

Sign Live! cloud suite gears

wp security

August 2024

intarsys GmbH

Sign Live! cloud suite gears

wp security

Version 8.13

cloud suite gears

intarsys GmbH
Sign Live! cloud suite gears wp security
Version 8.13

All rights reserved
© 2019 intarsys GmbH
www.intarsys.de

Preface

- Author and company

This book has been provided by different authors from the development staff of intarsys GmbH.

- Trademarks

Wherever possible and where the authors were aware of a trademark claim, such designations are marked as trademarks in this book.

jPod is a trademark of intarsys consulting.

Sun, Java and JavaScript are trademarks of Oracle.

Microsoft and Windows are trademarks of Microsoft Corporation.

- Who should read this book

This book provides both an overview of the product design and architecture and a reference for using the components and services.

So, this is the document for architects, developers and operators.

- Reviews and comments

We make constant efforts to improve our documentation and meet your requirements. Your comments are welcome and are a valuable resource for us.

Email support@intarsys.de

Website www.intarsys.de

Contents

Preface	5
▪ Author and company	5
▪ Trademarks	5
▪ Who should read this book	5
▪ Reviews and comments	5
Contents	6
1. Overview	10
2. Concepts	11
2.1 Security environment	11
2.2 Object capability	11
2.3 Principal	11
2.4 Identification	11
2.5 Authentication	12
2.6 Authorization	12
2.7 Security realm	12
3. Gears mechanisms	13
3.1 Web API	13
3.1.1 Peer to Peer	13
3.1.2 Browser (SPA)	14
3.2 Sessions	14
3.3 Flow/conversation	14
3.4 Principal	14
3.5 Spring security	16
3.6 Principal provider	17
3.6.1 Anonymous (no) principal	17
3.6.2 Static principal	17
3.6.3 Explicit principal	18
3.6.4 Spring security principal	18
3.7 Principal lookup	18
3.7.1 String expansion	18

4. Protocol security	20
4.1 Transport security	20
4.1.1 Server-side TLS, native Tomcat	20
4.1.2 Server-side TLS, Spring boot	22
4.1.3 Client TLS, native Tomcat	22
4.1.4 Client TLS, Spring boot	23
4.1.5 Client implementation	24
5. Browser security	25
5.1 Overview	25
5.2 Subdomains	25
5.3 Headers	25
5.3.1 Content-Security-Policy	25
5.3.2 Referrer policy	26
5.3.3 Strict-Transport-Security	26
5.3.4 X-Content-Type-Options	26
5.3.5 X-XSS-Protection	26
5.3.6 X-Frame-Options	26
5.4 Cookies	27
5.5 Session fixation	27
5.6 CORS	27
5.7 CSRF (XSRF)	28
5.8 XSS	28
6. Scenarios	29
6.1 BasicAuth, SSCD signature	29
6.2 BasicAuth, "anonymous" signature	29
6.2.1 Gears device configuration	29
6.2.2 Spring configuration	29
6.2.3 Gears principal configuration	30
6.2.4 Demo application	30
6.3 BasicAuth, individual signature	31
6.3.1 Gears device configuration	32
6.3.2 Spring configuration	32
6.3.3 Gears principal configuration	32
6.3.4 Demo application	33
6.4 OAuth2, individual signature	34
6.4.1 Gears device configuration	35
6.4.2 Spring configuration	35
6.4.3 Gears principal configuration	38
6.4.4 Demo application	39
6.5 X.509, individual signature	39
6.5.1 Gears device configuration	40
6.5.2 Spring configuration	40
6.5.3 Gears principal configuration	41
6.5.4 Demo application	42

Overview

6.6 Any authentication, attached viewer	42
6.7 Discard internal security definition	43
7. External References	46

1. Overview

This book is a collection of mechanisms and best practices around the security concepts that apply when running a gears instance.

On one hand these are mostly orthogonal concepts to gears built-in features and as such not part of the product but part of a company strategy and infrastructure.

On the other hand, the available scenarios for integration into existing infrastructure and the impact on security often result in similar solutions that can be reused among clients.

2. Concepts

2.1 Security environment

A security environment defines exactly the conditions of (valid) use for a security relevant product. In our case this is a combination of the security environment for the signature creation device which is used behind the gears services and the gears security environment itself.

This paper and the techniques explained replace in no way the security environments that are defined for the signature creation devices used.

It is an extension to this description that helps to understand how to fulfill the requirements described in the respective papers and how to integrate in an existing system and company infrastructure.

2.2 Object capability

Capability-based security is a well-known concept where a capability is a "communicable and unforgeable token of authority". Well, this definition from Wikipedia is quite dense but says it all...

Most of the security of gears services is capability-based: You get a random unguessable short-lived token to handle your process – the conversation id.

It is further important that this token is not communicated in cookies – it is always a service argument.

2.3 Principal

Principal denotes any party involved into a process with gears. Most often it is assumed to be a person, but can be an application, company or whatever.

From the gears manual:

Typical influence of a principal on the service execution or outcome:

- the user principal provides claims that are used in string expansion
- the user principal defines a profile whose properties enhance service arguments
- the client principal is billed for the service execution

2.4 Identification

Identification in our context simply denotes how an application collects information about the participating parties.

This can be as simple as getting a username from a calling client.

Your concrete usage scenario determines if identification of a principal is sufficient.

2.5 Authentication

Authentication is about getting evidence that a claim of identity is really true.

Classic example is a password. In this example an application confirms that a shared secret is known by both parties. If both parties are able to keep the secret, the application can assume the identity claim is correct.

2.6 Authorization

Authorization is about the decision if an (authenticated) user is allowed to access a certain resource.

In gears several resources can be access controlled.

2.7 Security realm

A gears security realm is a collection of services sharing a common security configuration.

There are currently three realms of interest (see [1]) for gears

- "Flow"
The gears "business functions" like viewer or signer
- "Control"
The gears operating console services (control gears runtime objects like pools etc.)
- "Manage"
The gears management console services (collecting production relevant data, Spring actuator)

Each of the realms can be configured individually.

3. Gears mechanisms

3.1 Web API

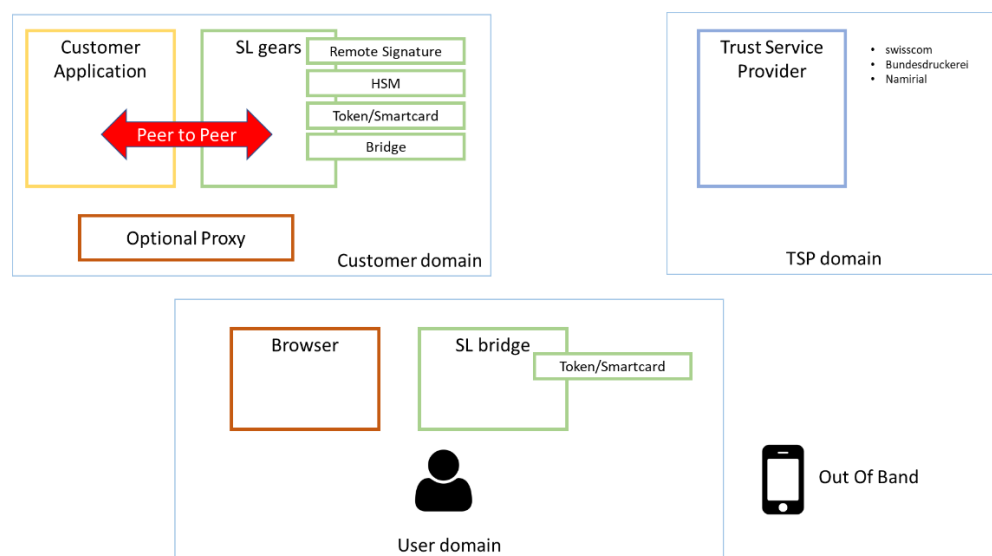
Even if obvious, we want to emphasize that we are talking of a public web service API, not a web application with a 1:1 internal API.

This has some impact on the attack surface and on the restrictions and security measures that can be applied.

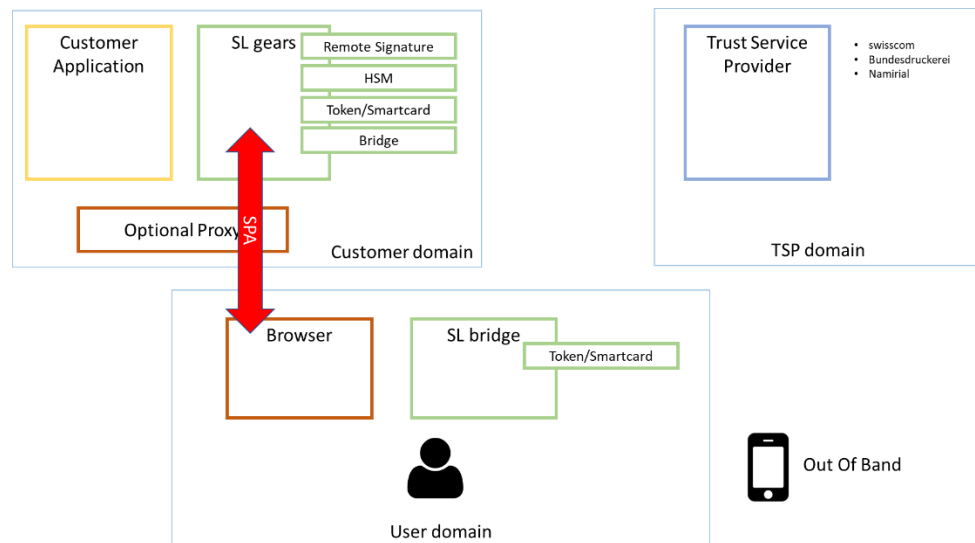
The API is stateless. If authentication is required, the credentials are always sent along with the request.

3.1.1 Peer to Peer

When we look at the reference sheet for a gears architecture, we talk about communication between two backend systems – shown in the upper left.



3.1.2 Browser (SPA)



3.2 Sessions

Gears does not use HTTP sessions at all.

By default, all Spring security realms are configured to be stateless, too. This means that authentication information has to be provided with each and every request. This is standard REST style.

If for some reason you need session-based security, you must redefine the security realm.

3.3 Flow/conversation

A flow is a "mini session" that handles the interaction between the user and the flow resources (e.g., a document). It is technically implemented on the basis of the "conversation" concept. You can use the terms interchangeably here.

It is based on "object capability security" as described above. Using the short-lived conversation id, you have automatically the right to manipulate the flow using its services.

E.g., having the conversation id for a viewer gives you the right to request a rendered page of the contained document.

The flow is associated with the user principal that represents the entity on behalf of which all requests are handled.

If a flow launches another flow (a viewer starts a signer), the principal is propagated to the child conversations.

3.4 Principal

The principal and its use in gears is described in [1]. Maybe you should revisit this section for the technical details.

Here we summarize the important parts and bring them in a context to other non-gears related concepts.

Gears supports three types of principals:

- a "tenant" represents a customer or organizational unit
(urn:intarsys:names:principal:1.0:role:Tenant)
- a "client" represents an application
(urn:intarsys:names:principal:1.0:role:Client)
- a "user" represents a single entity within the client world,
eventually holding a private key
(urn:intarsys:names:principal:1.0:role:User)

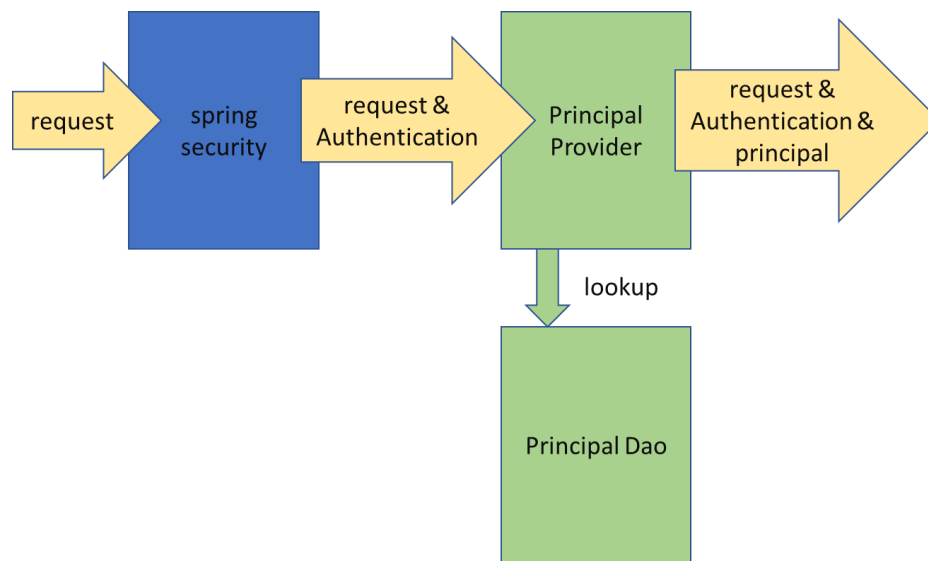
This paper is concerned with the identification and authentication of a user, so we ignore the "tenant" and "client" principals from here.

The "user" principal represents the entity on whose behalf the services are executed, e.g., a signature or a viewer is requested for a document of the user. He's the owner of the document and the one who is authorized to apply a signature on it.

A principal can be created from many sources – most of the scenarios here are concerned with deriving a principal from authentication information.

A principal is assigned to a flow (and therefore part of the "object capability" expressed by the conversation id). Unless stated otherwise, the principal is "inherited" if such a flow creates another flow, like for example a viewer that performs a signature (**principal propagation**).

Let's revisit the processing of a request w.r.t. the user principal:



For each incoming request

- the optional Spring security components ensure authentication
- this results in a contextual **Authentication** object
- a gears principal provider will
 - select raw information according to its strategy from the request context
 - optionally lookup associated model information using a principal DAO
 - link the request with the derived principal

Now we have a user principal in the context that can be used in the gears backend processing for

- profile selection
- key selection
- argument expansion

3.5 Spring security

Spring security is a general industry strength framework for securing web services.

We will not go into details of configuration or implementation but will highlight the most important features

- standard industry proven implementations
- Open Source
- pluggable implementations for authentication protocols
 - BasicAuth
 - OpenId Connect
 - other token-based authentication protocols

- ...
- pluggable implementations for data integration
 - JDBC
 - LDAP
 - ...
- configurable handling of well-known security issues
 - CORS
 - HTS
 - XSRF
 - ...
- completely independent of web service implementation

The Spring components add the required security context to the request, regardless of the protocol chosen and ensure industry best practices for the protocol handling.

3.6 Principal provider

The principal provider ensures the gears request has always a principal in its context.

The provider is pluggable but there exist default implementations for the common scenarios.

3.6.1 Anonymous (no) principal

There are scenarios where we simply **need** no user principal in the request (we may still have one for authorization purposes), e.g.

- signature on local token (knowledge and possession)
- signature with a seal (not user specific)

Still these scenarios are totally different from a security point of view.

In one case the request needs no authentication because security is implemented completely out-of-band – the user holds a token in his possession and usage of the key is secured by a token specific protocol (e.g., by at least a PIN).

The other case may still need authentication (depending on the TSP and security environment), but on tenant or client level. To achieve this, you can still apply the same techniques described in this white paper.

3.6.2 Static principal

A principal with a certain role can be statically configured. This allows for example simple distinction between different running gears instances or reuse of configuration components in different context.

3.6.3 Explicit principal

Here we filter the principal from web service arguments. The principal is sent literally with the request payload, either by reference or by value.

When sent by reference, the principal DAO is used to look up the complete principal instance. By value we simply deserialize the principal instance from the request.

This provides the "identification" part of the task described in this white paper.

The principal information at this point is unauthenticated as far as the principal itself is concerned. It is intentionally a pure service argument, not part of an authentication protocol.

This is a very weak assumption on the principal executing the transaction.

You can use this in scenarios comparable to the "no principal" case and whenever the targeted TSP backend itself has a complete authentication in place. You only pre-select the backend entity (the key) in this case.

3.6.4 Spring security principal

Here we rely on information collected from the authentication in the Spring security layer.

We have for example a username that is expanded to a principal instance with a database lookup, a token with a set of claims that are mapped directly to a principal or a X509 client certificate that is converted to a principal.

3.7 Principal lookup

After the infrastructure described above has built up principal information for each request, what do we do with it?

While this part of the process is not in the scope of this paper, we give a short summary of the principal usage in the final stages of request processing.

It is important to know that there's **no** built-in processing on behalf of gears itself. The information is nowhere used explicitly. It is made available to different plugins that allow to contextualize the processing.

3.7.1 String expansion

The principal context can be accessed everywhere where dynamic string expansion is in effect. An example of accessing principal data is

```
${principal.user.name}
```

or

```
${principal.user.claims.msisdn}
```

This information is available wherever strings are expanded dynamically.

4. Protocol security

4.1 Transport security

This paper assumes that any request to gears services is either in a known secure environment or has transport security (TLS) in place.

Any other use is out of scope.

As the name implies, transport security is not part of the gears application, it is provided by your infrastructure components and as such depends on the environment.

In the following chapters we will give a jump-start on how to set up such a scenario. Example keystore files for client and server are included in the "example" folder.

4.1.1 Server-side TLS, native Tomcat

Detailed information on Tomcat SSL configuration is available at <https://tomcat.apache.org/tomcat-8.0-doc/ssl-howto.html>.

We assume you already have a valid server SSL key and certificate available as a Java keystore. There's a plethora of tutorials on this topic available, both for self-signed and production scenarios (e.g., here <https://www.mulesoft.com/tcat/tomcat-ssl>).

All you need to do is add a connector declaration for an SSL listener on port 8443 (or whatever port you desire). This fragment has to be included into the "server.xml".

tomcat XML configuration fragment (JKS)

```
<Connector
  protocol="org.apache.coyote.http11.Http11NioProtocol"
  port="8443"
  maxThreads="200"
  scheme="https"
  secure="true"
  sslProtocol="TLS"
  SSLEnabled="true"
  keystoreFile="conf/demo-server-key.jks"
  keystoreType="JKS"
  keystorePass="server"
/>
```

tomcat XML configuration fragment (PKCS#12)

```
<Connector
...
  keystoreFile="conf/demo-server-key.pfx"
  keystoreType="PKCS12"
...
/>
```

If you copy the "demo-server-key.jks" from the "example/TLS" folder to the tomcat "conf" directory, you can use this out of the box.

Be sure to choose the same password for the key and the keystore, tomcat cannot access the key otherwise. After restart, you should be able to contact the server via SSL.

If you want to use this configuration along with the demo app, be sure to configure the demo correctly:

- Use the https protocol to talk to the gears server
- Add the "demo.properties" required for TLS (see "demo/example/configuration")
- Add the spring-demo-tls.xml to effectively disable hostname verification

Assuming you have connectors for http and https you force Tomcat to redirect http requests to https by adding the following security-constraint to conf\web.xml:

tomcat XML configuration fragment

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Entire Application</web-resource-name>
    <url-pattern>/*</url-pattern>
  </web-resource-collection>
  <!-- auth-constraint goes here if you require authentication -->
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>
```

4.1.2 Server-side TLS, Spring boot

Gears comes with the option to run as a standalone application with an embedded tomcat. Here's what to do to switch on TLS in this case.

spring properties

```
server.ssl.enabled=true
server.ssl.key-store=${cloudsuite.config.shared}/demo-server-key.jks
server.ssl.key-store-type=JKS
server.ssl.key-store-password=server
server.ssl.key-password=server
```

The key-store property must point to the keystore – you can use any Spring resource identifier here. If you copy the demo-server-key.jks from the "example/TLS" folder to the gears config directory, you can use this out of the box.

Be aware that Spring does not use the classpath that includes your gears "config/classes" and "config/lib" directories for many of its internal lookup operations, including this.

There are still more options to discover for Spring TLS support, you will find them here <https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-application-properties.html#server-properties>.

If you want to use this configuration along with the demo app, be sure to configure the demo correctly:

- use the https protocol to talk to the gears server
- add the demo.properties required for TLS (see "demo/example/configuration")
- add the spring-demo-tls.xml to effectively disable hostname verification

4.1.3 Client TLS, native Tomcat

You can be even more restrictive in your setup by requiring the client to present a valid certificate, too.

tomcat XML fragment

```
<Connector
  protocol="org.apache.coyote.http11.Http11NioProtocol"
  port="8443"
  maxThreads="200"
  scheme="https"
  secure="true"
  sslProtocol="TLS"
  SSLEnabled="true"
  keystoreFile="conf/demo-server-key.jks"
  keystoreType="JKS"
  keystorePass="server"
  clientAuth="true"
  truststoreFile="conf/demo-client-cert.jks"
  truststoreType="JKS"
  truststorePass=""
/>
```

The setup is similar to the one before but it is cumbersome to add a truststore containing the trusted certificates (or the CA certificates) for the clients.

If you want to use this configuration along with the demo app, be sure to configure the demo correctly:

- use the https protocol to talk to the gears server
- add the demo.properties required for TLS (see "demo/example/configuration")
- add the spring-demo-tls.xml to effectively disable hostname verification

4.1.4 Client TLS, Spring boot

Things are very similar when it comes to Spring boot

spring properties

```
server.ssl.enabled=true
server.ssl.key-store=${cloudsuite.config.shared}/demo-server-key.jks
server.ssl.key-store-type=JKS
server.ssl.key-store-password=server
server.ssl.key-password=server
# one of NEED | NONE | WANT
server.ssl.client-auth=NEED
server.ssl.trust-store=${cloudsuite.config.shared}/demo-client-cert.jks
server.ssl.trust-store-type=JKS
server.ssl.trust-store-password=
```

If you want to use this configuration along with the demo app, be sure to configure the demo correctly:

- use the https protocol to talk to the gears server
- add the demo.properties required for TLS (see "demo/example/configuration")
- add the spring-demo-tls.xml to effectively disable hostname verification

4.1.5 Client implementation

4.1.5.1 General

Accessing gears with TLS set up requires additional steps when executing the request. These steps naturally depend on your implementation language and environment.

For all scenarios the requirements and steps are basically the same:

- you have some factory that is responsible for creating a physical connection (transport layer)
- For server-side TLS
 - you provide a truststore with the server certificate or one of its CA certificates
 - password may or may not be required/supported – but these are public information anyway
 - You may have to provide a policy how to deal with the hostname (hostname verification). This is sometimes an issue in development scenarios. Do never forget to remove "accept all hostnames" policies from production code.
- For client-side TLS
 - you provide a keystore with the client key and certificate
 - you provide the required passwords for store / key

Now the runtime environment should take care of creating a valid TLS connection.

4.1.5.2 Client with JAX-RS

Here's example code that implements the above steps for Java JAX-RS. You can find a code snippet like this in the demo code that comes with gears.

Java code fragment

```
protected Client createTLSClient() {
    ClientBuilder builder = ClientBuilder.newBuilder();
    KeyStore keystore = getSslKeyStore();
    String keystorePassword = getSslKeyPassword();
    builder = builder.keyStore(keystore, keystorePassword);
    KeyStore truststore = getSslTrustStore();
    builder = builder.trustStore(truststore);
    HostnameVerifier hostnameVerifier = getGearsCoreSslHostnameVerifier();
    builder = builder.hostnameVerifier(hostnameVerifier);
    return builder.build();
}
```

5. Browser security

5.1 Overview

This chapter deals with issues that are relevant only if you use UI components of gears (e.g. the viewer).

5.2 Subdomains

Many security vulnerabilities are based on the difficult rules how subdomains are treated in different areas (cookie sending, same-site policies, SSL certificates...).

We recommend running gears on domains only where the hosting domains and all subdomains are trusted to avoid security issues in corner cases.

5.3 Headers

5.3.1 Content-Security-Policy

This header allows detailed configuration of the security settings applied in a conformant browser.

This header is

```
Content-Security-Policy
default-src 'self';
style-src 'self' 'unsafe-inline' fonts.googleapis.com;
font-src 'self' fonts.gstatic.com;
img-src 'self' blob;;
object-src 'none';
frame-ancestors 'none';
```

for all Angular UI code.

For other UI code this may vary as the default uses the most restrictive ruleset possible

This is the header you most probably need to tweak if you have any special integration requirements as:

- Embedding the viewer in a frame
- Loading resources from other than the enumerated destinations
- Separating UI code delivery from the API

You should do so only if you really know what you are doing. For special scenarios come back to intarsys professional services.

5.3.2 Referrer policy

To avoid leaks based on publishing the URL via the browser “referrer” header, all UI code is decorated with

```
Referrer-Policy: no-referrer
```

5.3.3 Strict-Transport-Security

Strict-Transport-Security instructs a browser to allow HTTPS connections only.

This header is

```
Strict-Transport-Security: max-age=31536000 ; includeSubDomains
```

for all UI code **and** is set only when HTTPS is used.

5.3.4 X-Content-Type-Options

This option disables the browser auto-detect feature for content to prevent content injection that might be executed on the client.

This header is

```
X-Content-Type-Options: nosniff
```

for all UI code.

5.3.5 X-XSS-Protection

This option enables detection of XSS attacks for older browsers. For newer browsers it is superseded by “Content-Security-Policy”.

This header is

```
X-XSS-Protection: 1; mode=block
```

for all UI code.

5.3.6 X-Frame-Options

This header controls if the content may be embedded in an iFrame.

The header is

X-Frame-Options: DENY

for all UI code.

5.4 Cookies

No security relevant cookies are used with gears.

5.5 Session fixation

Session fixation attacks try to guess or inject session identifiers and apply to the gears capabilities (conversations are short lived sessions) in theory, too.

Gears does not allow client generated tokens. The server generated tokens are defined for a single document (object capability) and as such are not usable to gain access to other user resources.

5.6 CORS

Cross-Origin Resource Sharing (short CORS) deals with sending requests to a web service from the browser in web pages that are loaded from another domain. It is a concept that is relevant for browser requests (SPA) only and is implemented in all modern browsers.

From a cross-origin point of view, the gears services are public content, intended to be called from any other domain. After the creation of a flow, we have a capability-based security **without** storing the token in local data stores like cookies or browser storage. So only the creational services are subject to be attacked.

Special care must be taken when authentication is applied, though. This means that you decided for some reason your (creational) gears services should not be publicly available. Depending on your application and the type of authentication (HTTP session based) you have set up, you may run into issues now and may want to have a stricter CORS policy in place.

While the default CORS settings for **all** gears services is to allow everything, you can override this using the following properties.

security.cors.allowedHeaders	
list of strings	A ", " separated list of allowed headers. Default: *
security.cors.allowedMethods	
list of strings	A ", " separated list of allowed methods. Default: *
security.cors.allowedOrigins	

list of headers	A ", " separated list of allowed origins. Default: *
security.cors.exposedHeaders	
list of headers	A ", " separated list of allowed exposed headers. Default: *
security.cors.allowCredentials	
boolean	Flag if user credentials are allowed. Default: true

5.7 CSRF (XSRF)

Cross-Site Request Forgery is an attack where a user of a malicious web site or another client app that is able to submit web requests (mail client) is tricked into submitting a request with his cached credentials.

In our concrete case we could construct a scenario where

- You have set up a gears environment with a seal creation service (or another unattended signature creation).
- You have set up authentication that is persisted in a cookie store.
- A malicious site causes a signature creation request behind the scenes.

Gears, in compliance with RFC 2616, is designed to make state changing requests using POST only. The payload is always JSON encoded. In addition, all capability-based requests can be considered safe.

To set up additional counter measures in the above scenario we must revert to the authentication layer – which is Spring in this case. Gears has no notion of an "authenticated session" and as such cannot establish relationships between successive calls.

- csrf token / double submit cookie (encrypted, HMAC)
 - token storage cookie/local/DOM ???
- custom header
- user interaction-based defense (OTP, Captcha, ...)

5.8 XSS

Cross-Site Scripting is an attack that injects code provided by the attacker in sites displayed to the user.

Gears does sanitize user input respective web service input that gets rendered in terms of XSS attacks.

6. Scenarios

6.1 BasicAuth, SSCD signature

This is the most common scenario, where gears is a frontend to some 3rd party TSP with a secured signature creation device (SSCD). The identification and authentication process is not a required part of gears, as the TSP sets up a security layer himself.

You can set up and experiment without Spring security or with any of the scenarios below, but none will compromise the signature.

Identification & authentication may be necessary for your environment for other reasons (accounting etc.), though.

6.2 BasicAuth, "anonymous" signature

There are interesting cases where a key is shared by many users, for example a "seal". In this case the individual user does not have the key's credentials, access is granted through the gears gateway. For sure you have to take care that only authorized users can use this key, even so the user's identity is not used anywhere in the gears process.

For this use case we assume

- a seal is used for signing
- access to the signature service using the seal is required to be authenticated to prevent unauthorized use
- BasicAuth is used
- no session is held on the server

6.2.1 Gears device configuration

To simplify matters, we use the default@demo device without a **signerIdentifier**. This simulates signature creation without an additional authentication step on the signature creation side.

6.2.2 Spring configuration

To set up BasicAuth you need a simple Spring configuration: we define the **securityRealmFlowAuthenticationFilter**, as explained in [1]

spring XML fragment

```
<bean
  id="securityRealmFlowAuthenticationFilter"
  class="org.springframework.security.web.authentication.www.BasicAuthenticationFilter">
  <constructor-arg ref="securityRealmFlowAuthenticationManager" />
</bean>
```

To simplify things, we leave the default authentication manager in place. It is backed by a simple in-memory data store holding a user with id/password "user"/"user".

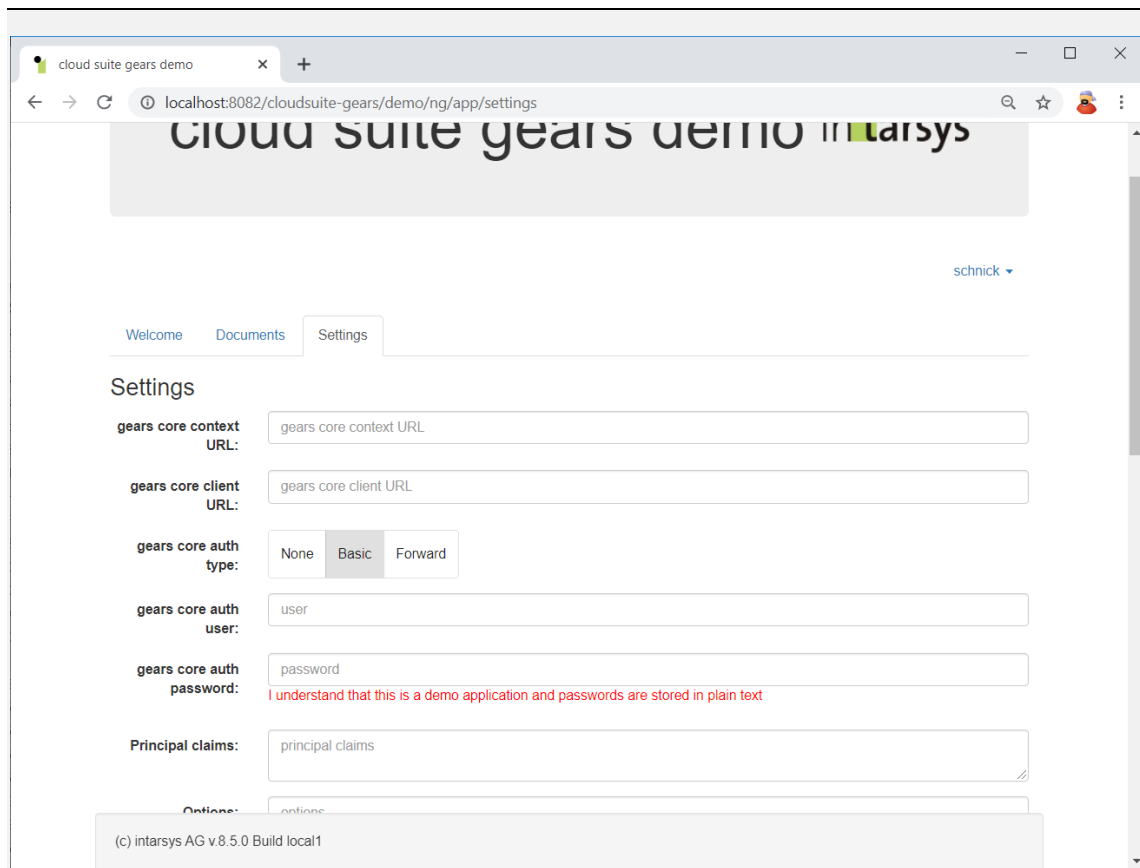
6.2.3 Gears principal configuration

In this example with a shared resource we do not make any use of a user principal, so we can get away with ignoring it.

The default (explicit principal provider) will be active, so that a request can carry any user principal information, but it will simply not be used.

6.2.4 Demo application

The demo application is built using a "Peer to Peer" architecture, so the request is sent by the demo backend, not the browser. What credentials are used can be entered on the settings page.



Set the "auth type" to "Basic" and enter the credentials for gears ("user"/"user" if you are still working with our defaults).

Now when calling "signer/create" or "viewer/create" the credentials are sent to gears. As a result, we receive a flow which is a self-contained "object capability" and further calls need not be authenticated.

The principal was derived with the first, authenticated call and is referenced by the flow.

6.3 BasicAuth, individual signature

In this scenario we need access to some user information to select the correct entity / key in the signature backend. The official way to do so is relying on the user principal.

While we still could use an explicit principal (one that is supplied in the request payload) this would collide with the original idea of the authenticated request. The explicit principal has "identification" qualities only.

If you want to separate these two (authenticated user from the protocol level and explicit user principal from the request) because of your business requirements, you can simply do so (proceed from the scenario "BasicAuth seal signature").

For our scenario we want to couple the both. This means that we can be sure that user principal information we access in the ongoing process is authentic and "owned" by the authenticated protocol level user.

6.3.1 Gears device configuration

We use a device that has individual keys for the user principals. Again, for test purposes the simplest solution is using the "default@demo" device, this time with a **signerIdentifier** argument. Instead of the global default identity a new identity for each individual principal is created automatically.

6.3.2 Spring configuration

The Spring configuration is the same as above, simple BasicAuth.

spring XML fragment

```
<bean
  id="securityRealmFlowAuthenticationFilter"
  class="org.springframework.security.web.authentication.www.BasicAuthenticationFilter">
  <constructor-arg ref="securityRealmFlowAuthenticationManager" />
</bean>
```

To have some more test users we change the Spring authentication manager, too (be sure to have the "security" XML namespaces added to your beans.xml).

spring XML fragment

```
<security:authentication-manager id="securityRealmFlowAuthenticationManager">
  <security:authentication-provider>
    <security:user-service>
      <security:user name="user" password="{noop}user" authorities="ROLE_USER" />
      <security:user name="foo" password="{noop}bar" authorities="ROLE_USER" />
      <security:user name="schnick" password="{noop}schnack" authorities="ROLE_USER" />
    </security:user-service>
  </security:authentication-provider>
</security:authentication-manager>
```

6.3.3 Gears principal configuration

We want the principal to be derived from Spring security, so we use the **SpringSecurityPrincipalProvider**.

spring XML fragment

```
<bean id="principalProviderUser"
  class="de.intarsys.cloudsuite.gears.security.spring.\
SpringSecurityPrincipalProvider">
  <property name="role" value="urn:intarsys:names:principal:1.0:role:User" />
  <property name="keyConverter">
    <bean class="de.intarsys.cloudsuite.gears.security.spring.\
AuthenticationToStringConverter"/>
  </property>
  <property name="principalDao" ref="modelPrincipalDao" />
</bean>
```

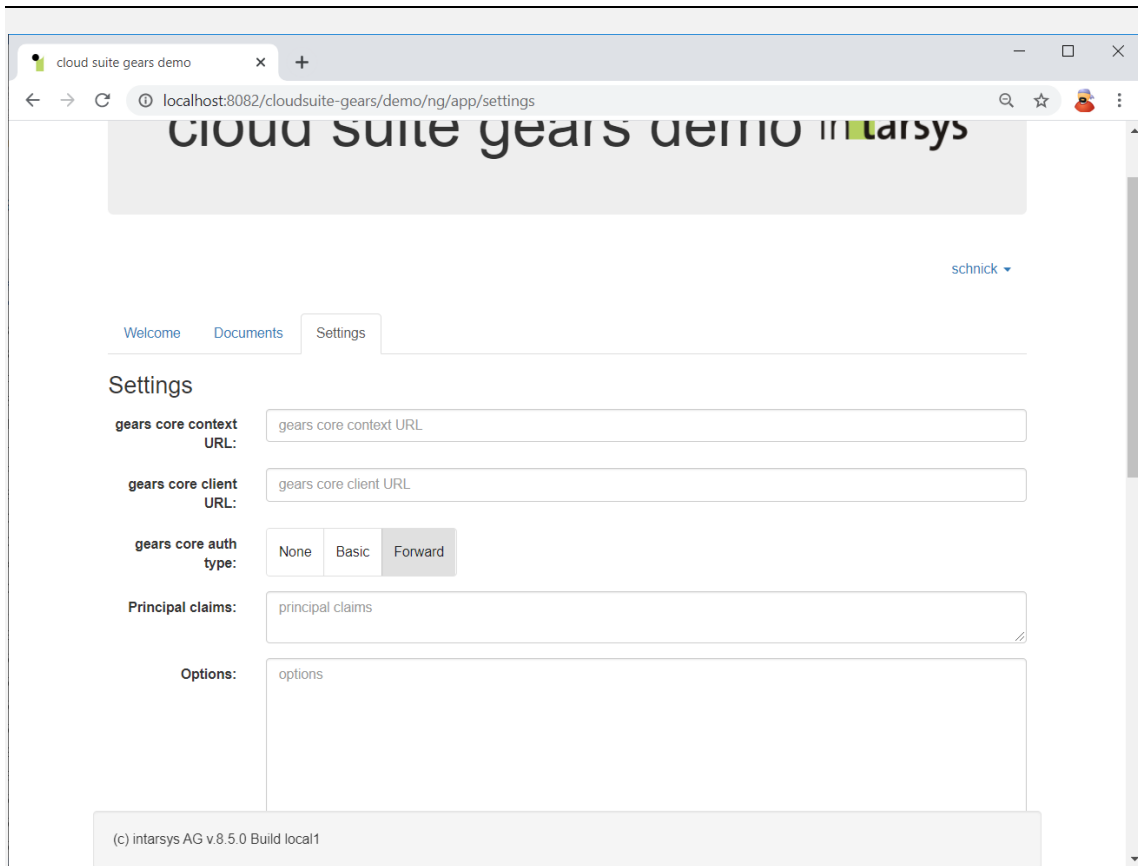
Base of this principal provider is the Spring authentication. In this example, we convert simply to a string (which is the user name) and use this as the key into a standard principal DAO.

To keep the complexity down, we choose to stay with the default DAO. You can choose from any supported data source here, like in-memory, JDBC and so on.

6.3.4 Demo application

If you keep the settings from above, each request would be authenticated with "user"/"user", so each signature would be created by a principal "user" (the authenticated one).

Change the demo authentication settings to "Forward". Now the user/password combination that has been used for demo authentication will be forwarded to gears. Note that this forwarding works only for user/password logon (as we have Basic Auth in place with gears).



To get the principal into the scope you add a corresponding parameter to the "signer arguments" in the settings:

demo settings signer args

```
{
  "documentSigner": {
    "args": {
      "digestSigner": {
        "args": {
          "signerIdentifier": "${principal.user.name}"
        }
      }
    }
  }
}
```

Now you can log on with any one of "user", "foo" or "schnick" and get a signature on your document that claims to be originated by one of these users.

If you log on to demo with another user, the request will fail.

6.4 OAuth2, individual signature

Now we try to combine an OAuth2 token-based authentication with an authenticated principal.

6.4.1 Gears device configuration

For this example, we use again the "default@demo" device with a **signerIdentifier** argument.

6.4.2 Spring configuration

The OAuth2 features of Spring are very manifold and as such require careful reading of the Spring security documentation. What we present here is only a small subset of the Spring features available.

Here we go for a simple token authentication against the Google ID provider.

Now we will split apart the different components that make up authentication.

6.4.2.1 The filter

First, we need a filter that guards the request. This time it is a token-based mechanism.

spring XML fragment

```
<bean id="securityRealmFlowAuthenticationFilter"
      class="org.springframework.security.oauth2.server.resource.web.\
BearerTokenAuthenticationFilter">
  <constructor-arg ref="securityRealmFlowAuthenticationManager" />
</bean>
```

The **BearerTokenAuthenticationFilter** is what we need. We already know the reference to an authentication manager. This is the component that can decide upon the authenticity of the token and assigns the "grants". It is required here, too, but we will need to set up another object for this purpose later.

You can configure the strategy of how to resolve the token from the request, too, but for this example we stick with the default and simply use the "Authentication" HTTP header.

6.4.2.2 The authentication manager

Let's work backward – we need an authentication manager. Spring has a default implementation for token-based authentication that constructs an authentication context solely based on the token and its claims itself.

spring XML fragment

```

<bean id="jwtAuthenticationProvider"
  class="org.springframework.security.oauth2.server.resource.authentication.\
JwtAuthenticationProvider">

  <constructor-arg ref="jwtDecoder" />

  <property name="jwtAuthenticationConverter">
    <bean class="org.springframework.security.oauth2.server.resource.authentication.\
JwtAuthenticationConverter">
      <property name="jwtGrantedAuthoritiesConverter">
        <bean class="de.intarsys.spring.security.JwtStaticGrantedAuthoritiesConverter">
          <property name="authorities" value="ROLE_USER" />
        </bean>
      </property>
    </bean>
  </property>
</bean>

<security:authentication-manager id="securityRealmFlowAuthenticationManager">
  <security:authentication-provider ref="jwtAuthenticationProvider" />
</security:authentication-manager>

```

The first component that is used by the **JwtAuthenticationProvider** is the **jwtDecoder**. This is the component that is responsible for the token parsing and validation according to the OAuth2 specs. We will come to this soon.

The **JwtAuthenticationConverter** takes the result of parsing and constructs a Spring internal object, along with the authorities (or roles) that the authentication carries with it.

As our Google token does not imply any roles, we have provided a static definition (**JwtStaticGrantedAuthoritiesConverter**) that simply assigns the "ROLE_USER" to any authenticated token.

In contrast, if you use a service that also sends a user's roles, such as a self-configured Keycloak instance, you use the **JwtGrantedAuthoritiesByRolesConverter**. This can be used to define where the roles are located in the token and how they may be converted, e.g. using a prefix or uppercase.

You see that there's a lot of possibilities to inject your own policies into the Spring hot spots and it all depends on your own security architecture and role model.

6.4.2.3 The introspection

In case of an opaque token, the token validation must be performed by the issuing authorization server using OAuth 2.0 introspection. The following example illustrates how to integrate token introspection into your authentication provider and thus query an authorization server (e.g. Keycloak) to verify the active state of the provided token. We need an **OpaqueTokenIntrospector** to send the request.

Spring Security offers the **SpringOpaqueTokenIntrospector**. The information received through successful introspection of the token is processed by an **AuthenticationProvider**, in this case the

OpaqueTokenAuthenticationProvider, which creates an Authentication instance that represents the token bearer's authentication information.

spring XML fragment

```
<bean id="keycloakTokenIntrospector"
class="org.springframework.security.oauth2.server.resource.introspection.SpringOpaqueTokenIntrospector">
    <constructor-arg index="0" type="java.lang.String"
value="KEYCLOAK_INTROSPECTION_ENDPOINT"/>
    <constructor-arg index="1" type="java.lang.String" value="KEYCLOAK_CLIENT_ID"/>
    <constructor-arg index="2" type="java.lang.String" value="KEYCLOAK_CLIENT_SECRET"/>
</bean>

<bean id="introAuthenticationProvider"
class="org.springframework.security.oauth2.server.resource.authentication.OpaqueTokenAuthenticationProvider">
    <constructor-arg ref="keycloakTokenIntrospector" />
</bean>
```

6.4.2.4 The decoder

The decoder has the task of parsing and validating the token with regard to the OAuth2 spec and the token issuer. This is a complex task, but it is completely hidden behind some simple configuration properties.

The token decoder responsibilities are

- syntactical check (spec conformity)
serialization format & completeness
- authenticity
can we confirm the authenticity of the issuer (signature)
- semantic check
the authentic token needs to fulfill additional predicates to authorize our implementation

The first one is completely internal and we are fine.

The second step requires at least that we provide information what we consider to be a trustworthy identity provider. We can do this by supplying the decoder factory with information on the key material that we have trust in. For Google this means that we provide a "jwk set uri" – a location where we can collect the official public keys for the Google ID provider. Other possibilities would be to hardcode a public key or a symmetric key.

The semantic check can now ensure that the claims have a certain content. By default the following checks are made:

- token not expired
- token issuer (if issuer configured)

Other validation predicates can be added.

spring XML fragment

```
<bean id="jwtDecoder" class="de.intarsys.spring.security.JwtDecoderFactoryBean">
  <property name="jwkSetUri" value="https://www.googleapis.com/oauth2/v3/certs" />
</bean>
```

6.4.3 Gears principal configuration

We want the principal to be derived from Spring security, so we use the **SpringSecurityPrincipalProvider**.

Regarding the **principalDao**, we could stick with the same configuration as with basic auth:

spring XML fragment

```
<bean id="principalProviderUser"
  class="de.intarsys.cloudsuite.gears.security.spring.\
SpringSecurityPrincipalProvider">
  <property name="role" value="urn:intarsys:names:principal:1.0:role:User" />
  <property name="keyConverter">
    <bean class="de.intarsys.cloudsuite.gears.security.spring.\
AuthenticationToStringConverter"/>
  </property>
  <property name="principalDao" ref="modelPrincipalDao" />
</bean>
```

This means that we convert the Spring authentication data to a key and then lookup principal information in our DAO. For the token authentication case this means that

- Spring derives the "authentication name" from the "sub" claim
- we derive a principal from the authentication name

In the Google case this is a "random" big integer that most probably is not useful with the default configuration where it is used literally as the principal.

To improve matters, we could implement a lookup (in memory or in a database) to attach our own profile data to the "sub" key.

An alternative would be to use the token claims itself immediately to fill the principal information. No key conversion is applied.

spring XML fragment

```
<bean id="principalProviderUser"
  class="de.intarsys.cloudsuite.gears.security.spring.\
SpringSecurityPrincipalProvider">
  <property name="role" value="urn:intarsys:names:principal:1.0:role:User" />
  <property name="principalDao">
    <bean class="de.intarsys.cloudsuite.gears.security.spring.\
JwtPrincipalDao" />
  </property>
</bean>
```

This configuration copies over all claims to the authenticated principal, the name will be the "sub" claim.

6.4.4 Demo application

To access the token authenticated gears services, you can use the demo application, too.

Change the demo authentication settings to "Forward". Any authentication information will be forwarded to gears.

In this case, you must use the Google authentication module as the gears backend cannot deal with user / password in this configuration.

You can access the authenticated principal for use in the demo signer by adding a corresponding parameter to the "signer arguments" in the demo settings page:

demo settings signer args

```
{
  "documentSigner": {
    "args": {
      "digestSigner": {
        "args": {
          "signerIdentifier": "${principal.user.name}"
        }
      }
    }
  }
}
```

or even better (with a Google token)

demo settings signer args

```
{
  "documentSigner": {
    "args": {
      "digestSigner": {
        "args": {
          "signerIdentifier": "${principal.user.claims.email}"
        }
      }
    }
  }
}
```

6.5 X.509, individual signature

A very high level of security can be achieved when TLS with client authentication is set up on transport level.

Spring security extracts this pre-authenticated information and makes it available as a Spring authentication.

6.5.1 Gears device configuration

For this example, we use again the "default@demo" device with a **signerIdentifier** argument.

6.5.2 Spring configuration

X.509 authentication can be customized in many variations and extensions based on your security architecture are normally quite easy.

Now we will split apart the different components that make up authentication.

6.5.2.1 The filter

As always, we start with the filter, this time based on the transport level client certificate.

spring XML fragment

```
<bean id="securityRealmFlowAuthenticationFilter"
  class="org.springframework.security.web.authentication.preauth.x509.\
X509AuthenticationFilter">
  <property name="authenticationManager" ref="securityRealmFlowAuthenticationManager" />
  <property name="authenticationDetailsSource">
    <bean class="de.intarsys.spring.security.StaticAuthenticationDetailsSource">
      <property name="grantedAuthorities" value="ROLE_USER" />
    </bean>
  </property>
  <property name="principalExtractor">
    <bean class="org.springframework.security.web.authentication.preauth.x509.\
SubjectDnX509PrincipalExtractor" />
  </property>
</bean>
```

The **X509AuthenticationFilter** is provided for this use case. We already know the reference to an **authenticationManager**. This is the component that can decide upon the authenticity of the token and assigns the "grants". It is required here, too, but we will need to set up a special certificate aware object for this purpose later.

An **authenticationDetailsSource** provides a component that defines the Spring authorities that are associated with the request. In this case we inject a static list of authorities.

The **principalExtractor** converts the X509Certificate to whatever principal representation you want – in this case the subject CN.

6.5.2.2 The authentication manager

The authentication manager is a specialized version for pre authentication cases. The pre authentication token is extracted by the filter and forwarded.

spring XML fragment

```
<bean id="x509AuthenticationProvider"
  class="org.springframework.security.web.authentication.preauth.\
PreAuthenticatedAuthenticationProvider">
  <property name="preAuthenticatedUserDetailsService">
    <bean
      class="org.springframework.security.web.authentication.preauth.\
PreAuthenticatedGrantedAuthoritiesUserDetailsService">
    </bean>
  </property>
</bean>

<security:authentication-manager id="securityRealmFlowAuthenticationManager">
  <security:authentication-provider ref="x509AuthenticationProvider" />
</security:authentication-manager>
```

The first component that is used by the **PreAuthenticatedAuthenticationProvider** is the **preAuthenticatedUserDetailsService**. This component creates user information from the certificate. Here we use a predefined component that create a simple token with the name of the subject CN. The authorities are copied from the static definition above in the filter.

Another attempt for creating the Spring authentication token could be a lookup in a user service – the configuration for the provider would be like this:

spring XML fragment

```
<security:user-service id="securityRealmFlowUserDetailsService">
  <security:user name="gears demo client ssl" password="{noop}user"
  authorities="ROLE_USER" />
</security:user-service>

<bean id="x509AuthenticationProvider"
  class="org.springframework.security.web.authentication.preauth.\
PreAuthenticatedAuthenticationProvider">
  <property name="preAuthenticatedUserDetailsService">
    <bean class="org.springframework.security.core.userdetails.\
UserDetailsServiceByNameServiceWrapper">
      <property name="userDetailsService" ref="securityRealmFlowUserDetailsService" />
    </bean>
  </property>
</bean>
```

Now we look up a user instance by the subject CN.

6.5.3 Gears principal configuration

Again, we want the principal to be derived from Spring security, so we use the **SpringSecurityPrincipalProvider**.

Regarding the **principalDao**, we use again the same configuration as with basic auth:

spring XML fragment

```
<bean id="principalProviderUser"
  class="de.intarsys.cloudsuite.gears.security.spring.\
SpringSecurityPrincipalProvider">
  <property name="role" value="urn:intarsys:names:principal:1.0:role:User" />
  <property name="keyConverter">
    <bean class="de.intarsys.cloudsuite.gears.security.spring.\
AuthenticationToStringConverter" />
  </property>
  <property name="principalDao">
    <bean class="de.intarsys.cloudsuite.gears.model.entity.principal.\
LiteralPrincipalDao" />
  </property>
</bean>
```

This means that we convert the Spring authentication data to a key and then lookup principal information in our DAO. For the X.509 this means that

- Spring derives the "authentication name" from the subject CN
- we derive a principal from the authentication name

6.5.4 Demo application

To use this authentication scenario, you must setup client TLS (see chapters above).

You can access the authenticated principal for use in the demo signer by adding a corresponding parameter to the "signer arguments" in the demo settings page:

demo settings signer args

```
{
  "documentSigner": {
    "args": {
      "digestSigner": {
        "args": {
          "signerIdentifier": "${principal.user.name}"
        }
      }
    }
  }
}
```

This should result in the certificate subject CN.

6.6 Any authentication, attached viewer

The "attached viewer" is created by the "viewer/create" service request. It is treated like any other request, which means if you have set up authentication for the "flow" realm you need to provide credentials.

The setup is exactly the same as in the scenario above.

After creation the resulting flow will return a redirect that is provisioned with the "object capability key" – the conversation id – of this authenticated flow.

The resource we redirect to (the gears SPA) is itself not access restricted. After starting the SPA uses the conversation id to manage its access to the flow constrained resources. No further authenticated API call is required here.

When launching a signature, the principal is propagated.

You see, we do not need special treatment when working with the viewer – all scenarios above simply work the same way.

6.7 Discard internal security definition

When the builtin security definition only barely matches your requirements its best to back off completely and create one from scratch.

You must start with the spring profile “customSecurity” in your properties

```
spring.profiles.active=customSecurity
```

Now spring security is completely off! Additional information on spring security you can find in the examples that come with gears and the great spring online documentation.

An example for your own security may be

```

<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xmlns:w="http://www.intarsys.de/schema/widget"
xmlns:tx="http://www.springframework.org/schema/tx"
xmlns:util="http://www.springframework.org/schema/util"
xmlns:security="http://www.springframework.org/schema/security"
xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd
http://www.intarsys.de/schema/widget http://www.intarsys.de/schema/widget/widget.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx.xsd
http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util-3.1.xsd
http://www.springframework.org/schema/security
http://www.springframework.org/schema/security/spring-security.xsd"
">

<description>
  This configuration wires a simple security environment.
</description>

<!--
spring AuthenticationEntryPoint for signaling HTTP FORBIDDEN

  This implementation both provides JSON structured error payload *and* gears
observations
-->
<bean id="apiUnauthorizedAuthenticationEntryPoint"
class="de.intarsys.spring.security.ApiForbiddenAccessDeniedHandler" />

<!--
spring AccessDeniedHandler for signaling HTTP FORBIDDEN

  This implementation both provides JSON structured error payload *and* gears
observations
-->
<bean id="apiForbiddenAccessDeniedHandler"
class="de.intarsys.spring.security.ApiForbiddenAccessDeniedHandler" />

<!--
The gateway between spring security and intarsys aaa.

This is required if you want to define additional ACLs (authorization)
-->
<bean class="de.intarsys.spring.security.SpringAuthProvider" />

<security:authentication-manager id="securityRealmSimpleAuthenticationManager">
  <security:authentication-provider>
    <security:user-service>
      <security:user name="dude" password="{noop}dude" authorities="ROLE_SIMPLE" />
    </security:user-service>
  </security:authentication-provider>
</security:authentication-manager>

<security:http name="securityRealmSimple" create-session="stateless"
authentication-manager-ref="securityRealmSimpleAuthenticationManager"
pattern="/api/v1/flow/**"
entry-point-ref="apiUnauthorizedAuthenticationEntryPoint">

  <security:http-basic />
  <security:access-denied-handler ref="apiForbiddenAccessDeniedHandler" />
  <security:csrf disabled="true" />

  <security:intercept-url pattern="/api/v1/flow/*/create" access="hasRole('SIMPLE')" />
  <security:intercept-url pattern="/api/v1/flow/private/**" access="hasRole('SIMPLE')" />
</security:http>

<!-- gears core api, session token -->
<security:intercept-url pattern="/api/v1/flow/**" access="permitAll" />
<!-- catch all -->
<security:intercept-url pattern="/**" access="denyAll" />

```

```
</security:http>  
  
<security:http pattern="/**" security="none">  
</security:http>  
  
</beans>
```

Some important observations:

To keep clients happy that are used to get structured error messages and proper audit logs, we recommend using the **apiUnauthorizedAuthenticationEntryPoint** and **apiForbiddenAccessDeniedHandler** as they are defined here. They insure consistent and complete error handling.

If you want to use ACL authorization, having **de.intarsys.spring.security.SpringAuthProvider** in place is required.

Anything else is to your will...

7. External References

- [1] intarsys GmbH, Sign Live! cloud suite gears manual.
- [2] Swisscom AG, "All-in Signing Service Reference Guide," [Online]. Available:
http://documents.swisscom.com/product/1000255-Digital_Signing_Service/Documents/Reference_Guide/Reference_Guide-All-in-Signing-Service-de.pdf.
- [3] intarsys GmbH, Sign Live! cloud suite gears cookbook.
- [4] intarsys GmbH, Sign Live! cloud suite gears incubator.
- [5] intarsys GmbH, Sign Live! cloud suite gears tutorial.
- [6] intarsys GmbH, Sign Live! Security Applications Developers Guide.
- [7] intarsys GmbH, Sign Live! cloud suite gears cookbook.