**How the JCA Might Be Used in a SSL/TLS Implementation**

With an understanding of the JCA classes, consider how these classes might be combined to implement an advanced network protocol like SSL/TLS. The [SSL/TLS Overview](https://docs.oracle.com/javase/7/docs/technotes/guides/security/jsse/JSSERefGuide.html#SSLOverview) section in the [JSSE Reference Guide](https://docs.oracle.com/javase/7/docs/technotes/guides/security/jsse/JSSERefGuide.html) describes at a high level how the protocols work. As asymmetric (public key) cipher operations are much slower than symmetric operations (secret key), public key cryptography is used to establish secret keys which are then used to protect the actual application data. Vastly simplified, the SSL/TLS handshake involves exchanging initialization data, performing some public key operations to arrive at a secret key, and then using that key to encrypt further traffic.

**NOTE:** The details presented here simply show how some of the above classes might be employed. This section will not present sufficient information for building a SSL/TLS implementation. For more details, please see the [JSSE Reference Guide](https://docs.oracle.com/javase/7/docs/technotes/guides/security/jsse/JSSERefGuide.html) and [RFC 2246: The TLS Protocol](http://www.ietf.org/rfc/rfc2246.txt).

Assume that this SSL/TLS implementation will be made available as a JSSE provider. A concrete implementation of the Provider class is first written that will eventually be registered in the Security class' list of providers. This provider mainly provides a mapping from algorithm names to actual implementation classes. (that is: "SSLContext.TLS"->"com.foo.TLSImpl") When an application requests an "TLS" instance (via SSLContext.getInstance("TLS"), the provider's list is consulted for the requested algorithm, and an appropriate instance is created.

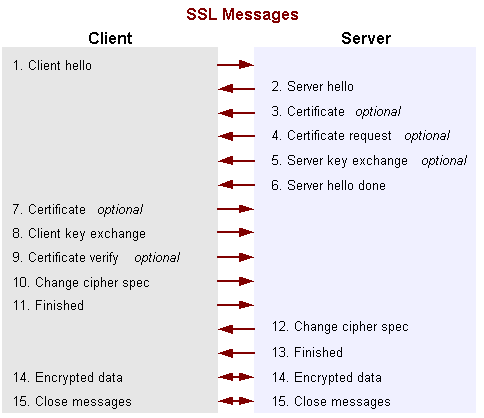
Before discussing details of the actual handshake, a quick review of some of the JSSE's architecture is needed. The heart of the JSSE architecture is the SSLContext. The context eventually creates end objects (SSLSocket and SSLEngine) which actually implement the SSL/TLS protocol. SSLContexts are initialized with two callback classes, KeyManager and TrustManager, which allow applications to first select authentication material to send and second to verify credentials sent by a peer.

A JSSE KeyManager is responsible for choosing which credentials to present to a peer. Many algorithms are possible, but a common strategy is to maintain a RSA or DSA public/private key pair along with a X509Certificate in a KeyStore backed by a disk file. When a KeyStore object is initialized and loaded from the file, the file's raw bytes are converted into PublicKey and PrivateKey objects using a KeyFactory, and a certificate chain's bytes are converted using a CertificateFactory. When a credential is needed, the KeyManager simply consults this KeyStore object and determines which credentials to present.

A KeyStore's contents might have originally been created using a utility such as keytool. keytool creates a RSA or DSA KeyPairGenerator and initializes it with an appropriate keysize. This generator is then used to create a KeyPair which keytool would store along with the newly-created certificate in the KeyStore, which is eventually written to disk.

A JSSE TrustManager is responsible for verifying the credentials received from a peer. There are many ways to verify credentials: one of them is to create a CertPath object, and let the JDK's built-in Public Key Infrastructure (PKI) framework handle the validation. Internally, the CertPath implementation might create a Signatureobject, and use that to verify that the each of the signatures in the certificate chain.

With this basic understanding of the architecture, we can look at some of the steps in the SSL/TLS handshake. The client begins by sending a ClientHello message to the server. The server selects a ciphersuite to use, and sends that back in a ServerHello message, and begins creating JCA objects based on the suite selection. We'll use server-only authentication in the following examples.



[Description of Figure 16: SSL Messages](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec_image_descriptions.html" \l "ssl-messages)

In the first example, the server tries to use a RSA-based ciphersuite such as TLS\_RSA\_WITH\_AES\_128\_CBC\_SHA. The server's KeyManager is queried, and returns an appropriate RSA entry. The server's credentials (that is: certificate/public key) are sent in the server's Certificate message. The client's TrustManager verifies the server's certificate, and if accepted, the client generates some random bytes using a SecureRandom object. This is then encrypted using an encrypting asymmetric RSA Cipher object that has been initialized with the PublicKey found in the server's certificate. This encrypted data is sent in a Client Key Exchange message. The server would use its corresponding PrivateKey to recover the bytes using a similar Cipher in decrypt mode. These bytes are then used to establish the actual encryption keys.

In a different example, an ephemeral Diffie-Hellman key agreement algorithm along with the DSA signature algorithm is chosen, such as TLS\_DHE\_DSS\_WITH\_AES\_128\_CBC\_SHA. The two sides must each establish a new temporary DH public/private keypair using a KeyPairGenerator. Each generator creates DH keys which can then be further converted into pieces using the KeyFactory and DHPublicKeySpec classes. Each side then creates a KeyAgreement object and initializes it with their respective DH PrivateKeys. The server sends its public key pieces in a ServerKeyExchange message (protected by the DSA signature algorithm, and the client sends its public key in a ClientKeyExchange message. When the public keys are reassembled using another KeyFactory, they are fed into the agreement objects. The KeyAgreement objects then generate agreed-upon bytes that are then used to establish the actual encryption keys.

Once the actual encryption keys have been established, the secret key is used to initialize a symmetric Cipher object, and this cipher is used to protect all data in transit. To help determine if the data has been modified, a MessageDigest is created and receives a copy of the data destined for the network. When the packet is complete, the digest (hash) is appended to data, and the entire packet is encrypted by the Cipher. If a block cipher such as AES is used, the data must be padded to make a complete block. On the remote side, the steps are simply reversed.

Again, this is vastly simplified, but gives one an idea of how these classes might be combined to create a higher level protocol.