BFH (Bern University of Applied Sciences), CH-2501 Biel, Switzerland

UniCert Architecture Specification

Version 0.1

Philémon von Bergen

September 10, 2014

Revision History

Revision	Date	Author(s)	Description
0.1	September 10, 2014	Philémon von Bergen	Initial draft.

Contents

1 Introduction

This document presents the architectural specification of UniCert. UniCert is a certification authority, that issues digital certificates used to authenticate users, to sign and/or encrypt messages. UniCert provides an interface where the user can authenticate themselves and request a certificate corresponding to their needs. UniCert uses UniBoard to publish all issued certificates. A general description of what UniCert does and how it works is available in document [2].

This document presents the detailed certificate issuing process and focuses on architectural and technical specifications. In a Chapter 2, the components and processes of authentication and issuance of the certificate will be described in more details. Chapter 3 describes the architecture of some important classes. Finally, Chapter 4 describes some important UniCert particularities.

2 Components and Processes

To give an overview of how UniCert works, we give a high-level picture of the architecture and the processes that compose UniCert.

2.1 Components

UniCert is composed of two components. The fist one, the authentication component called *unicert-authentication*, has following functions:

- authenticate the user
- allow the user to generate a key pair and select some options
- send all data to issuer component
- return the private key (generated on user side) and the issued certificate (received from the issuer component) to the user.

So, this component proposes a frontend for user interactions and a backend to process the requests coming from the user.

The second component, the issuer component called *unicert-issuer*, has following functions:

- issue the certificate based on the data received
- publish the certificate on UniBoard
- return the certificate to the authentication component

2.2 Authentication Process

Currently, the authentication component supports two identity providers to authenticate the user, namely SwitchAAI and Google. Since they do not use the same technologies, the flow of the authentication process is slightly different for these two providers. Following paragraphs describe the process for each of them.

SwitchAAI authentication SwitchAAI uses a Shibboleth module of Apache webserver to make the redirection to the authentication webpage and back. Figure 2.1 shows how the process works. To begin the procedure, the user must invoke a link containing some information about what parameters must be loaded for the authentication and issuance process (described in more details later in Section 4.3). This can for example be done by clicking on a link on the home page of UniCert. This link calls the *ParameterServlet* which loads the required parameters, among others the supported identity providers for authentication. If

more than one are supported, page *idpselection.xhtml* is displayed where the user can choose which provider to use.

For the present case, SwitchAAI is selected. Page *switchaai.xhtml* is requested. For this page, the Apache Shibboleth module expects a valid user to be logged in. If it is not the case, the user is redirected to SwitchAAI authentication page. Once the credentials entered and the user successfully authenticated by SwitchAAI, the Apache Shibboleth module verifies the response received from Switch and displays the *switchaai.xhtml* page where the user personal information received from Switch are read. After a few seconds, the user is redirected to the page where the certificate can be requested (*certificate-request.xhtml*).

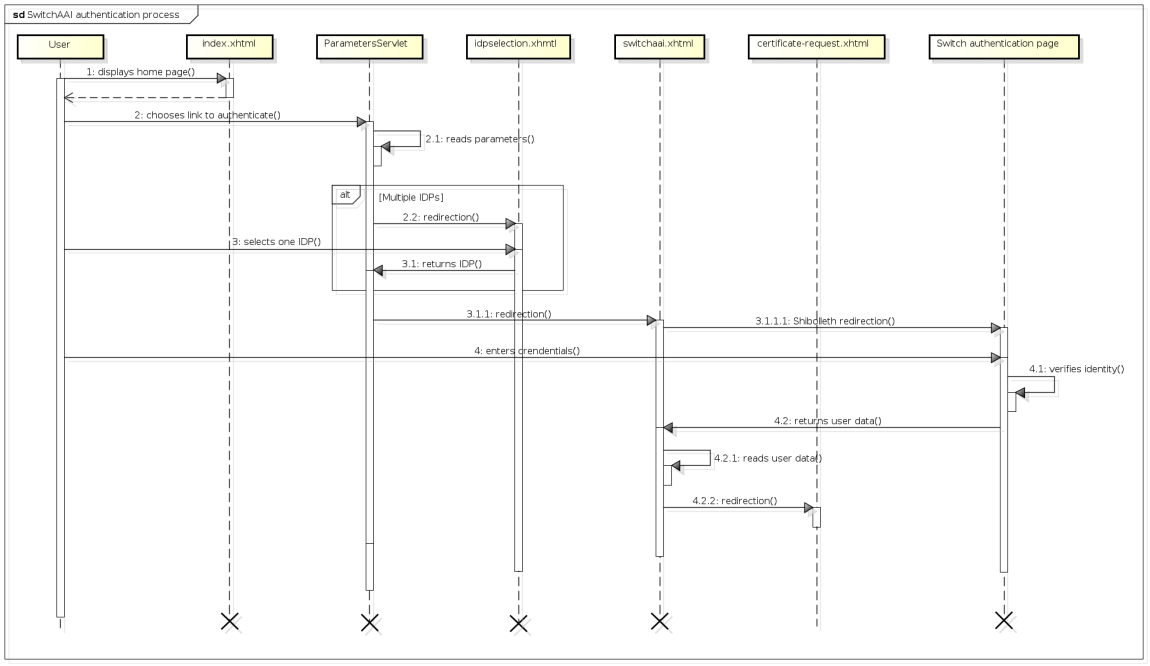

Figure 2.1: Authentication process with SwitchAAI

Google authentication Google uses protocol OAuth to manage the authentication. Because of this different technology, the process of authentication with Google is slightly different from the one with SwitchAAI. Figure 2.2 shows how it works. The first steps are identical until the identity provider is chosen (for this case Google). The *ParametersServlet* redirects the user to Google where the authentication process takes place. The response from Google is sent back to the *OAuth2CallbackServlet* which verifies the response from Google and reads the user personal information received. If everything was correct, the user is redirected to the page where the certificate can be requested (*certificate-request.xhtml*).

2.3 Certificate Issuance Process

The rest of the process is issuance of the certificate. This works in the same way independently of the identity provider used. The process is shown in Figure 2.3. This whole process takes place on the page *certificate-request.xhtml*. On this page, the user has to provide following information for the certificate they request:

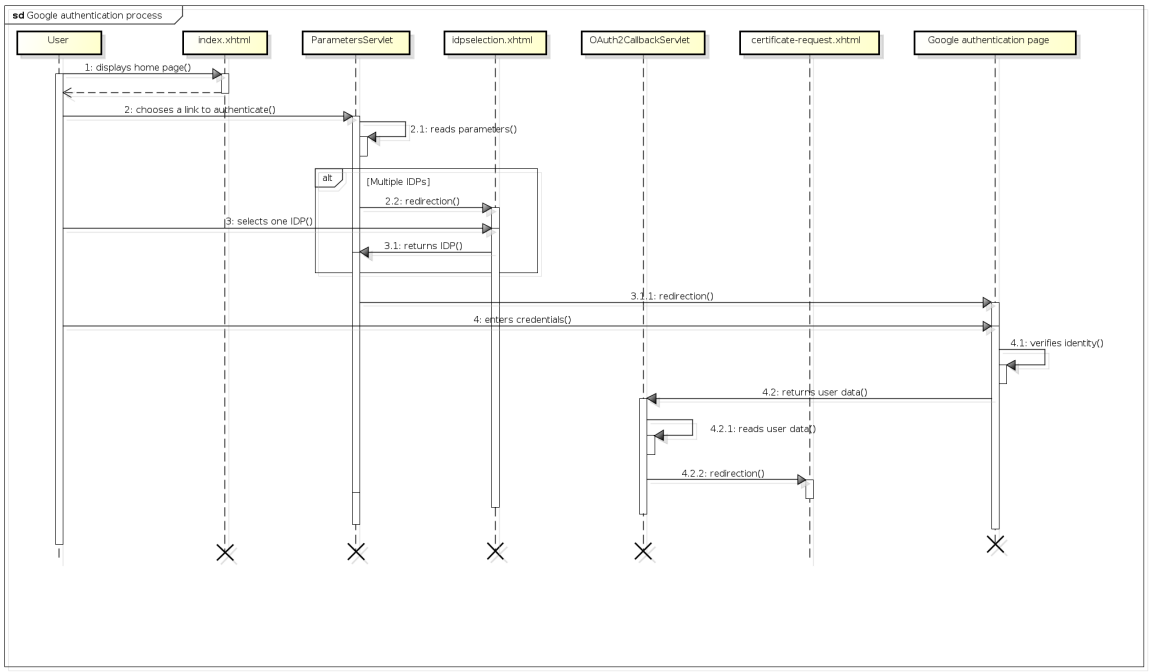

Figure 2.2: Authentication process with Google OAuth

- type of cryptographic values (currently supported: RSA and discrete logarithm)
- cryptographic parameters: size of key for RSA, values p, q, g for discrete logarithm
- a private key (generated locally with RSA)
- a password to encrypt the private key
- application identifier of the application which the certificate is issued for
- role in the application the certificate is issued for
- the identity function that must be applied on the identity information

The private key is generated on client side in javascript. The public key corresponding to the generated private key is computed together with a signature (or a proof in case of discrete logarithm setup) to prove the knowledge of the private key. Through javascript, the data listed above (except the private key and the password) are sent to the *CertificateRequestServlet*. In this servlet, the user personal information (also called *user data*) that were received during the authentication process (see Section 2.2) are used in combination with the identity function to generate the *identity data* that will be included in the certificate. All the processes described until here are realized by the *unicert-authentication* component. The data received and the identity data are then transmitted to the *unicert-issuer* component. This second component verifies the data received and issues the certificate as requested, publishes it on UniBoardand returns it to the *unicert-authentication* component which returns it to the user. In parallel, the secret key is encrypted with the password and sent by mail to the user through the *unicert-authentication* component.

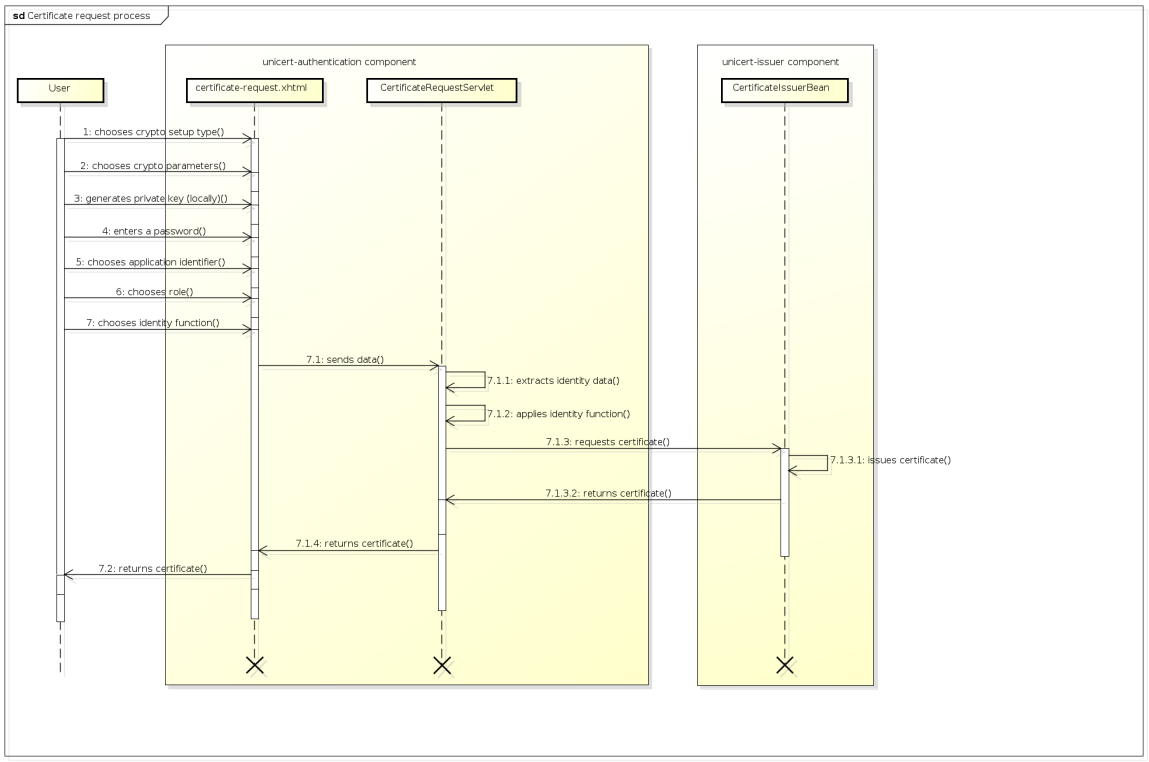

Figure 2.3: Process of requesting a certificate

3 Architecture

This chapter describes in more details some classes and other architecture concepts of the *unicert-authentication* component and of the *unicert-issuer* component.

3.1 unicert-authentication

unicert-authentication is a web application composed of two part, the frontend in JSF/XHTML/Javascript and the backend in Java. The interface between these two parts works over servlets and beans, namely *ParametersServlet*, *CertificateRequestServlet* and *SwitchUserDataBean*. Some important concepts related to the interface between the Java backend and the web frontend as well as some other classes will be described in this section.

3.1.1 UserData and IdentityFunction

The format of the user data returned by the identity provider is not the same for all identity providers. The way these data are transmitted to UniCert is also different for various identity providers. These facts require a flexible solution to provide support of multiple identity providers and to allow to easily add a new identity provider.

As already mentioned in Section 2.2, SwitchAAI use a Shibboleth Apache module for the authentication. This technology requires a user to be logged in before displaying a web page. In this concrete case, it is the page *switchaai.xhtml*. If the user is not logged in, a redirection takes place to SwitchAAI login page. After a successful login, the page *switchaai.xhtml* is displayed to the user, and in the background, a bean called *UserDataBean* is invoked to read the user data returned by Switch in the session. These user data are then stored in an object of the class *SwitchAAIUserData*.

For Google, the user data are returned to a servlet called *OAuth2CallbackServlet* after successful login. This servlet stores the user data in a object of the class *GoogleUserData* and creates the *UserDataBean*. Both *SwitchAAIUserData* and *GoogleUserData* inherits from the *UserData* interface but their content is not the same as shown in Figure 3.1, .

The *certificate-issuer* component however must know the content of the UserData object in order to be able to extract the data needed in the certificate. Therefore, an identity function is used. There is at least one identity function per identity provider that knows the format of the corresponding user data. So, the GoogleIdentityFunction class knows the fields present in a GoogleUserData object, and the SwitchAAIIdentityFunction knows the content of the SwitchAAIUserData. A concrete IdentityFunction object allows to transform a concrete UserData object into a generic class called *IdentityData* containing the identity information needed for the certificate. This IdentityData object can then be used by the *certificate-issuer* component.

The way how the identity information are extracted from a concrete UserData object is defined in a concrete IdentityFunction object. This process must be adaptable: for example an application could require the e-mail address to appear in certificate, another application could require an other unique identifier, or a third application could require an anonymized

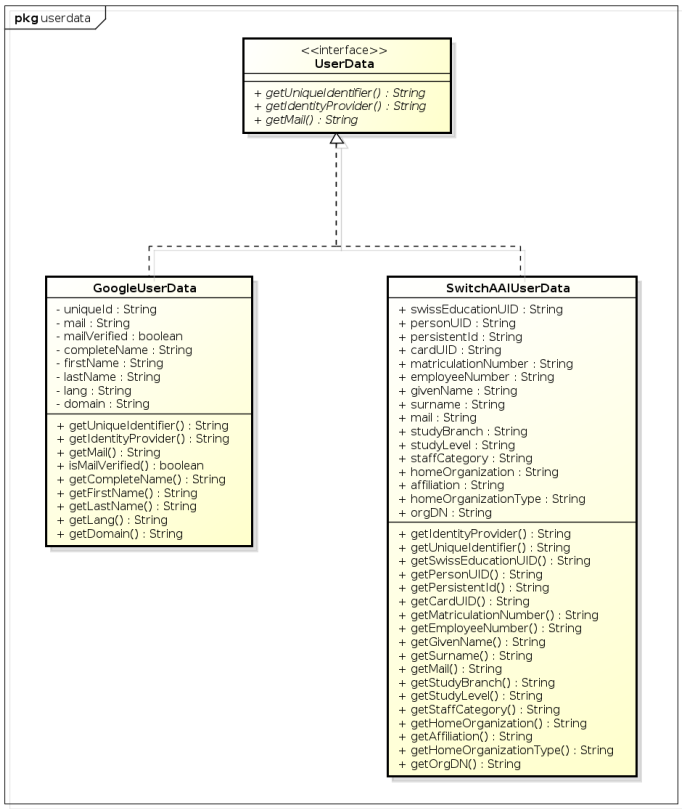

Figure 3.1: Class diagram of user data classes

form of the e-mail address, and so on. Therefore, multiple IdentityFunction can exist for the same identity provider. In this implementation, there is a standard and an anonymized identity function for each identity provider (*StandardSwitchAAIIdentityFunction*, *StandardGoogleIdentityFunction*, *AnonymizedSwitchAAIIdentityFunction*, *AnonymizedGoogleIdentityFunction*). For SwitchAAI, there is a third variant specially designed for the university of Zürich which uses the student identification number as common name (*ZurichSwitchAAIIdentityFunction*).

The organisation of these classes is shown in Figure 3.2. The *apply* method is the publicly available method extracting the information out of the given UserData object and returning the generated IdentityData object shown in Figure 3.3. Different protected methods are used for this process. *selectCommonName* is responsible for extracting the common name from the UserData. In standard functions, the e-mail address is used. In anonymized function, a hash of the e-mail address is used. In the special function for Zürich, the student identification number is used. *selectUniqueId* is responsible for extracting a unique identifier from the UserData. *putInOtherValues* allows to add information in extensions of the certificate in case it cannot be included in the fields proposed in the IdentityData class. Finally, method *createIdentityData* indicate which fields are put in the IdentityData object. This architecture allows to create subclasses which can overwrite some behaviour of the super class. For example, method *createIdentityData* in anonymized function does not include ”firstname” and ”surname” in the IdentityData object as this is done in standard functions.

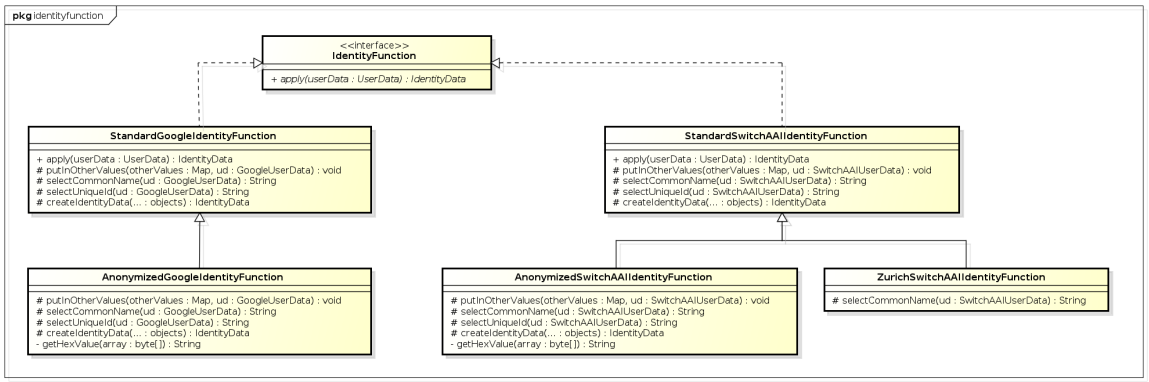

Figure 3.2: Class diagram of the identity function classes

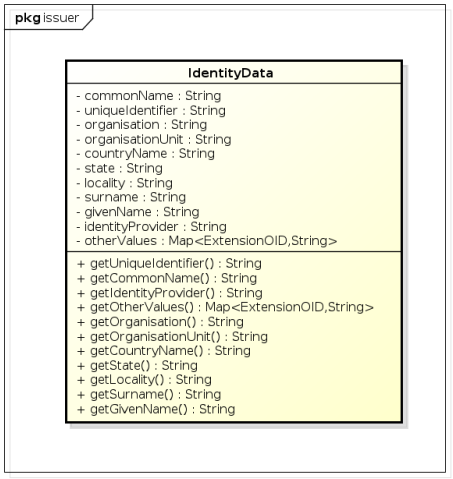

Figure 3.3: IdentityData class

3.1.2 Error description

Errors occurring during the issuance process should be communicated in an understandable way to the user. Exception caught in JSF result in a web page displaying the stack trace. This is not intuitive. An exception thrown in the backend result in an HTTP 500 JSON response with an error code and a default error description. Since the issuance request is done over Ajax, the response is also received through Ajax. The error code is then read in Javascript and the corresponding error description is selected in the language file. This message is then displayed to the user.

3.1.3 Javascript cryptography

In the frontend, some cryptographic computation must be realized in Javascript. It concerns the generation of the key pair and the computation of the signature and proofs sent to the backend. In Javascript, a BigInteger library called Leemon [1] is used to represent big integers. The needed cryptographic primitives are implemented on top. Because of the bad performance of Javascript in some browsers, some of these primitives have been implemented in an asynchronous way in order to avoid Javascript time outs.

These signatures/proofs generated are verified in the backend using a library called UniCrypt. So, the generation of the signatures and proofs in Javascript must be completely compatible with UniCrypt. Therefore, hash function SHA-256 is used on both side. Recursive hash method is used on UniCrypt side and the equivalent is generated on Javascript side. Strings are considered encoded in UTF-8 and big integers as big endian.

3.2 unicert-issuer

unicert-issuer is an EJB component responsible for the issuance of the certificate. The main class is the *CertificateIssuer* EJB which checks the data received and issues the certificate. Some other classes used in this process will be described in this section.

3.2.1 CryptographicSetup

The class *CryptographicSetup* is a container for all cryptographic values that are submitted by the user and that must be included in the certificate or verified in the *unicert-issuer* component. Two concrete CryptographicSetup exist: *DiscreteLogSetup* and *RSASetup* as shown in Figure 3.4. They both include all values that define such a setup and the signature or proof generated in the frontend. Objects of these classes are created by the *unicert-authentication* component on certificate request.

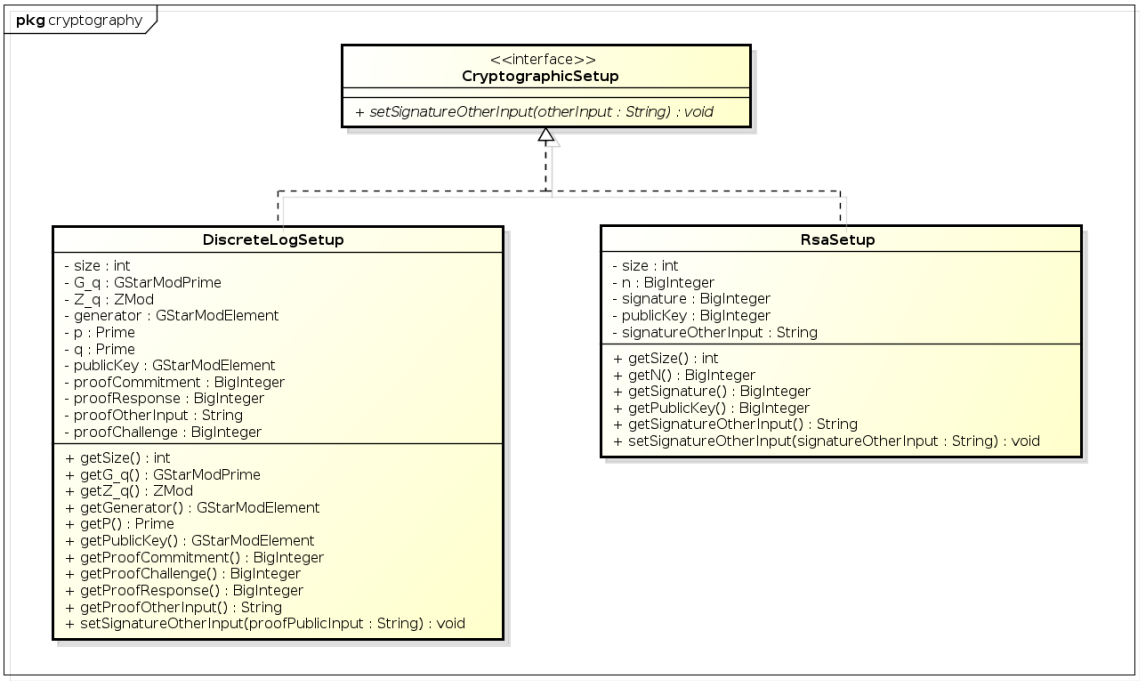

Figure 3.4: Class diagram of the cryptographic setup classes

3.2.2 Certificate class

Certificate class is the Java representation of the X.509 certificate that is issued. The fields that appear in that certificate are stored here together with the PEM format of the issued

certificate as shown in Figure 3.5. When posting the certificate on UniBoard, a JSON representation of this class is created through the method *toJSON*. This allows UniBoard to make query on the content of a certificate. The listing below shows the schema of the generated JSON certificate.

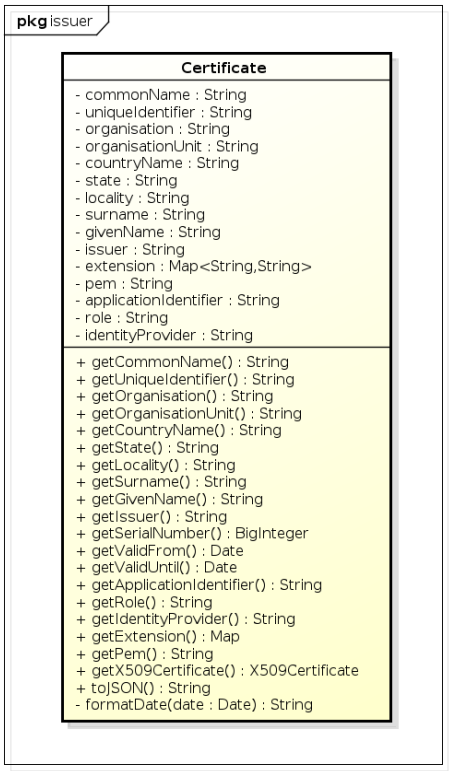

Figure 3.5: Certificate class

```
{
        "title": "Schema for UniCert certificates",
        "description": "This schema describes the format of a
           UniCert certificate in JSON format",
        "type":"object",
        "$schema": "http://json-schema.org/draft-04/schema",
        "properties": {
                "commonName": {
                        "type": "string",
                        "description": "Common name of
                           certificate owner"
                },
                "uniqueIdentifier": {
                        "type": "string",
                        "description":  "Unique identifier of
                           certificate owner"
                },
                "organisation": {
```

```
                        "type": "string",
                        "description": "Organisation of
                            certificate owner"
                },
                "organisationUnit": {
                        "type": "string",
                        "description": "Organisation unit of
                            certificate owner"
                },
                "countryName": {
                        "type": "string",
                        "description": "Country of certificate
                            owner"
                },
                "state": {
                        "type": "string",
                        "description": "State of certificate
                            owner"
                },
                "locality": {
                        "type": "string",
                        "description": "Locality certificate
                            owner"
                },
                "surname": {
                        "type": "string",
                        "description": "Surname of certificate
                            owner"
                },
                "givenName": {
                        "type": "string",
                        "description": "Given name of
                            certificate owner"
                },
                "issuer": {
                        "type": "string",
                        "description": "Issuer of the
                            certificate"
                },
                "serialNumber": {
                        "type":"string",
                        "description": "Serial number of the
                            certificate"
                },
                "validFrom": {
                        "type": "string",
                        "description": "Date when certificate
                            starts to be valid"
```

```
                },
                "validUntil": {
                        "type": "string",
                        "description": "Date when certificate
                           stops to be valid"
                },
                "applicationIdentifier": {
                        "type": "string",
                        "description": "Application the
                           certificate has been issued for"
                },
                "role": {
                        "type": "string",
                        "description": "Role inside an
                           application the certificate has been
                            issued for"
                },
                "identityProvider": {
                        "type": "string",
                        "description": "Identity provider used
                           to verify the identity of the
                           certificate owner"
                },
                "pem": {
                        "type": "string",
                        "description": "Certificate in PEM
                           format"
                }
        },
        "required": ["commonName", "issuer", "serialNumber", "
           validFrom", "validUntil", "identityProvider", "pem"
           ]
}
```

4 UniCert Particularities

This chapter describes in more details some particularities of UniCert.

4.1 X.509 Certificate Extensions

Some special information must be added by UniCert in the issued certificate, like the identity provider used for the authentication, the application and the role the certificate is issued for. This information cannot really be included in existing X.509 fields. So, we defined some extensions for each of these elements. In X.509 certificates, an extension must always be identified by an object identifier (OID). The Bern University of Applied Sciences own a group OID: 1.3.6.1.4.1.13305. Following subgroup is reserved for UniCert 1.3.6.1.4.1.13305.1.101.x. Table 4.1 presents the elements included in the certificate and their corresponding OIDs. The content of these extensions are always ASN1 strings.

Element	OID
Identity provider	1.3.6.1.4.1.13305.1.101.1
Application identifier	1.3.6.1.4.1.13305.1.101.2
Role	1.3.6.1.4.1.13305.1.101.3
Language (only for Google)	1.3.6.1.4.1.13305.1.101.4
Test extension	1.3.6.1.4.1.13305.1.101.999

Table 4.1: OIDs

4.2 JNDI Propreties for *unicert-issuer*

To work properly, *unicert-issuer* require some configuration values. These values are defined as JNDI properties. Table 4.2 shows what are these values. They are stored in a property set called *unicertProps*.

4.3 Simplification of the User Interface

As described in Section 2.3, when the user requests a certificate, they have to provide, among other, following information:

- type of cryptographic values (currently supported: RSA and discrete logarithm)
- cryptographic parameters: size of key for RSA, values p, q, g for discrete logarithm
- application identifier of the application the certificate is issued for
- role in the application the certificate is issued for

Property	Description
keystoreExternal	If the keystore must be loaded from an external location
keystorePath	Path to the keystore containing the certificate and private key of UniCert for signing issued certificates
keystorePass	Password for the keystore
privateKeyPass	Password used to protect private key of UniCert certificate
issuerId	Name of the issuer of the generated certificates
validityYears	Time slot in which the certificate is valid (2 years)
uniboardURL	URL of UniBoard webservice (if not indicated, certificates are not published)
googleRedirectURI	URL of the callback servlet after Google OAuth2 authentication
googleClientID	OAuth2 credentials for Google authentication
googleClientSecret	OAuth2 credentials for Google authentication

Table 4.2: JNDI properties for *unicert-issuer*

- the identity function that must be applied on the user's personal information

Most of the time however, these values are the same for a lot of users. For example, when a voter wants to issue a voter certificate for UniVote, the cryptographic values, the application identifier (in this case UniVote) and the role (in this case *Voter*) are the same for all users that request a certificate. Even the identity function is the same for all voters inside a given election. So, it is unintuitive to ask each user to enter all these data. Therefore, a feature has been implemented allowing to preset all these values. It works with JNDI properties. The administrator can define multiple JNDI property sets giving them a name like "/unicert/mypropertyset". In this property set, the administrator can define the properties they want not to be filled by the user. The ones that are not defined must then be filled by the user in the certificate request process. The property set is loaded by the *ParametersServlet* which receives the name of the property set to load through the link on which the user has clicked to start the authentication process, for example "authenticate?params=mypropertyset". If no property set is indicated, a default property set is loaded containing only the identity providers supported. All others information must be filled by the user. Table 4.3 shows the JNDI properties that can be preset in a property set.

Property	Description	Possible values	
identityProvider	The identity provider to use for authentication	”SwitchAAI”, ”Google”, ”SwitchAAI,Google”	Mandatory
keyType	Cryptographic setup type	”RSA”, ”DiscreteLog”	Optional
keySize	Size of RSA keys	integer	Optional
primeP	Value of prime p	base 10 representation	Optional
primeQ	Value of prime q	base 10 representation	Optional
generator	Value of generator	base 10 representation	Optional
applicationIdentifier	Application identifier	string	Optional
role	Role	string	Optional
identityFunctionIndex	Index of identiy function to select in dropdown	integer	Optional

Table 4.3: JNDI properties for presets in user interface

Bibliography

[1] Big Integer Library for Javascript. `http://www.leemon.com/crypto/BigInt.js`. Accessed: 2014-10-07.

[2] R. Haenni, R. Koenig, S. Hauser, P. von Bergen, P. Locher, S. Fischli . UniVote System Specification. Technical report, 2014.