Perceptive Connect Runtime Database Connector

API Guide

Version: 3.x for Perceptive Connect Runtime, version 3.0.x

Written by: Documentation Team, R&D
Date: November 2025

HYLAND™

Documentation Notice

Table of Contents

Perceptive Connect Runtime Database Connector API

The Perceptive Connect Runtime (PCR) Database Connector provides a centralized interface for managing how PCR components and developers access your databases. The Database Connector provides features for external access to your configured Statement and Connection Descriptions through a REST API. Using Java Database Connectivity (JDBC), you can execute statements that query your database and return database results sets.

Edbc.sql package

The `edbc.sql` package contains the classes that wrap the JDBC interfaces. The classes in this package create and store the information needed for database connections and prepared SQL statements. This package also provides classes that execute and report the results of a prepared statement.

Info Classes

The info classes are lightweight, immutable plain old Java objects (POJOs). You must create ConnectionInfo and StatementInfo instances using the respective Info Factories. You may instantiate the DriverInfo, HostInfo, and UserInfo classes directly.

ConnectionInfo

A **ConnectionInfo** contains the information needed to create a database connection. Each **ConnectionInfo** contains two Strings and three info classes. The Strings represent a connection's database name and description, and the info classes are DriverInfo, HostInfo, and UserInfo. The **ConnectionInfo** has a getter for both strings; each info class; the **DriverInfo**'s driver class name, driver name, and driver version; the **HostInfo**'s host name, port, and network protocol; and the **UserInfo**'s user name and password. The Strings and info classes are never null, but the port, network protocol, driver name, and driver version may be null.

Unless you need direct control over a database connection, always use a **ConnectionInfo**. A **ConnectinoInfo** does not contain a database connection, so you can pass it around PCR without risking a resource leak. Additionally, the Database Connector components and services that handle database connections only accept **ConnectInfo** instances.

StatementInfo

A **StatementInfo** contains the information needed to create and execute a PreparedStatement. Each **StatementInfo** contains an SQL statement, a String-ParameterMetaType Map of input parameters, a String-ParameterMetaType Map of output parameters, a CursorType, a ConcurrencyType, and the **ConnectionInfo** that the StatementExecutor uses when it executes the prepared statement. Each property has a getter.

The **StatementInfo**'s input parameter Map is an ordered ImmutableMap that describes how the StatementExecutor passes information into a parameterized statement. The Map's keys provide labels for each parameter in the statement, and the Map's values describe the corresponding type for each parameter. The keys can be any non-null, non-empty String. The order of the keys determines the order that the StatementExecutor applies the values to the statement. If the **StatementInfo**'s SQL statement is not parameterized, then the **StatementInfo** has an empty input Map.

The **StatementInfo**'s output parameter Map is an ImmutableMap that describes the values the Result extracts from the executed statement's ResultSet. The Map's keys must match the columns that the Result will extract from the ResultSet. If the statement selects `*` , then the keys must match a column from the target table. The output Map's values describe the expected type for each output column. The order of the output parameters is irrelevant. If the **StatementInfo**'s statement does not return a ResultSet, then the **StatementInfo** has an empty output Map. If you plan to retrieve only the ResultSet from the executed statement, then the output parameters are optional.

The CursorType and ConcurrencyType represent static integers from the ResultSet. These integers determine the behavior of an executed statement's Result wrapper and ResultSet. For the supported type values, refer to CursorType and ConcurrencyType.

The type values have two limitations. If either your CursorType or ConcurrencyType is `DATABASE_DEFAULT` , then the other type must be `DATABASE_DEFAULT` . There are two reasons for this limitation.

- You cannot set a ResultSet's cursor and concurrency types separately.
- The database does not report its default cursor and concurrency types.

Additionally, `TYPE_SCROLL_INSENSITIVE` only works with `CONCUR_READ_ONLY`. This requirement is a limitation of the underlying ResultSet.

For information on the different Result behaviors, see the Appendix A: Result Behavior.

DriverInfo

A **DriverInfo** contains the information a ConnectionService needs to select a database connection's driver. Each DriverInfo contains three Strings: a driver class name, a driver name, and a driver version. Each String has a getter. The driver name and version may be null or empty, but the driver class name cannot be null or empty. A null or empty driver name or version indicates that the desired driver was not registered with the respective property. Because a ConnectionService uses null to represent driver information that is "not present," a **DriverInfo** stores empty Strings as null values. If the driver name or version does not matter, use the static String `DriverInfo.ANY` during construction for the respective arguments.

DriverInfo has the following constructors.

- `DriverInfo(String driverClassName)`
- `DriverInfo(String driverClassName, String driverVersion)`
- `DriverInfo(String driverClassName, String driverVersion, String driverName)`

The first constructor creates a **DriverInfo** that allows a ConnectionService to use a driver that has the given class name and any name or version. Similarly, the second constructor creates a **DriverInfo** that allows a driver with the given driver class, name, driver version, and any driver name. The third constructor only allows the ConnectionService to use a driver with the given driver name, class name, and version.

HostInfo

A **HostInfo** contains the information a ConnectionService needs to initiate a database connection. Each HostInfo contains three Strings: a host name, a port, and a network protocol. Each String has a getter. The port and network protocol may be null or empty, but the host name cannot be null or empty. A null or empty network prootoocol indicates that the ConnectionService will use the default protocol when it initiates the connection. Similarly, a null or empty port indicates that the ConnectionService will use the default port. Because a ConnectionService uses null to indicate default options, a **HostInfo** stores empty strings as null values.

HostInfo has the following constructors.

- `HostInfo()`
- `HostInfo(String hostname)`
- `HostInfo(String hostName, String port)`
- `HostInfo(String hostname, String port, String networkProtocol)`

The first constructor creates a **HostInfo** that has a null host name, the default port, and network protocol. The second constructor creates a **HostInfo** that has the given host name, the default port, and the default network protocol. The third constructor creates a **HostInfo** with the given host name, given port, and the default network protocol. The fourth constructor creates a **HostInfo** with the given host name, port, and network protocol.

UserInfo

A **UserInfo** contains the account information needed for a ConnectionService to initiate a database connection. Each **UserInfo** contains two Strings: a user name and password. Each String has a getter. Both values may be null or empty. A null or empty value indicates that the account does not have the associated value. Because the ConnectionService uses empty strings to represent that an account does not have a given property, a **UserInfo** stores null values as empty strings.

UserInfo has the following constructors.

- `UserInfo()`
- `UserInfo(String username, String password)`

The default constructor creates a **UserInfo** for connections that do not have a user name or password. The second constructor creates a **UserInfo** that has the given user name and password.

Info Factories

The various info factories build new **ConnectionInfo** and **StatementInfo** classes using different sources. These factories are the only way to construct new info class instances. You can create your own info factory classes, but your factories must extend the abstract factory associated with the desired info type. The InfoFactoryManager controls all the info factories installed in PCR. If your info factory is completely stateless and does not rely on anything inside PCR, then you can bypass the **InfoFactoryManager** and directly instantiate your info directory.

ConnectionInfoFactory

ConnectionInfoFactories construct new ConnectionInfo instances. Every **ConnectionInfoFactory** must extend the **AbstractConnectionInfoFactory**. All concrete factories must use the following method to create new instances.

```
protected ConnectionInfo make(String databaseName, String description,
  DriverInfo driverInfo, HostInfo hostInfo, UserInfo userInfo)
```

The arguments of the protected `make` method correspond directly to the values of a **ConnectionInfo**. Additionally, concrete factories must implement two methods from the **ConnectionInfoFactory** interface.

- `public String getName();`
- `public ConnectionInfo make(String identifier);`

The `getName` method returns the concrete factory's name. The InfoFactoryManager uses a factory's name for storage and retrieval. Unless your factory requires a specific name, use the factory's simple class name. The simple name reduces the chances of a name conflict inside PCR. If you do not use the factory's simple class name, you should implement a static method or field to get the factory's name without an instance of the factory.

The public `make` method accepts a single `String` used to identify the **ConnectionInfo** that the factory should make. If your concrete factory cannot construct a **ConnectionInfo** instance using a single String, then your `make` should throw an `UnsupportedOperationException`.

ConnectionInfoFactory Implementations

The `edbc.sql` package provides two concrete ConnectionInfoFactories.

- `FromPifConnectionInfoFactory`
- `InjectionConnectionInfoFactory`

FromPifConnectionInfoFactory

When you need a **ConnectionInfo** from the PCR's configuration UI, use the FromPifConnectionInfoFactory. The FromPifConnectionInfoFactory creates **ConnectionInfo** instances using information stored in PCR as ConnectionDescriptions. Use PCR's configuration UI to create ConnectionDescriptions. (For more information on ConnectionDescriptions, see the *Perceptive Connect Runtime Database Connector Configuration Guide*.) This factory supports `make(String identifier)`, and the factory uses ConnectionDescription names as the identifiers.

Note Do not store **ConnectionInfo** instances created using this factory. If PCR updates a ConnectionDescription, then PCR does not update any corresponding ConnectionInfo instances. To avoid using a stale instance, always create a new **ConnectionInfo** instance immediately before you need it.

InjectionConnectionInfoFactory

When you need to programmatically make a new **ConnectionInfo** from its raw components, use the InjectionConnectinInfoFactory. The InjectionConnectionInfoFactory is a stateless factory, so you can directly instantiate it. This factory does not support `make(String identifier)`. Instead, this factory provides a public version of the AbstractConnectionInfoFactory's protected `make`. This public `make` accepts a String database name, a String description, a DriverInfo, a HostInfo, and a UserInfo.

StatementInfoFactory

StatementInfoFactories construct new **StatementInfo** instances. Every StatementInfoFactory must extend the **AbstractStatementInfoFactory**. All concrete factories must use one of the following methods to create new instances.

```
protected StatementInfo make(String sql,
   Map<String, ParameterMetaType> intputParameters,
   Map<String, ParameterMetaType> outputParameters,
   ConnectionInfo connectionInfo)
protected StatementInfo make(String sql,
   Map<String, ParameterMetaType> intputParameters,
   Map<String, ParameterMetaType> outputParameters,
   ConnectionInfo connectionInfo,
   CursorType cursorType,
   ConcurrencyType concurrencyType)
```

The arguments of these protected `make` methods correspond directly to the values of a **StatementInfo**. Additionally, the concrete factories must implement three methods from the **StatementInfoFactory** interface.

- `public String getName();`
- `public StatementInfo make(String identifier)`
- `public StatementInfo make(String identifier, ConnectionInfo connectionInfo);`

The `getName` method returns the concrete factory's name. Like ConnectionInfoFactories, the InfoFactoryManager uses a factory's name for storage and retrieval. Unless your factory requires a specific name, use the factory's simple class name. The simple name reduces the chances of a name conflict inside PCR. If you do not use the factory's simple class name, then you should implement a static method or field to get the factory's name without an instance of the factory.

The `make(String identifier)` method accepts a single String used to identify the **StatementInfo** the factory should make. The `make(String identifier, ConnectionInfo connectionInfo)` accepts the same identifier as `make(String identifier)` to identify the **StatementInfo** the factory should make. However, the two-argument `make` uses the given **ConnectionInfo** to overwrite the **ConnectionInfo** used by `make(String identifier)`. If your concrete factory cannot construct a **StatementInfo** instance using either `make` method, then the unsupported `make` methods should throw and `UnsupportedOperationException`.

StatementInfoFactory Implementations

The `edbc.sql` package provides two concrete StatementInfoFactories.

- `FromPifStatementInfoFactory`
- `InjectionStatementInfoFactory`

FromPifStatementInfoFactory

When you need a **StatementInfo** from PCR's configuration UI, use the FromPifStatementInfoFactory. The **FromPifStatementInfoFactory** creates **StatementInfo** instances using information stored in PCR as StatementDescriptions. (For more information on StatementDescriptions, refer to the *Perceptive Connect Runtime Database Connector Configuration Guide*.) You use PCR's configuration UI to create StatementDescriptions. This factory supports both `make` methods. The factory uses StatementDescription names as the identifiers and accepts any valid **ConnectionInfo** instance.

Do not store **StatementInfo** instances created using this factory. If PCR updates a StatementDescription or a StatementDescription's **ConnectionInfo**, then PCR does not update any corresponding **StatementInfo** instances. When using this factory, always create a new **StatementInfo** instance immediately before you need it to avoid using a stale instance.

InjectionStatementInfoFactory

When you need to programmatically make a new **StatementInfo** from its raw components, use the InjectionStatementInfoFactory. The InjectionStatementInfoFactory is a stateless factory, so you can directly instantiate it. This factory does not support `make(String identifier)` but does support `make(String identifier, ConnectionInfo connectionInfo)`. This two-argument `make` uses a String SQL statement as the identifier. The new **StatementInfo** uses the given **ConnectionInfo** during execution and does not have any input or output parameters.

In addition to `make(String identifier, ConnectionInfo connectionInfo)`, this factory provides public versions of the AbstractStatementInfoFactory's protected `make` methods. These `make` methods accept a SQL statement String, an ordered String-ParameterMetaType Map for input parameters, a String-ParameterMetaType Map for output parameters, and a **ConnectionInfo** to use during execution. The second `make` method also accepts a CursorType and a ConcurrencyType, so you can override the default Result behavior using these type arguments. For information on the different Result behaviors, refer to Appendix A: Result Behavior.

StatementExecutor and Result

The StatementExecutor and Result are the Database Connector's primary SQL classes. These classes use the Info Classes to create and execute statements and to retrieve SQL results. You can instantiate the StatementExecutor anywhere you need it, but only the StatementExecutor can create new Results. Additionally, the StatementExecutor can execute compound SQL statements, but the Result class only reports the first statement's ResultSet and update count.

StatementExecutor

The StatementExecutor executes prepared SQL statements and wraps the executed statements in Results. The StatementExecutor uses a ConnectionService to create database connections for its prepared statements. The StatementExecutor's Constructor allows you to create an executor with any ConnectionService.

A statement executor uses the `executeStatement()` method to create and execute prepared SQL statements.

```
public Result executeStatement(StatementInfo statementInfo, Map<String, Object>
parameters)
```

The `executeStatement()` method accepts any valid **StatementInfo** and a String-Object Map. The Map contains the values the executor will assign to the prepared statement. The method's Map keys must match the **StatementInfo**'s input parameter Map keys. The executor ignores any extra values in the method's Map, but if any values are missing, then the method throws an `EdbcException`. For each corresponding `key` in the method's Map and **StatementInfo**'s input parameter Map, the object's class from the method's Map must match the ParameterType from the **StatementInfo**'s input parameter Map.

Result

The Result class is a wrapper for executed SQL statements. The Result is an auto-closable resource, so make sure that you place Result in a try-with-resource or call `close` when you are finished with it. An open Result contains an open Statement and Connection, and it may contain an open ResultSet. If you never close a Result, then that Result's Connection will never close.

A Result provides two methods for determining its current state.

- `public boolean isClosed()`
- `public boolean isResultSet()`

The `isClosed` method determines if the Result's Connection, ResultSet, or Statement is closed. If the Connection, ResultSet, or Statement is closed, the data retrieval methods fail. After a Result is closed, you cannot reopen it.

The `isResultSet` method determines if the executed statement's data is a ResultSet. The value of the `isResultSet` is the value returned by an SQL statement's `execute` method, but this method does not re-execute the SQL statement.

The Result class also provides two methods for determining its ResultSet's behavior.

- `public CursorType getCursorType()`
- `public ConcurrencyType getConcurrenyType()`

The combination of CursorType and ConcurrencyType determine how you may access data from the ResultSet. For the different Result behaviors, refer to Appendix A: Result Behavior.

Result provides three primary methods for retrieving data from your statement.

- `public List<Map<String, Object>> getResultList()`
- `public ResultSet getSqlResultSet()`
- `public int getUpdateCount()`

Additionally, the Result provides one helper method for ResultSet data.

- `public Map<String, ParameterMetaType> getOutputParameters()`

The `getSqlResultSet` method returns the executed statement's ResultSet, and `getUpdateCount` returns the number of rows affected by the executed statement. Both methods behave like the SQL Statement's equivalent methods, except you can call the Result methods more than once.

If the **StatementInfo** used to create a Result has output parameters, then the `getOutputParameters` method returns a String-ParameterMetaType Map representing the column names and expected types of the data in the ResultSet's rows. If the **StatementInfo** does not have output parameters, then `getOutputParameters` returns an empty Map, so you need to get the expected column names and types from another source.

The `getResultList` method extracts row data from the executed statement's ResultSet and returns the data as an ordered List of String-Object Maps. Each Map represents a row from the ResultSet, each Map key represents a column from the row, and each value represents the data at the row-column location. The `getResultList` method does not limit the size of the return List nor does it page the ResultSet's data, so large ResultSets may take a long time to extract. If the Result's executed statement does not have output parameters or if it does not return a ResultSet, then `getResultList` returns null.

Edbc.connect.service

The `edbc.connect.service` package contains the Database Connector's helpers and services. You can instantiate these classes, as you need them.

ConnectionService

ConnectionService is an interface for classes that create a `java.sql.Connection` using a given **ConnectionInfo**. Every ConnectionService class must implement the following functionality.

```
public java.sqlConnection makeConnection(ConnectionInfo connectionInfo)
```

The StatementExecutor requires a ConnectionService to create and execute prepared SQL statements. The Database Connector wraps the connections created through this service, but you can use a ConnectionService to retrieve an exposed `java.sql.Connection`.

DataSourceConnectionService

The DataSourceConnectionService is a ConnectionService that uses DataSources and DataSourceFactories to create new `java.sql.Connections`. You must install your drivers and DataSourceFactories before you can use this implementation.

This service requires a BundleContext for its constructor, so you must use a PCR component or service to create a DataSourceConnectionService. This ConnectionService uses the BundleContext to retrieve the DataSourceFactories from PCR.

ParameterMetatype

ParameterMetatype is a Class that controls the information about a Parameter. It holds three pieces of information about a parameter: *label*, *type*, and *format*. The *label* is how the Parameter is associated with the data received as an Input Parameter, or as the data passed back as an Output Parameter. The *type* is the ParameterType for the object. The *format* is optional. It is only supported by three ParameterTypes: `DATE`, `TIME`, and `TIMESTAMP`.

ParameterMetatype provides two different constructors.

- `public ParameterMetatype(ParameterType type, String label);`
- `public ParameterMetatype(ParameterType type, String label, String format);`

Additionally, ParameterMetatype provides a single static method to parse a Parameter String into a ParameterMetatype. For more information on Parameter Strings, see "Statement Description Parameters" in the *Perceptive Content Runtime Database Connector Configuration Guide*.

- `public static fromString(String metatype, String delimiter);`

ParameterMetatype provides two different methods of Object conversion: Object -> String and String -> Object.

- `public String formatValue(Object value);`
- `public Object parseObject(String string);`

The `formatValue()` method accepts the actual Object and returns a String representation of that object using the ParameterMetatype's format. The `parseObject()` method accepts a String representation of an object. If the ParameterMetatype does not have a format, it uses the ParameterType.parseObject method to parse the value. Otherwise, it uses the given format to parse the value.

ParameterType

ParameterType is an enumeration that controls the Database Connector's supported data types. The Database Connector uses this enumeration to declare and verify StatementInfo parameter types and to retrieve the correct data type from an executed statement.

The ParameterTypes that the Database Connector fully supports are:

- `STRING`
- `LONG`
- `INTEGER`
- `SHORT`
- `BYTE`
- `DOUBLE`
- `FLOAT`
- `BOOLEAN`
- `DATE`
- `TIME`
- `TIMESTAMP`

The Database Connector partially supports `OBJECT`. Input parameters cannot have the `OBJECT` type, but output parameters can. `DATE`, `TIME`, and `TIMESTAMP` use the Java formats to parse their respective values. If a query uses DateTime, `TIMESTAMP` can be used as the corresponding ParameterType.

ParameterType provides two static methods to retrieve a ParameterType value.

- `public static ParameterType fromClass(Class<?> clazz)`
- `public static ParameterType fromString(String name)`

The `fromClass()` method accepts the class that corresponds with the desired ParameterType. The `fromString()` method accepts the `name()` method's String or the simple class name. The `fromString()` method's argument is not case sensitive. Both methods will throw an `IllegalArgumentExeption` if you call either method with an argument that does not match a ParameterType.

DateTime Formatting

The ParameterTypes `DATE`, `TIME`, and `TIMESTAMP` support custom formats for input and output formatting. There are two ways to provide an input parameter format.

- HTTP Request
- Parameter Declaration

HTTP Request. To declare a parameter's Date format using an HTTP request, add an additional parameter `name-format`, where name is the parameter you want to format. For example, given a StatementDescription named UpdateBirthdays with two Input Parameters, `Date#NewBirthday` and `String#Who`, the request would be:

```
http://hostname:port/rs/databaseConnector/statement/UpdateBirthdays?Who=Bob&NewBirthday=05-14- 2015&NewBirthday-format=MM- dd-yyyy
```

Parameter Declaration. To declare a parameter's format using the StatementDescription configuration, insert your delimiter after the parameter's name and add a custom format. For example: given a StatementDescription named UpdateBirthdays with two Input Parameters, `Date#NewBirthday#MM/dd/yyyy` and `String#Who`, the request would be:

```
http://hostname:port/rs/databaseConnector/statement/UpdateBirthdays?Who=Bob&NewBirthday=05-14- 2015&NewBirthday-format=MM- dd-yyyy
```

Note An HTTP request's date format overwrites a parameter's configured format.

Output Format. To declare a parameter's output format, insert the delimiter after the output parameter's name and add the custom format. For example: given a `Date` parameter named `NewBirthday`, a delimiter `#`, and the format `MM/dd/yyyy`, the parameter's declaration would be `Date#NewBirthday#MM/dd/yyyy`

All formats must follow Java's SimpleDateFormatter patterns.

Parsing Strings

ParameterType provides one instance method to parse a String to an Object of the Class defined by the given ParameterType.

```
public Object parseObject(String value)
```

If `parseObject()` is unable to parse the given String to the Class defined by the ParameterType, either a `NumberFormatException` or `Exception` is thrown.

Examples

```
ParameterType.INTEGER.parseObject("123"): // returns (Integer)123
ParameterType.BOOLEAN.parseObject("true"); // returns (Boolean)true
ParameterType.BOOLEAN.parseObject("NotABoolean"); // throws Exception
```

ConcurrencyType

ConcurrencyType is an Integer enumeration that affects the behavior of a Result. The StatementExecutor uses this enumeration when it creates new prepared Statements.

The Concurrency types are:

- `CONCUR_READ_ONLY`
- `CONCUR_UPDATABLE`
- `DATABASE_DEFAULT`

`CONCUR_READ_ONLY` and `CONCUR_UPDATABLE` correspond to the ResultSet's static integers with the same names. `DATABASE_DEFAULT` causes the StatementExecutor to use the database's concurrency setting.

ConcurrencyType has two static public methods.

- `public static ConcurrencyType getDefault()`
- `public static LinkedHashMap<String, String> getOptions()`

The `getDefault` method retrieves the default CursorType used by the Database Connector. The default value is `CONCUR_READ_ONLY`. The `getOptions` method retrieves the enumeration values as a String-String Map, where both Strings are the names of the enumeration values. Additionally, CurrencyType has one non-static public method.

```
public int getValue()
```

The `getValue` method returns the integer value associated with the current CurrencyType instance. For the effect of ConcurrencyTypes on a Result, see Appendix A: Result Behavior.

CursorType

CursorType is an Integer enumeration that affects the behavior of a Result. The StatementExecutor uses this enumeration when it creates new prepared Statements.

The CursorTypes are:

- `TYPE_FORWARD_ONLY`
- `TYPE_SCROLL_INSENSITIVE`
- `TYPE_SCROLL_SENSITIVE`
- `DATABASE_DEFAULT`

`TYPE_FORWARD_ONLY`, `TYPE_SCROLL_INSENSITIVE`, and `TYPE_SCROLL_SENSITIVE` correspond to the ResultSet's static integers with the same names. `DATABASE_DEFAULT` causes the StatementExecutor to use the database's cursor setting.

CursorType has two static public methods.

- `public static CursorType getDefault()`
- `public static LinkedHashMap<String, String> getOptions()`

The `getDefault` method retrieves the default CursorType sued by the Database Connector. The default value is `TYPE_FORWARD_ONLY`. The `getOptions` method retrieves the enumeration values as a String-String Map, where both Strings are the names of the enumeration values. Additionally, CursorType has one non-static public method.

```
public int getValue()
```

The `getValue` method returns the integer value associated with the current CursorType instance. For the effect of CursorType on a Result, see Appendix A: Result Behavior.

Edbc.connect.component

The `edbc.connect.component` package contains the Database Connector's PCR components. Your services and components can directly bind and unbind each component in this package.

InfoFactoryManager

The InfoFactoryManager provides access to every info factory registered inside your PCR. The manager can retrieve any service that provides either the ConnectionInfoFactory or StatementInfoFactory interface.

To retrieve a registered info factory, you must bind the InfoFactoryManager to your component or service. After you bind the manager, call one of the following methods to retrieve your desired factory.

- `public ConnectionInfoFactory getConnectionInfoFactory()`
- `public ConnectionInfoFactory getConnectionInfoFactory(String name)`
- `public StatementInfoFactory getStatementInfoFactory()`
- `public StatementInfoFactory getStatementInfoFactory(String name)`

The `getConnectionInfoFactory()` returns the FromPifConnectionInfoFactory and the `getStatementInfoFactory()` returns the FromPifStatementInfoFactory. These two factories are the default factories because they are in the same install bundle as the InfoFactoryManager.

The `getConnectionInfoFactory(String name)` returns the ConnectionInfoFactory with the given name, and `getStatementInfoFactory(String name)` returns the StatementInfoFactory with the given name. If a given name does not belong to a factory, then these two methods will return null. Additionally, if the given name is null, then these methods will return the "from PIF" version of their respective factories.

The manager also has four public methods for registering and unregistering info factories. You should not call these methods. These methods are required by PCR to register and unregister the info factories. Manually calling these methods may remove or replace a factory that another component is using. If you want your custom factory in the manager, install a component that provides your factory as a service. The manager automatically registers your factory.

The InfoFactoryManager is a standard PCR LifeCycleComponent. The manager does not have any external service requirements, and it provides itself as a service. Additionally, the manager has bind and unbind methods with `0..n` cardinality for the ConnectionInfoFactory and StatementInfoFactory interfaces.

FromPifConnectionInfoFactory

For the service functionality of this factory, see FromPifConnectionInfoFactory under the `edbc.sql` package.

The FromPifConnectionInfoFactory is a PCR LifeCycleComponent that implements the ConnectionInfoFactory interface. This component does not have any external service requirements, and it provides the FromPifConnectionInfoFactory and the ConnectionInfoFactory services.

FromPifStatementInfoFactory

For the service functionality of this factory, see FromPifStatementInfoFactory under the `edbc.sql` package.

The FromPifStatementInfoFactory is a PCR LifeCycleComponent that implements the StatementInfoFactory interface. This component requires the FromPifConnectionInfoFactory as an external service. This component provides the FromPifStatementInfoFactory and StatementInfoFactory services.

Appendix A: Result Behavior

The combination of a Result's ConcurrencyType and CursorType determine the behavior of the Result's underlying ResultSet. This appendix details the behavior for each combination. These section labels follow a "CursorType-ConcurrencyType" pattern.

TYPE_FORWARD_ONLY-CONCUR_READ_ONLY

In a `TYPE_FORWARD_ONLY-CONCUR_READ_ONLY` Result, you can make one forward pass through the ResultSet, and you cannot use the cursor to update data. The Result's `getResultList` and `getSqlResultSet` methods use the same underlying ResultSet, so these methods interfere with each other. After you completely iterate through the underlying ResultSet, neither method will return any useful data. The `getResultList` method completely iterates through a ResultSet. After this first call, `getResultList` will return an empty List and `getSqlResultSet` will return an iterated ResultSet. Also, if you manually iterate through a ResultSet from a fresh Result's `getSqlResultSet`, then `getResultList` will return an empty list.

TYPE_FORWARD_ONLY-CONCUR_UPDATABLE

The `TYPE_FORWARD_ONLY-CONCUR_UPDATABLE` combination behaves like the `TYPE_FORWARD_ONLY-CONCUR_READ_ONLY` combination. The only difference between the two combinations is that `TYPE_FORWARD_ONLY-CONCUR_UPDATABLE` uses a cursor that can update the database.

TYPE_SCROLL_INSENSITIVE-CONCUR_READ_ONLY

The `TYPE_SCROLL_INSENSITIVE-CONCUR_READ_ONLY` combination removes the limitations that `TYPE_FORWARD_ONLY` imposes on a Result's method. With a `TYPE_SCROLL_INSENSITIVE` cursor, `getResultList` and `getSqlResultSet` reset the underlying ResultSet's cursor before execution. So `getResultList` will always return the same Map List, and `getSqlResultSet` will always return a ResultSet with a fresh cursor. The `TYPE_SCROLL_INSENSITIVE` cursor has two limitations. This cursor uses a database snapshot and it cannot update any rows. Because of the database snapshot, any stored Results using this cursor become stale over time.

TYPE_SCROLL_SENSITIVE-CONCUR_READ_ONLY

The `TYPE_SCROLL_SENSITIVE-CONCUR_READ_ONLY` combination behaves like a `TYPE_SCROLL_INSENSITIVE-CONCUR_READ_ONLY` that does not use a database snapshot. The `TYPE_SCROLL_SENSITIVE` cursor can see any changes to existing rows in its ResultSet. This visibility allows the ResultSet to see row updates, and row deletions appear as missing data. However, inserted rows are not visible. Additionally, this cursor cannot update the database.

TYPE_SCROLL_SENSITIVE-CONCUR_UPDATABLE

The `TYPE_SCROLL_SENSITIVE-CONCUR_UPDATABLE` behaves like a `TYPE_SCROLL_SENSITIVE-CONCUR_READ_ONLY` that can update a database. This similarity includes the database-visibility behavior. Updates to current rows are visible, deletes appear as missing data, and inserts are not visible. Compared to the other combinations, this combination provides the most flexibility.

DATABASE_DEFAULT-DATABASE_DEFAULT

The `DATABASE_DEFAULT-DATABASE_DEFAULT` combination uses the database's default cursor and concurrency types. If you do not know your database's default and you need a specific behavior, explicitly use one of the other supported combinations.

Unsupported Combinations

The ResultSet's limitations prevent us from using the following combinations.

- `TYPE_SCROLL_INSENSITIVE-CONCUR_READ_ONLY`
- `DATABASE_DEFAULT-CONCUR_READ_ONLY`
- `DATABASE_DEFAULT-CONCUR_UPDATABLE`
- `TYPE_FORWARD_ONLY-DATABASE_DEFAULT`
- `TYPE_SCROLL_INSENSITIVE-DATABASE_DEFAULT`
- `TYPE_SCROLL_SENSITIVE-DATABASE_DEFAULT`

Appendix B: JPA Integration

The Database Connector's DataSources support JPA and its persistence units. When you configure a Connection in the Database Connector, the Connector registers a DataSource as a service in OSGi. PCR comes with EclipseLink/Gemini installed, and this appendix provides information for their specific integration. The information contained in this appendix may also help you integrate with other JPA providers.

There are four options for using the Database Connector's configured DataSources.

Notes

- In the following examples, `MyConnectionName` refers to the name configured for your connection in the configuration GUI.
- DataSources registered by the Database Connector do not support the Java Transaction API (JTA)
- Options one through three require a JNDI provider. Refer to JNDI Provider for more information.

Option 1: Non-JTA DataSource in Persistence Unit

Define a `non-jta-data-source` in your persistence unit XML file.

Note This option requires a JNDI provider installed in your PCR instance. Refer to JNDI Provider for more information.

Add `non-jta-data-source` to your persistence unit XML file.

```
<persistence-unit name="MyDatabase" transaction-type="RESOURCE_LOCAL">
    ...
    <non-jta-data-source>
        osgi:service/javaax.sql.DataSource/
        (osgi.jndi.service.name=MyConnectionName)
    </non-jta-data-source>
    ...
</persistence-unit>
```

When configured using a JNDI name, Gemini JPA does not correctly recognize `non-jta-data-source` and does not generate and register an `EntityFactoryManager.` Instead, Gemini attempts to configure a new DataSource and pass that to EclipseLink. To fix this, bind or otherwise obtain the Gemini-generated `EntityManagerFactoryBuilder` and create an `EntityManagerFactory`, passing the `gemini.jpa.providerConnectedDataSource` property.

Option 2: Non-JTA DataSource in Code

Define the `javax.persistence.nonJtaDataSource` property without modifying your persistence unit XML file.

Note This option requires a JNDI provider installed in your PCR instance. Refer to JDNI provider for more information.

```
public void bind(EntityManagerFactoryBuilder builder)
{
    Map<String, Object> props = new HashMap<String, Object>();
    props.put("javax.persistence.nonJtaDataSource",
        "osgi:service/javax.sql.DataSource/
         (osgi.jndi.service.name=MyConnectionName)");
    props.put("Gemini.jpa.providerConnectedDataSource", true);
    EntityManagerFactory factory = builder.createEntityManagerFactory(props);
    ... // Do something with the factory
}
```

Option 3: Directly Obtain the DataSource from JNDI

Obtain a DataSource from JNDI and set the `javax.persistence.nonJtaDataSource` property to that DataSource.

Note This option requires a JNDI provider installed in your PCR instance. Refer to JNDI Provider for more information.

```
public void bind(EntityManagerFactoryBuilder builder) throws NamingException
{
    Map<String, Object> props = new HashMap<String, Object>();
    javax.naming.Context context = new InitialContext();
    DataSource dataSource = (DataSource)
        context.lookup("osgi:service/javax.sql.DataSource/
                        (osgi.jndi.service.name=MyConnectionName)");
    props.put("java.persistence.nonJtaDataSource", dataSource);
    EntityManagerFactory factory = builder.createEntityManagerFactory(props);
    ... // Do something with the factory
}
```

Option 4: Bind to DataSource

Using DeclarativeServices, bind to the configured DataSource. This option lets you use the ConfigurationAdmin to set the target filter for the bound DataSource.

Note This option does not require a JNDI provider.

In your DeclarativeServices (DS) XML file, enter the following information.

```
<scr:component xmlns:scr=http://www.osgi.org/xmlns/scr/v1.1.0
               activate="activate"
                deactivate="deactivate"
                name="MyComponent">
    ...
    <reference bind="bind"
               cardinality="1..1"
               interface="org.osgi.service.jpa.EntityManagerFactoryBuilder"
               name="EntityManagerFactoryBuilder" policy="dynamic"
               target="(osgi.unit.name=MyDatabasee)"/>
    <reference bind="bind"
               cardinality="1..1"
               interface="javex.sql.DataSource"
               name="DataSource"
               policy="dynamic"
               target="(osgi.jndi.service.name=MyConnectionName)"/>
    ...
</scr:component>
```

In your component implementation, enter the following information.

```
DataSource dataSource = null;

public void bind(DataSource dataSource)
{
    this.dataSource = dataSource;
}

public void bind(EntityManagerFactoryBuilder builder)
{
    Map<String, Object> props = new HashMap<String, Object>();
    props.put("javax.persistence.nonJtaDataSource", dataSource);
    EntityManagerFactory factory = builder.createEntityManagerFactory(props);
    ... // Do something with the factory
}
```

JNDI Providers

Apache Aries is the recommended JNDI provider. You can download the required jars from the Apache Aries website.

Apache Aries provides a number of services, but only a few are required to enable JNDI in PCR.

- Apache Aries JNDI Bundle (org.apache.aries.jndi)
- Apache Aries Proxy Bundle (org.apache.aries.proxy)
- Apache Aries Util (org.apache.aries.util)

Download the latest version of each and install them in PCR like any other bundle.

Troubleshooting

In some cases, you may encounter an error like the following example.

```
java.lang.IllegalArgumentException: Object:
com.yourcompany.YourClass@268f34f6 is not a known entity type.
```

First, check that your persistence unit is aware of your model. Make sure that your class is listed. We recommend that you also set `exclude-unlisted-classes` to `true`.

```
<persistence-unit name="MyDatabase" transaction-type="RESOURCE_LOCAL">
    ...
    <class>com.yourcompany.YourClass</class>
    <exclude-unlisted-classes>true</exclude-unlisted-classes>
    ...
</persitence-unit>
```

If you have correctly configured your persistence unit, it is possible that a ClassLoader issue is to blame. This can be resolved by setting the `eclipselink.classloader` property when creating an `EntityManagerFactory` from your `EntityManagerFactoryBuilder`.

```
public void bind(EntityManagerFactoryBuilder builder)
{
    Map<String, Object> props = new HashMap<String, Object>();
    ...
    props.put("eclipselink.classloader", this.getClass().getClassLoader());
    EntityManagerFactory factory = builder.createEntityManagerFactory(props);
    ... // Do something with the factory
}
```