Advanced Server JDBC Connector Guide

Version 42.2.9.1

Built at 2020-12-17T15:34:14

1 What's New

The following features are added to create Advanced Server JDBC Connector 42.2.9.1:

- The EDB JDBC Connector is now supported on Red Hat Enterprise Linux and CentOS (x86_64) 8.x.

2 Requirements Overview

Supported Versions

The Advanced Server JDBC Connector is certified with Advanced Server version 9.4 and above.

Supported Platforms

The Advanced Server JDBC Connector native packages are supported on the following 64 bit Linux platforms:

- Red Hat Enterprise Linux and CentOS (x86_64) 6.x, 7.x and 8.x.
- OEL Linux 6.x and 7.x
- PPC-LE 8 running RHEL or CentOS 7.x
- SLES 12.x
- Debian 9.x
- Ubuntu 18.04

The Advanced Server JDBC Connector graphical installers are supported on the following Windows platforms:

64-bit Windows:

- Windows Server 2019
- Windows Server 2016
- Windows Server 2012 R2

32-bit Windows:

- Windows 10
- Windows 8
- Windows 7

3 Advanced Server JDBC Connector Overview

Sun Microsystems created a standardized interface for connecting Java applications to databases known as Java Database Connectivity (JDBC). The Advanced Server JDBC Connector connects a Java application to a Postgres database.

JDBC Driver Types

There are currently four different types of JDBC drivers, each with their own specific implementation, use and limitations. The Advanced Server JDBC Connector is a Type 4 driver.

Type 1 Driver

- This driver type is the JDBC-ODBC bridge.
- It is limited to running locally.
- Must have ODBC installed on computer.
- Must have ODBC driver for specific database installed on computer.
- Generally can’t run inside an applet because of Native Method calls.

Type 2 Driver

- This is the native database library driver.
- Uses Native Database library on computer to access database.
- Generally can’t run inside an applet because of Native Method calls.
- Must have database library installed on client.

Type 3 Driver

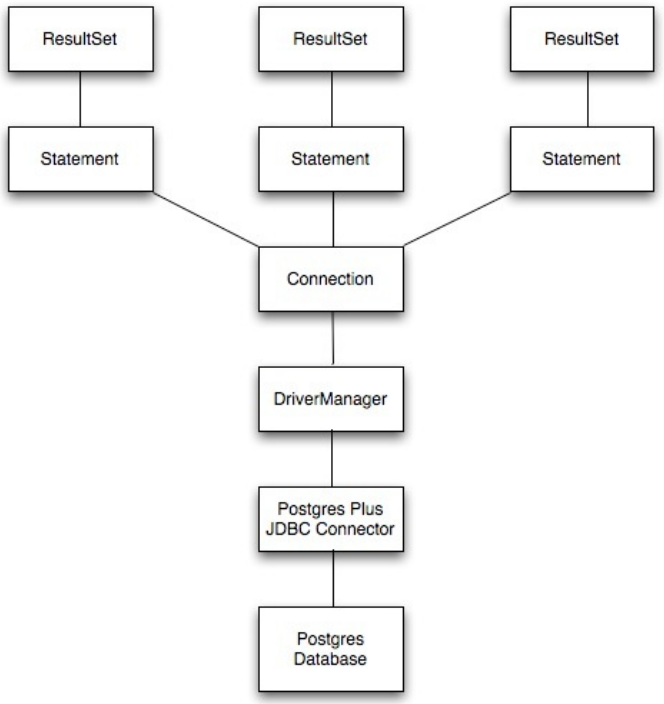
- 100% Java Driver, no native methods.
- Does not require pre-installation on client.
- Can be downloaded and configured on-the-fly just like any Java class file.
- Uses a proprietary protocol for talking with a middleware server.
- Middleware server converts from proprietary calls to DBMS specific calls

Type 4 Driver

- 100% Java Driver, no native methods.
- Does not require pre-installation on client.
- Can be downloaded and configured on-the-fly just like any Java class file.
- Unlike Type III driver, talks directly with the DBMS server.
- Converts JDBC calls directly to database specific calls.

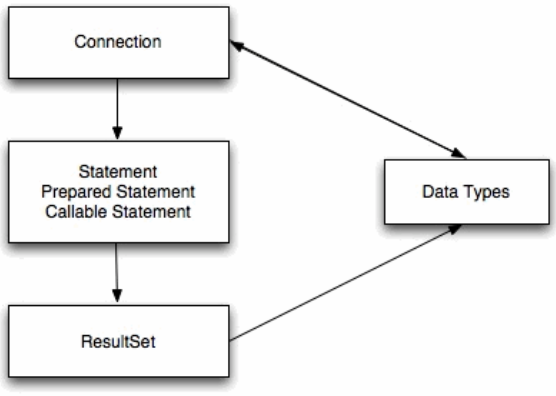
The JDBC Interface

Following figure shows the core API interfaces in the JDBC specification and how they relate to each other. These interfaces are implemented in the `java.sql` package.



JDBC Classes and Interfaces

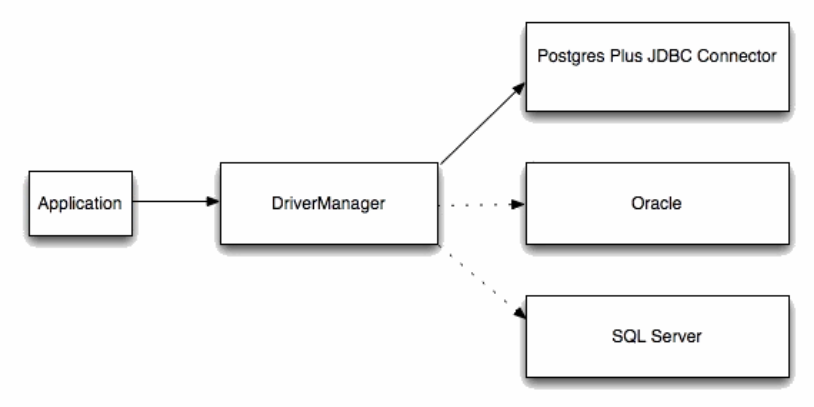
The core API is composed of classes and interfaces; these classes and interfaces work together as shown below:



The JDBC DriverManager

The figure below depicts the role of the `DriverManager` class in a typical JDBC application. The `DriverManager` acts as the bridge between a Java application and the backend database and determines which JDBC driver to use for the

target database.



Advanced Server JDBC Connector Compatibility

EDB provides support for multiple JRE/JDK versions by providing appropriate JDBC drivers for each version of Java Virtual Machine. Use the following table to determine compatibility between your JRE/JDK version and the Advanced Server Driver.

Table – Advanced Server JDBC Driver Compatibility

JRE/JDK Version(s)	JDBC Specification	Advanced Server JDBC Driver	Community JDBC Driver
1.4, 1.5 (see Note 1)	3	edb-jdbc15.jar	postgresql-9.3.1103.JDBC3.jar
1.6 (see Note 2)	4	edb-jdbc16.jar	postgresql-42.2.9.jre6.jar
1.7	4.1	edb-jdbc17.jar	postgresql-42.2.9.jre7.jar
1.8	4.2	edb-jdbc18.jar	postgresql-42.2.9.jar

Note

1. JRE/JDK versions 1.4 and 1.5 are no longer supported by the Advanced Server JDBC Connector as the JDBC driver file `edb-jdbc15.jar` is no longer provided.
2. The `edb-jdbc16.jar` file is not available for Linux on PowerPC 64 little endian

(ppc64le), Debian/Ubuntu and RHEL 8 platforms.
3. The `edb-jdbc17.jar` file is not available for Debian/Ubuntu and RHEL 8 platforms.
4. Community version numbers are based on the `pgjdbc` version, not the PostgreSQL version.
5. Advanced Server JDBC releases are decoupled with EDB Postgres Advanced Server releases.

The following JDBC Compatibility rules apply:

- From Community website: “The PostgreSQL JDBC driver has some unique properties that you should be aware of before starting to develop any code for it. The current development driver supports six server versions and six java environments. This doesn't mean that every feature must work in every combination, but a reasonable behaviour must be provided for non-supported versions. While this extra compatibility sounds like a lot of work, the actual goal is to reduce the amount of work by maintaining only one code base.”
- From the `pgjdbc` readme file: “PgJDBC regression tests are run against all PostgreSQL versions since 8.4, including “build PostgreSQL from git master” version. Don't assume pgjdbc 9.4.x is only PostgreSQL 9.4 compatible.”

For the supported community versions, see the PostgreSQL JDBC Driver website.

4 Installing and Configuring the JDBC Connector

This chapter describes how to install and configure the Advanced Server JDBC Connector.

Before installing the JDBC Connector, you must have Java installed on your system; you can download a Java installer that matches your environment from the Oracle Java Downloads website. Documentation that contains detailed installation instructions is available through the associated `Docs Installation Instruction` links on the same page.

You can use the Advanced Server graphical installer or an RPM package to add the JDBC Connector to your installation.

The following sections describe these installation methods.

4.1 Installing the Connector with an RPM Package

Before installing `JDBC Connector`, you must:

Install the `epel-release` package:

- On RHEL or CentOS 7:

```
yum -y install https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm
```

- On RHEL or CentOS 8:

```
dnf -y install https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm
```

Note

You may need to enable the `[extras]` repository definition in the `CentOS-Base.repo` file (located in `/etc/yum.repos.d`).

You must also have credentials that allow access to the EnterpriseDB repository. For information about requesting credentials, visit:

> https://info.enterprisedb.com/rs/069-ALB-339/../../../../../../images/Repository%20Access%2004-09-2019.pdf

After receiving your repository credentials you can:

1. Create the repository configuration file.
2. Modify the file, providing your user name and password.

3. Install `edb-jdbc`.

Creating a Repository Configuration File

To create the repository configuration file, assume superuser privileges, and invoke the following command:

- On RHEL or CentOS 7:

```
yum -y install https://yum.enterprisedb.com/edb-repo-rpms/edb-repo-latest.noarch.rpm
```

- On RHEL or CentOS 8:

```
dnf -y install https://yum.enterprisedb.com/edb-repo-rpms/edb-repo-latest.noarch.rpm
```

The repository configuration file is named `edb.repo`. The file resides in `/etc/yum.repos.d`.

Modifying the file, providing your user name and password

After creating the `edb.repo` file, use your choice of editor to ensure that the value of the enabled parameter is `1`, and replace the `username` and `password` placeholders in the `baseurl` specification with the name and password of a registered EnterpriseDB user.

```
[edb]
name=EnterpriseDB RPMs $releasever - $basearch
baseurl=https://<username>:
<password>@yum.enterprisedb.com/edb/redhat/rhel-$releasever-$basearch
enabled=1
gpgcheck=1
gpgkey=file:///etc/pki/rpm-gpg/ENTERPRISEDB-GPG-KEY
```

Installing JDBC Connector

After saving your changes to the configuration file, you can use the `yum install` command to install the `JDBC Connector` on RHEL or CentOS 7:

- On RHEL or CentOS 7:

```
yum install edb-jdbc
```

- On RHEL or CentOS 8:

```
dnf install edb-jdbc
```

When you install an RPM package that is signed by a source that is not recognized by your system, yum may ask for your permission to import the key to your local server. If prompted, and you are satisfied that the packages come from a trustworthy source, enter `y`, and press `Return` to continue.

During the installation, yum may encounter a dependency that it cannot resolve. If it does, it will provide a list of the required dependencies that you must manually resolve.

Updating an RPM Installation

If you have an existing `JDBC Connector` RPM installation, you can use yum or dnf to upgrade your repository configuration file and update to a more recent product version. To update the `edb.repo` file, assume superuser privileges and enter:

- On RHEL or CentOS 7:

```
yum upgrade edb-repo
```

- On RHEL or CentOS 8:

```
dnf upgrade edb-repo
```

yum or dnf will update the `edb.repo` file to enable access to the current EDB repository, configured to connect with the credentials specified in your `edb.repo` file. Then, you can use yum or dnf to upgrade any installed packages:

- On RHEL or CentOS 7:

```
yum upgrade edb-jdbc
```

- On RHEL or CentOS 8:

```
dnf upgrade edb-jdbc
```

4.2 Installing the Connector on an SLES 12 Host

You can use the zypper package manager to install the connector on an SLES 12 host. zypper will attempt to satisfy package dependencies as it installs a package, but requires access to specific repositories that are not hosted at EnterpriseDB.

Before installing the connector, use the following commands to add EnterpriseDB repository configuration files to your SLES host:

```
zypper addrepo https://zypp.enterprisedb.com/suse/epas12-sles.repo
zypper addrepo https://zypp.enterprisedb.com/suse/epas-sles-tools.repo
zypper addrepo https://zypp.enterprisedb.com/suse/epas-sles-dependencies.repo
```

Each command creates a repository configuration file in the `/etc/zypp/repos.d` directory. The files are named:

- `edbas12suse.repo`
- `edbasdependencies.repo`
- `edbastools.repo`

After creating the repository configuration files, use the `zypper refresh` command to refresh the metadata on your SLES host to include the EnterpriseDB repositories:

```
/etc/zypp/repos.d # zypper refresh
Repository 'SLES12-12-0' is up to date.
Repository 'SLES12-Pool' is up to date.
Repository 'SLES12-Updates' is up to date.
Retrieving repository 'EDB Postgres Advanced Server 12 12 - x86_64'
metadata ----------------------[\]
```

```
Authentication required for
'https://zypp.enterprisedb.com/12/suse/suse-12-x86_64'
User Name:
Password:

Retrieving repository 'EDB Postgres Advanced Server 12 12 - x86_64'
metadata.....................................[done]
Building repository 'EDB Postgres Advanced Server 12 12 - x86_64'
cache.........................[done]
All repositories have been refreshed.
...
```

When prompted for a `User Name` and `Password`, provide your connection credentials for the EnterpriseDB repository. If you need credentials, contact EnterpriseDB.

Before installing EDB Postgres Advanced Server or supporting components, you must also add SUSEConnect and the SUSE Package Hub extension to the SLES host, and register the host with SUSE, allowing access to SUSE repositories. Use the commands:

```
zypper install SUSEConnect
SUSEConnect -p PackageHub/12/x86_64
SUSEConnect -p sle-sdk/12/x86_64
```

For detailed information about registering a SUSE host, visit the SUSE website.

Then, you can use the zypper utility to install the connector:

```
zypper install edb-jdbc
```

4.3 Installing a DEB Package on a Debian or Ubuntu Host

To install a DEB package on a Debian or Ubuntu host, you must have credentials that allow access to the EnterpriseDB repository. To request credentials for the repository, visit the EDB website.

The following steps will walk you through on using the EnterpriseDB apt repository to install a DEB package. When using the commands, replace the `username` and `password` with the credentials provided by EnterpriseDB.

1. Assume superuser privileges:

   ```
   sudo su –
   ```

2. Configure the EnterpriseDB repository:

   ```
   sh -c 'echo "deb
   https://username:password@apt.enterprisedb.com/$(lsb_release -cs)-
   edb $(lsb_release -cs) main" > /etc/apt/sources.list.d/edb-$(lsb_release
   -cs).list'
   ```

3. Add support to your system for secure APT repositories:

   ```
   apt-get install apt-transport-https
   ```

4. Add the EDB signing key:

   > `wget -q -O - https://username:password@apt.enterprisedb.com/edb-deb.gpg.key | apt-key add -`

5. Update the repository metadata:

   ```
   apt-get update
   ```

6. Install DEB package:

   ```
   apt-get install edb-jdbc
   ```

Note

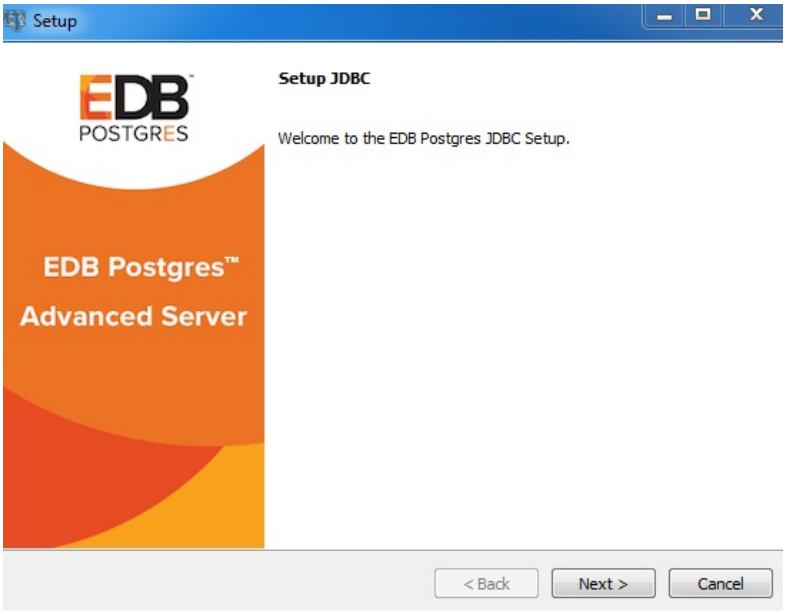
By default, the Debian 9x and Ubuntu 18.04 platform installs Java version 10. Make sure you install Java version 8 on your system to run the EDB Java-based components.

4.4 Using the Graphical Installer to Install the Connector

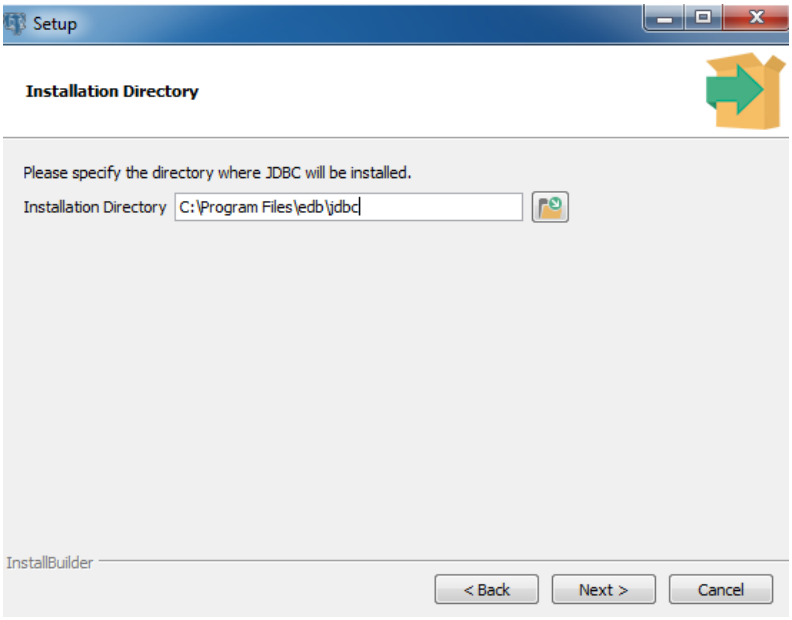
You can use the EnterpriseDB Connectors Installation wizard to add the JDBC connector to your system; the wizard is available from the EnterpriseDB Advanced Downloads page.

This section demonstrates using the Installation Wizard to install the Connectors on a Windows system. (Download the installer, and then, right-click on the installer icon, and select Run As Administrator from the context menu.)

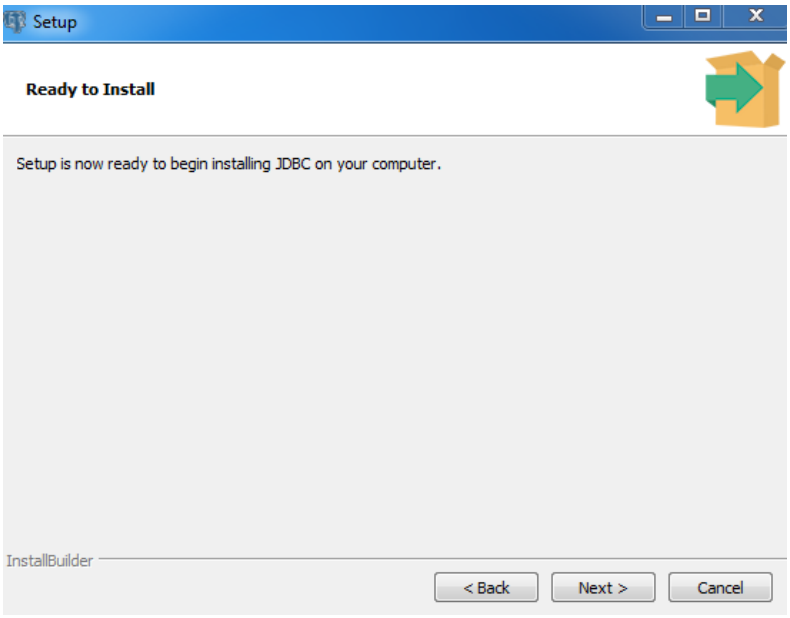
When the Language Selection popup opens, select an installation language and click OK to continue to the Setup window.



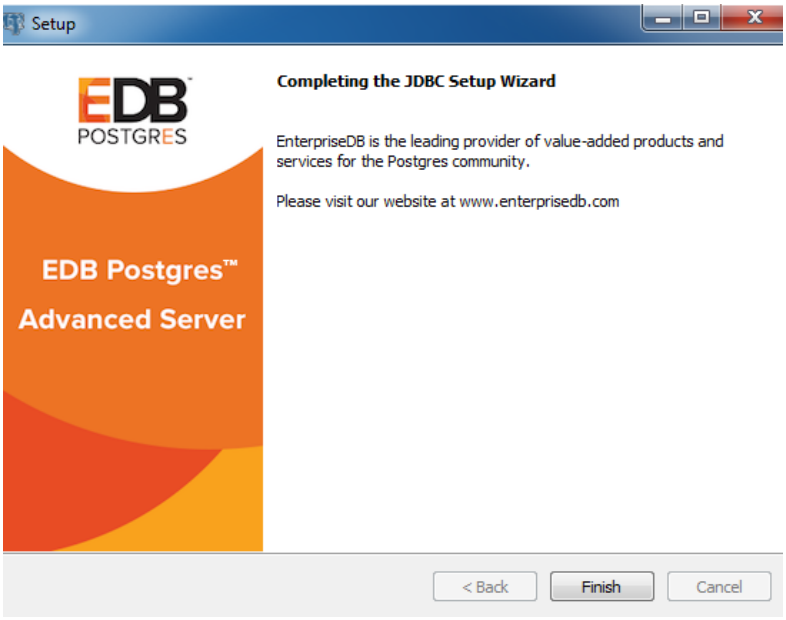
Click Next to continue.



Use the Installation Directory dialog to specify the directory in which the connector will be installed, and click Next to continue.

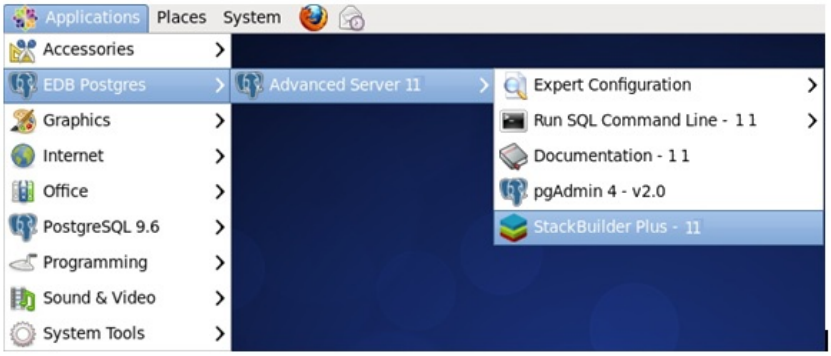


Click Next on the Ready to Install dialog to start the installation; popup dialogs confirm the progress of the installation wizard.

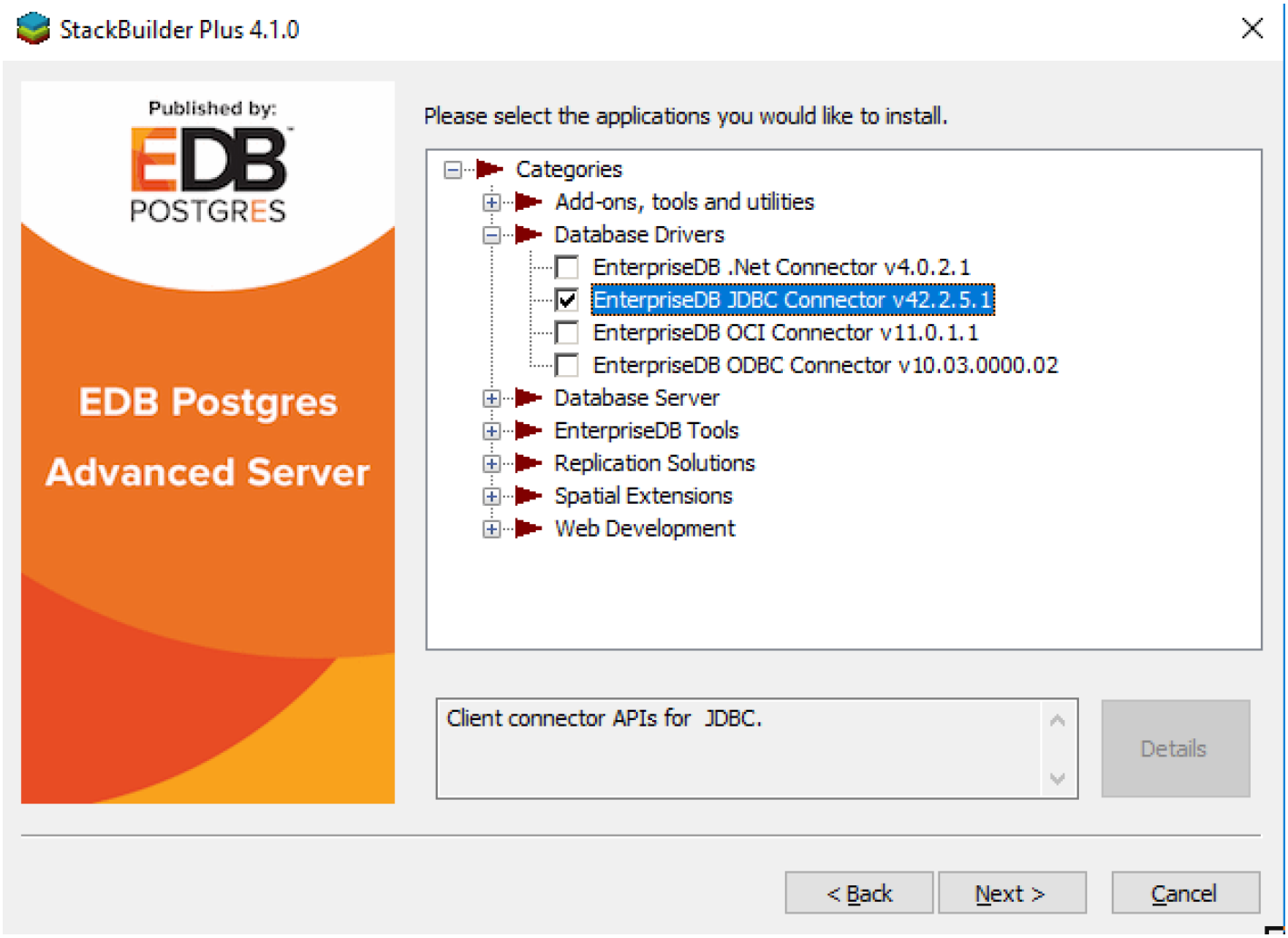


When the wizard informs you that it has completed the setup, click the Finish button to exit the dialog.

You can also use StackBuilder Plus to add or update the connector on an existing Advanced Server installation; to open StackBuilder Plus, select StackBuilder Plus from the Windows Apps menu or through Linux Applications menu.



When StackBuilder Plus opens, follow the onscreen instructions. Select the EnterpriseDB JDBC Connector option from the Database Drivers node of the tree control.



Follow the directions of the onscreen wizard to add or update an installation of the EnterpriseDB Connectors.

4.5 Configuring the Advanced Server JDBC Connector

Advanced Server ships with the following JDBC drivers:

- edb-jdbc16.jar supports JDBC version 4
- edb-jdbc17.jar supports JDBC version 4.1
- edb-jdbc18.jar supports JDBC version 4.2

Note

The `edb-jdbc16.jar` file is not available for Linux on PowerPC 64 little endian (ppc64le).

To make the JDBC driver available to Java, you must either copy the appropriate java `.jar` file for the JDBC version that you are using to your `$java_home/jre/lib/ext` directory or append the location of the `.jar` file to the `CLASSPATH` environment variable.

Note that if you choose to append the location of the `jar` file to the `CLASSPATH` environment variable, you must include the complete pathname:

```
/usr/edb/jdbc/edb-jdbcxx.jar
```

5 Using the Advanced Server JDBC Connector with Java applications

With Java and the Advanced Server JDBC Connector in place, a Java application can access an Advanced Server database. Listing 1.1 creates an application that executes a query and prints the result set.

Listing 1.1

```
import java.sql.*;
public class ListEmployees
{
  public static void main(String[] args)
  {
    try
    {
      Class.forName("com.edb.Driver");
      String url      = "jdbc:edb://localhost:5444/edb";
      String user     = "enterprisedb";
      String password = "enterprisedb";
```

```
    Connection con  = DriverManager.getConnection(url, user, password);
    Statement stmt  = con.createStatement();
    ResultSet rs    = stmt.executeQuery("SELECT * FROM emp");
    while(rs.next())
    {
      System.out.println(rs.getString(1));
    }

    rs.close();
    stmt.close();
    con.close();
    System.out.println("Command successfully executed");
   }
   catch(ClassNotFoundException e)
   {
    System.out.println("Class Not Found : " + e.getMessage());
   }
   catch(SQLException exp)
   {
    System.out.println("SQL Exception: " + exp.getMessage());
    System.out.println("SQL State:     " + exp.getSQLState());
    System.out.println("Vendor Error:  " + exp.getErrorCode());
   }
  }
}
```

This example is simple, but it demonstrates the fundamental steps required to interact with an Advanced Server database from a Java application:

- Load the JDBC driver
- Build connection properties
- Connect to the database server
- Execute an SQL statement
- Process the result set

- Clean up
- Handle any errors that may occur

5.1 Loading the Advanced Server JDBC Connector

The Advanced Server JDBC driver is written in Java and is distributed in the form of a compiled JAR (Java Archive) file. Use the `Class.forName()` method to load the driver. The `forName()` method dynamically loads a Java class at runtime. When an application calls the `forName()` method, the JVM (Java Virtual Machine) attempts to find the compiled form (the bytecode) that implements the requested class.

The Advanced Server JDBC driver is named `com.edb.Driver`:

```
Class.forName("com.edb.Driver");
```

After loading the bytecode for the driver, the driver registers itself with another JDBC class (named `DriverManager`) that is responsible for managing all the JDBC drivers installed on the current system.

If the JVM is unable to locate the named driver, it throws a `ClassNotFound` exception (which is intercepted with a `catch` block near the end of the program). The `DriverManager` is designed to handle multiple JDBC driver objects. You can write a Java application that connects to more than one database system via JDBC. The next section explains how to select a specific driver.

5.2 Connecting to the Database

After the driver has loaded and registered itself with the `DriverManager`, the `ListEmployees` class can attempt to connect to the database server, as shown in the following code fragment:

```
String url = "jdbc:edb://localhost:5444/edb";
String user = "enterprisedb";
String password = "enterprisedb";
Connection con = DriverManager.getConnection(url, user, password);
```

All JDBC connections start with the `DriverManager`. The `DriverManager` class offers a static method called `getConnection()` that is responsible for creating a connection to the database. When you call the `getConnection()` method, the `DriverManager` must decide which JDBC driver to use to connect to the database; that decision is based on a URL (Uniform Resource Locator) that you pass to `getConnection()`.

A JDBC URL takes the following general format:

> `jdbc:<driver>:<connection parameters>`

The first component in a JDBC URL is always `jdbc`. When using the Advanced Server JDBC Connector, the second component (the driver) is `edb`.

The Advanced Server JDBC URL takes one of the following forms:

> `jdbc:edb:<database>`
>
> `jdbc:edb://<host>/<database>`
>
> `jdbc:edb://<host>:<port>/<database>`

The following table shows the various connection parameters:

Table - Connection Parameters

Name	Description
host	The host name of the server. Defaults to localhost.
port	The port number the server is listening on. Defaults to the Advanced Server standard port number (5444).

Name	Description
database	The database name.

5.2.1 Additional Connection Properties

In addition to the standard connection parameters, the Advanced Server JDBC driver supports connection properties that control behavior specific to EnterpriseDB. You can specify these properties in the connection URL or as a Properties object parameter passed to DriverManager.getConnection(). Listing 1.2 demonstrates how to use a Properties object to specify additional connection properties:

Listing 1.2

```
String url = "jdbc:edb://localhost/edb";
Properties props = new Properties();

props.setProperty("user", "enterprisedb");
props.setProperty("password", "enterprisedb");
props.setProperty("sslfactory", "com.edb.ssl.NonValidatingFactory");
props.setProperty("ssl", "true");

Connection con = DriverManager.getConnection(url, props);
```

Note

By default the combination of SSL=true and setting the connection URL parameter sslfactory=org.postgresql.ssl.NonValidatingFactory encrypts the connection but does not validate the SSL certificate. To enforce certificate validation, you must use a Custom SSLSocketFactory. For more details about writing a Custom SSLSocketFactory, review the PostgreSQL JDBC driver documentation.

To specify additional connection properties in the URL, add a question mark

and an ampersand-separated list of keyword-value pairs:

```
String url = "jdbc:edb://localhost/edb?user=enterprisedb&ssl=true";
```

Some of the additional connection properties are shown in the following table:

Table 5-2 - Additional Connection Properties

Name	Type	Description
user	String	The database user on whose behalf the connection is being made.
password	String	The database user's password.
ssl	Boolean	Requests an authenticated, encrypted SSL connection
loglevel	Integer	The value of loglevel determines the amount of detail printed to the DriverManager's current value for LogStream or LogWriter. It currently supports values of: com.edb.Driver.DEBUG com.edb.Driver.INFO Set the value of loglevel to INFO to include sparse log information or to DEBUG to produce significant detail.
charSet	String	The value of charSet determines the character set used for data sent to or received from the database.
prepareThreshold	Integer	The value of prepareThreshold determines the number of PreparedStatement executions required before switching to server side prepared statements. The default is five.

Name	Type	Description
loadBalanceHosts	Boolean	In default mode (disabled) hosts are connected in the given order. If enabled, hosts are chosen randomly from the set of suitable candidates.
targetServerType	String	Allows opening connections to only servers with the required state. The allowed values are any, master, secondary, preferSecondary, and preferSyncSecondary. The master/secondary distinction is currently done by observing if the server allows writes. The value preferSecondary tries to connect to secondaries if any are available, otherwise allows connecting to the master. The Advanced Server JDBC Connector supports preferSyncSecondary, which permits connection to only synchronous secondaries or the master if there are no active synchronous secondaries. See Section 5.2.2 for information on preferSyncSecondary. Note: The values slave, preferSlave, and preferSyncSlave have been deprecated as they have been replaced by the “secondary” values. The “slave” values are currently still supported, and the use of the terms “slave” and “secondary” provide the same functionality, however, it is advised to use the “secondary” values.
skipQuotesOnReturning	Boolean	When set to true, column names from the RETURNING clause are not quoted. This eliminates a case-sensitive comparison of the column name. When set to false (the default setting), column names are quoted.

5.2.2 Preferring Synchronous Secondary Database Servers

The Advanced Server JDBC Connector supports the `preferSyncSecondary` option for the `targetServerType` connection property as noted in Table 5-2.

Note

The `targetServerType` values `slave`, `preferSlave`, and `preferSyncSlave` all provide the same corresponding set of functionality as the `secondary`, `preferSecondary`, and `preferSyncSecondary` values; the `slave` values have been deprecated. We advise you to use the `secondary` values.

The `preferSyncSecondary` option provides a preference for synchronous, standby servers for failover connection, and thus ignoring asynchronous servers.

The specification of this capability in the connection URL is shown by the following syntax:

```
jdbc:edb://master:port,secondary_1:port_1,secondary_2:port_2,.../
database?targetServerType=preferSyncSecondary
```

Parameters

`master:port`

The IP address or a name assigned to the master database server followed by its port number. If `master` is a name, it must be specified with its IP address in the `/etc/hosts` file on the host running the Java program. **Note:** The master database server can be specified in any location in the list. It does not have to precede the secondary database servers.

`secondary_n:port_n`

The IP address or a name assigned to a standby, secondary database server followed by its port number. If `secondary_n` is a name, it must be specified with

its IP address in the `/etc/hosts` file on the host running the Java program.

`database`

The name of the database to which the connection is to be made.

The following is an example of the connection URL:

```
String url = "jdbc:edb://master:5300,secondary1:5400/edb?
targetServerType=preferSyncSecondary";
con = DriverManager.getConnection(url, "enterprisedb", "edb");
```

The following characteristics apply to the `preferSyncSecondary` option:

- The master database server may be specified in any location in the connection list.
- Connection for accessing the database for usage by the Java program is first attempted on a synchronous secondary. The secondary servers are available for read-only operations.
- No connection attempt is made to any servers running in asynchronous mode.
- The order in which connection attempts are made is determined by the `loadBalanceHosts` connection property as described in Table 5-2. If disabled, which is the default setting, connection attempts are made in the left-to-right order specified in the connection list. If enabled, connection attempts are made randomly.
- If connection cannot be made to a synchronous secondary, then connection to the master database server is used. If the master database server is not active, then the connection attempt fails.

The synchronous secondaries to be used for the `preferSyncSecondary` option must be configured for hot standby usage.

The following section provides a brief overview of setting up the master and secondary database servers for hot standby, synchronous replication.

Configuring Master and Secondary Database Servers Overview

The process for configuring a master and secondary database servers are

described in the PostgreSQL documentation.

For general information on hot standby usage, which is needed for the `preferSyncSecondary` option, see the PostgreSQL core documentation.

For information about creating a base backup for the secondary database server from the master, see Section 25.3.2, *Making a Base Backup* (describes usage of the pg_basebackup utility program) or Section 25.3.3, *Making a Base Backup Using the Low Level API* within Section 25.3 *Continuous Archiving and Point-in-Time Recovery (PITR)* in The PostgreSQL Core Documentation.

For information on the configuration parameters that must be set for hot standby usage, see Section 19.6, Replication.

The following section provides a basic example of setting up the master and secondary database servers.

Example: Master and Secondary Database Servers

In the example that follows, the:

- Master database server resides on host `192.168.2.24`, port `5444`
- Secondary database server is named `secondary1` and resides on host `192.168.2.22`, port `5445`
- Secondary database server is named `secondary2` and resides on host `192.162.2.24`, port `5446` (same host as the master)

In the master database server's `pg_hba.conf` file, there must be a replication entry for each unique replication database `USER/ADDRESS` combination for all secondary database servers. In the following example, the database superuser `enterprisedb` is used as the replication database user for both the `secondary1` database server on `192.168.2.22` and the `secondary2` database server that is local relative to the master.

```
### TYPE  DATABASE        USER            ADDRESS                 METHOD
host    replication     enterprisedb    192.168.2.22/32         md5
host    replication     enterprisedb    127.0.0.1/32            md5
```

After the master database server has been configured in the `postgresql.conf`

file along with its `pg_hba.conf` file, database server `secondary1` is created by invoking the following command on host `192.168.2.22` for `secondary1`:

```
su – enterprisedb
Password:
-bash-4.1$ pg_basebackup -D /opt/secondary1 -h 192.168.2.24 -p 5444 -Fp -R
-X stream -l 'Secondary1'
```

On the secondary database server, `/opt/secondary1`, a `recovery.conf` file is generated in the database cluster, which has been edited in the following example by adding the `application_name=secondary1` setting as part of the `primary_conninfo` string and removal of some of the other unneeded options automatically generated by `pg_basebackup`. Also note the use of the `standby_mode = 'on'` parameter.

```
standby_mode = 'on'
primary_conninfo = 'user=enterprisedb password=password
host=192.168.2.24 port=5444 application_name=secondary1'
```

The application name `secondary1` must be included in the `synchronous_standby_names` parameter of the master database server's `postgresql.conf` file.

The secondary database server (`secondary2`) is created in an alternative manner on the same host used by the master:

```
su - enterprisedb
Password:
-bash-4.1$ psql -d edb -c "SELECT pg_start_backup('Secondary2')"
Password:
 pg_start_backup
-----------------
 0/6000028
(1 row)

-bash-4.1$ cp -rp /var/lib/edb/as12/data/opt/secondary2
-bash-4.1$ psql -d edb -c "SELECT pg_stop_backup()"
Password:
```

```
NOTICE:  pg_stop_backup complete, all required WAL segments have been
archived
 pg_stop_backup
----------------
 0/6000130
(1 row)
```

On the secondary database server `/opt/secondary2`, create the `recovery.conf` file in the database cluster. Note the `application_name=secondary2` setting as part of the `primary_conninfo` string as shown in the following example. Also be sure to include the `standby_mode = 'on'` parameter.

```
standby_mode = 'on'
primary_conninfo = 'user=enterprisedb password=password host=localhost
port=5444 application_name=secondary2'
```

The application name `secondary2` must be included in the `synchronous_standby_names` parameter of the master database server's `postgresql.conf` file.

You must ensure the configuration parameter settings in the `postgresql.conf` file of the secondary database servers are properly set (particularly `hot_standby=on`).

Note

As of EDB Postgres Advanced Server v12, the `recovery.conf` file is no longer valid; it is replaced by the `standby.signal` file. As a result, `primary_conninfo` is moved from the `recovery.conf` file to the `postgresql.conf` file. The presence of `standby.signal` file signals the cluster to run in standby mode. Please note that even if you try to create a `recovery.conf` file manually and keep it under the `data` directory, the server will fail to start and throw an error.

The parameter `standby_mode=on` is also removed from EDB Postgres Advanced Server v12, and the `trigger_file` parameter name is changed to `promote_trigger_file`.

The following table lists the basic `postgresql.conf` configuration parameter settings of the master database server as compared to the secondary database servers:

Table - Master/Secondary Configuration Parameters

Parameter	Master	Secondary	Description
archive_mode	on	off	Completed WAL segments sent to archive storage
archive_command	cp %p /*archive_dir*/%f	n/a	Archive completed WAL segments
wal_level	hot_standby (9.5 or earlier), replica (9.6 or later)	minimal	Information written to WAL segment
max_wal_senders	*n* (positive integer)	0	Maximum concurrent connections from standby servers
wal_keep_segments	*n* (positive integer)	0	Minimum number of past log segments to keep for standby servers

Parameter	Master	Secondary	Description
synchronous_standby_names	*n*(*secondary1*, *secondary2*,...)	n/a	List of standby servers for synchronous replication. Must be present to enable synchronous replication. These are obtained from the application_name option of the primary_conninfo parameter in the recovery.conf file of each standby server.
hot_standby	off	on	Client application can connect and run queries on the secondary server in standby mode

The secondary database server (`secondary1`) is started:

```
-bash-4.1$ pg_ctl start -D /opt/secondary1 -l logfile -o "-p 5445"
server starting
```

The secondary database server (`secondary2`) is started:

```
-bash-4.1$ pg_ctl start -D /opt/secondary2/data -l logfile -o "-p 5446"
server starting
```

To ensure that the secondary database servers are properly set up in synchronous mode, use the following query on the master database server. Note that the `sync_state` column lists applications `secondary1` and

secondary2 as sync.

```
edb=# SELECT usename, application_name, client_addr, client_port,
sync_state FROM pg_stat_replication;
   usename    | application_name | client_addr  | client_port | sync_state
--------------+------------------+--------------+-------------+------------
 enterprisedb | secondary1       | 192.168.2.22 |       53525 | sync
 enterprisedb | secondary2       | 127.0.0.1    |       36214 | sync
(2 rows)
```

The connection URL is:

```
String url = "jdbc:edb://master:5444,secondary1:5445,secondary2:5446/edb?
targetServerType=preferSyncSecondary";
con = DriverManager.getConnection(url, "enterprisedb", "password");
```

The /etc/hosts file on the host running the Java program contains the following entries with the server names specified in the connection URL string:

```
192.168.2.24          localhost.localdomain master
192.168.2.22          localhost.localdomain secondary1
192.168.2.24          localhost.localdomain secondary2
```

For this example, the preferred synchronous secondary connection option results in the first usage attempt made on secondary1, then on secondary2 if secondary1 is not active, then on the master if both secondary1 and secondary2 are not active as demonstrated by the following program that displays the IP address and port of the database server to which the connection is made.

```
import java.sql.*;
public class InetServer
{
  public static void main(String[] args)
  {
    try
    {
      Class.forName("com.edb.Driver");
```

```
      String url =
 "jdbc:edb://master:5444,secondary1:5445,secondary2:5446/edb?
targetServerType=preferSyncSecondary";
      String user     = "enterprisedb";
      String password = "password";
      Connection con  = DriverManager.getConnection(url, user, password);

      ResultSet rs = con.createStatement().executeQuery("SELECT
inet_server_addr() || ':' || inet_server_port()");
      rs.next();
      System.out.println(rs.getString(1));

      rs.close();
      con.close();
      System.out.println("Command successfully executed");
    }
    catch(ClassNotFoundException e)
    {
      System.out.println("Class Not Found : " + e.getMessage());
    }
    catch(SQLException exp)
    {
      System.out.println("SQL Exception: " + exp.getMessage());
      System.out.println("SQL State:     " + exp.getSQLState());
      System.out.println("Vendor Error:  " + exp.getErrorCode());
    }
  }
}
```

Case 1: When all database servers are active, connection is made to `secondary1` on `192.168.2.22` port `5445`.

```
$ java InetServer
192.168.2.22/32:5445
Command successfully executed
```

Case 2: When `secondary1` is shut down, connection is made to `secondary2` on `192.168.2.24` port `5446`.

```
bash-4.1$ /usr/edb/as12/bin/pg_ctl stop -D /opt/secondary1
waiting for server to shut down.... done
server stopped

$ java InetServer
192.168.2.24/32:5446
Command successfully executed
```

Case 3: When `secondary2` is also shut down, connection is made to the `master` on `192.168.2.24` port `5444`.

```
bash-4.1$ /usr/edb/as12/bin/pg_ctl stop -D /opt/secondary2/data
waiting for server to shut down.... done
server stopped

$ java InetServer
192.168.2.24/32:5444
Command successfully executed
```

5.3 Executing SQL Statements through Statement Objects

After loading the Advanced Server JDBC Connector driver and connecting to the server, the code in the sample application builds a JDBC `Statement` object, executes an SQL query, and displays the results.

A `Statement` object sends SQL statements to a database. There are three kinds of Statement objects. Each is specialized to send a particular type of SQL statement:

- A `Statement` object is used to execute a simple SQL statement with no parameters.
- A `PreparedStatement` object is used to execute a pre-compiled SQL statement with or without IN parameters.
- A `CallableStatement` object is used to execute a call to a database stored procedure.

You must construct a `Statement` object before executing an SQL statement. The `Statement` object offers a way to send a SQL statement to the server (and gain access to the result set). Each `Statement` object belongs to a `Connection`; use the `createStatement()` method to ask the `Connection` to create the `Statement` object.

A `Statement` object defines several methods to execute different types of SQL statements. In the sample application, the `executeQuery()` method executes a `SELECT` statement:

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM emp");
```

The `executeQuery()` method expects a single argument: the SQL statement that you want to execute. `executeQuery()` returns data from the query in a `ResultSet` object. If the server encounters an error while executing the SQL statement provided, it throws an `SQLException` (and does not return a `ResultSet`).

Using Named Notation with a CallableStatement Object

The JDBC Connector (Advanced Server version 9.6 and later) supports the use of named parameters when instantiating a `CallableStatement` object. This syntax is an extension of JDBC supported syntax, and does not conform to the JDBC standard.

You can use a `CallableStatement` object to pass parameter values to a stored procedure. You can assign values to `IN`, `OUT`, and `INOUT` parameters with a `CallableStatement` object.

When using the `CallableStatement` class, you can use ordinal notation or named notation to specify values for an actual arguments. You must set a value

for each `IN` or `INOUT` parameter marker in a statement.

When using ordinal notation to pass values to a `CallableStatement` object, you should use the setter method that corresponds to the parameter type. For example, when passing a `STRING` value, use the `setString` setter method. Each parameter marker within a statement (`?`) represents an ordinal value. When using ordinal parameters, you should pass the actual parameter values to the statement in the order that the formal arguments are specified within the procedure definition.

You can also use named parameter notation when specifying argument values for a `CallableStatement` object. Named parameter notation allows you to supply values for only those parameters that are required by the procedure, omitting any parameters that have acceptable default values. You can also specify named parameters in any order.

When using named notation, each parameter name should correspond to a `COLUMN_NAME` returned by a call to the `DatabaseMetaData.getProcedureColumns` method. You should use the `=>` token when including a named parameter in a statement call.

Use the `registerOutParameter` method to identify each `OUT` or `INOUT` parameter marker in the statement.

Examples

The following examples demonstrate using the `CallableStatement` method to provide parameters to a procedure with the following signature:

```
PROCEDURE hire_emp (ename VARCHAR2
empno NUMBER
job VARCHAR2
sal NUMBER
hiredate DATE DEFAULT now(),
mgr NUMBER DEFAULT 7100,
deptno NUMBER
)
```

The following example uses ordinal notation to provide parameters:

```
CallableStatement cstmt = con.prepareCall
("{CALL hire_emp(?,?,?,?,?,?,?)}");
//Bind a value to each parameter.
cstmt.setString(1, "SMITH");
cstmt.setInt(2, 8888);
cstmt.setString(3, "Sales");
cstmt.setInt(4, 5500);
cstmt.setDate(5, "2016-06-01");
cstmt.setInt(6, 7566);
cstmt.setInt(7, 30);
```

The following example uses named notation to provide parameters; using named notation, you can omit parameters that have default values or re-order parameters:

```
CallableStatement cstmt = con.prepareCall
("{CALL hire_emp(ename => ?,
job => ?,
empno => ?,
sal => ?,
deptno => ?
)}");

//Bind a value to each parameter.
cstmt.setString("ename", "SMITH");
cstmt.setInt("empno", 8888);
cstmt.setString("job", "Sales");
cstmt.setInt("sal", 5500);
cstmt.setInt("deptno", 30);
```

5.4 Retrieving Results from a ResultSet Object

A `ResultSet` object is the primary storage mechanism for the data returned by an SQL statement. Each `ResultSet` object contains both data and metadata (in the form of a `ResultSetMetaData` object). `ResultSetMetaData` includes useful information about results returned by the SQL command: column names, column count, row count, column length, and so on.

To access the row data stored in a `ResultSet` object, an application calls one or more `getter` methods. A `getter` method retrieves the value in particular column of the current row. There are many different `getter` methods; each method returns a value of a particular type. For example, the `getString()` method returns a `STRING` type; the `getDate()` method returns a `Date`, and the `getInt()` method returns an `INT` type. When an application calls a `getter` method, JDBC tries to convert the value into the requested type.

Each `ResultSet` keeps an internal pointer that point to the current row. When the `executeQuery()` method returns a `ResultSet`, the pointer is positioned before the first row; if an application calls a `getter` method before moving the pointer, the `getter` method will fail. To advance to the next (or first) row, call the `ResultSet's next()` method. `ResultSet.next()` is a boolean method; it returns `TRUE` if there is another row in the `ResultSet` or `FALSE` if you have moved past the last row.

After moving the pointer to the first row, the sample application uses the `getString()` `getter` method to retrieve the value in the first column and then prints that value. Since `ListEmployees` calls `rs.next()` and `rs.getString()` in a loop, it processes each row in the result set. `ListEmployees` exits the loop when `rs.next()` moves the pointer past the last row and returns `FALSE`.

```
while(rs.next())
{
System.out.println(rs.getString(1));
}
```

When using the `ResultSet` interface, remember:

- You must call `next()` before reading any values. `next()` returns `true` if another row is available and prepares the row for processing.
- Under the JDBC specification, an application should access each row in the `ResultSet` only once. It is safest to stick to this rule, although at the current time, the Advanced Server JDBC driver will allow you to access a field as many times as you want.

- When you've finished using a `ResultSet`, call the `close()` method to free the resources held by that object.

5.5 Freeing Resources

Every JDBC object consumes some number of resources. A `ResultSet` object, for example, may contain a copy of every row returned by a query; a `Statement` object may contain the text of the last command executed, and so forth. It's usually a good idea to free up those resources when the application no longer needs them. The sample application releases the resources consumed by the `Result`, `Statement`, and `Connection` objects by calling each object's `close()` method:

```
rs.close();
stmt.close();
con.close();
```

If you attempt to use a JDBC object after closing it, that object will throw an error.

5.6 Handling Errors

When connecting to an external resource (such as a database server), errors are bound to occur; your code should include a way to handle these errors. Both JDBC and the Advanced Server JDBC Connector provide various types of error handling. The ListEmployees class example demonstrates how to handle an error using `try/catch` blocks.

When a JDBC object throws an error (an object of type `SQLException` or of a

type derived from `SQLException`), the `SQLException` object exposes three different pieces of error information:

- The error message.
- The SQL State.
- A vendor-specific error code.

In the example, the following code displays the value of these components should an error occur:

```
System.out.println("SQL Exception: " + exp.getMessage());
System.out.println("SQL State: " + exp.getSQLState());
System.out.println("Vendor Error: " + exp.getErrorCode());
```

For example, if the server tries to connect to a database that does not exist on the specified host, the following error message is displayed:

```
SQL Exception: FATAL: database "acctg" does not exist
SQL State: 3D000
Vendor Error: 0
```

6 Executing SQL Commands with executeUpdate()

In the previous example `ListEmployees` executed a `SELECT` statement using the `Statement.executeQuery()` method. `executeQuery()` was designed to execute query statements so it returns a `ResultSet` that contains the data returned by the query. The `Statement` class offers a second method that you should use to execute other types of commands (`UPDATE`, `INSERT`, `DELETE`, and so forth). Instead of returning a collection of rows, the `executeUpdate()` method returns the number of rows affected by the SQL command it executes.

The signature of the `executeUpdate()` method is:

```
int executeUpdate(String sqlStatement)
```

Provide this method a single parameter of type `String`, containing the SQL command that you wish to execute.

Using executeUpdate() to INSERT Data

The example that follows demonstrates using the `executeUpdate()` method to add a row to the `emp` table.

NOTE: the following example is not a complete application, only a method - the samples in the remainder of this document do not include the code required to set up and tear down a `Connection`. To experiment with the example, you must provide a class that invokes the sample code.

Listing 1.3

```
public void updateEmployee(Connection con)
{
  try
  {
    Console console = System.console();
    Statement stmt  = con.createStatement();

    String empno = console.readLine("Employee Number :");
    String ename = console.readLine("Employee Name   :");
    int rowcount = stmt.executeUpdate("INSERT INTO emp(empno, ename)
      VALUES("+empno+",'"+ename+"')");
    System.out.println("");
    System.out.println("Success - "+rowcount+" rows affected.");
  }
  catch(Exception err)
  {
    System.out.println("An error has occurred.");
    System.out.println("See full details below.");
    err.printStackTrace();
```

```
  }
}
```

The `updateEmployee()` method expects a single argument from the caller, a `Connection` object that must be connected to an Advanced Server database:

```
public void updateEmployee(Connection con)
```

Next, `updateEmployee()` prompts the user for an employee name and number:

```
String empno = console.readLine("Employee Number :");
String ename = console.readLine("Employee Name :");
```

`updateEmployee()` concatenates the values returned by `console.readline()` into an `INSERT` statement and pass the result to the `executeUpdate()` method.

```
int rowcount = stmt.executeUpdate("INSERT INTO emp(empno, ename)
VALUES("+empno+",'"+ename+"')");
```

For example, if the user enters an employee number of `6000` and a name of `Jones`, the `INSERT` statement passed to `executeUpdate()` will look like this:

```
INSERT INTO emp(empno, ename) VALUES(6000, 'Jones');
```

The `executeUpdate()` method returns the number of rows affected by the SQL statement (an `INSERT` typically affects one row, but an `UPDATE` or `DELETE` statement can affect more). If `executeUpdate()` returns without throwing an error, the call to `System.out.println` displays a message to the user that shows the number of rows affected.

```
System.out.println("");
System.out.println("Success - "+rowcount+" rows affected.");
```

The catch block displays an appropriate error message to the user if the program encounters an exception:

```
{
System.out.println("An error has occurred.");
```

```
System.out.println("See full details below.");
err.printStackTrace();
}
```

executeUpdate() Syntax Examples

You can use `executeUpdate()` with any SQL command that does not return a result set. Some simple syntax examples using `executeUpdate()` with SQL commands follow:

To use the `UPDATE` command with `executeUpdate()` to update a row:

```
stmt.executeUpdate("UPDATE emp SET ename='"+ename+"' WHERE empno="+empno);
```

To use the `DELETE` command with `executeUpdate()` to remove a row from a table:

```
stmt.executeUpdate("DELETE FROM emp WHERE empno="+empno);
```

To use the DROP TABLE command with executeUpdate() to delete a table from a database:

```
stmt.executeUpdate("DROP TABLE tablename");
```

To use the `CREATE TABLE` command with `executeUpdate()` to add a new table to a database:

```
stmt.executeUpdate("CREATE TABLE tablename (fieldname NUMBER(4,2), fieldname2 VARCHAR2(30))");
```

To use the `ALTER TABLE` command with `executeUpdate()` to change the attributes of a table:

```
stmt.executeUpdate("ALTER TABLE tablename ADD COLUMN colname BOOLEAN");
```

7 Adding a Graphical Interface to a Java Program

With a little extra work, you can add a graphical user interface to a program - the next example (Listing 1.4) demonstrates how to write a Java application that creates a `JTable` (a spreadsheet-like graphical object) and copies the data returned by a query into that `JTable`.

Note

The following sample application is a method, not a complete application. To call this method, provide an appropriate main() function and wrapper class.

Listing 1.4

```
import java.sql.*;
import java.util.Vector;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTable;

...
public void showEmployees(Connection con)
{
  try
  {
    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery("SELECT * FROM emp");
    ResultSetMetaData rsmd = rs.getMetaData();
    Vector labels = new Vector();
    for(int column = 0; column < rsmd.getColumnCount(); column++)
      labels.addElement(rsmd.getColumnLabel(column + 1));

    Vector rows = new Vector();
    while(rs.next())
    {
```

```
      Vector rowValues = new Vector();
      for(int column = 0; column < rsmd.getColumnCount(); column++)
        rowValues.addElement(rs.getString(column + 1));
      rows.addElement(rowValues);
    }

    JTable table = new JTable(rows, labels);
    JFrame jf = new JFrame("Browsing table: EMP (from EnterpriseDB)");
    jf.getContentPane().add(new JScrollPane(table));
    jf.setSize(400, 400);
    jf.setVisible(true);
    jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    System.out.println("Command successfully executed");
  }
  catch(Exception err)
  {
    System.out.println("An error has occurred.");
    System.out.println("See full details below.");
    err.printStackTrace();
  }
}
```

Before writing the `showEmployees()` method, you must import the definitions for a few JDK-provided classes:

```
import java.sql.*;
import java.util.Vector;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTable;
```

The `showEmployees()` method expects a Connection object to be provided by the caller; the `Connection` object must be connected to the Advanced Server:

```
public void showEmployees(Connection con)
```

`showEmployees()` creates a `Statement` and uses the `executeQuery()` method

to execute an SQL query that generates an employee list:

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM emp");
```

As you would expect, `executeQuery()` returns a `ResultSet` object. The `ResultSet` object contains the metadata that describes the `shape` of the result set (that is, the number of rows and columns in the result set, the data type for each column, the name of each column, and so forth). You can extract the metadata from the `ResultSet` by calling the `getMetaData()` method:

```
ResultSetMetaData rsmd = rs.getMetaData();
```

Next, `showEmployees()` creates a `Vector` (a one dimensional array) to hold the column headers and then copies each header from the `ResultMetaData` object into the vector:

```
Vector labels = new Vector();
for(int column = 0; column < rsmd.getColumnCount(); column++)
{
  labels.addElement(rsmd.getColumnLabel(column + 1));
}
```

With the column headers in place, `showEmployees()` extracts each row from the `ResultSet` and copies it into a new vector (named `rows`). The `rows` vector is actually a vector of vectors: each entry in the `rows` vector contains a vector that contains the data values in that row. This combination forms the two-dimensional array that you will need to build a `JTable`. After creating the rows vector, the program reads through each row in the `ResultSet` (by calling `rs.next()`). For each column in each row, a `getter` method extracts the value at that row/column and adds the value to the `rowValues` vector. Finally, `showEmployee()` adds each `rowValues` vector to the `rows` vector:

```
Vector rows = new Vector();
while(rs.next())
{
  Vector rowValues = new Vector();
  for(int column = 0; column < rsmd.getColumnCount(); column++)
    rowValues.addElement(rs.getString(column + 1));
```

```
  rows.addElement(rowValues);
}
```

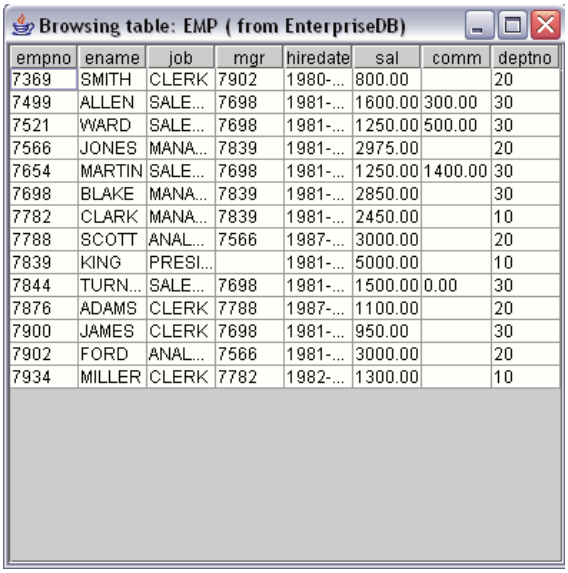
At this point, the vector (`labels`) contains the column headers, and a second two-dimensional vector (`rows`) contains the data for the table. Now you can create a `JTable` from the vectors and a `JFrame` to hold the `JTable`:

```
JTable table = new JTable(rows, labels);
JFrame jf = new JFrame("Browsing table: EMP (from EnterpriseDB)");
jf.getContentPane().add(new JScrollPane(table));
jf.setSize(400, 400);
jf.setVisible(true);
jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
System.out.println("Command successfully executed");
```

The `showEmployees()` method includes a `catch` block to intercept any errors that may occur and display an appropriate message to the user:

```
catch(Exception err)
{
  System.out.println("An error has occurred.");
  System.out.println("See full details below.");
  err.printStackTrace();
}
```

The result of calling the `showEmployees()` method is shown in below figure:



8 Advanced JDBC Connector Functionality

The previous example created a graphical user interface that displayed a result set in a `JTable`. Now we will switch gears and show you some of the more advanced features of the Advanced Server JDBC Connector.

To avoid unnecessary clutter, the rest of the code samples in this document will use the console to interact with the user instead of creating a graphical use interface.

8.1 Reducing Client-side Resource Requirements

The Advanced Server JDBC driver retrieves the results of a SQL query as a `ResultSet` object. If a query returns a large number of rows, using a batched `ResultSet` will:

- Reduce the amount of time it takes to retrieve the first row.
- Save time by retrieving only the rows that you need.
- Reduce the memory requirement of the client.

When you reduce the fetch size of a `ResultSet` object, the driver doesn't copy the entire `ResultSet` across the network (from the server to the client). Instead, the driver requests a small number of rows at a time; as the client application moves through the result set, the driver fetches the next batch of rows from the server.

Batched result sets cannot be used in all situations. Not adhering to the following restrictions will make the driver silently fall back to fetching the whole `ResultSet` at once:

- The client application must disable `autocommit`.
- The `Statement` object must be created with a `ResultSet` type of

`TYPE_FORWARD_ONLY` type (which is the default). `TYPE_FORWARD_ONLY` result sets can only step forward through the ResultSet.
- The query must consist of a single SQL statement.

Modifying the Batch Size of a Statement Object

Limiting the batch size of a `ResultSet` object can speed the retrieval of data and reduce the resources needed by a client-side application. Listing 1.5 creates a `Statement` object with a batch size limited to five rows:

```
// Make sure autocommit is off
conn.setAutoCommit(false);

Statement stmt = conn.createStatement();
// Set the Batch Size.
stmt.setFetchSize(5);

ResultSet rs = stmt.executeQuery("SELECT * FROM emp");
while (rs.next())
  System.out.println("a row was returned.");

rs.close();
stmt.close();
```

The call to `conn.setAutoCommit(false)` ensures that the server won't close the `ResultSet` before you have a chance to retrieve the first row. After preparing the `Connection`, you can construct a `Statement` object:

```
Statement stmt = db.createStatement();
```

The following code sets the batch size to five (rows) before executing the query:

```
stmt.setFetchSize(5);

ResultSet rs = stmt.executeQuery("SELECT * FROM emp");
```

For each row in the `ResultSet` object, the call to `println()` prints `a row was returned`.

```
System.out.println("a row was returned.");
```

Remember, while the `ResultSet` contains all of the rows in the table, they are only fetched from the server five rows at a time. From the client's point of view, the only difference between a `batched` result set and an `unbatched` result set is that a batched result may return the first row in less time.

Next, we will look at another feature (`the PreparedStatement`) that you can use to increase the performance of certain JDBC applications.

8.2 Using PreparedStatements to Send SQL Commands

Many applications execute the same SQL statement over and over again, changing one or more of the data values in the statement between each iteration. If you use a `Statement` object to repeatedly execute a SQL statement, the server must parse, plan, and optimize the statement every time. JDBC offers another `Statement` derivative, the `PreparedStatement` to reduce the amount of work required in such a scenario.

Listing 1.6 demonstrates invoking a `PreparedStatement` that accepts an employee ID and employee name and inserts that employee information in the `emp` table:

Listing 1.6

```
public void AddEmployee(Connection con)
{
  try
  {
    Console c = System.console();
```

```
    String command = "INSERT INTO emp(empno,ename) VALUES(?,?)";
    PreparedStatement stmt = con.prepareStatement(command);
    stmt.setObject(1,new Integer(c.readLine("ID:")));
    stmt.setObject(2,c.readLine("Name:"));
    stmt.execute();

    System.out.println("The procedure successfully executed.");
  }
  catch(Exception err)
  {
     System.out.println("An error has occurred.");
     System.out.println("See full details below.");
     err.printStackTrace();
  }
}
```

Instead of hard-coding data values in the SQL statement, you insert placeholders to represent the values that will change with each iteration. Listing 1.6 shows an INSERT statement that includes two placeholders (each represented by a question mark):

```
String command = "INSERT INTO emp(empno,ename) VALUES(?,?)";
```

With the parameterized SQL statement in hand, the AddEmployee() method can ask the Connection object to prepare that statement and return a PreparedStatement object:

```
PreparedStatement stmt = con.prepareStatement(command);
```

At this point, the PreparedStatement has parsed and planned the INSERT statement, but it does not know what values to add to the table. Before executing the PreparedStatement, you must supply a value for each placeholder by calling a setter method. setObject() expects two arguments:

- A parameter number; parameter number one corresponds to the first question mark, parameter number two corresponds to the second question mark, etc.
- The value to substitute for the placeholder.

The `AddEmployee()` method prompts the user for an employee ID and name and calls `setObject()` with the values supplied by the user:

```
stmt.setObject(1,new Integer(c.readLine("ID:")));
stmt.setObject(2, c.readLine("Name:"));
```

And then asks the `PreparedStatement` object to execute the statement:

```
stmt.execute();
```

If the SQL statement executes as expected, `AddEmployee()` displays a message that confirms the execution. If the server encounters an exception, the error handling code displays an error message.

8.3 Executing Stored Procedures

A stored procedure is a module that is written in EnterpriseDB's SPL and stored in the database. A stored procedure may define input parameters to supply data to the procedure and output parameters to return data from the procedure. Stored procedures execute within the server and consist of database access commands (SQL), control statements, and data structures that manipulate the data obtained from the database.

Stored procedures are especially useful when extensive data manipulation is required before storing data from the client. It is also efficient to use a stored procedure to manipulate data in a batch program.

Invoking Stored Procedures

The `CallableStatement` class provides a way for a Java program to call stored procedures. A `CallableStatement` object can have a variable number of parameters used for input (`IN` parameters), output (`OUT` parameters), or both (`IN OUT` parameters).

The syntax for invoking a stored procedure in JDBC is shown below. Note that the square brackets indicate optional parameters; they are not part of the command syntax.

```
{call procedure_name([?, ?, ...])}
```

The syntax to invoke a procedure that returns a result parameter is:

```
{? = call procedure_name([?, ?, ...])}
```

Each question mark serves as a placeholder for a parameter. The stored procedure determines if the placeholders represent `IN`, `OUT`, or `IN OUT` parameters and the Java code must match. We will show you how to supply values for `IN` (or `IN OUT`) parameters and how to retrieve values returned in `OUT` (or `IN OUT`) parameters in a moment.

Executing a Simple Stored Procedure

Listing 1.7-a shows a stored procedure that increases the salary of each employee by `10%`. `increaseSalary` expects no arguments from the caller and does not return any information:

```
CREATE OR REPLACE PROCEDURE increaseSalary
IS
  BEGIN
    UPDATE emp SET sal = sal * 1.10;
  END;
```

Listing 1.7-b demonstrates how to invoke the `increaseSalary` procedure:

```
public void SimpleCallSample(Connection con)
{
  try
  {
    CallableStatement stmt = con.prepareCall("{call increaseSalary()}");
    stmt.execute();
    System.out.println("Stored Procedure executed successfully");
```

```
  }
  catch(Exception err)
  {
    System.out.println("An error has occurred.");
    System.out.println("See full details below.");
    err.printStackTrace();
  }
}
```

To invoke a stored procedure from a Java application, use a `CallableStatement` object. The `CallableStatement` class is derived from the `Statement` class and, like the `Statement` class, you obtain a `CallableStatement` object by asking a `Connection` object to create one for you. To create a `CallableStatement` from a `Connection`, use the `prepareCall()` method:

```
CallableStatement stmt = con.prepareCall("{call increaseSalary()}");
```

As the name implies, the `prepareCall()` method prepares the statement, but does not execute it. As you will see in the next example, an application typically binds parameter values between the call to `prepareCall()` and the call to `execute()`. To invoke the stored procedure on the server, call the `execute()` method.

```
stmt.execute();
```

This stored procedure (`increaseSalary`) did not expect any `IN` parameters and did not return any information to the caller (using `OUT` parameters) so invoking the procedure is simply a matter of creating a `CallableStatement` object and then calling that object's `execute()` method.

The next section demonstrates how to invoke a stored procedure that requires data (`IN` parameters) from the caller.

Executing Stored Procedures with IN parameters

The code in the next example first creates and then invokes a stored procedure named `empInsert`; `empInsert` requires `IN` parameters that contain employee

information: `empno`, `ename`, `job`, `sal`, `comm`, `deptno`, and `mgr`. `empInsert` then inserts that information into the `emp` table.

Listing 1.8-a creates the stored procedure in the Advanced Server database:

```
CREATE OR REPLACE PROCEDURE empInsert(
    pEname  IN VARCHAR,
    pJob    IN VARCHAR,
    pSal    IN FLOAT4,
    pComm   IN FLOAT4,
pDeptno IN INTEGER,
pMgr    IN INTEGER
)
AS
DECLARE
  CURSOR getMax IS SELECT MAX(empno) FROM emp;
  max_empno INTEGER := 10;
BEGIN
  OPEN getMax;
  FETCH getMax INTO max_empno;
  INSERT INTO emp(empno, ename, job, sal, comm, deptno, mgr)
    VALUES(max_empno+1, pEname, pJob, pSal, pComm, pDeptno, pMgr);
  CLOSE getMax;
END;
```

Listing 1.8-b demonstrates how to invoke the stored procedure from Java:

```
public void CallExample2(Connection con)
{
  try
  {
    Console c = System.console();
    String commandText = "{call empInsert(?,?,?,?,?,?)}";
    CallableStatement stmt = con.prepareCall(commandText);
    stmt.setObject(1, new String(c.readLine("Employee Name :")));
    stmt.setObject(2, new String(c.readLine("Job :")));
    stmt.setObject(3, new Float(c.readLine("Salary :")));
```

```
    stmt.setObject(4, new Float(c.readLine("Commission :")));
    stmt.setObject(5, new Integer(c.readLine("Department No :")));
    stmt.setObject(6, new Integer(c.readLine("Manager")));
    stmt.execute();
  }
  catch(Exception err)
  {
    System.out.println("An error has occurred.");
    System.out.println("See full details below.");
    err.printStackTrace();
  }
}
```

Each placeholder (?) in the command (`commandText`) represents a point in the command that is later replaced with data:

```
String commandText = "{call EMP_INSERT(?,?,?,?,?,?)}";
CallableStatement stmt = con.prepareCall(commandText);
```

The `setObject()` method binds a value to an `IN` or `IN OUT` placeholder. Each call to `setObject()` specifies a parameter number and a value to bind to that parameter:

```
stmt.setObject(1, new String(c.readLine("Employee Name :")));
stmt.setObject(2, new String(c.readLine("Job :")));
stmt.setObject(3, new Float(c.readLine("Salary :")));
stmt.setObject(4, new Float(c.readLine("Commission :")));
stmt.setObject(5, new Integer(c.readLine("Department No :")));
stmt.setObject(6, new Integer(c.readLine("Manager")));
```

After supplying a value for each placeholder, this method executes the statement by calling the `execute()` method.

Executing Stored Procedures with OUT parameters

The next example creates and invokes an SPL stored procedure called

`deptSelect`. This procedure requires one `IN` parameter (department number) and returns two `OUT` parameters (the department name and location) corresponding to the department number. The code in Listing 1.9-a creates the `deptSelect` procedure:

```
CREATE OR REPLACE PROCEDURE deptSelect
(
  p_deptno IN  INTEGER,
  p_dname  OUT VARCHAR,
  p_loc    OUT VARCHAR
)
AS
DECLARE
  CURSOR deptCursor IS SELECT dname, loc FROM dept WHERE
deptno=p_deptno;
BEGIN
  OPEN deptCursor;
  FETCH deptCursor INTO p_dname, p_loc;

  CLOSE deptCursor;
END;
```

Listing 1.9-b shows the Java code required to invoke the `deptSelect` stored procedure:

```
public void GetDeptInfo(Connection con)
{
  try
  {
    Console c = System.console();
    String commandText = "{call deptSelect(?,?,?)}";
    CallableStatement stmt = con.prepareCall(commandText);
    stmt.setObject(1, new Integer(c.readLine("Dept No :")));
    stmt.registerOutParameter(2, Types.VARCHAR);
    stmt.registerOutParameter(3, Types.VARCHAR);
    stmt.execute();
    System.out.println("Dept Name: " + stmt.getString(2));
```

```
    System.out.println("Location : " + stmt.getString(3));
  }
  catch(Exception err)
  {
    System.out.println("An error has occurred.");
    System.out.println("See full details below.");
    err.printStackTrace();
  }
}
```

Each placeholder (?) in the command (commandText) represents a point in the command that is later replaced with data:

```
String commandText = "{call deptSelect(?,?,?)}";
CallableStatement stmt = con.prepareCall(commandText);
```

The setObject() method binds a value to an IN or IN OUT placeholder. When calling setObject() you must identify a placeholder (by its ordinal number) and provide a value to substitute in place of that placeholder: .. code-block:: text stmt.setObject(1, new Integer(c.readLine("Dept No :"))); The JDBC type of each OUT parameter must be registered before the CallableStatement object can be executed. Registering the JDBC type is done with the registerOutParameter() method. .. code-block:: text stmt.registerOutParameter(2, Types.VARCHAR); stmt.registerOutParameter(3, Types.VARCHAR); After executing the statement, the CallableStatement's getter method retrieves the OUT parameter values: to retrieve a VARCHAR value, use the getString() getter method. .. code-block:: text stmt.execute(); System.out.println("Dept Name: " + stmt.getString(2)); System.out.println("Location : " + stmt.getString(3)); In the current example GetDeptInfo() registers two OUT parameters and (after executing the stored procedure) retrieves the values returned in the OUT parameters. Since both OUT parameters are defined as VARCHAR values, GetDeptInfo() uses the getString() method to retrieve the OUT parameters. .. raw:: latex \newpage Executing Stored Procedures with IN OUT parameters ---------- .. index:: executing stored procedures with in out parameters The code in the next example creates and invokes a stored procedure named empQuery defined with one IN parameter (p_deptno), two IN OUT parameters (p_empno and p_ename) and three OUT parameters (p_job , p_hiredate and p_sal). empQuery then

returns information about the employee in the two IN OUT parameters and three OUT parameters. Listing 1.10-a creates a stored procedure named empQuery : .. code-block:: SQL CREATE OR REPLACE PROCEDURE empQuery (p_deptno IN NUMBER, p_empno IN OUT NUMBER, p_ename IN OUT VARCHAR2, p_job OUT VARCHAR2, p_hiredate OUT DATE, p_sal OUT NUMBER) IS BEGIN SELECT empno, ename, job, hiredate, sal INTO p_empno, p_ename, p_job, p_hiredate, p_sal FROM emp WHERE deptno = p_deptno AND (empno = p_empno OR ename = UPPER(p_ename)); END; Listing 1.10-b demonstrates invoking the empQuery procedure, providing values for the IN parameters, and handling the OUT and IN OUT parameters: .. code-block:: text public void CallSample4(Connection con) { try { Console c = System.console(); String commandText = "{call emp_query(?,?,?,?,?,?)}"; CallableStatement stmt = con.prepareCall(commandText); stmt.setInt(1, new Integer(c.readLine("Department No:"))); stmt.setInt(2, new Integer(c.readLine("Employee No:"))); stmt.setString(3, new String(c.readLine("Employee Name:"))); stmt.registerOutParameter(2, Types.INTEGER); stmt.registerOutParameter(3, Types.VARCHAR); stmt.registerOutParameter(4, Types.VARCHAR); stmt.registerOutParameter(5, Types.TIMESTAMP); stmt.registerOutParameter(6, Types.NUMERIC); stmt.execute(); System.out.println("Employee No: " + stmt.getInt(2)); System.out.println("Employee Name: " + stmt.getString(3)); System.out.println("Job : " + stmt.getString(4)); System.out.println("Hiredate : " + stmt.getTimestamp(5)); System.out.println("Salary : " + stmt.getBigDecimal(6)); } catch(Exception err) { System.out.println("An error has occurred."); System.out.println("See full details below."); err.printStackTrace(); } } Each placeholder (?) in the command (commandText) represents a point in the command that is later replaced with data: .. code-block:: text String commandText = "{call emp_query(?,?,?,?,?,?)}"; CallableStatement stmt = con.prepareCall(commandText); The setInt() method is a type-specific setter method that binds an Integer value to an IN or IN OUT placeholder. The call to setInt() specifies a parameter number and provides a value to substitute in place of that placeholder: .. code-block:: text stmt.setInt(1, new Integer(c.readLine("Department No:"))); stmt.setInt(2, new Integer(c.readLine("Employee No:"))); The setString() method binds a String value to an IN or IN OUT placeholder: .. code-block:: text stmt.setString(3, new String(c.readLine("Employee Name:"))); Before executing the CallableStatement , you must register the JDBC type of

each OUT parameter by calling the registerOutParameter() method. .. code-block:: text stmt.registerOutParameter(2, Types.INTEGER); stmt.registerOutParameter(3, Types.VARCHAR); stmt.registerOutParameter(4, Types.VARCHAR); stmt.registerOutParameter(5, Types.TIMESTAMP); stmt.registerOutParameter(6, Types.NUMERIC); Remember, before calling a procedure with an IN parameter, you must assign a value to that parameter with a setter method. Before calling a procedure with an OUT parameter, you register the type of that parameter; then you can retrieve the value returned by calling a getter method. When calling a procedure that defines an IN OUT`` parameter, you must perform all three actions:

- Assign a value to the parameter.
- Register the type of the parameter.
- Retrieve the value returned with a getter method.

8.4 Using REF CURSORS with Java

A REF CURSOR is a cursor variable that contains a pointer to a query result set returned by an OPEN statement. Unlike a static cursor, a REF CURSOR is not tied to a particular query. You may open the same REF CURSOR variable any number of times with the OPEN statement containing different queries; each time, a new result set is created for that query and made available via the cursor variable. A REF CURSOR can also pass a result set from one procedure to another.

Advanced Server supports the declaration of both strongly-typed and weakly-typed REF CURSORs. A strongly-typed cursor must declare the shape (the type of each column) of the expected result set. You can only use a strongly-typed cursor with a query that returns the declared columns; opening the cursor with a query that returns a result set with a different shape will cause the server to throw an exception. On the other hand, a weakly-typed cursor can work with a result set of any shape.

To declare a strongly-typed REF CURSOR:

```
TYPE <cursor_type_name> IS REF CURSOR RETURN <return_type>;
```

To declare a weakly-typed `REF_CURSOR`:

```
name SYS_REFCURSOR;

.. raw:: latex

  \newpage
```

Using a REF CURSOR to retrieve a ResultSet

The stored procedure shown in Listing 1.11-a (`getEmpNames`) builds two `REF CURSORs` on the server; the first `REF CURSOR` contains a list of commissioned employees in the `emp` table, while the second `REF CURSOR` contains a list of salaried employees in the `emp` table:

Listing 1.11-a

```
CREATE OR REPLACE PROCEDURE getEmpNames
(
  commissioned IN OUT SYS_REFCURSOR,
  salaried IN OUT SYS_REFCURSOR
)
IS
BEGIN
  OPEN commissioned FOR SELECT ename FROM emp WHERE comm is
NOT NULL;
  OPEN salaried FOR SELECT ename FROM emp WHERE comm is NULL;
END;
```

The `RefCursorSample()` method (see Listing 1.11-b) invokes the `getEmpName()` stored procedure and displays the names returned in each of the two `REF CURSOR` variables:

Listing 1.11-b

```
public void RefCursorSample(Connection con)
{
  try
  {
    con.setAutoCommit(false);
    String commandText = "{call getEmpNames(?,?)}";
    CallableStatement stmt = con.prepareCall(commandText);
    stmt.setNull(1, Types.REF);
    stmt.registerOutParameter(1, Types.REF);
    stmt.setNull(2, Types.REF);
    stmt.registerOutParameter(2, Types.REF);

    stmt.execute();
    ResultSet commissioned = (ResultSet)stmt.getObject(1);
    System.out.println("Commissioned employees:");
    while(commissioned.next())
    {
      System.out.println(commissioned.getString(1));
    }

    ResultSet salaried = (ResultSet)stmt.getObject(2);
    System.out.println("Salaried employees:");
    while(salaried.next())
    {
      System.out.println(salaried.getString(1));
    }
  }
  catch(Exception err)
  {
    System.out.println("An error has occurred.");
    System.out.println("See full details below.");
    err.printStackTrace();
  }
}
```

A `CallableStatement` prepares each `REF CURSOR` (`commissioned` and

salaried). Each cursor is returned as an IN OUT parameter of the stored procedure, getEmpNames():

```
String commandText = "{call getEmpNames(?,?)}";
CallableStatement stmt = con.prepareCall(commandText);
```

The call to registerOutParameter() registers the parameter type (Types.REF) of the first REF CURSOR (commissioned) :

```
stmt.setNull(1, Types.REF);
stmt.registerOutParameter(1, Types.REF);
```

Another call to registerOutParameter() registers the second parameter type (Types.REF) of the second REF CURSOR (salaried) :

```
stmt.setNull(2, Types.REF);
stmt.registerOutParameter(2, Types.REF);
```

A call to stmt.execute() executes the statement:

```
stmt.execute();
```

The getObject() method retrieves the values from the first parameter and casts the result to a ResultSet. Then, RefCursorSample iterates through the cursor and prints the name of each commissioned employee:

```
ResultSet commissioned = (ResultSet)stmt.getObject(1);
while(commissioned.next())
{
  System.out.println(commissioned.getString(1));
}
```

The same getter method retrieves the ResultSet from the second parameter and RefCursorExample iterates through that cursor, printing the name of each salaried employee:

```
ResultSet salaried = (ResultSet)stmt.getObject(2);
while(salaried.next())
```

```
{
  System.out.println(salaried.getString(1));
}
```

8.5 Using BYTEA Data with Java

The BYTEA data type stores a binary string in a sequence of bytes; digital ../../../../../../images and sound files are often stored as binary data. Advanced Server can store and retrieve binary data via the BYTEA data type.

The following Java sample stores BYTEA data in an Advanced Server database and then demonstrates how to retrieve that data. The example requires a bit of setup; Listings 1.12-a, 1.12-b, and 1.12-c create the server-side environment for the Java example.

Listing 1.12-a creates a table (emp_detail) that stores BYTEA data. emp_detail contains two columns: the first column stores an employee's ID number (type INT) and serves as the primary key for the table; the second column stores a photograph of the employee in BYTEA format:

```
CREATE TABLE emp_detail
(
  empno INT4 PRIMARY KEY,
  pic   BYTEA
);
```

Listing 1.12-b creates a procedure (ADD_PIC) that inserts a row into the emp_detail table:

```
CREATE OR REPLACE PROCEDURE ADD_PIC(p_empno IN int4, p_photo
IN bytea) AS

BEGIN
  INSERT INTO emp_detail VALUES(p_empno, p_photo);
END;
```

And finally, Listing 1.12-c creates a function (GET_PIC) that returns the photograph for a given employee:

```
CREATE OR REPLACE FUNCTION GET_PIC(p_empno IN int4) RETURN
BYTEA IS

DECLARE
  photo BYTEA;
BEGIN
  SELECT pic INTO photo from EMP_DETAIL WHERE empno = p_empno;
  RETURN photo;
END;
```

Inserting BYTEA Data into an Advanced Server Database

Listing 1.13 shows a Java method that invokes the ADD_PIC procedure (see Listing 1.12-b) to copy a photograph from the client file system to the emp_detail table on the server:

```
public void InsertPic(Connection con)
{
  try
  {
    Console c = System.console();
    int empno = Integer.parseInt(c.readLine("Employee No :"));
    String fileName = c.readLine("Image filename :");
    File f = new File(fileName);

    if(!f.exists())
```

```
    {
      System.out.println("Image file not found. Terminating...");
      return;
    }

    CallableStatement stmt = con.prepareCall("{call ADD_PIC(?, ?)}");
    stmt.setInt(1, empno);
    stmt.setBinaryStream(2, new FileInputStream(f), (int)f.length());
    stmt.execute();
    System.out.println("Added image for Employee "+empno);
  }
  catch(Exception err)
  {
    System.out.println("An error has occurred.");
    System.out.println("See full details below.");
    err.printStackTrace();
  }
}
```

`InsertPic()` prompts the user for an employee number and the name of an image file:

```
 int empno = Integer.parseInt(c.readLine("Employee No :"));
String fileName = c.readLine("Image filename :");
```

If the requested file does not exist, `InsertPic()` displays an error message and terminates:

```
 File f = new File(fileName);

if(!f.exists())
{
  System.out.println("Image file not found. Terminating...");
  return;
}
```

Next, `InsertPic()` prepares a `CallableStatement` object (`stmt`) that calls the

ADD_PIC procedure. The first placeholder (?) represents the first parameter expected by ADD_PIC (p_empno); the second placeholder represents the second parameter (p_photo). To provide actual values for those placeholders, InsertPic() calls two setter methods. Since the first parameter is of type INTEGER, InsertPic() calls the setInt() method to provide a value for p_empno. The second parameter is of type BYTEA, so InsertPic() uses a binary setter method; in this case, the method is setBinaryStream():

```
CallableStatement stmt = con.prepareCall("{call ADD_PIC(?, ?)}");
stmt.setInt(1, empno);
stmt.setBinaryStream(2 ,new FileInputStream(f), f.length());
```

Now that the placeholders are bound to actual values, InsertPic() executes the CallableStatement:

```
stmt.execute();
```

If all goes well, InsertPic() displays a message verifying that the image has been added to the table. If an error occurs, the catch block displays a message to the user:

```
System.out.println("Added image for Employee \""+empno);
catch(Exception err)
{
  System.out.println("An error has occurred.");
  System.out.println("See full details below.");
  err.printStackTrace();
}
```

Retrieving BYTEA Data from an Advanced Server Database

Now that you know how to insert BYTEA data from a Java application, Listing 1.14 demonstrates how to retrieve BYTEA data from the server:

```
public static void GetPic(Connection con)
{
```

```
  try
  {
    Console c = System.console();
    int empno = Integer.parseInt(c.readLine("Employee No :"));
    CallableStatement stmt = con.prepareCall("{?=call GET_PIC(?)}");
    stmt.setInt(2, empno);
    stmt.registerOutParameter(1, Types.BINARY);
    stmt.execute();
    byte[] b = stmt.getBytes(1);

    String fileName = c.readLine("Destination filename :");
    FileOutputStream fos = new FileOutputStream(new File(fileName));
    fos.write(b);
    fos.close();
    System.out.println("File saved at \""+fileName+"\"");
  }
  catch(Exception err)
  {
    System.out.println("An error has occurred.");
    System.out.println("See full details below.");
    err.printStackTrace();
  }
}
```

`GetPic()` starts by prompting the user for an employee ID number:

```
int empno = Integer.parseInt(c.readLine("Employee No :"));
```

Next, `GetPic()` prepares a `CallableStatement` with one `IN` parameter and one `OUT` parameter. The first parameter is the `OUT` parameter that will contain the photograph retrieved from the database. Since the photograph is `BYTEA` data, `GetPic()` registers the parameter as a `Type.BINARY`. The second parameter is the `IN` parameter that holds the employee number (an `INT`), so `GetPic()` uses the `setInt()` method to provide a value for the second parameter.

```
CallableStatement stmt = con.prepareCall("{?=call GET_PIC(?)}");
stmt.setInt(2, empno);
```

```
stmt.registerOutParameter(1, Types.BINARY);
```

Next, GetPic() uses the getBytes getter method to retrieve the BYTEA data from the CallableStatement:

```
stmt.execute();
byte[] b = stmt.getBytes(1);
```

The program prompts the user for the name of the file where it will store the photograph:

```
String fileName = c.readLine("Destination filename :");
```

The FileOutputStream object writes the binary data that contains the photograph to the destination filename:

```
FileOutputStream fos = new FileOutputStream(new File(fileName));
fos.write(b);
fos.close();
```

Finally, GetPic() displays a message confirming that the file has been saved at the new location:

```
System.out.println("File saved at \""+fileName+"\"");
```

8.6 Using Object Types and Collections with Java

The SQL CREATE TYPE command is used to create a user-defined object type, which is stored in the Advanced Server database. The CREATE TYPE command is also used to create a collection, commonly referred to as an array, which is also stored in the Advanced Server database.

These user-defined types can then be referenced within SPL procedures, SPL functions, and Java programs.

The basic object type is created with the `CREATE TYPE AS OBJECT` command along with optional usage of the `CREATE TYPE BODY` command.

A `nested table type` collection is created using the `CREATE TYPE AS TABLE OF` command. A `varray type` collection is created with the `CREATE TYPE VARRAY` command.

Example usage of an object type and a collection are shown in the following sections.

Listing 1.15 shows a Java method used by both examples to establish the connection to the Advanced Server database.

```
public static Connection getEDBConnection() throws
  ClassNotFoundException, SQLException {
  String url = "jdbc:edb://localhost:5444/test";
  String user = "enterprisedb";
  String password = "edb";
  Class.forName("com.edb.Driver");
  Connection conn = DriverManager.getConnection(url, user, password);
  return conn;
}
```

Using an Object Type

Create the object types in the Advanced Server database. Object type `addr_object_type` defines the attributes of an address:

```
CREATE OR REPLACE TYPE addr_object_type AS OBJECT
(
    street          VARCHAR2(30),
    city            VARCHAR2(20),
    state           CHAR(2),
    zip             NUMBER(5)
);
```

Object type `emp_obj_typ` defines the attributes of an employee. Note that one

of these attributes is object type `ADDR_OBJECT_TYPE` as previously described. The object type body contains a method that displays the employee information:

```
CREATE OR REPLACE TYPE emp_obj_typ AS OBJECT
(
   empno          NUMBER(4),
   ename          VARCHAR2(20),
   addr          ADDR_OBJECT_TYPE,
   MEMBER PROCEDURE display_emp(SELF IN OUT emp_obj_typ)
);

CREATE OR REPLACE TYPE BODY emp_obj_typ AS
 MEMBER PROCEDURE display_emp (SELF IN OUT emp_obj_typ)
 IS
 BEGIN
   DBMS_OUTPUT.PUT_LINE('Employee No   : ' || SELF.empno);
   DBMS_OUTPUT.PUT_LINE('Name          : ' || SELF.ename);
   DBMS_OUTPUT.PUT_LINE('Street        : ' || SELF.addr.street);
   DBMS_OUTPUT.PUT_LINE('City/State/Zip: ' || SELF.addr.city || ', ' ||
     SELF.addr.state || ' ' || LPAD(SELF.addr.zip,5,'0'));
 END;
END;
```

Listing 1.16 is a Java method that includes these user-defined object types:

```
public static void testUDT() throws SQLException {
 Connection conn = null;
 try {
   conn = getEDBConnection();
   String commandText = "{call emp_obj_typ.display_emp(?)}";
   CallableStatement stmt = conn.prepareCall(commandText);

   // initialize emp_obj_typ structure
   // create addr_object_type structure
   Struct address = conn.createStruct("addr_object_type",
     new Object[]{"123 MAIN STREET","EDISON","NJ",8817});
```

```
    Struct emp     = conn.createStruct("emp_obj_typ",
      new Object[]{9001,"JONES", address});

    // set emp_obj_typ type param
    stmt.registerOutParameter(1, Types.STRUCT, "emp_obj_typ");
    stmt.setObject(1, emp);
    stmt.execute();

    // extract emp_obj_typ object
    emp = (Struct)stmt.getObject(1);
    Object[] attrEmp = emp.getAttributes();
    System.out.println("empno: " + attrEmp[0]);
    System.out.println("ename: " + attrEmp[1]);

    // extract addr_object_type attributes
    address = (Struct) attrEmp[2];
    Object[] attrAddress = address.getAttributes();
    System.out.println("street: " + attrAddress[0]);
    System.out.println("city: " + attrAddress[1]);
    System.out.println("state: " + attrAddress[2]);
    System.out.println("zip: " + attrAddress[3]);
  } catch (ClassNotFoundException cnfe) {
    System.err.println("Error: " + cnfe.getMessage());
  } finally {
    if (conn != null) {
      conn.close();
    }
  }
}
```

A `CallableStatement` object is prepared based on the `display_emp()` method of the `emp_obj_typ` object type:

```
String commandText = "{call emp_obj_typ.display_emp(?)}";
CallableStatement stmt = conn.prepareCall(commandText);
```

`createStruct()` initializes and creates instances of object types `addr_object_type` and `emp_obj_typ` named `address` and `emp`, respectively:

```
Struct address = conn.createStruct("addr_object_type",
  new Object[]{"123 MAIN STREET","EDISON","NJ",8817});
Struct emp     = conn.createStruct("emp_obj_typ",
  new Object[]{9001,"JONES", address});
```

The call to `registerOutParameter()` registers the parameter type (`Types.STRUCT`) of `emp_obj_typ`:

```
stmt.registerOutParameter(1, Types.STRUCT, "emp_obj_typ");
```

The `setObject()` method binds the object instance `emp` to the `IN OUT` placeholder.

```
stmt.setObject(1, emp);
```

A call to `stmt.execute()` executes the call to the `display_emp()` method:

```
stmt.execute();
```

`getObject()` retrieves the `emp_obj_typ` object type. The attributes of the `emp` and `address` object instances are then retrieved and displayed:

```
emp = (Struct)stmt.getObject(1);
Object[] attrEmp = emp.getAttributes();
System.out.println("empno: " + attrEmp[0]);
System.out.println("ename: " + attrEmp[1]);

address = (Struct) attrEmp[2];
Object[] attrAddress = address.getAttributes();
System.out.println("street: " + attrAddress[0]);
System.out.println("city: " + attrAddress[1]);
System.out.println("state: " + attrAddress[2]);
System.out.println("zip: " + attrAddress[3]);
```

Using a Collection

Create collection types `NUMBER_ARRAY` and `CHAR_ARRAY` in the Advanced Server database:

```
CREATE OR REPLACE TYPE NUMBER_ARRAY  AS TABLE OF NUMBER;
CREATE OR REPLACE TYPE CHAR_ARRAY  AS TABLE OF
VARCHAR(50);
```

Listing 1.17-a is an SPL function that uses collection types `NUMBER_ARRAY` and `CHAR_ARRAY` as `IN` parameters and `CHAR_ARRAY` as the `OUT` parameter.

The function concatenates the employee ID from the `NUMBER_ARRAY IN` parameter with the employee name in the corresponding row from the `CHAR_ARRAY IN` parameter. The resulting concatenated entries are returned in the `CHAR_ARRAY OUT` parameter.

```
CREATE OR REPLACE FUNCTION concatEmpIdName
(
    arrEmpIds       NUMBER_ARRAY,
    arrEmpNames     CHAR_ARRAY
) RETURN CHAR_ARRAY
AS
DECLARE
    i               INTEGER := 0;
    arrEmpIdNames   CHAR_ARRAY;
BEGIN
  arrEmpIdNames := CHAR_ARRAY(NULL,NULL);
  FOR i IN arrEmpIds.FIRST..arrEmpIds.LAST LOOP
    arrEmpIdNames(i) := arrEmpIds(i) || '  ' || arrEmpNames(i);
  END LOOP;
  RETURN arrEmpIdNames;
END;
```

Listing 1.17-b is a Java method that calls the Listing 1.17-a function, passing and retrieving the collection types:

```
public static void testTableOfAsInOutParams() throws SQLException {
 Connection conn = null;
 try {
  conn = getEDBConnection();
  String commandText = "{? = call concatEmpIdName(?,?)}";
  CallableStatement stmt = conn.prepareCall(commandText);

  // create collections to specify employee id and name values
  Array empIdArray = conn.createArrayOf("integer",
   new Integer[]{7900, 7902});
  Array empNameArray = conn.createArrayOf("varchar",
   new String[]{"JAMES", "FORD"});

  // set TABLE OF VARCHAR as OUT param
  stmt.registerOutParameter(1, Types.ARRAY);

  // set TABLE OF INTEGER as IN param
  stmt.setObject(2, empIdArray, Types.OTHER);

  // set TABLE OF VARCHAR as IN param
  stmt.setObject(3, empNameArray, Types.OTHER);
  stmt.execute();
  java.sql.Array empIdNameArray = stmt.getArray(1);
  String[] emps = (String[]) empIdNameArray.getArray();

  System.out.println("items length: " + emps.length);
  System.out.println("items[0]: " + emps[0].toString());
  System.out.println("items[1]: " + emps[1].toString());

 } catch (ClassNotFoundException cnfe) {
  System.err.println("Error: " + cnfe.getMessage());
 } finally {
  if (conn != null) {
   conn.close();
  }
 }
```

```
}
```

A `CallableStatement` object is prepared to invoke the `concatEmpIdName()` function:

```
String commandText = "{? = call concatEmpIdName(?,?)}";
CallableStatement stmt = conn.prepareCall(commandText);
```

`createArrayOf()` initializes and creates collections named `empIdArray` and `empNameArray`:

```
Array empIdArray = conn.createArrayOf("integer",
  new Integer[]{7900, 7902});
Array empNameArray = conn.createArrayOf("varchar",
  new String[]{"JAMES", "FORD"});
```

The call to `registerOutParameter()` registers the parameter type (`Types.ARRAY`) of the `OUT` parameter:

```
stmt.registerOutParameter(1, Types.ARRAY);
```

The `setObject()` method binds the collections `empIdArray` and `empNameArray` to the `IN` placeholders:

```
stmt.setObject(2, empIdArray, Types.OTHER);
stmt.setObject(3, empNameArray, Types.OTHER);
```

A call to `stmt.execute()` invokes the `concatEmpIdName()` function:

```
stmt.execute();
```

`getArray()` retrieves the collection returned by the function. The first two rows consisting of the concatenated employee IDs and names are displayed:

```
java.sql.Array empIdNameArray = stmt.getArray(1);
String[] emps = (String[]) empIdNameArray.getArray();
System.out.println("items length: " + emps.length);
System.out.println("items[0]: " + emps[0].toString());
```

```
System.out.println("items[1]: " + emps[1].toString());
```

8.7 Asynchronous Notification Handling with NoticeListener

The Advanced Server JDBC Connector provides asynchronous notification handling functionality. A `notification` is a message generated by the server when an SPL (or PL/pgSQL) program executes a `RAISE NOTICE` statement. Each notification is sent from the server to the client application. To intercept a notification in a JDBC client, an application must create a `NoticeListener` object (or, more typically, an object derived from `NoticeListener`).

It is important to understand that a notification is sent to the client as a result of executing an SPL (or PL/pgSQL) program. To generate a notification, you must execute an SQL statement that invokes a stored procedure, function, or trigger: the notification is delivered to the client as the SQL statement executes. Notifications work with any type of statement object; `CallableStatement` objects, `PreparedStatement` objects, or simple `Statement` objects. A JDBC program intercepts a notification by associating a `NoticeListener` with a `Statement` object. When the `Statement` object executes an SQL statement that raises a notice, JDBC invokes the `noticeReceived()` method in the associated `NoticeListener`.

Listing 1.18-a shows an SPL procedure that loops through the `emp` table and gives each employee a 10% raise. As each employee is processed, `adjustSalary` executes a `RAISE NOTICE` statement (in this case, the message contained in the notification reports progress to the client application):

```
CREATE OR REPLACE PROCEDURE adjustSalary
IS
  v_empno        NUMBER(4);
  v_ename        VARCHAR2(10);
  CURSOR emp_cur IS SELECT empno, ename FROM emp;
BEGIN
  OPEN emp_cur;
  LOOP
    FETCH emp_cur INTO v_empno, v_ename;
    EXIT WHEN emp_cur%NOTFOUND;

    UPDATE emp SET sal = sal * 1.10 WHERE empno = v_empno;
    RAISE NOTICE 'Salary increased for %', v_ename;
  END LOOP;
  CLOSE emp_cur;
END;
```

Listing 1.18-b shows how to create a `NoticeListener` that intercepts notifications in a JDBC application:

```
public void NoticeExample(Connection con)
{
  CallableStatement stmt;
  try
  {
    stmt = con.prepareCall("{call adjustSalary()}");

    MyNoticeListener listener = new MyNoticeListener();
    ((BaseStatement)stmt).addNoticeListener(listener);
    stmt.execute();
    System.out.println("Finished");
  }
  catch (SQLException e)
  {
    System.out.println("An error has occurred.");
    System.out.println("See full details below.");
```

```
    e.printStackTrace();
  }
}
class MyNoticeListener implements NoticeListener
{
  public MyNoticeListener()
  {
  }

  public void noticeReceived(SQLWarning warn)
  {
    System.out.println("NOTICE: "+ warn.getMessage());
  }
}
```

The `NoticeExample()` method is straightforward; it expects a single argument, a `Connection` object, from the caller:

```
public void NoticeExample(Connection con)
```

`NoticeExample()` begins by preparing a call to the `adjustSalary` procedure shown in example 1.10-a. As you would expect, `con.prepareCall()` returns a `CallableStatement` object. Before executing the `CallableStatement`, you must create an object that implements the `NoticeListener` interface and add that object to the list of `NoticeListeners` associated with the `CallableStatement`:

```
CallableStatement stmt = con.prepareCall("{call adjustSalary()}");
MyNoticeListener listener = new MyNoticeListener();
((BaseStatement)stmt).addNoticeListener(listener);
```

Once the `NoticeListener` is in place, `NoticeExample` method executes the `CallableStatement` (invoking the `adjustSalary` procedure on the server) and displays a message to the user:

```
stmt.execute();
System.out.println("Finished");
```

Each time the `adjustSalary` procedure executes a `RAISE NOTICE` statement,

the server sends the text of the message (`"Salary increased for ..."`) to the `Statement` (or derivative) object in the client application. JDBC invokes the `noticeReceived()` method (possibly many times) `before` the call to `stmt.execute()` completes.

```
class MyNoticeListener implements NoticeListener
{
  public MyNoticeListener()
  {
  }

  public void noticeReceived(SQLWarning warn)
  {
    System.out.println("NOTICE: "+ warn.getMessage());
  }
}
```

When JDBC calls the `noticeReceived()` method, it creates an `SQLWarning` object that contains the text of the message generated by the `RAISE NOTICE` statement on the server.

Notice that each `Statement` object keeps a `list` of `NoticeListeners`. When the JDBC driver receives a notification from the server, it consults the list maintained by the Statement object. If the list is empty, the notification is saved in the Statement object (you can retrieve the notifications by calling stmt.getWarnings() once the call to execute() completes). If the list is not empty, the JDBC driver delivers an SQLWarning to each listener, in the order in which the listeners were added to the Statement`.

9 Security and Encryption

9.1 Using SSL

In this section, you will learn about:

- Configuring the server
- Configuring the client
- Testing the SSL JDBC Connection
- Using SSL without Certificate Validation
- Using Certificate Authentication (without a password)

9.1.1 Configuring the Server

For information about configuring PostgreSQL or Advanced Server for SSL, refer to:

> https://www.enterprisedb.com/edb-docs/d/postgresql/reference/manual/12.3/ssl-tcp.html

Note

Before you access your SSL enabled server from Java, ensure that you can log in to your server via `edb-psql`. The sample output should look similar to the one shown below if you have established a SSL connection:

```
$ ./bin/edb-psql -U enterprisedb -d edb
psql.bin (12.0.1)
SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384, bits: 256, compression: off)
Type "help" for help.
```

```
edb=#
```

9.1.2 Configuring the Client

There are a number of connection parameters for configuring the client for SSL. To know more about the SSL Connection parameters and Additional Connection Properties, refer to Section 5.2.

In this section, we will discuss more about the behavior of ssl connection parameter when passed with different values. When you pass the connection parameter `ssl=true` into the driver, the driver validates the SSL certificate and verifies the hostname. On contrary to this behavior, using `libpq` defaults to a non-validating SSL connection.

You can get better control of the SSL connection using the `sslmode` connection parameter. This parameter is the same as the `libpq sslmode` parameter and the existing SSL implements the following sslmode connection parameters.

sslmode Connection Parameters

sslmode=require

This mode makes the encryption mandatory and also requires the connection to fail if it can't be encrypted. The server is configured to accept SSL connections for this Host/IP address and that the server recognizes the client certificate.

Note

In this mode, the JDBC driver accepts all server certificates.

sslmode=verify-ca

If `sslmode=verify-ca`, the server is verified by checking the certificate chain up to the root certificate stored on the client.

sslmode=verify-full

If `sslmode=verify-full`, the server host name is verified to make sure it matches the name stored in the server certificate. The SSL connection fails if the server certificate cannot be verified. This mode is recommended in most security-sensitive environments.

In the case where the certificate validation is failing you can try `sslcert=` and `LibPQFactory` will not send the client certificate. If the server is not configured to authenticate using the certificate it should connect.

The location of the client certificate, client key and root certificate can be overridden with the `sslcert`, `sslkey`, and `sslrootcert` settings respectively. These default to `/defaultdir/postgresql.crt`, `/defaultdir/postgresql.pk8`, and `/defaultdir/root.crt` respectively where defaultdir is ${user.home}/.postgresql/ in unix systems and %appdata%/postgresql/ on windows.

In this mode, when establishing a SSL connection the JDBC driver will validate the server's identity preventing "man in the middle" attacks. It does this by checking that the server certificate is signed by a trusted authority, and that the host you are connecting to, is the same as the hostname in the certificate.

9.1.3 Testing the SSL JDBC Connection

If you are using Java's default mechanism (not `LibPQFactory`) to create the SSL connection, you need to make the server certificate available to Java, which can be achieved by implementing steps given below:

1. Set the below property in the Java program.

String url="jdbc:edb://localhost/test?user=fred&password=secret&ssl=true";

2. Convert the server certificate to Java format:

3. Import this certificate into Java's system truststore.
4. If you do not have access to the system cacerts truststore, create your own truststore as below:

```
$ keytool -keystore mystore -alias postgresql -import -file server.crt.der
```

5. Start your Java application and test the program.

```
$ java -Djavax.net.ssl.trustStore=mystore com.mycompany.MyApp
```

For example:

```
$java -classpath .:/usr/edb/jdbc/edb-jdbc18.jar–
Djavax.net.ssl.trustStore=mystore pg_test2 public
```

Note

For troubleshooting connection issues, add -Djavax.net.debug=ssl to the java command.

Using SSL without Certificate Validation

By default the combination of `SSL=true` and setting the connection URL parameter `sslfactory=com.edb.ssl.NonValidatingFactory` encrypts the connection but does not validate the SSL certificate. To enforce certificate validation, you must use a `Custom SSLSocketFactory`.

For more details about writing a `Custom SSLSocketFactory`, refer to:

> https://jdbc.postgresql.org/documentation/head/ssl-factory.html

---

9.1.4 Using Certificate Authentication (Without a

Password)

In order to use certificate authentication (without a password), follow the below steps:

1. Convert the client certificate to DER format.

```
$ openssl x509 –in postgresql.crt -out postgresql.crt.der -outform
der
```

2. Convert the client key to DER format.

```
$ openssl pkcs8 -topk8 -outform DER -in postgresql.key -out
postgresql.key.pk8 –nocrypt
```

3. Copy the client files (`postgresql.crt.der`, `postgresql.key.pk8`) and root certificate to the client machine and use the following properties in your java program to test it:

```
String url = "jdbc:edb://snvm001:5444/edbstore";
    Properties props = new Properties();
    props.setProperty("user","enterprisedb");
    props.setProperty("ssl","true");
    props.setProperty("sslmode","verify-full");
    props.setProperty("sslcert","postgresql.crt.der");
    props.setProperty("sslkey","postgresql.key.pk8");
    props.setProperty("sslrootcert","root.crt");
```

4. Compile the Java program and test it.

```
$ java -Djavax.net.ssl.trustStore=mystore -classpath
.:./edb-jdbc18.jar pg_ssl public
```

9.2 Scram Compatibility

The EDB JDBC driver provides SCRAM-SHA-256 support for Advanced Server versions 10, 11, and 12. For JRE/JDK version 1.8, this support is available from EDB JDBC Connector release 42.2.2.1 onwards; for JRE/JDK version 1.7, this support is available from EDB JDBC Connector release 42.2.5 onwards.

10 Advanced Server JDBC Connector Logging

The Advanced Server JDBC Connector supports the use of logging to help resolve issues with the JDBC Connector when used in your application. The JDBC Connector uses the logging APIs of `java.util.logging` that was part of Java since JDK 1.4. For information on `java.util.logging`, see The PostgreSQL JDBC Driver.

Note

Previous versions of the Advanced Server JDBC Connector used a custom mechanism to enable logging, which is now replaced by the use of `java.util.logging` in versions moving forward from community version 42.1.4.1. The older mechanism is no longer available.

Enabling Logging with Connection Properties (static)

You can directly configure logging within the connection properties of the JDBC Connector. The connection properties related to logging are `loggerLevel` and `loggerFile`.

`loggerLevel`

Logger level of the driver. Allowed values are `OFF`, `DEBUG`, or `TRACE`. This option enables the `java.util.logging.Logger` level of the driver to correspond to

the following Advanced Server JDBC levels:

loggerLevel	java.util.logging
OFF	OFF
DEBUG	FINE
TRACE	FINEST

`loggerFile`

File name output of the logger. The default is `java.util.logging.ConsoleHandler`. The following example sets the logging level to `TRACE (FINEST)` and the log file `to EDB-JDBC.LOG`:

```
jdbc:edb://myhost:5444/mydb?loggerLevel=TRACE&loggerFile=EDB-JDBC.LOG
```

Enabling Logging with logging.properties (dynamic)

You can use logging properties to configure the driver dynamically (for example, when using the JDBC Connector with an application server such as Tomcat, JBoss, WildFly, etc.), which makes it easier to enable/disable logging at runtime. The following example demonstrates configuring logging dynamically:

```
handlers = java.util.logging.FileHandler
//logging level
.level = OFF
```

The default file output is in the user's home directory:

```
java.util.logging.FileHandler.pattern = %h/EDB-JDBC%u.log
java.util.logging.FileHandler.limit = 5000000
java.util.logging.FileHandler.count = 20
java.util.logging.FileHandler.formatter = java.util.logging.SimpleFormatter
java.util.logging.FileHandler.level = FINEST
java.util.logging.SimpleFormatter.format=%1$tY-%1$tm-%1$td
```

```
%1$tH:%1$tM:%1$tS %4$s %2$s %5$s%6$s%n
```

Use the following command to set the logging level for the JDBC Connector to FINEST (maps to `loggerLevel`):

```
com.edb.level=FINEST
```

Then, execute the application with the logging configuration:

```
java –jar -Djava.util.logging.config.file=logging.properties run.jar
```

11 Reference - JDBC Data Types

The following table lists the JDBC data types supported by Advanced Server and the JDBC connector. If you are binding to an Advanced Server type (shown in the middle column) using the `setObject()` method, supply a JDBC value of the type shown in the left column. When you retrieve data, the `getObject()` method will return the object type listed in the right-most column:

JDBC Type	Advanced Server Type	getObject() returns
INTEGER	INT4	java.lang.Integer
TINYINT, SMALLINT	INT2	java.lang.Integer
BIGINT	INT8	java.lang.Long
REAL	FLOAT4	java.lang.Float
DOUBLE, FLOAT	FLOAT8	java.lang.Double (Float is same as double)
DECIMAL, NUMERIC	NUMERIC	java.math.BigDecimal
CHAR	BPCHAR	java.lang.String
VARCHAR, LONGVARCHAR	VARCHAR	java.lang.String
DATE	DATE	java.sql.Date
TIME	TIME, TIMETZ	java.sql.Timestamp

JDBC Type	Advanced Server Type	getObject() returns
TIMESTAMP	TIMESTAMP, TIMESTAMPTZ	java.sql.Timestamp
BINARY	BYTEA	byte[](primitive)
BOOLEAN, BIT	BOOL	java.lang.Boolean
Types.REF	REFCURSOR	java.sql.ResultSet
Types.REF_CURSOR	REFCURSOR	java.sql.ResultSet
Types.OTHER	REFCURSOR	java.sql.ResultSet
Types.OTHER	UUID	java.util.UUID
Types.SQLXML	XML	java.sql.SQLXML

Note

`Types.REF_CURSOR` is only supported for JRE 4.2.

`Types.OTHER` is not only used for UUID, but is also used if you do not specify any specific type and allow the server or the JDBC driver to determine the type. If the parameter is an instance of `java.util.UUID`, the driver determines the appropriate internal type and sends it to the server.