



This PDF is generated from authoritative online content, and is provided for convenience only. This PDF cannot be used for legal purposes. For authoritative understanding of what is and is not supported, always use the online content. To copy code samples, always use the online content.

Platform SDK Developer's Guide

Using the Platform SDK Commons Library

12/14/2025


Contents

- 1 Using the Platform SDK Commons Library
 - 1.1 Using the Platform SDK Commons Library to Configure TLS

Using the Platform SDK Commons Library

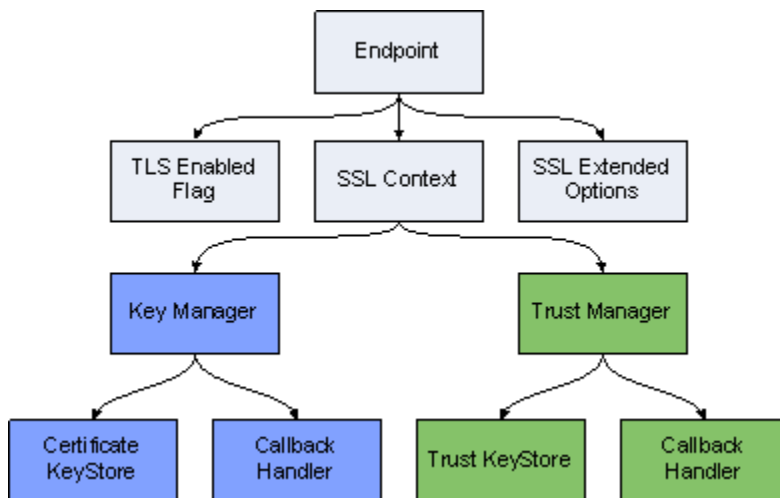
Using the Platform SDK Commons Library to Configure TLS

Starting with Platform SDK 8.1.1, the only way to configure connections is by using `Endpoint` objects, which contain all parameters related to the endpoint connection—including TLS parameters that indicate whether TLS is enabled and provide details about the SSL context and extended options.

 **Note:** In earlier releases, Platform SDK provided three ways to configure connections:

- using `ConnectionConfiguration` objects passed to `Protocol` constructors
- setting parameters in the protocol context
- adding a textual parameter representation to the URL query

The following diagrams show interdependencies among the Platform SDK objects used to establish network connections and support TLS.



TLS Configuration Objects Containment Hierarchy

This page outlines each step required to create supporting objects for a TLS-enabled `Endpoint`.

Callback Handlers

In many cases, certificate or key storage is password-protected. This means that Platform SDK will need the password to access storage. The Java `CallbackHandler` interface offers a flexible way to pass this type of credential data:

```
package javax.security.auth.callback;
```

```
...
public interface CallbackHandler {
    void handle(Callback[] callbacks)
        throws java.io.IOException, UnsupportedCallbackException;
}
```

The `handle()` method accepts credential requests in the form of `Callback` objects that have appropriate setter methods. The most common callback implementation is `PasswordCallback`. User code may use a GUI to ask the end user to:

- enter a password
- retrieve a password from a file, pipe, network, and so on

Here is an example of a `CallbackHandler` delegating password retrieval to a GUI:

```
CallbackHandler callbackHandler = new CallbackHandler() {
    public void handle(Callback[] callbacks) throws IOException,
        UnsupportedCallbackException {
        for (Callback c : callbacks) {
            if (c instanceof PasswordCallback) {
                PasswordCallback p = (PasswordCallback) c;
                p.setPassword(gui.getKeyStorePassword());
            }
        }
    }
};
```

When No Password is Required

In some cases, certificate storage does not need a password. The API may still dictate that a `CallbackHandler` be provided however, so the Platform SDK includes a predefined class that can be used as a "dummy" `CallbackHandler` for this scenario:

```
com.genesyslab.platform.commons.connection.tls.DummyPasswordCallbackHandler
```

Here is an example of using this dummy class:

```
CallbackHandler callbackHandler = new DummyPasswordCallbackHandler();
```

Key Managers

Java provides a `KeyManager` interface. This interface defines functionality that can be used to load and contain certificates or keys, or to select appropriate certificates or keys.

Classes based on the `KeyManager` interface are used by Java TLS support to retrieve certificates that will be sent over the network to a remote party for validation. They are also used to retrieve the corresponding private keys. On the client side, `KeyManager` classes retrieve client certificates or keys; on the server side they retrieve server certificates or keys.

The Platform SDK Commons library has a helper class, `KeyManagerHelper`, which makes it easy to create key managers using several types of key stores and security providers. The built-in key manager types are:

- **PEM** — reads certificate/key pairs from X.509 PEM files.
- **MSCAPI** — uses the Microsoft CryptoAPI and Windows certificate services to retrieve certificate/key

pairs.

- **PKCS11** — delegates to an external security provider plugged in via the PKCS#11 interface, for example, Mozilla NSS.
- **JKS** — retrieves a certificate/key pair from a Java Keystore file.
- **Empty** — does not retrieve anything. This type is for use as a dummy key manager. For example, clients that do not have certificates can use it.

Here are some examples of key manager creation:

```
// From PEM file
X509ExtendedKeyManager km = KeyManagerHelper.createPEMKeyManager(
    "c:/cert/client-cert.pem", "c:/cert/client-cert-key.pem");

// From MSCAPI
CallbackHandler cbh = new DummyPasswordCallbackHandler();
// Whitespace characters are allowed anywhere inside the string
String certThumbprint =
    "4A 3F E5 08 48 3A 00 71 8E E6 C1 34 56 A4 48 34 55 49 D9 0E";
X509ExtendedKeyManager km = KeyManagerHelper.createMSCAPIKeyManager(
    cbh, certThumbprint);

// From PKCS11
// This provider does not allow customization of Key Manager
// This is required for FIPS-140 certification
// Dummy callback handler will not work, must use strong password
CallbackHandler passCallback = ...;
X509ExtendedKeyManager km = KeyManagerHelper.createPKCS11KeyManager(
    passCallback);

// From JKS
// JKS key store does not allow callback usage (bug in Java?)
// Individual entries in JKS key store can be password-protected
char[] keyStorePass = "keyStorePass".toCharArray();
char[] entryPass = "entryPass".toCharArray();
X509ExtendedKeyManager km = KeyManagerHelper.createJKSKeyManager(
    "c:/cert/client-cert.jks", keyStorePass, entryPass);

// Empty key manager
// Using KeyManagerHelper class
X509ExtendedKeyManager km1 = KeyManagerHelper.createEmptyKeyManager();
// Direct creation
X509ExtendedKeyManager km2 = new EmptyX509ExtendedKeyManager();
```

Trust Managers

A Trust Manager is an entity that decides which certificates from a remote party are to be trusted. It performs certificate validation, checks the expiration date, matches the host name, checks the certificate against a CRL list, and builds and validates the chain of trust. The chain of trust starts from a certificate trusted by both sides (for example, a CA certificate) and continues with second-level certificates signed by CA, then possibly with third-level certificates signed by second-level authorities and so on. Chain length can vary, but Platform SDK was designed to explicitly support two-level chains consisting of a CA certificate and a leaf certificate signed by CA.

Trust manager instances are created based on storage that contains trusted certificates. The number of trusted certificates can vary depending on the type of trust manager being used. With PEM files, the storage contains only a single CA certificate; other provider types can have larger sets of trusted certificates.

The Platform SDK Commons library has a helper class, `TrustManagerHelper`, which makes it easy to create trust managers that use several types of certificate stores and security providers, and which can accept additional parameters that affect certificate validation. Built-in trust manager types are:

- **PEM** — Reads a CA certificate from an X.509 PEM file.
- **MSCAPI** — Uses the Microsoft CryptoAPI and Windows certificate services to retrieve CA certificates and validate certificates.
- **PKCS11** — Delegates certificate validation to an external security provider plugged in via the PKCS#11 interface, for example, Mozilla NSS.
- **JKS** — Retrieves a CA certificate from a Java Keystore file and uses Java built-in validation logic.
- **Default** — Uses trusted certificates shipped with or configured in Java Runtime and Java built-in validation logic.
- **TrustEveryone** — Trusts any certificates. Can be used on the server side when you do not expect any certificates from clients, or during testing.

Here are some examples of trust manager creation (with generic `crlPath` and `expectedHostName` parameters defined in the first example):

```
// Generic parameters for trust manager examples
String crlPath = "c:/cert/ca-crl.pem";
String expectedHostName = "serverhost";
// From PEM file
X509TrustManager tm = TrustManagerHelper.createPEMTrustManager(
    "c:/cert/ca.pem", crlPath, expectedHostName);

// From MSCAPI
// CRL is loaded from PEM file (Platform SDK supports only file-base CRLs)
// Concrete CA is not specified, all certificates from WCS Trusted Root are used
CallbackHandler cbh = new DummyPasswordCallbackHandler();
X509TrustManager tm = TrustManagerHelper.createMSCAPITrustManager(
    cbh, crlPath, expectedHostName);

// From PKCS#11
// This provider implementation in Java does not allow custom host name check,
// but CRL can still be used
X509TrustManager tm = TrustManagerHelper.createPKCS11TrustManager(
    cbh, crlPath);

// From JKS
// JKS key store does not allow callback usage (bug in Java?)
// Certificate-only entries cannot have passwords in JKS key store
// CRL and host name check are supported
char[] keyStorePass = "keyStorePass".toCharArray();
X509ExtendedKeyManager km = KeyManagerHelper.createJKSTrustManager(
    "c:/cert/ca-cert.jks", keyStorePass, crlPath, expectedHostName);

// From Java built-in trusted certificates
// This one does not support CRL and host name check
X509ExtendedKeyManager km = KeyManagerHelper.createDefaultTrustManager();

// Trust Everyone
X509ExtendedKeyManager km =
    KeyManagerHelper.createTrustEveryoneTrustManager();
```

SSLContext and SSLEngineOptions

An SSLContext instance serves as a container for all SSL and TLS parameters and objects and also as a factory for SSLEngine instances.

SSLEngine instances contain logic that deals directly with TLS handshaking, negotiation, and data encryption and decryption. SSLEngine instances are not reusable and must be created anew for each connection. This is a good reason for requiring users to provide an SSLContext instance rather than an instance of SSLEngine. SSLEngine instances are created by the Platform SDK connection layer and are not exposed to user code.

Only some of the parameters for SSLEngine can be pre-set in SSLContext. However, the SSLEngineOptions class may be used to collect additional parameters.

SSLEngineOptions currently contains two parameters:

- the "mutual TLS" flag
- a list of enabled cipher suites

The mutual TLS flag is used only by server applications. When the flag is turned on, the server will require connecting clients to send their certificates for validation. The connections of any clients that do not send certificates will fail.

The list of enabled cipher suites contains the names of all cipher suites that will be used as filters for SSLEngine. As a result, only ciphers that are supported by SSLEngine and that are contained in the enabled cipher suites list will be enabled for use.

Platform SDK includes the SSLContextHelper helper class to support one-line creation of SSLContext and SSLEngineOptions instances.

Here are some examples:

```
// Creating SSLContext
KeyManager km = ...;
TrustManager tm = ...;
SSLContext sslContext = SSLContextHelper.createSSLContext(km, tm);

String[] cipherList = new String[] {
    "TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA",
    "TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA",
    "TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA"};
// Can be single String with space-separated suite names
String cipherNames = "TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA " +
    "TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA " +
    "TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA";
boolean mutualTLS = false;

// Creating SSLEngineOptions directly
SSLEngineOptions sslOpts1 =
    new SSLEngineOptions(mutualTLS, cipherList);
SSLEngineOptions sslOpts2 =
    new SSLEngineOptions(mutualTLS, cipherNames);

// Create SSLEngineOptions using the helper class:
SSLEngineOptions sslOpts3 =
    SSLContextHelper.createSSLEngineOptions(mutualTLS, cipherList);
SSLEngineOptions sslOpts4 =
```

```
SSLContextHelper.createSSExtendedOptions(mutualTLS, cipherNames);
```

Endpoints

Now that supporting objects have been created and configured, you are ready to create an Endpoint.

The connection configuration parameters of an Endpoint are read-only—they cannot be changed after the Endpoint is created. This configuration information is then used by Protocol instances, the warm standby service, the connection layer and the TLS layer.

A sample Endpoint configuration is shown below:

```
ConnectionConfiguration connConf = ...;
SSLContext sslContext = ...;
SSExtendedOptions sslOpts = ...;
tlsEnabled = true;
// Specifying host name and port.
Endpoint ep1 = new Endpoint("Server-1", "serverhost", 9090, connConf,
    tlsEnabled, sslContext, sslOpts);
// Specifying URI. Query part is still supported.
String uri = "tcp://Server-1@serverhost:9090/" +
    "?protocol=addp&addp-remote-timeout=5&addp-trace=remote";
Endpoint ep2 = new Endpoint("Server-1", uri, connConf,
    tlsEnabled, sslContext, sslOpts);
```

Note: Configuration parameters can be set directly in a Protocol instance context, but will be overwritten and lost under the following conditions:

- a new Endpoint is set up
- the protocol is forced to reconnect
- a warm standby switchover occurs

Configuring TLS for Client Connections

Using the information above, you are now ready to configure actual client connections.

Example:

```
// Get TLS configuration objects for connection
String clientName = "ClientApp";
String host = "serverhost";
int port = 9000;
SSLContext sslContext = ...; // Assume it is created
SSExtendedOptions sslOptions = ...; // Assume it is created
boolean tlsEnabled = true;

ConnectionConfiguration connConf = new KeyValueConfiguration(new KeyValueCollection());
Endpoint epTSrv = new Endpoint(
    clientName, host, port, connConf, tlsEnabled, sslContext, sslOptions);

TServerProtocol tsProtocol = new TServerProtocol(epTSrv);
tsProtocol.setClientName(clientName);
tsProtocol.open();
```


Configuring TLS for Servers

Using the information above, you are now ready to configure actual server connections.

```
String serverName = "ServerApp";
String host = "serverhost";
int port = 9000;
SSLContext sslContext = ...; // Assume it is created
SSLExtendedOptions sslOptions = ...; // Assume it is created
boolean tlsEnabled = true;

ConnectionConfiguration connConf = new KeyValueConfiguration(new KeyValueCollection());
Endpoint epTSrv = new Endpoint(
    serverName, host, port, connConf, tlsEnabled, sslContext, sslOptions);

ExternalServiceProtocolListener serverChannel =
    new ExternalServiceProtocolListener(endpoint);
```