14

Securing Camel

This chapter covers

- Securing your Camel configuration
- Web service security
- Transport security
- Encryption and decryption
- Signing messages
- Authentication and authorization

Security in enterprise applications seems to become more and more important every year. As mobile and web access endpoints are the preferred method of access for customers, applications are becoming more open to the greater internet and consequently more open to attack. Unauthorized access to these exposed endpoints can become a costly thing to deal with. For example, having private customer data leaked on the internet has plagued retailers in recent years. These events definitely have an impact on the current and future bottom line of a company's finances.

With that said, it's important to note that Camel is by default *not* secured! There's a good reason for this: application security has many angles, and not all may be applicable to every use case. For instance, you probably don't need to encrypt your payload if the communication link is within your company's VPN. But authentication and authorization may be needed. Camel can help you implement as much or as little security as you require, with relative ease. We say *relative* because security configuration can become quite complex just by its nature.

This chapter covers the main areas of security in Camel. We'll start by covering how to secure sensitive information in your configuration. Next we'll cover security in web services. Then we'll show how to sign and encrypt messages within Camel. We'll then cover transport security, which

is all about securing the link between Camel and the remote service using Transport Layer Security (TLS). Finally, Camel also provides authentication and authorization services for controlling access to your routes.

Let's start by looking at how to secure your configuration.

14.1 Securing your configuration

As you saw in chapter 2, externalizing your configuration and referencing with property placeholders are great ways to ease the transition from testing to production. Often you need to store such things as usernames, access keys, passwords, or other sensitive data to access remote services in your configuration. If there's a chance that someone may view this information, having everything in plain text would give that person access to the remote resource. To prevent that from happening, the camel-jasypt component allows you to encrypt your configuration values and then decrypt them via the property placeholder mechanism at runtime.

Let's first look at using camel-jasypt to encrypt your sensitive data.

14.1.1 Encrypting configuration

Encrypting your configuration is a task that typically happens outside your runtime route. The next section covers the runtime scenario (decrypting configuration). The camel-jasypt component includes a command-line utility to assist with this encryption, although, in theory, any utility that uses the same Java Cryptography Architecture (JCA) algorithm would do. The camel-jasypt utility is included with the camel-jasypt JAR in the lib folder of the Apache Camel distribution. For example, you can get help with this utility as follows:

```
[janstey@bender]$ cd apache-camel-2.20.1/
[janstey@bender]$ wget http://repo1.maven.org/maven2/org/jasypt/jasypt/1
[janstey@bender]$ java -cp jasypt-1.9.2.jar:lib/camel-jasypt-2.20.1.jar
Apache Camel Jasypt takes the following options

  -h or -help = Displays the help screen
  -c or -command <command> = Command either encrypt or decrypt
  -p or -password <password> = Password to use
```

```
    -i or -input <input> = Text to encrypt or decrypt
    -a or -algorithm <algorithm> = Optional algorithm to use
```

As you can see, the utility supports both encryption and decryption for convenience. To encrypt the secret text `secret`, you'd use the following options:

```
[janstey@bender]$ java -cp jasypt-1.9.2.jar:lib/camel-jasypt-2.20.1.jar
Encrypted text: q+XT/4rR94ghCbNp5coaxg==
```

Pay special attention to the `-p` option; we used an encryption password of `supersecret`, which you'll need to use later when decrypting configuration. The output from the utility is the encrypted value of the `secret` text. You can just as easily go back to the original value:

```
[janstey@bender]$ java -cp jasypt-1.9.2.jar:lib/camel-jasypt-2.20.1.jar
Decrypted text: secret
```

In both cases, you didn't specify anything for the algorithm (`-a`) option, so the default algorithm of `PBEWithMD5AndDES` will be used. This is a standard algorithm name as defined in the JCA Standard Algorithm Name Documentation (http://docs.oracle.com/javase/8/docs/technotes/guides/security/StandardNames.html). If you decide to use another algorithm here, make sure to use the same when you're decrypting configuration within your route. Also, not all security providers provide the same set of algorithms, so before configuring something different, ensure that all systems in your deployment can support the algorithm.

Now that you have your password encrypted, let's look at how to use that within your Camel route.

14.1.2 Decrypting configuration

To use the camel-jasypt component within your Camel project, you need to add a dependency to it in your Maven POM:

```
<dependency>
  <groupId>org.apache.camel</groupId>
```

```
  <artifactId>camel-jasypt</artifactId>
  <version>2.20.1</version>
</dependency>
```

You then need to configure the Camel Properties component to use the `PropertyParser` provided by camel-jasypt. In XML DSL, you reference it with the `propertiesParserRef` attribute on `<propertyPlaceholder>` within the `camelContext`, as follows:

```
<bean id="jasypt"
       class="org.apache.camel.component.jasypt.JasyptPropertiesParser">
    <property name="password" value="supersecret"/>
</bean>

<camelContext xmlns="http://camel.apache.org/schema/spring">
    <propertyPlaceholder id="properties"
                         location="classpath:rider-test.properties"
                         propertiesParserRef="jasypt"/>

...
</camelContext>
```

In the rider-test.properties file, you can then define the encrypted properties using the special `ENC()` notation:

```
ftp.password=ENC(q+XT/4rR94ghCbNp5coaxg==)
```

The text inside the parentheses is the secret text that you encrypted in the previous section. You can now use the encrypted property directly in the endpoint URI as the FTP server password, as shown in bold in this route:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  ...
  <route>
    <from uri="file:target/inbox"/>
    <to uri="ftp://rider:{{ftp.password}}@localhost:21000/target/outbox"
  </route>
</camelContext>
```

You can do all of this from the Java DSL as well. The following listing shows how.

Listing 14.1 Referencing encrypting properties from the Java DSL

```
public class SecuringConfigTest extends CamelTestSupport {
    @EndpointInject(uri = "file:target/inbox")
    private ProducerTemplate inbox;
    private FtpServerBean ftp = new FtpServerBean();

    @Override
    protected CamelContext createCamelContext() throws Exception {
        CamelContext context = super.createCamelContext();

    JasyptPropertiesParser jasypt = new JasyptPropertiesParser();    (1)
```

(1) Creates the jasypt properties parser

```
    jasypt.setPassword("supersecret");    (2)
```

(2) Sets the encryption password

```
        PropertiesComponent prop =
            context.getComponent("properties", PropertiesComponent.class
        prop.setLocation("classpath:rider-test.properties");
    prop.setPropertiesParser(jasypt);    (3)
```

(3) Uses the jasypt properties parser so you can decrypt values

```
        return context;
    }

    ...
```

```
    @Override
    protected RouteBuilder createRouteBuilder() throws Exception {
        return new RouteBuilder() {
            @Override
            public void configure() throws Exception {
                from("file:target/inbox")
          .to("ftp://rider:{{ftp.password}}@localhost:21000"
```

References the encrypted property with an endpoint URI

```
                    + "/target/outbox");
            }
        };
    }
}
```

After creating `CamelContext`, you create the jasypt `PropertyParser` ❶ and set it on `PropertiesComponent` ❸. You also need to use the encryption password ❷ as you did in the XML DSL example before.

You can try this example using the following Maven goals from the chapter14/configuration directory:

```
mvn test -Dtest=SecuringConfigTest
mvn test -Dtest=SpringSecuringConfigTest
```

Because you have your security hat on right now, you may have been thinking that this plain-text encryption password is a security hole. You'd be right, of course! After all, what's the point of encrypting your remote service password if you then leave the encryption password in plain view?

Externalizing the encryption password

How do you deal with a plain-text encryption password? The solution is to not define it in your application but to refer to it from an OS environment variable or JVM system property at runtime. For example, to use an environment variable `CAMEL_ENCRYPTION_PASSWORD` instead of the `supersecret` password in Listing 14.1, you could do as follows:

```
jasypt.setPassword("sysenv:CAMEL_ENCRYPTION_PASSWORD");
```

The `sysenv` prefix directs camel-jasypt to use the value in the `CAMEL_ENCRYPTION_PASSWORD` environment variable. Now, before starting your Camel application, you'd set an additional environment variable:

```
export CAMEL_ENCRYPTION_PASSWORD=supersecret
```

You can also refer to JVM system properties in the same manner, but instead you need to use the `sys` prefix.

Now that you've secured your configuration, let's look at how to secure web services.

14.2 Web service security

Web services are an extremely useful integration technology for distributed applications. They are often associated with service-oriented architecture (SOA), in which each service is defined as a web service.

You can think of a web service as an API on the network. The API itself is defined using the Web Services Description Language (WSDL), specifying the operations you can call on a web service and the input and output types, among other things. Messages are typically XML, formatted to comply with the SOAP schema. In addition, these messages are typically sent over HTTP. Web services allow you to write Java code and make that Java code callable over the internet, which is pretty neat!

For accessing and publishing web services, Camel uses Apache CXF (http://cxf.apache.org). CXF is a popular web services framework that supports many web services standards. We covered Camel's use of CXF in chapter 10, so you can refer to that chapter for more general usage. Here we focus on CXF's support of the Web Services Security (WS-Security) standard. The WS-Security support in CXF allows you to do the following:

- Use authentication tokens
- Encrypt messages
- Sign messages
- Timestamp messages

To show these concepts in action, let's go back to Rider Auto Parts, which needs a new piece of functionality implemented. In chapter 2, you saw that customers could place orders in two ways:

- Uploading the order file to an FTP server
- Submitting the order from the Rider Auto Parts web store via HTTP

What we didn't say then was that this HTTP link to the back-end order processing systems needs to be a web service. Of course, you wouldn't want anonymous users to be able to submit orders, so you're going to use WS-Security to help you out.

14.2.1 Authentication in web services

Clients of the new Rider Auto Parts web service have a simple way to submit orders. The service endpoint in question looks like this:

```
import javax.jws.WebService;

@WebService
public interface OrderEndpoint {
    OrderResult order(Order order);
}
```

Clients can call the order method with an `Order` object as the only argument. The `Order` object contains a part name, amount to order, and the customer name. An `OrderResult` is returned and can tell the user whether the order succeeded or failed. The big missing piece here is that it's by default unsecured! Anyone who knows the endpoint address can submit an order—hardly ideal for Rider Auto Parts or the customer who will be charged for the parts. The Rider Auto Parts developer team clearly needs to add authentication to this service.

With authentication, a user proves who they are to a system, typically using a username and password. In the web-services world, this is encapsulated in a `<UsernameToken>` element within the SOAP message. The receiver of a SOAP message with a `UsernameToken` can check things such as whether a username exists, whether a password is valid, or if the timestamp on the token is still valid. Consider this example `UsernameToken`:

```
<wsse:UsernameToken
  wsu:Id="UsernameToken-db49b9e2-1051-4559-bd48-39877be634ad">
 <wsse:Username>jon</wsse:Username>
 <wsse:Password
  Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username
        -token-profile-1.0#PasswordDigest">ZbNxe8WcmHekEcR6pbQLaVzW6zk=
</wsse:Password>
 <wsse:Nonce
  EncodingType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
                soap-message-security-1.0#Base64Binary">Zi8UQpTp28/OeSNq
                9eTQ==
</wsse:Nonce>
 <wsu:Created>2017-10-25T18:06:46.052Z</wsu:Created>
</wsse:UsernameToken>
```

That's not something you want to write or interpret by hand. Fortunately, CXF provides interceptors that can be used on the client and server side to process this information before the underlying service endpoint is called.

Server-side WS-Security processing

First, let's look at how the server-side configuration will change when security is added. The route used to implement the web service won't change at all:

```
 <route>
   <from uri="cxf:bean:orderEndpoint"/>
   <to uri="seda:incomingOrders"/>
   <transform>
     <method beanType="camelinaction.order.OrderResultBean"
             method="orderOK"/>
   </transform>
 </route>
```

This simple route sends an order to an `incomingOrders` queue and then returns an `OK` result back to the client. At this point, the user is assumed authenticated. You need to look at the CXF configuration in the following listing to see how this authentication was set up.

Listing 14.2 Adding `UsernameToken` authentication to a CXF web service

```
<bean id="loggingOutInterceptor"
      class="org.apache.cxf.interceptor.LoggingOutInterceptor"/>

<bean id="loggingInInterceptor"
      class="org.apache.cxf.interceptor.LoggingInInterceptor"/>

<bean id="wss4jInInterceptor"
    class="org.apache.cxf.ws.security.wss4j.WSS4JInInterceptor">    (1)
```

(1) Creates a WSS4JInInterceptor to process the WS-Security SOAP header

```
  <constructor-arg>
    <map>
   <entry key="action" value="UsernameToken Timestamp"/>    (2)
```

(2) Sets the actions to use when processing the UsernameToken

```
      <entry key="passwordCallbackClass"
       value="camelinaction.wssecurity.ServerPasswordCallback"/>    (3)
```

(3) Specifies the callback to use when checking the username and password

```
    </map>
  </constructor-arg>
</bean>

<cxf:cxfEndpoint id="orderEndpoint"
                 address="http://localhost:9000/order"
                 serviceClass="camelinaction.order.OrderEndpoint">

  <cxf:inInterceptors>
    <ref bean="loggingInInterceptor"/>
  <ref bean="wss4jInInterceptor"/>
```

Adds the WSS4JInInterceptor to the cxfEndpoint of the OrderEndpoint

```
    </cxf:inInterceptors>

    <cxf:outInterceptors>
      <ref bean="loggingOutInterceptor"/>
    </cxf:outInterceptors>
  </cxf:cxfEndpoint>
```

Your `cxfEndpoint` bean again doesn't change much from the unsecured case. The security magic happens in CXF's `org.apache.cxf.ws.security.wss4j.WSS4JInInterceptor` ❶. `WSS4J` in the class name refers to the Apache WSS4J project, which provides the implementation of many of the security concepts in CXF. The `action` entry ❷ specifies the tasks that will be executed using the interceptor. In our case, this will be to process a `UsernameToken` and a `Timestamp`. Take note of these actions, as the client code will also need to provide them. The `passwordCallbackClass` ❸ entry specifies a class that'll be used to check the username and provide a password. In our case, you'll be using a hardcoded username/password combination to demonstrate the flow. Such a callback handler class is shown in the following listing.

Listing 14.3 A simple WS-Security callback for a server-side app

```
package camelinaction.wssecurity;

import java.io.IOException;

import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;  ❶
```

❶

You need to implement a CallbackHandler.

```
import javax.security.auth.callback.UnsupportedCallbackException;

import org.apache.wss4j.common.ext.WSPasswordCallback;
```

```
public class ServerPasswordCallback implements CallbackHandler {

    public void handle(Callback[] callbacks)
        throws IOException, UnsupportedCallbackException {
        WSPasswordCallback pc = (WSPasswordCallback) callbacks[0];
    if ("jon".equals(pc.getIdentifier())) {    ❷
```

❷

Checks the username

```
        pc.setPassword("secret");    ❸
```

❸

Sets the password on the callback if the username is valid

```
        } else {
            throw new IOException("Username incorrect!");
        }
    }
}
```

First, your `ServerPasswordCallback` implements `CallbackHandler` ❶, which has a single method: `handle`. In the `handle` method, you need to check whether the username is valid ❷ and, if so, set the password on the callback ❸. The verification of the password is done by WSS4J under the covers. The password itself may not be stored in plain text, so comparing it may be an effort in itself. It's best to leave the checking up to WSS4J.

Now, the Rider Auto Parts order web service has full authentication support, albeit for a single user, `jon`, but still full authentication support. Let's see next how clients of this web service can add user credentials.

Client-side WS-Security configuration

Like the server-side case, the main thing you're doing to add WS-Security authentication to your application is adding a WSS4J interceptor to your

CXF-based web service. It's easiest to explain with code, so let's look at the following listing in detail.

Listing 14.4 WS-Security client-side configuration

```
protected static OrderEndpoint createCXFClient(
    String url, String user, String passwordCallbackClass) {
    List<Interceptor<? extends Message>> outInterceptors = new ArrayList

    // Define WSS4j properties for flow outgoing
    Map<String, Object> outProps = new HashMap<>();
  outProps.put("action", "UsernameToken Timestamp");  (1)
```

(1) Sets the actions to use when processing the UsernameToken

```
  outProps.put("user", user);  (2)
```

(2) Specifies the user

```
  outProps.put("passwordCallbackClass", passwordCallbackClass);  (3)
```

(3) Specifies the callback to use to grab the password

```
    WSS4JOutInterceptor wss4j = new WSS4JOutInterceptor(outProps);
    // Add LoggingOutInterceptor
    LoggingOutInterceptor loggingOutInterceptor = new LoggingOutIntercep

    outInterceptors.add(wss4j);
    outInterceptors.add(loggingOutInterceptor);

    // we use CXF to create a client for us as its easier than JAXWS and
  JaxWsProxyFactoryBean factory = new JaxWsProxyFactoryBean();  (4)
```

❹

Uses a CXF factory bean to create the OrderEndpoint

```
        factory.setOutInterceptors(outInterceptors);
        factory.setServiceClass(OrderEndpoint.class);
        factory.setAddress(url);
        return (OrderEndpoint) factory.create();
    }

    @Test
    public void testOrderOk() throws Exception {
        OrderEndpoint client = createCXFClient(
            "http://localhost:9000/order", "jon",
            "camelinaction.wssecurity.ClientPasswordCallback");
      OrderResult reply = client.order(new Order("motor", 100, "honda"));
```

❺

Places an order using the OrderEndpoint

```
        assertEquals("OK", reply.getMessage());
    }
```

Before you create a business-as-usual JAX-WS web service using a CXF factory bean ❹, you need to configure `WSS4JOutInterceptor`. You use the same actions as on the server side ❶. You want to specify `UsernameToken` and `Timestamp` so a passed authentication doesn't last indefinitely. You're on the client side now, so you need to specify the user you want accessing the remote service. You do this by adding a `user` entry for the user named `jon` ❷. You specify a different callback handler on the client side ❸, which we'll discuss next. Finally, you place an order using `OrderEndpoint` created by the CXF JAX-WS factory bean ❺; this will invoke the remote web service and return a result. All the marshaling and networking is handled by CXF under the hood.

The callback handler for the client side is a little different in that you're providing a password to use when calling the web service. The following listing shows such a callback handler.

Listing 14.5 A simple WS-Security callback for a client-side app

```
package camelinaction.wssecurity;

import java.io.IOException;

import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;

import org.apache.wss4j.common.ext.WSPasswordCallback;

public class ClientPasswordCallback implements CallbackHandler {

    public void handle(Callback[] callbacks)
        throws IOException, UnsupportedCallbackException {
        WSPasswordCallback pc = (WSPasswordCallback) callbacks[0];
    pc.setPassword("secret");
```

Specifies the password to use when invoking the web service

```
    }
}
```

This overly simplified callback handler gets the job done for your hard-coded example, but you wouldn't want to always return the same password for every user.

You can try this example for yourself by navigating to the chapter14/webservice directory and executing the following command:

```
mvn test -Dtest=WssAuthTest
```

In particular, note that the web service invocation will fail with `javax.xml.ws.soap.SOAPFaultException` when either the username or password is incorrect. Recall that this is how you designed the server-side callback handler in listing 14.3. Of course, with hardcoded usernames and passwords, using an incorrect one is highly unlikely. Let's look at how to integrate your web service with JAAS authentication.

14.2.2 Authenticating web services using JAAS

The developers at Rider Auto Parts have successfully demonstrated their authenticated web service to management. Now they're tasked with authenticating against the users available in their companywide container standard: Apache Karaf. Users in Karaf can come from various sources such as LDAP, database, and properties files. All of these are accessed via the Java Authentication and Authorization Service (JAAS).

This process is made easy using an existing class from WSS4J. Recall that in listing 14.3 the server-side `CallbackHandler` set the password text on `WSPasswordCallback` without checking for its validity. We mentioned leaving the password validation up to WSS4J. Validating against JAAS is as easy as switching to using a `org.apache.wss4j.dom.validate.JAASUsernameTokenValidator` password validator. The following listing shows you how.

Listing 14.6 Using `JAASUsernameTokenValidator` to authenticate against JAAS

```
<bean id="karafJaasValidator"     (1)
  class="org.apache.wss4j.dom.validate.JAASUsernameTokenValidator">     (1)
  <property name="contextName" value="karaf"/>     (1)
</bean>

<bean id="wss4jInInterceptor"
      class="org.apache.cxf.ws.security.wss4j.WSS4JInInterceptor">
  <argument>
     <map>
    <entry key="action" value="UsernameToken"/>     (2)
```

(1) Points the JAASUsernameTokenValidator to the Karaf container JAAS context

(2) The WSS4JInInterceptor should check for UsernameToken.

```
      </map>
    </argument>
  </bean>

  <cxf:cxfEndpoint id="orderEndpoint"
                   address="/order"    ❹
```

❹

Uses relative path for web service

```
                   serviceClass="camelinaction.order.OrderEndpoint">

    <cxf:inInterceptors>
      <ref component-id="loggingInInterceptor"/>
      <ref component-id="wss4jInInterceptor"/>
    </cxf:inInterceptors>
    <cxf:outInterceptors>
      <ref component-id="loggingOutInterceptor"/>
    </cxf:outInterceptors>

    <cxf:properties>
      <entry key="ws-security.ut.validator">    ❸
```

❸

Overrides the default password validator with JAASUsernameTokenValidator

```
        <ref component-id="karafJaasValidator"/>    ❸
      </entry>    ❸
    </cxf:properties>

  </cxf:cxfEndpoint>
```

First, you construct an `org.apache.wss4j.dom.validate.JAASUsernameTokenValidator` that's configured to check `UsernameToken` credentials against `javax.security.auth.login.LoginContext` with the name `karaf` ❶.

When deployed into Apache Karaf, this will pick up whatever login module it's using for authentication (for example, LDAP, database, or properties file). Now when you create `WSS4JInInterceptor` ❷, it's different from before. The main difference is that you're now supplying no `CallbackHandler`. This is a feature of Apache CXF; because you provided no `CallbackHandler` and your `UsernameToken` has a password set, CXF creates a simple `CallbackHandler` for you. This simple `CallbackHandler` sets the password on `WSCallbackHandler`, similar to what you were doing manually before. Finally, you need to override the default password validator with your JAAS-powered one. You can do that using the `ws-security.ut.validator` CXF configuration property ❸. With this new configuration, your web service is now authenticated against container credentials.

NOTE Web services deployed into a container such as Apache Karaf typically reuse an HTTP service that's already running. In Karaf's case, HTTP services are by default on port 8181, and CXF web services are registered under the /cxf path. For your web service in listing 14.6, the /order path ❹ will be available at http://localhost:8181/cxf/order.

To demonstrate this, you need to deploy the example into Apache Karaf. At the time of writing, you're using the latest version: Apache Karaf 4.1.2. You start Apache Karaf using the following command:

```
bin/karaf
```

Then install Camel version 2.20.1 and CXF WS-Security support in Karaf:

```
feature:repo-add camel 2.20.1
feature:install camel-cxf cxf-ws-security camel-blueprint
```

Now you need to install your Rider Auto Parts web service in Karaf, which you can do using the chapter14/webservice-karaf example from the book's source code. At first, you need to build this example:

```
mvn clean install
```

You also need to build the chapter14/webservice example from the previous section if you haven't already. Now from the Karaf command line, type this:

```
install -s mvn:com.camelinaction/chapter14-webservice/2.0.0
install -s mvn:com.camelinaction/chapter14-webservice-karaf/2.0.0
```

This installs and starts the example (the `-s` flag refers to *start*).

The chapter14/webservice-karaf directory also contains a client that you can use to access the web service deployed in Karaf. This client is shown in the following listing.

Listing 14.7 A client to access the Rider Auto Parts web service

```
public class Client {

    public static void main(String[] args) {
       List<Interceptor<? extends Message>> outInterceptors = new ArrayL

       // Define WSS4j properties for flow outgoing
       Map<String, Object> outProps = new HashMap<>();
       outProps.put("action", "UsernameToken");
    outProps.put("user", "karaf");    (1)
```

(1) Uses the default Karaf user

```
    outProps.put("passwordType", "PasswordText");    (2)
```

(2) Uses text-based password to be compatible with the JAAS validator

```
    outProps.put("passwordCallbackClass",    (3)
```

❸

Uses a CallbackHandler that accepts a password from the console

```
        "camelinaction.StdInPasswordCallback");   ❸

        WSS4JOutInterceptor wss4j = new WSS4JOutInterceptor(outProps);
        outInterceptors.add(wss4j);

        JaxWsProxyFactoryBean factory = new JaxWsProxyFactoryBean();
         factory.setOutInterceptors(outInterceptors);
         factory.setServiceClass(OrderEndpoint.class);
         factory.setAddress("http://localhost:8181/cxf/order");

         OrderEndpoint client = (OrderEndpoint) factory.create();

         Order order = new Order("motor", 100, "honda");
         System.out.println("Placing order for: " + order);
         OrderResult reply = client.order(order);

         System.out.println("Rider Auto Web service returned: "
             + reply.getMessage());
    }
}
```

Your simple `Client` class looks similar to the web services test from the previous section. The first main difference is that you're specifying the default Karaf admin user `karaf` ❶ as the user. You then set the password type as `text` so that the WSS4J JAAS validator can compare it correctly ❷. Finally, you specify `CallbackHandler`, which reads the password from the console ❸. This simple `CallbackHandler` is shown in the following listing.

Listing 14.8 A `CallbackHandler` that reads passwords from the console

```
public class StdInPasswordCallback implements CallbackHandler {

    public void handle(Callback[] callbacks)
        throws IOException, UnsupportedCallbackException {
        WSPasswordCallback pc = (WSPasswordCallback) callbacks[0];
        Console console = System.console();
```

```
            console.printf("Please enter your Rider Auto Parts password: ");
            char[] passwordChars = console.readPassword();
            String passwordString = new String(passwordChars);
            pc.setPassword(passwordString);
        }
    }
```

To try this client, you can use the following command:

```
mvn exec:java -Pclient
```

You'll be prompted for a password to use:

```
Please enter your Rider Auto Parts password:
```

The default username and password in Karaf is karaf/karaf, so in the prompt you need to enter `karaf` as the password. If you don't enter the correct password, `javax.xml.ws.soap.SOAPFaultException` will be thrown from the CXF client.

The WS-Security support in CXF also allows you to do things such as sign or encrypt your messages, among other things. For more information on how to configure these, refer to the Apache CXF website: http://cxf.apache.org/docs/ws-security.html.

Now that you've secured your Rider Auto Parts web service, let's look at securing more general payloads you could be sending with Camel.

14.3 Payload security

You have two main options for securing the message payloads you're sending with Camel. One involves encrypting the entire contents of the message—a useful option when you're sending sensitive information that you wouldn't want viewed by any listening parties. But most often, you can pass the buck of encrypting your data down to the transport layer via the TLS protocol rather than handling it within a Camel route. These will probably be the most common secured endpoints you'll encounter out in the wild. To see more about configuring TLS, skip to section 14.4. There's a difference to handling encryption within Camel or delegating to the transport layer. Figures 14.1 and 14.2 illustrate this difference.

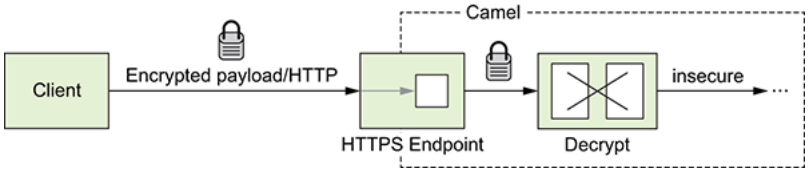

Figure 14.1 Using the camel-crypto data format, you can handle encryption and decryption of payloads within your Camel route. This gives you full control over what transport you send over—for example, an unsecured HTTP pipe.

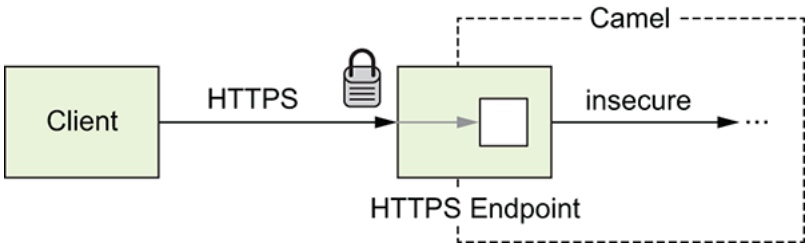

Figure 14.2 Using transport-based security to handle payload encryption gives you less choice over what transport you can use, because the underlying transport has to support JSSE. For a complete discussion on transport security, see section 14.4.

Another option in Camel to secure your payload is to send a digital signature along with the message so that the receiver can verify that the message is unchanged and that the sender is correct. Let's look at how to sign and verify a message first.

14.3.1 Digital signatures

Adding a digital signature to a message gives the receiver the ability to confirm that you sent the message and that no one tampered with it in transit. This is especially crucial in messages such as financial transactions, where a change of message origin or modification of a decimal place, say, would have large consequences. Digital signatures are generated using asymmetric cryptography: the sender generates the signature using a private key that only they have. The receiver then uses a different publicly available key to verify that the message is indeed the same. The mathematics behind this relationship is complex, but fortunately you don't have to delve into that to use the algorithms.

For signing messages in Camel, there's the camel-crypto component. To use the camel-crypto component within your Camel project, you need to add a dependency to it in your Maven POM:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-crypto</artifactId>
```

```
  <version>2.20.1</version>
</dependency>
```

Before digging into what the camel-crypto component gives you, you first have to create the public and private keys that will feed into the cryptography algorithms. For this, the keytool utility that ships with the Java Development Kit (JDK) is required.

Generating and loading public and private keys

Going over all the options of the keytool utility is beyond the scope of this book. To not confuse you (and even us), we'll stick to using default cryptographic algorithms where possible. If you do decide that the default algorithm isn't sufficient for your application, make sure to specify the same algorithm in keytool as you do in Camel.

The first step in using keytool is to generate a *key pair*—a private and public key for the sender. This is accomplished with the `-genkeypair` command:

```
keytool -genkeypair \
     -alias jon \
     -dname "CN=Jon Anstey, L=Paradise, ST=Newfoundland, C=Canada" \
     -validity 7300 \
     -keypass secret \
     -keystore cia_keystore.jks \
     -storepass supersecret
```

Here you're creating a public/private key pair with the alias `jon`, expiration in 7,300 days, and key password of `secret`. The `dname` argument specifies the distinguished name of the certificate issuer. Finally, you specify `cia_keystore.jks` as the keystore file with a password of `supersecret`. If you use the `-list` keytool command, you can see that there's one key-pair in your keystore:

```
[janstey@bender]$ keytool -list -keystore cia_keystore.jks \
-storepass supersecret

Keystore type: JKS
Keystore provider: SUN
```

```
Your keystore contains 1 entry

jon, Jun 29, 2017, PrivateKeyEntry,
Certificate fingerprint (SHA1): EE:B0:BC:F3:B0:8E:24:C4:7B:0E:60:47:11:7
```

Now this keystore is all ready for you to start signing messages. But you need to extract the public key into a separate keystore, because you can't give out your private key. This keystore that contains only public keys is called the *trust store*. To do this, you can use the `-exportcert` and `-importcert` keytool commands:

```
[janstey@bender]$ keytool -exportcert -rfc -alias jon -file jon.cer \
                  -keystore cia_keystore.jks -storepass supersecret
Certificate stored in file <jon.cer>
[janstey@bender]$ keytool -importcert -noprompt -alias jon \
                  -file jon.cer -keystore cia_truststore.jks \
                  -storepass supersecret
Certificate was added to keystore
```

Now before you can use the keys stored in the keystores in Camel, you need to load them into the registry. Fortunately, Camel has the `org.apache.camel.util.jsse.KeyStoreParameters` helper class so you don't need to do this by hand. In Java, using `KeyStoreParameters` looks like this:

```
@Override
protected JndiRegistry createRegistry() throws Exception {
    JndiRegistry registry = super.createRegistry();

    KeyStoreParameters keystore = new KeyStoreParameters();
    keystore.setPassword("supersecret");
    keystore.setResource("./cia_keystore.jks");
    registry.bind("keystore", keystore);

    KeyStoreParameters truststore = new KeyStoreParameters();
    truststore.setPassword("supersecret");
    truststore.setResource("./cia_truststore.jks");
    registry.bind("truststore", truststore);
```

```
    return registry;
}
```

In Spring or Blueprint XML, you can do this just as easily:

```
<keyStoreParameters xmlns="http://camel.apache.org/schema/spring"
  id="keystore" resource="./cia_keystore.jks" password="supersecret"/>
<keyStoreParameters xmlns="http://camel.apache.org/schema/spring"
  id="truststore" resource="./cia_truststore.jks" password="supersecret"
```

You have the key and trust stores ready, so you can start signing and verifying messages.

Signing and verifying messages using camel-crypto

The camel-crypto component provides two endpoints for signing and verifying messages. The URI format is as follows:

```
crypto:sign:endpointName[?options]
crypto:verify:endpointName[?options]
```

Intuitively, you'd use `crypto:sign` to add a digital signature to the message. Using our previously generated keystore, this would look something like this:

```
from("direct:sign")
    .to("crypto:sign://keystore?keyStoreParameters=#keystore     &alias=
...
```

In XML, this would look like the following:

```
<route>
    <from uri="direct:sign"/>
    <to uri="crypto:sign://keystore?keyStoreParameters=#keystore&amp;
                                   alias=jon&amp;
                                   password=secret"/>
    ...
</route>
```

This would add a `CamelDigitalSignature` header (defined in the `org.apache.camel.component.crypto.DigitalSignatureConstants.SIGNATURE` constant) to the message with a value calculated using the default SHA1WithDSA algorithm and the private key with alias `jon` in the keystore loaded via `KeyStoreParameters` in the `#keystore` reference. Verifying this message later is again a simple step in your route:

```
from("direct:verify")
    .toF("crypto:verify://keystore?keystore=%s&alias=%s&password=%s",
     "#truststore", "jon", "secret")
...
```

In XML DSL, this would look like the following:

```
<route>
    <from uri="direct:verify"/>
    <to uri="crypto:verify://keystore?keyStoreParameters=#truststore&amp
                                     alias=jon&amp;
                                     password=secret"/>
    ...
</route>
```

The only difference from before is that now you're using the `verify` endpoint and you're using the truststore instead of the keystore. On success, nothing will happen, and your route will continue as normal. But when something bad happens to your message in transit, the verify call will generate a `java.security.SignatureException`, which you can then handle via Camel's exception-handling facility. For example, let's try to simulate a man-in-the-middle type attack by modifying the message body before it gets to the verify step.

NOTE See https://en.wikipedia.org/wiki/Man-in-the-middle_attack for more information on this type of attack.

You'll use three routes to simulate this type of attack:

```
from("direct:sign")
    .toF("crypto:sign://keystore?keystore=%s&alias=%s&password=%s",
```

```
            "#keystore", "jon", "secret")
        .to("mock:signed")
        .to("direct:mitm");

    from("direct:mitm")
        .setBody().simple("I'm hacked!")
        .to("direct:verify");

    from("direct:verify")
        .toF("crypto:verify://keystore?keystore=%s&alias=%s&password=%s",
            "#truststore", "jon", "secret")
        .to("mock:verified");
```

Now when sending a message to the `direct:sign` endpoint

```
try {
    template.sendBody("direct:sign", "Hello World");
} catch (CamelExecutionException e) {
    assertIsInstanceOf(SignatureException.class, e.getCause());
}
```

you can expect its body to be modified to “I'm hacked!” and then the verify step will fail. Notice that Camel throws `CamelExecutionException` with the cause being `SignatureException`.

You can run this example for yourself using the following Maven goals from the chapter14/payload directory:

```
mvn test -Dtest=MessageSigningWithKeyStoreParamsTest
mvn test -Dtest=SpringMessageSigningWithKeyStoreParamsTest
mvn test -Dtest=ManInTheMiddleTest
```

Now that you’ve stopped a man-in-the-middle attack using digital signatures, let’s see how to encrypt entire payloads.

14.3.2 Payload encryption

Sometimes even viewing the contents of a message can be considered a security breach. For these cases, encrypting the entire payload can be useful. Encryption can be performed via symmetric or asymmetric cryptography. In *symmetric* cryptography, both the sender and receiver share a

secret key. This is in contrast to digital signatures we looked at previously, where the sender and receiver had different but complementary keys (they were *asymmetric*). Because you just looked at an asymmetric example, let's take a look at how a symmetric case would work.

NOTE Asymmetric encryption is handled with `PGPDataformat`. You can find more information on this data format on Camel's website, at http://camel.apache.org/crypto.html.

Similar to when signing messages in Camel, the camel-crypto component is used. You first have to create the key that will feed into the cryptography algorithms. Symmetric cryptography needs a completely different type of key than what was used for the digital signatures, so you can't reuse those keys.

Generating and loading secret keys

To generate a secret key using keytool, you can use the `-genseckey` command:

```
keytool -genseckey
    -alias ciasecrets
    -keypass secret
    -keystore cia_secrets.jceks
    -storepass supersecret
    -storetype JCEKS
```

Here you're creating a secret key with the alias `ciasecrets` and key password of `secret`. Note that you can't use the default store type anymore; you need to use `JCEKS` for storing secret keys. Finally, you specify `cia_secrets.jceks` as the keystore file with a password of `supersecret`.

As for loading the keystore in Camel, you need to load the keystores in the same way as you did for message signing, except you can't use the default keystore type anymore. You need to specify `JCEKS` for the type:

```
@Override
protected JndiRegistry createRegistry() throws Exception {
```

```
        JndiRegistry registry = super.createRegistry();

        KeyStoreParameters keystore = new KeyStoreParameters();
        keystore.setPassword("supersecret");
        keystore.setResource("./cia_secrets.jceks");
        keystore.setType("JCEKS");

        KeyStore store = keystore.createKeyStore();
        secretKey = store.getKey("ciasecrets", "secret".toCharArray());
        registry.bind("secretKey", secretKey);
        return registry;
    }
```

Here you use `KeyStoreParameters` again to do the grunt work of loading the keystore, but you're adding the secret key to the registry instead of `KeyStoreParameters`.

Encrypting and decrypting payloads using camel-crypto

Payload encryption in camel-crypto is implemented using data formats. We looked at data formats in chapter 3. Basically, using the `marshal` DSL method will encrypt the payload, and `unmarshal` will decrypt. Before you can call either `marshal` or `unmarshal`, you need to create the data format:

```
CryptoDataFormat crypto = new CryptoDataFormat("DES", secretKey);
```

Here you specify an encryption algorithm of Digital Encryption Standard (DES) and the secret key you created and loaded previously.

TIP All possible cipher algorithms are listed in the Java Cryptography Architecture Standard Algorithm Name Documentation. See https://docs.oracle.com/javase/8/docs/technotes/guides/security/StandardNames.html#C

Now you can use the data format in a route like this:

```
from("direct:start")
        .marshal(crypto)
```

```
        .to("mock:encrypted")
        .unmarshal(crypto)
        .to("mock:unencrypted");
```

In XML DSL, this would look like the following:

```
<bean id="secretKey"
       class="camelinaction.SpringMessageEncryptionTest"
    factory-method="loadKey"/>

<camelContext xmlns="http://camel.apache.org/schema/spring">
    <dataFormats>
        <crypto id="myCrypto" algorithm="DES" keyRef="secretKey"/>
    </dataFormats>
    <route>
        <from uri="direct:start"/>
        <marshal ref="myCrypto"/>
        <to uri="mock:encrypted"/>
        <unmarshal ref="myCrypto"/>
        <to uri="mock:unencrypted"/>
    </route>
</camelContext>
```

At the `mock:encrypted` endpoint, the payload is encrypted, and at the `mock:unencrypted` endpoint, the payload is back to plain text. This route demonstrates how easily you can encrypt and decrypt a payload in Camel. After you've loaded your key and set up your data format, that's all there is to it.

You can run this example for yourself using the following Maven goals from the chapter14/payload directory:

```
mvn test -Dtest=MessageEncryptionTest
mvn test -Dtest=SpringMessageEncryptionTest
```

Another option to ensure that your data is encrypted outside your Camel route is to use transport security, which is all about setting up the TLS protocol on the underlying transport.

14.4 Transport security

Many components in Camel allow you to configure TLS settings for the transport type being offered. In Java, TLS is typically configured using the Java Secure Socket Extension (JSSE) API provided with the JRE. But not all third-party libraries that Camel integrates with use JSSE in a common way. To make this a bit easier, Camel provides a layer on top of this, called the JSSE Utility. At the time of writing, many Camel components use this utility:

- camel-ahc
- camel-apns
- camel-box
- camel-cometd
- camel-ftp
- camel-http and camel-http4
- camel-irc
- camel-jetty9
- camel-linkedin
- camel-mail
- camel-mina2
- camel-netty and camel-netty4
- camel-olingo2 and camel-olingo4
- camel-restlet
- camel-salesforce
- camel-spring-ws
- camel-undertow
- camel-websocket

The central class you'll be using from this utility is `org.apache.camel.util.jsse.SSLContextParameters`. That's what you'll be passing into the Camel components to configure the TLS settings. You can configure this via pure Java or in Spring or Blueprint XML. For example, in Spring, a possible `SSLContextParameters` configuration looks like this:

```
<sslContextParameters id="sslContextParameters"
    xmlns="http://camel.apache.org/schema/spring">
```

```
    <keyManagers keyPassword="secret">
        <keyStore resource="./cia_keystore.jks" password="supersecret"/>
    </keyManagers>
    <trustManagers>
        <keyStore resource="./cia_truststore.jks" password="supersecret"
    </trustManagers>
</sslContextParameters>
```

Here you have a top-level `sslContextParameters` element and then `keyManagers` and `trustManagers` children. These configure the keystore and truststore files, respectively, that you created in section 14.3.1. From here, you can then reference the `sslContextParameters` bean in your endpoint URI:

```
<route>
    <from uri="jetty:https://localhost:8080/early?sslContextParameters=#
    <transform>
        <constant>Hi</constant>
    </transform>
</route>
```

With this route, you now have an HTTPS endpoint up and running. In Java, things get more verbose, but it's mostly the same:

```
@Override
protected JndiRegistry createRegistry() throws Exception {
    KeyStoreParameters ksp = new KeyStoreParameters();
    ksp.setResource("./cia_keystore.jks");
    ksp.setPassword("supersecret");
    KeyManagersParameters kmp = new KeyManagersParameters();
    kmp.setKeyPassword("secret");
    kmp.setKeyStore(ksp);

    KeyStoreParameters tsp = new KeyStoreParameters();
    tsp.setResource("./cia_truststore.jks");
    tsp.setPassword("supersecret");
    TrustManagersParameters tmp = new TrustManagersParameters();
    tmp.setKeyStore(tsp);

    SSLContextParameters sslContextParameters = new SSLContextParameters
    sslContextParameters.setKeyManagers(kmp);
```

```
        sslContextParameters.setTrustManagers(tmp);

        JndiRegistry registry = super.createRegistry();
        registry.bind("ssl", sslContextParameters);

        return registry;
    }
```

The Java route is also similar:

```
    from("jetty:https://localhost:8080/early?sslContextParameters=#ssl")
        .transform().constant("Hi");
```

When calling these HTTPS endpoints within Camel, you need to also provide `sslContextParameters`, which contains a trusted certificate. For the purposes of the example, you can reuse the server `sslContextParameters`. The URI syntax is exactly the same for the producer in this case:

```
    @Test
    public void testHttps() throws Exception {
        String reply = template.requestBody(
            "jetty:https://localhost:8080/early?sslContextParameters=#ssl",
            "Hi Camel!", String.class);
        assertEquals("Hi", reply);
    }
```

If you didn't provide any `sslContextParameters` with a valid trust store, you should expect Camel to throw an execution exception, because the server wouldn't let you connect:

```
    @Test(expected = CamelExecutionException.class)
    public void testHttpsNoTruststore() throws Exception {
        String reply = template.requestBody(
            "jetty:https://localhost:8080/early", "Hi Camel!", String.class);
        assertEquals("Hi", reply);
    }
```

You can try this for yourself using the following Maven goals from the chapter14/transport directory:

```
mvn test -Dtest=HttpsTest
mvn test -Dtest=SpringHttpsTest
```

14.4.1 Defining global SSL configuration

If you use the same SSL configuration in most of your routes, it makes sense to define this configuration globally. You can do this by first setting the global `SSLContextParameters` on `CamelContext`:

```
CamelContext context = super.createCamelContext();
context.setSSLContextParameters(createSSLContextParameters());
```

The `createSSLContextParameters` method in this case creates an `SSLContextParameters` object:

```
private SSLContextParameters createSSLContextParameters() {
    KeyStoreParameters ksp = new KeyStoreParameters();
    ksp.setResource("./cia_keystore.jks");
    ksp.setPassword("supersecret");
    KeyManagersParameters kmp = new KeyManagersParameters();
    kmp.setKeyPassword("secret");
    kmp.setKeyStore(ksp);

    KeyStoreParameters tsp = new KeyStoreParameters();
    tsp.setResource("./cia_truststore.jks");
    tsp.setPassword("supersecret");
    TrustManagersParameters tmp = new TrustManagersParameters();
    tmp.setKeyStore(tsp);

    SSLContextParameters sslContextParameters = new SSLContextParameters
    sslContextParameters.setKeyManagers(kmp);
    sslContextParameters.setTrustManagers(tmp);

    return sslContextParameters;
}
```

You can then enable this global SSL config for each component you're using. In this case, the SSL examples in this chapter use the Jetty component so you can enable global SSL like so:

```
((SSLContextParametersAware) context.getComponent("jetty"))
    .setUseGlobalSslContextParameters(true);
```

After this, you no longer have to reference the `SSLContextParameters` in our endpoint URIs. The example route you used previously simplifies down to this:

```
from("jetty:https://localhost:8080/early")
    .transform().constant("Hi");
```

You can try this for yourself using the following Maven goal from the chapter14/transport directory:

```
mvn test -Dtest=GlobalSSLContextParametersTest
```

Now that we've covered security at the transport level, let's move back into Camel and look at implementing authentication and authorization in your routes.

14.5 Route authentication and authorization

The process of authentication and authorization are separate concepts, but they always go together. *Authentication* is the first step, whereby a user proves who they claim to be to a system using, typically, a username and password. Next, after successful authentication, *authorization* is the process whereby the system checks what you're allowed to do. Camel supports both of these processes within a route so you can control who is allowed to use a particular route. There are two implementations of route security: one using Apache Shiro, and another using Spring Security. Both frameworks provide a means of authenticating and authorizing a user. They do more than that, but from a Camel point of view, we're using only these features. You can find out more about the respective projects on their websites:

- *Spring Security*—http://projects.spring.io/spring-security
- *Apache Shiro*—http://shiro.apache.org

Spring Security seems to be the most popular and active project of the two, so we focus on that in the examples of this section. For more infor-

mation on Camel's Shiro support, check out the website docs at http://camel.apache.org/shiro-security.html.

To get started using Spring Security in Camel, you need to add a dependency on the camel-spring-security module to your Maven POM:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-spring-security</artifactId>
  <version>2.20.1</version>
</dependency>
```

This brings in the necessary Spring Security JARs transitively, so you don't need to add them to your POM as well. If you're curious, though, or are looking at the reference documentation, Camel uses the spring-security-core and spring-security-config modules. Several more modules are available to support things like LDAP and ACLs. These additional modules are beyond the scope of this section, so to find out more, check out the Spring Security project's website.

Route security in Camel is handled via a *policy*, a class that implements `org.apache.camel.spi.Policy`. A route protected by a policy looks like this:

```
<route>
    <from uri="direct:start"/>
    <to uri="mock:unsecure"/>
    <policy ref="admin">
        <to uri="mock:secure"/>
    </policy>
</route>
```

Here you're referencing a policy defined by the `admin` bean. Note that the whole route isn't protected by the policy—only what's inside the policy element. You must take care to place the secured calls inside the policy element. A *policy* is a generic Camel concept, so you haven't really done any securing yet. Let's take a look at that `admin` bean to see how it's helping to secure your route:

```
<authorizationPolicy xmlns="http://camel.apache.org/schema/spring-securi
  id="admin" access="ROLE_ADMIN"
  authenticationManager="authenticationManager"
  accessDecisionManager="accessDecisionManager"/>
```

This `authorizationPolicy` element is syntactic sugar for the `org.apache.camel.component.spring.security.SpringSecurityAuthorizationPolic` class, which implements the `Policy` interface. You could have created a `SpringSecurityAuthorizationPolicy` bean here just the same.

Let's go over what the attributes of `authorizationPolicy` are doing. First, the `accessDecisionManager` attribute specifies the `org.springframework.security.access.AccessDecisionManager` that you'll be using. This is the class that will be deciding whether the user will be authorized to use the route. The `access` attribute specifies the authority name passed to the `AccessDecisionManager`. This is most often a role name (a string beginning with the prefix `ROLE_`). Finally, the `authenticationManager` attribute specifies the `org.springframework.security.authentication.AuthenticationManager` bean that you'll be using to authenticate the user.

14.5.1 Configuring Spring Security

Outside Camel, you need to configure two required Spring Security beans: `AccessDecisionManager` and `AuthenticationManager`. To configure these, you need to add an extra XML namespace to your Spring XML:

```
xmlns:spring-security="http://www.springframework.org/schema/security"
xsi:schemaLocation="
http://www.springframework.org/schema/security http://www.springframewor
...
```

From here, you can then define `AuthenticationManager`:

```
<spring-security:authentication-manager alias="authenticationManager">
    <spring-security:authentication-provider
        user-service-ref="userDetailsService"/>
</spring-security:authentication-manager>

<spring-security:user-service id="userDetailsService">
```

```
    <spring-security:user name="jon"
        password="secret" authorities="ROLE_USER"/>
    <spring-security:user name="claus"
        password="secret" authorities="ROLE_USER, ROLE_ADMIN"/>
</spring-security:user-service>
```

This is probably the simplest `AuthenticationManager` configuration you can get. All you do here is check that a password matches the one provided for a particular user. The usernames, passwords, and roles are all defined within the XML snippet. In a real-world deployment, you'd probably have users checked via JAAS or by querying an LDAP database. It's far easier to talk about an XML snippet over an LDAP server configuration! In the preceding snippet, you're defining two users: `jon` and `claus`. Each has a different set of roles. Only `claus` has the `admin` role. Recall that before, your Camel `authorizationPolicy` had an access parameter defined as `ROLE_ADMIN`, which means only users with the `admin` role will be allowed to access the route inside the policy element. But how is this role checking defined? The answer is `AccessDecisionManager`. For our role-checking example, you can use the following `AccessDecisionManager` configuration:

```
<bean id="accessDecisionManager"
      class="org.springframework.security.access.vote.AffirmativeBased"
    <property name="decisionVoters">
        <list>
            <bean class="org.springframework.security.access.vote.RoleV
        </list>
    </property>
</bean>
```

The concrete `AccessDecisionMaker` you're using here, `AffirmativeBased`, will grant access if one of the decision voters succeeds. You have only one decision voter configured here, so it's one vote or nothing. The sole vote is coming from `RoleVoter`, which will grant access only if the user has the specified role (in this case, it's `ROLE_ADMIN`).

Now your route is secured! But you need to do a little more work to use it now. Trying to access it at this point will certainly fail, unless you pass in the proper credentials. `SpringSecurityAuthorizationPolicy` looks for

`Exchange.AUTHENTICATION` headers (`CamelAuthentication`) for the `javax.security.auth.Subject` it will be authenticating. You can easily create this header from a username and password:

```
private void sendMessageWithAuth(String uri, String body,
                                 String username, String password) {
   Authentication authToken =
      new UsernamePasswordAuthenticationToken(username, password);

   Subject subject = new Subject();
   subject.getPrincipals().add(authToken);

   template.sendBodyAndHeader(uri, body,
      Exchange.AUTHENTICATION, subject);
}
```

Let's look at a test in action:

```
@Test
public void testAdminOnly() throws Exception {
    getMockEndpoint("mock:secure").expectedBodiesReceived("Davs Claus!")
    getMockEndpoint("mock:unsecure").expectedBodiesReceived("Davs Claus!
        , "Hello Jon!");
    sendMessageWithAuth("direct:start", "Davs Claus!", "claus", "secret"
    try {
        sendMessageWithAuth("direct:start",
                            "Hello Jon!", "jon", "secret");
    } catch (CamelExecutionException e) {
        assertIsInstanceOf(CamelAuthorizationException.class,
            e.getCause());
    }

    assertMockEndpointsSatisfied();
}
```

Here you're sending two messages: one using claus's credentials, and another using jon's. Because your policy requires the `admin` role, only claus's message gets through. On sending jon's credentials, Camel throws `CamelExecutionException`, with the cause being `CamelAuthorizationException`.

You can run this example for yourself using the following Maven goals from the chapter14/route directory:

```
mvn test -Dtest=SpringSecurityTest
```

This covers the basics of using Spring Security in Camel for route authentication and authorization. For more information, see the Camel website at http://camel.apache.org/spring-security.html.

14.6 Summary and best practices

This concludes our whirlwind tour of Camel's security features. As with many other areas of Camel, you can implement security in many ways. Camel doesn't force you to use a set library for security.

Let's recap some key ideas from this chapter:

- *Secure only what you need to*—Of the four types of security configuration in Camel, you often don't need to use all of them, or even more than one. For instance, you probably don't need to encrypt a payload when you have transport security enabled, as this will also encrypt the payload for you. Also, extra unnecessary security configuration takes away from the readability of your Camel applications.
- *Camel is unsecured by default*—Probably the most important point is that Camel has no security settings turned on by default. This is great for development, but before your application is deployed in the real world, you'll most likely need some form of security enabled.

Speaking of deploying Camel applications in the real world, in chapter 15 we'll be talking all about running and deploying Camel in various environments.