SEVENTH FRAMEWORK PROGRAMME

CloudSpaces

(FP7-ICT-2011-8)

Open Service Platform for the Next Generation of Personal Clouds

D5.1 Framework specs and API descriptions

Due date of deliverable: 30-11-2013
Actual submission date: 12-11-2013

Start date of project: 01-10-2012 Duration: 36 months

Summary of the document

Document Type	Deliverable
Dissemination level	Public
State	Final
Number of pages	49
WP/Task related to this document	WP5
WP/Task responsible	CNC
Author(s)	Refer to contributors list
Partner(s) Contributing	CNC, URV, EOS
Document ID	CLOUDSPACES_D5.1_131112_Public.pdf
Abstract	This report includes guidelines, open specifications, and documented best-practices for achieving syntactic interoperability of Personal Clouds. Data API specs and early prototypes. Persistence API spec and early prototypes. eyeOS early incomplete integration with data services including eyeSync and eyeFiles. Candidate tools specs for eyeOS demonstrator tools demonstrating contact, file, and calendar data use. Early prototype of OpenStack Web data management integration with data services. All software will be released under an open source license such as GPLv3.
Keywords	interoperability, API, prototype, persistence, services, Personal Clouds

Contributors

Name	Last name	Affiliation	Email
John	Lenton	CNC	john.lenton@canonical.com
Adrián	Moreno Martínez	URV	adrian.moreno@urv.cat
Pedro	García López	URV	pedro.garcia@urv.cat
José Miguel	García López	TST	jmgarcia@tissat.es
Ferran	Caceres	EOS	ferran.caceres@eyeos.com

Table of Contents

1 Executive summary

This deliverable includes guidelines, open specifications, and documented best-practices for achieving syntactic interoperability of Personal Clouds. This document describes an open service platform for Personal Clouds including a number of services ensuring both horizontal and vertical interoperability. In Section 3 we review and discuss some of the current Cloud standards and analyse whether they are suitable for the project.

Horizontal interoperability is focused on exchanging and sharing information between heterogeneous Personal Clouds. It includes services for data storage and sharing to shared workspaces. Horizontal interoperability will be demonstrated between StackSync and U1 Personal Clouds using the Share and Store APIs.

In Section 4 we describe the first version of the sharing protocol, which will enable different Personal Clouds to share resources among them via an API, without forcing users to be in the same provider. More generally, the sharing protocol will create a freely-implementable and generic methodology for allowing Personal Cloud interoperability. We provide the protocol specification as well as different use case scenarios that the protocol may face.

Next, we describe the storage API in Section 5. This API is meant to consolidate a standard among Personal Clouds to achieve an easier interoperability and facilitate access to third parties. We specify the different actions and resources available as well as the needed parameters to perform queries and the distinct error codes available.

The persistence API is presented in Section 6. Persistence relies in the U1DB implementation, which is a database API for synchronised databases of JSON documents. It allows applications to store documents and synchronise them between machines and devices. The design allows U1DB to be used everywhere, backed by the platform's native data storage capabilities. Meaning that it can used on different platforms, from different languages, and backed on to different databases, and synchronised between all of them.

Vertical Interoperability refers to external third-party applications accessing a Personal Cloud. This includes the client prototypes presented in Section 7 and the aforementioned services: the storage API and persistence service for applications requiring at least key-value and metadata services over the information stored in the Personal Cloud. Vertical interoperability will be demonstrated thanks to the eyeOs web desktop infrastructure and tools. eyeOS services like eyeFiles and eyeCalendar will demonstrate the aforementioned Store, Share, and Persistence services. Vertical interoperability will be also demonstrated with the integration of a Web Managament Interface on top of OpenStack Swift for accessing heterogeneous Personal Clouds data services.

The goal of this platform is that users retake control of their information stored in Personal Cloud. Users will be able to decide how (access control) and whom (users, applications) can access information stored in their Personal Clouds. The service platform aims to produce open specifications that may be adopted by third-party providers (Personal Clouds, Applications) and thus break the fragmentation of the market and its implicit vendor lock-in.

2 Motivation and Context

In the next few years, users will require ubiquitous and massive network storage to handle their ever-growing digital lives. Every user will handle hundreds of gigabytes to store digital information including photos, videos, work documents and communication flows like emails or social communication.

To meet this demand, current trends show an increasing number of enterprises and users migrating their data to Cloud storage providers. A major selling point for Cloud computing is that it offers on-demand storage capacity that otherwise might not be affordable.

In this line, the Personal Cloud model is a user-centric solution to manage such massive amounts of digital information. Unlike application-centric models where data is tied to a specific application, user-centric models provide a personal storage service for user data.

The Personal Cloud model defines a ubiquitous storage facility enabling the unified and location-agnostic access to information flows from any device and application. Commercial providers such as Dropbox, SugarSync or Ubuntu One are offering very popular Personal Cloud solutions that keep the information synchronized between different user devices. These solutions also allow sharing information with other users within the same Personal Cloud provider.

The popularity of these killer applications lies behind their easy-to-use Software as a Service (SaaS) storage facade to ubiquitous Infrastructure as a Service (IaaS) storage resources like Amazon S3 and others. In a recent report, Forrester research forecasts a market of $12 billion in the US in paid subscriptions to personal clouds by 2016. This growing popularity of Personal Clouds is also attracting the major players in the market, and Google, Microsoft, Amazon or Apple are offering integrated solutions in this field.

3 Related work

In this section, we will be reviewing and discussing related works that seek to tackle the problem of interoperability from different angles. Starting from the common storage interface to access Personal Cloud resources, and then moving to the sharing protocol to allow Personal Clouds to establish a relationship and allow their users to share resources with users in other Personal Clouds, preventing the vendor lock-in problem. Finally, we will address the persistence interface and examine the previous work done in this field.

We have analyzed different Cloud standards in order to consider their adoption when building the Personal Cloud storage API and sharing protocol. First, we evaluated the Cloud Data Management Interface (CDMI)[1], which is a standard by the Storage Networking Industry Association (SNIA). CDMI defines the functional interface that applications will use to create, retrieve, update and delete data elements from the Cloud. Among other features, CDMI allows clients to discover the capabilities available in the cloud storage offering and associate metadata with containers and the objects they contain. However, CDMI is applied at a lower level than Personal Clouds —i.e. Cloud object storage solutions such as OpenStack or Amazon Web Services— and does not provide any solution related to Personal Cloud.

The Open Cloud Computing Interface (OCCI)[2] defines a protocol and API for management tasks. OCCI was originally initiated to create a remote management API for IaaS model based Services, allowing for the development of interoperable tools for common tasks including deployment, autonomic scaling and monitoring. But again, OCCI is focused on Cloud providers and it is not suitable for Personal Clouds as it cannot address many distinctive features and functionalities offered by these kind of services. Other analyzed Cloud standards suffer from the same constraints that prevents their adoption on the Personal Cloud field.

We also evaluated existing storage API specifications from well-known Personal Clouds such as Dropbox[3], Box[4], SugarSync[5], and the likes. However, all analyzed APIs are very coupled to their services. This coupling is noticeable when dealing with files and other provider-specific models. For example, whereas SugarSync organizes its files and folders in albums, files are indexed by identifiers, and deleted files are moved to a special album that acts as a recycle bin; Dropbox does not enclose its folders and files in bigger containers, indexes files by their path instead of identifiers, and deleted files are tagged as deleted but not moved to anywhere.

As far as we know, there exists no sharing protocol to allow two heterogeneous Personal Clouds to reach an agreement and authorize users of both Personal Clouds to seamlessly share between them. Similarly, to address the data persistence issue we analyzed our partner's solution called U1DB[6]. U1DB is a database API for synchronized databases of JSON documents. It allows applications to store documents and synchronize them between machines and devices. U1DB itself is not a database: instead, it is an API which can be backed by any database for storage. This means that you can use U1DB on different platforms, from different languages, and backed on to different databases, and sync between all of them. U1DB completely fits the needs of the project and its specification is described later in this document.

[1]http://www.snia.org/cdmi
[2]http://occi-wg.org/
[3]https://www.dropbox.com/developers/core/docs
[4]http://developers.box.com/get-started/
[5]https://www.sugarsync.com/developer
[6]https://one.ubuntu.com/developer/data/u1db/index

4 Sharing protocol

The sharing protocol enables different Personal Clouds to share resources among them via an API, without forcing users to be in the same provider. More generally, the sharing protocol creates a freely-implementable and generic methodology for allowing Personal Cloud interoperability.

4.1 Prerequisites

Having two Personal Clouds (Personal Cloud 1 and Personal Cloud 2) that pretend to interoperate with each other. They must meet the following requirements before using the present specification.

1. Once the sharing process is completed, Personal Clouds must use APIs to access protected resources. In case they do not implement the Storage API proposed in Section 5, Personal Cloud 1 must implement an adapter to access Personal Cloud 2 API, and vice versa.
2. Personal Cloud 1 must be registered in Personal Cloud 2 and validated as an authorized service in order to obtain its credentials, and vice versa. The method in which Personal Clouds register with each other and agree to cooperate is beyond the scope of this specification.

4.2 Request URL

The sharing protocol defines three endpoints:

- **Share URL.** The URL used to present the sharing proposal to the user and obtain authorization.
- **Unshare URL.** The URL used to finish the sharing agreement.
- **Credentials URL.** The URL used to provide the access credentials.

4.3 Interoperability process overview

The sharing protocol is done in three steps:

1. User A invites User B to its folder located in Personal Cloud A.
2. Personal Cloud A creates the sharing proposal.
3. Personal Cloud A sends the access credentials to Personal Cloud B.

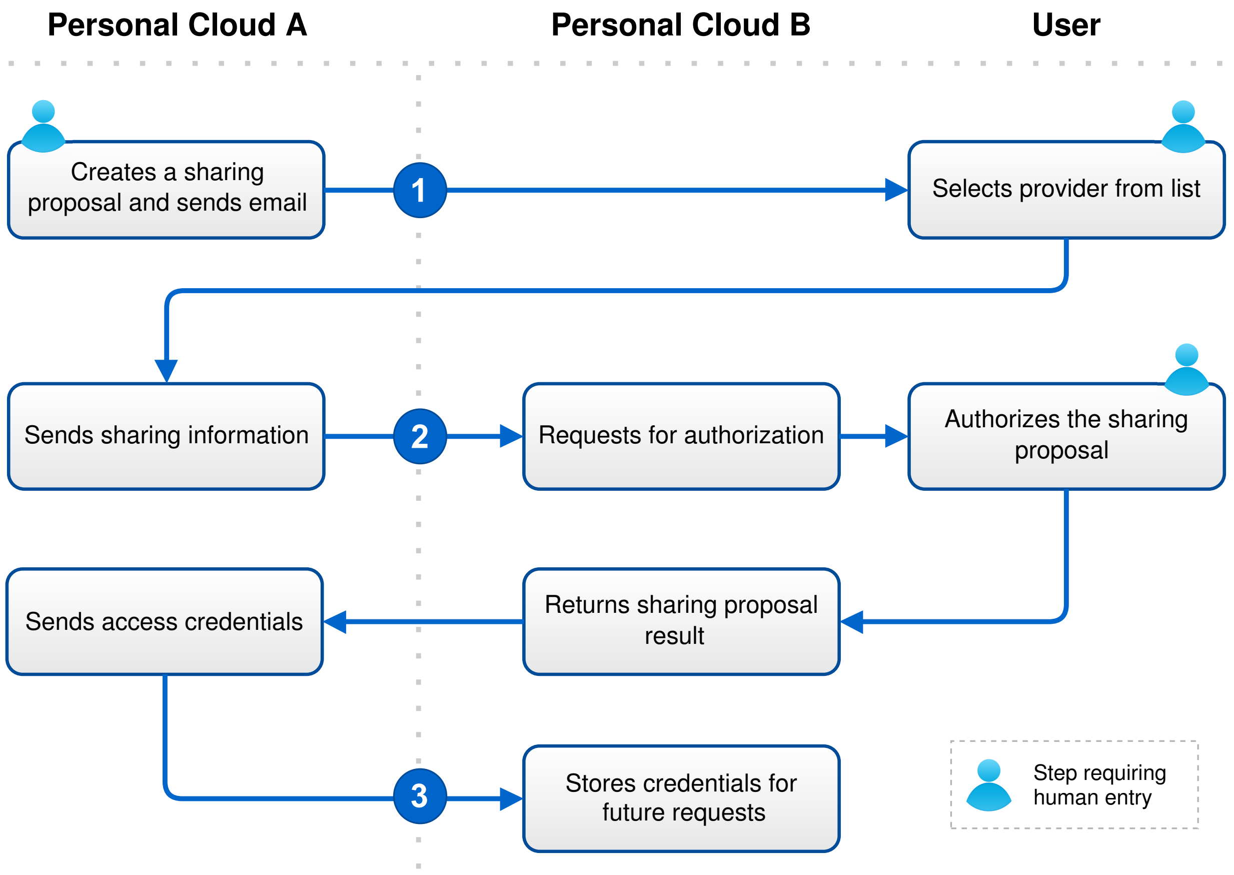

Personal Cloud A
Personal Cloud B
User
Creates a sharing proposal and sends email
1
Selects provider from list
Sends sharing information
2
Requests for authorization
Authorizes the sharing proposal
Sends access credentials
Returns sharing proposal result
3
Stores credentials for future requests
Step requiring human entry

Figure 1: CloudSpaces sharing process flow

In Figure 1 we can observe the sharing process divided in the three steps commented above. First, a user in Personal Cloud A expresses its intention of sharing a file with a external user (User B). Personal Cloud A will send an email with information about the proposal to the external user. The external user will select its favourite Personal Cloud, in which it has an account, namely Personal Cloud B. In the second step, Personal Cloud A will create the sharing proposal and send it to Personal Cloud B, which will require User B to authorize the proposal. The result of the proposal will be returned to Personal Cloud A. Finally, Personal Cloud A will hand the access credentials over the Personal Cloud B, granting forthcoming access to the shared resource.

4.4 User invitation

4.4.1 User sends an invitation

User A wants to share a folder with User B. Therefore, User A goes to Personal Cloud A, selects the folder he/she wants to share, and introduces the email of User B, who will receive an email indicating the intention of User A to share a folder with him/her and a link to a website located on Personal Cloud A.

4.4.2 The recipient selects its Personal Cloud

User B clicks on the link and is taken to the Personal Cloud A, where it is asked to select its Personal Cloud from a list of services that have an agreement with Personal Cloud A. User B selects Personal Cloud B.

4.4.3 Creating the sharing proposal

At this time, Personal Cloud A creates the sharing proposal. To create a sharing proposal, Personal Cloud A sends an HTTP request to Personal Cloud B's share URL. The Personal Cloud B documentation specifies the HTTP method for this request, and HTTP POST is RECOMMENDED.

Field	**Description**
`share_id`	A random value that uniquely identifies the sharing proposal
`resource_url`	An absolute URL to access the shared resource located in Personal Cloud A.
`owner_name`	The name corresponding to the owner of the folder
`owner_email`	The email corresponding to the owner of the folder
`folder_name`	The name of the folder
`permission`	Permissions granted to the recipient. Options are `READ-ONLY` and `READ-WRITE`.
`recipient`	The email corresponding to the user who the folder has been shared with
`callback`	An absolute URL to which the Personal Cloud B will redirect the User back when the Accepting the invitation step is completed.
`protocol_version`	MUST be set to 1.0. Services MUST assume the protocol version to be 1.0 if this parameter is not present.

4.5 Invitation acceptance

4.5.1 The user accepts the invitation

Personal Cloud B displays User B the details of the folder invitation request. User B must provide its credentials and explicitly accept the invitation.

4.5.2 Returning the proposal response

Once Personal Cloud B has obtained approval or denial from User B, Personal Cloud B must use the `callback` to inform Personal Cloud A about the User B decision.

Personal Cloud B uses the `callback` to construct an HTTP GET request, and directs the User's web browser to that URL with the following values added as query parameters:

Field	Description
`share_id`	A random value that uniquely identifies the sharing proposal
`accepted`	A string indicating whether the invitation has been accepted or denied. `true` and `false` are the only possible values.

4.6 Access credentials

4.6.1 Granting access to the service

When Personal Cloud A receives the proposal result it must provide the access credentials to Personal Cloud B in order to be able to obtain the shared resource. Personal Cloud A sends an HTTP request to Personal Cloud B's credential URL. The Personal Cloud B documentation specifies the HTTP method for this request, and HTTP POST is recommended.

Personal Cloud A specifies what type of authentication protocol and version must be used to access the resource. To this end, Personal Cloud B must check the `auth_protocol` and `auth_protocol_version` parameters. The authentication protocol and version used by Personal Cloud A is beyond the scope of this specification, but OAuth 1.0a or OAuth 2.0 is recommended.

Field	Description
`share_id`	A random value that uniquely identifies the sharing proposal
`auth_protocol`	The authentication protocol used to access the shared resource (e.g. `oauth`)
`auth_protocol_version`	The version of the authentication protocol (e.g. `1.0a`)

Other authentication-specific parameters are sent together with the above parameters, these parameters may include values like tokens, timestamps or signatures.

5 Store API

In this section we will describe the storage API. This API is meant to consolidate a standard among Personal Clouds to achieve an easier interoperability and facilitate access to third parties. We will specify the different actions and resources available as well as the needed parameters to perform queries and the distinct error codes available.

5.1 Error Handling

Errors are returned using standard HTTP error code syntax. Any additional info is included in the body of the return call, JSON-formatted. Error codes not listed in figure 5.1a are in the REST API methods listed below.

Code	**Description**
`400`	Bad input parameter. Error message should indicate which one and why.
`401`	Authorization required. The presented credentials, if any, were not sufficient to access the folder resource. Returned if an application attempts to use an access token after it has expired.
`403`	Forbidden. The requester does not have permission to access the specified resource.
`404`	File or folder not found at the specified path.
`405`	Request method not expected (generally should be `GET` or `POST`).
`5`*`xx`*	Server error.

Table 5.1a: Standard API errors

5.2 Getting user representation

Returns the JSON representation of the currently authorized user.

URL structure

`API_ROOT/`

Method

`GET`

Sample JSON return value

```
{
  "volumes" : [
    {
      "volume_id": "9834593",
      "root_node_id": "42342343242",
      "resource_path": "/api/metadata/volumes/42342343242",
      "name": "Photos",
      "when_created": <timestamp>
      "content_path": "/content/124324234"
      "x_attributes": {}
    },
    {
      "volume_id": "32423443",
      "root_node_id" : "124324234",
      "resource_path": "/api/metadata/volumes/124324234",
      "name": "Documents",
      "when_created": <timestamp>
      "content_path": "/content/124324234",
      "x_attributes": {}
    }
  ]
}
```

Where:	**`resource_path`**:	path to the resource metadata
	`x_attributes`:	a dictionary containing provider-specific attributes
	`when_created`:	timestamp of creation of the volume
	`content_path`:	path where files content can be retrieved for this volume

5.3 Metadata

Retrieves file and folder metadata.

URL structure

`API_ROOT``/metadata/`*`node_id`*

Method

`GET`

Parameters

`node_id`:	ID of a file or folder.
`list`:	*(optional)* Only applicable when `fileId` corresponds to a folder. The strings `true` and `false` are valid values. `true` is the default. If `true`, the folder's metadata will include a `contents` field with a list of metadata entries for the contents of the folder. If `false`, the `contents` field will be omitted. Note that the root folder will always return its content regardless of this flag.
`include_deleted`:	*(optional)* The strings `true` and `false` are valid values, `false` is the default. If this parameter is set to `true`, then response will include metadata of deleted objects. Note that the target of the metadata call is always returned even when it has been deleted regardless of this flag.
`version`:	*(optional)* If you include a particular version, then only the metadata for that version will be returned.

Returns

The metadata for the file or folder at the given `file_id`. If `file_id` represents a folder and the `list` parameter is `true`, the metadata will also include a listing of metadata for the folder's contents.

Sample JSON return value for a file

```
{
    "client_modified": "2013-03-08 10:36:41.997",
    "content_path": "/api/content/534824681",
    "hash": "-2678858962222278590",
    "is_deleted": false,
    "is_folder": false,
    "mimetype": "application/pdf",
    "node_id": "534824681",
    "parent_node_id": "0",
    "path": "/StackSync_Guide.pdf",
    "resource_path": "/api/metadata/534824681",
    "server_modified": "2013-03-08 10:36:41.997",
    "size": 775412,
    "user_id": "943274",
    "version": 2,
    "volume_id": "793249234",
    "x_attributes": {}
}
```

Sample JSON return value for a folder when list parameter is set to true

If `list` is `false` the `contents` key will simply be omitted from the result.

```
{
    "client_modified": "2013-03-08 10:36:41.994",
    "content_path": "/api/content/538757639",
    "contents": [
        {
            "checksum": "465328065684161006 6",
            "client_modified": "2013-03-08 10:36:42.893",
            "content_path": "/api/content/1916666835",
            "is_deleted": false,
            "is_folder": false,
            "mimetype": "text/plain",
            "node_id": "1916666835",
            "parent_node_id": "538757639",
            "path": "/Documents/important.txt",
            "resource_path": "/api/metadata/1916666835",
            "server_modified": "2013-03-08 10:36:42.893",
            "size": 9453,
            "user_id": "1234",
            "version": 1,
            "x_attributes": {}
        }
    ],
    "is_deleted": false,
    "is_folder": true,
    "is_root": false,
    "node_id": "538757639",
    "parent_node_id": "0",
    "path": "/Documents",
    "resource_path": "/api/metadata/538757639",
    "server_modified": "2013-03-08 10:36:41.994",
    "user_id": "1234",
    "version": 1,
    "volume_id": "793249234",
    "x_attributes": {
        "generation": "45",
        "generation_created": "42",
        "key": "abc:def"
    }
}
```

Return value definitions

`is_folder`	Whether the given entry is a folder.
`is_root`	Whether the given entry is the root folder.
`is_deleted`	.
`path`	The canonical path to the file or directory.
`size`	The file size in bytes (only for files).
`mimetype`	The media type of the file (only for files). `http://www.iana.org/assignments/media-types`
`hash`	The file's hash.
`version`	A unique identifier for the current version of a file. Can be used to detect changes and avoid conflicts.
`filename`	The name of the file or folder.
`user_id`	The name of the user that made modified this specific version.
`node_id`	A unique identifier for a file or folder.
`parent_node_id`	`file_id` of the folder's parent.
`client_modified`	This is the modification time set by the desktop client when the file was last modified, in the standard date format. Since this time is not verified (the server stores whatever the desktop client sends up), this should only be used for display purposes (such as sorting) and not, for example, to determine if a file has changed or not.
`server_modified`	This is the modification time set by the server at the time of processing the file.
`x_attributes`	A dictionary containing provider-specific attributes.
`content_path`	Path to the content.
`resource_path`	Path to the node resource.

Note that `status`, `parent_node_id`, `client_modified`, `server_modified`, `version`, `user`, `path`, and `checksum` are not in the metadata for the root folder.

Status codes

2*xx*	The request was successful. The folder information is in the response body.

See also the standard error codes in figure 5.1a.

5.4 Modifying an existing file

Modifies the metadata of a file or folder.

URL structure

`API_ROOT``/metadata/`*`node_id`*

Method

`PUT`

Returns

Metadata of the file just uploaded

Sample JSON return value

```
{
    "checksum": -2678858962222278590,
    "client_modified": "2013-03-08 10:36:41.997",
    "content_path": "/api/content/534824681",
    "is_deleted": false,
    "is_folder": false,
    "mimetype": "application/pdf",
    "node_id": 534824681,
    "parent_file_id": 0,
    "path": "/StackSync_Guide.pdf",
    "resource_path": "/api/metadata/534824681",
    "server_modified": "2013-03-08 10:36:41.997",
    "size": 775412,
    "user": "Adrian",
    "version": 2
}
```

Status codes

2xx The file has been successfully uploaded

See also the standard error codes in figure 5.1a.

5.5 Creating a file

Uploads a new file.

URL structure

API_ROOT/content/*folder_node_id*

Method

`POST`

Parameters

`file_name`	The file name.
`folder_node_id`	*(optional)* ID of the folder where the file or folder is going to be uploaded. If no ID is passed, it will use the top-level folder. This parameter should not point to a file.

Returns

Metadata of the file just uploaded.

Sample JSON return value

```
{
    "checksum": -2678858962222278590,
    "client_modified": "2013-03-08 10:36:41.997",
    "content_path": "/api/content/534824681",
    "is_deleted": false,
    "is_folder": false,
    "mimetype": "application/pdf",
    "node_id": "534824681",
    "parent_node_id": "0",
    "path": "/StackSync_Guide.pdf",
    "resource_path": "/api/metadata/534824681",
    "server_modified": "2013-03-08 10:36:41.997",
    "size": 775412,
    "user_id": "289342",
    "version": 2
}
```

Status codes

2*xx*	The file has been successfully uploaded
4*xx*	The file name already exists

See also the standard error codes in figure 5.1a.

5.6 Upload an existing file

Modifies a file.

URL structure

`API_ROOT/content/node_id`

Method

PUT

Returns

Metadata of the file just uploaded.

Sample JSON return value

```
{
    "checksum": -2678858962222278590,
    "client_modified": "2013-03-08 10:36:41.997",
    "content_path": "/api/content/534824681",
    "is_deleted": false,
    "is_folder": false,
    "mimetype": "application/pdf",
    "node_id": 534824681,
    "parent_file_id": 0,
    "path": "/StackSync_Guide.pdf",
    "resource_path": "/api/metadata/534824681",
    "server_modified": "2013-03-08 10:36:41.997",
    "size": 775412,
    "user": "Adrian",
    "version": 2
}
```

Status codes

$2xx$	The file has been successfully uploaded

See also the standard error codes in figure 5.1a.

5.7 Download a file

Retrieves a file's content.

URL structure

API_ROOT/content/*node_id*

Method

GET

Parameters

node_id	ID of the file to download.
version	*(optional)* If you want to download a specific version.

Returns

The content of the file in the body response.

5.8 Delete a file or folder

URL structure

API_ROOT/content/*node_id*

Method

DELETE

Parameters

node_id	ID of the file to download.

Returns

The metadata of the file just deleted.

5.9 Create a folder

Touch a folder in the server.

URL structure

API_ROOT`/metadata`

Method

`POST`

Parameters

`folder_name`	Name of the folder to be created.
`parent`	*(optional)* ID of the folder where the folder is going to be created. If no ID is passed, it will use the top-level folder. This parameter should *not* point to a file.

Returns

Metadata of the folder just created.

Status codes

201	The folder has been successfully created

See also the standard error codes in figure 5.1a.

6 Persistence API

The data persistance API implemented has been called “U1DB”.

U1DB is a database API for synchronised databases of JSON documents. It’s simple to use in applications, and allows apps to store documents and synchronise them between machines and devices. U1DB is the database designed to work everywhere, backed by the platform’s native data storage capabilities. This means that you can use U1DB on different platforms, from different languages, and backed on to different databases, and sync between all of them.

The API for U1DB looks similar across all different implementations. This API is described at The high-level API. To actually use U1DB you’ll an implementation; a version of U1DB made available on your choice of platform, in your choice of language, and on your choice of backend database.

If you’re interested in using U1DB in an application, look at The high-level API first, and then choose one of the implementations and read about exactly how the U1DB API is made available in that implementation. Get going quickly with the downloads and Quickstart guide.

If you’re interested in hacking on U1DB itself, read about the rules for U1DB and The reference implementation.

6.1 The high-level API

The U1DB API has three separate sections: document storage and retrieval, querying, and sync. Here we describe the high-level API. Remember that you will need to choose an implementation, and exactly how this API is defined is implementation-specific, in order that it fits with the language’s conventions.

6.1.1 Document storage and retrieval

U1DB stores documents. A document is a set of nested key-values; basically, anything you can express with JSON. Implementations are likely to provide a Document object “wrapper” for these documents; exactly how the wrapper works is implementation-defined.

Creating documents To create a document, use `create_doc()` or `create_doc_from_json()`. Code examples below are from the reference implementation in Python. `create_doc()` takes a dictionary-like object, and `create_doc_from_json()` a JSON string.

```
>>> import u1db
>>> db = u1db.open("mydb1.u1db", create=True)
>>> doc = db.create_doc({"key": "value"}, doc_id="testdoc")
>>> doc.content
{'key': 'value'}
>>> doc.doc_id
'testdoc'
```

Retrieving documents The simplest way to retrieve documents from a u1db is by calling `get_doc()` with a `doc_id`. This will return a `Document` object[7].

```
>>> import u1db
>>> db = u1db.open("mydb4.u1db", create=True)
>>> doc = db.create_doc({"key": "value"}, doc_id="testdoc")
>>> doc1 = db.get_doc("testdoc")
>>> doc1.content
{u'key': u'value'}
>>> doc1.doc_id
'testdoc'
```

And it's also possible to retrieve many documents by `doc_id`.

```
>>> import u1db
>>> db = u1db.open("mydb5.u1db", create=True)
>>> doc1 = db.create_doc({"key": "value"}, doc_id="testdoc1")
>>> doc2 = db.create_doc({"key": "value"}, doc_id="testdoc2")
>>> for doc in db.get_docs(["testdoc2","testdoc1"]):
...     print doc.doc_id
testdoc2
testdoc1
```

Note that `u1db.Database.get_docs()` returns the documents in the order specified.

Editing existing documents Editing an existing document is done with `put_doc()`. This is separate from `create_doc()` so as to avoid accidental overwrites. `put_doc()` takes a `Document` object, because the object encapsulates revision information for a particular document. This revision information must match what is stored in the database, so we can make sure you are not overwriting another version of the document that you do not know about (eg, new documents that came from a background sync while you were editing your copy).

```
>>> import u1db
>>> db = u1db.open("mydb2.u1db", create=True)
>>> doc1 = db.create_doc({"key1": "value1"}, doc_id="doc1")

>>> # the next line should fail because it's creating a doc that
   already exists
>>> db.create_doc({"key1fail": "value1fail"}, doc_id="doc1")
Traceback (most recent call last):
    ...
RevisionConflict

>>> # Now editing the doc with the doc object we got back...
>>> doc1.content["key1"] = "edited"
>>> db.put_doc(doc1)
'...'
```

[7] Alternatively if a factory function was passed into `u1db.open()`, `get_doc()` will return whatever type of object the factory function returns.

```
>>> doc2 = db.get_doc(doc1.doc_id)
>>> doc2.content
{u'key1': u'edited'}
```

Finally, deleting a document is done with `delete_doc()`.

```
>>> import u1db
>>> db = u1db.open("mydb3.u1db", create=True)
>>> doc = db.create_doc({"key": "value"})
>>> db.delete_doc(doc)
'...'
>>> db.get_doc(doc.doc_id)
>>> doc = db.get_doc(doc.doc_id, include_deleted=True)
>>> doc.content
```

Document functions

- `create_doc()`
- `create_doc_from_json()`
- `put_doc()`
- `get_doc()`
- `get_docs()`
- `get_all_docs()`
- `delete_doc()`
- `whats_changed()`

6.1.2 Querying

To retrieve documents other than by `doc_id`, you query the database. Querying a U1DB is done by means of an index. To retrieve only some documents from the database based on certain criteria, you must first create an index, and then query that index.

An index is created from ''index expressions''. An index expression names one or more fields in the document. A simple example follows: view many more examples here.

Given a database with the following documents:

```
>>> import u1db
>>> db1 = u1db.open("mydb6.u1db", create=True)
>>> jb = db1.create_doc({"firstname": "John", "surname": "Barnes"
  , "position": "left wing"})
>>> jm = db1.create_doc({"firstname": "Jan", "surname": "Molby",
  "position": "midfield"})
```

```
>>> ah = db1.create_doc({"firstname": "Alan", "surname": "Hansen"
    , "position": "defence"})
>>> jw = db1.create_doc({"firstname": "John", "surname": "Wayne",
    "position": "filmstar"})
```

an index expression of `"firstname"` will create an index that looks (conceptually) like this

index expression value	**document**
Alan	ah
Jan	jm
John	jb
John	jw

and that index is created with:

```
>>> db1.create_index("by-firstname", "firstname")
>>> sorted(db1.get_index_keys('by-firstname'))
[(u'Alan',), (u'Jan',), (u'John',)]
```

—that is, create an index with a name and one or more index expressions. (Exactly how to pass the name and the list of index expressions is something specific to each implementation.)

Index expressions An index expression describes how to get data from a document; you can think of it as describing a function which, when given a document, returns a value, which is then used as the index key.

Name a field. A basic index expression is a dot-delimited list of nesting fieldnames, so the index expression `field.sub1.sub2` applied to a document with below content:

```
>>> import u1db
>>> db = u1db.open('mydb7.u1db', create=True)
>>> db.create_index('by-subfield', 'field.sub1.sub2')
>>> doc1 = db.create_doc({"field": {"sub1": {"sub2": "hello", "
   sub3": "not selected"}}})
>>> db.get_index_keys('by-subfield')
[(u'hello',)]
```

gives the index key “hello”, and therefore an entry in the index of

Index key	**doc**
hello	doc1

Name a list. If an index expression names a field whose contents is a list of strings, the document will have multiple entries in the index, one per entry in the list. So, the index expression `field.tags` applied to a document with content:

```
>>> import u1db
>>> db = u1db.open('mydb8.u1db', create=True)
>>> db.create_index('by-tags', 'field.tags')
>>> doc2 = db.create_doc({"field": {"tags": [ "tag1", "tag2", "
   tag3" ]}})
>>> sorted(db.get_index_keys('by-tags'))
[(u'tag1',), (u'tag2',), (u'tag3',)]
```

gives index entries

Index key	doc
tag1	doc2
tag2	doc2
tag3	doc2

Subfields of objects in a list. If an index expression points at subfields of objects in a list, the document will have multiple entries in the index, one for each object in the list that specifies the denoted subfield. For instance the index expression `managers.phone_number` applied to a document with content:

```
>>> import u1db
>>> db = u1db.open('mydb9.u1db', create=True)
>>> db.create_index('by-phone-number', 'managers.phone_number')
>>> doc3 = db.create_doc(
...     {"department": "department of redundancy department",
...     "managers": [
...         {"name": "Mary", "phone_number": "12345"},
...         {"name": "Katherine"},
...         {"name": "Rob", "phone_number": "54321"}]})
>>> sorted(db.get_index_keys('by-phone-number'))
[(u'12345',), (u'54321',)]
```

would give index entries:

Index key	doc
12345	doc3
54321	doc3

Transformation functions. An index expression may be wrapped in any number of transformation functions. A function transforms the result of the contained index expression: for example, if an expression `name.firstname` generates “John” when applied to a document, then `lower(name.firstname)` generates “john”.

Available transformation functions are:

- `lower(index_expression)` - lowercase the value

- `split_words(index_expression)` - split the value on whitespace; will act like a list and add multiple entries to the index
- `number(index_expression,width)` - takes an integer value, and turns it into a string, left padded with zeroes, to make it at least as wide as width; or nothing if the field type is not an integer.
- `bool(index_expression)` - takes a boolean value and turns it into '0' if false and '1' if true, or nothing if the field type is not boolean.
- `combine(index_expression1,index_expression2,...)` - Combine the values of an arbitrary number of sub expressions into a single index.

So, the index expression `splitwords(lower(field.name))` applied to a document with content:

```
>>> import u1db
>>> db = u1db.open('mydb10.u1db', create=True)
>>> db.create_index('by-split-lower', 'split_words(lower(field.
    name))')
>>> doc4 = db.create_doc({"field": {"name": "Bruce David
    Grobbelaar"}})
>>> sorted(db.get_index_keys('by-split-lower'))
[(u'bruce',), (u'david',), (u'grobbelaar',)]
```

gives index entries

Index key	**doc**
bruce	doc3
david	doc3
grobbelaar	doc3

Querying an index Pass an index key or a tuple of index keys (if the index is on multiple fields) to `get_from_index`; the last index key in each tuple (and only the last one) can end with an asterisk, which matches initial substrings. So, querying our `by-firstname` index from above:

```
>>> johns = [d.doc_id for d in db1.get_from_index("by-firstname",
     "John")]
>>> assert(jw.doc_id in johns)
>>> assert(jb.doc_id in johns)
>>> assert(jm.doc_id not in johns)
```

will return the documents with ids: 'jw', 'jb'.

`get_from_index("by_firstname","J*")` will match all index keys beginning with "J", and so will return the documents with ids: 'jw', 'jb', 'jm'.

```
>>> js = [d.doc_id for d in db1.get_from_index("by-firstname", "J
    *")]
```

```
>>> assert(jw.doc_id in js)
>>> assert(jb.doc_id in js)
>>> assert(jm.doc_id in js)
```

Index functions

- `create_index()`
- `delete_index()`
- `get_from_index()`
- `get_range_from_index()`
- `get_index_keys()`
- `list_indexes()`

6.1.3 Synchronising

U1DB is a syncable database. Any U1DB can be synced with any U1DB server; most U1DB implementations are capable of being run as a server. Synchronising brings both the server and the client up to date with one another; save data into a local U1DB whether online or offline, and then sync when online.

Pass an HTTP URL to sync with that server.

Synchronising databases which have been independently changed may produce conflicts. Read about the U1DB conflict policy and more about synchronising at Conflicts, Synchronisation, and Revisions.

Running your own U1DB server is implementation-specific. The reference implementation is able to be run as a server.

6.1.4 Dealing with conflicts

Synchronising a database can result in conflicts; if your user changes the same document in two different places and then syncs again, that document will be ‘’in conflict‘‘, meaning that it has incompatible changes. If this is the case, `has_conflicts` will be true, and `put_doc` to a conflicted doc will give a `ConflictedDoc` error. To get a list of conflicted versions of the document, do `get_doc_conflicts()`. Deciding what the final unconflicted document should look like is obviously specific to the user's application; once decided, call `resolve_doc()` to resolve and set the final resolved content.

Synchronising Functions

- `sync()`
- `get_doc_conflicts()`
- `resolve_doc()`

6.2 U1DB implementations

Let us move to concrete U1DB implementations

6.2.1 U1DB APIs

Implementations should expose the APIs presented previously and choose and utilize a storage backend appropriate and meaningful for the platform they cover to realize the semantics sketched.

User API these APIs for databases and documents should be rendered with as little as possible language variations so that user can transfer and reuse their u1db knowledge between languages

Sync API this API is HTTP based and allows with few requests to exchange the delta between the current state of a database – that means the changed documents – and a previous synchronization point. The exchange is asymmetric insofar are there is a source and a target, the target is the HTTP server implementing the API and after a sync conflicts are registered/created only on the source. The Sync API is also the means by which different implementations can introperate and exchange data, possibly through an intermediate target if they only implement the source side.

6.2.2 U1DB test suite

Together with the reference implementation comes an extensive test suite that checks the methods and semantics of the API, both the user API and the syncing with a server aspect of it.

The test suite is written in Python as the implementation, but that as we proved can be reused for other languages and implementations using bridging techniques either pre-existing like Cython which we employ to reuse the suite for the C implementation, or ad hoc techniques like passing operation requests over a pipe for interpretation by a small driver as we use for the Javascript implementation.

An implementation to be considered compliant and able to interoperate and sync with others should pass these tests.

6.2.3 Available implementations

Currently the following implementations exist:

- compliant/complete
  - C backed by SQLite
  - Python backed by SQLite and also the reference implementation
  - Javascript for browser use, backed by local storage
- not compliant/incomplete
  - Qt C++ provides declarative, easy to use, local data storage for QML applications, part of the Ubuntu SDK
  - Vala
  - Go

6.2.4 The reference implementation

The u1db reference implementation is written in Python, with a SQLite back end. It can be used as a real working implementation by Python code. It is also used to document and test how u1db should work; it has a comprehensive test suite. Implementation authors should port the u1db reference test suite in order to test that their implementation is correct; in particular, sync conformance is defined as being able to sync with the reference implementation.

Fetch with `bzr branch lp:u1db` or from Launchpad.

To open a new database, use `u1db.open`:

`u1db.open(path, create, document_factory=None)`

Open a database at the given location.

Will raise `u1db.errors.DatabaseDoesNotExist` if `create=False` and the database does not already exist.

Parameters

`path` The filesystem path for the database to open.

`create` `True/False`, should the database be created if it doesn't already exist?

`document_factory` A function that will be called with the same parameters as `Document.__init__`.

Returns An instance of `Database`.

Opening returns a `Database` object.

`class u1db.Database`

A JSON Document data store.

This data store can be synchronized with other `u1db.Database` instances.

`close()`

Release any resources associated with this database.

`create_doc(content, doc_id=None)`

Create a new document.

You can optionally specify the document identifier, but the document must not already exist. See ‘`put_doc`’ if you want to override an existing document.

If the database specifies a maximum document size and the document exceeds it, create will fail and raise a `DocumentTooBig` exception.

Parameters

- **`content`** A Python dictionary.
- **`doc_id`** An optional identifier specifying the document id.

Returns An instance of `Document`.

`create_doc_from_json(json, doc_id=None)`

Create a new document.

You can optionally specify the document identifier, but the document must not already exist. See ‘`put_doc`’ if you want to override an existing document.

If the database specifies a maximum document size and the document exceeds it, create will fail and raise a `DocumentTooBig` exception.

Parameters

- **`json`** The JSON document string
- **`doc_id`** An optional identifier specifying the document id.

Returns An instance of `Document`.

`create_index(index_name, *index_expressions)`

Create an named index, which can then be queried for future lookups.

Creating an index which already exists is not an error, and is cheap.

Creating an index which does not match the `index_expressions` of the existing index is an error.

Creating an index will block until the expressions have been evaluated and the index generated.

Parameters

`index_name` A unique name which can be used as a key prefix

`index_expressions` Index expressions defining the index information.
Examples:
`fieldname`, or *`fieldname.subfieldname`* to index alphabetically sorted on the contents of a field.
`number(`*`fieldname, width`*`)` to index numerically sorted on the contents of a field.
`lower(`*`fieldname`*`)` to index alphabetically sorted on the case-normalized contents of a field.

`delete_doc(doc)`

Mark a document as deleted.

Will abort if the current revision doesn't match `doc.rev`. This will also set `doc.content` to `None`.

Parameters

`doc` The document we are removing.

`delete_index(index_name)`

Remove a named index.

Parameters

`index_name` The name of the index we are removing.

`get_all_docs(include_deleted=False)`

Get the JSON content for all documents in the database.

Parameters

`include_deleted` If set to True, deleted documents will be returned with empty content. Otherwise deleted documents will not be included in the results.

Returns `(generation,[Document])` The current generation of the database, followed by a list of all the documents in the database.

`get_doc(doc_id, include_deleted=False)`

Get the JSON string for the given document.

Parameters

`doc_id` The unique document identifier

`include_deleted` If set to `True`, deleted documents will be returned with empty content. Otherwise asking for a deleted document will return `None`.

Returns a `Document` object.

`get_doc_conflicts(doc_id)`

Get the list of conflicts for the given document.

The order of the conflicts is such that the first entry is the value that would be returned by "`get_doc`".

Parameters

`doc_id` The unique document identifier

Returns `[doc]` A list of the `Document` entries that are conflicted.

`get_docs(doc_ids, check_for_conflicts=True, include_deleted=False)`

Get the JSON content for many documents.

Parameters

`doc_ids` A list of document identifiers.

`check_for_conflicts` If set to `False`, then the conflict check will be skipped, and `None` will be returned instead of `True`/`False`.

`include_deleted` If set to `True`, deleted documents will be returned with empty content. Otherwise deleted documents will not be included in the results.

Returns iterable giving the `Document` object for each document id in matching `doc_ids` order.

`get_from_index(index_name, *key_values)`

Return documents that match the keys supplied.

You must supply exactly the same number of values as have been defined in the index. It is possible to do a prefix match by using `*` to indicate a wildcard match. You can only supply `*` to trailing entries (e.g. `val, *, *` is allowed, but `*, val, val` is not.). It is also possible to append a `*` to the last supplied value (e.g. `val*, *, *` or `val, val*, *`, but not `val*, val, *`)

Parameters

- **`index_name`** The index to query
- **`key_values`** values to match. eg, if you have an index with 3 fields then you would have: `get_from_index(index_name,val1,val2,val3)`

Returns List of `Document`

`get_index_keys(index_name)`

Return all keys under which documents are indexed in this index.

Parameters

- **`index_name`** The index to query

Returns A list of tuples of indexed keys.

`get_range_from_index(index_name, start_value, end_value)`

Return documents that fall within the specified range.

Both ends of the range are inclusive. For both `start_value` and `end_value`, one must supply exactly the same number of values as have been defined in the index, or pass `None`. In case of a single column index, a string is accepted as an alternative for a tuple with a single value. It is possible to do a prefix match by using `*` to indicate a wildcard match. You can only supply `*` to trailing entries (e.g. `val, *, *` is allowed, but `*, val, val` is not.). It is also possible to append a `*` to the last supplied value (e.g. `val*, *, *` or `val, val*, *`, but not `val*, val, *`)

Parameters

- **`index_name`** The index to query
- **`start_values`** tuples of values that define the lower bound of the range. eg, if you have an index with 3 fields then you would have: `(val1, val2, val3)`
- **`end_values`** tuples of values that define the upper bound of the range. eg, if you have an index with 3 fields then you would have: `(val1, val2, val3)`

Returns List of `Document`

`get_sync_target()`

Return a `SyncTarget` object, for another u1db to synchronize with.

Returns An instance of `SyncTarget`.

`list_indexes()`

List the definitions of all known indexes.

Returns A list of `(index-name,[field1,field2])` definitions.

`put_doc(doc)`

Update a document.

If the document currently has conflicts, put will fail.

If the database specifies a maximum document size and the document exceeds it, put will fail and raise a `DocumentTooBig` exception.

Parameters

`doc` A Document with new content.

Returns **`new_doc_rev`**, the new revision identifier for the document.

The `Document` object will also be updated.

`resolve_doc(doc, conflicted_doc_revs)`

Mark a document as no longer conflicted.

We take the list of revisions that the client knows about that it is superseding. This may be a different list from the actual current conflicts, in which case only those are removed as conflicted. This may fail if the conflict list is significantly different from the supplied information. (sync could have happened in the background from the time you `GET_DOC_CONFLICTS` until the point where you `RESOLVE`)

Parameters

`doc` A Document with the new content to be inserted.

`conflicted_doc_revs` A list of revisions that the new content supersedes.

`set_document_factory(factory)`

Set the document factory that will be used to create objects to be returned as documents by the database.

Parameters

`factory` A function that returns an object which at minimum must satisfy the same interface as does the class DocumentBase. Subclassing that class is the easiest way to create such a function.

`set_document_size_limit(limit)`

Set the maximum allowed document size for this database.

Parameters

`limit` Maximum allowed document size in bytes.

`sync(url, creds=None, autocreate=True)`

Synchronize documents with remote replica exposed at url.

Parameters

`url` the URL of the target replica to sync with.

`creds` optional dictionary giving credentials to authorize the operation with the server. For using OAuth the form of `creds` is:

```
{"oauth": {
    "consumer_key": ...,
    "consumer_secret": ...,
    "token_key": ...,
    "token_secret": ...
}}
```

`autocreate` ask the target to create the database if non-existent.

Returns

`local_gen_before_sync`, the local generation before the synchronisation was performed. This is useful to pass into `whats_changed`, if an application wants to know which documents were affected by a synchronisation.

`whats_changed(old_generation=0)`

Return a list of documents that have changed since `old_generation`. This allows apps to only store a db generation before going 'offline', and then when coming back online they can use this data to update whatever extra data they are storing.

Parameters

`old_generation` The generation of the database in the old state.

Returns `(generation,trans_id,[(doc_id,generation,trans_id),...])`
The current generation of the database, its associated transaction id, and a list of changed documents since `old_generation`, represented by tuples with for each document its `doc_id` and the generation and transaction id corresponding to the last intervening change and sorted by generation (old changes first)

`class u1db.Document(doc_id=None, rev=None, json="{}", has_conflicts=False)`

Container for handling a single document.

Variables

`doc_id` Unique identifier for this document.

`rev` The revision identifier of the document.

`json` The JSON string for this document.

`has_conflicts` Boolean indicating whether this document has conflicts

`content`

Content of the `Document`.

`get_json()`

Get the JSON serialization of this document.

`is_tombstone()`

Return `True` if the document is a tombstone, `False` otherwise.

`make_tombstone()`

Make this document into a tombstone.

`same_content_as(other)`

Return `True` if the documents have the same content, `False` otherwise.

`set_json(json)`

Set the json serialization of this document.

6.2.5 The JavaScript implementation

The u1db JavaScript implementation is meant to be used by web applications from browsers, currently it uses browser local storage as backend. This means that the data stored in the backend can be used and updated while offline, compatible sync to a remote target server is supported as well so to interoperate with other implementations and save the data for use across machines using a remote server.

The implementation can be fetched with `bzr branch lp:~pedronis/u1db/u1db-js` or from Launchpad.

The implementation itself lives in one `u1db.js` file of around 2000 lines. Code to reuse the reference implementation test suite is also present, this is based on a simple bridging approach in which API calls in the tests become operation requests piped over by the bridge to a simple interpretative driver running on the JavaScript side (either Rhino, or inside a browser page). This infrastructure is around a bit more than 1000 lines of Python and JavaScript, compared to the 8000 lines and counting in the reference test suite itself.

The API is quite close to the Python's one as it should be. This also allow for the automatic mapping performed by the bridge for testing. Arguments can usually be passed positionally or through a dictionary:

```
db = new U1DB("testing"); // creates a db

doc = db.create_doc({'v': 2}, 'my-doc');
// or equivalently
doc = db.create_doc({'v': 2}, {doc_id: 'my-doc'});

got = db.get_doc('my-other-doc');
```

Syncing to a remote target database can be requested with one line as with the reference implementation:

```
db.sync(target_db_url, {autocreate: true,
                        creds: {"oauth": {
                                "consumer_key": ...,
                                "consumer_secret": ...,
                                "token_key": ...,
                                "token_secret": ...
                        }}});
```

7 Client prototypes

7.1 EyeOS

7.1.1 Introduction

eyeOS is a web platform that provides a remote virtual desktop for the end user. The overall user experience is strongly influenced by the classic desktop design, widely known thanks to the most popular operating system on the market. eyeOS Personal Web Desktop includes several features such as: file manager, contacts, groups and other collaborative capabilities. eyeOS Personal Web Desktop is a disruptive technology that fits in perfectly with the CloudSpaces Open Personal Cloud paradigm. One of the key values that eyeOS provides is the possibility to work directly with files in the cloud. eyeOS does not require users to manually download any files onto their computer nor is it necessary to install anything locally, so the experience is totally transparent: users just log into a website and start working with their files normally.

Furthermore, eyeOS lets you add additional services and applications within the web desktop, so that all the company or organization's web resources are available within a single controlled environment that can be accessed using single sign-on.

By combining eyeOS' web file management capabilities with Personal Cloud, users can access their Personal Cloud contents via web, with a user experience very similar to local desktop environments.

7.1.2 Authentication

The eyeOS platform uses OAuth authentication in order to interact with the user's protected data stored in Personal Cloud. OAuth is an authorization protocol that enables the user (resource owner) to authorize eyeOS to access the resources on their behalf without giving eyeOS their authentication credentials i.e. username and password.

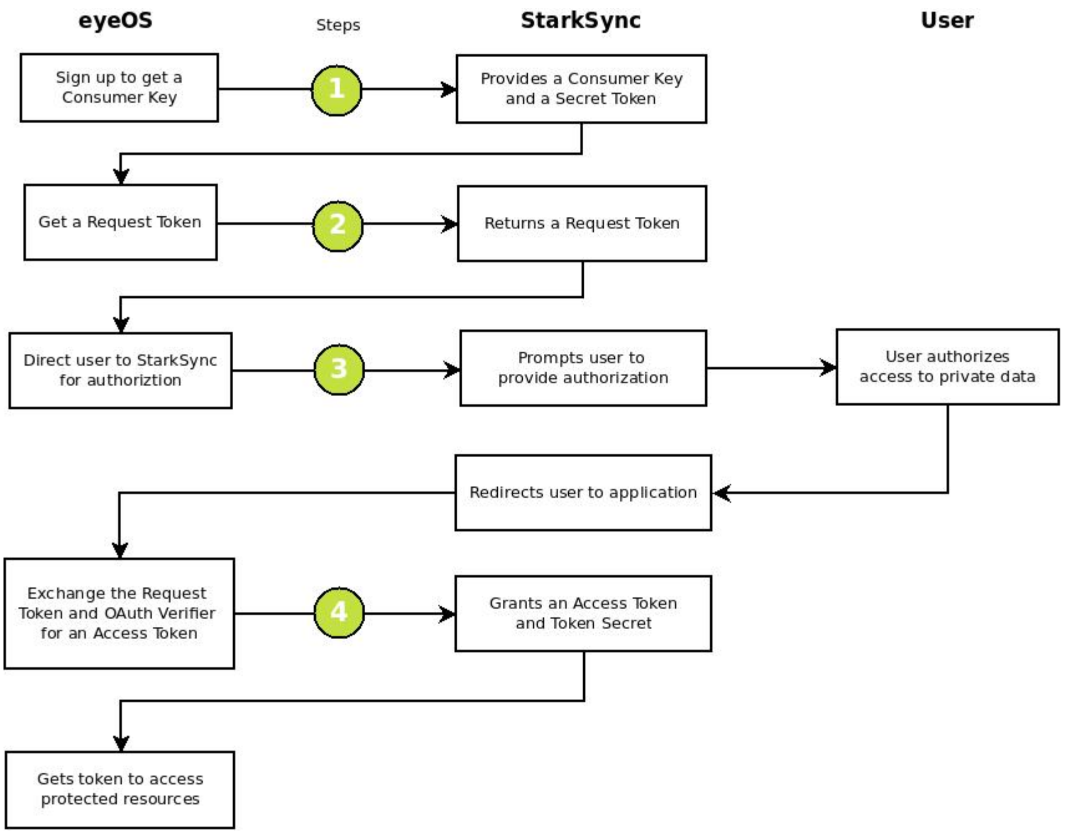

eyeOS
Steps
StarkSync
User
Sign up to get a Consumer Key
1
Provides a Consumer Key and a Secret Token
Get a Request Token
2
Returns a Request Token
Direct user to StarkSync for authoriztion
3
Prompts user to provide authorization
User authorizes access to private data
Redirects user to application
Exchange the Request Token and OAuth Verifier for an Access Token
4
Grants an Access Token and Token Secret
Gets token to access protected resources

Figure 2: eyeOS authentication flow

When the user accesses eyeOS, a newly developed plugin is used to get a security token with which the keys required for interacting with user data stored in Personal Cloud can be obtained. The Access Token and Token Secret keys are stored in a relational database (managed via a RDBMS based on MySQL). These keys are linked with the user who logged onto the platform, meaning the system can determine at any stage the access token for a specific user who attempts to use the service.

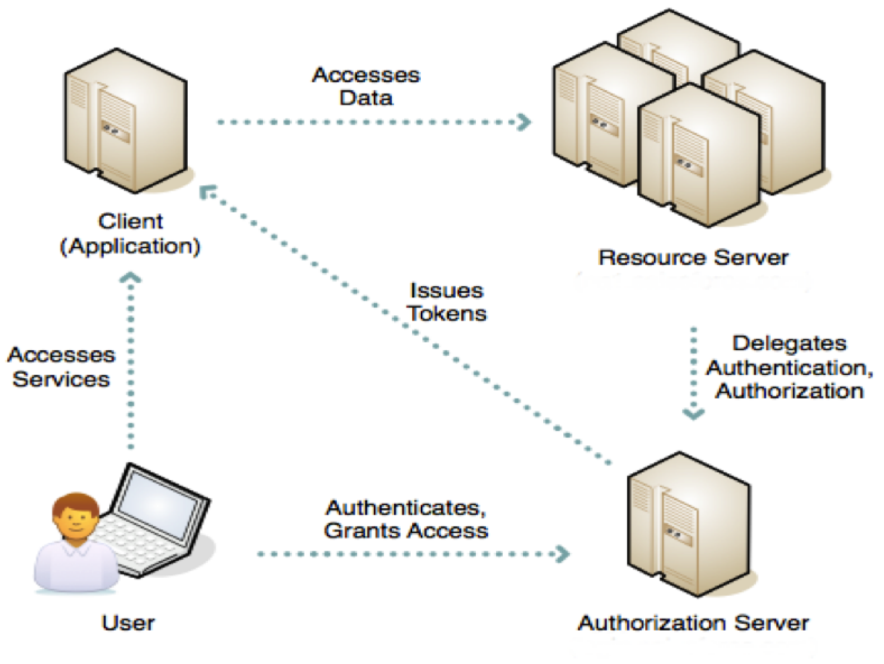
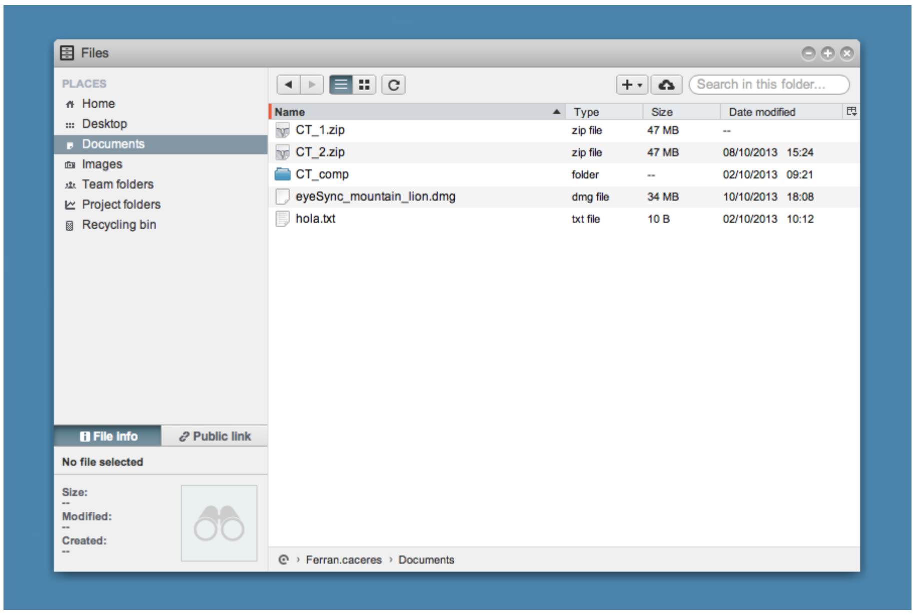


Figure 3: eyeOS file management

7.1.3 Integrated file management

Within the eyeOS Personal Web Desktop, one of the key features of the platform is file management. eyeOS includes a web file manager developed in JavaScript, HTML and CSS that enables users to manage all their files directly from their browser, but with an experience similar to a file manager of any desktop operating system, such as Microsoft Windows™ or GNU/Linux.



Name	Type	Size	Date modified
CT_1.zip	zip file	47 MB	--
CT_2.zip	zip file	47 MB	08/10/2013 15:24
CT_comp	folder	--	02/10/2013 09:21
eyeSync_mountain_lion.dmg	dmg file	34 MB	10/10/2013 18:08
hola.txt	txt file	10 B	02/10/2013 10:12

No file selected

Size:
--
Modified:
--
Created:
--

Ferran.caceres › Documents

Figure 4: eyeOS file management

By integrating Personal Cloud services in the eyeOS platform, users can use eyeOS' web file manager and all its features with their Personal Cloud files. For example, users can display online their MS Office or OpenOffice documents saved in their Personal Cloud, create directories, move files, view PDFs, share documents, etc.

7.1.4 Storage Provider

The storage provider eyeOS uses to store user files can point to a NFS, CIFS or arbitrary mount point; this way you can indicate which storage provider to use for managing files instead of the local disk when installing eyeOS.

eyeOS as a platform is not limited to the web environment. Its web file manager is only one of the many services the platform includes. eyeOS has CIFS endpoints so users can access their files in a way similar to a Windows shared drive, for example.

Given the wide range of services eyeOS provides for user files, the disk access layer is not implemented at web app level, as then the files would not be visible from the CIFS or any other external service.

eyeOS' web app delegates resource access to the system. Hence the operating system is responsible for resolving this access, not the web app.

To do this, eyeOS has a FUSE module that lets it create mount points in the system (virtual disk drives) that map disk access to the appropriate provider. There is only one provider for the web app: FUSE.

To integrate the Personal Cloud storage service in the solution, an event notification service is created which will interact with Personal Cloud's API, updating the file manager and notifying the user with a message.

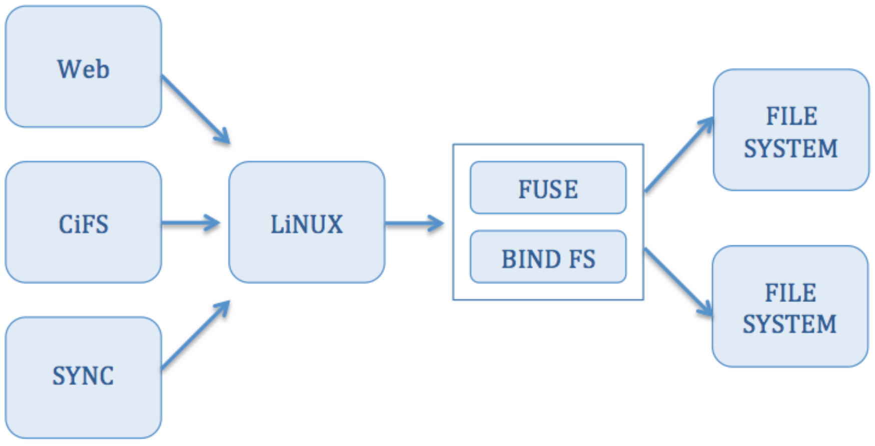


Figure 5: eyeOS storage flow

This service will extend **StoreStackSync**, which will enable requests to be sent and received from Personal Cloud's Store API:

```
@Interface StoreStackSync
interface StoreStackSync {
   @ Metadata request, returns a ResponseStackSync class
   public function retrieveMetadata(int $file_id = 0, bool $list = true,
   bool $include_deleted = false, int $version = 0);
   @ Versions request, returns a ResponseStackSync class
   public function getVersions(int $file_id, int $limit_versions);
   @ Restore files request, returns a ResponseStackSync class
   public function restoreFile(int $file_id, int $version);
   @ Upload files request, returns a ResponseStackSync class
   public function uploadFile(string $file_name, int $parent=0, bool $overwrite);
   @ Download file content request, returns a
   ResponseStackSync class
   public function downloadFile(int $file_id, int $version);
   @ Delete files or folders request, returns a ResponseStackSync class
   public function deleteElement(int $file_id);
   @ Create files or folders request, returns a ResponseStackSync class
   public function createElment(string $name, bool $file = true, int $parent = 0);
}
```

```
@class ResponseStoreStackSync
class ResponseStoreStackSync {
   @ var error
   private $error;
   @ var json
   private $json;
   @ Set variable value error
   public function setError($error);
   @ Get variable value error
   public function getError();
   @ Set variable value json
   public function setJson($json);
   @ Get variable value json
   public function getJson();
}
```

Through the Persistence API, the files in use by the user and that cannot be changed are controlled, although the service gets any differences from the Store API. In these cases, the user is informed so a decision can be made to resolve any conflicts that have occurred.

The following diagram shows the process in detecting a new file in the user's Personal Cloud:

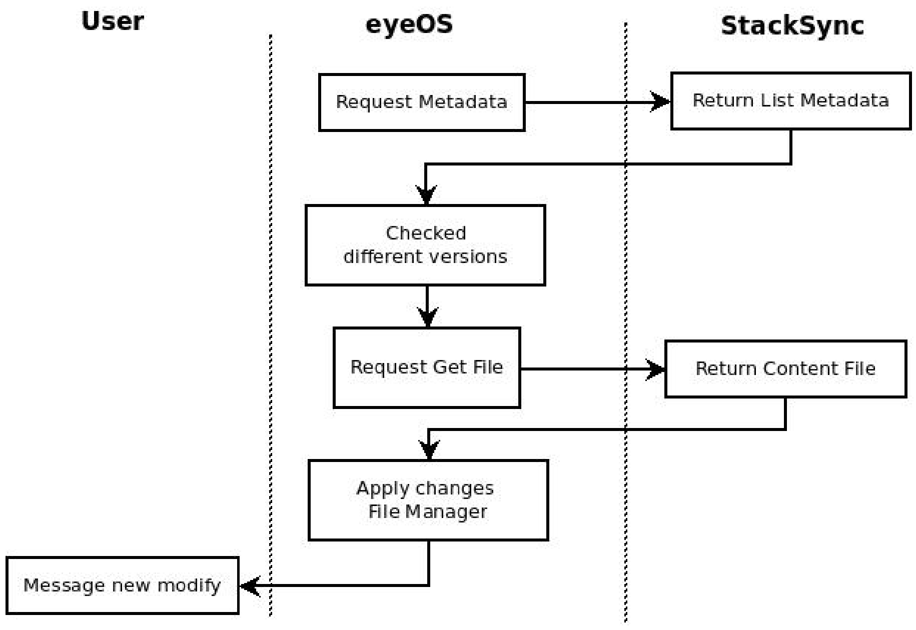


Figure 6: eyeOS file manager flow

All applications/services that the eyeOS platform provides, including the web app that has a web file manager, are limited to requesting files from the operating system.

The operating system (GNU/Linux) delegates FUSE to resolve this request, which in turn calls on Personal Cloud's Rest API to resolve the request and get the results.

However, this is not enough for all functionalities, as eyeOS' web file manager also needs to be able to perform file-sharing operations on Personal Cloud resources. The operations described are beyond the scope of the typical operations of a file provider and cannot be completed by FUSE.

The additional functionalities that do not form part of the interface of a file storage provider are covered directly by the web app itself. By using the files through CIFS or other services, files can be simply viewed and worked on as usual. By accessing through the eyeOS file manager, additional operations such as share file can be performed.

These operations are performed by the web app directly against Personal Cloud's API. As a result, the final scheme is as follows:

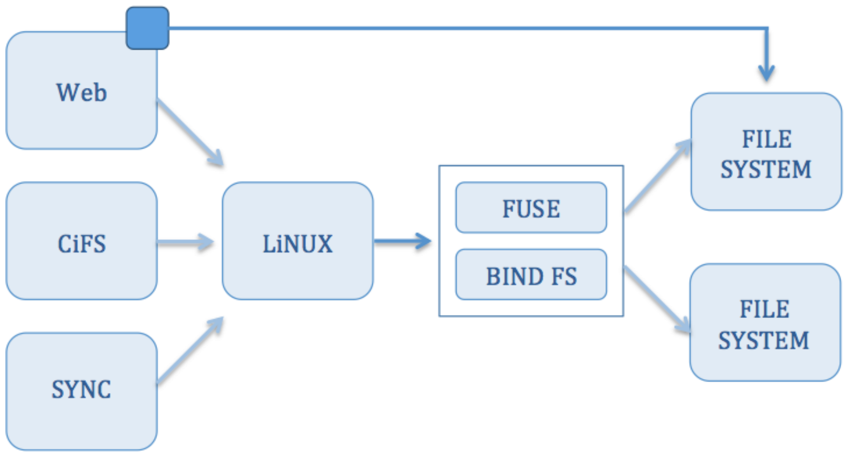


Figure 7: eyeOS direct access to file system

As can be seen above, the basic operations of the file system are performed transparently by the system through FUSE. Additional aggregated functions are performed by the web app by directly accessing the Personal Cloud API.

Both the FUSE module and the web platform have access tokens to the Personal Cloud services on behalf of the user, which have been obtained during authentication.

The final operation is as follows:

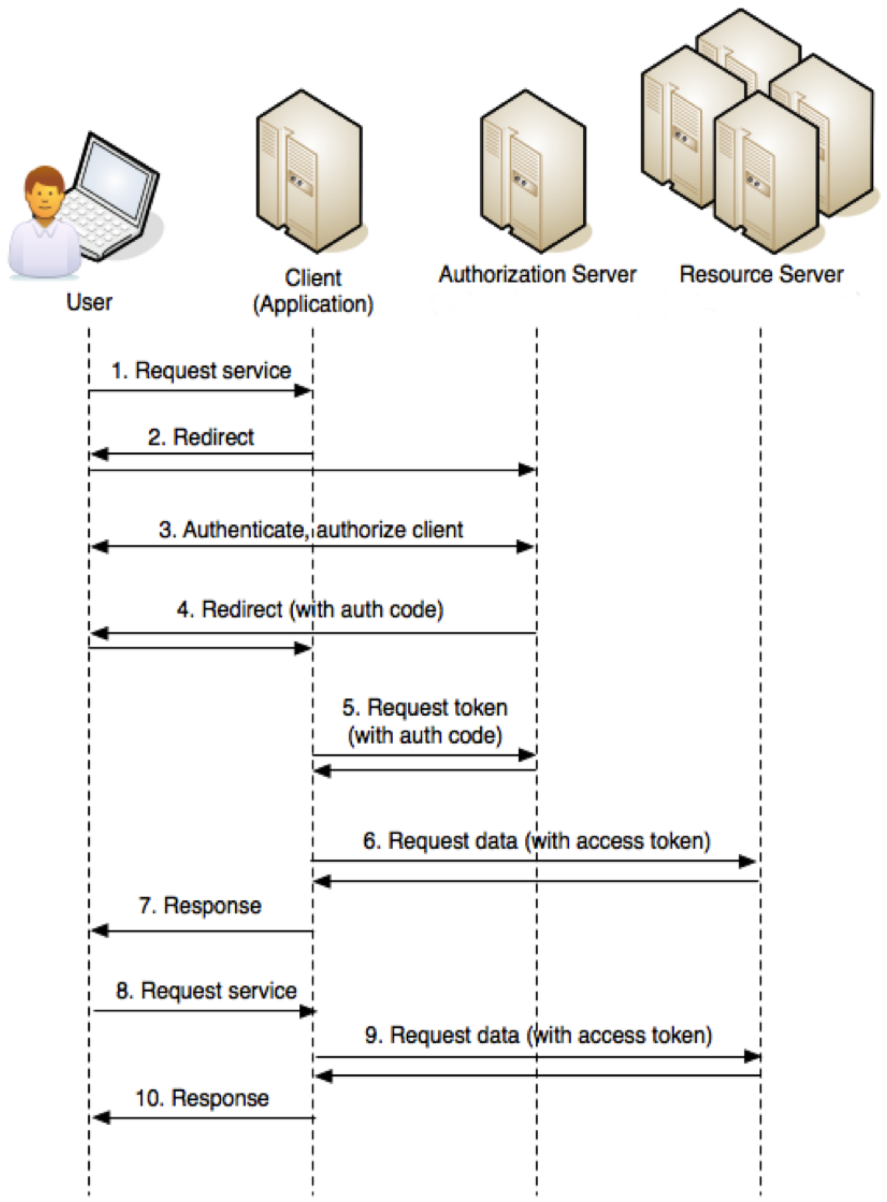

User
Client (Application)
Authorization Server
Resource Server
1. Request service
2. Redirect
3. Authenticate, authorize client
4. Redirect (with auth code)
5. Request token (with auth code)
6. Request data (with access token)
7. Response
8. Request service
9. Request data (with access token)
10. Response

Figure 8: eyeOS final sequence diagram

7.1.5 Token management

To operate, the system must ensure that it has the appropriate user access tokens for Personal Cloud. If these tokens are not available, then the services cannot be accessed via the Personal Cloud API and the user cannot access their files via the web manager or any other way in eyeOS.

The access tokens have a specific timespan and a refresh token is obtained that can be used to get a new token when the access token has expired. The system can continue to request tokens while the user has specified eyeOS as authorized application within Personal Cloud's OAuth.

A daemon has been developed that periodically checks in the DBMS which tokens are about to expire and uses the refresh token to get new tokens with a renewed timespan. Both FUSE and the web app operate under the premise that there are valid tokens at all times.

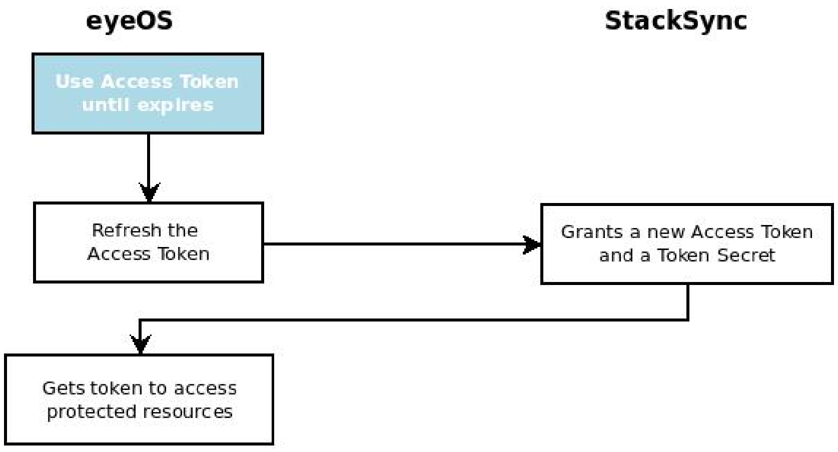

eyeOS
StackSync
Use Access Token until expires
Refresh the Access Token
Grants a new Access Token and a Token Secret
Gets token to access protected resources

Figure 9: eyeOS token renewal flow

If the token stops operating for some reason, the system warns the user that it has lost access to Personal Cloud and closes the session. On closing the session, the user returns to the authentication page where they can repeat the login process against Personal Cloud's OAuth, enabling the system to get new tokens and continue operating with the service as normal.

Token management is crucial in ensuring service continuity; hence, the token management daemon (tokensd) is monitored and self-healing functionalities have been added via monitd.

7.1.6 Scalability

eyeOS architecture is based on stateless web front-ends that can be replicated horizontally without limitation. A traditional load balancer is used in front-ends to distribute requests. Since the front-ends are stateless, sticky sessions are not necessary. The only state is in the database and the file system. Since the file system is delegated to Personal Cloud, the state that remains is in the database. eyeOS uses a MySQL cluster based on Percona to horizontally scale the DBMS. The platform's messaging systems use their own daemon, webqueued, which has a shared state and can be horizontally scaled without any problems. Access to Personal Cloud is done by each front-end node using the tokens taken from the DBMS (managed by tokensd). Requests to Personal Cloud are done asynchronously and are non-blocking, meaning FUSE continues to serve requests while it waits for responses to previous requests.

In our tests, after having integrated Personal Cloud as an authentication and storage provider, the architecture of the solution has not changed essentially and it continues to allow horizontal scalability, as described in the standard eyeOS scalability documents.

However, the responsiveness of the file system has worsened significantly. By using a Rest API in an external storage provider, the disk requests are slower and the file manager responds with latency to certain operations. There is the overall perception that the experience has worsened, although it still remains within acceptable limits and is comparable to the experience achieved in integration with other similar storage systems.

7.2 Web Management Interface

7.2.1 Web client

The web client seeks to create a web interface able to manage different users and their data storage environment.

The system is being designed considering aspects of i18n (internationalization).

The framework developed is the following process flow:

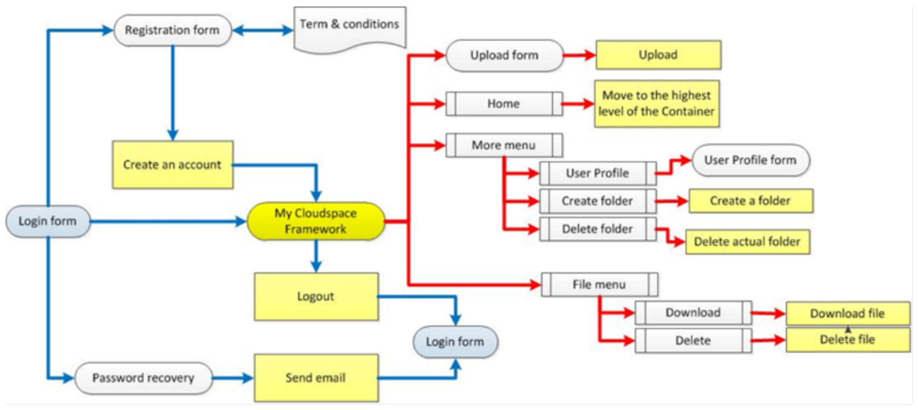


Figure 10: Framework flow

As indicated in the flow chart, we have incorporated features that allow self-registration of users and password management, new options to upload, download and delete files from the user's environment and functionality to create and delete folders.

Below we show the screens developed.

Login Form

This form includes two text input areas, allowing the user to be logged in at the application with its email and password.

The user is the same for the web application than the Openstack user; this coincidence forces us to encrypt personal data for avoiding security vulnerabilities, and also avoids to have two different passwords (one for web interface, another for openstack platform).

The Login Screen links to:

1. Create an account. if you are a new user, you can register to the platform.

2. Forgot my password. Send an email to a registered user, if the password is forgotten.

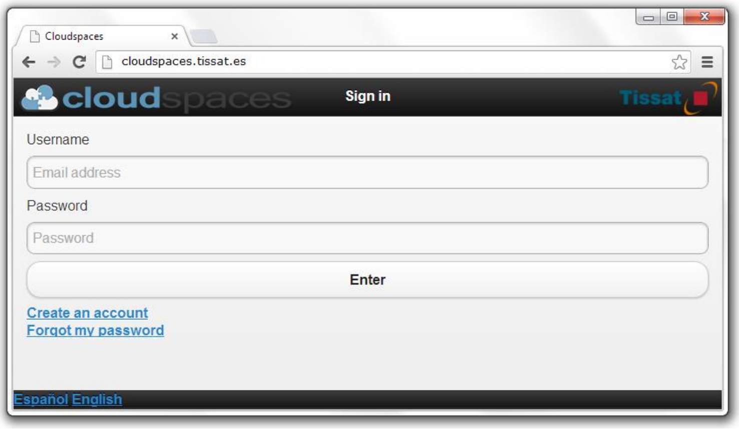


Figure 11: Login form

Registration Form

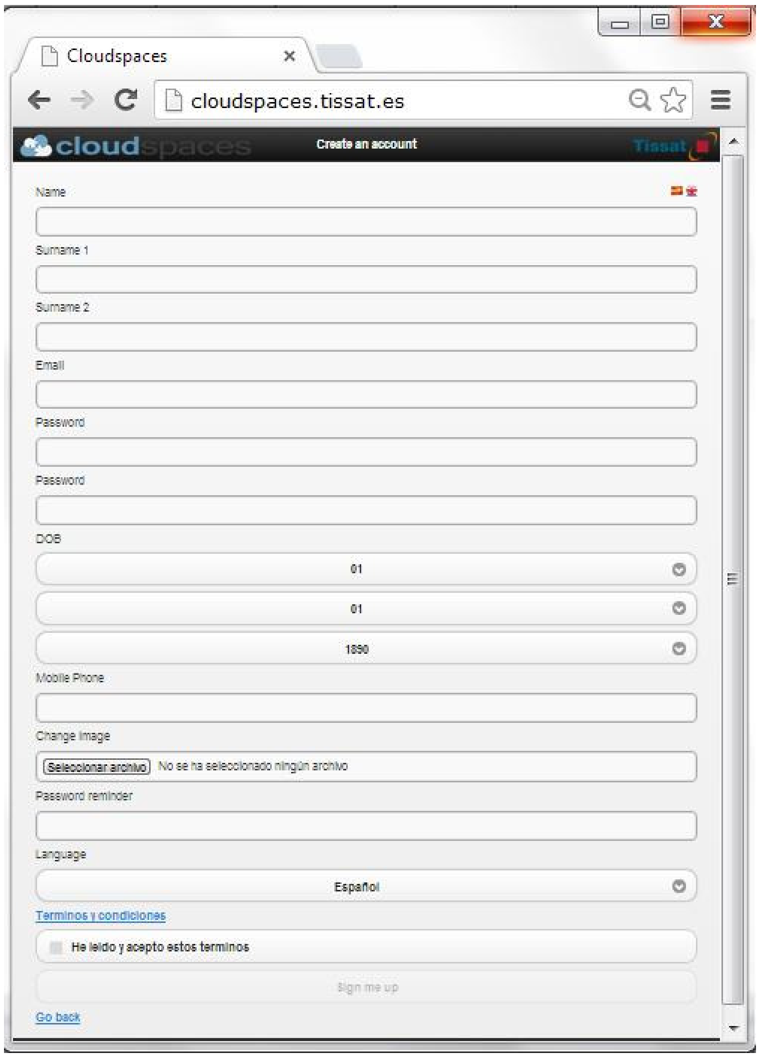
The registration form allows to new users, register at the Openstack platform, and have access their own disk space.

Several fields should be filled by the user, like name, birth date, user icon, and of course email and password fields.

The user must read and accept terms and conditions in order to complete successfully the process. This check field gives us the warranty that the user is currently informed about payment and privacy terms, noticing the user that this is a research project, and stability, privacy and protection against data loss are not assured at this platform.

The user language is also asked at this form, in more mature versions, this field will let us to personalize contents and to implement a default visualization for each language.

When the user confirms the register, the system calls Openstack for creating a new user, it's 'tenant' and space, with a pre-assigned default quota.



Create an account

Name

Surname 1

Surname 2

Email

Password

Password

DOB

01

01

1890

Mobile Phone

Change image

Seleccionar archivo No se ha seleccionado ningún archivo

Password reminder

Language

Español

Terminos y condiciones

He leido y acepto estos terminos

Sign me up

Go back

Figure 12: Registration form

Password Recovery

This is a simple form, with an email field.

The user can write its email address, and a email will be sent, with instructions about how to restore the password.

The next version will include a captcha field, to avoid automatic bot actions.

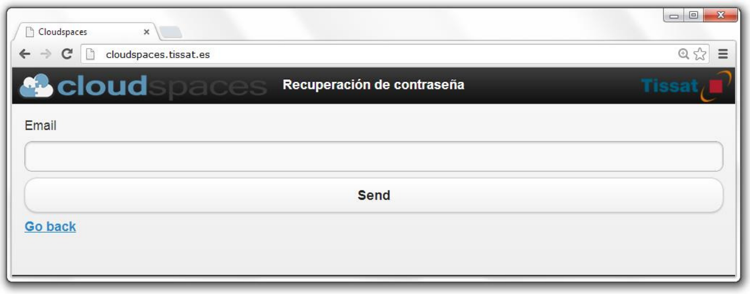


Figure 13: Password recovery form

CloudSpaces Framework

This is the main interface, that allows the user to access to its CloudSpaces Openstack environment upload or download files and work with folders.

There are four main buttons:

- Home. Direct Access to the highest level of the file tree.
- Upload. For uploading files.
- More. Expand the main menu with other options.
- Log out. Close the application, logging out the user.

The user can also enter in a folder when clicking on its name, or visualize to the file options clicking in a existing file.

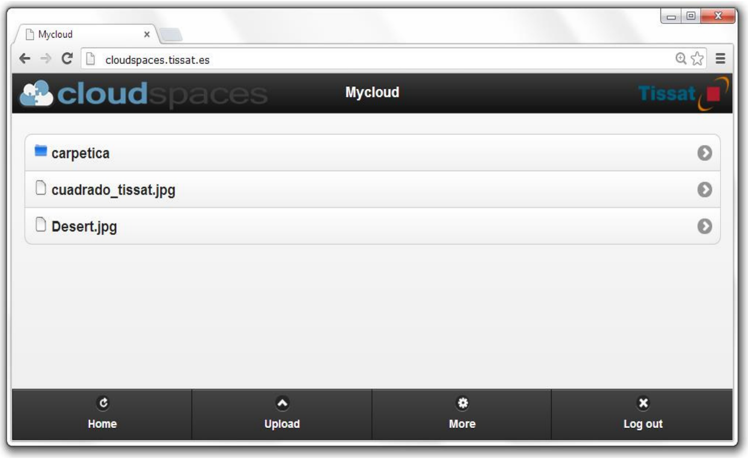


Figure 14: Listing content in the framework

More menu

At this option menu, It will be shown to the user new options:

Profile. The user can access to its own data for updating.

Add folder. It adds a folder in the actual folder.

Delete folder. It deletes the actual folder and its files.

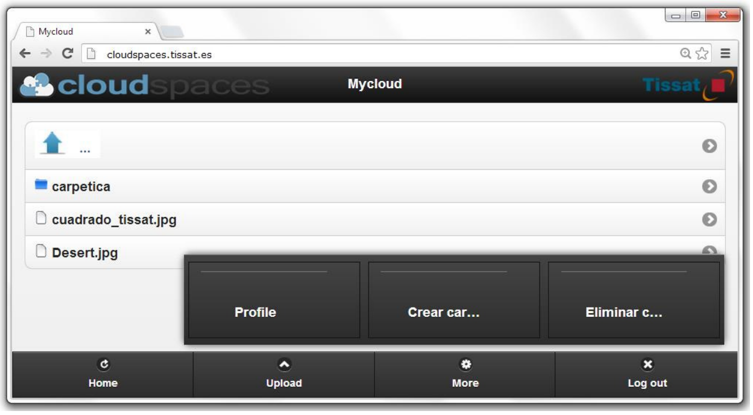


Figure 15: “More” menu

File menu

When clicking in a file, the interface will show options for the selected file. Actual version includes:

Download. Download the file directly.

Delete. Delete the selected file.

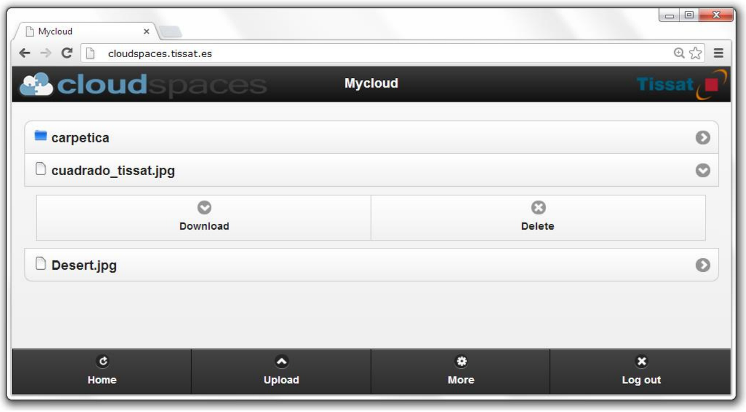


Figure 16: File menu

7.2.2 Admin panel

The developed administration panel aims to be a tool for centralized management of users of different platforms, giving to the system new properties of scalability, ease of use and user control.

Although we have developed a functionality that allows a user to self-register by calling the OpenStack APIs. Tissat has developed an interface that lists active users on the platform, making easiest the management of the platform with thousand of users.

The Administration Panel functionalities are divided into the following aspects:

- Basic Management

Allows the system administrator to register a new user by editing its properties and setting quota limits.

This section allows the obtaining of user lists through their interconnection with the OpenStack API, as well as searching and filtering basic data.

Specifically, the developed areas which will be housed on a secure server, not accessible from any location, includes the functionalities named below:

- Listing the users of the platform.
- Edit User.
- Delete User.
- Add User.
- Definition of quotas to user or groups.

By default, all users are belonging to the same generic group, but the system will provide the possibility of creating different types of groups with their own characteristics.

- Advanced User Management

Provides for the creation of groups of users who share the same container, creates a Group Profile, to allow easy management (add user, delete user ...) and easy monitorization of a specific group of users, including quota definition for them.

The Web Management Interface will get traces about the use of the platform, registering logging data and enabling queries about how is used, controlling bandwidth consumption by registering the size of files uploaded and downloaded per month, including also capabilities for add or cancel monitoring of a user or group, and get listings of use and access to the environment, in order to perform security audits and performance.