```
/*
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

'use strict';

const utils = require('./utils.js');
const config = utils.getConfig();
const logger = utils.getLogger('FabricCAClient.js');
const http = require('http');
const https = require('https');
const util = require('util');
const IdentityService = require('./IdentityService');
const AffiliationService = require('./AffiliationService');
const CertificateService = require('./CertificateService');

/**
 * Client for communciating with the Fabric CA APIs
 *
 * @class
 */
const FabricCAClient = class {

	/**
	 * constructor
	 *
	 * @param {object} connect_opts Connection options for communciating with the Fabric
CA server
	 * @param {string} connect_opts.protocol The protocol to use (either HTTP or HTTPS)
	 * @param {string} connect_opts.hostname The hostname of the Fabric CA server
endpoint
	 * @param {number} connect_opts.port The port of the Fabric CA server endpoint
```

```
	* @param {TLSOptions} connect_opts.tlsOptions The TLS settings to use when the Fabric CA endpoint uses "https"
	* @param {string} connect_opts.caname The optional name of the CA. Fabric-ca servers support multiple Certificate Authorities from
	*  a single server. If omitted or null or an empty string, then the default CA is the target of requests
	* @throws Will throw an error if connection options are missing or invalid
	*
	*/
	constructor(connect_opts, cryptoPrimitives) {

		// check connect_opts
		try {
			this._validateConnectionOpts(connect_opts);
		} catch (err) {
			throw new Error('Invalid connection options. ' + err.message);
		}

		this._caName = connect_opts.caname,
		this._httpClient = (connect_opts.protocol === 'http') ? http : https;
		this._hostname = connect_opts.hostname;
		if (connect_opts.port) {
			this._port = connect_opts.port;
		} else {
			this._port = 7054;
		}
		if (typeof connect_opts.tlsOptions === 'undefined' || connect_opts.tlsOptions === null) {
			this._tlsOptions = {
				trustedRoots: [],
				verify: false
			};
		} else {
			this._tlsOptions = connect_opts.tlsOptions;
			if (typeof this._tlsOptions.verify === 'undefined') {
				this._tlsOptions.verify = true;
			}
			if (typeof this._tlsOptions.trustedRoots === 'undefined') {
				this._tlsOptions.trustedRoots = [];
			}
		}
		this._baseAPI = '/api/v1/';
```

```
		this._cryptoPrimitives = cryptoPrimitives;

		logger.debug('Successfully constructed Fabric CA client from options - %j',
connect_opts);
	}

	/**
	 * @typedef {Object} KeyValueAttribute
	 * @property {string} name The key used to reference the attribute
	 * @property {string} value The value of the attribute
	 * @property {boolean} ecert Optional, A value of true indicates that this attribute
	 *  should be included in an enrollment certificate by default
	 */

	/**
	 * Register a new user and return the enrollment secret
	 * @param {string} enrollmentID ID which will be used for enrollment
	 * @param {string} enrollmentSecret Optional enrollment secret to set for the registered
user.
	 *	If not provided, the server will generate one.
	 *	When not including, use a null for this parameter.
	 * @param {string} role Optional type of role for this user.
	 *	When not including, use a null for this parameter.
	 * @param {string} affiliation Affiliation with which this user will be associated
	 * @param {number} maxEnrollments The maximum number of times the user is
permitted to enroll
	 * @param {KeyValueAttribute[]} attrs Array of key/value attributes to assign to the user
	 * @param {SigningIdentity} signingIdentity The instance of a SigningIdentity
encapsulating the
	 * signing certificate, hash algorithm and signature algorithm
	 * @returns {Promise} The enrollment secret to use when this user enrolls
	 */
	register(enrollmentID, enrollmentSecret, role, affiliation, maxEnrollments, attrs,
signingIdentity) {

		const self = this;

		// all arguments are required
		if (arguments.length < 7) {
			throw new Error('Missing required parameters. \'enrollmentID\',
\'enrollmentSecret\', \'role\', \'affiliation\', ' +
				'\'maxEnrollments\', \'attrs\' and \'signingIdentity\' are all required.');
		}
```

```
		if (typeof maxEnrollments !== 'number') {
			throw new Error('Parameter \'maxEnrollments\' must be a number');
		}

		return new Promise(((resolve, reject) => {
			const regRequest = {
				'id': enrollmentID,
				'affiliation': affiliation,
				'max_enrollments': maxEnrollments
			};

			if (role) {
				regRequest.type = role;
			}

			if (attrs) {
				regRequest.attrs = attrs;
			}

			if (typeof enrollmentSecret === 'string' && enrollmentSecret !== '') {
				regRequest.secret = enrollmentSecret;
			}

			return self.post('register', regRequest, signingIdentity)
				.then((response) => {
					return resolve(response.result.secret);
				}).catch((err) => {
					return reject(err);
				});
		}));
	}

	/**
	 * Revoke an existing certificate (enrollment certificate or transaction certificate), or revoke
	 * all certificates issued to an enrollment id. If revoking a particular certificate, then both
	 * the Authority Key Identifier and serial number are required. If revoking by enrollment id,
	 * then all future requests to enroll this id will be rejected.
	 * @param {string} enrollmentID ID to revoke
	 * @param {string} aki Authority Key Identifier string, hex encoded, for the specific certificate to revoke
```

```
         * @param {string} serial Serial number string, hex encoded, for the specific certificate to
revoke
         * @param {string} reason The reason for revocation. See
https://godoc.org/golang.org/x/crypto/ocsp
         *  for valid values
         * @param {SigningIdentity} signingIdentity The instance of a SigningIdentity
encapsulating the
         * signing certificate, hash algorithm and signature algorithm
         * @returns {Promise} The revocation results
         */
        revoke(enrollmentID, aki, serial, reason, signingIdentity) {

                const self = this;

                // all arguments are required
                if (arguments.length < 5) {
                        throw new Error('Missing required parameters. \'enrollmentID\', \'aki\',
\'serial\', \'reason\', ' +
                                '\'callerID\' and \'signingIdentity\' are all required.');
                }

                return new Promise(((resolve, reject) => {

                        const regRequest = {
                                'id': enrollmentID,
                                'aki': aki,
                                'serial': serial,
                                'reason': reason
                        };

                        return self.post('revoke', regRequest, signingIdentity)
                                .then((response) => {
                                        return resolve(response);
                                }).catch((err) => {
                                        return reject(err);
                                });
                }));
        }

        /**
         * Re-enroll an existing user.
         * @param {string} csr PEM-encoded PKCS#10 certificate signing request
```

```
	 * @param {SigningIdentity} signingIdentity The instance of a SigningIdentity
encapsulating the
	 * signing certificate, hash algorithm and signature algorithm
	 * @param {AttributeRequest[]} attr_reqs An array of {@link AttributeRequest}
	 * @returns {Promise} {@link EnrollmentResponse}
	 */
	reenroll(csr, signingIdentity, attr_reqs) {

		const self = this;

		// First two arguments are required
		if (arguments.length < 2) {
			throw new Error('Missing required parameters.  \'csr\', \'signingIdentity\'
are all required.');
		}

		return new Promise(((resolve, reject) => {

			const request = {
				certificate_request: csr
			};

			if (attr_reqs) {
				request.attr_reqs = attr_reqs;
			}

			return self.post('reenroll', request, signingIdentity)
				.then((response) => {
					return resolve(response);
				}).catch((err) => {
					return reject(err);
				});
		}));
	}

	/**
	 * Creates a new {@link IdentityService} instance
	 *
	 * @returns {IdentityService} instance
	 */
	newIdentityService() {
		return new IdentityService(this);
	}
```

```
	/**
	 * Create a new {@link AffiliationService} instance
	 *
	 * @returns {AffiliationService} instance
	 */
	newAffiliationService() {
		return new AffiliationService(this);
	}

	/**
	 * Create a new {@link CertificateService} instance
	 *
	 * @returns {CertificateService} instance
	 */
	newCertificateService() {
		return new CertificateService(this);
	}

	post(api_method, requestObj, signingIdentity) {
		return this.request('POST', api_method, signingIdentity, requestObj);
	}

	delete(api_method, signingIdentity) {
		return this.request('DELETE', api_method, signingIdentity);
	}

	get(api_method, signingIdentity) {
		return this.request('GET', api_method, signingIdentity);
	}

	put(api_method, requestObj, signingIdentity) {
		return this.request('PUT', api_method, signingIdentity, requestObj);
	}

	request(http_method, api_method, signingIdentity, requestObj) {

		// Check for required args (requestObj optional)
		if (arguments.length < 3) {
			return Promise.reject('Missing required parameters.  \'http_method\',
\'api_method\' and \'signingIdentity\' are all required.');
		}
```

```
		if (requestObj) {
			requestObj.caName = this._caName;
		}
		// establish socket timeout
		// default: 3000ms
		const CONNECTION_TIMEOUT = config.get('connection-timeout', 3000);
		// SO_TIMEOUT is the timeout that a read() call will block,
		// it means that if no data arrives within SO_TIMEOUT,
		// socket will throw an error
		// default: infinite
		const SO_TIMEOUT = config.get('socket-operation-timeout');
		logger.debug('CONNECTION_TIMEOUT = %s, SO_TIMEOUT = %s',
CONNECTION_TIMEOUT, SO_TIMEOUT ? SO_TIMEOUT : 'infinite');

		const self = this;
		const requestOptions = {
			hostname: self._hostname,
			port: self._port,
			path: self._baseAPI + api_method,
			method: http_method,
			headers: {
				Authorization: self.generateAuthToken(requestObj,
signingIdentity)
			},
			ca: self._tlsOptions.trustedRoots,
			rejectUnauthorized: self._tlsOptions.verify,
			timeout: CONNECTION_TIMEOUT
		};

		return new Promise(((resolve, reject) => {

			const request = self._httpClient.request(requestOptions, (response) => {

				const responseBody = [];
				response.on('data', (data) => {
					responseBody.push(data);
				});

				response.on('end', (data) => {
					const payload = responseBody.join('');

					if (!payload) {
						return reject(new Error(
```

```
                                                util.format('fabric-ca request %s failed with
HTTP status code %s', api_method, response.statusCode)));
                                        }
                                        // response should be JSON
                                        let responseObj;
                                        try {
                                                responseObj = JSON.parse(payload);
                                                if (responseObj.success) {
                                                        return resolve(responseObj);
                                                } else {
                                                        return reject(new Error(
                                                                util.format('fabric-ca request %s
failed with errors [%s]', api_method, JSON.stringify(responseObj && responseObj.errors ?
responseObj.errors : responseObj))));
                                                }

                                        } catch (err) {
                                                return reject(new Error(
                                                        util.format('Could not parse %s response
[%s] as JSON due to error [%s]', api_method, payload, err)));
                                        }
                                });
                        });

                        request.on('socket', (socket) => {
                                socket.setTimeout(CONNECTION_TIMEOUT);
                                socket.on('timeout', () => {
                                        request.abort();
                                        reject(new Error(util.format('Calling %s endpoint failed,
CONNECTION Timeout', api_method)));
                                });
                        });

                        // If socket-operation-timeout is not set, read operations will not time out
(infinite timeout).
                        if (SO_TIMEOUT && Number.isInteger(SO_TIMEOUT) && SO_TIMEOUT
> 0) {
                                request.setTimeout(SO_TIMEOUT, () => {
                                        reject(new Error(util.format('Calling %s endpoint failed,
READ Timeout', api_method)));
                                });
                        }
```

```
                        request.on('error', (err) => {
                                reject(new Error(util.format('Calling %s endpoint failed with error
[%s]', api_method, err)));
                        });

                        if (requestObj) {
                                request.write(JSON.stringify(requestObj));
                        }
                        request.end();
                }));
        }

        /*
         * Generate authorization token required for accessing fabric-ca APIs
         */
        generateAuthToken(reqBody, signingIdentity) {
                // specific signing procedure is according to:
                // https://github.com/hyperledger/fabric-ca/blob/master/util/util.go#L213
                const cert = Buffer.from(signingIdentity._certificate).toString('base64');
                let bodyAndcert;
                if (reqBody) {
                        const body = Buffer.from(JSON.stringify(reqBody)).toString('base64');
                        bodyAndcert = body + '.' + cert;
                } else {
                        bodyAndcert = '.' + cert;
                }

                const sig = signingIdentity.sign(bodyAndcert, {hashFunction:
this._cryptoPrimitives.hash.bind(this._cryptoPrimitives)});
                logger.debug(util.format('bodyAndcert: %s', bodyAndcert));

                const b64Sign = Buffer.from(sig, 'hex').toString('base64');
                return cert + '.' + b64Sign;
        }

        /**
         * @typedef {Object} AttributeRequest
         * @property {string} name - The name of the attribute to include in the certificate
         * @property {boolean} optional - throw an error if the identity does not have the attribute
         */

        /**
         * @typedef {Object} EnrollmentResponse
```

```
         * @property {string} enrollmentCert PEM-encoded X509 enrollment certificate
         * @property {string} caCertChain PEM-encoded X509 certificate chain for the issuing
         * certificate authority
         */

        /**
         * Enroll a registered user in order to receive a signed X509 certificate
         * @param {string} enrollmentID The registered ID to use for enrollment
         * @param {string} enrollmentSecret The secret associated with the enrollment ID
         * @param {string} csr PEM-encoded PKCS#10 certificate signing request
         * @param {string} profile The profile name.  Specify the 'tls' profile for a TLS certificate;
otherwise, an enrollment certificate is issued.
         * @param {AttributeRequest[]} attr_reqs An array of {@link AttributeRequest}
         * @returns {Promise} {@link EnrollmentResponse}
         * @throws Will throw an error if all parameters are not provided
         * @throws Will throw an error if calling the enroll API fails for any reason
         */
        enroll(enrollmentID, enrollmentSecret, csr, profile, attr_reqs) {

                const self = this;

                // check for required args
                if (arguments.length < 3) {
                        return Promise.reject('Missing required parameters.  \'enrollmentID\',
\'enrollmentSecret\' and \'csr\' are all required.');
                }

                const requestOptions = {
                        hostname: self._hostname,
                        port: self._port,
                        path: self._baseAPI + 'enroll',
                        method: 'POST',
                        auth: enrollmentID + ':' + enrollmentSecret,
                        ca: self._tlsOptions.trustedRoots,
                        rejectUnauthorized: self._tlsOptions.verify
                };

                const enrollRequest = {
                        caName: self._caName,
                        certificate_request: csr
                };

                if (profile) {
```

```
		enrollRequest.profile = profile;
	}

	if (attr_reqs) {
		enrollRequest.attr_reqs = attr_reqs;
	}

	return new Promise(((resolve, reject) => {

		const request = self._httpClient.request(requestOptions, (response) => {

			const responseBody = [];
			response.on('data', (chunk) => {
				responseBody.push(chunk);
			});

			response.on('end', (data) => {

				const payload = responseBody.join('');

				if (!payload) {
					return reject(new Error(
						util.format('Enrollment failed with HTTP
status code', response.statusCode)));
				}
				// response should be JSON
				try {
					const res = JSON.parse(payload);
					if (res.success) {
						// we want the result field which is
Base64-encoded PEM
						const enrollResponse = new Object();
						// Cert field is Base64-encoded PEM
						enrollResponse.enrollmentCert =
Buffer.from(res.result.Cert, 'base64').toString();
						enrollResponse.caCertChain =
Buffer.from(res.result.ServerInfo.CAChain, 'base64').toString();
						return resolve(enrollResponse);
					} else {
						return reject(new Error(
							util.format('Enrollment failed with
errors [%s]', JSON.stringify(res.errors))));
					}
```

```
                                        } catch (err) {
                                                return reject(new Error(
                                                        util.format('Could not parse enrollment
response [%s] as JSON due to error [%s]', payload, err)));
                                        }
                                });

                                response.on('error', (error) => {
                                        reject(new Error(
                                                util.format('Enrollment failed with error [%s]',
error)));
                                });
                        });

                        request.on('error', (err) => {
                                reject(new Error(util.format('Calling enrollment endpoint failed with
error [%s]', err)));
                        });

                        const body = JSON.stringify(enrollRequest);
                        request.end(body);

                }));

        }

        generateCRL(revokedBefore, revokedAfter, expireBefore, expireAfter, signingIdentity) {
                const self = this;

                if (arguments.length !== 5) {
                        return Promise.reject(new Error('Missing required parameters.
\'revokedBefore\', \'revokedAfter\' ,' +
                                '\'expireBefore\', \'expireAfter\' and \'signingIdentity\' are all
required.'));
                }

                const request = {};
                request.revokedBefore = revokedBefore;
                request.revokedAfter = revokedAfter;
                request.expireBefore = expireBefore;
                request.expireAfter = expireAfter;
                request.caname = self._caName;
```

```
		return new Promise(((resolve, reject) => {
			return self.post('gencrl', request, signingIdentity)
				.then((response) => {
					if (response.success && response.result) {
						return resolve(response.result.CRL);
					} else {
						return reject(response.errors);
					}
				}).catch((err) => {
					return reject(err);
				});
		}));
	}

	/**
	 * Validate the connection options
	 * @throws Will throw an error if any of the required connection options are missing or
invalid
	 * @ignore
	 */
	_validateConnectionOpts(connect_opts) {
		// check for protocol
		if (!connect_opts.protocol) {
			throw new Error('Protocol must be set to \'http\' or \'https\'');
		}

		if (connect_opts.protocol !== 'http') {
			if (connect_opts.protocol !== 'https') {
				throw new Error('Protocol must be set to \'http\' or \'https\'');
			}
		}

		if (!connect_opts.hostname) {
			throw new Error('Hostname must be set');
		}

		if (connect_opts.port) {
			if (!Number.isInteger(connect_opts.port)) {
				throw new Error('Port must be an integer');
			}
		}
```

```
        }
};

module.exports = FabricCAClient;
```