Technical Specification

Title	Shamirs Webservice
Author	Christof Reichardt Paul-Ehrlich-Weg 1 D-63110 Rodgau E-Mail: projektstudien@christofreichardt.de
Created on	2021-03-05
Version	1.0.0
Status	Draft

Table of Contents

1 Introduction

The software intends to provide several cryptographic services – such as the generation of symmetric and assymetric keys, the distribution of secret shares or the digital signature and encryption of electronic documents – by exposing an appropriate REST API. The cryptographic keys used for signing and encrypting usually remain on the server within PKCS12 keystores. Key entries within such PKCS12 keystores are typically encrypted as well, e.g. via password based encryption algorithms.

1.1 Terminology

AES
: Advanced Encryption Standard.

ECDSA
: Elliptic Curve Digital Signature Algorithm.

GET
: A method defined by the HTTP protocol.

HTTP
: Hypertext Transfer Protocol.

JSON
: A lightweight data-interchange format.

Participant
: A user which has subscribed to a secret sharing sheme.

PKCS12
: Defines a file format used to store cryptographical objects, e.g. private keys, within a single file.

POST
: A method defined by the HTTP protocol.

PUT
: A method defined by the HTTP protocol.

REST
: Representational state transfer denotes an architectural style related to interactive applications using web services.

Secret Sharing
: Methods for distributing shares of a secret, e.g. a password, between certain participants. The participants need to combine a subset of the shares in able to recover the original secret.

URI
: Uniform Resource Identifier.

XML
: An extensible markup language for encoding of documents.

1.2 Product Outline

The center piece of the service is the administration of keystores which belong to certain participants. These keystores will be generated by the server on demand together with specified cryptographic keys according to submitted instructions. The participants may use their associated keystores for the encryption/decryption or rather digital signing/verification of electronic documents. In order to use a keystore someone must utilize its provisioned session which can only be activated if sufficient and appropriate password shares are available. The expired sessions of a keystore maintain a history of actions which have been executed by using the keystore. Password shares are only valid for a single session. After the closure of a session new password shares will be generated for the keystore and its entries will be reencrypted via usual password based encryption algorithms. As an option the just now computed password shares will be actively distributed to the participants. Alternatively the password shares might be fetched from the server by their owners via the REST API. If the password shares remaining on the server fall below a certain threshold the associated keystore becomes unloadable.

1.3 Security Goals

1.3.1 Multiple-Eye Principles

It should be possible to enforce the multiple-eye principle when granting access to one of the keystores comprising the cryptographic keys. Consider, for example, a company signing key within such a keystore and an electronic payment order which must be reviewed and digitally signed. This can be achieved by applying a secret sharing algorithm in addition to the password based encryption of the keystores.

1.3.2 System Administration

Ideally, the system administration of the cryptographic services should not be able to gain access to the cryptographic keys even with root access to the server. This means that the passwords – or rather password shares – of the keystores should not remain on the server after their initial generation. Either the

password shares will be distributed, e.g. via (encrypted) E-Mail messages, immediately after their creation or the clients fetch them via the REST API in due course. In order to open a session related to a certain keystore the appropriate shares must be again transmitted back to the server by the participants. After the closure of the session the key entries of the involved keystore will be re-encrypted and new fitting password shares will be distributed. That means that a seizure of the server alone should normally not lead to a security breach. Encrypted documents or private keys required for digital signatures remain safe because the related keystores cannot be loaded without the appropriate password shares.

1.3.3 Data Protection

The keystores can be optionally exported to a physically separate backup system on a regular basis to prevent data loss e.g. due to a hard disk failure of the main system. As soon as the associated password shares become invalid after a session closure, the backup system will be notified to wipe out the corresponding expired keystore incarnations. The whole database system might be backed up when a save point is reached. A save point is reached when all expired password shares have been cleaned up and all keystore instances have become unloadable due to insufficient available password shares.

1.4 Design Principles

The design of the REST API follows Fieldings ideas where it seems applicable and desirable. The domain objects (keystores, participants, shares or slices, sessions and documents) will be mapped on resources and can be manipulated through representations (or sometimes rather instructions). The service will answer calls by a set of URIs representing possible state transfers (“Hypermedia as the engine of application state”). For example POSTing a keystore representation (or rather instructions how such a keystore should be generated) will be answered with a HTTP 201 CREATED together with a JSON representation of a keystore comtaining a “self” link identifying the new keystore resource on the server. Following the “self” link gives a more complete JSON representation together with some more URIs denoting related domain objects, e.g. session resources. Such a session resource could be used to POST documents for further processing (encryption, signing, …).

The resources of the REST API are backed by an object oriented model of the domain objects which in turn are mirroring the physical database schema.

2 Use Cases

The use cases listed below describe the “happy path” within the Postconditions section. Many things can go wrong. For example, the preconditons weren’t met or the decryption of a document fails because an authentication tag does not have matched the computed value. In these cases an appropriate HTTP 4xx code will be returned together with a message in JSON format hinting the error in more detail if possible.

2.1 Generation of Keystores

Participants may trigger the generation of PKCS12 keystores together with the desired key entries by POSTing keystore instructions. Such key entries are always related to certain algorithms, e.g. ECDSA or AES. The password needed to access the keystore will be splitted into shares and made available for the distribution between the denoted participants.

URI

/shamir/v1/keystores

Method

POST

Preconditions

(1) The actor POSTing a set of instructions must be a registered user with the necessary authorizations.
(2) The denoted participants must be known to the system.

Postconditions

(1) The keystore together with the required key entries has been created.
(2) The related password shares have been computed and (optionally) distributed.
(3) A session with state PROVISIONED has been assigned to the keystore.
(4) The server has responded with HTTP 201 CREATED together with a JSON response comprising a keystore representation.

2.2 Query all Keystores

Infos about the available keystores can be requested. The infos include, inter alia, the keystore IDs, descriptive names of the keystores, the particular sharing shemes (number of shares and threshold), the creation time and an URI to each keystore.

URI

/shamir/v1/keystores

Method

GET

Preconditions

(1) The actor must own the necessary permissions to request the infos.

Postconditions

(1) The server has responded with HTTP 200 OK together with the infos in JSON format.

2.3 Query single Keystore

Extended information about a single keystore can be requested. Besides the name of the keystore, the sharing sheme and the creation date an overwiew of the key entries (alias, algorithm, key size) will be delivered if possible. The keystore may be unloadable due to an insufficient number of available shares. Furthermore a complete list of URIs regarding related entities (sessions and participants) will be presented.

URI

/shamir/v1/keystores/<keystore_id>

Method

GET

Preconditions

(1) The actor must own the necessary permissions to request the infos.

Postconditions

(1) The server has responded with HTTP 200 OK together with the infos in JSON format.

2.4 Import of the Participants

Appropriate information about the participants and actors must be imported to system. It is out of scope to maintain an independent identity management system. Instead the system relies on a separate OpenId Connect system for this purpose. In order to relate the participants referenced by the instructions to generate a certain keystore the system must maintain or buffer some information about the approved participants.

Preconditions

(1) The participants together with their roles and corresponding permissions must be known to a OpenID Connect Provider.

Postconditions

(1) The system has created the participant entities and is able to relate them to the users maintained by the OpenId Connect provider.

2.5 Session Activation

In order to encrypt or digitally sign electronic documents a session belonging to a certain keystore must be activated by PUTting the appropriating session instructions as JSON object.

URI

/shamir/v1/keystores/<keystore_id>/sessions/<session_id>

Method

PUT

Preconditions

(1) The present session belonging to the given keystore has the state PROVISIONED.

(2) A sufficient subset of password shares required to recover the password of the keystore is available.

(3) The actor that opens the session owns the required authorizations.

Postconditions

(1) The session belonging to the keystore has got the state ACTIVE.

(2) The session is ready to encrypt/sign documents by applying key entries of the related keystore.

(3) The server has responded with HTTP 200 OK and has transmitted a session representation in JSON format to the actor.

(4) Pending documents which had been scheduled for digital signing or encryption have been processed.

2.6 Session Closure

2.6.1 Automatic Closure

During the session activation a maximum idle period had been transmitted. A concurrent thread checks periodically for idle sessions and closes them.

Preconditions

(1) One ore more sessions have been inactive for longer than the denoted idle time.

Postconditions

(1) The idle sessions have been asssigned the state CLOSED and won't process documents anymore.

(2) The related keystores have been assigned new sessions with state PROVISIONED.
(3) The related keystores have been given new passwords and the comprising key entries have been re-encrypted.
(4) Corresponding password shares have been computed and (optionally) distributed.
(5) The preceding password shares related to the idle session have been marked as EXPIRED.

2.6.2 On Demand

A Session can be closed on demand by PUTting an appropriate session instruction as JSON object.

URI

/shamir/v1/keystores/<keystore_id>/sessions/<session_id>

Method

PUT

Preconditions

(1) The actor PUTting the session instruction has the required authorizations.
(2) The session has the state ACTIVE.

Postconditions

(1) The session has been asssigned the state CLOSED and won't process documents anymore.
(2) The related keystore has been assigned a new session with state PROVISIONED.
(3) The related keystore has been given a new password and the comprising key entries have been re-encrypted.
(4) Corresponding password shares have been computed and (optionally) distributed.
(5) The preceding password shares related to the closed session have been marked as EXPIRED.
(6) The server has responded with HTTP 204 NO CONTENT.

2.7 Query all Sessions of a Keystore

Infos about the sessions belonging to a keystore can be requested. This includes already expired sessions.

URI

/shamir/v1/keystores/<keystore_id>/sessions

Method

GET

Preconditions

(1) The actor must own the necessary permissions to request the infos.

Postconditions

(1) The infos have been delivered in JSON format (HTTP 200 OK).

2.8 Submission of Documents for Review

Documents can be submitted to provisioned sessions scheduling them for digital signing or encryption.

URI

/shamir/v1/sessions/<session_id>/documents

Method

POST

Preconditions

(1) The actor must own the necessary permissions to submit the documents.

(2) The session has the state PROVISIONED.

Postconditions

(1) The document has been stored together with its metadata and can be reviewed by the participants.

(2) A metadata representation in JSON format has been returned (HTTP 201 CREATED) including an URI indicating the location on the server.

2.9 Digital Signature, Encryption and Decryption

Documents of certain media types can be digitally signed or rather verified by POSTing them to an URL referencing an active session. The actual behaviour of the service depends on the media type. A posted XML document will be digitally signed by applying XML Signature Processing whereas a PDF document will be processed by different rules. Two query parameters indicate the key entry to be used by its alias and the desired action which can be any of: encrypt, decrypt, sign or verify. The posted documents will be stored on the server. Depending on the size of the transferred document the service might decide to process the request asynchronously.

URI

/shamir/v1/sessions/<session_id>/documents

Method

POST

Preconditions

(1) The denoted session is ACTIVE.

(2) The actor has the required permissions.

Postconditions

(1) The document has been processed (encrypted, decrypted, signed or verified) and the the resulting document and its metadata has been stored.

(2) The server has responded either with HTTP 201 CREATED or HTTP 202 ACCEPTED and has transmitted a JSON response comprising the result of the operation and an URI indicating the location of the processed document if applicable.

2.10 Query all Metadata of Documents belonging to a Session

The metadata of the documents submitted to a session can be requested. The metadata include the document title, the current state – that might be pending, processed or faulty – its media type, the intended action, timestamps and the self link..

URI

/shamir/v1/sessions/<session_id>/documents

Method

GET

Preconditions

(1) The actor must own the necessary permissions to request the metadata.

Postconditions

(1) The server has responded with HTTP 200 OK together with the metadata in JSON format.

2.11 Query the Metadata of a single Document

The metadata of a single document can be requested. That includes along with the usual metadata links to all related domain objects, e.g. the link needed to retrieve the actual document content.

URI

/shamir/v1/sessions/<session_id>/metadata/<document_id>

Method

GET

Preconditions

(1) The actor must own the necessary permissions to request the metadata.

Postconditions

(1) The server has responded with HTTP 200 OK together with the metadata in JSON format.

2.12 Request the content of a Document

Often the actual content of a document needs to be fetched, most notably digitally signed documents. Encrypted document content may be retrieved to submit it again for decryption.

URI

/shamir/v1/sessions/<session_id>/documents/<document_id>

Method

POST

Preconditions

(1) The actor must own the necessary permissions to fetch the content.

Postconditions

(1) The server has responded with HTTP 200 OK together with the content as octet stream.

3 Solution

The service is organized into three logical tiers: presentation tier (view), the application tier (business logic) and the data tier (database). The presentation tier for this service is razor thin. It comprises solely the (JSON) representations of the domain objects. But following RESTful API design principles these representations are transporting the application state to the clients and might be used to build a rich internet application. The business logic layer (application tier) processes these representations by taking into account the current database state. The application tier itself is completely stateless.

3.1 External Dependencies

The actual service is a Spring based application and has been built by using Spring Boot. The most important external dependencies are Jakarta Persistence backed by Hibernate and JAX-RS (Jakarta RESTful Web Services) backed by Jersey. Incoming and outgoing representations in JSON format will be mapped on the generic object model provided by Jakarta JSON Processing. The secret sharing algorithm and corresponding keystore engine is depending on the JCA provider Shamirs Keystore. XML encryption will be implemented drawing upon Apache Santuario. Bouncy Castle contributes some cryptographic algorithms or rather procedures related to PKCS #7 (Cryptographic Message Syntax) and PKCS #12 keystores.

The development database is a MariaDB instance. Since any database access will be handled by the Jakarta Persistence API, other databases might be supported as well at a later time.

The identy management will be given to an OpenId Connect provider, namely Keycloak.

Unit and integration tests are relying on JUnit 5 together with AssertJ.

3.2 Data Model

Participants of secret sharing shemes might have got shares of several keystores and a particular keystore has at least one owner but might have been shared between several participants. A slice consists of one or more shares. Different participants might have been given a different number of shares. A slice bundles up shares into one JSON file. Therefore, participants might have slices – bundled shares – of more than one keystore and vice versa access to a certain keystore has been splitted into different slices. Furthermore, the validity of slices expires when an active session of a keystore closes. Thereupon a new batch of slices will be created for this keystore and its participants. A Keystore entity might reference several sessions. One of these sessions – the most recently created – is the present session. Newly created sessions are in state PROVISIONED. Documents scheduled for encryption or digital sign-

ing can be submitted to provisioned sessions for review. As soon as a sufficient subset of slices comprising of password shares is available the provisioned session can be switched to state ACTIVE and all pending documents will be processed. Documents submitted to active sessions will be processed at once. The metadata and the document itself will be separately stored. See Figure 1 for details.

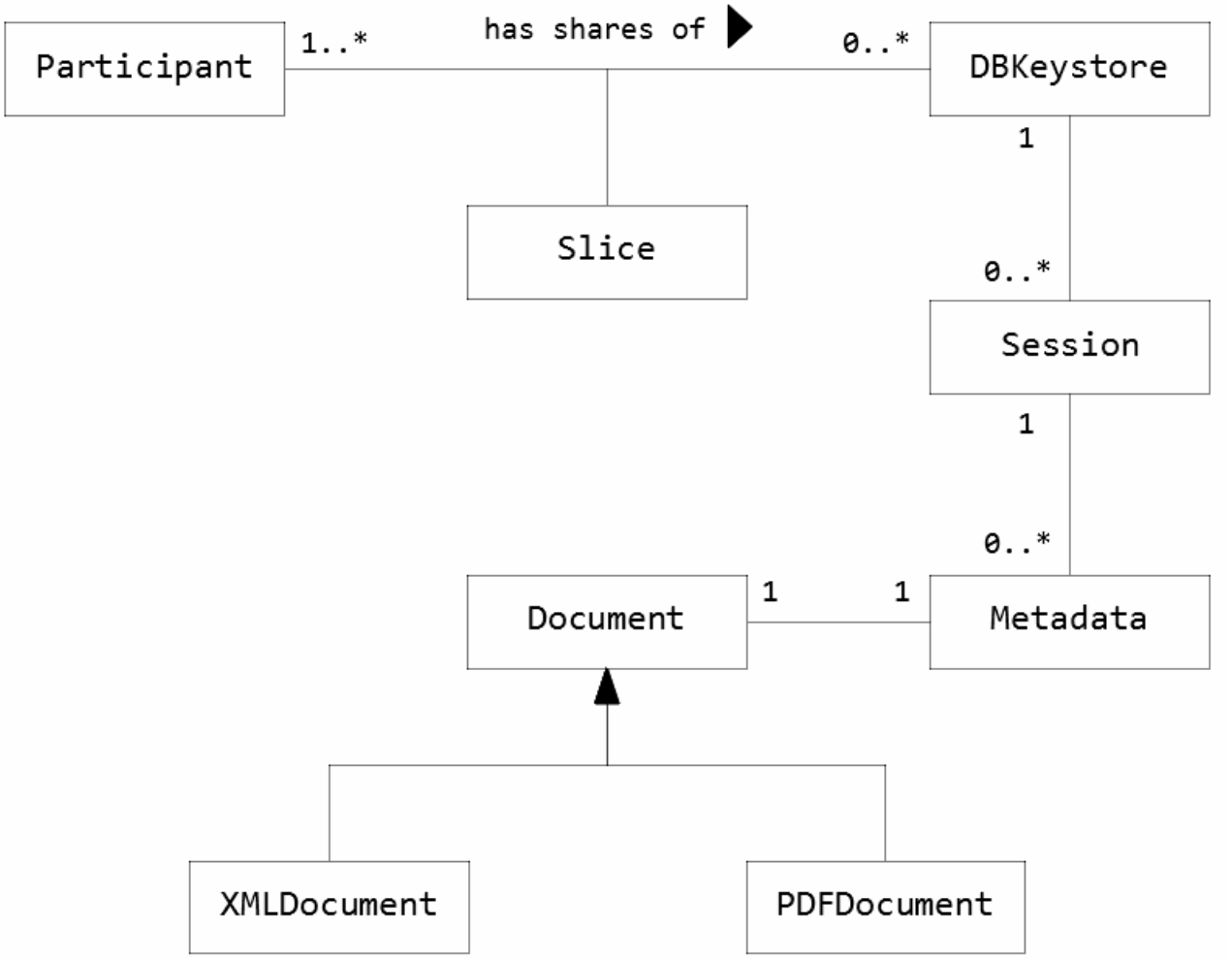

Figure 1: Data Model

3.3 Business Logic

The application tier has interfaces to the presentation tier and the back end. The classes responsible for processing the representations are called resource classes. These resources are primarily responsible to interprete the incoming instructions and delivering the appropriate presentations to the clients. In doing so, they must first sanitize the input provided as generic JSON object model. Illegal or invalid inputs result in HTTP 400 BAD REQUEST responses. POST requests usually lead to creating of domain objects, stitching them together and feeding the object graph to the database access routines. PUT re-

quests need to query the database first to process the instructions for a resource. GET requests simply query the database and deliver the representations in JSON format. The domain classes provide a `toJson()`-method which produces such a representation to a given time.

All resource classes inherit from a abstract base class which provides some utilities useful for all of them, see Figure 2.

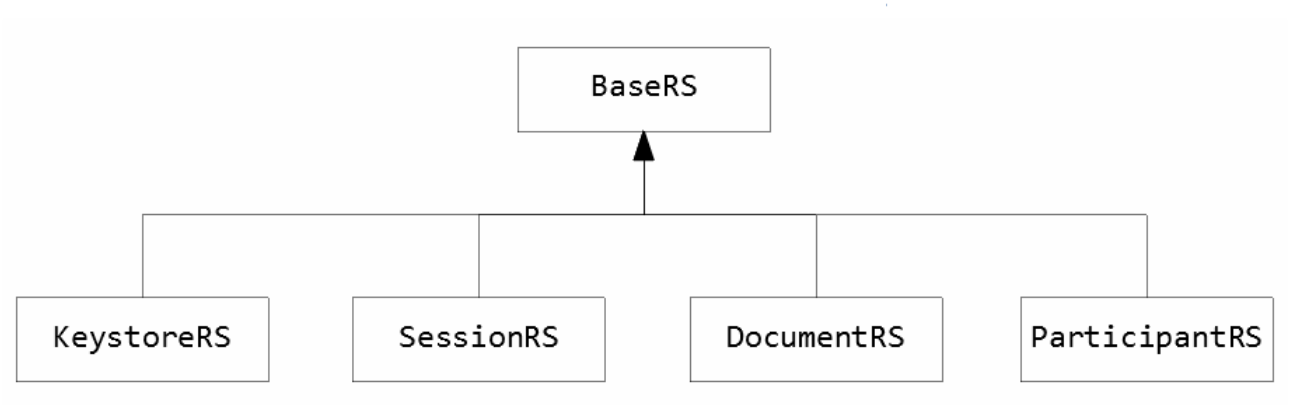

Figure 2: Resource classes

Some parts of the business logic are attached to the domain objects where convenient (rich domain model) leaving some glue code to the resource classes.

The database access routines are bundled into several services. Each of these services is spezialized in a particular domain class.

3.4 Resources and Representations

The subsequent sections are somewhat related to the corresponding sections within chapter 2 Use Cases but with special attention to parameters and responses including the actual entities transmitted within the HTTP body. The examples shown have been sampled from different test runs. The randomly created UUIDs aren't always consistent. The prefix /shamir/v1 is omitted from all URIs.

3.4.1 Keystore Resource

3.4.1.1 Keystore Generation

POST /keystores

CONSUMES application/json
PRODUCES application/json

Parameter

Type	Name	Description	Schema
Body	Keystore Instructions	Instructions for generating Keystores	object, see Keystore Instructions

Keystore Instructions

Name	Required	Schema
shares	☑	integer
threshold	☑	integer
descriptiveName	☑	string
keyInfos	☑	array, see Secret Key Info and Private Key Info
sizes	☑	array

Secret Key Info

Name	Required	Schema
alias	☑	string
algorithm	☑	string
keysize	☑	integer
type	☑	string

Private Key Info

Name	Required	Schema
alias	☑	string
algorithm	☑	string
type	☑	string
x509	☑	object, see X509

X509

Name	Required	Schema
validity	☑	integer
commonName	☑	string
locality	☑	string
state	☑	string
country	☑	string

Response

HTTP Code	Status Info	Schema
201	Created	object, see Keystore Representation

Example

Supposing for instance that we want to generate a keystore comprising a secret key entry suitable for the AES algorithm and a private key entry suitable for the Elliptic Curve algorithm. Furthermore the access to the keystore should be partitioned into 12 shares whereas 4 of them are required to recover the secret. The 12 shares are given to 7 users. One user receives 4 shares, two user receive in each case 2 shares and four user are given in each case 1 share. That would be encoded by the subsequent JSON object:

```
POST /keystores
```

```
{
  "shares": 12,
  "threshold": 4,
  "descriptiveName": "my-posted-keystore",
  "keyinfos": [
    {
      "alias": "my-secret-key",
      "algorithm": "AES",
      "keySize": 256,
      "type": "secret-key"
    },
    {
      "alias": "donalds-private-ec-key",
      "algorithm": "EC",
      "type": "private-key",
      "x509": {
          "validity": 100,
          "commonName": "Donald Duck",
          "locality": "Entenhausen",
          "state": "Bayern",
          "country": "Deutschland"
      }
    }
  ],
  "sizes": [
    {
      "size": 4,
      "participant": "test-user-0"
    },
    {
      "size": 2,
      "participant": "test-user-1"
    },
    {
      "size": 2,
      "participant": "test-user-2"
    },
    {
      "size": 1,
      "participant": "test-user-3"
    },
    {
      "size": 1,
      "participant": "test-user-4"
    },
    {
      "size": 1,
      "participant": "test-user-5"
    },
    {
      "size": 1,
      "participant": "test-user-6"
    }
  ]
}
```

The server might answer with HTTP 201 CREATED and the subsequent message body:

```
{
  "id": "e54e1637-15a5-41dd-8a2c-7352f2932dbe",
  "descriptiveName": "my-posted-keystore",
  "currentPartitionId": "dcaf16a9-80b3-4217-aa0b-63425c36a29b",
  "shares": 12,
  "threshold": 4,
  "creationTime": "2021-06-04T14:57:54.7066307",
  "mofificationTime": "2021-06-04T14:57:54.7066307",
  "links": [
    {
      "rel": "self",
      "href": "/keystores/e54e1637-15a5-41dd-8a2c-7352f2932dbe",
      "type": [
        "GET"
      ]
    }
  ]
}
```


3.4.1.2 Query single Keystore

GET /keystores/<keystore_id>

PRODUCES application/json

Parameter

Type	Name	Description	Schema
Path	keystore_id	Keystore Id	string

Response

HTTP Code	Status Info	Schema
200	Ok	object, see Keystore Representation

Example

```
GET /keystores/e54e1637-15a5-41dd-8a2c-7352f2932dbe
```

This gives the full representation of the single keystore including the key entries, provided that the keystore is loadable. The service might answer with HTTP 200 OK and the subsequent message body:


```
{
  "id": "e54e1637-15a5-41dd-8a2c-7352f2932dbe",
  "descriptiveName": "my-posted-keystore",
  "currentPartitionId": "dcaf16a9-80b3-4217-aa0b-63425c36a29b",
  "shares": 12,
  "threshold": 4,
  "creationTime": "2021-06-04T14:57:54",
  "mofificationTime": "2021-06-04T14:57:54",
  "keyEntries": [
    {
      "alias": "my-secret-key",
      "localKeyID": "54:69:6d:65:20:31:36:32:32:38:31:31:34:37:32:38:31:30",
      "friendlyName": "my-secret-key"
    },
    {
      "alias": "donalds-private-ec-key",
      "localKeyID": "54:69:6d:65:20:31:36:32:32:38:31:31:34:37:33:33:37:33",
      "friendlyName": "donalds-private-ec-key"
    }
  ],
```

```
  "links": [
    {
      "rel": "self",
      "href": "/keystores/e54e1637-15a5-41dd-8a2c-7352f2932dbe",
      "type": [
          "GET"
      ]
    },
    {
      "rel": "sessions",
      "href": "/keystores/e54e1637-15a5-41dd-8a2c-7352f2932dbe/sessions",
      "type": [
          "GET"
      ]
    },
    {
      "rel": "currentSession",
      "href": "/keystores/e54e1637-15a5-41dd-8a2c-7352f2932dbe/sessions/e7f64346-0d72-4048-aea7-0263b33d68c4",
      "type": [
          "GET",
          "PUT"
      ]
    }
  ]
}
```


3.4.1.3 Query all Keystores

GET /keystores

PRODUCES application/json

Response

HTTP Code	Status Info	Schema
200	Ok	object, see List of Keystores

List of Keystores

Name	Schema
keystores	array, see Keystore Representation

Example

Note that every retrieved keystore is outlined by the light representation without key entries and only with self links. An example response is shown below:

```
GET /keystores
```

The service might answer with HTTP 200 OK and the subsequent message body:


```
{
  "keystores": [
    {
      "id": "5adab38c-702c-4559-8a5f-b792c14b9a43",
      "descriptiveName": "my-first-keystore",
      "currentPartitionId": "467b268d-1a7f-4f00-993c-672b82494822",
      "shares": 12,
      "threshold": 4,
      "creationTime": "2021-06-07T18:01:17",
      "mofificationTime": "2021-06-07T18:01:17",
      "links": [
        {
```

```
          "rel": "self",
          "href": "/keystores/5adab38c-702c-4559-8a5f-b792c14b9a43",
          "type": [
            "GET"
          ]
        }
      ]
    },
    {
      "id": "7660d95e-4962-4fd0-8d4d-306a66c57ac6",
      "descriptiveName": "my-posted-keystore",
      "currentPartitionId": "291133b2-44e9-4a63-addc-1fdc17d351da",
      "shares": 12,
      "threshold": 4,
      "creationTime": "2021-06-07T18:01:28",
      "mofificationTime": "2021-06-07T18:01:28",
      "links": [
        {
          "rel": "self",
          "href": "/keystores/7660d95e-4962-4fd0-8d4d-306a66c57ac6",
          "type": [
            "GET"
          ]
        }
      ]
    },
    {
      "id": "e509eaf0-3fec-4972-9e32-48e6911710f7",
      "descriptiveName": "the-idle-keystore",
      "currentPartitionId": "75821741-ffce-49a6-bc94-5a2b9239a65e",
      "shares": 12,
      "threshold": 4,
      "creationTime": "2021-06-07T18:01:17",
      "mofificationTime": "2021-06-07T18:01:25",
      "links": [
        {
          "rel": "self",
          "href": "/keystores/e509eaf0-3fec-4972-9e32-48e6911710f7",
          "type": [
            "GET"
          ]
        }
      ]
    }
  ]
}
```


3.4.1.4 Keystore Representation

Name	Full	Schema
id		string
descriptiveName		string
currentPartitionId		string
shares		integer
threshold		integer
creationTime		string

keyEntries	☑	array
modificationTime		string
links		array

3.4.2 Session Resource

3.4.2.1 Query all Sessions of a Keystore

GET /keystores/<keystore_id>/sessions

PRODUCES application/json

Parameter

Type	Name	Description	Schema
Path	keystore_id	Keystore Id	string

Response

HTTP Code	Status Info	Schema
200	Ok	object, see List of Sessions

List of Sessions

Name	Schema
sessions	array, see Session Representation

Example

```
GET /keystores/5adab38c-702c-4559-8a5f-b792c14b9a43/sessions
```

The service might answer with HTTP 200 OK and the subsequent message body:


```
{
  "sessions": [
    {
      "id": "8bff8ac6-fc31-40de-bd6a-eca4348171c5",
      "phase": "PROVISIONED",
      "idleTime": 0,
      "creationTime": "2021-06-09T15:46:11",
      "modificationTime": "2021-06-09T15:46:11",
      "expirationTime": "null",
      "links": [
        {
          "rel": "self",
          "href": "/keystores/5adab38c-702c-4559-8a5f-b792c14b9a43/sessions/8bff8ac6-fc31-40de-bd6a-eca4348171c5",
          "type": [
            "GET",
            "PUT"
          ]
        }
      ]
    }
  ]
}
```

3.4.2.2 Update Session

PUT /keystores/<keystore_id>/sessions/<session_id>

CONSUMES application/json
PRODUCES application/json

Parameter

Type	Name	Description	Schema
Path	keystore_id	Keystore Id	string
Path	session_id	Session Id	string
Body	Session Instructions	Instructions for updating Sessions	object, see Session Instructions

Session Instructions

Name	Required	Schema
session	☑	object, see Session Activation or Session Closure

Session Instructions concern either the activation of provisioned sessions together with parameters denoting the maximal idle time or the explicit closure of active sessions. Therefore you have to include either an activation or a closure object.

Session Activation

Name	Required	Schema
activation		object, see Automatic Close

Automatic Close

Name	Required	Schema
automaticClose	☑	object, see Idle Time

Idle Time

Name	Required	Schema
idleTime	☑	integer
temporalUnit		string, default is “SECONDS”

Session Closure

Name	Required	Schema
closure		empty object

Response

Use Case	HTTP Code	Status Info	Schema
Activation	200	Ok	object, see Session Representation

Closure	204	No Content	

Example 1

This is an example concerning the activation of provisioned sessions:

```
PUT /keystores/5adab38c-702c-4559-8a5f-b792c14b9a43/sessions/8bff8ac6-fc31-40de-bd6a-eca4348171c5
```

```
{
  "session": {
    "activation": {
      "automaticClose": {
        "idleTime": 10,
        "temporalUnit": "SECONDS"
      }
    }
  }
}
```

The service might answer with HTTP 200 OK and the subsequent message body:

```
{
  "id": "8bff8ac6-fc31-40de-bd6a-eca4348171c5",
  "phase": "ACTIVE",
  "idleTime": 10,
  "creationTime": "2021-06-10T14:16:49",
  "modificationTime": "2021-06-10T14:16:57.6542639",
  "expirationTime": "2021-06-10T14:17:07.6542639",
  "links": [
    {
      "rel": "self",
      "href": "/keystores/5adab38c-702c-4559-8a5f-b792c14b9a43/sessions/8bff8ac6-fc31-40de-bd6a-eca4348171c5",
      "type": [
        "GET",
        "PUT"
      ]
    }
  ]
}
```

Example 2

Subsequent snippet instructs the explicit and immediate closure of an activated session:

```
PUT /keystores/5adab38c-702c-4559-8a5f-b792c14b9a43/sessions/8bff8ac6-fc31-40de-bd6a-eca4348171c5
```

```
{
  "session": {
    "closure": {
    }
  }
}
```

The service might answer with HTTP 204 NO CONTENT.

3.4.2.3 Query single session

GET /keystores/<keystore_id>/sessions/<session_id>

PRODUCES application/json

Parameter

Type	Name	Description	Schema
Path	keystore_id	Keystore Id	string
Path	session_id	Session Id	string

Response

HTTP Code	Status Info	Schema
200	Ok	object, see Session Representation

Example

```
GET /keystores/5adab38c-702c-4559-8a5f-b792c14b9a43/sessions/8bff8ac6-fc31-40de-bd6a-eca4348171c5
```

The service might answer with HTTP 200 OK and the subsequent message body:


```
{
  "id": "8bff8ac6-fc31-40de-bd6a-eca4348171c5",
  "phase": "PROVISIONED",
  "idleTime": 0,
  "creationTime": "2021-06-10T14:16:49",
  "modificationTime": "2021-06-10T14:16:49",
  "expirationTime": "null",
  "links": [
    {
      "rel": "self",
      "href": "/keystores/5adab38c-702c-4559-8a5f-b792c14b9a43/sessions/8bff8ac6-fc31-40de-bd6a-eca4348171c5",
      "type": [
        "GET",
        "PUT"
      ]
    },
    {
      "rel": "documents",
      "href": "/sessions/8bff8ac6-fc31-40de-bd6a-eca4348171c5/documents",
      "type": [
        "GET",
        "POST"
      ]
    }
  ]
}
```


3.4.2.4 Session Representation

Name	Schema
id	string
phase	string
idleTime	string
creationTime	string
modificationTime	string
expirationTime	string
links	array

3.4.3 Document Resource

3.4.3.1 Process Document

POST /sessions/<session_id>/documents

CONSUMES application/xml, application/pdf
PRODUCES application/json

Parameter

Type	Name	Description	Schema
Path	session_id	Session Id	string
Query	action	The intended processing, e.g. signing.	string
Query	alias	Points to the key to be used.	string
Header	content-type	The media type	string
Header	doc-title	Part of the metadata	string
Body	document	The to be processed document	xml | pdf

Response

HTTP Code	Status Info	Schema
201	Created	object
202	Accepted	object

Example

A xml document will be submitted to a provisioned session for digital signing. That will usually result in a created but pending document.

```
POST /sessions/8bff8ac6-fc31-40de-bd6a-eca4348171c5/documents?action=SIGN&alias=test-ec-key

Content-Type:  application/xml
doc-title:     payment-order-1
```

The service might answer with HTTP 201 CREATED and the subsequent message body:

```
{
  "id": "a1396065-dc1e-4889-8eae-b8dbca3f54a6",
  "title": "payment-order-1",
  "state": "PENDING",
  "action": "SIGN",
  "alias": "test-ec-key",
  "mediaType": "application/xml",
  "creationTime": "2021-06-11T15:32:15",
  "modificationTime": "2021-06-11T15:32:15",
  "links": [
    {
      "rel": "self",
      "href": "/sessions/8bff8ac6-fc31-40de-bd6a-eca4348171c5/metadata/a1396065-dc1e-4889-8eae-b8dbca3f54a6",
      "type": [
        "GET"
      ]
```

```
    }
  ]
}
```

3.4.3.2 Query all Metadata of a Session

GET /sessions/<session_id>/documents

PRODUCES application/json

Parameter

Type	Name	Description	Schema
Path	session_id	Session Id	string

Response

HTTP Code	Status Info	Schema
200	Ok	object, see List of Metadata

List of Metadata

Name	Schema
documents	array, see Metadata Representation

Example

```
GET /sessions/8bff8ac6-fc31-40de-bd6a-eca4348171c5/documents
```

The service might answer with HTTP 200 OK and the subsequent message body:

```
{
  "documents": [
    {
      "id": "8baf5021-c6b5-404e-a98b-f324efe4d13d",
      "title": "payment-order-1",
      "state": "PENDING",
      "action": "SIGN",
      "alias": "test-ec-key",
      "mediaType": "application/xml",
      "creationTime": "2021-06-14T14:57:22",
      "modificationTime": "2021-06-14T14:57:22",
      "links": [
        {
          "rel": "self",
          "href": "/sessions/8bff8ac6-fc31-40de-bd6a-eca4348171c5/metadata/8baf5021-c6b5-404e-a98b-f324efe4d13d",
          "type": [
            "GET"
          ]
        }
      ]
    }
  ]
}
```

3.4.3.3 Query Metadata of a single Document

GET /sessions/<session_id>/metadata/<document_id>

PRODUCES application/json

Parameter

Type	Name	Description	Schema
Path	session_id	Session Id	string
Path	document_id	Document Id	string

Response

HTTP Code	Status Info	Schema
200	Ok	object, see Metadata Representation

Example

```
GET /sessions/8bff8ac6-fc31-40de-bd6a-eca4348171c5/metadata/8baf5021-c6b5-404e-a98b-f324efe4d13d
```

The service might answer with HTTP 200 OK and the subsequent message body:


```
{
  "id": "8baf5021-c6b5-404e-a98b-f324efe4d13d",
  "title": "payment-order-1",
  "state": "PENDING",
  "action": "SIGN",
  "alias": "test-ec-key",
  "mediaType": "application/xml",
  "creationTime": "2021-06-14T14:57:22",
  "modificationTime": "2021-06-14T14:57:22",
  "links": [
    {
      "rel": "self",
      "href": "/sessions/8bff8ac6-fc31-40de-bd6a-eca4348171c5/metadata/8baf5021-c6b5-404e-a98b-f324efe4d13d",
      "type": [
        "GET"
      ]
    },
    {
      "rel": "content",
      "href": "/sessions/8bff8ac6-fc31-40de-bd6a-eca4348171c5/documents/8baf5021-c6b5-404e-a98b-f324efe4d13d",
      "type": [
        "GET"
      ]
    },
    {
      "rel": "session",
      "href": "/keystores/5adab38c-702c-4559-8a5f-b792c14b9a43/sessions/8bff8ac6-fc31-40de-bd6a-eca4348171c5",
      "type": [
        "GET",
        "PUT"
      ]
    }
  ]
}
```


3.4.3.4 Retrieve the content of a Document

GET /sessions/<session_id>/documents/<document_id>

PRODUCES application/octet-stream

Parameter

Type	Name	Description	Schema
Path	session_id	Session Id	string
Path	document_id	Document Id	string

Response

HTTP Code	Status Info	Schema
200	Ok	binary data

Example

```
GET /sessions/8bff8ac6-fc31-40de-bd6a-eca4348171c5/documents/8baf5021-c6b5-404e-a98b-f324efe4d13d
```

The service might answer with HTTP 200 OK

```
Content-Type:          application/octet-stream
Content-Length:        1070
```

and subsequent with the binary data.

3.4.3.5 Metadata Representation

Name	Schema
id	string
title	string
state	string
action	string
validated	true | false
alias	string
mediaType	string
creationTime	string
modificationTime	string
links	array

3.5 Test Plan

3.6 Deployment

4 Security Considerations

5 Privacy