gauth

gauth Component

Available in Camel 2.3

The gauth component is used by web applications to implement a Google-specific OAuth consumer. It will be later extended to support other OAuth providers as well. Although this component belongs to the Camel Components for Google App Engine (GAE), it can also be used to OAuth-enable non-GAE web applications. For a detailed description of Google's OAuth implementation refer to the Google OAuth API reference.

URI format

```
gauth://name[?options]
```

The endpoint name can be either authorize or upgrade. An authorize endpoint is used to obtain an unauthorized request token from Google and to redirect the user to the authorization page. The upgrade endpoint is used to process OAuth callbacks from Google and to upgrade an authorized request token to a long-lived access token. Refer to the usage section for an example.

Options

Name	Default Value	Required	Description
callback	null	true (can alternatively be set via GAuthAuthorizeBinding.GAUTH_CALLBACK message header)	URL where to redirect the user after having granted or denied access.
scope	null	true (can alternatively be set via GAuthAuthorizeBinding.GAUTH_SCOPE message header)	URL identifying the service(s) to be accessed. Scopes are defined by each Google service; see the service's documentation for the correct value. To specify more than one scope, list each one separated with a comma. Example: http://www.google.com/calendar/feeds/.
consumerKey	null	true (can alternatively be set on component-level).	Domain identifying the web application. This is the domain used when registering the application with Google. Example: camelcloud.appspot.com. For a non-registered application use anonymous.
consumerSecret	null	one of consumerSecret or keyLoaderRef is required (can alternatively be set on component-level).	Consumer secret of the web application. The consumer secret is generated when when registering the application with Google. It is needed if the HMAC-SHA1 signature method shall be used. For a non-registered application use anonymous.
keyLoaderRef	null	one of consumerSecret or keyLoaderRef is required (can be alternatively set on component-level)	Reference to a private key loader in the registry. Part of camel-gae are two key loaders: GAuthPk8Loader for loading a private key from a PKCS#8 file and GAuthJksLoader to load a private key from a Java key store. It is needed if the RSA-SHA1 signature method shall be used. These classes are defined in the org.apache.camel.component.gae.auth package.
authorizeBindingRef	Reference to GAuthAuthorizeBinding	false	Reference to a OutboundBinding<GAuthEndpoint, GoogleOAuthParameters, GoogleOAuthParameters> in the registry for customizing how an Exchange is bound to GoogleOAuthParameters. This binding is used for teh authorization phase. Most applications won't change the default value.
upgradeBindingRef	Reference to GAuthAuthorizeBinding	false	Reference to a OutboundBinding<GAuthEndpoint, GoogleOAuthParameters, GoogleOAuthParameters> in the registry. for customizing how an Exchange is bound to GoogleOAuthParameters. This binding is used for teh token upgrade phase. Most applications won't change the default value.

Message headers

Name	Type	Endpoint	Message	Description
GAuthAuthorizeBinding.GAUTH_CALLBACK	String	gauth:authorize	in	Overrides the callback option.
GAuthAuthorizeBinding.GAUTH_SCOPE	String	gauth:authorize	in	Overrides the scope option.
GAuthUpgradeBinding.GAUTH_ACCESS_TOKEN	String	gauth:upgrade	out	Contains the long-lived access token. This token should be stored by the applications in context of a user.
GAuthUpgradeBinding.GAUTH_ACCESS_TOKEN_SECRET	String	gauth:upgrade	out	Contains the access token secret. This token secret should be stored by the applications in context of a user.

Message body

The gauth component doesn't read or write message bodies.

Component configuration

Some endpoint options such as `consumerKey`, `consumerSecret` or `keyLoader` are usually set to the same values on `gauth:authorize` and `gauth:upgrade` endpoints. The `gauth` component allows to configure them on component-level. These settings are then inherited by `gauth` endpoints and need not be set redundantly in the endpoint URIs. Here are some configuration examples.

component configuration for a registered web application using the HMAC-SHA1 signature method

```
<bean id="gauth" class="org.apache.camel.component.gae.auth.GAuthComponent">
    <property name="consumerKey" value="example.appspot.com" />
    <property name="consumerSecret" value="QAtA...HfQ" />
</bean>
```

component configuration for an unregistered web application using the HMAC-SHA1 signature method

```
<bean id="gauth" class="org.apache.camel.component.gae.auth.GAuthComponent">
    <!-- Google will display a warning message on the authorization page -->
    <property name="consumerKey" value="anonymous" />
    <property name="consumerSecret" value="anonymous" />
</bean>
```

component configuration for a registered web application using the RSA-SHA1 signature method

```
<bean id="gauth" class="org.apache.camel.component.gae.auth.GAuthComponent">
    <property name="consumerKey" value="ipfcloud.appspot.com" />
    <property name="keyLoader" ref="jksLoader" />
    <!--<property name="keyLoader" ref="pk8Loader" />-->
</bean>

<!-- Loads the private key from a Java key store -->
<bean id="jksLoader"
    class="org.apache.camel.component.gae.auth.GAuthJksLoader">
    <property name="keyStoreLocation" value="myKeytore.jks" />
    <property name="keyAlias" value="myKey" />
    <property name="keyPass" value="myKeyPassword" />
    <property name="storePass" value="myStorePassword" />
</bean>

<!-- Loads the private key from a PKCS#8 file -->
<bean id="pk8Loader"
    class="org.apache.camel.component.gae.auth.GAuthPk8Loader">
    <property name="keyStoreLocation" value="myKeyfile.pk8" />
</bean>
```

Usage

Here's the minimum setup for adding OAuth to a (non-GAE) web application. In the following example, it is assumed that the web application is running on `gauth.example.org`.

GAuthRouteBuilder.java

```
import java.net.URLEncoder;
import org.apache.camel.builder.RouteBuilder;

public class GAuthRouteBuilder extends RouteBuilder {

    @Override
    public void configure() throws Exception {

        // Calback URL to redirect user from Google Authorization back to the web application
        String encodedCallback = URLEncoder.encode("https://gauth.example.org:8443/handler", "UTF-8");
        // Application will request for authorization to access a user's Google Calendar
        String encodedScope = URLEncoder.encode("http://www.google.com/calendar/feeds/", "UTF-8");

        // Route 1: A GET request to http://gauth.example.org/authorize will trigger the the OAuth
        // sequence of interactions. The gauth:authorize endpoint obtains an unauthorized request
        // token from Google and then redirects the user (browser) to a Google authorization page.
        from("jetty:http://0.0.0.0:8080/authorize")
            .to("gauth:authorize?callback=" + encodedCallback + "&scope=" + encodedScope);

        // Route 2: Handle callback from Google. After the user granted access to Google Calendar
        // Google redirects the user to https://gauth.example.org:8443/handler (see callback) along
        // with an authorized request token. The gauth:access endpoint exchanges the authorized
        // request token against a long-lived access token.
        from("jetty:https://0.0.0.0:8443/handler")
            .to("gauth:upgrade")
            // The access token can be obtained from
            // exchange.getOut().getHeader(GAuthUpgradeBinding.GAUTH_ACCESS_TOKEN)
            // The access token secret can be obtained from
            // exchange.getOut().getHeader(GAuthUpgradeBinding.GAUTH_ACCESS_TOKEN_SECRET)
            .process(/* store the tokens in context of the current user ... */);
    }

}
```

The OAuth sequence is triggered by sending a GET request to `http://gauth.example.org/authorize`. The user is then redirected to a Google authorization page. After having granted access on this page, Google redirects the user to the web application which handles the callback and finally obtains a long-lived access token from Google.

These two routes can perfectly co-exist with any other web application framework. The framework provides the basis for web application-specific functionality whereas the OAuth service provider integration is done with Apache Camel. The OAuth integration part could even use resources from an existing servlet container by using the `servlet` component instead of the `jetty` component.

What to do with the OAuth access token?

- Application should store the access token in context of the current user. If the user logs in next time, the access token can directly be loaded from the database, for example, without doing the *OAuth dance* again.
- The access token is then used to get access to Google services, such as a Google Calendar API, on behalf of the user. Java applications will most likely use the GData Java library for that. See below for an example how to use the access token with the GData Java library to read a user's calendar feed.
- The user can revoke the access token at any time from his Google Accounts page. In this case, access to the corresponding Google service will throw an authorization exception. The web application should remove the stored access token and redirect the user again to the Google authorization page for creating another one.

The above example relies on the following component configuration.

```
<bean id="gauth" class="org.apache.camel.component.gae.auth.GAuthComponent">
    <property name="consumerKey" value="anonymous" />
    <property name="consumerSecret" value="anonymous" />
</bean>
```

If you don't want that Google displays a warning message on the authorization page, you'll need to register your web application and change the `consumerKey` and `consumerSecret` settings.

GAE example

To OAuth-enable a Google App Engine application, only some small changes in the route builder are required. Assuming the GAE application hostname is `camelcloud.appspot.com` a configuration might look as follows. Here, the ghttp component is used to handle HTTP(S) requests instead of the `jetty` component.

GAuthRouteBuilder

```
import java.net.URLEncoder;
import org.apache.camel.builder.RouteBuilder;

public class TutorialRouteBuilder extends RouteBuilder {

    @Override
    public void configure() throws Exception {

        String encodedCallback = URLEncoder.encode("https://camelcloud.appspot.com/handler", "UTF-8");
        String encodedScope = URLEncoder.encode("http://www.google.com/calendar/feeds/", "UTF-8");

        from("ghttp:///authorize")
            .to("gauth:authorize?callback=" + encodedCallback + "&scope=" + encodedScope);

        from("ghttp:///handler")
            .to("gauth:upgrade")
            .process(/* store the tokens in context of the current user ... */);
    }

}
```

Access token usage

Here's an example how to use an access token to access a user's Google Calendar data with the GData Java library. The example application writes the titles of the user's public and private calendars to `stdout`.

Access token usage

```
import com.google.gdata.client.authn.oauth.OAuthHmacSha1Signer;
import com.google.gdata.client.authn.oauth.OAuthParameters;
import com.google.gdata.client.calendar.CalendarService;
import com.google.gdata.data.calendar.CalendarEntry;
import com.google.gdata.data.calendar.CalendarFeed;

import java.net.URL;

public class AccessExample {

    public static void main(String... args) throws Exception {
        String accessToken = ...
        String accessTokenSecret = ...

        CalendarService myService = new CalendarService("exampleCo-exampleApp-1.0");
        OAuthParameters params = new OAuthParameters();
        params.setOAuthConsumerKey("anonymous");
        params.setOAuthConsumerSecret("anonymous");
        params.setOAuthToken(accessToken);
        params.setOAuthTokenSecret(accessTokenSecret);
        myService.setOAuthCredentials(params, new OAuthHmacSha1Signer());

        URL feedUrl = new URL("http://www.google.com/calendar/feeds/default/");
        CalendarFeed resultFeed = myService.getFeed(feedUrl, CalendarFeed.class);

        System.out.println("Your calendars:");
        System.out.println();

        for (int i = 0; i < resultFeed.getEntries().size(); i++) {
            CalendarEntry entry = resultFeed.getEntries().get(i);
            System.out.println(entry.getTitle().getPlainText());
        }
    }
}
```

Dependencies

Maven users will need to add the following dependency to their `pom.xml`.

pom.xml

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-gae</artifactId>
    <version>${camel-version}</version>
</dependency>
```

where `${camel-version}` must be replaced by the actual version of Camel (2.3.0 or higher).

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started