



# ZeroKit

Tresorit encryption platform

API version 4.0 | SDK version 4.0

January 2017

## ~ Table of contents ~

### *~ Table of contents ~*

#### **1. Overview**

#### **2. Web based example**

#### **3. Basic concepts**

#### **4. JavaScript SDK**

- 4.1 User Management
- 4.2 Tresor management
- 4.3 Encryption/Decryption
- 4.4 Invitation Links
- 4.5 Customization

#### **5. Administrative API**

- 5.1 Authentication (request signing)
- 5.2 API reference

#### **6. Built-in IDP**

#### **7. Common flows**

#### **8. Changelog**

- 8.1 V1-V2 Upgrade action list

## 1. Overview

Tresorit offers an authentication and encryption platform (ZeroKit), that is unique in terms of the level of security among other solutions on the market. We employ *zero-knowledge* techniques in our stack to provide the highest available security, while maintaining a simple, easy to use interface for application developers. We solve problems for:

- Companies who handle healthcare, financial or personal data and wish to remain compliant and safely store the data of their customers.
- For application providers as they no longer have to trust the cloud provider and can be sure that application developers cannot make mistakes with leaking important data.

Our SDK allows application developers to encrypt the data at the moment of creation - right in the browser or mobile device of the end-user. The data never travels to the cloud in a non-encrypted form; thus neither your cloud provider, nor your application developer, nor Tresorit, nor potential hackers are able to interpret the data, even if they gained access to it. Our goal is to provide a holistic answer to data encryption: covering user identity, password and key management, data encryption on client devices, in transit and at rest.

Based on the industry standard practices, we can define "levels" of security that a cloud file storage provider can offer to its customers:

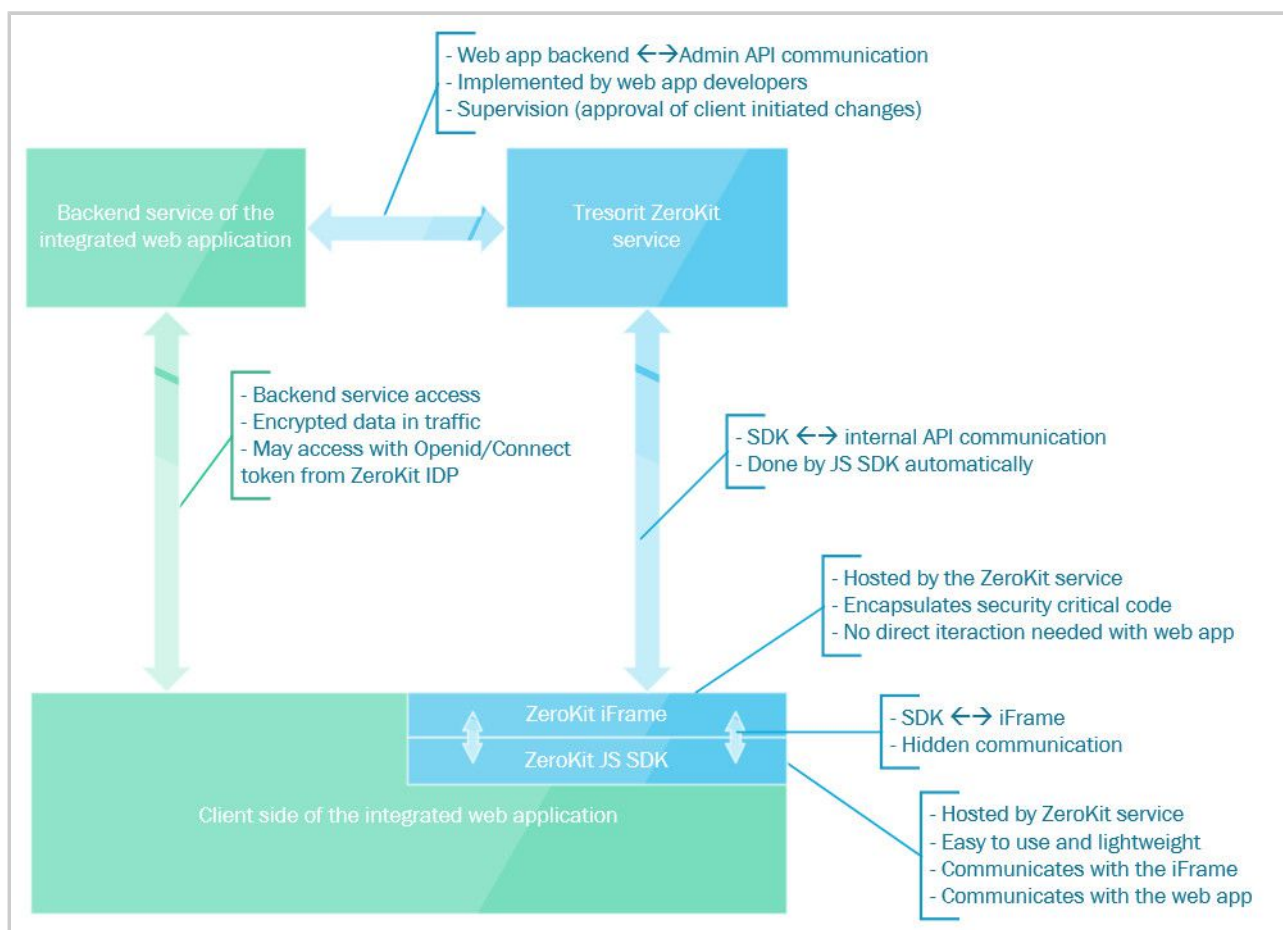
1. **No encryption at all:** Fortunately this is not common, but an example of this would be an FTP server.
2. (Transport) **channel encryption:** The communication channels between the end user's device and the service provider's server are encrypted, most commonly by utilizing HTTPS (SSL/TLS) communication protocol. This way malicious people cannot eavesdrop the communication (for example by listening to public WiFi hotspot communications). However, your data is stored on the servers in plain text. So the employees of the company can access files, or much worse, if hackers can break into their systems, they could also access all data.
3. **At rest encryption:** This adds encryption to the server side: the service provider encrypts the data after receiving it and before storing it. This gives additional protection against hackers, because it would not be enough for them to access stored files, but they would have to obtain the encryption keys as well. This is not impossible, but makes hacking harder. Note, that in this model the encryption keys are handled by the service provider, so their administrators, scripts, etc. are still able to see files in an unencrypted format.
4. **Client side encryption:** The encryption happens on the end-users device, only they have access to the encryption keys. The files are protected throughout the whole journey: they leave the computer, get transferred to the server, stored there and finally downloaded again to another device. Please note that this alone would leave channel and at rest encryption unnecessary. Also, it's clear that client side encryption is secure only if the encryption keys are handled properly. Sending or sharing the encrypted keys in any insecure manner (eg. via email or unprotected transport channel) would be equivalent of sending the files the same insecure way. Managing and sharing encryption keys securely becomes a challenge with client side encryption.
5. **Zero knowledge encryption:** This scheme does not add additional security in the common sense, but rather guarantees that client side encryption is used properly. Zero knowledge algorithms and protocols ensure that no keys, passwords, files, or any sensitive material ever get transferred in an unencrypted or reversible form. There is no point in time when encryption keys or unencrypted files are visible to the servers or service administrators. This rules out another attack factor: the service provider.

The ZeroKit SDK offers zero-knowledge encryption, because the keys and passwords of the end-users never leave their computer in a way that it would be readable to eavesdroppers, service administrators or even the developers of the application.

## Suggested architecture

The ZeroKit tenant server provides basic user management with authentication, an internal zero-knowledge key management service, and an OpenID Connect identity provider. This is not a full application, only a building block that an integrating application can use. It's important to know, that the tenant server doesn't store any user data besides the minimum required for authentication and encryption. The application backend must store all additional data on it's own. This data usually consists of at least the needed tokens for the registration flow (7. Common flows), the username-userid mapping, list of tresors, the structure of shared tresors and all application specific encrypted data.

The ZeroKit Web SDK (referred to as `zkit_sdk` in the source) is hosted on the tenant server by default but can be hosted on the application server as well.



## 2. Web based example

The web based example can be downloaded along with the mobile examples from the administrative portal or from GitHub (<https://github.com/tresorit/ZeroKit-simple-example>).

### What's in the example

- HTML5 application which uses the 0-Kit library
- A Node.js backend program which
  - Serves the previous applications static files
  - Provides a minimal and functional API

### Supported browsers

We recommend to use the latest Chrome or Firefox for testing.

A full list of supported browsers:

- Chrome 38+
- Firefox 35.0+
- Microsoft Edge (25.10586+)
- Safari 8+
- Internet Explorer 11

### Setting up the example application

You can get a customized downloader script on the management portal that you should run with node. It downloads the example template from GitHub and generates your own example app.

1. Download script from management portal
2. Place inside empty folder
3. Run script with node e.g.: `node generate_example_tenantId`
4. Move the generated "app" subfolder to a place of your liking or leave it in place
5. Step into the app folder
6. Run: `"npm start"`

This will start serving your own zero-knowledge example application of your own tenant on port 3000.

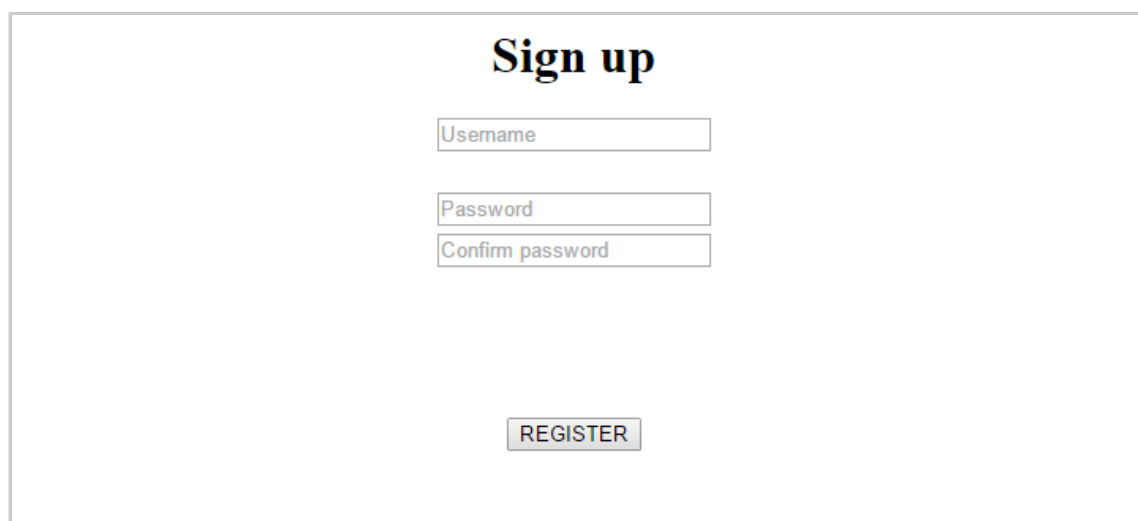
## Using the example

1. Create a new example user of the example application
  - a. Go to the Registration page



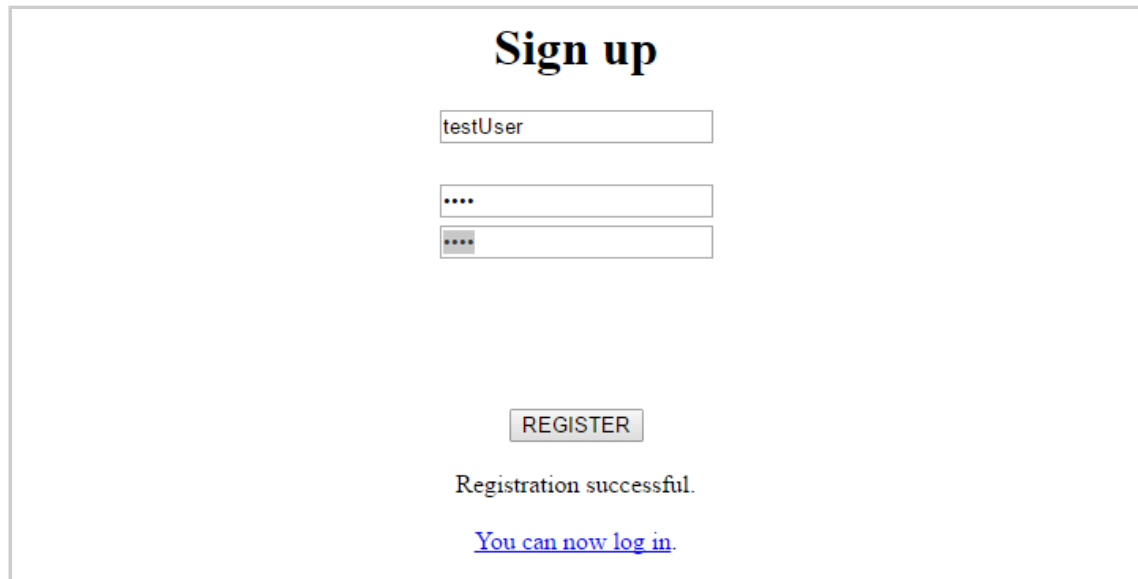
A screenshot of a web form titled "Please sign in". It contains two input fields: "Username" and "Password". Below the fields is a "SIGN IN" button. At the bottom, there is a link that says "Don't have a user yet? [Sign up!](#)".

- i. Navigate to ``http://localhost:3000/`` in your favorite browser (See a list of supported browsers at the beginning of this chapter)
  - ii. Click Sign up
- b. Provide a username and a password
  - i. You can keep it simple, these credentials are only for testing
- c. Click on the Register button



A screenshot of a web form titled "Sign up". It contains three input fields: "Username", "Password", and "Confirm password". Below the fields is a "REGISTER" button.

- d. Shortly after, you should see a "Successful Registration!" message near the bottom of the page.



**Sign up**

testUser

....

....

REGISTER

Registration successful.

[You can now log in.](#)

e. Repeat the registration process once more to register a second user.

2. Log in with the user

- Click on the Log in link on the bottom of the registration page, or hit back in your browser.
- Enter the login information into the form



**Please sign in**

testUser1

....

SIGN IN

Don't have a user yet? [Sign up!](#)

c. Click Login

3. Click Encrypt on the logged in page

Logged in as 20161012090748.ek2nd2j7@zkaastest.tresorit.io

[Encrypt](#) or [Decrypt](#)

4. Create a tresor: a tresor is an encrypted, shareable, revocable keychain that you can use to encrypt data shared with a group of users.

Logged in as user 20161012090748.ek2nd2j7@zkaastest.tresorit.io

## Let's encrypt!

### Step 1: Create a tresor.

A tresor is a keychain that that you can share with other users.

CREATE TRESOR

- a. Simply click on the "Create tresor" button
- b. Step 2 will appear with the Tresor id already filled

### Step 2: Enter a line of text to encrypt

Tresor ID:

0000o4jdrfouqh3e6uasjvc5

Text to encrypt:

ENCRYPT

5. Create your first encrypted message



- a. Put some secret message in the text area

Logged in as user 20161012090748.ek2nd2j7@zkaastest.tresorit.io

## Let's encrypt!

**Step 1: Create a tresor.**

A tresor is a keychain that that you can share with other users.

**Step 2: Enter a line of text to encrypt**

Tresor ID:

Text to encrypt:

- b. Click on Encrypt
- c. A new, seemingly random, long, base64 encoded text will appear in the textbox below

Logged in as user 20161012090748.ek2nd2j7@zkaastest.tresorit.io

## Let's encrypt!

**Step 1: Create a tresor.**

A tresor is a keychain that that you can share with other users.

**CREATE TRESOR**

**Step 2: Enter a line of text to encrypt**

Tresor ID:  
0000o4jdrfouqh3e6uasjvc5

Text to encrypt:  
testText

**ENCRYPT**

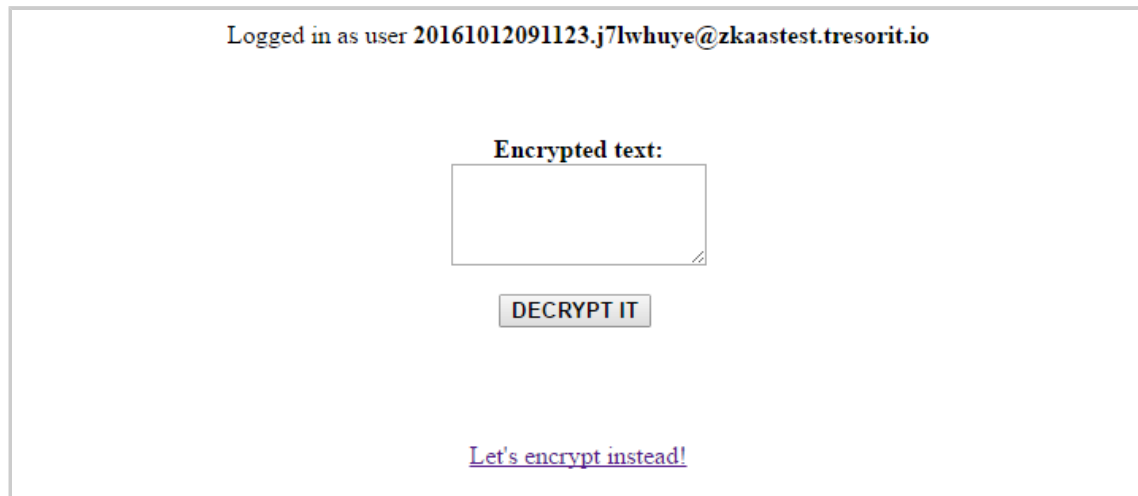
**Encrypted text:**  
AQEwMDAwbzRqZHJmb3VxaDNlN  
nVhc2p2YzUA8qpw73AEM5Mwr  
Unj6y0W0wJppDuxsFy6C1TDxf  
N/JnQzNrjdNgbLQ==

**TEST DECRYPT**

**Share the Tresor with somebody else?**  
Enter UserID **SHARE TRESOR**

- d. Click on "Test Decrypt"
  - e. The original plaintext message you entered will appear below the button
  - f. Save the encrypted message for later use, for example copy it to a textfile or put it on your clipboard
6. Try to decrypt with another user

- a. **Open your browser in Incognito/Private browsing mode.** Important: don't just open a new tab in your current browser as it will use your current login session, and now we'll want to log in simultaneously with your second user. If you cannot start a new session easily, fully close your current browser and log in with the second user.
- b. Register a new user if you didn't register a second user and log in as before
- c. After log in click decrypt and the decryption page should appear



Logged in as user 20161012091123.j7lwhuye@zkaastest.tresorit.io

Encrypted text:

DECRYPT IT

[Let's encrypt instead!](#)

- d. Paste the encrypted text into the text area and click Decrypt It
  - e. You should look out for a message below the button to see the decryption fail
  - f. Save the id of the second user you
7. Share the first user's tresor with the second user

- Bring the previous browser window into focus
- Paste the id of the second user into the input below Share the Tresor and hit the Share Tresor button.

Logged in as user 20161012090748.ek2nd2j7@zkaastest.tresorit.io

## Let's encrypt!

**Step 1: Create a tresor.**

A tresor is a keychain that that you can share with other users.

**Step 2: Enter a line of text to encrypt**

Tresor ID:

Text to encrypt:

**Encrypted text:**

```
AQEwMDAwbzRqZHJmb3VxaDNlN  
nVhc2p2YzUA8qpw73AEM5Mwr  
Unj6y0W0wJppDuxsFy6C1TDxf  
N/JnQzNrjdNgblQ==
```

**Share the Tresor with somebody else?**

Shared

- If the operation was successful a 'Shared' message should appear below the input.
- Now the second user should be able to decrypt the same encrypted message which it couldn't before, to test this bring the second window into focus and hit Decrypt again (without refreshing)



- e. The originally entered text should appear below the button.

## 3. Basic concepts

When planning and developing the security features of your product, it is crucial to know what tools the 0-kit does provide and to understand the underlying concepts.

### 3.1 Architecture

Since encryption and key handling is a complicated subject, it is important to agree on the names of each part of the architecture, so no key gets misplaced or leaked to a third party.

- **Application server:** This is the backend part of your application that handles communication with our Administrative API to approve or reject user actions on the Tenant server. It may be made up of multiple servers, but for simplicity's sake, we consider it a single unit in this documentation.
- **Application client:** When we talk about the frontend or client part of your application, it means all the code running on the users' machine.
  - **Embedding domains:** This setting is stored on the Tenant server but defines the domains where the iframes accept messages from. This can be set on the administrative portal
- **Tenant Server:** This is where your instance of the 0-Kit infrastructure resides. This server hosts the APIs, the service-access iframes and the related data needed to identify and manage the keys of users. This infrastructure is owned by Tresorit and used exclusively by the subscriber and their users during the subscription period. These instances run on a dedicated infrastructure and have a unique API domain address. All user registrations, operations and shares are available strictly within that single tenant.
  - **AdminKey:** The admin key is a 64 character long string that encodes your key that can be used to sign requests to the Administrative API. This is the hexadecimal representation of a 32 byte random key. This secret belongs to the application server and should never be shared with any untrusted parties or any of your clients. If your key has been compromised, you should regenerate it as soon as possible on the administrative portal to prevent abuse. It is important to note, that this can't be used to access or decrypt user data but can be used to seriously disrupt the operation of your applications on all your clients.
  - **TenantId:** Your tenant ID is a 3 to 10 character long, lowercase code, which uniquely identifies your tenant. It is part of your service access URL and your user IDs.
  - **HostId:** This ID identifies the host on which your tenant is hosted.
  - **Administrative API:** The Administrative API is a REST API provided by the Tenant Server. It provides the application server control over user actions: it is used to approve or reject user actions like registration, tresor creation and sharing.
  - **SDK:** The SDK is a small JavaScript code that handles the adding of the service access iframes to your site and the communication with them through a message passing interface.

#### 3.1.1 Responsibilities

- **Password handling:** User passwords never leave the iframes provided by 0-Kit and are never entered or used in the application itself. Even inside our implementation they are used immediately to derive secure keys and are not stored.
- **Encryption keys:** Encryption keys never leave the iframes, they are handled entirely by 0-Kit.
- **Storing data:** Storing data is the responsibility of the application, the tenant server only stores the data needed for authentication and encryption.

## 3.2 Integration

It is important to understand that 0-Kit is not a stand-alone application but rather a set of building blocks and tools to build secure applications, therefore it needs to be integrated. While 0-Kit provides many features that simplify implementing secure user-handling and client-side encryption, it will still take some effort to use it in your product.

### 3.2.1 Iframes

In the SDK, we do all computations and store all sensitive data inside iframes. This is important because modern browsers provide complete separation for iframes loaded from different domains, and accessing them is only possible through a strictly message passing interface. Since the iframes are loaded from your tenant server they are completely separated from all third party code, this way we can ensure that no attacker can gain access to your users' passwords or encryption keys by XSS, and it also provides a separation of responsibilities that is conducive to achieving zero-knowledge.

All the iframes can be customized by uploading custom CSS files through the administrative API, and dynamically through a set of customization methods adding and removing CSS classes, and setting placeholder texts.

There are a few types of iframes used in the SDK:

- **Registration:** this iframe contains only two password fields. It is responsible for getting the user's password and registering the user. It provides methods to query meta-information about the password input (matching, password strength) and to commit a registration to the 0-Kit system. Registering with 0-Kit is slightly more complicated than usual: for more details see [7. Common flows](#).
- **Login:** the login iframe contains a password input field, and it provides a method to initiate the login process.
- **CreateInvitationLink:** this iframe provides the password input fields for invitation link creation and a method to create password protected links.
- **AcceptInvitationLink:** this iframe contains the password input and the necessary method to accept password protected invitation links.
- **ChangePassword:** this iframe contains an input for the current and 2 password input for the new password, this is needed, because a password change operation needs a fresh login.
- **Api:** this iframe does most of the heavy lifting in the system and is responsible for all other operations beside the above. It provides many methods, for a detailed documentation see [4. JavaScript SDK](#).

### 3.2.2 SDK

The SDK can be loaded from the tenant server with the URL we provide. This is a fairly small script that handles loading and wrapping the iframes in custom objects that provide simple interfaces. Loading the script will inject a `zkit_sdk` object into the global namespace; this object can be used to access the features provided by 0-Kit.

### 3.2.3 Administrative API

The administrative API provided by the tenant server is a REST API that can be used to control user actions in 0-Kit. Most user actions, everything besides login and en/decryption require approval from the application server to be effective. This includes registration, which has a slightly different than usual flow (documented in [7. Common flows](#)). Approvals could be done manually, but we recommend some kind of automated process to handle this, since for most operations approving one can invalidate other pending operations. Every request against this API has to be signed by the AdminKey you received at the time of your purchase. The available methods are documented in [5.2 API reference](#), and the exact signing procedure is documented in [5.1 Authentication \(request signing\)](#).

The administrative API is the application's way to enforce its ACL, it allows the application to control which operation a given user is able to perform, e.g. it can be controlled *who can share what with whom*. Another important feature of this API is that it provides transactional control and ensures that the server knows about significant user actions.

## 3.3 Users

The user concept of 0-Kit is a really bare-bones one: we only store the necessary data for authentication and key management. We don't store passwords and we store no profile data.

### 3.3.1 Registration

During registration the user's password is entered in the registration iframe which exposes a register method that is used as a part of the registration process, committing the registration to the 0-Kit database. The SDK only provides meta-information about the password, the iframe's wrapper object provides methods to query whether the entered passwords match and the strength of the password is acceptable. We use zxcvbn (<https://github.com/dropbox/zxcvbn>) to measure password strength.

### 3.3.2 Login

To log in, users enter their password into the login iframe and your application provides the user ID when calling the login method. This login is only effective in the SDK. A logged in user can en/decrypt data and initiate other user actions provided by the SDK. This alone however is not sufficient to most applications, using our IDP is required in most cases. This is because while the SDK returns the ID of the logged in user, it provides no way of proving it to the application server, so a malicious user could impersonate someone else just by knowing the user ID. This doesn't affect the security of any encrypted data, because without the password of the impersonated user the attacker cannot decrypt any data, however it could still cause problems.

### 3.3.3 IDP - Identity Provider

IDP makes it possible for users to prove their identities to the server. This is necessary because the application doesn't handle any login data and has no way of ensuring the users are who they say they are. In our case the only information we provide about users is the identification information as we store no other profile data, and since we only provide this for the application there is no consent screen involved. In most use-cases server side identification is a must and so is using IDP. This is documented in detail in [6. Built-in IDP](#).



### 3.4 Tresor

Tresors are the basic units of key management in 0-Kit. They are basically groups that share a history of common keys that can be used to en/decrypt data belonging to the tresor. Using this simple concept 0-Kit covers up a great deal of complexity stemming from shareable encryption.

#### 3.4.1 Members

A tresor can have multiple members, and all current users can decrypt data encrypted by the tresor and encrypt using the current key of the tresor. Access level of tresor members (ACL) is implemented and controlled by the application - through admin API approvals and/or rejects.

#### 3.4.2 Who can access/decrypt the data?

As data is stored by the application, access to data is controlled solely by the application's ACL, 0-Kit only provides access control for keys. Current members of a tresor can download and decrypt its keys and then use it to decrypt any data that was encrypted by that tresor at any time. It is important to notice that - cryptographically speaking - anybody who has had access to a chunk of data could have saved it, so will have 'access' to it in the future. This is also true of the keys used: while we don't allow users to download even the encrypted keys of tresors that they are not members of, any user could potentially save the keys they had access to and use it later to decrypt any data encrypted by those keys. Refreshing keys is handled by 0-Kit, no user action is needed after removing a user from a tresor.

#### When to re-encrypt?

Cryptographically speaking if you once had access to a chunk of data, you know that version of it, so in general there is no need to re-encrypt anything. If somehow the removed user gains access to that bit of data again, the only new information they could get from it is that it hasn't changed. Re-encrypting may prevent this, but it can be a costly operation and in most cases it is not recommended.

#### 3.4.3 Example

Let's use the example of an encrypted chat service. In this case, the user has many different tresors. First, each user would have a private profile tresor, encrypted with this the application can store data that enhances the user experience, like preferences, but not necessarily public like channel memberships and contacts. In addition to this, each user could have a public profile tresor, something that would be shared with all contacts containing a profile picture and a name. Lastly, each channel would have its own tresor and all messages on the channel would be encrypted by that tresor. Users invited to a channel are added to it by a member of the channel/tresor. Permission to invite users into the channel should be managed by both appropriate tresor permissions e.g.: only channel admins/owner gets the necessary Administrator permission, and the applications own ACL, only approving the appropriate invitations. This way the history of messages and all profile data can be stored safely and securely in the application's database. Users could access all needed data, and have great control over what they share with others. Chat history is accessible by anyone who was a member of the channel after the message was sent, access to this history has to be controlled by the application.

### 3.5 Secrets & Keys

This section will give you a high-level overview of what keys are used in the system and grant a brief glimpse into the complexity covered up by 0-Kit.

### 3.5.1 User keys

Users own a fairly big set of keys:

- **Master key:** a key derived from the password, used to encrypt the cryptographic profile of the user and used to decrypt this profile upon login.
- **Additional keys:** these symmetric encryption keys are stored in the above mentioned profile and are used to encrypt different 'fragments' of information belonging to the user. These fragments are stored server side.
- **Agreement key pair:** This private-public pair of keys can be used to grant the user to access to tresors. The private key is stored in a fragment that the user can access and decrypt after logging in and the public key is public information that other users can query.

### 3.5.2 Sharable encryption

Inside the tresor there are a few different kinds of keys:

1. **Encryption key history:** this list of keys is stored encrypted in the tresor, and has a new key added to it after removing a user from the tresor. Data encrypted by the tresor is always encrypted by the latest key in this list.
2. **Common key:** a key used to encrypt the above key history, that every member has access to and is refreshed after adding/removing users. An encrypted version of this is stored in the tresor for each member, encrypted by their on keys.
3. **Key encryption key:** the key encryption key (KEK) is different for each member of the tresor, and is used to encrypt the common key. This is still stored in the tresor, but encrypted with the public agreement key of the user it belongs to.

You can add or remove users to/from this structure and share keys with them as you like. These are done in the following way:

#### Adding a new user:

1. Add the new member to the tresor, save the user ID and public agreement key
2. Generate a new common key
3. Generate a KEK for all members including the new one
4. Encrypt the common key by the KEK of each member
5. Encrypt the KEK for each user with their public agreement key
6. Re-encrypt the key history with the new common key
7. Upload the updated tresor data

### Removing a user:

1. Remove the member from the tresor
2. Generate a new common key
3. Generate a KEK for all current members
4. Encrypt the common key by the KEK of each member
5. Encrypt the KEK for each user with their public agreement key
6. Add a new key version to the key history
7. Re-encrypt the key history with the new common key
8. Upload the updated tresor data

## 3.6 Under the hood

### 3.6.1 WebCrypto and support in different browsers

[WebCrypto](#) is a low level API that we use in our SDK to access native cryptographic functions. This is really important as it provides us with a cryptographically safe random generator that is crucial to the security of our product. Also, native implementations are much faster than JavaScript ones, so it is also important from a performance viewpoint.

### 3.6.2 Key types

- **Keys from the user password:** we use PBKDF2 and scrypt to derive a safe key from the password
- **Asymmetric keys:** we use elliptic keys from the X25519 curve as asymmetric keys
- **Symmetric keys:** our symmetric keys are, if not derived, random strings generated using WebCrypto or msCrypto

### 3.6.3 Used algorithms

- **Login:** When logging in we use SRP protocol to agree on a session key that is used to sign requests to the Tenant Server
- **Encryption:** All data is encrypted using AES256-GCM

## 4. JavaScript SDK

### Components

#### Iframes

Most of the ZeroKit JavaScript code runs inside iframes hosted on the Tenant Server. This is to ensure the separation of security critical data e.g.: keys and authentication information and to provide protection against certain XSS attacks. This way the separation isn't done only on the JavaScript level: since the code runs inside an iframe, hosted on a different domain, the browser itself ensures separation, so no code running in the application can get access to keys or other data inside, including accidental leaks/uploads by the application itself.

The iframes that have a user interface can all be customized by providing your own css files and they all provide a common set of customization methods, described in [4.5 Customization](#).

#### SDK

The SDK itself is a fairly small script hosted on the Tenant server. This handles loading the iframes, communicating with them through messages, and checking the origin of every message to increase security. Every operation is asynchronous, and returns promises, so it won't block the UI.

### Setup

Setting the SDK up is an easy 3 step process:

1. Load the JavaScript file from the provided URL (see the example below). You may need to modify the version number later. This will inject `zkit_sdk` into the global namespace.
2. Configure SDK by calling the `setup` method. The first argument is the origin of your tenant. The second is the path to your tenant. (see the example below). You can get these by splitting up the service url you can get on the management portal. Please mind the lack of `'` at the end of both parameters.
3. The iframe will be loaded asynchronously before the first call.

```
<script
src="https://host-${hostId}.api.tresorit.io/tenant-${tenantId}/static/v4/zkit
-sdk.js"></script>
```

```
zkaas_sdk.setup(`https://host-${hostId}.api.tresorit.io`,
`/tenant-${tenantId}`);
```

## 4.1 User Management

### Registration

User registration is a 3 step process and only a part of it is done through the SDK. Here we only discuss the SDK related parts, the full registration flow is documented in detail in [7. Common flows](#).

### Registration iframe

This iframe contains two input fields to enter and confirm the user's password. It should be included in the registration form, either added dynamically by the SDK, or insert it manually into the HTML document and then wrapped by the SDK. The SDK wrapper handles the communication with the code running in the iframe and provides the methods below. This wrapped object provides a strict interface to make sure no code running on your site can access the actual password, only some meta-information about it. Through it, the application can register the user on the Tenant Server.

```
var zkitReg =  
zkitSdk.getRegistrationIframe(document.getElementById('parentElement'));
```

or

```
<iframe  
src="https://host-`${hostId}`.api.tresorit.io/tenant-`${tenantId}`/static/v4/embedded-register.html" id="zkitRegIframe"></iframe>  
<script type="text/javascript">  
  var zkitRegSDK =  
  ZkitSdk.wrapRegistrationIframe(document.getElementById('zkitRegIframe'));  
</script>
```

**Methods of the RegistrationIframeObject:****register(regSessionId: string, userId: string): Promise<{RegValidationVerifier: string}>**

This method will read the password fields inside the iframe and check if they match, then registers the user with the provided userId. The regSessionId is used to make sure that your application started the registration process and that the user matches the one that the application started the registration for. The returned promise resolves to the regValidationVerifier which is used during user validation, so it should be saved on the application server.

**Parameters:**

- **regSessionId:** The regSessionId provided by the InitUserRegistration API call for the given alias (see [5. Administrative API](#) for details)
- **userId:** The userId provided by the InitUserRegistration API call for the given alias

**Returns:**

**Promise<{RegValidationVerifier: string}>:** This method returns a Promise, that resolves to an object, with a RegValidationVerifier property.

**Rejections:**

Code	Reason
RegSessionNotExists	The provided regSessionId is invalid
UserIdMismatch	The provided userId and regSessionId doesn't match

**checkPasswordsMatch(): Promise<bool>**

This methods check if the passwords match.

**Parameters:****Returns:**

**Promise<bool>:** True if the passwords entered in the iframes input fields match.

**getPasswordStrength(): Promise<PasswordMetric>**

This methods gives meta-information about the password the user entered. Currently returns the length of the password.

**Parameters:****Returns:**

**Promise<PasswordMetric>**: Part of the result of running zxcvbn on the password.

### **login(userId, function(userId: string): Promise<string>**

This method is completely analogous to the login method of the LoginIframe and it is provided so that the user could log in after registration, but it has to be noted, that the user has to be validated before it is able to log in. It tries to log the given user in with the password entered into the iframe. This method returns a promise that will resolve to the userId of the logged in user.

#### **Parameters:**

- **userId**: The userId or alias of the user to log in.
- **callback**: This callback will be called after the user successfully logs in. It's function is described in the summary.

#### **Returns:**

**Promise<string>**: Resolves to the userId of the logged in user. Important: will never resolve if this is done as a part of the IDP login flow, the SDK will redirect instead.

#### **Rejections:**

Code	Reason
InvalidAuthorization	Invalid username or password
UserNotExists	The user does not exist
UserNotValidated	The user wasn't validated before logging in

### **PasswordMetric**

The result of the getPasswordStrength method of the registration, password change and link creation iframes. It shows the length, the strength of the passwords and gives estimates of time number of guesses and time required to crack the password. We calculate this by running zxcvbn (<https://github.com/dropbox/zxcvbn>).

#### **Fields:**

- **length**: Integer showing the length of the password entered
- **score**: The zxcvbn score of the password: integer from 0 to 4, for exact definitions see the link above
- **guesses\_log10**: The log10 of the estimated number of guesses needed to crack the password.
- **crack\_times\_seconds**: The time required to crack the password in different scenarios, based on the above estimate.
- **feedback**: Some feedback provided by the library.

## Login

There are multiple types of login, here we only discuss the SDK related parts. Login through IDP and an overview of the different login flows available is provided in [7. Common flows](#)

### LoginIframe

The login iframe contains a password input field. It should be included in the login page, either added dynamically by the SDK, or inserted manually into the HTML document and then wrapped by the SDK. The SDK wrapper will handle communication with the code running in the iframe and provide the methods below. Through this object the application can start the client login process. This login will persist across tabs and refreshes.

```
var zkitLogin =  
zkitSdk.getLoginIframe(document.getElementById('parentElement'));
```

or

```
<iframe  
src="https://host-`${hostId}.api.tresorit.io/tenant-`${tenantId}/static/v4/embedded-register.html" id="zkitLoginIframe"></iframe>  
<script type="text/javascript">  
  var zkitLogin =  
  zkitSdk.wrapLoginIframe(document.getElementById('zkitLoginIframe'));  
</script>
```



## Methods of the LoginIframe object

### **login(userId: string, callback: (userId: string, willRedirect: bool) => any): Promise<string>**

This method tries to log the given user in with the password entered into the iframe. If the login is a part of the IDP login flow, the SDK will redirect the user back to the IDP login process, then, after a few steps back to the page that requested the login. The callback parameter is a function that will always be called after the user successfully logs in. It will get the userId of the user and a bool value indicating if the SDK will redirect as parameters. If it returns a promise, the redirection will be delayed until that Promise resolves, otherwise it redirects immediately after the function returns. The login method returns a promise that will resolve to the userId of the logged in user if and only if there is no need for a redirection.

#### **Parameters:**

- **userId:** The userId or alias of the user to log in.
- **callback:** This callback will be called after the user successfully logs in. It's function is described in the summary.

#### **Returns:**

**Promise.<string>:** Resolves to the userId of the logged in user. Important: it will never resolve if this is done as a part of the IDP login flow, the SDK will redirect instead.

#### **Rejections:**

Code	Reason
InvalidAuthorization	Invalid username or password
UserNotExists	The user does not exist
UserNotValidated	The user wasn't validated before logging in

## Password change

Changing the password needs a completely freshly entered password along with the new password and a confirmation.

### ChangePasswordIframe

The password change iframe contains a password input field to for the current password and two for the new password and the confirmation. It can either be added dynamically by the SDK, or inserted manually into the HTML document and then wrapped by the SDK. The SDK wrapper will handle communication with the code running in the iframe and provide the methods below. Through this object the application can start the password change process.

```
var zkitChangePassword =  
zkitSdk.getChangePasswordIframe(document.getElementById('parentElement'));
```

or

```
<iframe
src="https://host-`${hostId}`.api.tresorit.io/tenant-`${tenantId}`/static/v4/embedded-changePassword.html" id="zkitChangePasswordIframe"></iframe>
<script type="text/javascript">
  var zkitChangePassword =
zkitSdk.wrapChangePasswordIframe(document.getElementById('zkitChangePasswordIframe'));
</script>
```

### Methods of the ChangePasswordIframe object

#### **changePassword(userId): Promise<string>**

This method logs into a security session and changes the password of the user.

##### **Parameters:**

- **userId:** Optional parameter to specify the id of the user changing password. This is only required if the user is not logged in.

##### **Returns:**

**Promise<string>:** Resolves to the id of the user.

##### **Rejections:**

Code	Reason
InvalidAuthorization	Invalid username or password
UserNameDoesntExist	The user does not exist

#### **checkPasswordsMatch(): Promise<bool>**

This methods check if the passwords match.

##### **Parameters:**

##### **Returns:**

**Promise<bool>:** True if the passwords entered in the iframes input fields match.

#### **getPasswordStrength(): Promise<PasswordMetric>**

This methods gives meta-information about the password the user entered. Currently returns the

length of the password.

**Parameters:**

**Returns:**

**Promise<PasswordMetric>:** Part of the result of running zxcvbn on the password. The result type is defined in the registration section.

## 4.2 Tresor management

Tresors are the basic unit of key handling and sharing. They can be referenced by a server generated id, returned on tresor creation. We provide no means to list a user's tresors, so the application should save these ids. Both tresor creation and sharing needs administrative approval to be effective. Since the encrypted data has the tresor id included, it can be decrypted even if the tresorId is lost from the application database.

### **createTresor()**

The createTresor call will create a tresor with the logged in user as a member, but it will only be usable once it's approved. The resolved value of the returned promise should be saved, as it is the only way to identify the tresor.

#### **Parameters:**

#### **Returns:**

**Promise.<string>:** Resolves to the tresorId of the newly created tresor. This id can be used to approve the tresor creation and to encrypt/decrypt using the tresor.

#### **Rejections:**

Code	Reason
NotLoggedIn	There is no user logged in

### **shareTresor(tresorId: string, userId: string)**

The shareTresor method will share the tresor with the given user. The operation will only be effective after it is approved using the returned OperationId. This uploads a modified tresor, but the new version is downloadable only after it has been approved. This should be done as soon as possible, as approving any operation to a tresor may invalidate any pending ones.

#### **Parameters:**

- **tresorId:** The id of the tresor to invite the user to.
- **userId:** The id of the user to invite.

#### **Returns:**

**Promise.<string>:** Resolves to the OperationId that can be used to approve this share.

#### **Rejections:**

Code	Reason
BadInput	Invalid tresor or userId
TresorNotExists	Couldn't find a tresor by the give tresorId
UserNotFound	There is no user by that id

CantInviteYourself	You can't invite yourself to a tresor
AlreadyAMember	The invitee is already a member of the tresor
CallerUserIsNotAMemberOfTresor	The caller user is not a member of the tresor
NotLoggedIn	There is no user logged in

### **kickFromTresor(tresorId: string, userId: string): Promise<string>**

This method will remove a user from a tresor you are a member of. The operation will only be effective after it is approved using the returned OperationId. This uploads a modified tresor, but the new version is downloadable only after it has been approved. This should be done as soon as possible, as approving any operation to a tresor may invalidate any pending ones.

#### **Parameters:**

- **tresorId:** The id of the tresor to invite the user to.
- **userId:** The id of the user to kick.

#### **Returns:**

**Promise.<string>:** Resolves to the OperationId that can be used to approve this share.

#### **Rejections:**

Code	Reason
BadInput	Invalid tresor or userId
TresorNotExists	Couldn't find a tresor by the give tresorId
UserNotFound	There is no user by that id
CantKickYourself	You can't kick yourself from a tresor
NotMember	The user to kick is not a member of the tresor
CallerUserIsNotAMemberOfTresor	The caller user is not a member of the tresor
NotLoggedIn	There is no user logged in

## 4.3 Encryption/Decryption

All data encrypted by ZeroKit is bound to a tresor, in that all current users of a tresor can decrypt any data encrypted by that tresor, even if it was encrypted before the user was added to it without reencryption. Removed users lose access to the keys necessary to decrypt data immediately after the kick operation was approved and they can't decrypt any data encrypted by that tresor in the future even if they saved/stored their keys. In the sdk the keys cached, but the keys used for encryption has to be refreshed from the server at most 5 seconds before use.

### Text/stringified data

#### **encrypt(tresorId: string, plainText: string): Promise<string>**

Encrypts the plaintext by the given tresor.

##### **Parameters:**

- **tresorId**: The id of the tresor, that will be used to encrypt the text
- **plainText**: The plainText to encrypt

##### **Returns:**

**Promise<string>**: Resolves to the cipher text. It contains the tresorId, so the it can be decrypted by itself.

##### **Rejections:**

Code	Reason
BadInput	The tresorId and plainText has to be a non-empty string
BadInput	Invalid tresorId
TresorNotExists	Couldn't find a tresor by the given id
CallerUserIsNotMemberOfTresor	This user does not have access to the tresor

#### **decrypt(cipherText: string): Promise<string>**

Decrypts the given cipherText

##### **Parameters:**

- **cipherText**: ZeroKit encrypted text

##### **Returns:**

**Promise<string>**: Resolves to the plain text.

##### **Rejections:**

Code	Reason
BadInput	Invalid cipherText

BadInput	Invalid tresorId
CallerUserIsNotMemberOfTresor	This user does not have access to the tresor

**File/Blob****encryptBlob(tresorId: string, plainText: Blob): Promise<Blob>**

Encrypts the plaintext Blob or File by the given tresor.

**Parameters:**

- **tresorId**: The id of the tresor, that will be used to encrypt the text
- **plainText**: The plainText Blob or File object to encrypt

**Returns:**

**Promise<Blob>**: Resolves to the encrypted blob. It contains the tresorId, so the it can be decrypted by itself.

**Rejections:**

Code	Reason
BadInput	The tresorId has to be a non-empty string and plainText has to be a Blob or File
BadInput	Invalid tresorId
TresorNotExists	Couldn't find a tresor by the given id
CallerUserIsNotMemberOfTresor	This user does not have access to the tresor

**decryptBlob(cipherText: Blob): Promise<Blob>**

Decrypts the given encrypted Blob.

**Parameters:**

- **cipherText**: ZeroKit encrypted Blob or File

**Returns:**

**Promise<Blob>**: Resolves to the decrypted Blob.

**Rejections:**

Code	Reason
BadInput	Invalid cipherText
CallerUserIsNotMemberOfTresor	This user does not have access to the tresor

## Uint8Array

**encryptBytes(tresorId: string, plainBytes: Uint8Array): Promise<Uint8Array>**

Encrypts the plaintext bytes by the given tresor.

**Parameters:**

- **tresorId**: The id of the tresor, that will be used to encrypt the text
- **plainBytes**: The data to encrypt in a Uint8Array format.

**Returns:**

**Promise<Uint8Array>**: Resolves to the encrypted blob. It contains the tresorId, so the it can be decrypted by itself.

**Rejections:**

Code	Reason
BadInput	The tresorId has to be a non-empty string. plainText has to be a Blob or File
BadInput	Invalid tresorId
TresorNotExists	Couldn't find a tresor by the given id
CallerUserIsNotMemberOfTresor	This user does not have access to the tresor

**decryptBytes(cipherBytes: Uint8Array): Promise<Uint8Array>**

Decrypts the given encrypted bytes.

**Parameters:**

- **cipherBytes**: ZeroKit encrypted data in a Uint8Array

**Returns:**

**Promise<Uint8Array>**: Resolves to the decrypted Blob.

**Rejections:**

Code	Reason
BadInput	Invalid cipherBytes
CallerUserIsNotMemberOfTresor	This user does not have access to the tresor



## 4.4 Invitation Links

Invitation links are used to invite someone into a tresor who is not a registered user. This method of invitation is made this way to communicate a best-practice. Using the link format below (placing the secret inside the fragment identifier of the url) you can ensure that the credentials necessary to get access to the tresor doesn't travel to your server and subsequently through the network. We advise, that you don't store any of the invitation links on your server unencrypted as it is a security critical information, which, in case of a breach, can be used to get access to user data. Even so, to achieve the best security you should use only password protected links and ask the users to transfer the password and the link to the invitee through different channels (e.g.: email the link and text/phone the password).

### Creating a link

You can create an invitation link with no password through the basic api, but password protected links have to be created through an iframe, since the user has to enter a password. You can load and wrap the iframe by calling `getCreateInvitationLinkPasswordIframe` or just wrap a manually loaded one with `wrapCreateInvitationLinkPassword` both of which return a wrapper object described below.

**`createInvitationLinkNoPassword(linkBase: string, tresorId: string, message: string): Promise<\{url: string, id: string\}>`**

Creates an invitation link that can be used by anyone to gain access to the tresor. The secret that can be used to open the invitation link is concatenated to the end of the link after a '#'. We recommend that you use password protected links. This operation needs administrative approval.

**Parameters:**

- **linkBase:** the base of the link. The link secret is appended to this after a '#'
- **tresorId:** the id of the tresor
- **message:** optional arbitrary string data that can be retrieved without a password or any other information

**Returns:**

**`Promise<\{url: string, id: string\}>`:** Resolves to the operation id and the url of the created link. The operation must be approved before the link is enabled.

```
var createLinkIframe =  
zkitSdk.getCreateLinkPasswordIframe(document.getElementById( 'placeholder' ));
```

or

```
<iframe
src="https://{tenantId}.api.tresorit.io/static/v4/embedded-register.html"
id="zkitRegIframe"></iframe>
<script type="text/javascript">
  var createLinkIframe =
zkitSdk.wrapCreateLinkPasswordIframe(document.getElementById('zkitRegIframe'))
);
</script>
```

#### Methods of the iframe

**createInvitationLink(linkBase:string, tresorId:string, message:string?): Promise<\{url: string, id: string}>**

This method creates an invitation link with the password entered into the iframe.

**Parameters:**

- **linkBase:** the base of the link. The link secret is appended to this after a '#'
- **tresorId:** the id of the tresor
- **message:** optional arbitrary string data that can be retrieved without a password or any other information

**Returns:**

**Promise<\{url: string, id: string}>:** Resolves to the operation id and the url of the created link. The operation must be approved before the link is enabled.

**checkPasswordsMatch(): Promise<bool>**

This methods check if the passwords match.

**Parameters:**

**Returns:**

**Promise<bool>:** True if the passwords entered in the iframes input fields match.

**getPasswordStrength(): Promise<PasswordMetric>**

This methods gives meta-information about the password the user entered. Currently returns the length of the password.

**Parameters:**

**Returns:**

**Promise<PasswordMetric>:** Part of the result of running zxcvbn on the password. The result type is defined at the end of the registration section.

## Retrieving info about a link

You can get some information about the link by calling `getInvitationLinkInfo` with the link secret. The secret is in the fragment identifier of the link. The returned object contains a token necessary to accept the invitation. This also is a client side secret, that should never be uploaded to your site as that would compromise the zero knowledge nature of the system by providing ways to open the tresor if the link was not password protected.

### **class InvitationLinkPublicInfo**

This represents the information that can be retrieved by the sdk without the password of the link, public in a sense that anyone in possession of the link can view it.

#### **Fields:**

- **creatorUserId:** the user id of the creator of this link
- **isPasswordProtected:** bool value indicating if the link is password protected
- **message:** arbitrary string data set at the time of creation of the link
- **\$token:** link information for internal use, used as a parameter for `acceptInvitationLink`

### **getInvitationLinkInfo(secret: string): Promise<LinkPublicInfo>**

Retrieves information about the link.

#### **Parameters:**

- **secret:** The secret is the one that was concatenated to the end of the url in `createInvitationLink`.

#### **Returns:**

**Promise<LinkPublicInfo>:** Resolves to all the information available.

## Accepting a link

A link with no password can be accepted by any logged in user that has access to the token returned by `getInvitationLinkInfo` through the basic sdk. Passworded links work the same way, but the user has to enter the password into an iframe. Loading and wrapping this iframe is done in much the same way as the others.

### **acceptInvitationLinkNoPassword(token: LinkToken): Promise<string>**

This method will add the user to the tresor of the link.

#### **Parameters:**

- **token:** The token is the `$token` field of the `InvitationLinkPublicInfo` of the link returned by `getInvitationLinkInfo`.

#### **Returns:**

**Promise<string>:** Resolves to the operation id that must be approved for the operation to be effective.

```
var acceptLinkIframe =  
zkitSdk.getAcceptLinkPasswordIframe(document.getElementById( 'placeholder' ));
```

or

```
<iframe  
src="https://{tenantId}.api.tresorit.io/static/v4/embedded-register.html"  
id="zkitRegIframe"></iframe>  
<script type="text/javascript">  
  var acceptLinkIframe =  
  zkitSdk.wrapAcceptLinkPasswordIframe(document.getElementById( 'zkitRegIframe' )  
  );  
</script>
```

#### Methods of the iframe

##### **acceptInvitationLink(token: Object): Promise<string>**

This method will add the user to the tresor of the link using the password entered into the iframe. This operation need administrative approval.

##### **Parameters:**

- **token:** The token is the \$token field of the InvitationLinkPublicInfo of the link returned by getInvitationLinkInfo.

##### **Returns:**

**Promise<string>:** Resolves to the operation id that must be approved for the operation to be effective.

#### Revoking a link

Invitation links can be revoked, but to do this the user has to be both a member of the tresor and have the link secret.

##### **revokeInvitationLink(tresorId: string, secret: string): Promise<string>**

Revokes the link from the tresor with the secret provided

##### **Parameters:**

- **tresorId:** the id of the tresor
- **secret:** The secret is the one that was concatenated to the end of the url in createInvitationLink.

##### **Returns:**

**Promise<string>:** Resolves to the operation id that must be approved for the operation to be effective.

## 4.5 Customization

The Iframes can be customized by providing your own css file, but if that's not enough, all the Iframes have a common set of customization methods:

### Css classes

**addClass(id: string, className: string)**

Adds the given class to the element selected by the provided id.

**Parameters:**

- **id:** The id of the element
- **className:** The name of the class to be added to the element

**getClasses(id: string)**

Returns the css classes of the element selected by the provided id.

**Parameters:**

- **id:** The id of the element

**removeClass(id: string, className: string)**

Removes the given class to the element selected by the provided id.

**Parameters:**

- **id:** The id of the element
- **className:** The name of the class to be removed from the element

### Events

**onEnter(callback: (targetId: string) => void)**

Adds the provided callback function the list of callbacks called when the user presses enter in one of the input fields.

**Parameters:**

- **callback:** The event handler, expecting a type and a targetId parameter

**onFocus(callback: (targetId: string) => void)**

Adds the provided callback function the list of callbacks called when one of the inputs gains focus.

**Parameters:**

- **callback:** The event handler, expecting a type and a targetId parameter

**onBlur(callback: (targetId: string) => void)**

Adds the provided callback function the list of callbacks called when one of the inputs loses focus.

**Parameters:**

- **callback:** The event handler, expecting a type and a targetId parameter

## Placeholders

**setPlaceholder(id: string, placeholder: string)**

Sets the placeholder of the input selected by the provided id.

**Parameters:**

- **id:** The id of the element
- **placeholder:** The placeholder to be set to the element

## 5. Administrative API

Tresorit encryption platform is a service to support the rapid development of client side encryption enabled apps. Most of the cryptographic operations (including invites and sharing) must be done client side by the SDK library. To provide control over these operations, and to prevent possible abuse by tampering the client, we introduced the admin API. All client initiated changes which has a permanent effect on the server has to be approved through the Admin API (typically by the server backend of the integrated app).

There are some other cases when the admin API can be used to adjust the settings of a tresor or a user, but these cases are admin-only calls (not the client initiates it.)

### Architecture

Any API call on the SDK that should have a permanent effect is not executed, only checked and saved by the platform server. The result of a call like this is an operation ID, which can be sent to the application backend of the integrated web app. The backend service then can access the Admin API with the Admin keys and use the operation ID to retrieve info about the operation and then approve or reject it. This flow splits these calls into two distinct phases, a request and an approval phase. If the request succeeds, then the request itself is syntactically and semantically correct, and the execution of the request is (at the time of the call) is possible, but the change is not effective yet. If the approval succeeds, then the change becomes effective instantaneously. Please note, that more conflicting requests may be made to the platform concurrently, and this may result in calls that can't be approved, because the approval one of the concurrent calls may make it impossible to execute the other request. This is a natural behavior of the deferred APIs, and in this case the client should retry the complete request cycle.

### Administrative operations

There are two groups of administrative operations. The first group is formed by the operations used for the approval / supervision of client-initiated operations. The second group contains administrative functions to track and adjust the service behavior for a single or more users or safes.

#### **Operations need to be approved:**

- User registration
- User invitation to a tresor
- Tresor creation
- Invitation link creation for a tresor
- Invitation link acceptance
- Invitation link revocation
- Kicking out a user from a tresor

#### **Administrative queries / settings:**

- Listing the user base of the tenant
- Listing the members of a tresor
- Delete a tresor
- Settings user state (enabled / disabled)
- Allow / prohibit tresor sharing for a user
- Allow / prohibit tresor creation for a user
- Upload custom content files (i.e. .css files)



- List custom content files
- Delete custom content files

## Authentication

The Admin API provides a JSON webservice interface described in the next chapter. As this is a stateless web API all calls made against it has to be authenticated individually. This is done by the "Admin key signing" signature algorithm, also described in the next chapter.

### Using the two admin keys

The system provides two admin keys since API v3\*. The api signing can be done with any of the two keys, the system will recognize it automatically. The reason for having two distinct keys is providing rolling key change.

It's recommended to use only one of the keys (either primary or secondary). When the used key somehow leaks, it has to be changed for security reasons. The first step to change the key without downtime is to start using the other key, and eliminate usage of the compromised one. After all integrated systems has been configured to use the new key, the old one can be re-generated. (Please feel free to contact tresorit for advice about key change or for a key-regeneration.)

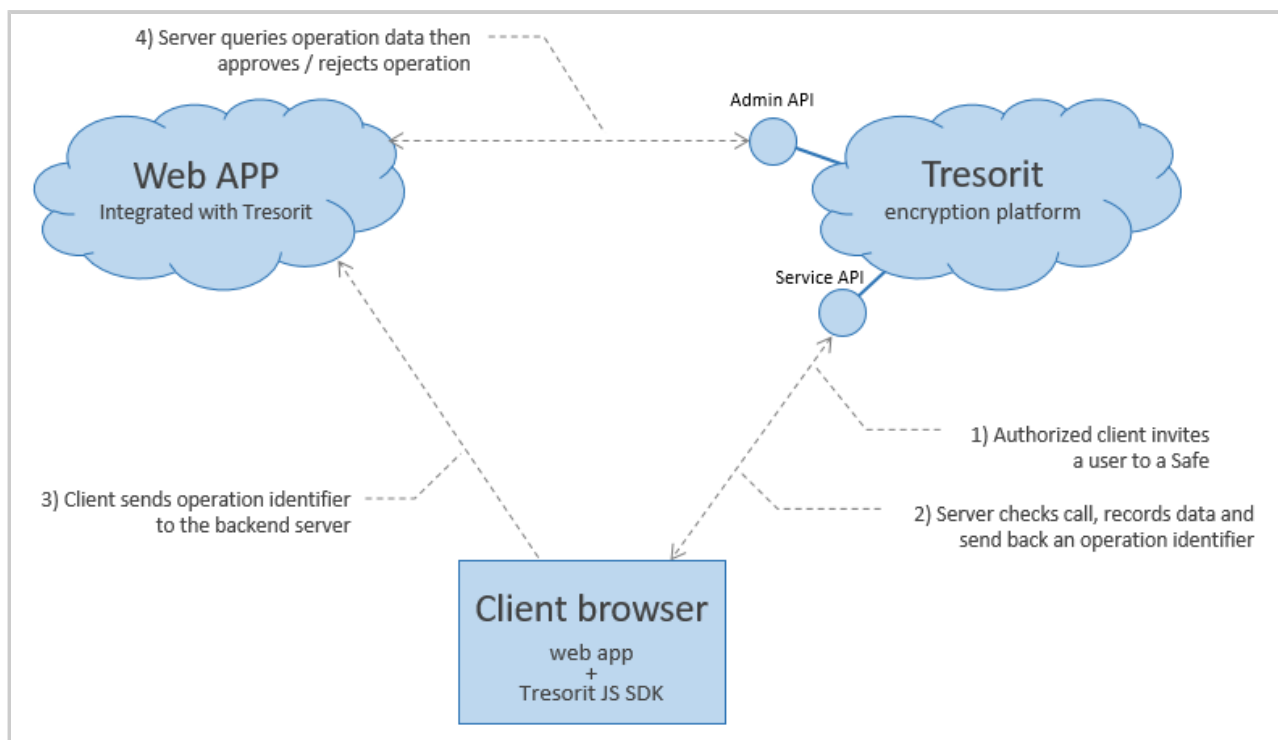
*\*For tenants created before this version their old single admin key became their primary key and a new secondary key is generated for them. Please feel free to contact tresorit for the second key.*

## Example

To understand the administrative workflow briefly, consider the following situation:

**Prerequisites:** *A user is signed in to the service successfully, then started to use the integrated application.*

1. The user makes changes in the App which result in the invitation of other users to a tresor (through the SDK).
2. The Tresorit encryption platform server checks this call, saves the pending data and sends back a unique operation id to the client.
3. The client then send this ID to the application server and requests the approval of it.
4. The application server then uses the administrative API to query the details of the operation, then decides about the approval or rejection of it.
5. If the operation is approved by the server (also through the admin API), and the approval succeeds, the operation takes effect instantly. Then the client can be notified about the result.



### Common approval workflow

**Note:** If the approval fails because the preceding approval of a concurrent call, then the client should retry the request-approval cycle again.

## 5.1 Authentication (request signing)

As all server-side APIs of the platform are stateless HTTP services, all requests made against it have to be authenticated individually. This is done by signing all request with the Tenant-specific admin key with the special scheme described below.

### Signing procedure

The signing procedure has 4 major steps. In the first step the request should be assembled completely but without the signature. In the second step the request is analyzed and canonicalized to a string which will be the subject of the signing. The third step is the signature calculation itself, and in the final step we amend the original request with the computed signature header(s).

#### Step 1. - Assemble the request

Assemble the request according to the API description and also provide the following headers:

##### Required headers:

###### Content-Type

Please fill this header according of the API description. **Do not** provide it with GET calls. (In the case of POST calls the value should be "application/json", otherwise it should be omitted.)

Example	Content-Type: application/json
---------	--------------------------------

###### Content-SHA256

If the request has a body (typically POST requests), always provide the Content-SHA256 header with the proper value. (The value should be the SHA256 fingerprint of the contents converted into a lower-case hex string.)

Example*	Content-SHA256: e3b0c44298fc1c149afb4c8996fb92427ae41e4649b934ca495991b7852b855
----------	--

*\*Note: in the example above the hash is the hash of the empty string.*

###### TresoritDate

Always provide this header and populate it with the current UTC timestamp in ISO-8601 format. The request is accepted only when the value of this header is within a 15 minute range from the server clock.

Example*	TresoritDate: 2014-05-05T05:05:05Z
----------	------------------------------------

*\*Note: Please always use the "Z" time-zone specifier in the date for indicating the UTC zone.*

###### Userid

Always provide this header for all administrative calls and fill the value with the admin user ID of your tenant. The admin user id is always computed with the following format string:

"admin@{TenantId}.tresorit.io" where {TenantId} is the placeholder for your tenant's ID.

Example*	UserId: admin@exampletenant.tresorit.io
----------	---

**\*Note:** in the example above the Tenant ID is "exampletenant"

### HMACHeaders header

To indicate the list of signed header to the platform server, please construct and add an additional header, called "HMACHeaders" to the request. The HMACHeaders header value is the comma separated list of the names of the signed headers. (This header itself can also be signed, but its not necessary. Note, that there should be no whitespaces in the list.)

The rules to assemble the list of the signed headers can be read below, in Step 2.

Example	HMACHeaders: Content-Type,Content-SHA256,TresoritDate,UserId
---------	--

### Request examples

Here you can see two examples for API calls. One for a GET and one for a POST call.

#### Get call example (listing userbase)

Example	GET <a href="https://exampletenant.api.tresorit.io/api/v1/users/admin/listusers">https://exampletenant.api.tresorit.io/api/v1/users/admin/listusers</a> HTTP/1.1 Host: <a href="https://exampletenant.api.tresorit.io">exampletenant.api.tresorit.io</a> TresoritDate: 2014-05-05T05:05:05Z UserId: <a href="mailto:admin@exampletenant.tresorit.io">admin@exampletenant.tresorit.io</a> HMACHeaders: TresoritDate,UserId
---------	---

#### Post call example (setting user state)

Example	POST <a href="https://exampletenant.api.tresorit.io/api/v1/users/admin/setuserstate">https://exampletenant.api.tresorit.io/api/v1/users/admin/setuserstate</a> HTTP/1.1 Host: <a href="https://exampletenant.api.tresorit.io">exampletenant.api.tresorit.io</a> Content-Type: application/json TresoritDate: 2014-05-05T05:05:05Z UserId: <a href="mailto:admin@exampletenant.tresorit.io">admin@exampletenant.tresorit.io</a> Content-SHA256: b11b56c53beb010850dbc00bf8f0ea12cdc9343075d7756efff556ea5163f43f HMACHeaders: Content-Type,Content-SHA256,TresoritDate,UserId  <body>
---------	---

### Step 2. - Canonicalize request

In this step a canonical request string should be assembled, which will be the subject of the signing. For this, the list of headers to sign should be determined.

**When constructing the signature string, keep the following in mind:**

- The VERB portion of the string is the HTTP verb, such as GET or PUT, and must be uppercase.
- All new-line characters (\n) shown are required within the signature string.
- The following headers **must be signed** if present. You can sign any other header too.
  - Content-Type, Content-SHA256, TresoritDate, UserId
- Please note, that the order of the header names in the HMACHeaders header must be the same as they appear in the canonical string.
- Also note, that this procedure is case-sensitive because of the cryptographic signature, so leave any text casing unchanged, as the texts appear in the HTTP request.

### Construction of the canonical string:

1. In the first line write the HTTP request method with uppercase letters, and terminate the line with a single "\n" character (line feed).
2. In the second line insert the request combined request path and query string of the request URI. (this is the part that follows the domain after a forward slash, but not containing the slash). For example, the request path and query of the following address `https://sub.example.com/part1/part2?value=42` is `"part1/part2?value=42"`. terminate the line with a line feed (\n) character.
3. In the next lines populate the headers listed in the HMACHeaders header in the same order, exactly one-header in each line. The format is **{HeaderName}:{HeaderValue}** where {HeaderName} stands for the header name and {HeaderValue} is the value of the header. Note, that there is only a semicolon between the header name and value, and there are no white spaces. terminate each line with a "\n" char, EXCEPT the last one.

Example	Original HTTP request
	<pre>POST https://exampletenant.api.tresorit.io/api/v1/users/admin/setuser state HTTP/1.1 Host: exampletenant.api.tresorit.io Content-Type: application/json TresoritDate: 2014-05-05T05:05:05Z UserId: admin@exampletenant.tresorit.io Content-SHA256: b11b56c53beb010850dbc00bf8f0ea12cdc9343075d7756efff556ea5163f43f HMACHeaders: Content-Type,Content-SHA256,TresoritDate,UserId  &lt;body&gt;</pre>
	<h3 data-bbox="279 1619 774 1653">Assembling canonical request string</h3> <pre>"POST" + "\n" + "api/v1/users/admin/setuserstate" + "\n" + "Content-Type" + ":" + "application/json" + "Content-SHA256" + ":" + "b11b56c53beb010850dbc00bf8f0ea12cdc9343075d7756efff556ea5163f43f" + "\n" + "TresoritDate" + ":" + "2014-05-05T05:05:05Z" + "\n" + "UserId" + ":" + "admin@exampletenant.tresorit.io"</pre>

Canonical request string
POST
api/v1/users/admin/setuserstate
Content-Type:application/json
Content-SHA256:b11b56c53beb010850dbc00bf8f0ea12cdc9343075d7756efff556ea5163f43f
TresoritDate:2014-05-05T05:05:05Z
UserId:admin@exampletenant.tresorit.io

### Step 3. - Signing

In this step the canonical request will be signed. The signing algorithm is the following:

**BASE64ENCODE( HMACSHA256( key = HEXTOBIN( AdminKey ), data = UTF8TOBINRAY( CanonicalString ) ) )**

Where:

- *AdminKey* is any of the the primary or secondary admin keys provided at subscription time. (See part 5. [Administrative API](#) for more information on primary and secondary admin keys.)
- *CanonicalString* is the canonicalized request string assembled in the previous step
- *BASE64ENCODE()* is a function which transforms binary data to a Base64 encoded string.
- *HEXTOBIN()* is a function which converts a hex encoded string to binary data.
- *UTF8TOBINRAY()* is a function which encodes all the characters in the specified string into a sequence of bytes using the UTF8 text codec
- *HMACSHA256(key, data)* is a function to compute the keyed-hash message authentication code (HMAC) of the input **data**, keyed with the binary input key **key** and using the SHA256 algorithm as hash function.

**Note:** Please note, that the binary-ascii conversions are needed because the cryptographical functions are usually take their inputs (key and data) in binary form.

Example	AdminKey
	AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
	Canonical string
	POST /api/v1/users/admin/setuserstate Content-Type:application/json Content-SHA256:b11b56c53beb010850dbc00bf8f0ea12cdc9343075d7756efff556e TresoritDate:2014-05-05T05:05:05Z UserId:admin@exampletenant.tresorit.io  <body>
	Signature

```
4eZKn+xk03mtLphU2v+UtNtVAHOA5fje/Co4irJV2ds=
```

#### Step 4. - Adding authorization header

The last step is where the Authorization header is assembled and added to the prepared request.

The header value is assembled according to the following format: **AdminKey {SignatureBase64}**

Note, that the value always starts with the constant string "AdminKey" followed by a single space and then with the base64 encoded value of the previously computed signature.

Example	Authorization: AdminKey 4eZKn+xk03mtLphU2v+UtNtVAHOA5fje/Co4irJV2ds=
---------	---

#### The assembled final query:

Example	<pre>POST https://exampletenant.api.tresorit.io/api/v1/users/admin/setuser state Host: exampletenant.api.tresorit.io Content-Type: application/json TresoritDate: 2014-05-05T05:05:05Z UserId: admin@exampletenant.tresorit.io Content-SHA256: b11b56c53beb010850dbc00bf8f0ea12cdc9343075d7756efff556ea5163f43f HMACHeaders: Content-Type,Content-SHA256,TresoritDate,UserId Authorization: AdminKey 4eZKn+xk03mtLphU2v+UtNtVAHOA5fje/Co4irJV2ds=  &lt;body&gt;</pre>
---------	---

## 5.2 API reference

- Endpoints
  - ApproveInvitationLinkAcception
  - ApproveInvitationLinkCreation
  - ApproveInvitationLinkRevocation
  - ApproveKick
  - ApproveShare
  - ApproveTresorCreation
  - DeleteCustomContent
  - DeleteTresor
  - GetInvitationLinkAcceptionDetails
  - GetInvitationLinkCreationDetails
  - GetInvitationLinkRevocationDetails
  - GetKickDetails
  - GetShareDetails
  - InitUserRegistration
  - ListCustomContents
  - ListMembers
  - ListTresors
  - ListUsers
  - RejectInvitationLinkAcception
  - RejectInvitationLinkCreation
  - RejectInvitationLinkRevocation
  - RejectKick
  - RejectShare
  - RejectTresorCreation
  - SetUserStatus
  - SetUserTresorCreationMode
  - SetUserTresorSharingMode
  - UploadCustomContent
  - ValidateUser
- Entities
  - ApproveAcceptInvitationLinkRequest
  - ApproveCreateInvitationLinkRequest
  - ApproveKickRequest
  - ApproveRevokeInvitationLinkRequest
  - ApproveShareRequest
  - ApproveTresorCreationRequest
  - CustomContentDetailsResponse
  - DeleteTresorRequest
  - GetAcceptInvitationLinkDetailsResponse
  - GetCreateInvitationLinkDetailsResponse
  - GetKickDetailsResponse
  - GetRevokeInvitationLinkDetailsResponse



- [GetShareDetailsResponse](#)
  - [InitUserRegistrationResponse](#)
  - [ListCustomContentResponse](#)
  - [ListMembersResponse](#)
  - [ListTresorsResponse](#)
  - [ListUsersResponse](#)
  - [RejectAcceptInvitationLinkRequest](#)
  - [RejectCreateInvitationLinkRequest](#)
  - [RejectKickRequest](#)
  - [RejectRevokeInvitationLinkRequest](#)
  - [RejectShareRequest](#)
  - [RejectTresorCreationRequest](#)
  - [SetUserStatusRequest](#)
  - [SetUserTresorCreationModeRequest](#)
  - [SetUserTresorSharingModeRequest](#)
  - [UserListItem](#)
  - [ValidateUserRequest](#)
- [Error codes](#)

## Endpoints

---

### ApproveInvitationLinkAcception

**summary:** Approves a client initiated invitation link acception

**url:** <https://<TenantID>.api.tresorit.io/api/v4/admin/tresor/approve-invitation-link-acception>

**method:** POST

**body:** [ApproveAcceptInvitationLinkRequest](#)

**remarks:**

To share a tresor with people who are not registered yet, invitation links can be created. If the link is distributed to them, they can use it to register and accept the share in one step. Both the creation and the acception (as it is a way of sharing) of invitation links must be approved or rejected by the backend service of the integrating web application.

This endpoint completes the link acception by approving it.

Please note, that the approval may fail due to concurrently accepted other links, or kick and share operations on the same tresor. (This behavior occurs because of the client-side encrypted nature of the platform.) In case of a failure, the complete link acception process must be retried from the beginning. (The client should re-initiate it again.)

### Input

- **request :** [ApproveAcceptInvitationLinkRequest](#)
  - Approval request as a JSON object

## Output

-

---

### ApproveInvitationLinkCreation

**summary:** Approves a client initiated invitation link creation

**url:** <https://<TenantID>.api.tresorit.io/api/v4/admin/tresor/approve-invitation-link-creation>

**method:** POST

**body:** [ApproveCreateInvitationLinkRequest](#)

#### remarks:

To share a tresor with people who are not registered yet, invitation links can be created. If the link is distributed to them, they can use it to register and accept the share in one step. Both the creation and the acceptance (as it is a way of sharing) of invitation links must be approved or rejected by the backend service of the integrating web application.

This endpoint completes the link creation by accepting it.

Please note, that the approval may fail due to concurrently approved other links, or kick and share operations on the same tresor. (This behavior occurs because of the client-side encrypted nature of the platform.) In case of a failure, the complete link creation process must be retried from the beginning. (The client should re-initiate it again.)

## Input

- **request :** [ApproveCreateInvitationLinkRequest](#)
  - Approval request as a JSON object

## Output

-

---

### ApproveInvitationLinkRevocation

**summary:** Approves a client initiated invitation link revocation

**url:** <https://<TenantID>.api.tresorit.io/api/v4/admin/tresor/approve-invitation-link-revocation>

**method:** POST

**body:** [ApproveRevokeInvitationLinkRequest](#)

#### remarks:

To share a tresor with people who are not registered yet, invitation links can be created. If the link is distributed to them, they can use it to register and accept the share in one step. The revocation of invitation links is the way of invalidating them to prevent further users to join the tresor. The revocation of links must be approved or rejected by the backend service of the integrating web application.

This endpoint completes the link creation by accepting it.

Please note, that the approval may fail due to concurrently approved other links, or kick and share operations on the same tresor. (This behavior occurs because of the client-side encrypted nature of the platform.) In case of a failure, the complete link creation process must be retried from the beginning. (The client should re-initiate it again.)

### Input

- **request** : [ApproveRevokeInvitationLinkRequest](#)
  - Approval request as a JSON object

### Output

-

---

## ApproveKick

**summary:** Approves the kick-out of a user from a tresor

**url:** <https://<TenantID>.api.tresorit.io/api/v4/admin/tresor/approve-kick>

**method:** POST

**body:** [ApproveKickRequest](#)

### remarks:

When a tresor member is kicked out by an admin member of the same tresor, he / she loses all rights in the share.

This operation is initialized by the client of the admin user and must be approved or rejected by the backend service of the integrating web application.

This call finishes the kick process by approving it.

Please note, that the approval may fail due to concurrently approved other kick or share operations on the same tresor. (This behavior occurs because of the client-side encrypted nature of the platform.) In case of a failure, the complete kick process must be retried from the beginning. (The client should re-initiate it again.)

### Input

- **request** : [ApproveKickRequest](#)
  - Request object in JSON

### Output

-

---

## ApproveShare

**summary:** Approves a client-initiated tresor sharing operation

**url:** <https://<TenantID>.api.tresorit.io/api/v4/admin/tresor/approve-share>

**method:** POST

**body:** [ApproveShareRequest](#)

**remarks:**

After a user initiated a tresor sharing, the backend service of the integrating web application must approve or reject the operation. This call finishes the sharing process by approving it.

Please note, that the approval may fail due to concurrently approved other kick or share operations on the same tresor. (This behavior occurs because of the client-side encrypted nature of the platform.) In case of a failure, the complete sharing process must be retried from the beginning. (The client should re-initiate it again.)

**Input**

- **request :** [ApproveShareRequest](#)
  - Approval request as a JSON object

**Output**

-

---

## ApproveTresorCreation

**summary:** Approves the given tresor creation

**url:** <https://<TenantID>.api.tresorit.io/api/v4/admin/tresor/approve-tresor-creation>

**method:** POST

**body:** [ApproveTresorCreationRequest](#)

**remarks:**

Only tresors wich are created with API V2 (or newer API versions) can be approved or rejected.

**Input**

- **request :** [ApproveTresorCreationRequest](#)

**Output**

-

---

## DeleteCustomContent

**summary:** Deletes a custon content file

**url:** <https://<TenantID>.api.tresorit.io/api/v4/admin/tenant/delete-custom-content?filename=<fileName>>

**method:** DELETE

**remarks:**

This method operates in a "delete-if-exists" manner, which means that if the target file does not exist, this method throws no error.

**Input**

- **fileName** : string
  - Name of the file to delete

**Output**

-

---

**DeleteTresor**

**summary:** Deletes the given tresor

**url:** <https://<TenantID>.api.tresorit.io/api/v4/admin/tresor/delete-tresor>

**method:** POST

**body:** [DeleteTresorRequest](#)

**remarks:**

Only tresors which are approved and not already deleted can be deleted

**Input**

- **request** : [DeleteTresorRequest](#)

**Output**

-

---

**GetInvitationLinkAcceptanceDetails**

**summary:** Retrieves the details of a client-initialized invitation link acceptance operation

**url:**

<https://<TenantID>.api.tresorit.io/api/v4/admin/tresor/get-invitation-link-acceptance-details?operationid=<operationId>>

**method:** GET

**remarks:**

To share a tresor with people who are not registered yet, invitation links can be created. If the link is distributed to them, they can use it to register and accept the share in one step. Both the creation and the acceptance (as it is a way of sharing) of invitation links must be approved or rejected by the backend service of the integrating web application.

This endpoint lets the service to fetch the details of the link acceptance operation to investigate it before the decision is made about the approval or rejection.

### Input

- **operationId** : string
  - Identifier of the link creation operation

### Output

- [GetAcceptInvitationLinkDetailsResponse](#)
- 

### GetInvitationLinkCreationDetails

**summary:** Retrieves the details of a client-initialized invitation link creation operation

**url** :

`https://<TenantID>.api.tresorit.io/api/v4/admin/tresor/get-invitation-link-creation-details?operationid=<operationId>`

**method:** GET

### remarks:

To share a tresor with people who are not registered yet, invitation links can be created. If the link is distributed to them, they can use it to register and accept the share in one step. Both the creation and the acceptance (as it is a way of sharing) of invitation links must be approved or rejected by the backend service of the integrating web application.

This endpoint lets the service to fetch the details of the link creation operation to investigate it before the decision is made about the approval or rejection.

### Input

- **operationId** : string
  - Identifier of the link creation operation

### Output

- [GetCreateInvitationLinkDetailsResponse](#)
- 

### GetInvitationLinkRevocationDetails

**summary:** Retrieves the details of a client-initialized invitation link revocation operation

**url** :

`https://<TenantID>.api.tresorit.io/api/v4/admin/tresor/get-invitation-link-revocation-details?operationid=<operationId>`

**method:** GET

### remarks:

To share a tresor with people who are not registered yet, invitation links can be created. If the link is distributed to them, they can use it to register and accept the share in one step. The revocation of invitation links is the way of invalidating them to prevent further users to join the tresor. The revocation of links must be approved or rejected by the backend service of the integrating web application.

This endpoint lets the service to fetch the details of the link revocation operation to investigate it before the decision is made about the approval or rejection.

### Input

- **operationId** : string
  - Identifier of the link revocation operation

### Output

- [GetRevokeInvitationLinkDetailsResponse](#)
- 

### GetKickDetails

**summary:** Retrieves the details of a user initiated kick operation

**url:** <https://<TenantID>.api.tresorit.io/api/v4/admin/tresor/get-kick-details?operationid=<operationId>>

**method:** GET

### remarks:

When a tresor member is kicked out by an admin member of the same tresor, he / she loses all rights in the share.

This operation is initialized by the client of the admin user and must be approved or rejected by the backend service of the integrating web application.

This endpoint lets the service to fetch the details of the operation to investigate them before the decision is made about the approval or rejection.

### Input

- **operationId** : string
  - Identifier of the kick operation

### Output

- [GetKickDetailsResponse](#)
- 

### GetShareDetails

**summary:** Retrieves the details of a pending tresor-sharing operation initiated by a user

**url:** <https://<TenantID>.api.tresorit.io/api/v4/admin/tresor/get-share-details?operationid=<operationId>>

**method:** GET

**remarks:**

After a user initiated a tresor sharing, the backend service of the integrating web application must approve or decline the operation. This endpoint lets the service to fetch the details of the operation to investigate them before the decision is made about the approval or rejection.

**Input**

- **operationId** : string
  - ID of the tresor-sharing operation

**Output**

- [GetShareDetailsResponse](#)
- 

**InitUserRegistration**

**summary:** Initiates a user registration process

**url:** <https://<TenantID>.api.tresorit.io/api/v4/admin/user/init-user-registration>

**method:** POST

**body:** -

**remarks:**

The initialization must be done by the backend service of the web application which uses the platform. A new user id is generated and returned to the caller.

The platform creates and returns a registration session, a session identifier and a verifier secret which must be provided to the next administration call of the process along with the session identifier. These secrets (session id and verifier) ensure that the process cannot be tampered between any of its steps by a 3rd party attacker.

Note: The generated user id is bound to the registration session.

**Input**

-

**Output**

- [InitUserRegistrationResponse](#)
- 

**ListCustomContents**

**summary:** Lists the custom content files from the tenant

**url:** <https://<TenantID>.api.tresorit.io/api/v4/admin/tenant/list-custom-contents>

**method:** GET



**Input**

-

**Output**

- [ListCustomContentResponse](#)
- 

**ListMembers**

**summary:** Lists all members of the given tresor

**url:** `https://<TenantID>.api.tresorit.io/api/v4/admin/tresor/list-members?tresorid=<tresorId>`

**method:** GET

**remarks:**

On success the resulting JSON object will contain the list of the tresor members' user IDs.

**Input**

- **tresorId** : string
  - Id of the tresor to list its users

**Output**

- [ListMembersResponse](#)
- 

**ListTresors**

**summary:** Lists all tresors with paging

**u r l :**

`https://<TenantID>.api.tresorit.io/api/v4/admin/tresor/list-tresors?pagesize=<pagesize>&continuefrom=<continueFrom>`

**method:** GET

**remarks:**

Page size cant be larger than 100 tresors Warning: The usage of this endpoint is not recommended and this will be removed in the future releases of the server. The tresor list should be maintained by the integrated application. This can be done by monitoring tresor approvals/rejections (available from API V2)

**Input**

- **pagesize** : int
- **continueFrom** : string

**Output**

- [ListTresorsResponse](#)
- 

### ListUsers

**summary:** Lists the user base of the tenant

**url:** <https://<TenantID>.api.tresorit.io/api/v4/admin/user/list-users>

**method:** GET

**remarks:**

This operation lists all registered users of the tenant, both the validated and unvalidated ones.

#### Input

-

#### Output

- [ListUsersResponse](#)
    - Separate lists of the identifiers of the validated and unvalidated tenant users wrapped into a JSON object
- 

### RejectInvitationLinkAcception

**summary:** Rejects a client initiated invitation link acception

**url:** <https://<TenantID>.api.tresorit.io/api/v4/admin/tresor/reject-invitation-link-acception>

**method:** POST

**body:** [RejectAcceptInvitationLinkRequest](#)

**remarks:**

To share a tresor with people who are not registered yet, invitation links can be created. If the link is distributed to them, they can use it to register and accept the share in one step. Both the creation and the acception (as it is a way of sharing) of invitation links must be approved or rejected by the backend service of the integrating web application.

This call finishes the revocation process by rejecting it.

The rejection cannot fail due to concurrent operations.

#### Input

- **request :** [RejectAcceptInvitationLinkRequest](#)
  - Approval request as a JSON object

#### Output

-

---

## RejectInvitationLinkCreation

**summary:** Rejects a client initiated invitation link creation

**url:** <https://<TenantID>.api.tresorit.io/api/v4/admin/tresor/reject-invitation-link-creation>

**method:** POST

**body:** [RejectCreateInvitationLinkRequest](#)

**remarks:**

To share a tresor with people who are not registered yet, invitation links can be created. If the link is distributed to them, they can use it to register and accept the share in one step. Both the creation and the acceptance (as it is a way of sharing) of invitation links must be approved or rejected by the backend service of the integrating web application.

This call finishes the revocation process by rejecting it.

The rejection cannot fail due to concurrent operations.

### Input

- **request :** [RejectCreateInvitationLinkRequest](#)
  - Approval request as a JSON object

### Output

-

---

## RejectInvitationLinkRevocation

**summary:** Rejects a client initiated invitation link revocation

**url:** <https://<TenantID>.api.tresorit.io/api/v4/admin/tresor/reject-invitation-link-revocation>

**method:** POST

**body:** [RejectRevokeInvitationLinkRequest](#)

**remarks:**

To share a tresor with people who are not registered yet, invitation links can be created. If the link is distributed to them, they can use it to register and accept the share in one step. The revocation of invitation links is the way of invalidating them to prevent further users to join the tresor. The revocation of links must be approved or rejected by the backend service of the integrating web application.

This call finishes the revocation process by rejecting it.

The rejection cannot fail due to concurrent operations.

### Input

- **request :** [RejectRevokeInvitationLinkRequest](#)

- Approval request as a JSON object

**Output**

-

---

**RejectKick**

**summary:** Rejects the kick-out of a user from a tresor

**url:** https://<TenantID>.api.tresorit.io/api/v4/admin/tresor/reject-kick

**method:** POST

**body:** [RejectKickRequest](#)

**remarks:**

When a tresor member is kicked out by an admin member of the same tresor, he / she loses all rights in the share.

This operation is initialized by the client of the admin user and must be approved or rejected by the backend service of the integrating web application.

This call finishes the kick process by rejecting it.

The rejection cannot fail due to concurrent operations.

**Input**

- **request :** [RejectKickRequest](#)
  - Request object in JSON

**Output**

-

---

**RejectShare**

**summary:** Rejects a client-initiated tresor sharing operation

**url:** https://<TenantID>.api.tresorit.io/api/v4/admin/tresor/reject-share

**method:** POST

**body:** [RejectShareRequest](#)

**remarks:**

After a user initiated a tresor sharing, the backend service of the integrating web application must approve or reject the operation. This call finishes the sharing process by rejecting it.

The rejection cannot fail due to concurrent operations.

**Input**

- **request** : [RejectShareRequest](#)
  - Approval request as a JSON object

### Output

-

---

### RejectTresorCreation

**summary:** Rejects the given tresor creation

**url:** <https://<TenantID>.api.tresorit.io/api/v4/admin/tresor/reject-tresor-creation>

**method:** POST

**body:** [RejectTresorCreationRequest](#)

### remarks:

Only tresors wich are created with API V2 (or newer API versions) can be approved or rejected.

### Input

- **request** : [RejectTresorCreationRequest](#)

### Output

-

---

### SetUserStatus

**summary:** Enables or disables a user account

**url:** <https://<TenantID>.api.tresorit.io/api/v4/admin/user/set-user-state>

**method:** POST

**body:** [SetUserStatusRequest](#)

### remarks:

The backend service of the web application that is using the encryption platform can enable or disable user accounts of the tenant. If an account is disabled, the owner user cannot log in to the service anymore.

This setting can be done at any time and does not affect other users that share content with the disabled account, it just disables the login of the user.

Warning: account state (enable/disable) is NOT the same as the initial validation of a newly registered user account. This is rather another policy to control the user's state individually.

### Input

- **request** : [SetUserStatusRequest](#)
  - State settings request as a JSON object

## Output

-

---

### SetUserTresorCreationMode

**summary:** Sets the tresor creation policy of a user account

**url:** <https://<TenantID>.api.tresorit.io/api/v4/admin/user/set-tresor-creation-policy>

**method:** POST

**body:** [SetUserTresorCreationModeRequest](#)

#### remarks:

The ability of a user to create tresors can be controlled by the backend service of the integrated web application. This is done by setting the tresor creation policy of the user through the admin api.

This API call sets the value of this policy. The default value is "true", which enables tresor creation, but this can be changed to "false" (which disables the creation of new tresors) or back at any time.

## Input

- **request :** [SetUserTresorCreationModeRequest](#)
  - Policy settings request as a JSON object

## Output

-

---

### SetUserTresorSharingMode

**summary:** Sets the tresor sharing policy of a user account

**url:** <https://<TenantID>.api.tresorit.io/api/v4/admin/user/set-tresor-sharing-policy>

**method:** POST

**body:** [SetUserTresorSharingModeRequest](#)

#### remarks:

The ability of a user to share tresors can be controlled by the backend service of the integrated web application. This is done by setting the tresor sharing policy of the user through the admin api.

This API call sets the value of this policy. The default value is "FreeShare", which enables sharing, but this can be changed to "NoShare" (which disables sharing) or back at any time.

## Input

- **request :** [SetUserTresorSharingModeRequest](#)
  - Policy settings request as a JSON object

## Output

-

---

## UploadCustomContent

**summary:** Uploads a custom content file

**url:**

https://<TenantID>.api.tresorit.io/api/v4/admin/tenant/upload-custom-content?filename=<fileName>&contenttype=<contentType>

**method:** PUT

**body:** [Stream](#)

**remarks:**

This method overwrites the target file if it already exists without any warning! The filename can include a path which will be added to the custom-file base-url. The filenames can contain english small and capital letters, numbers, dot ".", underscore "\_", hyphen "-" and slashes "/" as path separators.

Examples:

example.css (https://<tenant-id>.api.tresorit.io/custom/example.css) subfolder/example.css  
(https://<tenant-id>.api.tresorit.io/custom/subfolder/example.css)

## Input

- **fileName** : string
  - File name, including the desired file-path, but without a trailing slash.
- **contentType** : string
  - Content-type of the file
- **data** : Stream
  - File contents as body-stream

## Output

- [CustomContentDetailsResponse](#)
- 

## ValidateUser

**summary:** Validates a user registration

**url:** https://<TenantID>.api.tresorit.io/api/v4/admin/user/validate-user-registration

**method:** POST

**body:** [ValidateUserRequest](#)

**remarks:**

After a user completed the registration, the created account is not valid yet. In this state the account can be listed and other users may share a tresor with it, but the user cannot log in yet, because he / she is not validated. To complete the whole process, the account must be validated by the backend service of the web application.

If the application wants to validate the user's identity with any out-of-band method (like email or SMS), that process should take place before the account validation.

If this call succeeds, it completes the registration process by validating the user. The caller service must attach the registration session ID and the verification secrets (which are received in the earlier stage of the registration flow) to the request.

### Input

- **request** : [ValidateUserRequest](#)
  - Validation request message as a JSON object

### Output

-

---

## Entities

---

### ApproveAcceptInvitationLinkRequest

- **OperationId** : string
    - The ID of the AcceptInvitationLink operation.
- 

### ApproveCreateInvitationLinkRequest

- **OperationId** : string
    - The ID of the CreateInvitationLink operation.
  - **AdditionalInfo** : string
    - Additional info for the approved invitation link.
- 

### ApproveKickRequest

- **OperationId** : string
    - The ID of the Kick operation.
- 

### ApproveRevokeInvitationLinkRequest

- **OperationId** : string
    - The ID of the RevokeInvitationLink operation.
-



### ApproveShareRequest

- **OperationId** : string
    - The ID of the Share operation.
- 

### ApproveTresorCreationRequest

- **TresorId** : string
    - The ID of the tresor that was created.
- 

### CustomContentDetailsResponse

- **Name** : string
    - Gets or sets the file name
  - **Path** : string
    - Gets or sets the file path (under the custom contents folder / url)
  - **Url** : string
    - Gets or sets the full public URL of the file
  - **Size** : ulong
    - Gets or sets the size of the file in bytes
  - **ContentType** : string
    - Gets or sets the content type of the file
  - **Etag** : string
    - Gets or sets the etag of the file
- 

### DeleteTresorRequest

- **TresorId** : string
    - The ID of the tresor to delete.
- 

### GetAcceptInvitationLinkDetailsResponse

- **ByUserId** : string
  - The identifier of the user that created the invitation link.
- **ForUserId** : string
  - The identifier of the user that accepted the invitation.
- **TresorId** : string
  - The identifier of the tresor that the user is invited to.
- **Timestamp** : string
  - The timestamp of the operation.

ISO 8601 date format: "2000-01-23T23:59:59Z"

---

### GetCreateInvitationLinkDetailsResponse

- **ByUserId** : string
  - The identifier of the user that executed the CreateInvitationLink operation.
- **TresorId** : string
  - The identifier of the tresor that the user created the invitation link in.
- **Timestamp** : string
  - The timestamp of the operation.

ISO 8601 date format: "2000-01-23T23:59:59Z"

---

### GetKickDetailsResponse

- **ByUserId** : string
  - The identifier of the user that executed the Kick operation.
- **ForUserId** : string
  - The identifier of the user to be kicked.
- **TresorId** : string
  - The identifier of the tresor that the user is kicked from.
- **Timestamp** : string
  - The timestamp of the operation.

ISO 8601 date format: "2000-01-23T23:59:59Z"

---

### GetRevokeInvitationLinkDetailsResponse

- **ByUserId** : string
  - The identifier of the user that executed the RevokeInvitationLink operation.
- **TresorId** : string
  - The identifier of the tresor that the user revokes the invitation link in.
- **Timestamp** : string
  - The timestamp of the operation.

ISO 8601 date format: "2000-01-23T23:59:59Z"

---

### GetShareDetailsResponse

- **ByUserId** : string
  - The identifier of the user that executed the Invite operation.
- **ForUserId** : string
  - The identifier of the user to be invited.
- **TresorId** : string
  - The identifier of the tresor that the user is invited to.

### InitUserRegistrationResponse

- **UserId** : string
    - The newly generated user ID.
  - **RegSessionId** : string
    - Registration session ID
  - **RegSessionVerifier** : string
    - Registration session verifier
- 

### ListCustomContentResponse

- **CustomContentCount** : [uint](#)
    - Gets or sets the count of custom content files
  - **TotalUsedBytes** : [ulong](#)
    - Gets or sets the count of bytes used by custom content files
  - **Contents** : [List<CustomContentDetailsResponse>](#)
    - Gets or sets the list of custom contents
- 

### ListMembersResponse

- **Members** : [List<string>](#)
    - Gets the list of the user identifiers of the Tresor members
- 

### ListTresorsResponse

- **Tresors** : [List<string>](#)
    - List of tresor IDs
  - **NextTresor** : [string](#)
    - ID of the next tresor which can be used as a continuation marker when querying the next page
- 

### ListUsersResponse

- **Users** : [List<UserListItem>](#)
    - User list
- 

### RejectAcceptInvitationLinkRequest

- **OperationId** : [string](#)
    - The ID of the AcceptInvitationLink operation.
- 

### RejectCreateInvitationLinkRequest

- **OperationId** : [string](#)
    - The ID of the CreateInvitationLink operation.
-

**RejectKickRequest**

- **OperationId** : string
    - The ID of the Kick operation.
- 

**RejectRevokeInvitationLinkRequest**

- **OperationId** : string
    - The ID of the RevokeInvitationLink operation.
- 

**RejectShareRequest**

- **OperationId** : string
    - The ID of the Share operation.
- 

**RejectTresorCreationRequest**

- **TresorId** : string
    - The ID of the tresor that was created.
- 

**SetUserStatusRequest**

- **UserId** : string
    - The ID of the user.
  - **Enable** : bool
    - Indicates whether the user should be enabled or disabled.
- 

**SetUserTresorCreationModeRequest**

- **UserId** : string
    - The ID of the user.
  - **Enable** : bool
    - Indicates whether tresor creation is enabled or not.
- 

**SetUserTresorSharingModeRequest**

- **UserId** : string
  - The ID of the user.
- **Enable** : bool
  - Indicates whether tresor sharing is enabled or not.

True: TresorSharingMode.FreeShare

False: TresorSharingMode.NoShare

---

### UserListItem

- **UserId** : string
  - User identifier
- **UserRegistrationState** : string
  - User registration state

Currently available values: Unvalidated, Validated

---

### ValidateUserRequest

- **UserId** : string
    - The ID of the user.
  - **RegSessionId** : string
    - Registration session ID
  - **RegValidationVerifier** : string
    - Verifier secret for user validation
  - **RegSessionVerifier** : string
    - Registration session verifier
-

## Error codes

Error code	HTTP status code	Error message
AgreeCertDoesNotExist	Forbidden	Requested agree certificate doesn't exist.
AliasAlreadyExists	Forbidden	Alias already exists.
AliasNotExists	Forbidden	Alias not exists.
AuthorizationRequiredForNonAnonymousInviteLinks	Forbidden	Authorization is required for non-anonymous invite links.
BadInput	BadRequest	Input validation failed.
CallerAndTargetUserIsTheSame	BadRequest	The caller user and the target user of the call must not be the same!
CallerUserAlreadyTresorMember	BadRequest	The caller user has already access to the tresor.
CallerUserIsNotMemberOfTresor	Forbidden	Caller user has no permission for this tresor
Certificate	Forbidden	General certificate related error.
CertificateDataCollision	Conflict	Certificate data collide during insert batch.
CertificateIssuingFailed	Forbidden	Certificate signing error.
CertificateParsing	Forbidden	Error occurred while parsing certificate.
CertificateSigning	Forbidden	Certificate signing error.
CertToRevokeNotFound	BadRequest	The certificate to revoke does not exist.
ClientDisconnected	InternalServerError	The client is disconnected.
ConfigurationError	InternalServerError	The configuration of the tenant contains one or more errors.
Crypto	Forbidden	General cryptography related error.

DbUploadConflict	InternalServerError	Azure DBA block format conflicts with the block format of the Azure library! (Probably overlapped DBA and normal upload)
DeploymentEnvironmentDoesNotExist	NotFound	The requested deployment environment does not exist
DeviceCertAlreadyRegistered	Forbidden	Device certificate is already registered.
DeviceCertFragmentNotFound	Forbidden	Device cert/profile fragment not exists for this certificate.
DeviceCertificateMissing	Forbidden	Device certificate missing
DeviceCertificateRevoked	BadRequest	The device certificate is revoked.
DeviceCertificateSigningRequestsConsistency	Forbidden	Device Certificate Signing Requests not consistent.
DomainDoesNotExist	NotFound	The requested domain does not exists
EmailAddressBlocked	Forbidden	Email Address is blocked.
EntityExpiredButNotRemovedFromMasterFragment	Forbidden	Entity expired but not removed from MasterFragment
EntityRevokedButNotRemovedFromMasterFragment	Forbidden	Entity revoked but not removed from MasterFragment
EntityValidButRemovedFromMasterFragment	Forbidden	Entity valid but removed from MasterFragment
EtagNotMatch	PreconditionFailed	Etag mismatch.
GetDeviceCertificateCallConsistency	Forbidden	Get Device Certificate call not consistent.
GkfBlobConflict	InternalServerError	Gkf upload conflicted during uploading gkf file to the blob.
GkfIntegrity	Forbidden	Integrity check of Group Key File failed.
HostAlreadyExists	BadRequest	The requested host already exist
HostDoesNotExist	NotFound	The requested host does not exist

IllegalClaimValueInjection	BadRequest	User claim value injection is invalid.
IllegalTestValueInjection	Forbidden	Test value injection is strictly forbidden in release builds and for non-test users.
InvalidActivationId	Forbidden	The provided activation ID is invalid.
InvalidAuthorization	Forbidden	Authentication failed for user
InvalidAuthSession	Forbidden	AuthSession not found or invalid.
InvalidCertificate	Forbidden	Certificate contains invalid values.
InvalidHostState	BadRequest	The requested change is invalid in the current state of the host
InvalidHostStateTransition	BadRequest	The requested change in the state of the host or its installation is invalid.
InvalidInvitationLink	Forbidden	Invitation link is not found or invalid.
InvalidMasterFragmentChange	Forbidden	InvalidMasterFragmentChange
InvalidRegistrationSessionVerifier	Forbidden	The provided registration session verifier is invalid for the registration session
InvalidRegistrationValidationVerifier	Forbidden	The provided registration validation verifier is invalid for the registration session
InvalidSession	Forbidden	Session not found or invalid.
InvalidTenantStateTransition	BadRequest	The requested change in the state of the tenant or its installation is invalid.
InvalidTenantType	BadRequest	Invalid tenant type
InvalidToken	Forbidden	Login token is invalid.
InvitationLinkAlreadyExists	BadRequest	Invitation link already exists.
InvitationLinkBadPasswordTryLimitExceeded	Forbidden	Invitation link's password try limit is exceeded.
IpAddressBlocked	Forbidden	IP Address is blocked.



LatestAgreeCertNotKnownByClient	PreconditionFailed	Latest agree certificate is unknown by the client.
LatestGkfNotKnownByClient	PreconditionFailed	Latest group key file is unknown by the client.
MasterFragmentEntityKeyTypeViolation	Forbidden	MasterFragment entity key type violation
MasterFragmentEntityKeyViolation	Forbidden	MasterFragment entity key violation
MasterFragmentEntityUniqueIdViolation	Forbidden	MasterFragment entity unique Id violation
MasterFragmentKeyNotFound	BadRequest	Master fragment key not found.
MasterFragmentNotFound	NotFound	Master fragment not found.
MasterFragmentUserIdMismatch	BadRequest	MasterFragment user Id mismatch
MasterFragmentValidationDataMismatch	Forbidden	MasterFragment validation data mismatch
MasterFragmentValidationDataMissing	BadRequest	MasterFragment validation data missing
MasterFragmentVersionMismatch	BadRequest	The given version is not the latest version of the master fragment.
MembershipRecordAlreadyExistsForCallerUser	BadRequest	The caller user's permission record already exists.
MembershipRecordAlreadyExistsForTargetUser	BadRequest	The target user's permission record already exists.
MembershipRecordNotFoundForCallerUser	BadRequest	The caller user's permission record was not found.
MembershipRecordNotFoundForTargetUser	BadRequest	The target user's permission record was not found.
NotModified	NotModified	Not modified.
OnlyForTestUser	Forbidden	Not a test user.
PermissionChangeLogVersionNotExists	NotFound	The requested permission changelog version was not found.

PolicyEntityNotExists	NotFound	Policy not exists.
ProfileCertEntityNotFound	BadRequest	The profile cert entity does not exist.
ProfileCertificateMissing	Forbidden	Profile certificate missing
ProfileSchemaMismatch	BadRequest	Profile schema mismatch.
PublicKeysNotDifferent	BadRequest	The public keys are not different.
PublicKeysNotEqual	BadRequest	Public keys are not equal
RegistrationSessionIdMismatch	BadRequest	The provided registration session id is valid, but belongs to another user is then the given one.
RegistrationSessionNotExists	BadRequest	The provided registration alias / session id does not belong to an existing registration session.
RequiredClaimNotGranted	Forbidden	Required claim is not granted for the user.
StandardHttpErrorWrapping		
StorageConflict	Conflict	An unexpected, conflicting storage error happened.
StorageNetwork	InternalServerError	An unexpected, network related storage error happened.
SubscriptionDoesNotExist	NotFound	The requested subscription does not exists
SubscriptionTenantLimitReached	BadRequest	The tenant limit of the subscription has already been reached.
TargetUserAlreadyTresorMember	BadRequest	The target user has already access to the tresor.
TargetUserIsNotMemberOfTresor	Forbidden	Target user has no permission for this tresor
TenantAlreadyExist	BadRequest	The tenant is already exists or the tenant id is taken
TenantConfigurationFailed	InternalServerError	The configuration of the tenant contains failed.

TenantDisabled	ServiceUnavailable	Tenant is disabled, therefor the request is unsatisfiable.
TenantDoesNotExist	NotFound	The requested tenant does not exists
TenantInstallationAlreadyExists	BadRequest	Could not create teh requested tenant installation because its already exists
ThereIsNoAvailableTenant	ServiceUnavailable	Currently there is no available tenant to reserve. Try again later.
TheseArentTheDroidsYouAreLookingFor	NotFound	These aren't the droids you're looking for. (How could this happened to you?! Possibly you typed in an invalid url, so the server could not handle your request. Check your code, and try again. May the force be with you!)
TresorAlreadyDeleted	BadRequest	Tresor has been deleted.
TresorAlreadyExists	BadRequest	Tresor already exists.
TresorCreationForbidden	Forbidden	Tresor creation is forbidden by policy.
TresorIsAlreadyApproved	BadRequest	Tresor is exists, and already approved by the administrator.
TresorIsNotApproved	BadRequest	Tresor is exists, but not approved by the administrator yet.
TresorNotExists	NotFound	Tresor head not found.
TresorSharingForbidden	Forbidden	Tresor sharing is forbidden by policy.
UnauthorizedAccess	Forbidden	Unauthorized access
UnexpectedError	InternalServerError	An unexpected, internal server error has happened.
UnexpectedMasterFragmentIntegrityError	BadRequest	Unexpected error occurred during MasterFragment integrity check.
UnexpectedStorage	InternalServerError	An unexpected, internal server error has happened.
UnknownMasterFragmentEntityType	Forbidden	MasterFragment entity unique Id violation

UserAlreadyRegistered	Forbidden	User already registered.
UserCertFragmentAlreadySet	Forbidden	UserCertFragment has been already set.
UserDeviceEntityNotFound	BadRequest	The device entity does not exist.
UserDoesNotHaveAgreeCert	Forbidden	User does not have agree certificate yet.
UserInWrongState	Forbidden	User is in wrong state.
UserIsDisabledByPolicy	Forbidden	User is disabled by policy.
UserIsNotDomainMember	Forbidden	The user is not member of the domain
UserNotExists	Forbidden	User not exists.
UserUnderDeletion	Forbidden	The user's account is under deletion and the worker did not fully finish the process yet.
WrongDeviceCertOrigin	Forbidden	The device certificate was not issued by the current infrastructure.



## 6. Built-in IDP

All tenants have a built-in, optionally enabled OpenID Connect identity provider component, which can be used to delegate the logged in user's identity from the encryption platform to the integrated application(s) in a secure and standardized way.

### Concept

The idea behind the integrated IDP component is to provide a standard way for the integrated web app to consume the identity of the users. Because the user login takes place through the ZeroKit web or mobile SDK, identity propagation needed to ensure the backend service of the web application about the user's identity. The ZeroKit SDK itself can determine and provide the logged in user's identity, but that information can only be trusted at the client-side part of the integrated app. The backend of the app may need a more trustworthy source than what the client side can provide. This source is the optionally enabled IDP component.

### Protocol

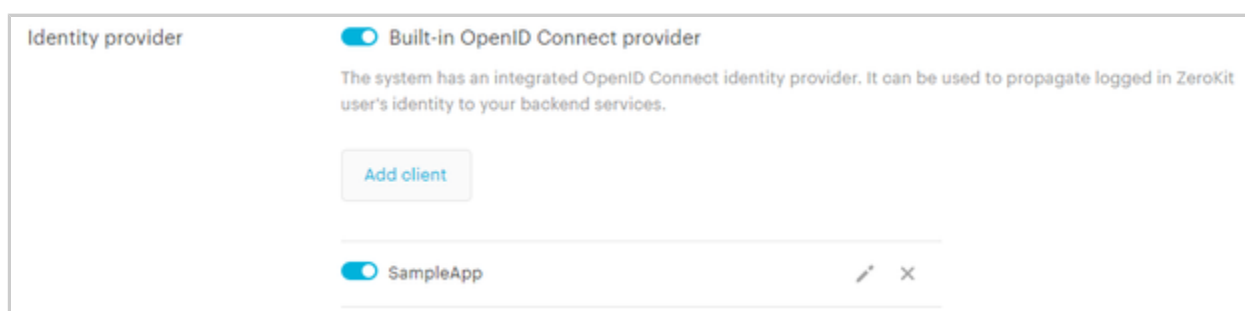
The OpenID/Connect protocol was chosen because this standard is one of the most modern and secure protocols, which offers easy integration with web applications. The integrated IDP component is an officially OpenID/Connect certified component, which offers full support of the protocol.



The OIDC API itself is accessible at the <https://host-{hostId}.api.tresorit.io/tenant-{tenantId}/idp> address where {tenantId} stands for your tenant id and {hostId} for your host id.

### Configuration

Before usage the IDP component must be activated and configured. Clients can be configured and enabled/disabled on the administrative portal, on the configuration page of your tenant in the "Identity provider section".



#### Identity provider configuration section

Here you can enable/disable the whole IDP module, and also add, configure or remove clients. On the basic configuration page you can edit the Name of your client, edit redirection URL list and copy the client ID and secret.

×

**Edit OpenID Connection properties of Client #1**

Client name ?

SampleApp

Client ID ?

sampleapp

Copy

Client secret ?

Copy

Redirect URLs ?

Add URL

Add

http://localhost:3000/auth/callback

×

Advanced options ▾

Apply

Cancel

### Basic IDP client configuration page

**Client name:** A readable name which helps to identify your client properly. This name used only by the portal and in logs, but not by any code. You can change it any time.

**Client ID:** this is an auto-generated id of your client. This ID is immutable, and used by the OpenID Connect client libraries to identify your client.

**Client secret:** this is an auto-generated secret value used by the OpenID Connect client libraries with some OpenID flows.

**Redirect URLs:** a list of web URLs where your user is allowed to be redirected by the consumer of the IDP (your applications) after a successful login.

If you enable the "Advanced options", you can also set the used OpenID Connect flow and the list of

allowed CORS origins.

×

**Edit OpenID Connection properties of Client #1**

Client name ?

SampleApp 📋

Client ID ?

sampleapp Copy

Client secret ?

Copy

Redirect URLs ?

Add URL 📋

Add

http://localhost:3000/auth/callback ×

Flow ?

Authorization Code ↕

☐ Requires proof key (DHCE) ?

CORS origins ?

Add

Advanced options ^

Apply

Cancel

Advanced IDP client configuration page



**Flow:** Selected OpenID flow for the client. The flows are well-defined sequences in the OpenID Connect standard. You can choose *Hybrid (default)*, *Authorization code* or *Implicit* flow.

**Requires proof key:** In the case of *Hybrid* or *Authorization code* flows you can turn on this option. Proof key usage is an extension of the OpenID connect standard which improves security in some cases, especially when the consumer is a mobile app. Your client has to support this option.

**Cors origins:** Some IDP APIs can be called from JavaScript code as well (especially when *Implicit* flow is used). As your app (consumer) and the platform's server (provider) is hosted on different domains, you have to allow these calls by configuring your origin domains.

**Note:** The allowed scopes are *"openid"* and *"profile"*, and the only returned claim will be the user identifier as the value of the *sub* (subject identifier) claim.

**Note:** Please note, that IDP configuration changes like any other changes on the admin portal made against a tenant, may take up to 5 minutes for a change to be effective.

**Note:** The integrated IDP will not show consent screens as this is not a public IDP service and your application(s) will be the only client(s) of it.

## Debugging

If the client configuration either on client or server side contains an error, it can be very hard to find out what causes the problem. To make this easier, we implemented an extensive logging for the IDP. This logging can be turned on by Tresorit if you ask for it on the red carpet channel, and then we can provide logs. This logging should only be activated for as long as it's needed, because the logging is resource wasting and can cause security issues if left on for long periods of time, because it may also log sensitive token data. It's a very useful tool for debugging, but please be considerate with its usage.

## Example

Here you can see the shortest possible client config of the IDP component of the ZeroKit platform and the client-side pair of the same config implemented in a C# client using standard OWIN components.

**Client config (at ZeroKit Admin portal):**

×

**Add OpenID Connection Client**

**Client name** ?  
 2

**Client ID** ?  
 Copy

**Client secret** ?  
 Copy

**Redirect URLs** ?  

2 Add

×

**Advanced options** ▼

Apply Cancel

**Example IDP client configuration****Client config (web app side):**

**Example**

```
public class Startup
{
    public void Configuration(IApplicationBuilder app)
    {
        app.UseCookieAuthentication(new CookieAuthenticationOptions {
            AuthenticationType = "cookies" });
        app.UseOpenIdConnectAuthentication(new
            OpenIdConnectAuthenticationOptions()
            {
                AuthenticationType = "oidc",
                SignInAsAuthenticationType = "cookies",
                Authority = "http://host-myhost.api.tresorit.io/tenant-mytenant/
                    idp", // Use the service URL of your tenant
                ClientId = "ua8zpwlxq5_4777cy43iL", // Use your client id from
                    the portal
                ClientSecret = "ytjwe8skrjg2zhvs", // Use the secret of your
                    IDP client from the portal
                RedirectUri = "http://localhost:4455/", // The desired redirect
                    location after login
                ResponseType = "id_token token",
                Scope = "openid profile"
            });
    }
}
```

**Note:** This is a full configuration of an OWIN web app, the absolute relevant config lines are lines 8-14.

## 7. Common flows

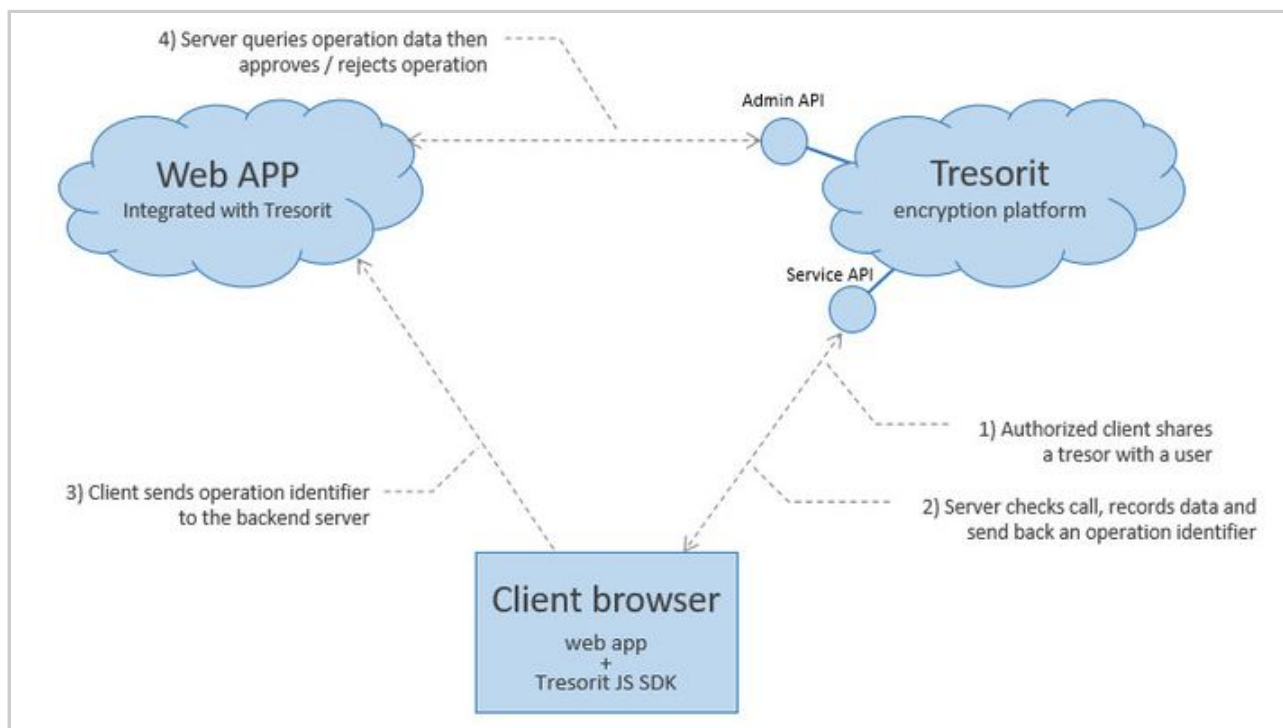
Brief description of common flows of the framework.

### Basic approval flow

All client initiated operations against the ZeroKit API (beside encryption, decryption and login) must be supervised (approved or rejected) by the backend service of your application through the administration API of ZeroKit. This gives you great control over the user activity on the encryption platform. The other reason for this is that the ZeroKit platform does not provide detailed ACLs (that is the responsibility of your application), the granularity of ACL that ZeroKit provides determines whether a given user can or cannot access the right credentials for encrypting and decrypting data.

A typical user-initiated flow against the platform consist of three steps.

1. **Initiation:** The user does something on your web UI (for example: shares encrypted data, which triggers an invite to one of their tresors) which needs an operation against the ZeroKit API. Your app logic first checks if the user can do the given operation, and then calls the ZeroKit SDK, which executes the needed cryptographic operations and the init tasks against the tenant server. The requested operation is now prepared on the platform server but not effective yet. The SDK returns a unique identifier which can be used to refer to this operation.
2. **Backend notification:** The client-side part of your application notifies your backend service about the initiated operation and also sends the operation identifier to the backend.
3. **Supervision:** Your backend service can query the encryption platform for the details of the operation (by its identifier) and then it can decide to approve or reject it. The operation is completed when the backend successfully completes the operation by an approval or rejection. Then the client can be notified that the requested operation is done. For most operations early approval is critical, as approving other operations against the same resource may invalidate every other pending operation. You can see this flow on the next figure.



### Communication flow of the client-initiated operation

**Note:** In some cases the approval can fail due to concurrently approved / rejected operations against the same resource (tresor, user, invitation link etc.). This behavior occurs because the client-side encrypted nature of the platform. (The client-side encrypted structures must be re-created by the client because the server is unable to modify or merge (or even to understand) what is encrypted by the client). In case of a failure, the whole sharing process must be retried from the beginning. (The client should re-initiate it again.)

## Registration

The registration flow is the first process that end-users will encounter, and also the most complex one under the hood because of the involved cryptographic operations. Fortunately the ZeroKit SDK makes it fairly easy to implement it.

As ZeroKit offers client-side encryption, its main goal is to prevent any 3rd party code (even the code of your web application) or attackers from accessing the user's password(s) and key(s). This is achieved by iframes hosted by the tenant server. These iframes handle different processes (e.g. registration, login etc.), and they can be integrated into your page and customized by your CSS. The user interface of the registration iframe contains only the password boxes, but internally it encapsulates the complex client-side encryption logic. As all browsers separate the hosting webpage and the embedded iframes completely, there is no way to reach sensitive information from outside of the iframe. The iframe is controlled by the ZeroKit JS SDK through a messaging protocol, so your webpage only has to communicate with the SDK.

### The registration form

The registration form has to be assembled from the registration iframe (which contains two password input fields), your desired custom fields (like email, username, address etc.) and your CSS to format the page. If the page is assembled correctly, the user will see no difference between your input fields and the embedded password fields.

The diagram shows a registration form titled "Registration form". It contains four input fields: "username", "Password", "Confirm password", and a "Register" button. Annotations with dashed lines point to these fields:

- "User name and any additional fields are added by your application logic" points to the "username" field.
- "Both password fields exist on the embedded register iframe. Custom css hides iframe borders and ensures that the form looks consistent." points to the "Password" and "Confirm password" fields.
- "Submit button and logic provided by your client-side application logic" points to the "Register" button.

### Example registration form

### Communication flow during registration

The standard one-step registration flow (submit reg form and done) has to be separated into three background steps to ensure maximum security and control over the system. If you understand the sequence, this will be quite easy, and users will see no difference from the standard flow.

**Steps:**

1. User fills out the form (user data and password fields) and clicks on "Submit".
2. The submit logic has to be implemented by you in JavaScript using Ajax calls. First, the fields of the form (except iframe/passwords) have to be sent to your application backend. Your backend then can inspect the fields and validate the data.
3. If everything is fine, the backend should make a call to the ZeroKit backend through the administration API to initiate a user registration. (Note: the call from the submit logic to the application backend is unanswered yet!) The platform server will send back a generated user ID, a registration session ID and a registration verifier secret on success. The application backend should save all user data along with these three fields.
4. Now the backend can answer the submit call by sending back the freshly generated user ID and registration session ID to the client.

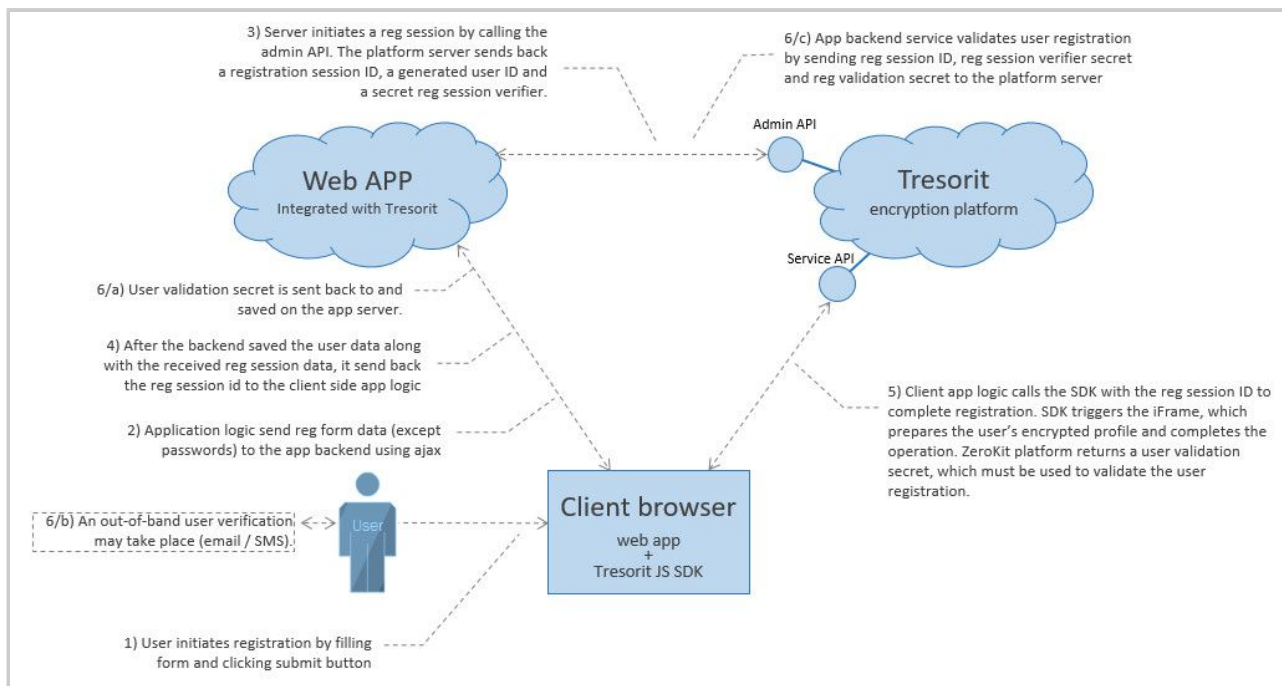
**Warning!** Never ever send the **registration verifier secret** to the client side. This value is security-critical to the process.

5. When the submit logic receives this answer, it can call the SDK to complete the registration. (Internally the SDK will send the user ID and the registration session data along with the prepared encrypted data to the ZeroKit backend). After this call completes, the user is registered in both your system and the encryption platform. The call also returns a secret value - called validation secret - which should be sent and stored along with the user data at your backend.

**Warning!** Albeit the user is registered, it is still not active, as the registration has to be validated. Tresors can be shared with unvalidated users, but they cannot log in to the system.

6. The validation will be done by an admin call from your server backend to the ZeroKit server. This call needs the registration session ID, the registration verifier secret and the validation verifier secret as input.
  - a) These secrets are returned by the previous calls to the ZeroKit backend and SDK. They ensure that the process cannot be tampered with by a 3rd party between the steps.
  - b) (Optional) If you want to verify the identity of the registering user by any out-of-band method (such as an email or SMS) you should do that validation at this point - before the registration is completed by the SDK and before your backend validates the user registration. (An out-of-band verification is strongly recommended, but if you do not want to implement it, you can validate the user immediately after the registration completion.)
  - c) The actual validation is done by an admin call from your server backend to the ZeroKit server with the registration session ID and the verifiers.

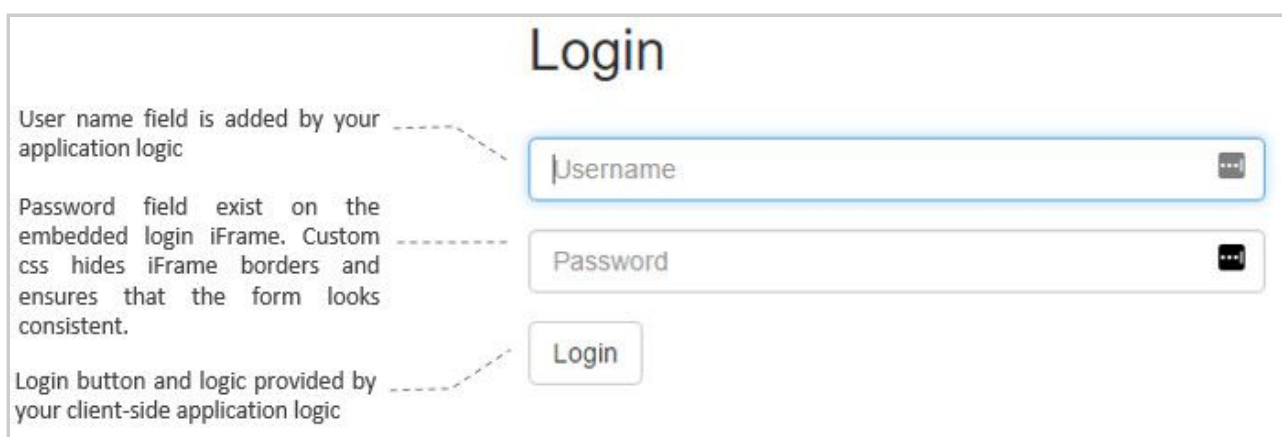
The described process consists of four main parts: 1) user fills out the form and sends data to the app backend, 2) app backend checks data and initiates a ZeroKit user registration session 3) client side logic finishes the registration 4) User registration is validated by the application backend (after an optional out-of-band validation.). Notice that the first three steps are the same as the previously described client-initiated flows, the only difference is the additional verification steps. The following diagram will help you to understand the flow:



### Steps and data flow of the registration process

## Login

The login flow is also a slightly different from the conventional one. Login is the process when the user proves his/her identity by providing a secret (in this case the password). As the login page has to be hosted by the integrated web app and the system should also maintain its zero knowledge nature, the secret must be entered to the login iframe provided by ZeroKit. The iframe internally does all the cryptographic work needed to prove the identity to the server without leaking the password.



### Login form explained

From the perspective of the initialization of the login process we have two distinct login flows. If the flow is initialized by the user (or by any other component) by navigating explicitly to the login page, the flow is called *explicit login*. After the successful explicit login flow the user will be able to use the ZeroKit JS SDK



and the ZeroKit API as a logged in user. The SDK can tell the user's identity to the client side JS logic of the integrating web app, but it cannot prove the user's identity directly to the backend of the web app. To delegate the user's identity to the web app backend, the integrated OpenId/Connect IDP component of the ZeroKit backend can be used by configuring the web app as a client of the IDP. The IDP can delegate the user's identity automatically when the user is logged in to the ZeroKit system. When the application requests the user's identity from the IDP and the user is not logged in yet, the IDP can also initiate the login process and after a successful login, it will complete the identity request automatically. If the login flow is initiated by the IDP, it is called *implicit login*.

In the following we will discuss the detailed steps of the two login flows.

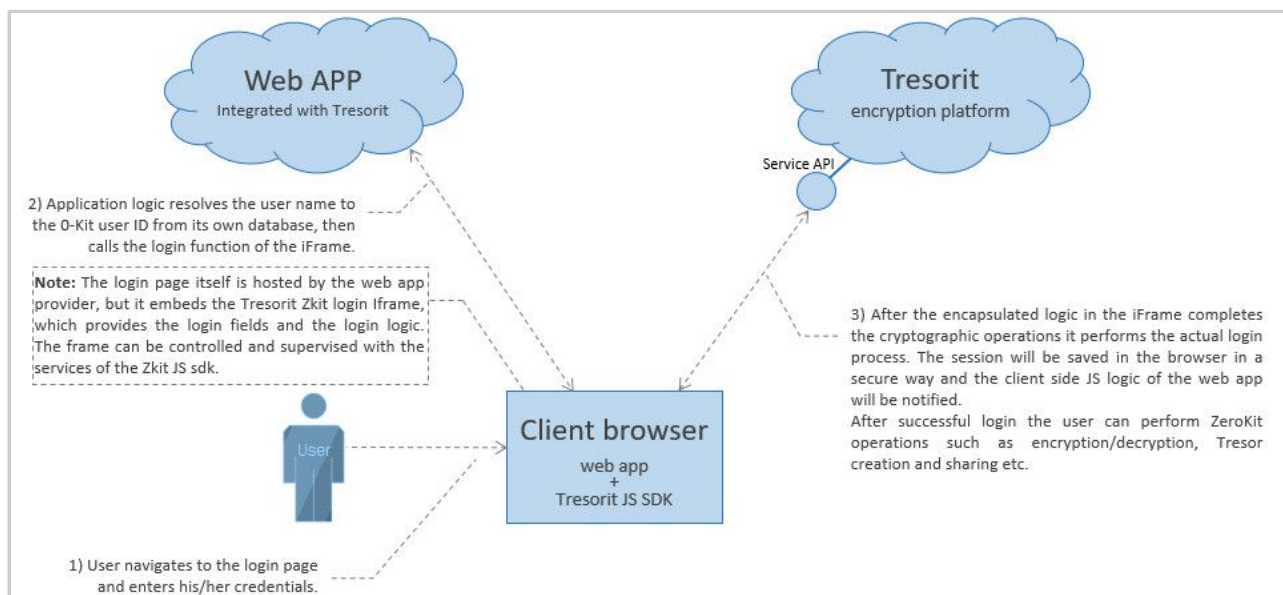
**Note:** The core process (the logging in to the ZeroKit server) is the same in both flows, only the initiation and maybe the final redirect differ.

## Explicit login

In this case the login flow is initiated by an explicit navigation to the login page. The initiator can be the user or any part of the web app which made a redirect.

The login iframe contains only the password field and the login logic. The presentation of the username field and the login button is the responsibility of the app. When the user presses the login button it should resolve the username entered by the user using the application's own API and then pass the ZeroKit user ID of the user to the SDK as an argument of the login method. The result of the method will be a JS promise which can be used to wait for the result of the login request. If the login succeeds, the SDK will automatically recognize the user's session and all further calls will be made on the behalf of the logged in user until logout. If the app needs, it can redirect the user to any other page when the login is completed. (No automatic redirection will be made by the SDK.)

If the user has a valid ZeroKit session in the browser, the IDP component can automatically delegate his/her identity to its configured clients without asking for his/her credentials again.



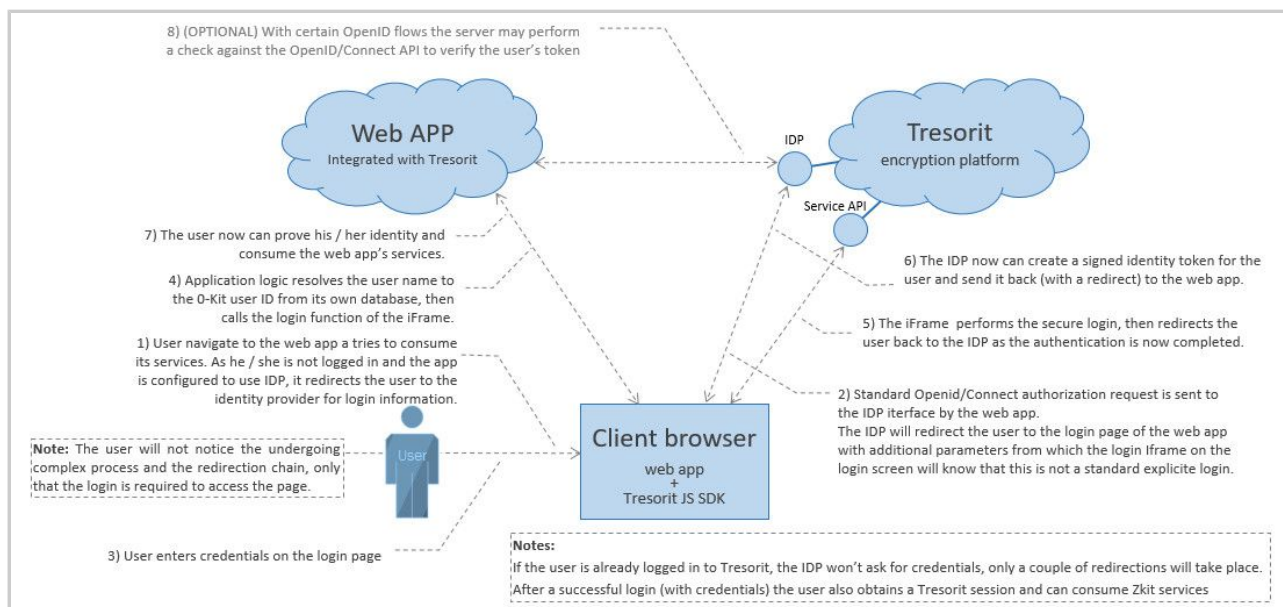
### Explicit login flow explained



## Implicit login (IDP)

In this case the login flow is initiated implicitly by the built-in OpenID Connect identity provider of the ZeroKit server. This occurs when a configured client of the IDP (typically the web app itself) requests the user's identity from the IDP and the provider realizes that the user has no active ZeroKit session. In this case the identity delegation is paused by an intervening login flow (as described previously). After the login succeeded (on the same page and the same way as in the explicit case), the SDK automatically redirects the logged in user to the IDP which will complete the flow and delegate the identity of the user to the web app backend. Please note, that the Promise returned by the login method will not resolve before redirection.

**Note:** Notice the differences between the two flows: 1) The first one is initiated by an explicit navigation to the login page by the user or by the application 2) In the first case, there is no automatic redirect after the successful login, while after the successful login in the implicit flow the SDK will redirect the user back to the IDP endpoint to finalize the identity request.



### Implicit login flow explained

## Tresor creation

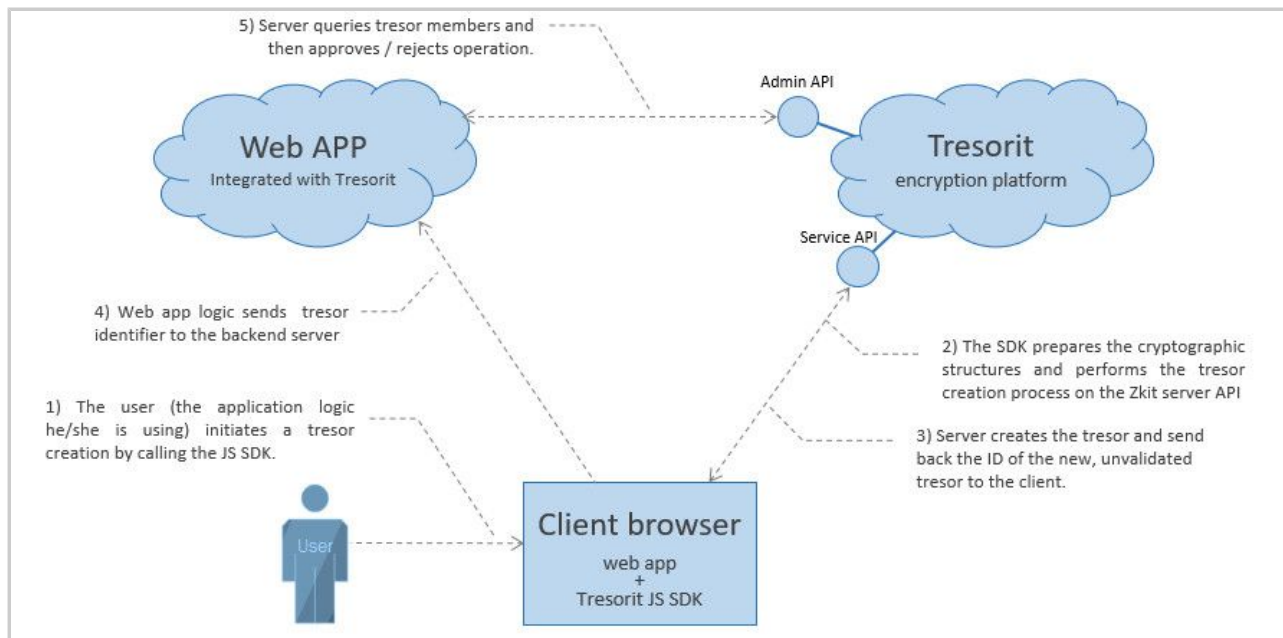
Tresors are the basic units of encryption and sharing. They take care of all the necessary cryptographic operations such as key sharing, key revocation and key change. If a piece of data was once encrypted with the keys of a tresor, the same tresor has to be used to decrypt the data. Tresors can be created by any active user, but the created tresors have to be approved by the application through the administrative API. This process grants application control over the creation of tresors (key-containers) and an opportunity to save the tresor ID in their database before the users start to use them.

**Note:** No metadata about the tresor is handled by Tresorit ZeroKit!

The tresor creation itself is initiated on the client side by calling the Create tresor operation, which will take care for the cryptographic initialization and will return the ID of the newly created tresor. This ID then can be passed to the application backend and used as "operation ID" to approve / reject tresor creation. To get

information about the creator (initial members) of the tresor, the tresor member listing feature of the administrative API can be used.

**Note:** The initial membership(s) of the tresor does not need further acceptance, after the validation of the tresor creation they instantly became valid members.



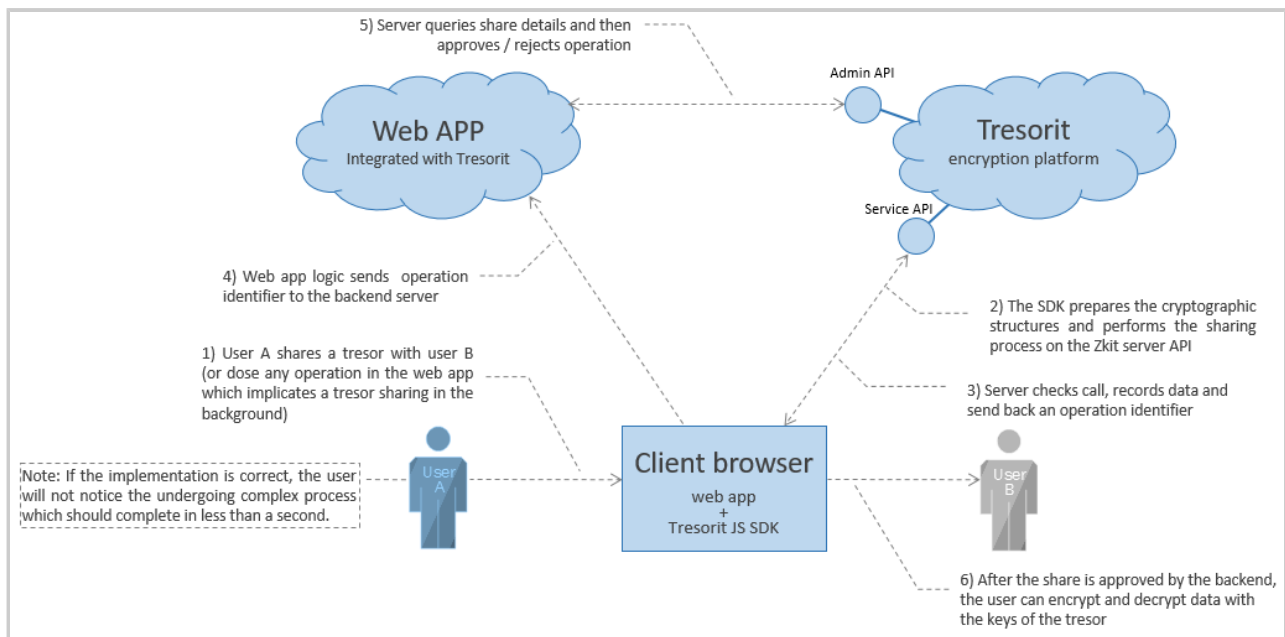
**Tresor creation flow explained**

## Tresor sharing

Tresors can be shared with other users, so they can use the tresor to decrypt data which was encrypted by them or other members of the tresor. Sharing is a key-concept of the tresor and a share can be revoked at any time. After revocation the kicked out member will not be able to retrieve the metadata of the tresor from the platform server, therefore he/she loses the ability to encrypt or decrypt data within the tresor as well.

If a user wants to (or any operation he/she did in the web app needs to) share a tresor with another user, this process must be initialized from the JS SDK and the other user must be also registered into the web app and to the ZeroKit server. After a share is initialized, it has to be approved by the web app backend through the administration API. This approval cycle is the same as described above in the basic approval flow section.

**Note:** The approval may fail due to concurrent operations on the tresor. In that case, the whole process has to be retried. To minimize the chance of failure, the approval (or rejection) should take place immediately after the initialization.



### Tresor sharing explained

## Data handling (encryption / decryption)

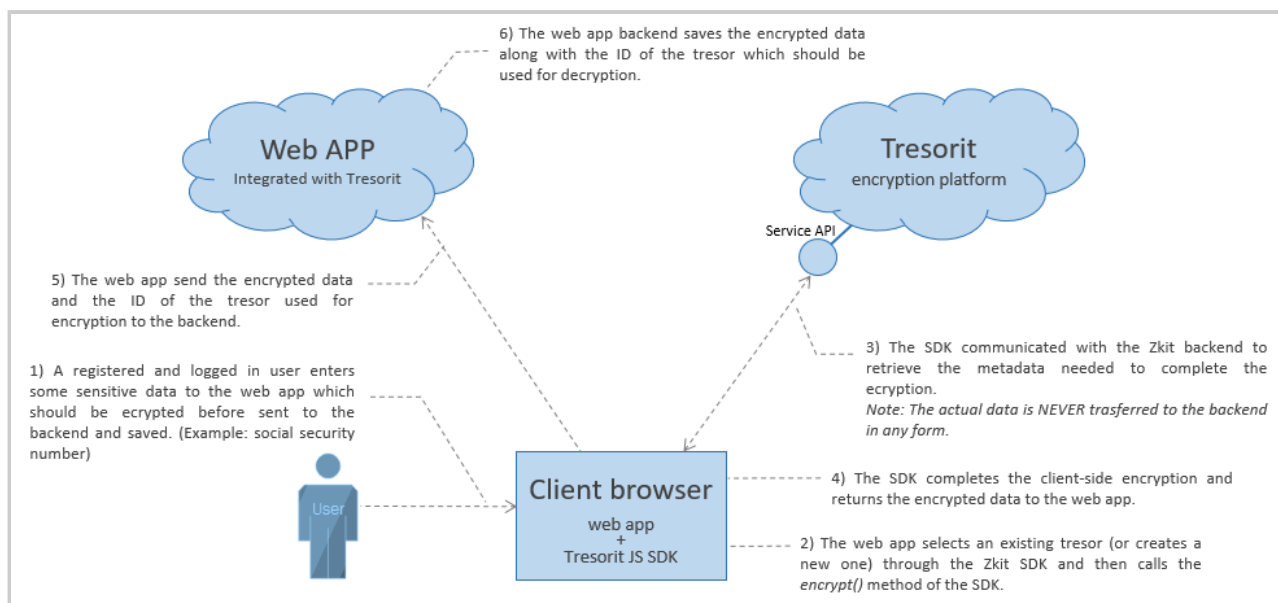
The most common use-case of the ZeroKit SDK is the actual data handling: encryption and decryption. Both operations are executed on the client side through a tresor in which the user is a member. The encrypted data should be stored by the web app backend and provided to the proper clients when they need it, along with the ID of the tresor used for encryption.

### Important:

- The same tresor has to be used for decryption which was used for the encryption of the data.
- Your data never reaches the Tresorit server in any form (even in encrypted form).
- Tresorit does not store any data, it is the responsibility of the web app backend.

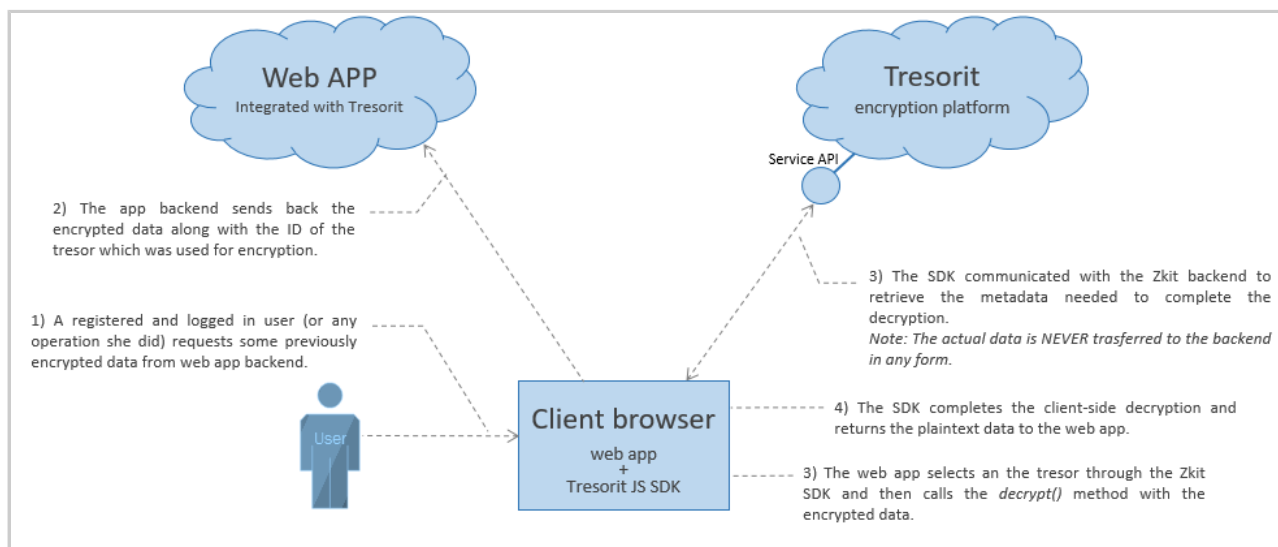
The following figures will help to understand the encryption and decryption flows from the viewpoint of the user and the web app.

## Encryption



### Encryption explained

### Decryption



### Decryption explained

## 8. Changelog

Changes of public API and SDK.

### v1 (2016. June)

- Initial release

### v2 (2016. August)

- **[ change ]** Tresor creation has to be approved by admin API
  - **[ new ]** New error code: `TresorsNotApproved` - thrown when a user tries to use an unvalidated tresor. This also applies when a tresor is created on API V2, but not validated and the user tries to use it with API V1.
  - **[ new ]** New error code: `TresorsAlreadyApproved` - thrown when the administrative API is called twice to validate a tresor.
  - **[ change ]** If the user tries to use an unapproved tresor he / she will receive a "TresorsNotApproved" error.
  - **[ change ]** If the user tries to use an already rejected tresor he / she will receive a "TresorAlreadyDeleted" error.
  - **[ new ]** If the administrative API is called to validate / reject an already validated tresor the caller would receive a "TresorsAlreadyApproved" error
  - **[ new ]** If the administrative API is called to validate / reject an already rejected tresor the caller would receive a "TresorAlreadyDeleted" error.
  - **[ new ]** Tresors created through API/SDK v1 will be automatically approved.
- **[ new ]** Tresor deletion - tresors can be deleted through administrative API
- **[ removed ]** User ID alias handling feature is removed from product.
  - **[ change ]** Init user registration administrative endpoint does not accept an alias anymore.
  - **[ remove ]** Alias (username) field is removed from login iframe.
  - **[ change ]** login call only accepts 0-kit user ID.
  - **[ change ]** User names have to be resolved to user aliases by the application.
- **[ removed ]** Basic ACL handling of tresors (Admin and Contributor role) removed, as this functionality can be fully covered by proper administrative logic of the application.
- **[ new ]** Added support for invitation links, using which users can invite not registered users into a tresor
  - **[ new ]** New iframes are introduced into the system to handle password protected links: `CreateInvitationLinkIframe` and `AcceptInvitationLinkIframe`
  - **[ new ]** New methods added to the sdk to support creating link without passwords.
  - **[ new ]** Added method to get information about invitation links.
- **[ new ]** Added methods to the sdk to support encrypting streamed data

### v3 (2016. October)

- [ **new** ] Added support for modification of custom content files through admin api (i.e. .css files for embedded iFrames)
  - [ **new** ] New method for uploading custom file
  - [ **new** ] New method for listing custom files
  - [ **new** ] New method for deleting custom file
- [ **new** ] Password change feature
  - [ **new** ] New iFrame for password change
- [ **change** ] Embedding URL list (tenant setting) now accepts \* wildcards
- [ **new** ] Password strength now returns more information using zxcvbn
- [ **new** ] Added a common set of cosmetic customization methods to all iframes
- [ **change** ] Administrative API now supports two keys (primary and secondary) for rolling key changes without downtime.

### v4 (2016. December)

- [ **BREAKING** ] Changed the internal representation of user password salts and slightly altered the algorithm used to generate verifiers to improve security, breaking all existing users.

## 8.1 V1-V2 Upgrade action list

### 1. Tresor creation approval

From now on tresor creation need administrative approval, before all requests went through, an equivalent of approving all requests.

1. Implement calling the approve-tresor-creation endpoint and rejection
2. This endpoint should be called when saving the tresorId
3. To keep current functionality you should approve all tresor creation requests
4. You should either make sure a user never receives the id of an unapproved tresor or handle the related errors.

### 2. Removed alias feature

The alias or username feature of the sdk was removed, including the respective field on the login iframe.

1. You must update how you call the init-user-registration api endpoint, you shouldn't pass it the username.
2. You should update your sites structure and include the username input in your login form
3. You should implement mapping usernames to tresorit userId-s on your application backend.
4. When logging in, you should make a call to the application backend to figure out the userId belonging to the user currently logging in
5. When calling the login function on the iframe you should pass it the tresorit userId of the user logging in as the first parameter. This means that the optional callback function moved to the second parameter.