Crypto (Digital Signatures)

Crypto component for Digital Signatures

Available as of Camel 2.3

With Camel cryptographic endpoints and Java's Cryptographic extension it is easy to create Digital Signatures for Exchanges. Camel provides a pair of flexible endpoints which get used in concert to create a signature for an exchange in one part of the exchange's workflow and then verify the signature in a later part of the workflow.

Maven users will need to add the following dependency to their `pom.xml` for this component:

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>

Introduction

Digital signatures make use of Asymmetric Cryptographic techniques to sign messages. From a (very) high level, the algorithms use pairs of complimentary keys with the special property that data encrypted with one key can only be decrypted with the other. One, the private key, is closely guarded and used to 'sign' the message while the other, public key, is shared around to anyone interested in verifying the signed messages. Messages are signed by using the private key to encrypting a digest of the message. This encrypted digest is transmitted along with the message. On the other side the verifier recalculates the message digest and uses the public key to decrypt the the digest in the signature. If both digests match the verifier knows only the holder of the private key could have created the signature.

Camel uses the Signature service from the Java Cryptographic Extension to do all the heavy cryptographic lifting required to create exchange signatures. The following are some excellent resources for explaining the mechanics of Cryptography, Message digests and Digital Signatures and how to leverage them with the JCE.

- Bruce Schneier's Applied Cryptography
- Beginning Cryptography with Java by David Hook
- The ever insightful Wikipedia Digital_signatures

URI format

As mentioned Camel provides a pair of crypto endpoints to create and verify signatures

crypto:sign:name[?options] crypto:verify:name[?options]

- `crypto:sign` creates the signature and stores it in the Header keyed by the constant `org.apache.camel.component.crypto.DigitalSignatureConstants.SIGNATURE`, i.e. `"CamelDigitalSignature"`.
- `crypto:verify` will read in the contents of this header and do the verification calculation.

In order to correctly function, the sign and verify process needs a pair of keys to be shared, signing requiring a `PrivateKey` and verifying a `PublicKey` (or a `Certificate` containing one). Using the JCE it is very simple to generate these key pairs but it is usually most secure to use a KeyStore to house and share your keys. The DSL is very flexible about how keys are supplied and provides a number of mechanisms.

Note a `crypto:sign` endpoint is typically defined in one route and the complimentary `crypto:verify` in another, though for simplicity in the examples they appear one after the other. It goes without saying that both signing and verifying should be configured identically.

Options

confluenceTableSmall

Name	Type	Default	Description
`algorithm`	`String`	`SHA1WithDSA`	The name of the JCE Signature algorithm that will be used.
`alias`	`String`	`null`	An alias name that will be used to select a key from the keystore.
`bufferSize`	`Integer`	`2048`	the size of the buffer used in the signature process.
`certificate`	`Certificate`	`null`	A Certificate used to verify the signature of the exchange's payload. Either this or a Public Key is required.
`keystore`	`KeyStore`	`null`	A reference to a JCE Keystore that stores keys and certificates used to sign and verify.
keyStoreParameters **Camel 2.14.1**	KeyStoreParameters	null	A reference to a Camel KeyStoreParameters Object which wraps a Java KeyStore Object
`provider`	`String`	`null`	The name of the JCE Security Provider that should be used.
`privateKey`	`PrivateKey`	`null`	The private key used to sign the exchange's payload.
`publicKey`	`PublicKey`	`null`	The public key used to verify the signature of the exchange's payload.
`secureRandom`	`secureRandom`	`null`	A reference to a `SecureRandom` object that will be used to initialize the Signature service.
`password`	`char[]`	`null`	The password to access the private key from the keystore

`clearHeaders`	`String`	`true`	Remove camel crypto headers from Message after a verify operation (value can be `"true"`/`"false"`).

Using

1) Raw keys

The most basic way to way to sign and verify an exchange is with a KeyPair as follows.{snippet:id=basic|lang=java|url=camel/trunk/components/camel-crypto/src/test/java/org/apache/camel/component/crypto/SignatureTests.java}The same can be achieved with the Spring XML Extensions using references to keys{snippet:id=basic|lang=xml|url=camel/trunk/components/camel-crypto/src/test/resources/org/apache/camel/component/crypto/SpringSignatureTests.xml}

2) KeyStores and Aliases.

The JCE provides a very versatile keystore concept for housing pairs of private keys and certificates, keeping them encrypted and password protected. They can be retrieved by applying an alias to the retrieval APIs. There are a number of ways to get keys and Certificates into a keystore, most often this is done with the external 'keytool' application. This is a good example of using keytool to create a KeyStore with a self signed Cert and Private key.

The examples use a Keystore with a key and cert aliased by 'bob'. The password for the keystore and the key is 'letmein'

The following shows how to use a Keystore via the Fluent builders, it also shows how to load and initialize the keystore.{snippet:id=keystore|lang=java|url=camel/trunk/components/camel-crypto/src/test/java/org/apache/camel/component/crypto/SignatureTests.java}Again in Spring a ref is used to lookup an actual keystore instance.{snippet:id=keystore|lang=xml|url=camel/trunk/components/camel-crypto/src/test/resources/org/apache/camel/component/crypto/SpringSignatureTests.xml}

3) Changing JCE Provider and Algorithm

Changing the Signature algorithm or the Security provider is a simple matter of specifying their names. You will need to also use Keys that are compatible with the algorithm you choose.{snippet:id=algorithm|lang=java|url=camel/trunk/components/camel-crypto/src/test/java/org/apache/camel/component/crypto/SignatureTests.java}{snippet:id=provider|lang=java|url=camel/trunk/components/camel-crypto/src/test/java/org/apache/camel/component/crypto/SignatureTests.java}or{snippet:id=algorithm|lang=xml|url=camel/trunk/components/camel-crypto/src/test/resources/org/apache/camel/component/crypto/SpringSignatureTests.xml}{snippet:id=provider|lang=xml|url=camel/trunk/components/camel-crypto/src/test/resources/org/apache/camel/component/crypto/SpringSignatureTests.xml}

4) Changing the Signature Message Header

It may be desirable to change the message header used to store the signature. A different header name can be specified in the route definition as follows{snippet:id=signature-header|lang=java|url=camel/trunk/components/camel-crypto/src/test/java/org/apache/camel/component/crypto/SignatureTests.java}or{snippet:id=signature-header|lang=xml|url=camel/trunk/components/camel-crypto/src/test/resources/org/apache/camel/component/crypto/SpringSignatureTests.xml}

5) Changing the buffersize

In case you need to update the size of the buffer...{snippet:id=buffersize|lang=java|url=camel/trunk/components/camel-crypto/src/test/java/org/apache/camel/component/crypto/SignatureTests.java}or{snippet:id=buffersize|lang=xml|url=camel/trunk/components/camel-crypto/src/test/resources/org/apache/camel/component/crypto/SpringSignatureTests.xml}

6) 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 be neither feasible nor desirable. It would be useful to be able to specify signature keys dynamically on a per-exchange basis. The exchange could then be dynamically enriched with the key of its target recipient prior to signing. To facilitate this the signature mechanisms allow for keys to be supplied dynamically via the message headers below

- `Exchange.SIGNATURE_PRIVATE_KEY`, `"CamelSignaturePrivateKey"`
- `Exchange.SIGNATURE_PUBLIC_KEY_OR_CERT`, `"CamelSignaturePublicKeyOrCert"`

{snippet:id=headerkey|lang=java|url=camel/trunk/components/camel-crypto/src/test/java/org/apache/camel/component/crypto/SignatureTests.java}or{snippet:id=headerkey|lang=xml|url=camel/trunk/components/camel-crypto/src/test/resources/org/apache/camel/component/crypto/SpringSignatureTests.xml}Even better would be to dynamically supply a keystore alias. Again the alias can be supplied in a message header

- `Exchange.KEYSTORE_ALIAS`, `"CamelSignatureKeyStoreAlias"`

{snippet:id=alias|lang=java|url=camel/trunk/components/camel-crypto/src/test/java/org/apache/camel/component/crypto/SignatureTests.java}or{snippet:id=alias|lang=xml|url=camel/trunk/components/camel-crypto/src/test/resources/org/apache/camel/component/crypto/SpringSignatureTests.xml}The header would be set as follows

Exchange unsigned = getMandatoryEndpoint("direct:alias-sign").createExchange(); unsigned.getIn().setBody(payload); unsigned.getIn().setHeader(DigitalSignatureConstants.KEYSTORE_ALIAS, "bob"); unsigned.getIn().setHeader(DigitalSignatureConstants.KEYSTORE_PASSWORD, "letmein".toCharArray()); template.send("direct:alias-sign", unsigned); Exchange signed = getMandatoryEndpoint("direct:alias-sign").createExchange(); signed.getIn().copyFrom(unsigned.getOut()); signed.getIn().setHeader(KEYSTORE_ALIAS, "bob"); template.send("direct:alias-verify", signed);

Endpoint See Also

- Crypto Crypto is also available as a Data Format