may constrain permissible values. Domains are useful when you are creating multiple tables that contain comparable columns; you can create a domain that defines constraints that are common to the columns and re-use the domain definition when creating the columns, rather than individually defining each set of constraints.

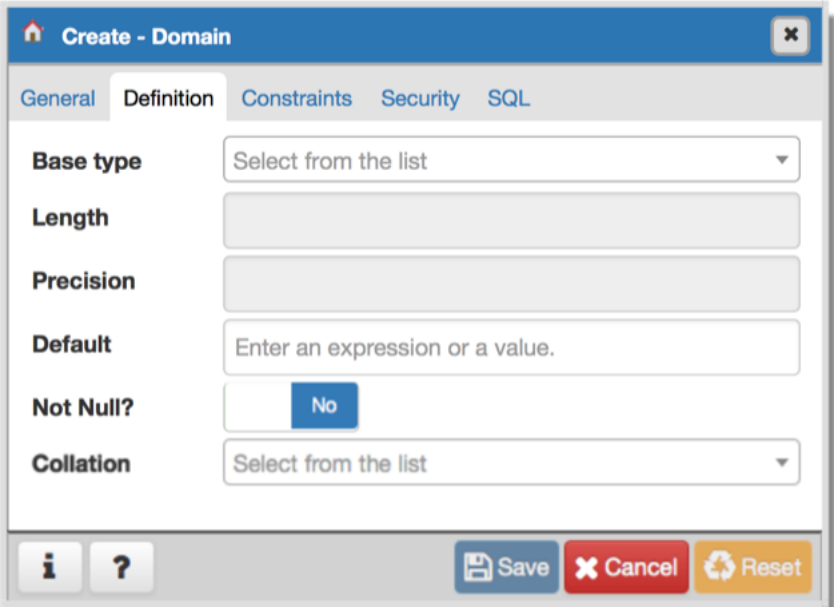
The *Domain* dialog organizes the development of a domain through the following tabs: *General*, *Definition*, *Constraints*, and *Security*. The *SQL* tab displays the SQL code generated by dialog selections.



Use the fields on the *General* tab to identify a domain:

- Use the *Name* field to add a descriptive name for the domain. The name will be displayed in the *pgAdmin* tree control.
- Use the drop-down listbox next to *Owner* to select a role that will own the domain.
- Select the name of the schema in which the domain will reside from the drop-down listbox in the *Schema* field.
- Store notes about the domain in the *Comment* field.

Click the *Definition* tab to continue.

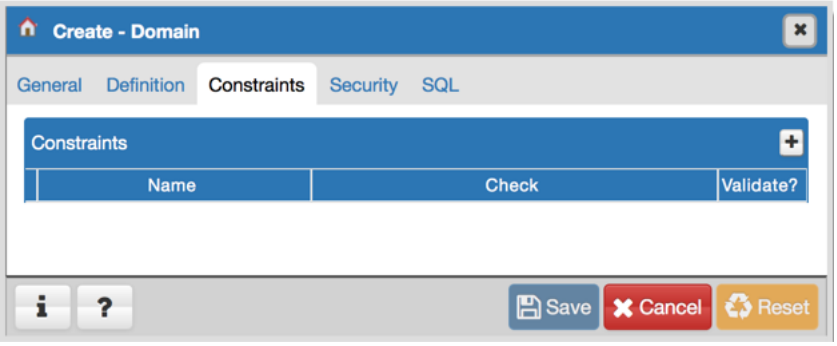


Use the fields in the *Definition* tab to describe the domain:

- Use the drop-down listbox next to *Base type* to specify a data type.

- Use the context-sensitive *Length* field to specify a numeric length for a numeric type.
- Use the context-sensitive *Precision* field to specify the total count of significant digits for a numeric type.
- Specify a default value for the domain data type in the *Default* field. The data type of the default expression must match the data type of the domain. If no default value is specified, then the default value is the null value.
- Move the *Not Null* switch to specify the values of this domain are prevented from being null.
- Use the drop-down listbox next to *Collation* to apply a collation cast. If no collation is specified, the underlying data type's default collation is used. The underlying type must be collatable if COLLATE is specified.

Click the *Constraints* tab to continue.



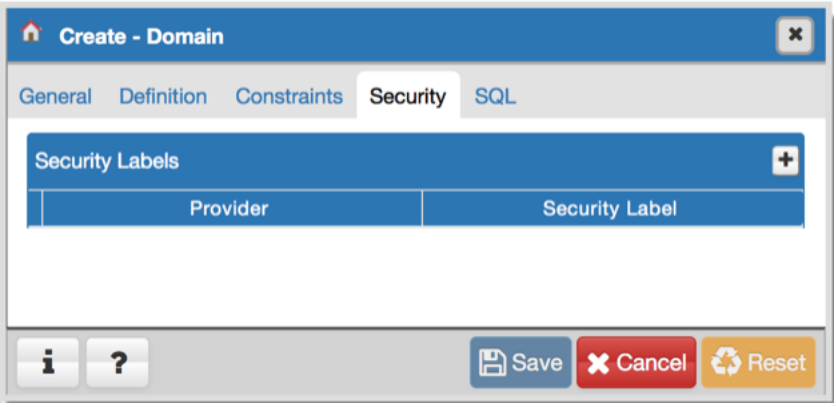
Use the fields in the *Constraints* tab to specify rules for the domain. Click the *Add* icon (+) to set constraints:

- Use the *Name* field to specify a name for the constraint.
- Use the *Check* field to provide an expression for the constraint.
- Use the *Validate* checkbox to determine whether the constraint will be validated. The default checkbox is checked and sets a validation requirement.

A CHECK clause specifies an integrity test which values of the domain must satisfy. Each constraint must be an expression that produces a Boolean result. Use the key word VALUE to refer to the value being tested. Expressions evaluating to TRUE or UNKNOWN succeed. If the expression produces a FALSE result, an error is reported and the value is not allowed to be converted to the domain type. A CHECK expression cannot contain subqueries nor refer to variables other than VALUE. If a domain has multiple CHECK constraints, they will be tested in alphabetical order by name.

Click the *Add* icon (+) to set additional constraints; to discard a constraint, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Click the *Security* tab to continue.



Use the *Security Labels* panel to assign security labels. Click the *Add* icon (+) to add a label:

- Specify a security label provider in the *Provider* field. The named provider must be loaded and must consent to the proposed labeling operation.
- Specify a a security label in the *Security Label* field. The meaning of a given label is at the discretion of the label provider. PostgreSQL places no restrictions on whether or how a label provider must interpret security labels; it merely provides a mechanism for storing them.

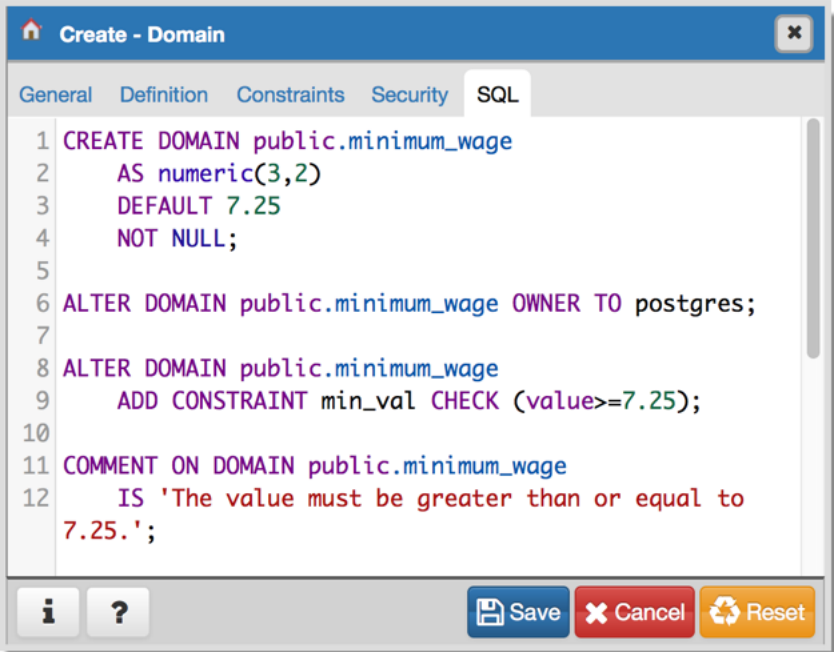
Click the *Add* icon (+) to specify each additional label; to discard a label, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Click the *SQL* tab to continue.

Your entries in the *Domain* dialog generate a SQL command (see an example below). Use the *SQL* tab for review; revisit or switch tabs to make any changes to the SQL command.

Example

The following is an example of the sql command generated by selections made in the *Domain* dialog:



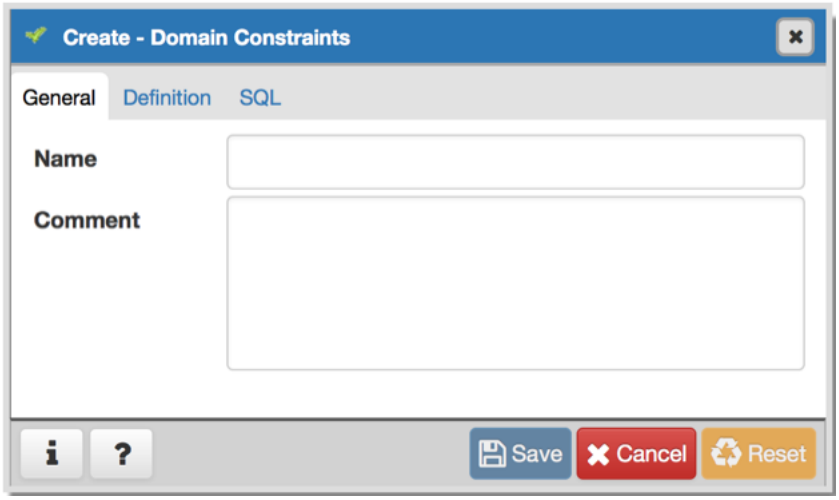
The example shown demonstrates creating a domain named *minimum-wage* that confirms that the value entered is greater than or equal to *7.25*.

- Click the *Info* button (i) to access online help. View context-sensitive help in the *Tabbed browser*, where a new tab displays the PostgreSQL core documentation.
- Click the *Save* button to save work.
- Click the *Cancel* button to exit without saving work.
- Click the *Reset* button to restore configuration parameters.

3.4 The Domain Constraints Dialog

Use the *Domain Constraints* dialog to create or modify a domain constraint. A domain constraint confirms that the values provided for a domain meet a defined criteria. The *Domain Constraints* dialog implements options of the ALTER DOMAIN command.

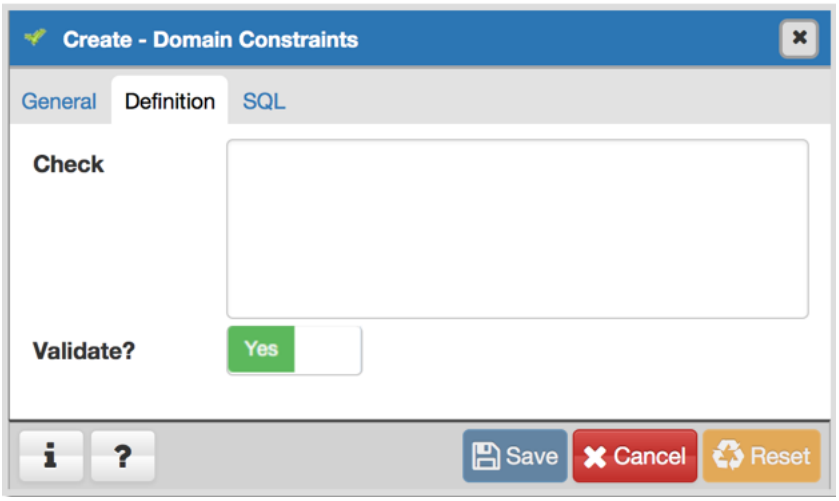
The *Domain Constraints* dialog organizes the development of a domain constraint through the following dialog tabs: *General* and *Definition*. The *SQL* tab displays the SQL code generated by dialog selections.



Use the fields in the *General* tab to identify the domain constraint:

- Use the *Name* field to add a descriptive name for the constraint. The name will be displayed in the *pgAdmin* tree control.
- Store notes about the constraint in the *Comment* field.

Click the *Definition* tab to continue.



Use the fields in the *Definition* tab to define the domain constraint:

- Use the *Check* field to provide a CHECK expression. A CHECK expression specifies a constraint that the domain must satisfy. A constraint must produce a Boolean result; include the key word VALUE to refer to the value being tested. Only those expressions that evaluate to TRUE or UNKNOWN will succeed. A CHECK expression cannot contain subqueries or refer to variables other than VALUE. If a domain has multiple CHECK constraints, they will be tested in alphabetical order.
- Move the *Validate?* switch to the *No* position to mark the constraint NOT VALID. If the constraint is marked NOT VALID, the constraint will not be applied to existing column data. The default value is *Yes*.

Click the *SQL* tab to continue.

Your entries in the *Domain Constraints* dialog generate a SQL command (see an example below). Use the *SQL* tab for review; revisit or switch tabs to make any changes to the SQL command.

Example

The following is an example of the sql command generated by user selections in the *Domain Constraints* dialog:

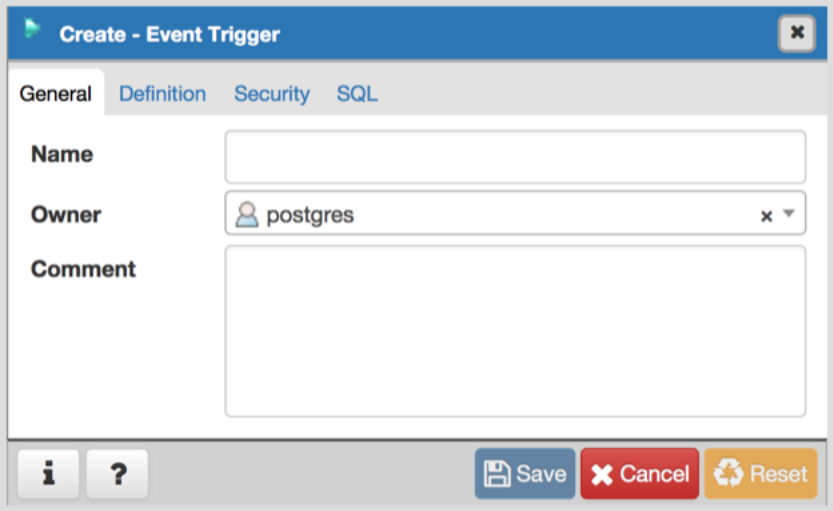
The example shown demonstrates creating a domain constraint on the domain *timesheets* named *weekday*. It constrains a value to equal *Friday*.

- Click the *Info* button (i) to access online help. View context-sensitive help in the *Tabbed browser*, where a new tab displays the PostgreSQL core documentation.
- Click the *Save* button to save work.
- Click the *Cancel* button to exit without saving work.
- Click the *Reset* button to restore configuration parameters.

3.5 The Event Trigger Dialog

Use the *Domain Trigger* dialog to define an event trigger. Unlike regular triggers, which are attached to a single table and capture only DML events, event triggers are global to a particular database and are capable of capturing DDL events. Like regular triggers, event triggers can be written in any procedural language that includes event trigger support, or in C, but not in SQL.

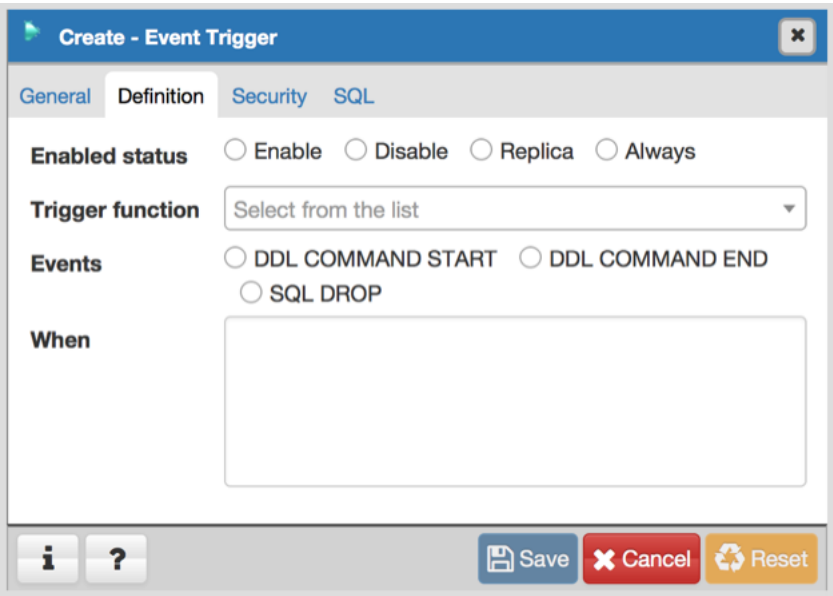
The *Domain Trigger* dialog organizes the development of a event trigger through the following dialog tabs: *General*, *Definition*, and *Security Labels*. The *SQL* tab displays the SQL code generated by dialog selections.



Use the fields in the *General* tab to identify the event trigger:

- Use the *Name* field to add a descriptive name for the event trigger. The name will be displayed in the *pgAdmin* tree control.
- Use the drop-down listbox next to *Owner* to specify the owner of the event trigger.
- Store notes about the event trigger in the *Comment* field.

Click the *Definition* tab to continue.

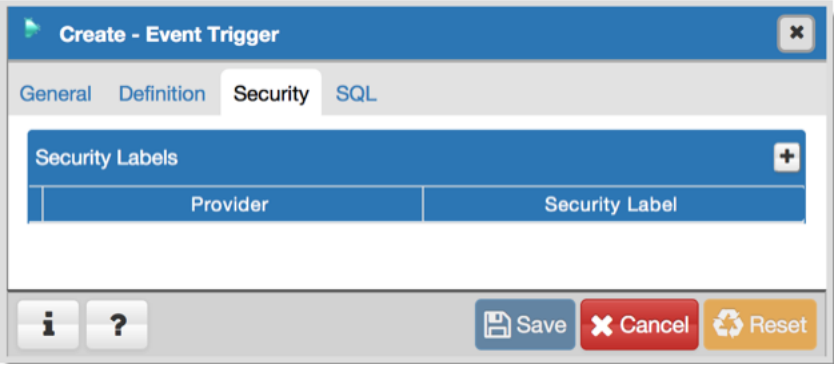


Use the fields in the *Definition* tab to define the event trigger:

- Select a radio button in the *Enabled Status* field to specify a status for the trigger: *Enable Disable*, *Replica Always*.
- Use the drop-down listbox next to *Trigger function* to specify an existing function. A trigger function takes an empty argument list, and returns a value of type event_trigger.
- Select a radio button in the *Events* field to specify when the event trigger will fire: *DDL COMMAND START*, *DDL COMMAND END*, or *SQL DROP*.

- Use the *When* field to write a condition for the event trigger that must be satisfied before the event trigger can execute.

Click the *Security Labels* tab to continue.



Use the *Security* tab to define security labels applied to the trigger. Click the *Add* icon (+) to add each security label.

- Specify a security label provider in the *Provider* field. The named provider must be loaded and must consent to the proposed labeling operation.
- Specify a security label in the *Security Label* field. The meaning of a given label is at the discretion of the label provider. PostgreSQL places no restrictions on whether or how a label provider must interpret security labels; it merely provides a mechanism for storing them.

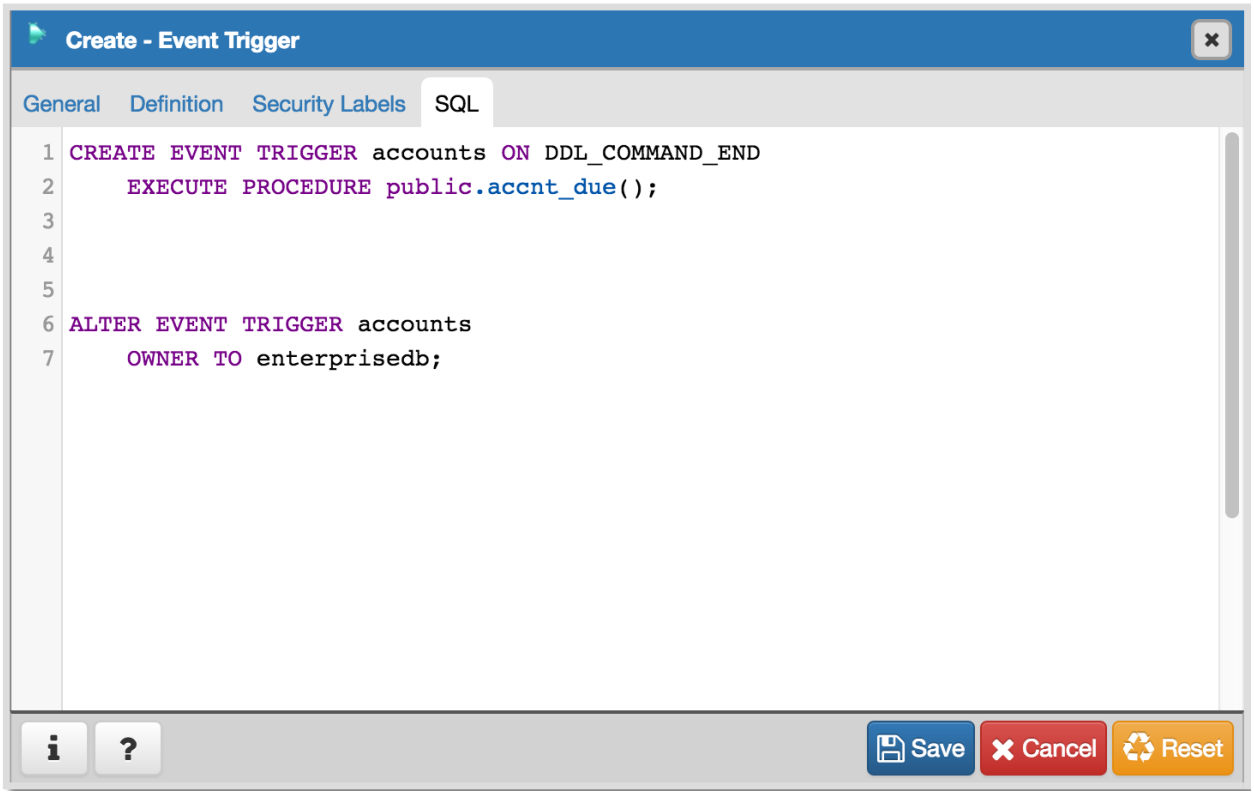
Click the *Add* icon (+) to assign additional security labels; to discard a security label, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Click the *SQL* tab to continue.

Your entries in the *Domain Trigger* dialog generate a generate a SQL command. Use the *SQL* tab for review; revisit or switch tabs to make any changes to the SQL command.

Example

The following is an example of the sql command generated by user selections in the *Domain Trigger* dialog:



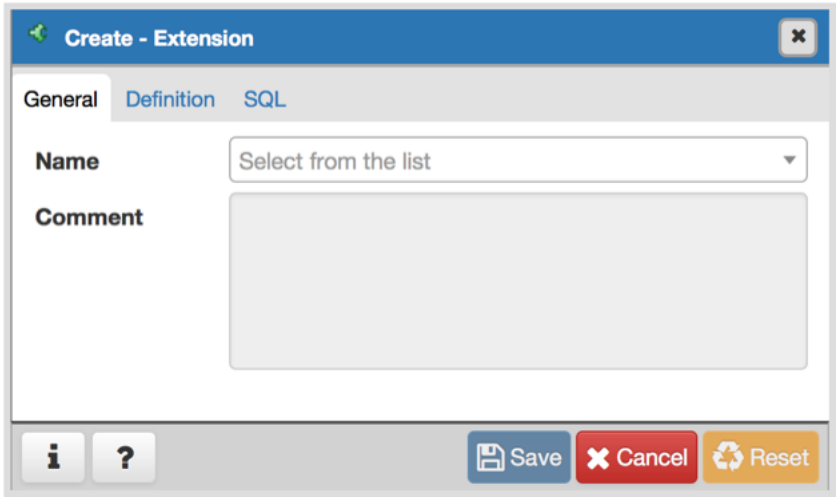
The command creates an event trigger named *accounts* that invokes the procedure named *acct_due*.

- Click the *Info* button (i) to access online help. View context-sensitive help in the *Tabbed browser*, where a new tab displays the PostgreSQL core documentation.
- Click the *Save* button to save work.
- Click the *Cancel* button to exit without saving work.
- Click the *Reset* button to restore configuration parameters.

3.6 The Extension Dialog

Use the *Extension* dialog to install a new extension into the current database. An extension is a collection of SQL objects that add targeted functionality to your Postgres installation. The *Extension* dialog adds the functionality of an extension to the current database only; you must register the extension in each database that use the extension. Before you load an extension into a database, you should confirm that any pre-requisite files are installed.

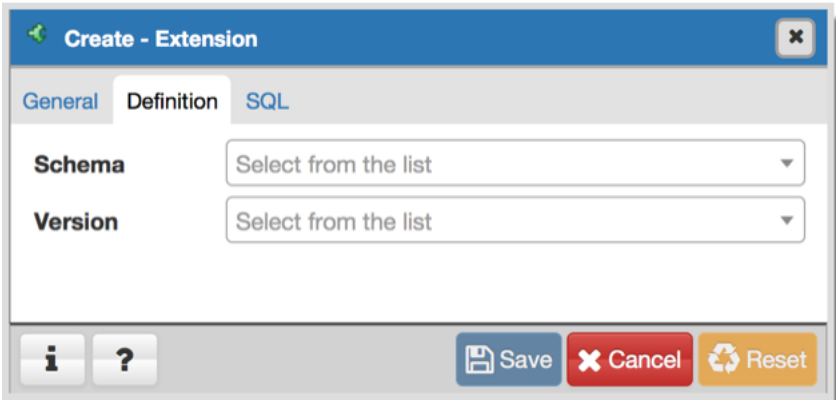
The *Extension* dialog allows you to implement options of the CREATE EXTENSION command through the following dialog tabs: *General* and *Definition*. The *SQL* tab displays the SQL code generated by dialog selections.



Use the fields in the *General* tab to identify an extension:

- Use the drop-down listbox in the *Name* field to select the extension. Each extension must have a unique name.
- Store notes about the extension in the *Comment* field.

Click the *Definition* tab to continue.



Use the *Definition* tab to select the *Schema* and *Version*:

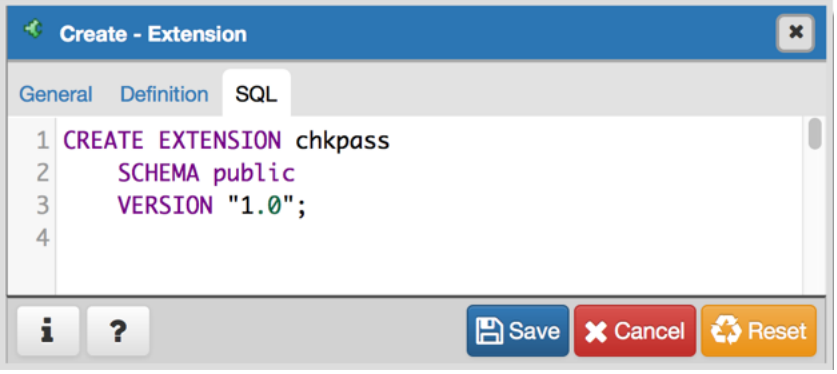
- Use the drop-down listbox next to *Schema* to select the name of the schema in which to install the extension's objects.
- Use the drop-down listbox next to *Version* to select the version of the extension to install.

Click the *SQL* tab to continue.

Your entries in the *Extension* dialog generate a SQL command (see an example below). Use the *SQL* tab for review; revisit or switch tabs to make any changes to the SQL command.

Example

The following is an example of the sql command generated by user selections in the *Extension* dialog:



The command creates the *chkpass* extension in the *public* schema. It is version *1.0* of *chkpass*.

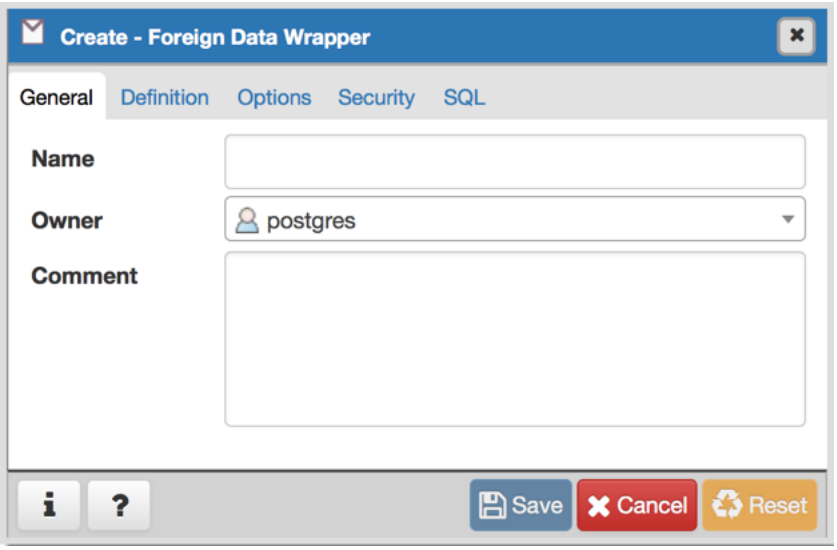
- Click the *Info* button (i) to access online help. View context-sensitive help in the *Tabbed browser*, where a new tab displays the PostgreSQL core documentation.
- Click the *Save* button to save work.
- Click the *Cancel* button to exit without saving work.
- Click the *Reset* button to restore configuration parameters.

3.7 The Foreign Data Wrapper Dialog

Use the *Foreign Data Wrapper* dialog to create or modify a foreign data wrapper. A foreign data wrapper is an adapter between a Postgres database and data stored on another data source.

You must be a superuser to create a foreign data wrapper.

The *Foreign Data Wrapper* dialog organizes the development of a foreign data wrapper through the following dialog tabs: *General*, *Definition*, *Options*, and *Security*. The *SQL* tab displays the SQL code generated by dialog selections.

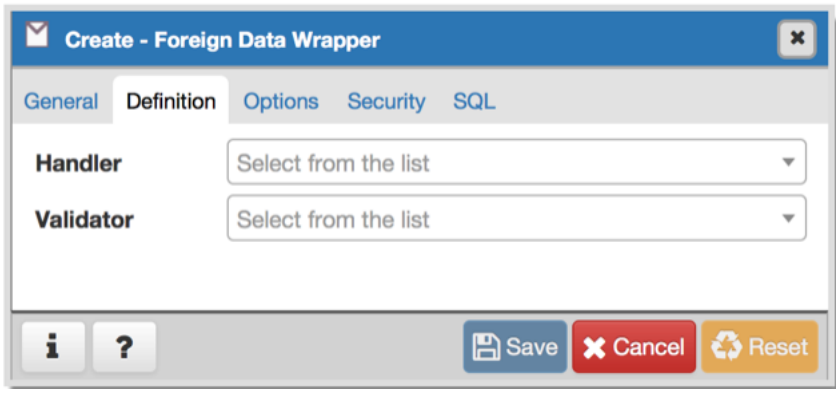


Use the fields in the *General* tab to identify the foreign data wrapper:

- Use the *Name* field to add a descriptive name for the foreign data wrapper. A foreign data wrapper name must be unique within the database. The name will be displayed in the *pgAdmin* tree control.
- Use the drop-down listbox next to *Owner* to select the name of the role that will own the foreign data wrapper.

- Store notes about the foreign data wrapper in the *Comment* field.

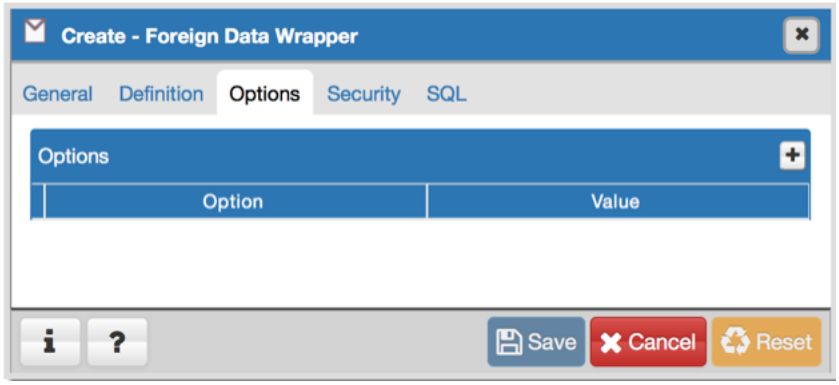
Click the *Definition* tab to continue.



Use the fields in the *Definition* tab to set parameters:

- Select the name of the handler function from the drop-down listbox in the *Handler* field. This is the name of an existing function that will be called to retrieve the execution functions for foreign tables.
- Select the name of the validator function from the drop-down listbox in the *Validator* field. This is the name of an existing function that will be called to check the generic options given to the foreign data wrapper, as well as options for foreign servers, user mappings and foreign tables using the foreign data wrapper.

Click the *Options* tab to continue.

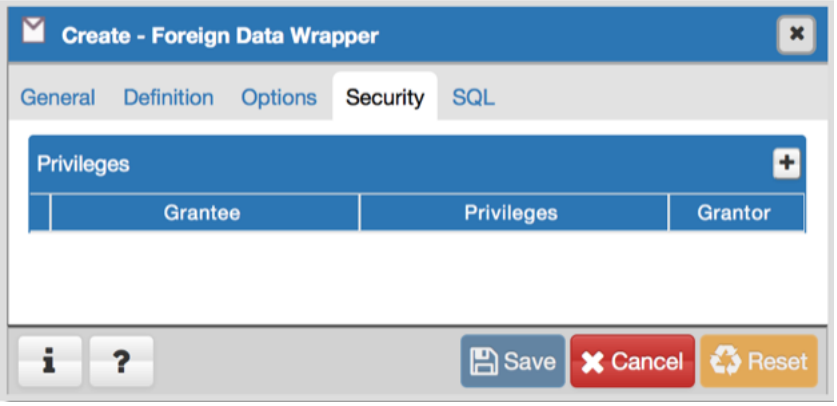


Use the fields in the *Options* tab to specify options:

- Click the the *Add* icon (+) button to add an option/value pair for the foreign data wrapper. Supported option/value pairs will be specific to the selected foreign data wrapper.
- Specify the option name in the *Option* field and provide a corresponding value in the *Value* field.

Click the *Add* icon (+) to specify each additional pair; to discard an option, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Click the *Security* tab to continue.



Use the *Security* tab to assign security privileges. Click the *Add* icon (+) to assign a set of privileges.

- Select the name of the role from the drop-down listbox in the *Grantee* field.
- Click inside the *Privileges* field. Check the boxes to the left of one or more privileges to grant the selected privileges to the specified user.
- Select the name of the role granting the privileges from the drop-down listbox in the *Grantor* field. The default grantor is the owner of the foreign data wrapper.

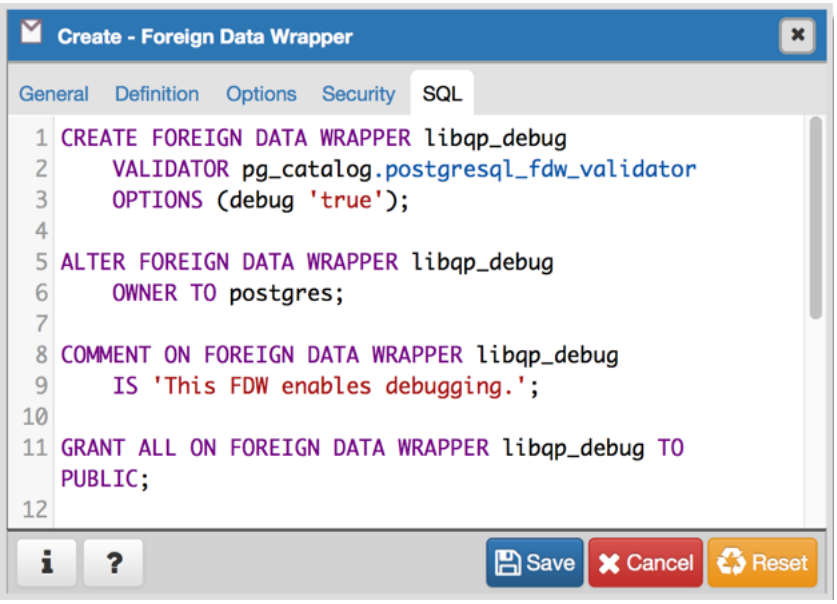
Click add to assign additional privileges; to discard a privilege, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Click the *SQL* tab to continue.

Your entries in the *Foreign Data Wrapper* dialog generate a SQL command (see an example below). Use the *SQL* tab for review; revisit or switch tabs to make any changes to the SQL command.

Example

The following is an example of the sql command generated by user selections in the *Foreign Data Wrapper* dialog:



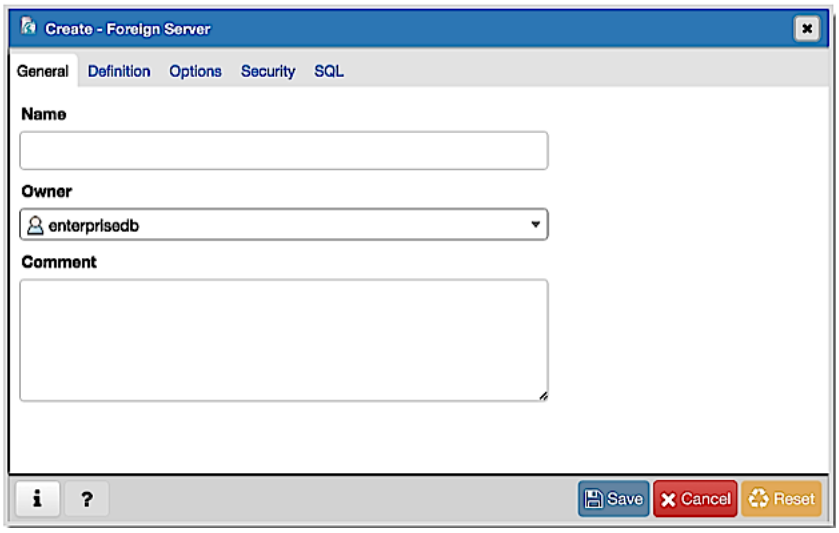
The example creates a foreign data wrapper named *libpq_debug* that uses pre-existing validator and handler functions, *dblink_fdw_validator* and *libpg_fdw_handler*. Selections on the *Options* tab set *debug* equal to *true*. The foreign data wrapper is owned by *postgres*.

- Click the *Help* button (?) to access online help. View context-sensitive help in the *Tabbed browser*, where a new tab displays the PostgreSQL core documentation.
- Click the *Save* button to save work.
- Click the *Cancel* button to exit without saving work.
- Click the *Reset* button to restore configuration parameters.

3.8 The Foreign Server Dialog

Use the *Foreign Server* dialog to create a foreign server. A foreign server typically encapsulates connection information that a foreign-data wrapper uses to access an external data resource. Each foreign data wrapper may connect to a different foreign server; in the *pgAdmin* tree control, expand the node of the applicable foreign data wrapper to launch the *Foreign Server* dialog.

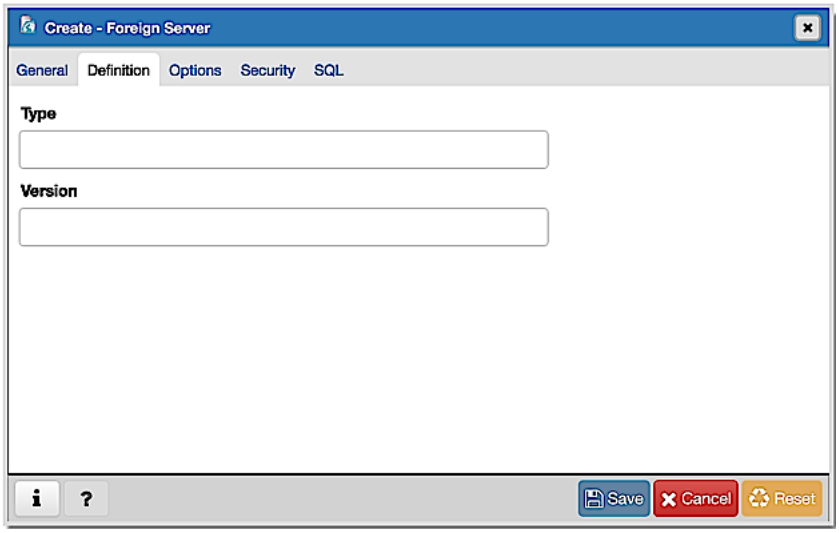
The *Foreign Server* dialog organizes the development of a foreign server through the following dialog tabs: *General*, *Definition*, *Options*, and *Security*. The *SQL* tab displays the SQL code generated by dialog selections.



Use the fields in the *General* tab to identify the foreign server:

- Use the *Name* field to add a descriptive name for the foreign server. The name will be displayed in the *pgAdmin* tree control. It must be unique within the database.
- Use the drop-down listbox next to *Owner* to select a role.
- Store notes about the foreign server in the *Comment* field.

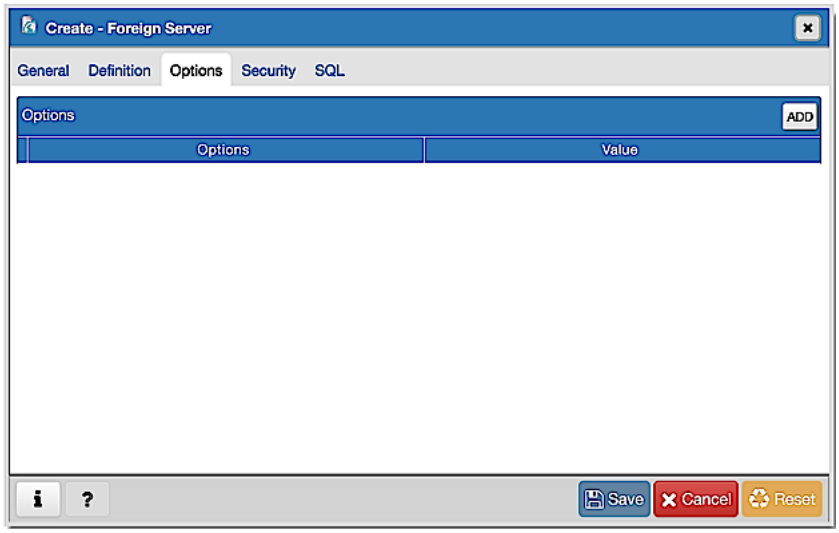
Click the *Definition* tab to continue.



Use the fields in the *Definition* tab to set parameters:

- Use the *Type* field to specify a server type.
- Use the *Version* field to specify a server version.

Click the *Options* tab to continue.

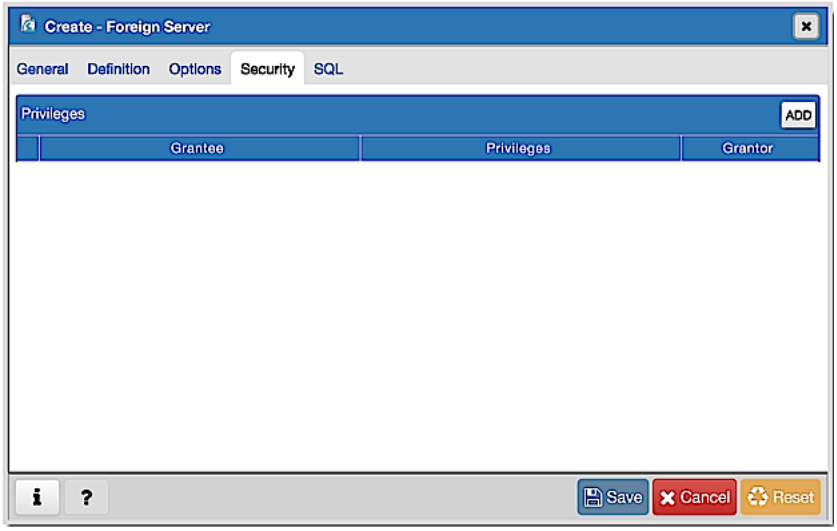


Use the fields in the *Options* tab to specify options. Click the *Add* button to create an option clause for the foreign server.

- Specify the option name in the *Option* field.
- Provide a corresponding value in the *Value* field.

Click *Add* to create each additional clause; to discard an option, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Click the *Security* tab to continue.



Use the *Security* tab to assign security privileges to the foreign server. Click *Add* before you assign a set of privileges.

- Select the name of the role from the drop-down listbox in the *Grantee* field.
- Click inside the *Privileges* field. Check the boxes to the left of one or more privileges to grant the selected privileges to the specified user.
- Select the name of the role from the drop-down listbox in the *Grantor* field. The default grantor is the owner of the foreign server. This is a required field.

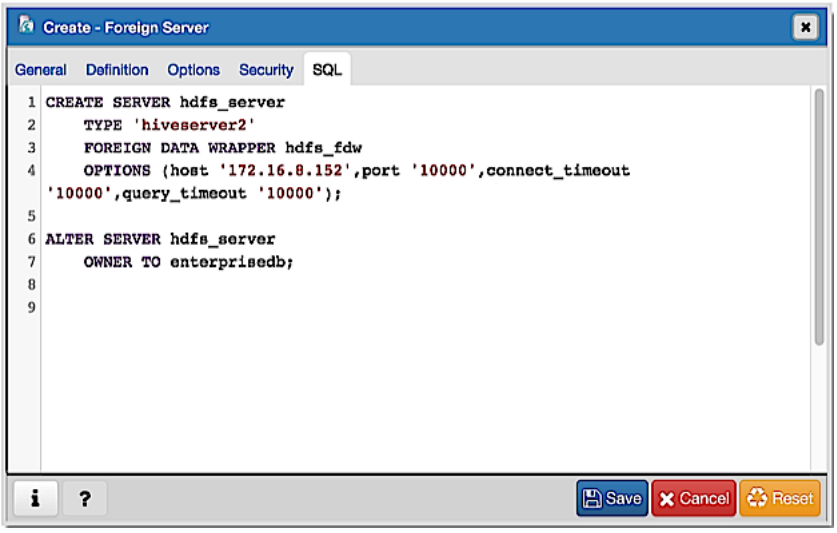
Click *Add* to assign a new set of privileges; to discard a privilege, click the trash icon to the left of the row and confirm deletion in the *Delete Row* dialog.

Click the *SQL* tab to continue.

Your entries in the *Foreign Server* dialog generate a SQL command (see an example below). Use the *SQL* tab for review; revisit or switch tabs to make any changes to the SQL command.

Example

The following is an example of the sql command generated by user selections in the *Foreign Server* dialog:



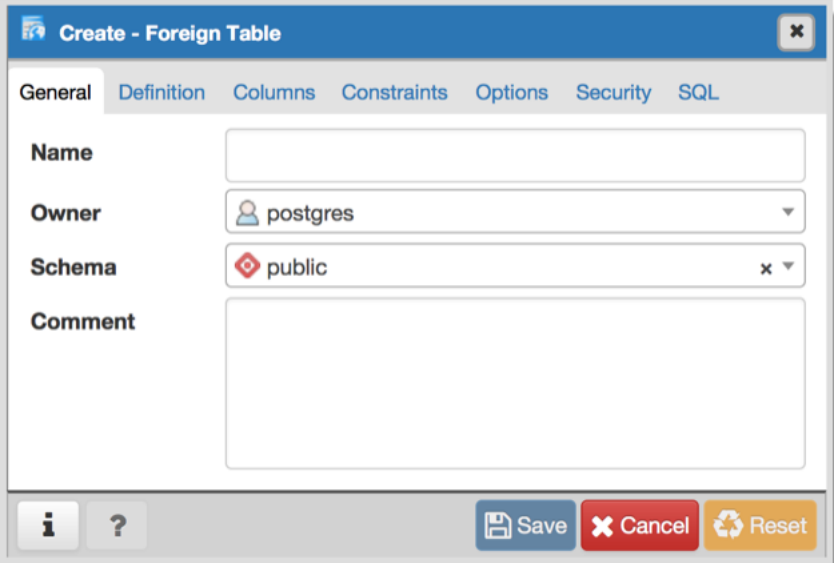
The example shown demonstrates creating a foreign server for the foreign data wrapper *hdfs_fdw*. It has the name *hdfs_server*; its type is *hiveserver2*. Options for the foreign server include a host and a port.

- Click the *Info* button (i) to access online help. View context-sensitive help in the *Tabbed browser*, where a new tab displays the PostgreSQL core documentation.
- Click the *Save* button to save work.
- Click the *Cancel* button to exit without saving work.
- Click the *Reset* button to restore configuration parameters.

3.9 The Foreign Table Dialog

Use the *Foreign Table* dialog to define a foreign table in the current database. Foreign tables define the structure of an external data source that resides on a foreign server.

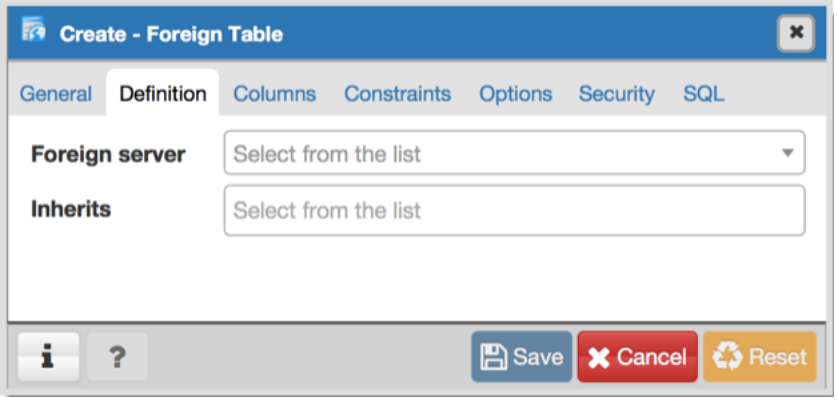
The *Foreign Table* dialog organizes the development of a foreign table through the following dialog tabs: *General*, *Definition*, *Columns*, *Constraints*, *Options*, and *Security*. The *SQL* tab displays the SQL code generated by dialog selections.



Use the fields in the *General* tab to identify the foreign table:

- Use the *Name* field to add a descriptive name for the foreign table. The name of the foreign table must be distinct from the name of any other foreign table, table, sequence, index, view, existing data type, or materialized view in the same schema. The name will be displayed in the *pgAdmin* tree control.
- Use the drop-down listbox next to *Owner* to select the name of the role that will own the foreign table.
- Select the name of the schema in which the foreign table will reside from the drop-down listbox in the *Schema* field.
- Store notes about the foreign table in the *Comment* field.

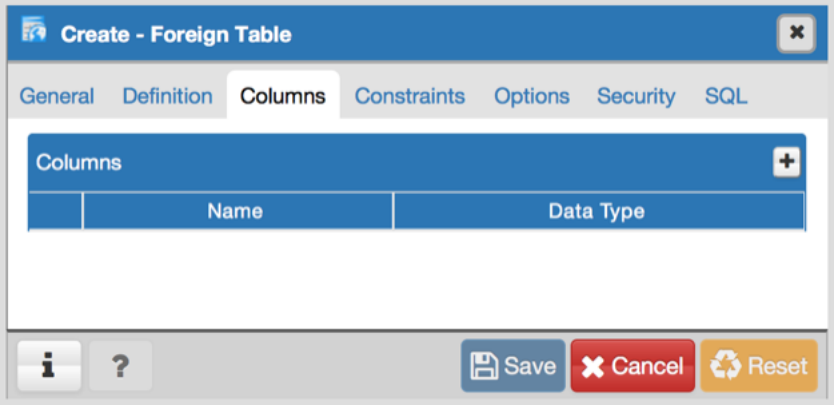
Click the *Definition* tab to continue.



Use the fields in the *Definition* tab to define the external data source:

- Use the drop-down listbox next to *Foreign server* to select a foreign server. This list is populated with servers defined through the *Foreign Server* dialog.
- Use the drop-down listbox next to *Inherits* to specify a parent table. The foreign table will inherit all of its columns. This field is optional.

Click the *Columns* tab to continue.

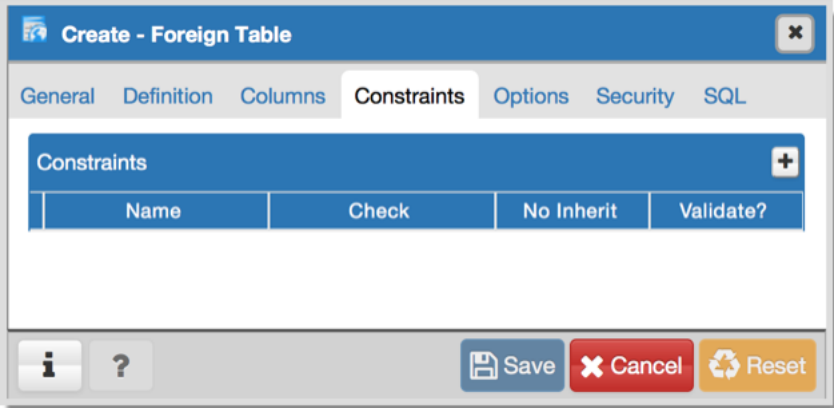


Use the fields in the *Columns* tab to to add columns and their attributes to the table. Click the *Add* icon (+) to define a column:

- Use the *Name* field to add a descriptive name for the column.
- Use the drop-down listbox in the *Data Type* field to select a data type for the column. This can include array specifiers. For more information on which data types are supported by PostgreSQL, refer to Chapter 8 of the core documentation.

Click the *Add* icon (+) to specify each additional column; to discard a column, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Click the *Constraints* tab to continue.

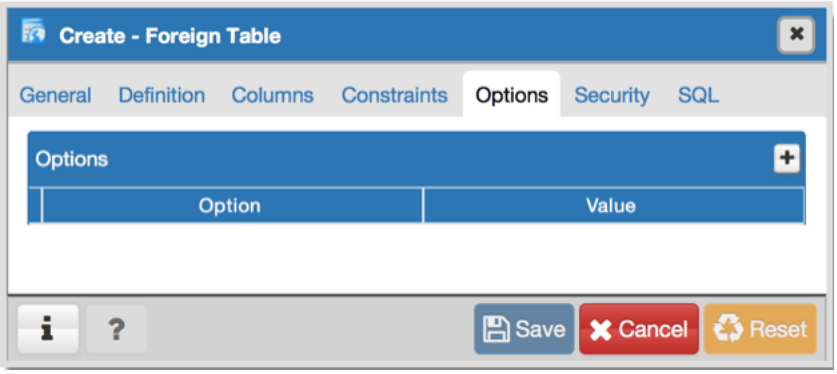


Use the fields in the *Constraints* tab to apply a table constraint to the foreign table. Click the *Add* icon (+) to define a constraint:

- Use the *Name* field to add a descriptive name for the constraint. If the constraint is violated, the constraint name is present in error messages, so constraint names like *col must be positive* can be used to communicate helpful information.
- Use the *Check* field to write a check expression producing a Boolean result. Each row in the foreign table is expected to satisfy the check expression.
- Check the *No Inherit* checkbox to specify that the constraint will not propagate to child tables.
- Uncheck the *Validate* checkbox to disable validation. The database will not assume that the constraint holds for all rows in the table.

Click the *Add* icon (+) to specify each additional constraint; to discard a constraint, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Click the *Options* tab to continue.

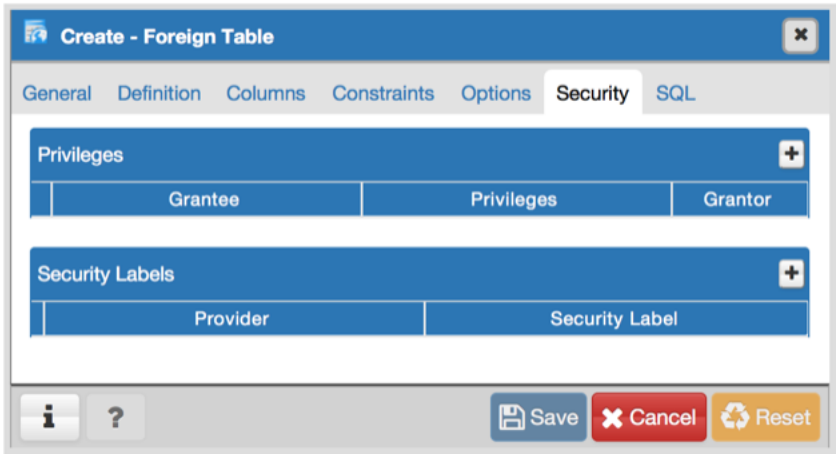


Use the fields in the *Options* tab to specify options to be associated with the new foreign table or one of its columns; the accepted option names and values are specific to the foreign data wrapper associated with the foreign server. Click the *Add* icon (+) to add an option/value pair.

- Specify the option name in the *Option* field. Duplicate option names are not allowed.
- Provide a corresponding value in the *Value* field.

Click the *Add* icon (+) to specify each additional option/value pair; to discard an option, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Click the *Security* tab to continue.



Use the *Security* tab to assign privileges and define security labels.

Use the *Privileges* panel to assign privileges to a role. Click the *Add* icon (+) to set privileges for database objects:

- Select the name of the role to which privileges will be assigned from the drop-down listbox in the *Grantee* field.
- Click inside the *Privileges* field. Check the boxes to the left of one or more privileges to grant the selected privilege to the specified user.
- Select the name of the role that owns the foreign table from the drop-down listbox in the *Grantor* field. The default grantor is the owner of the database.

Click the *Add* icon (+) to assign additional privileges; to discard a privilege, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Use the *Security Labels* panel to define security labels applied to the function. Click the *Add* icon (+) to add each security label selection:

- Specify a security label provider in the *Provider* field. The named provider must be loaded and must consent to the proposed labeling operation.
- Specify a a security label in the *Security Label* field. The meaning of a given label is at the discretion of the label provider. PostgreSQL places no restrictions on whether or how a label provider must interpret security labels; it merely provides a mechanism for storing them.

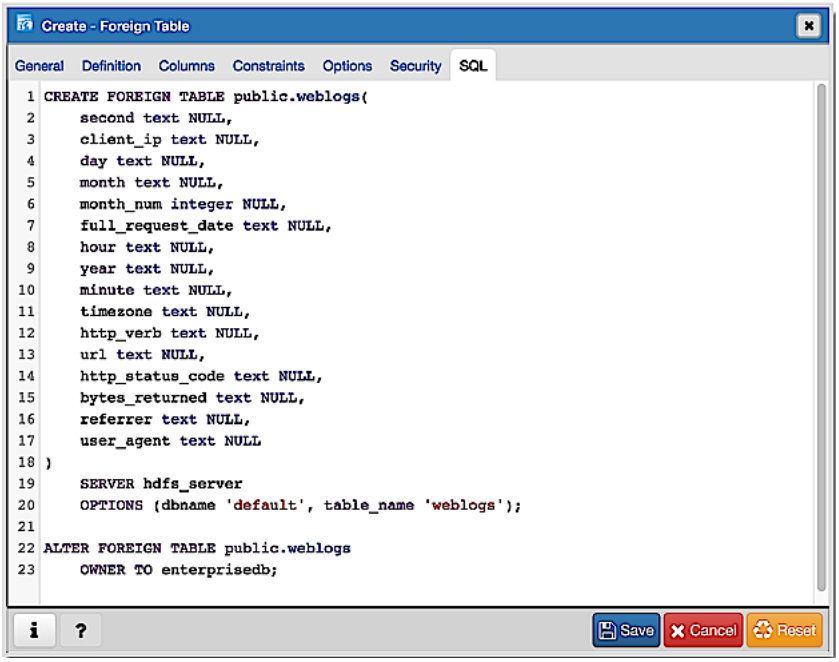
Click the *Add* icon (+) to assign additional security labels; to discard a security label, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Click the *SQL* tab to continue.

Your entries in the *Foreign Table* dialog generate a SQL command (see an example below). Use the *SQL* tab for review; revisit or switch tabs to make any changes to the SQL command.

Example

The following is an example of the sql command generated by user selections in the *Foreign Table* dialog:



The example shown demonstrates creating a foreign table *weblogs* with multiple columns and two options.

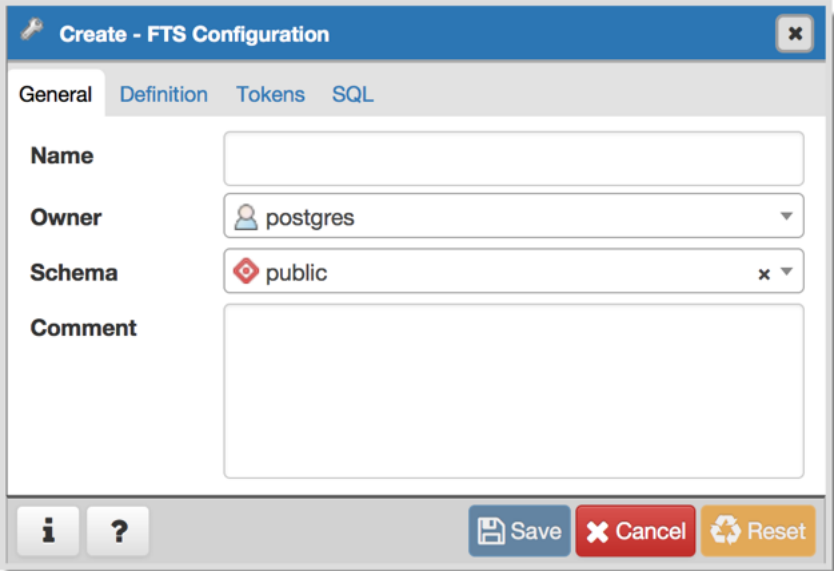
- Click the *Info* button (i) to access online help. View context-sensitive help in the *Tabbed browser*, where a new tab displays the PostgreSQL core documentation.
- Click the *Save* button to save work.
- Click the *Cancel* button to exit without saving work.
- Click the *Reset* button to restore configuration parameters.

3.10 The FTS Configuration dialog

Use the *FTS Configuration* dialog to configure a full text search. A text search configuration specifies a text search parser that can divide a string into tokens, along with dictionaries that can identify searchable tokens.

The *FTS Configuration* dialog organizes the development of a FTS configuration through the following dialog tabs: "*General*, *Definition*, and *Tokens*. The *SQL* tab displays the SQL code generated by dialog selections.

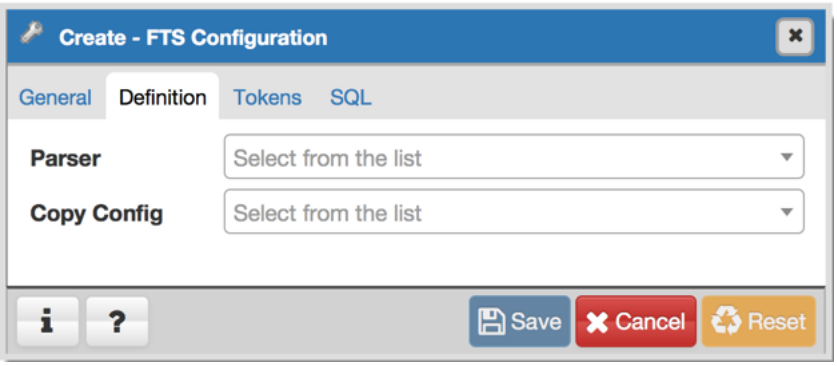
Click the *General* tab to begin.



Use the fields in the *General* tab to identify a FTS configuration:

- Use the *Name* field to add a descriptive name for the FTS configuration. The name will be displayed in the *pgAdmin* tree control.
- Use the drop-down listbox next to *Owner* to specify the role that will own the configuration.
- Select the name of the schema in which the FTS configuration will reside from the drop-down listbox in the *Schema* field.
- Store notes about the FTS configuration in the *Comment* field.

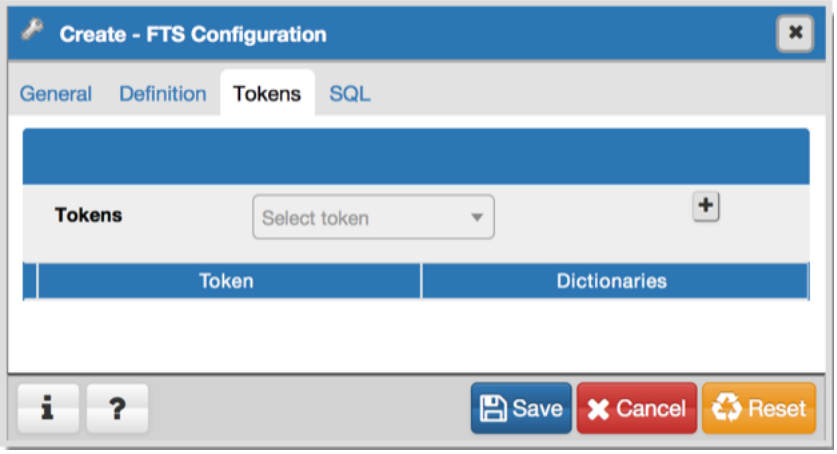
Click the *Definition* tab to continue.



Use the fields in the *Definition* tab to define parameters:

- Select the name of the text search parser from the drop-down listbox in the *Parser* field.
- Select a language from the drop-down listbox in the *Copy Config* field.

Click the *Tokens* tab to continue.



Use the fields in the *Tokens* tab to add a token:

- Use the *Tokens* field to specify the name of a token.
- Click the *Add* icon (+) to create a token.
- Use the *Dictionaries* field to specify a dictionary.

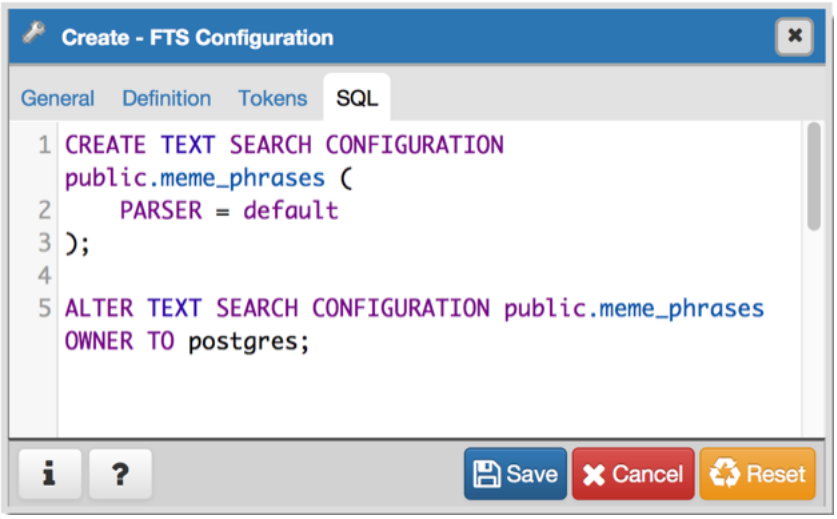
Repeat these steps to add additional tokens; to discard a token, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Click the *SQL* tab to continue.

Your entries in the *FTS Configuration* dialog generate a SQL command (see an example below). Use the *SQL* tab for review; revisit or switch tabs to make any changes to the SQL command.

Example

The following is an example of the sql command generated by user selections in the *FTS Configuration* dialog:



The example shown demonstrates creating a FTS configuration named *meme_phrases*. It uses the *default* parser.

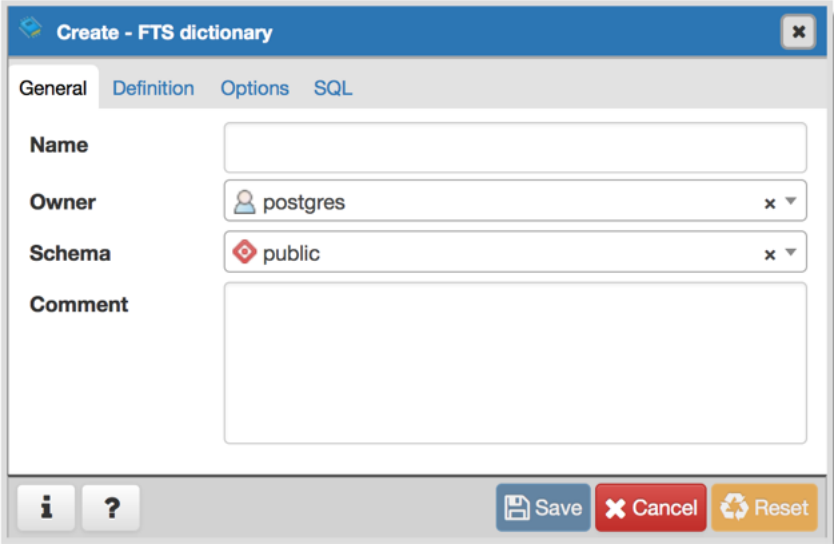
- Click the *Info* button (i) to access online help. View context-sensitive help in the *Tabbed browser*, where a new tab displays the PostgreSQL core documentation.
- Click the *Save* button to save work.
- Click the *Cancel* button to exit without saving work.

- Click the *Reset* button to restore configuration parameters.

3.11 The FTS Dictionary Dialog

Use the *FTS Dictionary* dialog to create a full text search dictionary. You can use a predefined templates or create a new dictionary with custom parameters.

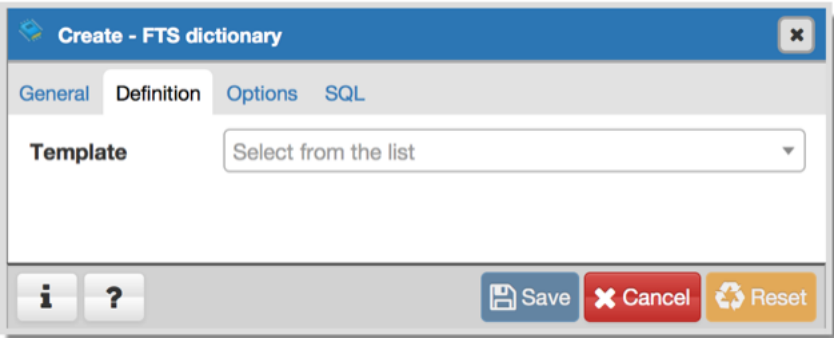
The *FTS Dictionary* dialog organizes the development of a FTS dictionary through the following dialog tabs: *General*, *Definition*, and *Options*. The *SQL* tab displays the SQL code generated by dialog selections.



Use the fields in the *General* tab to identify the dictionary:

- Use the *Name* field to add a descriptive name for the dictionary. The name will be displayed in the *pgAdmin* tree control.
- Use the drop-down listbox next to *Owner* to select the role that will own the FTS Dictionary.
- Select the name of the schema in which the dictionary will reside from the drop-down listbox in the *Schema* field.
- Store notes about the dictionary in the *Comment* field.

Click the *Definition* tab to continue.

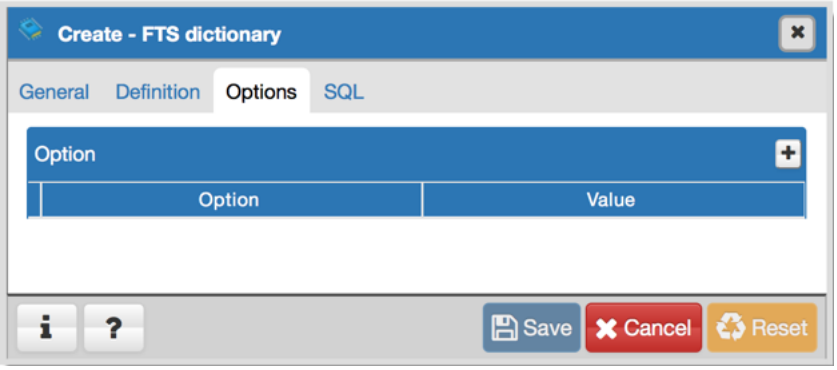


Use the field in the *Definition* tab to choose a template from the drop-down listbox:

*Select *ispell* to select the Ispell template. The Ispell dictionary template supports morphological dictionaries, which can normalize many different linguistic forms of a word into the same lexeme. For example, an English Ispell dictio-

nary can match all declensions and conjugations of the search term bank, e.g., banking, banked, banks, banks', and bank's. Ispell dictionaries usually recognize a limited set of words, so they should be followed by another broader dictionary; for example, a Snowball dictionary, which recognizes everything. *Select *simple* to select the simple template. The simple dictionary template operates by converting the input token to lower case and checking it against a file of stop words. If it is found in the file then an empty array is returned, causing the token to be discarded. If not, the lowercased form of the word is returned as the normalized lexeme. Alternatively, the dictionary can be configured to report non-stop-words as unrecognized, allowing them to be passed on to the next dictionary in the list. *Select *snowball* to select the Snowball template. The Snowball dictionary template is based on a project by Martin Porter, inventor of the popular Porter's stemming algorithm for the English language. Snowball now provides stemming algorithms for many languages (see the Snowball site for more information). Each algorithm understands how to reduce common variant forms of words to a base, or stem, spelling within its language. A Snowball dictionary recognizes everything, whether or not it is able to simplify the word, so it should be placed at the end of the dictionary list. It is useless to have it before any other dictionary because a token will never pass through it to the next dictionary. *Select *synonym* to select the synonym template. This dictionary template is used to create dictionaries that replace a word with a synonym. Phrases are not supported (use the thesaurus template (Section 12.6.4) for that). A synonym dictionary can be used to overcome linguistic problems, for example, to prevent an English stemmer dictionary from reducing the word Paris to pari. *Select *thesaurus* to select the thesaurus template. A thesaurus dictionary replaces all non-preferred terms by one preferred term and, optionally, preserves the original terms for indexing as well. PostgreSQL's current implementation of the thesaurus dictionary is an extension of the synonym dictionary with added phrase support.

Click the *Options* tab to continue.



Use the fields in the *Options* tab to provide template-specific options. Click the *Add* icon (+) to add an option clause:

- Specify the name of an option in the *Option* field
- Provide a value for the option in the *Value* field.

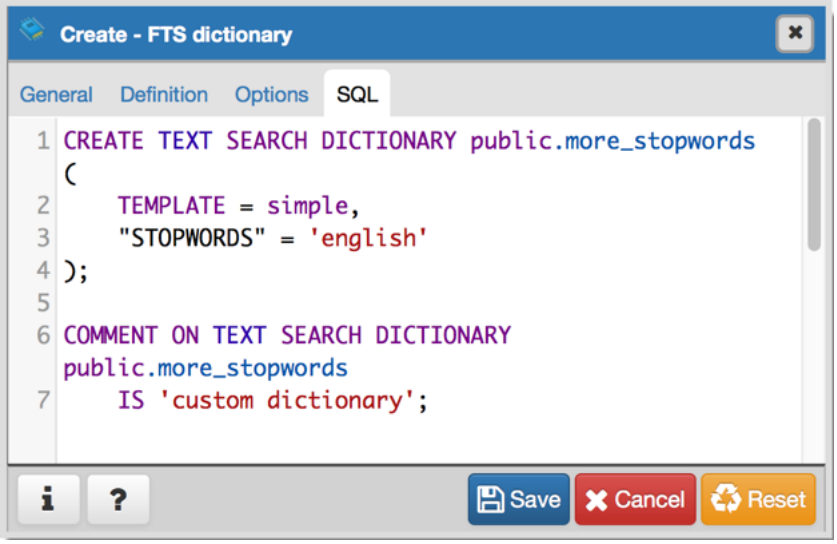
Click the *Add* icon (+) to specify each additional option/value pair; to discard an option, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Click the *SQL* tab to continue.

Your entries in the *FTS Dictionary* dialog generate a generate a SQL command. Use the *SQL* tab for review; revisit or switch tabs to make any changes to the SQL command.

Example

The following is an example of the sql command generated by user selections in the *FTS Dictionary* dialog:



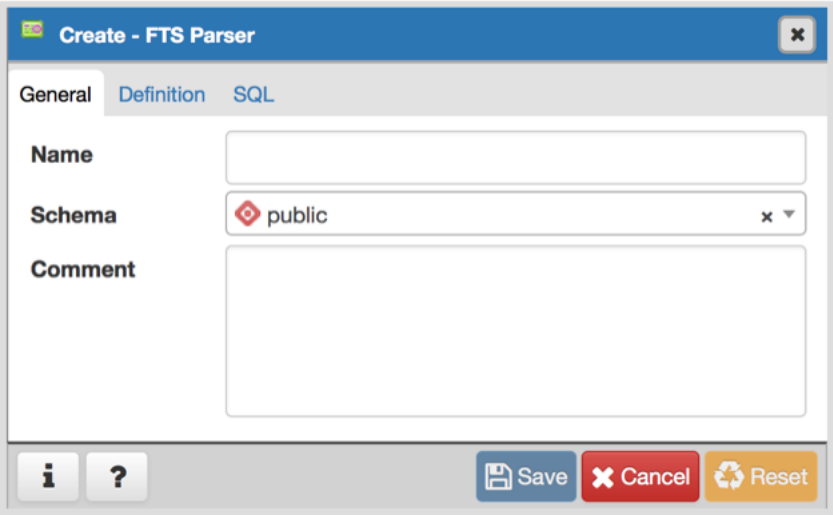
The example shown demonstrates creating a custom dictionary named *more_stopwords* which is based on the simple template and is configured to use standard English.

- Click the *Info* button (i) to access online help. View context-sensitive help in the *Tabbed browser*, where a new tab displays the PostgreSQL core documentation.
- Click the *Save* button to save work.
- Click the *Cancel* button to exit without saving work.
- Click the *Reset* button to restore configuration parameters.

3.12 The FTS Parser Dialog

Use the *FTS Parser* dialog to create a new text search parser. A text search parser defines a method for splitting a text string into tokens and assigning types (categories) to the tokens.

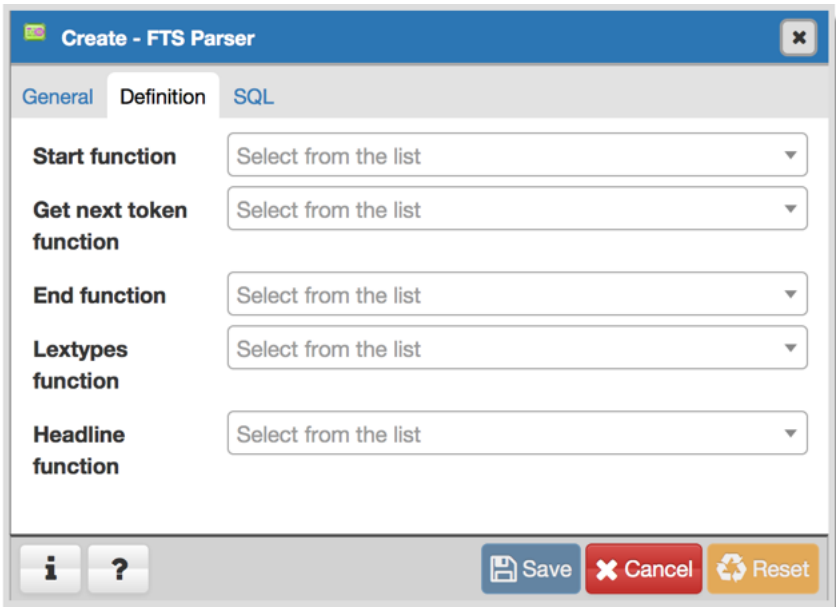
The *FTS Parser* dialog organizes the development of a text search parser through the following dialog tabs: *General*, and *Definition*. The *SQL* tab displays the SQL code generated by dialog selections.



Use the fields in the *General* tab to identify a text search parser:

- Use the *Name* field to add a descriptive name for the parser. The name will be displayed in the *pgAdmin* tree control.
- Select the name of the schema in which the parser will reside from the drop-down listbox in the *Schema* field.
- Store notes about the domain in the *Comment* field.

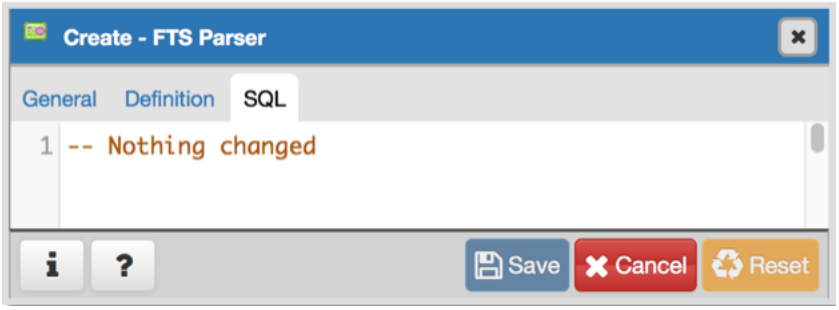
Click the *Definition* tab to continue.



Use the fields in the *Definition* tab to define parameters:

- Use the drop-down listbox next to *Start function* to select the name of the function that will initialize the parser.
- Use the drop-down listbox next to *Get next token function* to select the name of the function that will return the next token.
- Use the drop-down listbox next to *End function* to select the name of the function that is called when the parser is finished.
- Use the drop-down listbox next to *Lextypes function* to select the name of the lextypes function for the parser. The lextypes function returns an array that contains the id, alias, and a description of the tokens used by the parser.
- Use the drop-down listbox next to *Headline function* to select the name of the headline function for the parser. The headline function returns an excerpt from the document in which the terms of the query are highlighted.

Click the *SQL* tab to continue.



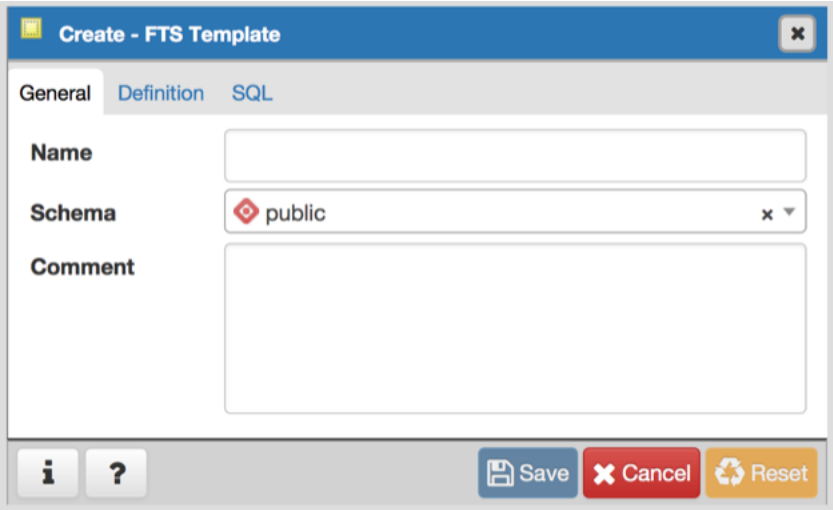
Your entries in the *FTS Parser* dialog generate a generate a SQL command. Use the *SQL* tab for review; revisit or switch tabs to make any changes to the SQL command.

- Click the *Info* button (i) to access online help. View context-sensitive help in the *Tabbed browser*, where a new tab displays the PostgreSQL core documentation.
- Click the *Save* button to save work.
- Click the *Cancel* button to exit without saving work.
- Click the *Reset* button to restore configuration parameters.

3.13 The FTS Template Dialog

Use the *FTS Template* dialog to create a new text search template. A text search template defines the functions that implement text search dictionaries.

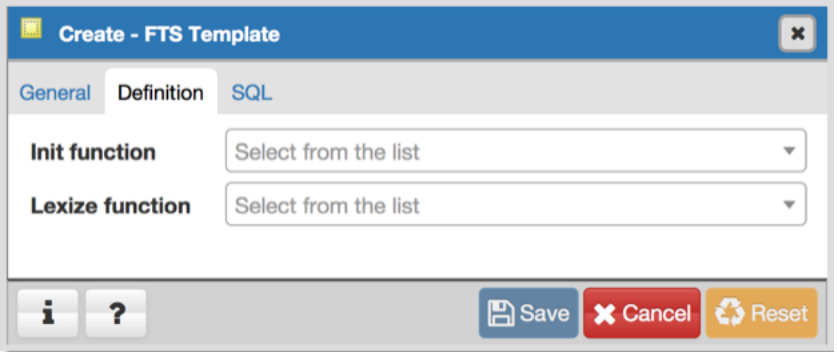
The *FTS Template* dialog organizes the development of a text search Template through the following dialog tabs: *General*, and *Definition*. The *SQL* tab displays the SQL code generated by dialog selections.



Use the fields in the *General* tab to identify a template:

- Use the *Name* field to add a descriptive name for the template. The name will be displayed in the *pgAdmin* tree control.
- Select the name of the schema in which the template will reside from the drop-down listbox in the *Schema* field.
- Store notes about the template in the *Comment* field.

Click the *Definition* tab to continue.



Use the fields in the *Definition* tab to define function parameters:

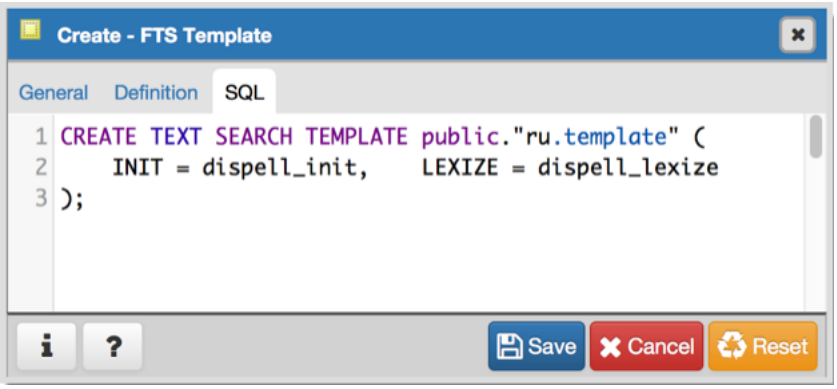
- Use the drop-down listbox next to *Init function* to select the name of the init function for the template. The init function is optional.
- Use the drop-down listbox next to *Lexize function* to select the name of the lexize function for the template. The lexize function is required.

Click the *SQL* tab to continue.

Your entries in the *FTS Template* dialog generate a SQL command (see an example below). Use the *SQL* tab for review; revisit or switch tabs to make any changes to the SQL command.

Example

The following is an example of the sql command generated by user selections in the *FTS Template* dialog:



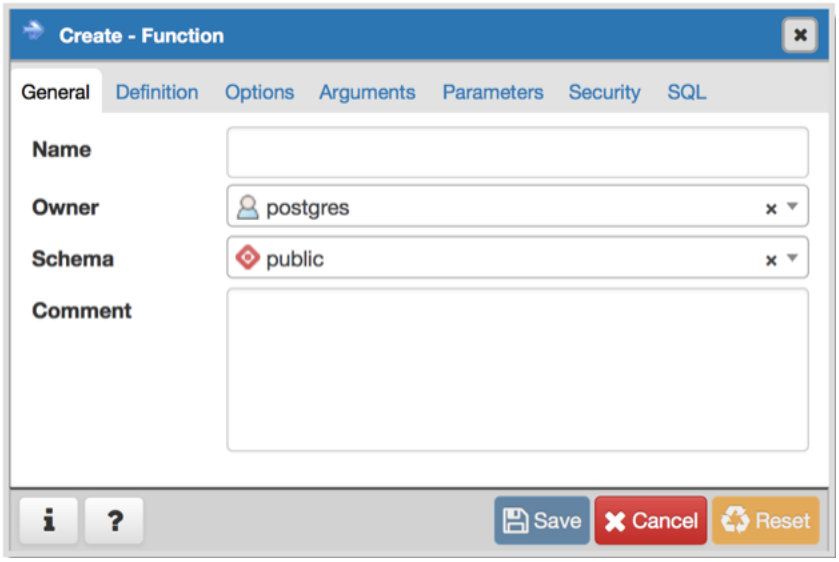
The example shown demonstrates creating a fts template named *ru_template* that uses the ispell dictionary.

- Click the *Info* button (i) to access online help. View context-sensitive help in the *Tabbed browser*, where a new tab displays the PostgreSQL core documentation.
- Click the *Save* button to save work.
- Click the *Cancel* button to exit without saving work.
- Click the *Reset* button to restore configuration parameters.

3.14 The Function Dialog

Use the *Function* dialog to define a function. If you drop and then recreate a function, the new function is not the same entity as the old; you must drop existing rules, views, triggers, etc. that refer to the old function.

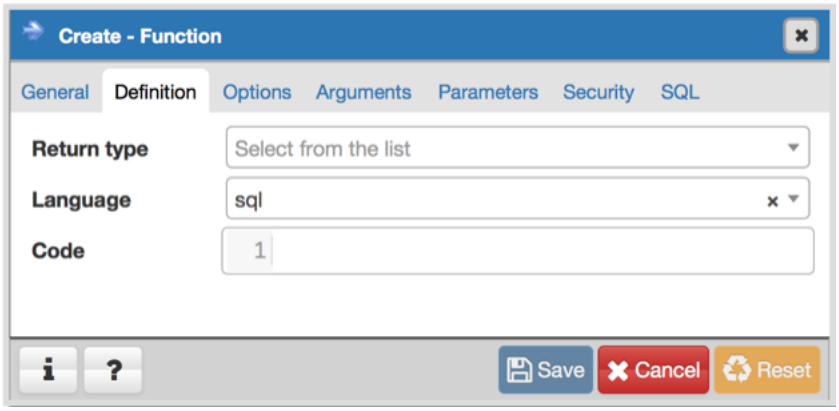
The *Function* dialog organizes the development of a function through the following dialog tabs: *General*, *Definition*, *Options*, *Arguments*, *Parameters*, and *Security*. The *SQL* tab displays the SQL code generated by dialog selections.



Use the fields in the *General* tab to identify a function:

- Use the *Name* field to add a descriptive name for the function. The name will be displayed in the *pgAdmin* tree control.
- Use the drop-down listbox next to *Owner* to select the name of the role that will own the function.
- Use the drop-down listbox next to *Schema* to select the schema in which the function will be created.
- Store notes about the function in the *Comment* field.

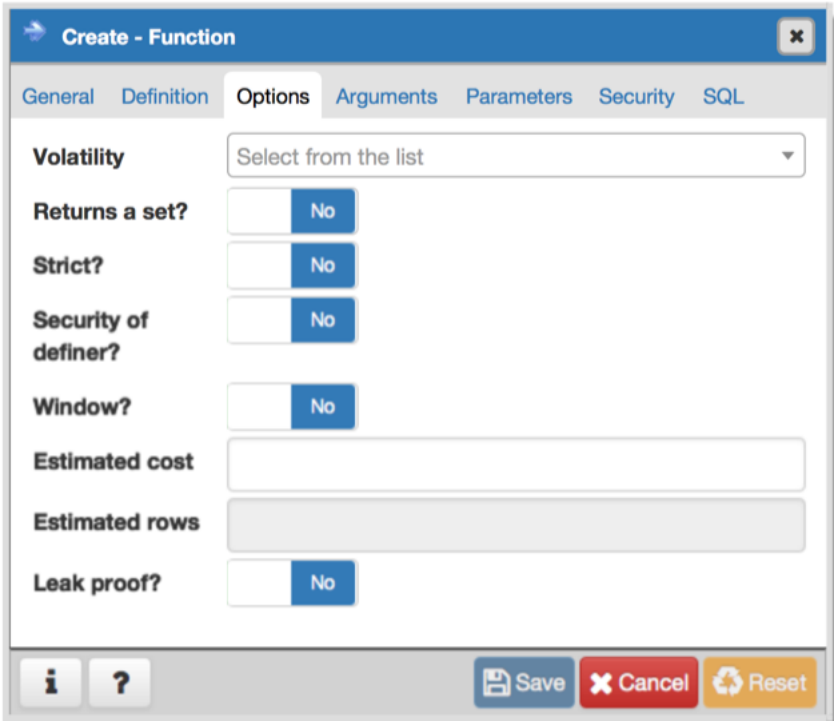
Click the *Definition* tab to continue.



Use the fields in the *Definition* tab to define the function:

- Use the drop-down listbox next to *Return type* to select the data type returned by the function, if any.
- Use the drop-down listbox next to *Language* to select the implementation language. The default is *sql*.
- Use the *Code* field to write the code that will execute when the function is called.

Click the *Options* tab to continue.



Use the fields in the *Options* tab to describe or modify the action of the function:

- Use the drop-down listbox next to *Volatility* to select one of the following. *VOLATILE* is the default value.
  - *VOLATILE* indicates that the function value can change even within a single table scan, so no optimizations can be made.
  - *STABLE* indicates that the function cannot modify the database, and that within a single table scan it will consistently return the same result for the same argument values.
  - *IMMUTABLE* indicates that the function cannot modify the database and always returns the same result when given the same argument values.
- Move the *Returns a Set?* switch to indicate if the function returns a set that includes multiple rows. The default is *No*.
- Move the *Strict?* switch to indicate if the function always returns NULL whenever any of its arguments are NULL. If *Yes*, the function is not executed when there are NULL arguments; instead a NULL result is assumed automatically. The default is *No*.
- Move the *Security of definer?* switch to specify that the function is to be executed with the privileges of the user that created it. The default is *No*.
- Move the *Window?* switch to indicate that the function is a window function rather than a plain function. The default is *No*. This is currently only useful for functions written in C. The WINDOW attribute cannot be changed when replacing an existing function definition. For more information about the CREATE FUNCTION command, see the PostgreSQL core documentation available at:

  http://www.postgresql.org/docs/9.5/static/functions-window.html
- Use the *Estimated cost* field to specify a positive number representing the estimated execution cost for the function, in units of cpu_operator_cost. If the function returns a set, this is the cost per returned row.
- Use the *Estimated rows* field to specify a positive number giving the estimated number of rows that the query planner should expect the function to return. This is only allowed when the function is declared to return a set. The default assumption is 1000 rows.

- Move the *Leak proof?* switch to indicate whether the function has side effects. The default is *No*. This option can only be set by the superuser.

Click the *Arguments* tab to continue.

Use the fields in the *Arguments* tab to define an argument. Click the *Add* icon (+) to set parameters and values for the argument:

- Use the drop-down listbox in the *Data type* field to select a data type.
- Use the drop-down listbox in the *Mode* field to select a mode. Select *IN* for an input parameter; select *OUT* for an output parameter; select *INOUT* for both an input and an output parameter; or, select *VARIADIC* to specify a VARIADIC parameter.
- Provide a name for the argument in the *Argument Name* field.
- Specify a default value for the argument in the *Default Value* field.

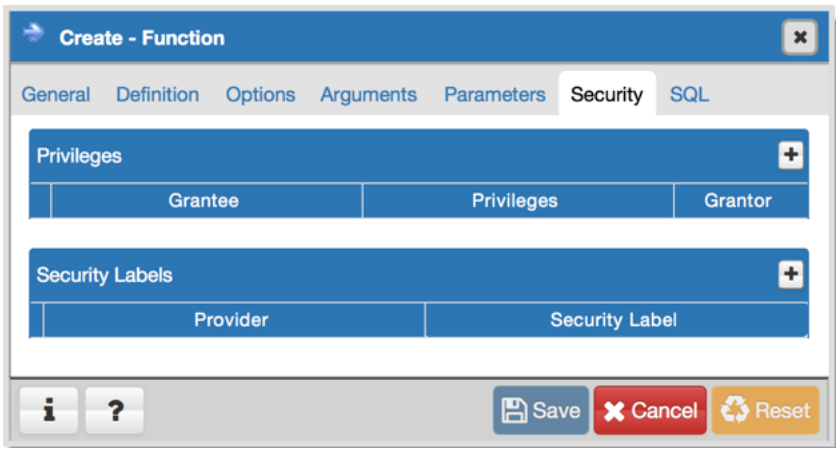
Click the *Add* icon (+) to define another argument; to discard an argument, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Click the *Parameters* tab to continue.

Use the fields in the *Parameters* tab to specify settings that will be applied when the function is invoked. Click the *Add* icon (+) to add a *Name/Value* field in the table.

- Use the drop-down listbox in the *Name* column in the *Parameters* panel to select a parameter.
- Use the *Value* field to specify the value that will be associated with the selected variable. This field is context-sensitive.

Click the *Security* tab to continue.



Use the *Security* tab to assign privileges and define security labels.

Use the *Privileges* panel to assign usage privileges for the function to a role.

- Select the name of the role from the drop-down listbox in the *Grantee* field.
- Click inside the *Privileges* field. Check the boxes to the left of one or more privileges to grant the selected privilege to the specified user.
- Select the name of the role from the drop-down listbox in the *Grantor* field. The default grantor is the owner of the database.

Click the *Add* icon (+) to assign additional privileges; to discard a privilege, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Use the *Security Labels* panel to define security labels applied to the function. Click the *Add* icon (+) to add each security label selection:

- Specify a security label provider in the *Provider* field. The named provider must be loaded and must consent to the proposed labeling operation.
- Specify a a security label in the *Security Label* field. The meaning of a given label is at the discretion of the label provider. PostgreSQL places no restrictions on whether or how a label provider must interpret security labels; it merely provides a mechanism for storing them.

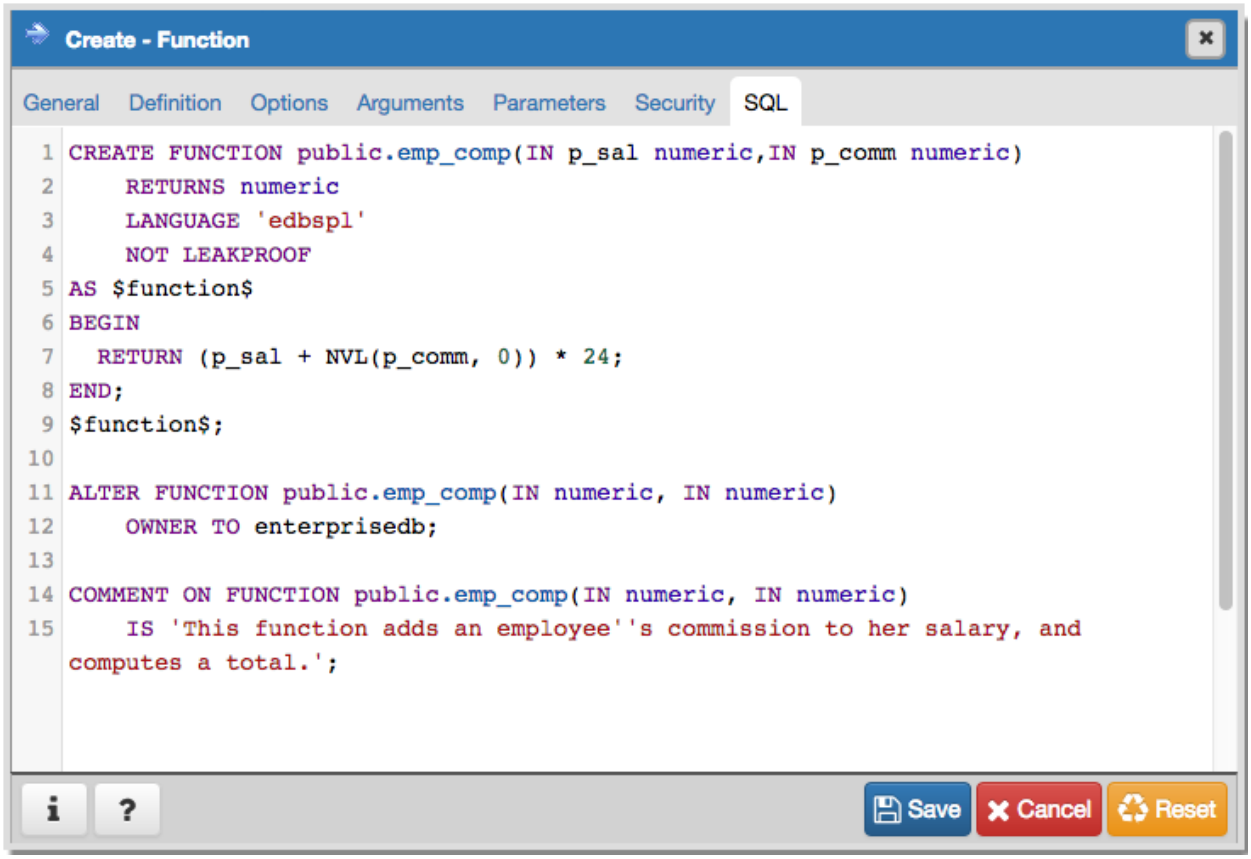
Click the *Add* icon (+) to assign additional security labels; to discard a security label, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Click the *SQL* tab to continue.

Your entries in the *Function* dialog generate a generate a SQL command (see an example below). Use the *SQL* tab for review; revisit or switch tabs to make any changes to the SQL command.

Example

The following is an example of the sql command generated by selections made in the *Function* dialog:



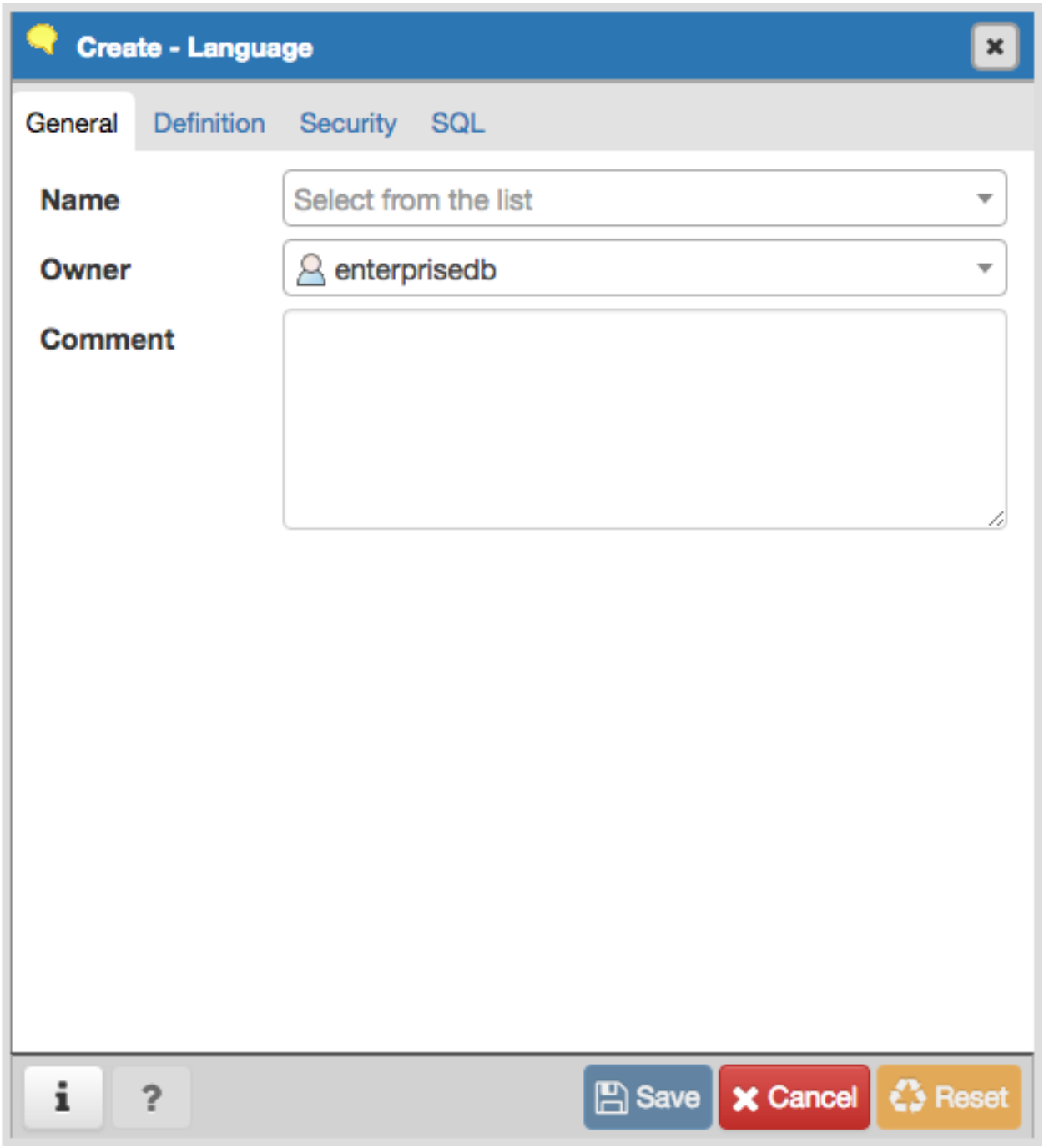
The example demonstrates creating an *edbspl* function named *emp_comp*. The function adds two columns (p_sal and p_comm), and then uses the result to compute a yearly salary, returning a NUMERIC value.

- Click the *Info* button (i) to access online help.View context-sensitive help in the *Tabbed browser*, where a new tab displays the PostgreSQL core documentation.
- Click the *Save* button to save work.
- Click the *Cancel* button to exit without saving work.
- Click the *Reset* button to restore configuration parameters.

3.15 The Language Dialog

Use the CREATE LANGUAGE dialog to register a new procedural language.

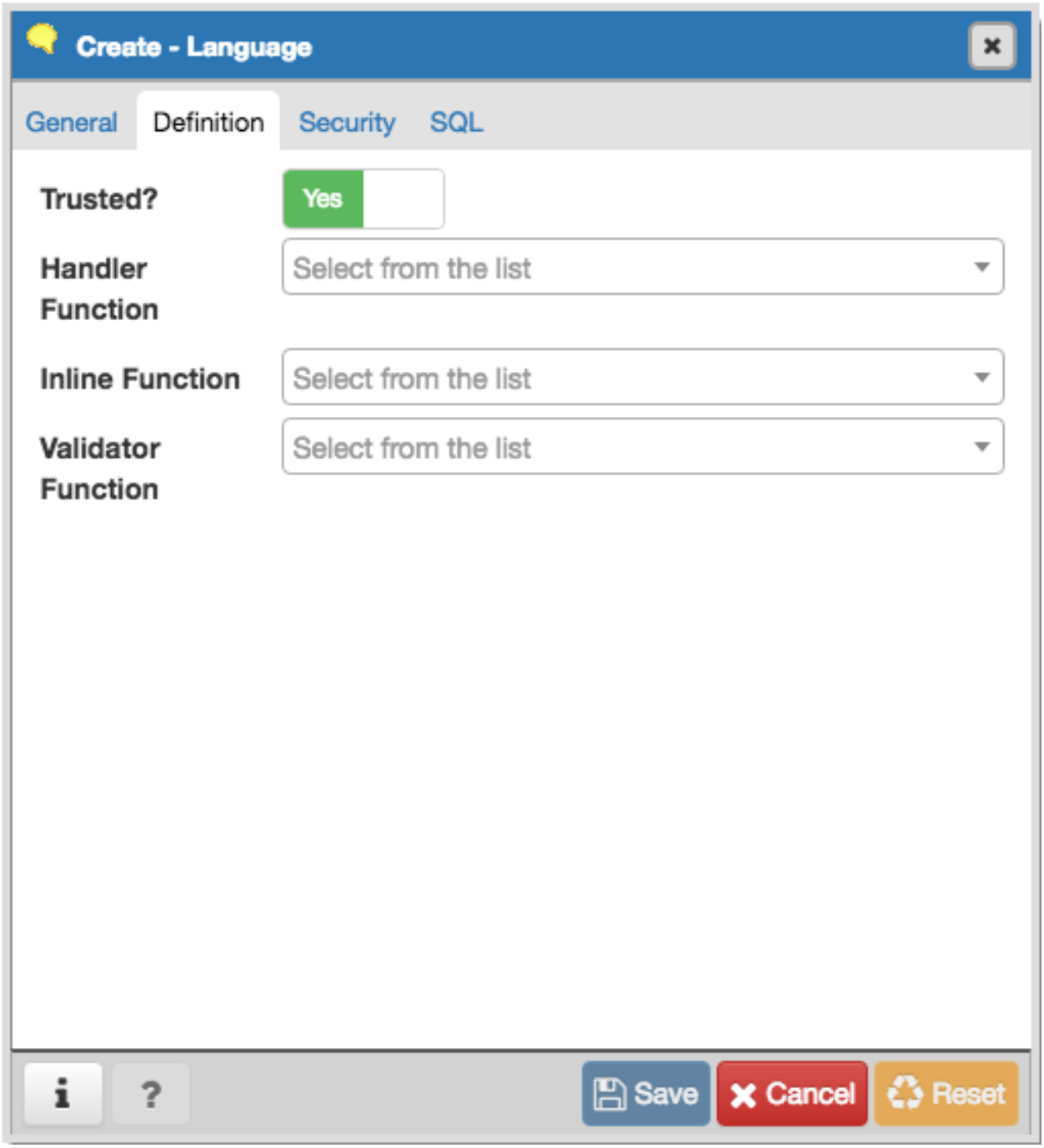
The *Language* dialog organizes the registration of a procedural language through the following dialog tabs: *General*, *Definition*, and *Security*. The *SQL* tab displays the SQL code generated by dialog selections.



Use the fields in the *General* tab to identify a language:

- Use the drop-down listbox next to *Name* to select a language script.
- Use the drop-down listbox next to *Owner* to select a role.
- Store notes about the language in the *Comment* field.

Click the *Definition* tab to continue.

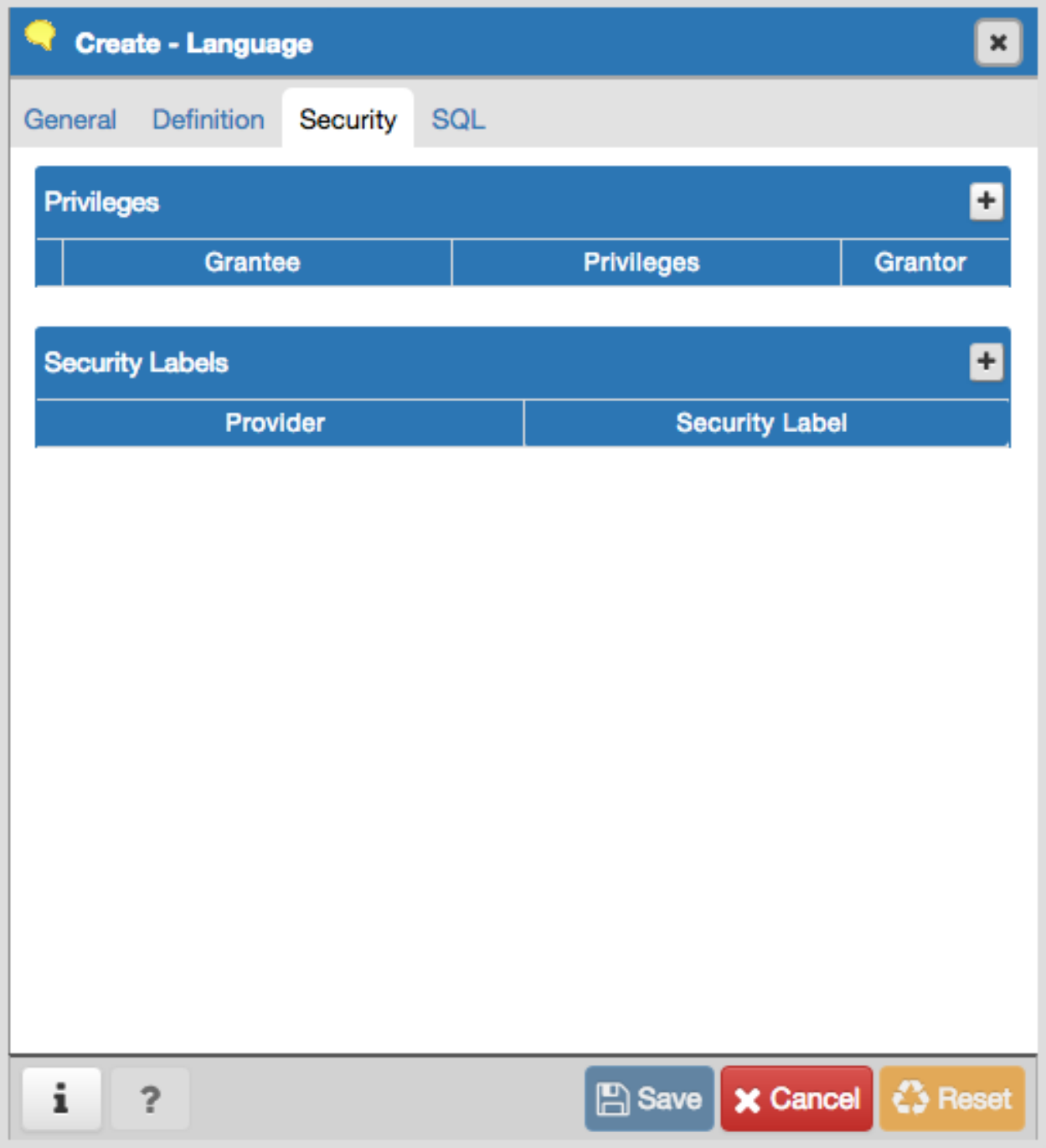


Use the fields in the *Definition* tab to define parameters:

- Move the *Trusted?* switch to the *No* position to specify only users with PostgreSQL superuser privilege can use this language. The default is *Yes*.
- When enabled, use the drop-down listbox next to *Handler Function* to select the function that will be called to execute the language's functions.
- When enabled, use the drop-down listbox next to *Inline Function* to select the function that will be called to execute an anonymous code block (DO command) in this language.
- When enabled, use the drop-down listbox next to *Validator Function* to select the function that will be called

when a new function in the language is created, to validate the new function.

Click the *Security* tab to continue.



Use the *Security* tab to assign privileges and define security labels.

Use the *Privileges* panel to assign privileges to a role. Click the *Add* icon (+) to set privileges for database objects:

- Select the name of the role from the drop-down listbox in the *Grantee* field.
- Click inside the *Privileges* field. Check the boxes to the left of one or more privileges to grant the selected privilege to the specified user.

- Select the name of the role from the drop-down listbox in the *Grantor* field. The default grantor is the owner of the database.

Click the *Add* icon (+) to assign additional privileges; to discard a privilege, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Use the *Security Labels* panel to define security labels applied to the function. Click the *Add* icon (+) to add each security label selection:

- Specify a security label provider in the *Provider* field. The named provider must be loaded and must consent to the proposed labeling operation.
- Specify a a security label in the *Security Label* field. The meaning of a given label is at the discretion of the label provider. PostgreSQL places no restrictions on whether or how a label provider must interpret security labels; it merely provides a mechanism for storing them.

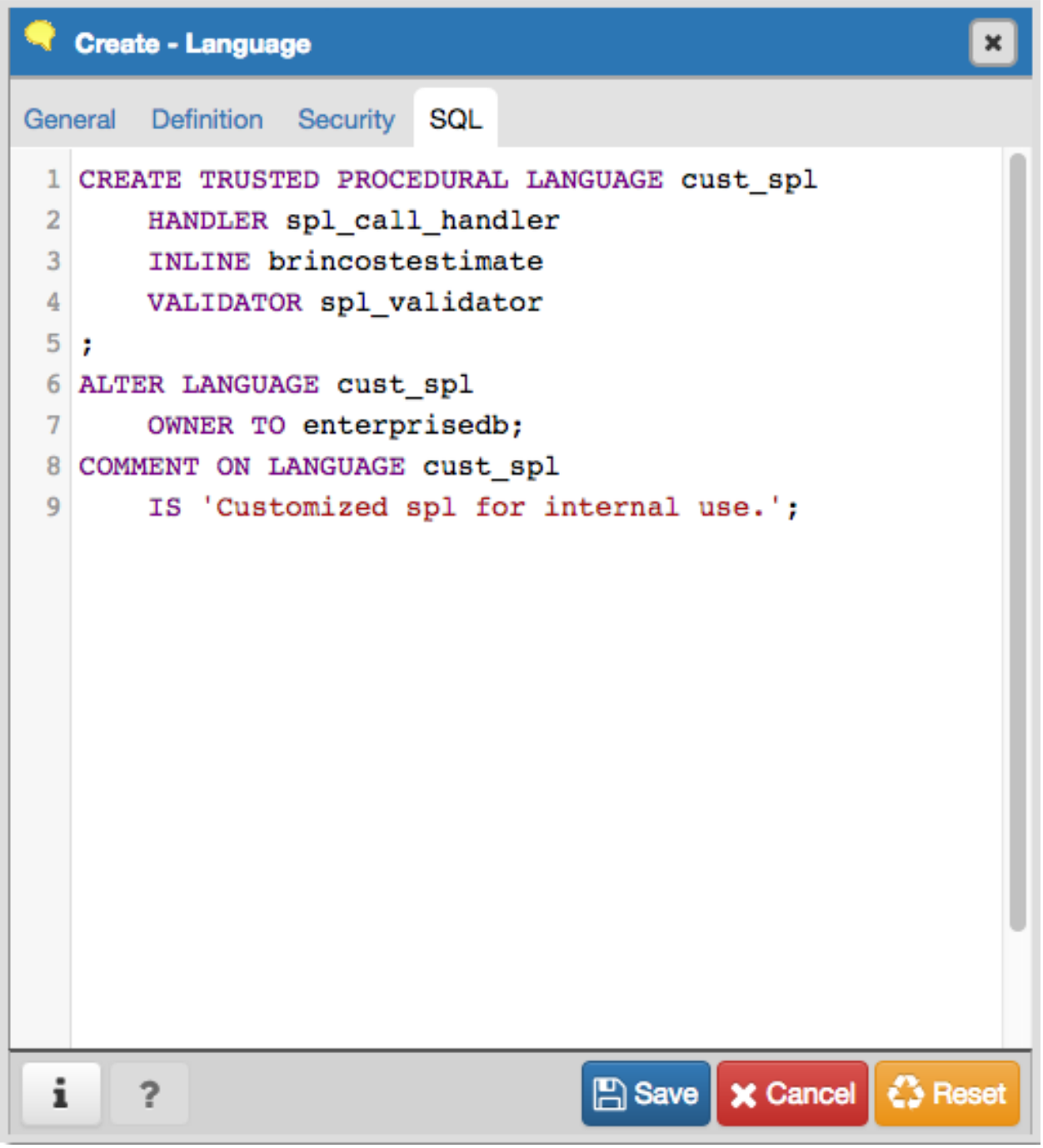
Click the *Add* icon (+) to assign additional security labels; to discard a security label, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Click the *SQL* tab to continue.

Your entries in the *Language* dialog generate a SQL command (see an example below). Use the *SQL* tab for review; revisit or switch tabs to make any changes to the SQL command.

Example

The following is an example of the sql command generated by user selections in the *Language* dialog:



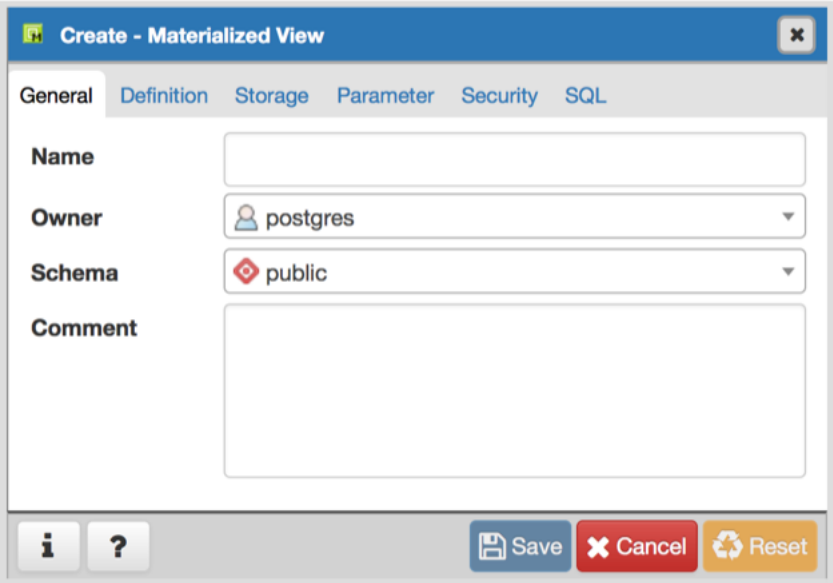
"The example shown demonstrates creating the procedural language named *plperl*."

- Click the *Info* button (i) to access online help. View context-sensitive help in the *Tabbed browser*, where a new tab displays the PostgreSQL core documentation.
- Click the *Save* button to save work.
- Click the *Cancel* button to exit without saving work.
- Click the *Reset* button to restore configuration parameters.

3.16 The Materialized View Dialog

Use the *Materialized View* dialog to define a materialized view. A materialized view is a stored or cached view that contains the result set of a query. Use the REFRESH MATERIALIZED VIEW command to update the content of a materialized view.

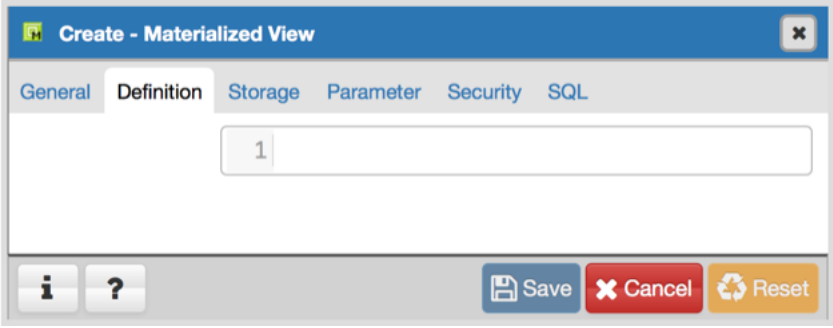
The *Materialized View* dialog organizes the development of a materialized_view through the following dialog tabs: *General*, *Definition*, *Storage*, *Parameter*, and *Security*. The *SQL* tab displays the SQL code generated by dialog selections.



Use the fields in the *General* tab to identify the materialized view:

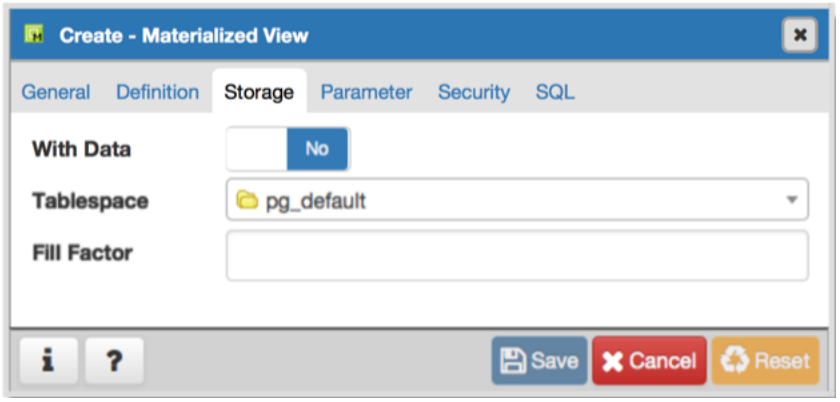
- Use the *Name* field to add a descriptive name for the materialized view. The name will be displayed in the *pgAdmin* tree control.
- Use the drop-down listbox next to *Owner* to select the role that will own the materialized view.
- Select the name of the schema in which the materialized view will reside from the drop-down listbox in the *Schema* field.
- Store notes about the materialized view in the *Comment* field.

Click the *Definition* tab to continue.



Use the text editor field in the *Definition* tab to provide the query that will populate the materialized view.

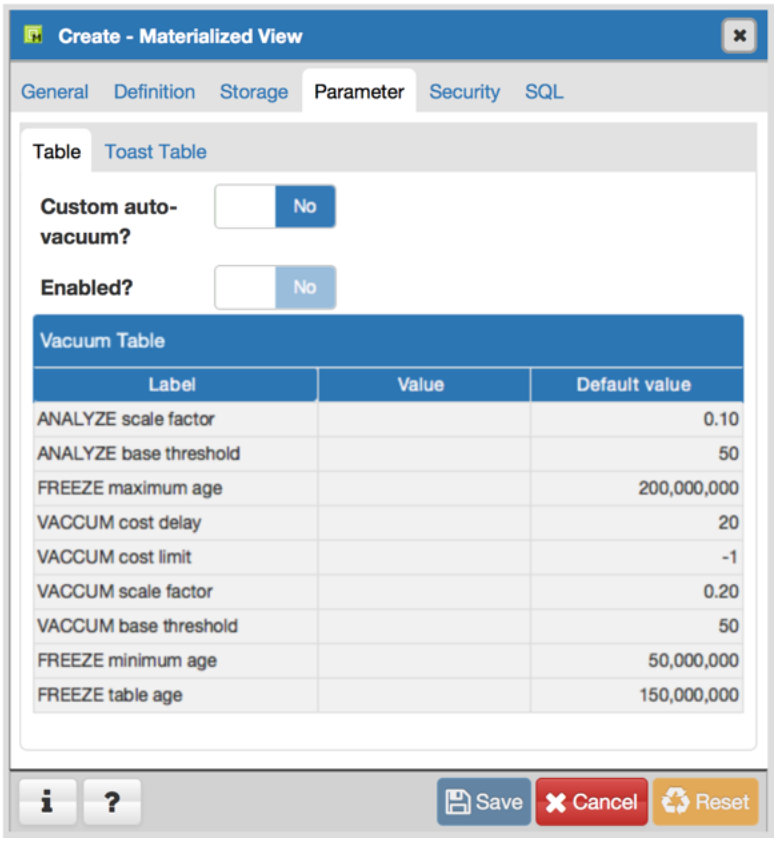
Click the *Storage* tab to continue.



Use the fields in the *Storage* tab to maintain the materialized view:

- Move the *With Data* switch to the *Yes* position to specify the materialized view should be populated at creation time. If not, the materialized view cannot be queried until you invoke REFRESH MATERIALIZED VIEW.
- Use the drop-down listbox next to *Tablespace* to select a location for the materialized view.
- Use the *Fill Factor* field to specify a fill factor for the materialized view. The fill factor for a table is a percentage between 10 and 100. 100 (complete packing) is the default.

Click the *Parameter* tab to continue.

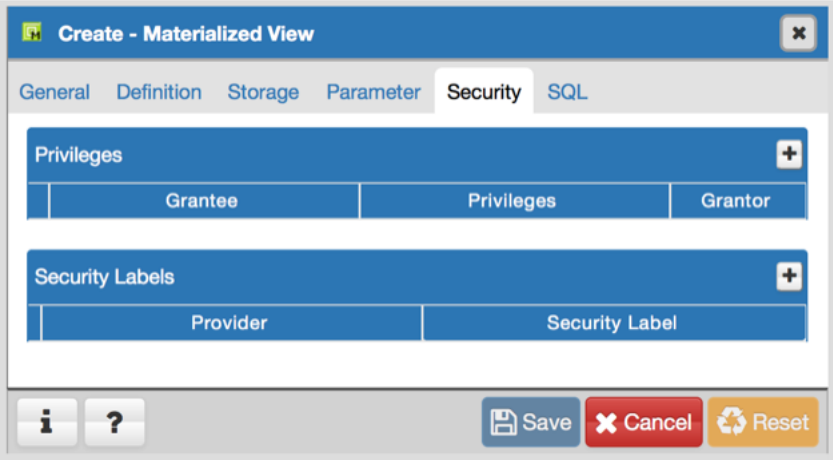


Use the tabs nested inside the *Parameter* tab to specify VACUUM and ANALYZE thresholds; use the *Table* tab and the *Toast Table* tab to customize values for the table and the associated toast table. To change the default values:

- Move the *Custom auto-vacuum?* switch to the *Yes* position to perform custom maintenance on the materialized view.

- Move the *Enabled?* switch to the *Yes* position to select values in the *Vacuum table*. Provide values for each row in the *Value* column.

Click the *Security* tab to continue.



Use the *Security* tab to assign privileges and define security labels.

Use the *Privileges* panel to assign privileges to a role. Click the *Add* icon (+) to set privileges for the materialized view:

- Select the name of the role from the drop-down listbox in the *Grantee* field.
- Click inside the *Privileges* field. Check the boxes to the left of one or more privileges to grant the selected privilege to the specified user.
- Select the name of the role from the drop-down listbox in the *Grantor* field. The default grantor is the owner of the database.

Click the *Add* icon (+) to assign additional privileges; to discard a privilege, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Use the *Security Labels* panel to define security labels applied to the materialized view. Click the *Add* icon (+) to add each security label selection:

- Specify a security label provider in the *Provider* field. The named provider must be loaded and must consent to the proposed labeling operation.
- Specify a a security label in the *Security Label* field. The meaning of a given label is at the discretion of the label provider. PostgreSQL places no restrictions on whether or how a label provider must interpret security labels; it merely provides a mechanism for storing them.

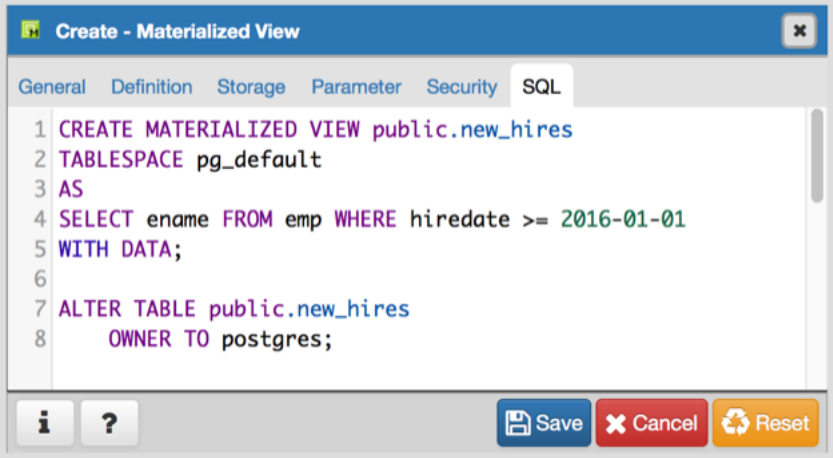
Click the *Add* icon (+) to assign additional security labels; to discard a security label, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Click the *SQL* tab to continue.

Your entries in the *Materialized View* dialog generate a SQL command (see an example below). Use the *SQL* tab for review; revisit or switch tabs to make any changes to the SQL command.

Example

The following is an example of the sql command generated by user selections in the *Materialized View* dialog:



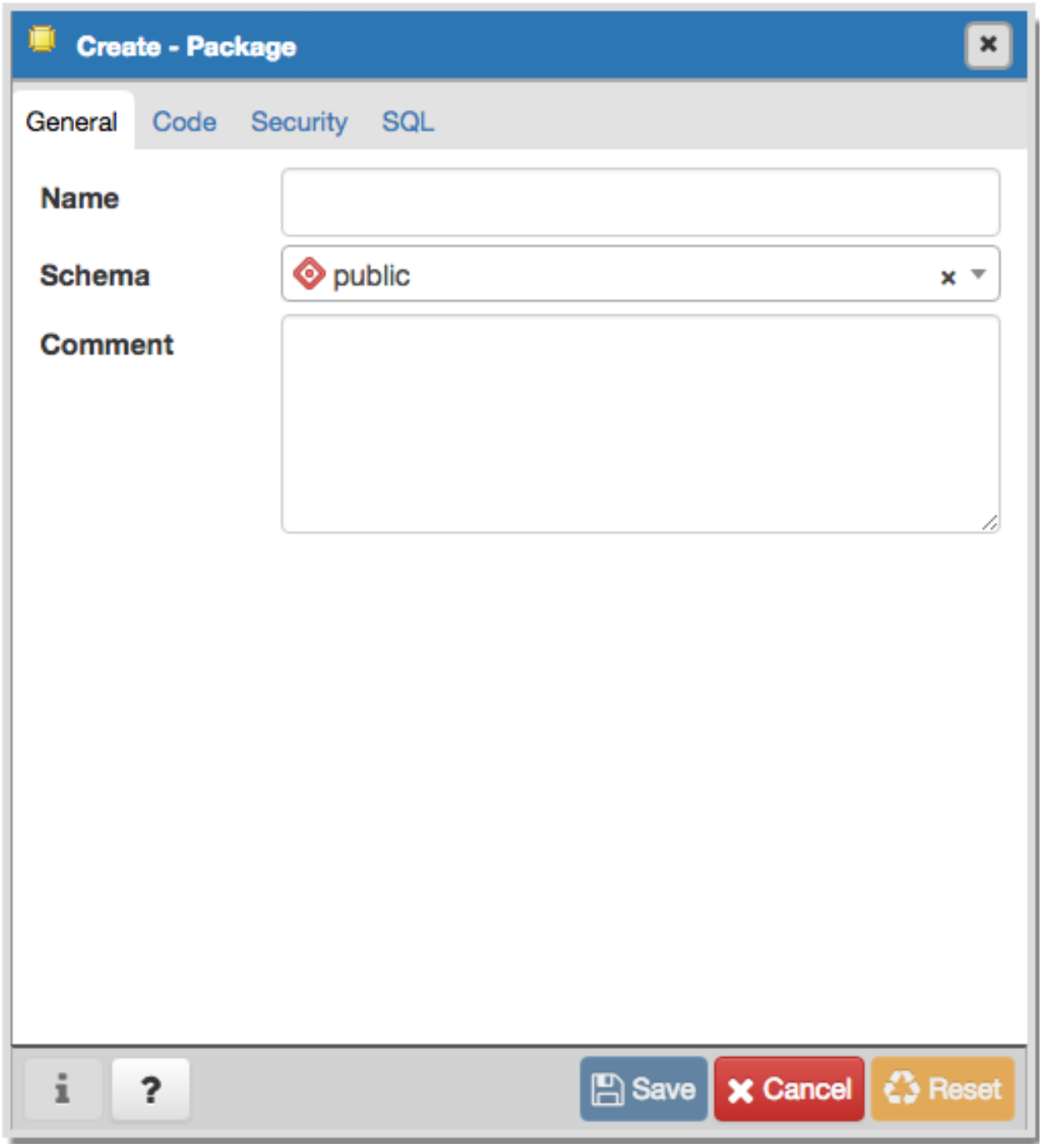
The example shown creates a query named *new_hires* that stores the result of the displayed query in the *pg_default* tablespace.

- Click the *Info* button (i) to access online help. View context-sensitive help in the *Tabbed browser*, where a new tab displays the PostgreSQL core documentation.
- Click the *Save* button to save work.
- Click the *Cancel* button to exit without saving work.
- Click the *Reset* button to restore configuration parameters.

3.17 The Package Dialog

Use the *Package* dialog to create a (user-defined) package specification.

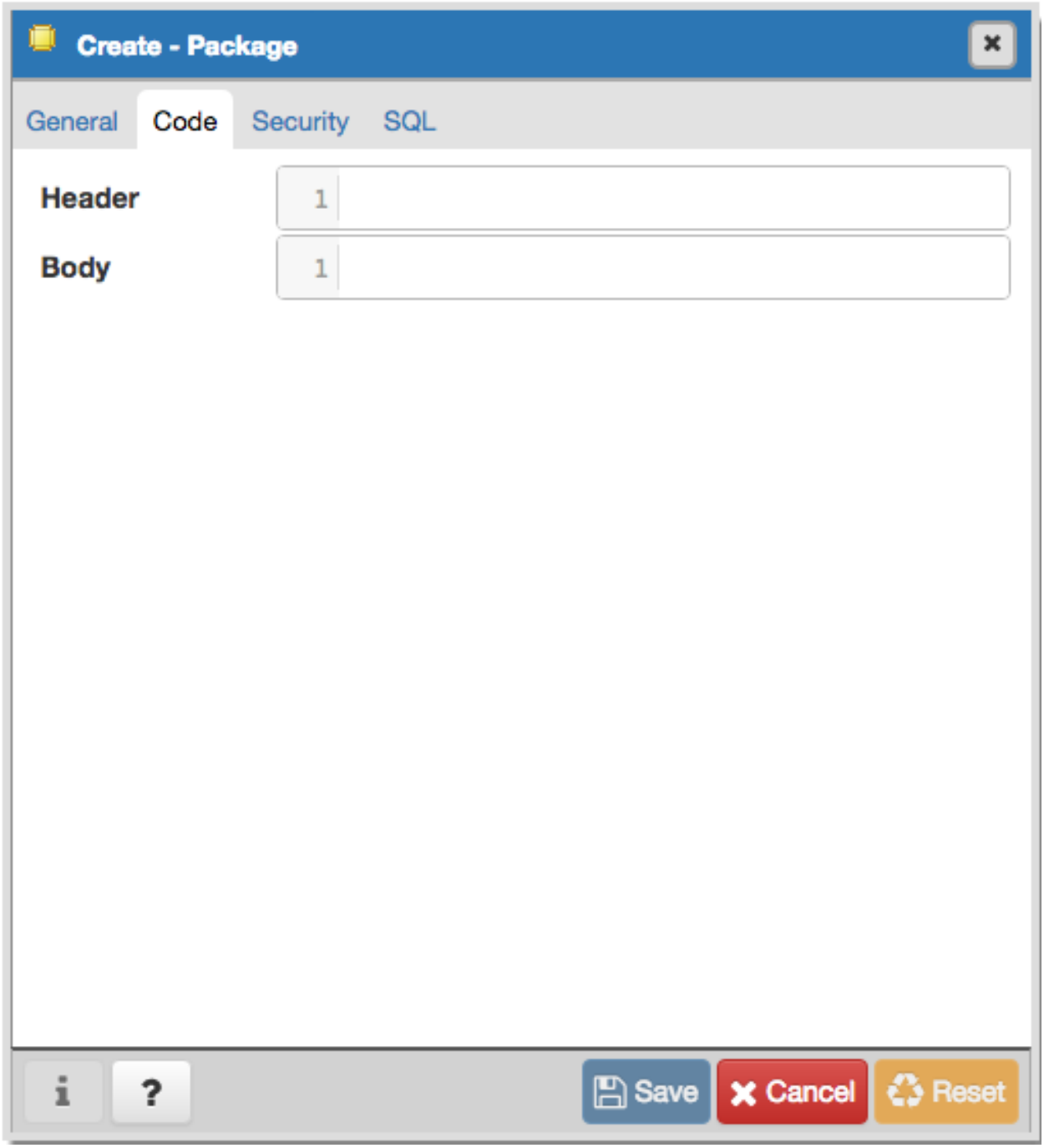
The *Package* dialog organizes the management of a package through the following dialog tabs: *General*, *Code*, and *Security*. The *SQL* tab displays the SQL code generated by dialog selections.



Use the fields in the *General* tab to identify the package:

- Use the *Name* field to add a descriptive name for the package. The name of a new package must not match any existing package in the same schema.
- Select the schema in which the package will reside from the drop-down listbox in the *Schema* field.
- Store notes about the package in the *Comment* field.

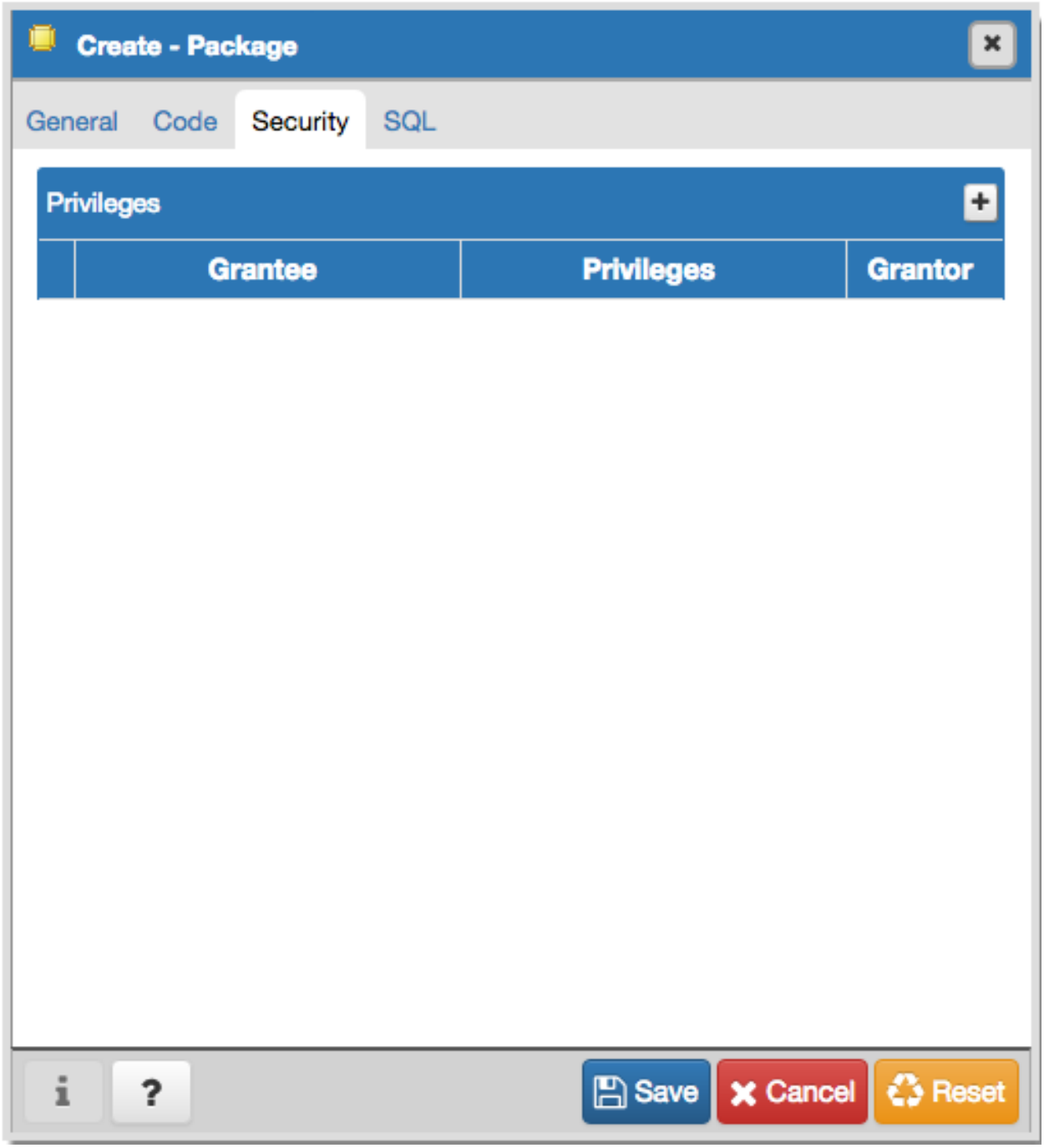
Click the *Code* tab to continue.



Use the fields in the *Code* tab to specify the package contents and to provide implementation details:

- Use the *Header* field to define the public interface for the package.
- Use the *Body* field to provide the code that implements each package object.

Click the *Security* tab to continue.



Use the fields in the *Security* tab to to assign EXECUTE privileges for the package to a role. Click the *Add* icon (+) to set privileges for the package:

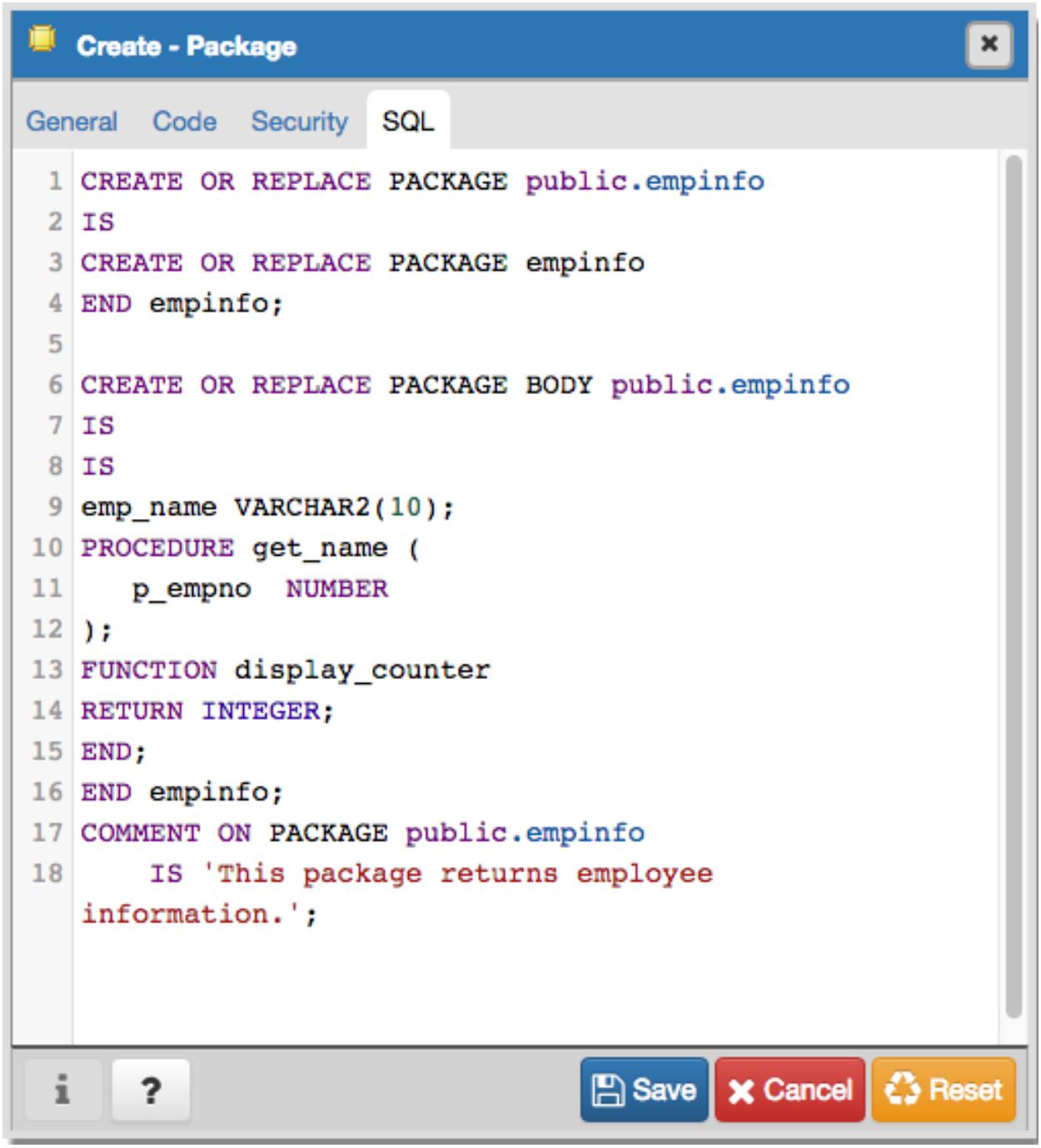
- Select the name of the role from the drop-down listbox in the *Grantee* field.
- Click inside the *Privileges* field. Check the boxes to the left of a privilege to grant the selected privilege to the specified user.
- Select the name of a role from the drop-down listbox in the *Grantor* field. The default grantor is the owner of the package.

Click the *Add* icon (+) to assign additional privileges; to discard a privilege, click the trash icon to the left of the row,

and confirm the deletion in the *Delete Row* popup.

Click the *SQL* tab to continue.

Your entries in the *Package* dialog generate a SQL command that creates or modifies a package definition:



The example shown demonstrates creating a package named *empinfo* that includes one function and one procedure.

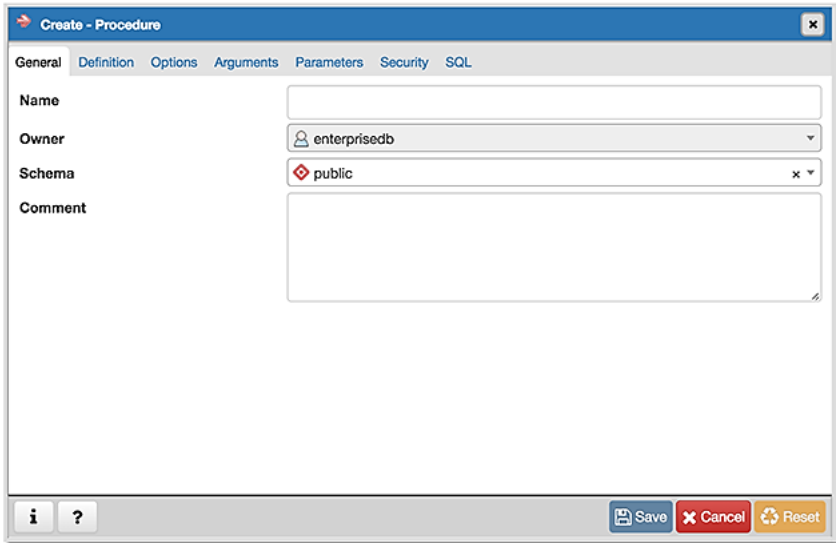
- Click the *Save* button to save work.
- Click the *Cancel* button to exit without saving work.
- Click the *Reset* button to delete any changes to the dialog.

3.18 The Procedure Dialog

Use the *Procedure* dialog to create a procedure; procedures are supported by EDB Postgres Advanced Server. The *Procedure* dialog allows you to implement options of the CREATE PROCEDURE command; for more information about the CREATE PROCEDURE SQL command, please see the Database Compatibility for Oracle Developer's, available at:

http://www.enterprisedb.com

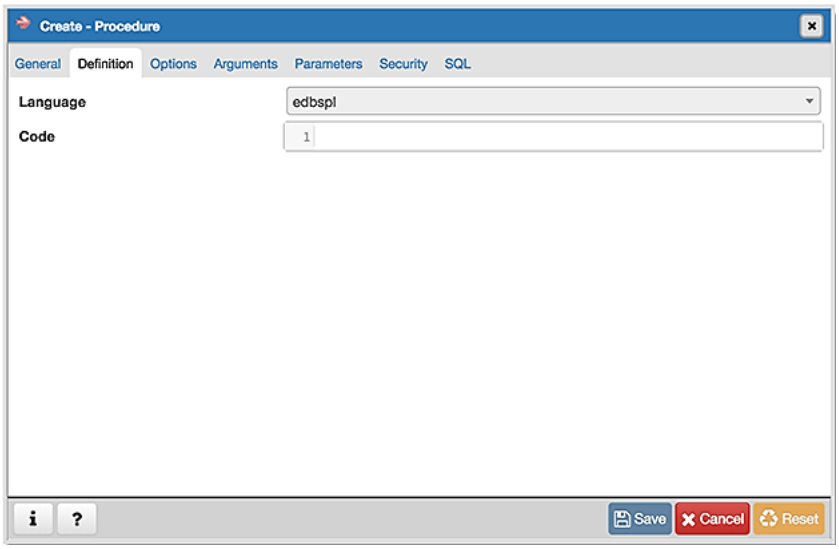
The *Procedure* dialog organizes the development of a procedure through the following dialog tabs: *General*, *Definition*, *Options*, *Arguments*, *Parameters*, and *Security*. The *SQL* tab displays the SQL code generated by dialog selections.



Use the fields in the *General* tab to identify a procedure:

- Use the *Name* field to add a descriptive name for the procedure. The name will be displayed in the *pgAdmin* tree control.
- Use the drop-down listbox next to *Owner* to select a role.
- Select the name of the schema in which the procedure will reside from the drop-down listbox in the *Schema* field.
- Store notes about the procedure in the *Comment* field.

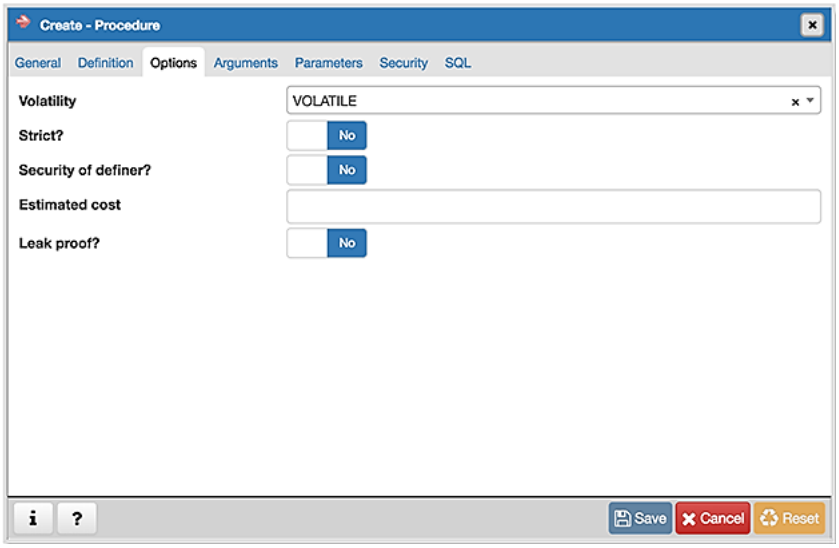
Click the *Definition* tab to continue.



Use the fields in the *Definition* tab to define the procedure:

- Use the drop-down listbox next to *Language* to select a language. The default is *edbspl*.
- Use the *Code* field to specify the code that will execute when the procedure is called.

Click the *Options* tab to continue.

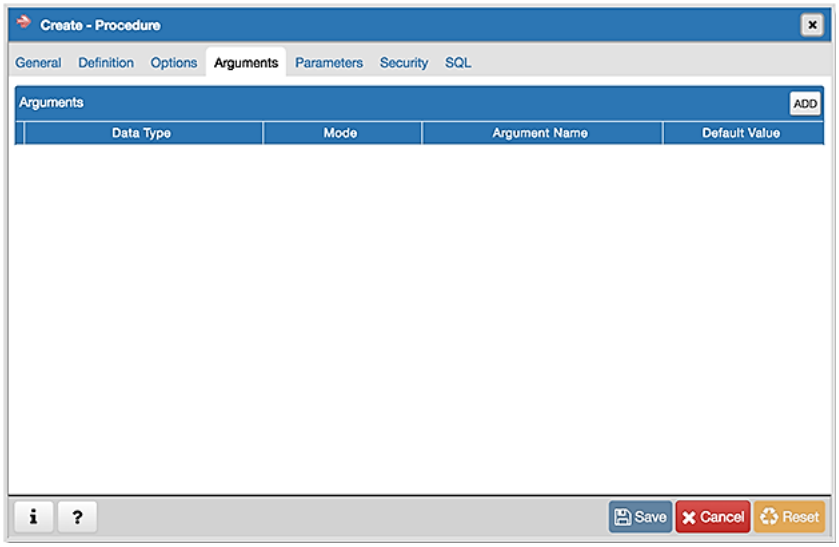


Use the fields in the *Options* tab to describe or modify the behavior of the procedure:

- Use the drop-down listbox under *Volatility* to select one of the following. *VOLATILE* is the default value.
  - *VOLATILE* indicates that the value can change even within a single table scan, so no optimizations can be made.
  - *STABLE* indicates that the procedure cannot modify the database, and that within a single table scan it will consistently return the same result for the same argument values, but that its result could change across SQL statements.
  - *IMMUTABLE* indicates that the procedure cannot modify the database and always returns the same result when given the same argument values.

- Move the *Strict?* switch to indicate if the procedure always returns NULL whenever any of its arguments are NULL. If *Yes*, the procedure is not executed when there are NULL arguments; instead a NULL result is assumed automatically. The default is *No*.
- Move the *Security of definer?* switch to specify that the procedure is to be executed with the privileges of the user that created it. The default is *No*.
- Use the *Estimated cost* field to specify a positive number representing the estimated execution cost for the procedure, in units of cpu_operator_cost. If the procedure returns a set, this is the cost per returned row.
- Move the *Leak proof?* switch to indicate whether the procedure has side effects — it reveals no information about its arguments other than by its return value. The default is *No*.

Click the *Arguments* tab to continue.

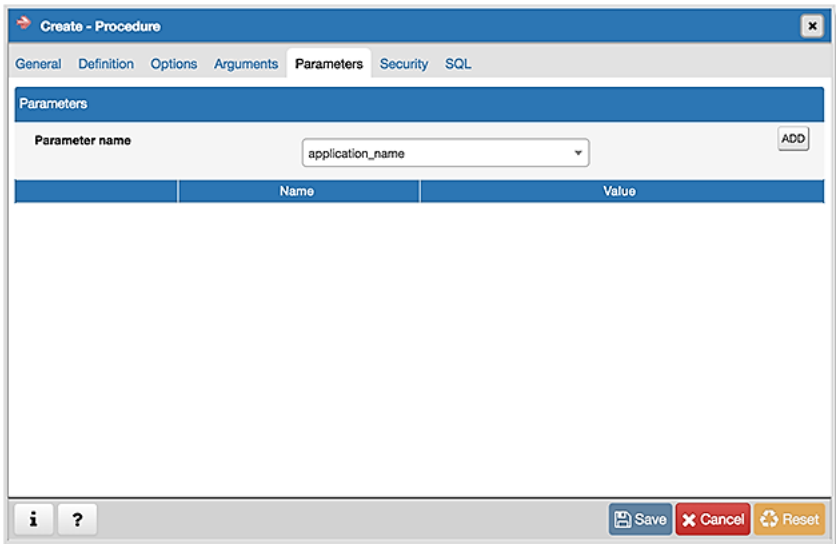


Use the fields in the *Arguments* tab to define an argument. Click *Add* to set parameters and values for the argument:

- Use the drop-down listbox next to *Data type* to select a data type.
- Use the drop-down listbox next to *Mode* to select a mode. Select *IN* for an input parameter; select *OUT* for an output parameter; select *INOUT* for both an input and an output parameter; or, select *VARIADIC* to specify a VARIADIC parameter.
- Write a name for the argument in the *Argument Name* field.
- Specify a default value for the argument in the *Default Value* field.

Click *Add* to define another argument; to discard an argument, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

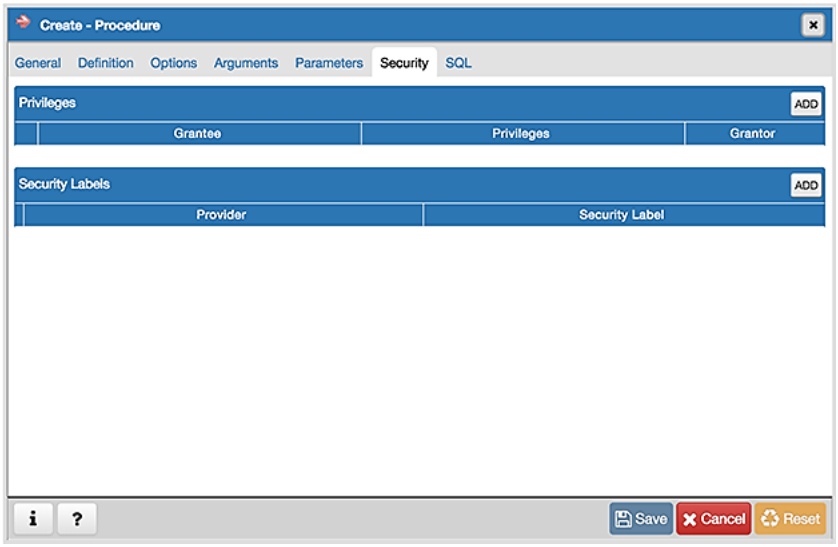
Click the *Parameters* tab to continue.



Use the fields in the *Parameters* tab to specify settings that will be applied when the procedure is invoked:

- Use the drop-down listbox next to *Parameter Name* in the *Parameters* panel to select a parameter.
- Click the *Add* button to add the variable to *Name* field in the table.
- Use the *Value* field to specify the value that will be associated with the selected variable. This field is context-sensitive.

Click the *Security* tab to continue.



Use the *Security* tab to assign privileges and define security labels.

Use the *Privileges* panel to assign execute privileges for the procedure to a role:

- Select the name of the role from the drop-down listbox in the *Grantee* field.
- Click inside the *Privileges* field. Check the boxes to the left of one or more privileges to grant the selected privilege to the specified user.
- Select the name of the role from the drop-down listbox in the *Grantor* field. The default grantor is the owner of the database.

Click *Add* to assign additional privileges; to discard a privilege, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Use the *Security Labels* panel to define security labels applied to the procedure. Click *Add* to add each security label selection:

- Specify a security label provider in the *Provider* field. The named provider must be loaded and must consent to the proposed labeling operation.
- Specify a a security label in the *Security Label* field. The meaning of a given label is at the discretion of the label provider. PostgreSQL places no restrictions on whether or how a label provider must interpret security labels; it merely provides a mechanism for storing them.

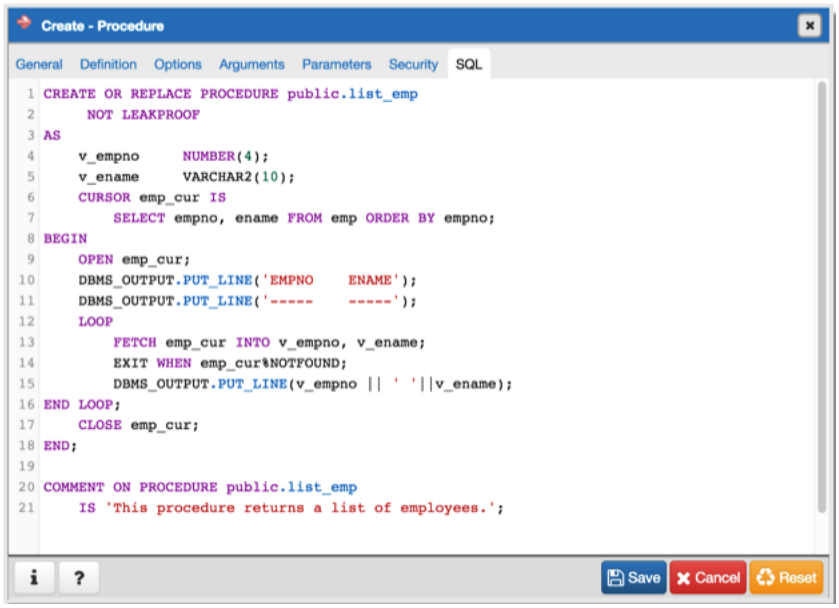
Click *Add* to assign additional security labels; to discard a security label, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Click the *SQL* tab to continue.

Your entries in the *Procedure* dialog generate a SQL command (see an example below). Use the *SQL* tab for review; revisit or switch tabs to make any changes to the SQL command.

Example

The following is an example of the sql command generated by selections made in the *Procedure* dialog:



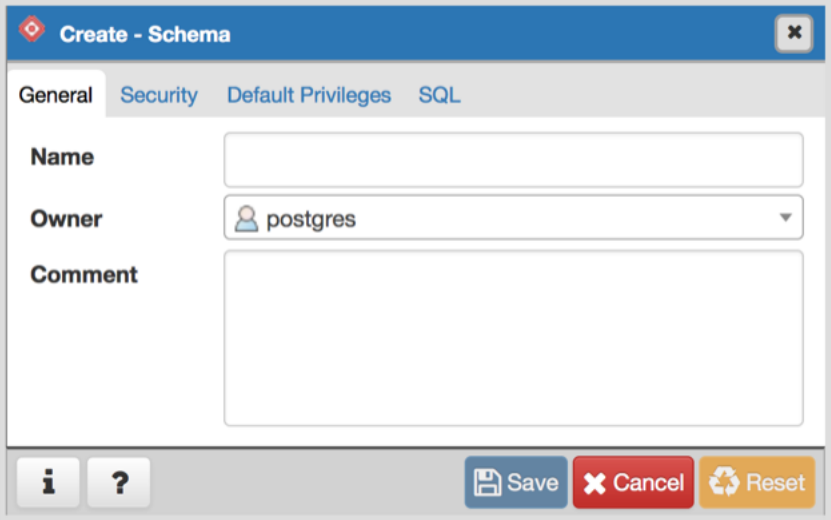
The example demonstrates creating a procedure that returns a list of employees from a table named *emp*. The procedure is a SECURITY DEFINER, and will execute with the privileges of the role that defined the procedure.

- Click the *Info* button (i) to access online help. View context-sensitive help in the *Tabbed browser*, where a new tab displays the PostgreSQL core documentation.
- Click the *Save* button to save work.
- Click the *Cancel* button to exit without saving work.
- Click the *Reset* button to restore configuration parameters.

3.19 The Schema Dialog

Use the *Schema* dialog to define a schema. A schema is the organizational workhorse of a database, similar to directories or namespaces. To create a schema, you must be a database superuser or have the CREATE privilege.

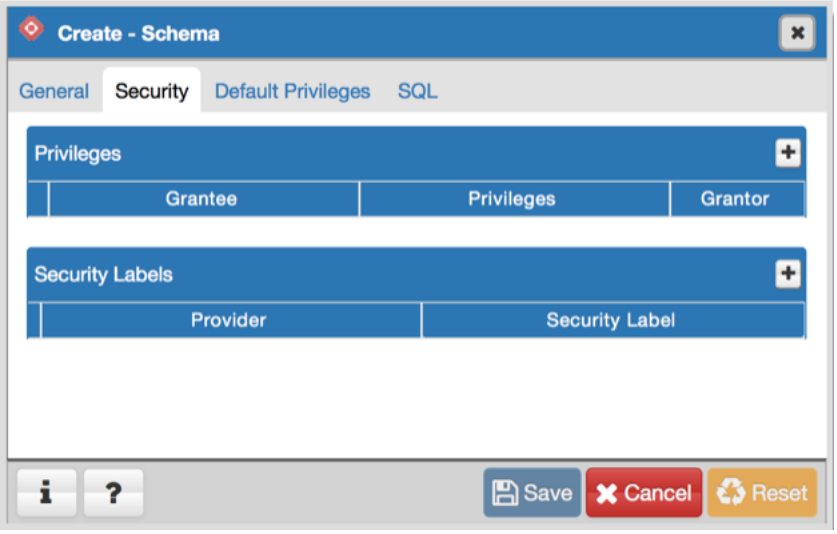
The *Schema* dialog organizes the development of schema through the following dialog tabs: *General* and *Security*. The *SQL* tab displays the SQL code generated by dialog selections.



Use the fields on the *General* tab to identify the schema.

- Use the *Name* field to add a descriptive name for the schema. The name will be displayed in the *pgAdmin* tree control.
- Select the owner of the schema from the drop-down listbox in the *Owner* field.
- Store notes about the schema in the *Comment* field.

Click the *Security* tab to continue.



Use the *Security* tab to assign privileges and security labels for the schema.

Click the *Add* icon (+) to assign a set of privileges in the *Privileges* panel:

- Select the name of the role from the drop-down listbox in the *Grantee* field.
- Click inside the *Privileges* field. Check the boxes to the left of one or more privileges to grant the selected privileges to the specified user.
- Select the name of the role that is granting the privilege from the drop-down listbox in the *Grantor* field. The default grantor is the owner of the schema.

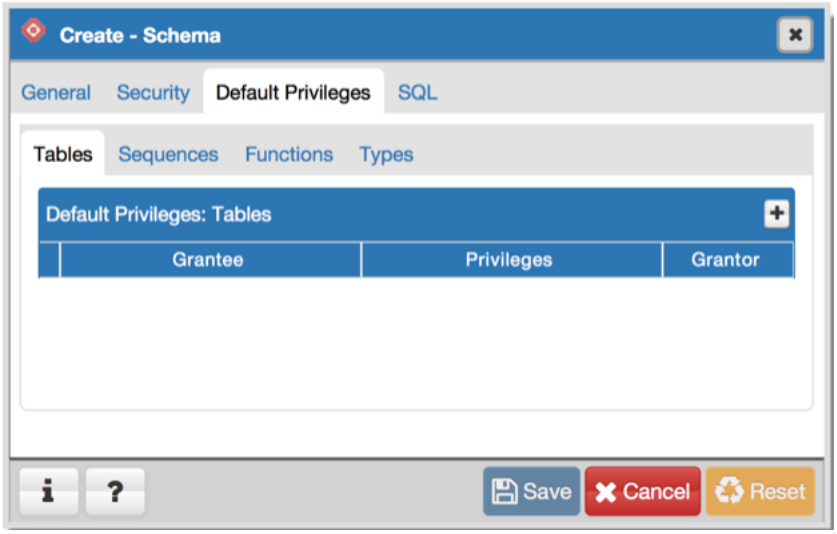
Click the *Add* icon (+) to assign additional sets of privileges; to discard a privilege, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Click the *Add* icon (+) to assign a security label in the *Security Labels* panel:

- Specify a security label provider in the *Provider* field. The named provider must be loaded and must consent to the proposed labeling operation.
- Specify a a security label in the *Security Label* field. The meaning of a given label is at the discretion of the label provider. PostgreSQL places no restrictions on whether or how a label provider must interpret security labels; it merely provides a mechanism for storing them.

Click the *Add* icon (+) to assign additional security labels; to discard a security label, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Click the *Default Privileges* tab to continue.



Use the *Default Privileges* tab to grant privileges for tables, sequences, functions and types. Use the tabs nested inside the *Default Privileges* tab to specify the database object and click the *Add* icon (+) to assign a set of privileges:

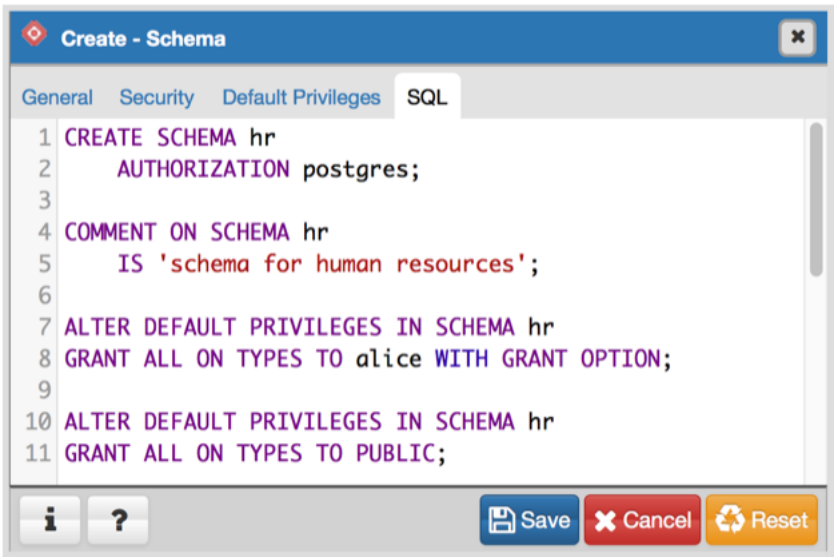
- Select the name of a role that will be granted privileges in the schema from the drop-down listbox in the *Grantee* field.
- Click inside the *Privileges* field. Check the boxes to the left of one or more privileges to grant the selected privileges to the specified user.
- Select the name of the role that is granting the privilege from the drop-down listbox in the *Grantor* field. The default grantor is the owner of the schema.

Click the *SQL* tab to continue.

Your entries in the *Schema* dialog generate a SQL command (see an example below). Use the *SQL* tab for review; revisit or switch tabs to make any changes to the SQL command.

Example

The following is an example of the sql command generated by selections made in the *Schema* dialog:



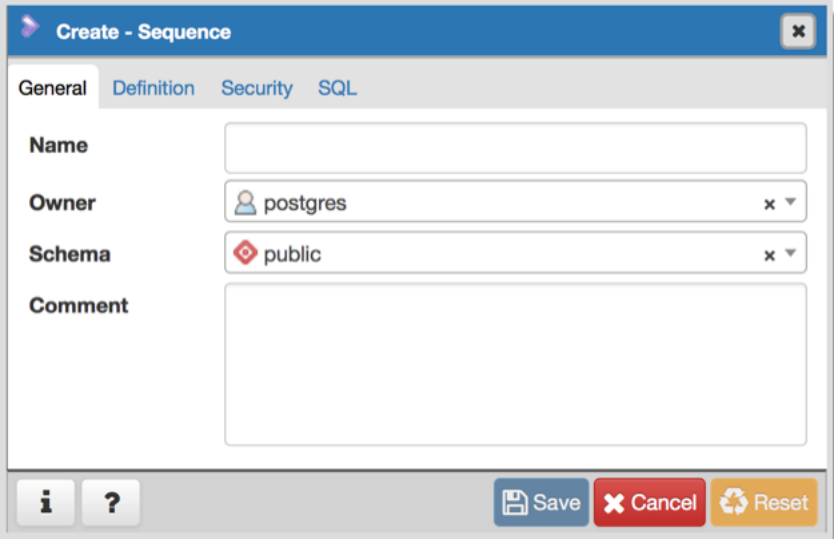
The example creates a schema named hr; the command grants *USAGE* privileges to *public* and assigns the ability to grant privileges to *alice*.

- Click the *Info* button (i) to access online help. View context-sensitive help in the *Tabbed browser*, where a new tab displays the PostgreSQL core documentation.
- Click the *Save* button to save work.
- Click the *Cancel* button to exit without saving work.
- Click the *Reset* button to restore configuration parameters.

3.20 The Sequence Dialog

Use the *Sequence* dialog to create a sequence. A sequence generates unique values in a sequential order (not necessarily contiguous).

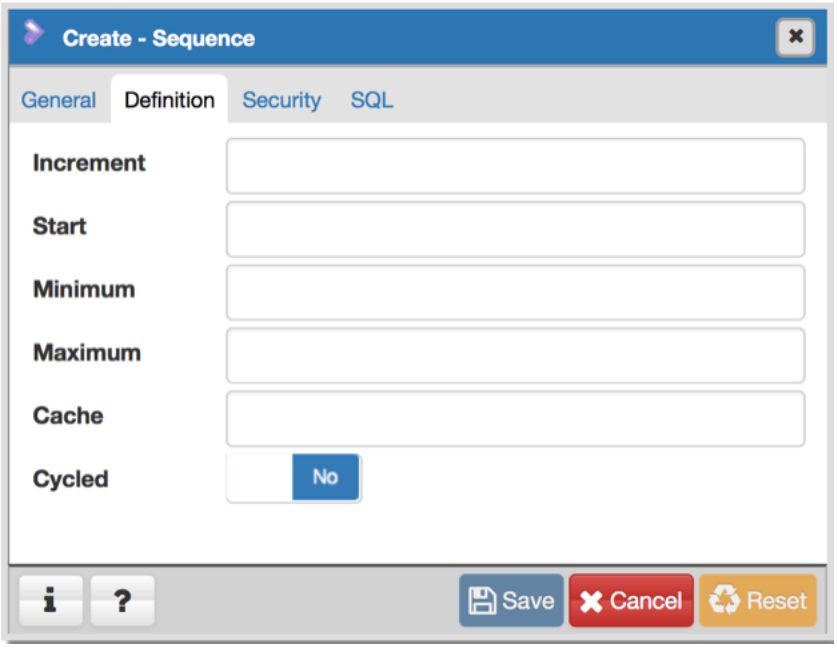
The *Sequence* dialog organizes the development of a sequence through the following dialog tabs: *General*, *Definition*, and *Security*. The *SQL* tab displays the SQL code generated by dialog selections.



Use the fields in the *General* tab to identify a sequence:

- Use the *Name* field to add a descriptive name for the sequence. The name will be displayed in the *pgAdmin* tree control. The sequence name must be distinct from the name of any other sequence, table, index, view, or foreign table in the same schema.
- Use the drop-down listbox next to *Owner* to select the name of the role that will own the sequence.
- Use the drop-down listbox next to *Schema* to select the schema in which the sequence will reside.
- Store notes about the sequence in the *Comment* field.

Click the *Definition* tab to continue.



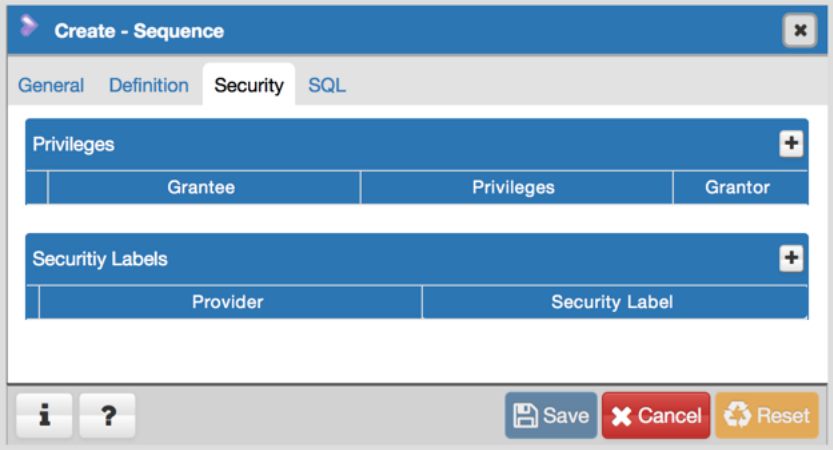
Use the fields in the *Definition* tab to define the sequence:

- Use the *Increment* field to specify which value is added to the current sequence value to create a new value.
- Provide a value in the *Start* field to specify the beginning value of the sequence. The default starting value is

MINVALUE for ascending sequences and MAXVALUE for descending ones.

- Provide a value in the *Minimum* field to specify the minimum value a sequence can generate. If this clause is not supplied or NO MINVALUE is specified, then defaults will be used. The defaults are 1 and -263-1 for ascending and descending sequences, respectively.
- Provide a value in the *Maximum* field to specify the maximum value for the sequence. If this clause is not supplied or NO MAXVALUE is specified, then default values will be used. The defaults are 263-1 and -1 for ascending and descending sequences, respectively.
- Provide a value in the *Cache* field to specify how many sequence numbers are to be preallocated and stored in memory for faster access. The minimum value is 1 (only one value can be generated at a time, i.e., no cache), and this is also the default.
- Move the *Cycled* switch to the *Yes* position to allow the sequence to wrap around when the MAXVALUE or the MINVALUE has been reached by an ascending or descending sequence respectively. If the limit is reached, the next number generated will be the MINVALUE or MAXVALUE, respectively. The default is *No*.

Click the *Security* tab to continue.



Use the *Security* tab to assign privileges and define security labels for the sequence.

Use the *Privileges* panel to assign privileges. Click the *Add* icon (+) to set privileges:

- Select the name of a role that will be granted privileges from the drop-down listbox in the *Grantee* field.
- Click inside the *Privileges* field. Check the boxes to the left of one or more privileges to grant the selected privilege to the specified user.
- Select the name of the role from the drop-down listbox in the *Grantor* field. The default grantor is the owner of the database.

Click the *Add* icon (+) to assign additional privileges; to discard a privilege, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Use the *Security Labels* panel to define security labels applied to the sequence. Click the *Add* icon (+) to add each security label selection:

- Specify a security label provider in the *Provider* field. The named provider must be loaded and must consent to the proposed labeling operation.
- Specify a a security label in the *Security Label* field. The meaning of a given label is at the discretion of the label provider. PostgreSQL places no restrictions on whether or how a label provider must interpret security labels; it merely provides a mechanism for storing them.

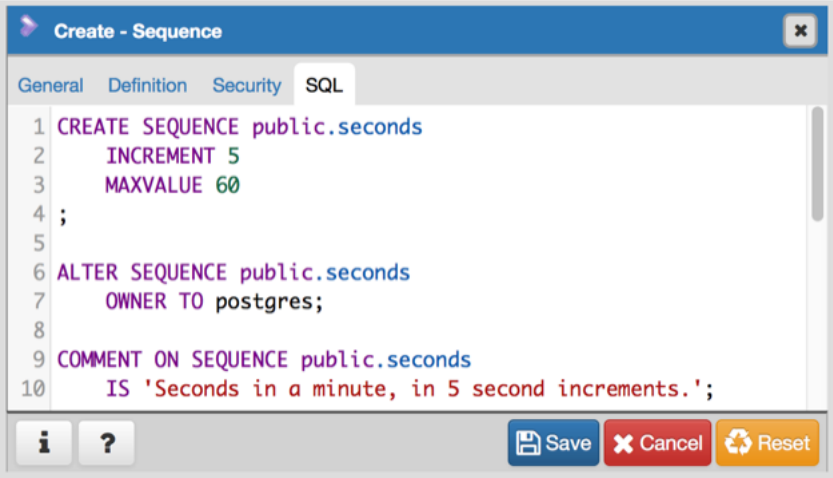
Click the *Add* icon (+) to assign additional security labels; to discard a security label, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Click the *SQL* tab to continue.

Your entries in the *Sequence* dialog generate a generate a SQL command (see an example below). Use the *SQL* tab for review; revisit or switch tabs to make any changes to the SQL command.

Example

The following is an example of the sql command generated by user selections in the *Sequence* dialog:



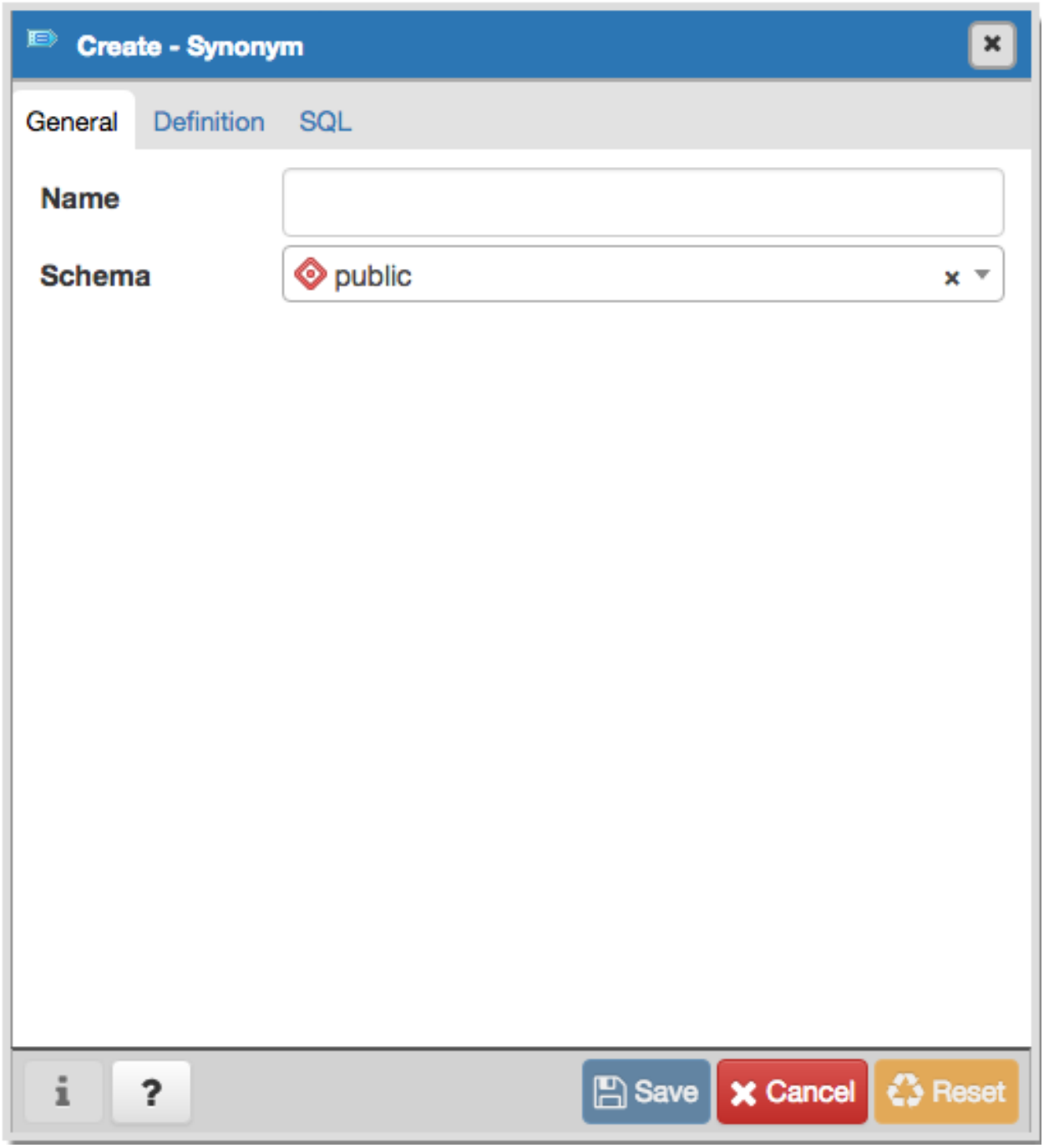
The example shown demonstrates a sequence named *seconds*. The sequence will increase in *5* second increments, and stop when it reaches a maximum value equal of *60*.

- Click the *Info* button (i) to access online help. View context-sensitive help in the *Tabbed browser*, where a new tab displays the PostgreSQL core documentation.
- Click the *Save* button to save work.
- Click the *Cancel* button to exit without saving work.
- Click the *Reset* button to restore configuration parameters.

3.21 The Synonym Dialog

Use the *Synonym* dialog to substitute the name of a target object with a user-defined synonym.

The *Synonym* dialog organizes the development of a synonym through the *General* tab. The *SQL* tab displays the SQL code generated by dialog selections.



Use the fields in the *General* tab to identify the synonym:

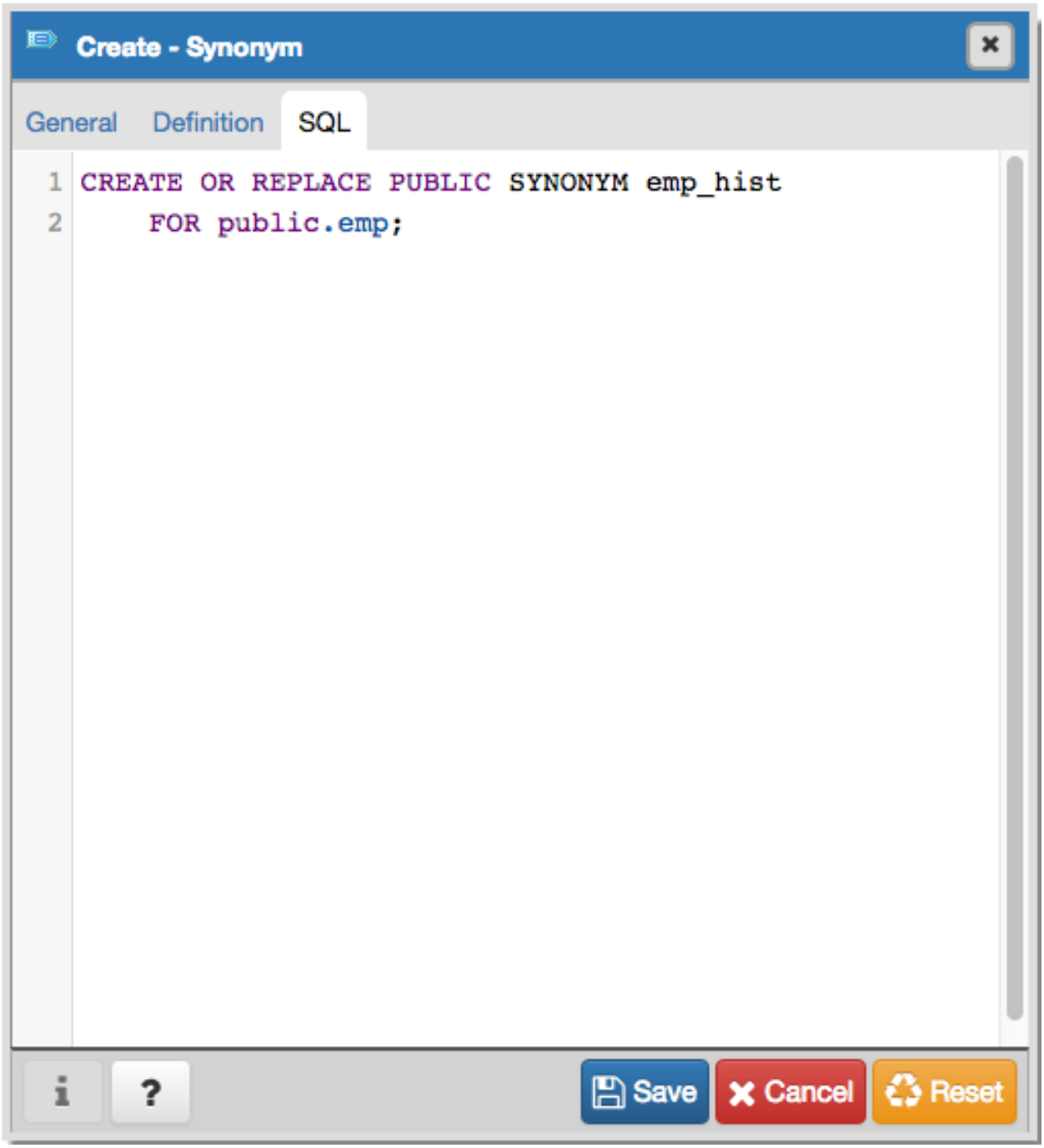
- Use the *Name* field to specify the name of synonym. The name will be displayed in the *pgAdmin* tree control.
- Select the name of the schema in which the synonym will reside from the drop-down listbox in the *Schema* field.

In the definition panel, identify the target:

- Use the drop-down listbox next to *Target Type* to select the the type of object referenced by the synonym.
- Use the drop-down listbox next to *Target Schema* to select the name of the schema in which the object resides.
- Use the drop-down listbox next to *Target Object* to select the name of the object referenced by the synonym.

Click the *SQL* tab to continue.

Your selections and entries in the *Synonym* dialog generate a SQL command.



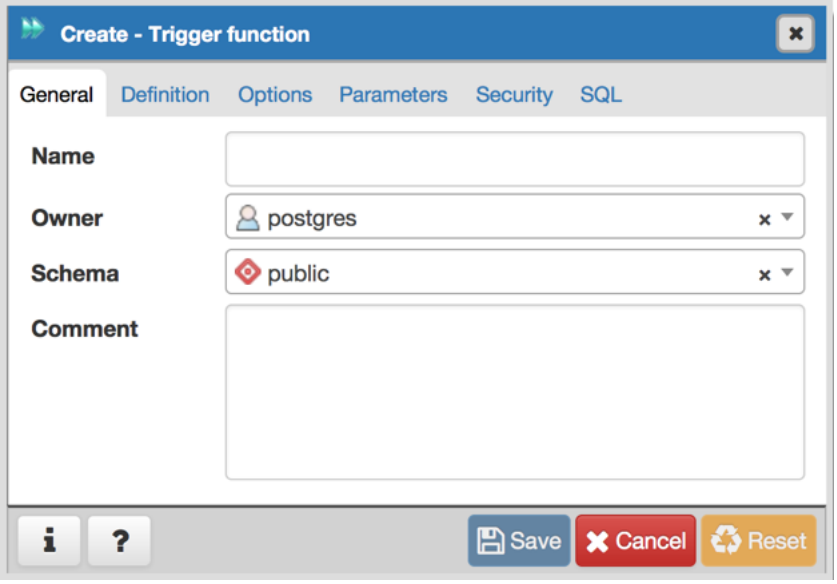
The example creates a synonym for the *emp* table named *emp_hist*.

- Click the *Save* button to save work.
- Click the *Cancel* button to exit without saving work.
- Click the *Reset* button to restore configuration parameters.

3.22 The Trigger function Dialog

Use the *Trigger function* dialog to create or manage a trigger_function. A trigger function defines the action that will be invoked when a trigger fires.

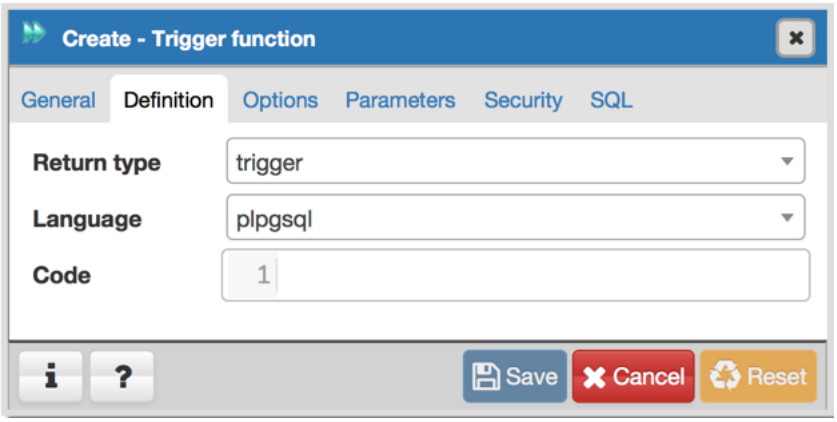
The *Trigger function* dialog organizes the development of a trigger function through the following dialog tabs: *General*, *Definition*, *Options*, *Parameters* and *Security*. The *SQL* tab displays the SQL code generated by dialog selections.



Use the fields in the *General* tab to identify the trigger function:

- Use the *Name* field to add a descriptive name for the trigger function. The name will be displayed in the *pgAdmin* tree control. Please note that trigger functions will be invoked in alphabetical order.
- Use the drop-down listbox next to *Owner* to select the role that will own the trigger function.
- Select the name of the schema in which the trigger function will reside from the drop-down listbox in the *Schema* field.
- Store notes about the trigger function in the *Comment* field.

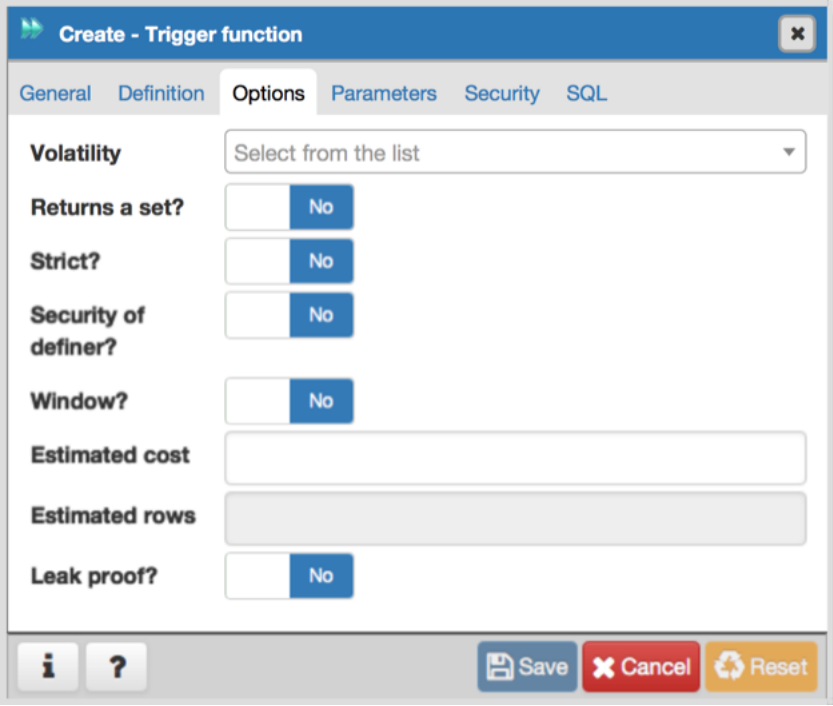
Click the *Definition* tab to continue.



Use the fields in the *Definition* tab to define the trigger function:

- Use the drop-down listbox next to *Return type* to specify the pseudotype that is associated with the trigger function:
  - Select *trigger* if you are creating a DML trigger.
  - Select *event_trigger* if you are creating a DDL trigger.
- Use the drop-down listbox next to *Language* to select the implementation language. The default is *plpgsql*.
- Use the *Code* field to write the code that will execute when the trigger function is called.

Click the *Options* tab to continue.

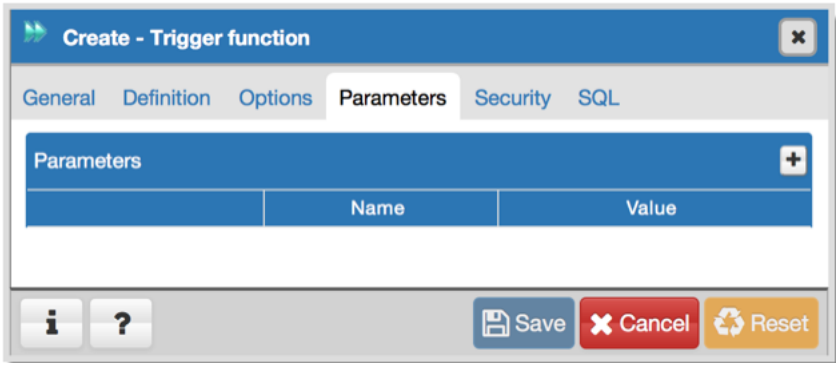


Use the fields in the *Options* tab to describe or modify the action of the trigger function:

- Use the drop-down listbox next to *Volatility* to select one of the following:
  - *VOLATILE* indicates that the trigger function value can change even within a single table scan.
  - *STABLE* indicates that the trigger function cannot modify the database, and that within a single table scan it will consistently return the same result for the same argument values.
  - *IMMUTABLE* indicates that the trigger function cannot modify the database and always returns the same result when given the same argument values.
- Move the *Returns a Set?* switch to indicate if the trigger function returns a set that includes multiple rows. The default is *No*.
- Move the *Strict?* switch to indicate if the trigger function always returns NULL whenever any of its arguments are NULL. If *Yes*, the function is not executed when there are NULL arguments; instead a NULL result is assumed automatically. The default is *No*.
- Move the *Security of definer?* switch to specify that the trigger function is to be executed with the privileges of the user that created it. The default is *No*.
- Move the *Window?* switch to indicate that the trigger function is a window function rather than a plain function. The default is *No*. This is currently only useful for trigger functions written in C.

- Use the *Estimated cost* field to specify a positive number representing the estimated execution cost for the trigger function, in units of cpu_operator_cost. If the function returns a set, this is the cost per returned row.
- Use the *Estimated rows* field to specify a positive number giving the estimated number of rows that the query planner should expect the trigger function to return. This is only allowed when the function is declared to return a set. The default assumption is 1000 rows.
- Move the *Leak proof?* switch to indicate whether the trigger function has side effects. The default is *No*. This option can only be set by the superuser.

Click the *Parameters* tab to continue.

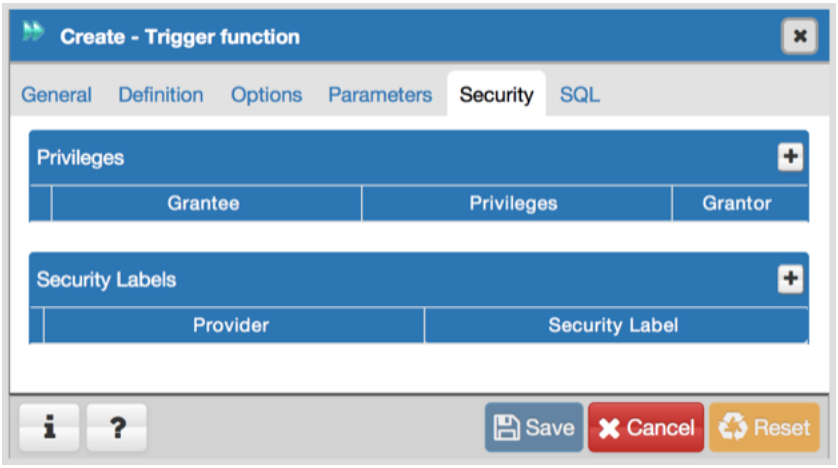


Use the fields in the *Parameters* tab to specify settings that will be applied when the trigger function is invoked. Click the *Add* icon (+) to add a *Name*/*Value* pair to the table below.

- Use the drop-down listbox in the *Name* field to select a parameter.
- Use the *Value* field to specify the value that will be associated with the selected parameter. This field is context-sensitive.

Click the *Add* icon (+) to set additional parameters; to discard a parameter, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Click the *Security* tab to continue.



Use the *Security* tab to assign privileges and define security labels.

Use the *Privileges* panel to assign usage privileges for the trigger function to a role. Click the *Add* icon (+) to to add a role to the table.

- Select the name of the role from the drop-down listbox in the *Grantee* field.

- Click inside the *Privileges* field. Check the boxes to the left of one or more privileges to grant the selected privilege to the specified user.
- Select the name of a role from the drop-down listbox in the *Grantor* field. The default grantor is the owner of the database.

Click the *Add* icon (+) to assign additional privileges; to discard a privilege, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Use the *Security Labels* panel to define security labels applied to the trigger function. Click the *Add* icon (+) to add each security label selection:

- Specify a security label provider in the *Provider* field. The named provider must be loaded and must consent to the proposed labeling operation.
- Specify a a security label in the *Security Label* field. The meaning of a given label is at the discretion of the label provider. PostgreSQL places no restrictions on whether or how a label provider must interpret security labels; it merely provides a mechanism for storing them.

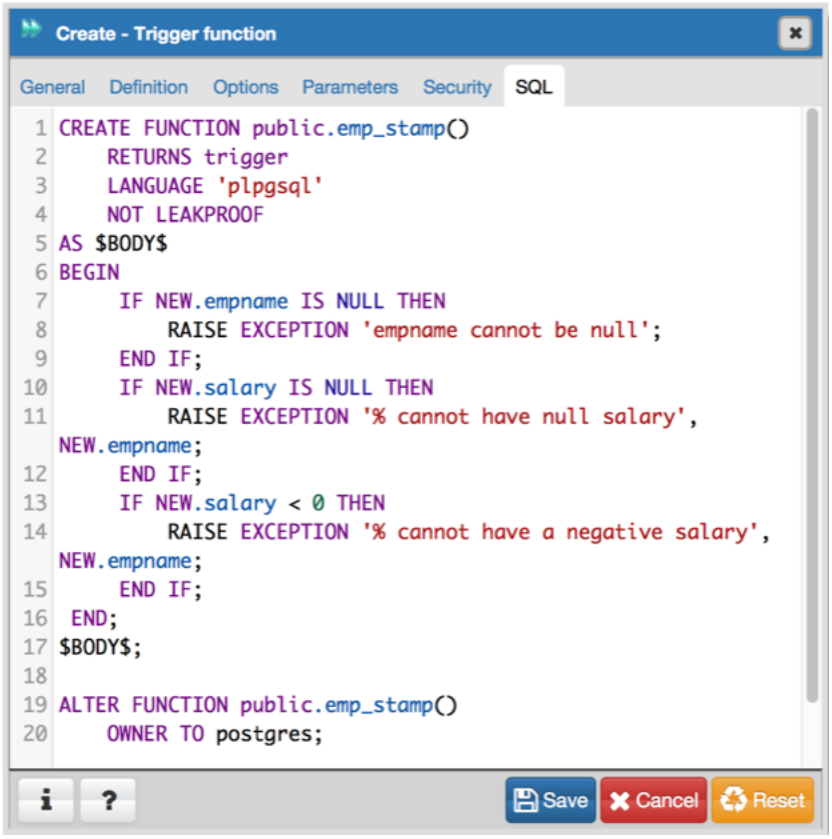
Click the *Add* icon (+) to assign additional security labels; to discard a security label, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Click the *SQL* tab to continue.

Your entries in the *Trigger function* dialog generate a SQL command (see an example below). Use the *SQL* tab for review; revisit other tabs to modify the SQL command.

Example

The following is an example of the sql command generated by user selections in the *Trigger function* dialog:



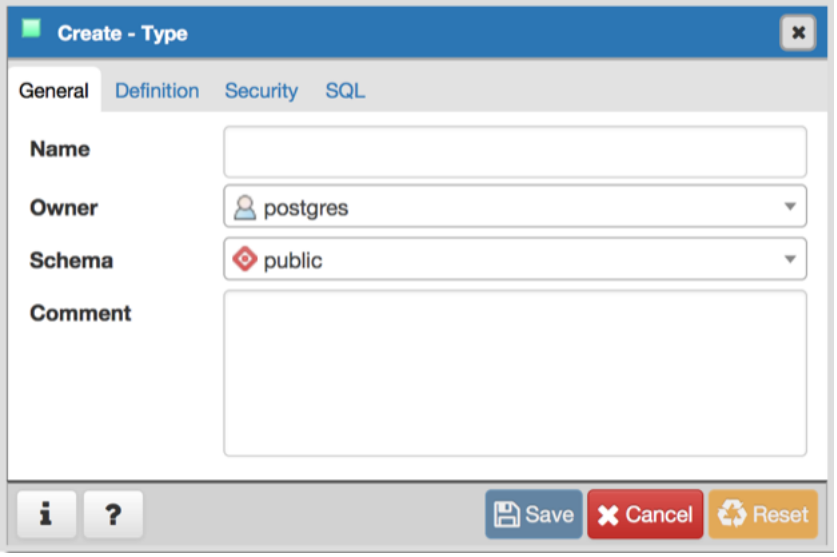
The example shown demonstrates creating a trigger function named *emp_stamp* that checks for a new employee's name, and checks that the employee's salary is a positive value.

- Click the *Info* button (i) to access online help. View context-sensitive help in the *Tabbed browser*, where a new tab displays the PostgreSQL core documentation.
- Click the *Save* button to save work.
- Click the *Cancel* button to exit without saving work.
- Click the *Reset* button to restore configuration parameters.

3.23 The Type Dialog

Use the *Type* dialog to register a custom data type.

The *Type* dialog organizes the development of a data type through the following dialog tabs: *General*, *Definition*, and *Security*. The *SQL* tab displays the SQL code generated by dialog selections.



Use the fields in the *General* tab to identify the custom data type:

- Use the *Name* field to add a descriptive name for the type. The name will be displayed in the *pgAdmin* tree control. The type name must be distinct from the name of any existing type, domain, or table in the same schema.
- Use the drop-down listbox next to *Owner* to select the role that will own the type.
- Select the name of the schema in which the type will reside from the drop-down listbox in the *Schema* field.
- Store notes about the type in the *Comments* field.

Click the *Definition* tab to continue.

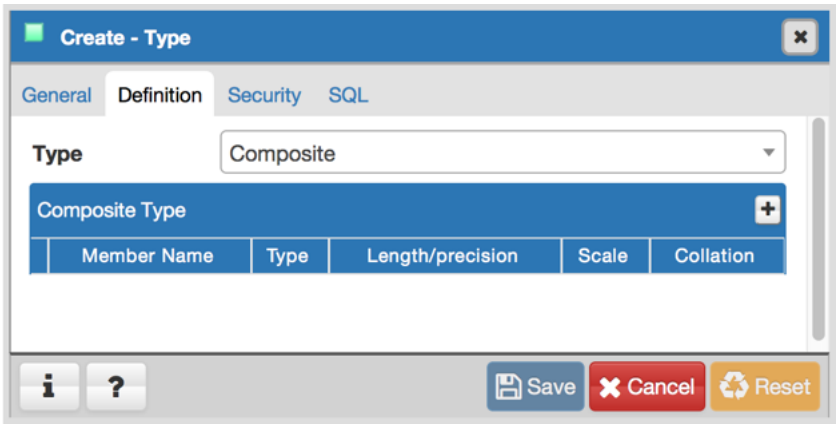
Select a data type from the drop-down listbox next to *Type* on the *Definition* tab; the panel below changes to display the options appropriate for the selected data type. Use the fields in the panel to define the data type.

There are five data types:

- *Composite Type*
- *Enumeration Type*

- *Range Type*
- *External Type* (or *Base Type*)
- *Shell Type*

If you select *Composite* in the *Type* field, the *Definition* tab displays the *Composite Type* panel:

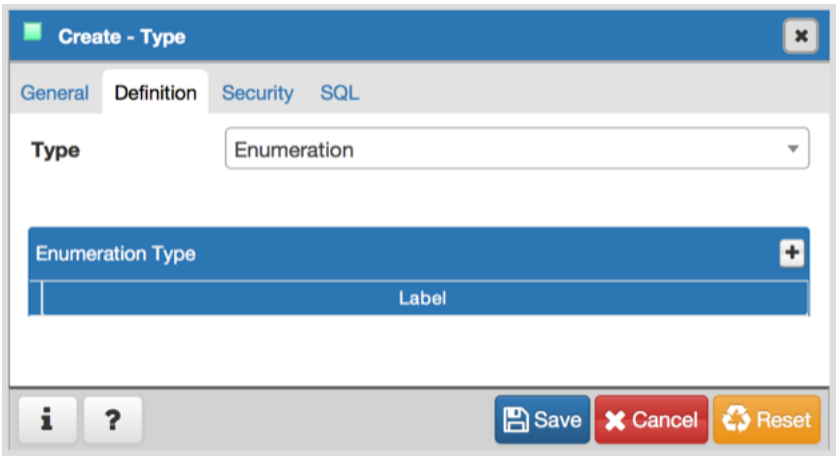


Click the *Add* icon (+) to provide attributes of the type. Fields on the *General* panel are context sensitive and may be disabled.

- Use the *Member Name* field to add an attribute name.
- Use the drop-down listbox in the *Type* field to select a datatype.
- Use the *Length/Precision* field to specify the maximum length of a non-numeric type, or the total count of significant digits in a numeric type.
- Use the *Scale* field to specify the number of digits to the right of the decimal point.
- Use the drop-down listbox in the *Collation* field to select a collation (if applicable).

Click the *Add* icon (+) to define an additional member; click the trash icon to the left of the row to discard a row.

If you select the *Enumeration* in the *Type* field, the *Definition* tab displays the *Enumeration Type* panel:

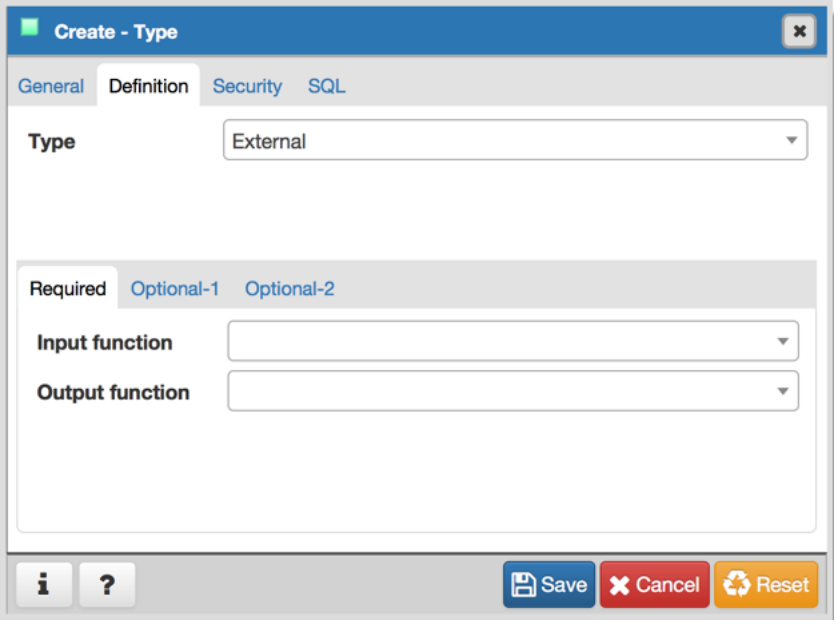


Click the *Add* icon (+) to provide a label for the type.

- Use the *Label* field to add a label, which must be less than 64 bytes long.

Click the *Add* icon (+) after each selection to create additional labels; to discard a label, click the trash icon to the left of the row.

If you select *External*, the *Definition* tab displays the *External Type* panel:



On the *Required* tab:

- Use the drop-down listbox next to the *Input function* field to add an input_function. The input_function converts the type's external textual representation to the internal representation used by the operators and functions defined for the type.
- Use the drop-down listbox next to the *Output function* field to add an output_function. The output_function converts the type's internal representation used by the operators and functions defined for the type to the type's external textual representation.

On the *Optional-1* tab:

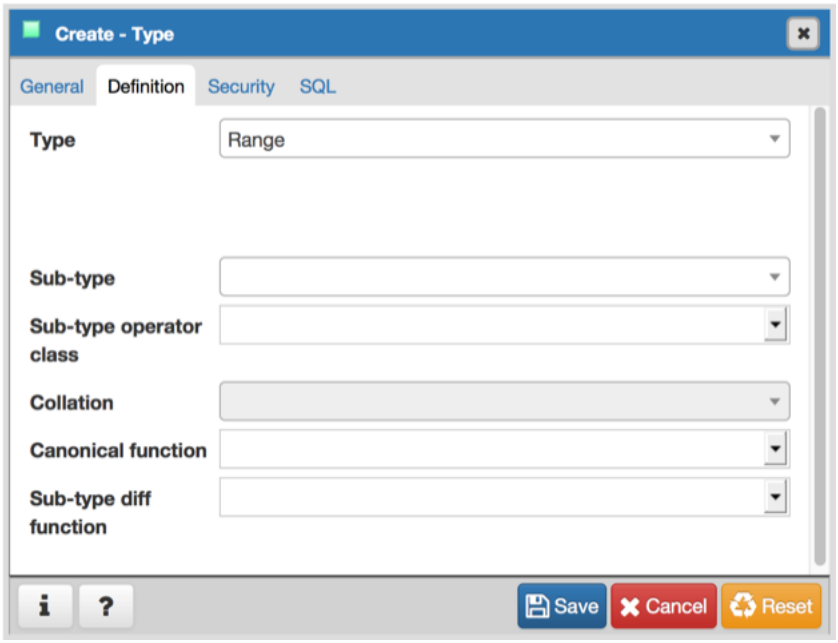
- Use the drop-down listbox next to the optional *Receive Function* field to select a receive_function. The optional receive_function converts the type's external binary representation to the internal representation. If this function is not supplied, the type cannot participate in binary input.
- Use the drop-down listbox next to the optional *Send function* field to select a send_function. The optional send_function converts from the internal representation to the external binary representation. If this function is not supplied, the type cannot participate in binary output.
- Use the drop-down listbox next to the optional *Typmod in function* field tab to select a type_modifier_input_function.
- Use the drop-down listbox next to the optional *Typmod out function* field tab to select a type_modifier_output_function. It is allowed to omit the type_modifier_output_function, in which case the default display format is the stored typmod integer value enclosed in parentheses.
- Use the optional *Internal length* to specify a value for internal representation.
- Move the *Variable?* switch to specify the internal representation is of variable length (VARIABLE). The default is a fixed length positive integer.
- Specify a default value in the optional *Default* field in cases where a column of the data type defaults to something other than the null value. Specify the default with the DEFAULT key word. (A default can be overridden by an explicit DEFAULT clause attached to a particular column.)
- Use the drop-down listbox next to the optional *Analyze function* field to select a function for performing type-specific statistics collection for columns of the data type.

- Use the drop-down listbox next to the optional *Category type* field to help control which implicit cast will be applied in ambiguous situations.
- Move the *Preferred?* switch to *Yes* to specify the selected category type is preferred. The default is *No*.

On the *Optional-2* tab:

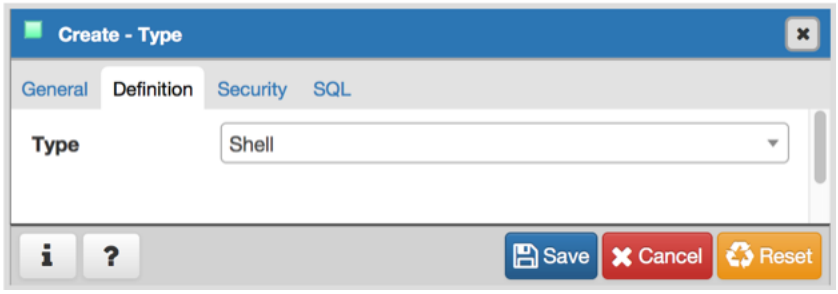
- Use the drop-down listbox next to the optional *Element type* field to specify a data type.
- Use the optional *Delimiter* field to indicate the delimiter to be used between values in the external representation of arrays for this data type. The default delimiter is the comma (,). Note that the delimiter is associated with the array element type, not the array type itself.
- Use the drop-down listbox next to *Alignment type* to specify the storage alignment required for the data type. The allowed values (char, int2, int4, and double) correspond with alignment on 1, 2, 4, or 8 byte boundaries.
- Use the drop-down listbox next to optional *Storage type* to select a strategy for storing data.
- Move the *Passed by value?* switch to *Yes* to override the existing data type value. The default is *No*.
- Move the *Collatable?* switch to *Yes* to specify column definitions and expressions of the type may carry collation information through use of the COLLATE clause. The default is *No*.

If you select *Range* in the *Type* field, the *Definition* tab displays the *Range* panel. Fields on the *Range* panel are context-sensitive and may be disabled.



- Use the drop-down listbox next to *Sub-type* to select an associated b-tree operator class (to determine the ordering of values for the range type).
- Use the drop-down listbox next to *Sub-type operator class* to use a non-default operator class.
- Use the drop-down listbox next to *Collation* to use a non-default collation in the range's ordering if the sub-type is collatable.
- Use the drop-down listbox next to *Canonical function* to convert range values to a canonical form.
- Use the drop-down listbox next to *Sub-type diff function* to select a user-defined subtype_diff function.

If you select *Shell* in the *Type* field, the *Definition* tab displays the *Shell* panel:



A shell type is a placeholder for a type and has no parameters.

Click the *Security* tab to continue.

Use the *Security* tab to assign privileges and define security labels.

Use the *Privileges* panel to assign privileges for the type; click the *Add* icon (+) to grant privileges:

- Select the name of the role that will be granted privileges on the type from the drop-down listbox in the *Grantee* field.
- Click inside the *Privileges* field. Check the boxes to the left of one or more privileges to grant the selected privilege to the specified user.
- Select the name of the role that is granting privileges from the drop-down listbox in the *Grantor* field. The default grantor is the owner of the database.

Click the *Add* icon (+) to assign additional privileges; to discard a privilege, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Use the *Security Labels* panel to define security labels applied to the type. Click the *Add* icon (+) to add each security label selection:

- Specify a security label provider in the *Provider* field. The named provider must be loaded and must consent to the proposed labeling operation.
- Specify a security label in the *Security Label* field. The meaning of a given label is at the discretion of the label provider. PostgreSQL places no restrictions on whether or how a label provider must interpret security labels; it merely provides a mechanism for storing them.

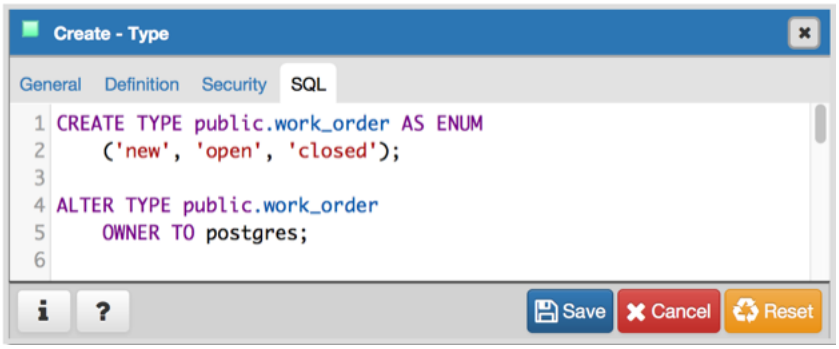
Click the *Add* icon (+) to assign additional security labels; to discard a security label, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Click the *SQL* tab to continue.

Your entries in the *Type* dialog generate a SQL command (see an example below). Use the *SQL* tab for review; revisit or switch tabs to make any changes to the SQL command.

Example

The following is an example of a sql command generated by user selections made in the *Type* dialog:



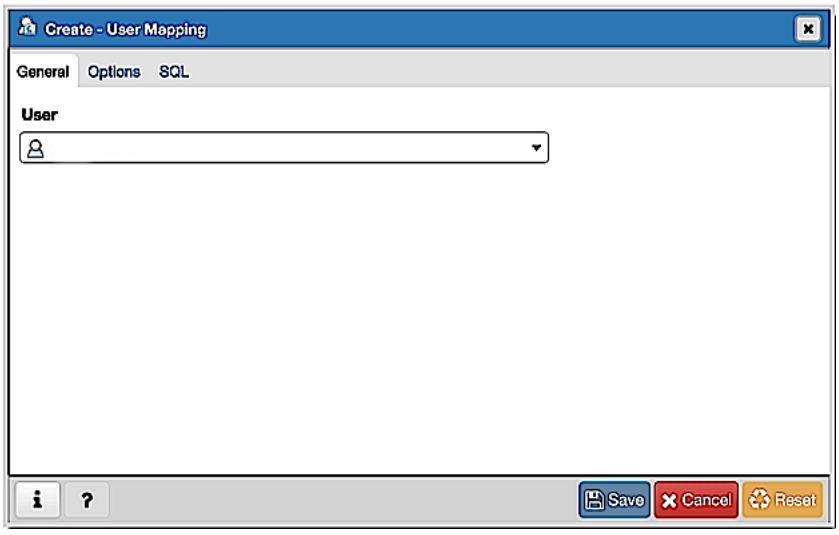
The example shown demonstrates creating a data type named *work_order*. The data type is an enumerated type with three labels: new, open and closed.

- Click the *Info* button (i) to access online help. View context-sensitive help in the *Tabbed browser*, where a new tab displays the PostgreSQL core documentation.
- Click the *Save* button to save work.
- Click the *Cancel* button to exit without saving work.
- Click the *Reset* button to restore configuration parameters.

3.24 The User Mapping Dialog

Use the *User Mapping* dialog to define a new mapping of a user to a foreign server.

The *User Mapping* dialog organizes the development of a user mapping through the following dialog tabs: *General* and *Options*. The *SQL* tab displays the SQL code generated by dialog selections.

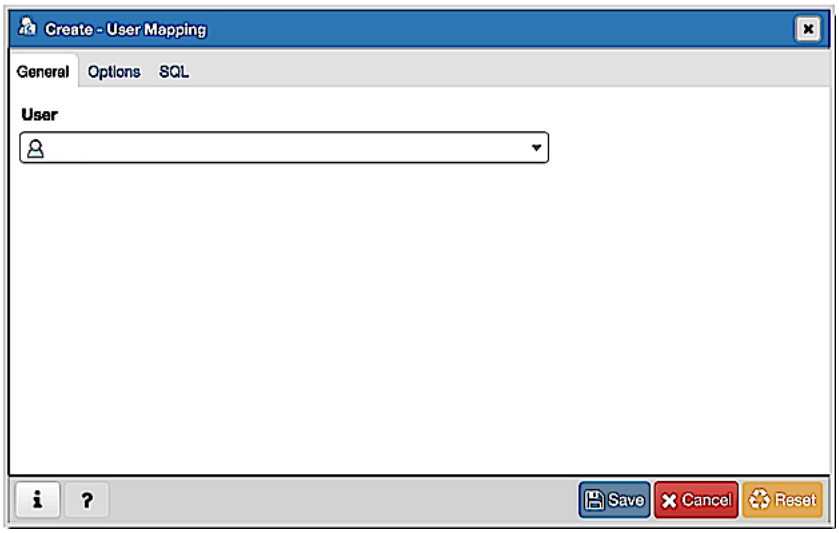


Use the drop-down listbox in the *User* field in the *General* tab to identify the connecting role:

- Select *CURRENT_USER* to use the name of the current role.

- Select *PUBLIC* if no other user-specific mapping is applicable.
- Select a pre-defined role name to specify the name of an existing user.

Click the *Options* tab to continue.



Use the fields in the *Options* tab to specify connection options; the accepted option names and values are specific to the foreign data wrapper associated with the server specified in the user mapping. Click the *Add* button to add an option/value pair.

- Specify the option name in the *Option* field.
- Provide a corresponding value in the *Value* field.

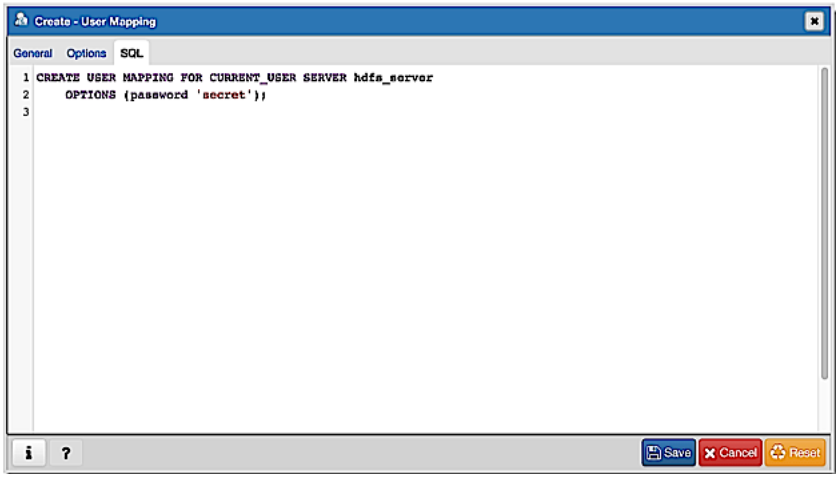
Click *Add* to specify each additional option/value pair; to discard an option, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Click the *SQL* tab to continue.

Your entries in the *User Mapping* dialog generate a SQL command (see an example below). Use the *SQL* tab for review; revisit or switch tabs to make any changes to the SQL command.

Example

The following is an example of the sql command generated by user selections in the *User Mapping* dialog:



The example shown demonstrates a user mapping for the *hdfs_server*. The user is *CURRENT_USER* with a password *secret*.

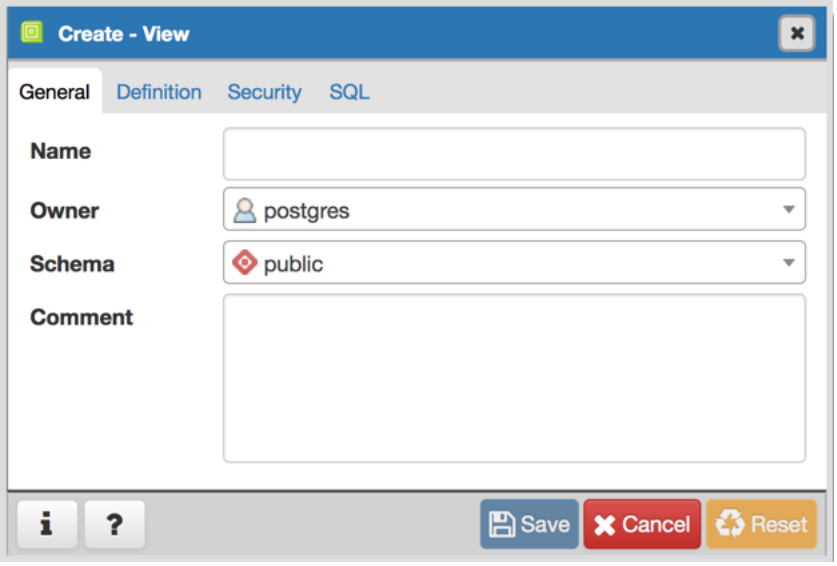
- Click the *Info* button (i) to access online help. View context-sensitive help in the *Tabbed browser*, where a new tab displays the PostgreSQL core documentation.
- Click the *Save* button to save work.
- Click the *Cancel* button to exit without saving work.
- Click the *Reset* button to restore configuration parameters.

3.25 The View Dialog

Use the *View* dialog to define a view. The view is not physically materialized; the query is executed each time the view is referenced in a query.

The *View* dialog organizes the development of a View through the following dialog tabs: *General*, *Definition*, and *Security*". The *SQL* tab displays the SQL code generated by dialog selections.

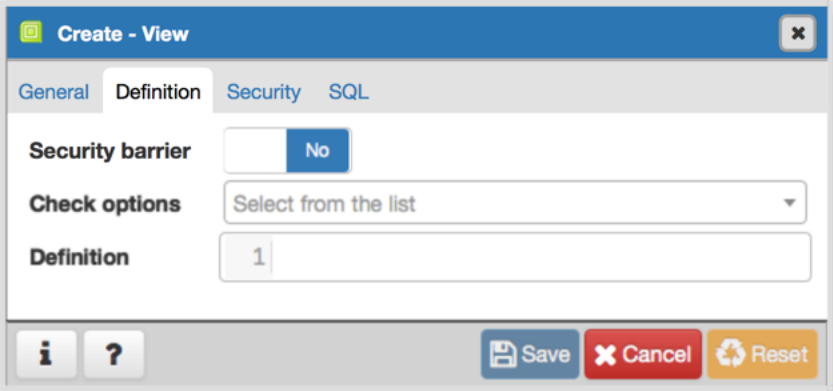
Click the *General* tab to begin.



Use the fields in the *General* tab to identify a view:

- Use the *Name* field to add a descriptive name for the view. The name of the view must be distinct from the name of any other view, table, sequence, index or foreign table in the same schema. The name will be displayed in the *pgAdmin* tree control.
- Use the drop-down listbox next to *Owner* to select the role that will own the view.
- If applicable, select the name of the schema in which the view will reside from the drop-down listbox in the *Schema* field.
- Store notes about the view in the *Comments* field.

Click the *Definition* tab to continue.



Use the fields in the *Definition* tab to define properties of the view:

- Set the *Security Barrier* switch to *Yes* to indicate that the view is to act as a security barrier. For more information about defining and using a security barrier rule, see Section 38.5 of the PostgreSQL documentation.
- **Use the drop-down listbox next to *Check options* to select from *No*, *Local* or *Cascaded*.** The *Local* option specifies that new rows are only checked against the conditions defined in the view. Any conditions defined on underlying base views are not checked (unless you specify the CHECK OPTION). The *Cascaded* option specifies new rows are checked against the conditions of the view and all underlying base views.
- Use the workspace in the *Definition* field to write a query to create a view.

Click the *Security* tab to continue.

Use the *Security* tab to assign privileges and define security labels.

Use the *Privileges* panel to assign privileges to a role. Click the *Add* icon (+) to set privileges for the view:

- Select the name of the role that will be granted privileges from the drop-down listbox in the *Grantee* field.
- Click inside the *Privileges* field. Check the boxes to the left of one or more privileges to grant the selected privilege to the specified user.
- Select the name of a role with sufficient privileges to grant privileges on the view from the drop-down listbox in the *Grantor* field. The default grantor is the owner of the database.

Click the *Add* icon (+) to assign additional privileges; to discard a privilege, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Use the *Security Labels* panel to define security labels applied to the view. Click the *Add* icon (+) to add each security label selection:

- Specify a security label provider in the *Provider* field. The named provider must be loaded and must consent to the proposed labeling operation.
- Specify a a security label in the *Security Label* field. The meaning of a given label is at the discretion of the label provider. PostgreSQL places no restrictions on whether or how a label provider must interpret security labels; it merely provides a mechanism for storing them.

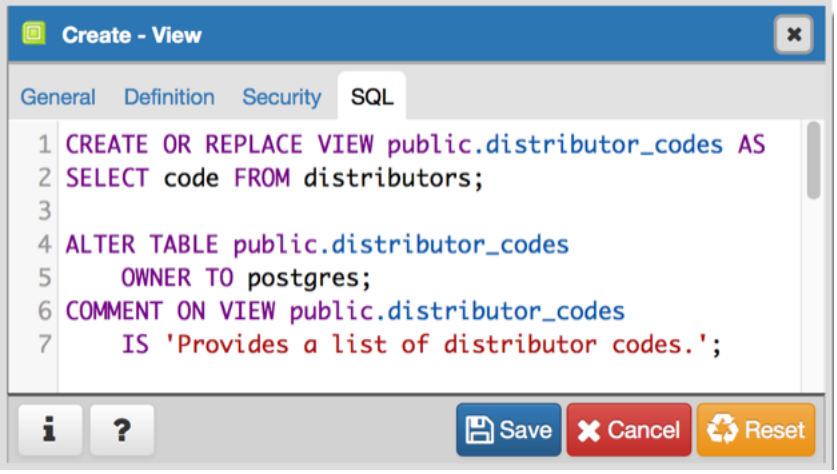
Click the *Add* icon (+) to assign additional security labels; to discard a security label, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Click the *SQL* tab to continue.

Your entries in the *View* dialog generate a SQL command (see an example below). Use the *SQL* tab for review; revisit or switch tabs to make any changes to the SQL command.

Example

The following is an example of the sql command generated by user selections in the *View* dialog:



The example shown demonstrates creating a view named *distributor_codes* that includes the content of the *code* column from the *distributors* table.

- Click the *Info* button (i) to access online help. View context-sensitive help in the *Tabbed browser*, where a new tab displays the PostgreSQL core documentation.
- Click the *Save* button to save work.
- Click the *Cancel* button to exit without saving work.
- Click the *Reset* button to restore configuration parameters.

CHAPTER

FOUR

CREATING OR MODIFYING A TABLE

pgAdmin 4 provides dialogs that allow you to modify all table properties and attributes.

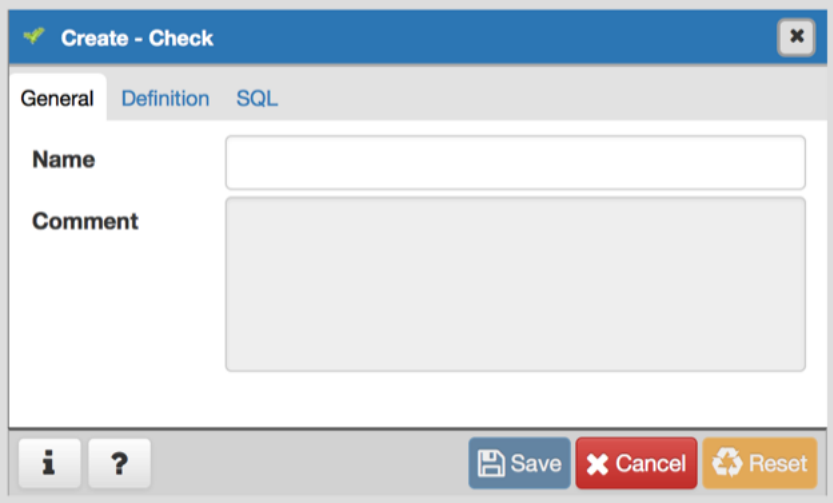
To access a dialog that allows you to create a database object, right-click on the object type in the pgAdmin tree control, and select the *Create* option for that object. For example, to create a new database, right-click on the *Casts* node, and select *Create Cast...*

Contents:

4.1 The Check Dialog

Use the *Check* dialog to define or modify a check constraint. A check constraint specifies an expression that produces a Boolean result that new or updated rows must satisfy for an insert or update operation to succeed.

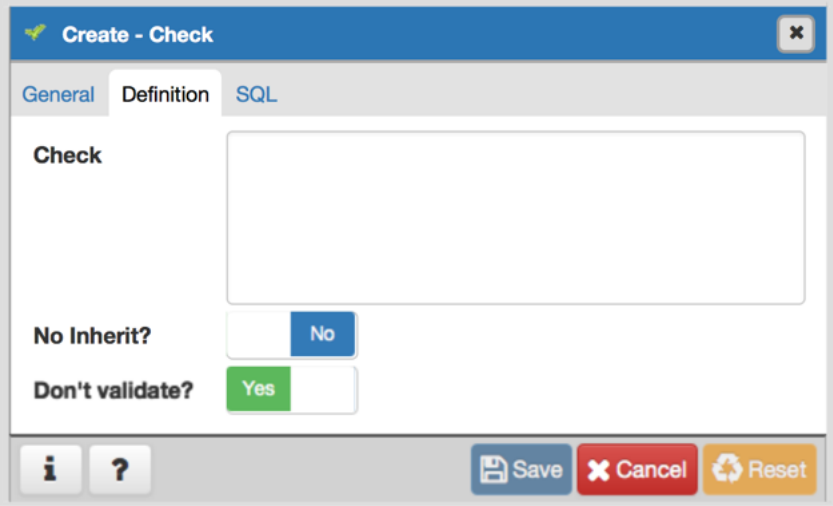
The *Check* dialog organizes the development of a check constraint through the *General* and *Definition* tabs. The *SQL* tab displays the SQL code generated by dialog selections.



Use the fields in the *General* tab to identify the check constraint:

- Use the *Name* field to provide a descriptive name for the check constraint that will be displayed in the *pgAdmin* tree control. With PostgreSQL 9.5 forward, when a table has multiple check constraints, they will be tested for each row in alphabetical order by name and after NOT NULL constraints.
- Store notes about the check constraint in the *Comment* field.

Click the *Definition* tab to continue.



Use the fields in the *Definition* tab to define the check constraint:

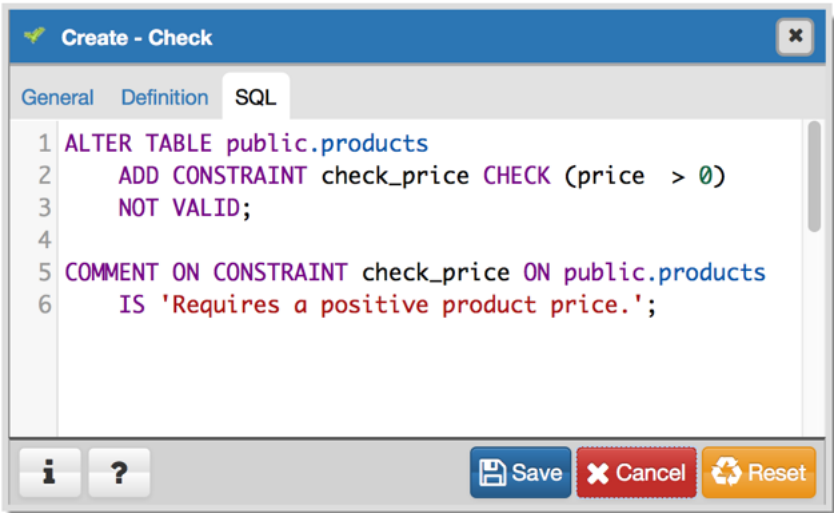
- Provide the expression that a row must satisfy in the *Check* field.
- Move the *No Inherit?* switch to the *Yes* position to specify this constraint is automatically inherited by a table's children. The default is *No*.
- Move the *Don't validate?* switch to the *No* position to skip validation of existing data; the constraint may not hold for all rows in the table. The default is *Yes*.

Click the *SQL* tab to continue.

Your entries in the *Check* dialog generate a SQL command (see an example below). Use the *SQL* tab for review; revisit or switch tabs to make any changes to the SQL command.

Example

The following is an example of the sql command generated by user selections in the *Check* dialog:



The example shown demonstrates creating a check constraint named *check_price* on the *price* column of the *products* table. The constraint confirms that any values added to the column are greater than 0.

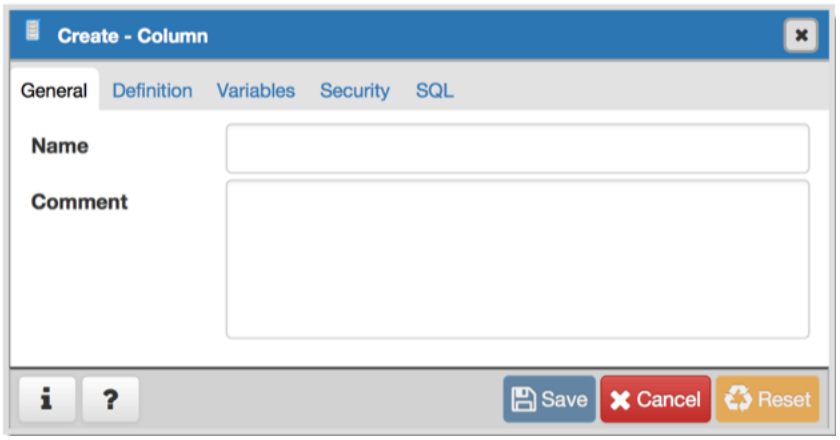
- Click the *Info* button (i) to access online help. View context-sensitive help in the *Tabbed browser*, where a new tab displays the PostgreSQL core documentation.
- Click the *Save* button to save work.

- Click the *Cancel* button to exit without saving work.
- Click the *Reset* button to restore configuration parameters.

4.2 The Column Dialog

Use the *Column* dialog to add a column to an existing table or modify a column definition.

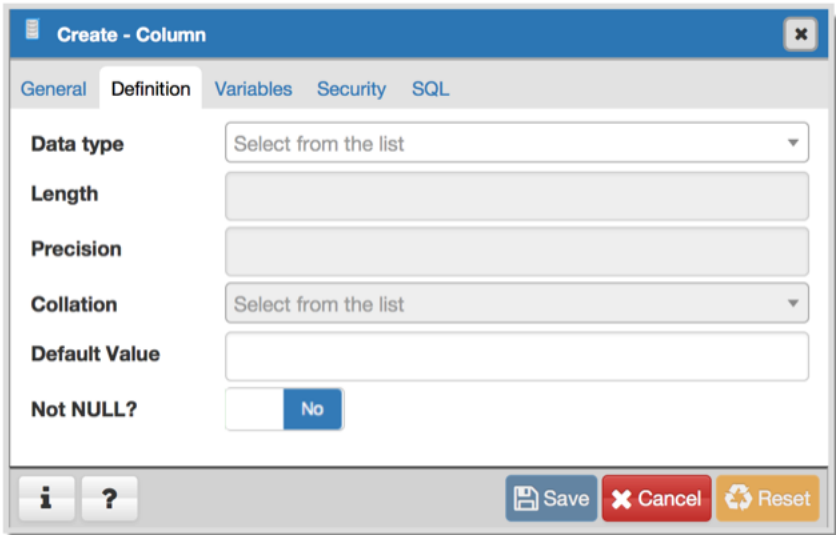
The *Column* dialog organizes the development of a column through the following dialog tabs: *General*, *Definition*, and *Security*. The *SQL* tab displays the SQL code generated by dialog selections.



Use the fields in the *General* tab to identify the column:

- Use the *Name* field to add a descriptive name for the column. The name will be displayed in the *pgAdmin* tree control. This field is required.
- Store notes about the column in the *Comment* field.

Click the *Definition* tab to continue.



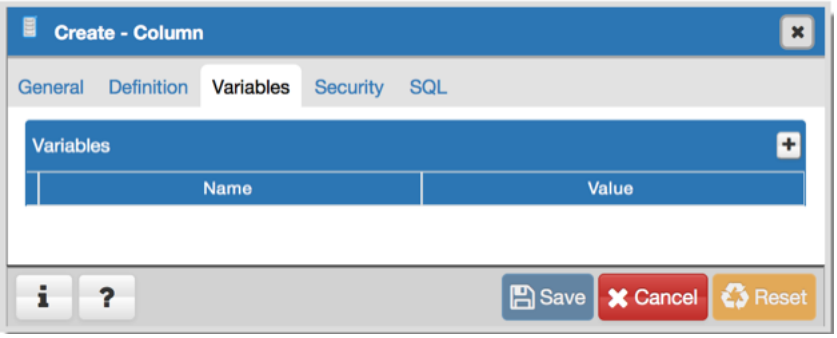
Use the fields in the *Definition* tab to add parameters for the column. (Fields are disabled if inapplicable.)

- Use the drop-down listbox next to *Data Type* to select a data type for the column. For more information on the data types that are supported by PostgreSQL, refer to Chapter 8 of the Postgres core documentation. This field

is required.

- Use the *Length* and *Precision* fields to specify the maximum number of significant digits in a numeric value, or the maximum number of characters in a text value.
- Use the drop-down listbox next to *Collation* to apply a collation setting to the column.
- Use the *Default Value* field to specify a default data value.
- Move the *Not Null* switch to the *Yes* position to specify the column may not contain null values. The default is *No*.

Click the *Variables* tab to continue.

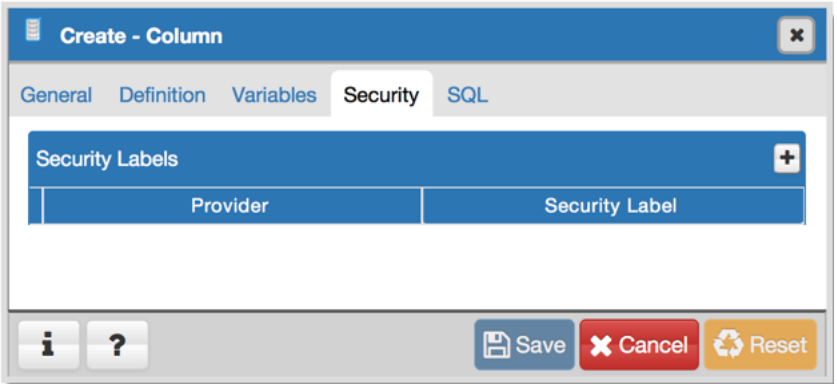


Use the *Variables* tab to to specify the number of distinct values that may be present in the column; this value overrides estimates made by the ANALYZE command. Click the *Add* icon (+) to add a *Name*/*Value* pair:

- Select the name of the variable from the drop-down listbox in the *Name* field.
  - Select *n_distinct* to specify the number of distinct values for the column.
  - Select *n_distinct_inherited* to specify the number of distinct values for the table and its children.
- Specify the number of distinct values in the *Value* field. For more information, see the documentation for ALTER TABLE.

Click the *Add* icon (+) to specify each additional *Name*/*Value* pair; to discard a variable, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Click the *Security* tab to continue.



Use the *Security* tab to assign attributes and define security labels. Click the *Add* icon (+) to add each security label selection:

- Specify a security label provider in the *Provider* field. The named provider must be loaded and must consent to the proposed labeling operation.

- Specify a a security label in the *Security Label* field. The meaning of a given label is at the discretion of the label provider. PostgreSQL places no restrictions on whether or how a label provider must interpret security labels; it merely provides a mechanism for storing them.

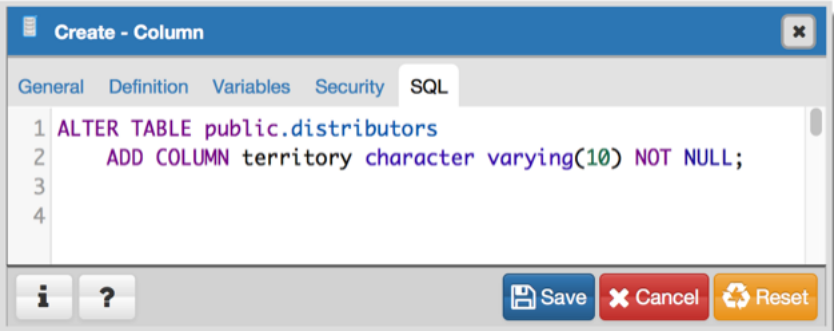
Click the *Add* icon (+) to assign additional security labels; to discard a security label, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Click the *SQL* tab to continue.

Your entries in the *Column* dialog generate a SQL command (see an example below). Use the *SQL* tab for review; revisit or switch tabs to make any changes to the SQL command.

Example

The following is an example of the sql command generated by user selections in the *Column* dialog:



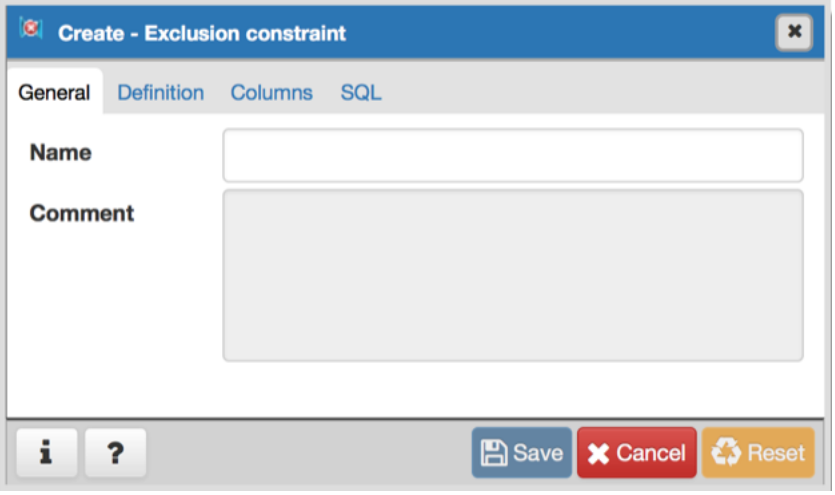
The example shown demonstrates creating a column named *territory* in the table named *distributors*.

- Click the *Info* button (i) to access online help. View context-sensitive help in the *Tabbed browser*, where a new tab displays the PostgreSQL core documentation.
- Click the *Save* button to save work.
- Click the *Cancel* button to exit without saving work.
- Click the *Reset* button to restore configuration parameters.

4.3 The Exclusion constraint Dialog

Use the *Exclusion constraint* dialog to define or modify the behavior of an exclusion constraint. An exclusion constraint guarantees that if any two rows are compared on the specified column or expression (using the specified operator), at least one of the operator comparisons will return false or null.

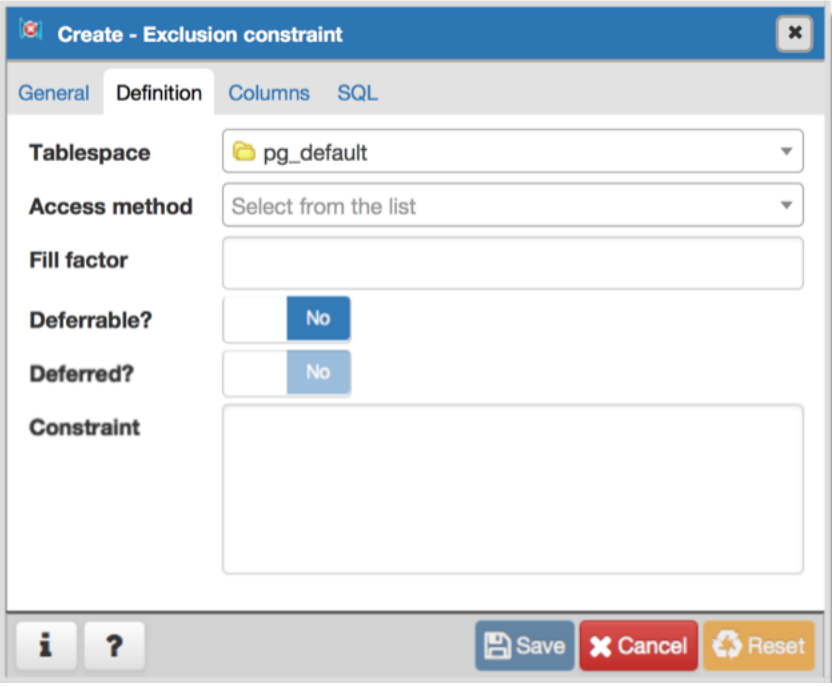
The *Exclusion constraint* dialog organizes the development of an exclusion constraint through the following dialog tabs: *General*, *Definition*, and *Columns*. The *SQL* tab displays the SQL code generated by dialog selections.



Use the fields in the *General* tab to identify the exclusion constraint:

- Use the *Name* field to provide a descriptive name for the exclusion constraint. The name will be displayed in the *pgAdmin* tree control.

Click the *Definition* tab to continue.



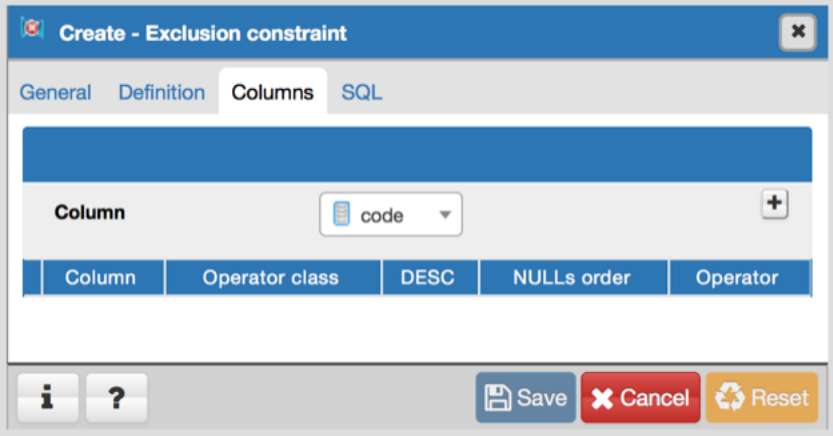
Use the fields in the *Definition* tab to define the exclusion constraint:

- Use the drop-down listbox next to *Tablespace* to select the tablespace in which the index associated with the exclude constraint will reside.
- Use the drop-down listbox next to *Access method* to specify the type of index that will be used when implementing the exclusion constraint:
  - Select *gist* to specify a GiST index.
  - Select *spgist* to specify a space-partitioned GiST index.
  - Select *btree* to specify a B-tree index.

  - Select *hash* to specify a hash index.

- Use the *Fill Factor* field to specify a fill factor for the table and associated index. The fill factor is a percentage between 10 and 100. 100 (complete packing) is the default.
- Move the *Deferrable?* switch to the *Yes* position to specify that the timing of the constraint is deferrable, and can be postponed until the end of the statement. The default is *No*.
- If enabled, move the *Deferred?* switch to the *Yes* position to specify the timing of the constraint is deferred to the end of the statement. The default is *No*.
- Use the *Constraint* field to provide a condition that a row must satisfy to be included in the table.

Click the *Columns* tab to continue.



Use the fields in the *Columns* tab to to specify the column(s) to which the constraint applies. Use the drop-down listbox next to *Column* to select a column and click the *Add* icon (+) to provide details of the action on the column:

- The *Column* field is populated with the selection made in the *Column* drop-down listbox.
- If applicable, use the drop-down listbox in the *Operator class* to specify the operator class that will be used by the index for the column.
- Move the *DESC* switch to *DESC* to specify a descending sort order. The default is *ASC* which specifies an ascending sort order.
- Use the *NULLs order* column to specify the placement of NULL values (when sorted). Specify *FIRST* or *LAST*.
- Use the drop-down list next to *Operator* to specify a comparison or conditional operator.

Click the *SQL* tab to continue.

Your entries in the *Exclusion Constraint* dialog generate a SQL command (see an example below). Use the *SQL* tab for review; revisit or switch tabs to make any changes to the SQL command.

Example

The following is an example of the sql command generated by user selections in the *Exclusion Constraint* dialog:

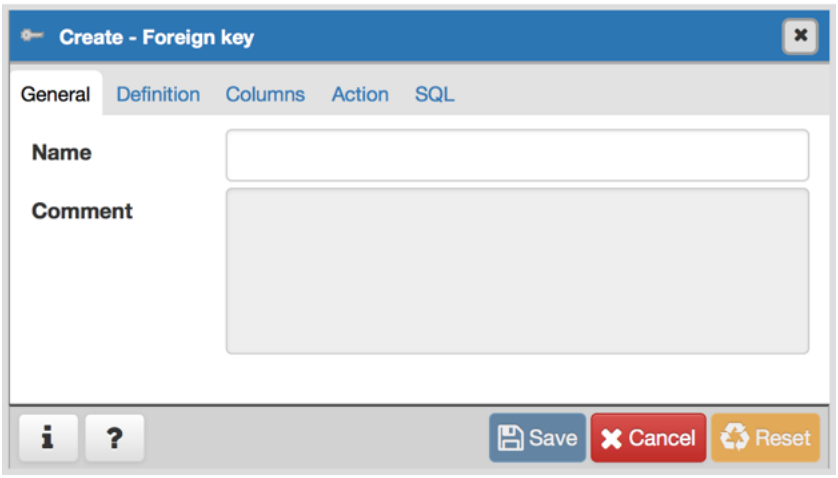
The example shown demonstrates creating an exclusion constraint named *exclude_department* that restricts additions to the dept table to those additions that are not equal to the value of the *deptno* column. The constraint uses a btree index.

- Click the *Info* button (i) to access online help. View context-sensitive help in the *Tabbed browser*, where a new tab displays the PostgreSQL core documentation.
- Click the *Save* button to save work.
- Click the *Cancel* button to exit without saving work.
- Click the *Reset* button to restore configuration parameters.

4.4 The Foreign key Dialog

Use the *Foreign key* dialog to specify the behavior of a foreign key constraint. A foreign key constraint maintains referential integrity between two tables. A foreign key constraint cannot be defined between a temporary table and a permanent table.

The *Foreign key* dialog organizes the development of a foreign key constraint through the following dialog tabs: *General*, *Definition*, *Columns*, and *Action*. The *SQL* tab displays the SQL code generated by dialog selections.

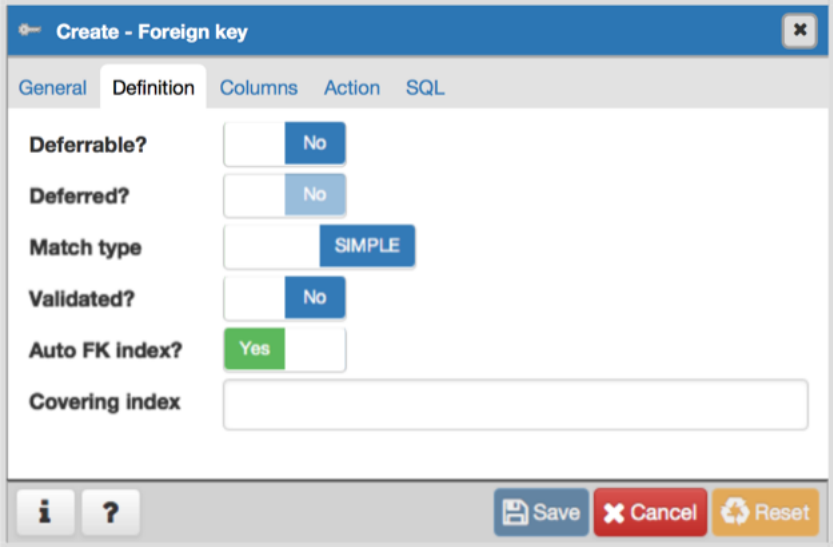


Use the fields in the *General* tab to identify the foreign key constraint:

- Use the *Name* field to add a descriptive name for the foreign key. The name will be displayed in the *pgAdmin* tree control.

- Store notes about the foreign key constraint in the *Comment* field.

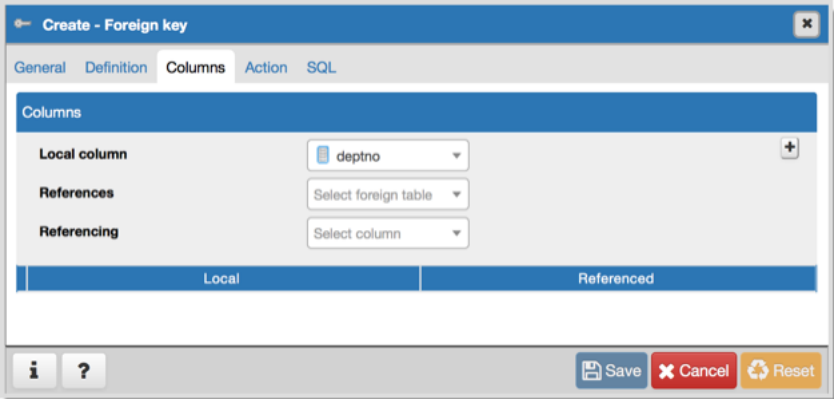
Click the *Definition* tab to continue.



Use the fields in the *Definition* tab to define the foreign key constraint:

- Move the *Deferrable?* switch to the *Yes* position to specify the timing of the constraint is deferrable and can be postponed until the end of the statement. The default is *No*.
- If enabled, move the *Deferred?* switch to the *Yes* position to specify the timing of the constraint is deferred to the end of the statement. The default is *No*.
- Move the *Match type* switch specify the type of matching that is enforced by the constraint:
  - Select *Full* to indicate that all columns of a multicolumn foreign key must be null if any column is null; if all columns are null, the row is not required to have a match in the referenced table.
  - Select *Simple* to specify that a single foreign key column may be null; if any column is null, the row is not required to have a match in the referenced table.
- Move the *Validated* switch to the *Yes* position to instruct the server to validate the existing table content (against a foreign key or check constraint) when you save modifications to this dialog.
- Move the *Auto FK Index* switch to the *No* position to disable the automatic index feature.
- The field next to *Covering Index* generates the name of an index if the *Auto FK Index* switch is in the *Yes* position; or, this field is disabled.

Click the *Columns* tab to continue.

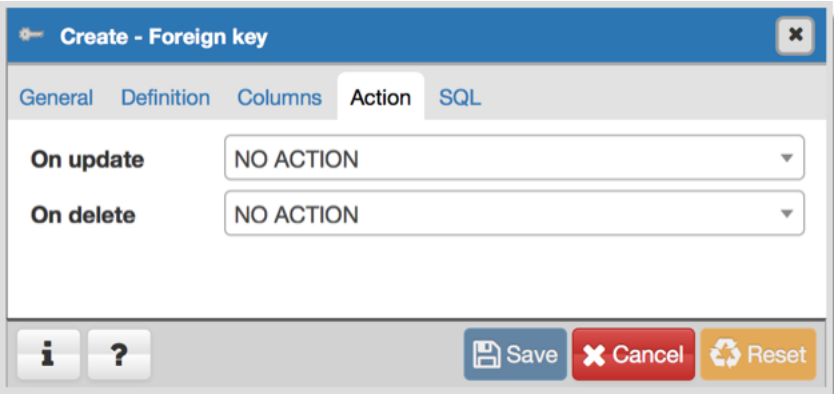


Use the fields in the *Columns* tab to specify one or more reference column(s). A Foreign Key constraint requires that one or more columns of a table must only contain values that match values in the referenced column(s) of a row of a referenced table:

- Use the drop-down listbox next to *Local column* to specify the column in the current table that will be compared to the foreign table.
- Use the drop-down listbox next to *References* to specify the name of the table in which the comparison column(s) resides.
- Use the drop-down listbox next to *Referencing* to specify a column in the foreign table.

Click the *Add* icon (+) to add a column to the list; repeat the steps above and click the *Add* icon (+) to add additional columns. To discard an entry, click the trash icon to the left of the entry and confirm deletion in the *Delete Row* popup.

Click the *Action* tab to continue.



Use the drop-down listboxes on the *Action* tab to specify behavior related to the foreign key constraint that will be performed when data within the table is updated or deleted:

- Use the drop-down listbox next to *On update* to select an action that will be performed when data in the table is updated.
- Use the drop-down listbox next to *On delete* to select an action that will be performed when data in the table is deleted.

The supported actions are:

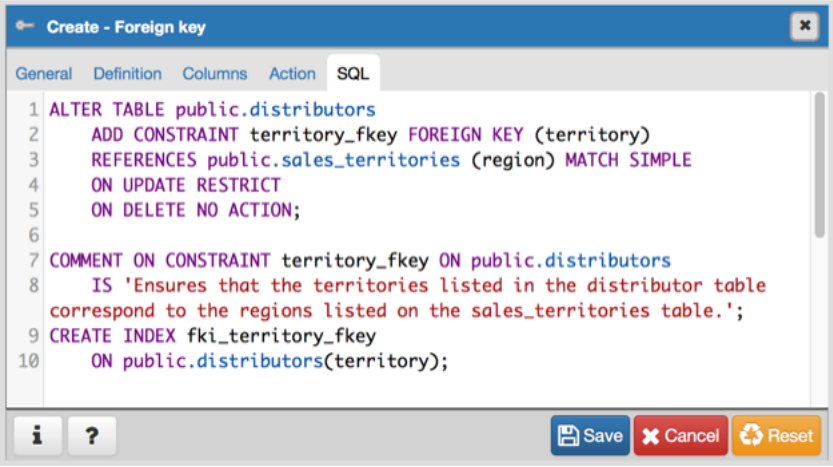
NO ACTION	Produce an error indicating that the deletion or update will create a foreign key constraint violation. If the constraint is deferred, this error will be produced at constraint check time if any referencing rows still exist. This is the default.
RESTRICT	Throw an error indicating that the deletion or update would create a foreign key constraint violation. This is the same as NO ACTION except that the check is not deferrable.
CASCADE	Delete any rows referencing the deleted row, or update the values of the referencing column(s) to the new values of the referenced columns, respectively.
SET NULL	Set the referencing column(s) to null.
SET DEFAULT	Set the referencing column(s) to their default values. There must be a row in the referenced table that matches the default values (if they are not null), or the operation will fail.

Click the *SQL* tab to continue.

Your entries in the *Foreign key* dialog generate a SQL command (see an example below). Use the *SQL* tab for review; revisit or switch tabs to make any changes to the SQL command.

Example

The following is an example of the sql command generated by user selections in the *Foreign key* dialog:



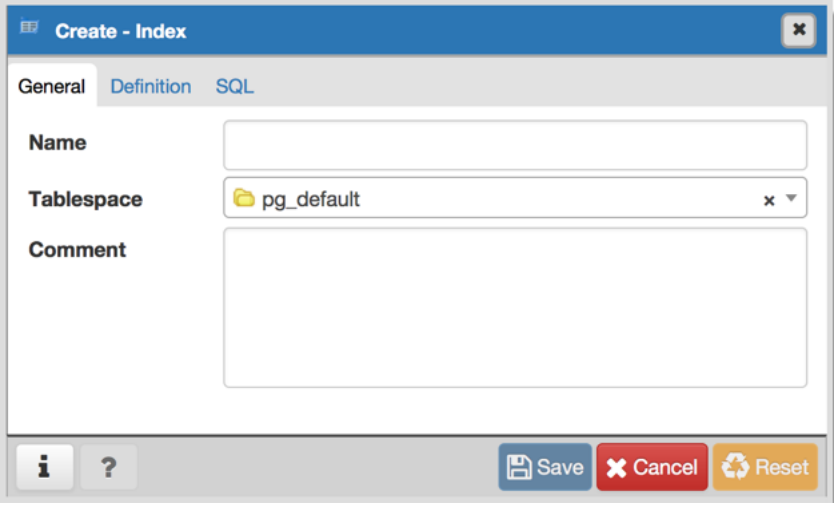
The example shown demonstrates creating a foreign key constraint named *territory_fkey* that matches values in the *distributors* table *territory* column with those of the *sales_territories* table *region* column.

- Click the *Info* button (i) to access online help. View context-sensitive help in the *Tabbed browser*, where a new tab displays the PostgreSQL core documentation.
- Click the *Save* button to save work.
- Click the *Cancel* button to exit without saving work.
- Click the *Reset* button to restore configuration parameters.

4.5 The Index Dialog

Use the *Index* dialog to create an index on a specified table or materialized view.

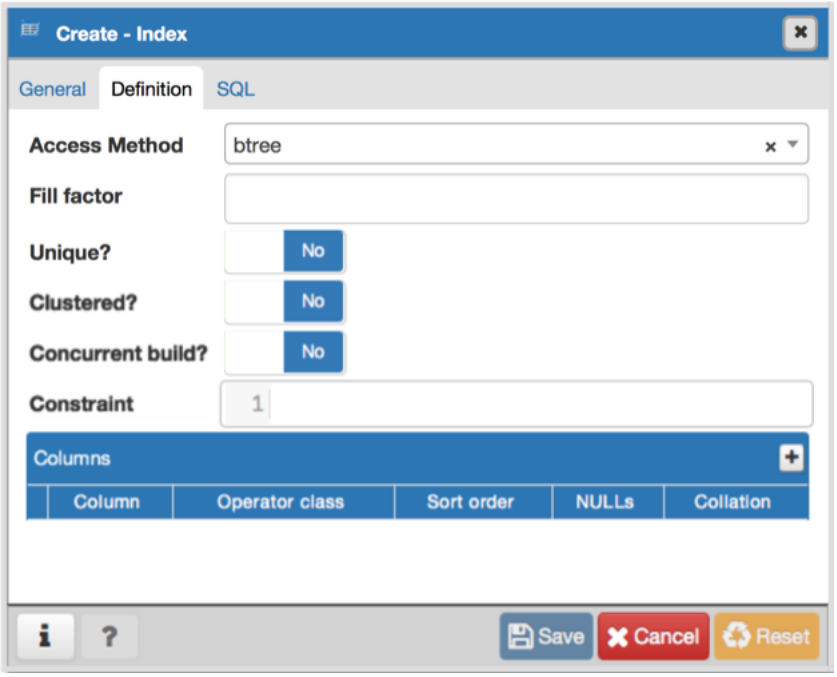
The *Index* dialog organizes the development of a index through the following dialog tabs: *General* and *Definition*. The *SQL* tab displays the SQL code generated by dialog selections.



Use the fields in the *General* tab to identify the index:

- Use the *Name* field to add a descriptive name for the index. The name will be displayed in the *pgAdmin* tree control.
- Use the drop-down listbox next to *Tablespace* to select the tablespace in which the index will reside.
- Store notes about the index in the *Comment* field.

Click the *Definition* tab to continue.



Use the fields in the *Definition* tab to define the index:

- Use the drop-down listbox next to *Access Method* to select an index type:
  - Select *btree* to create a B-tree index. A B-tree index may improve performance when managing equality and range queries on data that can be sorted into some ordering (the default).
  - Select *hash* to create a hash index. A hash index may improve performance when managing simple equality comparisons.

  - Select *gist* to create a GiST index. A GiST index may improve performance when managing values with more than one key.
  - Select *gin* to create a GIN index. A GIN index may improve performance when managing two-dimensional geometric data types and nearest-neighbor searches.
  - Select *spgist* to create a space-partitioned GiST index. A SP-GiST index may improve performance when managing non-balanced data structures.
  - Select *brin* to create a BRIN index. A BRIN index may improve performance when managing minimum and maximum values and ranges.
- Use the *Fill Factor* field to specify a fill factor for the index. The fill factor specifies how full the selected method will try to fill each index page.
- Move the *Unique?* switch to the *Yes* position to check for duplicate values in the table when the index is created and when data is added. The default is *No*.
- Move the *Clustered?* switch to the *Yes* position to instruct the server to cluster the table.
- Move the *Concurrent build?* switch to the *Yes* position to build the index without taking any locks that prevent concurrent inserts, updates, or deletes on the table.
- Use the *Constraint* field to provide a constraint expression; a constraint expression limits the entries in the index to those rows that satisfy the constraint.

Use the context-sensitive fields in the *Columns* panel to specify which column(s) the index queries. Click the *Add* icon (+) to add a column:

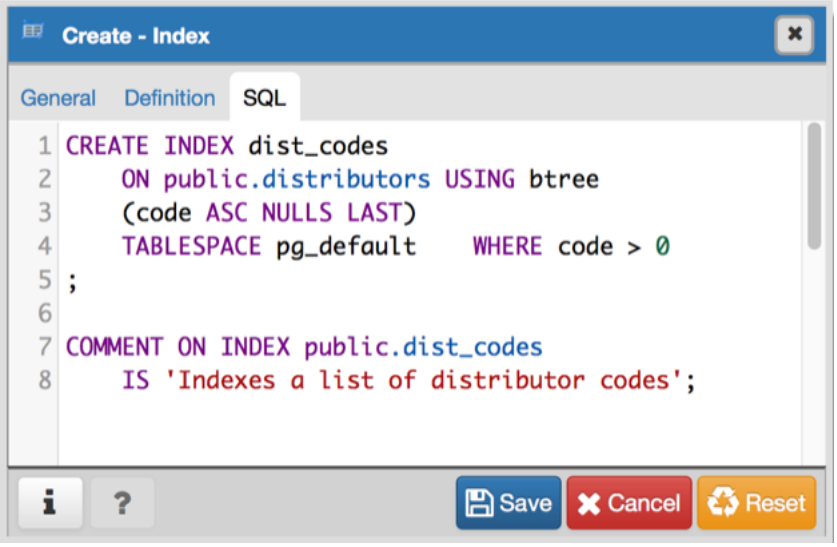
- Use the drop-down listbox in *Column* field to select the name of the column from the table.
- If enabled, use the drop-down listbox to select an available *Operator class* to specify the type of action performed on the column.
- If enabled, move the *Sort order* switch to specify the sort order:
  - Select *ASC* to specify an ascending sort order (the default);
  - Select *DESC* to specify a descending sort order.
- If enabled, move the *Nulls* switch to specify the sort order of nulls:
  - Select *First* to specify nulls sort before non-nulls;
  - Select *Last* to specify nulls sort after non-nulls (the default).
- Use the drop-down listbox in the *Collation* field to select a collation to use for the index.

Click the *SQL* tab to continue.

Your entries in the *Index* dialog generate a SQL command (see an example below). Use the *SQL* tab for review; revisit or switch tabs to make any changes to the SQL command.

Example

The following is an example of the sql command generated by user selections in the *Index* dialog:



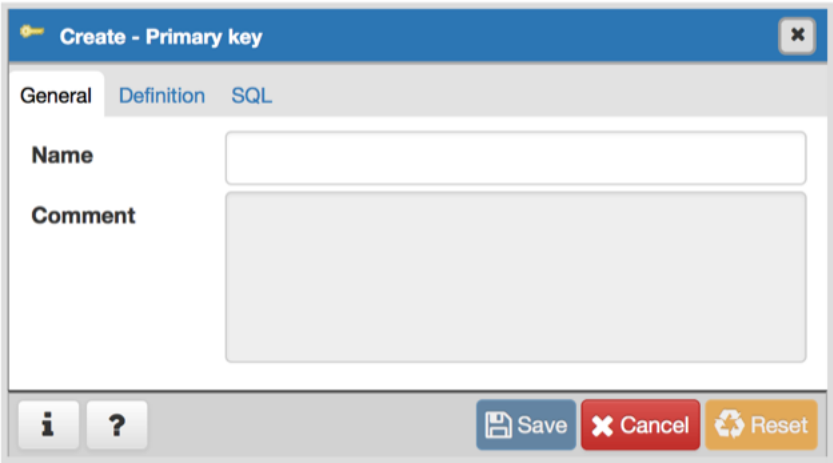
The example shown demonstrates creating an index named *dist_codes* that indexes the values in the *code* column of the *distributors* table.

- Click the *Info* button (i) to access online help. View context-sensitive help in the *Tabbed browser*, where a new tab displays the PostgreSQL core documentation.
- Click the *Save* button to save work.
- Click the *Cancel* button to exit without saving work.
- Click the *Reset* button to restore configuration parameters.

4.6 The Primary key Dialog

Use the *Primary key* dialog to create or modify a primary key constraint. A primary key constraint indicates that a column, or group of columns, uniquely identifies rows in a table. This requires that the values in the selected column(s) be both unique and not null.

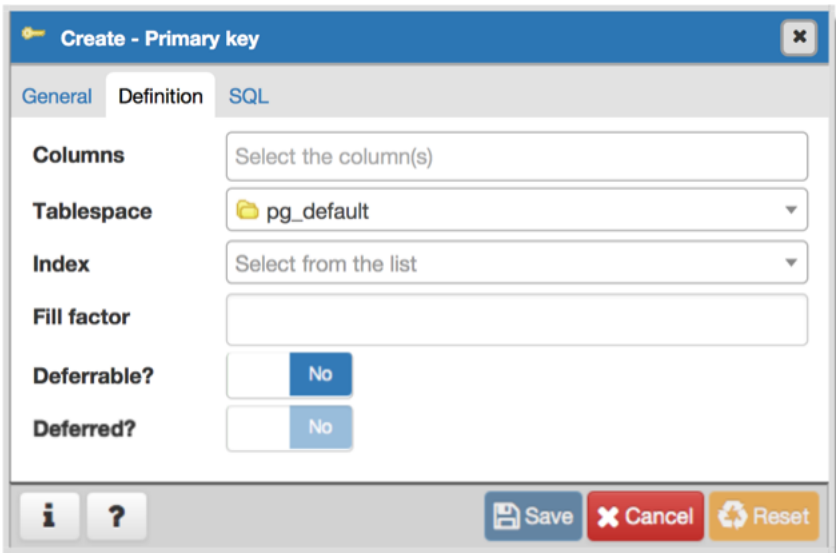
The *Primary key* dialog organizes the development of a primary key constraint through the *General* and *Definition* tabs. The *SQL* tab displays the SQL code generated by dialog selections.



Use the fields in the *General* tab to identify the primary key:

- Use the *Name* field to add a descriptive name for the primary key constraint. The name will be displayed in the *pgAdmin* tree control.

Click the *Definition* tab to continue.



Use the fields in the *Definition* tab to define the primary key constraint:

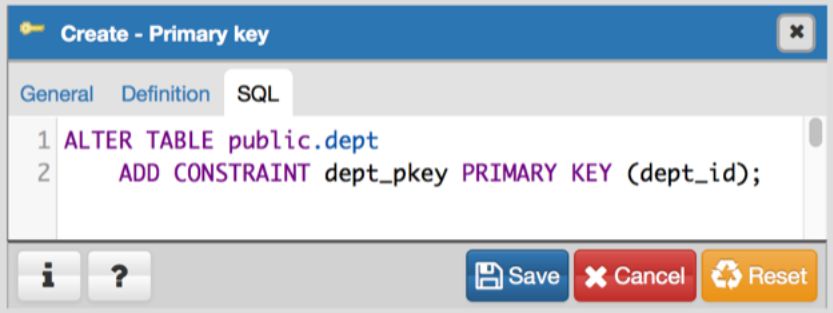
- Click inside the *Columns* field and select one or more column names from the drop-down listbox. To delete a selection, click the *x* to the left of the column name. The primary key constraint should be different from any unique constraint defined for the same table; the selected column(s) for the constraints must be distinct.
- Select the name of the tablespace in which the primary key constraint will reside from the drop-down listbox in the *Tablespace* field.
- Select the name of an index from the drop-down listbox in the *Index* field. This field is optional. Adding a primary key will automatically create a unique B-tree index on the column or group of columns listed in the primary key, and will force the column(s) to be marked NOT NULL.
- Use the *Fill Factor* field to specify a fill factor for the table and index. The fill factor for a table is a percentage between 10 and 100. 100 (complete packing) is the default.
- Move the *Deferrable?* switch to the *Yes* position to specify the timing of the constraint is deferrable and can be postponed until the end of the statement. The default is *No*.
- If enabled, move the *Deferred?* switch to the *Yes* position to specify the timing of the constraint is deferred to the end of the statement. The default is *No*.

Click the *SQL* tab to continue.

Your entries in the *Primary key* dialog generate a SQL command (see an example below). Use the *SQL* tab for review; revisit or switch tabs to make any changes to the SQL command.

Example

The following is an example of the sql command generated by user selections in the *Primary key* dialog:



The example shown demonstrates creating a primary key constraint named *dept_pkey* on the *dept_id* column of the *dept* table.

- Click the *Info* button (i) to access online help. View context-sensitive help in the *Tabbed browser*, where a new tab displays the PostgreSQL core documentation.
- Click the *Save* button to save work.
- Click the *Cancel* button to exit without saving work.
- Click the *Reset* button to restore configuration parameters.

4.7 The Rule Dialog

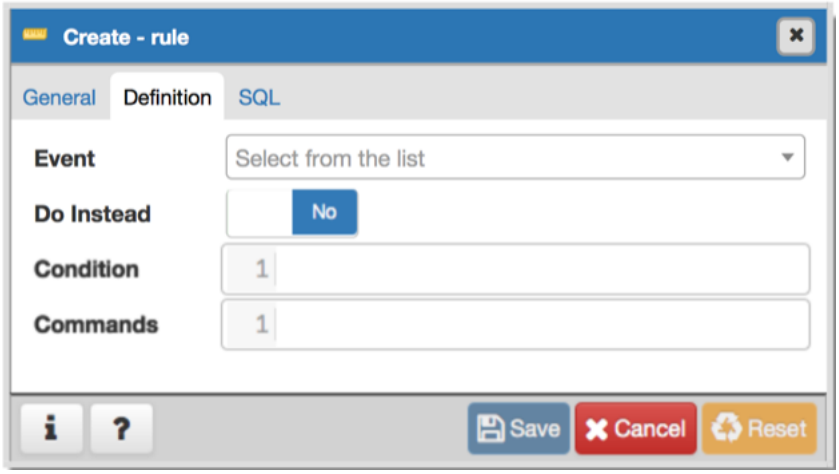
Use the *Rule* dialog to define or modify a rule for a specified table or view. A PostgreSQL rule allows you to define an additional action that will be performed when a SELECT, INSERT, UPDATE, or DELETE is performed against a table.

The *Rule* dialog organizes the development of a rule through the *General*, and *Definition* tabs. The *SQL* tab displays the SQL code generated by dialog selections.

Use the fields in the *General* tab to identify the rule:

- Use the *Name* field to add a descriptive name for the rule. The name will be displayed in the *pgAdmin* tree control. Multiple rules on the same table are applied in alphabetical name order.
- Store notes about the rule in the *Comment* field.

Click the *Definition* tab to continue.



Use the fields in the *Definition* tab to write parameters:

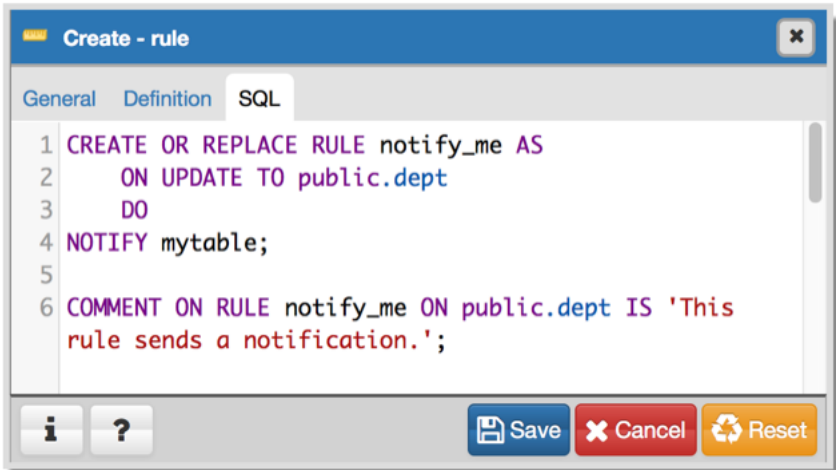
- Click inside the *Event* field to select the type of event that will invoke the rule; event may be *Select*, *Insert*, *Update*, or *Delete*.
- Move the *Do Instead* switch to *Yes* indicate that the commands should be executed instead of the original command; if Do Instead specifies *No*, the rule will be invoked in addition to the original command.
- Specify a SQL conditional expression that returns a boolean value in the *Condition* editor.
- Provide a command in the *Commands* editor that defines the action performed by the rule.

Click the *SQL* tab to continue.

Your entries in the *Rule* dialog generate a SQL command (see an example below). Use the *SQL* tab for review; revisit or switch tabs to make any changes to the SQL command.

Example

The following is an example of the sql command generated by user selections in the *Rule* dialog:



The example sends a notification when an UPDATE executes against a table.

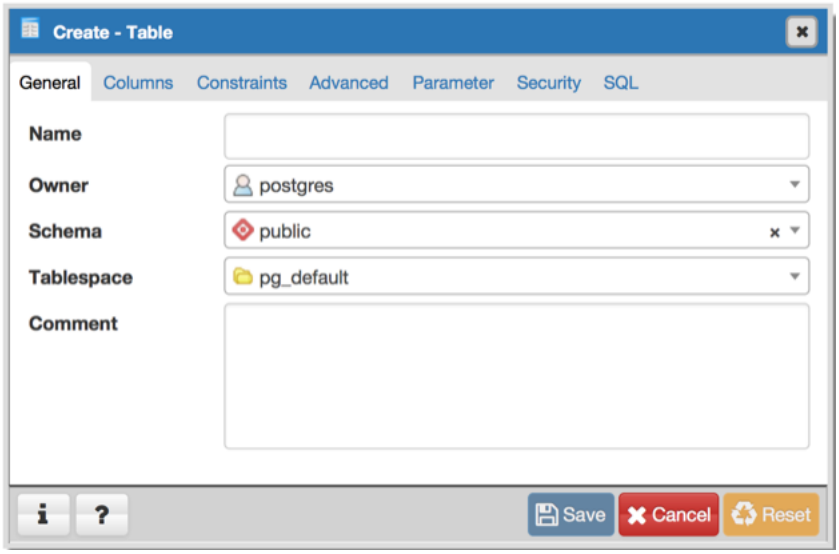
- Click the *Info* button (i) to access online help. View context-sensitive help in the *Tabbed browser*, where a new tab displays the PostgreSQL core documentation.
- Click the *Save* button to save work.
- Click the *Cancel* button to exit without saving work.

- Click the *Reset* button to restore configuration parameters.

4.8 The Table Dialog

Use the *Table* dialog to create or modify a table.

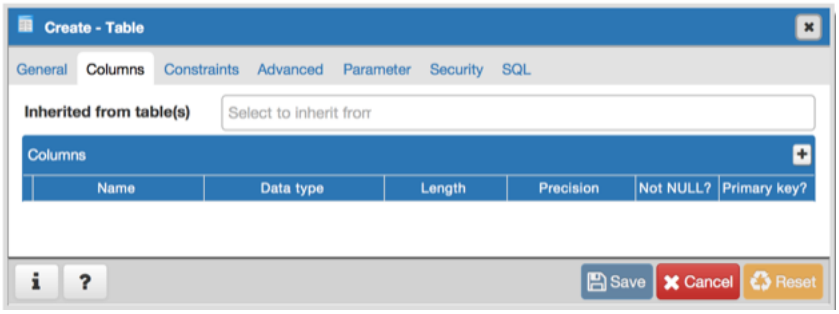
The *Table* dialog organizes the development of a table through the following dialog tabs: *General*, *Columns*, *Constraints*, *Advanced*, *Parameter*, and *Security*. The *SQL* tab displays the SQL code generated by dialog selections.



Use the fields in the *General* tab to identify the table:

- Use the *Name* field to add a descriptive name for the table. A table cannot have the same name as any existing table, sequence, index, view, foreign table, or data type in the same schema. The name specified will be displayed in the *pgAdmin* tree control. This field is required.
- Select the owner of the table from the drop-down listbox in the *Owner* field. By default, the owner of the table is the role that creates the table.
- Select the name of the schema in which the table will reside from the drop-down listbox in the *Schema* field.
- Use the drop-down listbox in the *Tablespace* field to specify the tablespace in which the table will be stored.
- Store notes about the table in the *Comment* field.

Click the *Columns* tab to continue.



Use the drop-down listbox next to *Inherited from table(s)* to specify any parent table(s); the table will inherit columns from the selected parent table(s). Click inside the *Inherited from table(s)* field to select a table name from a drop-down list. Repeat to add any other parent tables. Delete a selected table by clicking the *x* to the left of the parent name. Note

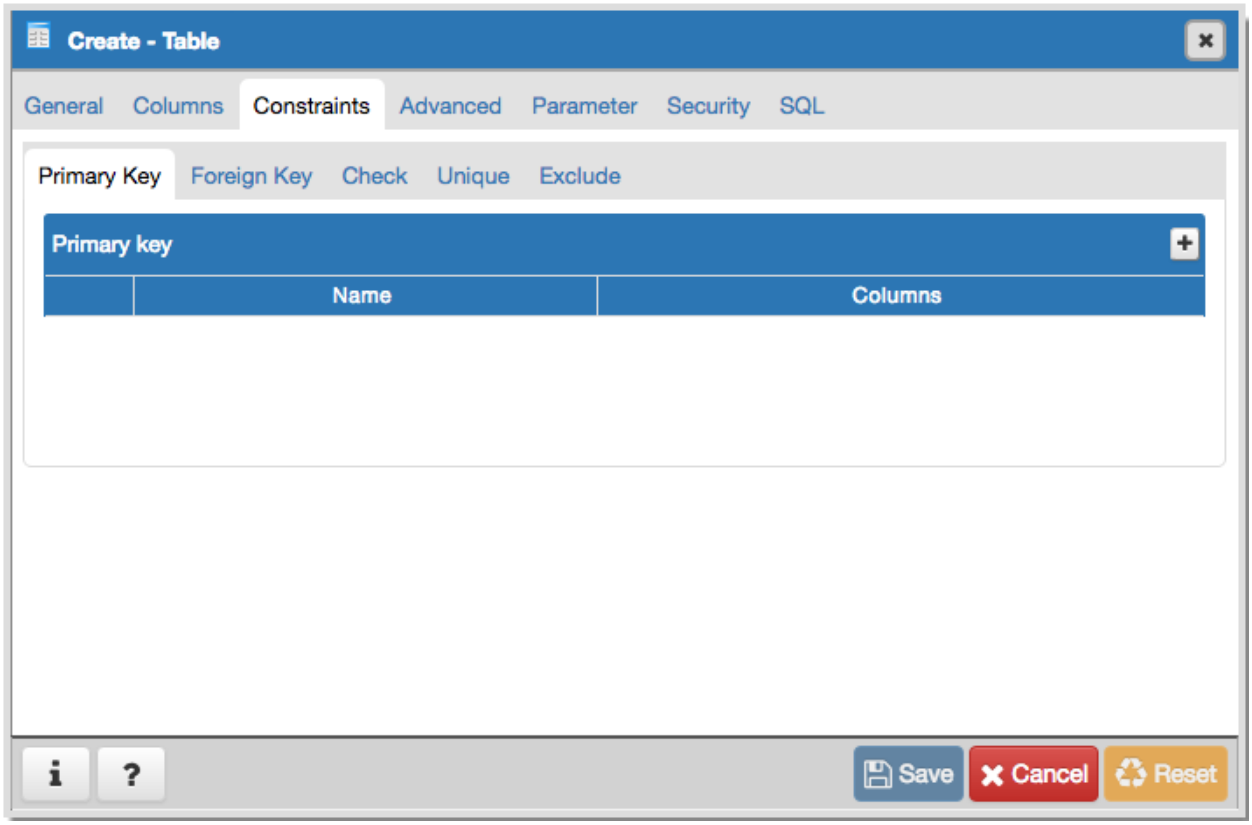
that inherited column names and datatypes are not editable in the current dialog; they must be modified at the parent level.

Click the *Add* icon (+) to specify the names of columns and their datatypes in the *Columns* table:

- Use the *Name* field to add a descriptive name for the column.
- Use the drop-down listbox in the *Data type* field to select a data type for the column. This can include array specifiers. For more information on the data types supported by PostgreSQL, refer to Chapter 8 of the core documentation.
- If enabled, use the *Length* and *Precision* fields to specify the maximum number of significant digits in a numeric value, or the maximum number of characters in a text value.
- Move the *Not NULL?* switch to the *Yes* position to require a value in the column field.
- Move the *Primary key?* switch to the *Yes* position to specify the column is the primary key constraint.

  Click the *Add* icon (+) to add additional columns; to discard a column, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Click the *Constraints* tab to continue.



Use the fields in the *Constraints* tab to provide a table or column constraint. Optional constraint clauses specify constraints (tests) that new or updated rows must satisfy for an *INSERT* or *UPDATE* operation to succeed. Select the appropriate constraint type by selecting one of the following tabs on the *Constraints* panel:

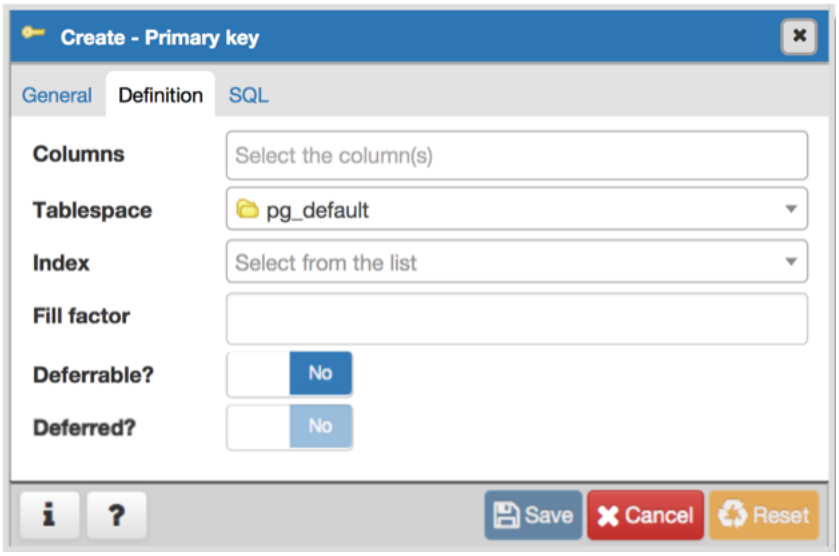
Tab Name	Constraint
Primary Key	Provides a unique identifier for each row in the table.
Foreign Key	Maintains referential integrity between two tables.
Check	Requires data satisfies an expression or condition before insertion or modification.
Unique	Ensures that the data contained in a column, or a group of columns, is unique among all the rows in the table.
Exclude	Guarantees that if any two rows are compared on the specified column or expression (using the specified operator), at least one of the operator comparisons will return false or null.

To add a primary key for the table, select the *Primary Key* tab, and click the *Add* icon (+). To define the primary key, click the *Edit* icon to the left of the *Trash* icon. A dialog similar to the *Primary key* dialog (accessed by right clicking on *Constraints* in the *pgAdmin* tree control) opens.

Use the fields in the *General* tab to identify the primary key:

- Use the *Name* field to add a descriptive name for the primary key constraint. The name will be displayed in the *pgAdmin* tree control.
- Provide notes about the primary key in the *Comment* field.

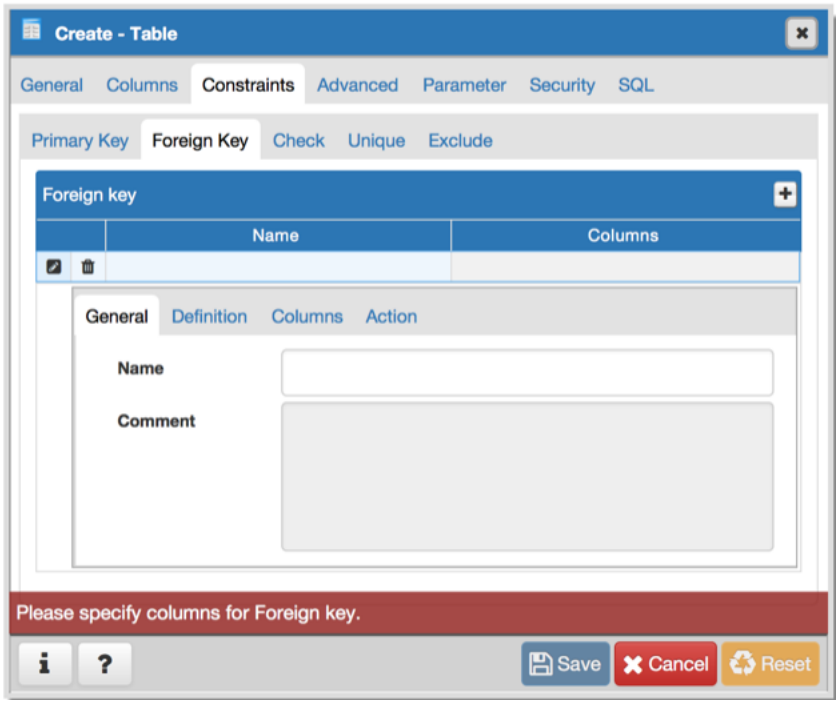
Click the *Definition* tab to continue.



Use the fields in the *Definition* tab to define the primary key constraint:

- Click inside the *Columns* field and select one or more column names from the drop-down listbox. To delete a selection, click the *x* to the left of the column name. The primary key constraint should be different from any unique constraint defined for the same table; the selected column(s) for the constraints must be distinct.
- Select the name of the tablespace in which the primary key constraint will reside from the drop-down listbox in the *Tablespace* field.
- Use the *Fill Factor* field to specify a fill factor for the table and index. The fill factor for a table is a percentage between 10 and 100. 100 (complete packing) is the default.

- Move the *Deferrable?* switch to the *Yes* position to specify the timing of the constraint is deferrable and can be postponed until the end of the statement. The default is *No*.
- If enabled, move the *Deferred?* switch to the *Yes* position to specify the timing of the constraint is deferred to the end of the statement. The default is *No*.

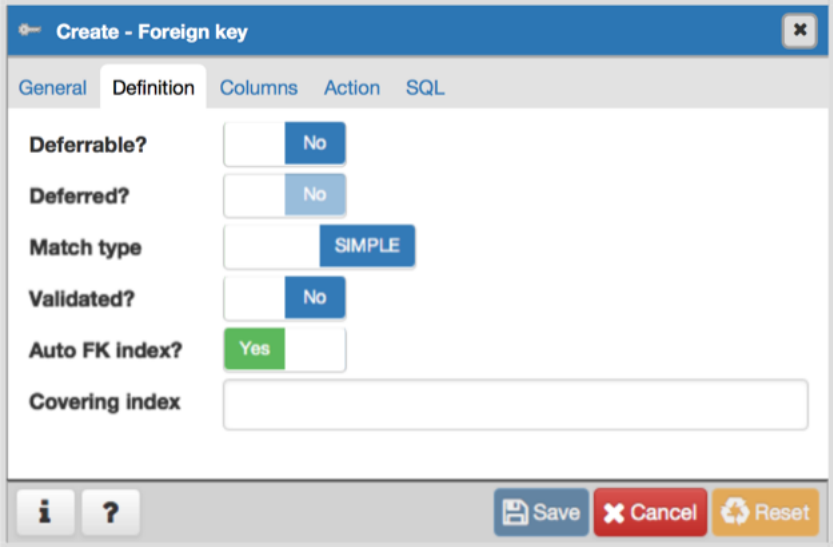


To add a foreign key constraint, select the *Foreign Key* tab, and click the *Add* icon (+). To define the constraint, click the *Edit* icon to the left of the *Trash* icon. A dialog similar to the *Foreign key* dialog (accessed by right clicking on *Constraints* in the *pgAdmin* tree control) opens.

Use the fields in the *General* tab to identify the foreign key constraint:

- Use the *Name* field to add a descriptive name for the foreign key constraint. The name will be displayed in the *pgAdmin* tree control.
- Provide notes about the foreign key in the *Comment* field.

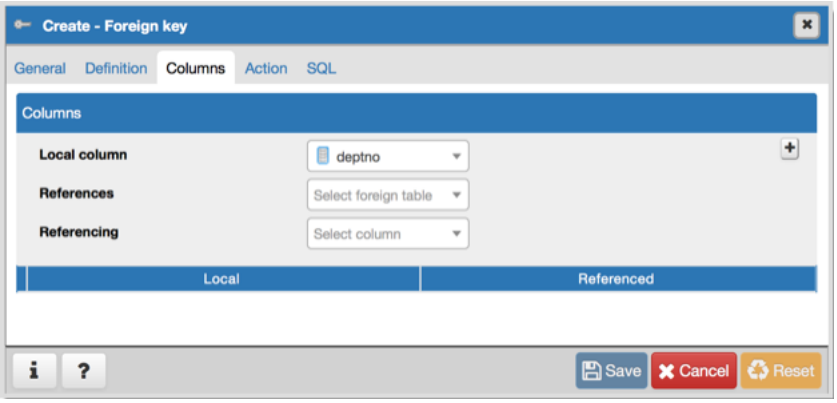
Click the *Definition* tab to continue.



Use the fields in the *Definition* tab to define the foreign key constraint:

- Move the *Deferrable?* switch to the *Yes* position to specify the timing of the constraint is deferrable and can be postponed until the end of the statement. The default is *No*.
- If enabled, move the *Deferred?* switch to the *Yes* position to specify the timing of the constraint is deferred to the end of the statement. The default is *No*.
- Move the *Match type* switch specify the type of matching that is enforced by the constraint:
  - Select *Full* to indicate that all columns of a multicolumn foreign key must be null if any column is null; if all columns are null, the row is not required to have a match in the referenced table.
  - Select *Simple* to specify that a single foreign key column may be null; if any column is null, the row is not required to have a match in the referenced table.
- Move the *Validated* switch to the *Yes* position to instruct the server to validate the existing table content (against a foreign key or check constraint) when you save modifications to this dialog.
- Move the *Auto FK Index* switch to the *No* position to disable the automatic index feature.
- The field next to *Covering Index* generates the name of an index if the *Auto FK Index* switch is in the *Yes* position; or, this field is disabled.

Click the *Columns* tab to continue.



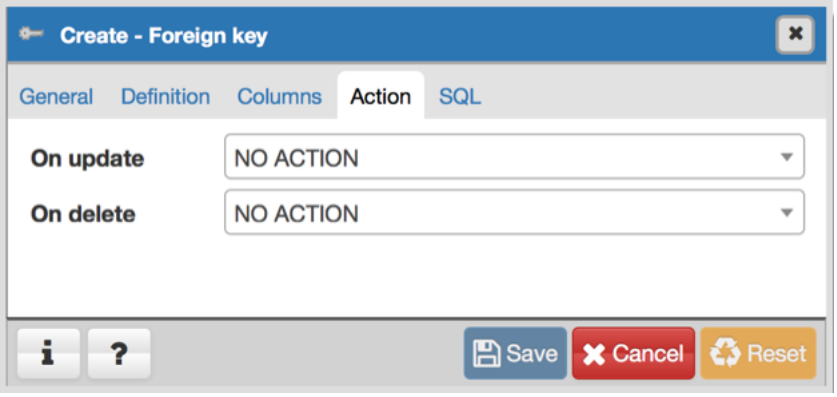
Use the fields in the *Columns* tab to specify one or more reference column(s). A Foreign Key constraint requires that one or more columns of a table must only contain values that match values in the referenced column(s) of a row of a

referenced table:

- Use the drop-down listbox next to *Local column* to specify the column in the current table that will be compared to the foreign table.
- Use the drop-down listbox next to *References* to specify the name of the table in which the comparison column(s) resides.
- Use the drop-down listbox next to *Referencing* to specify a column in the foreign table.

Click the *Add* icon (+) to add a column to the list; repeat the steps above and click the *Add* icon (+) to add additional columns. To discard an entry, click the trash icon to the left of the entry and confirm deletion in the *Delete Row* popup.

Click the *Action* tab to continue.

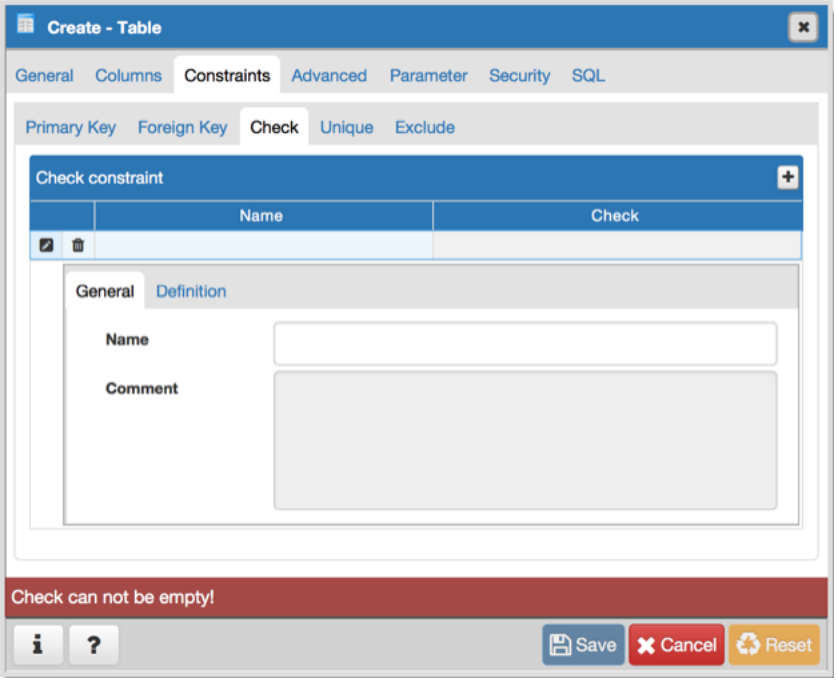


Use the drop-down listboxes on the *Action* tab to specify behavior related to the foreign key constraint that will be performed when data within the table is updated or deleted:

- Use the drop-down listbox next to *On update* to select an action that will be performed when data in the table is updated.
- Use the drop-down listbox next to *On delete* to select an action that will be performed when data in the table is deleted.

The supported actions are:

NO ACTION	Produce an error indicating that the deletion or update will create a foreign key constraint violation. If the constraint is deferred, this error will be produced at constraint check time if any referencing rows still exist. This is the default.
RESTRICT	Throw an error indicating that the deletion or update would create a foreign key constraint violation. This is the same as NO ACTION except that the check is not deferrable.
CASCADE	Delete any rows referencing the deleted row, or update the values of the referencing column(s) to the new values of the referenced columns, respectively.
SET NULL	Set the referencing column(s) to null.
SET DEFAULT	Set the referencing column(s) to their default values. There must be a row in the referenced table that matches the default values (if they are not null), or the operation will fail.

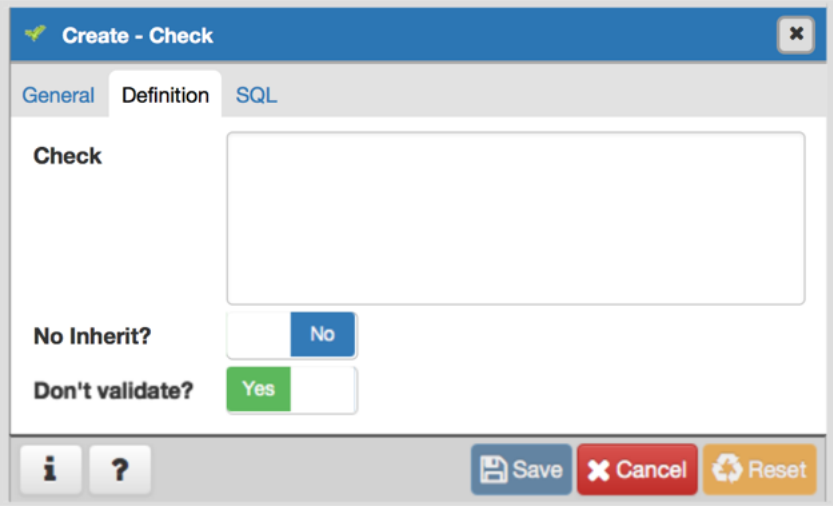


To add a check constraint, select the *Check* tab on the panel, and click the *Add* icon (+). To define the check constraint, click the *Edit* icon to the left of the *Trash* icon. A dialog similar to the *Check* dialog (accessed by right clicking on *Constraints* in the *pgAdmin* tree control) opens.

Use the fields in the *General* tab to identify the check constraint:

- Use the *Name* field to add a descriptive name for the check constraint. The name will be displayed in the *pgAdmin* tree control. With PostgreSQL 9.5 forward, when a table has multiple check constraints, they will be tested for each row in alphabetical order by name and after NOT NULL constraints.
- Provide notes about the check constraint in the *Comment* field.

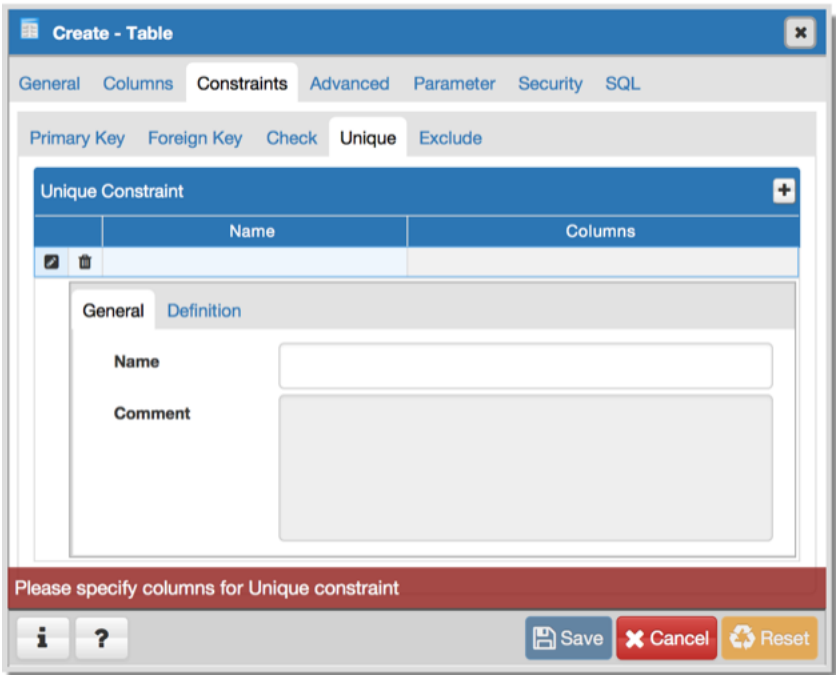
Click the *Definition* tab to continue.



Use the fields in the *Definition* tab to define the check constraint:

- Provide the expression that a row must satisfy in the *Check* field. This field is required.

- Move the *No Inherit?* switch to the *Yes* position to specify this constraint is automatically inherited by a table's children. The default is *No*.
- Move the *Don't validate?* switch to the *No* position to skip validation of existing data; the constraint may not hold for all rows in the table. The default is *Yes*.

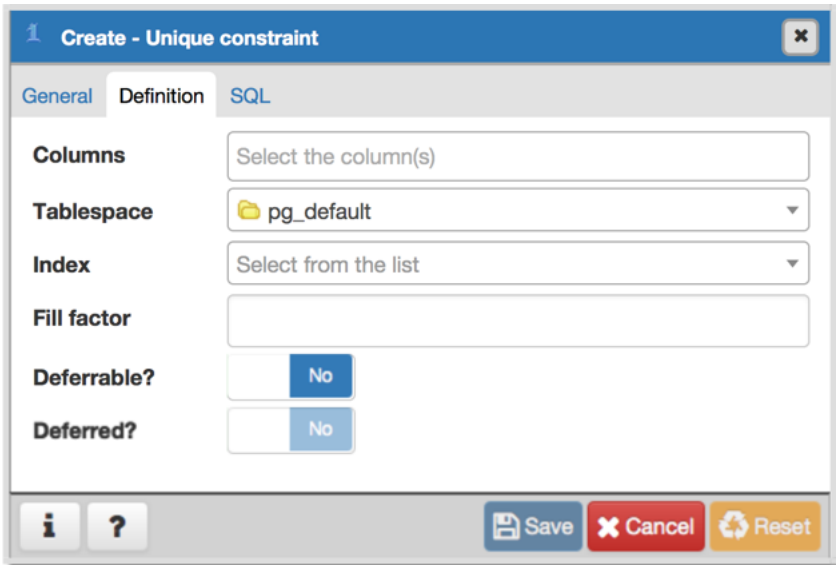


To add a unique constraint, select the *Unique* tab on the panel, and click the *Add* icon (+). To define the constraint, click the *Edit* icon to the left of the *Trash* icon. A dialog similar to the *Unique constraint* dialog (accessed by right clicking on *Constraints* in the *pgAdmin* tree control) opens.

Use the fields in the *General* tab to identify the unique constraint:

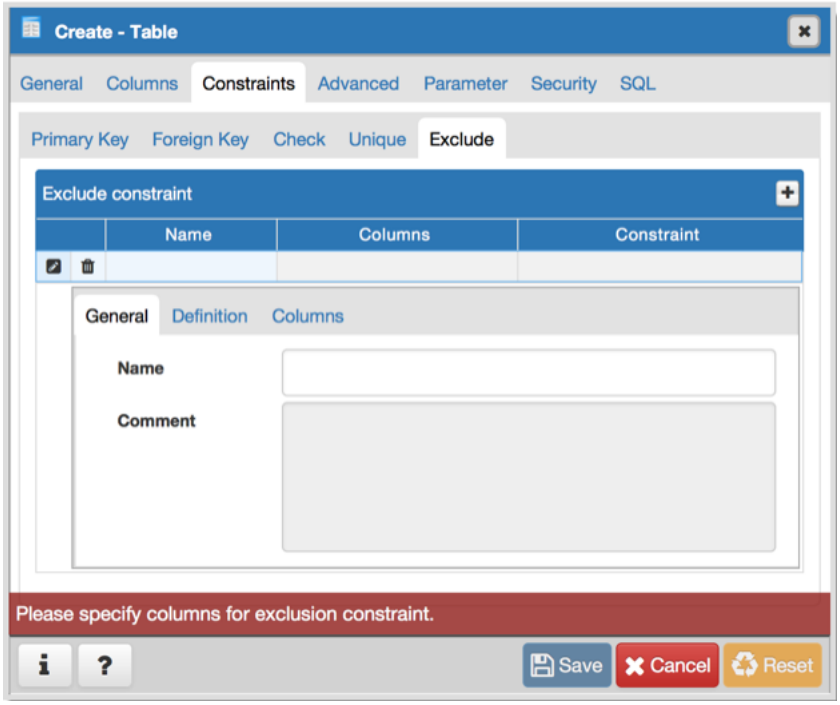
- Use the *Name* field to add a descriptive name for the unique constraint. The name will be displayed in the *pgAdmin* tree control.
- Provide notes about the unique constraint in the *Comment* field.

Click the *Definition* tab to continue.



Use the fields in the *Definition* tab to define the unique constraint:

- Click inside the *Columns* field and select one or more column names from the drop-down listbox. To delete a selection, click the *x* to the left of the column name. The unique constraint should be different from the primary key constraint defined for the same table; the selected column(s) for the constraints must be distinct.
- Select the name of the tablespace in which the unique constraint will reside from the drop-down listbox in the *Tablespace* field.
- Use the *Fill Factor* field to specify a fill factor for the table and index. The fill factor for a table is a percentage between 10 and 100. 100 (complete packing) is the default.
- Move the *Deferrable?* switch to the *Yes* position to specify the timing of the constraint is deferrable and can be postponed until the end of the statement. The default is *No*.
- If enabled, move the *Deferred?* switch to the *Yes* position to specify the timing of the constraint is deferred to the end of the statement. The default is *No*.

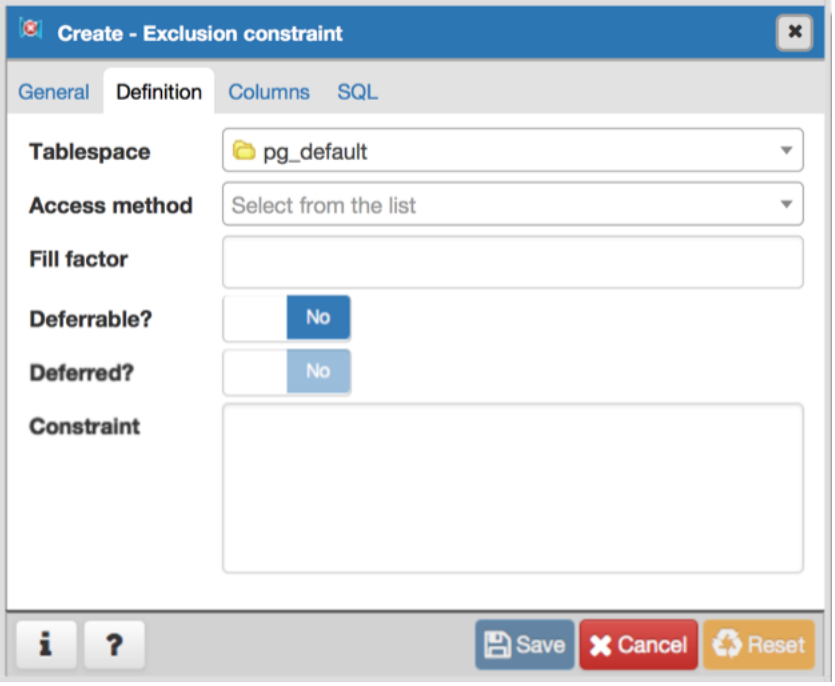


To add an exclusion constraint, select the *Exclude* tab on the panel, and click the *Add* icon (+). To define the constraint, click the *Edit* icon to the left of the *Trash* icon. A dialog similar to the *Exclusion constraint* dialog (accessed by right clicking on *Constraints* in the *pgAdmin* tree control) opens.

Use the fields in the *General* tab to identify the exclusion constraint:

- Use the *Name* field to provide a descriptive name for the exclusion constraint. The name will be displayed in the *pgAdmin* tree control.
- Provide notes about the exclusion constraint in the *Comment* field.

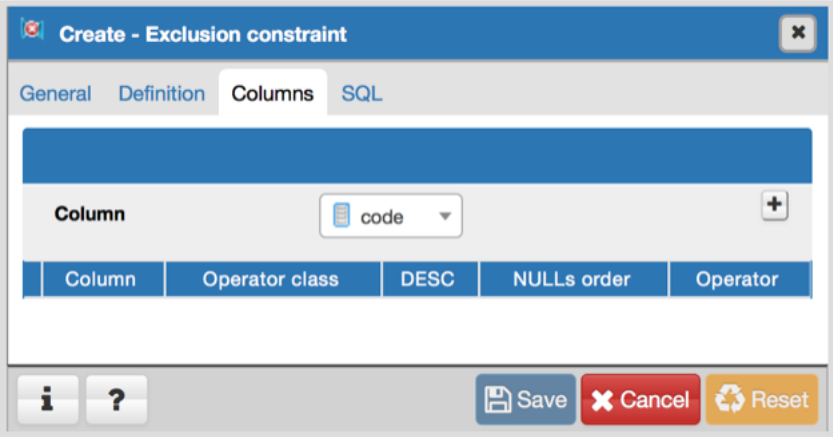
Click the *Definition* tab to continue.



Use the fields in the *Definition* tab to define the exclusion constraint:

- Use the drop-down listbox next to *Tablespace* to select the tablespace in which the index associated with the exclude constraint will reside.
- Use the drop-down listbox next to *Access method* to specify the type of index that will be used when implementing the exclusion constraint:
  - Select *gist* to specify a GiST index (the default).
  - Select *spgist* to specify a space-partitioned GiST index.
  - Select *btree* to specify a B-tree index.
  - Select *hash* to specify a hash index.
- Use the *Fill Factor* field to specify a fill factor for the table and associated index. The fill factor is a percentage between 10 and 100. 100 (complete packing) is the default.
- Move the *Deferrable?* switch to the *Yes* position to specify that the timing of the constraint is deferrable, and can be postponed until the end of the statement. The default is *No*.
- If enabled, move the *Deferred?* switch to the *Yes* position to specify the timing of the constraint is deferred to the end of the statement. The default is *No*.
- Use the *Constraint* field to provide a condition that a row must satisfy to be included in the table.

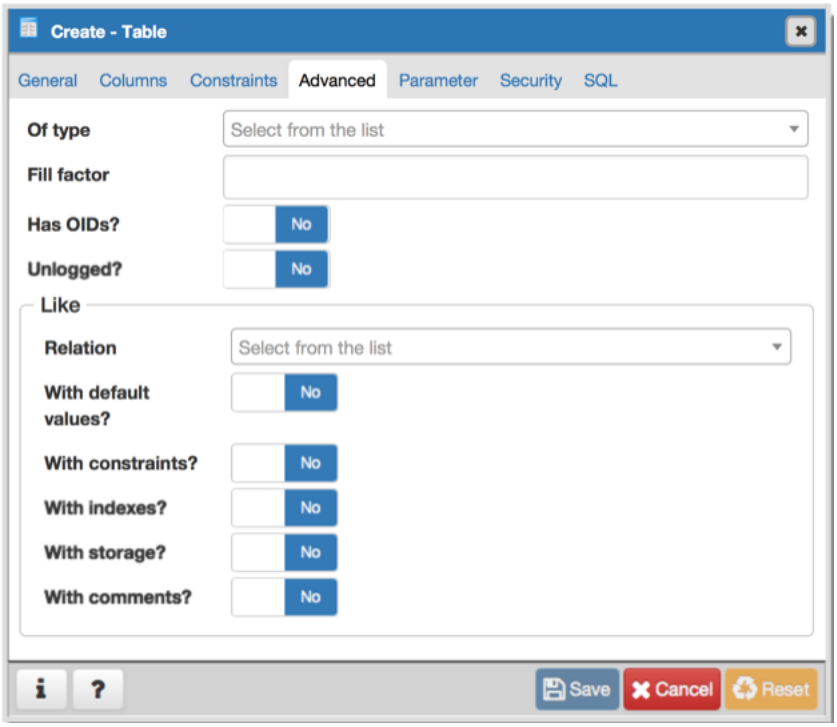
Click the *Columns* tab to continue.



Use the fields in the *Columns* tab to to specify the column(s) to which the constraint applies. Use the drop-down listbox next to *Column* to select a column and click the *Add* icon (+) to provide details of the action on the column:

- The *Column* field is populated with the selection made in the *Column* drop-down listbox.
- If applicable, use the drop-down listbox in the *Operator class* to specify the operator class that will be used by the index for the column.
- Move the *DESC* switch to *DESC* to specify a descending sort order. The default is *ASC* which specifies an ascending sort order.
- Move the *NULLs order* switch to *LAST* to define an ascending sort order for NULLs. The default is *FIRST* which specifies a descending order.
- Use the drop-down list next to *Operator* to specify a comparison or conditional operator.

Click the *Advanced* tab to continue.



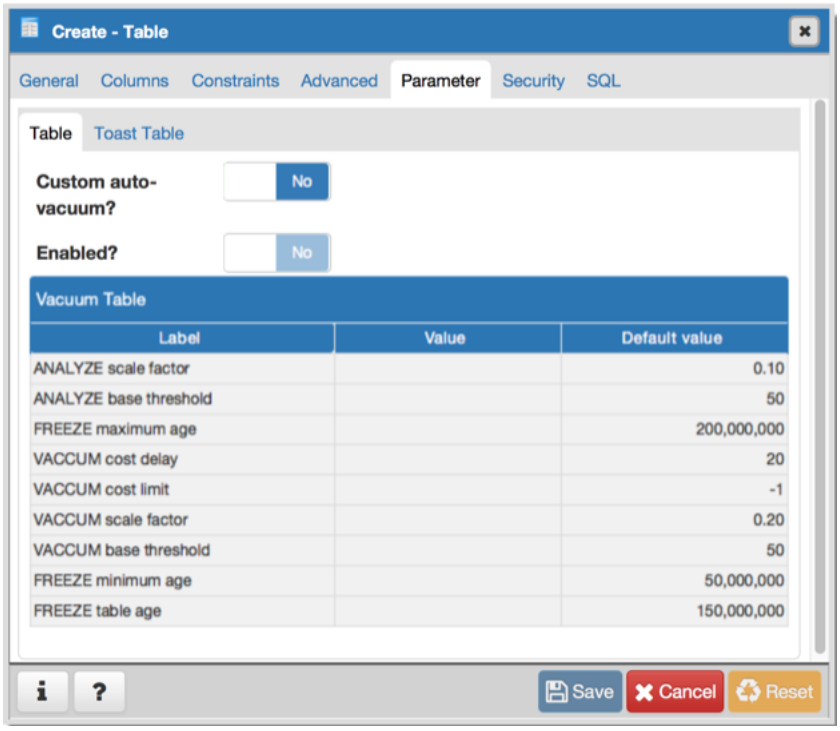
Use the fields in the *Advanced* tab to define advanced features for the table:

- Use the drop-down listbox next to *Of type* to copy the table structure from the specified composite type. Please note that a typed table will be dropped if the type is dropped (with DROP TYPE ... CASCADE).
- Use the *Fill Factor* field to specify a fill factor for the table. The fill factor for a table is a percentage between 10 and 100. 100 (complete packing) is the default.
- Move the *Has OIDs?* switch to the *Yes* position to specify that each row within a table has a system-assigned object identifier. The default is *No*.
- Move the *Unlogged?* switch to the *Yes* position to disable logging for the table. Data written to an unlogged table is not written to the write-ahead log. Any indexes created on an unlogged table are automatically unlogged as well. The default is *No*.

Use the fields in the **Like** box to specify which attributes of an existing table from which a table will automatically copy column names, data types, and not-null constraints; after saving the new or modified table, any changes to the original table will not be applied to the new table.

- Use the drop-down listbox next to *Relation* to select a reference table.
- Move the *With default values?* switch to the *Yes* position to copy default values.
- Move the *With constraints?* switch to the *Yes* position to copy table and column constraints.
- Move the *With indexes?* switch to the *Yes* position to copy indexes.
- Move the *With storage?* switch to the *Yes* position to copy storage settings.
- Move the *With comments?* switch to the *Yes* position to copy comments.

Click the *Parameter* tab to continue.

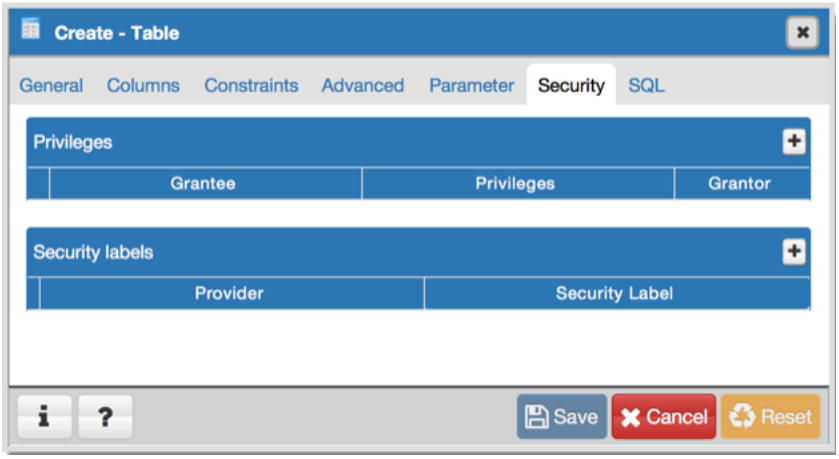


Use the tabs nested inside the *Parameter* tab to specify VACUUM and ANALYZE thresholds; use the *Table* tab and the *Toast Table* tab to customize values for the table and the associated toast table:

- Move the *Custom auto-vacuum?* switch to the *Yes* position to perform custom maintenance on the table.
- Move the *Enabled?* switch to the *Yes* position to select values in the *Vacuum table*. The *Vacuum Table* provides default values for maintenance operations.

Provide a custom value in the *Value* column for each metric listed in the *Label* column.

Click the *Security* tab to continue.



Use the *Security* tab to assign privileges and define security labels.

Use the *Privileges* panel to assign privileges to a role. Click the *Add* icon (+) to set privileges for database objects:

- Select the name of the role from the drop-down listbox in the *Grantee* field.
- Click inside the *Privileges* field. Check the boxes to the left of one or more privileges to grant the selected privilege to the specified user.
- Select the name of the role from the drop-down listbox in the *Grantor* field. The default grantor is the owner of the database.

Click the *Add* icon (+) to assign additional privileges; to discard a privilege, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Use the *Security Labels* panel to define security labels applied to the function. Click the *Add* icon (+) to add each security label selection:

- Specify a security label provider in the *Provider* field. The named provider must be loaded and must consent to the proposed labeling operation.
- Specify a a security label in the *Security Label* field. The meaning of a given label is at the discretion of the label provider. PostgreSQL places no restrictions on whether or how a label provider must interpret security labels; it merely provides a mechanism for storing them.

Click the *Add* icon (+) to assign additional security labels; to discard a security label, click the trash icon to the left of the row and confirm deletion in the *Delete Row* popup.

Click the *SQL* tab to continue.

Your entries in the *Table* dialog generate a SQL command (see an example below). Use the *SQL* tab for review; revisit or switch tabs to make any changes to the SQL command.

Example

The following is an example of the sql command generated by user selections in the *Table* dialog:

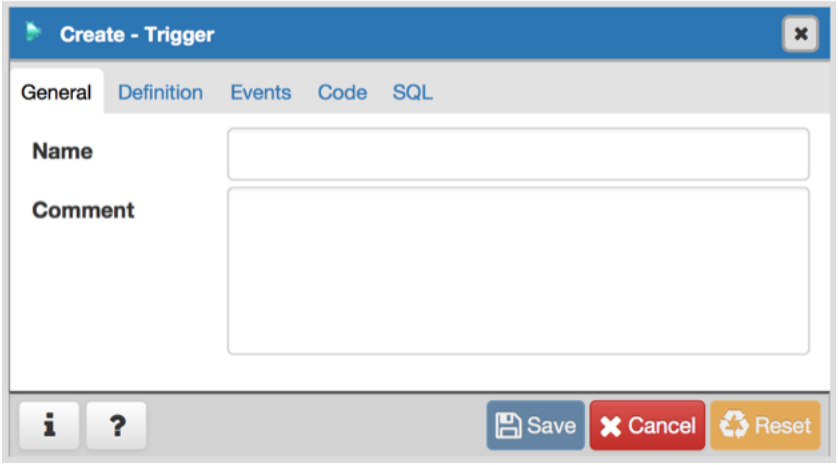
The example shown demonstrates creating a table named *product_category*. It has three columns and a primary key constraint on the *category_id* column.

- Click the *Info* button (i) to access online help. View context-sensitive help in the *Tabbed browser*, where a new tab displays the PostgreSQL core documentation.
- Click the *Save* button to save work.
- Click the *Cancel* button to exit without saving work.
- Click the *Reset* button to restore configuration parameters.

4.9 The Trigger Dialog

Use the *Trigger* dialog to create a trigger or modify an existing trigger. A trigger executes a specified function when certain events occur.

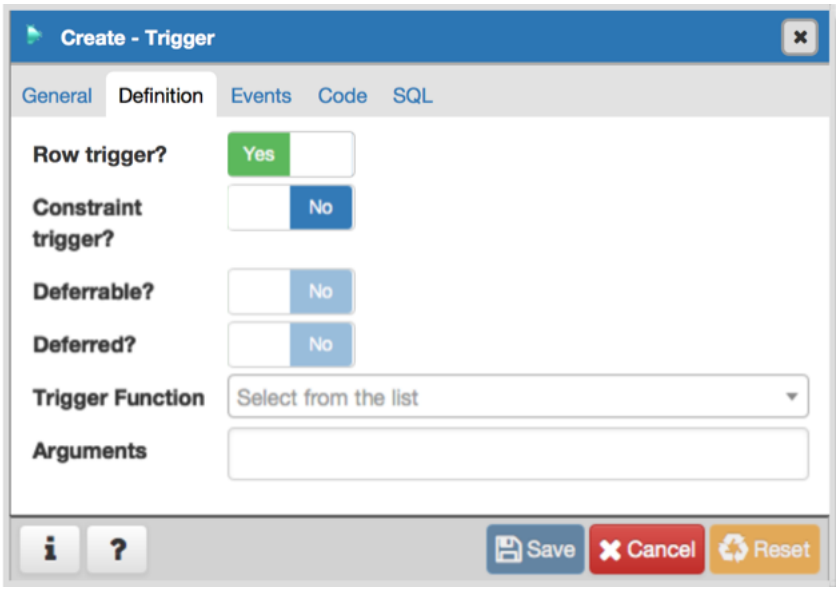
The *Trigger* dialog organizes the development of a trigger through the following dialog tabs: *General*, *Definition*, *Events*, and *Code*. The *SQL* tab displays the SQL code generated by dialog selections.



Use the fields in the *General* tab to identify the trigger:

- Use the *Name* field to add a descriptive name for the trigger. This must be distinct from the name of any other trigger for the same table. The name will be displayed in the *pgAdmin* tree control. Note that if multiple triggers of the same kind are defined for the same event, they will be fired in alphabetical order by name.
- Store notes about the trigger in the *Comment* field.

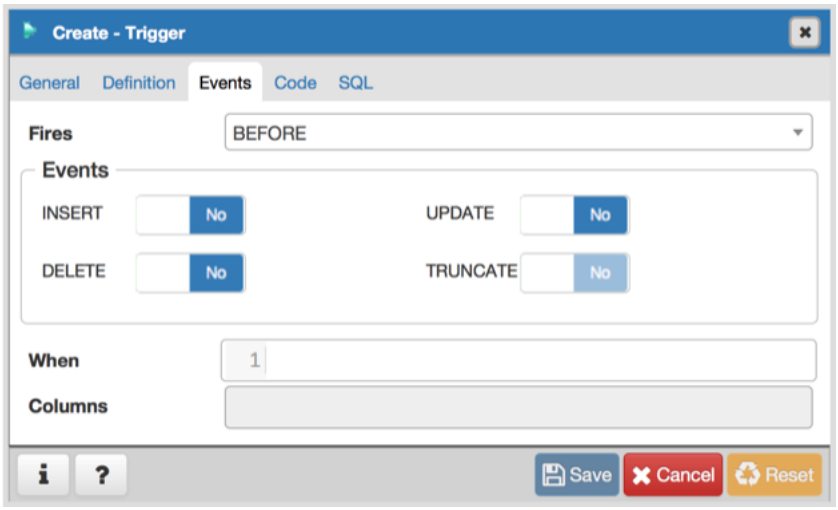
Click the *Definition* tab to continue.



Use the fields in the *Definition* tab to define the trigger:

- Move the *Row trigger?* switch to the *No* position to disassociate the trigger from firing on each row in a table. The default is *Yes*.
- Move the *Constraint trigger?* switch to the *Yes* position to specify the trigger is a constraint trigger.
- If enabled, move the *Deferrable?* switch to the *Yes* position to specify the timing of the constraint trigger is deferrable and can be postponed until the end of the statement. The default is *No*.
- If enabled, move the *Deferred?* switch to the *Yes* position to specify the timing of the constraint trigger is deferred to the end of the statement causing the triggering event. The default is *No*.
- Use the drop-down listbox next to *Trigger Function* to select a trigger function or procedure.
- Use the *Arguments* field to provide an optional (comma-separated) list of arguments to the function when the trigger is executed. The arguments are literal string constants.

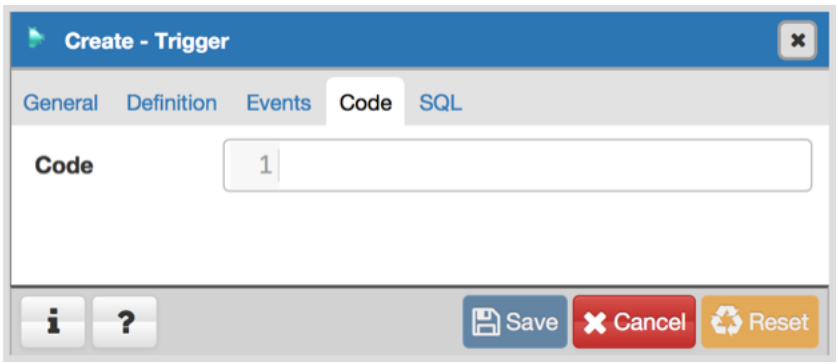
Click the *Events* tab to continue.



Use the fields in the *Events* tab to specify how and when the trigger fires:

- Use the drop-down listbox next to the *Fires* fields to determine if the trigger fires *BEFORE* or *AFTER* a specified event. The default is *BEFORE*.
- Select the type of event(s) that will invoke the trigger; to select an event type, move the switch next to the event to the *YES* position. The supported event types are *INSERT*, *UPDATE*, *DELETE*, and *TRUNCATE*.
- Use the *When* field to provide a boolean condition that will invoke the trigger.
- If defining a column-specific trigger, use the *Columns* field to specify the columns or columns that are the target of the trigger.

Click the *Code* tab to continue.



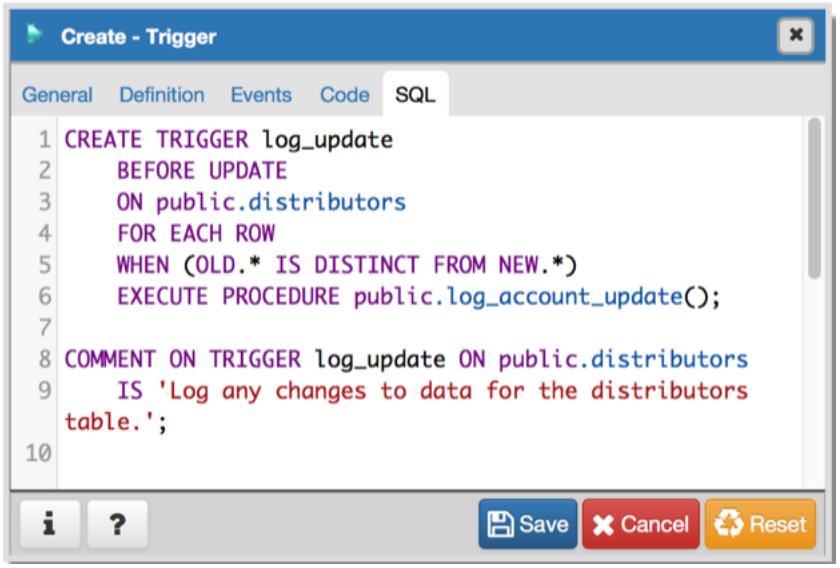
Use the *Code* field to specify any additional code that will be invoked when the trigger fires.

Click the *SQL* tab to continue.

Your entries in the *Trigger* dialog generate a SQL command (see an example below). Use the *SQL* tab for review; revisit or switch tabs to make any changes to the SQL command.

Example

The following is an example of the sql command generated by user selections in the *Trigger* dialog:



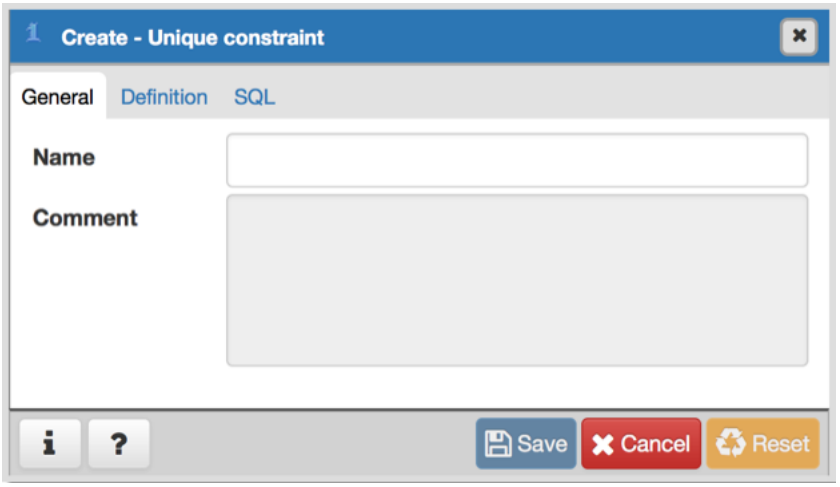
The example demonstrates creating a trigger named *log_update* that calls a procedure named *log_account_update* that logs any updates to the *distributors* table.

- Click the *Info* button (i) to access online help. View context-sensitive help in the *Tabbed browser*, where a new tab displays the PostgreSQL core documentation.
- Click the *Save* button to save work.
- Click the *Cancel* button to exit without saving work.
- Click the *Reset* button to restore configuration parameters.

4.10 The Unique Constraint Dialog

Use the *Unique constraint* dialog to define a unique constraint for a specified table. Unique constraints ensure that the data contained in a column, or a group of columns, is unique among all the rows in the table.

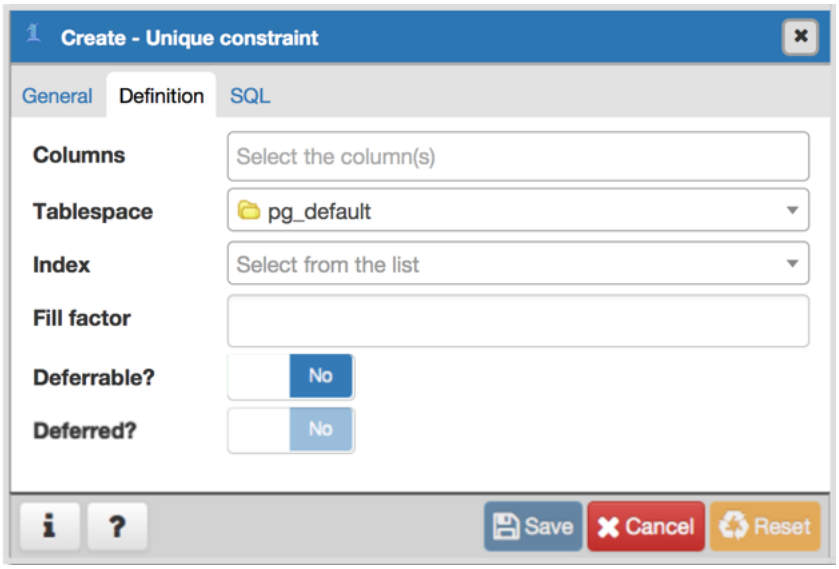
The *Unique constraint* dialog organizes the development of a unique constraint through the following dialog tabs: *General* and *Definition*. The *SQL* tab displays the SQL code generated by dialog selections.



Use the fields in the *General* tab to identify the unique constraint:

- Use the *Name* field to add a descriptive name for the unique constraint. The name will be displayed in the *pgAdmin* tree control.

Click the *Definition* tab to continue.



Use the fields in the *Definition* tab to define the unique constraint:

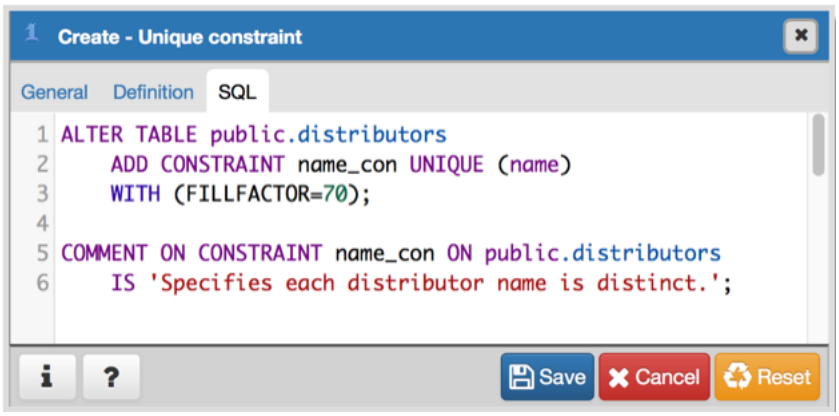
- Click inside the *Columns* field and select one or more column names from the drop-down listbox. To delete a selection, click the *x* to the left of the column name. The unique constraint should be different from the primary key constraint defined for the same table; the selected column(s) for the constraints must be distinct.
- Select the name of the tablespace in which the unique constraint will reside from the drop-down listbox in the *Tablespace* field.
- Select the name of an index from the drop-down listbox in the *Index* field. This field is optional. Adding a unique constraint will automatically create a unique B-tree index on the column or group of columns listed in the constraint, and will force the column(s) to be marked NOT NULL.
- Use the *Fill Factor* field to specify a fill factor for the table and index. The fill factor for a table is a percentage between 10 and 100. 100 (complete packing) is the default.
- Move the *Deferrable?* switch to the *Yes* position to specify the timing of the constraint is deferrable and can be postponed until the end of the statement. The default is *No*.
- If enabled, move the *Deferred?* switch to the *Yes* position to specify the timing of the constraint is deferred to the end of the statement. The default is *No*.

Click the *SQL* tab to continue.

Your entries in the *Unique constraint* dialog generate a SQL command (see an example below). Use the *SQL* tab for review; revisit or switch tabs to make any changes to the SQL command.

Example

The following is an example of the sql command generated by user selections in the *Unique constraint* dialog:



The example shown demonstrates creating a unique constraint named *name_con* on the *name* column of the *distributors* table.

- Click the *Info* button (i) to access online help. View context-sensitive help in the *Tabbed browser*, where a new tab displays the PostgreSQL core documentation.
- Click the *Save* button to save work.
- Click the *Cancel* button to exit without saving work.
- Click the *Reset* button to restore configuration parameters.

CHAPTER

FIVE

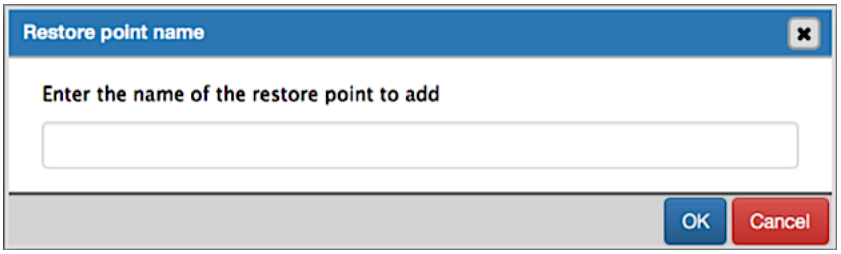
MANAGEMENT BASICS

pgAdmin provides point and click dialogs that help you perform server management functions. Dialogs simplify tasks such as managing named restore points, granting user privileges, and performing VACUUM, ANALYZE and REINDEX functions.

Contents:

5.1 The Add named restore point Dialog

Use the *Add named restore point* dialog to take a named snapshot of the state of the server for use in a recovery file. To create a named restore point, the server's postgresql.conf file must specify a *wal_level* value of *archive*, *hot_standby*, or *logical*. You must be a database superuser to create a restore point.



When the *Restore point name* window launches, use the field *Enter the name of the restore point to add* to provide a descriptive name for the restore point.

For more information about using a restore point as a recovery target, please see the PostgreSQL documentation.

- Click the *OK* button to save the restore point.
- Click the *Cancel* button to exit without saving work.

5.2 The Change Password Dialog

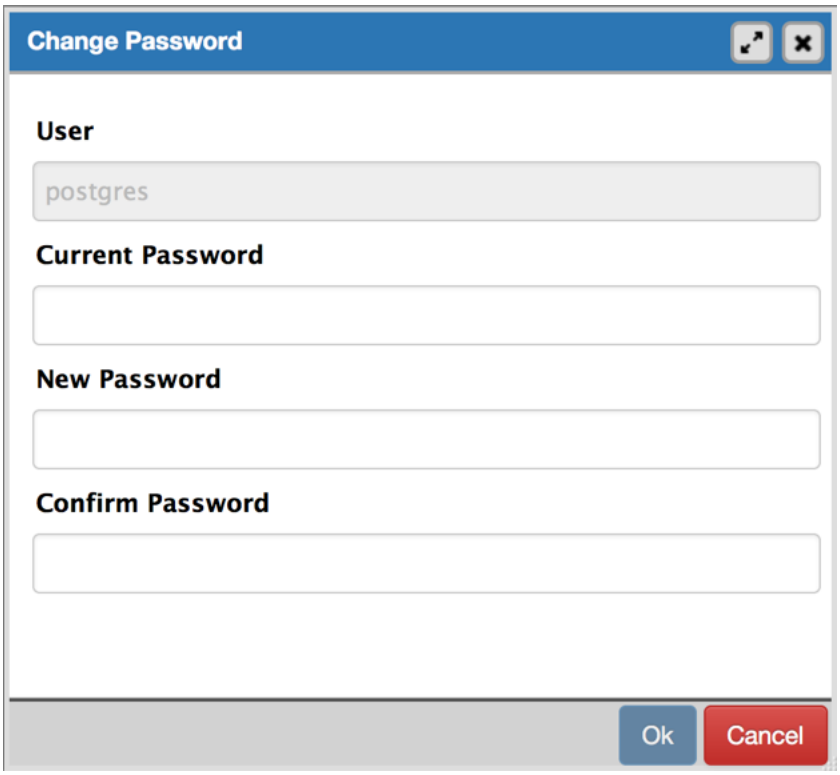
It is a good policy to routinely change your password to protect data, even in what you may consider a 'safe' environment. In the workplace, failure to apply an appropriate password policy could leave you in breach of Data Protection laws.

Please consider the following guidelines when selecting a password:

- Ensure that your password is an adequate length; 6 characters should be the absolute minimum number of characters in the password.

- Ensure that your password is not open to dictionary attacks. Use a mixture of upper and lower case letters and numerics, and avoid words or names. Consider using the first letter from each word in a phrase that you will remember easily but is an unfamiliar acronym.
- Ensure that your password is changed regularly; at minimum, change it every ninety days.

The above should be considered a starting point: It is not a comprehensive list and it **will not guarantee security**.



Use the *Change Password* dialog to change your password:

- The name displayed in the *User* field is the role for which you are modifying the password; it is the role that is associated with the server connection that is highlighted in the tree control.
- Enter the password associated with the role in the *Current Password* field.
- Enter the desired password for in the *New Password* field.
- Re-enter the new password in the *Confirm Password* field.

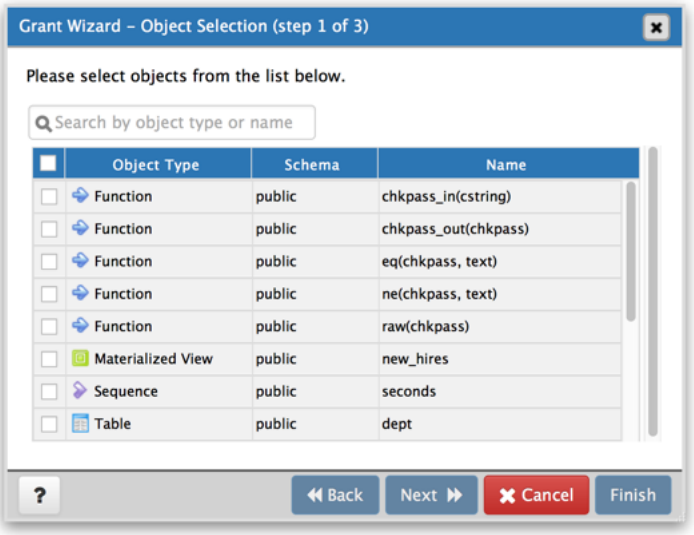
Click the *OK* button to change your password; click *Cancel* to exit the dialog without changing your password.

5.3 Grant Wizard

The *Grant Wizard* tool is a graphical interface that allows you to manage the privileges of one or more database objects in a point-and-click environment. A search box, dropdown lists, and checkboxes facilitate quick selections of database objects, roles and privileges.

The wizard organizes privilege management through a sequence of windows: *Object Selection (step 1 of 3)*, *Privileges Selection (step 2 of 3)* and *Final (Review Selection) (step 3 of 3)*. The *Final (Review Selection)* window displays the SQL code generated by wizard selections.

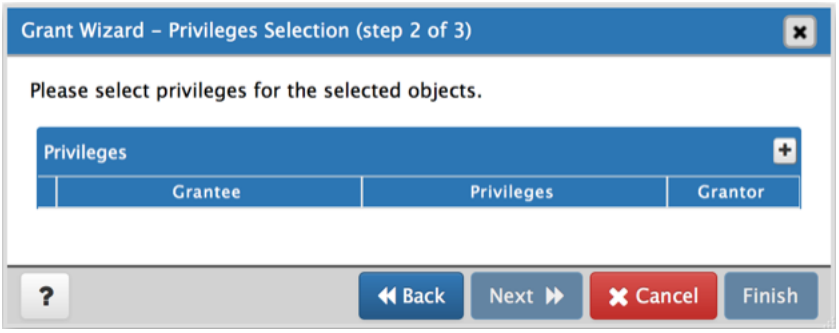
To launch the *Grant Wizard* tool, select a database object in the *pgAdmin* tree control, then navigate through *Tools* on the menu bar to click on the *Grant Wizard* option.



Use the fields in the *Object Selection (step 1 of 3)* window to select the object or objects on which you are modifying privileges. Use the *Search by object type or name* field to locate a database object, or use the scrollbar to scroll through the list of all accessible objects.

- Each row in the table lists object identifiers; check the checkbox in the left column to include an object as a target of the Grant Wizard. The table displays:
  - The object type in the *Object Type* field
  - The schema in which the object resides in the *Schema* field
  - The object name in the *Name* field.

Click the *Next* button to continue, or the *Cancel* button to close the wizard without modifying privileges.



Use the fields in the *Privileges Selection (step 2 of 3)* window to grant privileges. If you grant a privilege WITH GRANT OPTION, the Grantee will have the right to grant privileges on the object to others. If WITH GRANT OPTION is subsequently revoked, any role who received access to that object from that Grantee (directly or through a chain of grants) will lose thier privileges on the object.

- Click the *Add* icon (+) to assign a set of privileges.
- Select the name of the role from the drop-down listbox in the *Grantee* field.
- Click inside the *Privileges* field. Check the boxes to the left of one or more privileges to grant the selected privileges to the specified user. If privileges have previously been granted on a database object, unchecking a

privilege for a group or user will result in revoking that privilege.

- If enabled, select the name of the role from the drop-down listbox in the *Grantor* field. The default grantor is the owner of the database.
- Click the *Add* icon (+) to assign a set of privileges to another role; to discard a privilege, click the trash icon to the left of the row and confirm deletion in the *Delete Row* dialog.

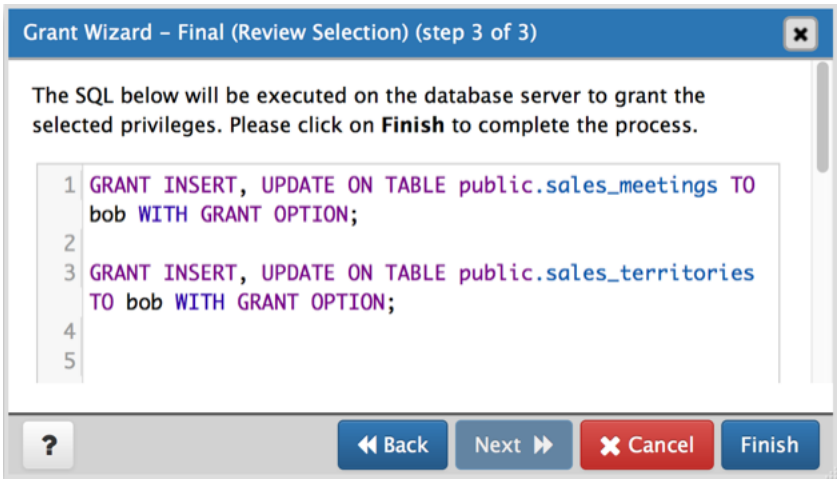
For more information about granting privileges on database objects, see the PostgreSQL core documentation.

Click the *Next* button to continue, the *Back* button to select or deselect additional database objects, or the *Cancel* button to close the wizard without modifying privileges.

Your entries in the *Grant Wizard* tool generate a SQL command; you can review the command in the *Final (Review Selection) (step 3 of 3)* window (see an example below).

Example

The following is an example of the sql command generated by user selections in the *Grant Wizard* tool:



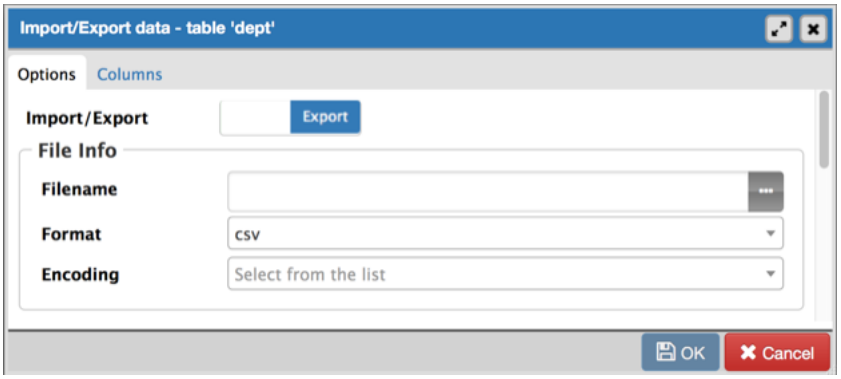
The commands displayed assign a role named *Bob INSERT* and *UPDATE* privileges *WITH GRANT OPTION* on the *sales_meetings* and the *sales_territories* tables.

- Click the *Back* button to select or deselect additional database objects, roles and privileges.
- Click the *Cancel* button to exit without saving work.
- Click the *Finish* button to save selections and exit the wizard.

5.4 The Import/Export data Dialog

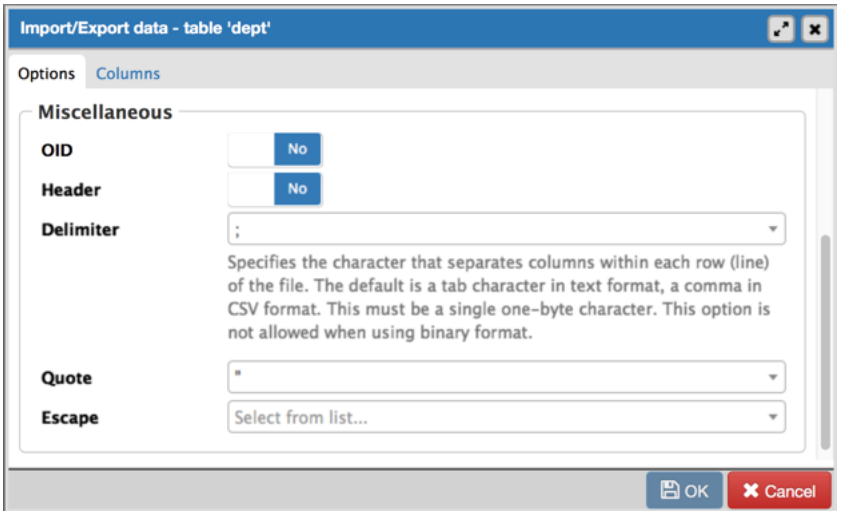
Use the *Import/Export data* dialog to copy data from a table to a file, or copy data from a file into a table.

The *Import/Export data* dialog organizes the import/export of data through the *Options* and *Columns* tabs.



Use the fields in the *Options* tab to specify import and export preferences:

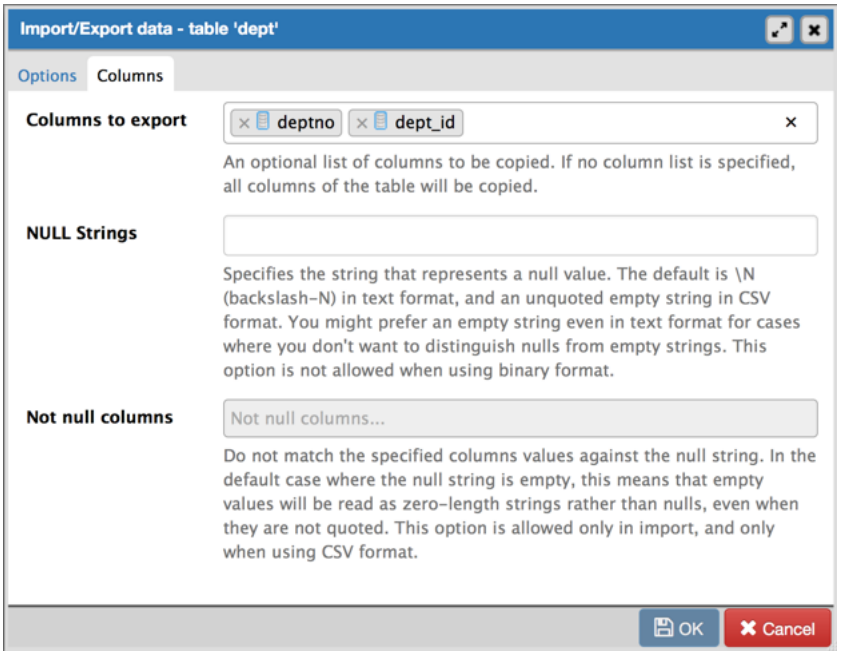
- Move the *Import/Export* switch to the *Import* position to specify that the server should import data to a table from a file. The default is *Export*.
- Use the fields in the *File Info* field box to specify information about the source or target file:
  - Enter the name of the source or target file in the *Filename* field. Optionally, select the *Browser* icon (ellipsis) to the right to navigate into a directory and select a file.
  - Use the drop-down listbox in the *Format* field to specify the file type. Select:
    - *binary* for a .bin file.
    - *csv* for a .csv file.
    - *text* for a .txt file.
  - Use the drop-down listbox in the *Encoding* field to specify the type of character encoding.



- Use the fields in the *Miscellaneous* field box to specify additional information:
  - Move the *OID* switch to the *Yes* position to include the *OID* column. The *OID* is a system-assigned value that may not be modified. The default is *No*.
  - Move the *Header* switch to the *Yes* position to include the table header with the data rows. If you include the table header, the first row of the file will contain the column names.
  - If you are exporting data, specify the delimiter that will separate the columns within the target file in the *Delimiter* field. The separating character can be a colon, semicolon, a vertical bar, or a tab.

- Specify a quoting character used in the *Quote* field. Quoting can be applied to string columns only (i.e. numeric columns will not be quoted) or all columns regardless of data type. The character used for quoting can be a single quote or a double quote.
- Specify a character that should appear before a data character that matches the *QUOTE* value in the *Escape* field.

Click the *Columns* tab to continue.



Use the fields in the *Columns* tab to select the columns that will be imported or exported:

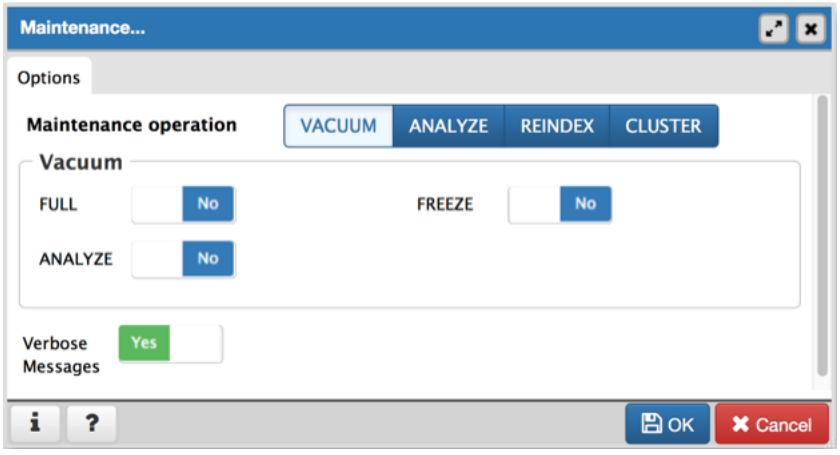
- Click inside the *Columns to export/import* field to deselect one or more columns from the drop-down listbox. To delete a selection, click the *x* to the left of the column name. Click an empty spot inside the field to access the drop-down list.
- Use the *NULL Strings* field to specify a string that will represent a null value within the source or target file.
- If enabled, click inside the *Not null columns* field to select one or more columns that will not be checked for a NULL value. To delete a column, click the *x* to the left of the column name.

After completing the *Import/Export data* dialog, click the *OK* button to perform the import or export. pgAdmin will inform you when the background process completes:

Use the *Click here for details* link on the notification to open the *Process Watcher* and review detailed information about the execution of the command that performed the import or export:

5.5 The Maintenance Dialog

Use the *Maintenance* dialog to VACUUM, ANALYZE, REINDEX or CLUSTER a database or selected database objects.



While this utility is useful for ad-hoc maintenance purposes, you are encouraged to perform automatic VACUUM jobs on a regular schedule.

Select a button next to *Maintenance operation* to specify the type of maintenance:

- Click *VACUUM* to scan the selected database or table to reclaim storage used by dead tuples.
  - Move the *FULL* switch to the *Yes* position to compact tables by writing a completely new version of the table file without dead space. The default is *No*.

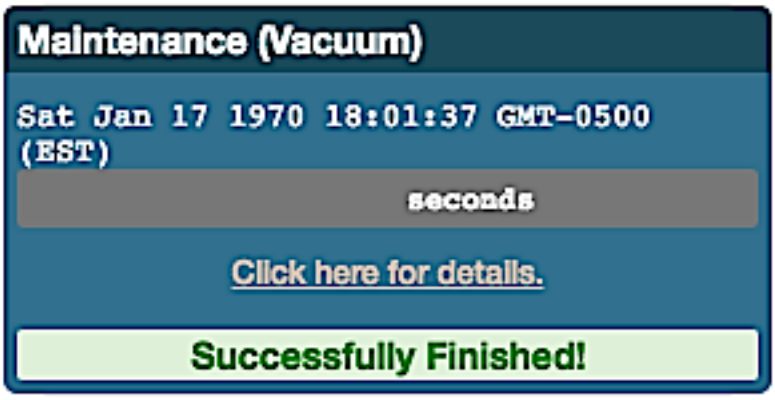
  - Move the *FREEZE* switch to the *Yes* position to freeze data in a table when it will have no further updates. The default is *No*.
  - Move the *ANALYZE* switch to the *Yes* position to issue ANALYZE commands whenever the content of a table has changed sufficiently. The default is *No*.

- Click *ANALYZE* to update the stored statistics used by the query planner. This enables the query optimizer to select the fastest query plan for optimal performance.
- Click *REINDEX* to rebuild any index in case it has degenerated due to the insertion of unusual data patterns. This happens, for example, if you insert rows with increasing index values, and delete low index values.
- Click *CLUSTER* to instruct PostgreSQL to cluster the selected table.

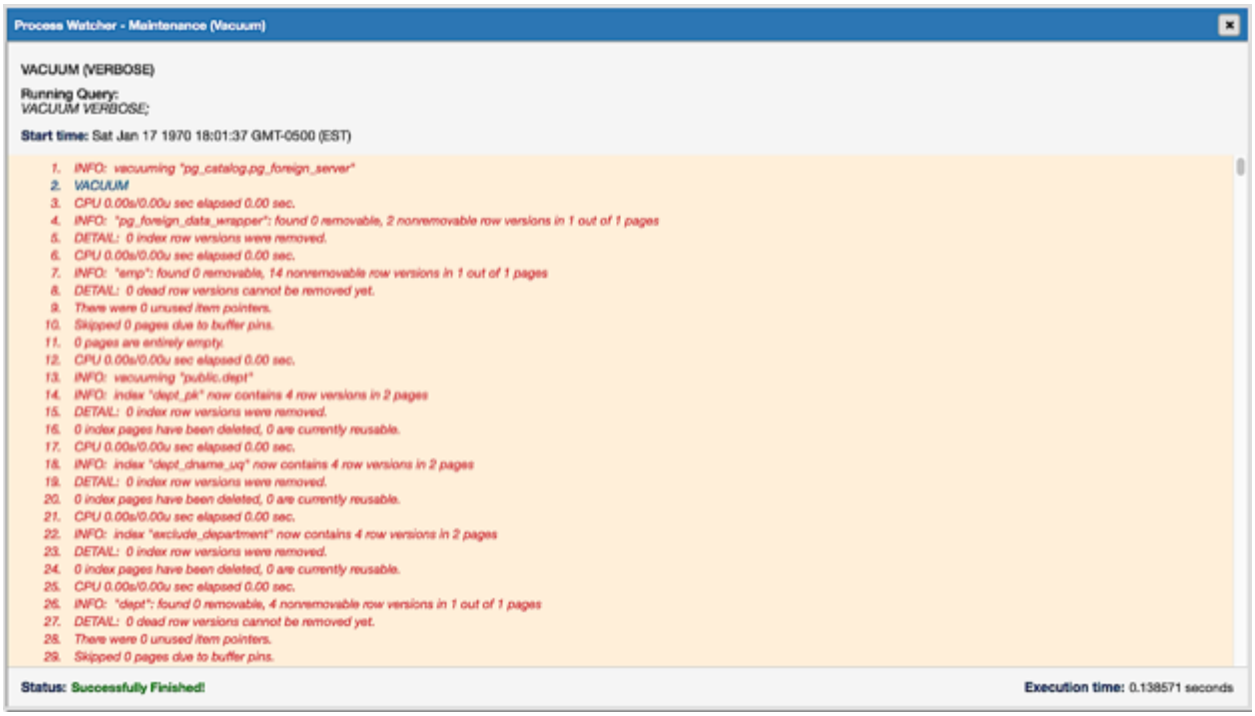
To exclude status messages from the process output, move the *Verbose Messages* switch to the *No* position; by default, status messages are included.

When you've completed the dialog, click *OK* to start the background process; to exit the dialog without performing maintenance operations, click *Cancel*.

pgAdmin will inform you when the background process completes:



Use the *Click here for details* link on the notification to open the *Process Watcher* and review detailed information about the execution of the command that performed the import or export:


Process Watcher - Maintenance (Vacuum)
VACUUM (VERBOSE)
Running Query:
VACUUM VERBOSE;
Start time: Sat Jan 17 1970 18:01:37 GMT-0500 (EST)
1. INFO: vacuuming "pg_catalog.pg_foreign_server"
2. VACUUM
3. CPU 0.00s/0.00u sec elapsed 0.00 sec.
4. INFO: "pg_foreign_data_wrapper": found 0 removable, 2 nonremovable row versions in 1 out of 1 pages
5. DETAIL: 0 index row versions were removed.
6. CPU 0.00s/0.00u sec elapsed 0.00 sec.
7. INFO: "emp": found 0 removable, 14 nonremovable row versions in 1 out of 1 pages
8. DETAIL: 0 dead row versions cannot be removed yet.
9. There were 0 unused item pointers.
10. Skipped 0 pages due to buffer pins.
11. 0 pages are entirely empty.
12. CPU 0.00s/0.00u sec elapsed 0.00 sec.
13. INFO: vacuuming "public.dept"
14. INFO: index "dept_pk" now contains 4 row versions in 2 pages
15. DETAIL: 0 index row versions were removed.
16. 0 index pages have been deleted, 0 are currently reusable.
17. CPU 0.00s/0.00u sec elapsed 0.00 sec.
18. INFO: index "dept_dname_uq" now contains 4 row versions in 2 pages
19. DETAIL: 0 index row versions were removed.
20. 0 index pages have been deleted, 0 are currently reusable.
21. CPU 0.00s/0.00u sec elapsed 0.00 sec.
22. INFO: index "exclude_department" now contains 4 row versions in 2 pages
23. DETAIL: 0 index row versions were removed.
24. 0 index pages have been deleted, 0 are currently reusable.
25. CPU 0.00s/0.00u sec elapsed 0.00 sec.
26. INFO: "dept": found 0 removable, 4 nonremovable row versions in 1 out of 1 pages
27. DETAIL: 0 dead row versions cannot be removed yet.
28. There were 0 unused item pointers.
29. Skipped 0 pages due to buffer pins.
Status: Successfully Finished!
Execution time: 0.138571 seconds

CHAPTER

SIX

BACKUP AND RESTORE

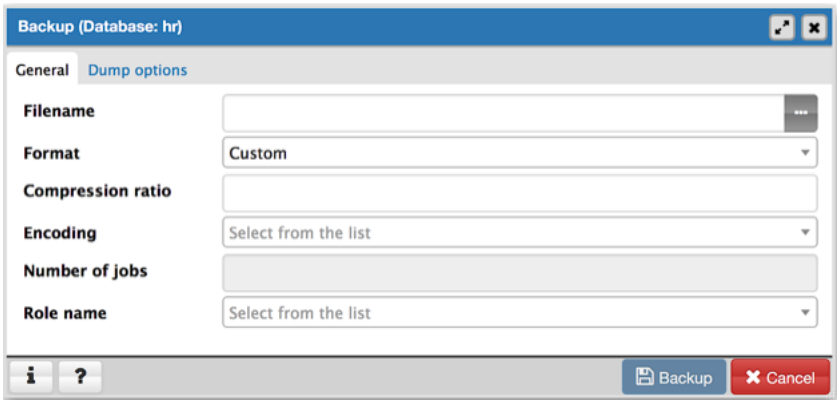
A powerful, but user-friendly Backup and Restore tool provides an easy way to use pg_dump, pg_dumpall, and pg_restore to take backups and create copies of databases or database objects for use in a development environment.

Contents:

6.1 The Backup Dialog

Using the *pg_dump* utility, *pgAdmin* provides an easy way to create a backup in a plain-text or archived format. You can then use a client application (like *psql* or the *Query Tool*) to restore a plain-text backup file, or use the Postgres *pg_restore* utility to restore an archived backup. The *pg_dump* utility must have read access to all database objects that you want to back up.

You can backup a single table, a schema, or a complete database. Select the name of the backup source in the *pgAdmin* tree control, right click to open the context menu, and select *Backup...* to open the *Backup* dialog. The name of the object selected will appear in the dialog title bar.



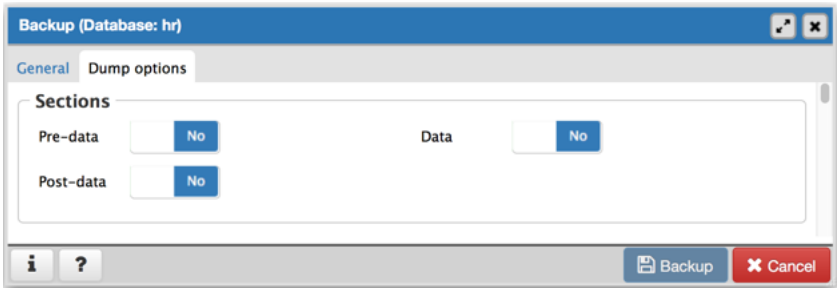
Use the fields in the *General* tab to specify parameters for the backup:

- Enter the name of the backup file in the *Filename* field. Optionally, select the *Browser* icon (...) to the right to navigate into a directory and select a file that will contain the archive.
- Use the drop-down listbox in the *Format* field to select the format that is best suited for your application. Each format has advantages and disadvantages:
  - Select *Custom* to create a custom archive file that you can use with *pg_restore* to create a copy of a database. Custom archive file formats must be restored with *pg_restore*. This format offers the opportunity to select which database objects to restore from the backup file. *Custom* archive format is recommended for medium to large databases as it is compressed by default.

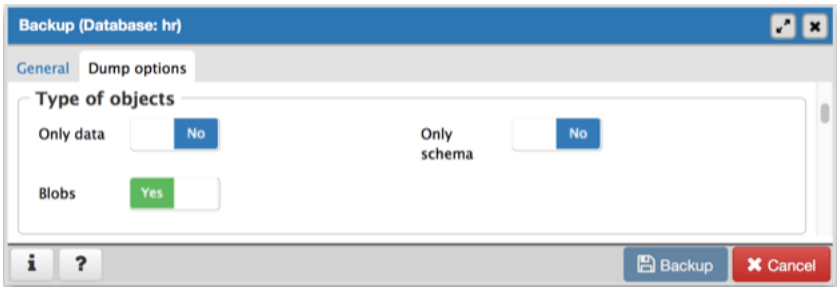
  - Select *Tar* to generate a tar archive file that you can restore with *pg_restore*. The tar format does not support compression.
  - Select *Plain* to create a plain-text script file. A plain-text script file contains SQL statements and commands that you can execute at the *psql* command line to recreate the database objects and load the table data. A plain-text backup file can be edited in a text editor, if desired, before using the *psql* program to restore database objects. *Plain* format is normally recommended for smaller databases; script dumps are not recommended for blobs. The SQL commands within the script will reconstruct the database to the last saved state of the database. A plain-text script can be used to reconstruct the database on another machine, or (with modifications) on other architectures.
  - Select *Directory* to generate a directory-format archive suitable for use with *pg_restore*. This file format creates a directory with one file for each table and blob being dumped, plus a *Table of Contents* file describing the dumped objects in a machine-readable format that *pg_restore* can read. This format is compressed by default.

- Use the *Compression Ratio* field to select a compression level for the backup. Specify a value of zero to mean use no compression; specify a maximum compression value of 9. Please note that tar archives do not support compression.
- Use the *Encoding* drop-down listbox to select the character encoding method that should be used for the archive.
- Use the *Number of Jobs* field (when applicable) to specify the number of tables that will be dumped simultaneously in a parallel backup.
- Use the dropdown listbox next to *Rolename* to specify the role that owns the backup.

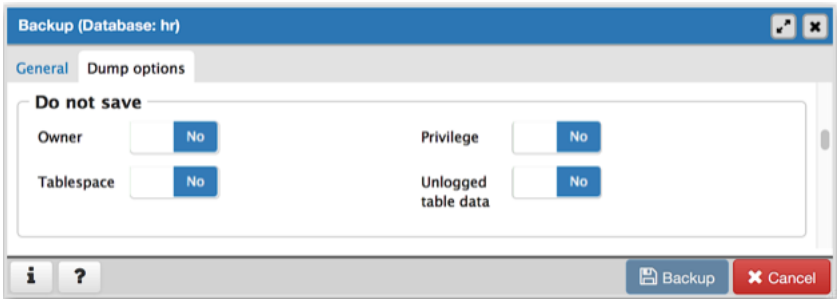
Click the *Dump options* tab to continue. Use the box fields in the *Dump options* tab to provide options for *pg_dump*.



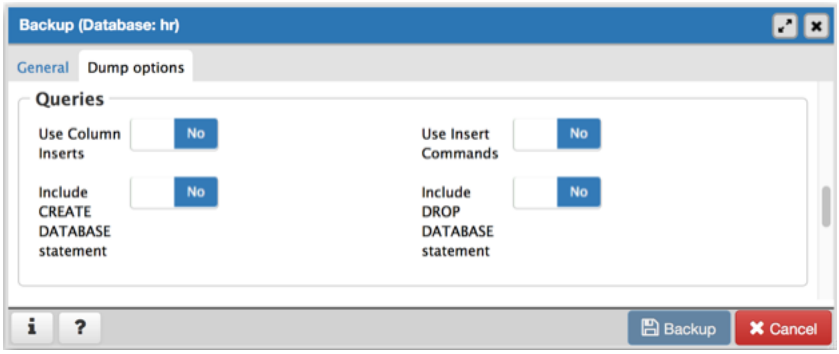
- Move switches in the **Sections** field box to select a portion of the object that will be backed up.
  - Move the switch next to *Pre-data* to the *Yes* position to include all data definition items not included in the data or post-data item lists.
  - Move the switch next to *Data* to the *Yes* position to backup actual table data, large-object contents, and sequence values.
  - Move the switch next to *Post-data* to the *Yes* position to include definitions of indexes, triggers, rules, and constraints other than validated check constraints.



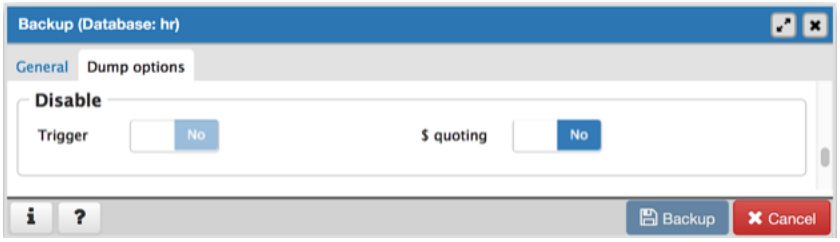
- Move switches in the **Type of objects** field box to specify details about the type of objects that will be backed up.
  - Move the switch next to *Only data* to the *Yes* position to limit the back up to data.
  - Move the switch next to *Only schema* to limit the back up to schema-level database objects.
  - Move the switch next to *Blobs* to the *No* position to exclude large objects in the backup.



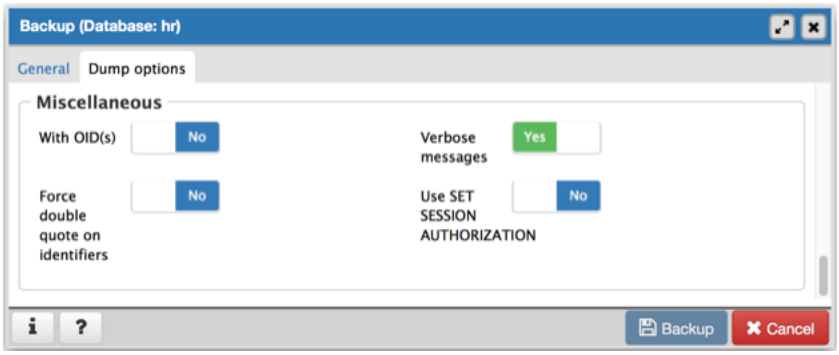
- Move switches in the **Do not save** field box to select the objects that will not be included in the backup.
  - Move the switch next to *Owner* to the *Yes* position to include commands that set object ownership.
  - Move the switch next to *Privilege* to the *Yes* position to include commands that create access privileges.
  - Move the switch next to *Tablespace* to the *Yes* position to include tablespaces.
  - Move the switch next to *Unlogged table data* to the *Yes* position to include the contents of unlogged tables.



- Move switches in the **Queries** field box to specify the type of statements that should be included in the backup.
  - Move the switch next to *Use Column Inserts* to the *Yes* position to dump the data in the form of INSERT statements and include explicit column names. Please note: this may make restoration from backup slow.
  - Move the switch next to *Use Insert commands* to the *Yes* position to dump the data in the form of INSERT statements rather than using a COPY command. Please note: this may make restoration from backup slow.
  - Move the switch next to *Include CREATE DATABASE statement* to the *Yes* position to include a command in the backup that creates a new database when restoring the backup.
  - Move the switch next to *Include DROP DATABASE statement* to the *Yes* position to include a command in the backup that will drop any existing database object with the same name before recreating the object during a backup.



- Move switches in the **Disable** field box to specify the type of statements that should be excluded from the backup.
  - Move the switch next to *Trigger* (active when creating a data-only backup) to the *Yes* position to include commands that will disable triggers on the target table while the data is being loaded.
  - Move the switch next to *$ quoting* to the *Yes* position to enable dollar quoting within function bodies; if disabled, the function body will be quoted using SQL standard string syntax.

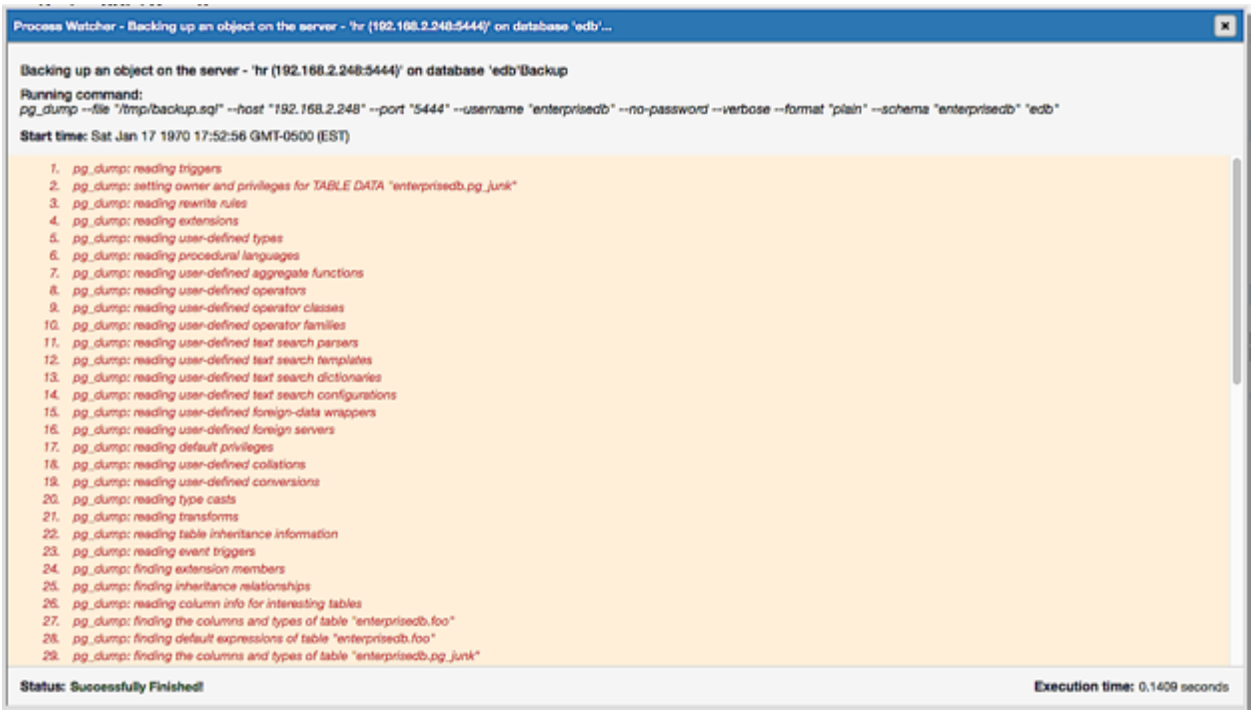


- Move switches in the **Miscellaneous** field box to specify miscellaneous backup options.
  - Move the switch next to *With OIDs* to the *Yes* position to include object identifiers as part of the table data for each table.
  - Move the switch next to *Verbose messages* to the *No* position to instruct *pg_dump* to exclude verbose messages.
  - Move the switch next to *Force double quotes on identifiers* to the *Yes* position to force the quoting of all identifiers.
  - Move the switch next to *Use SET SESSION AUTHORIZATION* to the *Yes* position to include a statement that will use a SET SESSION AUTHORIZATION command to determine object ownership (instead of an ALTER OWNER command).

When you've specified the details that will be incorporated into the pg_dump command:

- Click the *Backup* button to build and execute a command that builds a backup based on your selections on the *Backup* dialog.
- Click the *Cancel* button to exit without saving work.

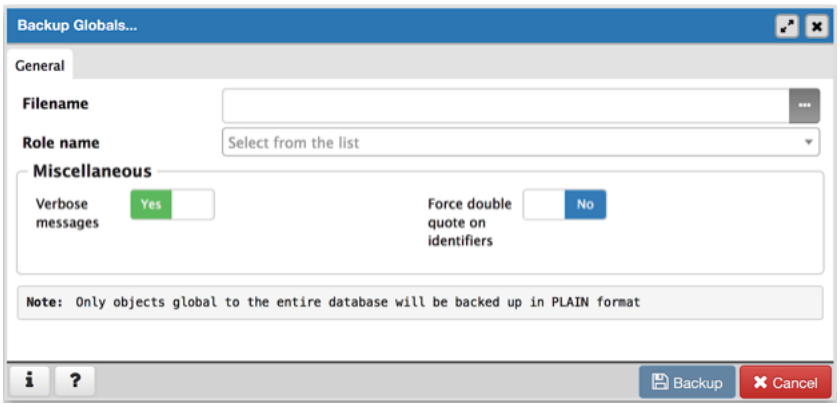
If the backup is successful, a popup window will confirm success. Click *Click here for details* on the popup window to launch the *Process Watcher*. The *Process Watcher* logs all the activity associated with the backup and provides additional information for troubleshooting.



If the backup is unsuccessful, you can review the error messages returned by the backup command on the *Process Watcher*.

6.2 The Backup Globals Dialog

Use the *Backup Globals* dialog to create a plain-text script that recreates all of the database objects within a cluster, and the global objects that are shared by those databases. Global objects include tablespaces, roles, and object properties. You can use the pgAdmin *Query Tool* to play back a plain-text script, and recreate the objects in the backup.



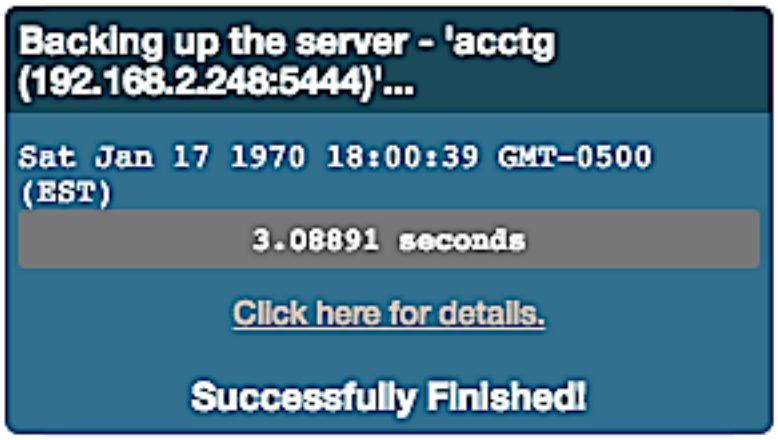
Use the fields in the *General* tab to specify the following:

- Enter the name of the backup file in the *Filename* field. Optionally, select the *Browser* icon (ellipsis) to the right to navigate into a directory and select a file that will contain the archive.
- Use the drop-down listbox next to *Role name* to specify a role with connection privileges on the selected server. The role will be used for authentication during the backup.

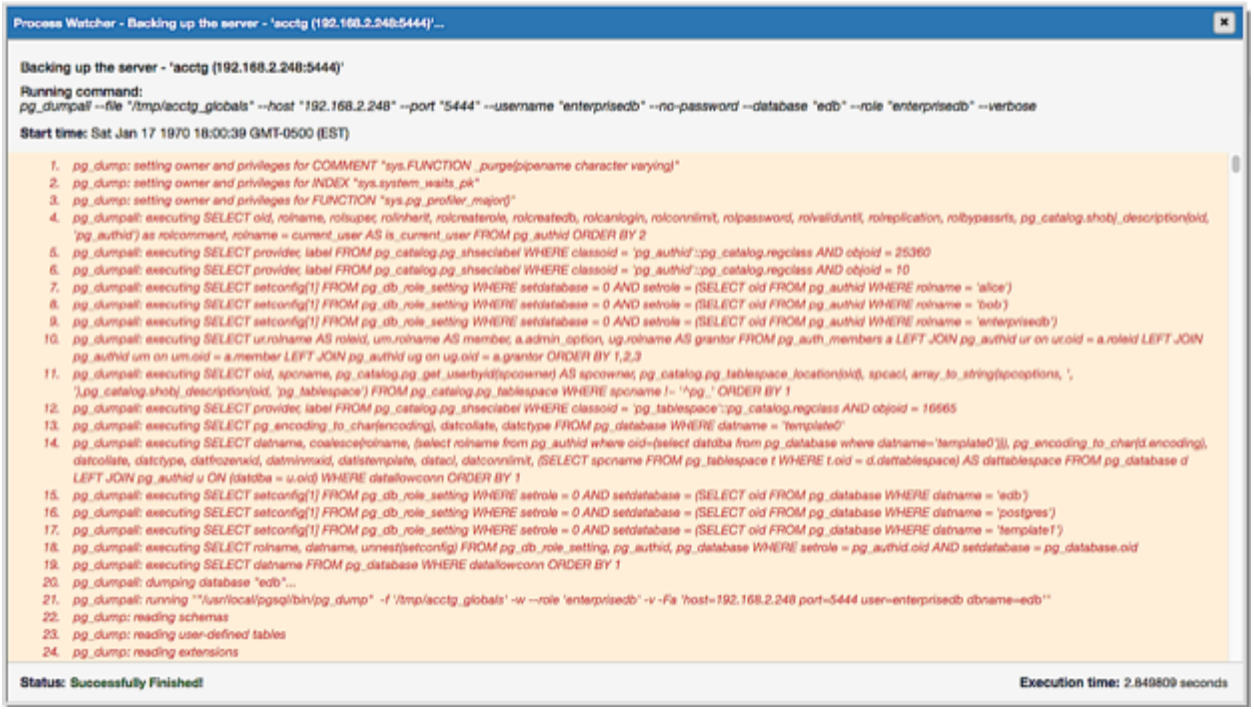
Move switches in the **Miscellaneous** field box to specify the type of statements that should be included in the backup.

- Move the *Verbose messages* switch to the *No* position to exclude status messages from the backup. The default is *Yes*.
- Move the *Force double quote on identifiers* switch to the *Yes* position to name identifiers without changing case. The default is *No*.

Click the *Backup* button to build and execute a command based on your selections; click the *Cancel* button to exit without saving work.



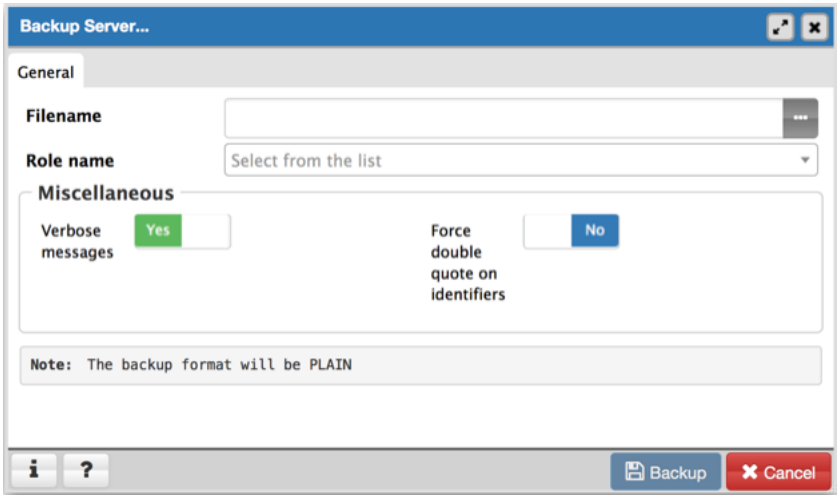
If the backup is successful, a popup window will confirm success. Click *Click here for details* on the popup window to launch the *Process Watcher*. The *Process Watcher* logs all the activity associated with the backup and provides additional information for troubleshooting.



If the backup is unsuccessful, review the error message returned by the *Process Watcher* to resolve any issue.

6.3 The Backup Server Dialog

Use the *Backup Server* dialog to create a plain-text script that will recreate the selected server. You can use the pgAdmin *Query Tool* to play back a plain-text script, and recreate the server.



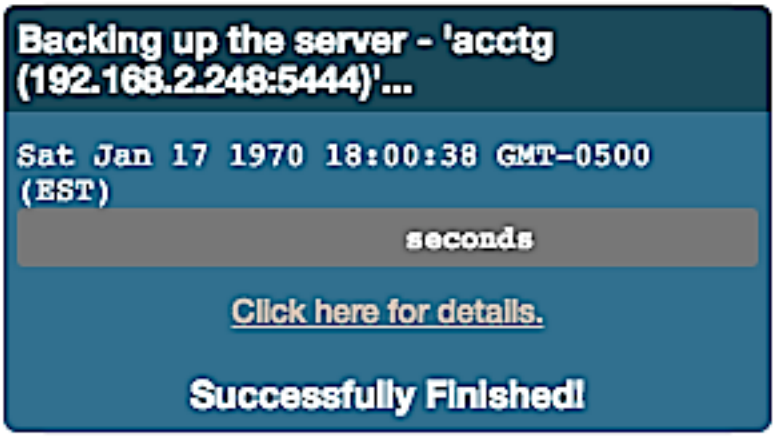
Use the fields in the *General* tab to specify the following:

- Enter the name of the backup file in the *Filename* field. Optionally, select the *Browser* icon (ellipsis) to the right to navigate into a directory and select a file that will contain the archive.
- Use the drop-down listbox next to *Role name* to specify a role with connection privileges on the selected server. The role will be used for authentication during the backup.

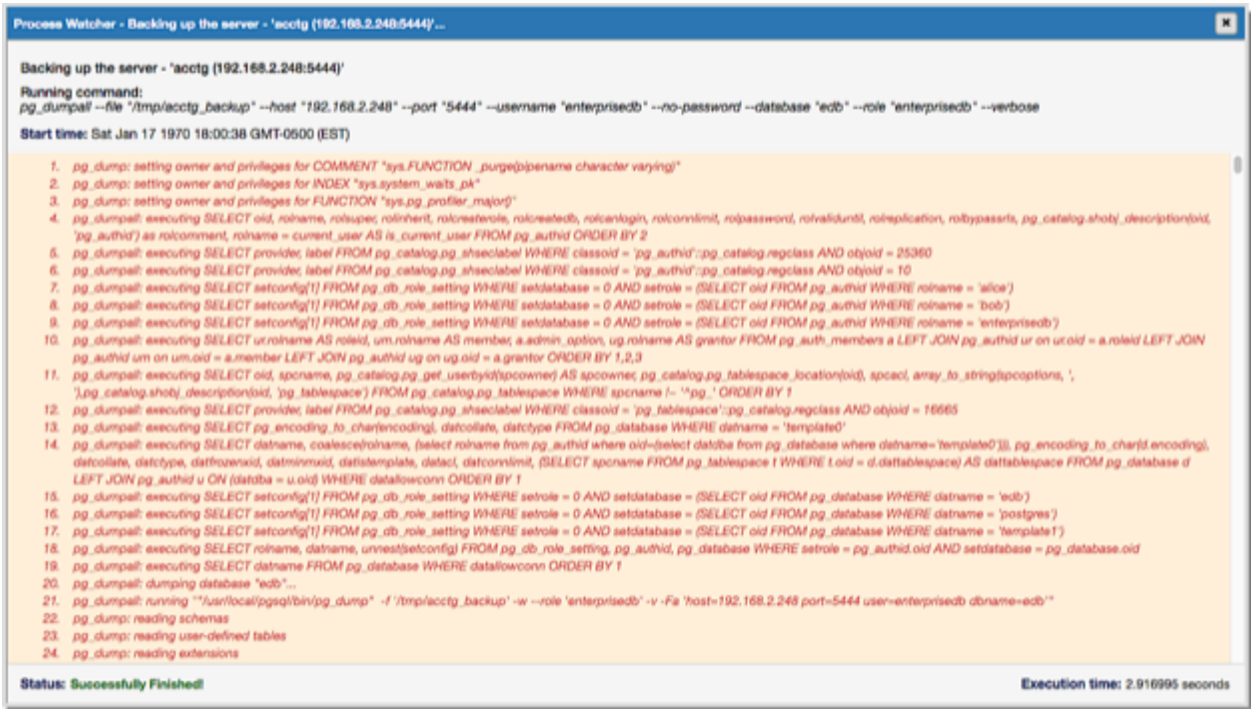
Move switches in the *Miscellaneous* box to specify the type of statements that should be included in the backup.

- Move the *Verbose messages* switch to the *No* position to exclude status messages from the backup. The default is *Yes*.
- Move the *Force double quote on identifiers* switch to the *Yes* position to name identifiers without changing case. The default is *No*.

Click the *Backup* button to build and execute a command based on your selections; click the *Cancel* button to exit without saving work.



If the backup is successful, a popup window will confirm success. Click *Click here for details* on the popup window to launch the *Process Watcher*. The *Process Watcher* logs all the activity associated with the backup and provides additional information for troubleshooting.

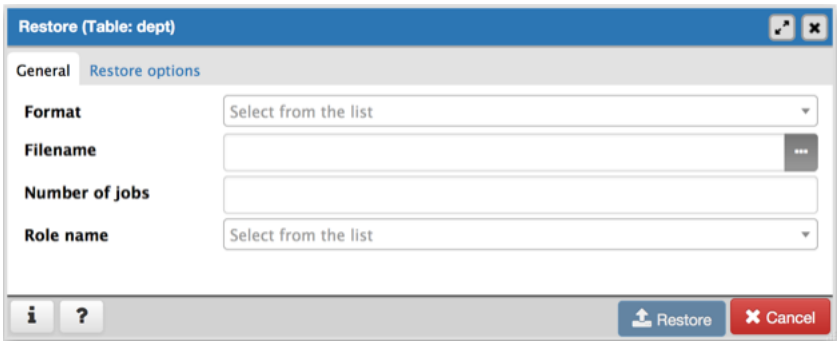


If the backup is unsuccessful, review the error message returned by the *Process Watcher* to resolve any issue.

6.4 The Restore Dialog

The *Restore* dialog provides an easy way to use a Custom, tar, or Directory format backup taken with the pgAdmin *Backup* dialog to recreate a database or database object. The *Backup* dialog invokes options of the pg_dump client utility; the *Restore* dialog invokes options of the pg_restore client utility.

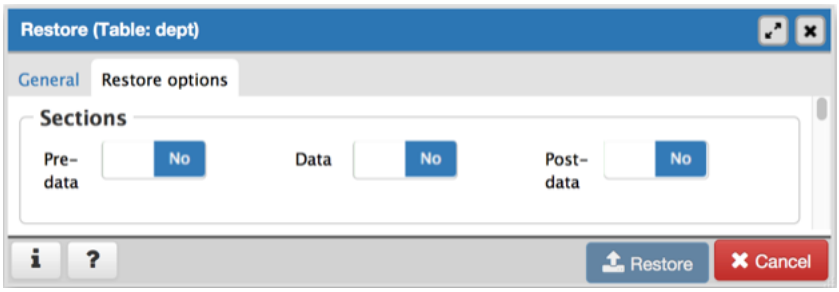
You can use the *Query Tool* to play back the script created during a plain-text backup made with the *Backup* dialog. For more information about backing up or restoring, please refer to the documentation for pg_dump or pg_restore.



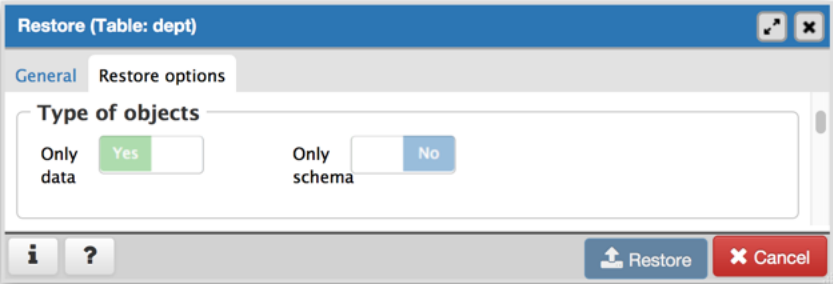
Use the fields on the *General* tab to specify general information about the restore process:

- Use the drop-down listbox in the *Format* field to select the format of your backup file.
  - Select *Custom or tar* to restore from a custom archive file to create a copy of the backed-up object.
  - Select *Directory* to restore from a compressed directory-format archive.
- Enter the complete path to the backup file in the *Filename* field. Optionally, select the *Browser* icon (ellipsis) to the right to navigate into a directory and select the file that contains the archive.
- Use the *Number of Jobs* field to specify if pg_restore should use multiple (concurrent) jobs to process the restore. Each job uses a separate connection to the server.
- Use the drop-down listbox next to *Rolename* to specify the role that will be used to authenticate with the server during the restore process.

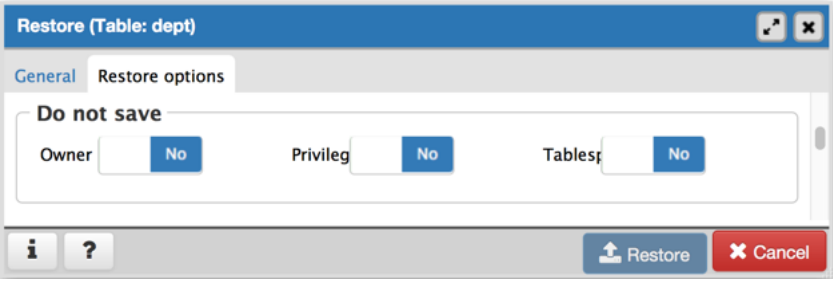
Click the *Restore options* tab to continue. Use the fields on the *Restore options* tab to specify options that correspond to *pg_restore* options.



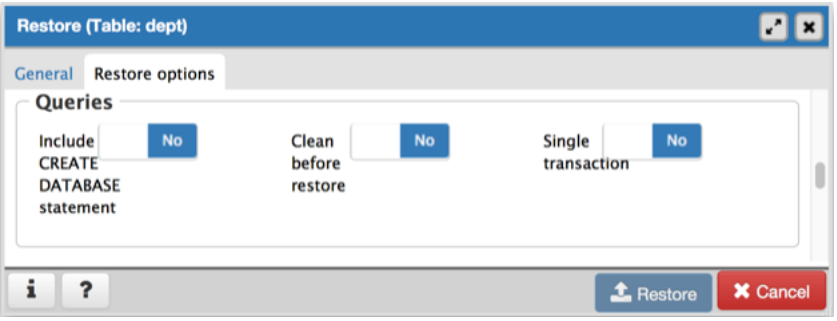
- Use the switches in the **Sections** box to specify the content that will be restored:
  - Move the switch next to *Pre-data* to the *Yes* position to restore all data definition items not included in the data or post-data item lists.
  - Move the switch next to *Data* to the *Yes* position to restore actual table data, large-object contents, and sequence values.
  - Move the switch next to *Post-data* to the *Yes* position to restore definitions of indexes, triggers, rules, and constraints (other than validated check constraints).



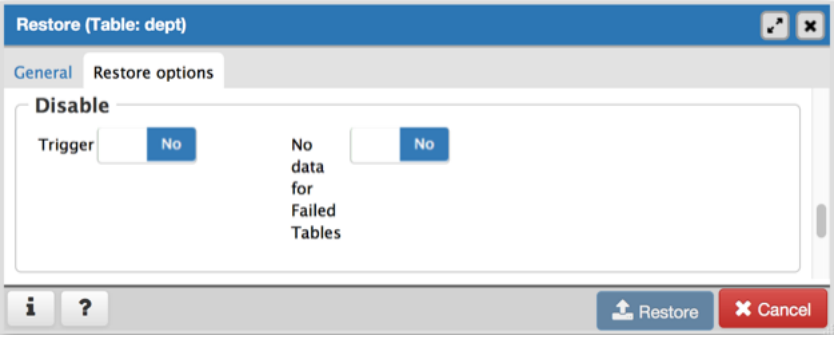
- Use the switches in the **Type of objects** box to specify the objects that will be restored:
  - Move the switch next to *Only data* to the *Yes* position to limit the restoration to data.
  - Move the switch next to *Only schema* to limit the restoration to schema-level database objects.



- Use the switches in the **Do not save** box to specify which objects will not be restored:
  - Move the switch next to *Owner* to the *Yes* position to exclude commands that set object ownership.
  - Move the switch next to *Privilege* to the *Yes* position to exclude commands that create access privileges.
  - Move the switch next to *Tablespace* to the *Yes* position to exclude tablespaces.



- Use the switches in the **Queries** box to specify the type of statements that should be included in the restore:
  - Move the switch next to *Include CREATE DATABASE statement* to the *Yes* position to include a command that creates a new database before performing the restore.
  - Move the switch next to *Clean before restore* to the *Yes* position to drop each existing database object (and data) before restoring.
  - Move the switch next to *Single transaction* to the *Yes* position to execute the restore as a single transaction (that is, wrap the emitted commands in *BEGIN/COMMIT*). This ensures that either all the commands complete successfully, or no changes are applied. This option implies *–exit-on-error*.

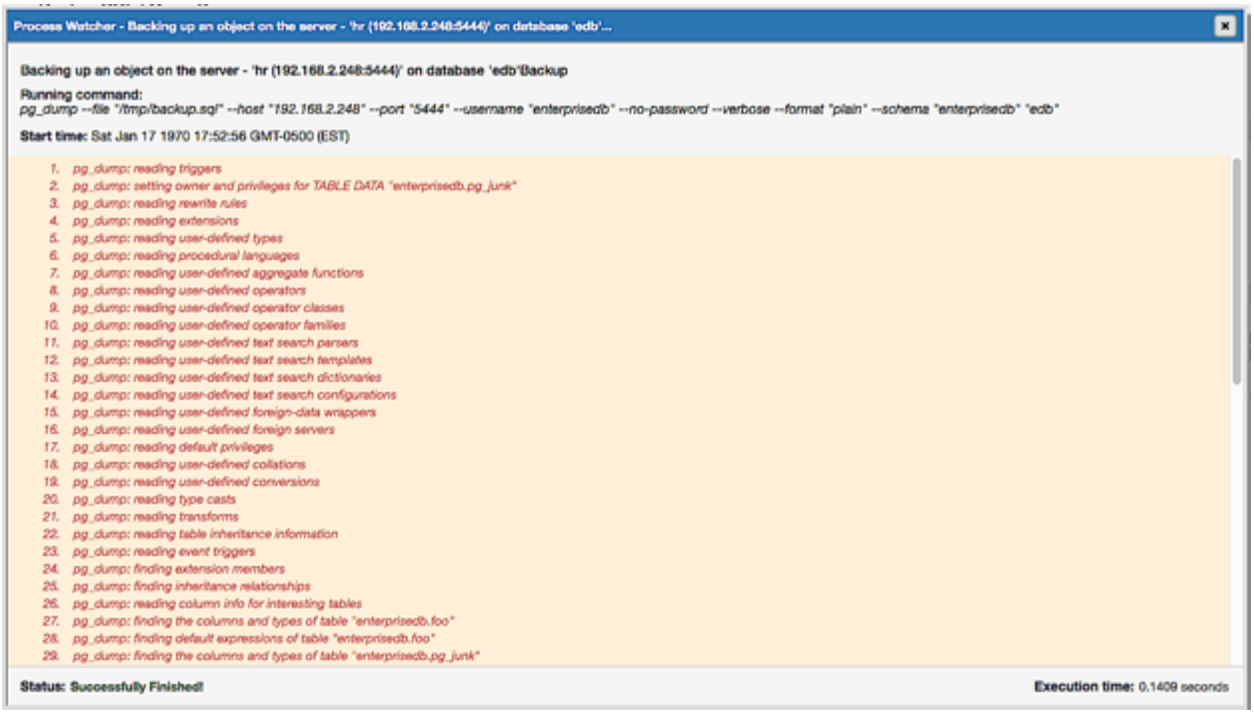


- Use the switches in the **Disable** box to specify the type of statements that should be excluded from the restore:
  - Move the switch next to *Trigger* (active when creating a data-only restore) to the *Yes* position to include commands that will disable triggers on the target table while the data is being loaded.
  - Move the switch next to *No data for Failed Tables* to the *Yes* position to ignore data that fails a trigger.

- Use the switches in the **Miscellaneous/Behavior** box to specify miscellaneous restore options:
  - Move the switch next to *Verbose messages* to the *No* position to instruct *pg_restore* to exclude verbose messages.
  - Move the switch next to *Use SET SESSION AUTHORIZATION* to the *Yes* position to include a statement that will use a SET SESSION AUTHORIZATION command to determine object ownership (instead of an ALTER OWNER command).
  - Move the switch next to *Exit on error* to the *Yes* position to instruct *pg_restore* to exit restore if there is an error in sending SQL commands. The default is to continue and to display a count of errors at the end of the restore.

When you've specified the details that will be incorporated into the pg_restore command, click the *Restore* button to start the process, or click the *Cancel* button to exit without saving your work. A popup will confirm if the restore is successful.

Click *Click here for details* on the popup to launch the *Process Watcher*. The *Process Watcher* logs all the activity associated with the restore, and provides additional information for troubleshooting should the restore command encounter problems.



CHAPTER

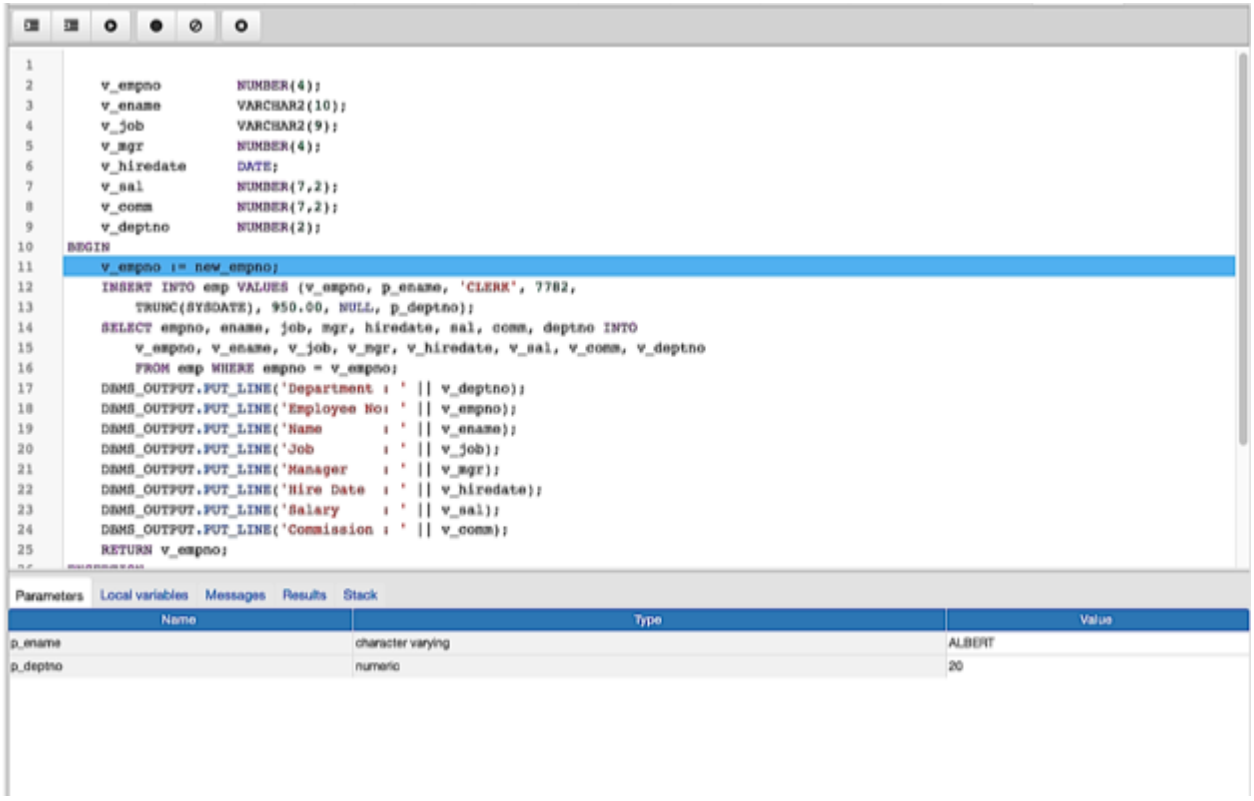
SEVEN

DEVELOPER TOOLS

The pgAdmin *Tools* menu displays a list of powerful developer tools that you can use to execute and analyze complex SQL commands, manage data, and debug PL/SQL code.

Contents:

7.1 pgAdmin Debugger



The debugger may be used to debug PL/pgSQL functions in PostgreSQL, as well as EDB-SPL functions, stored procedures and packages in Advanced Server. The Debugger is available as an extension for your PostgreSQL installation, and is distributed as part of Advanced Server. You must have superuser privileges to use the debugger.

Before using the debugger, you must modify the *postgresql.conf* file, adding the server-side debugger components to the the value of the *shared_preload_libraries* parameter:

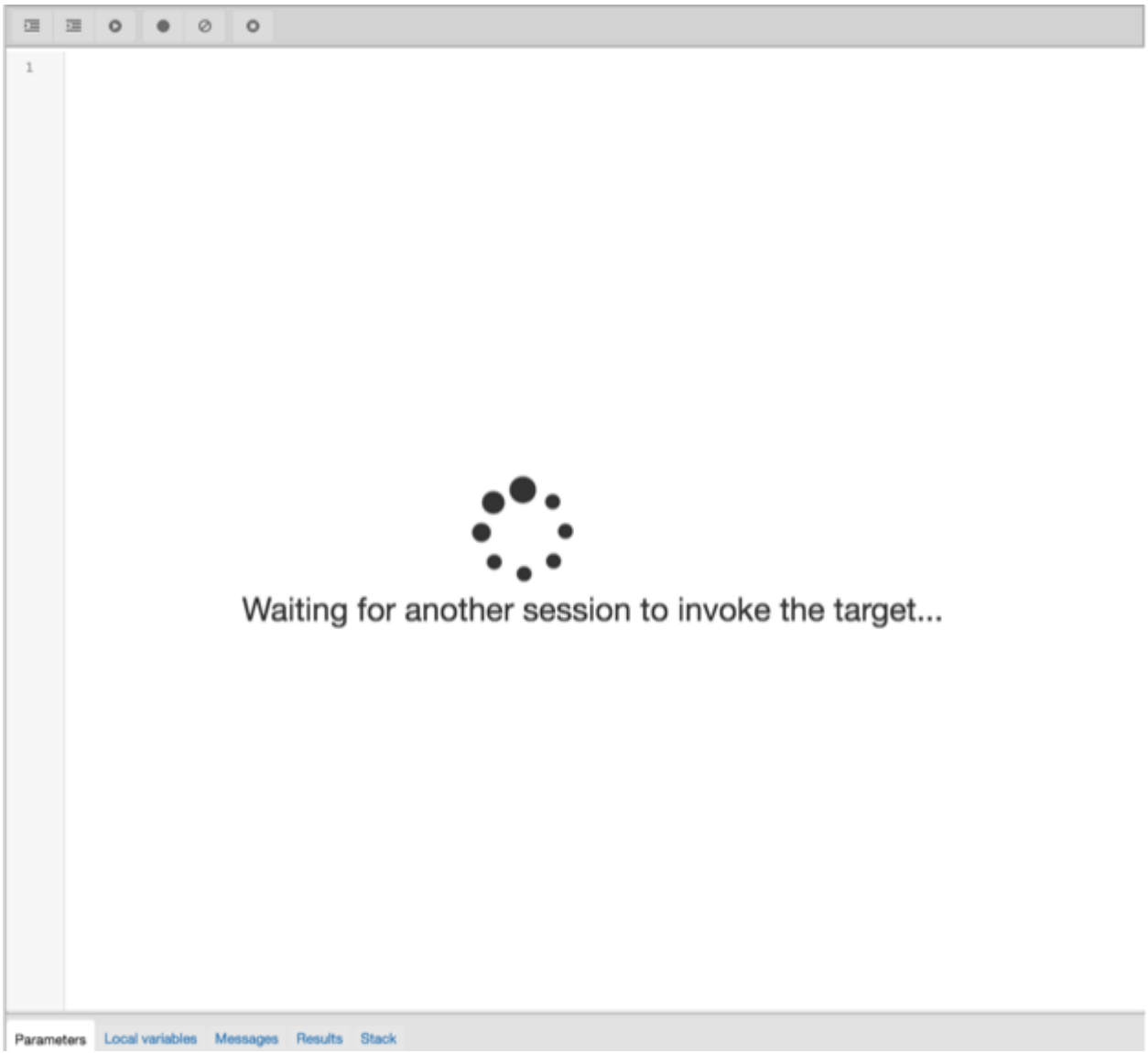
shared_preload_libraries = '$libdir/*other_libraries*/plugin_debugger'

After modifying the *shared_preload_libraries* parameter, restart the server to apply the changes.

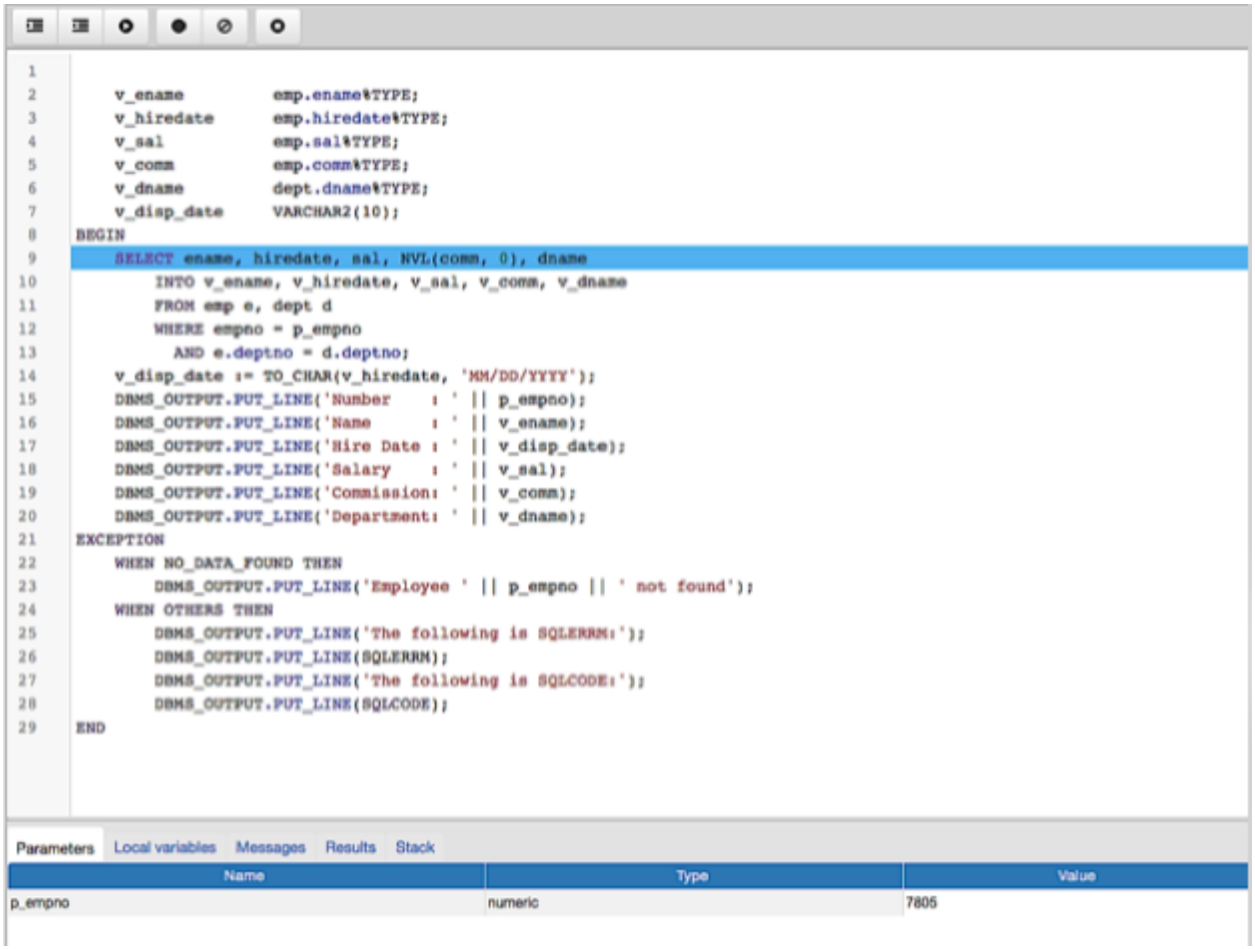
The debugger may be used for either in-context debugging or direct debugging of a target function or procedure. When you use the debugger for in-context debugging, you set a breakpoint at the first line of a program; when a session invokes the target, control is transferred to the debugger. When using direct debugging, the debugger prompts you for any parameters required by the target, and then allows you to step through the code.

In-context Debugging

To set a breakpoint at the first line of a program, right-click the name of the object you would like to debug, and select *Set breakpoint* from the *Debugging* sub-menu. The debugger window will open, waiting for another session to invoke the program.

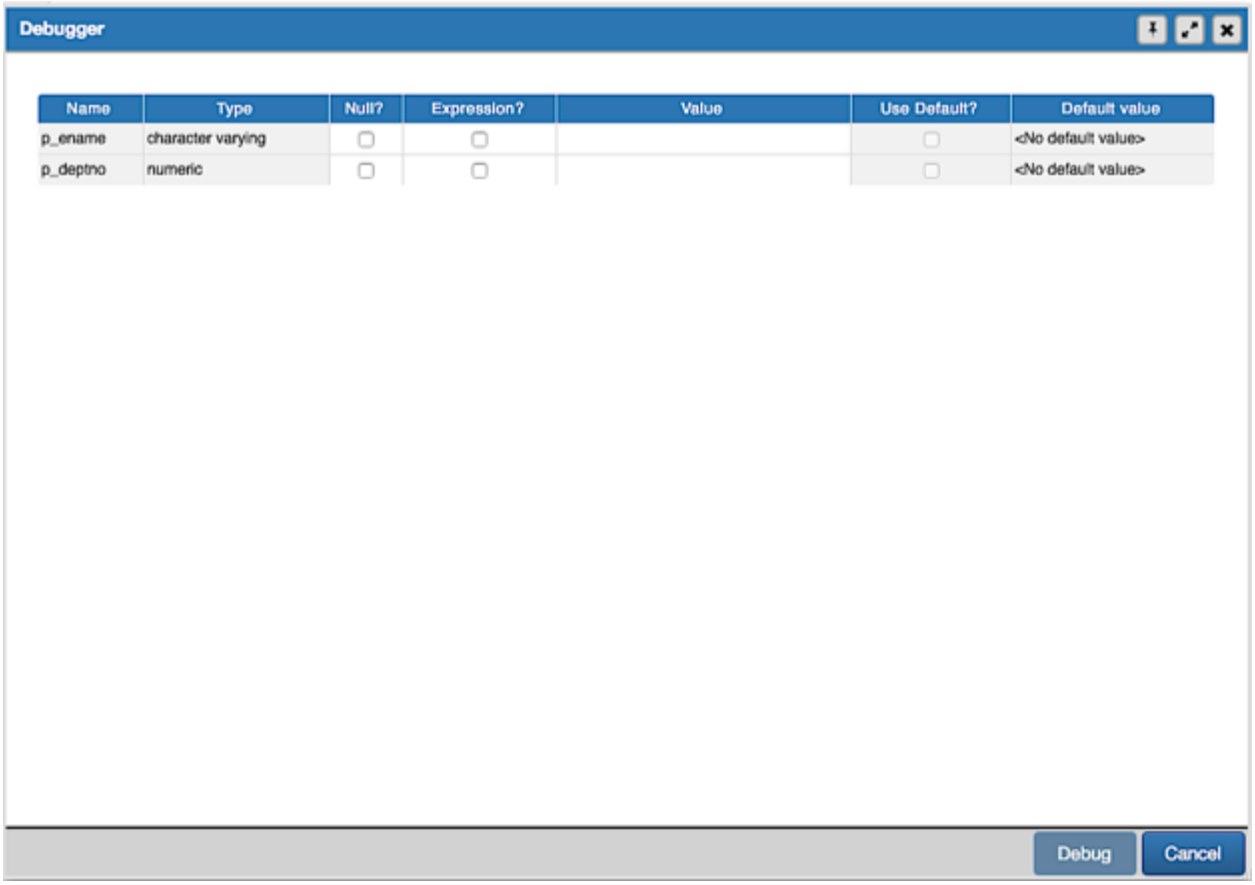


When another session invokes the target, the debugger will display the code, allowing you to add break points, or step through line-by-line. The other session is suspended until the debugging completes; then control is returned to the session.



Direct Debugging

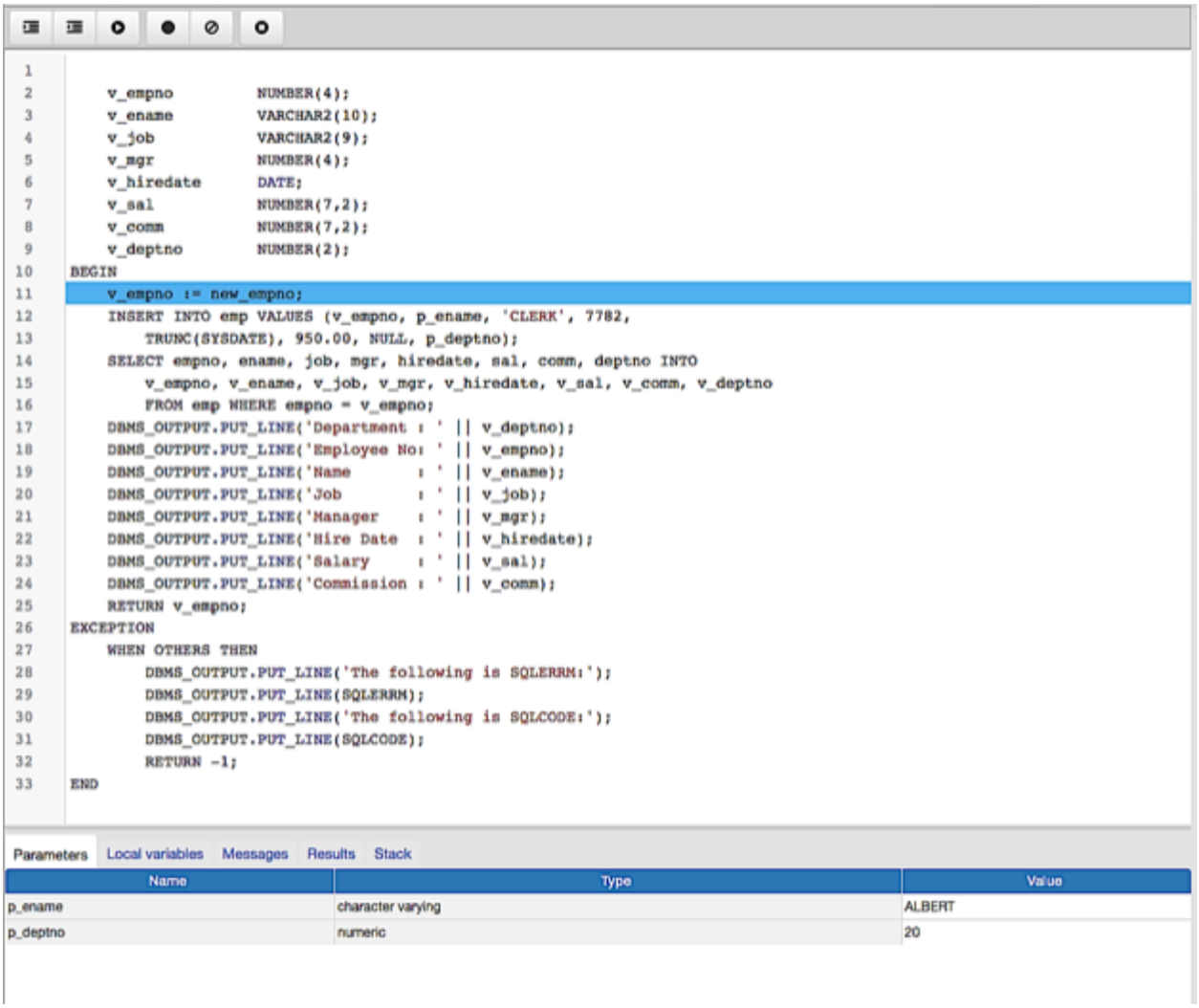
To use the debugger for direct debugging, right click on the name of the object that you wish to debug in the pgAdmin tree control and select *Debug* from the *Debugging* sub-menu. The debugger window will open, prompting you for any values required by the program:



Use the fields on the *Debugger* dialog to provide a value for each parameter:

- The *Name* field contains the formal parameter name.
- The *Type* field contains the parameter data type.
- Check the *Null?* checkbox to indicate that the parameter is a NULL value.
- Check the *Expression?* checkbox if the Value field contains an expression.
- Use the *Value* field to provide the parameter value that will be passed to the program. When entering parameter values, type the value into the appropriate cell on the grid, or, leave the cell empty to represent NULL, enter '' (two single quotes) to represent an empty string, or to enter a literal string consisting of just two single quotes, enter ''''. PostgreSQL 8.4 and above supports variadic function parameters. These may be entered as a comma-delimited list of values, quoted and/or cast as required.
- Check the *Use default?* checkbox to indicate that the program should use the value in the Default Value field.
- The *Default Value* field contains the default value of the parameter.

Provide values required by the program, and click the *Debug* button to start stepping through the program.

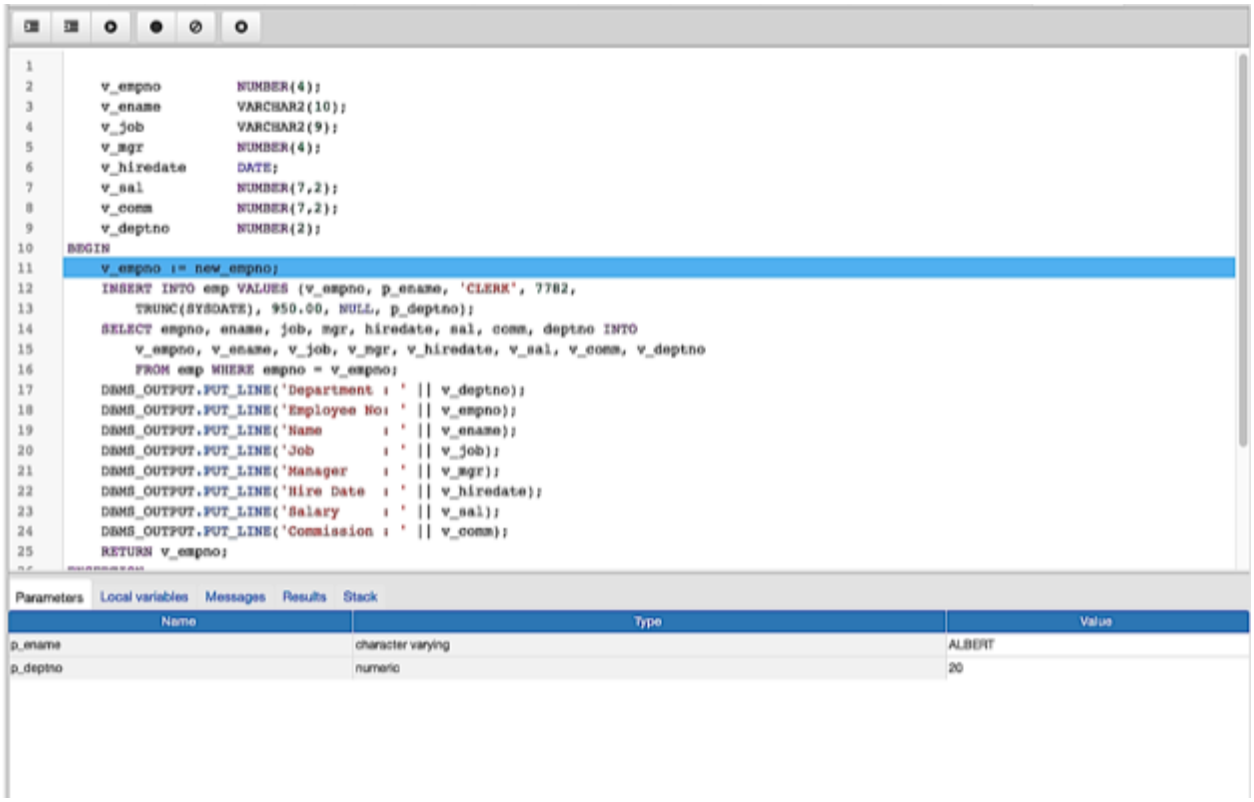


Using the Debugger

The main debugger window consists of two panels and a context-sensitive toolbar. Use toolbar icons to manage breakpoints and step into or through code; hover over an icon for a tooltip that identifies the option associated with the icon. The toolbar options are:

Option	Action
Step into	Click the *Step into* icon to execute the currently highlighted line of code.
Step over	Click the *Step over* icon to execute a line of code, stepping over any sub-functions invoked by the code. The sub-function executes, but is not debugged unless it contains a breakpoint.
Continue/Start	Click the *Continue/Start* icon to execute the highlighted code, and continue until the program encounters a breakpoint or completes.
Toggle breakpoint	Use the *Toggle breakpoint* icon to enable or disable a breakpoint (without removing the breakpoint).
Clear all breakpoints	Click the *Clear all breakpoints* icon to remove all breakpoints from the program.
Stop	Click the *Stop* icon to halt the execution of a program.

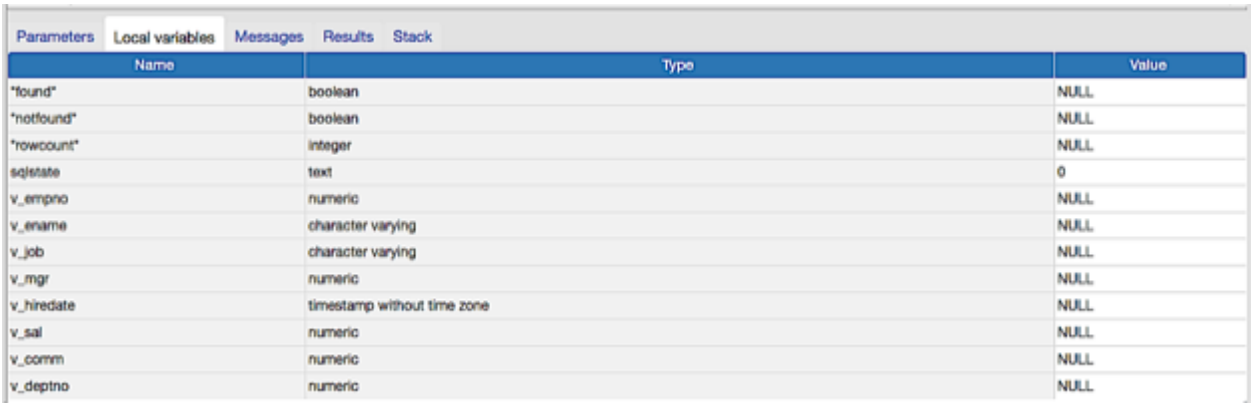
The top panel of the debugger window displays the program body; click in the grey margin next to a line number to add a breakpoint. The highlighted line in the top panel is the line that is about to execute.



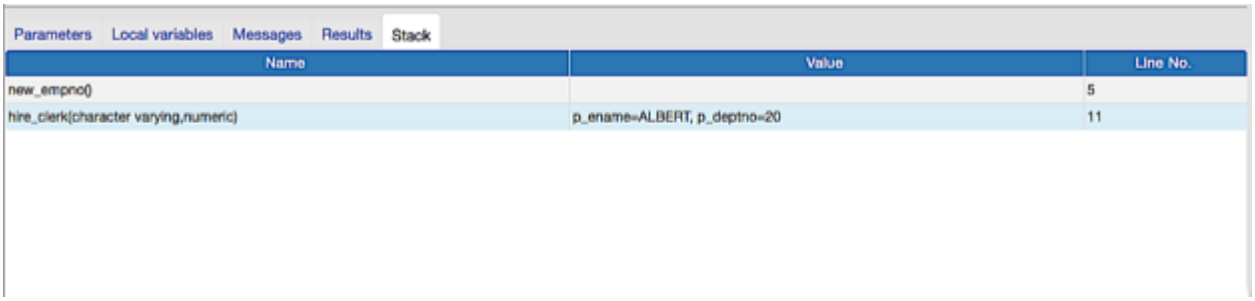
The lower panel of the debugger window provides a set of tabs that allow you to review information about the program:

- The *Parameters* tab displays the value of each parameter.
- The *Local* variables tab displays the current value of the program variables.
- The *Messages* tab displays any messages returned by the server (errors, warnings and informational messages).
- The *Results* tab displays the server message when the program completes.
- The *Stack* tab displays the list of functions that have been invoked, but which have not yet completed.

As you step through a program, the *Local variables* tab displays the current value of each variable:



When you step into a subroutine, the *Stack* tab displays the call stack, including the name of each caller, the parameter values for each caller (if any), and the line number within each caller:



Select a caller to change focus to that stack frame and display the state of the caller in the upper panel.

When the program completes, the *Results* tab displays the message returned by the server. If the program encounters an error, the *Messages* tab displays details:

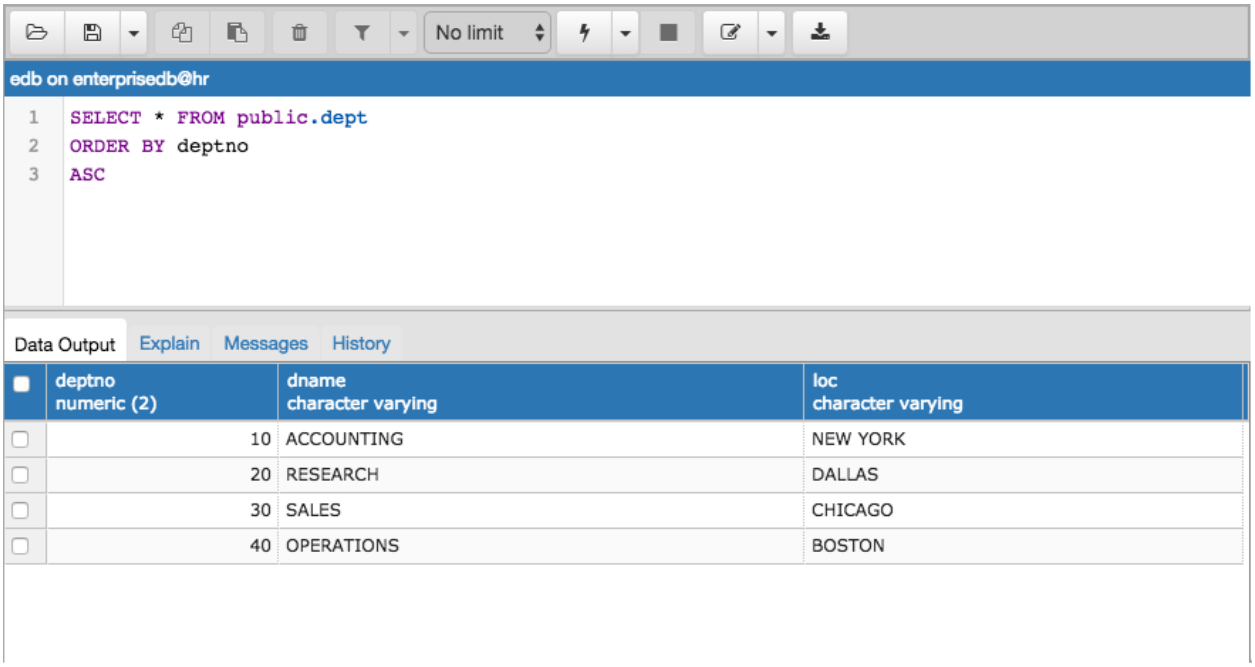
7.2 The Query tool

The Query tool is a powerful, feature-rich environment that allows you to execute arbitrary SQL commands and review the result set. If you access the Query tool via the *Query Tool* menu option on the *Tools* menu, you can:

- Issue ad-hoc SQL queries.
- Execute arbitrary SQL commands.
- Save the data displayed in the output panel to a CSV file.
- Review the execution plan of a SQL statement in either a text or a graphical format.
- View analytical information about a SQL statement.

If you open the Query tool via the *View Data* context-menu, the Query tool acts as a data editor, allowing you to:

- View or modify the data that is stored in a table.
- Filter the result set.
- Save the data displayed in the output panel to a CSV file.
- Review the execution plan of a SQL statement in either a text or a graphical format.
- View analytical information about a SQL statement.



The Query tool features a toolbar that allows quick access to frequently used options, and a work environment divided into two panels:

- The upper panel of the Query tool contains the *SQL Editor*. You can use the panel to manually enter a query, or review the query that generated the result set displayed in the lower panel.
- The lower panel of the Query tool contains the *Data Output* panel. The output panel displays the result of a query, or information about a query's execution plan.

pgAdmin allows you to open multiple copies of the Query tool (in individual tabs) simultaneously. For example, if you select *Query tool* from the *Tools* menu, the Query tool opens in a tab labeled *Query-1*; if you open the Query tool again (without closing *Query-1*), a second copy will open in *Query-2*. To close a copy of the Query tool, click the *X* in the upper-right hand corner of the tab bar.

The Query tool Toolbar

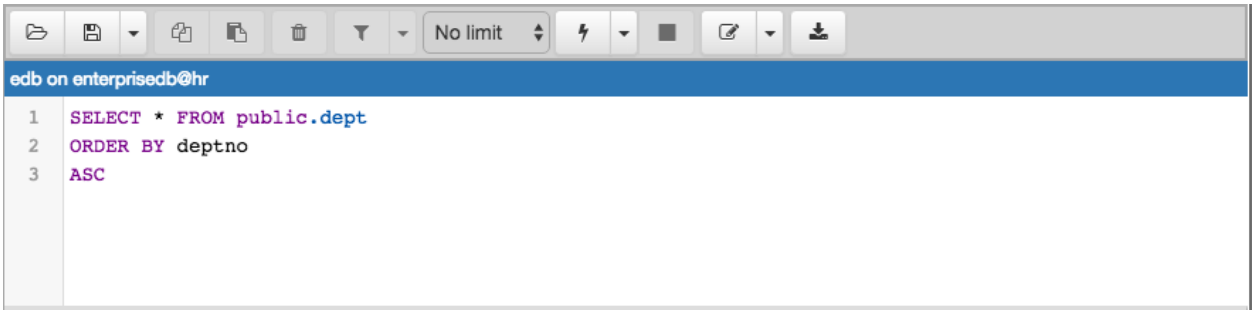
The *Query tool* toolbar uses context-sensitive icons that provide shortcuts to frequently performed tasks. If an icon is highlighted, the option is enabled; if the icon is grayed-out, the task is disabled.

Hover over an icon to display a tooltip that describes the icon's functionality:

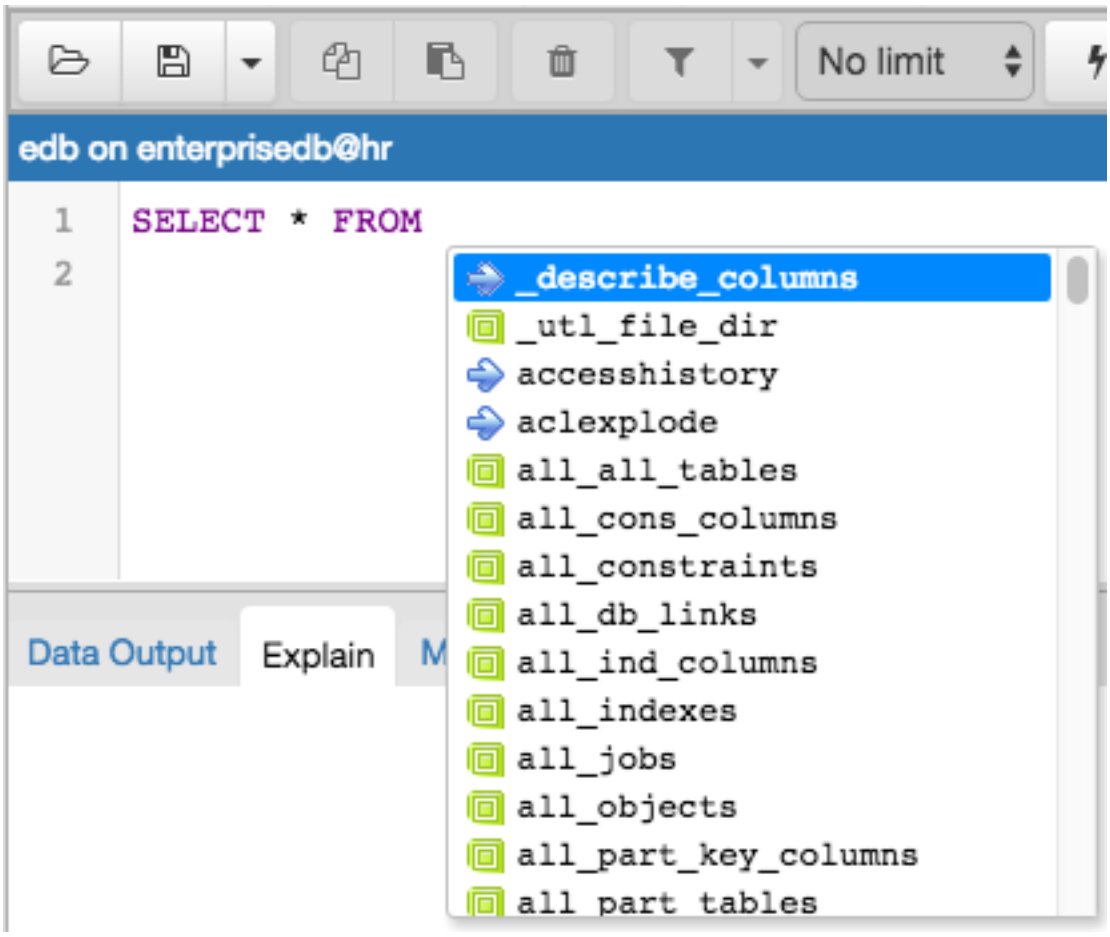
Icon	Behavior
Open File	Click the *Open File* icon to display a previously saved query in the SQL Editor.
Save	Click the *Save* icon to save the query that is currently displayed in the SQL Editor.
Copy	Click the *Copy* icon to copy the currently selected row.
Paste	Click the *Paste* icon to paste the content that is currently on the clipboard.
Add New Row	Use the *Add New Row* icon to add a new row in the output panel.
Filter	Click the *Filter* icon to open a dialog that allows you to write and apply a filter for the content currently displayed in the output panel. Click the down arrow to open the *Filter* drop- down menu and select from predefined options: Select *Remove* to remove the currently applied filter and display the complete result set. Select *By selection* to refresh the displayed data, displaying only those rows that have columns that match the currently highlighted value. Select *Exclude selection* to refresh the displayed data, excluding those rows that have columns that match the currently highlighted value.
No limit	Use the *No limit* drop-down listbox to specify how many rows to display in the output panel. Select from: *No limit* (the default), *1000 rows*, *500 rows*, or *100 rows*.
Execute/Refresh	Click the *Execute/Refresh* icon to either execute or refresh the query highlighted in the SQL editor panel. Click the down arrow to access other execution options: Select *Execute/Refresh* to invoke the SQL command and refresh the result set. Select *Explain* to view an explanation plan for the current query. The result of the EXPLAIN is displayed graphically on the *Explain* tab of the output panel, and in text form on the *Data Output* tab. Select *Explain analyze* to invoke an EXPLAIN ANALYZE command on the current query. Navigate through the *Explain Options* menu to select options for the EXPLAIN command: Select *Verbose* to display additional information regarding the query plan. Select *Costs* to include information on the estimated startup and total cost of each plan node, as well as the estimated number of

The SQL Editor Panel

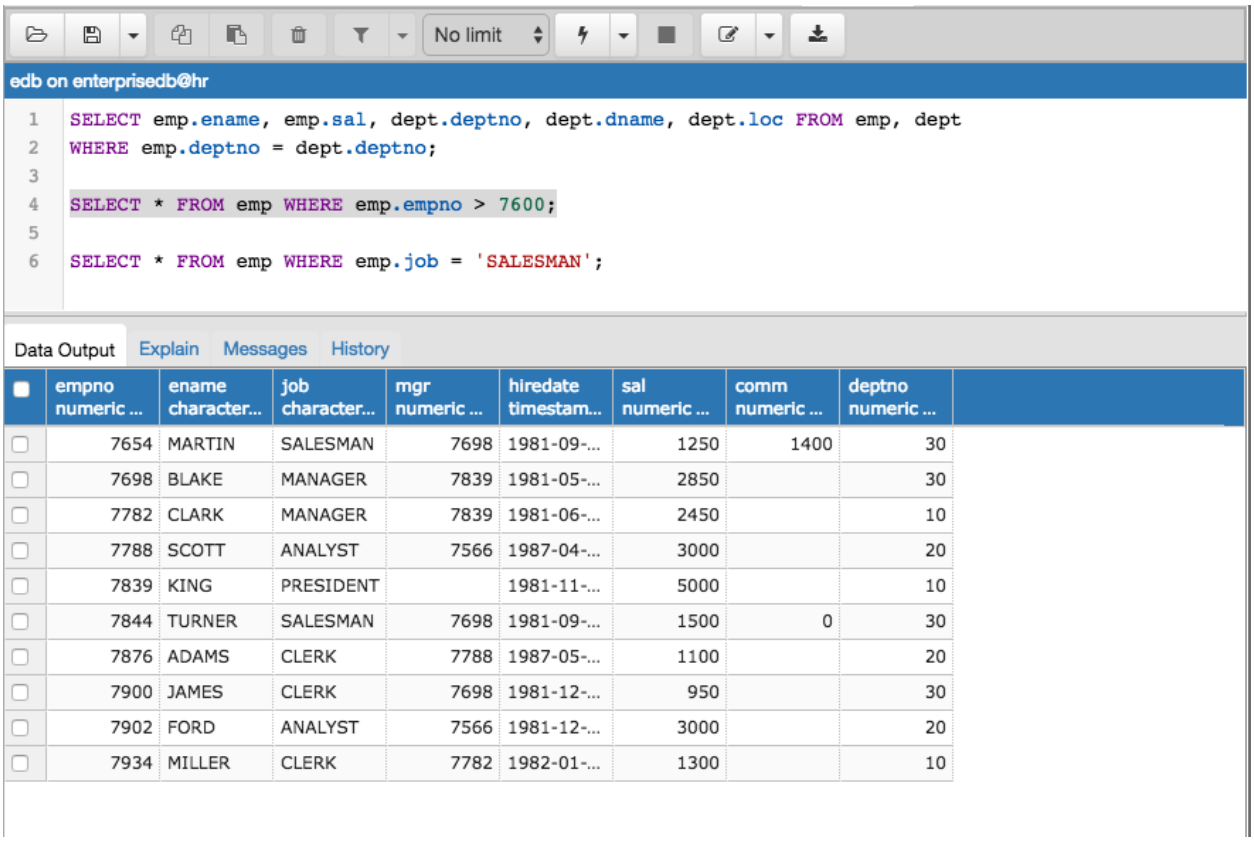
The *SQL editor* panel contains a workspace for entering commands; you can read a query from a file, or type a query. The SQL editor features syntax coloring and autocompletion to help you develop queries.



To use autocomplete, begin typing your query; when you would like the Query editor to suggest object names or commands that might be next in your query, press the Control+Space key combination. For example, type "*SELECT * FROM* " (without quotes, but with a trailing space), and then press the Control+Space key combination to select from a popup menu of autocomplete options.



After entering a query, select the *Execute/Refresh* icon from the toolbar. The complete contents of the SQL editor panel will be sent to the database server for execution. To execute only a section of the code that is displayed in the SQL editor, highlight the text that you want the server to execute, and click the *Execute/Refresh* icon:



The message returned by the server when a command executes is displayed on the *Messages* tab of the output panel. If the command is successful, the *Messages* tab displays execution details:

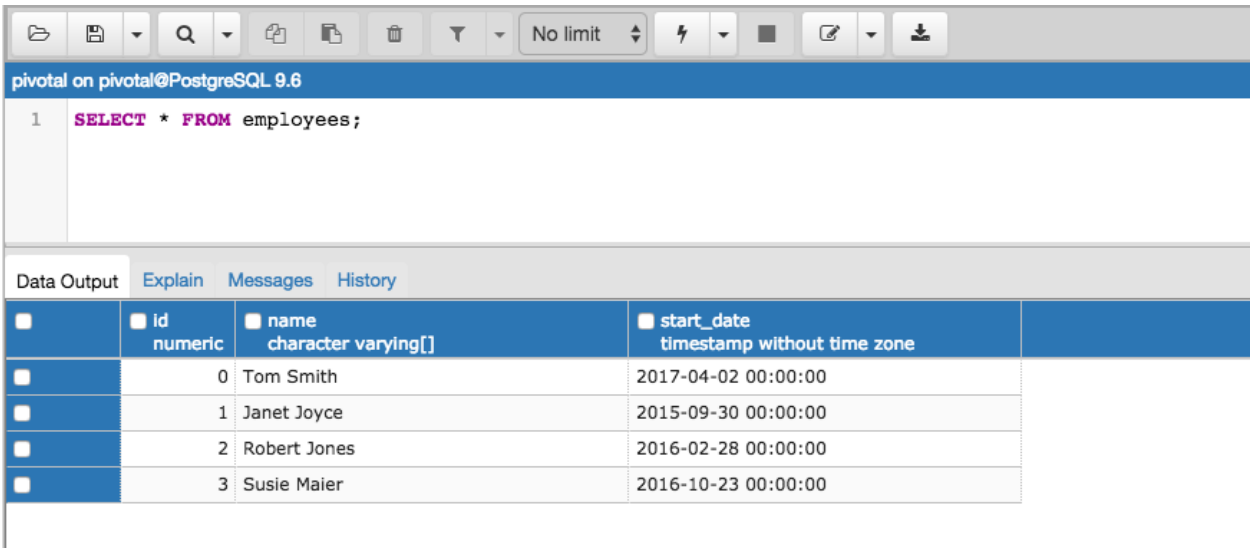
The editor also offers several features that help with code formatting:

- The auto-indent feature will automatically indent text to the same depth as the previous line when you press the Return key.
- Block indent text by selecting two or more lines and pressing the Tab key.

The Data Output Panel

Use the *Data Output* panel of the *Query tool* to view data and information generated by a query in the *SQL editor*, or to *View Data* for an object currently selected in the *pgAdmin* tree control.

The *Data Output* panel organizes output through the following tabs: *Data Output*, *Explain*, *Messages*, and *History*.



If the Query tool is opened through the *Query tool* menu option on the *Tools* menu, you can use the *Data Output* tab to view the results of an arbitrary query in a table format. If the Query tool is opened through a *View Data* context menu, the *Data Output* tab will display the data stored in the table from which the Query tool was opened.

- If enabled, use the *Filter* options from the Query tool toolbar to refine the result set displayed on the *Data Output* tab.
- If enabled, use the *No limit* drop-down to specify how many rows to display on the *Data Output* tab.
- If enabled, use the *Execute/Refresh* options to retrieve query execution information and set query execution options.
- Use the *Download as CSV* icon to download the content of the *Data Output* tab as a comma-delimited file.

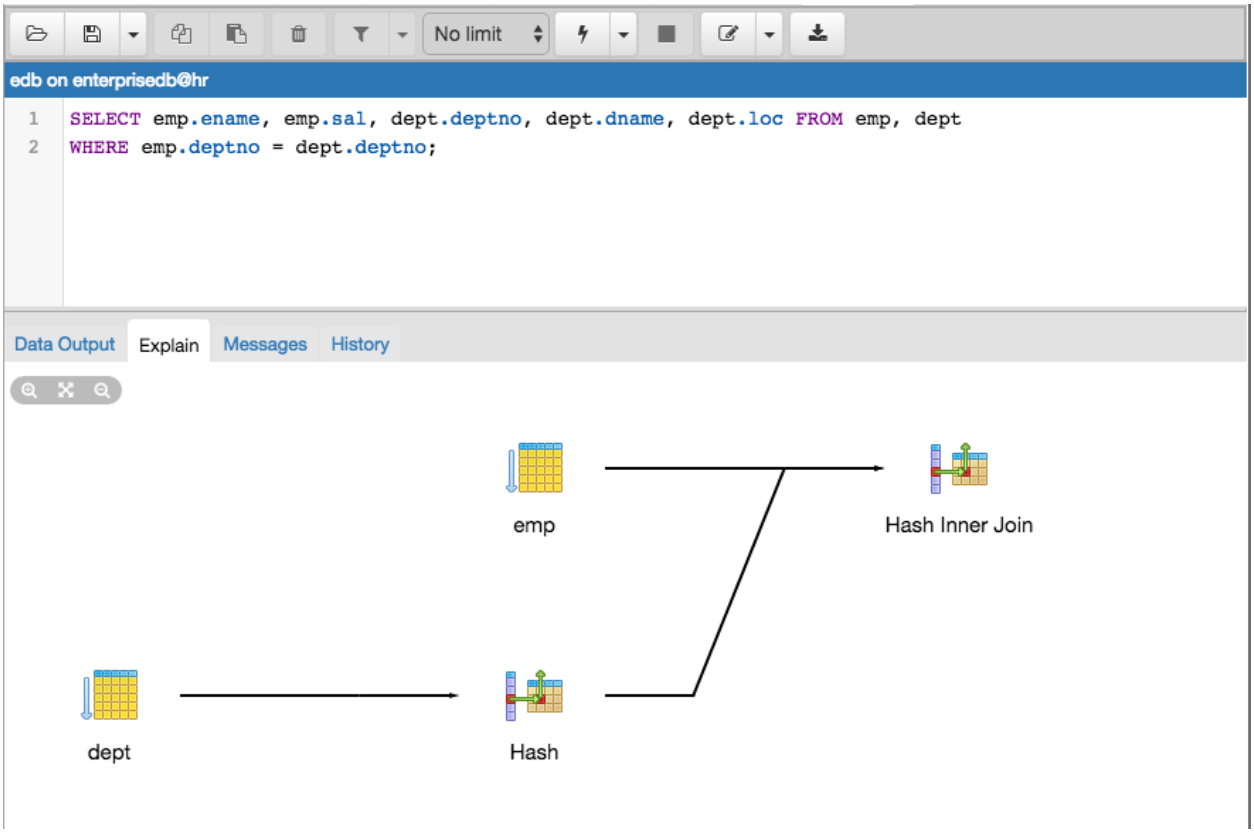
All rowsets from previous queries or commands that are displayed in the *Data Output* panel will be discarded when you invoke another query; open another query tool browser tab to keep your previous results available.

If the Query Tool is opened using the *View Data* menu option and the data is updatable and has a primary key, then you can double-click on values on the *Data Output* tab and edit them:

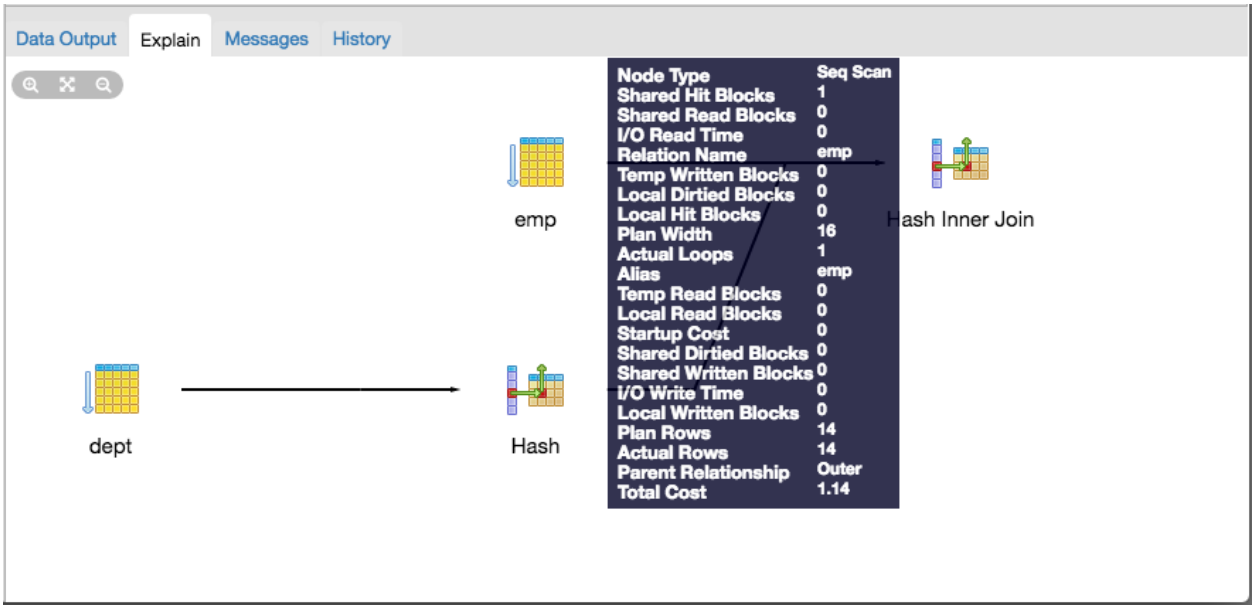
- To enter a NULL, clear the value of the string.
- To enter a blank set the value of the cell to ‘’.
- To enter the string ‘’. enter the value ‘’.

Once the data has been edited as required, use the Save button to save the changes to the database.

Use the *Explain* tab to view a graphical representation of a query:

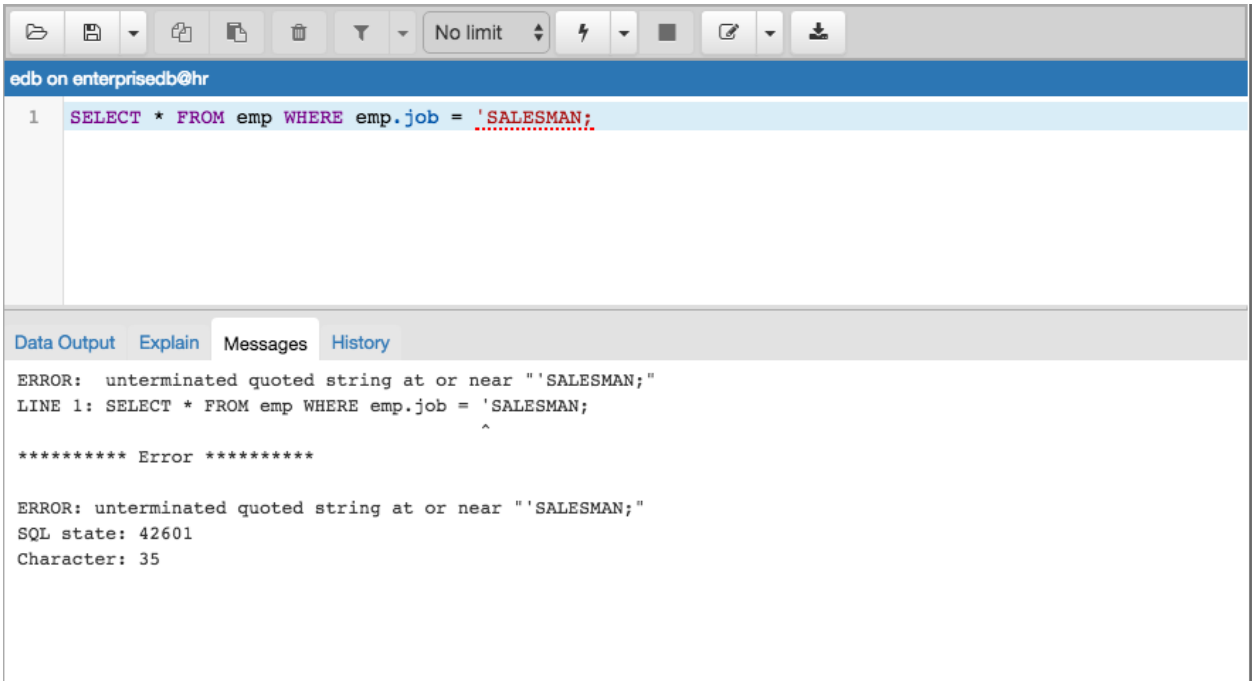


To generate a graphical explain diagram, open the *Explain* tab, and select *Explain*, *Explain Analyze*, or one or more options from the *Explain options* menu on the *Execute/Refresh* drop-down. Please note that *EXPLAIN VERBOSE* cannot be displayed graphically. Hover over an icon on the *Explain* tab to review information about that item; a popup window will display information about the selected object:



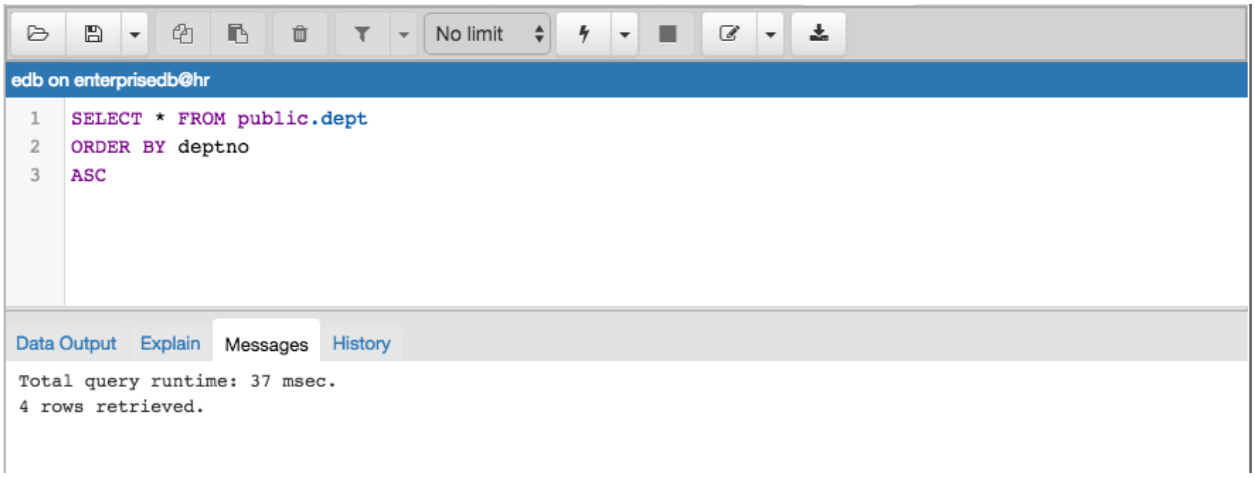
Note that the query plan that accompanies the *Explain analyze* is available on the *Data Output* tab.

Use the *Messages* tab to view information about the last-executed query:

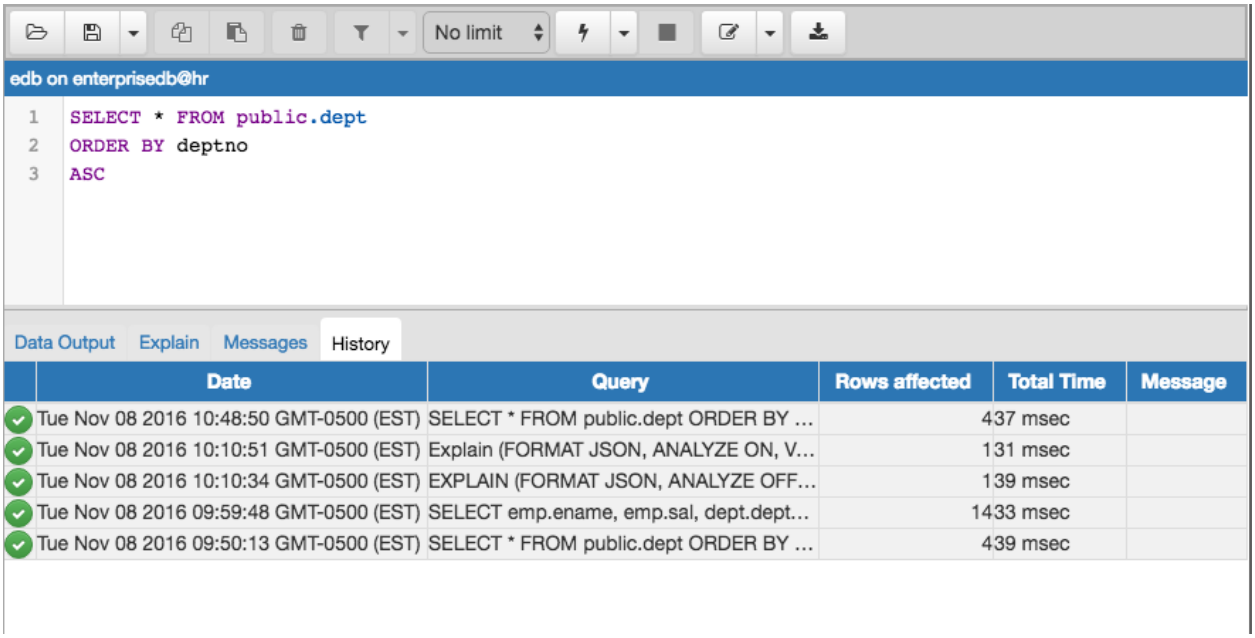


If the server returns an error, the error message will be displayed on the *Messages* tab, and the syntax that cause the error will be underlined in the SQL editor.

If a query succeeds, the *Messages* tab displays how long the query took to complete and how many rows were retrieved:



Use the *History* tab to review activity for the current session:



The History tab displays:

- The date and time that a query was invoked.
- The text of the query.
- The number of rows returned by the query.
- The amount of time it took the server to process the query and return a result set.
- Messages returned by the server (not noted on the *Messages* tab).

To erase the content of the *History* tab, select *Clear history* from the *Clear query window* drop-down menu.

CHAPTER

EIGHT

PGADMIN DEPLOYMENT

Pre-compiled and configured installation packages for pgAdmin 4 are available for a number of desktop environments; we recommend using an installer whenever possible. If you are interested in learning more about the project, or if a pgAdmin installer is not available for your environment, the pages listed below will provide detailed information about creating a custom deployment.

Contents:

CHAPTER

NINE

PGADMIN PROJECT CONTRIBUTIONS

pgAdmin is an open-source project that invites you to get involved in the development process. For more information about contributing to the pgAdmin project, contact the developers on the pgAdmin mailing list pgadmin-hackers@postgresql.org to discuss any ideas you might have for enhancements or bug fixes.

In the sections listed below, you'll find detailed information about the development process used to develop, improve, and maintain the pgAdmin client.

Contents:

9.1 Submitting Patches

Before developing a patch for pgAdmin you should always contact the developers on the mailing list pgadmin-hackers@postgresql.org to discuss your plans. This ensures that others know if you're fixing a bug and can then avoid duplicating your work, and in the case of large patches, gives the community the chance to discuss and refine your ideas before investing too much time writing code that may later be rejected.

You should always develop patches against a checkout of the source code from the GIT source code repository, and not a release tarball. This ensures that you're working with the latest code on the branch and makes it easier to generate patches correctly. You can checkout the source code with a command like:

```
$ git clone git://git.postgresql.org/git/pgadmin4.git
```

Once you've made the changes you wish to make, commit them to a private development branch in your local repository. Then create a patch containing the changes in your development branch against the upstream branch on which your work is based. For example, if your current branch contains your changes, you might run:

```
$ git diff origin/master > my_cool_feature.diff
```

to create a patch between your development branch and the public master branch.

Once you have your patch, check it thoroughly to ensure it meets the pgAdmin *Coding Standards*, and review it against the *Code Review Notes* to minimise the chances of it being rejected. Once you're happy with your work, mail it as an attachment to the mailing list pgadmin-hackers@postgresql.org. Please ensure you include a full description of what the patch does, as well as the rationale for any important design decisions.

9.2 Code Overview

The bulk of pgAdmin is a Python web application written using the Flask framework on the backend, and HTML5 with CSS3, Bootstrap and jQuery on the front end. A desktop runtime is also included for users that prefer a desktop application to a web application, which is written in C++ using the QT framework.

9.2.1 Runtime

The runtime is essentially a Python webserver and browser in a box. Found in the **/runtime** directory in the source tree, it is a relatively simple QT application that is most easily modified using the **QT Creator** application.

9.2.2 Web Application

The web application forms the bulk of pgAdmin and can be found in the **/web** directory in the source tree. The main file is **pgAdmin4.py** which can be used to run the built-in standalone web server, or as a WSGI application for production use.

Configuration

The core application configuration is found in **config.py**. This file includes all configurable settings for the application, along with descriptions of their use. It is essential that various settings are configured prior to deployent on a web server; these can be overriden in **config_local.py** to avoid modifying the main configuration file.

User Settings

When running in desktop mode, pgAdmin has a single, default user account that is used for the desktop user. When running in server mode, there may be unlimited users who are required to login prior to using the application. pgAdmin utilised the **Flask-Security** module to manage application security and users, and provides options for self-service password reset and password changes etc.

Whether in desktop or server mode, each user's settings are stored in a SQLite database which is also used to store the user accounts. This is initially created using the **setup.py** script which will create the database file and schema within it, and add the first user account (with administrative privileges) and a default server group for them. A **settings** table is also used to store user configuration settings in a key-value fashion. Although not required, setting keys (or names) are typically formatted using forward slashes to artificially namespace values, much like the pgAdmin 3 settings files on Linux or Mac.

Note that the local configuration must be setup prior to **setup.py** being run. The local configuration will determine how the script sets up the database, particularly with regard to desktop vs. server mode.

9.2.3 pgAdmin Core

The heart of pgAdmin is the **pgadmin** package. This contains the globally available HTML templates used by the Jinja engine, as well as any global static files such as images, Javascript and CSS files that are used in multiple modules.

The work of the package is handled in it's constructor, **__init__.py**. This is responsible for setting up logging and authentication, dynamically loading other modules, and a few other tasks.

9.2.4 Modules

Units of functionality are added to pgAdmin through the addition of modules. Theses are Python object instance of classes, inherits the PgAdminModule class (a Flask Blueprint implementation), found in **web/pgadmin/utils.py**. It provide various hook points for other modules to utilise (primarily the default module - the browser).

To be recognised as a module, a Python package must be created. This must:

1) Be placed within the **web/pgadmin/** directory, and

2) Implements pgadmin.utils.PgAdminModule class

3) An instance variable (generally - named **blueprint**) representing that particular class in that package.

Each module may define a **template** and **static** directory for the Blueprint that it implements. To avoid name collisions, templates should be stored under a directory within the specified template directory, named after the module itself. For example, the **browser** module stores it's templates in **web/pgadmin/browser/templates/browser/**. This does not apply to static files which may omit the second module name.

In addition to defining the Blueprint, the **views** module is typically responsible for defining all the views that will be rendered in response to client requests, we must provide a REST API url(s) for these views. These must include appropriate route and security decorators. Take a look at the NodeView class, which uses the same approach as Flask's MethodView, it can be found in **web/pgadmin/browser/utils.py**. This specific class is used by browser nodes for creating REST API url(s) for different operation on them. i.e. list, create, update, delete, fetch children, get statistics/reversed SQL/dependencies/dependents list for that node, etc. We can use the same class for other purpose too. You just need to inherit that class, and overload the member variables operations, parent_ids, ids, node_type, and then register it as node view with PgAdminModule instance.

Most pgAdmin modules will also implement the **hooks** provided by the PgAdminModule class. This is responsible for providing hook points to integrate the module into the rest of the application - for example, a hook might tell the caller what CSS files need to be included on the rendered page, or what menu options to include and what they should do. Hook points need not exist if they are not required. It is the responsiblity of the caller to ensure they are present before attempting to utilise them.

Hooks currently implemented are:

```
class MyModule(PgAdminModule):
    """
    This is class implements the pgadmin.utils.PgAdminModule, and
    implements the hooks
    """

    ...

    def get_own_stylesheets(self):
        """
        Returns:
            list: the stylesheets used by this module, not including any
                  stylesheet needed by the submodules.
        """
        return [url_for('static', 'css/mymodule.css')]

    def get_own_javascripts(self):
        """
        Returns:
            list of dict:
            - contains the name (representation for this javascript
              module), path (url for it without .js suffix), deps (array of
              dependents), exports window object by the javascript module,
              and when (would you like to load this javascript), etc
              information for this module, not including any script needed
              by submodules.
        """
        return [
            {
                'name': 'pgadmin.extension.mymodule',
                'path': url_for('static', filename='js/mymodule'),
                'exports': None,
                'when': 'server'
                }
```

(continues on next page)

(continued from previous page)

```
            ]

    def get_own_menuitems(self):
        """
        Returns:
            dict: the menuitems for this module, not including
                  any needed from the submodules.
        """
        return {
            'help_items': [
                MenuItem(
                    name='mnu_mymodule_help',
                    priority=999,
                    # We need to create javascript, which registers itself
                    # as module
                    module="pgAdmin.MyModule",
                    callback='about_show',
                    icon='fa fa-info-circle',
                    label=gettext('About MyModule'
                    )
                ]
            }
    def get_panels(self):
        """
        Returns:
            list: a list of panel objects to add implemented in javascript
                  module
        """
        return []
    ...



blueprint = MyModule('mymodule', __name__, static_url_path='/static')
```

pgAdmin Modules may include any additional Python modules that are required to fulfill their purpose, as required. They may also reference other dynamically loaded modules, but must use the defined hook points and fail gracefully in the event that a particular module is not present.

9.2.5 Nodes

Nodes are very similar to modules, it represents an individual node or, collection object on the browser treeview. To recognised as a node module, a Python package (along with javascript modules) must be created. This must:

1) Be placed within the **web/pgadmin/browser/** directory, and

2) Implements the BrowserPluginModule, and registers the node view, which exposes required the REST APIs

3) An instance of the class object

9.2.6 Front End

pgAdmin uses javascript extensively for the front-end implementation. It uses require.js to allow the lazy loading (or, say load only when required), bootstrap for UI look and feel, Backbone for data manipulation of a node, Backform for generating properties/create dialog for selected node. We have divided each module in small chunks as much as

possible. Not all javascript modules are required to be loaded (i.e. loading a javascript module for database will make sense only when a server node is loaded competely.) Please look at the the javascript files node.js, browser.js, menu.js, panel.js, etc for better understanding of the code.

9.3 Coding Standards

pgAdmin uses multiple technologies and multiple languages, each of which have their own coding standards.

9.3.1 General

In all languages, indentations should be made with 4 spaces, and excessively long lines wrapped where appropriate to ensure they can be read on smaller displays (80 characters is used in many places, but this is not a required maximum size as it's quite wasteful on modern displays). Typically lines should not be longer than 120 characters.

Comments should be included in all code where required to explain its purpose or how it works if not obvious from a quick review of the code itself.

9.3.2 CSS 3

CSS3 is used for styling and layout throughout the application. Extensive use is made of the Bootstrap Framework to aid in that process, however additional styles must still be created from time to time.

Most custom styling comes from individual modules which may advertise static stylesheets to be included in the module that is loading them via hooks.

Styling overrides (for example, to alter the Bootstrap look and feel) will typically be found in the **overrides.css** file in the main static file directory for the application.

Styling should never be applied inline in HTML, always through an external stylesheet, which should contain comments as appropriate to explain the usage or purpose for the style.

Styles should be specified clearly, one per line. For example:

```
/* iFrames should have no border */
iframe {
    border-width: 0;
}

/* Ensure the codemirror editor displays full height gutters when resized */
.CodeMirror, .CodeMirror-gutters {
    height: 100% !important;
}
```

All stylesheets must be CSS3 compliant.

9.3.3 HTML 5

HTML 5 is used for page structure throughout the application, in most cases being rendered from templates by the Jinja2 template engine in Flask.

All HTML must be HTML 5 compliant.

9.3.4 Javascript

Client-side code is written in Javascript using jQuery and various plugins. Whilst much of the code is rendered from static files, there is also code that is rendered from templates using Jinja2 (often to inject the users settings) or constructed on the fly from module hooks.

A typical Javascript function might be formatted like this (this snipped is from a template):

```
// Delete a server group
function delete_server_group(item) {
    alertify.confirm(
        'Delete server group?',
        'Are you sure you wish to delete the server group "{0}"?'.replace('{0}', tree.
↪getLabel(item)),
        function() {
            var id = tree.getId(item)
            $.post("{{ url_for('NODE-server-group.delete') }}", { id: id })
                .done(function(data) {
                    if (data.success == 0) {
                        report_error(data.errormsg, data.info);
                    } else {
                        var next = tree.next(item);
                        var prev = tree.prev(item);
                        tree.remove(item);
                        if (next.length) {
                            tree.select(next);
                        } else if (prev.length) {
                            tree.select(prev);
                        }
                    }
                }
            )
        },
        null
    )
}
```

Note the use of a descriptive function name, using the underscore character to separate words in all lower case, and short but descriptive lower case variable names.

C++

C++ code is used in the desktop runtime for the application, primarily with the QT framework and an embedded Python interpreter. Note the use of hanging braces, which may be omitted if on a single statement is present:

```
// Ping the application server to see if it's alive
bool PingServer(QUrl url)
{
    QNetworkAccessManager manager;
    QEventLoop loop;
    QNetworkReply *reply;
    QVariant redirectUrl;

    url.setPath("/utils/ping");

    do
    {
```

(continues on next page)

(continued from previous page)

```
        reply = manager.get(QNetworkRequest(url));

        QObject::connect(reply, SIGNAL(finished()), &loop, SLOT(quit()));
        loop.exec();

        redirectUrl = reply->attribute(QNetworkRequest::RedirectionTargetAttribute);
        url = redirectUrl.toUrl();

        if (!redirectUrl.isNull())
            delete reply;

    } while (!redirectUrl.isNull());

    if (reply->error() != QNetworkReply::NoError)
        return false;

    QString response = reply->readAll();

    if (response != "PING")
    {
        qDebug() << "Failed to connect, server response: " << response;
        return false;
    }

    return true;
}
```

9.3.5 Python

Python is used for the backend web server. All code must be compatible with Python 2.7 and should include PyDoc comments whilst following the official Python coding standards defined in PEP 8. An example function along with the required file header is shown below:

```
##########################################################################
#
# pgAdmin 4 - PostgreSQL Tools
#
# Copyright (C) 2013 - 2017, The pgAdmin Development Team
# This software is released under the PostgreSQL Licence
#
##########################################################################

"""Integration hooks for server groups."""

from flask import render_template, url_for
from flask.ext.security import current_user

from pgadmin.settings.settings_model import db, ServerGroup

def get_nodes():
    """Return a JSON document listing the server groups for the user"""
    groups = ServerGroup.query.filter_by(user_id=current_user.id)

    value = ''
    for group in groups:
```

(continues on next page)

(continued from previous page)

```
        value += '{"id":%d,"label":"%s","icon":"icon-server-group","inode":true},' \
                 % (group.id, group.name)

    value = value[:-1]

    return value
```

9.4 Code Snippets

This document contains code for some of the important classes, listed as below:

- *PgAdminModule*
- *NodeView*
- *BaseDriver*
- *BaseConnection*

9.4.1 PgAdminModule

PgAdminModule is inherted from Flask.Blueprint module. This module defines a set of methods, properties and attributes, that every module should implement.

```
class PgAdminModule(Blueprint):
    """
    Base class for every PgAdmin Module.

    This class defines a set of method and attributes that
    every module should implement.
    """

    def __init__(self, name, import_name, **kwargs):
        kwargs.setdefault('url_prefix', '/' + name)
        kwargs.setdefault('template_folder', 'templates')
        kwargs.setdefault('static_folder', 'static')
        self.submodules = []

        super(PgAdminModule, self).__init__(name, import_name, **kwargs)

        def create_module_preference():
            # Create preference for each module by default
            if hasattr(self, 'LABEL'):
                self.preference = Preferences(self.name, self.LABEL)
            else:
                self.preference = Preferences(self.name, None)

            self.register_preferences()

        # Create and register the module preference object and preferences for
        # it just before the first request
        self.before_app_first_request(create_module_preference)

    def register_preferences(self):
```

(continues on next page)

(continued from previous page)

```
        pass

    def register(self, app, options, first_registration=False):
        """
        Override the default register function to automagically register
        sub-modules at once.
        """
        if first_registration:
            self.submodules = list(app.find_submodules(self.import_name))

        super(PgAdminModule, self).register(app, options, first_registration)

        for module in self.submodules:
            app.register_blueprint(module)

    def get_own_stylesheets(self):
        """
        Returns:
            list: the stylesheets used by this module, not including any
                stylesheet needed by the submodules.
        """
        return []

    def get_own_messages(self):
        """
        Returns:
            dict: the i18n messages used by this module, not including any
                messages needed by the submodules.
        """
        return dict()

    def get_own_javascripts(self):
        """
        Returns:
            list: the javascripts used by this module, not including
                any script needed by the submodules.
        """
        return []

    def get_own_menuitems(self):
        """
        Returns:
            dict: the menuitems for this module, not including
                any needed from the submodules.
        """
        return defaultdict(list)

    def get_panels(self):
        """
        Returns:
            list: a list of panel objects to add
        """
        return []

    @property
    def stylesheets(self):
        stylesheets = self.get_own_stylesheets()
```

(continues on next page)

(continued from previous page)

```
        for module in self.submodules:
            stylesheets.extend(module.stylesheets)
        return stylesheets

    @property
    def messages(self):
        res = self.get_own_messages()

        for module in self.submodules:
            res.update(module.messages)
        return res

    @property
    def javascripts(self):
        javascripts = self.get_own_javascripts()
        for module in self.submodules:
            javascripts.extend(module.javascripts)
        return javascripts

    @property
    def menu_items(self):
        menu_items = self.get_own_menuitems()
        for module in self.submodules:
            for key, value in module.menu_items.items():
                menu_items[key].extend(value)
        menu_items = dict((key, sorted(value, key=attrgetter('priority')))
                          for key, value in menu_items.items())
        return menu_items
```

9.4.2 NodeView

NodeView class helps exposing basic REST APIs for different operations used by pgAdmin Browser. The basic idea has been taken from the Flask's MethodView class. Because - we need a lot more operations (not, just CRUD), we can not use it directly.

```
class NodeView(with_metaclass(MethodViewType, View)):
    """
    A PostgreSQL Object has so many operaions/functions apart from CRUD
    (Create, Read, Update, Delete):
    i.e.
    - Reversed Engineered SQL
    - Modified Query for parameter while editing object attributes
      i.e. ALTER TABLE ...
    - Statistics of the objects
    - List of dependents
    - List of dependencies
    - Listing of the children object types for the certain node
      It will used by the browser tree to get the children nodes

    This class can be inherited to achieve the diffrent routes for each of the
    object types/collections.

       OPERATION   |             URL              | HTTP Method |    Method
    ---------------+------------------------------+-------------+--------------
    List           | /obj/[Parent URL]/           | GET         | list
```

(continues on next page)

(continued from previous page)

```
    Properties      | /obj/[Parent URL]/id          | GET         | properties
    Create          | /obj/[Parent URL]/            | POST        | create
    Delete          | /obj/[Parent URL]/id          | DELETE      | delete
    Update          | /obj/[Parent URL]/id          | PUT         | update

    SQL (Reversed   | /sql/[Parent URL]/id          | GET         | sql
    Engineering)    |
    SQL (Modified   | /msql/[Parent URL]/id         | GET         | modified_sql
    Properties)     |

    Statistics      | /stats/[Parent URL]/id        | GET         | statistics
    Dependencies    | /dependency/[Parent URL]/id   | GET         | dependencies
    Dependents      | /dependent/[Parent URL]/id    | GET         | dependents

    Nodes           | /nodes/[Parent URL]/          | GET         | nodes
    Current Node    | /nodes/[Parent URL]/id        | GET         | node

    Children        | /children/[Parent URL]/id     | GET         | children

    NOTE:
    Parent URL can be seen as the path to identify the particular node.

    i.e.
    In order to identify the TABLE object, we need server -> database -> schema
    information.
    """
    operations = dict({
        'obj': [
            {'get': 'properties', 'delete': 'delete', 'put': 'update'},
            {'get': 'list', 'post': 'create'}
        ],
        'nodes': [{'get': 'node'}, {'get': 'nodes'}],
        'sql': [{'get': 'sql'}],
        'msql': [{'get': 'modified_sql'}],
        'stats': [{'get': 'statistics'}],
        'dependency': [{'get': 'dependencies'}],
        'dependent': [{'get': 'dependents'}],
        'children': [{'get': 'children'}],
        'module.js': [{}, {}, {'get': 'module_js'}]
    })

    @classmethod
    def generate_ops(cls):
        cmds = []
        for op in cls.operations:
            idx = 0
            for ops in cls.operations[op]:
                meths = []
                for meth in ops:
                    meths.append(meth.upper())
                if len(meths) > 0:
                    cmds.append({
                        'cmd': op, 'req': (idx == 0),
                        'with_id': (idx != 2), 'methods': meths
                    })
                idx += 1
        return cmds
```

(continues on next page)

(continued from previous page)

```
    # Inherited class needs to modify these parameters
    node_type = None
    # This must be an array object with attributes (type and id)
    parent_ids = []
    # This must be an array object with attributes (type and id)
    ids = []

    @classmethod
    def get_node_urls(cls):
        assert cls.node_type is not None, \
            "Please set the node_type for this class ({0})".format(
                str(cls.__class__.__name__))
        common_url = '/'
        for p in cls.parent_ids:
            common_url += '<{0}:{1}>/'.format(str(p['type']), str(p['id']))

        id_url = None
        for p in cls.ids:
            id_url = '{0}<{1}:{2}>'.format(common_url if not id_url else id_url,
                                           p['type'], p['id'])

        return id_url, common_url

    def __init__(self, **kwargs):
        self.cmd = kwargs['cmd']

    # Check the existance of all the required arguments from parent_ids
    # and return combination of has parent arguments, and has id arguments
    def check_args(self, **kwargs):
        has_id = has_args = True
        for p in self.parent_ids:
            if p['id'] not in kwargs:
                has_args = False
                break

        for p in self.ids:
            if p['id'] not in kwargs:
                has_id = False
                break

        return has_args, has_id and has_args

    def dispatch_request(self, *args, **kwargs):
        meth = flask.request.method.lower()
        if meth == 'head':
            meth = 'get'

        assert self.cmd in self.operations, \
            "Unimplemented command ({0}) for {1}".format(
                self.cmd,
                str(self.__class__.__name__)
            )

        has_args, has_id = self.check_args(**kwargs)

        assert (self.cmd in self.operations and
```

(continues on next page)

(continued from previous page)

```
                (has_id and len(self.operations[self.cmd]) > 0 and
                 meth in self.operations[self.cmd][0]) or
                (not has_id and len(self.operations[self.cmd]) > 1 and
                 meth in self.operations[self.cmd][1]) or
                (len(self.operations[self.cmd]) > 2 and
                 meth in self.operations[self.cmd][2])), \
            "Unimplemented method ({0}) for command ({1}), which {2} an id".format(
                meth, self.cmd,
                'requires' if has_id else 'does not require'
            )

        meth = self.operations[self.cmd][0][meth] if has_id else \
            self.operations[self.cmd][1][meth] if has_args and \
                                                  meth in self.operations[self.
↪cmd][1] else \
                self.operations[self.cmd][2][meth]

        method = getattr(self, meth, None)

        if method is None:
            return make_json_response(
                status=406,
                success=0,
                errormsg=gettext(
                    "Unimplemented method ({0}) for this url ({1})".format(
                        meth, flask.request.path)
                )
            )

        return method(*args, **kwargs)

    @classmethod
    def register_node_view(cls, blueprint):
        cls.blueprint = blueprint
        id_url, url = cls.get_node_urls()

        commands = cls.generate_ops()

        for c in commands:
            if c['with_id']:
                blueprint.add_url_rule(
                    '/{0}{1}'.format(
                        c['cmd'], id_url if c['req'] else url
                    ),
                    view_func=cls.as_view(
                        '{0}{1}'.format(
                            c['cmd'], '_id' if c['req'] else ''
                        ),
                        cmd=c['cmd']
                    ),
                    methods=c['methods']
                )
            else:
                blueprint.add_url_rule(
                    '/{0}'.format(c['cmd']),
                    view_func=cls.as_view(
                        '{0}'.format(c['cmd']), cmd=c['cmd']
```

(continues on next page)

(continued from previous page)

```
                    ),
                    methods=c['methods']
                )

    def module_js(self, **kwargs):
        """
        This property defines (if javascript) exists for this node.
        Override this property for your own logic.
        """
        return flask.make_response(
            flask.render_template(
                "{0}/js/{0}.js".format(self.node_type)
            ),
            200, {'Content-Type': 'application/x-javascript'}
        )

    def children(self, *args, **kwargs):
        """Build a list of treeview nodes from the child nodes."""
        children = []

        for module in self.blueprint.submodules:
            children.extend(module.get_nodes(*args, **kwargs))
        # Return sorted nodes based on label
        return make_json_response(
            data=sorted(
                children, key=lambda c: c['label']
            )
        )
```

9.4.3 BaseDriver

```
class BaseDriver(object):
    """
    class BaseDriver(object):

    This is a base class for different server types.
    Inherit this class to implement different type of database driver
    implementation.

    (For PostgreSQL/EDB Postgres Advanced Server, we will be using psycopg2)

    Abstract Properties:
    -------- ----------
    * Version (string):
        Current version string for the database server

    Abstract Methods:
    -------- -------
    * get_connection(*args, **kwargs)
    - It should return a Connection class object, which may/may not be
      connected to the database server.

    * release_connection(*args, **kwargs)
    - Implement the connection release logic
```

(continues on next page)

(continued from previous page)

```
    * gc()
    - Implement this function to release the connections assigned in the
      session, which has not been pinged from more than the idle timeout
      configuration.
    """

    @abstractproperty
    def Version(cls):
        pass

    @abstractmethod
    def get_connection(self, *args, **kwargs):
        pass

    @abstractmethod
    def release_connection(self, *args, **kwargs):
        pass

    @abstractmethod
    def gc(self):
        pass
```

9.4.4 BaseConnection

```
class BaseConnection(object):
    """
    class BaseConnection(object)

        It is a base class for database connection. A different connection
        drive must implement this to expose abstract methods for this server.

        General idea is to create a wrapper around the actual driver
        implementation. It will be instantiated by the driver factory
        basically. And, they should not be instantiated directly.


    Abstract Methods:
    -------- -------
    * connect(**kwargs)
      - Define this method to connect the server using that particular driver
        implementation.

    * execute_scalar(query, params, formatted_exception_msg)
      - Implement this method to execute the given query and returns single
        datum result.

    * execute_async(query, params, formatted_exception_msg)
      - Implement this method to execute the given query asynchronously and returns␣
↪result.

    * execute_void(query, params, formatted_exception_msg)
      - Implement this method to execute the given query with no result.

    * execute_2darray(query, params, formatted_exception_msg)
      - Implement this method to execute the given query and returns the result
```

(continues on next page)

(continued from previous page)

```
    as a 2 dimensional array.

* execute_dict(query, params, formatted_exception_msg)
  - Implement this method to execute the given query and returns the result
    as an array of dict (column name -> value) format.

* connected()
  - Implement this method to get the status of the connection. It should
    return True for connected, otherwise False

* reset()
  - Implement this method to reconnect the database server (if possible)

* transaction_status()
  - Implement this method to get the transaction status for this
    connection. Range of return values different for each driver type.

* ping()
  - Implement this method to ping the server. There are times, a connection
    has been lost, but - the connection driver does not know about it. This
    can be helpful to figure out the actual reason for query failure.

* _release()
  - Implement this method to release the connection object. This should not
    be directly called using the connection object itself.

  NOTE: Please use BaseDriver.release_connection(...) for releasing the
        connection object for better memory management, and connection pool
        management.

* _wait(conn)
  - Implement this method to wait for asynchronous connection to finish the
    execution, hence - it must be a blocking call.

* _wait_timeout(conn, time)
  - Implement this method to wait for asynchronous connection with timeout.
    This must be a non blocking call.

* poll(formatted_exception_msg)
  - Implement this method to poll the data of query running on asynchronous
    connection.

* cancel_transaction(conn_id, did=None)
  - Implement this method to cancel the running transaction.

* messages()
  - Implement this method to return the list of the messages/notices from
    the database server.

* rows_affected()
  - Implement this method to get the rows affected by the last command
    executed on the server.
"""

ASYNC_OK = 1
ASYNC_READ_TIMEOUT = 2
ASYNC_WRITE_TIMEOUT = 3
```

(continues on next page)

(continued from previous page)

```
    ASYNC_NOT_CONNECTED = 4
    ASYNC_EXECUTION_ABORTED = 5

    @abstractmethod
    def connect(self, **kwargs):
        pass

    @abstractmethod
    def execute_scalar(self, query, params=None, formatted_exception_msg=False):
        pass

    @abstractmethod
    def execute_async(self, query, params=None, formatted_exception_msg=True):
        pass

    @abstractmethod
    def execute_void(self, query, params=None, formatted_exception_msg=False):
        pass

    @abstractmethod
    def execute_2darray(self, query, params=None, formatted_exception_msg=False):
        pass

    @abstractmethod
    def execute_dict(self, query, params=None, formatted_exception_msg=False):
        pass

    @abstractmethod
    def connected(self):
        pass

    @abstractmethod
    def reset(self):
        pass

    @abstractmethod
    def transaction_status(self):
        pass

    @abstractmethod
    def ping(self):
        pass

    @abstractmethod
    def _release(self):
        pass

    @abstractmethod
    def _wait(self, conn):
        pass

    @abstractmethod
    def _wait_timeout(self, conn, time):
        pass

    @abstractmethod
    def poll(self, formatted_exception_msg=True):
```

(continues on next page)

(continued from previous page)

```
        pass

    @abstractmethod
    def status_message(self):
        pass

    @abstractmethod
    def rows_affected(self):
        pass

    @abstractmethod
    def cancel_transaction(self, conn_id, did=None):
        pass
```

9.5 Code Review Notes

This document lists a number of standard items that will be checked during the review process for any patches submitted for inclusion in pgAdmin.

- Ensure all code follows the pgAdmin *Coding Standards*.
- Copyright years must be correct and properly formatted (to make it easy to make bulk updates every year). The start date should always be 2013, and the end year the current year, e.g.

  Copyright (C) 2013 - 2017, The pgAdmin Development Team
- Ensure there's a blank line immediately following any copyright headers.
- Include PyDoc comments for functions, classes and modules. Node modules should be """Implements the XXXX node""".
- Ensure that any generated SQL does not have any leading or trailing blank lines and consistently uses 4 space indents for nice formatting.
- Don't special-case any Slony objects. pgAdmin 4 will have no direct knowledge of Slony, unlike pgAdmin 3.
- If you copy/paste modules, please ensure any comments are properly updated.
- Read all comments, and ensure they make sense and provide useful commentary on the code.
- Ensure that field labels both use PostgreSQL parlance, but also are descriptive. A good example is the "Init" field on an FTS Template - Init is the PG term, but adding the word "Function" after it makes it much more descriptive.
- Re-use code whereever possible, but factor it out into a suitably central location - don't copy and paste it unless modifications are required!
- Format code nicely to make it readable. Break up logical chunks of code with blank lines, and comment well to describe what different sections of code are for or pertain to.
- Ensure that form validation works correctly and is consistent with other dialogues in the way errors are displayed.
- On dialogues with Schema or Owner fields, pre-set the default values to the current schema/user as appropriate. In general, if there are common or sensible default values available, put them in the fields for the user.
- 1 patch == 1 feature. If you need to fix/update existing infrastructure in your patch, it's usually easier if it's in a separate patch. Patches containing multiple new features or unrelated changes are likely to be rejected.

- Ensure the patch is fully functional, and works! If a patch is being sent as a work in progress, not intended for commit, clearly state that it's a WIP, and note what does or does not yet work.

9.6 Translations

pgAdmin supports multiple languages using the Flask-Babel Python module. A list of supported languages is included in the **web/config.py** configuration file and must be updated whenever languages are added or removed with ISO 639-1 (two letter) language codes. The codes are named **$LANG** in this document.

9.6.1 Translation Marking

Strings can be marked for translation in either Python code (using **gettext**()) or Jinja templates (using _()). Here are some examples that show how this is achieved.

Python:

```
errormsg = gettext('No server group name was specified')
```

Jinja:

```
<input type="submit" value="{{ _('Change Password') }}">
```

```
<title>{{ _('%(appname)s Password Change', appname=config.APP_NAME) }}</title>
```

```
define(['sources/gettext', ...], function(gettext, ...){
    ...
    var alert = alertify.prompt(
        gettext('Password Change'),
        gettext('New password for %(userName)s', {userName: 'jsmith' }),
        ...
    )
})
```

9.6.2 Updating and Merging

Whenever new strings are added to the application, the template catalogue (**web/pgadmin/messages.pot**) and the existing translation catalogues (**web/pgadmin/translations/$LANG/LC_MESSAGES/messages.po**) must be updated and compiled. This can be achieved using the following commands from the **web** directory in the Python virtual environment for pgAdmin:

```
(pgadmin4) user$ pybabel extract -F babel.cfg -o pgadmin/messages.pot pgadmin
```

Once the template has been updated it needs to be merged into the existing message catalogues:

```
(pgadmin4) user$ pybabel update -i pgadmin/messages.pot -d pgadmin/translations
```

Finally, the message catalogues can be compiled for use:

```
(pgadmin4) user$ pybabel compile -d pgadmin/translations
```

9.6.3 Adding a New Language

Adding a new language is simple. First, add the language name and identifier to **web/config.py**:

```
# Languages we support in the UI
LANGUAGES = {
    'en': 'English',
    'zh': 'Chinese (Simplified)',
    'de': 'German',
    'pl': 'Polish'
}
```

Then, create the new message catalogue from the **web** directory in the source tree in the Python virtual environment for pgAdmin:

```
(pgadmin4) user$ pybabel init -i pgadmin/messages.pot -d pgadmin/translations -l $LANG
```

CHAPTER

TEN

PGAGENT

pgAgent is a job scheduling agent for Postgres databases, capable of running multi-step batch or shell scripts and SQL tasks on complex schedules.

pgAgent is distributed independently of pgAdmin. You can download pgAgent from the download area of the pgAdmin website.

Contents:

10.1 Using pgAgent

pgAgent is a scheduling agent that runs and manages jobs; each job consists of one or more steps and schedules. If two or more jobs are scheduled to execute concurrently, pgAgent will execute the jobs in parallel (each with it's own thread).

A step may be a series of SQL statements or an operating system batch/shell script. Each step in a given job is executed when the previous step completes, in alphanumeric order by name. Switches on the *pgAgent Job* dialog (accessed through the *Properties* context menu) allow you to modify a job, enabling or disabling individual steps as needed.

Each job is executed according to one or more schedules. Each time the job or any of its schedules are altered, the next runtime of the job is re-calculated. Each instance of pgAgent periodically polls the database for jobs with the next runtime value in the past. By polling at least once every minute, all jobs will normally start within one minute of the specified start time. If no pgAgent instance is running at the next runtime of a job, it will run as soon as pgAgent is next started, following which it will return to the normal schedule.

When you highlight the name of a defined job in the pgAdmin tree control, the *Properties* tab of the main pgAdmin window will display details about the job, and the *Statistics* tab will display details about the job's execution.

10.1.1 Security concerns

pgAgent is a very powerful tool, but does have some security considerations that you should be aware of:

Database password - *DO NOT* be tempted to include a password in the pgAgent connection string - on Unix systems it may be visible to all users in 'ps' output, and on Windows systems it will be stored in the registry in plain text. Instead, use a libpq *~/.pgpass* file to store the passwords for every database that pgAgent must access. Details of this technique may be found in the PostgreSQL documentation on .pgpass file.

System/database access - all jobs run by pgAgent will run with the security privileges of the pgAgent user. SQL steps will run as the user that pgAgent connects to the database as, and batch/shell scripts will run as the operating system user that the pgAgent service or daemon is running under. Because of this, it is essential to maintain control over the users that are able to create and modify jobs. By default, only the user that created the pgAgent database objects will be able to do this - this will normally be the PostgreSQL superuser.

10.2 Installing pgAgent

pgAgent runs as a daemon on Unix systems, and a service on Windows systems. In most cases it will run on the database server itself - for this reason, pgAgent is not automatically configured when pgAdmin is installed. In some cases however, it may be preferable to run pgAgent on multiple systems, against the same database; individual jobs may be targeted at a particular host, or left for execution by any host. Locking prevents execution of the same instance of a job by multiple hosts.

10.2.1 Database setup

Before using pgAdmin to manage pgAgent, you must create the pgAgent extension in the maintenance database registered with pgAdmin. To install pgAgent on a PostgreSQL host, connect to the *postgres* database, and navigate through the *Tools* menu to open the Query tool. For server versions 9.1 or later, and pgAgent 3.4.0 or later, enter the following command in the query window, and click the *Execute* icon:

```
CREATE EXTENSION pgagent;
```

This command will create a number of tables and other objects in a schema called 'pgagent'.

The database must also have the pl/pgsql procedural language installed - use the PostgreSQL CREATE LANGUAGE command to install pl/pgsql if necessary. To install pl/pgsql, enter the following command in the query window, and click the *Execute* icon:

```
CREATE LANGUAGE plpgsql;
```

If you are using an earlier version of PostgreSQL or pgAgent, use the *Open file* icon on the Query Tool toolbar to open a browse

- On Windows, it is usually located under *C:Program filespgAdmin III* (or *C:Program filesPostgreSQL8.xpgAdmin III* if installed with the PostgreSQL server installer).
- On Linux, it is usually located under */usr/local/pgadmin3/share/pgadmin3* or */usr/share/pgadmin3*.

After loading the file into the Query Tool, click the *Execute* icon to execute the script. The script will create a number of tables and other objects in a schema named *pgagent*.

10.2.2 Daemon installation on Unix

To install the pgAgent daemon on a Unix system, you will normally need to have root privileges to modify the system startup scripts. Modifying system startup scripts is quite system-specific so you should consult your system documentation for further information.

The program itself takes few command line options, most of which are only needed for debugging or specialised configurations:

```
Usage:
  /path/to/pgagent [options] <connect-string>

options:
  -f run in the foreground (do not detach from the terminal)
  -t <poll time interval in seconds (default 10)>
  -r <retry period after connection abort in seconds (>=10, default 30)>
  -s <log file (messages are logged to STDOUT if not specified)>
  -l <logging verbosity (ERROR=0, WARNING=1, DEBUG=2, default 0)>
```

The connection string is a standard PostgreSQL libpq connection string (see the PostgreSQL documentation on the connection string for further details). For example, the following command line will run pgAgent against a server listening on the localhost, using a database called 'pgadmin', connecting as the user 'postgres':

```
/path/to/pgagent hostaddr=127.0.0.1 dbname=postgres user=postgres
```

10.2.3 Service installation on Windows

pgAgent can install itself as a service on Windows systems. The command line options available are similar to those on Unix systems, but include an additional parameter to tell the service what to do:

```
Usage:
  pgAgent REMOVE <serviceName>
  pgAgent INSTALL <serviceName> [options] <connect-string>
  pgAgent DEBUG [options] <connect-string>

  options:
    -u <user or DOMAIN\user>
    -p <password>
    -d <displayname>
    -t <poll time interval in seconds (default 10)>
    -r <retry period after connection abort in seconds (>=10, default 30)>
    -l <logging verbosity (ERROR=0, WARNING=1, DEBUG=2, default 0)>
```

The service may be quite simply installed from the command line as follows (adjust the path as required):

```
"C:\Program Files\pgAdmin III\pgAgent" INSTALL pgAgent -u postgres -p secret␣
↪hostaddr=127.0.0.1 dbname=postgres user=postgres
```

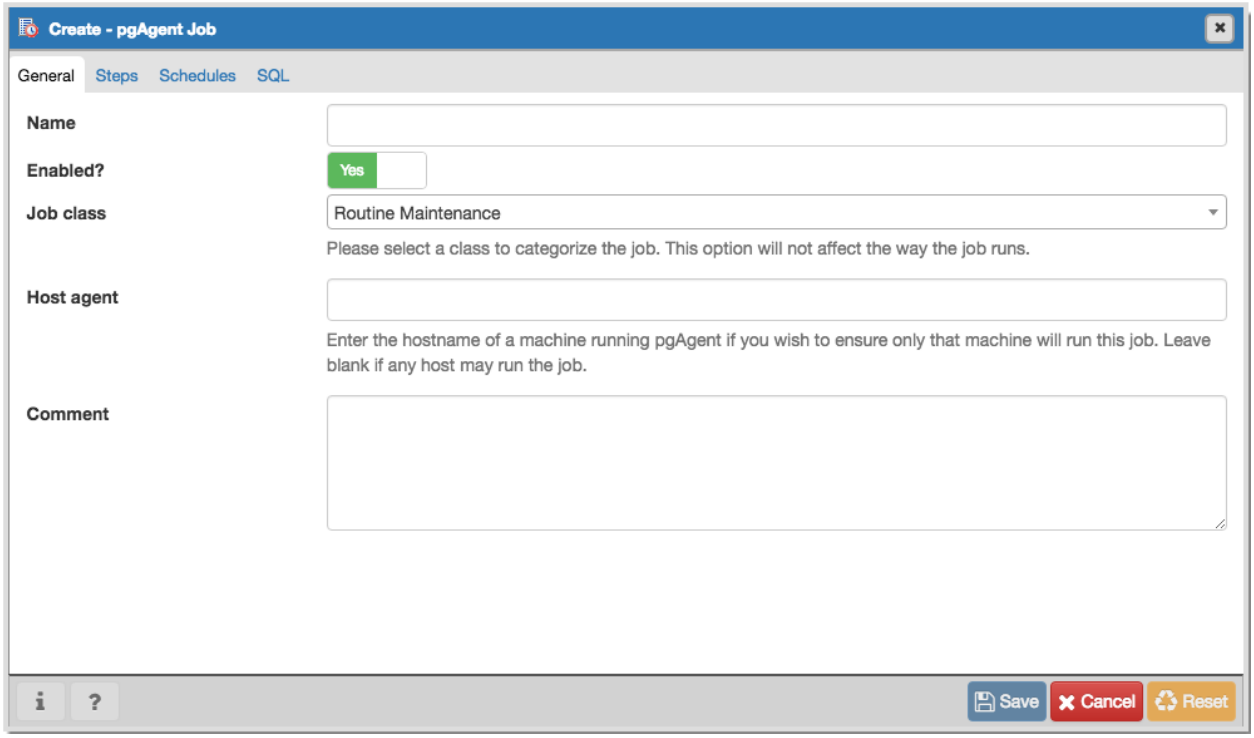
You can then start the service at the command line using *net start pgAgent*, or from the *Services* control panel applet. Any logging output or errors will be reported in the Application event log. The DEBUG mode may be used to run pgAgent from a command prompt. When run this way, log messages will output to the command window.

10.3 Creating a pgAgent Job

pgAgent is a scheduling agent that runs and manages jobs; each job consists of steps and schedules.

To create or manage a job, use the pgAdmin tree control to browse to the server on which the pgAgent database objects were created. The tree control will display a *pgAgent Jobs* node, under which currently defined jobs are displayed. To add a new job, right click on the *pgAgent Jobs* node, and select *Create pgAgent Job...* from the context menu.

When the pgAgent dialog opens, use the tabs on the *pgAgent Job* dialog to define the steps and schedule that make up a pgAgent job.



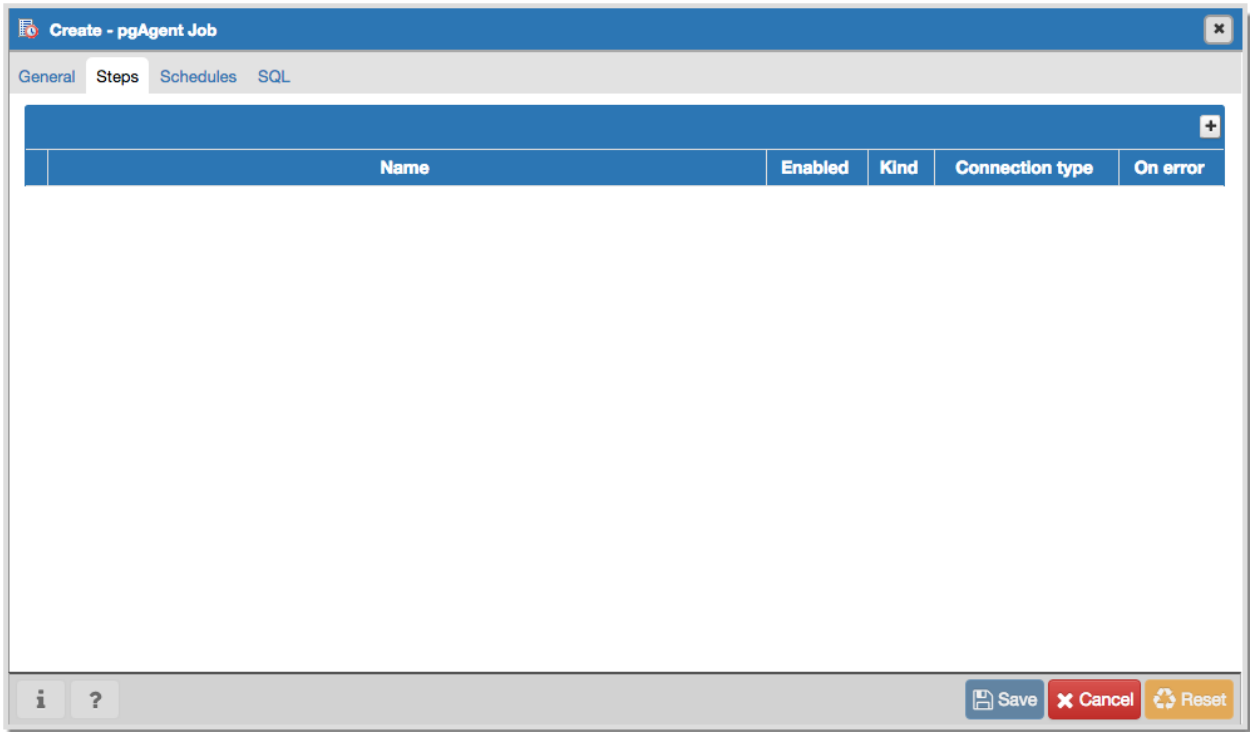
Use the fields on the *General* tab to provide general information about a job:

- Provide a name for the job in the *Name* field.
- Move the *Enabled* switch to the *Yes* position to enable a job, or *No* to disable a job.
- Use the *Job Class* drop-down to select a class (for job categorization).
- Use the *Host Agent* field to specify the name of a machine that is running pgAgent to indicate that only that machine may execute the job. Leave the field blank to specify that any machine may perform the job.

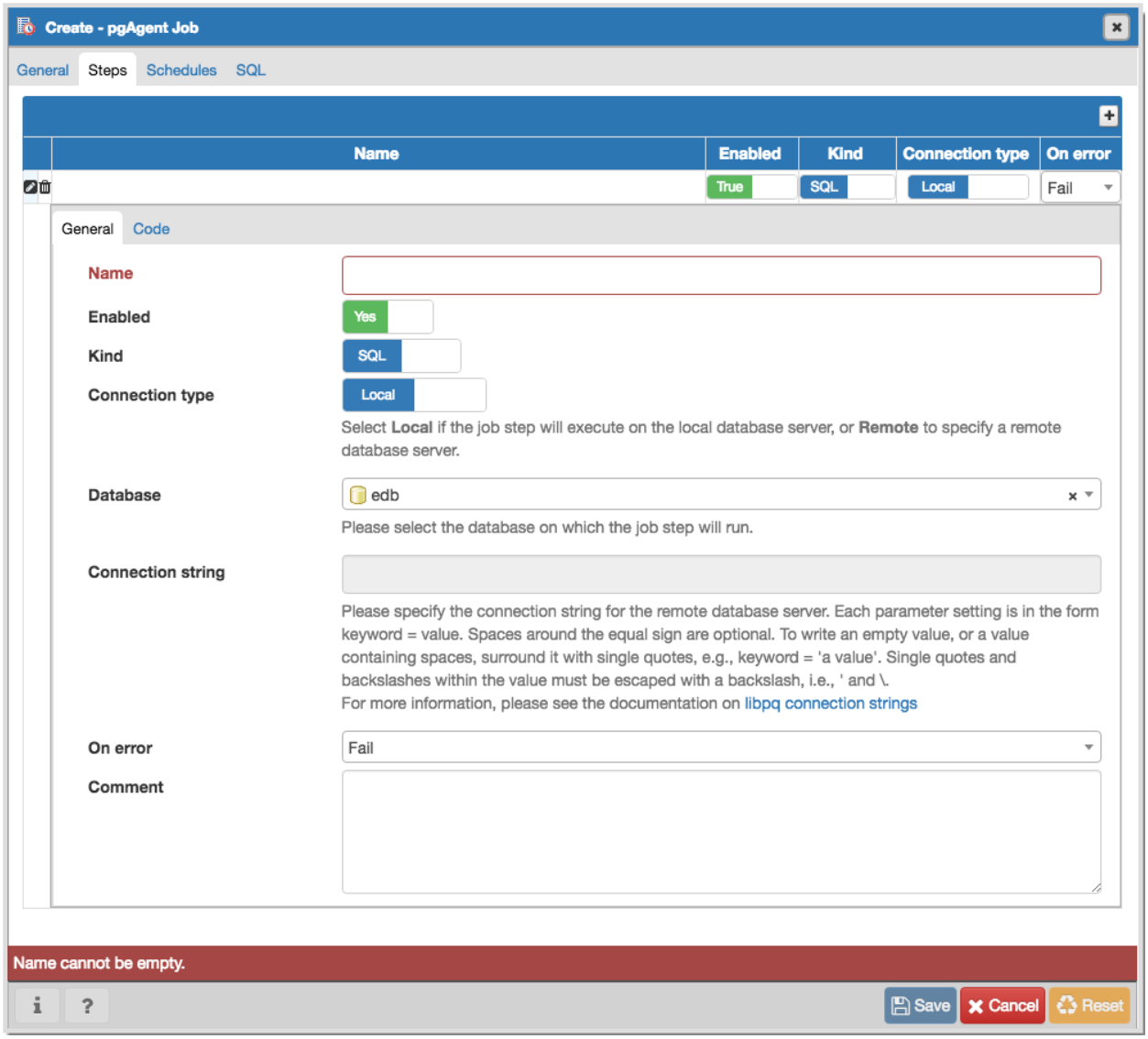
  Note: It is not always obvious what value to specify for the Host Agent in order to target a job step to a specific machine. With pgAgent running on the required machines and connected to the scheduler database, you can use the following query to view the hostnames as reported by each agent:

  ```
  SELECT jagstation FROM pgagent.pga_jobagent
  ```

  Use the hostname exactly as reported by the query in the Host Agent field.
- Use the *Comment* field to store notes about the job.



Use the *Steps* tab to define and manage the steps that the job will perform. Click the Add icon (+) to add a new step; then click the compose icon (located at the left side of the header) to open the step definition dialog:

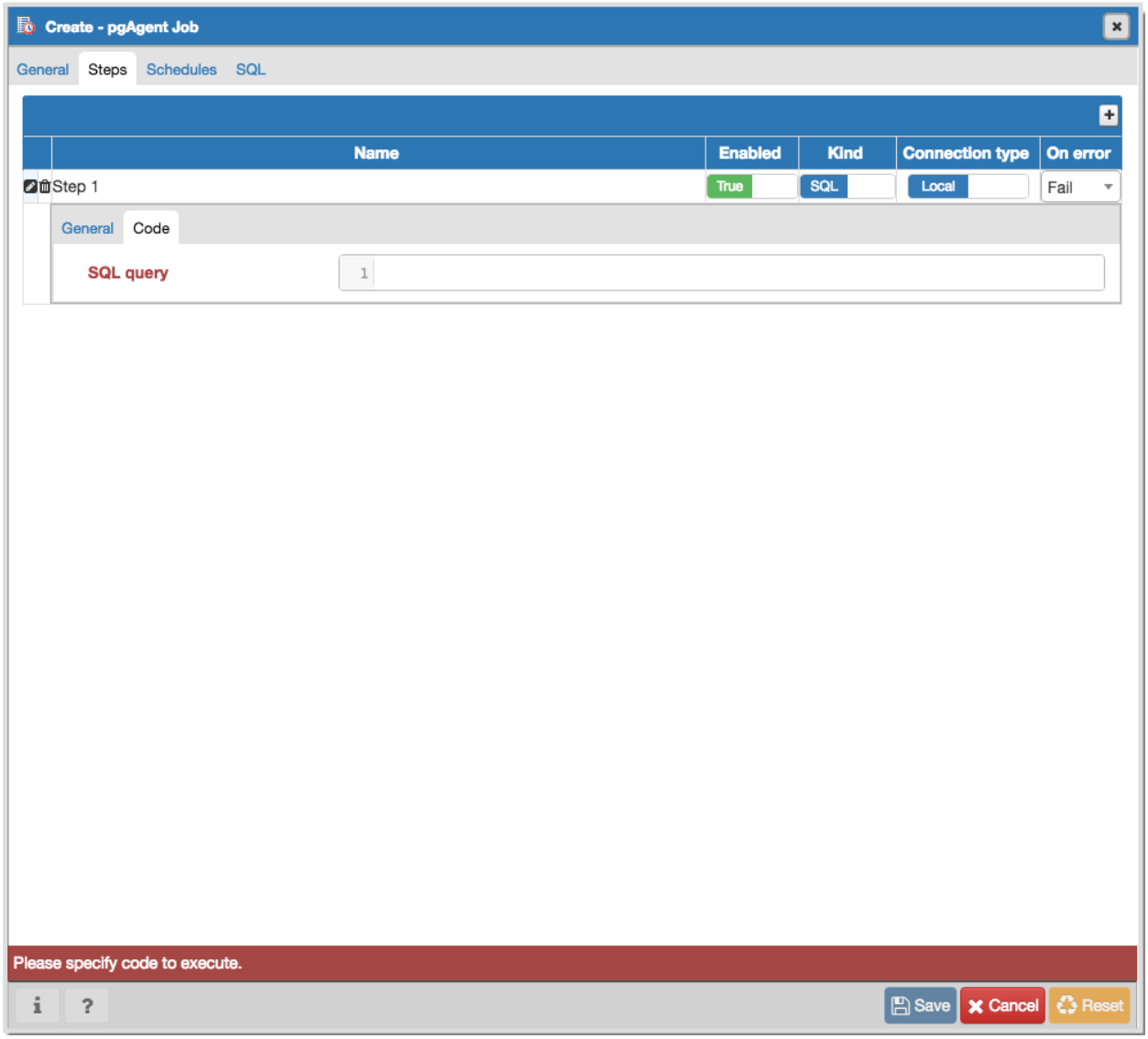


Use fields on the step definition dialog to define the step:

- Provide a name for the step in the *Name* field; please note that steps will be performed in alphanumeric order by name.
- Use the *Enabled* switch to include the step when executing the job (*True*) or to disable the step (*False*).
- Use the *Kind* switch to indicate if the job step invokes SQL code (*SQL*) or a batch script (*Batch*).
- If you select *SQL*, use the *Code* tab to provide SQL code for the step.
- If you select *Batch*, use the *Code* tab to provide the batch script that will be executed during the step.
- Use the *Connection type* switch to indicate if the step is performed on a local server (*Local*) or on a remote host (*Remote*). If you specify a remote connection should be used for the step, the *Connection string* field will be enabled, and you must provide a libpq-style connection string.
- Use the *Database* drop-down to select the database on which the job step will be performed.
- Use the *Connection string* field to specify a libpq-style connection string to the remote server on which the step will be performed. For more information about writing a connection string, please

see the PostgreSQL documentation.

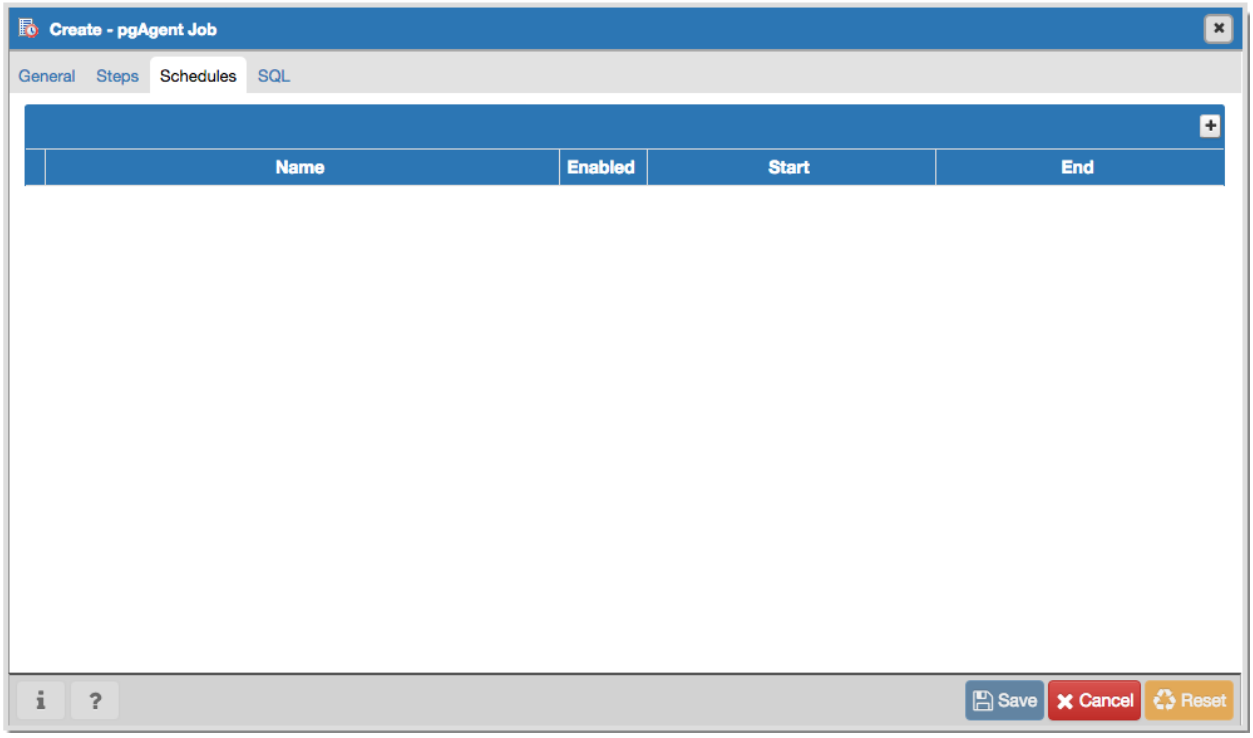
- Use the *On error* drop-down to specify the behavior of pgAgent if it encounters an error while executing the step. Select from:
- *Fail* - Stop the job if you encounter an error while processing this step.
- *Success* - Mark the step as completing successfully, and continue.
- *Ignore* - Ignore the error, and continue.
- Use the *Comment* field to provide a comment about the step.



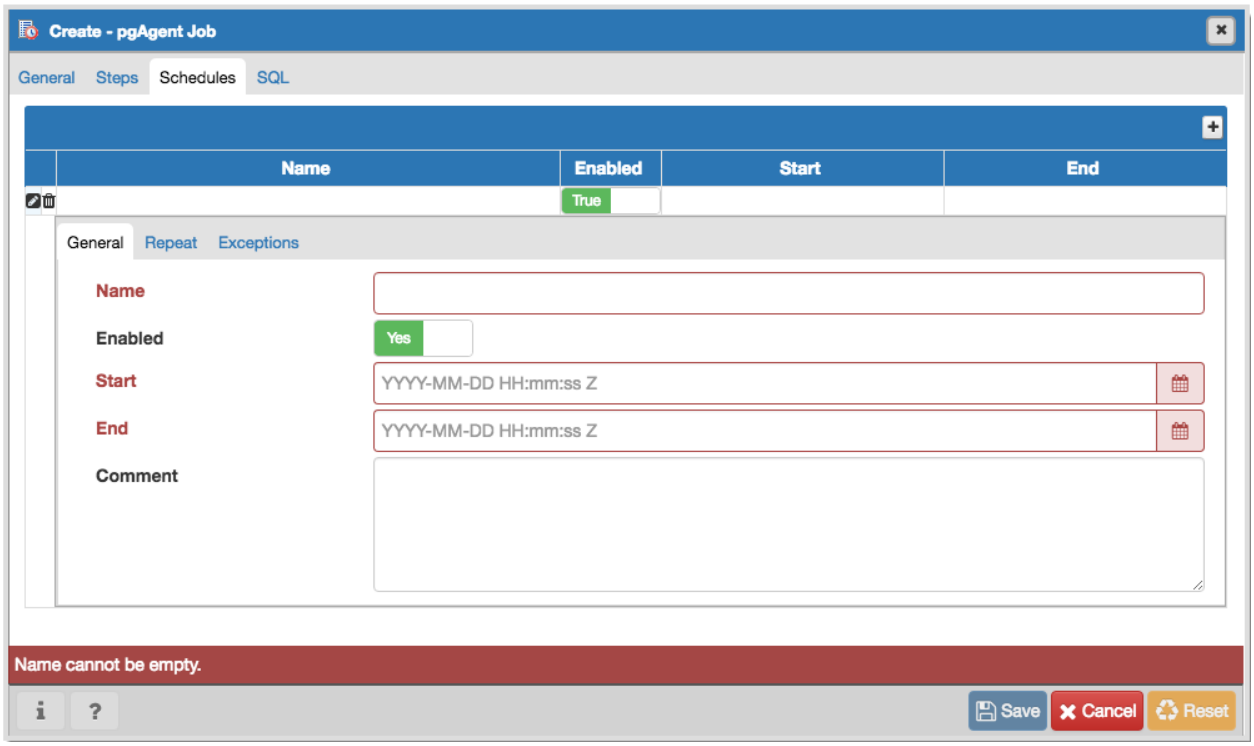
Use the context-sensitive field on the step definition dialog's *Code* tab to provide the SQL code or batch script that will be executed during the step:

- If the step invokes SQL code, provide one or more SQL statements in the *SQL query* field.
- If the step performs a batch script, provide the script in the *Script* field. If you are running on a Windows server, standard batch file syntax must be used. When running on a Linux server, any shell script may be used, provided that a suitable interpreter is specified on the first line (e.g. *#!/bin/sh*).

When you've provided all of the information required by the step, click the compose icon to close the step definition dialog. Click the add icon (+) to add each additional step, or select the *Schedules* tab to define the job schedule.



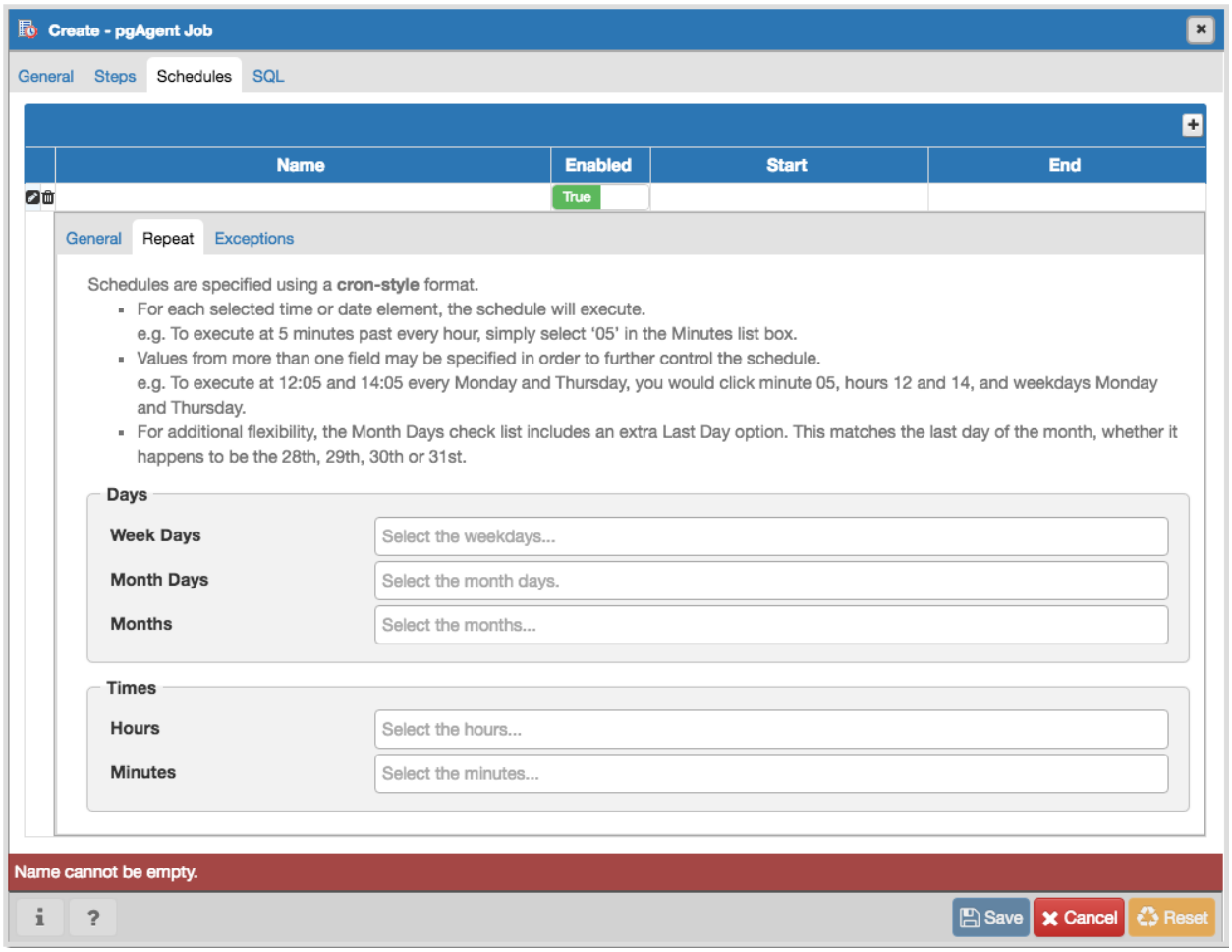
Click the Add icon (+) to add a schedule for the job; then click the compose icon (located at the left side of the header) to open the schedule definition dialog:



Use the fields on the schedule definition tab to specify the days and times at which the job will execute.

- Provide a name for the schedule in the *Name* field.
- Use the *Enabled* switch to indicate that pgAgent should use the schedule (*Yes*) or to disable the schedule (*No*).
- Use the calendar selector in the *Start* field to specify the starting date and time for the schedule.
- Use the calendar selector in the *End* field to specify the ending date and time for the schedule.
- Use the *Comment* field to provide a comment about the schedule.

Select the *Repeat* tab to define the days on which the schedule will execute.



Use the fields on the *Repeat* tab to specify the details about the schedule in a cron-style format. The job will execute on each date or time element selected on the *Repeat* tab.

Click within a field to open a list of valid values for that field; click on a specific value to add that value to the list of selected values for the field. To clear the values from a field, click the X located at the right-side of the field.

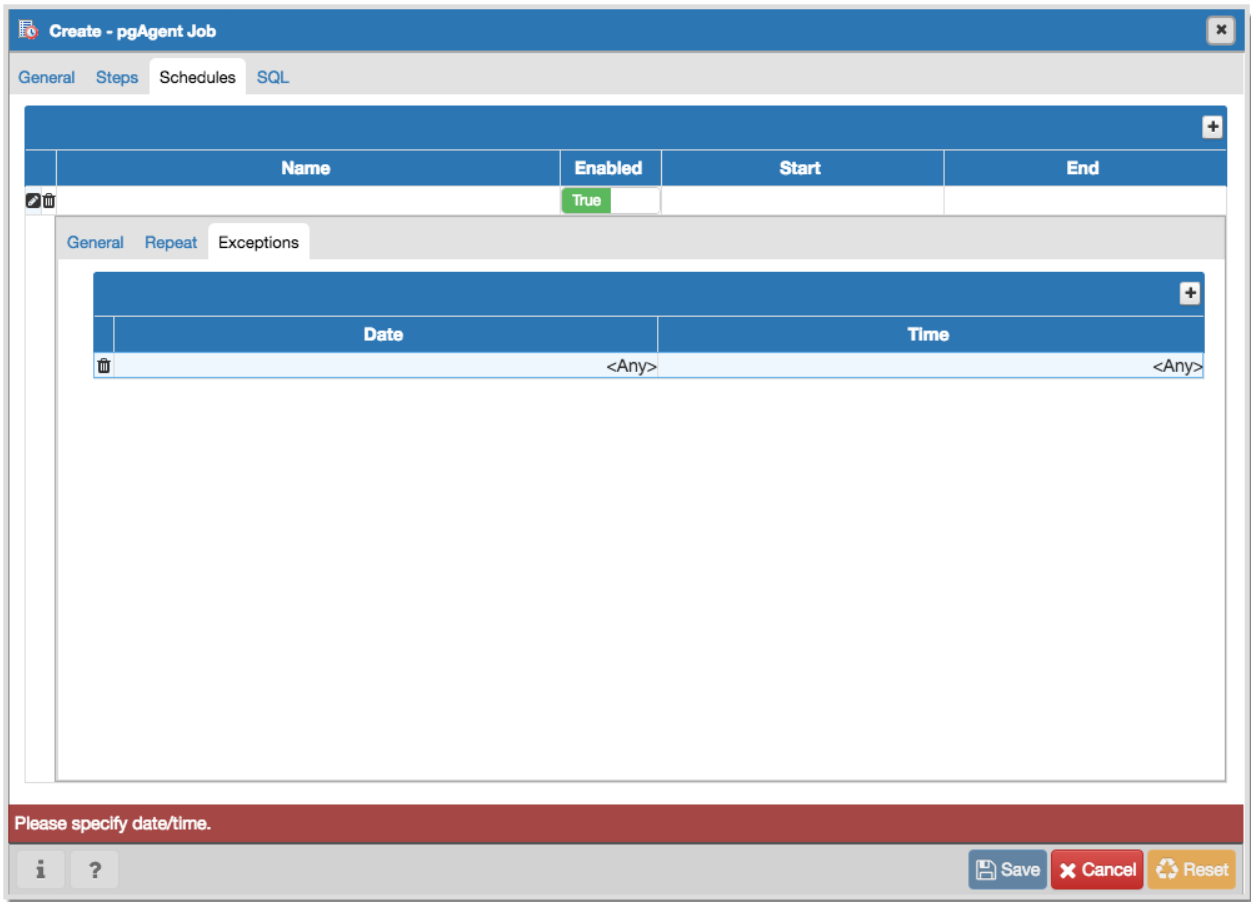
Use the fields within the *Days* box to specify the days on which the job will execute:

- Use the *Week Days* field to select the days on which the job will execute.
- Use the *Month Days* field to select the numeric days on which the job will execute. Specify the *Last Day* to indicate that the job should be performed on the last day of the month, irregardless of the date.
- Use the *Months* field to select the months in which the job will execute.

Use the fields within the *Times* box to specify the times at which the job will execute:

- Use the *Hours* field to select the hour at which the job will execute.
- Use the *Minutes* field to select the minute at which the job will execute.

Select the *Exceptions* tab to specify any days on which the schedule will *not* execute.

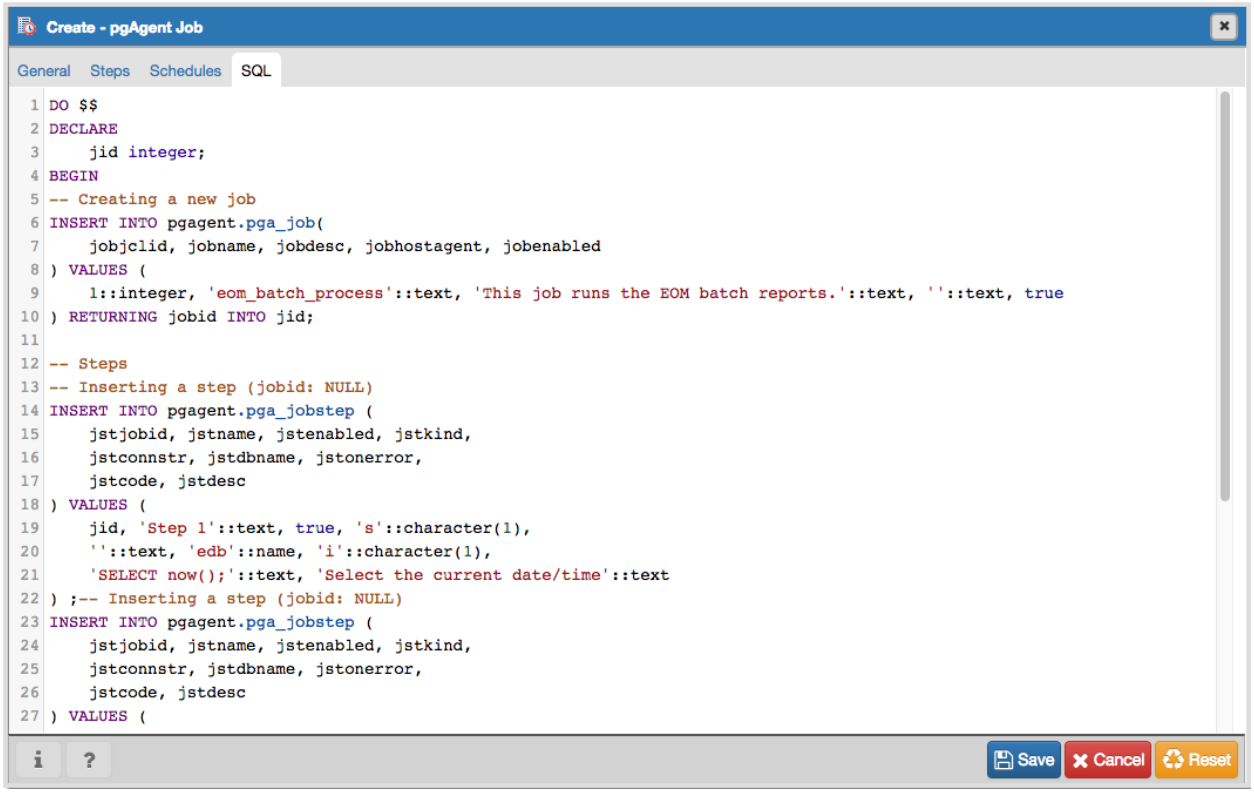


Use the fields on the *Exceptions* tab to specify days on which you wish the job to not execute; for example, you may wish for jobs to not execute on national holidays.

Click the Add icon (+) to add a row to the exception table, then:

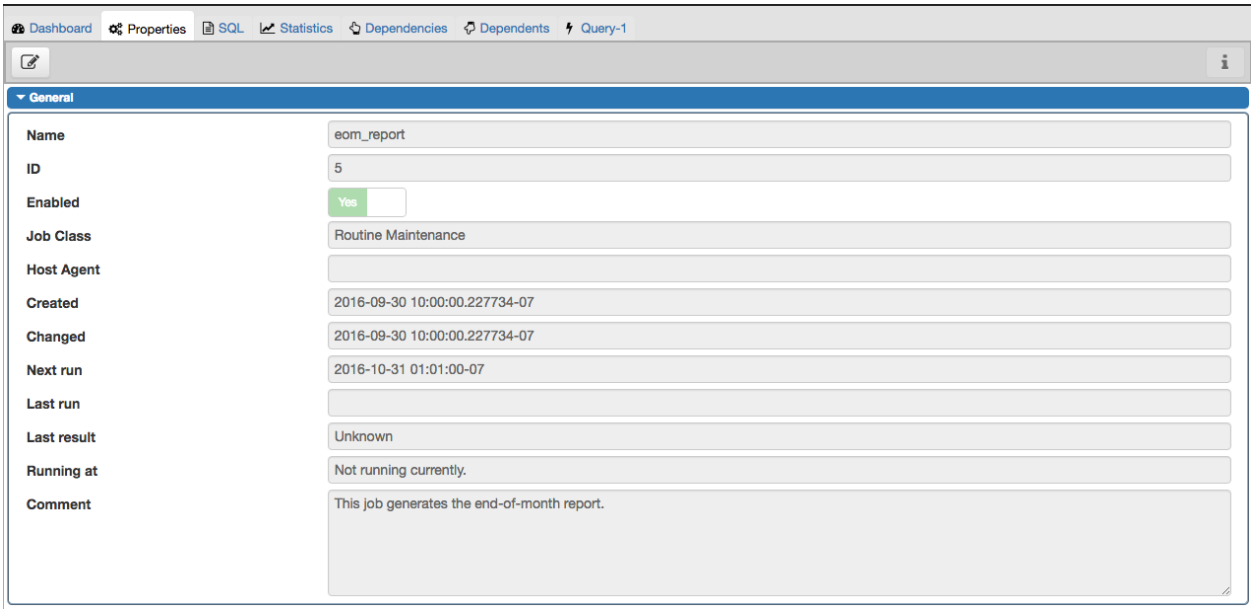
- Click within the *Date* column to open a calendar selector, and select a date on which the job will not execute. Specify *<Any>* in the *Date* column to indicate that the job should not execute on any day at the time selected.
- Click within the *Time* column to open a time selector, and specify a time on which the job will not execute. Specify *<Any>* in the *Time* column to indicate that the job should not execute at any time on the day selected.

When you've finished defining the schedule, you can use the *SQL* tab to review the code that will create or modify your job.



Click the *Save* button to save the job definition, or *Cancel* to exit the job without saving. Use the *Reset* button to remove your unsaved entries from the dialog.

After saving a job, the job will be listed under the *pgAgent Jobs* node of the pgAdmin tree control of the server on which it was defined. The *Properties* tab in the main pgAdmin window will display a high-level overview of the selected job, and the *Statistics* tab will show the details of each run of the job.



To modify an existing job or to review detailed information about a job, right-click on a job name, and select *Properties* from the context menu.

CHAPTER

ELEVEN

LICENCE

pgAdmin is released under the PostgreSQL Licence, which is a liberal Open Source licence similar to BSD or MIT, and approved by the Open Source Initiative. The copyright for the project source code, website and documentation is attributed to the pgAdmin Development Team.