# *Oracle Berkeley DB*

# *Getting Started with Distributed Applications for Java*

## *12c Release 1*

**Library Version 12.1.6.2**

## Legal Notice



*Published 3/28/2016*

# Table of Contents

# Preface

This document describes how to write client-server applications for Berkeley DB 12*c* Release 1 (library version 12.1.6.2). This release includes a Java-based server program that provides accesses to Berkeley DB APIs to remote client applications. Client applications communicate with the server through driver APIs.

The steps to start the server and the driver APIs used to implement client applications are described here. This book describes the concepts surrounding distributed Berkeley DB applications, the scenarios under which you might choose to use it, and the requirements of using the server program and the driver APIs.

This book is aimed at the software engineer responsible for writing a distributed DB application.

This book assumes that you have already read and understood the concepts contained in the *Berkeley DB Getting Started with Transaction Processing* guide.

## Conventions Used in this Book

The following typographical conventions are used within in this manual:

Class names are represented in `monospaced font`, as are `method names`. For example: "The `Environment()` constructor returns an `Environment` class object."

Variable or non-literal text is presented in *italics*. For example: "Go to your *DB_INSTALL* directory."

Program examples are displayed in a `monospaced font` on a shaded background. For example:

```
import com.sleepycat.client.SDatabaseConfig;

...

// Allow the database to be created.
SDatabaseConfig myDbConfig = new SDatabaseConfig();
myDbConfig.setAllowCreate(true);
```

In some situations, programming examples are updated from one chapter to the next. When this occurs, the new code is presented in **`monospaced bold`** font. For example:

```
import com.sleepycat.client.SDatabase;
import com.sleepycat.client.SDatabaseConfig;

...

// Allow the database to be created.
SDatabaseConfig myDbConfig = new SDatabaseConfig();
myDbConfig.setAllowCreate(true);
SDatabase myDb = env.openDatabase(null, "mydb.db", null, myDbConfig);
```

### Note

Finally, notes of special interest are represented using a note block such as this.

## For More Information

Beyond this manual, you may also find the following sources of information useful when building a transactional DB application:

- Getting Started with Transaction Processing for Java
- Getting Started with Berkeley DB for Java
- Berkeley DB Collections Tutorial
- Berkeley DB Programmer's Reference Guide
- Berkeley DB Javadoc

To download the latest Berkeley DB documentation along with white papers and other collateral, visit http://www.oracle.com/technetwork/indexes/documentation/index.html.

For the latest version of the Oracle Berkeley DB downloads, visit http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/downloads/index.html.

### Contact Us

You can post your comments and questions at the Oracle Technology (OTN) forum for Oracle Berkeley DB at: https://forums.oracle.com/forums/forum.jspa?forumID=271, or for Oracle Berkeley DB High Availability at: https://forums.oracle.com/forums/forum.jspa?forumID=272.

For sales or support information, email to: berkeleydb-info_us@oracle.com You can subscribe to a low-volume email announcement list for the Berkeley DB product family by sending email to: bdb-join@oss.oracle.com

# Chapter 1. Introduction to Building Distributed Applications with Berkeley DB

In addition to being an embedded database, DB also supports the client-server architecture by providing a stand-alone server program and client driver APIs. The server program offers remote access to DB features. The client driver APIs provide building blocks for applications that communicate with a database server. Multiple client applications can communicate with a single server simultaneously.

This book provides a thorough introduction and discussion on how to build distributed applications with Berkeley DB (DB). It describes the features supported by the server, and how to configure and start the server program. Finally, it describes the driver APIs that you use to implement client applications.

You should understand the concepts from the *Berkeley DB Getting Started with Transaction Processing* guide before reading this book.

## Berkeley DB Server

The server is capable of managing multiple DB environments and databases on behalf of client applications. All environments and databases managed by a server are protected by transactions. Non-transactional environments or databases are not supported by the server. The server can serve multiple client applications simultaneously.

Not all features offered by the DB library are supported by the server. For example, features that require callback functions are not supported. An exception to this is secondary databases, where a callback function is needed to create the set of secondary keys corresponding to a given primary key and data pair.

To avoid leaking DB handles, the server can be configured to close long-inactive handles automatically. This is useful when client applications may disconnect unexpectedly. For example, a client application may open a transactional cursor, update a few records using the cursor and then disconnect unexpectedly (e.g. the client application crashes). Without the ability to close (e.g. abort) the transaction handle by the server, the transaction will be open forever, and the locks held by the transaction will never be released. This would block other client applications that require any of the locks.

### Supported Features

The following high-level features are supported by Berkeley DB server:

- B-Tree, Hash and Recno access methods
- Transaction
- In-memory databases
- Secondary databases, foreign constraints and join cursors
- Sequences

- Environment and database statistics
- Close long-inactive handles manually and automatically
- SSL over TCP connections

# Client Driver APIs

The client driver APIs are modeled after Berkeley DB Java base APIs with most class names prefixed with an 'S', which stands for server. For example, the `Database` class is named the `SDatabase` class in the driver APIs. All classes in the client driver APIs are located in the `com.sleepycat.client` package.

In addition, the client driver provides the `BdbServerConnection` class, which can be used to establish a connection to a server and also serves as the starting point to everything else in the client driver APIs. For example, client environment handles are created like this:

```
BdbServerConnection conn =
  BdbServerConnection.connect("localhost", 8080);
SEnvironment env = conn.openEnvironment("env",
  new SEnvironmentConfig().setAllowCreate(true));
```

The `BdbServerConnection` class is not thread-safe. Furthermore, all objects created from a `BdbServerConnection` instance use the connection object internally. Thus, they must be accessed in the same thread as the connection object, or at least their accesses must be serialized. This rule applies recursively to handles created from environment handles (which are created from connection objects). Therefore, the entire object graph rooted at a single `BdbServerConnection` instance must be accessed in a single thread, or accessed serially.

# Secure Connections

Secure Socket Layer (SSL) can be enabled to secure communications between clients and servers. Both uni- and bi-directional authentications are supported.

Java key stores are used to manage private and public keys for SSL authentication. Depending on whether uni- or bi-directional authentication is used, key stores and/or trust stores must be configured properly on both client and server machines.

# Chapter 2. Berkeley DB Server

In this chapter, we describe the Berkeley DB Server - the standalone server program. It begins by explaining how to start and stop the server. Also, how to enable SSL. Then it describes the configuration files used by the server. Finally, it lists the constraints of Berkeley DB utility programs when used on environments and databases managed by the server.

## System Requirements

The server program requires JDK 8 or above, slf4j 1.5.8, log4j 1.2.14, Apache Thrift's Java library 0.9.2, and Berkeley DB Java library. All libraries are required at both compile and run time.

### Note

The 3rd party libraries: slf4j, log4j and Apache Thrift are bundled in the release package. You do not need to install them separately.

## Starting the Server

On UNIX/POSIX systems, a shell script `start_server.sh` is installed under the `bin` directory if `--enable-server` is specified to the build configuration. This script sets up the proper classpath and JVM options, and starts the server program.

On Windows systems, no batch script is provided because the place of the `db.jar` depends on the build configuration. To start a server, you need to do the following:

1. Copy `bdb.properties` and `log4j.xml` from `lang/thrift/server` to your current directory.
2. Include the `db.jar` and all jar files under `lang/thrift/jars` to the classpath.
3. Set the `java.library.path` JVM option to point to the directory where native Berkeley DB libraries are placed.
4. Run the `com.sleepycat.server.BdbServer` program.

For example, if the native libraries and `db.jar` are placed under `build_windows/Win32/Release`, a batch script placed under it would look like:

```
set THRIFT_HOME=..\..\..\lang\thrift

copy %THRIFT_HOME%\server\bdb.properties .
copy %THRIFT_HOME%\server\log4j.xml .

set CLASSPATH=.;%THRIFT_HOME%\jars\slf4j-api.jar
set CLASSPATH=%CLASSPATH%;%THRIFT_HOME%\jars\slf4j-log4j12.jar
set CLASSPATH=%CLASSPATH%;%THRIFT_HOME%\jars\log4j.jar
```

```
set CLASSPATH=%CLASSPATH%;%THRIFT_HOME%\jars\libthrift.jar
set CLASSPATH=%CLASSPATH%;%THRIFT_HOME%\jars\db_thrift_interface.jar
set CLASSPATH=%CLASSPATH%;%THRIFT_HOME%\jars\db_thrift_server.jar
set CLASSPATH=%CLASSPATH%;db.jar

java -Djava.library.path=. com.sleepycat.server.BdbServer
```

The `BdbServer` supports the following options:

- -v

  Print the version of the server program.

- -h

  Print the help information of the server program.

- -config-file

  Specify the configuration file for the server program.

- -log-config

  Specify the log4j XML configuration file for the server program.

# Stopping the server

To stop a server, use `Ctrl-C` from within the shell where the server is running.

# Enabling SSL for the Server

To enable SSL for a server, you need to setup the appropriate Java key store and/or trust store files and then configure the server to use these key stores.

If you want to authenticate the server so that clients know that they are connecting to the correct server, a key store with the server's private key must be setup on the server. For example, the following command creates a key store *keystore.jks* containing a generated private/public key pair.

```
keytool -genkeypair -alias certificatekey -keyalg RSA \
-validity 7 -keystore keystore.jks
```

If you want to authenticate clients, a trust store with trusted clients' public keys must be setup on the server. For more information, see Connecting to a Server with SSL (page 13).

Once the key store and/or trust store are setup, you should list them in the server configuration file. For example:

```
ssl.host=localhost
```

```
# Configure the key store for SSL.

ssl.keyStore=keystore.jks
ssl.keyStore.password=<password>

# Configure the trust store for SSL.

#ssl.trustStore=truststore.jks
#ssl.trustStore.password=<password>
```

For more information on the server configuration file, see Server Configuration File (page 6).

# Configuring the Server

Berkeley DB server uses two configuration files, one for log4j, and the other for the server itself. After a change to a configuration file, the server must be restarted to pick up the change.

## log4j Configuration File

Berkeley DB server uses log4j, and thus requires a log4j configuration file. Berkeley DB server accepts configuration files in XML only. By default, it looks for a `log4j.xml` file under the current working directory. This can be overridden by the `-log-config` command-line argument when the server is started.

An example log4j configuration file for a Berkeley DB server:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">

<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
    <appender name="File" class="org.apache.log4j.RollingFileAppender">
        <param name="File" value="path-to-log-file"/>
        <param name="MaxBackupIndex" value="20"/>
        <param name="MaxFileSize" value="50MB"/>
        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern"
                value="%d [%t] %-5p %c - %m%n"/>
        </layout>
    </appender>
    <appender name="Async" class="org.apache.log4j.AsyncAppender">
        <param name="BufferSize" value="256"/>
        <appender-ref ref="File" />
    </appender>
    <root>
        <level value="INFO"/>
        <appender-ref ref="Async"/>
    </root>
</log4j:configuration>
```

## Server Configuration File

Berkeley DB server also uses a Java properties file to control its own behavior. By default, the server looks for a `bdb.properties` file under the current working directory. This can be overridden by the `-config-file` command-line argument when the server is started.

### Note

Because this file may contain sensitive information, its permissions should be limited to the minimum. If the file is accessible globally (globally readable, writable or executable), the server would not start.

An example configuration file for a Berkeley DB server:

```
###################################
# Server configuration
###################################

# The server's listening port

port=8080

# If SSL is enabled, this indicates the maximum number of concurrent
# client connections.
#
# If SSL is disabled, there is no limit on the number of concurrent
# connections, and this parameter indicates the maximum number of
# requests that can be processed concurrently.

workers=10

# Configure the host name of this server. SSL is enabled if ssl.host
# is set.

#ssl.host=server-host-name

# Configure the key store for SSL.

#ssl.keyStore=path-to-key-store
#ssl.keyStore.password=key-store-password
#ssl.keyStore.type=jks
#ssl.keyStore.manager=SunX509

# Configure the trust store for SSL.

#ssl.trustStore=path-to-trust-store
#ssl.trustStore.password=trust-store-password
#ssl.trustStore.type=jks
#ssl.trustStore.manager=PKIX
```

```
###################################
# Database service configurations
###################################

# The root directory for all environments, the home directory for an
# environment with name "<env>" is $root.home/<env>

root.home=path-to-environment-root-directory

# The root directory for all data directories, the data directory
# for an environment with name "<env>" is $root.data/<env>

root.data=path-to-data-root-directory

# The root directory for all blob directories, the blob directory
# for an environment with name "<env>" is $root.blob/<env>

root.blob=path-to-blob-root-directory

# The root directory for all log directories, the log directory for
# an environment with name "<env>" is $root.log/<env>

root.log=path-to-db-log-root-directory

# Configure the timeout value for open handles. The timeout value
# is set in seconds.
#
# For example, if the timeout is set to 600 seconds (10 minutes),
# and a handle has not been accessed for longer than 600 seconds,
# then this handle will be closed automatically when the cleanup
# worker runs the next time.

handle.timeout=600

# The frequency the cleanup worker runs. The frequency is set in
# seconds.
#
# For example, if the frequency is set to 300 seconds, the
# cleanup worker runs every 300 seconds.

cleanup.interval=300
```

### Server Configuration Options

The server can be configured with the following options:

- **port**

  Specifies the port number the server listens for client connections. The default value is 8080.

- **workers**

  Specifies the number of worker threads used for handling client requests. The default value is 20.

  If SSL is not enabled, client requests are handled by a shared pool of worker threads. A free worker thread is chosen to handle each client request, and the worker thread becomes free again after handling the request. The number of worker threads determines the maximum number of client requests that can be handled concurrently. If the number of concurrent requests exceeds the number of worker threads, later requests will be blocked until a free worker thread is available.

  ## Note

  In this mode, the number of concurrent client connections is limited only by system resources. For example, the server can serve 200 concurrent client connections even if the *workers* is set to 20.

  If SSL is enabled, each client is handled by a dedicated worker thread. Therefore, the number of worker threads determines the maximum number of concurrent client connections. For example, if *workers* is set to 100, the server can serve a maximum of 100 concurrent client connections. If more clients try to connect to the server, they are blocked until some client disconnects from the server or the connection times out (the timeout is specified by the user).

- **ssl.host**

  SSL is enabled if this property is specified. This property specifies the host name of the server. Clients should use this name when connecting to the server. There is no default value, and SSL is disabled by default.

- **ssl.keyStore**

  Specifies the path to the Java key store file which manages the server's private key. There is no default value.

  ## Note

  This file maintains sensitive information. If the file is accessible globally (globally readable, writable or executable), the server would not start.

- **ssl.keyStore.password**

  Specifies the password of the key store file specified in *ssl.keyStore*. There is no default value.

- **ssl.keyStore.type**

  Specifies the type of the key store file specified in *ssl.keyStore*. There is no default value.

- **ssl.keyStore.manager**

Specifies the algorithm name of the key manager factory. There is no default value.

- **ssl.trustStore**

  Specifies the path to the Java trust store file which manages the server's trusted public certificates. There is no default value.

  ## Note

  This file maintains sensitive information. If the file is accessible globally (globally readable, writable or executable), the server would not start.

- **ssl.trustStore.password**

  Specifies the password of the trust store file specified in *ssl.trustStore*. There is no default value.

- **ssl.trustStore.type**

  Specifies the type of the trust store file specified in *ssl.trustStore*. There is no default value.

- **ssl.trustStore.manager**

  Specifies the algorithm name of the trust manager factory. There is no default value.

- **root.home**

  The root directory for all environment's home directory. For example, if the environment home specified by a client is *env_home*, the actual home directory on the server is <*root.home*>/*env_home*. The default value is the current working directory.

  ## Note

  The environment home specified by a client cannot escape the specified root directory. Absolute paths or relative paths that escape the root directory are not allowed.

- **root.data**

  The root directory for all environment's data directories. For example, if the environment home specified by a client is *env_home*, the actual directory for access method database files of this environment is <*root.data*>/*env_home*.

  If this property is not specified, *root.home* is used.

- **root.blob**

  The root directory for all environment's blob directories. For example, if the environment home specified by a client is *env_home*, the actual directory for blob files of this environment is <*root.blob*>/*env_home*.

If this property is not specified, *root.home* is used.

- **root.log**

  The root directory for all environment's log directories. For example, if the environment home specified by a client is *env_home*, the actual directory for log files of this environment is <*root.log*>/*env_home*.

  If this property is not specified, *root.home* is used.

- **handle.timeout**

  Specifies the timeout for open handles on the server, in seconds. The server scans all open handles periodically, and closes those handles that have not been accessed for the last *handle.timeout* seconds. The default value is 600.

- **cleanup.interval**

  Specifies the interval between consecutive scans of open handles, in seconds. The default value is 300.

# Impacts on Utility Programs

All Berkeley DB utility programs can be run on environments and databases managed by Berkeley DB servers. For example, you can use `db_hotbackup` to backup databases, or `db_stat` to get database statistics.

The following constraints apply when running utility programs on managed environments and databases that are accessed simultaneously by other processes:

## Note

Constraints do not apply if the accessed environment or database has no client handles opened or if the server is offline.

- **db_archive**

  No constraint applies.

  ## Note

  If *root.data* and/or *root.log* are different from *root.home*, you need a `DB_CONFIG` file which sets the proper paths using the `add_data_dir` , or `set_lg_dir` configuration parameters.

- **db_dump**

  Options `-d, -R` and `-r` cannot be used on databases actively accessed by clients.

- **db_load**

  `db_load` cannot be used on databases actively accessed by clients.

### Note

If *root.blob* is different from *root.home*, you need to set the proper blob path using the `-b` option.

- **db_log_verify**

  No constraint applies.

### Note

If *root.log* is different from *root.log*, you need a `DB_CONFIG` file which sets the proper path using the `set_lg_dir` configuration parameter.

- **db_printlog**

  No constraint applies.

### Note

If *root.log* is different from *root.log*, you need a `DB_CONFIG` file which sets the proper path using the `set_lg_dir` configuration parameter.

- **db_recover**

  `db_recover` cannot be used on environments actively accessed by clients.

  `db_recover` must have exclusive access to the environment being recovered. An alternative way to perform normal recovery of an environment is to open the environment with the `DB_RECOVER` flag from a client application.

- **db_replicate**

  `db_replicate` cannot be used on databases managed by Berkeley DB servers.

### Note

Replication is not supported by Berkeley DB servers. Therefore, even when the server is offline, `db_replicate` still cannot be used on databases managed by Berkeley DB server.

- **db_sql_codegen**

  `db_sql_codegen` does not access environments or databases, and is irrelevant.

- **dbsql**

  `dbsql` cannot be used on databases managed by Berkeley DB servers because they are not created with the SQL APIs.

- **db_stat**

  Options `-r` and `-R` cannot be used on databases managed by Berkeley DB servers because replication is not supported.

- **db_upgrade**

  `db_upgrade` cannot be used on databases actively accessed by clients.

- **db_verify**

  `db_verify` cannot be used on databases actively accessed by clients.

  ### Note

  If *root.blob* is different from *root.home*, you need to set the proper blob path using the `-b` option.

- **sqlite3**

  `sqlite3` cannot be used on databases managed by Berkeley DB servers.

# Secondary Databases

To manage secondary keys created for secondary databases, an auxiliary database is maintained on the server for each secondary database. If the secondary database is file based, the auxiliary database is also file based. If the secondary database is in-memory, the auxiliary database is also in-memory.

For file-based auxiliary databases, its file name is constructed by appending `"__aux.db"` to the file name of its corresponding secondary database. For example, if the file name of a secondary database is `"secondary"`, the file name of its corresponding auxiliary database is `"secondary__aux.db"`.

Make sure that the auxiliary database is in sync with its corresponding secondary databases. Otherwise, operations on the secondary database may fail. For example, when backing up or restoring databases, the secondary databases and their corresponding auxiliary databases must be backed up or restored together. When restoring databases, the databases must be restored together.

# Chapter 3. Berkeley DB Client Driver APIs

In this chapter, we describe in detail all the aspects of Berkeley DB client driver APIs. The client driver APIs are modeled after Java base APIs. Therefore a good understanding of *Berkeley DB Getting Started with Transaction Processing* is necessary.

This chapter begins by describing how to connect to a server. It then follows with a detailed list of differences between the client driver APIs and Java base APIs.

## System Requirements

The client driver requires JDK 8 or above, slf4j 1.5.8, log4j 1.2.14, and Apache Thrift's Java library 0.9.2. All libraries are required at both compile and run time.

### Note

The 3rd party libraries: slf4j, log4j and Apache Thrift are bundled in the release package. You do not need to install them separately.

For Java applications, you should add the following jar files under the `lang/thrift/jars` directory to your applications classpath:

- `log4j.jar`
- `slf4j-api.jar`
- `slf4j-log4j12.jar`
- `libthrift.jar`
- `db_thrift_interface.jar`
- `db_thrift_client.jar`

## Connecting to a Server without SSL

To establish a connection to a server, you must use one of the `connect` methods of the `BdbServerConnection` class. To connect to a server without SSL, use `BdbServerConnection.connect` and specify the host name and port of the server:

```
BdbServerConnection conn =
     BdbServerConnection.connect("localhost", 8080);
```

## Connecting to a Server with SSL

To connect to a server with SSL, you need to setup the trust store files and configure the client to use them through `SslConfig`.

To continue from the example of authenticating the server found in Enabling SSL for the Server (page 4), to setup the trust store files and connect to a server with SSL, do the following:

1. Export the certificate containing the server's public key from the key store on the server using the following command:

```
keytool -export -alias certificatekey \
-keystore keystore.jks -rfc -file cert.cer
```

2. Create a trust store on the client machine and import the certificate to it using the following command:

```
keytool -import -alias certificatekey \
-file cert.cer -keystore truststore.jks
```

3. Then, connect to the server by using `BdbServerConnection.connectSsl` and specifying the host name, port and a `SslConfig`:

```
SslConfig sslConfig = new SslConfig()
.setTrustStore("truststore.jks", "password");
BdbServerConnection conn =
     BdbServerConnection.connectSsl("localhost", 8080, sslConfig);
```

# Administrative Functions

Several functions are added to the client driver APIs to control the server, to clean up open handles and to delete unused environments. These functions do not rely on open handles and are put together into the `BdbServerAdmin` class.

## Shutdown the Server

You can shutdown the server using the `shutdownServer` method:

```
BdbServerConnection conn =
     BdbServerConnection.connect("localhost", 8080);

// Get an instance of the BdbServerAdmin class
BdbServerAdmin adminService = conn.adminService();

// Shutdown the server. After this method, close the connection.
adminService.shutdownServer();

// Close the connection.
conn.close();
```

## Using ping

You can test the server's reachability using the `ping` method:

```
BdbServerConnection conn =
     BdbServerConnection.connect("localhost", 8080);

// Get an instance of the BdbServerAdmin class
BdbServerAdmin adminService = conn.adminService();

// Test if the server is reachable. If the server is not
```

```
// reachable, an SDatabaseException is thrown.
adminService.ping();

...
```

## Getting the server version

You can get the version of Berkeley DB library used by the server using the `getServerBdbVersion` method:

```
BdbServerConnection conn =
     BdbServerConnection.connect("localhost", 8080);

// Get an instance of the BdbServerAdmin class
BdbServerAdmin adminService = conn.adminService();

// Get the version of the Berkeley DB library used by the server.
String serverBdbVersion = adminService.getServerBdbVersion();

...
```

## Cleaning up Inactive Handles

Sometimes a handle may be left open for a long time on the server. This can be caused by a client application error, crash or hang. If this handle holds critical resources (e.g locks), it may slow down or even block other client applications.

The Berkeley DB server can be configured to close these inactive handles periodically with the `cleanup.interval` and `handle.timeout` properties. For more information, see Server Configuration Options (page 7)

In addition, the driver APIs also provide functions to clean up inactive environment and database handles. For example:

### Note

When an environment or database handle is closed, all handles that depend on it are also closed. For example, when an environment handle is closed, all transaction handles opened under the environment handles are also closed.

```
BdbServerConnection conn =
     BdbServerConnection.connect("localhost", 8080);

// Get an instance of the BdbServerAdmin class
BdbServerAdmin adminService = conn.adminService();

// Close handles opened on the environment whose home directory
// is "env", if they have not been accessed in the last 10 seconds.
adminService.closeEnvironmentHandles("env", 10 * 1000L);

// Close handles opened on the database specified with the given
```

```
// environment home directory, database file name and sub-database
// name, if they have not been accessed in the last 10 seconds.
adminService.closeDatabaseHandles("home", "dbFile", "subDb", 10 * 1000L);

// Shutdown the server. After this method, close the connection.
adminService.shutdownServer();

// Close the connection.
conn.close();
```

### Deleting an Environment and Its Databases

With the embedded Berkeley DB, you can remove an environment and all its databases by removing the environment home directory and any additional data, log and blob directories. The driver APIs also provide a way to do this:

```
BdbServerConnection conn =
     BdbServerConnection.connect("localhost", 8080);

// Get an instance of the BdbServerAdmin class
BdbServerAdmin adminService = conn.adminService();

// Delete the environment whose home directory is "env" and all the
// databases opened in it. Close all open handles on the environment
// and databases.
adminService.deleteEnvironmentAndDatabases("env", true);

// Shutdown the server. After this method, close the connection.
adminService.shutdownServer();

// Close the connection.
conn.close();
```

## Removing and Renaming Objects

Operations that remove or rename an environment, a database or a sequence object require care. First, operations on the same object must be serialized. For example, you cannot rename and remove a database simultaneously. Second, they cannot be performed if there are still open handles on the object. To solve the first issue, operations on the same objects are serialized explicitly by the server program. To solve the second issue, the driver APIs for these operations behave slightly differently from the Java base APIs.

In the driver APIs, each of these functions has a boolean argument `force`. If this argument is set to `true`, all handles opened on the object to be removed or renamed are automatically closed by the server, so the operation can proceed. If this argument is set to `false`, the operation is executed only if there is no open handle on the object; otherwise, the function throws a `SResourceInUseException` exception.

For example, if you want to remove a database even if someone is still using it, you can call `SEnvironment.removeDatabase` with the `force` parameter set to `true`:

```
BdbServerConnection conn = ...;
SEnvironment env = conn.openEnvironment("env", null);

// Force the database to be deleted.
env.removeDatabase(null, "db", null, true);
```

If you want to remove a database, but only when no one is using it, you can call the same function with the `force` parameter set to `false`:

```
BdbServerConnection conn = ...;
SEnvironment env = conn.openEnvironment("env", null);

try {
    // Try to remove the database if no one is using it.
    env.removeDatabase(null, "db", null, false);
    // The database is removed successfully.
} catch (SResourceInUseException e) {
    // Someone is still using the database and the database
    // is not removed
}
```

# Key/Data Pairs and Bulk Operations

Bulk operations in the driver APIs are quite different from the Java base APIs. In Java base APIs, multiple entries are encoded into a single byte array. In the driver APIs, they are simply maintained as a list of entries. As a consequence, the `MultipleEntry` class has no counterpart in the driver APIs, and `SMultipleDataEntry` , `SMultipleKeyDataEntry` and `SMultipleRecnoDataEntry` are no longer subclasses of `SDatabaseEntry`. To support functions that accept any type of data entries, `SDatabaseEntryBase` is added as the base interface of all data entry classes, and `SMultiplePairs` is added as the base class of `SMultipleKeyDataEntry` and `SMultipleRecnoDataEntry`.

In driver APIs, the amount of data to be returned in a single bulk retrieval is not limited by a buffer size, because `SMultipleDataEntry` , `SMultipleKeyDataEntry` or `SMultipleRecnoDataEntry` does not use a byte array buffer. Instead, the number of entries to be returned is specified with the `setBatchSize` method:

```
SEnvironment env = ...

// Open the database with the default configuration.
SDatabase db = env.openDatabase(null, "db", null, null);

// Create a SMultipleDataEntry for bulk retrieval.
SMultipleDataEntry dataEntries = new SMultipleDataEntry();

// Return at maximum of 10 data items from a single retrieval.
dataEntries.setBatchSize(10);

// Perform a bulk retrieval to return multiple data items
// for the same duplicated key.
```

```
db.get(null, new SDatabaseEntry("key".getBytes()), dataEntries, null);

// Iterate over the retrieved items.
SDatabaseEntry item = new SDatabaseEntry();
while (dataEntries.next(item)) {
    // Do something for each item.
}
```

## Statistic Functions

All statistic functions in the Java base APIs are supported in the driver APIs. In addition, a function that returns multiple environment statistics at once is provided to save a few client-server round trips when multiple statistics are needed:

```
SEnvironment env = ...

SMultiStatsConfig config = new SMultiStatsConfig();

// Include cache file statistics in the result.
config.setCacheFileConfig(true, null);

// Include log statistics in the result and reset it.
config.setLogConfig(true, new SStatsConfig().setClear(true));

// Retrieve both statistics.
SMultiStats multiStats = env.getMultipleStats(config);

// Get cache file statistics.
SCacheFileStats[] cacheFileStats = multiStats.getCacheFileStats();

// Get log statistics.
SLogStats logStats = multiStats.getLogStats();
```

## Secondary Databases

The secondary-key callback configured for a secondary database is called at the client node only. To make it work, the client driver tries to maintain enough information for the callback. However, certain operations are still not supported by the client driver.

- Partial update is not supported on databases that have associated secondary databases.
- `Cursor.putAfter()`, `Cursor.putBefore()` or `Cursor.putCurrent()` cannot be called immediately after a multiple-key get operation on the same cursor. A multiple-key get operation is a get operation where the data parameter is an instance of `SMultiplePairs`.

## Multi-Threaded Applications

The `BdbServerConnection` class is not thread-safe. Furthermore, objects created from a `BdbServerConnection` instance use the connection object internally for remote invocations. Therefore, they are not thread-safe either. The connection object together with all objects

created from it must be used in a single thread, because they all share the same connection object. For example, if we create a `BdbServerAdmin` object, a `SEnvironment` object from a `BdbServerConnection` object, a `SDatabase` object, and a `STransaction` object from the `SEnvironment` object, all five objects must be accessed in a single thread.

For a multi-threaded application, you can create multiple instances of `BdbServerConnection` each in a separate thread. Then, if necessary, you can open the same environment or database from multiple connections to access the same environment or database concurrently.