

# Technical Specification

Title	Shamirs Webservice
Author	Christof Reichardt Paul-Ehrlich-Weg 1 D-63110 Rodgau E-Mail: <a href="mailto:projektstudien@christofreichardt.de">projektstudien@christofreichardt.de</a>
Created on	2021-03-05
Version	1.0.0
Status	Draft

## Table of Contents

1	Introduction.....	3
1.1	Terminology.....	3
1.2	Product Outline.....	4
1.3	Security Goals.....	4
1.3.1	Multiple-Eye Principles.....	4
1.3.2	System Administration.....	4
1.3.3	Data Protection.....	5
1.4	Design Principles.....	5
2	Use Cases.....	6
2.1	Generation of Keystores.....	6
2.2	Query all Keystores.....	6
2.3	Query single Keystore.....	7
2.4	Import of the Participants.....	7
2.5	Session Activation.....	8
2.6	Session Closure.....	8
2.6.1	Automatic Closure.....	8
2.6.2	On Demand.....	9
2.7	Query all Sessions of a Keystore.....	9
2.8	Submission of Documents for Review.....	10
2.9	Digital Signature, Encryption and Decryption.....	10
3	Solution.....	12
3.1	External Dependencies.....	12
3.2	Data Model.....	12
3.3	Business Logic.....	13
3.4	Resources and Representations.....	14
3.4.1	Keystore Resource.....	14
3.4.1.1	Keystore Generation.....	14
3.4.1.2	Query single Keystore.....	17
3.4.1.3	Query all Keystores.....	18
3.4.1.4	Keystore Representation.....	19
3.5	Test Plan.....	19
3.6	Deployment.....	19
4	Security Considerations.....	19
5	Privacy.....	19

# 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 scheme.

### 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 it must be assigned a session which in turn can only be created 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 containing 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 preconditions 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 schemes (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 scheme and the creation date an overview 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 assigned the state CLOSED and won't process documents anymore.



- (2) The related keystores have been assigned new sessions with state PROVIDED.
- (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 assigned the state CLOSED and won't process documents anymore.
- (2) The related keystore has been assigned a new session with state PROVIDED.
- (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

#### Parameter

Name	Type	Description
alias	query	Identifies the key entry to be used (required)
action	query	encrypt   sign (required)
content-type	header	application/xml   application/pdf (required)
doc-title	header	document title (required)

#### Method

POST

#### Preconditions

- (1) The actor must own the necessary permissions to request the infos.
- (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. Encrypted, decrypted or digitally signed document 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

## Parameter

Name	Type	Description
alias	query	Identifies the key entry to be used (required)
action	query	encrypt   sign   verify   decrypt (required)
content-type	header	application/xml   application/pdf (required)
doc-title	header	document title (required)

## 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 has been stored if applicable. Documents sent for verification won't be stored.
- (2) The server has responded either with HTTP 201 CREATED, HTTP 202 ACCEPTED or HTTP 200 OK and has transmitted a JSON response comprising the result of the operation and an URI indicating the location of the processed document if applicable. In the case of the verification of digitally signed documents the status code is HTTP 200 OK together with a response body in JSON format hinting the outcome of the verification.

## 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 identity 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 schemes 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 signing 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.

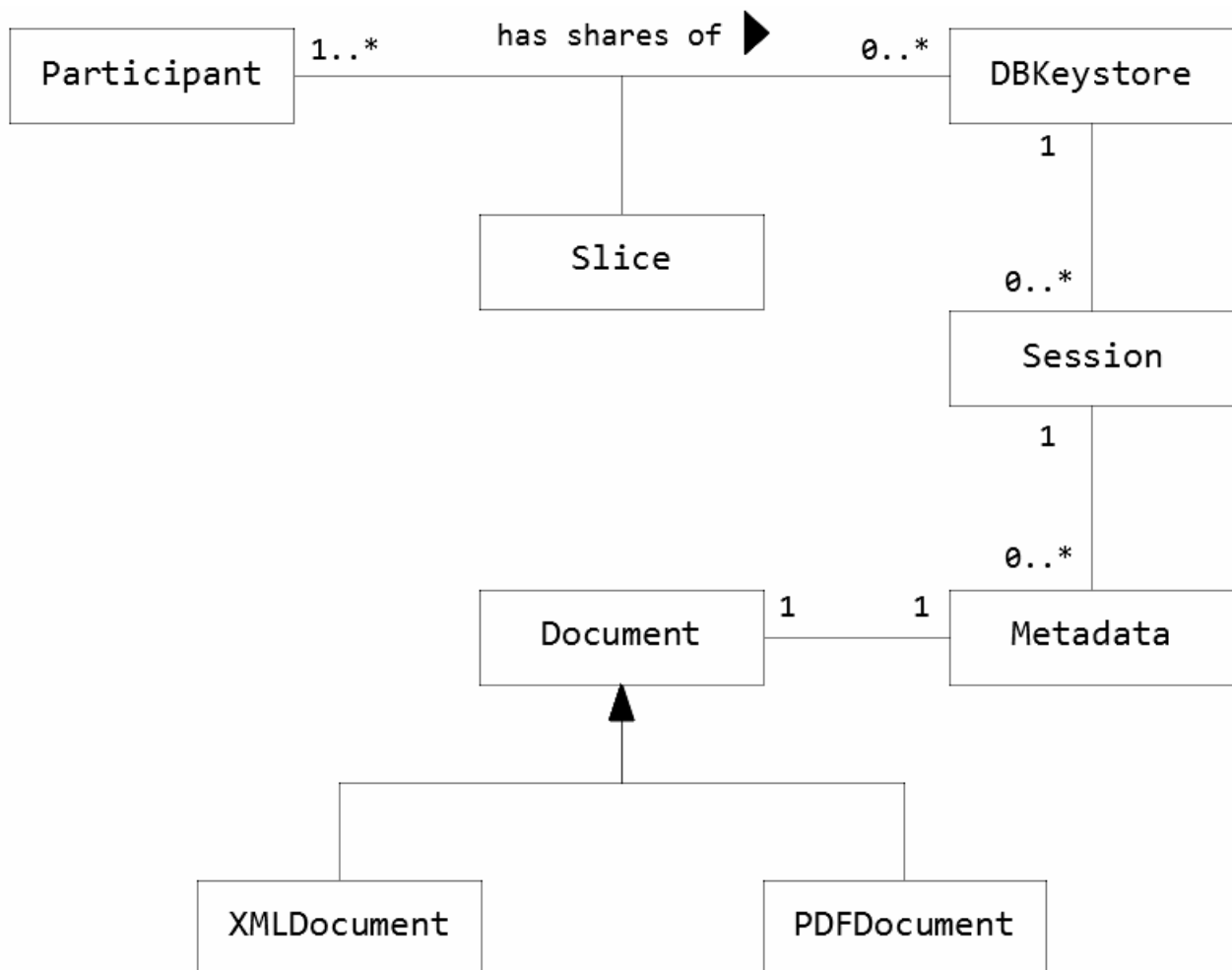


Figure 1: Data Model

### 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 interpret 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 requests 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.

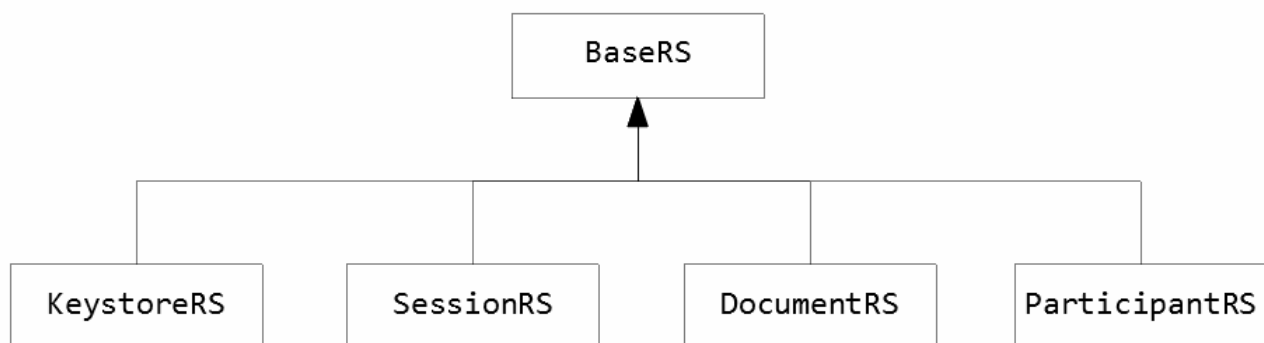


Figure 2: Resource classes

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 specialized 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 are sampled from different test runs. The randomly created UUIDs aren't always consistent.

### 3.4.1 Keystore Resource

#### 3.4.1.1 Keystore Generation

POST /shamir/v1/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	<input checked="" type="checkbox"/>	integer
threshold	<input checked="" type="checkbox"/>	integer
descriptiveName	<input checked="" type="checkbox"/>	string
keyInfos	<input checked="" type="checkbox"/>	array, see Secret Key Info and Private Key Info
sizes	<input checked="" type="checkbox"/>	array

### Secret Key Info

Name	Required	Schema
alias	<input checked="" type="checkbox"/>	string
algorithm	<input checked="" type="checkbox"/>	string
keysize	<input checked="" type="checkbox"/>	integer
type	<input checked="" type="checkbox"/>	string

### Private Key Info

Name	Required	Schema
alias	<input checked="" type="checkbox"/>	string
algorithm	<input checked="" type="checkbox"/>	string
type	<input checked="" type="checkbox"/>	string
x509	<input checked="" type="checkbox"/>	object, see X509

### X509

Name	Required	Schema
validity	<input checked="" type="checkbox"/>	integer
commonName	<input checked="" type="checkbox"/>	string
locality	<input checked="" type="checkbox"/>	string
state	<input checked="" type="checkbox"/>	string
country	<input checked="" type="checkbox"/>	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:

```
{
  "shares": 12,
  "threshold": 4,
  "descriptiveName": "my-posted-keystore",
  "keyinfos": [
    {
      "alias": "my-secret-key",
      "algorithm": "AES",
      "keySize": 256,
      "type": "secret-key"
    },
    {
      "alias": "donaldds-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",
}
```



```
"modificationTime": "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 /shamir/v1/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

An example response is shown below:

```
{
  "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",
  "modificationTime": "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 /shamir/v1/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

An example response is shown below:

```

{
  "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",
      "modificationTime": "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",
      "modificationTime": "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",
  "modificationTime": "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	<input checked="" type="checkbox"/>	array
modificationTime		string
links		array

### 3.5 Test Plan

### 3.6 Deployment

## 4 Security Considerations

## 5 Privacy