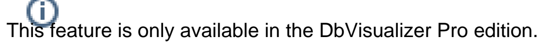
Security



Only in DbVisualizer Pro

This feature is only available in the DbVisualizer Pro edition.

- Using an SSH Tunnel
- Using SSL/TLS
  - Certificates
    - Trusting the server – One way authentication
    - Using a truststore for the whole JVM
    - Mutual authentication – two way authentication
      - Example
- Common problems
  - "invalid privatekey: [B@........"
- Single Sign-On (SSO)

When DbVisualizer communicates with a database server, all communication (except passwords) is done in plain text; by intercepting the communication, someone can access the transmitted data and even modify it while in transit. To protect your communication you need to encrypt the communication between DbVisualizer and the server. Kerberos, LDAP, Radius, security mechanisms, external authentication options, SSH server forwarding, and a lot of other database specific features are not directly related to DbVisualizer, but we will do our best to assist in these scenarios.

You find additional information on security related to specific databases in our support portal.

Using an SSH Tunnel

You can use SSH (Secure SHell) to encrypt the network connection between DbVisualizer and a server even for non-SSL-capable clients. A database that sits behind a firewall cannot be accessed directly from a client on the other side of the firewall, but it can often be accessed through an SSH tunnel. The firewall must be configured to accept SSH connections and you also need to have an account on the SSH host for this to work.

If you need to access a database that can only be accessed vi an SSH tunnel, you need to specify additional information in the Use SSH Tunnel area of the Connection tab.

This area is only shown when the **Server Info** settings format is selected, and only for databases identified by at least a **Database Server** and a **Database Port** (i.e. not for embedded databases or when using the TNS Connections Type for an Oracle database, or similar).

Enable SSH tunneling by clicking on the checkbox. When it is enabled, five additional fields are shown.

Database Connection: CRM Ahoa

CRM Ahoa

Actions

Disconnected

Connection | Properties

Connection	
Name	CRM Ahoa
Notes	
Database	
Settings Format	Server Info
Database Type	Auto Detect (Oracle)
Driver (JDBC)	Oracle Thin
Connection Type	Service
Database Server	192.168.1.21
Database Port	1521
Service	XE
Authentication	
Database Userid	hr
Database Password	••
Use SSH Tunnel	☑
SSH Host	192.168.1.100
SSH Port	22
SSH Userid	hans
SSH Password	
Private Key File	
Options	

Connect | Disconnect | Ping Server

Connection Message

Disconnected.

The **SSH Host** is the name or IP address for the host accepting SSH connections. The SSH Host is typically the same as the Database Server. Enter the port for SSH connections in the **SSH Port** field. The default value is 22.

You may also enter the userid and password for your SSH host account in the **SSH Userid** and **SSH Password** fields, but see Setting Common Authentication Options for other options. Alternatively, you can enter the path to a private key file (using either the RSA or DSA algorithms) in the **Private Key File** field. The SSH Password field is then replaced by a Key Passphrase field where you can enter the passphrase if the private key is protected with one.

When SSH tunneling is enabled, a tunnel is established when you connect to the database and the connection is then made through the tunnel by constructing a JDBC URL that uses information from both the Connection and Use SSH Tunnel sections.

If you're familiar with using the `ssh` command to set up a tunnel manually, you may be interested in more details. The tunnel corresponds to the tunnel you would set up with the ssh command like this:

```
ssh -p <SSHPort> -L<LocalPort>:<DatabaseServer>:<DatabasePort> <SSHUserid>@<SSHHost>
```

Where the placeholders correspond to the fields in the Connect and Use SSH Tunnel sections, except for <LocalPort> which is any available port, determined at connect time.

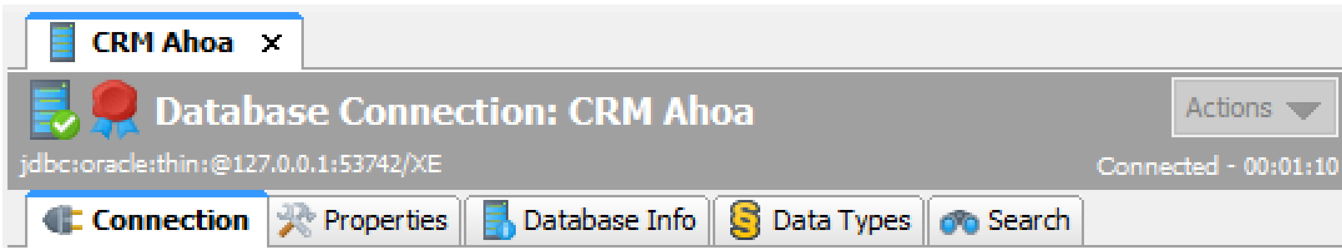
Note that when using an SSH tunnel, the **Database Server** is evaluated on the SSH host. If the database server is running on the SSH host, you can therefore set **Database Server** to `localhost` in case the database only accepts local connections.

The JDBC URL is constructed using 127.0.0.1 as the Database Server portion and <LocalPort> as the Database Port portion, e.g. like this for the Oracle Thin driver when <LocalPort> is 50538:

```
jdbc:oracle:thin@127.0.0.1:50538/XE
```

In other words, the JDBC driver connects to the SSH tunnel's local port, which then forwards all communication to the database server.

The URL that is used for the connection is shown at the top of the **Object View** tab for the database connection when a connection is established, along with a certificate icon if the connection is made through an SSH tunnel.



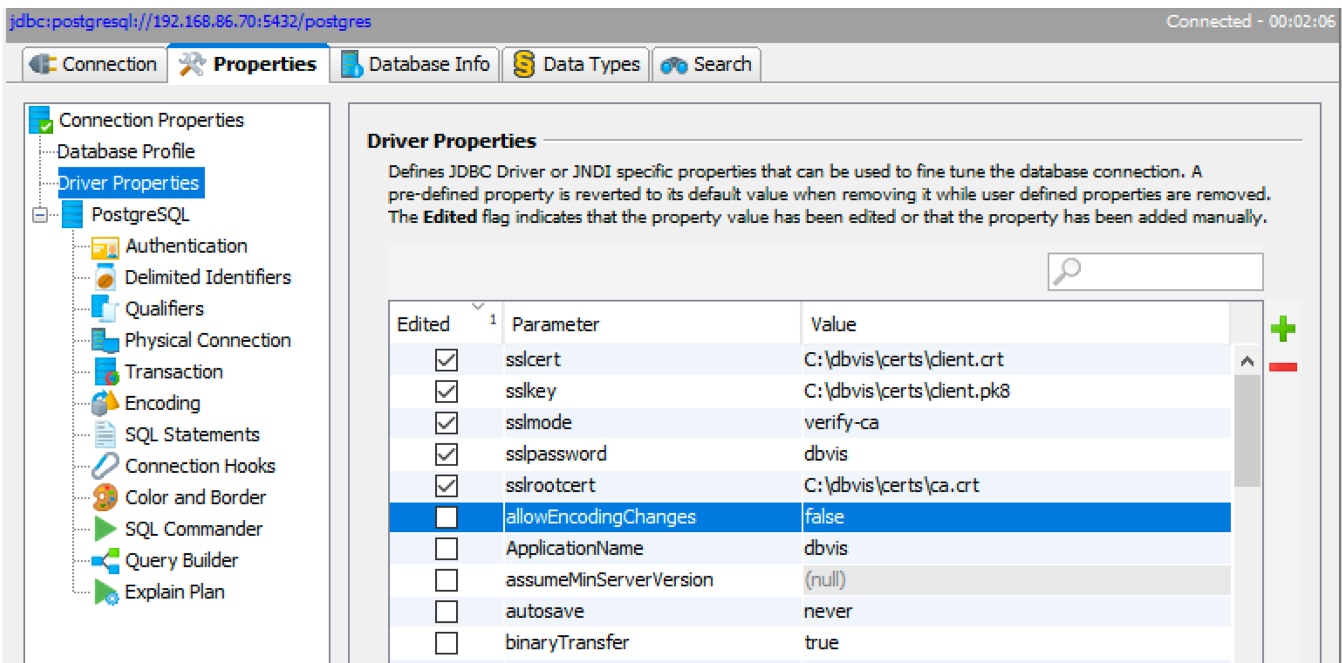
If you use the SSH Tunnel feature, you may also want to configure a few things in **Tools->Tool Properties**. In the **Database Connection/SSH Settings** category under the General tab, you can specify:

- **SSH Keep-Alive Interval** to minimize the risk that the tunnel is disconnected due to inactivity
- **SSH Known Hosts File** so you don't have to accept connections to known SSH hosts every time you connect
- **SSH Config File** containing optional SSH configuration. More on SSH config files.
- **SSH Authentication** settings
- **Number of SSH Authentication Tries** that limits the number of allowed connection attempts
- **SSH Password/Passphrase** settings

Using SSL/TLS

Depending on the database and the JDBC driver, you may be able to use SSL (Secure Socket Layer) to encrypt client/server communications and securely authenticate client and server.

In case the JDBC driver supports SSL, you define the SSL settings as Driver Properties for your connection according to the documentation for the JDBC driver. The exact details depend on the versions of the database and the driver, but using PostgreSQL as an example, the settings can look like this:



Certificates

For ensuring security of the data being transferred between a client and server, SSL can be implemented either one-way or two-way (aka mutual authentication). In one way SSL, only client validates the server to ensure that it receives data from the intended server. In case of two-way SSL, both client and server authenticate each other to ensure that both parties involved in the communication are trusted.

Trusting the server – One way authentication

A server certificate that is signed by a trusted Certificate Authority (CA) should always work fine, since the Java distribution includes a truststore with all the CA public keys.

When a self-signed server certificate is used, some additional configuration is needed. Depending on the actual JDBC driver this may include importing certificates to a truststore using the Java keytool. Some drivers allow this truststore to be configured per connection instead of for the whole Java VM. When using a truststore that affects the whole JVM special considerations must be taken.

Using a truststore for the whole JVM

Many forums on the net suggests using the Java keytool to import the certificate into the Java VM's default truststore. The drawback with that solution is that it does not survive a Java upgrade; when the Java VM bundled with DbVisualizer is used, upgrading DbVisualizer effectively causes SSL connections to fail. We therefore recommend creating a truststore separate from the Java VM (e.g. in */Users/me/MyTrustStore*) and import the certificate to that Trust Store.

Note that setting the truststore on Java VM level may affect other functionality of DbVisualizer as well as other database connections. E.g. if the certificate of dbvis.com cannot be verified neither **Help->Contact Support** nor **Help->Check for Update** will work.

When creating the truststore you should always start with a trust store containing the needed certificates (E.g. the default java Trust Store) and add/import your custom certificate to it.

Do the following:

1. Copy the Java default truststore to your location (e.g. /Users/me/MyTrustStore).
   The location of the Java default truststore is in most cases
   <Java Home>/lib/security/cacerts
2. Import your server certificate to the truststore using keytool. The password of the Java VM truststore is in most cases *changeit*. Following is an example of using the keytool when importing a certificate.

   ```
   keytool -importcert -alias mycert -file ca.pem -keystore /Users/me/MyTrustStore -storepass changeit
   ```

Replace the paths to your fit your environment.

Java VM properties for pointing to the truststore can then be added in **Tools->Tool Properties**, in the **Java VM Properties** area in the **General** category:

```
-Djavax.net.ssl.trustStore=/Users/me/MyTrustStore
-Djavax.net.ssl.trustStorePassword=mytruststore_password
```

There is also a Java VM property that can be used to get debugging information from the SSL handshake process.

```
-Djavax.net.debug=all
```

Mutual authentication – two way authentication

Some database servers can be configured to require clients connecting to authenticate using a certificate.

The configuration on the client side (DbVisualizer) resembles the way a truststore is configured. In this case you may create a keystore containing a single certificate. DbVisualizer functionality is not depending on any keys in the JVM keystore.

Java VM properties for pointing to the keystore can then be added in Tool Properties, in the Java VM Properties area in the General category.

```
-Djavax.net.ssl.keyStore=/Users/me/MyKeyStore
-Djavax.net.ssl.keyStorePassword=mykeystore_password
```

Again, there may be drivers supporting configuration of keystores (and truststores) per connection. If this is supported this is the preferred way of configuration.

Example

For an example on how such a certificate can be created, see description below. Note that this and the following commands are just examples and should only be seen as a guideline.

Create your local keystore directory, for example as */users/me/MyKeyStore*. Go to this directory and create your keystore and key pair with a command like below:

```
keytool -genkey -alias dbvisuser -keyalg RSA -validity 365 -keystore client.keystore -storetype pkcs12
```

Give a password to be used for the keystore and answer the questions. As a result you will finally have the keystore in a created file with name as given: *client.keystore*

You can now view your keystore with command:

```
keytool -list -v -keystore client.keystore
```

And, you can create a certificate (in the example given the name *mydomain.crt*) to be used by a database server with command:

```
keytool -export -alias dbvisuser -file mydomain.crt -keystore client.keystore
```

This new certificate file can then be given to the DBA(s) for the database that you connect to.

Common problems

Establishing an SSH tunnel may result in various errors and problems due to algorithms, key lengths, and much more. Here we list a few of them and possible solutions.

"invalid privatekey: [B@........"

This error is reported when using a private key file that is not in a valid format. If you are sure it is a valid private key this error may also occur when using keys that are in the "new OpenSSH" format rather than classic. One potential fix on the OpenSSH problem is to ensure you generate the keys with the **-t PEM** option to ssh-keygen. The following example shows the command used to create new keys and the other how to convert existing keys.

```
# Create new keys with the -m PEM option
ssh-keygen -t rsa -m PEM

# Converting existing keys with -m PEM option
ssh-keygen -p -f /home/me/.ssh/id_rsa -m PEM
```

Single Sign-On (SSO)

Availability and configuration of SSO options depend on the database and JDBC driver. Please refer to our support portal for more details.