Derby Security Guide

Version 10.16

Derby Document build:
May 18, 2022, 1:44:21 PM (PDT)

Contents

Copyright....4

License....5

About this guide....9
- **Purpose of this guide**....9
- **Audience**....9
- **How this guide is organized**....9

Part One: Introduction to database security....11
- **Why databases need security**....11
  - Vulnerabilities of unsecured databases....11
  - Threats to unsecured databases....11
- **Defenses against security threats**....12
  - Derby defenses against threats....12
  - Defenses outside of Derby....13
- **Defenses mapped to threats**....13
- **Designing safer Derby applications**....14
- **Security terminology**....15

Part Two: Configuring security for Derby....16
- **Basic security configuration tasks**....16
  - Configuring security in an embedded environment....17
  - Configuring security in a client/server environment....17
- **Configuring database encryption**....19
  - Requirements for Derby encryption....20
  - Working with encryption....20
- **Using signed jar files**....26
- **Configuring SSL/TLS**....26
  - Creating a client key pair and certificate....27
  - Creating a server key pair and certificate....28
  - Importing certificates....28
  - Booting the server and connecting to it....29
  - Key and certificate handling....30
  - Starting the server with SSL/TLS....31
  - Running the client with SSL/TLS....31
  - Other server commands....32
- **Understanding identity in Derby**....33
  - Users and authorization identifiers....33
  - Database Owner....34
- **Configuring user authentication**....35
  - Configuring LDAP authentication....36
  - Configuring NATIVE authentication....39
  - Specifying authentication with a user-defined class....42
  - List of user authentication properties....44
  - Programming applications for Derby user authentication....45
  - Login failure exceptions with user authentication....46
  - Configuring Network Server authentication in special circumstances....46
- **Configuring user authorization**....48
  - Configuring coarse-grained user authorization....48
  - Configuring fine-grained user authorization....51
- **Restricting file permissions**....69
- **Putting it all together**....70
  - Starting a secured Network Server....70

Creating and using a secure database..... 70
Stopping the secured Network Server..... 71

Trademarks..... 73

Copyright

Related information

License

License

The Apache License, Version 2.0

```
      that are managed by, or on behalf of, the Licensor for the
      purpose of discussing and improving the Work, but excluding
      communication that is conspicuously marked or otherwise
      designated in writing by the copyright owner as "Not a
      Contribution."

      "Contributor" shall mean Licensor and any individual or Legal
      Entity on behalf of whom a Contribution has been received by
      Licensor and subsequently incorporated within the Work.

   2. Grant of Copyright License. Subject to the terms and conditions
      of this License, each Contributor hereby grants to You a
      perpetual, worldwide, non-exclusive, no-charge, royalty-free,
      irrevocable copyright license to reproduce, prepare Derivative
      Works of, publicly display, publicly perform, sublicense, and
      distribute the Work and such Derivative Works in Source or
      Object form.

   3. Grant of Patent License. Subject to the terms and conditions of
      this License, each Contributor hereby grants to You a perpetual,
      worldwide, non-exclusive, no-charge, royalty-free, irrevocable
      (except as stated in this section) patent license to make, have
      made, use, offer to sell, sell, import, and otherwise transfer
      the Work, where such license applies only to those patent claims
      licensable by such Contributor that are necessarily infringed by
      their Contribution(s) alone or by combination of their
      Contribution(s) with the Work to which such Contribution(s) was
      submitted. If You institute patent litigation against any entity
      (including a cross-claim or counterclaim in a lawsuit) alleging
      that the Work or a Contribution incorporated within the Work
      constitutes direct or contributory patent infringement, then any
      patent licenses granted to You under this License for that Work
      shall terminate as of the date such litigation is filed.

   4. Redistribution. You may reproduce and distribute copies of the
      Work or Derivative Works thereof in any medium, with or without
      modifications, and in Source or Object form, provided that You
      meet the following conditions:

      (a) You must give any other recipients of the Work or
          Derivative Works a copy of this License; and

      (b) You must cause any modified files to carry prominent notices
          stating that You changed the files; and

      (c) You must retain, in the Source form of any Derivative Works
          that You distribute, all copyright, patent, trademark, and
          attribution notices from the Source form of the Work,
          excluding those notices that do not pertain to any part of
          the Derivative Works; and

      (d) If the Work includes a "NOTICE" text file as part of its
          distribution, then any Derivative Works that You distribute
          must include a readable copy of the attribution notices
          contained within such NOTICE file, excluding those notices
          that do not pertain to any part of the Derivative Works, in
          at least one of the following places: within a NOTICE text
          file distributed as part of the Derivative Works; within the
          Source form or documentation, if provided along with the
          Derivative Works; or, within a display generated by the
          Derivative Works, if and wherever such third-party notices
          normally appear. The contents of the NOTICE file are for
          informational purposes only and do not modify the License.
          You may add Your own attribution notices within Derivative
          Works that You distribute, alongside or as an addendum to
          the NOTICE text from the Work, provided that such additional
          attribution notices cannot be construed as modifying the
          License.

      You may add Your own copyright statement to Your modifications
```

```
Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.  See the License for the specific language governing
permissions and limitations under the License.
```

About this guide

For general information about the Derby documentation, such as a complete list of books, conventions, and further reading, see *Getting Started with Derby*.

For more information about Derby, visit the Derby website at http://db.apache.org/derby/. The website provides pointers to the Derby Wiki and other resources, such as the derby-users mailing list, where you can ask questions about issues not covered in the documentation.

Purpose of this guide

This guide provides information on securing Derby databases.

Derby provides several ways to protect the correctness and privacy of your data as well as to prevent accidental or malicious misuse of the database software itself. This guide explains how to improve the database security of applications and machines that use Derby. It describes how to configure security for both embedded applications and applications that use the Derby Network Server.

Audience

This guide is intended for software developers who already know some SQL and Java.

Derby users who are not familiar with the SQL standard or the Java programming language will benefit from consulting books on those subjects.

How this guide is organized

This guide includes the following two parts.

Part One: Introduction to database security

- *Why databases need security*

  Describes the vulnerabilities and threats that databases face.
- *Defenses against security threats*

  Describes the kinds of defenses that databases can use.
- *Defenses mapped to threats*

  Shows how the defenses available to databases map to the threats that they face.
- *Designing safer Derby applications*

  Describes important techniques for securing databases.
- *Security terminology*

  Provides a glossary of security terms used in this part.

Part Two: Configuring security for Derby

- *Basic security configuration tasks*

  Lists basic tasks for configuring security in an embedded or client/server environment.
- *Configuring database encryption*

  Explains how to encrypt Derby databases.
- *Using signed jar files*

Explains how to use signed jar files in Derby databases.

- *Configuring SSL/TLS*

  Explains how to use SSL/TLS to encrypt network traffic in a client/server environment.
- *Understanding identity in Derby*

  Describes the concepts of identity, users, and authorization identifiers in Derby.
- *Configuring user authentication*

  Explains how to configure authentication, which determines whether someone is a legal user.
- *Configuring user authorization*

  Explains how to configure authorization, which determines what operations can be performed by a user's identity.
- *Restricting file permissions*

  Explains how to take advantage of file system protections.
- *Putting it all together*

  Shows how to enable all the available Derby defenses.

Part One: Introduction to database security

This part of the manual provides a conceptual introduction to database security.

Why databases need security

An unsecured database has numerous vulnerabilities to different kinds of threats.

This section does not provide a complete list of these vulnerabilities and threats. No survey of security concerns can hope to be complete. However, this section attempts to list the major vulnerabilities and threats known today.

The remainder of this guide describes how you can combat these threats.

Vulnerabilities of unsecured databases

If you do not configure Derby security, you must be aware of the following vulnerabilities.

- **Network JDBC**: Network JDBC connections expose sensitive operations to use by persons who may not have account privileges on the database machine.
- **Cleartext traffic**: By default, network traffic travels in cleartext.
- **Unbounded growth**: Tables can grow arbitrarily large.
- **CPU hogging**: Unbounded CPU cycles can be consumed by connection attempts, SQL queries, and user code running in the database.
- **Superusers**: By default, all Derby users enjoy extensive powers to read and write in all databases.
- **Launch privileges**: Derby procedural code executes with the operating system privileges of the account that launched the virtual machine. This includes system-supplied procedures as well as custom, user-coded procedures.
- **User code**: Arbitrary user code can execute in the Derby virtual machine by means of user-coded functions and procedures.
- **Open source**: Derby's code itself is publicly visible as part of the Apache Derby open source project. This means that an attacker can write subtle malware after studying the code and file formats. Note that while closed source code enjoys the advantage of "security by obscurity", openness can confer other, countervailing security advantages.

Threats to unsecured databases

A threat is potential damage caused by an attacker using a technique to exploit a vulnerability. We have already seen examples of Derby vulnerabilities. Examples of damages, attackers, and techniques follow.

Significant damages include the following.

- **Denial-of-service attacks**: An attacker can monopolize resources on the host machine. For instance, an attacker can launch a runaway procedure on the Derby virtual machine, fill up the file system, or pepper the Derby server with incessant connection requests.
- **Theft**: An attacker can read private information stored in a Derby database or transmitted across the network. With enough privileges and by exploiting application code visible on the classpath, an attacker can use Derby to read private information stored elsewhere on the server machine or even on other machines inside the firewall.
- **Corruption**: An attacker can modify or destroy information stored in a Derby database or elsewhere inside the firewall.

Attackers include the following.

- **Insiders**: These are privileged persons who enjoy access to systems inside the firewall and maybe even to restricted machine rooms. Drunken System Administrators and disgruntled co-workers can cause significant damage.
- **Outsiders**: These include politically motivated governments and guerillas, commercially motivated businesses and criminals, and thrill-seeking attackers.

Techniques of attack include the following.

- **SQL injection**: This technique plagues applications that construct queries by concatenating input from clients. A clever client can put SQL into these fields. That SQL, not intended by the application, then runs inside the database.
- **Man-in-the-middle**: In this technique, the client believes that it is talking to the server. In reality, the connection has been intercepted by another machine. The device in the middle can examine and alter the traffic.
- **Eavesdropping**: This is a special case of the man-in-the-middle attack. The attacker listens to the network traffic but does not interfere with it. An example of this technique is password sniffing, in which a machine in the middle intercepts the credentials handshake between client and server.
- **Malware**: This is a general term for viruses, worms, trojan horses, and other intrusive or destructive code that can infect a machine.
- **Probing**: This is the technical equivalent of jiggling door handles to see what doors are unlocked.
- **Physical access**: This refers to the low-tech, brute-force technique of gaining physical access to a restricted area or machine and, for instance, exploiting superuser powers that might be available from a system's console.
- **Social engineering**: This refers to the low-tech technique of gaining and abusing the confidence of someone who has the keys.

Defenses against security threats

Defenses against threats can be divided into two categories.

- Derby defenses
- Other defenses

The following terms are useful in discussing these defenses.

- **System Administrator**: This is the person who configures Derby's system-wide behavior. Typically, this is a highly privileged user responsible for allocating machine resources, managing the network, configuring security, and actually launching the Virtual Machine (VM).
- **Database Owner**: This is the person who creates and tends the databases needed by a particular application. Of course, the Database Owner could be the System Administrator.
- **User**: This is a person authorized to use an application. This includes end-users, technical support engineers, and developers who maintain the application.

Derby defenses against threats

Derby provides numerous defenses against security threats.

These defenses are described in the following table.

Table 1. Derby defenses

Defense	Task Owner	Description
SSL/TLS	System Administrator	The System Administrator can require that SSL/TLS be used to encrypt network traffic between Derby clients and servers, along the way raising an extra authentication hurdle.
Encryption	Database Owner	A Database Owner can require that the data for an application be encrypted before being stored on disk. This makes it expensive to steal and corrupt the data.
Authentication	Database Owner	Using usernames and passwords, a Database Owner can restrict access to an application's data.
Coarse-grained authorization	Database Owner	A Database Owner can divide an application's users into three groups: those with no privileges, those with read-only privileges, and those with read-write privileges.
Fine-grained SQL authorization	Database Owner	By using SQL GRANT and REVOKE statements, a Database Owner can further restrict access to fine-grained pieces of data and code.

Defenses outside of Derby

In addition to the defenses provided by Derby, you should take advantage of defenses provided by your machine and intranet.

It is important to configure these defenses to protect Derby from attacks by both outsiders and insiders.

- **Firewalls**: Limit network access to the machine that runs Derby.
- **Accounts**: Limit login access to the machine that runs Derby. Centrally administer accounts using, for instance, an LDAP server.
- **Physical locks**: Limit physical access to the machine that runs Derby.
- **Secure traffic**: Encrypt the traffic that flows on your internal network.
- **File permissions**: Restrict the file permissions granted to the account that launches Derby.
- **Quotas**: Limit the file space and CPU that an account can consume.
- **Containerization**: Use an operating system container to limit resource consumption and access.

Defenses mapped to threats

The following table maps defenses to examples of threats that they parry.

This matrix can help you decide whether you need to configure specific defenses. Consult this table if you decide NOT to configure a defense -- make sure that you are still shielded from the corresponding threats.

Table 2. Derby defenses

Defense	Damages	Attackers	Techniques	Vulnerabilities
SSL/TLS	Theft and corruption	Insiders and outsiders	Man-in-the middle, eavesdropping, physical access	Network JDBC, cleartext traffic
Encryption	Theft and corruption	Chiefly insiders	Physical access	Open source
Authentication	Theft, corruption, denial of service	Insiders and outsiders	Probing	Superusers
Coarse-grained authorization	Theft, corruption, denial of service	Insiders and outsiders	Probing	Superusers
Fine-grained SQL authorization	Theft, corruption, denial of service	Insiders and outsiders	Probing	Superusers
Firewalls	Theft, corruption, denial of service	Insiders and outsiders	Probing	Network JDBC
Accounts	Theft, corruption, denial of service	Insiders	Man-in-the-middle, malware, physical access	Launch privileges, user code
Physical locks	Theft, corruption, denial of service	Insiders	Man-in-the-middle, malware, physical access	Launch privileges, user code
Secure traffic	Theft and corruption	Insiders	Man-in-the-middle, eavesdropping	Cleartext traffic
File permissions	Theft, corruption, denial of service	Insiders and outsiders	Malware	Launch privileges, user code, open source

Designing safer Derby applications

The following tips should help you write and deploy safer applications that use Derby.

- **Create a launch account**: Create an operating system account for the System Administrator. This will be the account that launches Derby. This account should not be the operating system's superuser.
- **Limit file permissions**: Limit the file permissions of this System Administrator account to just the directories that the application should be allowed to read and write. Do not grant read or write access on these directories to any other operating system accounts.
- **Prevent JDBC leaks**: Do not let JDBC connections leak outside your intranet's firewall. If possible, design your application so that external clients talk to an application server, which in turn communicates with Derby. Limit the JDBC connections to communication between the application server and Derby.
- **Protect against injection**: Do not construct queries by concatenating strings that are filled in by clients. To parameterize your queries, use JDBC ? parameters in `PreparedStatements`.
- **Deploy your shields**: By default, enable all defenses mentioned in this section. If you need to turn off a defense for performance reasons, then carefully consider how you will protect your application from the threats which that defense parries.

Security terminology

In discussing Derby defenses, the following terms are useful.

attacker
: A person or organization that seeks to compromise the security of a system.

damages
: The harm done to a system by an attacker. Includes denial-of-service, theft of secrets, and corruption of data.

Database Owner
: The person who creates a database and configures its security.

insider
: An attacker, such as a disgruntled co-worker, who operates inside the firewall and enjoys the presumption of friendliness.

malware
: A program that compromises security, such as a virus, worm, or spider.

outsider
: An attacker who operates outside the firewall.

System Administrator
: The account that launches Derby and is responsible for configuring the security of the Derby system.

technique
: A mechanism for compromising the security of a system, such as man-in-the-middle or SQL injection.

user
: A person authorized to use a Derby application.

vulnerability
: A feature of Derby that attackers can exploit in order to cause damage.

Part Two: Configuring security for Derby

This part of the manual describes the specific tasks involved in securing Derby databases.

Derby can be deployed in a number of ways and in a number of different environments, ranging from a single-user deployment for small-scale development and testing to a multi-user deployment of a large database. For all but the smallest deployments, however, it is essential to make the Derby system secure.

To secure a Derby database or databases, take the following steps.

1. Understand the basic tasks involved in configuring security in a client-server environment or an embedded environment.

   See Basic security configuration tasks for details.
2. Encrypt your databases.

   Derby provides ways to encrypt data stored on disk.

   For more information about encryption, see Configuring database encryption.
3. Sign any jar files that you use in your databases.

   Derby validates certificates for classes loaded from signed jar files.

   For more information about using signed jar files, see Using signed jar files.
4. Encrypt network traffic with SSL/TLS.

   SSL/TLS certificate authentication is also supported. See Configuring SSL/TLS for details.
5. Understand the concept of identity in Derby.

   See Understanding identity in Derby for details.
6. Configure *authentication* by setting up users and passwords.

   Authentication determines whether someone is a legal user. It establishes a user's identity. Derby verifies user names and passwords before permitting access to the Derby system.

   For more information about authentication, see Configuring user authentication.
7. Configure *user authorization* for the system.

   Authorization determines what operations can be performed by a user's Derby identity. Authorization grants users or roles permission to read a database or to write to a database.

   For more information about authorization, see Configuring user authorization.
8. If necessary, restrict database file access to the operating system account that started the JVM.

   For details, see Restricting file permissions.

See the *Derby Reference Manual* for information about many security-related properties and system procedures, as well as such statements as GRANT, REVOKE, CREATE ROLE, DROP ROLE, CREATE PROCEDURE, and CREATE FUNCTION.

Basic security configuration tasks

In most cases, you enable Derby security features through the use of properties. It is important to understand the best way to set properties for your environment.

Derby does not come with a built-in superuser. For that reason, be careful to follow these steps when you configure Derby for user authentication and user authorization.

1. When first working with security, work with system-level properties only so that you can easily override them if you make a mistake. See "Scope of properties" and "Setting system-wide properties" in the *Derby Developer's Guide* for more information.
2. Be sure to create at least one valid user, and grant that user full (read-write) access. For example, you might always want to create a user called `sa` with the password `derby` while you are developing.
3. Test the authentication system while it is still configured at the system level. Be absolutely certain that you have configured the system correctly before setting the properties as database-level properties.
4. Before disabling system-level properties (by setting `derby.database.propertiesOnly` to true), test that at least one database-level read-write user (such as `sa`) is valid. If you do not have at least one valid user that the system can authenticate, you will not be able to access your database.

Configuring security in an embedded environment

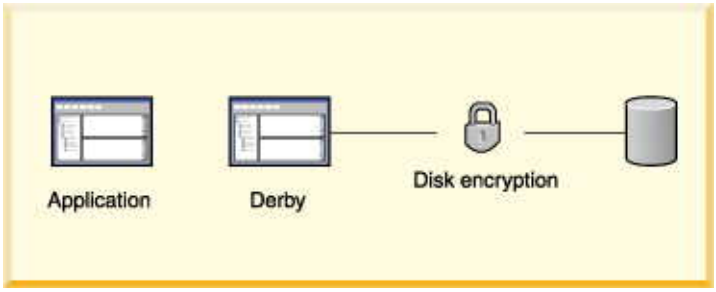
In an embedded environment, typically there is only one database per system, and there are no administrative resources to protect databases.

To configure security in an embedded environment:

1. Encrypt the database when you create it.
2. Configure all security features as database-level properties. These properties are stored in the database (which is encrypted). See "Scope of properties" and "Setting database-wide properties" in the *Derby Developer's Guide* for more information.
3. Turn on protection for database-level properties so that they cannot be overridden by system properties by setting the `derby.database.propertiesOnly` property to true. See the *Derby Reference Manual* for details on this property.
4. To prevent unauthorized users from accessing databases once they are booted, turn on user authentication and SQL authorization for the database. Use NATIVE authentication or, alternatively, LDAP or a user-defined class.

The following figure shows how disk encryption protects data when the recipient might not know how to protect data. It is useful for databases deployed in an embedded environment.

Figure 1. Using disk encryption to protect data



Configuring security in a client/server environment

This procedure requires a system with multiple databases and some administrative resources.

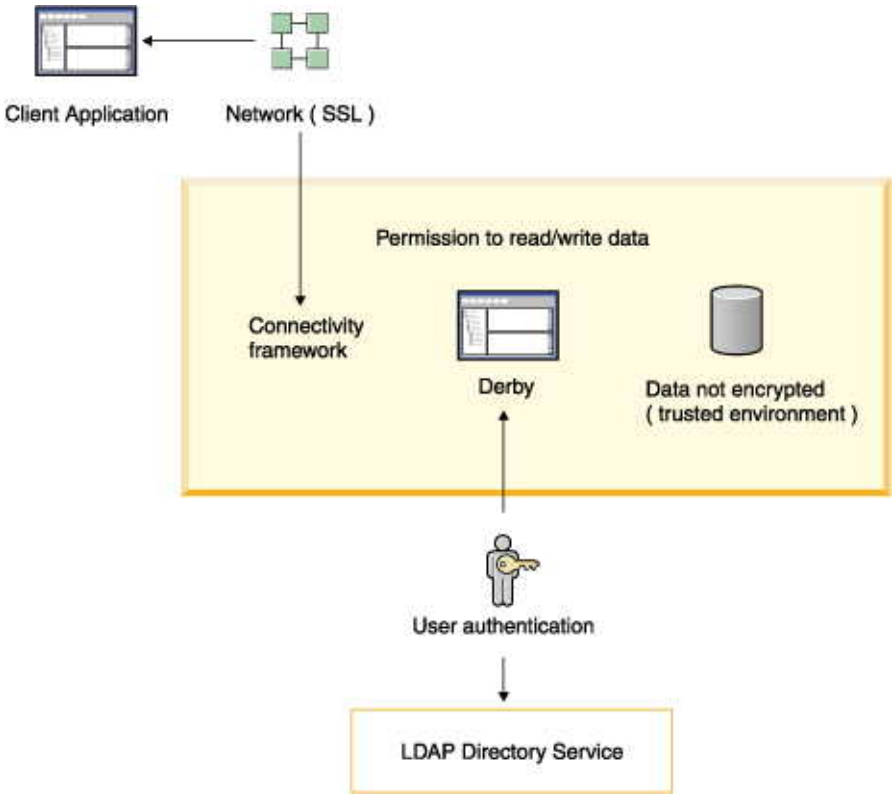
1. Configure security features as system-level properties.

2. Provide administrative-level protection for the `derby.properties` file and Derby databases. For example, you can protect these files and directories with operating system permissions and firewalls.
3. Turn on user authentication for your system. All users must provide valid user IDs and passwords to access the Derby system. Use NATIVE authentication (or, alternatively, LDAP or a user-defined class).

   > Important: It is also strongly recommended that production systems protect network connections with SSL/TLS.
4. Configure fine-grained user authorization (SQL authorization) for your databases.

The following figure shows some of the Derby security mechanisms at work in a client/server environment. User authentication is performed by accessing an LDAP directory service. The data in the database is not encrypted in this trusted environment.

Figure 2. Using an LDAP directory service in a trusted environment



Network Server security

By default, the Derby Network Server listens only on the localhost. Clients must use the localhost host name to connect.

By default, clients cannot access the Network Server from another host. To enable connections from other hosts, set the `derby.drda.host` property, or start the Network Server with the `-h` option in the `java org.apache.derby.drda.NetworkServerControl start` command.

In the following example, the server will listen only on the localhost, and clients cannot access the server from another host:

```
java org.apache.derby.drda.NetworkServerControl start
```

In the following example, the server runs on the host machine `sampleserver.example.com` and also listens for clients from other hosts. Clients must specify the server in the URL or DataSource as `sampleserver.example.com`:

```
java org.apache.derby.drda.NetworkServerControl start \
-h sampleserver.example.com
```

To start the Network Server so that it will listen on all interfaces, start with an IP address of `0.0.0.0`, as shown in the following example:

```
java org.apache.derby.drda.NetworkServerControl start -h 0.0.0.0
```

A server that is started with the `-h 0.0.0.0` option will listen to client requests that originate from both `localhost` and from other machines on the network.

However, administrative commands (for example, `org.apache.derby.drda.NetworkServerControl shutdown`) can run only on the host where the server was started, even if the server was started with the `-h` option.

Configuring database encryption

Derby provides a way for you to encrypt your data on disk.

By default, Derby stores its data unencrypted in ordinary operating system files. An attacker who can view those files can simply type them out, exposing all sorts of data stored in string columns. Knowing Derby's file formats, a clever attacker could even view numeric data stored in those files. Even worse, a clever attacker could change the data itself.

Fortunately, Derby can encrypt databases. On a shared machine, that helps protect data from other users, including disgruntled or curious superusers. Encryption helps protect private financial data from thieves who physically steal your laptop.

Before encrypting a database, you need to make two choices:

- A **boot password**: This is the password that unlocks your encrypted data when you want to use it.
- An **encryption algorithm**: This is a transformation name as described in the API documentation for the `javax.crypto.Cipher` class. Derby encryption relies on the JCE libraries supplied with the virtual machine. For more information on those libraries, see the *Java Cryptography Architecture (JCA) Reference Guide* (http://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html).

Here is a `ij` command that creates an encrypted database. Notice the additional attributes in bold on the database creation URL: `dataEncryption`, `encryptionAlgorithm`, and `bootPassword`. The URL string must be all on one line.

```
connect 'jdbc:derby:myEncryptedDatabaseName;create=true;
dataEncryption=true;encryptionAlgorithm=Blowfish/CBC/NoPadding;
bootPassword=mySuperSecretBootPassword';
```

Once you have created an encrypted database, you can work in it. After you shut down the encrypted database, you can reconnect to it by simply supplying your boot password in the connection URL, as shown in the following `ij` command:

```
connect 'jdbc:derby:myEncryptedDatabaseName;
bootPassword=mySuperSecretBootPassword';
```

Keep in mind that by booting a database with its boot password, you unlock the database for the lifetime of the virtual machine. This means that other threads can connect to the database without supplying the boot password. This situation lasts until the database is explicitly shut down or the virtual machine exits. For a single-user, shrink-wrapped application, this is generally not a problem. However, for a multi-user application, you need to take steps to keep the data secure during the various stages of working with the database:

1. **Unlocking the database**: The boot password is used to initially unlock encrypted data. Once the Database Owner has unlocked the database, other users can connect to it without supplying the boot password.
2. **Working with the database**: For that reason, you should configure Derby authorization (see below) to restrict the users who may access the unlocked data.
3. **Relocking the database**: To relock your data, simply shut down the database.

The following sections provide detailed information about database encryption.

Note: Jar files stored in a database are not encrypted.

Requirements for Derby encryption

Derby supports disk encryption and requires an encryption provider.

An encryption provider implements the Java cryptography concepts. The Java Runtime Environment (JRE) for Java SE includes Java Cryptographic Extensions (JCE, part of the Java Cryptography Architecture) and one or more default encryption providers. For more information, see the *Java Cryptography Architecture (JCA) Reference Guide* at http://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html.

The JRE determines the default encryption provider as follows:

- The JRE's provider is the default.
- If your environment for some reason does not include a provider, it must be specified.

Working with encryption

This section describes using encryption in Derby.

Encrypting databases on creation

You configure a Derby database for encryption when you create the database by specifying attributes on the connection URL.

- To enable encryption, use the `dataEncryption=true` attribute.
- To provide a key for the encryption, specify either the `bootPassword=`*`key`* attribute or the `encryptionKey=`*`key`* attribute.

The following connection URL specifies a boot password:

```
jdbc:derby:encryptedDB;create=true;dataEncryption=true;
bootPassword=DBpassword
```

The following URL specifies an encryption key:

```
jdbc:derby:encryptedDB;create=true;dataEncryption=true;
encryptionKey=6162636465666768
```

The default encryption algorithm is DES.

You can specify an encryption provider and/or encryption algorithm other than the defaults by using the `encryptionProvider=`*`providerName`* and `encryptionAlgorithm=`*`algorithm`* attributes. See Specifying an alternate encryption provider and Specifying an alternate encryption algorithm for more information.

See the *Derby Reference Manual* for details on the connection URL attributes.

Encrypting an existing unencrypted database

You can encrypt an unencrypted Derby database by specifying attributes on the connection URL when you boot the database. The attributes that you specify depend on how you want the database encrypted.

- If the database is configured with log archival, you must disable log archival and perform a shutdown before you can encrypt the database.
- If any global transactions are in the prepared state after recovery, the database cannot be encrypted.

When you encrypt an existing, unencrypted database, you can specify whether the database should be encrypted using a boot password (`bootPassword=`*`key`*) or an external encryption key (`encryptionKey=`*`key`*). You can also specify the `encryptionProvider=`*`providerName`* attribute and the `encryptionAlgorithm=`*`algorithm`* attribute on the connection URL. The database is configured with the specified encryption attributes, and all of the existing data in the database is encrypted.

See the *Derby Reference Manual* for details on the connection URL attributes.

Encrypting a database is a time-consuming process because it involves encrypting all of the existing data in the database. If the process is interrupted before completion, all the changes are rolled back the next time the database is booted. If the interruption occurs immediately after the database is encrypted but before the connection is returned to the application, you might not be able to boot the database without the boot password or external encryption key. In these rare circumstances, you should try to boot the database with the boot password or the external encryption key.

Recommendation: Ensure that you have enough free disk space before you encrypt a database. In addition to the disk space required for the current size of the database, temporary disk space is required to store the old version of the data to restore the database back to its original state if the encryption is interrupted or returns errors. All of the temporary disk space is released back to the operating system after the database is encrypted.

To encrypt an existing unencrypted database:

1. Specify the `dataEncryption=true` attribute and either the `encryptionKey=`*`key`* attribute or the `bootPassword=`*`key`* attribute in a connection URL and boot the database.

   For example, to encrypt the `salesdb` database with the boot password `abc1234xyz`, specify the following attributes in the URL:

   ```
   jdbc:derby:salesdb;dataEncryption=true;bootPassword=abc1234xyz
   ```

   If authentication and SQL authorization are both enabled, the credentials of the Database Owner must be supplied as well, since encryption is a restricted operation.

   After you encrypt an existing, unencrypted database, be sure to check for `SQLWarnings`. The encryption succeeded only if there were no `SQLWarnings` or `SQLExceptions`.

   If you disabled log archival before you encrypted the database, create a new backup of the database after the database is encrypted. For more information, see the section "Backing up and restoring databases" in the *Derby Server and Administration Guide*, particularly "Roll-forward recovery".

Creating a boot password

When you encrypt a database, you usually specify a boot password, which is an alphanumeric string used to generate the encryption key. (You can also specify an encryption key directly.)

The length of the encryption key depends on the algorithm used:

- AES (128, 192, and 256 bits)
- DES (the default) (56 bits)
- DESede (168 bits)
- All other algorithms (128 bits)

Note: The boot password should have at least as many characters as number of bytes in the encryption key (56 bits=8 bytes, 168 bits=24 bytes, 128 bits=16 bytes). The minimum number of characters for the boot password allowed by Derby is eight.

It is a good idea not to use words that would be easily guessed, such as a login name or simple words or numbers. A boot password, like any password, should be a mix of numbers and uppercase and lowercase letters.

You turn on and configure encryption and specify the corresponding boot password on the connection URL for a database when you create it:

```
jdbc:derby:encryptionDB1;create=true;dataEncryption=true;
bootPassword=clo760uds2caPe
```

Note: If you lose the boot password and the database is not currently booted, you will not be able to connect to the database any more. (If you know the current boot password, you can change it. See Encrypting databases with a new key.)

Specifying an alternate encryption provider:

You can specify an alternate provider when you create the database with the `encryptionProvider=`*`providerName`* attribute.

You must specify the full package and class name of the provider, and you must also add the libraries to the application's classpath.

```
-- using the the provider library bcprov-jdk15on-147.jar
-- available from www.bouncycastle.org
jdbc:derby:encryptedDB3;create=true;dataEncryption=true;
bootPassword=clo760uds2caPe;
encryptionProvider=org.bouncycastle.jce.provider.BouncyCastleProvider;
encryptionAlgorithm=DES/CBC/NoPadding

-- using a provider available from
-- http://jce.iaik.tugraz.at/sic/Download
jdbc:derby:encryptedDB3;create=true;dataEncryption=true;
bootPassword=clo760uds2caPe;
encryptionProvider=iaik.security.provider.IAIK;
encryptionAlgorithm=DES/CBC/NoPadding
```

Specifying an alternate encryption algorithm:

Derby supports the following encryption algorithms.

- DES (the default)
- DESede (also known as triple DES)
- Any encryption algorithm that fulfills the following requirements:
  - It is symmetric
  - It is a block cipher, with a block size of 8 bytes
  - It uses the `NoPadding` padding scheme
  - Its secret key can be represented as an arbitrary byte array
  - It requires exactly one initialization parameter, an initialization vector of type `javax.crypto.spec.IvParameterSpec`

- It can use `javax.crypto.spec.SecretKeySpec` to represent its key

For example, the algorithm `Blowfish` implemented in the Java Cryptography Extension (JCE) packages (`javax.crypto.*`) fulfills these requirements.

By Java convention, an encryption algorithm is specified like this:

```
algorithmName/feedbackMode/padding
```

The only feedback modes allowed are:

- CBC
- CFB
- ECB
- OFB

The only padding mode allowed is `NoPadding`.

By default, Derby uses the DES algorithm of `DES/CBC/NoPadding`.

To specify an alternate encryption algorithm when you create a database, use the `encryptionAlgorithm=`*`algorithm`* attribute. If the algorithm you specify is not supported by the provider you have specified, Derby throws an exception.

To specify the AES encryption algorithm with a key length other than the default of 128, specify the `encryptionKeyLength=`*`length`* attribute. For example, you might specify the following connection attributes:

```
jdbc:derby:encdbcbc_192;create=true;dataEncryption=true;
encryptionKeyLength=192;encryptionAlgorithm=AES/CBC/NoPadding;
bootPassword=Thursday
```

To use the AES algorithm with a key length of 192 or 256, you must use unrestricted policy jar files for your JRE. You can obtain these files from your Java provider. They might have a name like "Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files." If you specify a non-default key length using the default policy jar files, a Java exception occurs.

Encrypting databases with a new key

You can apply a new encryption key to a Derby database by specifying a new boot password or a new external key.

Encrypting a database with a new encryption key is a time-consuming process because it involves encrypting all of the existing data in the database with the new encryption key. If the process is interrupted before completion, all the changes are rolled back the next time the database is booted. If the interruption occurs immediately after the database is encrypted with the new encryption key but before the connection is returned to the application, you might not be able to boot the database with the old encryption key. In these rare circumstances, you should try to boot the database with the new encryption key.

Recommendation: Ensure that you have enough free disk space before you encrypt a database with a new key. In addition to the disk space required for the current size of the database, temporary disk space is required to store the old version of the data to restore the database back to its original state if the new encryption is interrupted or returns errors. All of the temporary disk space is released back to the operating system after the database is reconfigured to work with the new encryption key.

To encrypt a database with a new encryption key:

1. Use the type of encryption that is currently used to encrypt the database:
   - To encrypt the database with a new boot password key, use the `newBootPassword=`*`newPassword`* attribute.

- To encrypt the database with a new external encryption key, use the `newEncryptionKey=`*`key`* attribute.

If authentication and SQL authorization are both enabled, the credentials of the Database Owner must be supplied, since reencryption is a restricted operation.

Encrypting databases with a new boot password:

You can apply a new boot password to a Derby database by specifying the `newBootPassword=`*`newPassword`* attribute on the connection URL when you boot the database.

- If the database is configured with log archival for roll-forward recovery, you must disable log archival and perform a shutdown before you can encrypt the database with a new boot password.
- If any global transactions are in the prepared state after recovery, the database cannot be encrypted with a new boot password.
- If the database is currently encrypted with an external encryption key, use the `newEncryptionKey=`*`key`* attribute to encrypt the database.

When you use the `newBootPassword=`*`newPassword`* attribute, a new encryption key is generated internally by the engine, and the key is protected using the new boot password. The newly generated encryption key encrypts the database, including the existing data. You cannot change the encryption provider or encryption algorithm when you apply a new boot password.

To encrypt a database with a new boot password:

1. Specify the `newBootPassword=`*`newPassword`* attribute in a URL and reboot the database.

   For example, if you use the following URL to reboot the `salesdb` database, the database is encrypted with the new encryption key and is protected by the password `new1234xyz`:

   ```
   jdbc:derby:salesdb;bootPassword=abc1234xyz;newBootPassword=new1234xyz
   ```

   If authentication and SQL authorization are both enabled, the credentials of the Database Owner must be supplied as well, since reencryption is a restricted operation.

   After you change the boot password, be sure to check for `SQLWarnings`. The change succeeded only if there were no `SQLWarnings` or `SQLExceptions`.

   If you disabled log archival before you applied the new boot password, create a new backup of the database after the database is reconfigured with the new boot password. For more information, see the section "Backing up and restoring databases" in the *Derby Server and Administration Guide*, particularly "Roll-forward recovery".

Encrypting databases with a new external encryption key:

You can apply a new external encryption key to a Derby database by specifying the `newEncryptionKey=`*`key`* attribute on the connection URL when you boot the database.

- If the database is configured with log archival for roll-forward recovery, you must disable log archival and perform a shutdown before you can encrypt the database with a new external encryption key.
- If any global transaction are in the prepared state after recovery, the database cannot be encrypted with a new encryption key.
- If the database is currently encrypted with a boot password, use the `newBootPassword=`*`newPassword`* attribute to encrypt the database.

To encrypt a database with a new external encryption key:

1. Specify the `newEncryptionKey=`*`key`* attribute in a URL and reboot the database.

   For example, if you use the following URL to reboot the `salesdb` database, the database is encrypted with the new encryption key `6862636465666768`:

   ```
   jdbc:derby:salesdb;encryptionKey=6162636465666768;
   newEncryptionKey=6862636465666768'
   ```

   If authentication and SQL authorization are both enabled, the credentials of the Database Owner must be supplied as well, since encryption is a restricted operation.

   After you change the encryption key, be sure to check for `SQLWarnings`. The change succeeded only if there were no `SQLWarnings` or `SQLExceptions`.

   If you disabled log archival before you applied the new encryption key, create a new backup of the database after the database is reconfigured with the new encryption key. For more information, see the section "Backing up and restoring databases" in the *Derby Server and Administration Guide*, particularly "Roll-forward recovery".

Booting an encrypted database

If you create an encrypted database using the `bootPassword=`*`key`* attribute, you must specify the boot password to reboot the database. If you create an encrypted database using the `encryptionKey=`*`key`* attribute, you must specify the encryption key to reboot the database.

Encrypted databases cannot be booted automatically along with all other system databases on system startup (see "`derby.system.bootAll`" in the *Derby Reference Manual*). Instead, you boot an encrypted database when you first connect to the database.

Booting a database with the `bootPassword=`*`key`* attribute

To access an encrypted database called `wombat` that was created with the boot password `clo760uds2caPe`, use the following connection URL:

```
jdbc:derby:wombat;bootPassword=clo760uds2caPe
```

Booting a database with the `encryptionKey=`*`key`* attribute

To access an encrypted database called `flintstone` that was created with the attributes `encryptionKey=c566bab9ee8b62a5ddb4d9229224c678` and `encryptionAlgorithm=AES/CBC/NoPadding`, use the following connection URL:

```
jdbc:derby:flintstone;encryptionKey=c566bab9ee8b62a5ddb4d9229224c678
```

After the database is booted, all connections can access the database without the boot password. Only a connection that boots the database requires the key.

For example, the following connections would boot the database and require the boot password or encryption key, depending on what mechanism was used to encrypt the database originally:

- The first connection to the database in the JVM session
- The first connection to the database after the database has been explicitly shut down
- The first connection to the database after the system has been shut down and then rebooted

Note: The boot password and the encryption key are not meant to prevent unauthorized connections to the database after the database is booted. To protect a database after it has been booted, turn on user authentication (see Configuring user authentication).

Decrypting an encrypted database

You can return an encrypted database to an unencrypted state by specifying attributes on the connection URL.

To decrypt an encrypted database, specify the `decryptDatabase=true` attribute in conjunction with either the `bootPassword=`*key* attribute or the `encryptionKey=`*key* attribute.

See the *Derby Reference Manual* for details on the connection URL attributes.

Recommendation: Ensure that you have enough free disk space before you decrypt a database. In addition to the disk space required for the unencrypted size of the database, temporary disk space is required to store the encrypted version of the data to restore the database to its encrypted state if the decryption is interrupted or returns errors. All of the temporary disk space is released back to the operating system after the database is decrypted.

You must shut down the database before you decrypt it. An attempt to decrypt a booted database has no effect.

If the database is configured with log archival, you must disable log archival in addition to shutting down the database before you can decrypt the database. You should also create a new backup of the database before you decrypt it, and create another after you decrypt it. For more information, see the section "Backing up and restoring databases" in the *Derby Server and Administration Guide*, particularly "Roll-forward recovery".

If any global transactions are in the prepared state after recovery, the database cannot be decrypted.

If authentication and SQL authorization are both enabled, the credentials of the Database Owner must be supplied as well, since decryption is a restricted operation.

After you decrypt the database, be sure to check for `SQLWarnings`. The decryption succeeded only if there were no `SQLWarnings` or `SQLExceptions`.

Using signed jar files

In a Java SE environment, Derby can detect digital signatures on jar files. When attempting to load a class from a signed jar file stored in the database, Derby will verify the validity of the signature.

Note: The Derby class loader only validates the integrity of the signed jar file and verifies that the certificate has not expired. Derby cannot ascertain whether the validity or identity of declared signer is correct.

When loading classes from an application jar file in a Java SE environment, Derby behaves as follows if the class is signed:

- Verifies that the jar file was signed using a X.509 certificate (that is, it can be represented by the class `java.security.cert.X509Certificate`). If not, throws an exception.
- Verifies that the digital signature matches the contents of the file. If not, throws an exception.
- Checks that the set of signing certificates are all valid for the current date and time. If any certificate has expired or is not yet valid, throws an exception.
- Passes the array of certificates to the `setSigners()` method of `java.lang.ClassLoader`. This allows security managers to obtain the list of signers for a class (using `java.lang.Class.getSigners`) and then validate the identity of the signers using the services of a Public Key Infrastructure (PKI).

For more information about signed jar files, see http://docs.oracle.com/javase/8/docs/technotes/guides/jar/jar.html.

Configuring SSL/TLS

By default, network traffic travels in cleartext between Derby clients and servers.

By using a man-in-the-middle ploy, a clever attacker can read all of the string data shipped to and from the server. By knowing the Derby wire protocol, a clever attacker can read numeric data too. Even worse, the man-in-the-middle can change the data while it is traveling between the client and the server.

Fortunately, Derby can encrypt network traffic using the SSL/TLS (Secure Socket Layer/Transport Layer Security) logic supplied with the virtual machine. As a side effect, SSL/TLS raises an extra authentication hurdle too, involving *peer authentication*.

The term *peer* is used for the other part of the server-client communication: the server's peer is the client, and the client's peer is the server.

SSL/TLS for Derby (both for client and for server) operates in three possible modes:

off
: The default, no SSL/TLS encryption

basic
: SSL/TLS encryption, no peer authentication

peerAuthentication
: SSL/TLS encryption and peer authentication

Peer authentication may be set on the server, on the client, or on both. Peer authentication means that the other side of the SSL/TLS connection is authenticated based on a trusted certificate installed locally.

Before using this encryption technology, you will want to familiarize yourself with SSL/TLS concepts such as key pairs and certificates, and with the JDK's `keytool` application. You can find useful overviews of SSL/TLS at Apache and Wikipedia (http://httpd.apache.org/docs/2.0/ssl/ssl_intro.html and http://en.wikipedia.org/wiki/Secure_Sockets_Layer, respectively). You can find `keytool` documentation for Unix and for Windows at http://docs.oracle.com/javase/8/docs/technotes/tools/unix/keytool.html and http://docs.oracle.com/javase/8/docs/technotes/tools/windows/keytool.html, respectively.

Network encryption requires the following setup steps:

1. **Client certificates:** Each client must generate a client key pair and certificate. The client certificates must be loaded into the server's trust store.
2. **Server certificate:** The server must generate a server key pair and certificate. All of the clients must load the server's certificate into their respective trust stores.
3. **Server startup:** The server must be booted with system properties and a startup option that turn on SSL/TLS encryption.
4. **Client startup:** The client must be started with system properties that turn on SSL/TLS encryption. In addition, an extra attribute is added to the JDBC connection URL.

To use SSL/TLS to encrypt Derby's network traffic, the client must have a key store for holding its own public/private key pair. The client must also have a trust store for holding the server's certificate. If the key store and trust store do not already exist, the `keytool` program will create them. Suppose that the client stores its public/private key pair in `~/vault/ClientKeyStore`, and suppose that the client stores certificates from other systems in `~/vault/ClientTrustStore`.

Creating a client key pair and certificate

Follow these steps to create a client key pair and a client certificate.

1. Choose a password for the key store.

   Suppose you choose the password `secretClientPassword`.
2. On the client system, issue the following command to create the client's public/private key pair.

   You will be prompted to enter the password plus some identifying information (your input is marked **`bold`**):

```
keytool -genkey -alias MyClientName -keystore ~/vault/ClientKeyStore
Enter keystore password: secretClientPassword
What is your first and last name?
[Unknown]: MyFirstName MyLastName
What is the name of your organizational unit?
[Unknown]: Proofreading Department
What is the name of your organization?
[Unknown]: Name of my bookstore
What is the name of your City or Locality?
[Unknown]: New York
What is the name of your State or Province?
[Unknown]: NY
What is the two-letter country code for this unit?
[Unknown]: US
Is CN=MyFirstName MyLastName, OU=Proofreading Department, O=Name of
 my bookstore, L=New York, ST=NY, C=US correct?
[no]: yes


Enter key password for <MyClientName>
   (RETURN if same as keystore password):
```

3. Next, create a certificate for this client. Enter the command all on one line:

```
keytool -export -alias MyClientName \
-keystore ~/vault/ClientKeyStore  -rfc -file ClientCertificate \
-storepass secretClientPassword
```

   This command creates a file called `ClientCertificate`. Later, you will import this file into the server's trust store.

Creating a server key pair and certificate

Follow these steps to create a server key pair and a server certificate.

1. On the server system, issue the following command to to create a server key pair in a key store guarded by the `secretServerPassword` password:

```
keytool -genkey -alias MyServerName -keystore ~/vault/ServerKeyStore
Enter keystore password: secretServerPassword
...
```

2. Issue the following command (all on one line) to create a certificate named `ServerCertificate` from this key:

```
keytool -export -alias MyServerName \
-keystore ~/vault/ServerKeyStore -rfc -file ServerCertificate \
-storepass secretServerPassword
```

Importing certificates

Follow these steps to import each certificate into the other's trust store.

1. On the client, import the server certificate into the client's trust store:

```
keytool -import -alias favoriteServerCertificate \
-file ServerCertificate -keystore ~/vault/ClientTrustStore \
-storepass secretClientTrustStorePassword
```

2. On the server, import the client certificate into the server's trust store:

```
keytool -import -alias Client_1_Certificate \
-file ClientCertificate -keystore ~/vault/ServerTrustStore \
-storepass secretServerTrustStorePassword
```

Booting the server and connecting to it

Finally, boot the server and start the client.

The previous three topics covered the first two setup steps described in Configuring SSL/TLS, creating client certificates and creating a server certificate, then importing the certificates. This topic describes the remaining two steps, server startup and client startup.

Every time that we bring up the server, we must remember to turn on network encryption. We must set four VM properties that declare the locations and passwords for the server's key store and trust store:

- `javax.net.ssl.keyStore`
- `javax.net.ssl.keyStorePassword`
- `javax.net.ssl.trustStore`
- `javax.net.ssl.trustStorePassword`

In addition, we specify the `-ssl peerAuthentication` startup option. The command to start the server, therefore, looks something like this:

```
java -Djavax.net.ssl.keyStore=/Users/me/vault/ServerKeyStore \
-Djavax.net.ssl.keyStorePassword=secretServerPassword \
-Djavax.net.ssl.trustStore=/Users/me/vault/ServerTrustStore \
-Djavax.net.ssl.trustStorePassword=secretServerTrustStorePassword \
org.apache.derby.drda.NetworkServerControl start -p 8246 \
-ssl peerAuthentication
```

The `-p 8246` option starts the server on a nondefault port (rather than the default port of 1527).

The final step is to bring up a client. As with server startup, we must tell the VM the locations and passwords of the local key store and trust store. This example is a simple `ij` script. Notice the extra `ssl` attribute on the connection URL. That attribute tells the client to authenticate the server's identity using a certificate, and it tells the client that the network traffic must be encrypted:

```
java -Djavax.net.ssl.trustStore=/Users/me/vault/ClientTrustStore \
-Djavax.net.ssl.trustStorePassword=secretClientTrustStorePassword \
-Djavax.net.ssl.keyStore=/Users/me/vault/ClientKeyStore \
-Djavax.net.ssl.keyStorePassword=secretClientPassword \
org.apache.derby.tools.ij
ij version 10.11
ij> connect
 'jdbc:derby://localhost:8246/testdb;create=true;ssl=peerAuthentication';
ij> select schemaName, authorizationID from sys.sysschemas;
```

You will get errors from `ij` if you do not specify the extra VM properties and/or if you do not specify the `ssl` attribute on the connection URL. Here, for instance, is the output from running `ij` without the VM properties and `ssl` attribute:

```
java org.apache.derby.tools.ij
ij version 10.11
ij> connect 'jdbc:derby://localhost:8246/testdb;create=true';
ERROR 08006: A network protocol error was encountered and the connection
 has been
terminated: A PROTOCOL Data Stream Syntax Error was detected. Reason:
 0x3.
Plaintext connection attempt to an SSL enabled server?
```

When you want to administer the server (for instance, to bring it down), you will need to specify the locations and passwords of a valid key store and trust store as well as the extra `ssl` option on the server command line:

```
java -Djavax.net.ssl.trustStore=/Users/me/vault/ClientTrustStore \
-Djavax.net.ssl.trustStorePassword=secretClientTrustStorePassword \
-Djavax.net.ssl.keyStore=/Users/me/vault/ClientKeyStore \
-Djavax.net.ssl.keyStorePassword=secretClientPassword \
org.apache.derby.drda.NetworkServerControl shutdown -p 8246 \
-ssl peerAuthentication
```

Key and certificate handling

For SSL operation, the server always needs a key pair. If the server runs in peer authentication mode (the server authenticates the clients), each client needs its own key pair. In general, if one end of the communication wants to authenticate its partner, the first end needs to install a certificate generated by the partner.

The key pair is located in a file which is called a *key store*, and the JDK's SSL provider needs the system properties `javax.net.ssl.keyStore` and `javax.net.ssl.keyStorePassword` to access the keystore.

The certificates of trusted parties are installed in a file called a *trust store*. The JDK's SSL provider needs the system properties `javax.net.ssl.trustStore` and `javax.net.ssl.trustStorePassword` to access the trust store.

Key pair generation

Key pairs are generated with `keytool -genkey`. The simplest way to generate a key pair is to do the following:

```
keytool -genkey alias -keystore keystore
```

`keytool` will prompt for needed information, such as identity details and passwords.

Consult the JDK documentation for more information on `keytool`.

Certificate generation

Certificates are generated with `keytool -export` as follows:

```
keytool -export -alias alias -keystore keystore -rfc \
-file certificate-file
```

The certificate file may then be distributed to the relevant parties.

Certificate installation

Installation of a certificate in a trust store is done with `keytool -import` as follows:

```
keytool -import -alias alias -file certificate-file -keystore truststore
```

Examples

Generate the server key pair:

```
keytool -genkey -alias myDerbyServer -keystore serverKeyStore.key
```

Generate a server certificate:

```
keytool -export -alias myDerbyServer -keystore serverKeyStore.key -rfc \
-file myServer.cert
```

Generate a client key pair:

```
keytool -genkey -alias aDerbyClient -keystore clientKeyStore.key
```

Generate a client certficate:

```
keytool -export -alias aDerbyClient -keystore clientKeyStore.key -rfc \
-file aClient.cert
```

Install a client certificate in the server's trust store:

```
keytool -import -alias aDerbyClient -file aClient.cert \
-keystore serverTrustStore.key
```

Install the server certificate in a client's trust store:

```
keytool -import -alias myDerbyServer -file myServer.cert \
-keystore clientTrustStore.key
```

Starting the server with SSL/TLS

For server SSL/TLS, a server key pair needs to be generated. If the server is going to do client authentication, the client certificates need to be installed in the trust store.

These operations are described in Key and certificate handling.

SSL at the server side is activated with the property `derby.drda.sslMode` (default off) or the `-ssl` option for the server start command.

Starting the server with basic SSL encryption

When the SSL mode is set to `basic`, the server will only accept SSL encrypted connections.

The properties `javax.net.ssl.keyStore` and `javax.net.ssl.keyStorePassword` need to be set with the proper values.

Example

```
java -Djavax.net.ssl.keyStore=serverKeyStore.key \
-Djavax.net.ssl.keyStorePassword=qwerty \
-jar derbyrun.jar server start -ssl basic
```

Starting a server which authenticates clients

When the server's SSL mode is set to `peerAuthentication`, the server authenticates its clients' identity in addition to encrypting network traffic. In this situation, the server's *trust store* must contain a certificate for each client which will connect.

The `javax.net.ssl.trustStore` and `javax.net.ssl.trustStorePassword` need to be set in addition to the properties above.

See Running the client with SSL/TLS for client settings when the server does client authentication.

Example

```
java -Djavax.net.ssl.keyStore=serverKeyStore.key \
-Djavax.net.ssl.keyStorePassword=qwerty \
-Djavax.net.ssl.trustStore=serverTrustStore.key \
-Djavax.net.ssl.trustStorePassword=qwerty \
-jar derbyrun.jar server start -ssl peerAuthentication
```

Running the client with SSL/TLS

Basic SSL encryption on the client is enabled either by the URL attribute `ssl`, the property `ssl`, or the datasource attribute `ssl` set to `basic`.

Example

```
Connection c = getConnection("jdbc:derby://myhost:1527/db;ssl=basic");
```

Running a client which authenticates the server

If the client wants to authenticate the server, then the client's *trust store* must contain the server's certificate. See Key and certificate handling.

Client SSL with server authentication is enabled by the URL attribute `ssl` or the property `ssl` set to `peerAuthentication`. In addition, the system properties `javax.net.ssl.trustStore` and `javax.net.ssl.trustStorePassword` need to be set.

Example

```
System.setProperty("javax.net.ssl.trustStore","clientTrustStore.key");
System.setProperty("javax.net.ssl.trustStorePassword","qwerty");
Connection c =
   getConnection("jdbc:derby://myhost:1527/db;ssl=peerAuthentication");
```

Running the client when the server does client authentication

If the server does client authentication, the client will need a key pair and a client certificate which is installed in the server's *trust store*. See Key and certificate handling.

The client needs to set `javax.net.ssl.keyStore` and `javax.net.ssl.keyStorePassword`.

Example

```
System.setProperty("javax.net.ssl.keyStore","clientKeyStore.key");
System.setProperty("javax.net.ssl.keyStorePassword","qwerty");
Connection c = getConnection("jdbc:derby://myhost:1527/db;ssl=basic");
```

Running the client when both parties do peer authentication

This is a combination of the last two variants.

Example

```
System.setProperty("javax.net.ssl.keyStore","clientKeyStore.key");
System.setProperty("javax.net.ssl.keyStorePassword","qwerty");
System.setProperty("javax.net.ssl.trustStore","clientTrustStore.key");
System.setProperty("javax.net.ssl.trustStorePassword","qwerty");
Connection c =
   getConnection("jdbc:derby://myhost:1527/db;ssl=peerAuthentication");
```

Other server commands

The other server commands (`shutdown`, `ping`, `sysinfo`, `runtimeinfo`, `logconnections`, `maxthreads`, `timeslice`, `trace`, and `tracedirectory`) are implemented as clients, and they behave exactly as clients with regards to SSL.

See Running the client with SSL/TLS for more information.

The SSL mode is set with the property `derby.drda.sslMode` or the server command option `-ssl`.

Examples

The following command will shut down an SSL-enabled server:

```
java -jar derbyrun.jar server shutdown -ssl basic
```

Similarly, if you have `peerAuthentication` on both sides, use the following command:

```
java -Djavax.net.ssl.keyStore=clientKeyStore.key \
-Djavax.net.ssl.keyStorePassword=qwerty \
```

```
-Djavax.net.ssl.trustStore=clientTrustStore.key \
-Djavax.net.ssl.trustStorePassword=qwerty \
-jar derbyrun.jar server shutdown -ssl peerAuthentication
```

Understanding identity in Derby

Derby provides two kinds of identity, *system-wide identity* and *database-specific identity*.

- System-wide identity: Currently, any legal system-wide identity enjoys authorization to perform the following operations:
  - Create databases
  - Restore databases
  - Shut down the Derby engine
- Database-specific identity: If you are a legal identity in a specific database, you may enjoy the following rights:
  - You can connect to that database, provided that coarse-grained connection authorization has not been set to `noAccess`.
  - You can shut down that database, encrypt it, and upgrade it, provided that you are the Database Owner.
  - You can create your own SQL objects and write data to your own tables, provided that your coarse-grained connection authorization has not been set to `readOnlyAccess`.
  - You can access other SQL objects, provided that the owners have granted you fine-grained SQL access to those objects, and provided you have not been limited by coarse-grained `readOnlyAccess`.

The distinction between fine-grained SQL authorization and coarse-grained connection authorization is described in Configuring user authorization.

Users and authorization identifiers

User names within the Derby system are known as *authorization identifiers*. The authorization identifier is a string that represents the name of the user, if one was provided in the connection request.

For example, the built-in function CURRENT_USER returns the authorization identifier for the current user.

Once the authorization identifier is passed to the Derby system, it becomes an *`SQL92Identifier`*. An *`SQL92Identifier`* -- the kind of identifier that represents database objects such as tables and columns -- is case-insensitive (it is converted to all caps) unless it is delimited with double quotes, is limited to 128 characters, and has other limitations.

User names must be valid authorization identifiers even if user authentication is turned off, and even if all users are allowed access to all databases.

For more information about *`SQL92Identifiers`*, see the *Derby Reference Manual*.

Authorization identifiers, user authentication, and user authorization

When working with both user authentication and user authorization, you need to understand how user names are treated by each system.

If you use an external authentication system such as LDAP, the conversion of the user's name to an authorization identifier happens *after* authentication has occurred but *before* user authorization has occurred. Imagine, for example, a user named Fred.

- Within the user authentication system, Fred is known as `FRed`. Your external user authorization service is case-sensitive, so Fred must always type his name that way.

```
Connection conn = DriverManager.getConnection(
    "jdbc:derby:myDB", "FRed", "flintstone");
```

- Within the Derby user authorization system, Fred becomes a case-insensitive authorization identifier. Fred is known as `FRED`.

Let's take a second example, where Fred has a slightly different name within the user authentication system.

- Within the user authentication system, Fred is known as `Fred!`. You must now put double quotes around the name, because it is not a valid *`SQL92Identifier`*. (Derby knows to remove the double quotes when passing the name to the external authentication system.)

  ```
  Connection conn = DriverManager.getConnection(
      "jdbc:derby:myDB", "\"Fred!\"", "flintstone");
  ```
- Within the Derby user authorization system, `Fred` becomes a case-sensitive authorization identifier. Fred is known as `Fred!`.

As shown in the first example, your external authentication system may be case-sensitive, whereas the authorization identifier within Derby may not be. If your authentication system allows two distinct users whose names differ by case, delimit all user names within the connection request to make all user names case-sensitive within the Derby system. In addition, you must also delimit user names that do not conform to *`SQL92Identifier`* rules with double quotes.

User names and schemas

User names can affect a user's default schema.

For information about user names and schemas, see "SET SCHEMA statement" in the *Derby Reference Manual.*

Exceptions when using authorization identifiers

Specifying an invalid authorization identifier in a database user authorization property raises an exception. Specifying an invalid authorization identifier in a connection request raises an exception.

Database Owner

The term *Database Owner* refers to the current authorization identifier when the database is created, that is, the user creating the database. If you use NATIVE authentication, or if you manually enable or plan to enable SQL authorization, controlling the identity of the Database Owner becomes important.

When a database is created, the Database Owner of that database is implicitly set to the authorization identifier used in the connect operation that creates the database, for example, by supplying the URL attribute "user". Note that this applies even if authentication is not (yet) enabled. In SQL, the built-in functions USER and the equivalent CURRENT_USER return the current authorization identifier.

If the database is created *without* supplying a user (this is possible only if authentication is not enabled), the Database Owner is set to the default authorization identifier, "APP", which is also the name of the default schema. See "SET SCHEMA statement" in the *Derby Reference Manual* for details.

The Database Owner has automatic SQL level permissions when SQL authorization is enabled. For more information, see Configuring fine-grained user authorization.

To further enhance security, when *both* authentication and SQL authorization are enabled for a database, Derby restricts some special powers to the Database Owner: only the Database Owner is allowed to shut down the database, to encrypt or reencrypt

the database, or to perform a full upgrade of the database. These powers cannot be delegated.

Attention: There is currently no way of changing the Database Owner once the database is created. This means that if you plan to run with SQL authorization enabled, you should make sure to create the database as the user you want to be the owner.

Configuring user authentication

By default, Derby runs without any credentials checking. This situation may be fine for many shrink-wrapped, embedded applications. However, it means that anyone can connect to an unsecured database and steal or corrupt the data there. Fortunately, it's easy to frustrate these attacks by requiring authentication.

Derby provides support for user authentication and user authorization. *User authentication* determines whether a user is a valid user. It establishes the user's identity. *User authorization* determines what operations a user's established identity can perform. You are strongly urged to implement both authentication and authorization on any multi-user database used in production.

When user authentication is enabled, the user that requests a connection must provide a valid name and password, which Derby verifies against the repository of users defined for the system. After Derby authenticates the user as valid, user authorization determines what operations the user can perform on the database to which the user is requesting a connection.

Derby supports three kinds of authentication schemes:

LDAP
: In this scheme, the customer points Derby at an external LDAP directory service. The customer manages users with the external LDAP service, and Derby retrieves credentials from LDAP. See Configuring LDAP authentication for more information.

NATIVE
: In this scheme, user names and passwords are stored in a Derby database. See Configuring NATIVE authentication for details.

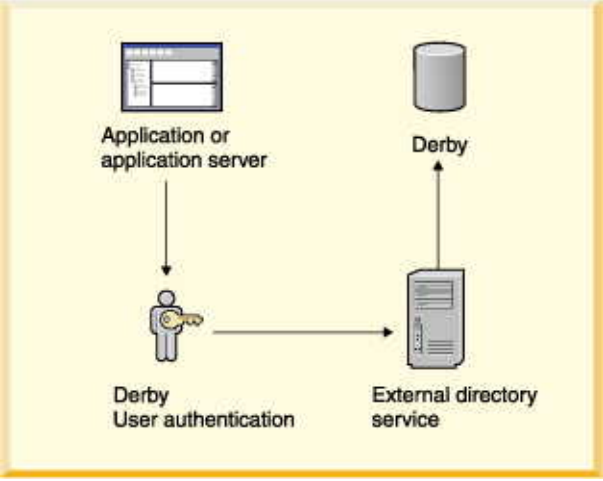
User-defined
: In this scheme, the customer provides all of the logic needed to authenticate users. See Specifying authentication with a user-defined class for more information.

You can define a repository of users for a particular database or for an entire system, depending on whether you use system-wide or database-wide properties.

A directory service stores names and attributes of those names. A typical use for a directory service is to store user names and passwords for a computer system. Derby uses the Java Naming and Directory Interface (JNDI) to interact with external directory services that can provide authentication of users' names and passwords.

When Derby user authentication is enabled and Derby uses an external directory service, the architecture looks something like that shown in the following figure. The application can be a single-user application with an embedded Derby engine or a multi-user application server.

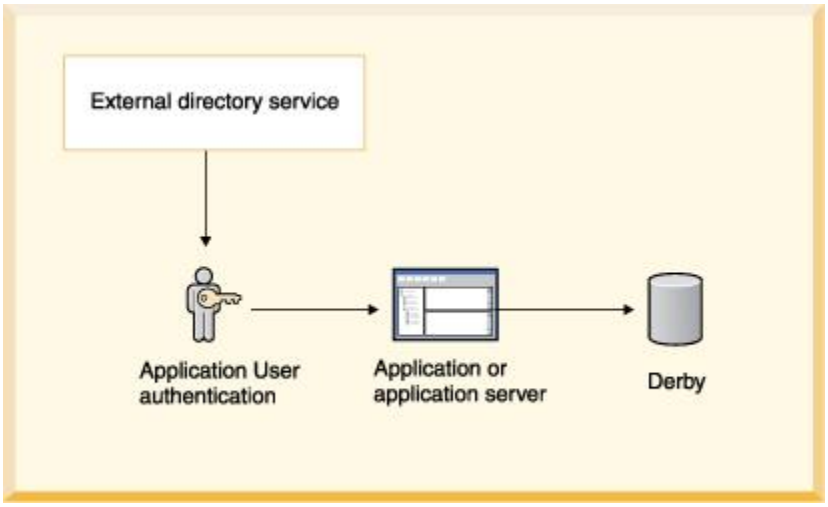
Figure 3. Derby user authentication using an external service



Derby always runs embedded in another Java application, whether that application is a single-user application or a multiple-user application server or connectivity framework.

A database can be accessed by only one JVM at a time, so it is possible to deploy a system in which the application in which Derby is embedded, not Derby, handles the user authentication by connecting to an external directory service. The application can be a single-user application with an embedded Derby engine or a multi-user application server. The following figure shows this kind of deployment.

Figure 4. Application user authentication using an external service



Configuring LDAP authentication

You can allow Derby to authenticate users against an existing LDAP directory service within your enterprise. LDAP (lightweight directory access protocol) provides an open directory access protocol running over TCP/IP. An LDAP directory service can quickly authenticate a user's name and password.

The runtime library provided with the Java Development Kit (JDK) includes libraries that allow you to access an LDAP directory service. See the API documentation for the `javax.naming.ldap` package at http://docs.oracle.com/javase/8/docs/api/, the LDAP section of the JNDI tutorial at http://docs.oracle.com/javase/tutorial/jndi/ldap/,

and the LDAP section of the JNDI specification at http://docs.oracle.com/javase/1.5.0/docs/guide/jndi/spec/jndi/jndi.5.html#pgfId=999241.

To use an LDAP directory service, set `derby.authentication.provider` to `LDAP`.

Booting an LDAP server

To begin, launch the OpenDS QuickSetup JNLP (Java Web Start) installer, then follow the installation steps to set up your directory server.

You can obtain the installer from https://opends.java.net/ by clicking the "Install with QuickSetup" link.

As part of this installation, you will specify a password, which we will call *YOUR_SELECTED_PASSWORD*.

Next, load some credentials into the directory server. Download this sample file of credentials: http://today.java.net/today/2007/03/22/secArticle.LDIF. Now load it into your directory server using the `import-ldif` tool in the `bin` directory of your OpenDS installation. (Make sure that OpenDS is not running when you import credentials; otherwise you will receive an error message indicating that the import utility cannot acquire a lock over storage.)

```
import-ldif --backendID userRoot --ldifFile secArticle.LDIF
```

Now bring up the OpenDS server by running the `start-ds` script in the `bin` directory of your OpenDS installation.

Setting up Derby to use your LDAP directory service

When specifying LDAP as your authentication service, you must specify what LDAP server to use.

To connect to the OpenDS LDAP server, add the following lines to your Derby configuration file, `derby.properties`. You may also want to store these properties in your database and lock them down by setting the `derby.database.propertiesOnly` property (see Configuring coarse-grained user authorization for an example of how to lock down database properties):

```
derby.connection.requireAuthentication=true
derby.authentication.server=ldap://127.0.0.1:1389
derby.authentication.provider=LDAP
derby.authentication.ldap.searchAuthPW=YOUR_SELECTED_PASSWORD
derby.authentication.ldap.searchAuthDN=cn=Directory Manager
derby.authentication.ldap.searchBase=dc=example,dc=com
derby.authentication.ldap.searchFilter=objectClass=person
```

Finally, start `ij` in the directory where you created your `derby.properties` (this ensures that embedded Derby will come up with the authentication settings listed above). Run the following commands:

```
java org.apache.derby.tools.ij
ij version 10.11
ij> connect 'jdbc:derby:testdb;create=true;user=tquist;password=tquist';
```

Verify that authentication works by trying to connect again, this time with bad credentials:

```
java org.apache.derby.tools.ij
ij version 10.11
ij> connect
 'jdbc:derby:testdb;create=true;user=tquist;password=badpassword';
ERROR 08004: Connection authentication failure occurred. Reason: Invalid
 authentication...
```

When you set the property `derby.authentication.server`, you can specify the LDAP server using just the server name, the server name and its port number separated

by a colon, or an `ldap` URL as shown in the example. If you do not provide a full URL, Derby will by default use unencrypted LDAP. To use SSL encrypted LDAP, specify a URL that starts with `ldaps://`. For details on the `derby.authentication.server` and `derby.authentication.provider` properties, see the *Derby Reference Manual*.

Guest access to search for DNs

In an LDAP system, users are hierarchically organized in the directory as a set of entries. An *entry* is a set of name-attribute pairs identified by a unique name, called a DN (distinguished name).

An entry is unambiguously identified by a DN, which is the concatenation of selected attributes from each entry in the tree along a path leading from the root down to the named entry, ordered from right to left. For example, a DN for a user might look like this:

```
cn=mary,ou=People,o=example.com

uid=mary,ou=People,o=example.com
```

The allowable entries for the name are defined by the entry's `objectClass`.

An LDAP client can bind to the directory (successfully log in) if it provides a user ID and password. The user ID must be a DN, the fully qualified list of names and attributes. This means that the user must provide a very long name.

Typically, the user knows only a simple user name (for example, the first part of the DN above, `mary`). With Derby, you do not need the full DN, because an LDAP client (Derby) can go to the directory first as a guest or even an anonymous user, search for the full DN, then rebind to the directory using the full DN (and thus authenticate the user).

Derby typically initiates a search for a full DN before binding to the directory using the full DN for user authentication. Derby does not initiate a search in the following cases:

- You have set `derby.authentication.ldap.searchFilter` to `derby.user`.
- A user DN has been cached locally for the specific user with the `derby.user.UserName` property.

For more information, see "`derby.authentication.ldap.searchFilter`" in the *Derby Reference Manual*.

Some systems permit anonymous searches; others require a user DN and password. You can specify a user's DN and password for the search with the properties listed below. In addition, you can limit the scope of the search by specifying a filter (definition of the object class for the user) and a base (directory from which to begin the search) with the properties listed below.

- `derby.authentication.ldap.searchAuthDN (optional)`

  Specifies the DN with which to bind (authenticate) to the server when searching for user DNs. This parameter is optional if anonymous access is supported by your server. If specified, this value must be a DN recognized by the directory service, and it must also have the authority to search for the entries.

  If not set, it defaults to an anonymous search using the root DN specified by the `derby.authentication.ldap.searchBase` property. For example:

  ```
  uid=guest,o=example.com
  ```
- `derby.authentication.ldap.searchAuthPW` (optional)

  Specifies the password to use for the guest user configured above to bind to the directory service when looking up the DN. If not set, it defaults to an anonymous search using the root DN specified by the `derby.authentication.ldap.searchBase` property.

```
myPassword
```

- `derby.authentication.ldap.searchBase` (optional)

  Specifies the root DN of the point in your hierarchy from which to begin a guest search for the user's DN. For example:

  ```
  ou=people,o=example.com
  ```

To narrow the search, you can specify a user's `objectClass`.

- `derby.authentication.ldap.searchFilter` (optional)

  Set `derby.authentication.ldap.searchFilter` to a logical expression that specifies what constitutes a user for your LDAP directory service. The default value of this property is `objectClass=inetOrgPerson`. For example:

  ```
  objectClass=person
  ```

See the *Derby Reference Manual* for details on all these properties.

LDAP performance issues

For performance reasons, the LDAP directory server should be in the same LAN as Derby. Derby does not cache the user's credential information locally and thus must connect to the directory server every time a user connects.

Connection requests that provide the full DN are faster than those that must search for the full DN.

LDAP restrictions

Derby does not support LDAP groups.

JNDI-specific properties for external directory services

Derby allows you to set a few advanced JNDI properties, which you can set in any of the supported ways of setting Derby properties. Typically you would set these at the same level (database or system) for which you configured the external authentication service.

The list of supported properties can be found in "Appendix A: JNDI Standard Environment Properties" in the Java Naming and Directory API at http://docs.oracle.com/javase/1.5.0/docs/guide/jndi/spec/jndi/properties.html. The external directory service must support the property.

Each JNDI provider has its set of properties that you can set within the Derby system.

For example, you can set the property `java.naming.security.authentication` to allow user credentials to be encrypted on the network if the provider supports it. You can also specify that SSL be used with LDAP (LDAPS).

Configuring NATIVE authentication

Derby's simplest authentication mechanism is NATIVE authentication.

When you use NATIVE authentication, user names and encrypted passwords are stored in a database. You can specify a dedicated credentials database for this purpose, or you can store the credentials in the same database you use for your application data. The credentials are stored in the SYSUSERS system table of the database.

To configure NATIVE authentication, follow these steps.

1. Use the `SYSCS_UTIL.SYSCS_CREATE_USER` system procedure to add credentials for the Database Owner. Remember that the Database Owner is the user who created the database.
2. Add credentials for other users.

3. Shut down the database, then reboot it. When the database reboots, NATIVE authentication is enabled.

For example, you can issue the following commands:

```
java org.apache.derby.tools.ij
ij version 10.11
ij> connect 'jdbc:derby:testdb;create=true;user=tquist';
ij> -- the Database Owner must be the first user you create
call SYSCS_UTIL.SYSCS_CREATE_USER( 'tquist', 'tquist' );
0 rows inserted/updated/deleted
ij> -- now add other users
call SYSCS_UTIL.SYSCS_CREATE_USER( 'thardy', 'thardy' );
0 rows inserted/updated/deleted
ij> call SYSCS_UTIL.SYSCS_CREATE_USER( 'jhallett', 'jhallett' );
0 rows inserted/updated/deleted
ij> call SYSCS_UTIL.SYSCS_CREATE_USER( 'mchrysta', 'mchrysta' );
0 rows inserted/updated/deleted
ij> -- shut down the database in order to turn on NATIVE authentication
connect 'jdbc:derby:testdb;shutdown=true';
ERROR 08006: Database 'testdb' shutdown.
ij> -- these connection attempts fail because of bad credentials
connect 'jdbc:derby:testdb;user=tquist';
ERROR 08004: Connection authentication failure occurred.  Reason: Invalid
 authentication..
ij> connect 'jdbc:derby:testdb;user=thardy;password=tquist';
ERROR 08004: Connection authentication failure occurred.  Reason: Invalid
 authentication..
ij> -- these connection attempts present good credentials, so they
 succeed
connect 'jdbc:derby:testdb;user=tquist;password=tquist';
ij(CONNECTION1)> connect 'jdbc:derby:testdb;user=thardy;password=thardy';
ij(CONNECTION2)> connect
 'jdbc:derby:testdb;user=jhallett;password=jhallett';
ij(CONNECTION3)> connect
 'jdbc:derby:testdb;user=mchrysta;password=mchrysta';
```

Enabling NATIVE authentication explicitly

You can turn on NATIVE authentication explicitly by using a property.

To do so, specify one of the following values for the `derby.authentication.provider` property:

- NATIVE:*credentialsDB*

  This value tells Derby to use *credentialsDB*, a dedicated database, to store user credentials. This value must be set by using system-wide Java Virtual Machine (JVM) properties or by using the `derby.properties` file; it cannot be set in the database by using the `SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY` procedure. When this system-wide value is set, *credentialsDB* is used to authenticate all operations. If an individual database holds credentials for the Database Owner, the global credentials database is used only to authenticate system-wide operations such as engine shutdown.

  The value of *credentialsDB* must be a valid name for a database.
- NATIVE:*credentialsDB*:LOCAL

  This value tells Derby to use *credentialsDB* for system-wide operations, but to use an individual database's SYSUSERS system table to authenticate connections to that database. This value must be set by using system-wide JVM properties or by using the `derby.properties` file; it cannot be set in the database by using the `SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY` system procedure.

See the *Derby Reference Manual* for details on the `derby.authentication.provider` property.

Working with a credentials database

With NATIVE authentication, a database can become a credentials database in any of several ways.

- When the database is being created, it is identified as the credentials database by the system-level property setting `derby.authentication.provider=NATIVE:`*`credentialsDB`*.
- When the database is being created, LOCAL authentication of connections is specified by the system-level property setting `derby.authentication.provider=NATIVE:`*`credentialsDB`*`:LOCAL`.
- When the database already exists, the Database Owner calls the `SYSCS_UTIL.SYSCS_CREATE_USER` system procedure to store the Database Owner's credentials in the database. If the Database Owner calls this procedure to store another user's credentials first, an error is raised.

When a database becomes a credentials database, the following things happen:

- The value of `derby.authentication.provider=NATIVE::LOCAL` is stored in the database, marking it as a credentials database.
- From this point forward, the value of `derby.authentication.provider` cannot be overridden or changed for connections to this database.
- If the database is being newly created, the Database Owner's credentials (provided in the connection arguments) are stored in the database's SYSUSERS system table.
- All future connections to the database are authenticated against the credentials in its SYSUSERS system table.

NATIVE authentication and other database properties

When NATIVE authentication is enabled, Derby behaves as if the `derby.connection.requireAuthentication` and `derby.database.sqlAuthorization` properties are also set.

That is, a user name and password must be specified whenever a user connects to a database, and object owners control access to database objects. See Configuring fine-grained user authorization for more information, and see NATIVE authentication and SQL authorization example for an example of the use of NATIVE authentication.

For maximum security, the passwords that users specify when they connect to databases have an expiration date that you can modify by using the property `derby.authentication.native.passwordLifetimeMillis`. The password of the Database Owner never expires. By default, ordinary user passwords expire after 31 days.

If a password is about to expire, or if the Database Owner's password is near what would be the expiration date, Derby issues a warning that the password will soon expire (or, in the Database Owner's case, that the password is stale). By default, the warning is issued if the password is due to expire in one-eighth of the password's lifetime. For example, if the password has a 31-day lifetime, the warning will be issued 3.875 days before the expiration date. You can change this proportion by using the property `derby.authentication.native.passwordLifetimeThreshold`.

Use the `derby.authentication.builtin.algorithm` property to change the way passwords are encrypted when they are stored in the SYSUSERS system table. The default algorithm is SHA-256. Two related properties are `derby.authentication.builtin.saltLength` and `derby.authentication.builtin.iterations`, which can be used to make the hashed passwords harder for attackers to crack.

See the *Derby Reference Manual* for details on these properties.

Managing users and passwords

To manage users and passwords, Derby provides a group of system procedures.

- To create users for a database, the Database Owner calls `SYSCS_UTIL.SYSCS_CREATE_USER`, which takes a user name and password as arguments. This procedure can also be executed by a user or role to which the Database Owner has granted sufficient privileges.
- To remove a user, the Database Owner calls `SYSCS_UTIL.SYSCS_DROP_USER`, which takes one argument, the user name of the user. This procedure can also be executed by a user or role to which the Database Owner has granted sufficient privileges.
- To reset a forgotten or expired password, the Database Owner calls `SYSCS_UTIL.SYSCS_RESET_PASSWORD`, with a user name and password as arguments. This procedure can also be executed by a user or role to which the Database Owner has granted sufficient privileges.
- To change a user's own password, any user can call the system procedure `SYSCS_UTIL.SYSCS_MODIFY_PASSWORD`, which takes only one argument, the password. Typically, a user calls this procedure when their password is about to expire.

See the *Derby Reference Manual* for details on these procedures.

Converting an existing database to use NATIVE authentication

If you wish to apply NATIVE authentication to a database that was created without it, the procedure is slightly different depending on whether you specify NATIVE:*credentialsDB* or NATIVE:*credentialsDB*:LOCAL.

- If you specify NATIVE:*credentialsDB*, add users of the existing database to the *credentialsDB*. For instance, if the old database was created without any authentication, then its default user name is APP, and you could do the following:

```
CALL SYSCS_UTIL.SYSCS_CREATE_USER('app', 'app');
```

- If you plan to specify NATIVE:*credentialsDB*:LOCAL, first connect to the existing database as its Database Owner using its old authentication scheme. Call `SYSCS_UTIL.SYSCS_CREATE_USER` to add credentials for the Database Owner. For example, if the existing database was created with no authentication, the Database Owner is APP, and you would add credentials for APP as shown above.

Specifying authentication with a user-defined class

You can set the `derby.authentication.provider` property to the full name of a class that implements the public interface `org.apache.derby.authentication.UserAuthenticator`.

By writing your own class that fulfills some minimal requirements, you can hook Derby up to an external authentication service other than LDAP. To do so, specify an external authentication service by setting the property `derby.authentication.provider` to a class name that you want Derby to load at startup.

The class that provides the external authentication service must implement the public interface `org.apache.derby.authentication.UserAuthenticator` and throw exceptions of the type `java.sql.SQLException` where appropriate.

Using a user-defined class makes Derby adaptable to various naming and directory services.

Example of setting a user-defined class

This is a very simple example of a class that implements the `org.apache.derby.authentication.UserAuthenticator` interface.

```
import org.apache.derby.authentication.UserAuthenticator;
import java.io.FileInputStream;
import java.util.Properties;
import java.sql.SQLException;
/**
  * A simple example of a specialized Authentication scheme.
  * The system property 'derby.connection.requireAuthentication'
  * must be set to true, and 'derby.authentication.provider' must
  * contain the full class name of the overridden authentication
  * scheme, (that is, the name of this class).
  *
  * @see org.apache.derby.authentication.UserAuthenticator
  */
public class MyAuthenticationSchemeImpl implements
        UserAuthenticator {
    private static final String USERS_CONFIG_FILE = "myUsers.cfg";
    private static Properties usersConfig;

    // Constructor
    // We get passed some Users properties if the
    // authentication service could not set them as
    // part of the System properties.
    //
    public MyAuthenticationSchemeImpl() {
    }

    /* Static block where we load the users definition from a
       users configuration file. */
    static {
        /* Load users config file as Java properties.
           File must be in the same directory where
           Derby is started.
           Otherwise, full path must be specified. */
        FileInputStream in = null;
        usersConfig = new Properties();
        try {
            in = new FileInputStream(USERS_CONFIG_FILE);
            usersConfig.load(in);
            in.close();
        } catch (java.io.IOException ie) {
            // No Config file. Raise error message
            System.err.println(
                "WARNING: Error during Users Config file retrieval");
            System.err.println("Exception: " + ie);
        }
    }

    /**
     * Authenticate the passed-in user's credentials.
     * A more complex class could make calls
     * to any external users directory.
     *
     * @param userName              The user's name
     * @param userPassword          The user's password
     * @param databaseName          The database
     * @param info                  Additional jdbc connection info.
     * @exception SQLException on failure
     */
    public boolean authenticateUser(String userName,
            String userPassword,
            String databaseName,
            Properties info)
        throws SQLException {
        /* Specific Authentication scheme logic.
           If user has been authenticated, then simply return.
           If user name and/or password are invalid,
           then raise the appropriate exception.

           This example allows only users defined in the
           users config properties object.
```

```
            Check if the passed-in user has been defined for the system.
            We expect to find and match the property corresponding to
            the credentials passed in. */
        if (userName == null)
            // We do not tolerate 'guest' user for now.
            return false;

        /* Check if user exists in our users config (file)
            properties set.
            If we did not find the user in the users config set, then
            try to find if the user is defined as a System property. */
        String actualUserPassword;
        actualUserPassword = usersConfig.getProperty(userName);
        if (actualUserPassword == null)
            actualUserPassword = System.getProperty(userName);
        if (actualUserPassword == null)
            // No such passed-in user found
            return false;
        // Check if the password matches
        if (!actualUserPassword.equals(userPassword))
            return false;
        // Now, check if the user is a valid user of the database
        if (databaseName != null) {
         /* If database users restriction lists are present, then
            check if there is one for this database and if so,
            check if the user is a valid one for that database.
            For this example, the only user we authorize in database
            DarkSide is user 'DarthVader'. This is the only database
            users restriction list we have for this example.
            We authorize any valid (login) user to access the
            OTHER databases in the system.
            Note that database users ACLs could be set in the same
            properties file or a separate one and implemented as you
            wish. */
            if (databaseName.equals("DarkSide")) {
                // Check if user is a valid one
                if (!userName.equals("DarthVader"))
                    // This user is not a valid one of the passed-in
                    return false;
            }
        }
        // The user is a valid one in this database
        return true;
    }
}
```

List of user authentication properties

The following table summarizes the Derby properties related to user authentication.

For details on these properties, see the *Derby Reference Manual*.

Table 3. User authentication properties

Property Name	Use
`derby.authentication.provider`	Specifies the kind of user authentication to use.
`derby.authentication.builtin.algori`	Specifies the message digest algorithm to use to protect the passwords that are stored in the database when using NATIVE authentication.

Property Name	Use
derby.authentication.builtin.iterat	Specifies the number of times to apply the hash function specified by the message digest algorithm.
derby.authentication.builtin.saltLe	Specifies the number of bytes of random salt that will be added to users' credentials before hashing them.
derby.authentication.native.passwor	Specifies the number of milliseconds that a password used for NATIVE authentication remans valid.
derby.authentication.native.passwor	Specifies the threshold that triggers a password-expiration warning for NATIVE authentication.
derby.connection.requireAuthenticat	Turns on user authentication. If NATIVE authentication is used, Derby behaves as if this property is set to TRUE.
derby.authentication.server	For LDAP user authentication, specifies the location of the server.
derby.authentication.ldap.searchAut derby.authentication.ldap.searchAut derby.authentication.ldap.searchFil and derby.authentication.ldap.search	Configures the way that DN searches are performed.
derby.user.UserName	Caches user DNs locally for LDAP authentication when derby.authentication.ldap.s is set to derby.user.
java.naming.*	JNDI properties. See Appendix A in the JNDI API reference (http://docs.oracle.com/javase/1.5.0/docs/guide/jndi/spec/jndi/properties.html) for more information about these properties.

Programming applications for Derby user authentication

To program user authentication into Derby applications, use the `DriverManager.getConnection` call to specify the user name and password.

An application can provide the user name and password in the following ways.

- Separately, as arguments to the following signature of the method: `getConnection(String url, String user, String password)`

```
Connection conn = DriverManager.getConnection(
    "jdbc:derby:myDB", "mary", "little7xylamb");
```

- As attributes to the database connection URL:

```
Connection conn = DriverManager.getConnection(
    "jdbc:derby:myDB;user=mary;password=little7xylamb");
```

- By setting the `user` and `password` properties in a `Properties` object as with other connection URL attributes:

```
Properties p = new Properties();
p.put("user", "mary");
p.put("password", "little7xylamb");
Connection conn = DriverManager.getConnection(
    "jdbc:derby:myDB", p);
```

Note: The password is not encrypted. When you are using Derby in the context of a server framework, the framework should be responsible for encrypting the password across the network. If your framework does not encrypt the password, it is strongly recommended that you protect network connections with SSL/TLS (see Configuring SSL/TLS).

For information about the treatment of user names within the Derby system, see Users and authorization identifiers.

Login failure exceptions with user authentication

If user authentication is turned on and a valid user name and password are not provided, *SQLException* 08004 is raised.

```
ERROR 08004: Connection refused : Invalid authentication.
```

Configuring Network Server authentication in special circumstances

Some advanced Network Server configuration features may be useful in certain situations.

Configuring Network Client authentication without SSL/TLS

If you do not encrypt network traffic with SSL/TLS, you can use properties to specify the encryption of user names and passwords on the client side.

> Important: Using SSL/TLS is strongly recommended for production applications. Use the properties only under unusual circumstances.

The `securityMechanism=value` property specifies a security mechanism for the Derby Network Client. See the *Derby Reference Manual* for details on this property.

You can set the `securityMechanism` property in one of the following ways:

- When you are using the `java.sql.DriverManager` class, set `securityMechanism=`*`value`* in a `java.util.Properties` object before you invoke the form of the `DriverManager.getConnection` method that includes the `java.util.Properties` parameter.
- When you are using the `ClientDataSource` interface to create and deploy your own DataSource objects, invoke the `ClientDataSource.setSecurityMechanism` method after you create a ClientDataSource object.

The following table lists the security mechanisms that the Derby Network Client supports, and the corresponding property value to specify to obtain this security mechanism. The default security mechanism is the user id only if no password is set. If the password is set, the default security mechanism is both the user id and password. The default user is APP if no other user is specified.

Table 4. Security mechanisms supported by the Derby Network Client

Security Mechan	securityMechanism Property Value	Comments
User id and password	`ClientDataSource.CLEAR_TEXT_PASSWORD_` (0x03)	Default if password is set

Security Mechan	securityMechanism Property Value	Comments
User id only	`ClientDataSource.USER_ONLY_SECURITY` (0x04)	Default if password is not set
Encrypted user id and encrypted password	`ClientDataSource.ENCRYPTED_USER_AND_P` (0x09)	Encryption requires a JCE implementation that supports the Diffie-Hellman algorithm with a public prime of 256 bits.

Derby provides two ClientDataSource implementations. Use the `org.apache.derby.jdbc.ClientDataSource` class on all supported Java SE versions except Java SE 8 Compact Profile 2. On Java SE 8 Compact Profile 2, use the `org.apache.derby.jdbc.BasicClientDataSource40` class.

Configuring Network Server authentication without SSL/TLS

If you do not encrypt network traffic with SSL/TLS, you can use properties to specify the encryption of user names and passwords on the Network Server side.

> Important: Using SSL/TLS is strongly recommended for production applications. Use the properties only under unusual circumstances.

When you run Derby in embedded mode or when you use the Derby Network Server, you can enable or disable server-side user authentication. (Enabling user authentication is strongly recommended.) However, when you use the Network Server, the default security mechanism (`CLEAR_TEXT_PASSWORD_SECURITY`) requires that you supply both the user name and password.

In addition to the default user name and password security mechanism, `org.apache.derby.jdbc.ClientDataSource.CLEAR_TEXT_PASSWORD_SECURITY`, Derby Network Server supports the following security properties:

- UserID: `org.apache.derby.jdbc.ClientDataSource.USER_ONLY_SECURITY`

  When you use this mechanism, you must specify only the `user` property. All other mechanisms require you to specify both the user name and the password.
- Encrypted UserID and encrypted password: `org.apache.derby.jdbc.ClientDataSource.ENC`

  When you use this mechanism, both password and user id are encrypted.

The user name that is specified upon connection is the default schema for the connection, if a schema with that name exists. See the *Derby Developer's Guide* for more information on schema and user names.

If you specify any other security mechanism, you will receive an exception.

To change the default, you can specify another security mechanism either as a property or on the URL (using the `securityMechanism=`*`value`* attribute) when you make the connection. For details, see Configuring Network Client authentication without SSL/TLS and "securityMechanism=value attribute" in the *Derby Reference Manual*.

Whether the security mechanism you specify for the client actually takes effect depends upon the setting of the `derby.drda.securityMechanism` property for the Network Server. If the `derby.drda.securityMechanism` property is set, the Network Server accepts only connections that use the security mechanism specified by the property setting. If the `derby.drda.securityMechanism` property is not set, clients can use

any valid security mechanism. For details, see "derby.drda.securityMechanism property" in the *Derby Server and Administration Guide*.

Security mechanism options when user authentication is enabled on the Network Server

When user authentication is enabled in Derby, you can use either of the following security mechanisms.

- Clear text user name and password security, the default
- Encrypted user name and password security

Security mechanism options when user authentication is disabled on the Network Server

When user authentication is turned off in Derby, you can use any of the security mechanism options.

You must provide a user and password for all security mechanisms except `USER_ONLY_SECURITY`. However, because user authentication is disabled in the Derby server, the user name and password that you supply do not have to be among those recognized as valid by Derby.

Enabling the encrypted user ID and password security mechanism

To use the encrypted user ID and password security mechanism, you need a Java environment with a JCE (Java Cryptography Extension) that supports the Diffie-Hellman algorithm with a public prime of 256 bits.

The Java Platform, Standard Edition (Java SE) requires a public prime of 512 bits or more.

To use the encrypted user id and password security mechanism during JDBC connection using the network client, specify the `securityMechanism=`*`value`* connection property. **Note:** If an encrypted database is booted in the Network Server, users can connect to the database without giving the `bootPassword`. The first connection to the database must provide the `bootPassword`, but all subsequent connections do not need to supply it. To remove access from the encrypted database, use the `shutdown=true` option to shut down the database. See Configuring database encryption for more information.

Configuring user authorization

While authentication determines whether someone is a legal database user, *authorization* determines what operations can be performed by a user's identity.

Once you have set up authentication, you can configure authorization.

Derby offers two kinds of authorization:

- **Coarse-grained authorization**, in which the Database Owner divides an application's users into two groups. One group has full authority to read and write all data. The other group merely has permission to read data.
- **Fine-grained authorization**, in which the Database Owner and individual users issue SQL GRANT/REVOKE statements to declare who can read or write specific pieces of data and who can exercise specific application functions.

Configuring coarse-grained user authorization

You can manipulate coarse-grained access by using the builtin procedure SYSCS_SET_DATABASE_PROPERTY to set the database properties `derby.database.fullAccessUsers` and `derby.database.readOnlyAccessUsers`.

The following example shows how to do this. The example assumes that you are reusing the credentials-protected database you created in Configuring NATIVE authentication. The example commands first set the read/write and read-only users and then verify that the settings work correctly.

```
java org.apache.derby.tools.ij
ij> ij version 10.11
ij> connect 'jdbc:derby:testdb;user=tquist;password=tquist';
ij> --
-- Prevent our settings from being overridden on the
-- command line or in derby.properties.
--
call SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY
( 'derby.database.propertiesOnly', 'true' );
0 rows inserted/updated/deleted
ij> --
-- Now we can configure read/write and read-only users.
--
call SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY
( 'derby.database.fullAccessUsers', 'tquist,mchrysta' );
0 rows inserted/updated/deleted
ij> call SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY
( 'derby.database.readOnlyAccessUsers', 'thardy,jhallett' );
0 rows inserted/updated/deleted
ij> --
-- Next verify that a read/write user has those powers:
--
connect 'jdbc:derby:testdb;user=mchrysta;password=mchrysta';
ij(CONNECTION1)> create table mchrysta.t1( a varchar( 20 ) );
0 rows inserted/updated/deleted
ij(CONNECTION1)> insert into mchrysta.t1( a ) values ( 'mchrysta' );
1 row inserted/updated/deleted
ij(CONNECTION1)> select * from mchrysta.t1;
A
--------------------
mchrysta

1 row selected
ij(CONNECTION1)> --
-- Finally, verify that a read-only user can read data but not write it:
--
connect 'jdbc:derby:testdb;user=thardy;password=thardy';
ij(CONNECTION2)> -- the user can select from public data
select count(*) from sys.systables;
1
-----------
24

1 row selected
ij(CONNECTION2)> -- but this user can't even create a table
create table thardy.t1( a varchar( 20 ) );
ERROR 25503: DDL is not permitted for a read-only connection, user or
 database.
```

Coarse-grained authorization details

Use a CALL statement to call the SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY system procedure.

To specify multiple user IDs, use a comma-separated list, with no spaces between the comma and the next user ID.

To specify read-write access for a user ID that contains special characters, use delimited identifiers for the user ID. For example:

```
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.fullAccessUsers', '"Elena!"')
```

For extra security, you should configure the `derby.database.propertiesOnly` property so that users cannot override database behavior using system-wide properties specified on the command line or in the `derby.properties` file.

Read-only and full access permissions

The actions that users can perform on a Derby database with coarse-grained authorization are determined by the type of access that users have to the database objects.

The following table lists the actions that users can perform based on the type of access that a user is granted on a database.

Table 5. Actions that are authorized by type of access

Action	Read-only access	Full access
Executing SELECT statements	Yes	Yes
Reading database properties	Yes	Yes
Loading database classes from jar files	Yes	Yes
Executing INSERT, UPDATE, or DELETE statements	No	Yes
Executing DDL statements	No	Yes
Adding or replacing jar files	No	Yes
Setting database properties	No	Yes

Setting the default connection access mode

You can use the `derby.database.defaultConnectionMode` property to specify the default type of access that users have when they connect to the database.

The valid settings for the `derby.database.defaultConnectionMode` property are:

- `noAccess`
- `readOnlyAccess`
- `fullAccess`

If you do not specify a setting for the `derby.database.defaultConnectionMode` property, the default access setting is `fullAccess`.

To set the default connection access mode, specify the access in a CALL statement. For example:

To specify read-write access for the System Administrator user ID `sa` and read-only access as the default for anyone else who connects to the database, issue these CALL statements:

```
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.fullAccessUsers', 'sa')

CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.defaultConnectionMode',
    'readOnlyAccess')
```

To specify read-write access for the user ID Fred and no access for other users, issue these CALL statements:

```
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.fullAccessUsers', 'Fred')

CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.defaultConnectionMode',
```

```
    'noAccess')
```

User authorization exceptions

SQL exceptions are returned when errors occur with coarse-grained user authorizations.

Derby validates the database properties when you set the properties. An exception is returned if you specify an invalid value when you set these properties.

If a user attempts to connect to a database but is not authorized to connect to that database, the `SQLException` 04501 is returned.

If a user with read-only access attempts to write to a database, the `SQLException` 08004 (connection refused) is returned.

Configuring fine-grained user authorization

You can use fine-grained user authorization, also called SQL standard authorization, to restrict access to specific pieces of data.

You can use fine-grained authorization by itself or in conjunction with coarse-grained authorization.

Fine-grained authorization, like coarse-grained authorization, requires that we run Derby with authentication turned on. If you are using LDAP authentication, then you will need to enable fine-grained authorization by setting the `derby.database.sqlAuthorization` property to `true`.

The following example uses the same database you created in Configuring NATIVE authentication, the database that relies on NATIVE authentication. If you use NATIVE authentication, there is no need to set the `derby.database.sqlAuthorization` property. NATIVE authentication automatically enables fine-grained authorization.

The example creates two tables. One table can be viewed by anyone. The other table can be viewed only by specific users.

```
java org.apache.derby.tools.ij
ij version 10.11
ij> connect 'jdbc:derby:testdb;user=mchrysta;password=mchrysta';
ij> -- create and populate some tables
create table publicTable( a int );
0 rows inserted/updated/deleted
ij> create table restrictedTable( a int );
0 rows inserted/updated/deleted
ij> insert into publicTable( a ) values ( 1 );
1 row inserted/updated/deleted
ij> insert into restrictedTable( a ) values( 100 );
1 row inserted/updated/deleted
ij> -- set up fine-grained checks
grant select on publicTable to public;
0 rows inserted/updated/deleted
ij> grant select on restrictedTable to thardy;
0 rows inserted/updated/deleted
ij> --
--Now verify that thardy can view both tables...
--
connect 'jdbc:derby:testdb;user=thardy;password=thardy';
ij(CONNECTION1)> select * from mchrysta.publicTable;
A
-----------
1

1 row selected
ij(CONNECTION1)> select * from mchrysta.restrictedTable;
A
-----------
```

```
100

1 row selected
ij(CONNECTION1)> --
-- ...but other users can only view the public table:
--
connect 'jdbc:derby:testdb;user=jhallett;password=jhallett';
ij(CONNECTION2)> select * from mchrysta.publicTable;
A
-----------
1

1 row selected
ij(CONNECTION2)> select * from mchrysta.restrictedTable;
ERROR 42502: User 'JHALLETT' does not have SELECT permission on column
 'A' of
table 'MCHRYSTA'.'RESTRICTEDTABLE'.
```

You can also use the GRANT command to restrict write access to your tables, to control who executes your functions and procedures, to limit who can add triggers to your tables, and to limit who can create foreign keys referencing your tables. You can also control users' ability to create, set, and drop roles.

Coarse-grained and fine-grained authorization are not mutually exclusive. You may want to configure both. Using coarse-grained authorization, you can prevent truly read-only users from creating and populating any table; this defends your database against an unbounded growth vulnerability (see Vulnerabilities of unsecured databases. Using additional fine-grained authorization checks prevents your read-write users from accessing restricted data.

After the `derby.database.sqlAuthorization` property has been set to `true`, you cannot set the property back to `false`.

You can set the `derby.database.sqlAuthorization` property as a system property or as a database property. If you set this property as a system property before you create the databases, all new databases will automatically have SQL authorization enabled:

```
derby.database.sqlAuthorization=true
```

If the databases already exist, you can set this property only as a database property:

```
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.sqlAuthorization',
    'true')
```

Using fine-grained user authorization

When the SQL standard authorization mode is enabled, object owners can use the GRANT and REVOKE SQL statements to set the user privileges for specific database objects or for specific SQL actions. They can also use roles to administer privileges.

The GRANT statement is used to grant specific privileges to users or to roles, or to grant roles to users or to roles. The REVOKE statement is used to revoke privileges and role grants. The grant and revoke privileges are:

- DELETE
- EXECUTE
- INSERT
- SELECT
- REFERENCES
- TRIGGER
- UPDATE

When a table, view, function, procedure, type, or aggregate is created, the person that creates the object is referred to as the *owner* of the object. Only the object owner and the

Database Owner have full privileges on the object. No other users have privileges on the object until the object owner grants privileges to them.

Another way of saying that privileges on objects belong to the owner is to call them *definer rights*, as opposed to *invoker rights*. This is the terminology used by the SQL standard.

See the *Derby Reference Manual* for more information on the GRANT and REVOKE statements.

Public and individual user privileges

The object owner can grant and revoke privileges for specific users, for specific roles, or for all users.

The keyword PUBLIC is used to specify all users. When PUBLIC is specified, the privileges affect all current and future users. The privileges granted and revoked to PUBLIC and to individual users or roles are independent. For example, suppose that a SELECT privilege on table `t` is granted to both PUBLIC and to the user `harry`. The SELECT privilege is later revoked from user `harry`, but user `harry` has access to table `t` through the PUBLIC privilege.

Exception: When you create a view, trigger, or constraint, Derby first checks to determine if you have the required privileges at the user level. If you have the user-level privileges, the object is created and is dependent on that user-level privilege. If you do not have the required privileges at the user level, Derby checks to determine if you have the required privileges at the PUBLIC level. If you have the PUBLIC level privileges, the object is created and is dependent on that PUBLIC level privilege. After the object is created, if the privilege on which the object depends is revoked, the object is automatically dropped. Derby does not try to determine if you have other privileges that can replace the privileges that are being revoked.

Example 1

User `zhi` creates table `t1` and grants SELECT privileges to user `harry` on table `t1`. User `zhi` grants SELECT privileges to PUBLIC on table `t1`. User `harry` creates view `v1` with the statement `SELECT * from zhi.t1`. The view depends on the user-level privilege that user `harry` has on `t1`. Subsequently, user `zhi` revokes SELECT privileges from user `harry` on table `t1`. As a result, the view `harry.v1` is dropped.

Example 2

User `anita` creates table `t1` and grants SELECT privileges to PUBLIC. User `harry` creates view `v1` with the statement `SELECT * from anita.t1`. The view depends on the PUBLIC level privilege that user `harry` has on `t1`, since user `harry` does not have user-level privileges on table `t1` when he creates the view `harry.v1`. Subsequently, user `anita` grants SELECT privileges to user `harry` on table `anita.t1`. The view `harry.v1` continues to depend on the PUBLIC level privilege that user `harry` has on `t1`. When user `anita` revokes SELECT privileges from PUBLIC on table `t1`, the view `harry.v1` is dropped.

See Privileges on views, triggers, constraints, and generated columns for more information.

Privileges on views, triggers, constraints, and generated columns

Views, triggers, constraints, and generated columns operate with the privileges of the owner of the view, trigger, constraint, or generated column.

For example, suppose that user `anita` wants to create a view using the following statement:

```
CREATE VIEW s.v(vc1,vc2,vc3)
    AS SELECT t1.c1,t1.c2,f(t1.c3)
```

```
    FROM t1 JOIN t2 ON t1.c1 = t2.c1
    WHERE t2.c2 = 5
```

User `anita` needs the following privileges to create the view:

- Ownership of the schema `s`, so that she can create something in the schema
- Ownership of the table `t1`, so that she can allow others to see columns in the table
- SELECT privilege on column `t2.c1` and column `t2.c2`
- EXECUTE privilege on function `f`

When the view is created, only user `anita` has the SELECT privilege on it. User `anita` can grant the SELECT privilege on any or all of the columns of view `s.v` to anyone, even to users that do not have the SELECT privilege on `t1` or `t2`, or the EXECUTE privilege on `f`. User `anita` then grants the SELECT privilege on view `s.v` to user `harry`. When user `harry` issues a SELECT statement on the view `s.v`, Derby checks to determine if user `harry` has the SELECT privilege on view `s.v`. Derby does not check to determine if user `harry` has the SELECT privilege on `t1` or `t2`, or the EXECUTE privilege on `f`.

Privileges on triggers, constraints, and generated columns work the same way as privileges on views. When one of these objects is created, Derby checks that the owner has the required privileges. Other users do not need to have those privileges to perform actions on a view, trigger, constraint, or generated column.

If the required privileges are revoked from the owner of a view, trigger, constraint, or generated column, the object is dropped as part of the REVOKE statement.

Another way of saying that privileges on objects belong to the owner is to call them *definer rights*, as opposed to *invoker rights*. This is the terminology used by the SQL standard.

Using SQL roles

When the SQL standard authorization mode is enabled, object owners can use the SQL roles facility to administer privileges.

SQL roles are useful for administering privileges when a database has many users. Roles provide a more powerful way to grant privileges to users' sessions than to grant privileges to each user of the database, which easily becomes tedious and error-prone when many users are involved. Roles do not in and of themselves give better database security, but used correctly, they facilitate better security. Only the Database Owner can create, grant, revoke, and drop roles. However, object owners can grant and revoke privileges for those objects to and from roles, as well as to and from individual users and PUBLIC (all users).

Note: Derby implements a subset of SQL roles. The fact that only the Database Owner can create, grant, revoke, and drop roles is an implementation restriction.

Creating and granting roles

Roles are available only when SQL authorization mode is enabled (that is, when NATIVE authentication is being used, or when the property `derby.database.sqlAuthorization` is explicitly set to TRUE).

Old databases must be fully upgraded to at least Release 10.5 before roles can be used. See "Upgrades" in the *Derby Developer's Guide* for more information.

If SQL authorization mode is enabled, the Database Owner can use the CREATE ROLE statement to create roles. The Database Owner can then use the GRANT statement to grant a role to one or more users, to PUBLIC, or to another role.

A role A *contains* another role B if role B is granted to role A, or is contained in a role C granted to role A. Privileges granted to a contained role are inherited by the containing

roles. So the set of privileges identified by role A is the union of the privileges granted to role A and the privileges granted to any contained roles of role A.

For example, suppose the Database Owner issued the following statements:

```
create role reader;
create role updater;
create role taskLeaderA;
create role taskLeaderB;
create role projectLeader;
grant reader to updater;
grant updater to taskLeaderA;
grant updater to taskLeaderB;
grant taskLeaderA to projectLeader;
grant taskLeaderB to projectLeader;
```

The roles would then have the following containment relationships:

- The `projectLeader` role contains the `taskLeaderA` and `taskLeaderB` roles.
- The `taskLeaderA` and `taskLeaderB` roles both contain the `updater` role.
- The `updater` role contains the `reader` role.

In this case, the `projectLeader` role contains all the other roles and has all their privileges. If the Database Owner then revokes `updater` from `taskLeaderA`, `projectLeader` still contains that role through `taskLeaderB`.

The SYSCS_DIAG.CONTAINED_ROLES diagnostic table function can be used to determine the set of contained roles for a role.

Cycles are not permitted in role grants. That is, if a role contains another role, you cannot grant the container role to the contained role. For example, the following statement would not be permitted:

```
grant projectLeader to updater;
```

Setting roles

When a user first connects to Derby, no role is set, and the CURRENT_ROLE function returns null. During a session, the user can call the SET ROLE statement to set the current role for that session. The role can be any role that has been granted to the session's current user or to PUBLIC. To unset the current role, call SET ROLE with an argument of NONE. At any time during a session, there is always a current user, but there is a current role only if SET ROLE has been called with an argument other than NONE. If a current role is not set, the session has only the privileges granted to the user directly or to PUBLIC.

For example, if the Database Owner created and granted the roles shown in the previous session, a user would have to issue a SET ROLE statement to have them take effect. Suppose a user issued the following statement:

```
SET ROLE taskLeaderA;
```

Assuming that the Database Owner had granted the `taskLeaderA` role to the user, the user would be allowed to set the role as shown and would have all the privileges granted to the `taskLeaderA`, `updater`, and `reader` roles.

To retrieve the current role identifier in SQL, call the CURRENT_ROLE function.

Within stored procedures and functions that contain SQL, the current role depends on whether the routine executes with invoker's rights or with definer's rights, as specified by the EXTERNAL SECURITY clause in the CREATE FUNCTION or CREATE PROCEDURE statements in the *Derby Reference Manual*. During execution, the current user and current role are kept on an authorization stack, which is pushed during a stored routine call.

- Within routines that execute with invoker's rights, the following applies: initially, inside a nested connection, the current role is set to that of the calling context. So is the current user. Such routines may set any role granted to the invoker or to PUBLIC.
- Within routines that execute with definer's rights, the following applies: initially, inside a nested connection, the current role is NULL, and the current user is that of the definer. Such routines may set any role granted to the definer or to PUBLIC.

Upon return from the stored procedure or function, the authorization stack is popped, so the current role of the calling context is not affected by any setting of the role inside the called procedure or function. If the stored procedure opens more than one nested connection, these all share the same (stacked) current role (and user) state. Any dynamic result set passed out of a stored procedure sees the current role (or user) of the nested context.

Granting privileges to roles

Once a role has been created, both the Database Owner and the object owner can grant privileges on tables and routines to that role. You can grant the same privileges to roles that you can grant to users. Granting a privilege to a role implicitly grants privileges to all roles that contain that role. For example, if you grant delete privileges on a table to `updater`, every user in the `updater`, `taskLeaderA`, `taskLeaderB`, and `projectLeader` role will also have delete privileges on that table, but users in the `reader` role will not.

Revoking privileges from a role

Either the Database Owner or the object owner can revoke privileges from a role.

When a privilege is revoked from a role A, that privilege is no longer held by role A, unless A otherwise inherits that privilege from a contained role.

If a privilege to an object is revoked from role A, a session will lose that privilege if it has a current role set to A or a role that contains A, unless one or more of the following is true:

- The privilege is granted directly to the current user
- The privilege is granted to PUBLIC
- The privilege is also granted to another role B in the current role's set of contained roles
- The session's current user is the Database Owner or the object owner

Revoking roles

The Database Owner can use the REVOKE statement to revoke a role from a user, from PUBLIC, or from another role.

When a role is revoked from a user, that session can no longer keep that role, nor can it take on that role in a SET ROLE statement, unless the role is also granted to PUBLIC. If that role is the current role of an existing session, the current privileges of the session lose any extra privileges obtained through setting that role.

The default drop behavior is CASCADE. Therefore, all persistent objects (constraints, views and triggers) that rely on that role are dropped. Although there may be other ways of fulfilling that privilege at the time of the revoke, any dependent objects are still dropped. This is an implementation limitation. Any prepared statement that is potentially affected will be checked again on the next execute. A result set that depends on a role will remain open even if that role is revoked from a user.

When a role is revoked from a role, the default drop behavior is also CASCADE. Suppose you revoke role A from role B. Revoking the role will have the effect of revoking all additional applicable privileges obtained through A from B. Roles that contain B will

also lose those privileges, unless A is still contained in some other role C granted to B, or the privileges come through some other role. See Creating and granting roles for an example.

Dropping roles

Only the Database Owner can drop a role. To drop a role, use the DROP ROLE statement.

Dropping a role effectively revokes all grants of this role to users and other roles.

Further information

For details on the following statements, functions, and system table related to roles, see the *Derby Reference Manual*.

- CREATE ROLE statement
- SET ROLE statement
- DROP ROLE statement
- GRANT statement
- REVOKE statement
- CURRENT_ROLE function
- SYSCS_DIAG.CONTAINED_ROLES table function
- SYSROLES system table

Upgrading an old database to use SQL standard authorization

An old, unprotected database can be shielded with authentication and SQL authorization later on.

Upgrading authentication and authorization

To protect a single-user database and convert it to a shared, multi-user database, simply enable authentication and SQL authorization. To do this, first turn on user authentication as described in Configuring user authentication. Make sure that you supply login credentials for the Database Owner. In most single-user databases, the Database Owner is APP. However, the Database Owner could be some other user if the original database creation URL specified a user name; for details, see Database Owner. If you are unsure about who owns the database, run the following query:

```
select authorizationid from sys.sysschemas where schemaname = 'SYS'
```

After enabling user authentication, turn on SQL authorization. To do this, connect to the database as the Database Owner and issue the following command:

```
call syscs_util.syscs_set_database_property(
 'derby.database.sqlAuthorization', 'true' )
```

Now shut down the database to activate the new value of `derby.database.sqlAuthorization`. The next time you boot the database, it will be protected by authentication and SQL authorization.

Behavior of upgraded databases

You will notice the following behavior changes in your upgraded database:

- **Data**: Users can access data in their own schemas. However, users cannot access data in schemas owned by other users. In particular, other users cannot access data in schemas belonging to the Database Owner. The Database Owner may need to GRANT access to that data.
- **Database Maintenance**: In a single-user database, anyone can run maintenance procedures to backup/restore and import/export data. In the upgraded multi-user database, only the Database Owner can perform these sensitive operations.

SQL standard authorization exceptions

SQL exceptions are returned when errors occur with SQL authorization.

The following errors can result from the CREATE ROLE statement:

- You cannot create a role if you are not the Database Owner. An attempt to do so raises the `SQLException` 4251A.
- You cannot create a role if a role with that name already exists. An attempt to do so raises the `SQLException` X0Y68.
- You cannot create a role name if there is a user by that name. An attempt to create a role name that conflicts with an existing user name raises the `SQLException` X0Y68.
- A role name cannot start with the prefix SYS (after case normalization). Use of the prefix SYS raises the `SQLException` 4293A.
- You cannot create a role with the name PUBLIC (after case normalization). PUBLIC is a reserved authorization identifier. An attempt to create a role with the name PUBLIC raises `SQLException` 4251B.

The following errors can result from the DROP ROLE statement:

- You cannot drop a role if you are not the Database Owner. An attempt to do so raises the `SQLException` 4251A.
- You cannot drop a role that does not exist. An attempt to do so raises the `SQLException` 0P000.

The following errors can result from the SET ROLE statement:

- You cannot set a role if you are not the Database Owner. An attempt to do so raises the `SQLException` 4251A.
- You cannot set a role that does not exist. An attempt to do so raises the `SQLException` 0P000.
- You cannot set a role when a transaction is in progress. An attempt to do so raises the `SQLException` 25001.
- You cannot use NONE or a malformed identifier as a string or ? argument to SET ROLE. An attempt to do so raises the `SQLException` XCXA0.

The following errors can result from the GRANT statement:

- You cannot grant a role if you are not the Database Owner. An attempt to do so raises the `SQLException` 4251A.
- You cannot grant a role that does not exist. An attempt to do so raises the `SQLException` 0P000.
- You cannot grant the role "PUBLIC". An attempt to do so raises the `SQLException` 4251B.
- You cannot grant a role if doing so would create a circularity by granting a container role to a contained role. An attempt to do so raises the `SQLException` 4251C.

The following errors can result from the REVOKE statement:

- You cannot revoke a role if you are not the Database Owner. An attempt to do so raises the `SQLException` 4251A.
- You cannot revoke a role that does not exist. An attempt to do so raises the `SQLException` 0P000.
- You cannot revoke the role "PUBLIC". An attempt to do so raises the `SQLException` 4251B.

For all statements, an attempt to specify an identifier argument more than 128 characters long raises the `SQLException` 42622.

For more information about exceptions, see "SQL error messages and exceptions" in the *Derby Reference Manual*.

NATIVE authentication and SQL authorization example

This example consists of the program `NativeAuthenticationExample.java`, which shows how to use Derby's NATIVE user authentication and SQL authorization with either the embedded or the client driver.

See Configuring NATIVE authentication for information on NATIVE authentication. See the other topics under Configuring user authorization for more information on using SQL authorization.

The program does the following:

1. Uses a system property to set the authentication provider to `NATIVE:nativeAuthDB:LOCAL`, meaning that `nativeAuthDB` is the credentials database and that all user credentials are stored there.
2. If you are running the program using the client driver, starts the Network Server.
3. Creates a database named `nativeAuthDB` as the user `sysadm`, who is therefore the Database Owner. Only the Database Owner has the right to set and read database properties.
4. Calls the `SYSCS_UTIL.SYSCS_CREATE_USER` system procedure to create several users: `noacc`, `guest`, and `sqlsam`. The user `sysadm` has already been created automatically.
5. Creates the roles `adder` and `viewer`.
6. Grants the role `adder` to `sqlsam`, and grants the role `viewer` to `guest`.
7. Creates a table, `accessibletbl`, and inserts a value into it.
8. Grants SELECT and INSERT privileges on `accessibletbl` to `adder`.
9. Tries to connect to the database without supplying credentials, and fails, as expected.
10. Connects to the database as a user who has not been granted any privileges. The connection succeeds, but the user does not attempt to perform any operations, since no operations would be permitted.
11. Connects to the database as `guest`, who has the role `viewer`.
12. Sets the current role to `viewer`; the user succeeds in executing a SELECT statement on the table, but cannot execute an INSERT statement.
13. Connects to the database as `sqlsam`, who has the role `adder`.
14. Sets the current role to `adder`; the user succeeds in executing both a SELECT and an INSERT statement, but is unable to execute a DELETE statement.
15. Using the connection of the Database Owner `sysadm`, deletes the table, the two roles, and the three users created previously.
16. If you are running the program using the client driver, shuts down the Network Server.
17. Closes the connection and shuts down Derby, using the Database Owner's credentials.

The instructions for compiling and running the program are in the comment at the beginning of the program. `DERBY_LIB` is the directory that contains the Derby jar files, typically `DERBY_HOME/lib`.

Source code for `NativeAuthenticationExample.java`

```
// does not use derby.properties

import java.io.PrintWriter;
import java.sql.*;

import org.apache.derby.drda.NetworkServerControl;

/*
 * <p>
 * This program showcases how SQL authorization is automatically turned
 * on when you run with NATIVE authentication. You can run this program
```

```
 * either embedded or client server.
 * </p>
 *
 * <p>
 * Here's how you compile the program:
 * </p>
 *
 * <pre>
 * javac -cp ${DERBY_LIB}/derbynet.jar NativeAuthenticationExample.java
 * </pre>
 *
 * <p>
 * Here's how you run the program embedded:
 * </p>
 *
 * <pre>
 * java -cp ${DERBY_LIB}/derby.jar:. NativeAuthenticationExample embedded
 * </pre>
 *
 * <p>
 * Here's how you run the program client/server:
 * </p>
 *
 * <pre>
 * java -cp \
 *
 ${DERBY_LIB}/derby.jar:${DERBY_LIB}/derbynet.jar:${DERBY_LIB}/
derbyclient.jar:. \
 * NativeAuthenticationExample client
 * </pre>
 */
public class NativeAuthenticationExample
{
    /////////////////////////////////////////////////////////////////////
    //
    //  CONSTANTS
    //
    /////////////////////////////////////////////////////////////////////

    private static final String DB_NAME="nativeAuthDB";

    // stored as SYSADM
    private static final String DB_OWNER="sysadm";
    private static final String DB_OWNER_PASSWORD="shh123ihtybb87m";

    private static final String USER_WITHOUT_ROLE="NOACC";
    private static final String USER_WITHOUT_ROLE_PASSWORD="ajaxj3x9";

    private static final String READER="GUEST";
    private static final String READER_PASSWORD="java5w6x";

    private static final String WRITER="SQLSAM";
    private static final String WRITER_PASSWORD="light8q9bulb";

    private static final String EMBEDDED = "embedded";
    private static final String CLIENT = "client";

    /////////////////////////////////////////////////////////////////////
    //
    //  STATE
    //
    /////////////////////////////////////////////////////////////////////

    private boolean _runningEmbedded;
    private NetworkServerControl _server;

    /////////////////////////////////////////////////////////////////////
    //
    //  ENTRY POINT
    //
```

```
    ////////////////////////////////////////////////////////////////////

    public static void main( String... args )
    {
        NativeAuthenticationExample demo = parseArgs( args );

        if ( demo !=  null )
        {
            demo.execute();
        }
        else
        {
            println( "Bad command line args." );
        }
    }

    private static NativeAuthenticationExample parseArgs(
            String... args )
    {
        if ( (args == null) || (args.length != 1) )
        {
            return null;
        }

        String  mode = args[ 0 ];

        if ( EMBEDDED.equals( mode ) )
        {
            return new NativeAuthenticationExample( true );
        }
        else if ( CLIENT.equals( mode ) )
        {
            return new NativeAuthenticationExample( false );
        }
        else
        {
            return null;
        }
    }

    ////////////////////////////////////////////////////////////////////
    //
    //  CONSTRUCTOR
    //
    ////////////////////////////////////////////////////////////////////

    private NativeAuthenticationExample( boolean runningEmbedded )
    {
        _runningEmbedded = runningEmbedded;
    }

    ////////////////////////////////////////////////////////////////////
    //
    //  FEATURE SHOWCASE
    //
    ////////////////////////////////////////////////////////////////////

    /**
     * Run all of the experiments
     */
    private void execute()
    {
        try
        {
            String  authenticationProvider =
                                            "NATIVE:" + DB_NAME + ":LOCAL";

            // this turns on NATIVE authentication as well as
            // SQL authorization
            println( "Setting authentication provider to " +
```

```
                authenticationProvider );
            System.setProperty(  "derby.authentication.provider",
                authenticationProvider );

            if ( !_runningEmbedded )
            {
                startServer();
            }

            Connection  dboConn = createDatabase();

            createUsers( dboConn );
            createRoles( dboConn );
            createTable( dboConn );

            tryToConnectWithoutCredentials();   //should fail

            // a valid user can connect even if they haven't been
            // assigned any roles
            getConnection( USER_WITHOUT_ROLE,
                           USER_WITHOUT_ROLE_PASSWORD );

            verifyReaderPrivileges();
            verifyWriterPrivileges();

            println( "Using Database Owner connection again" );

            dropTable( dboConn );
            dropRoles( dboConn );
            dropUsers( dboConn );

            cleanUpAndShutDown();

        } catch (Exception e)
        {
            errorPrintAndExit( e );
        }
    }

    /**
     * Create more users. Note that the credentials for the Database
     * Owner were stored in the database automatically when the
     * database was created.
     */
    public void createUsers( Connection conn )
        throws SQLException
    {
        println( "Storing some sample users in the database." );

        PreparedStatement   ps = prepare
            ( conn, "call syscs_util.syscs_create_user( ?, ? )" );

        createUser( ps, USER_WITHOUT_ROLE, USER_WITHOUT_ROLE_PASSWORD );
        createUser( ps, READER, READER_PASSWORD );
        createUser( ps, WRITER, WRITER_PASSWORD );

        ps.close();
    }

    private void createUser( PreparedStatement ps, String userName,
            String password )
        throws SQLException
    {
        println( "Creating user " + userName );
        ps.setString( 1, userName );
        ps.setString( 2, password );
        ps.execute();
    }

    /**
```

```
     * Create roles and grant them privileges.
     */
    private void createRoles( Connection conn )
        throws SQLException
    {
        println( "Creating roles and granting privileges to them..." );

        execute( conn, "CREATE ROLE adder" );
        execute( conn, "CREATE ROLE viewer" );

        execute( conn, "GRANT adder TO " + WRITER );
        execute( conn, "GRANT viewer TO " + READER );
    }

    /**
     * Create and populate a table and grant privileges related to it.
     */
    private void createTable( Connection conn )
        throws SQLException
    {
        println("Creating table accessibletbl...");
        execute( conn,
                 "CREATE TABLE accessibletbl(textcol VARCHAR(6))" );
        execute( conn, "INSERT INTO accessibletbl VALUES('hello')" );

        println( "Granting select/insert privileges to adder..." );
        execute( conn,
                 "GRANT SELECT, INSERT ON accessibletbl TO adder" );

        println( "Granting select privileges to viewer" );
        execute( conn, "GRANT SELECT ON accessibletbl TO viewer" );
    }

    /**
     * Drop users except for Database Owner.
     */
    public void dropUsers( Connection conn )
        throws SQLException
    {
        println( "Dropping sample users from the database..." );

        PreparedStatement   ps = prepare
            ( conn, "call syscs_util.syscs_drop_user( ? )" );

        dropUser( ps, USER_WITHOUT_ROLE );
        dropUser( ps, READER );
        dropUser( ps, WRITER );

        ps.close();
    }

    private void dropUser( PreparedStatement ps, String userName )
        throws SQLException
    {
        println( "Dropping user " + userName );
        ps.setString( 1, userName );
        ps.execute();
    }

    /**
     * Drop roles.
     */
    private void dropRoles( Connection conn )
        throws SQLException
    {
        println( "Dropping roles..." );

        execute( conn, "DROP ROLE adder" );
        execute( conn, "DROP ROLE viewer" );
    }
```

```
    /**
     * Drop the table.
     */
    private void dropTable( Connection conn )
        throws SQLException
    {
        execute( conn, "DROP TABLE accessibletbl" );
    }

    /**
     * Try to connect without supplying credentials
     */
    private void tryToConnectWithoutCredentials()
        throws Exception
    {
        println( "Trying to connect without supplying credentials..." );

        try {
            getConnection( null, null );
            println( "ERROR: Unexpectedly connected to database " +
                     DB_NAME );
            cleanUpAndShutDown();
        } catch (SQLException e)
        {
            if ( e.getSQLState().equals("08004") )
            {
                println
                    (
                     "As expected, could not get a connection without " +
                     "supplying credentials."
                     );
            } else
            {
                errorPrintAndExit( e );
            }
        }
    }

    /**
     * Verify that the READER user can select but not insert
     */
    private void verifyReaderPrivileges()
        throws Exception
    {
        Connection  readerConn = getConnection( READER,
                                                READER_PASSWORD );

        println( "Setting role to VIEWER" );
        execute( readerConn, "SET ROLE VIEWER" );

        readRow( readerConn );    // should succeed

        try {
            writeRow( readerConn );
            println( "ERROR: Unexpectedly allowed to insert into table"
);
            cleanUpAndShutDown();
        } catch (SQLException e)
        {
            if ( e.getSQLState().equals("42500") )
            {
                println( "As expected, failed to insert row." );
            }
            else
            {
                errorPrintAndExit(e);
            }
        }
```

```
        readerConn.close();
    }

    /**
     * Verify that the WRITER can read and write but not delete
     */
    private void verifyWriterPrivileges()
        throws Exception
    {
        Connection  writerConn = getConnection( WRITER,
                                                WRITER_PASSWORD );

        // set role to ADDER
        println( "Setting role to ADDER" );
        execute( writerConn, "SET ROLE ADDER" );

        // should succeed
        readRow( writerConn );
        writeRow( writerConn );

        try {
            deleteRow( writerConn );    // should fail

            println( "ERROR: Unexpectedly allowed to DELETE." );
            cleanUpAndShutDown();
        } catch (SQLException e)
        {
            if ( e.getSQLState().equals("42500") )
            {
                println( "As expected, failed to delete rows." );
            }
            else
            {
                errorPrintAndExit(e);
            }
        }

        writerConn.close();
    }

    private void readRow( Connection conn ) throws SQLException
    {
        PreparedStatement   ps = prepare
            ( conn, "SELECT * FROM sysadm.accessibletbl" );
        ResultSet   rs = ps.executeQuery();
        while( rs.next() )
        {
            println
                ( "Value of sysadm.accessibletbl/textcol = " +
                    rs.getString( 1 ) );
        }
        rs.close();
        ps.close();
    }

    private void writeRow( Connection conn ) throws SQLException
    {
        execute( conn,
                 "INSERT INTO sysadm.accessibletbl VALUES('guest')" );
    }

    private void    deleteRow( Connection conn ) throws SQLException
    {
        execute( conn, "DELETE FROM sysadm.accessibletbl" );
    }

    ////////////////////////////////////////////////////////////////////
    //
    //  SQL HELPERS
    //
```

```
    ////////////////////////////////////////////////////////////////////

    /**
     * Execute a statement
     */
    private void execute( Connection conn, String text )
        throws SQLException
    {
        PreparedStatement   ps = prepare( conn, text );

        ps.execute();
        ps.close();
    }

    /**
     * Prepare a statement
     */
    private PreparedStatement prepare( Connection conn, String text )
        throws SQLException
    {
        println( "    Preparing: " + text );
        return conn.prepareStatement( text );
    }

    ////////////////////////////////////////////////////////////////////
    //
    //  CONNECTION MANAGEMENT
    //
    ////////////////////////////////////////////////////////////////////

    /**
     * Create the database
     */
    private Connection createDatabase()
        throws SQLException
    {
        String  connectionURL = getConnectionURL
            ( DB_NAME, DB_OWNER, DB_OWNER_PASSWORD, true, false );

        println( "Creating database via this URL: " + connectionURL );

        return DriverManager.getConnection( connectionURL );
    }

    /**
     * Shut down the engine and exit.
     */
    private void cleanUpAndShutDown()
        throws Exception
    {
        // Shut down the server before the engine. this is so that
        // we can authenticate the shutdown credentials in the
        // booted database.
        if ( _server != null )
        {
            stopServer();
        }

        // the engine should only be brought down locally
        _runningEmbedded = true;
        shutdownEngine();

        System.exit(1);
    }

    private void shutdownEngine()
    {
        String shutdownURL = getConnectionURL
            ( null, DB_OWNER, DB_OWNER_PASSWORD, false, true );
```

```
        try
        {
            println( "Shutting down engine via this URL: " +
                     shutdownURL );
            DriverManager.getConnection(  shutdownURL );
        } catch (SQLException se)
        {
            if ( se.getSQLState().equals("XJ015") )
            {
                println( "Derby engine shut down normally" );
            }
            else
            {
                printSQLException( se );
            }
        }
    }

    /**
     * Get a connection to the database
     */
    private Connection getConnection( String userName, String password )
        throws SQLException
    {
        String  connectionURL = getConnectionURL
            ( DB_NAME, userName, password, false, false );

        println( "Getting connection via this URL: " + connectionURL );

        return DriverManager.getConnection( connectionURL );
    }

    private String getConnectionURL( String dbName, String userName,
        String password, boolean createDB, boolean shutdownDB )
    {
        String  connectionURL = _runningEmbedded ?
            "jdbc:derby:" :
            "jdbc:derby://localhost:1527/";

        if ( dbName != null )
        {
            connectionURL = connectionURL + DB_NAME;
        }
        if ( userName != null )
        {
            connectionURL = connectionURL + ";user=" + userName;
        }
        if ( password != null)
        {
            connectionURL = connectionURL + ";password=" + password;
        }
        if ( createDB )
        {
            connectionURL = connectionURL + ";create=true";
        }
        if ( shutdownDB )
        {
            connectionURL = connectionURL + ";shutdown=true";
        }

        return connectionURL;
    }

    ////////////////////////////////////////////////////////////////////
    //
    //  SERVER MANAGEMENT
    //
    ////////////////////////////////////////////////////////////////////

    /**
```

```
     * Start the Derby server
     */
    private void startServer()
        throws Exception
    {
        _server = new NetworkServerControl( DB_OWNER,
                                            DB_OWNER_PASSWORD );

        println( "Starting the Derby server..." );
        _server.start( new PrintWriter( System.out ) );

        // pause to let the server come up
        Thread.sleep( 5000L );
    }

    /**
     * Shut down the Derby server
     */
    private void stopServer()
        throws Exception
    {
        println( "Stopping the Derby server..." );
        _server.shutdown();

        // pause to let the server come down
        Thread.sleep( 5000L );
    }

    ////////////////////////////////////////////////////////////////////
    //
    //  DIAGNOSTIC PRINTING
    //
    ////////////////////////////////////////////////////////////////////

    /**
     * Report exceptions and exit.
     */
    private void errorPrintAndExit( Throwable e )
    {
        if ( e instanceof SQLException )
        {
            printSQLException((SQLException) e);
        }
        else
        {
            println("A non-SQL error occurred.");
            e.printStackTrace();
        }

        System.exit(1);
    }

    /**
     * Print a list of SQLExceptions.
     */
    private void printSQLException( SQLException sqle )
    {
        while (sqle != null)
        {
            println("\n---SQLException Caught---\n");
            println("    SQLState:   " + (sqle).getSQLState());
            println("    Severity: " + (sqle).getErrorCode());
            println("    Message:  " + (sqle).getMessage());

            sqle.printStackTrace();

            sqle = sqle.getNextException();
        }
    }
```

```
    /**
     * Print a diagnostic line to the console
     */
    private static void println( String text )
    {
        System.out.println( text );
    }
}
```

Restricting file permissions

Additional file protections are available on some file systems, including Windows NTFS, Unix, and Linux. You can configure Derby to take advantage of these extra file protections.

By default, Derby creates new directories and files with the default permissions of the operating system account that started the VM (the umask setting on Unix and Linux). You can configure Derby to override those default permissions and to restrict access to just that account. If you configure Derby this way, only that account can access the directories and files created by Derby. You can configure this extra protection by setting the following system property, either on the VM command line or in `derby.properties`:

```
derby.storage.useDefaultFilePermissions=false
```

For more information, see "derby.storage.useDefaultFilePermissions" in the *Derby Reference Manual*.

If you set this property, other operating system accounts will have no access to directories or files created by Derby. This behavior can be helpful in enhancing default security for database files.

The exact behavior is determined by two factors: how the Derby engine is started, and the presence or absence and specified value of the property `derby.storage.useDefaultFilePermissions`.

The following table shows how file access works. In this table,

- "Environment" means that access is controlled entirely by the JVM environment and the file location only (that is, by the umask setting on UNIX and Linux systems and by the default file permissions on Windows NTFS).
- "Restricted" means that Derby restricts access to the operating system account that started the JVM.

The following table shows how file access works with various settings of the `derby.storage.useDefaultFilePermissions` property.

Table 6. File access

Property Setting	Server Started from Command Line	Server Started Programmatically or Embedded
No property specified	Restricted	Environment
Property set to true	Environment	Environment
Property set to false	Restricted	Restricted

Putting it all together

This section shows how to enable all available Derby defenses.

This example uses SSL encryption, NATIVE authentication, and both coarse-grained and fine-grained authorization.

Starting a secured Network Server

Bring up the server and turn on SSL.

This example first brings up the server and turns on SSL. It also tells the server that NATIVE credentials will be stored in the `mchrystaEncryptedDB` database. That last directive causes the Database Owner's credentials to be stored when `mchrystaEncryptedDB` is created.

```
java -Djavax.net.ssl.keyStore=/Users/me/vault/ServerKeyStore \
-Djavax.net.ssl.keyStorePassword=secretServerPassword \
-Djavax.net.ssl.trustStore=/Users/me/vault/ServerTrustStore \
-Djavax.net.ssl.trustStorePassword=secretServerTrustStorePassword \
-Dderby.storage.useDefaultFilePermissions=false \
-Dderby.authentication.provider=NATIVE:mchrystaEncryptedDB \
org.apache.derby.drda.NetworkServerControl start -p 8246 \
-ssl peerAuthentication
```

Creating and using a secure database

Now the Database Owner creates an encrypted database, turns on coarse-grained authorization, and creates some data that everyone can read but only he can write.

Fine-grained authorization is automatically turned on because we are using NATIVE authentication.

Connection URLs are shown on multiple lines, but must be entered on one line.

```
java -Djavax.net.ssl.trustStore=/Users/me/vault/ClientTrustStore \
-Djavax.net.ssl.trustStorePassword=secretClientTrustStorePassword \
-Djavax.net.ssl.keyStore=/Users/me/vault/ClientKeyStore \
-Djavax.net.ssl.keyStorePassword=secretClientPassword \
org.apache.derby.tools.ij
ij version 10.9
ij> connect 'jdbc:derby://localhost:8246/mchrystaEncryptedDB;create=true;
user=mchrystа;password=mchrystа;dataEncryption=true;
encryptionAlgorithm=Blowfish/CBC/NoPadding;
bootPassword=mySuperSecretBootPassword;ssl=peerAuthentication';
ij> --
-- Prevent our authentication properties from being overridden on the
-- command line or in derby.properties.
--
call SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY
( 'derby.database.propertiesOnly','true');
Statement executed.
ij> --
-- This time around, there is no need to add credentials for the database
-- owner. That is because the database owner's credentials were
-- automatically added when we created the NATIVE database, advertised to
-- the server by setting
-- -Dderby.authentication.provider=NATIVE:mchrystaEncryptedDB.
--
--call SYSCS_UTIL.SYSCS_CREATE_USER( 'mchrysta', 'mchrysta' );

-- now add other users
call SYSCS_UTIL.SYSCS_CREATE_USER( 'thardy', 'thardy' );
Statement executed.
```

```
ij> call SYSCS_UTIL.SYSCS_CREATE_USER( 'jhallett', 'jhallett' );
Statement executed.
ij> call SYSCS_UTIL.SYSCS_CREATE_USER( 'tquist', 'tquist' );
Statement executed.
ij> --
-- Turn on coarse-grained authorization
--
call SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY
( 'derby.database.fullAccessUsers', 'tquist,mchrysta' );
Statement executed.
ij> call SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY
( 'derby.database.readOnlyAccessUsers', 'thardy,jhallett' );
Statement executed.
ij> --
-- Shut down the database and bring it back up. This will turn on NATIVE
-- authentication and fine-grained authorization.
--
connect 'jdbc:derby://localhost:8246/mchrystaEncryptedDB;shutdown=true;
user=mchrysta;password=mchrysta;ssl=peerAuthentication';
ERROR 08006: DERBY SQL error: SQLCODE: -1, SQLSTATE: 08006, SQLERRMC:
Database 'mchrystaEncryptedDB' shutdown.
ij> --
-- Reboot the encrypted, password-protected database.
--
connect 'jdbc:derby://localhost:8246/mchrystaEncryptedDB;user=mchrysta;
password=mchrysta;bootPassword=mySuperSecretBootPassword;
ssl=peerAuthentication';
ij(CONNECTION1)> --
-- Create some data and let everyone see it.
--
create table mchrysta.t1( a varchar( 20 ) );
0 rows inserted/updated/deleted
ij(CONNECTION1)> insert into mchrysta.t1( a ) values ( 'mchrysta' );
1 row inserted/updated/deleted
ij(CONNECTION1)> grant select on table mchrysta.t1 to public;
0 rows inserted/updated/deleted
ij(CONNECTION1)> --
-- Verify that another user can read the newly created data but not write
-- it:
--
connect 'jdbc:derby://localhost:8246/mchrystaEncryptedDB;user=tquist;
password=tquist;ssl=peerAuthentication';
ij(CONNECTION2)> --
-- Verify that this user can see the data ...
--
select * from mchrysta.t1;
A
--------------------
mchrysta

1 row selected
ij(CONNECTION2)> --
-- ... but not write the data:
--
insert into mchrysta.t1( a ) values ( 'tquist' );
ERROR 42500: User 'TQUIST' does not have INSERT permission on table
'MCHRYSTA'.'T1'.
```

Stopping the secured Network Server

Now you can bring down the secured server.

```
java -Djavax.net.ssl.trustStore=/Users/me/vault/ClientTrustStore \
-Djavax.net.ssl.trustStorePassword=secretClientTrustStorePassword \
-Djavax.net.ssl.keyStore=/Users/me/vault/ClientKeyStore \
-Djavax.net.ssl.keyStorePassword=secretClientPassword \
org.apache.derby.drda.NetworkServerControl shutdown -p 8246 \
-user mchrysta -password mchrysta \
```

```
-ssl peerAuthentication
```

Trademarks

The following terms are trademarks or registered trademarks of other companies and have been used in at least one of the documents in the Apache Derby documentation library:

Cloudscape, DB2, DB2 Universal Database, DRDA, and IBM are trademarks of International Business Machines Corporation in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.