# Crypto

## Crypto

**Available as of Camel 2.3**
**PGP Available as of Camel 2.9**

The Crypto Data Format integrates the Java Cryptographic Extension into Camel, allowing simple and flexible encryption and decryption of messages using Camel's familiar marshall and unmarshal formatting mechanism. It assumes marshalling to mean encryption to cyphertext and unmarshalling to mean decryption back to the original plaintext. This data format implements only symmetric (shared-key) encryption and decyption.

### Options

| Name | Type | Default | Description |
|---|---|---|---|
| `algorithm` | `String` | `DES/CBC /PKCS5Padding` | The JCE algorithm name indicating the cryptographic algorithm that will be used. |
| `algorithmParameterSpec` | `java.security.spec. AlgorithmParameterSpec` | `null` | A JCE AlgorithmParameterSpec used to initialize the Cipher. |
| `bufferSize` | `Integer` | `4096` | the size of the buffer used in the signature process. |
| `cryptoProvider` | `String` | `null` | The name of the JCE Security Provider that should be used. |
| `initializationVector` | `byte[]` | `null` | A byte array containing the Initialization Vector that will be used to initialize the Cipher. |
| `inline` | `boolean` | `false` | Flag indicating that the configured IV should be inlined into the encrypted data stream. |
| `macAlgorithm` | `String` | `null` | The JCE algorithm name indicating the Message Authentication algorithm. |
| `shouldAppendHMAC` | `boolean` | `null` | Flag indicating that a Message Authentication Code should be calculated and appended to the encrypted data. |

### Basic Usage

At its most basic all that is required to encrypt/decrypt an exchange is a shared secret key. If one or more instances of the Crypto data format are configured with this key the format can be used to encrypt the payload in one route (or part of one) and decrypted in another. For example, using the Java DSL as follows:{snippet:id=basic|lang=java|url=camel/trunk/components/camel-crypto/src/test/java/org/apache/camel/converter/crypto/CryptoDataFormatTest.java}In Spring the dataformat is configured first and then used in routes

xml<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring"> <dataFormats> <crypto id="basic" algorithm="DES" keyRef="desKey" /> </dataFormats> ... <route> <from uri="direct:basic-encryption" /> <marshal ref="basic" /> <to uri="mock:encrypted" /> <unmarshal ref="basic" /> <to uri="mock:unencrypted" /> </route> </camelContext>

### Specifying the Encryption Algorithm

Changing the algorithm is a matter of supplying the JCE algorithm name. If you change the algorithm you will need to use a compatible key.{snippet:id=algorithm|lang=java|url=camel/trunk/components/camel-crypto/src/test/java/org/apache/camel/converter/crypto/CryptoDataFormatTest.java}A list of the available algorithms in Java 7 is available via the Java Cryptography Architecture Standard Algorithm Name Documentation.

### Specifying an Initialization Vector

Some crypto algorithms, particularly block algorithms, require configuration with an initial block of data known as an Initialization Vector. In the JCE this is passed as an AlgorithmParameterSpec when the Cipher is initialized. To use such a vector with the CryptoDataFormat you can configure it with a byte[] containing the required data e.g.{snippet:id=init-vector|lang=java|url=camel/trunk/components/camel-crypto/src/test/java/org/apache/camel/converter/crypto/CryptoDataFormatTest.java}or with spring, suppling a reference to a byte[]{snippet:id=init-vector|lang=xml|url=camel/trunk/components/camel-crypto/src/test/resources/org/apache/camel/component/crypto/SpringCryptoDataFormatTest.xml}The same vector is required in both the encryption and decryption phases. As it is not necessary to keep the IV a secret, the DataFormat allows for it to be inlined into the encrypted data and subsequently read out in the decryption phase to initialize the Cipher. To inline the IV set the /oinline flag.{snippet:id=inline-init-vector|lang=java|url=camel/trunk/components/camel-crypto/src/test/java/org/apache/camel/converter/crypto/CryptoDataFormatTest.java}or with spring.{snippet:id=inline|lang=xml|url=camel/trunk/components/camel-crypto/src/test/resources/org/apache/camel/component/crypto/SpringCryptoDataFormatTest.xml}For more information of the use of Initialization Vectors, consult

- http://en.wikipedia.org/wiki/Initialization_vector
- http://www.herongyang.com/Cryptography/
- http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation

### Hashed Message Authentication Codes (HMAC)

To avoid attacks against the encrypted data while it is in transit the CryptoDataFormat can also calculate a Message Authentication Code for the encrypted exchange contents based on a configurable MAC algorithm. The calculated HMAC is appended to the stream after encryption. It is separated from the stream in the decryption phase. The MAC is recalculated and verified against the transmitted version to insure nothing was tampered with in transit.For more information on Message Authentication Codes see http://en.wikipedia.org/wiki/HMAC{snippet:id=hmac|lang=java|url=camel/trunk/components/camel-crypto/src/test/java/org/apache/camel/converter/crypto/CryptoDataFormatTest.java}or with spring.{snippet:id=hmac|lang=xml|url=camel/trunk/components/camel-crypto/src/test/resources/org/apache/camel/component/crypto/SpringCryptoDataFormatTest.xml}By default the HMAC is calculated using the HmacSHA1 mac algorithm though this can be easily changed by supplying a different algorithm name. See here for how to check what algorithms are available through the configured security providers{snippet:id=hmac-algorithm|lang=java|url=camel/trunk/components/camel-crypto/src/test/java/org/apache/camel/converter/crypto/CryptoDataFormatTest.java}or with spring.{snippet:id=hmac-algorithm|lang=xml|url=camel/trunk/components/camel-crypto/src/test/resources/org/apache/camel/component/crypto/SpringCryptoDataFormatTest.xml}

## Supplying Keys Dynamically

When using a Recipient list or similar EIP the recipient of an exchange can vary dynamically. Using the same key across all recipients may neither be feasible or desirable. It would be useful to be able to specify keys dynamically on a per exchange basis. The exchange could then be dynamically enriched with the key of its target recipient before being processed by the data format. To facilitate this the DataFormat allow for keys to be supplied dynamically via the message headers below

- `CryptoDataFormat.KEY` `"CamelCryptoKey"`

{snippet:id=key-in-header|lang=java|url=camel/trunk/components/camel-crypto/src/test/java/org/apache/camel/converter/crypto/CryptoDataFormatTest.java}or with spring.{snippet:id=header-key|lang=xml|url=camel/trunk/components/camel-crypto/src/test/resources/org/apache/camel/component/crypto/SpringCryptoDataFormatTest.xml}

## PGP Message

The PGP Data Formater can create and decrypt/verify PGP Messages of the following PGP packet structure (entries in brackets are optional and ellipses indicate repetition, comma represents  sequential composition, and vertical bar separates alternatives):

Public Key Encrypted Session Key ..., Symmetrically Encrypted Data | Sym. Encrypted and Integrity Protected Data, (Compressed Data,) (One Pass Signature ...,) Literal Data, (Signature ...,)

**Since Camel 2.16.0** the Compressed Data packet is optional, before it was mandatory.

## PGPDataFormat Options

| Name | Type | Default | Description |
|---|---|---|---|
| `keyUserid` | `String` | `null` | The user ID of the key in the PGP keyring used during encryption. See also option `keyUserids`. Can also be only a part of a user ID. For example, if the user ID is "Test User <test@camel.com>" then you can use the part "Test User" or "<test@camel.com>" to address the user ID. |
| `keyUserids` | `List<String>` | `null` | **Since camel 2.12.2**: PGP allows to encrypt the symmetric key by several asymmetric public receiver keys. You can specify here the User IDs or parts of User IDs of several public keys contained in the PGP keyring. If you just have one User ID, then you can also use the option `keyUserid`. The User ID specified in `keyUserid` and the User IDs in `keyUserids` will be merged together and the corresponding public keys will be used for the encryption. |
| `password` | `String` | `null` | Password used when opening the private key (not used for encryption). |
| `keyFileName` | `String` | `null` | Filename of the keyring; must be accessible as a classpath resource (but you can specify a location in the file system by using the "file:" prefix). |
| `encryptionKeyRing` | `byte[]` | `null` | **Since camel 2.12.1**: encryption keyring; you can not set the keyFileName and encryptionKeyRing at the same time. |
| `signatureKeyUserid` | `String` | `null` | **Since Camel 2.11.0**; optional User ID of the key in the PGP keyring used for signing (during encryption) or signature verification (during decryption). During the signature verification process the specified User ID restricts the public keys from the public keyring which can be used for the verification. If no User ID is specified for the signature verficiation then any public key in the public keyring can be used for the verification. Can also be only a part of a user ID. For example, if the user ID is "Test User <test@camel.com>" then you can use the part "Test User" or "<test@camel.com>" to address the User ID. |
| `signatureKeyUserids` | `List<String>` | `null` | **Since Camel 2.12.3**: optional list of User IDs of the key in the PGP keyring used for signing (during encryption) or signature verification (during decryption). You can specify here the User IDs or parts of User IDs of several keys contained in the PGP keyring. If you just have one User ID, then you can also use the option `keyUserid`. The User ID specified in `keyUserid` and the User IDs in `keyUserids` will be merged together and the corresponding keys will be used for the signing or signature verification. If the specified User IDs reference several keys then for each key a signature is added to the PGP result during the encryption-signing process. In the decryption-verifying process the list of User IDs restricts the list of public keys which can be used for signature verification. If the list of User IDs is empty then any public key in the public keyring can be used for the signature verification. |
| `signaturePassword` | `String` | `null` | **Since Camel 2.11.0**: optional password used when opening the private key used for signing (during encryption). |

| | | | |
|---|---|---|---|
| `signatureKeyFileName` | `String` | `null` | **Since Camel 2.11.0**: optional filename of the keyring to use for signing (during encryption) or for signature verification (during decryption); must be accessible as a classpath resource (but you can specify a location in the file system by using the "file:" prefix). |
| `signatureKeyRing` | `byte[]` | `null` | **Since camel 2.12.1**: signature keyring; you can not set the signatureKeyFileName and signatureKeyRing at the same time. |
| `algorithm` | `int` | `SymmetricKeyAlgorithmTags.CAST5` | **Since camel 2.12.2**: symmetric key encryption algorithm; possible values are defined in `org.bouncycastle.bcpg.SymmetricKeyAlgorithmTags`; for example 2 (= TRIPLE DES), 3 (= CAST5), 4 (= BLOWFISH), 6 (= DES), 7 (= AES_128). Only relevant for encrypting. |
| `compressionAlgorithm` | `int` | `CompressionAlgorithmTags.ZIP` | **Since camel 2.12.2**: compression algorithm; possible values are defined in `org.bouncycastle.bcpg.CompressionAlgorithmTags`; for example 0 (= UNCOMPRESSED), 1 (= ZIP), 2 (= ZLIB), 3 (= BZIP2). Only relevant for encrypting. |
| `hashAlgorithm` | `int` | `HashAlgorithmTags.SHA1` | **Since camel 2.12.2**: signature hash algorithm; possible values are defined in `org.bouncycastle.bcpg.HashAlgorithmTags`; for example 2 (= SHA1), 8 (= SHA256), 9 (= SHA384), 10 (= SHA512), 11 (=SHA224). Only relevant for signing. |
| `armored` | `boolean` | `false` | This option will cause PGP to base64 encode the encrypted text, making it available for copy/paste, etc. |
| `integrity` | `boolean` | `true` | Adds an integrity check/sign into the encryption file. |
| `passphraseAccessor` | `PGPPassphraseAccessor` | `null` | **Since Camel 2.12.2**: provides passphrases corresponding to user Ids. If no passpharase can be found from the option `password` or `signaturePassword` and from the headers `CamelPGPDataFormatKeyPassword` or `CamelPGPDataFormatSignatureKeyPassword` then the passphrase is fetched from the passphrase accessor. You provide a bean which implements the interface PGPPassphraseAccessor. A default implementation is given by DefaultPGPPassphraseAccessor. The passphrase accessor is especially useful in the decrypt case; see chapter 'PGP Decrypting/Verifying of Messages Encrypted/Signed by Different Private/Public Keys' below. |
| `signatureVerificationOption` | `String` | `"optional"` | **Since Camel 2.13.0**: controls the behavior for verifying the signature during unmarshaling. There are three values possible:<br>• `"optional"`: The PGP message may or may not contain signatures; if it does contain signatures, then a signature verification is executed. Use the constant PGPKeyAccessDataFormat.SIGNATURE_VERIFICATION_OPTION_OPTIONAL.<br>• `"required"`: The PGP message must contain at least one signature; if this is not the case an exception (PGPException) is thrown. A signature verification is executed. Use the constant PGPKeyAccessDataFormat.SIGNATURE_VERIFICATION_OPTION_REQUIRED.<br>• `"ignore"`: Contained signatures in the PGP message are ignored; no signature verification is executed. Use the constant PGPKeyAccessDataFormat.SIGNATURE_VERIFICATION_OPTION_IGNORE.<br>• `"no_signature_allowed"`: The PGP message must not contain a signature; otherwise an exception (PGPException) is thrown. Use the constant PGPKeyAccessDataFormat.SIGNATURE_VERIFICATION_OPTION_NO_SIGNATURE_ALLOWED. |
| `FileName` | `String` | `"_CONSOLE"` | **Since camel 2.15.0**: Sets the file name for the literal data packet. Can be overwritten by the header {@link Exchange#FILE_NAME}.<br>"`_CONSOLE`" indicates that the message is considered to be "for your eyes only". This advises that the message data is unusually sensitive, and the receiving program should process it more carefully, perhaps avoiding storing the received data to disk, for example. Only used for marshaling. |
| `withCompressedDataPacket` | `boolean` | `true` | **Since Camel 2.16.0**: Indicator whether the PGP Message shall be created with or without a Compressed Data packet. If the value is set to false, then no Compressed Data packet is added and the compressionAlgorithm value is ignored. Only used for marshaling. |

## PGPDataFormat Message Headers

You can override the PGPDataFormat options by applying below headers into message dynamically.

| Name | Type | Description |
|---|---|---|
| `CamelPGPDataFormatKeyFileName` | `String` | **Since Camel 2.11.0**; filename of the keyring; will override existing setting directly on the PGPDataFormat. |
| `CamelPGPDataFormatEncryptionKeyRing` | `byte[]` | **Since Camel 2.12.1**; the encryption keyring; will override existing setting directly on the PGPDataFormat. |
| `CamelPGPDataFormatKeyUserid` | `String` | **Since Camel 2.11.0**; the User ID of the key in the PGP keyring; will override existing setting directly on the PGPDataFormat. |
| `CamelPGPDataFormatKeyUserids` | `List<String>` | **Since camel 2.12.2**: the User IDs of the key in the PGP keyring; will override existing setting directly on the PGPDataFormat. |
| `CamelPGPDataFormatKeyPassword` | `String` | **Since Camel 2.11.0**; password used when opening the private key; will override existing setting directly on the PGPDataFormat. |

| | | |
|---|---|---|
| `CamelPGPDataFormatSignatureKeyFileName` | `String` | **Since Camel 2.11.0**; filename of the signature keyring; will override existing setting directly on the PGPDataFormat. |
| `CamelPGPDataFormatSignatureKeyRing` | `byte[]` | **Since Camel 2.12.1**; the signature keyring; will override existing setting directly on the PGPDataFormat. |
| `CamelPGPDataFormatSignatureKeyUserid` | `String` | **Since Camel 2.11.0**; the User ID of the signature key in the PGP keyring; will override existing setting directly on the PGPDataFormat. |
| `CamelPGPDataFormatSignatureKeyUserids` | `List<String>` | **Since Camel 2.12.3**; the User IDs of the signature keys in the PGP keyring; will override existing setting directly on the PGPDataFormat. |
| `CamelPGPDataFormatSignatureKeyPassword` | `String` | **Since Camel 2.11.0**; password used when opening the signature private key; will override existing setting directly on the PGPDataFormat. |
| `CamelPGPDataFormatEncryptionAlgorithm` | `int` | **Since Camel 2.12.2**; symmetric key encryption algorithm; will override existing setting directly on the PGPDataFormat. |
| `CamelPGPDataFormatSignatureHashAlgorithm` | `int` | **Since Camel 2.12.2**; signature hash algorithm; will override existing setting directly on the PGPDataFormat. |
| `CamelPGPDataFormatCompressionAlgorithm` | `int` | **Since Camel 2.12.2**; compression algorithm; will override existing setting directly on the PGPDataFormat. |
| `CamelPGPDataFormatNumberOfEncryptionKeys` | `Integer` | **Since Camel 2.12.3;** number of public keys used for encrypting the symmectric key, set by PGPDataFormat during encryptiion process |
| `CamelPGPDataFormatNumberOfSigningKeys` | `Integer` | **Since Camel 2.12.3;** number of private keys used for creating signatures, set by PGPDataFormat during signing process |

## Encrypting with PGPDataFormat

The following sample uses the popular PGP format for encrypting/decrypting files using the Bouncy Castle Java libraries:{snippet:id=pgp-format|lang=java|url=camel/trunk/components/camel-crypto/src/test/java/org/apache/camel/converter/crypto/PGPDataFormatTest.java}The following sample performs signing + encryption, and then signature verification + decryption. It uses the same keyring for both signing and encryption, but you can obviously use different keys:{snippet:id=pgp-format-signature|lang=java|url=camel/trunk/components/camel-crypto/src/test/java/org/apache/camel/converter/crypto/PGPDataFormatTest.java}Or using Spring:{snippet:id=pgp-xml-basic|lang=xml|url=camel/trunk/components/camel-crypto/src/test/resources/org/apache/camel/component/crypto/SpringPGPDataFormatTest.xml}

### To work with the previous example you need the following

- A public keyring file which contains the public keys used to encrypt the data
- A private keyring file which contains the keys used to decrypt the data
- The keyring password

### Managing your keyring

To manage the keyring, I use the command line tools, I find this to be the simplest approach in managing the keys. There are also Java libraries available from http://www.bouncycastle.org/java.html if you would prefer to do it that way.

1. Install the command line utilities on linux

   apt-get install gnupg
2. Create your keyring, entering a secure password

   gpg --gen-key
3. If you need to import someone elses public key so that you can encrypt a file for them.

   gpg --import <filename.key
4. The following files should now exist and can be used to run the example

   ls -l ~/.gnupg/pubring.gpg ~/.gnupg/secring.gpg

## PGP Decrypting/Verifying of Messages Encrypted/Signed by Different Private/Public Keys

Since **Camel 2.12.2**.

A PGP Data Formater can decrypt/verify messages which have been encrypted by different public keys or signed by different private keys. Just, provide the corresponding private keys in the secret keyring, the corresponding public keys in the public keyring, and the passphrases in the passphrase accessor.

javaMap<String, String> userId2Passphrase = new HashMap<String, String>(2); // add passphrases of several private keys whose corresponding public keys have been used to encrypt the messages userId2Passphrase.put("UserIdOfKey1","passphrase1"); // you must specify the exact User ID! userId2Passphrase.put("UserIdOfKey2","passphrase2"); PGPPassphraseAccessor passphraseAccessor = new PGPPassphraseAccessorDefault (userId2Passphrase); PGPDataFormat pgpVerifyAndDecrypt = new PGPDataFormat(); pgpVerifyAndDecrypt.setPassphraseAccessor (passphraseAccessor); // the method getSecKeyRing() provides the secret keyring as byte array containing the private keys pgpVerifyAndDecrypt. setEncryptionKeyRing(getSecKeyRing()); // alternatively you can use setKeyFileName(keyfileName) // the method getPublicKeyRing() provides the public keyring as byte array containing the public keys pgpVerifyAndDecrypt.setSignatureKeyRing((getPublicKeyRing()); // alternatively you can use

setSignatureKeyFileName(signatgureKeyfileName) // it is not necessary to specify the encryption or signer User Id from("direct:start") ... .unmarshal (pgpVerifyAndDecrypt) // can decrypt/verify messages encrypted/signed by different private/public keys ...

- The functionality is especially useful to support the key exchange. If you want to exchange the private key for decrypting you can accept for a period of time messages which are either encrypted with the old or new corresponding public key. Or if the sender wants to exchange his signer private key, you can accept for a period of time, the old or new signer key.
- Technical background: The PGP encrypted data contains a Key ID of the public key which was used to encrypt the data. This Key ID can be used to locate the private key in the secret keyring to decrypt the data. The same mechanism is also used to locate the public key for verifying a signature. Therefore you no longer must specify User IDs for the unmarshaling.

## Restricting the Signer Identities during PGP Signature Verification

Since **Camel 2.12.3.**

If you verify a signature you not only want to verify the correctness of the signature but you also want check that the signature comes from a certain identity or a specific set of identities. Therefore it is possible to restrict the number of public keys from the public keyring which can be used for the verification of a signature.

javaSignature User IDs// specify the User IDs of the expected signer identities List<String> expectedSigUserIds = new ArrayList<String>(); expectedSigUserIds.add("Trusted company1"); expectedSigUserIds.add("Trusted company2"); PGPDataFormat pgpVerifyWithSpecificKeysAndDecrypt = new PGPDataFormat(); pgpVerifyWithSpecificKeysAndDecrypt.setPassword("my password"); // for decrypting with private key pgpVerifyWithSpecificKeysAndDecrypt.setKeyFileName(keyfileName); pgpVerifyWithSpecificKeysAndDecrypt.setSignatureKeyFileName (signatgureKeyfileName); pgpVerifyWithSpecificKeysAndDecrypt.setSignatureKeyUserids(expectedSigUserIds); // if you have only one signer identity then you can also use setSignatureKeyUserid("expected Signer") from("direct:start") ... .unmarshal(pgpVerifyWithSpecificKeysAndDecrypt) ...

- If the PGP content has several signatures the verification is successful as soon as one signature can be verified.
- If you do not want to restrict the signer identities for verification then do not specify the signature key User IDs. In this case all public keys in the public keyring are taken into account.

## Several Signatures in One PGP Data Format

Since **Camel 2.12.3.**

The PGP specification allows that one PGP data format can contain several signatures from different keys. Since Camel 2.13.3 it is possible to create such kind of PGP content via specifying signature User IDs which relate to several private keys in the secret keyring.

javaSeveral Signatures PGPDataFormat pgpSignAndEncryptSeveralSignerKeys = new PGPDataFormat(); pgpSignAndEncryptSeveralSignerKeys. setKeyUserid(keyUserid); // for encrypting, you can also use setKeyUserids if you want to encrypt with several keys pgpSignAndEncryptSeveralSignerKeys.setKeyFileName(keyfileName); pgpSignAndEncryptSeveralSignerKeys.setSignatureKeyFileName (signatgureKeyfileName); pgpSignAndEncryptSeveralSignerKeys.setSignaturePassword("sdude"); // here we assume that all private keys have the same password, if this is not the case then you can use setPassphraseAccessor List<String> signerUserIds = new ArrayList<String>(); signerUserIds.add ("company old key"); signerUserIds.add("company new key"); pgpSignAndEncryptSeveralSignerKeys.setSignatureKeyUserids(signerUserIds); from ("direct:start") ... .marshal(pgpSignAndEncryptSeveralSignerKeys) ...

## Support of Sub-Keys and Key Flags in PGP Data Format Marshaler

Since **Camel 2.12.3.**
An OpenPGP V4 key can have a primary key and sub-keys. The usage of the keys is indicated by the so called Key Flags. For example, you can have a primary key with two sub-keys; the primary key shall only be used for certifying other keys (Key Flag 0x01), the first sub-key shall only be used for signing (Key Flag 0x02), and the second sub-key shall only be used for encryption (Key Flag 0x04 or 0x08). The PGP Data Format marshaler takes into account these Key Flags of the primary key and sub-keys in order to determine the right key for signing and encryption. This is necessary because the primary key and its sub-keys have the same User IDs.

## Support of Custom Key Accessors

Since **Camel 2.13.0.**
You can implement custom key accessors for encryption/signing. The above PGPDataFormat class selects in a certain predefined way the keys which should be used for signing/encryption or verifying/decryption. If you have special requirements how your keys should be selected you should use the PGPKeyAccessDataFormat class instead and implement the interfaces PGPPublicKeyAccessor and PGPSecretKeyAccessor as beans. There are default implementations DefaultPGPPublicKeyAccessor and DefaultPGPSecretKeyAccessor which cache the keys, so that not every time the keyring is parsed when the processor is called.

PGPKeyAccessDataFormat has the same options as PGPDataFormat except password, keyFileName, encryptionKeyRing, signaturePassword, signatureKeyFileName, and signatureKeyRing.

## Dependencies

To use the Crypto dataformat in your camel routes you need to add the following dependency to your pom.

xml<dependency> <groupId>org.apache.camel</groupId> <artifactId>camel-crypto</artifactId> <version>x.x.x</version> <!-- use the same version as your Camel core version --> </dependency>

## See Also

- Data Format

- Crypto (Digital Signatures)
- http://www.bouncycastle.org/java.html