# A PQC Almanac

**9th February 2025**

## What is the current state of play?

### In Brief:

There are two sets of algorithms that current standards and draft standards (at least NIST standards) are derived from:

two stateful signature algorithms – LMS and XMSS – detailed in SP 800-208,

one KEM, ML-KEM, three signature algorithms, ML-DSA, SLH-DSA, and FN-DSA – all finalists from the NIST PQC Competition Round 3.

There are further algorithms under investigation:

3 code-based KEMs, BIKE, HQC, and Classic McEliece – now in the NIST PQC Competition Round 4.

A range of types of PQC signature algorithms in the new PQC: Digital Signature Schemes selection process (still in it's early days).

### Notes:

LMS and XMSS are initially derived from RFC 8554 and RFC 8391. “stateful” means, in this case, that the algorithms both require the computation of a new private key for each signature. The main reason for this is they are both developed based on Merkle trees, which are Quantum Safe without relying on trap-door functions, but on the other hand break badly if the same private key values are used to compute a signature. Updating a private key involves incrementing a counter. Both schemes offer a way to break the key space into sub-trees: HSS for LMS, and XMSS/MT for XMSS.

ML-KEM and ML-DSA were formerly known as Kyber and Dilithium – of the 4 algorithms getting standardised now NIST has expressed a preference for their use. They are both based on lattice theory. SLH-DSA, formerly SPHINCS+, is a state-less hash based signature scheme, so offers, at the expense of key and signature size, a safer hash-based signature mechanism than the stateful schemes. FN-DSA, formerly Falcon (the FN though stands for FFT over NTRU lattices), is also lattice based, but uses a different approach to Dilithium built on Fast-Fourier-Transforms, which provides smaller signatures at the expense of some additional complexity in calculation. At the moment FN-DSA requires the use of some floating point calculations – there are still concerns that in some cases this may lead to side channels as secret data is involved in these calculations.

It is worth also being aware that FrodoKEM (another NIST round 3 candidate) and Classic McEliece are also being standardised by the BSI in Germany. NTRU, which was listed as the runner up for Kyber and also considered as the replacement if an acceptable solution to patents that affected the Kyber submission could not be addressed, is also seeing a lot of use, especially in Japan, where Nippon Telecom are making extensive use of it.

## How are these algorithms different?

### In Brief:

In general the algorithms differ from classical public key algorithms due the larger key, signature, and in the case of KEMs, payload sizes required.

Signatures, at least the stateless algorithms are otherwise the same as what we are used to.

KEM, short for Key Encapsulation Mechanism, algorithms do not quite fit with either the conventional key transport model, or the conventional key agreement model. They are kind of in-between. This does have wider implications, as protocols which assume it's possible to generate a symmetric session key use it and then apply a public key based transport mechanism after the fact require modification.

### Notes:

Okay, first thing to remember in the key size discussion, is a lot of conversation is about key sizes relative to RSA 2048 and EC P-256, which are worth 112 bits and 128 bits of security respectively. You need at least RSA 3072 to get 128 bits of security. An RSA key with 256 bits of pre-quantum security would now be around 15360 bits long and P-521 is the closest with have to a 256 bit secure EC key. Second thing to remember is that with further advances in the classical space, these sizes are only going to go up, and in the face of a Cryptographically Relevant Quantum Computer security basically goes to 0.

So keeping that in mind,

- ML-KEM public keys range from 6400 bits (800 bytes) to 12544 bits (1568 bytes), payloads (encapsulations) are the same as the corresponding public key. The private keys are  just over twice the size of the public keys (1632 bytes to 3168 bytes).  Private keys can be stored just by using the seed (64 bytes) where the holder is able to do the expansion on demand.

- ML-DSA public keys range from 10496 bits (1312 bytes) to 20736 bits (2592 bytes), signature lengths range from 19832 bits (2479 bytes) to 37232 bits (4654 bytes). In this case the private keys are just under double the size of the public keys (2528 bytes to 4864 bytes). Private keys can be stored just by using the seed (32 bytes) where the holder is able to do the expansion on demand.

- SLH-DSA public keys range from 256 bits (32 bytes) to 512 bits (64 bytes). Before everyone breathes a sigh of relief, signature sizes range from 62848 bits (7856 bytes) to 398848 bits (49856 bytes). The private keys are twice the size of their corresponding public key (64 bytes to 128 bytes).

- FN-DSA public keys range from 7176 bits (897 bytes) to 14344 bits (1793 bytes), signature sizes range from 5728 bits (716 bytes) to 10680 bits (1335 bytes). Private keys are around 50% bigger than public keys (1281 bytes to 2305 bytes).

## So how does a KEM differ from what we do now?

### In Brief:

With key transport (say RSA or ElGamal), we usually generate a session key and then encrypt it producing an encapsulation using the public key. Recovering the session key involves the private key corresponding to the public key being used

With key agreement (Diffie-Hellman, either over a finite field or an elliptic curve), we have a coordinated exchange of public keys between users to allow the creation of a shared secret value. This calculation involves the use of the private keys as well.

With a KEM a public key is used to generate a secret and an encapsulation in one step. The secret is then used to generate a local session key using a Key Derivation Function (KDF).

### Notes:

In some ways the KEM construct is not new, there's an RSA-based one that appears in NIST SP 800-56B and was also written up in RFC 5990 (published 2010). The PQC KEMs all appear to follow the same principal – the shared secret passed in the KEM encapsulation is supposed to be used as source material to derive any session keys associated using a KDF, some KEMs, such as the RSA one require this already, it could be argued that the original formulation of ML-KEM did not, however for anyone taking “KEM agility” seriously the KDF is always required as some KEMs always require one, as is the case in the draft RFC for using KEMs with CMS and a KDF does no harm. A late modification for ML-KEM removed the extra hashing on the encapsulated secret, so ML-KEM is actually in this family now.

On the client side of a protocol this means the holder of a public key needs to follow a path of:

1. [secret, encapsulation] = generate(public key)

2. key = KDF(secret, otherInfo)

3. (use key, send processed data), send encapsulation (possibly in opposite order)

where otherInfo is some other, usually all public, data both the sender and the receiver will use in their KDFs.

In a way this is a lot closer to key agreement at the start, but finishes with a quasi key transport when the encapsulation is sent.

On the server side protocol falls out as:

1. secret = extract(private key, encapsulation)

2. key = KDF(secret, otherInfo)

3. receive data, use key

which for all intents and purposes is identical to what you would see in RSA key transport with a KDF.

## Composite Signature/Certificate Formats

Dual key certificates are primarily proposed as migration aids. The issue people are facing is that it's highly unlikely flicking a “magic switch” to move everything to PQC from classical algorithms will go well.

### Catalyst

There's one dual key certificate format which is already standardised – known generally as Catalyst (from the original ISARA naming), or the Trustkovsky Draft (the now expired IETF draft which was meant to describe it). X.509 now includes certificate extensions that allow the specification of an alternative public key, alternative signature algorithm, and an alternative signature value.

While the altPublicKey and altSignatureAlgorithm are easy to deal with, the alternative signature value does have a specific method for determining the signature value which is different from the traditional X.509 approach. Another issue is there is currently no standard for how to describe these in a certification request (at the moment people appear to be falling back to the Trustkovsky Draft to deal with it).

### Chameleon

Also known as Delta, is another dual key approach going through the IETF which takes things a step further as it's really a dual certificate approach. Also X.509 extension based, a Chameleon extension contains set of differences which can be used to create a second certificate using the extension carrying certificate, or host certificate, as the basis. In this case it's possible to swap out the host certificate for the delta constructed certificate if desired, the certificate verification algorithm remains the same for both the host and the constructed delta.

### Certificate Binding

Also an extension based mechanism, currently working its way through the IETF. Certificate Binding provides an attribute for certification requests as well as an additional extension which allows a CA to issue two certificates specifically marked as related to each other, ideally allowing a user to swap certificates seamlessly depending on whether they need to use a classical algorithm or a PQC for establishing their identity.

### Composite

Composite, also an IETF standard, is built on the idea of specifying a new signature algorithm, which is a composite of two existing one. Originally a single OID was provided for the algorithm, however, as the draft has progressed specific OIDs have been chosen for mixing specific algorithms. While composite does provide a general way of including multiple public keys and signatures in a certificate, the downside is the it does require support for the algorithm OIDs used – unlike extension based approaches the signature and public key algorithm identifiers in an X.509 certificate cannot be safely ignored.

# Java Examples

## Ex. 1: Generating an ML-DSA signature

Note: This example will generate a deterministic ML-DSA signature as initSign() has not been provided with a SecureRandom. There is a stated preference for the use of non-deterministic signatures so unless deterministic signatures are specifically required, initSign() should be passed a SecureRandom.

```java
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.SecureRandom;
import java.security.Security;
import java.security.Signature;

import org.bouncycastle.jcajce.spec.MLDSAParameterSpec;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
import org.bouncycastle.util.Strings;
import org.bouncycastle.util.encoders.Hex;

/**
 * Example of ML-DSA signature generation using the ML-DSA-65 parameter set.
 */
public class MLDSAExample
{
    private static byte[] msg = Strings.toByteArray("Hello, world!");

    public static void main(String[] args)
        throws Exception
    {
        Security.addProvider(new BouncyCastleProvider());

        // Generate ML-DSA key pair.
        KeyPairGenerator kpGen = KeyPairGenerator.getInstance("MLDSA", "BC");

        kpGen.initialize(MLDSAParameterSpec.ml_dsa_65);

        KeyPair kp = kpGen.generateKeyPair();

        // Create ML-DSA signature object.
        Signature mlDsa = Signature.getInstance("MLDSA");

        // Create ML-DSA signature - without a SecureRandom we are
        // indicating we want to create a Deterministic one.
        mlDsa.initSign(kp.getPrivate());

        mlDsa.update(msg);

        byte[] siganture = mlDsa.sign();

        // Verify ML-DSA signature.
        mlDsa.initVerify(kp.getPublic());

        mlDsa.update(msg);

        if (mlDsa.verify(siganture))
        {
            System.out.println("ML-DSA-65 signature created and verified successfully");
            System.exit(0);
        }

        System.exit(1);
    }
}
```

## Ex. 2: Generating a Pre-hash ML-DSA signature

```java
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.Security;
import java.security.Signature;

import org.bouncycastle.jcajce.spec.MLDSAParameterSpec;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
import org.bouncycastle.util.Strings;

/**
 * Example of Pre-hash ML-DSA signature generation using the ML-DSA-65 parameter set.
 */
public class PrehashMLDSAExample
{
    private static byte[] msg = Strings.toByteArray("Hello, world!");

    public static void main(String[] args)
        throws Exception
    {
        Security.addProvider(new BouncyCastleProvider());

        // Generate ML-DSA key pair - this will result in a key pair specifically encoded
        // for use with pre-hash ML-DSA signatures only.
        KeyPairGenerator kpGen = KeyPairGenerator.getInstance("SHA512withMLDSA", "BC");

        kpGen.initialize(MLDSAParameterSpec.ml_dsa_65_with_sha512);

        KeyPair kp = kpGen.generateKeyPair();

        // Create ML-DSA signature object.
        Signature mlDsa = Signature.getInstance("SHA512withMLDSA");

        // Create ML-DSA signature - without a SecureRandom we are
        // indicating we want to create a Deterministic one.
        mlDsa.initSign(kp.getPrivate());

        mlDsa.update(msg);

        byte[] siganture = mlDsa.sign();

        // Verify ML-DSA signature.
        mlDsa.initVerify(kp.getPublic());

        mlDsa.update(msg);

        if (mlDsa.verify(siganture))
        {
            System.out.println("Pre-hash ML-DSA-65 signature created and verified successfully");
            System.exit(0);
        }

        System.exit(1);
    }
}
```

## Ex. 3: Generating an AES key with a KEM in Java 21

```java
import org.bouncycastle.jcajce.spec.KTSParameterSpec;
import org.bouncycastle.jcajce.spec.MLKEMParameterSpec;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
import org.bouncycastle.util.Arrays;
import org.bouncycastle.util.encoders.Hex;

import javax.crypto.KEM;
import javax.crypto.SecretKey;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PublicKey;
import java.security.SecureRandom;
import java.security.Security;

/**
 * Use of a Java 21 KEM based on ML-KEM to generate a 128 bit AES key.
 */
public class Java21MLKEMExample
{
    public static void main(String[] args)
        throws Exception
    {
        Security.addProvider(new BouncyCastleProvider());

        // Receiver side
        KeyPairGenerator g = KeyPairGenerator.getInstance("ML-KEM", "BC");

        g.initialize(MLKEMParameterSpec.ml_kem_768, new SecureRandom());

        KeyPair kp = g.generateKeyPair();
        PublicKey pkR = kp.getPublic();

        // Sender side
        KEM kemS = KEM.getInstance("ML-KEM");

        // Specify we want a 128 bit AES key using pkR
        KTSParameterSpec ktsSpec = new KTSParameterSpec.Builder("AES", 128).build();
        KEM.Encapsulator e = kemS.newEncapsulator(pkR, ktsSpec, null);

        KEM.Encapsulated enc = e.encapsulate();

        SecretKey secS = enc.key();

        byte[] em = enc.encapsulation();

        // Receiver side
        KEM kemR = KEM.getInstance("ML-KEM");
        KEM.Decapsulator d = kemR.newDecapsulator(kp.getPrivate(), ktsSpec);
        SecretKey secR = d.decapsulate(em);

        // secS and secR will be identical
        if (Arrays.areEqual(secS.getEncoded(), secR.getEncoded()))
        {
            System.out.println("AES key generated successfully: " + Hex.toHexString(secS.getEncoded()));
            System.exit(0);
        }

        System.exit(1);
    }
}
```

## Ex. 4: Pre-Java 21 ML-KEM AES key generation example

```java
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.SecureRandom;
import java.security.Security;

import javax.crypto.KeyGenerator;

import org.bouncycastle.jcajce.SecretKeyWithEncapsulation;
import org.bouncycastle.jcajce.spec.KEMExtractSpec;
import org.bouncycastle.jcajce.spec.KEMGenerateSpec;
import org.bouncycastle.jcajce.spec.MLKEMParameterSpec;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
import org.bouncycastle.util.Arrays;
import org.bouncycastle.util.encoders.Hex;

/**
 * Pre Java 21 ML-KEM shared key generation using the KeyGenerator class.
 */
public class PreJDK21MLKEMKeyGenExample
{
    public static void main(String[] args)
        throws Exception
    {
        Security.addProvider(new BouncyCastleProvider());

        // generate ML-KEM-512 key pair.
        KeyPairGenerator kpg = KeyPairGenerator.getInstance("ML-KEM", "BC");

        kpg.initialize(MLKEMParameterSpec.ml_kem_512, new SecureRandom());

        KeyPair kp = kpg.generateKeyPair();

        // Create the KeyGenerator - there are specific specs for creating encapsulations (KEMGenerateSpec) and for
        // extracting a secret key from an encapsulation using the private key (KEMExtractSpec)
        KeyGenerator keyGen = KeyGenerator.getInstance("ML-KEM", "BC");

        // initialise for creating an encapsulation and shared secret.
        keyGen.init(new KEMGenerateSpec(kp.getPublic(), "AES", 128), new SecureRandom());

        // SecretKeyWithEncapsulation is the class to use as the secret key, it has additional
        // methods on it for recovering the encapsulation as well.
        SecretKeyWithEncapsulation secEnc1 = (SecretKeyWithEncapsulation)keyGen.generateKey();

        keyGen.init(new KEMExtractSpec(kp.getPrivate(), secEnc1.getEncapsulation(), "AES", 128));

        // initialise for extracting the shared secret from the encapsulation.
        SecretKeyWithEncapsulation secEnc2 = (SecretKeyWithEncapsulation)keyGen.generateKey();

        // a quick check to make sure we got the same answer on both sides.
        if (Arrays.areEqual(secEnc1.getEncoded(), secEnc2.getEncoded()))
        {
            System.out.println("AES key generated successfully: " + Hex.toHexString(secEnc1.getEncoded()));
            System.exit(0);
        }

        System.exit(1);
    }
}
```

## Ex. 5: Pre-Java 21 ML-KEM AES-KWP Example

```java
import java.security.Key;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.SecureRandom;
import java.security.Security;

import javax.crypto.Cipher;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;
import org.bouncycastle.jcajce.provider.asymmetric.MLKEM;
import org.bouncycastle.jcajce.spec.KEMParameterSpec;
import org.bouncycastle.jcajce.spec.KTSParameterSpec;
import org.bouncycastle.jcajce.spec.MLKEMParameterSpec;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
import org.bouncycastle.pqc.jcajce.spec.KyberParameterSpec;
import org.bouncycastle.util.Arrays;
import org.bouncycastle.util.encoders.Hex;

/**
 * Pre Java 21 ML-KEM shared key based key wrap using the Cipher class.
 */
public class PreJDK21MLKEMWrapExample
{
    public static void main(String[] args)
        throws Exception
    {
        Security.addProvider(new BouncyCastleProvider());

        // define our key to be wrapped
        byte[] aesKey = Hex.decode("0102030405060708090a0b0c0d0e0f");

        SecretKey key = new SecretKeySpec(aesKey, "AES");

        // generate an ML-KEM-768 key pair
        KeyPairGenerator kpg = KeyPairGenerator.getInstance("MLKEM", "BC");

        kpg.initialize(MLKEMParameterSpec.ml_kem_768, new SecureRandom());

        KeyPair kp = kpg.generateKeyPair();

        // specify we want to use 256 bit AES-KWP for wrapping our key
        KTSParameterSpec ktsParameterSpec = new KTSParameterSpec.Builder("AES-KWP", 256).build();

        // set up the wrapping cipher
        Cipher w1 = Cipher.getInstance("MLKEM", "BC");

        w1.init(Cipher.WRAP_MODE, kp.getPublic(), ktsParameterSpec);

        // wrap the key
        byte[] data = w1.wrap(key);

        // set up the unwrapping cipher
        Cipher w2 = Cipher.getInstance("MLKEM", "BC");

        w2.init(Cipher.UNWRAP_MODE, kp.getPrivate(), ktsParameterSpec);

        // unwrap the encrypted key
        Key k = w2.unwrap(data, "AES", Cipher.SECRET_KEY);

        // check that the key was recovered successfully
        if (Arrays.areEqual(aesKey, k.getEncoded()))
        {
            System.out.println("AES key successfully wrapped and unwrapped");
            System.exit(0);
        }

        System.exit(1);
    }
}
```

## Ex. 6: ML-DSA PKCS#10 Generation

```
import java.math.BigInteger;
import java.security.KeyPair;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.SecureRandom;
import java.security.Security;

import org.bouncycastle.asn1.x500.X500Name;
import org.bouncycastle.jcajce.spec.MLDSAParameterSpec;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
import org.bouncycastle.operator.ContentSigner;
import org.bouncycastle.operator.jcajce.JcaContentSignerBuilder;
import org.bouncycastle.operator.jcajce.JcaContentVerifierProviderBuilder;
import org.bouncycastle.pkcs.PKCS10CertificationRequest;
import org.bouncycastle.pkcs.PKCS10CertificationRequestBuilder;
import org.bouncycastle.pkcs.jcajce.JcaPKCS10CertificationRequestBuilder;

/**
 * Create a basic PKCS10 request using ML-DSA-44.
 */
public class MLDSAPKCS10Example
{
    public static void main(String[] args)
        throws Exception
    {
        Security.addProvider(new BouncyCastleProvider());
        // generate an ML-DSA-44 key pair
        KeyPairGenerator kpGen = KeyPairGenerator.getInstance("ML-DSA", "BC");

        kpGen.initialize(MLDSAParameterSpec.ml_dsa_44, new SecureRandom());

        KeyPair kp = kpGen.generateKeyPair();

        PrivateKey privKey = kp.getPrivate();
        PublicKey pubKey = kp.getPublic();

        X500Name subject = new X500Name("CN=ML-DSA Certification Request");

        //
        // Create
        //
        ContentSigner sigGen = new JcaContentSignerBuilder("ML-DSA-44").setProvider("BC").build(privKey);

        PKCS10CertificationRequestBuilder pkcs10Gen = new JcaPKCS10CertificationRequestBuilder(subject, pubKey);

        PKCS10CertificationRequest pkcs10 = pkcs10Gen.build(sigGen);

        if (pkcs10.isSignatureValid(new
JcaContentVerifierProviderBuilder().setProvider("BC").build(pkcs10.getSubjectPublicKeyInfo())))
        {
            System.out.println("ML-DSA PKCS#10 request verified");
            System.exit(0);
        }

        System.exit(1);
    }
}
```

## Ex. 7: ML-DSA X.509 Certificate Generation

```
import java.math.BigInteger;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.SecureRandom;
import java.security.Security;
import java.security.cert.X509Certificate;
import java.util.Date;

import org.bouncycastle.asn1.ASN1ObjectIdentifier;
import org.bouncycastle.asn1.DERSequence;
import org.bouncycastle.asn1.x500.X500Name;
import org.bouncycastle.asn1.x509.Extension;
import org.bouncycastle.asn1.x509.GeneralName;
import org.bouncycastle.asn1.x509.GeneralNames;
import org.bouncycastle.asn1.x509.KeyPurposeId;
import org.bouncycastle.cert.X509v3CertificateBuilder;
import org.bouncycastle.cert.jcajce.JcaX509CertificateConverter;
import org.bouncycastle.cert.jcajce.JcaX509v3CertificateBuilder;
import org.bouncycastle.jcajce.spec.MLDSAParameterSpec;
import org.bouncycastle.jce.X509KeyUsage;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
import org.bouncycastle.operator.ContentSigner;
import org.bouncycastle.operator.jcajce.JcaContentSignerBuilder;

/**
 * Create a basic self-signed certificate using ML-DSA-44.
 */
public class MLDSACertificateExample
{
 private static long ONE_YEAR = 365 * 24 * 60 * 60 * 1000L;

 public static void main(String[] args)
 throws Exception
 {
 Security.addProvider(new BouncyCastleProvider());

 // generate an ML-DSA-44 key pair
 KeyPairGenerator kpGen = KeyPairGenerator.getInstance("ML-DSA", "BC");

 kpGen.initialize(MLDSAParameterSpec.ml_dsa_44, new SecureRandom());

 KeyPair kp = kpGen.generateKeyPair();

 PrivateKey privKey = kp.getPrivate();
 PublicKey pubKey = kp.getPublic();

 X500Name issuer = new X500Name("CN=ML-DSA Certificate");

 //
 // create base certificate - version 3
 //
 ContentSigner sigGen = new JcaContentSignerBuilder("ML-DSA-44").setProvider("BC").build(privKey);

 X509v3CertificateBuilder certGen = new JcaX509v3CertificateBuilder(
 issuer, BigInteger.valueOf(1),
 new Date(System.currentTimeMillis() - 50000),
 new Date(System.currentTimeMillis() + ONE_YEAR),
 issuer, pubKey)
 .addExtension(Extension.keyUsage, true,
 new X509KeyUsage(X509KeyUsage.encipherOnly))
 .addExtension(Extension.extendedKeyUsage, true,
 new DERSequence(KeyPurposeId.anyExtendedKeyUsage))
 .addExtension(Extension.subjectAlternativeName, true,
 new GeneralNames(new GeneralName(GeneralName.rfc822Name, "test@test.test")));

        X509Certificate cert = new JcaX509CertificateConverter().setProvider("BC").getCertificate(certGen.build(sigGen));

        //
        // check validity
        //
        cert.checkValidity(new Date());

        try
        {
            cert.verify(cert.getPublicKey());

            System.out.println("ML-DSA certificate verified");
            System.exit(0);
        }
        catch (Exception e)
        {
            System.exit(1);
        }
    }
}
```

## Ex. 8: ECDSA ML-DSA X.509 Dual Key Certificate Generation

```
import java.math.BigInteger;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.SecureRandom;
import java.security.Security;
import java.security.cert.X509Certificate;
import java.util.Date;

import org.bouncycastle.asn1.ASN1ObjectIdentifier;
import org.bouncycastle.asn1.DERSequence;
import org.bouncycastle.asn1.x500.X500Name;
import org.bouncycastle.asn1.x500.X500NameBuilder;
import org.bouncycastle.asn1.x509.Extension;
import org.bouncycastle.asn1.x509.GeneralName;
import org.bouncycastle.asn1.x509.GeneralNames;
import org.bouncycastle.asn1.x509.KeyPurposeId;
import org.bouncycastle.asn1.x509.SubjectAltPublicKeyInfo;
import org.bouncycastle.asn1.x509.SubjectPublicKeyInfo;
import org.bouncycastle.cert.X509CertificateHolder;
import org.bouncycastle.cert.X509v3CertificateBuilder;
import org.bouncycastle.cert.jcajce.JcaX509CertificateConverter;
import org.bouncycastle.cert.jcajce.JcaX509CertificateHolder;
import org.bouncycastle.cert.jcajce.JcaX509v3CertificateBuilder;
import org.bouncycastle.jcajce.spec.MLDSAParameterSpec;
import org.bouncycastle.jce.X509KeyUsage;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
import org.bouncycastle.jce.spec.ECNamedCurveGenParameterSpec;
import org.bouncycastle.operator.ContentSigner;
import org.bouncycastle.operator.jcajce.JcaContentSignerBuilder;
import org.bouncycastle.operator.jcajce.JcaContentVerifierProviderBuilder;

/**
 * Create a self-signed P-256 ECDSA certificate with an alt signature and public key based on ML-DSA-44
 */
public class X509AltCertificateExample
{
    private static long ONE_YEAR = 365 * 24 * 60 * 60 * 1000L;

    public static void main(String[] args)
        throws Exception
    {
        Security.addProvider(new BouncyCastleProvider());

        KeyPairGenerator kpGen = KeyPairGenerator.getInstance("ML-DSA", "BC");

        kpGen.initialize(MLDSAParameterSpec.ml_dsa_44, new SecureRandom());

        KeyPair kp = kpGen.generateKeyPair();

        PrivateKey privKey = kp.getPrivate();
        PublicKey pubKey = kp.getPublic();

        KeyPairGenerator ecKpGen = KeyPairGenerator.getInstance("EC", "BC");

        ecKpGen.initialize(new ECNamedCurveGenParameterSpec("P-256"), new SecureRandom());

        KeyPair ecKp = ecKpGen.generateKeyPair();
        PrivateKey ecPrivKey = ecKp.getPrivate();
        PublicKey ecPubKey = ecKp.getPublic();

        X500Name issuer = new X500Name("CN=ML-DSA ECDSA Alt Extension Certificate");

        //
        // create base certificate - version 3
        //
        ContentSigner sigGen = new JcaContentSignerBuilder("SHA256withECDSA").setProvider("BC").build(ecPrivKey);

        ContentSigner altSigGen = new JcaContentSignerBuilder("ML-DSA-44").setProvider("BC").build(privKey);

        X509v3CertificateBuilder certGen = new JcaX509v3CertificateBuilder(
            issuer, BigInteger.valueOf(1),
            new Date(System.currentTimeMillis() - 50000),
            new Date(System.currentTimeMillis() + ONE_YEAR),
            issuer, ecPubKey)
            .addExtension(Extension.keyUsage, true,
                new X509KeyUsage(X509KeyUsage.digitalSignature))
            .addExtension(Extension.extendedKeyUsage, true,
                new DERSequence(KeyPurposeId.anyExtendedKeyUsage))
            .addExtension(new ASN1ObjectIdentifier("2.5.29.17"), true,
                new GeneralNames(new GeneralName(GeneralName.rfc822Name, "test@test.test")))
            .addExtension(Extension.subjectAltPublicKeyInfo, false,
SubjectAltPublicKeyInfo.getInstance(kp.getPublic().getEncoded()));

        X509Certificate cert = new JcaX509CertificateConverter().setProvider("BC").getCertificate(certGen.build(sigGen, false,
altSigGen));

        // check validity and verify

        cert.checkValidity(new Date());

        cert.verify(cert.getPublicKey());

        // create a certificate holder to allow checking of the altSignature.

        X509CertificateHolder certHolder = new JcaX509CertificateHolder(cert);

        SubjectPublicKeyInfo altPubKey =
SubjectPublicKeyInfo.getInstance(certHolder.getExtension(Extension.subjectAltPublicKeyInfo).getParsedValue());

        if (certHolder.isAlternativeSignatureValid(new JcaContentVerifierProviderBuilder().setProvider("BC").build(altPubKey)))
        {
            System.out.println("alternative signature verified on certificate");
            System.exit(0);
        }

        System.exit(1);
    }
}
```

## Ex. 9: ML-KEM with CMS

```
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.SecureRandom;
import java.security.Security;
import java.util.Arrays;
import java.util.Collection;

import org.bouncycastle.asn1.cmp.CMPCertificate;
import org.bouncycastle.asn1.nist.NISTObjectIdentifiers;
import org.bouncycastle.asn1.x509.AlgorithmIdentifier;
import org.bouncycastle.cert.cmp.CMSProcessableCMPCertificate;
import org.bouncycastle.cert.jcajce.JcaX509CertificateConverter;
import org.bouncycastle.cert.jcajce.JcaX509ExtensionUtils;
import org.bouncycastle.cms.CMSAlgorithm;
import org.bouncycastle.cms.CMSEnvelopedData;
import org.bouncycastle.cms.CMSEnvelopedDataGenerator;
import org.bouncycastle.cms.CMSProcessableByteArray;
import org.bouncycastle.cms.KEMRecipientId;
import org.bouncycastle.cms.RecipientInformation;
import org.bouncycastle.cms.RecipientInformationStore;
import org.bouncycastle.cms.jcajce.JceCMSContentEncryptorBuilder;
import org.bouncycastle.cms.jcajce.JceKEMEnvelopedRecipient;
import org.bouncycastle.cms.jcajce.JceKEMRecipientInfoGenerator;
import org.bouncycastle.jcajce.spec.MLKEMParameterSpec;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
import org.bouncycastle.util.Strings;

/**
 * Simple message encryption using ML-KEM-768
 */
public class MLKEMCMSExample
{
 private static byte[] msg = Strings.toByteArray("Hello, world!");
```

```java
 public static void main(String[] args)
throws Exception
 {
     Security.addProvider(new BouncyCastleProvider());

     // Generate the ML-KEM-768 key.
     KeyPairGenerator kpg = KeyPairGenerator.getInstance("MLKEM", "BC");

     kpg.initialize(MLKEMParameterSpec.ml_kem_768, new SecureRandom());

     KeyPair kp = kpg.generateKeyPair();

     // Setup the CMS EnvelopedData generator
     CMSEnvelopedDataGenerator edGen = new CMSEnvelopedDataGenerator();

     // we will skip generating a certificate and just use a subject key ID.
     JcaX509ExtensionUtils x509Utils = new JcaX509ExtensionUtils();
     byte[] subjectKeyId = x509Utils.createSubjectKeyIdentifier(kp.getPublic()).getKeyIdentifier();

     // add the KEM recipientInfo RFC 9629 style - we're using SHAKE256 as the KDF
     edGen.addRecipientInfoGenerator(new JceKEMRecipientInfoGenerator(subjectKeyId, kp.getPublic(), CMSAlgorithm.AES256_WRAP).setKDF(
            new AlgorithmIdentifier(NISTObjectIdentifiers.id_shake256)));

     // Add the payload and specify encryption using AES-256.
     CMSEnvelopedData encryptedData = edGen.generate(
            new CMSProcessableByteArray(msg),
            new JceCMSContentEncryptorBuilder(CMSAlgorithm.AES256_CBC).setProvider("BC").build());

     // These are the steps to recover the data - usually we would use the issuer/serial-number
     RecipientInformation recInfo = encryptedData.getRecipientInfos().get(new KEMRecipientId(subjectKeyId));

     byte[] recData = recInfo.getContent(new JceKEMEnvelopedRecipient(kp.getPrivate()).setProvider("BC"));

     if (Arrays.equals(msg, recData))
     {
            System.out.println("message successfully encrypted and decrypted");
            System.exit(0);
     }

     System.exit(1);
   }
}
```

## Ex. 10: ML-KEM with CRMF/CMP

```java
import java.math.BigInteger;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.Security;
import java.util.Date;

import org.bouncycastle.asn1.cmp.CMPCertificate;
import org.bouncycastle.asn1.cmp.PKIBody;
import org.bouncycastle.asn1.cmp.PKIStatus;
import org.bouncycastle.asn1.cmp.PKIStatusInfo;
import org.bouncycastle.asn1.crmf.CertTemplate;
import org.bouncycastle.asn1.crmf.SubsequentMessage;
import org.bouncycastle.asn1.nist.NISTObjectIdentifiers;
import org.bouncycastle.asn1.x500.X500Name;
import org.bouncycastle.asn1.x509.AlgorithmIdentifier;
import org.bouncycastle.asn1.x509.BasicConstraints;
import org.bouncycastle.asn1.x509.Extension;
import org.bouncycastle.asn1.x509.GeneralName;
import org.bouncycastle.asn1.x509.SubjectPublicKeyInfo;
import org.bouncycastle.cert.CertException;
import org.bouncycastle.cert.CertIOException;
import org.bouncycastle.cert.X509CertificateHolder;
import org.bouncycastle.cert.X509v3CertificateBuilder;
import org.bouncycastle.cert.cmp.CMSProcessableCMPCertificate;
import org.bouncycastle.cert.cmp.CertificateConfirmationContent;
import org.bouncycastle.cert.cmp.CertificateConfirmationContentBuilder;
import org.bouncycastle.cert.cmp.ProtectedPKIMessage;
import org.bouncycastle.cert.cmp.ProtectedPKIMessageBuilder;
import org.bouncycastle.cert.crmf.CertificateRepMessage;
import org.bouncycastle.cert.crmf.CertificateRepMessageBuilder;
import org.bouncycastle.cert.crmf.CertificateReqMessages;
import org.bouncycastle.cert.crmf.CertificateReqMessagesBuilder;
import org.bouncycastle.cert.crmf.CertificateRequestMessage;
import org.bouncycastle.cert.crmf.CertificateResponse;
import org.bouncycastle.cert.crmf.CertificateResponseBuilder;
import org.bouncycastle.cert.crmf.jcajce.JcaCertificateRequestMessageBuilder;
import org.bouncycastle.cert.jcajce.JcaX509CertificateConverter;
import org.bouncycastle.cert.jcajce.JcaX509v3CertificateBuilder;
import org.bouncycastle.cms.CMSAlgorithm;
import org.bouncycastle.cms.CMSEnvelopedData;
import org.bouncycastle.cms.CMSEnvelopedDataGenerator;
import org.bouncycastle.cms.jcajce.JceCMSContentEncryptorBuilder;
import org.bouncycastle.cms.jcajce.JceKEMEnvelopedRecipient;
import org.bouncycastle.cms.jcajce.JceKEMRecipientInfoGenerator;
import org.bouncycastle.jcajce.spec.MLDSAParameterSpec;
import org.bouncycastle.jcajce.spec.MLKEMParameterSpec;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
import org.bouncycastle.operator.ContentSigner;
import org.bouncycastle.operator.MacCalculator;
import org.bouncycastle.operator.OperatorCreationException;
import org.bouncycastle.operator.PBEMacCalculatorProvider;
import org.bouncycastle.operator.jcajce.JcaContentSignerBuilder;
import org.bouncycastle.operator.jcajce.JcaContentVerifierProviderBuilder;
import org.bouncycastle.operator.jcajce.JcaDigestCalculatorProviderBuilder;
import org.bouncycastle.pkcs.jcajce.JcePBMac1CalculatorBuilder;
import org.bouncycastle.pkcs.jcajce.JcePBMac1CalculatorProviderBuilder;

/**
 * Simple message encryption using ML-KEM-768
 */
public class MLKEMCRMFExample
{
    private static X509CertificateHolder makeV3Certificate(String _subDN, KeyPair issKP)
        throws OperatorCreationException, CertException, CertIOException
    {
        PrivateKey issPriv = issKP.getPrivate();
        PublicKey issPub = issKP.getPublic();

        X509v3CertificateBuilder certGen = new JcaX509v3CertificateBuilder(
            new X500Name(_subDN),
            BigInteger.valueOf(System.currentTimeMillis()),
            new Date(System.currentTimeMillis()),
            new Date(System.currentTimeMillis() + (1000L * 60 * 60 * 24 * 100)),
            new X500Name(_subDN),
            issKP.getPublic());

        certGen.addExtension(Extension.basicConstraints, true, new BasicConstraints(0));

        ContentSigner signer = new JcaContentSignerBuilder("ML-DSA").build(issPriv);

        X509CertificateHolder certHolder = certGen.build(signer);

        return certHolder;
    }

    private static X509CertificateHolder makeEndEntityCertificate(SubjectPublicKeyInfo pubKey, X500Name _subDN,
                                                                  KeyPair issKP, String _issDN)
        throws OperatorCreationException, CertException, CertIOException
    {
        PrivateKey issPriv = issKP.getPrivate();
        PublicKey issPub = issKP.getPublic();

        X509v3CertificateBuilder certGen = new JcaX509v3CertificateBuilder(
            new X500Name(_issDN),
            BigInteger.valueOf(System.currentTimeMillis()),
            new Date(System.currentTimeMillis()),
            new Date(System.currentTimeMillis() + (1000L * 60 * 60 * 24 * 100)),
            _subDN,
            pubKey);

        certGen.addExtension(Extension.basicConstraints, true, new BasicConstraints(false));

        ContentSigner signer = new JcaContentSignerBuilder("ML-DSA").build(issPriv);

        X509CertificateHolder certHolder = certGen.build(signer);

        return certHolder;
    }

    public static void main(String[] args)
        throws Exception
    {
        Security.addProvider(new BouncyCastleProvider());

        // set up - in this section we are just establishing the private key and the certificate for our ML-DSA CA
        GeneralName sender = new GeneralName(new X500Name("CN=ML-KEM Subject"));
        GeneralName recipient = new GeneralName(new X500Name("CN=ML-DSA Issuer"));

        KeyPairGenerator dilKpGen = KeyPairGenerator.getInstance("ML-DSA", "BC");

        dilKpGen.initialize(MLDSAParameterSpec.ml_dsa_87);

        KeyPair dilKp = dilKpGen.generateKeyPair();

        X509CertificateHolder caCert = makeV3Certificate("CN=ML-DSA Issuer", dilKp);

        // First step for the client - we generate a key pair
        KeyPairGenerator mlKemKpGen = KeyPairGenerator.getInstance("ML-KEM", "BC");

        mlKemKpGen.initialize(MLKEMParameterSpec.ml_kem_768);

        KeyPair mlKemKp = mlKemKpGen.generateKeyPair();

        // Second Step: We generate our certification request (CRMF).

        BigInteger certReqId = BigInteger.valueOf(System.currentTimeMillis());
        JcaCertificateRequestMessageBuilder certReqBuild = new JcaCertificateRequestMessageBuilder(certReqId);

        certReqBuild
            .setPublicKey(mlKemKp.getPublic())
            .setSubject(X500Name.getInstance(sender.getName()))
            .setProofOfPossessionSubsequentMessage(SubsequentMessage.encrCert);

        CertificateReqMessagesBuilder certReqMsgsBldr = new CertificateReqMessagesBuilder();

        certReqMsgsBldr.addRequest(certReqBuild.build());

        // Third Step: We wrap the CRMF certification request in a CMP message for sending, protecting it using a MAC.
        char[] senderMacPassword = "secret".toCharArray();

        MacCalculator senderMacCalculator = new JcePBMac1CalculatorBuilder("HmacSHA256",
256).setProvider("BC").build(senderMacPassword);

        ProtectedPKIMessage message = new ProtectedPKIMessageBuilder(sender, recipient)
            .setBody(PKIBody.TYPE_INIT_REQ, certReqMsgsBldr.build())
            .build(senderMacCalculator);

        System.out.println("PKIBody.TYPE_INIT_REQ sent");
        // extract

        PBEMacCalculatorProvider macCalcProvider = new JcePBMac1CalculatorProviderBuilder().setProvider("BC").build();

        if (message.verify(macCalcProvider, senderMacPassword))
        {
            System.out.println("PKIBody.TYPE_INIT_REQ verified");
        }
        else
        {
            System.exit(1);
        }

        CertificateReqMessages requestMessages = CertificateReqMessages.fromPKIBody(message.getBody());
        CertificateRequestMessage senderReqMessage = requestMessages.getRequests()[0];
        CertTemplate certTemplate = senderReqMessage.getCertTemplate();

        X509CertificateHolder cert = makeEndEntityCertificate(certTemplate.getPublicKey(), certTemplate.getSubject(), dilKp, "CN=ML-
DSA Issuer");

        // Send response with encrypted certificate
        CMSEnvelopedDataGenerator edGen = new CMSEnvelopedDataGenerator();

        // note: use cert req ID as key ID, don't want to use issuer/serial in this case!

        edGen.addRecipientInfoGenerator(new JceKEMRecipientInfoGenerator(senderReqMessage.getCertReqId().getEncoded(),
            new JcaX509CertificateConverter().setProvider("BC").getCertificate(cert).getPublicKey(),
CMSAlgorithm.AES256_WRAP).setKDF(
            new AlgorithmIdentifier(NISTObjectIdentifiers.id_shake256)));

        CMSEnvelopedData encryptedCert = edGen.generate(
            new CMSProcessableCMPCertificate(cert),
            new JceCMSContentEncryptorBuilder(CMSAlgorithm.AES128_CBC).setProvider("BC").build());

        CertificateResponseBuilder certRespBuilder = new CertificateResponseBuilder(senderReqMessage.getCertReqId(), new
PKIStatusInfo(PKIStatus.granted));

        certRespBuilder.withCertificate(encryptedCert);

        CertificateRepMessageBuilder repMessageBuilder = new CertificateRepMessageBuilder(caCert);

        repMessageBuilder.addCertificateResponse(certRespBuilder.build());

        ContentSigner signer = new JcaContentSignerBuilder("ML-DSA").setProvider("BC").build(dilKp.getPrivate());

        CertificateRepMessage repMessage = repMessageBuilder.build();

        ProtectedPKIMessage responsePkixMessage = new ProtectedPKIMessageBuilder(sender, recipient)
            .setBody(PKIBody.TYPE_INIT_REP, repMessage)
            .build(signer);

        // decrypt the certificate

        System.out.println("PKIBody.TYPE_INIT_REP sent");

        if (responsePkixMessage.verify(new JcaContentVerifierProviderBuilder().build(caCert)))
        {
            System.out.println("PKIBody.TYPE_INIT_REP verified");
        }
        else
        {
            System.exit(1);
        }

        CertificateRepMessage certRepMessage = CertificateRepMessage.fromPKIBody(responsePkixMessage.getBody());

        CertificateResponse certResp = certRepMessage.getResponses()[0];

        if (certResp.hasEncryptedCertificate())
        {
            System.out.println("PKIBody.TYPE_INIT_REP contains encrypted certificate");
        }
        else
        {
            System.exit(1);
        }

        // this is the preferred way of recovering an encrypted certificate
```

```java
        CMPCertificate receivedCMPCert = certResp.getCertificate(new JceKEMEnvelopedRecipient(mlKemKp.getPrivate()));

        X509CertificateHolder receivedCert = new X509CertificateHolder(receivedCMPCert.getX509v3PKCert());

        X509CertificateHolder caCertHolder = certRepMessage.getX509Certificates()[0];

        if (receivedCert.isSignatureValid(new JcaContentVerifierProviderBuilder().build(caCertHolder)))
        {
            System.out.println("Received certificate decrypted and verified against CA certificate");
        }
        else
        {
            System.exit(1);
        }

        // confirmation message calculation

        CertificateConfirmationContent content = new CertificateConfirmationContentBuilder()
            .addAcceptedCertificate(cert, BigInteger.ONE)
            .build(new JcaDigestCalculatorProviderBuilder().build());

        message = new ProtectedPKIMessageBuilder(sender, recipient)
            .setBody(PKIBody.TYPE_CERT_CONFIRM, content)
            .build(senderMacCalculator);

        System.out.println("PKIBody.TYPE_CERT_CONFIRM sent");
        // confirmation receiving

        CertificateConfirmationContent recContent = CertificateConfirmationContent.fromPKIBody(message.getBody());

        if (recContent.getStatusMessages()[0].isVerified(receivedCert, new JcaDigestCalculatorProviderBuilder().build()))
        {
            System.out.println("PKIBody.TYPE_CERT_CONFIRM verified");
        }
        else
        {
            System.exit(1);
        }

        System.exit(0);
    }
}
```

# C# Examples

## Ex. 1: Generating an ML-DSA Signature

```
using System.Text;

using Org.BouncyCastle.Crypto.Generators;
using Org.BouncyCastle.Crypto.Parameters;
using Org.BouncyCastle.Crypto.Signers;
using Org.BouncyCastle.Security;

namespace PqcAlmanac;

/**
 * Example of ML-DSA signature generation using the ML-DSA-65 parameter set.
 */
public class MLDsaExample
{
    private static byte[] Msg = Encoding.UTF8.GetBytes("Hello, world!");

    public static int Main(string[] args)
    {
        var random = new SecureRandom();

        // Generate ML-DSA key pair.
        var kpg = new MLDsaKeyPairGenerator();
        kpg.Init(new MLDsaKeyGenerationParameters(random, MLDsaParameters.ml_dsa_65));
        var kp = kpg.GenerateKeyPair();

        // Create ML-DSA signer.
        var signer = SignerUtilities.InitSigner("ML-DSA-65", forSigning: true, kp.Private, random);

        // Generate ML-DSA signature.
        signer.BlockUpdate(Msg, 0, Msg.Length);
        byte[] signature = signer.GenerateSignature();

        // Verify ML-DSA signature.
        var verifier = SignerUtilities.InitSigner("ML-DSA-65", forSigning: false, kp.Public, random: null);

        verifier.BlockUpdate(Msg, 0, Msg.Length);
        if (verifier.VerifySignature(signature))
        {
            Console.WriteLine("ML-DSA-65 signature created and verified successfully");
            return 0;
        }

        return 1;
    }
}
```

## Ex. 2:  Generating a Shared Secret with a KEM

```
using Org.BouncyCastle.Crypto.Generators;
using Org.BouncyCastle.Crypto.Parameters;
using Org.BouncyCastle.Security;
using Org.BouncyCastle.Utilities;
using Org.BouncyCastle.Utilities.Encoders;

namespace PqcAlmanac;

/**
 * Example of ML-KEM shared secret generation using the ML-KEM-512 parameter set.
 */
public class MLKemExample
{
    public static int Main(string[] args)
    {
        var random = new SecureRandom();

        // Generate ML-KEM-512 key pair.
        var kpg = new MLKemKeyPairGenerator();
        kpg.Init(new MLKemKeyGenerationParameters(random, MLKemParameters.ml_kem_512));
        var kp = kpg.GenerateKeyPair();

        // Generate an encapsulation to the public key and store the secret.
        var encapsulator = KemUtilities.GetEncapsulator("ML-KEM-512");
        encapsulator.Init(kp.Public);

        byte[] encapsulation = new byte[encapsulator.EncapsulationLength];
        byte[] encapSecret = new byte[encapsulator.SecretLength];
        encapsulator.Encapsulate(encapsulation, 0, encapsulation.Length, encapSecret, 0, encapSecret.Length);

        // Extract the secret using the private key.
        var decapsulator = KemUtilities.GetDecapsulator("ML-KEM-512");
        decapsulator.Init(kp.Private);

        byte[] decapSecret = new byte[decapsulator.SecretLength];
        decapsulator.Decapsulate(encapsulation, 0, encapsulation.Length, decapSecret, 0, decapSecret.Length);

        // Check we got the same secret on both sides.
        if (Arrays.AreEqual(encapSecret, decapSecret))
        {
            Console.WriteLine("Shared secret generated successfully: " + Hex.ToHexString(encapSecret));
            return 0;
        }
        return 1;
    }
}
```

## Ex. 3: Creating a Self-Signed Certificate using ML-DSA

```
using Org.BouncyCastle.Asn1;
using Org.BouncyCastle.Asn1.X509;
using Org.BouncyCastle.Crypto.Generators;
using Org.BouncyCastle.Crypto.Operators;
using Org.BouncyCastle.Crypto.Parameters;
using Org.BouncyCastle.Math;
using Org.BouncyCastle.Security;
using Org.BouncyCastle.X509;

namespace PqcAlmanac;

/**
* Create a basic self-signed certificate using ML-DSA-44.
*/
public class MLDsaCertificateExample
{
    public static int Main(string[] args)
    {
        var random = new SecureRandom();

        // Generate an ML-DSA-44 key pair.
        var kpg = new MLDsaKeyPairGenerator();
        kpg.Init(new MLDsaKeyGenerationParameters(random, MLDsaParameters.ml_dsa_44));
        var keyPair = kpg.GenerateKeyPair();

        // Create certificate - version 3
        var signatureFactory = new Asn1SignatureFactory("ML-DSA-44", keyPair.Private);
        var subject = new X509Name("CN=ML-DSA Certificate");

        var certGen = new X509V3CertificateGenerator();
        certGen.SetIssuerDN(issuer: subject);
        certGen.SetNotAfter(DateTime.UtcNow.AddYears(1));
        certGen.SetNotBefore(DateTime.UtcNow.AddMinutes(-1));
        certGen.SetSerialNumber(BigInteger.One);
        certGen.SetSubjectDN(subject);
        certGen.SetPublicKey(keyPair.Public);

        certGen.AddExtension(X509Extensions.KeyUsage, critical: true, new KeyUsage(KeyUsage.DigitalSignature));
        certGen.AddExtension(X509Extensions.ExtendedKeyUsage, critical: true,
            new DerSequence(KeyPurposeID.AnyExtendedKeyUsage));
        certGen.AddExtension(X509Extensions.SubjectAlternativeName, critical: true,
            new GeneralNames(new GeneralName(GeneralName.Rfc822Name, "test@test.test")));

        X509Certificate cert = certGen.Generate(signatureFactory);

        // Check validity
        cert.CheckValidity();

        // Verify signature
        if (cert.IsSignatureValid(keyPair.Public))
        {
            Console.WriteLine("ML-DSA certificate verified");
            return 0;
        }

        return 1;
    }
}
```

## Ex 4. Creating a PKCS#10 Certification Request using ML-DSA

```
using Org.BouncyCastle.Asn1.X509;
using Org.BouncyCastle.Crypto.Generators;
using Org.BouncyCastle.Crypto.Operators;
using Org.BouncyCastle.Crypto.Parameters;
using Org.BouncyCastle.Pkcs;
using Org.BouncyCastle.Security;

namespace PqcAlmanac;

/**
* Create a basic PKCS#10 request using ML-DSA-44.
*/
public class MLDsaPkcs10Example
{
    public static int Main(string[] args)
    {
        var random = new SecureRandom();

        // Generate an ML-DSA-44 key pair.
        var kpg = new MLDsaKeyPairGenerator();
        kpg.Init(new MLDsaKeyGenerationParameters(random, MLDsaParameters.ml_dsa_44));
        var keyPair = kpg.GenerateKeyPair();

        // Create PKCS#10 request
        var signatureFactory = new Asn1SignatureFactory("ML-DSA-44", keyPair.Private);
        var subject = new X509Name("CN=ML-DSA Certification Request");
        var pkcs10 = new Pkcs10CertificationRequest(signatureFactory, subject, keyPair.Public, attributes: null);

        // Verify signature
        if (pkcs10.Verify(keyPair.Public))
        {
            Console.WriteLine("ML-DSA PKCS#10 request verified");
            return 0;
        }

        return 1;
    }
}
```