EDB JDBC Connector

Version 42.7.3.4

Built at 2026-01-27T09:46:49

1 EDB JDBC Connector

The EDB JDBC Connector provides connectivity between a Java application and an EDB Postgres Advanced Server database. The EDB JDBC Connector is written in Java and conforms to Sun's JDK architecture. For more information, see JDBC driver types

The EDB JDBC Connector is built on and supports all of the functionality of the PostgreSQL community driver. For more information about the features and functionality of the driver, please see the community documentation.

2 Release notes

The EDB JDBC connector documentation describes the latest version of EDB JDBC connector.

These release notes describe what's new in each release. When a minor or patch release introduces new functionality, indicators in the content identify the version that introduced the new feature.

Version	Release Date
42.7.3.4	25 Nov 2025
42.7.3.3	21 May 2025
42.7.3.2	21 Nov 2024
42.7.3.1	10 Sep 2024
42.5.4.2	26 Feb 2024
42.5.4.1	16 Mar 2023
42.5.1.2	14 Feb 2023
42.5.1.1	09 Dec 2022
42.5.0.1	01 Sep 2022
42.3.3.1	20 Apr 2022
42.3.2.1	15 Feb 2022
42.2.24.1	5 Nov 2021
42.2.19.1	15 Apr 2021
42.2.12.3	22 Oct 2020
42.2.9.1	18 May 2020
42.2.8.1	21 Oct 2019

2.1 EDB JDBC Connector 42.7.3.4 release notes

Released: 25 Nov 2025

The EDB JDBC connector provides connectivity between a Java application and an EDB Postgres Advanced Server database.

New features, enhancements, bug fixes, and other changes in the EDB JDBC Connector 42.7.3.4 include:

Type	Description	Addresses
Enhancement	Added support for EDB Postgres Advanced Server 18.	

2.2 EDB JDBC Connector 42.7.3.3 release notes

Released: 21 May 2025

The EDB JDBC connector provides connectivity between a Java application and an EDB Postgres Advanced Server database.

New features, enhancements, bug fixes, and other changes in the EDB JDBC Connector 42.7.3.3 include:

Type	Description	Addresses
Enhancement	Added support for EDB Postgres Advanced Server 13 to 17.	
Bug fix	Fixed an issue where `getUpdateCounts` was incorrect when using `edb_stmt_level_tx` to `on`.	#45496
Bug Fix	Fixed an issue where duplicate messages appeared in the JMS queue.	

2.3 EDB JDBC Connector 42.7.3.2 release notes

Released: 22 Nov 2024

The EDB JDBC connector provides connectivity between a Java application and an EDB Postgres Advanced Server database.

New features, enhancements, bug fixes, and other changes in the EDB JDBC Connector 42.7.3.2 include:

Type	Description
Performance	Improved parsing performance with large SQL (MTK/SQL Plus).
Enhancement	Added support for EDB Postgres Advanced Server 17.2.
Bug fix	Fixes an incompatibility issue with JDK 8 that was found in version 42.7.3.1 of the `edb-jdbc18` driver.
Bug Fix	`edb-jdbc` installation should not install a lower JDK version when a higher version is installed.
Bug Fix	Fixed issue where `Message.getJMSMessageID()` returns `null`.
Bug Fix	Fixed issue with determining the queue table for a queue when there is more than one queue defined within a single schema.
Bug Fix	Fixed issue where `EDBJmsMessageConsumer.receiveNoWait()` returns `null` even when messages are available on the queue.
Bug Fix	Fixed issue where `EDBJmsMessageConsumer.receive()` [without time parameter] fails to block until a message is available.
Bug Fix	Fixed issue where `EDBJmsMessageConsumer.receive(timeout)` doesn't honor the timeout specified.

2.4 EDB JDBC Connector 42.7.3.1 release notes

Released: 10 Sep 2024

The EDB JDBC connector provides connectivity between a Java application and an EDB Postgres Advanced Server database.

New features, enhancements, bug fixes, and other changes in the EDB JDBC Connector 42.7.3.1 include:

Type	Description	Addresses
Upstream Merge	Merged with the upstream community driver version 42.7.3. See the community JDBC documentation for details.	
Enhancement	Improved the parsing issue with the large SQL statements (for MTK/SQL Plus).	
Enhancement	JMS Enhancements - The EDB JMS API has been made according to the JMS standard. All supported JMS classes related to Factory, Connection, Session, Producer, Consumer and Message types can now be used in a standard way. - DefaultMessageListenerContainer can now be used to continuously pull messages from EDB JMS Queue. -Transacted Sessions are implemented.	
Enhancement	Fixed null pointer exception in case of timeout or end-of-fetch during message dequeue.	#37882
Enhancement	EDB JMS API now supports the basic Apache Camel Route concept as a source and destination.	#37882
Enhancement	JMS message types, such as message, text message, bytes message, and object message, are now supported.	#37884
Enhancement	EDBJmsConnectionFactory now has an alternative constructor that takes SQL Connection as a parameter.	#38465
Enhancement	- EDBJmsConnection now implements the critical lifecycle methods start() and stop(). - EDBJmsSession now implements the critical close() method. - EDBJmsSession.createQueue now returns a valid queue instance. - EDB JMS message types are now aligned with the JMS standard. The following message types are now supported: 1. aq$_jms_message 2. aq$_jms_text_message 3. aq$_jms_bytes_message 4. aq$_jms_object_message - All message types now support set*Property() and get*Property() for setting and getting properties of JMS supported types.	#38542

2.5 EDB JDBC Connector 42.5.4.2 release notes

Released: 26 Feb 2024

The EDB JDBC connector provides connectivity between a Java application and an EDB Postgres Advanced Server database.

New features, enhancements, bug fixes, and other changes in the EDB JDBC Connector 42.5.4.2 include:

Type	Description
Security Fix	CVE-2024-1597 - As outlined in the Security Advisory, SQL injection is possible while using a non-default connection property (preferQueryMode=simple) along with application code that has a vulnerable SQL that negates a parameter value. There is no vulnerability in the driver while using the default query mode.

2.6 EDB JDBC Connector 42.5.4.1 release notes

Released: 16 Mar 2023

The EDB JDBC connector provides connectivity between a Java application and an EDB Postgres Advanced Server database.

New features, enhancements, bug fixes, and other changes in the EDB JDBC Connector 42.5.4.1 include:

Type	Description
Upstream Merge	Merged with the upstream community driver version 42.5.4. See the community JDBC documentation for details.
Bug fix	Fixed an issue in which there was missing information in the MANIFEST.MF file. [Support Ticket #89609]

2.7 EDB JDBC Connector 42.5.1.2 release notes

Released: 14 Feb 2023

The EDB JDBC connector provides connectivity between a Java application and an EDB Postgres Advanced Server database.

New features, enhancements, bug fixes, and other changes in the EDB JDBC Connector 42.5.1.2 include:

Type	Description
Enhancement	Support for EDB Postgres Advanced Server 15.2.0.

2.8 EDB JDBC Connector 42.5.1.1 release notes

Released: 09 Dec 2022

The EDB JDBC connector provides connectivity between a Java application and an EDB Postgres Advanced Server database.

New features, enhancements, bug fixes, and other changes in the EDB JDBC Connector 42.5.1.1 include:

Type	Description
Upstream Merge	Merged with the upstream community driver version 42.5.1. See the community JDBC documentation for details.
Security Fix	CVE-2022-41946 - StreamWrapper spills to disk if setText or setBytea sends very large strings or arrays to the server. createTempFile creates a file that can be read by other users on Unix-like systems (not MacOS).

2.9 EDB JDBC Connector 42.5.0.1 release notes

Released: 01 Sep 2022

The EDB JDBC connector provides connectivity between a Java application and an EDB Postgres Advanced Server database.

New features, enhancements, bug fixes, and other changes in the EDB JDBC Connector 42.5.0.1 include:

Type	Description
Upstream Merge	Merged with the upstream community driver version 42.5.0. See the community JDBC documentation for details.
Security Fix	CVE-2022-31197 - Fixes the SQL generated in PgResultSet.refresh() to escape column identifiers in order to prevent SQL injection. Previously, the column names for both key and data columns were copied as-is into the generated SQL. This allowed for a malicious table with column names that included a statment terminator to be parsed and executed as multiple separate commands. Also, this fix adds a new test class ResultSetRefreshTest to verify this change.
Change	Migrated build to Gradle.
Enhancement	Added new `changeServerName` connection property. If the value for `changeServerName` is set to true, the `getServerName()` call returns a value as `PostgreSQL`. The default value is `false`.
Enhancement	Added new `forceBinaryTransfer` connection property. If the value is set to `true`, forces the transfer of all binary types from the PostgreSQL server to the JDBC driver in their binary form. The default value is `false`.

2.10 EDB JDBC Connector 42.3.3.1 release notes

Released: 20 Apr 2022

The EDB JDBC connector provides connectivity between a Java application and an EDB Postgres Advanced Server database.

New features, enhancements, bug fixes, and other changes in the EDB JDBC Connector 42.3.3.1 include:

Type	Description
Upstream Merge	Merged with the upstream community driver version 42.3.3. See the community JDBC documentation for details.
Security Fix	GHSA-673j-qm5f-xpv8: Removed the loggerFile and loggerLevel configuration properties as part of this fix. While the properties still exist, they can no longer be used to configure the driver logging. Instead use java.util.logging configuration mechanisms such as `logging.properties`.
Change	As part of security fix GHSA-673j-qm5f-xpv, the ability to enable logging using the connection properties is no longer available as of version 42.3.3.

2.11 EDB JDBC Connector 42.3.2.1 release notes

Released: 15 Feb 2022

The EDB JDBC connector provides connectivity between a Java application and an EDB Postgres Advanced Server database.

New features, enhancements, bug fixes, and other changes in the EDB JDBC Connector 42.3.2.1 include:

Type	Description
Upstream merge	Merged with the upstream community driver version 42.3.2. See the community JDBC documentation for details.
New feature	`org.checkerframework.*` was previously packaged in the EDB JDBC jar file; causing conflicts with other applications utilizing `org.checkerfamework.*` with different versions. New feature is packaging the checker framework under a custom namespace in the connector using the shade plugin. [Support Ticket: #74134]
New feature	JMS based API to interact with DBMS_AQ package seamlessly. This API has been made part of edb-jdbc code under com.edb.jms and com.edb.aq packages.
Enhancement	New property oidTimestamp used to change default behavior of driver when using setTimeStamp method for preparedStatement. If property oidTimestamp it is set to true, sets the oid to Oid.TIMESTAMP, otherwise uses default behavior.
Bug fix	Issue: Change in date format nls_date_format='YYYY/MM/DD' in EDB*PLUS gives error. [Support Ticket: #75812]
Bug fix	Rounding differences between EDB and Oracle. [Support Ticket: #72708]
Security fix	CVE-2022-21724 as part of community merge with v42.3.2
Security fix	CVE-2021-36373 - Removed dependency for org.apache.ant
Security fix	CVE-2020-15250 - junit fix for temporary folder.

2.12 EDB JDBC Connector 42.2.24.1 release notes

Released: 05 Nov 2021

The EDB JDBC connector provides connectivity between a Java application and an Advanced Server database.

New features, enhancements, bug fixes, and other changes in the EDB JDBC Connector 42.2.24.1 include:

Type	Description
Upstream merge	Merged with the upstream community driver version 42.2.24. See the community JDBC documentation for details.

2.13 EDB JDBC Connector 42.2.19.1 release notes

Released: 15 Apr 2021

The EDB JDBC connector provides connectivity between a Java application and an Advanced Server database.

New features, enhancements, bug fixes, and other changes in the EDB JDBC Connector 42.2.19.1 include:

Type	Description
Upstream merge	Merged with the upstream community driver version 42.2.19. See the community JDBC documentation for details.
Enhancement	EDB JDBC Connector now supports GSSAPI encrypted connection. See Support for GSSAPI Encrypted Connection.

> **Note**
>
> EDB JDBC Connector v42.2.19.1 does not support Java 1.6 and 1.7. Previous versions of EDB JDBC Connector support Java 1.6 and 1.7 but will not get any future updates, enhancements or bug fixes.

2.14 EDB JDBC Connector 42.2.12.3 release notes

Released: 22 Oct 2020

The EDB JDBC connector provides connectivity between a Java application and an Advanced Server database.

New features, enhancements, bug fixes, and other changes in the EDB JDBC Connector 42.2.12.3 include:

Type	Description
Enhancement	EDB JDBC Connector now supports EDB Postgres Advanced Server 13.

2.15 EDB JDBC Connector 42.2.9.1 release notes

Released: 18 May 2020

The EDB JDBC connector provides connectivity between a Java application and an Advanced Server database.

New features, enhancements, bug fixes, and other changes in the EDB JDBC Connector 42.2.9.1 include:

Type	Description
Enhancement	EDB JDBC Connector is now supported on Red Hat Enterprise Linux and CentOS (x86_64) 8.x.

2.16 EDB JDBC Connector 42.2.8.1 release notes

Released: 21 Oct 2019

The EDB JDBC connector provides connectivity between a Java application and an Advanced Server database.

New features, enhancements, bug fixes, and other changes in the EDB JDBC Connector 42.2.8.1 include:

Type	Description
Upstream merge	Merged with the upstream community driver version 42.2.8. See the community JDBC documentation for details.
Enhancement	EDB JDBC Connector now supports EDB Postgres Advanced Server 12.
Enhancement	EDB JDBC Connector is now supported on the Windows Server 2019 platform.

3 Supported platforms

The JDBC Connector is supported on the same platforms as EDB Postgres Advanced Server. To determine the platform support for the JDBC Connector, you can either refer to the platform support for EDB Postgres Advanced Server on the Platform Compatibility page on the EDB website or refer to Installing EDB JDBC Connector.

Supported database versions

This table lists the latest JDBC Connector versions and their supported corresponding EDB Postgres Advanced Server (EPAS) versions.

JDBC Connector	EPAS18	EPAS17	EPAS16	EPAS 15	EPAS 14	EPAS 13
42.7.3.4	Y	Y	Y	Y	Y	Y
42.7.3.3	N	Y	Y	Y	Y	Y
42.7.3.2	N	Y	Y	Y	Y	Y
42.7.3.1	N	N	Y	Y	Y	Y
42.5.4.2	N	N	Y	Y	Y	Y
42.5.4.1	N	N	N	Y	Y	Y
42.5.1.2	N	N	N	Y	Y	Y
42.5.1.1	N	N	N	N	Y	Y
42.5.0.1	N	N	N	N	Y	Y
42.3.3.1	N	N	N	N	Y	Y
42.3.2.1	N	N	N	N	Y	Y
42.2.24.1	N	N	N	N	Y	Y
42.2.19.1	N	N	N	N	N	Y
42.2.12.3	N	N	N	N	N	Y
42.2.9.1	N	N	N	N	N	N
42.2.8.1	N	N	N	N	N	N

Supported JDK distribution

Java Virtual Machine (JVM): Java SE 8 or higher (LTS version), including Oracle JDK, OpenJDK, and IBM SDK (Java) distributions.

4 EDB JDBC Connector overview

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

JDBC driver types

There are currently four types of JDBC drivers, each with its own implementation, use, and limitations. The EDB JDBC Connector is a Type 4 driver.

Type 1 driver

- This driver type is the JDBC-ODBC bridge.
- It's 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

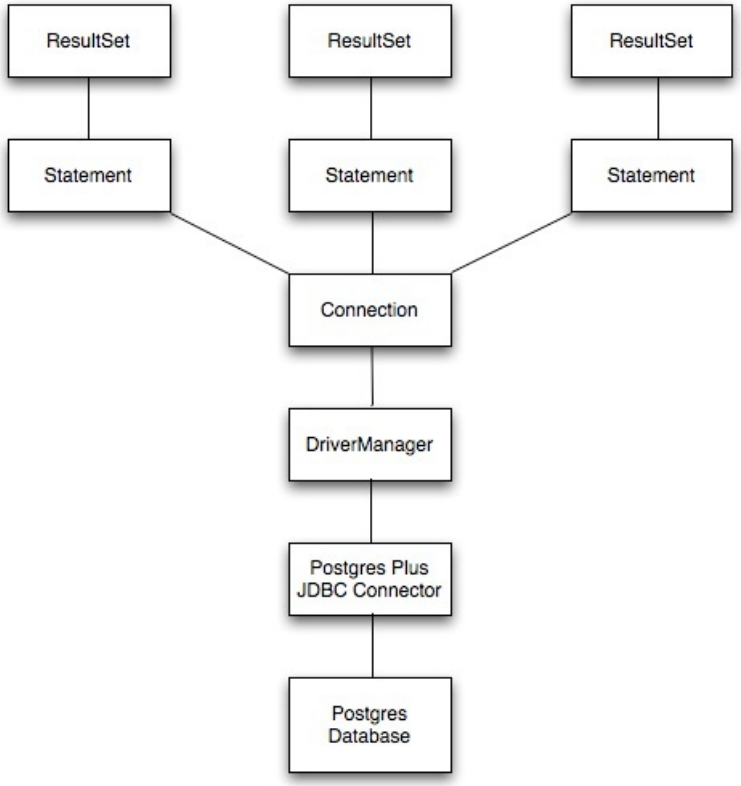
- 100% Java Driver, no native methods.
- Doesn't require preinstallation on client.
- Can be downloaded and configured on-the-fly just like any Java class file.
- Uses a proprietary protocol for talking with a middleware server.
- Middleware server converts from proprietary calls to DBMS specific calls.

Type 4 driver

- 100% Java driver, no native methods.
- Doesn't require preinstallation on client.
- Can be downloaded and configured on-the-fly just like any Java class file.
- Unlike Type 3 driver, talks directly with the DBMS server.
- Converts JDBC calls directly to database specific calls.

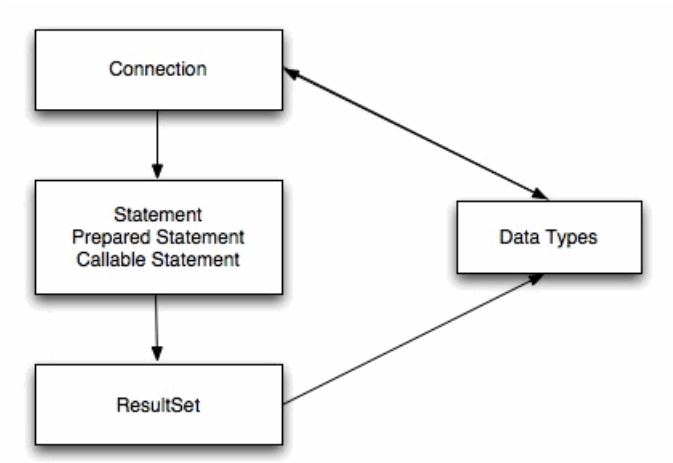
The JDBC interface

The 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.



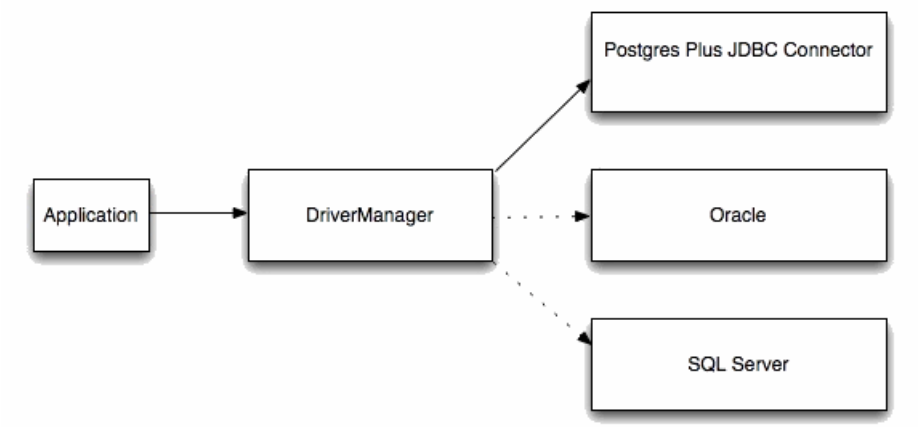
JDBC classes and interfaces

The core API is composed of classes and interfaces. These classes and interfaces work together as shown in the figure:



The JDBC DriverManager

This figure 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 the JDBC driver to use for the target database.



EDB Postgres Advanced Server JDBC Connector compatibility

This is the current version of the driver. Unless you have unusual requirements (running old applications or JVMs), this is the driver you should be using. This driver supports PostgreSQL 10 or higher versions and requires Java 8 or higher versions. It contains support for SSL and the javax.sql package.

> **Note**
>
> Deprecated support for Java 1.6 and 1.7. Previous version of EDB JDBC Connector v42.2.12.3 will continue to support Java 1.6 and 1.7 versions.

5 Installing EDB JDBC Connector

Select a link to access the applicable installation instructions:

Linux x86-64 (amd64)

Red Hat Enterprise Linux (RHEL) and derivatives

- RHEL 9, RHEL 8
- Oracle Linux (OL) 9, Oracle Linux (OL) 8
- Rocky Linux 9, Rocky Linux 8
- AlmaLinux 9, AlmaLinux 8

SUSE Linux Enterprise (SLES)

- SLES 15

Debian and derivatives

- Ubuntu 24.04, Ubuntu 22.04
- Debian 12, Debian 11

Linux IBM Power (ppc64le)

Red Hat Enterprise Linux (RHEL) and derivatives

- RHEL 9, RHEL 8

SUSE Linux Enterprise (SLES)

- SLES 15

Linux AArch64 (ARM64)

Red Hat Enterprise Linux (RHEL) and derivatives

- RHEL 9
- Oracle Linux (OL) 9

Debian and derivatives

- Debian 12

Windows

- Windows Server 2019

5.1 Installing EDB JDBC Connector on Linux IBM Power (ppc64le)

Operating system-specific install instructions are described in the corresponding documentation:

Red Hat Enterprise Linux (RHEL)

- RHEL 9
- RHEL 8

SUSE Linux Enterprise (SLES)

- SLES 15

5.1.1 Installing EDB JDBC Connector on RHEL 9 ppc64le

Prerequisites

Before you begin the installation process:

- Install Postgres on a host that the product can connect to using a connection string. It doesn't need to be on the same host. See:
  - Installing EDB Postgres Advanced Server
  - Installing PostgreSQL
- Ensure that Java is 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 `Installation Instruction` links on the same page.
- Review Supported JDBC distributions.
- Set up the EDB repository.

  Setting up the repository is a one-time task. If you already set up your repository, you don't need to perform this step.

  To determine if your repository exists, enter:

  ```
  dnf repolist | grep enterprisedb
  ```

  If no output is generated, the repository is installed.

  To set up the EDB repository:

  1. Go to EDB repositories.
  2. Select the button that provides access to the EDB repository.
  3. Select the platform and software that you want to download.
  4. Follow the instructions for setting up the EDB repository.
- Install the EPEL repository:

  ```
  sudo dnf -y install https://dl.fedoraproject.org/pub/epel/epel-release-latest-9.noarch.rpm
  ```
- Refresh the cache:

  ```
  sudo dnf makecache
  ```

Install the package

```
sudo dnf -y install edb-jdbc
```

5.1.2 Installing EDB JDBC Connector on RHEL 8 ppc64le

Prerequisites

Before you begin the installation process:

- Install Postgres on a host that the product can connect to using a connection string. It doesn't need to be on the same host. See:
  - Installing EDB Postgres Advanced Server
  - Installing PostgreSQL
- Ensure that Java is 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 `Installation Instruction` links on the same page.
- Review Supported JDBC distributions.
- Set up the EDB repository.

  Setting up the repository is a one-time task. If you already set up your repository, you don't need to perform this step.

  To determine if your repository exists, enter:

  ```
  dnf repolist | grep enterprisedb
  ```

  If no output is generated, the repository is installed.

  To set up the EDB repository:

  1. Go to EDB repositories.
  2. Select the button that provides access to the EDB repository.
  3. Select the platform and software that you want to download.
  4. Follow the instructions for setting up the EDB repository.
- Install the EPEL repository:

  ```
  sudo dnf -y install https://dl.fedoraproject.org/pub/epel/epel-release-latest-8.noarch.rpm
  ```
- Refresh the cache:

  ```
  sudo dnf makecache
  ```

Install the package

```
sudo dnf -y install edb-jdbc
```

5.1.3 Installing EDB JDBC Connector on SLES 15 ppc64le

Prerequisites

Before you begin the installation process:

- Install Postgres on a host that the product can connect to using a connection string. It doesn't need to be on the same host. See:
  - Installing EDB Postgres Advanced Server
  - Installing PostgreSQL
- Ensure that Java is 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 `Installation Instruction` links on the same page.
- Review Supported JDBC distributions.
- Set up the EDB repository.

  Setting up the repository is a one-time task. If you already set up your repository, you don't need to perform this step.

  To determine if your repository exists, enter:

  ```
  zypper lr -E | grep enterprisedb
  ```

  If no output is generated, the repository is installed.

  To set up the EDB repository:

  1. Go to EDB repositories.
  2. Select the button that provides access to the EDB repository.
  3. Select the platform and software that you want to download.
  4. Follow the instructions for setting up the EDB repository.
- Activate the required SUSE module:

  ```
  sudo SUSEConnect -p PackageHub/15.7/ppc64le
  ```

- Refresh the metadata:

  ```
  sudo zypper refresh
  ```

Install the package

```
sudo zypper -n install edb-jdbc
```

5.1.4 Installing EDB JDBC Connector on SLES 12 ppc64le

Prerequisites

Before you begin the installation process:

- Install Postgres on a host that the product can connect to using a connection string. It doesn't need to be on the same host. See:
  - Installing EDB Postgres Advanced Server
  - Installing PostgreSQL
- Ensure that Java is 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 `Installation Instruction` links on the same page.
- Review Supported JDBC distributions.
- Set up the EDB repository.

  Setting up the repository is a one-time task. If you have already set up your repository, you don't need to perform this step.

  To determine if your repository exists, enter:

  ```
  zypper lr -E | grep enterprisedb
  ```

  If no output is generated, the repository isn't installed.

  To set up the EDB repository:

  1. Go to EDB repositories.
  2. Select the button that provides access to the EDB repository.
  3. Select the platform and software that you want to download.
  4. Follow the instructions for setting up the EDB repository.
- Activate the required SUSE module:

  ```
  sudo SUSEConnect -p PackageHub/12.5/ppc64le
  sudo SUSEConnect -p sle-sdk/12.5/ppc64le
  ```

- Refresh the metadata:

  ```
  sudo zypper refresh
  ```

Install the package

```
sudo zypper -n install edb-jdbc
```

5.2 Installing EDB JDBC Connector on Linux x86 (amd64)

Operating system-specific install instructions are described in the corresponding documentation:

Red Hat Enterprise Linux (RHEL) and derivatives

- RHEL 9
- RHEL 8
- Oracle Linux (OL) 9
- Oracle Linux (OL) 8
- Rocky Linux 9
- Rocky Linux 8
- AlmaLinux 9
- AlmaLinux 8

SUSE Linux Enterprise (SLES)

- SLES 15

Debian and derivatives

- Ubuntu 24.04
- Ubuntu 22.04
- Debian 12
- Debian 11

5.2.1 Installing EDB JDBC Connector on RHEL 9 or OL 9 x86_64

Prerequisites

Before you begin the installation process:

- Install Postgres on a host that the product can connect to using a connection string. It doesn't need to be on the same host. See:
  - Installing EDB Postgres Advanced Server
  - Installing PostgreSQL
- Ensure that Java is 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 `Installation Instruction` links on the same page.
- Review Supported JDBC distributions.
- Set up the EDB repository.

  Setting up the repository is a one-time task. If you already set up your repository, you don't need to perform this step.

  To determine if your repository exists, enter:

  ```
  dnf repolist | grep enterprisedb
  ```

  If no output is generated, the repository is installed.

  To set up the EDB repository:

  1. Go to EDB repositories.
  2. Select the button that provides access to the EDB repository.
  3. Select the platform and software that you want to download.
  4. Follow the instructions for setting up the EDB repository.
- Install the EPEL repository:

  ```
  sudo dnf -y install https://dl.fedoraproject.org/pub/epel/epel-release-latest-9.noarch.rpm
  ```

Install the package

```
sudo dnf -y install edb-jdbc
```

5.2.2 Installing EDB JDBC Connector on RHEL 8 or OL 8 x86_64

Prerequisites

Before you begin the installation process:

- Install Postgres on a host that the product can connect to using a connection string. It doesn't need to be on the same host. See:
  - Installing EDB Postgres Advanced Server
  - Installing PostgreSQL
- Ensure that Java is 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 `Installation Instruction` links on the same page.
- Review Supported JDBC distributions.
- Set up the EDB repository.

  Setting up the repository is a one-time task. If you already set up your repository, you don't need to perform this step.

  To determine if your repository exists, enter:

  ```
  dnf repolist | grep enterprisedb
  ```

  If no output is generated, the repository is installed.

  To set up the EDB repository:

  1. Go to EDB repositories.
  2. Select the button that provides access to the EDB repository.
  3. Select the platform and software that you want to download.
  4. Follow the instructions for setting up the EDB repository.
- Install the EPEL repository:

  ```
  sudo dnf -y install https://dl.fedoraproject.org/pub/epel/epel-release-latest-8.noarch.rpm
  ```

Install the package

```
sudo dnf -y install edb-jdbc
```

5.2.3 Installing EDB JDBC Connector on AlmaLinux 9 or Rocky Linux 9 x86_64

Prerequisites

Before you begin the installation process:

- Install Postgres on a host that the product can connect to using a connection string. It doesn't need to be on the same host. See:
  - Installing EDB Postgres Advanced Server
  - Installing PostgreSQL
- Ensure that Java is 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 `Installation Instruction` links on the same page.
- Review Supported JDBC distributions.
- Set up the EDB repository.

  Setting up the repository is a one-time task. If you already set up your repository, you don't need to perform this step.

  To determine if your repository exists, enter:

  ```
  dnf repolist | grep enterprisedb
  ```

  If no output is generated, the repository is installed.

  To set up the EDB repository:

  1. Go to EDB repositories.
  2. Select the button that provides access to the EDB repository.
  3. Select the platform and software that you want to download.
  4. Follow the instructions for setting up the EDB repository.
- Install the EPEL repository:

  ```
  sudo dnf -y install epel-release
  ```

- Enable additional repositories to resolve dependencies:

  ```
  sudo dnf config-manager --set-enabled crb
  ```

Install the package

```
sudo dnf -y install edb-jdbc
```

5.2.4 Installing EDB JDBC Connector on AlmaLinux 8 or Rocky Linux 8 x86_64

Prerequisites

Before you begin the installation process:

- Install Postgres on a host that the product can connect to using a connection string. It doesn't need to be on the same host. See:
  - Installing EDB Postgres Advanced Server
  - Installing PostgreSQL
- Ensure that Java is 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 `Installation Instruction` links on the same page.
- Review Supported JDBC distributions.
- Set up the EDB repository.

  Setting up the repository is a one-time task. If you already set up your repository, you don't need to perform this step.

  To determine if your repository exists, enter:

  ```
  dnf repolist | grep enterprisedb
  ```

  If no output is generated, the repository is installed.

  To set up the EDB repository:

  1. Go to EDB repositories.
  2. Select the button that provides access to the EDB repository.
  3. Select the platform and software that you want to download.
  4. Follow the instructions for setting up the EDB repository.
- Install the EPEL repository:

  ```
  sudo dnf -y install epel-release
  ```
- Enable additional repositories to resolve dependencies:

  ```
  sudo dnf config-manager --set-enabled powertools
  ```

Install the package

```
sudo dnf -y install edb-jdbc
```

5.2.5 Installing EDB JDBC Connector on SLES 15 x86_64

Prerequisites

Before you begin the installation process:

- Install Postgres on a host that the product can connect to using a connection string. It doesn't need to be on the same host. See:
  - Installing EDB Postgres Advanced Server
  - Installing PostgreSQL
- Ensure that Java is 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 `Installation Instruction` links on the same page.
- Review Supported JDBC distributions.
- Set up the EDB repository.

  Setting up the repository is a one-time task. If you already set up your repository, you don't need to perform this step.

  To determine if your repository exists, enter:

  ```
  zypper lr -E | grep enterprisedb
  ```

  If no output is generated, the repository is installed.

  To set up the EDB repository:

  1. Go to EDB repositories.
  2. Select the button that provides access to the EDB repository.
  3. Select the platform and software that you want to download.
  4. Follow the instructions for setting up the EDB repository.
- Activate the required SUSE module:

  ```
  sudo SUSEConnect -p PackageHub/15.7/x86_64
  ```

- Refresh the metadata:

  ```
  sudo zypper refresh
  ```

Install the package

```
sudo zypper -n install edb-jdbc
```

5.2.6 Installing EDB JDBC Connector on Ubuntu 24.04 x86_64

Prerequisites

Before you begin the installation process:

- Install Postgres on a host that the product can connect to using a connection string. It doesn't need to be on the same host. See:
  - Installing EDB Postgres Advanced Server
  - Installing PostgreSQL
- Ensure that Java is 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 `Installation Instruction` links on the same page.
- Review Supported JDBC distributions.
- Set up the EDB repository.

  Setting up the repository is a one-time task. If you already set up your repository, you don't need to perform this step.

  To determine if your repository exists, enter:

  ```
  apt-cache search enterprisedb
  ```

  If no output is generated, the repository is installed.

  To set up the EDB repository:

  1. Go to EDB repositories.
  2. Select the button that provides access to the EDB repository.
  3. Select the platform and software that you want to download.
  4. Follow the instructions for setting up the EDB repository.

Install the package

```
sudo apt-get -y install edb-jdbc
```

5.2.7 Installing EDB JDBC Connector on Ubuntu 22.04 x86_64

Prerequisites

Before you begin the installation process:

- Install Postgres on a host that the product can connect to using a connection string. It doesn't need to be on the same host. See:
  - Installing EDB Postgres Advanced Server
  - Installing PostgreSQL
- Ensure that Java is 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 `Installation Instruction` links on the same page.
- Review Supported JDBC distributions.
- Set up the EDB repository.

  Setting up the repository is a one-time task. If you already set up your repository, you don't need to perform this step.

  To determine if your repository exists, enter:

  ```
  apt-cache search enterprisedb
  ```

  If no output is generated, the repository is installed.

  To set up the EDB repository:

  1. Go to EDB repositories.
  2. Select the button that provides access to the EDB repository.
  3. Select the platform and software that you want to download.
  4. Follow the instructions for setting up the EDB repository.

Install the package

```
sudo apt-get -y install edb-jdbc
```

5.2.8 Installing EDB JDBC Connector on Debian 12 x86_64

Prerequisites

Before you begin the installation process:

- Install Postgres on a host that the product can connect to using a connection string. It doesn't need to be on the same host. See:
  - Installing EDB Postgres Advanced Server
  - Installing PostgreSQL
- Ensure that Java is 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 `Installation Instruction` links on the same page.
- Review Supported JDBC distributions.
- Set up the EDB repository.

  Setting up the repository is a one-time task. If you already set up your repository, you don't need to perform this step.

  To determine if your repository exists, enter:

  ```
  apt-cache search enterprisedb
  ```

  If no output is generated, the repository is installed.

  To set up the EDB repository:

  1. Go to EDB repositories.
  2. Select the button that provides access to the EDB repository.
  3. Select the platform and software that you want to download.
  4. Follow the instructions for setting up the EDB repository.

Install the package

```
sudo apt-get -y install edb-jdbc
```

5.2.9 Installing EDB JDBC Connector on Debian 11 x86_64

Prerequisites

Before you begin the installation process:

- Install Postgres on a host that the product can connect to using a connection string. It doesn't need to be on the same host. See:
  - Installing EDB Postgres Advanced Server
  - Installing PostgreSQL
- Ensure that Java is 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 `Installation Instruction` links on the same page.
- Review Supported JDBC distributions.
- Set up the EDB repository.

  Setting up the repository is a one-time task. If you already set up your repository, you don't need to perform this step.

  To determine if your repository exists, enter:

  ```
  apt-cache search enterprisedb
  ```

  If no output is generated, the repository is installed.

  To set up the EDB repository:

  1. Go to EDB repositories.
  2. Select the button that provides access to the EDB repository.
  3. Select the platform and software that you want to download.
  4. Follow the instructions for setting up the EDB repository.

Install the package

```
sudo apt-get -y install edb-jdbc
```

5.2.10 Installing EDB JDBC Connector on SLES 12 x86_64

Prerequisites

Before you begin the installation process:

- Install Postgres on a host that the product can connect to using a connection string. It doesn't need to be on the same host. See:
  - Installing EDB Postgres Advanced Server
  - Installing PostgreSQL
- Ensure that Java is 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 `Installation Instruction` links on the same page.
- Review Supported JDBC distributions.
- Set up the EDB repository.

  Setting up the repository is a one-time task. If you have already set up your repository, you don't need to perform this step.

  To determine if your repository exists, enter:

  ```
  zypper lr -E | grep enterprisedb
  ```

  If no output is generated, the repository isn't installed.

  To set up the EDB repository:

  1. Go to EDB repositories.
  2. Select the button that provides access to the EDB repository.
  3. Select the platform and software that you want to download.
  4. Follow the instructions for setting up the EDB repository.
- Activate the required SUSE module:

  ```
  sudo SUSEConnect -p PackageHub/12.5/x86_64
  sudo SUSEConnect -p sle-sdk/12.5/x86_64
  ```

- Refresh the metadata:

  ```
  sudo zypper refresh
  ```

Install the package

```
sudo zypper -n install edb-jdbc
```

5.3 Installing EDB JDBC Connector on Linux AArch64 (ARM64)

Operating system-specific install instructions are described in the corresponding documentation:

Red Hat Enterprise Linux (RHEL) and derivatives

- RHEL 9
- Oracle Linux (OL) 9

Debian and derivatives

- Debian 12

5.3.1 Installing EDB JDBC Connector on RHEL 9 or OL 9 arm64

Prerequisites

Before you begin the installation process:

- Install Postgres on a host that the product can connect to using a connection string. It doesn't need to be on the same host. See:
  - Installing EDB Postgres Advanced Server
  - Installing PostgreSQL
- Ensure that Java is 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 `Installation Instruction` links on the same page.
- Review Supported JDBC distributions.
- Set up the EDB repository.

  Setting up the repository is a one-time task. If you already set up your repository, you don't need to perform this step.

  To determine if your repository exists, enter:

  ```
  dnf repolist | grep enterprisedb
  ```

  If no output is generated, the repository is installed.

  To set up the EDB repository:

  1. Go to EDB repositories.
  2. Select the button that provides access to the EDB repository.
  3. Select the platform and software that you want to download.
  4. Follow the instructions for setting up the EDB repository.
- Install the EPEL repository:

  ```
  sudo dnf -y install https://dl.fedoraproject.org/pub/epel/epel-release-latest-9.noarch.rpm
  ```

Install the package

```
sudo dnf -y install edb-jdbc
```

5.3.2 Installing EDB JDBC Connector on Debian 12 arm64

Prerequisites

Before you begin the installation process:

- Install Postgres on a host that the product can connect to using a connection string. It doesn't need to be on the same host. See:
  - Installing EDB Postgres Advanced Server
  - Installing PostgreSQL
- Ensure that Java is 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 `Installation Instruction` links on the same page.
- Review Supported JDBC distributions.
- Set up the EDB repository.

  Setting up the repository is a one-time task. If you already set up your repository, you don't need to perform this step.

  To determine if your repository exists, enter:

  ```
  apt-cache search enterprisedb
  ```

  If no output is generated, the repository is installed.

  To set up the EDB repository:

  1. Go to EDB repositories.
  2. Select the button that provides access to the EDB repository.
  3. Select the platform and software that you want to download.
  4. Follow the instructions for setting up the EDB repository.

Install the package

```
sudo apt-get -y install edb-jdbc
```

5.4 Installing EDB JDBC Connector on Windows

EDB provides a graphical installer for Windows. You can access it two ways:

- Download the graphical installer from the Downloads page, and invoke the installer directly. See Installing directly.
- Use Stack Builder (with PostgreSQL) or StackBuilder Plus (with EDB Postgres Advanced Server) to download the EDB installer package and invoke the graphical installer. See Using Stack Builder or StackBuilder Plus.

Installing directly

After downloading the graphical installer, to start the installation wizard, assume sufficient privileges (superuser or administrator) and double-click the installer icon. If prompted, provide a password.

In some versions of Windows, to invoke the installer with administrator privileges, you need to right-click the installer icon and select **Run as Administrator** from the context menu.

Proceed to Using the graphical installer.

Using Stack Builder or StackBuilder Plus

If you're using PostgreSQL, you can invoke the graphical installer with Stack Builder. SeeUsing Stack Builder.

If you're using EDB Postgres Advanced Server, you can invoke the graphical installer with StackBuilder Plus. SeeUsing StackBuilder Plus.

1. In Stack Builder or StackBuilder Plus, follow the prompts until you get to the module selection page.

   On the Welcome page, from the list of available servers, select the target server installation. If your network requires you to use a proxy server to access the internet, select **Proxy servers** and specify a server. Select **Next**.

2. Expand the **Database Drivers** node and do one of the following:

   - In Stack Builder, select **pgJDBC**.
   - In StackBuilder Plus, select **EnterpriseDB JDBC Connector**.

3. Proceed to Using the graphical installer.

Using the graphical installer

1. Select the installation language and select **OK**.
2. On the Setup JDBC page, select **Next**.
3. Browse to a directory where you want to install JDBC, or leave the directory set to the default location. Select **Next**.

4. On the Ready to Install page, select **Next**.

   An information box shows the installation progress of the selected components.

5. When the installation is complete, select **Finish**.

5.5 Installing EDB JDBC Connector using Maven

EDB supports installing EDB JDBC Connector using the Maven dependency manager. EDB-JDBC is published in theMaven Central Repository with the following groupId and artifactId:

- groupId: `com.enterprisedb`
- artifactId: `edb-jdbc`

Add the following dependency for EDB-JDBC in your `pom.xml` file to install and configure the EDB JDBC Connector. Ensure you provide the correct version to install:

```
<dependency>
    <groupId>com.enterprisedb</groupId>
    <artifactId>edb-jdbc</artifactId>
    <version>42.5.4.2</version>
</dependency>
```

5.6 Configuring EDB JDBC Connector for Java

edb-jdbc18.jar supports JDBC version 4.2.

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

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-jdbc18.jar
```

5.7 Upgrading a Linux installation

If you have an existing JDBC Connector installation on a Linux platform, you can upgrade your repository configuration file, which enables access to the current EDB repository. Then you can upgrade to a more recent version of JDBC Connector.

To update the `edb.repo` file:

```
# Update your repository configuration file
sudo <package-manager> upgrade edb-repo

# Upgrade the installed product
sudo <package-manager> upgrade edb-repo
```

Where `<package-manager>` is the package manager used with your operating system.

Package manager	Operating system
dnf	RHEL 8/9 and derivatives
zypper	SLES
apt-get	Debian and Ubuntu

6 Using the EDB JDBC Connector with Java applications

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

```
import java.sql.*;
public class ListEmployees
{
  public static void main(String[] args)
  {
    try
    {
      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(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 shows the fundamental steps required to interact with an EDB Postgres Advanced Server database from a Java application:

- Load the JDBC driver.
- Build connection properties.
- Connect to the database server.
- Execute a SQL statement.
- Process the result set.
- Clean up.
- Handle any errors that occur.

6.1 Loading EDB JDBC Connector

The EDB Postgres Advanced Server JDBC driver is written in Java and is distributed as a compiled Java Archive (JAR) file. Include the driver's JAR file in your classpath so that the Java runtime can register the driver as it starts up. The registered driver will be used when an application requests a connection with a URL beginning with the "jdbc:edb:" schema.

6.2 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's 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. The decision is based on a URL 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 EDB Postgres 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.

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

6.2.1 Additional connection properties

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

```
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 doesn't validate the SSL certificate. To enforce certificate validation, you must use a `Custom SSLSocketFactory`. For more details about writing a `Custom SSLSocketFactory`, see the 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.

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.
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.
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`, `primary`, `secondary`, `preferSecondary`, and `preferSyncSecondary`. The primary/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 primary. The EDB Postgres Advanced Server JDBC Connector supports `preferSyncSecondary`, which permits connection to only synchronous secondaries or the primary if there are no active synchronous secondaries.
skipQuotesOnReturning	Boolean	When set to `true`, column names from the `RETURNING` clause aren't quoted. This eliminates a case-sensitive comparison of the column name. When set to `false` (the default setting), column names are quoted.
changeServerName	Boolean	The `getServerName()` call in `PgConnection.java` returns `EnterpriseDB`. If `changeServerName` is set to `true`, it returns the value as `PostgreSQL`. The default value is `false`.

Name	Type	Description
forceBinaryTransfer	Boolean	If the value is set to `true`, forces the transfer of all binary types from the PostgreSQL server to the JDBC driver in their binary form. The default value is `false`.

6.2.2 Preferring synchronous secondary database servers

The EDB Postgres Advanced Server JDBC Connector supports the `preferSyncSecondary` option for the `targetServerType` connection property.

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

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

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

Parameters

`primary:port`

The IP address or a name assigned to the primary database server followed by its port number. If `primary` is a name, you must specify it with its IP address in the `/etc/hosts` file on the host running the Java program.

> **Note**
>
> You can specify the primary database server in any location in the list. It doesn't 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, you must specify it with its IP address in the `/etc/hosts` file on the host running the Java program.

`database`

The name of the database to which to make the connection.

The following is an example of the connection URL:

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

The following characteristics apply to the `preferSyncSecondary` option:

- You cam specify the primary database server in any location in the connection list.
- Connection for accessing the database for use 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. 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 can't be made to a synchronous secondary, then connection to the primary database server is used. If the primary database server isn't active, then the connection attempt fails.

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

Configuring primary and secondary database servers overview

The process for configuring a primary and secondary database servers is 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 primary, 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* in Section 25.3 *Continuous Archiving and Point-in-Time Recovery (PITR)* in The PostgreSQL Core Documentation.

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

Example: Primary and secondary database servers

In the example that follows, the:

- Primary 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 primary).

In the primary 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 primary.

```
# 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 primary database server is 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 was edited in the following example by adding the `application_name=secondary1` setting as part of the `primary_conninfo` string and removing 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 primary database server's `postgresql.conf` file.

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

```
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. The `application_name=secondary2` setting is 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 primary 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's 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 the `standby.signal` file signals the cluster to run in standby mode. Even if you try to create a `recovery.conf` file manually and keep it under the `data` directory, the server fails to start and reports 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 primary database server as compared to the secondary database servers.

Parameter	Primary	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 (10 or later)	replica	minimal	Information written to WAL segment
max_wal_senders	*n* (positive integer)	0	Maximum concurrent connections from standby servers

Parameter	Primary	Secondary	Description
wal_keep_segments	*n* (positive integer)	0	Minimum number of past log segments to keep for standby servers
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 primary database server. 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;
```

output

```
   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://primary: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 primary
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, and then on the primary if both `secondary1` and `secondary2` aren't active as shown by the following program. The program 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
    {
      String url
=
 "jdbc:edb://primary: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 `primary` 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
```

6.3 Executing SQL statements through statement objects

After loading the EDB Postgres Advanced Server JDBC Connector driver and connecting to the server, the code in the sample application builds a JDBC `Statement` object, executes a 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 precompiled 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 a 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 encountered an error while executing the SQL statement provided, it returns an `SQLException` and doesn't return a `ResultSet`.

Using named notation with a CallableStatement object

The JDBC Connector (EDB Postgres Advanced Server version 10 and later) supports the use of named parameters when instantiating a `CallableStatement` object. This syntax is an extension of JDBC supported syntax and doesn't 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 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, 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 in a statement (`?`) represents an ordinal value. When using ordinal parameters, pass the actual parameter values to the statement in the order that the formal arguments are specified in 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 must correspond to a `COLUMN_NAME` returned by a call to the `DatabaseMetaData.getProcedureColumns` method. 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 show using the `CallableStatement` method to provide parameters to a procedure with the following signature:

```
CREATE OR REPLACE PROCEDURE hire_emp (ename
VARCHAR2
empno NUMBER,
job VARCHAR2,
sal NUMBER,
hiredate DATE DEFAULT
now(),
mgr NUMBER DEFAULT 7100,
deptno NUMBER
)
IS
    BEGIN
        INSERT INTO emp VALUES (empno, ename, job, mgr, hiredate, sal,
deptno);
    END;
```

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, Date.valueOf("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 reorder 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);
```

6.4 Retrieving results from a ResultSet object

A `ResultSet` object is the primary storage mechanism for the data returned by a 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 a 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 points 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 fails. 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's another row in the `ResultSet` or `FALSE` if you 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:

- 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 must access each row in the `ResultSet` only once. It's safest to stick to this rule, although currently the EDB Postgres Advanced Server JDBC driver allows you to access a field as many times as you want.
- When you finish using a `ResultSet`, call the `close()` method to free the resources held by that object.

6.5 Freeing resources

Every JDBC object consumes resources. A `ResultSet` object, for example, might contain a copy of every row returned by a query. A `Statement` object might contain the text of the last command executed. 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 returns an error.

6.6 Handling errors

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

When a JDBC object returns 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 this example, the code displays the value of these components if an error occurs:

```
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 doesn't 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
```

7 Using advanced queueing

> **New feature**
>
> Advanced queueing is available in JDBC 42.3.2.1 and later.

EDB Postgres Advanced Server advanced queueing provides message queueing and message processing for the EDB Postgres Advanced Server database. User-defined messages are stored in a queue, and a collection of queues is stored in a queue table. You must first create a queue table before creating a queue that depends on it.

On the server side, procedures in the `DBMS_AQADM` package create and manage message queues and queue tables. Use the `DBMS_AQ` package to add or remove messages from a queue or register or unregister an SPL callback procedure. For more information about `DBMS_AQ` and `DBMS_AQADM`, see DBMS_AQ in the EDB Postgres Advanced Server documentation.

On the client side, the application uses the EDB-JDBC driver's JMS API to enqueue and dequeue message.

Enqueueing or dequeueing a message

For more information about using EDB Postgres Advanced Server's advanced queueing functionality, see Built-in packages in the EDB Postgres Advanced Server documentation.

7.1 Server-side setup

To use advanced queueing functionality on your JMS-based Java application, in EDB-PSQL or EDB-JDBC:

1. Create a user-defined message type, which can be one of the standard JMS message types. However, EDB-JDBC also supports any user-defined message types. See Message types for details.
2. Create a queue table specifying the payload type. This type is typically the one created in step 1.
3. Create a queue using the queue table created in the previous step.
4. Start the queue on the database server.
5. You can use either EDB-PSQL or EDB-JDBC JMS API in your Java application.

Using EDB-PSQL

Invoke EDB-PSQL and connect to the EDB Postgres Advanced Server host database. Use the SPL commands in EDB-PSQL to:

- Create a user-defined type
- Create the queue table
- Create the queue
- Start the queue

Create a user-defined type

To specify a RAW data type, create a user-defined type.

This example shows how to create a user-defined type named `mytype`:

```
CREATE OR REPLACE TYPE mytype AS (code INT, project TEXT, manager
VARCHAR(10));
```

Create the queue table

A queue table can hold multiple queues with the same payload type.

This example shows how to create a queue table named `MSG_QUEUE_TABLE`:

```
EXEC
DBMS_AQADM.CREATE_QUEUE_TABLE
      (queue_table => 'MSG_QUEUE_TABLE',
       queue_payload_type => 'mytype',
       comment => 'Message queue
table');
END;
```

Create the queue

This example shows how to create a queue named `MSG_QUEUE` in the table `MSG_QUEUE_TABLE`:

```
EXEC DBMS_AQADM.CREATE_QUEUE
      (queue_name => 'MSG_QUEUE',
       queue_table => 'MSG_QUEUE_TABLE',
       comment => 'This queue contains pending
messages.');
```

Start the queue

Once the queue is created, start the queue.

This example shows how to start a queue in the database:

```
EXEC DBMS_AQADM.START_QUEUE(queue_name => 'MSG_QUEUE');
commit;
```

Using EDB-JDBC JMS API

> **Tip**
>
> The following sequence of steps is required only if you want to create message types, queue tables, and queues programmatically. If you create the message types, queue table, and queue using EDB-PSQL, then you can use the standard JMS API.

The following JMS API calls perform the same steps performed using EDB-PSQL to:

- Connect to the EDB Postgres Advanced Server database
- Create the user-defined type
- Create the queue table and queue
- Start the queue

```
edbJmsFact = new EDBJmsConnectionFactory("localhost", 5444, "edb", "edb", "edb");

conn = (EDBJmsQueueConnection) edbJmsFact.createQueueConnection();

session = (EDBJmsQueueSession) conn.createQueueSession(true, Session.CLIENT_ACKNOWLEDGE);

String sql = "CREATE OR REPLACE TYPE mytype AS (code int, project
TEXT);";
UDTType udtType = new UDTType(conn.getConn(), sql,
"mytype");
Operation operation = new UDTTypeOperation(udtType);
operation.execute();

queueTable = session.createQueueTable(conn.getConn(), "MSG_QUEUE_TABLE", "mytype", "Message queue
table");

Queue queue1 = new Queue(conn.getConn(), "MSG_QUEUE", "MSG_QUEUE_TABLE", "Message
Queue");
operation = new QueueOperation(queue1);
operation.execute();

queue = (EDBJmsQueue) session.createQueue("MSG_QUEUE");
queue.setEdbQueueTbl(queueTable);

queue.start();
'''
```

```
<span
data-original-path='product_docs/docs/jdbc_connector/42.7.3.4/05a_using_advanced_queueing/jms_application
.mdx:5'></span>

# 7.2          Setting up the JMS
application

After creating the queue table and queue for the message types and starting the queue, you can set up
your JMS application:

1. Create a [connection factory]
(#connection-factory).
1. Create a [connection](#connection) using the connection
factory.
1. Create a [session](#session) using the
connection.
1. Get the queue from the
session.
1. Create a [message producer](#message-producer) using the session and queue to send
messages.
1. Create a [message consumer](#message-consumer) using the session and queue to receive
messages.



### Connection factory

Use the connection factory to create connections. `EDBJmsConnectionFactory` is an implementation of
`ConnectionFactory` and `QueueConnectionFactory`, which you use to create `Connection` and
`QueueConnection`. You can create a connection factory using one of the constructors of the
`EDBJmsConnectionFactory` class. You can use all three constructors to create either a
`ConnectionFactory` or `QueueConnectionFactory`.

```java
//Constructor with connection related
properties.
public EDBJmsConnectionFactory(String host, int port, String database,
     String username, String
password);
//Constructor with connection string, user name and
password.
public EDBJmsConnectionFactory(String connectionString,
     String username, String
password);
//Constructor with SQL
Connection.
public EDBJmsConnectionFactory(java.sql.Connection connection);
```

This example shows how to create a `ConnectionFactory` using an existing `java.sql.Connection`:

```
javax.jms.ConnectionFactory connFactory = new
EDBJmsConnectionFactory(connection);
```

This example shows how to create a `QueueConnectionFactory` using a connection string, username, and password:

```
javax.jms.QueueConnectionFactory connFactory = new
EDBJmsConnectionFactory
   ("jdbc:edb//localhost:5444/edb", "enterprisedb", "edb");
```
```

Connection

A connection is a client's active connection that can be created from the `ConnectionFactory` and used to create sessions. `EDBJmsConnection` is an implementation of `Connection`, and `EDBJmsQueueConnection` is an implementation of `QueueConnection` and extends `EDBJmsConnection`. You can create a `Connection` using `ConnectionFactory` and a `QueueConnection` from `QueueConnectionFactory`.

This example shows how to create a `Connection` and a `QueueConnection`:

```
//Connection from ConnectionFactory. Assuming connFactory is
ConnectionFactory.
javax.jms.Connection connection =
connFactory.createConnection();

////Connection from QueueConnectionFactory. Assuming connFactory is
QueueConnectionFactory.
javax.jms.QueueConnection queueConnection =
connFactory.createQueueConnection();
```

You must start a connection for the consumer to receive messages. However, a producer can send messages without starting the connection.

This example shows how to start a connection:

```
queueConnection.start();
```

You can stop a connection at any time to stop receiving messages, and you can restart it when needed. However, you can't restart a closed connection.

This example shows how to stop and close the connection:

```
queueConnection.stop();
queueConnection.close();
```

Session

A session in EDBJms is used for creating producers and consumers and for sending and receiving messages. `EDBJmsSession` implements the basic `Session` functionality, and `EDBJmsQueueSession` extends `EDBJmsSession` and implements `QueueSession`. A `Session` can be created from a `Connection`.

This example shows how to create a `Session` and a `QueueSession`:

```
// Session
javax.jms.Session session = connection.createSession(false, javax.jms.Session.AUTO_ACKNOWLEDGE);
// QueueSession
javax.jms.QueueSession session = queueConnection.createQueueSession(false,
javax.jms.Session.AUTO_ACKNOWLEDGE);
```

You can also use a `Session` or `QueueSession` to create queues.

> **Important**
>
> In this context, "creating a queue" doesn't refer to physically creating the queue. As discussed earlier, you need to create and start the queue as part of the server-side setup. In this context, creating a queue means getting the queue, related queue table, and payload type that were already created.

This example shows how to create a queue:

```
javax.jms.Queue queue = session.createQueue("MSG_QUEUE");
```

Message producer

A message producer is responsible for creating and sending messages. You create it using a session and queue. `EDBJmsMessageProducer` is an implementation of `MessageProducer`, but in most cases you use the standard `MessageProducer`.

This example shows how to create a message producer, create a message, and send it. To create messages of different types, see Message types.

```
javax.jms.MessageProducer messageProducer = session.createProducer(queue);

javax.jms.Message msg =
session.createMessage();
msg.setStringProperty("myprop1", "test value 1");

messageProducer.send(msg);
```

Message consumer

A message consumer receives messages. You create it using a session and a queue. `EDBJmsMessageConsumer` is an implementation of `MessageConsumer`, but you'll most often use the standard `MessageConsumer`.

This example shows how to create a message consumer and receive a message:

```
javax.jms.MessageConsumer messageConsumer = session.createConsumer(queue);

javax.jms.Message message = messageConsumer.receive();
```

7.3 Message acknowledgement

Acknowledgement messages are controlled by the two arguments to the `createSession()` and `createQueueSession()` methods:

```
EDBJmsConnection.createSession(boolean transacted, int acknowledgeMode)

EDBJmsQueueConnection.createQueueSession(boolean transacted, int acknowledgeMode)
```

If the first argument is true, it indicates that the session mode is transacted, and the second argument is ignored. However, if the first argument is false, then the second argument comes into play, and the client can specify different acknowledgment modes.

These acknowledgment modes include:

- Session.AUTO_ACKNOWLEDGE
- Session.CLIENT_ACKNOWLEDGE
- Session.DUPS_OK_ACKNOWLEDGE

Transacted session

In transacted sessions, messages are both sent and received during a transaction. These messages are acknowledged by making an explicit call to `commit()`. If `rollback()` is called, all received messages are marked as not acknowledged.

A transacted session always has an active transaction. When a client calls the `commit()` or `rollback()` method, the current transaction is either committed or rolled back, and a new transaction is started.

This example shows how the transacted session works:

```
    MessageProducer messageProducer = (MessageProducer) session.createProducer(queue);

    //Send a message in transacted session and commit
it.

    //Send message
    TextMessage msg1 =
session.createTextMessage();
    String messageText1 = "Hello 1";
    msg1.setText(messageText1);
    messageProducer.send(msg1);

    //Commit the
transaction.
    session.commit();

    //Now we have one message in the
queue.

    //Next, we want to send and receive in the same
transaction.

    MessageConsumer messageConsumer = (MessageConsumer) session.createConsumer(queue);

    //Send a Message in
transaction.
    TextMessage msg2 =
session.createTextMessage();
    String messageText2 = "Hello 2";
```

```
    msg2.setText(messageText2);
    messageProducer.send(msg2);

    //Receive message in the same transaction. There should be 1 message
available.
    Message message1 =
messageConsumer.receive();
    TextMessage txtMsg1 = (TextMessage)
message1;

    //Send another Message in transaction.
    TextMessage msg3 =
session.createTextMessage();
    String messageText3 = "Hello 3";
    msg3.setText(messageText3);
    messageProducer.send(msg3);

    //Commit the
transaction.
    //This should remove the one message we sent initially and received above and send 2
messages.
    session.commit();

    //2 messages are in the queue so we can receive these 2
messages.

    //Receive
1
    Message message2 =
messageConsumer.receive();
    TextMessage txtMsg2 = (TextMessage)
message2;

    //Receive
2
    Message message3 =
messageConsumer.receive();
    TextMessage txtMsg3 = (TextMessage)
message3;

    //Commit the transaction. This will consume the two
messages.
    session.commit();

    //Receive should fail now as there should be no messages
available.
    Message message4 =
messageConsumer.receive();
    //message4 will be null here.
```

AUTO_ACKNOWLEDGE mode

If the first argument to `createSession()` or `createQueueSession()` is false and the second argument is `Session.AUTO_ACKNOWLEDGE`, the messages are acknowledged automatically.

DUPS_OK_ACKNOWLEDGE mode

This mode instructs the session to lazily acknowledge the message and that it's okay if some messages are redelivered. However, in EDB JMS, this option is implemented the same way as `Session.AUTO_ACKNOWLEDGE`, where messages are acknowledged automatically.

CLIENT_ACKNOWLEDGE mode

If the first argument to `createSession()` or `createQueueSession()` is false and the second argument is `Session.CLIENT_ACKNOWLEDGE`, the messages are acknowledged when the client acknowledges the message by calling the `acknowledge()` method on a message. Acknowledging happens at the session level, and acknowledging one message causes all the received messages to be acknowledged.

For example, if you send five messages and then receive the five messages, acknowledging the fifth message causes all five messages to be acknowledged.

```
    MessageProducer messageProducer = (MessageProducer) session.createProducer(queue);

    //Send 5
messages
    for(int i=1; i<=5; i++)
{
        TextMessage msg =
session.createTextMessage();
        String messageText = "Hello " +
i;

msg.setText(messageText);
        messageProducer.send(msg);
    }

    MessageConsumer messageConsumer = (MessageConsumer) session.createConsumer(queue);

    //Receive
4
    for(int i=1; i<=4; i++)
{
        Message message = messageConsumer.receive();
        TextMessage txtMsg = (TextMessage)
message;
    }

    //Receive the 5th
message
    Message message5 =
messageConsumer.receive();
    TextMessage txtMsg5 = (TextMessage)
message5;

    //Now acknowledge it and all the messages will be
acknowledged.
    txtMsg5.acknowledge();

    //Try to receive again. This should return null as there is no message
available.
    Message messageAgain = messageConsumer.receive();
```

7.4 Message types

EDB-JDBC JMS API supports the following message types and can be used in a standard way.

Message type	JMS type
aq$_jms_message	javax.jms.Message
aq$_jms_text_message	javax.jms.TextMessage
aq$_jms_bytes_message	javax.jms.BytesMessage
aq$_jms_object_message	javax.jms.ObjectMessage

> **Note**
>
> The corresponding payload types (user-defined types) aren't predefined. You must create them before configuring the queue table, as shown in the examples that follow.

You can specify schema-qualified user-defined types, but the property types and message types must be in the same schema.

Message properties

All of the supported message types support setting and getting message properties. Before creating the actual message type, you must create the corresponding user-defined type for message properties.

This example shows how to create the user-defined type for message properties:

```
CREATE OR REPLACE TYPE AQ$_JMS_USERPROPERTY
AS object
(
    NAME VARCHAR2(100),
    VALUE VARCHAR2(2000)
);
```

All primitive types of message properties are supported.

TextMessage

You can send text messages using the `TextMessage` interface. `EDBTextMessageImpl` is an implementation of `TextMessage`, but for most cases you use the standard `TextMessage`. Before using the text message, you need to create a user-defined type for it.

This example shows how to create a user-defined message type for `TextMessage`:

```
CREATE OR REPLACE TYPE AQ$_JMS_TEXT_MESSAGE AS object(PROPERTIES AQ$_JMS_USERPROPERTY[], STRING_VALUE
VARCHAR2(4000));
```

Once the user-defined type is created, you can create the queue table using this type. This example shows how to create the queue table using the user-defined message created in the previous example:

```
EXEC DBMS_AQADM.CREATE_QUEUE_TABLE (queue_table => 'MSG_QUEUE_TABLE', queue_payload_type =>
'AQ$_JMS_TEXT_MESSAGE', comment => 'Message queue table');
```

After setting up the queue table, you can send and receive `TextMessages` using the standard procedure outlined in this Java code snippet:

```
MessageProducer messageProducer = (MessageProducer) session.createProducer(queue);
// Create text
message
TextMessage msg =
session.createTextMessage();
String messageText = "Hello
there!";
msg.setText(messageText);
msg.setStringProperty("myprop1", "test value 1");
// Send message
messageProducer.send(msg);

MessageConsumer messageConsumer = (MessageConsumer) session.createConsumer(queue);
// Receive Message
Message message = messageConsumer.receive();
TextMessage txtMsg = (TextMessage)
message;
System.out.println(txtMsg.getText());
System.out.println(txtMsg.getStringProperty("myprop1"));
```

BytesMessage

`BytesMessage` is used to send a stream of bytes. `EDBBytesMessageImpl` is an implementation of `BytesMessage`, but in most cases you use the standard `BytesMessage`. Before using `BytesMessage`, you must create a user-defined type.

This example shows how to create the user-defined type for `BytesMessage`:

```
CREATE OR REPLACE TYPE AQ$_JMS_BYTES_MESSAGE AS OBJECT (PROPERTIES AQ$_JMS_USERPROPERTY[], RAW_VALUE
CLOB);
```

Now, you can send and receive `BytesMessage` in the standard way.

This example shows how to create and use a `BytesMessage` in Java:

```
MessageProducer messageProducer = (MessageProducer) session.createProducer(queue);
BytesMessage msg =
session.createBytesMessage();
String messageText = "Hello
there!";
msg.writeBytes(messageText.getBytes());
messageProducer.send(msg);

MessageConsumer messageConsumer = (MessageConsumer) session.createConsumer(queue);
Message message = messageConsumer.receive();
BytesMessage byteMsg = (BytesMessage) message;
byteMsg.reset();
byte[] bytes = new byte[(int) byteMsg.getBodyLength()];
byteMsg.readBytes(bytes);
System.out.println(new String(bytes));
```

ObjectMessage

`ObjectMessage` is used to send a serializable object as a message. `EDBObjectMessageImpl` is an implementation of `ObjectMessage`, but the standard `ObjectMessage` is most commonly used.

Before using the `ObjectMessage`, you need to create the user-defined type for the object message.

This example shows how to create the user-defined type for `ObjectMessage`:

```
CREATE OR REPLACE TYPE AQ$_JMS_OBJECT_MESSAGE AS object(PROPERTIES AQ$_JMS_USERPROPERTY[], OBJECT_VALUE
CLOB);
```

For example, consider the following serializable Java class:

```
import java.io.Serializable;

public class Emp implements Serializable
{
    private int id;
    private String name;
    private String role;

    // Getter and setter
methods
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getRole() {
        return role;
    }

    public void setRole(String role) {
        this.role = role;
    }
}
```

This example shows how to use `ObjectMessage` to send a message containing an object of this class:

```
MessageProducer messageProducer = (MessageProducer) session.createProducer(queue);

// Create object
message
ObjectMessage msg =
session.createObjectMessage();
Emp emp = new
Emp();
emp.setId(1);
emp.setName("Joe");
emp.setRole("Manager");
msg.setObject(emp);

// Send message
messageProducer.send(msg);

MessageConsumer messageConsumer = (MessageConsumer) session.createConsumer(queue);

// Receive Message
Message message = messageConsumer.receive();
ObjectMessage objMsg = (ObjectMessage)
message;
Emp empBack = (Emp)
objMsg.getObject();
System.out.println("ID: " +
empBack.getId());
System.out.println("Name: " +
empBack.getName());
System.out.println("Role: " +
empBack.getRole());
```

Message

`Message` can be used to send a message with only properties and no body. `EDBMessageImpl` is an implementation of a `Message`, but you most often use the standard `Message`. Before using `Message`, create a user-defined type.

This example shows how to create a user-defined type for `Message`:

```
CREATE OR REPLACE TYPE AQ$_JMS_MESSAGE AS object(PROPERTIES
AQ$_JMS_USERPROPERTY[]);
```

This example shows how to send a message that contains only properties and no body:

```
MessageProducer messageProducer = (MessageProducer) session.createProducer(queue);
// Create message.
Message msg =
session.createMessage();
msg.setStringProperty("myprop1", "test value 1");
msg.setStringProperty("myprop2", "test value 2");
msg.setStringProperty("myprop3", "test value 3");
// Send message
messageProducer.send(msg);
MessageConsumer messageConsumer = (MessageConsumer) session.createConsumer(queue);
// Receive Message
message = messageConsumer.receive();
System.out.println("myprop1: " +
message.getStringProperty("myprop1"));
System.out.println("myprop2: " +
message.getStringProperty("myprop2"));
System.out.println("myprop3: " +
message.getStringProperty("myprop3"));
```

7.5 Non-standard message

EDB-JDBC JMS allows you to send and receive non-standard messages that are fully controlled by the API user. These messages don't support setting and getting properties. The process involves creating a user-defined type and setting it as the payload for the queue table.

This example shows how to create a Java Bean corresponding to the type you created:

```
package mypackage;
import com.edb.jms.common.CompareValue;
import java.util.ArrayList;
public class MyType extends com.edb.aq.UDTType
{
    private Integer code;
    private String project;
    private String manager;
    public MyType() {
    }
    /**
     * @param code the code to
set
     */
    @CompareValue(0)
    public void setCode(Integer code) {
      this.code = code;
    }
    /**
     * @return the
code
     */
    public Integer getCode() {
      return code;
    }
    /**
     * @param project the project to
set
     */
    @CompareValue(1)
    public void setProject(String project) {
      this.project = project;
    }
    /**
     * @return the
project
     */
    public String getProject() {
      return project;
    }
    @CompareValue(2)
    public void setManager(String manager) {
      this.manager = manager;
    }
    public String getManager() {
      return manager;
    }
    public String valueOf() {
        StringBuilder sql = new StringBuilder("CREATE TYPE
");
        sql.append(getName() + "
");
```

```
        sql.append("AS
(");
        sql.append("code int,
");
        sql.append("project
TEXT);");
        return
sql.toString();
    }
  /**
   * Override this method and call getter methods in the same order as in CREATE TYPE
statement.
   * CREATE OR REPLACE TYPE mytype AS object (code int, project text, manager
varchar(10))
   * @return object array containing
parameters.
   */
  @Override
  public Object[] getParamValues() {
    ArrayList<Object> params = new ArrayList<>
();

params.add(getCode());

params.add(getProject());

params.add(getManager());
    return
params.toArray();
  }
}
```

> **Note**
>
> - When you create a user-defined class, it must extend the `com.edb.aq.operations.UDTType` class and override the `getParamValues()` method. In this method, add the attribute values to an `ArrayList` in the same order as they appear in the CREATE TYPE SQL statement in the database.
> - Also make sure to use the annotation @CompareValue(0) with better methods, as it specifies the order of methods when using the reflection API to reconstruct the object after dequeuing the message from the queue.
>
> Failure to meet these requirements may result in errors.

This example shows how to send an object of this class as a message:

```
messageProducer = (EDBJmsMessageProducer) session.createProducer(queue);
      MyType udtType1 = new
MyType();
      udtType1.setProject("Test
Project");

udtType1.setManager("Joe");

udtType1.setCode(321);
      udtType1.setName("mytype"); //type name used in "CREATE
TYPE"
      messageProducer.send(udtType1);
```

This example shows how to receive this object as a message:

```
messageConsumer = (EDBJmsMessageConsumer) session.createConsumer(queue);

Message message = messageConsumer.receive();

MyType myt = (MyType)
message;
System.out.println("Code: "+
myt.getCode());
System.out.println("Project: "+
myt.getProject());
System.out.println("Manager: "+
myt.getManager());
```

Nested types

This example shows how to use nested types in the user-defined types:

```
CREATE OR REPLACE TYPE innermostcustom AS object (testing_field_1
text);

CREATE OR REPLACE TYPE innercustom AS object (testing_field_1 text, innermost
innermostcustom);

CREATE OR REPLACE TYPE custom_type AS object (testing_field text, inner
innercustom);
```

In this example, `custom_type` is using `innercustom` as another user-defined type that in turn is using the `innermostcustom` user-defined type. EDB Postgres Advanced Server supports the nested types this manner. However, it may have performance implications at a certain level. EDB JMS API also provides flexibility to read such nested types at the cost of an added performance impact.

To illustrate this using the EDB JMS API, you must first create the equivalent objects that represent nested custom types as shown in the examples that follow.

InnermostCustom.java

```
package mypackage;

import com.edb.aq.UDTType;
import com.edb.jms.common.CompareValue;
```

```
import java.util.ArrayList;

public class InnermostCustom extends UDTType {

    public InnermostCustom() {
    }

    private String testing_field_1;

    public String getTesting_field_1() {
        return testing_field_1;
    }

    @CompareValue(0)
    public void setTesting_field_1(String testing_field_1) {
        this.testing_field_1 = testing_field_1;
    }
  @Override
  public Object[] getParamValues(){
    ArrayList<Object> params = new ArrayList<Object>
();

params.add(getTesting_field_1());
    return
params.toArray();
  }
}
```

InnerCustom.java

```
package mypackage;

import com.edb.aq.UDTType;
import com.edb.jms.common.CompareValue;
```

```
import java.util.ArrayList;

public class InnerCustom  extends UDTType
{

    public InnerCustom() {
    }

    private String testing_field_1;
    private InnermostCustom innermostCustom;

    public String getTesting_field_1() {
        return testing_field_1;
    }

    @CompareValue(0)
    public void setTesting_field_1(String testing_field_1) {
        this.testing_field_1 = testing_field_1;
    }

  public InnermostCustom getInnermostCustom() {
    return innermostCustom;
  }

  @CompareValue(1)
  public void setInnermostCustom(InnermostCustom innermostCustom) {
    this.innermostCustom = innermostCustom;
  }
  @Override
  public Object[] getParamValues(){
    ArrayList<Object> params = new ArrayList<Object>
();

params.add(getTesting_field_1());

params.add(getInnermostCustom());
    return
params.toArray();
  }
}
```

CustomType.java

```
package mypackage;

import com.edb.aq.UDTType;
import com.edb.jms.common.CompareValue;
```

```
import java.util.ArrayList;

public class CustomType extends UDTType {

    private String testing_field;
    private InnerCustom
innerCustom;

    public String getTesting_field() {
        return testing_field;
    }

    @CompareValue(0)
    public void setTesting_field(String testing_field) {
        this.testing_field = testing_field;
    }

    public InnerCustom getInnerCustom()
{
        return
innerCustom;
    }

    @CompareValue(1)
    public void setInnerCustom(InnerCustom innerCustom)
{
        this.innerCustom =
innerCustom;
    }

    public CustomType() {

    }

    public Object[] getParamValues(){
        ArrayList<Object> params = new ArrayList<Object>
();

params.add(getTesting_field());

params.add(getInnerCustom());
        return
params.toArray();
    }
}
```

This example shows how to read these nested types:

```
      EDBJmsMessageProducer messageProducer = (EDBJmsMessageProducer)
session.createProducer(queue_1);

      InnermostCustom innermostCustom = new InnermostCustom();
      innermostCustom.setTesting_field_1("Innermost set");
      innermostCustom.setName("innermostCustom");

      InnerCustom innerCustom = new
InnerCustom();
      innerCustom.setTesting_field_1("Inner
set");

innerCustom.setInnermostCustom(innermostCustom);

innerCustom.setName("innercustom");

      CustomType customType = new CustomType();
      customType.setTesting_field("EDB");
      customType.setInnerCustom(innerCustom);
      customType.setName("custom_type");

      messageProducer.send(customType);

      EDBJmsMessageConsumer messageConsumer = (EDBJmsMessageConsumer)
session.createConsumer(queue_1);

      Message message = messageConsumer.receive();

      CustomType myType = (CustomType)
message;
      InnerCustom innerCustom_1 =
myType.getInnerCustom();
      InnermostCustom innermostCustom1 =
innerCustom_1.getInnermostCustom();

      System.out.println("Outer type test field: " +
myType.getTesting_field());
      System.out.println("Inner type test field: " +
innerCustom_1.getTesting_field_1());
      System.out.println("Most Inner type test field: " +
innermostCustom1.getTesting_field_1());
```

8 Executing SQL commands with executeUpdate() or through PrepareStatement objects

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 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 with a single parameter of type `String` containing the SQL command that you want to execute.

> **Avoid user-sourced values**
>
> We recommend that this string does not contain any user-sourced values. Avoid concatenating strings and values to compose your SQL command. Instead, use PreparedStatements which are reusable, parameterized SQL statements which safely manage the use of variable values in the SQL statement.

Using executeUpdate() to INSERT data

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

> **Code samples**
>
> The following examples are not a complete application, only example methods. These code samples don't 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.

```
    public void addOneEmployee(Connection
con)
    {
        try (Statement stmt=con.createStatement();)
        {
            int rowcount = stmt.executeUpdate("INSERT INTO emp(empno, ename)
VALUES(6000,'Jones')");
            System.out.println();
            System.out.printf("Success - %d - rows
affected.\n",rowcount);
        } catch(Exception err)
{
            System.out.println("An error has
occurred.");
            System.out.println("See full details below.");

err.printStackTrace();
        }
    }
```

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

```
    public void addOneEmployee(Connection
con);
```

A `Statement` object is needed to run `ExecuteUpdate()`. This can be obtained by using `createStatement()` on the Connection object. We use the try-resource style here to ensure the statement object is released when the try block is exited.

```
    try (Statement stmt=con.createStatement()) {
```

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).

```
    int rowcount = stmt.executeUpdate("INSERT INTO emp(empno, ename)
VALUES(6000,'Jones')");
```

If `executeUpdate()` returns without an error, the call to `System.out.printf` displays a message to the user that shows the number of rows affected.

```
    System.out.println();
    System.out.printf("Success - %d - rows
affected.\n",rowcount);
```

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

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

err.printStackTrace();
    }
```

You can use `executeUpdate()` with any SQL command that doesn't return a result set. It is best suited to situations where a specific command needs to be executed and that command takes no parameters.

To use the `DROP TABLE` command to delete a table from a database:

```
   Statement stmt=con.createStatement();
   stmt.executeUpdate("DROP TABLE tableName");
```

To use the `CREATE TABLE` command to add a new table to a database:

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

To use the `ALTER TABLE` command to change the attributes of a table:

```
    Statement stmt=con.createStatement();
    stmt.executeUpdate("ALTER TABLE tablename ADD COLUMN colname BOOLEAN ";
```

However, you should use `PreparedStatement` when passing values to an SQL insert or update statement, especially if those values have come from user input.

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.

The following shows invoking a `PreparedStatement` that accepts an employee ID and employee name and inserts that employee information in the `emp` table:

```
    public void addEmployee(Connection con, Integer id, String
name)
    {
        String command = "INSERT INTO emp(empno,ename)
VALUES(?,?)";
        try(PreparedStatement addstmt = con.prepareStatement(command)
{
            addstmt.setObject(1,id);
            addstmt.setObject(2,name);
            addstmt.execute();
            System.out.println("Employee
added");
        } catch(Exception err)
{
            System.out.println("An error has
occurred.");
            System.out.println("See full details below.");

err.printStackTrace();
        }
    }
```

This version of an add employee method takes as parameters the connection and values for the employee number (an integer) and name (a string).

Instead of hard coding data values in the SQL statement, you insert placeholders to represent the values to change with each iteration. The example 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:

```
     try(PreparedStatement addstmt = con.prepareStatement(command)
{
```

At this point, the `PreparedStatement` has parsed and planned the `INSERT` statement, but it doesn't know the values to add to the table. Before executing `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 in the parameters:

```
    addstmt.setObject(1,id);
    addstmt.setObject(2,name);
```

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

```
    addstmt.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.

Some simple syntax examples using `PreparedStatement` sending SQL commands follow:

To use the `UPDATE` command to update a row:

```
    public static void updateEmployee(Connection con, Integer id, String
name)
    {
        String command = "UPDATE emp SET ename=? where empno=?";
        try (PreparedStatement updateStmt = con.prepareStatement(command)) {
            updateStmt.setObject(1,id);
            updateStmt.setObject(2,name);
            updateStmt.execute();
        } catch(Exception err)
{
            System.out.println("An error has
occurred.");
            System.out.println("See full details below.");

err.printStackTrace();
        }
    }
```

For regularly and repeatedly used statements, the prepared statement can be initialized and reused.

```
    PreparedStatement preparedAddStmt;

    public void prepareStatements(Connection con)
{
        try {
            preparedAddStmt=con.prepareStatement("INSERT INTO emp(empno,ename)
VALUES(?,?)");
        } catch (SQLException e)
{
            throw new RuntimeException(e);
        }
    }

    public void addPreparedEmployee(Integer id, String name)
    {
        try {
            preparedAddStmt.setObject(1,id);
            preparedAddStmt.setObject(2,name);
            preparedAddStmt.execute();
        } catch(Exception err)
{
            System.out.println("An error has
occurred.");
            System.out.println("See full details below.");

err.printStackTrace();
        }
    }
```

This saves the system having to reparse and initialize the statement every time it is executed. Note that the prepared statement is prepared without a try-with-resource wrapper to ensure it is not closed when it leaves the `prepareStatements` method.

9 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 shows 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.

```
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 EDB Postgres 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'd 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 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`:

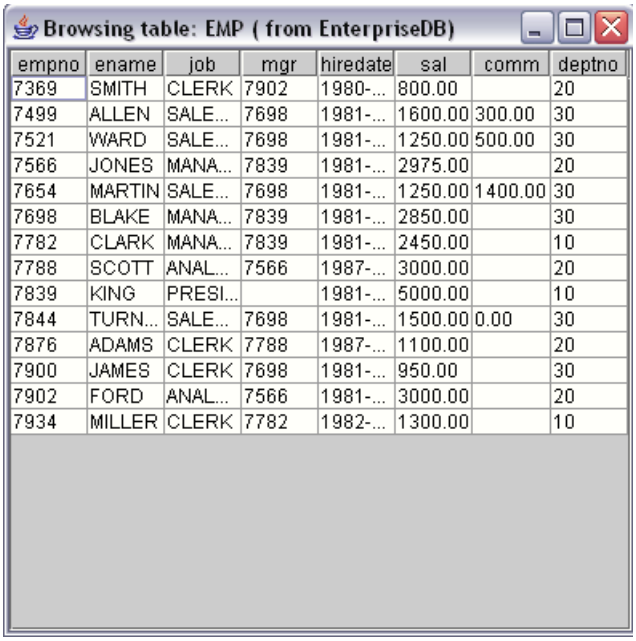
```
JTable table = new JTable(rows,
labels);
JFrame jf = new JFrame("Browsing table: EMP (from
EnterpriseDB)");
jf.getContentPane().add(new JScrollPane(table));
jf.setSize(400, 400);
jf.setVisible(true);
jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
System.out.println("Command successfully executed");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

The `showEmployees()` method includes a `catch` block to intercept any errors that 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 figure:



10 Advanced JDBC Connector functionality

These examples show you some of the advanced features of the EDB Postgres Advanced Server JDBC Connector.

10.1 Reducing client-side resource requirements

The EDB Postgres 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`:

- Reduces the amount of time it takes to retrieve the first row.
- Saves time by retrieving only the rows that you need.
- Reduces 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.

You can't use batched result sets in all situations. Not adhering to the following restrictions causes the driver to silently fall back to fetching the whole `ResultSet` at once:

- The client application must disable `autocommit`.
- You must create the `Statement` object with a `ResultSet` type of `TYPE_FORWARD_ONLY` type (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. The following code 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.");
```

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 can return the first row in less time.

10.2 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 this scenario.

The following code shows invoking a `PreparedStatement` that accepts an employee ID and employee name and inserts that employee information in the `emp` table:

```
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 change with each iteration. The following 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 doesn't know the 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:"));
```

It 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.

10.3 Executing stored procedures

A stored procedure is a module that's written in EDB's SPL and stored in the database. A stored procedure can define input parameters to supply data to the procedure and output parameters to return data from the procedure. Stored procedures execute in 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's 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. The square brackets indicate optional parameters. They aren't 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.

Executing a simple stored procedure

The following shows a stored procedure that increases the salary of each employee by 10%. `increaseSalary` expects no arguments from the caller and doesn't return any information:

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

The following shows 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 doesn't execute it. As Executing stored procedures with IN parameters shows, 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`) didn't expect any `IN` parameters and didn't return any information to the caller (using `OUT` parameters), so invoking the procedure is a matter of creating a `CallableStatement` object and then calling that object's `execute()` method.

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.

The following creates the stored procedure in the EDB Postgres 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;
```

The following shows 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's 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:

```
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;
```

The following 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's 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:

```
stmt.setObject(1, new Integer(c.readLine("Dept No :")));
```

Register the JDBC type of each `OUT` parameter before executing the `CallableStatement` objects. Registering the JDBC type is done with the `registerOutParameter()` method.

```
stmt.registerOutParameter(2, Types.VARCHAR);
stmt.registerOutParameter(3, Types.VARCHAR);
```

After executing the statement, the `CallableStatement` getter method retrieves the `OUT` parameter values. To retrieve a `VARCHAR` value, use the `getString()` getter method.

```
stmt.execute();
System.out.println("Dept Name: " + stmt.getString(2));
System.out.println("Location : " + stmt.getString(3));
```

In this 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.

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.

This code creates a stored procedure named `empQuery` :

```
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;
```

The following code shows invoking the `empQuery` procedure, providing values for the `IN` parameters, and handling the `OUT` and `IN OUT` parameters:

```
public void CallSample4(Connection
con)
{
  try
  {
    Console c =
System.console();
    String commandText = "{call
empQuery(?,?,?,?,?,?)}";
    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's later replaced with data:

```
String commandText = "{call empQuery(?,?,?,?,?,?)}";
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:

```
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:

```
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.

```
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);
```

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.

10.4 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` isn't tied to a particular query. You can 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 by way of the cursor variable. A `REF CURSOR` can also pass a result set from one procedure to another.

EDB Postgres Advanced Server supports the declaration of both strongly typed and weakly typed `REF CURSOR` variables. A strongly typed cursor must declare the `shape` (the type of each column) of the expected result set. You can use only 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 causes the server to return 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;
```

Using a REF CURSOR to retrieve a ResultSet

The stored procedure shown in the following (`getEmpNames`) builds two `REF CURSOR` variabes on the server. The first `REF CURSOR` contains a list of commissioned employees in the `emp` table. The second `REF CURSOR` contains a list of salaried employees in the `emp` table:

```
CREATE OR REPLACE PROCEDURE
getEmpNames
(
  commissioned OUT
SYS_REFCURSOR,
  salaried 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 shown in the following invokes the `getEmpName()` stored procedure and displays the names returned in each of the two `REF CURSOR` variables:

```
public void RefCursorSample(Connection
con)
{
  try
  {

con.setAutoCommit(false);
    String commandText = "{call
getEmpNames(?,?)}";
    CallableStatement stmt =
con.prepareCall(commandText);
    stmt.registerOutParameter(1, 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 `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.registerOutParameter(1, Types.REF);
```

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

```
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));
}
```

10.5 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. EDB Postgres Advanced Server can store and retrieve binary data by way of the `BYTEA` data type.

The following Java sample stores `BYTEA` data in an EDB Postgres Advanced Server database and then shows how to retrieve that data.

First, the following 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
);
```

The following 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;
```

Then, the following 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 EDB Postgres Advanced Server

The following shows a Java method that invokes the `ADD_PIC` procedure 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 doesn't 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());
```

Once 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 was 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 EDB Postgres Advanced Server database

Now that you know how to insert `BYTEA` data from a Java application, the following shows 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 to store the photograph:

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

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

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

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

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

10.6 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 EDB Postgres 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 EDB Postgres Advanced Server database.

These user-defined types can then be referenced in 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.

The following shows a Java method used by both upcoming examples to establish the connection to the EDB Postgres Advanced Server database.

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

Using an object type

Create the object types in the EDB Postgres 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. One of these attributes is object type `ADDR_OBJECT_TYPE`. 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;
```

The following 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 collections types `NUMBER_ARRAY` and `CHAR_ARRAY` in the EDB Postgres Advanced Server database:

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

The following 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;
```

The following is a Java method that calls the previous 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());
```

10.7 Asynchronous notification handling with NoticeListener

The EDB Postgres 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's 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`.

The following 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;
```

The following 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 from the caller, a `Connection` object:

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

`NoticeExample()` begins by preparing a call to the `adjustSalary` procedure shown previously. 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, the `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.

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 isn't empty, the JDBC driver delivers an `SQLWarning` to each listener in the order in which the listeners were added to the `Statement` .

11 Security and encryption

PostgreSQL has native support for using SSL connections to encrypt client/server communications for increased security. This requires that OpenSSL is installed on both client and server systems and that support in PostgreSQL is enabled at build time.

11.1 Using SSL

When using SSL, consider the following:

- Configuring the server
- Configuring the client
- Testing the SSL JDBC connection
- Using SSL without certificate Validation
- Using certificate authentication without a password

11.1.1 Configuring the server

For information about configuring PostgreSQL or EDB Postgres Advanced Server for SSL, see thePostgreSQL documentation.

> **Note**
>
> Before you access your SSL-enabled server from Java, ensure that you can log in to your server via `edb-psql`. If you've established an SSL connection, the output looks similar to this:

```
$ ./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=#
```

11.1.2 Configuring the client

A number of connection parameters are available for configuring the client for SSL. To know more about the SSL connection parameters and additional connection properties, see Connecting to the database.

When passed different values, the behavior of SSL connection parameters differs. When you pass the connection parameter `ssl=true` into the driver, the driver validates the SSL certificate and verifies the hostname. Conversely, using `libpq` defaults to a nonvalidating 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 hostname is verified to make sure it matches the name stored in the server certificate. The SSL connection fails if it can't verify the server certificate. 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 isn't configured to authenticate using the certificate, it should connect.

You can override the location of the client certificate, client key, and root certificate 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 an SSL connection, the JDBC driver validates 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're connecting to is the same as the hostname in the certificate.

11.1.3 Testing the SSL JDBC connection

If you're using Java's default mechanism (not `LibPQFactory`) to create the SSL connection, you need to make the server certificate available to Java.

1. Set the following 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:

```
$ openssl x509 -in server.crt -out server.crt.der -outform der
```

3. Import this certificate into Java's system truststore.

```
$ keytool -keystore $JAVA_HOME/lib/security/cacerts -alias postgresql-import -file server.crt.der
```

4. If you don't have access to the system cacerts truststore, create your own truststore.

```
$ 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**
>
> To troubleshoot 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 doesn't validate the SSL certificate. To enforce certificate validation, you must use a `Custom SSLSocketFactory`.

For more details about writing a `Custom SSLSocketFactory`, see the PostgreSQL documentation.

11.1.4 Using certificate authentication without a password

To use certificate authentication without a password:

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
```

11.2 Scram compatibility

The EDB JDBC driver provides SCRAM-SHA-256 support for EDB Postgres 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.

11.3 Support for GSSAPI-encrypted connection

> **New Feature **
>
> Support for GSSAPI-ecncrypted connections is available in EDB JDBC Connector release 42.2.19.1 and later.

The EDB JDBC driver supports GSSAPI-encrypted connections for EDB Postgres Advanced Server 12 onwards.

The `gssEncMode` parameter controls GSSAPI-encrypted connection. The parameter can have any of these values:

- `Disable`. Disables any attempt to connect using GSS-encrypted mode.
- `Allow`. Attempts to connect in plain text. Then, if the server requests it, it switches to encrypted mode.
- `Prefer`. Attempts to connect in encrypted mode and falls back to plain text if it fails to acquire an encrypted connection.
- `Require`. Attempts to connect in encrypted mode and fails to connect if that isn't possible.

GSSAPI/SSPI authentication

The default behavior of GSSAPI/SSPI authentication on Windows and Linux platforms is as following:

- On Windows, the EDB JDBC driver tries to connect using SSPI.
- On Linux, the EDB JDBC driver tries to connect using GSSAPI.

This default behavior is controlled using the `gsslib` connection parameter that takes one of the following values:

- `auto`. The driver attempts for SSPI authentication when the server requests it, the EDB JDBC client is running on Windows, and the waffle libraries required for SSPI are on the CLASSPATH. Otherwise it opts for Kerberos/GSSAPI authentication via JSSE. Unlike libpq, the EDB JDBC driver doesn't use the Windows SSPI libraries for Kerberos (GSSAPI) requests.
- `gssapi`. This option forces JSSE's GSSAPI authentication even when SSPI is available.
- `sspi`. This option forces SSPI authentication. This authentication fails on Linux or where SSPI is unavailable.

Using SSPI (Windows-only environment)

When the EDB Postgres Advanced Server and JDBC client both are on Windows, the JDBC driver connects with SSPI authentication using one of the following connection strings:

```
con = DriverManager.getConnection("jdbc:edb://localhost:5444/edb");
OR
con = DriverManager.getConnection("jdbc:edb://localhost:5444/edb?gsslib=sspi");
```

> **Note**
>
> - `gsslib=sspi` is optional because the server requires SSPI authentication.
> - There is no need to specify username and password. The logged-in user credentials are used to authenticate the user.

Example

The example assumes that SSPI authentication is configured on a Windows machine. Suppose the `edb-jdbc18.jar` path is `<PATH_DRIVER>` and the waffle libraries path is `<PATH_WAFFLE>`. Here's how to set `CLASSPATH` and run the JEdb sample:

```
set CLASSPATH=<PATH_DRIVER>\edb-jdbc18.jar;<PATH_WAFFLE>\*;
javac JEdb.java
java JEdb
```

Using GSSAPI (Linux-only environment)

When the EDB Postgres Advanced Server and JDBC client both are on Linux, the JDBC driver connects with GSSAPI authentication using the following connection string:

```
Properties connectionProps = new Properties();
connectionProps.setProperty("user", "postgres/myrealm.example@MYREALM.EXAMPLE");
String databaseUrl =
"jdbc:edb://myrealm.example:5444/edb";
con = DriverManager.getConnection(databaseUrl, connectionProps);
```

> **Note**
>
> `gsslib=gssapi` is optional because the server requires GSSAPI authentication.

Example

This example assumes that GSS authentication is configured on a Linux machine.

Create a file named `pgjdbc.conf` with the following contents.

```
pgjdbc {
com.sun.security.auth.module.Krb5LoginModule
required
doNotPrompt=true
useTicketCache=true
renewTGT=true
debug=true;
};
```

Suppose `pgjdbc.conf` is placed at `/etc/pgjdbc.conf`. Here's how to run JEdb sample:

```
javac JEdb.java
java -Djava.security.auth.login.config=/etc/pgjdbc.conf -cp .:edb-jdbc18.jar JEdb
```

Using SSPI/GSSAPI (Windows and Linux environment)

When the EDB Postgres Advanced Server is on Linux with authentication configured as GSSAPI, and the JDBC client is on Windows, the EDB JDBC connects either using SSPI or GSSAPI authentication.

For `gsslib=sspi` or `gsslib=auto`, EDB JDBC uses SSPI. For `gsslib=gssapi` it uses GSSAPI authentication.

Example

This example assumes that GSS authentication is configured between Windows Active Directory and a Linux machine.

SSPI

In this scenario, JDBC is using SSPI authentication. Create the connection using the following code:

```
Properties connectionProps = new Properties();
connectionProps.setProperty("user", "david@MYREALM.EXAMPLE");
String databaseUrl = "jdbc:edb://pg.myrealm.example:5444/edb?
gsslib=sspi";
con = DriverManager.getConnection(databaseUrl, connectionProps);
```

Running an EDB JDBC-based app in this case is the same as described in Using SSPI (Windows-only environment).

GSSAPI

In this scenario, JDBC is using GSSAPI authentication. Create the connection using the following code:

```
Properties connectionProps = new Properties();
connectionProps.setProperty("user", "david@MYREALM.EXAMPLE");
String databaseUrl = "jdbc:edb://pg.myrealm.example:5444/edb?
gsslib=gssapi";
con = DriverManager.getConnection(databaseUrl, connectionProps);
```

Set up the Kerberos credential cache file and obtain a ticket.

Create a new directory, say `c:\temp`, and a system environment variable `KRB5CCNAME`. In the variable value field, enter `c:\temp\krb5cache`.

> **Note**
>
> `krb5cache` is a file that's managed by the Kerberos software.

Obtain a ticket for a Kerberos principal either using MIT Kerberos Ticket Manager or using a `keytab` file using the `ktpass` utility.

Create the `pgjdbc.conf` file with the same contents described in Using GSSAPI (Linux-only environment).

Suppose `pgjdbc.conf` is placed at `c:\pgjdbc.conf`. Here's how to run JEdb sample:

```
set CLASSPATH=C:\Program Files\edb\jdbc\edb-jdbc18.jar;
java -Djava.security.auth.login.config=c:\pgjdbc.conf JEdb
```

12 EDB JDBC Connector logging

The EDB Postgres 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 EDB Postgres Advanced Server JDBC Connector used a custom mechanism to enable logging. It's 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.
>
> Previous versions of the Advanced Server JDBC Connector can enable logging using the connection properties, however it is no longer available from 42.3.3 onwards.

Enabling logging with logging.properties

The default Java logging framework stores its configuration in a file called `logging.properties`. 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
```

13 Reference - JDBC data types

The following table lists the JDBC data types supported by EDB Postgres Advanced Server and the JDBC Connector. If you're binding to an EDB Postgres 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 returns 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
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 supported only for JRE 4.2.

`Types.OTHER` is not only used for UUID but is also used if you don't specify a 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.