| | [**Overview**](http://docs.google.com/overview-summary.html) | **Package** | Class | [**Use**](http://docs.google.com/package-use.html) | [**Tree**](http://docs.google.com/package-tree.html) | [**Deprecated**](http://docs.google.com/deprecated-list.html) | [**Index**](http://docs.google.com/index-files/index-1.html) | [**Help**](http://docs.google.com/help-doc.html) | | --- | --- | --- | --- | --- | --- | --- | --- | | | ***Java™ Platform***  ***Standard Ed. 6*** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [**PREV PACKAGE**](http://docs.google.com/javax/sql/rowset/serial/package-summary.html)   [**NEXT PACKAGE**](http://docs.google.com/javax/swing/package-summary.html) | [**FRAMES**](http://docs.google.com/index.html?javax/sql/rowset/spi/package-summary.html)    [**NO FRAMES**](http://docs.google.com/package-summary.html)     [**All Classes**](http://docs.google.com/allclasses-noframe.html) |

## Package javax.sql.rowset.spi

The standard classes and interfaces that a third party vendor has to use in its implementation of a synchronization provider.

**See:**

[**Description**](#3znysh7)

| **Interface Summary** | |
| --- | --- |
| [**SyncResolver**](http://docs.google.com/javax/sql/rowset/spi/SyncResolver.html) | Defines a framework that allows applications to use a manual decision tree to decide what should be done when a synchronization conflict occurs. |
| [**TransactionalWriter**](http://docs.google.com/javax/sql/rowset/spi/TransactionalWriter.html) | A specialized interface that facilitates an extension of the standard SyncProvider abstract class so that it has finer grained transaction control. |
| [**XmlReader**](http://docs.google.com/javax/sql/rowset/spi/XmlReader.html) | A specialized interface that facilitates an extension of the SyncProvider abstract class for XML orientated synchronization providers. |
| [**XmlWriter**](http://docs.google.com/javax/sql/rowset/spi/XmlWriter.html) | A specialized interface that facilitates an extension of the SyncProvider abstract class for XML orientated synchronization providers. |

| **Class Summary** | |
| --- | --- |
| [**SyncFactory**](http://docs.google.com/javax/sql/rowset/spi/SyncFactory.html) | The Service Provider Interface (SPI) mechanism that generates SyncProvider instances to be used by disconnected RowSet objects. |
| [**SyncProvider**](http://docs.google.com/javax/sql/rowset/spi/SyncProvider.html) | The synchronization mechanism that provides reader/writer capabilities for disconnected RowSet objects. |

| **Exception Summary** | |
| --- | --- |
| [**SyncFactoryException**](http://docs.google.com/javax/sql/rowset/spi/SyncFactoryException.html) | Indicates an error with SyncFactory mechanism. |
| [**SyncProviderException**](http://docs.google.com/javax/sql/rowset/spi/SyncProviderException.html) | Indicates an error with the SyncProvider mechanism. |

## Package javax.sql.rowset.spi Description

The standard classes and interfaces that a third party vendor has to use in its implementation of a synchronization provider. These classes and interfaces are referred to as the Service Provider Interface (SPI). A vendor may have its implementation included on the JDBC web page that lists available SyncProvider implementations by sending email to jdbc@sun.com. Doing this helps make developers aware of the implementation. To make it possible for a RowSet object to use an implementation, the vendor must register it with the SyncFactory singleton. (See the class comment for SyncProvider for a full explanation of the registration process and the naming convention to be used.)

## Table of Contents

* [1.0 Package Specification](#_2et92p0)
* [2.0 Service Provider Architecture](#tyjcwt)
* [3.0 Implementer's Guide](#3dy6vkm)
* [4.0 Resolving Synchronization Conflicts](#_1t3h5sf)
* [5.0 Related Specifications](#4d34og8)
* [6.0 Related Documentation](#2s8eyo1)

### 1.0 Package Specification

The following classes and interfaces make up the javax.sql.rowset.spi package:

* SyncFactory
* SyncProvider
* SyncFactoryException
* SyncProviderException
* SyncResolver
* XmlReader
* XmlWriter
* TransactionalWriter

The following interfaces, in the javax.sql package, are also part of the SPI:

* RowSetReader
* RowSetWriter

A SyncProvider implementation provides a disconnected RowSet object with the mechanisms for reading data into it and for writing data that has been modified in it back to the underlying data source. A *reader*, a RowSetReader or XMLReader object, reads data into a RowSet object when the CachedRowSet methods execute or populate are called. A *writer*, a RowSetWriter or XMLWriter object, writes changes back to the underlying data source when the CachedRowSet method acceptChanges is called.

The process of writing changes in a RowSet object to its data source is known as *synchronization*. The SyncProvider implementation that a RowSet object is using determines the level of synchronization that the RowSet object's writer uses. The various levels of synchronization are referred to as *grades*.

The lower grades of synchronization are known as *optimistic* concurrency levels because they optimistically assume that there will be no conflicts or very few conflicts. A conflict exists when the same data modified in the RowSet object has also been modified in the data source. Using the optimistic concurrency model means that if there is a conflict, modifications to either the data source or the RowSet object will be lost.

Higher grades of synchronization are called *pessimistic* because they assume that others will be accessing the data source and making modifications. These grades set varying levels of locks to increase the chances that no conflicts occur.

The lowest level of synchronization is simply writing any changes made to the RowSet object to its underlying data source. The writer does nothing to check for conflicts. If there is a conflict and the data source values are overwritten, the changes other parties have made by to the data source are lost.

The RIXMLProvider implementation uses the lowest level of synchronization and just writes RowSet changes to the data source. This is true because typically XML data sources do not enable transaction techniques for maintaining the integrity of data. However, specific standards groups have considered offering XML-based synchronization. For details, see

<http://www.syncml.org>

For the the next level up, the writer checks to see if there are any conflicts, and if there are, it does not write anything to the data source. The problem with this concurrency level is that if another party has modified the corresponding data in the data source since the RowSet object got its data, the changes made to the RowSet object are lost. The RIOptimisticProvider implementation uses this level of synchronization.

At higher levels of synchronization, referred to as pessimistic concurrency, the writer take steps to avoid conflicts by setting locks. Setting locks can vary from setting a lock on a single row to setting a lock on a table or the entire data source. The level of synchronization is therefore a tradeoff between the ability of users to access the data source concurrently and the ability of the writer to keep the data in the RowSet object and its data source synchronized.

It is a requirement that all disconnected RowSet objects (CachedRowSet, FilteredRowSet, JoinRowSet, and WebRowSet objects) obtain their SyncProvider objects from the SyncFactory mechanism.

The reference implementation (RI) provides two synchronization providers.

* **RIOptimisticProvider**  
  The default provider that the SyncFactory instance will supply to a disconnected RowSet object when no provider implementation is specified.  
  This synchronization provider uses an optimistic concurrency model, assuming that there will be few conflicts among users who are accessing the same data in a database. It avoids using locks; rather, it checks to see if there is a conflict before trying to synchronize the RowSet object and the data source. If there is a conflict, it does nothing, meaning that changes to the RowSet object are not persisted to the data source.
* **RIXMLProvider**  
  A synchronization provider that can be used with a WebRowSet object, which is a rowset that can be written in XML format or read from XML format. The RIXMLProvider implementation does no checking at all for conflicts and simply writes any updated data in the WebRowSet object to the underlying data source. WebRowSet objects use this provider when they are dealing with XML data.

These SyncProvider implementations are bundled with the reference implementation, which makes them always available to RowSet implementations. SyncProvider implementations make themselves available by being registered with the SyncFactory singleton. When a RowSet object requests a provider, by specifying it in the constructor or as an argument to the CachedRowSet method setSyncProvider, the SyncFactory singleton checks to see if the requested provider has been registered with it. If it has, the SyncFactory creates an instance of it and passes it to the requesting RowSet object. If the SyncProvider implementation that is specified has not been registered, the SyncFactory singleton causes a SyncFactoryException object to be thrown. If no provider is specified, the SyncFactory singleton will create an instance of the default provider implementation, RIOptimisticProvider, and pass it to the requesting RowSet object.

If a WebRowSet object does not specify a provider in its constructor, the SyncFactory will give it an instance of RIOptimisticProvider. However, the constructor for WebRowSet is implemented to set the provider to the RIXMLProvider, which reads and writes a RowSet object in XML format.

See the [SyncProvider](http://docs.google.com/SyncProvider.html) class specification for further details.

Vendors may develop a SyncProvider implementation with any one of the possible levels of synchronization, thus giving RowSet objects a choice of synchronization mechanisms. A vendor can make its implementation available by registering the fully qualified class name with Sun Microsystems at jdbc@sun.com. This process is discussed in further detail below.

### 2.0 Service Provider Interface Architecture

**2.1 Overview** The Service Provider Interface provides a pluggable mechanism by which SyncProvider implementations can be registered and then generated when required. The lazy reference mechanism employed by the SyncFactory limits unnecessary resource consumption by not creating an instance until it is required by a disconnected RowSet object. The SyncFactory class also provides a standard API to configure logging options and streams that **may** be provided by a particular SyncProvider implementation. **2.2 Registering with the SyncFactory** A third party SyncProvider implementation must be registered with the SyncFactory in order for a disconnected RowSet object to obtain it and thereby use its javax.sql.RowSetReader and javax.sql.RowSetWriter implementations. The following registration mechanisms are available to all SyncProvider implementations:

* + **System properties** - Properties set at the command line. These properties are set at run time and apply system-wide per invocation of the Java application. See the section ["Related Documentation"](#2s8eyo1) further related information.
  + **Property Files** - Properties specified in a standard property file. This can be specified using a System Property or by modifying a standard property file located in the platform run-time. The reference implementation of this technology includes a standard property file than can be edited to add additional SyncProvider objects.
  + **JNDI Context** - Available providers can be registered on a JNDI context. The SyncFactory will attempt to load SyncProvider objects bound to the context and register them with the factory. This context must be supplied to the SyncFactory for the mechanism to function correctly.

Details on how to specify the system properties or properties in a property file and how to configure the JNDI Context are explained in detail in the [SyncFactory](http://docs.google.com/SyncFactory.html) class description.**2.3 SyncFactory Provider Instance Generation Policies**The SyncFactory generates a requested SyncProvider object if the provider has been correctly registered. The following policies are adhered to when either a disconnected RowSet object is instantiated with a specified SyncProvider implementation or is reconfigured at runtime with an alternative SyncProvider object.

* + If a SyncProvider object is specified and the SyncFactory contains *no* reference to the provider, a SyncFactoryException is thrown.
  + If a SyncProvider object is specified and the SyncFactory contains a reference to the provider, the requested provider is supplied.
  + If no SyncProvider object is specified, the reference implementation provider RIOptimisticProvider is supplied.

These policies are explored in more detail in the  [SyncFactory](http://docs.google.com/SyncFactory.html) class.

### 3.0 SyncProvider Implementer's Guide

**3.1 Requirements** A compliant SyncProvider implementation that is fully pluggable into the SyncFactory **must** extend and implement all abstract methods in the [SyncProvider](http://docs.google.com/SyncProvider.html) class. In addition, an implementation **must** determine the grade, locking and updatable view capabilities defined in the SyncProvider class definition. One or more of the SyncProvider description criteria **must** be supported. It is expected that vendor implementations will offer a range of grade, locking, and updatable view capabilities.Furthermore, the SyncProvider naming convention **must** be followed as detailed in the [SyncProvider](http://docs.google.com/SyncProvider.html) class description.**3.2 Grades**JSR 114 defines a set of grades to describe the quality of synchronization a SyncProvider object can offer a disconnected RowSet object. These grades are listed from the lowest quality of service to the highest.

* + - **GRADE\_NONE** - No synchronization with the originating data source is provided. A SyncProvider implementation returning this grade will simply attempt to write any data that has changed in the RowSet object to the underlying data source, overwriting whatever is there. No attempt is made to compare original values with current values to see if there is a conflict. The RIXMLProvider is implemented with this grade.
    - **GRADE\_CHECK\_MODIFIED\_AT\_COMMIT** - A low grade of optimistic synchronization. A SyncProvider implementation returning this grade will check for conflicts in rows that have changed between the last synchronization and the current synchronization under way. Any changes in the originating data source that have been modified will not be reflected in the disconnected RowSet object. If there are no conflicts, changes in the RowSet object will be written to the data source. If there are conflicts, no changes are written. The RIOptimisticProvider implementation uses this grade.
    - **GRADE\_CHECK\_ALL\_AT\_COMMIT** - A high grade of optimistic synchronization. A SyncProvider implementation returning this grade will check all rows, including rows that have not changed in the disconnected RowSet object. In this way, any changes to rows in the underlying data source will be reflected in the disconnected RowSet object when the synchronization finishes successfully.
    - **GRADE\_LOCK\_WHEN\_MODIFIED** - A pessimistic grade of synchronization. SyncProvider implementations returning this grade will lock the row in the originating data source that corresponds to the row being changed in the RowSet object to reduce the possibility of other processes modifying the same data in the data source.
    - **GRADE\_LOCK\_WHEN\_LOADED** - A higher pessimistic synchronization grade. A SyncProvider implementation returning this grade will lock the entire view and/or table affected by the original query used to populate a RowSet object.

**3.3 Locks**JSR 114 defines a set of constants that specify whether any locks have been placed on a RowSet object's underlying data source and, if so, on which constructs the locks are placed. These locks will remain on the data source while the RowSet object is disconnected from the data source.These constants **should** be considered complementary to the grade constants. The default setting for the majority of grade settings requires that no data source locks remain when a RowSet object is disconnected from its data source. The grades GRADE\_LOCK\_WHEN\_MODIFIED and GRADE\_LOCK\_WHEN\_LOADED allow a disconnected RowSet object to have a fine-grained control over the degree of locking.

* + - **DATASOURCE\_NO\_LOCK** - No locks remain on the originating data source. This is the default lock setting for all SyncProvider implementations unless otherwise directed by a RowSet object.
    - **DATASOURCE\_ROW\_LOCK** - A lock is placed on the rows that are touched by the original SQL query used to populate the RowSet object.
    - **DATASOURCE\_TABLE\_LOCK** - A lock is placed on all tables that are touched by the query that was used to populate the RowSet object.
    - **DATASOURCE\_DB\_LOCK** A lock is placed on the entire data source that is used by the RowSet object.

**3.4 Updatable Views**A RowSet object may be populated with data from an SQL VIEW. The following constants indicate whether a SyncProvider object can update data in the table or tables from which the VIEW was derived.

* + - **UPDATABLE\_VIEW\_SYNC** Indicates that a SyncProvider implementation supports synchronization to the table or tables from which the SQL VIEW used to populate a a RowSet object is derived.
    - **NONUPDATABLE\_VIEW\_SYNC** Indicates that a SyncProvider implementation does **not** support synchronization to the table or tables from which the SQL VIEW used to populate a RowSet object is derived.

**3.5 Usage of SyncProvider Grading and Locking**In the example below, the reference CachedRowSetImpl implementation reconfigures its current SyncProvider object by calling the setSyncProvider method. CachedRowSetImpl crs = new CachedRowSetImpl();  
 crs.setSyncProvider("com.foo.bar.HASyncProvider");  
An application can retrieve the SyncProvider object currently in use by a disconnected RowSet object. It can also retrieve the grade of synchronization with which the provider was implemented and the degree of locking currently in use. In addition, an application has the flexibility to set the degree of locking to be used, which can increase the possibilities for successful synchronization. These operation are shown in the following code fragment. SyncProvider sync = crs.getSyncProvider();  
  
 switch (sync.getProviderGrade()) {  
 case: SyncProvider.GRADE\_CHECK\_ALL\_AT\_COMMIT  
 //A high grade of optimistic synchronization  
 break;  
 case: SyncProvider.GRADE\_CHECK\_MODIFIED\_AT\_COMMIT   
 //A low grade of optimistic synchronization   
 break;  
 case: SyncProvider.GRADE\_LOCK\_WHEN\_LOADED   
 // A pessimistic synchronization grade   
 break;  
 case: SyncProvider.GRADE\_LOCK\_WHEN\_MODIFIED   
 // A pessimistic synchronization grade   
 break;  
 case: SyncProvider.GRADE\_NONE   
 // No synchronization with the originating data source provided  
 break;  
 }  
   
 switch (sync.getDataSourcLock() {  
 case: SyncProvider.DATASOURCE\_DB\_LOCK  
 // A lock is placed on the entire datasource that is used by the  
 // RowSet object   
 break;  
  
 case: SyncProvider.DATASOURCE\_NO\_LOCK  
 // No locks remain on the originating data source.  
 break;  
  
 case: SyncProvider.DATASOURCE\_ROW\_LOCK  
 // A lock is placed on the rows that are touched by the original   
 // SQL statement used to populate  
 // the RowSet object that is using the SyncProvider  
 break;  
  
 case: DATASOURCE\_TABLE\_LOCK  
 // A lock is placed on all tables that are touched by the original   
 // SQL statement used to populated  
 // the RowSet object that is using the SyncProvider  
 break;  
  
It is also possible using the static utility method in the SyncFactory class to determine the list of SyncProvider implementations currently registered with the SyncFactory. Enumeration e = SyncFactory.getRegisteredProviders();  
4.0 Resolving Synchronization Conflicts The interface SyncResolver provides a way for an application to decide manually what to do when a conflict occurs. When the CachedRowSet method acceptChanges finishes and has detected one or more conflicts, it throws a SyncProviderException object. An application can catch the exception and have it retrieve a SyncResolver object by calling the method SyncProviderException.getSyncResolver(). A SyncResolver object, which is a special kind of CachedRowSet object or a JdbcRowSet object that has implemented the SyncResolver interface, examines the conflicts row by row. It is a duplicate of the RowSet object being synchronized except that it contains only the data from the data source this is causing a conflict. All of the other column values are set to null. To navigate from one conflict value to another, a SyncResolver object provides the methods nextConflict and previousConflict. The SyncResolver interface also provides methods for doing the following:

### finding out whether the conflict involved an update, a delete, or an insert

* + getting the value in the data source that caused the conflict
  + setting the value that should be in the data source if it needs to be changed or setting the value that should be in the RowSet object if it needs to be changed

When the CachedRowSet method acceptChanges is called, it delegates to the RowSet object's SyncProvider object. How the writer provided by that SyncProvider object is implemented determines what level (grade) of checking for conflicts will be done. After all checking for conflicts is completed and one or more conflicts has been found, the method acceptChanges throws a SyncProviderException object. The application can catch the exception and use it to obtain a SyncResolver object. The application can then use SyncResolver methods to get information about each conflict and decide what to do. If the application logic or the user decides that a value in the RowSet object should be the one to persist, the application or user can overwrite the data source value with it. The comment for the SyncResolver interface has more detail. 5.0 Related Specifications

### [JNDI 1.3](http://java.sun.com/products/jndi)

* + [Java Logging APIs](http://docs.google.com/technotes/guides/logging/index.html)

6.0 Related Documentation

### [System properties](http://docs.google.com/technotes/tools/index.html#basic)

* + Resource Files
  + [DataSource for JDBC Connections](http://java.sun.com/tutorial/jdbc)

| | [**Overview**](http://docs.google.com/overview-summary.html) | **Package** | Class | [**Use**](http://docs.google.com/package-use.html) | [**Tree**](http://docs.google.com/package-tree.html) | [**Deprecated**](http://docs.google.com/deprecated-list.html) | [**Index**](http://docs.google.com/index-files/index-1.html) | [**Help**](http://docs.google.com/help-doc.html) | | --- | --- | --- | --- | --- | --- | --- | --- | | | ***Java™ Platform*** ***Standard Ed. 6*** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [**PREV PACKAGE**](http://docs.google.com/javax/sql/rowset/serial/package-summary.html)   [**NEXT PACKAGE**](http://docs.google.com/javax/swing/package-summary.html) | [**FRAMES**](http://docs.google.com/index.html?javax/sql/rowset/spi/package-summary.html)    [**NO FRAMES**](http://docs.google.com/package-summary.html)     [**All Classes**](http://docs.google.com/allclasses-noframe.html) |

[Submit a bug or feature](http://bugs.sun.com/services/bugreport/index.jsp)  
For further API reference and developer documentation, see [Java SE Developer Documentation](http://docs.google.com/webnotes/devdocs-vs-specs.html). That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.Copyright 2006 Sun Microsystems, Inc. All rights reserved. Use is subject to [license terms](http://docs.google.com/legal/license.html). Also see the [documentation redistribution policy](http://java.sun.com/docs/redist.html).