Pinyto-Cloud Documentation

Release 1.0-alpha1

Johannes Merkert

August 05, 2016

pinyto-cloud is the software you want to install on your server. There are applications for your clients to connect to the Pinyto cloud which are not covered in this documentation. If you are confused by this structure go to https://pinyto.de and read the introduction there.

Contents:

CHAPTER 1

Introduction

Pinyto is your own private and secure database. You decide which data is used for which application and how the data is processed even before it reaches the device. It achieves that with a document based database (MongoDB) which saves your data in the structure you want it to have. You can access your data through Assemblies which process and prepare your data at the server before it gets transferred to an app on a device. It is up to you which Assemblies you want to install and you can even create your own Assemblies by programming them in Python. This structure lets you expose the minimum of your data to achieve just what you want. Assemblies can also do things on their own like searching for data on the internet to complete your saved datasets computationally on the server. This structure lets your apps feel smart because they can access the information they need while keeping your data in your control. Assemblies are always OpenSource and you can read their sourcecode if you want to know what they are doing.

Pinyto is designed as a framework to make our vision of a personal cloud accessible for you. We provide some webapps and applications for your devices to show how Pinyto is meant to be used and what it is capable of. They may also be useful as they are.

1.1 Structure

There are two main parts of the Pinyto-Cloud component hosted on your server:

1. The Django application talking to your database and executing code from the assemblies.
2. The Webapps which are hosted on the server.

The Django application is structured in six Django-Apps:

1. **pinytoCloud** is the main app which contains the settings.py and the main urls.py. Its models.py contains all the models used for administration of your personal cloud. This data is saved in the SQL-database specified in settings.py. Your user-data does not live in this database. views.py contains the views used for administration of the cloud including registration and authentication.
2. **keyserver** contains models and views which are needed to access the cloud with username and password. Pinyto normally uses public-key-authentication which is not usable for webapps. The keyserver does the public-key-authentication for all users who supply the correct credentials. The models in this app store private keys and password hashes for user accounts.
3. **database** wraps all calls to the document based database used to store the data. It uses pymongo but adds some functionality specific for Pinyto.
4. **service** contains helpers which can be called from assemblies to perform certain tasks. As assemblies are very limited in their ability to process data all the work is done in those services.
5. **api_prototype** contains the views handling all requests concerning api-calls and jobs at assemblies. If necessary a sandbox is initialized and code from the assemblies is executed there.

6. **api** contains trusted assemblies which are executed without a sandbox. This is generally not necessary but can improve the performance of assembly calls and job execution.

The Webapps are structured in folders matching the name of the assembly. For example the files for the "pinyto/Todo" assembly live in /webapps/pinyto/Todo/. Every webapp is a separate application which is bootstrapped with its index.html. At the moment all Webapps are based on Angular.js and are structured like typical Angular applications.

CHAPTER 2

Administration

Pinyto uses two separate databases to store data. The SQL database is used for user accounts, public-keys, code from assemblies and so on. The real data user store inside of Pinyto is stored in a document-based database described in database. This section is about the database models and view functions used for registration, authentication and administration.

In Pinyto all api calls accept only json data for parameters and payload data. Every request except registration and authentication must contain a "token" which identifies the current user and her permissions. An example would be `{"token": "12323A38B3", "name": "test", "author": "mustermann"}`.

2.1 Users

Users are saved in this class:

As users need to get initialized with empty budgets a function is needed which is hooked into the `post_init` signal from Django:

2.2 Public Keys

Users have one public key for each device they use to connect to the cloud. This class saves the key:

2.3 Sessions

If a user authenticates at the cloud a session is created. Each session has a random token which has to be present at all requests a user makes.

2.4 The user's code

Users can write their own assemblies and they are stored in the following objects. The Assembly class stores the basic information about the assembly while ApiFunction an Job store the code.

2.4.1 Assembly

2.4.2 ApiFunction

2.4.3 Job

2.5 Registration

Registering new accounts is done in the register function in views.py.

Although the register function does the real work it does not accept a Django request object as parameter and is therefore not fit to be called by the url dispatcher. For this task the register_request function exists which accepts a request object and can easily be referenced in the url configuration. It internally calls register after extracting relevant the request data.

Registration requests must supply json-encoded data with a "username" and "key_data" while the key_data itself consists of a "N" and "e" value. Supply the numbers as strings.

Example: `{"username": "MaxMustermann", "key_data": {"N": "123456789123456789213456", "e": "54263"}}` For a real key N must be a much bigger number.

2.6 Authentication

Similar to the registration the authentication also consists of two functions. The real work is done in `authenticate`:

The matching function which accepts requests and which is wired into the url config is:

The authentication request needs a "username" and a "key_hash" which identifies the public key used for authentication. The key_hash consists of the first 10 bytes of a sha256 hash of (N+e).

Authenticate returns a challenge for this request containing an encrypted token and a signature of this token signed with the key of the server. The client can check the signature to verify the identity of the server. The client also decrypts the token with the private key matching the public key which was used for the hash in the request. By encrypting the token the server makes sure that only the client which possesses the private key can decrypt the token and use it for authentication.

Authenticate starts a session with a token which is transmitted with every request and which is used to identify the client and ensure that no attacker can access api functions without a correct token. If an attacker reads the token he can make requests as he likes as long as the session is active. Because of that the whole connection must be secured with https and the client must make sure the token is not accessed by malware.

2.7 Logout

`logout` only needs a "token".

2.8 Key Management

There are four functions for the key management. `list_keys` lists all public keys of the user:

Each key listed by this function should match a device used for Pinyto. The first key is usually the one used by webapps like the backoffice.

If a key is to be added `register_new_key` is called:

The function accepts "key_data" with a "N" and "e" encoded as strings.

Keys can also be deleted:

`delete_key` expects a "key_hash" with the first 10 bytes of a sha256 hash of (str(N)+str(e)).

Keys can be deactivated and deactivated again. For this functionality there is only one function which accepts the state:

The function needs a "key_hash" which identifies the key and an "active_state" as a boolean.

2.9 Assembly Management

Assemblies in Pinyto belong to a user but any user can install them. So it is important to distinguish between the assemblies owned by the user which she can edit and the ones owned by other users which can only be installed and deinstalled.

2.9.1 save_assembly

If a user wants to save a new assembly she can use this function:

`save_assembly` accepts all the data defining an assembly in one big json datastructure:

- "original_name" is the name the assembly had. If the original_name does not specify an assembly of the user a new assembly is created using the data from "data".
- **"data" contains the data of the assembly as it should be after it is saved to the database.**
    - "name" is the new name of the assembly.
    - "description" is the new description.
    - **"api_functions" is a list of dictionaries containing:**
        - "name" - the name of the function
        - "code" - the code of the function
    - **"jobs" is a list of dictionaries containing:**
        - "name" - the name of the job
        - "code" - the code of the job
        - "schedule" - an integer defining how many minutes the cloud should wait until running the job again. 0 means the job is run only once.

`save_assembly` returns `{"success": true}` if the assembly is saved.

2.9.2 delete_assembly

Call `delete_assembly` with a "name" defining an existing assembly of this user.

`delete_assembly` returns `{"success": true}` if the assembly is deleted.

2.9.3 list_own_assemblies

2.9.4 list_all_assemblies

This function lists all assemblies, the ones of the user and all assemblies which could be installed by the user,

2.9.5 get_assembly_source

In many cases the description of an assembly is not sufficient for the user to decide if she can safely install the assembly. With the following function the complete sourcecode of the assembly can be loaded:

The assembly is specified with an “author” and a “name”. If written as a combination it is `author/name`.

2.9.6 install_assembly

If the user decides that an assembly is useful and not harmful she can install the assembly with `install_assembly`.

The assembly is specified with an “author” and a “name”. If written as a combination it is `author/name`.

2.9.7 uninstall_assembly

The assembly is specified with an “author” and a “name”. If written as a combination it is `author/name`.

2.9.8 list_installed_assemblies

This function lists only the installed assemblies of the user.

2.10 Urls

Below is the main URL configuration in `pinytoCloud/urls.py` which also includes the urls from keyserver and the api calls handeled in api_prototype.

`pinytoCloud.urls.`**`urlpatterns`**

```
# coding=utf-8
"""
Pinyto cloud - A secure cloud database for your personal data
Copyright (C) 2105 Johannes Merkert <jonny@pinyto.de>

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.
"""
```

```
from django.conf.urls import include, url
from pinytoCloud.views import authenticate_request, logout, list_keys, set_key_active, delete_key, re
from pinytoCloud.views import register_request, list_own_assemblies, save_assembly, delete_assembly
from pinytoCloud.views import list_installed_assemblies, list_all_assemblies, install_assembly, unins
from pinytoCloud.views import get_assembly_source, home
from database.views import statistics, store
from api_prototype.views import api_call

urlpatterns = [
    url(r'^authenticate$', authenticate_request, name='authenticate'),
    url(r'^logout$', logout, name='logout'),
    url(r'^list_keys$', list_keys, name='list_keys'),
    url(r'^set_key_active$', set_key_active, name='set_key_active'),
    url(r'^delete_key$', delete_key, name='delete_key'),
    url(r'^register_new_key$', register_new_key, name='register_new_key'),
    url(r'^register$', register_request, name='register'),
    url(r'^list_own_assemblies$', list_own_assemblies, name='list_own_assemblies'),
    url(r'^save_assembly$', save_assembly, name='save_assembly'),
    url(r'^delete_assembly$', delete_assembly, name='delete_assembly'),
    url(r'^list_installed_assemblies$', list_installed_assemblies, name='list_installed_assemblies'),
    url(r'^list_all_assemblies$', list_all_assemblies, name='list_all_assemblies'),
    url(r'^install_assembly$', install_assembly, name='install_assembly'),
    url(r'^uninstall_assembly$', uninstall_assembly, name='uninstall_assembly'),
    url(r'^get_assembly_source$', get_assembly_source, name='get_assembly_source'),
]

urlpatterns += [
    url(r'^keyserver/', include('keyserver.urls')),
]

urlpatterns += [
    url(r'^(?P<user_name>\w+)/(?P<assembly_name>\w+)/store$', store, name='store'),
    url(r'^statistics$', statistics, name='statistics'),
]

urlpatterns += [
    url(r'^(?P<user_name>\w+)/(?P<assembly_name>\w+)/(?P<function_name>\w+)$', api_call, name='api_ca
]

urlpatterns += [
    url(r'^.*', home, name='home'),
]
```

CHAPTER 3

Keyserver

The Keyserver is integrated into Pinyto to support webapps. Normally Pinyto uses public-key authentication which is more secure than username and password. However this method needs clients which create private and public key pairs and store the private key securely. For webapps this approach is simply not possible. To solve this the Keyserver stands in between webapps and the cloud and saves one private key for each username and password. If a webapp wants to authenticate it can send the users credentials to the keyserver which does the authentication with the stored key if the credentials are correct. The keyserver sends the decrypted token which is ready to use over an https connection to the webapp. The webapp can use this token for all requests in this session.

3.1 Administration-API

Similar to the administration of the cloud does the keyserver provide an API to administer the Accounts.

`register` function expects a "username" and a "password" in the request data.

`authenticate` function expects a "username" and a "password" in the request data.

`change_password` function expects the new password as "password" in the request data.

3.1.1 Urls

`keyserver.urls.`**`urlpatterns`**

```
# coding=utf-8
"""
Pinyto cloud - A secure cloud database for your personal data
Copyright (C) 2105 Johannes Merkert <jonny@pinyto.de>

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.
"""
```

```
from django.conf.urls import url
from keyserver.views import authenticate, register, change_password

urlpatterns = [
    url(r'^authenticate$', authenticate, name='keyserver_authenticate'),
    url(r'^register$', register, name='keyserver_register'),
    url(r'^change_password$', change_password, name='change_password'),
]
```

CHAPTER 4

Database

4.1 Wrapper-Service

Pinyto internally uses pymongo to access the MongoDB database on the server. Because of the restrictive sandbox architecture a wrapper service for the database interface is used which basically exposes the most used functionality of pymongo to the assemblies. For security reasons no direct access to pymongo could be allowed because Pinyto must make sure the data of other users stays untouched.

class `service.database.`**`CollectionWrapper`**(*collection*, *assembly_name*, *only_own_data=True*)

This wrapper is user to expose the db to the users assemblies.

`count`(*query*)

Use this function to get a count from the database.

Parameters **`query`** (`dict`) –

Returns The number of documents matching the query

Return type int

`find`(*query*, *skip=0*, *limit=0*, *sorting=None*, *sort_direction='asc'*)

Use this function to read from the database. This method encodes all fields beginning with _ for returning a valid json response.

Parameters

- **`query`** (`dict`) –
- **`skip`** (`int`) – Count of documents which should be skipped in the query. This is useful for pagination.
- **`limit`** (`int`) – Number of documents which should be returned. This number is of course the maximum.
- **`sorting`** (`str`) – String identifying the key which is used for sorting.
- **`sort_direction`** (`str`) – 'asc' or 'desc'

Returns The list of found documents. If no document is found the list is empty.

Return type list

`find_distinct`(*query*, *attribute*)

Return a list representing the diversity of a given attribute in the documents matched by the query.

Parameters

- **`query`** (`str`) – json

- **`attribute`** (*`str`*) – String describing the attribute

Returns A list of values the attribute can have in the set of documents described by the query

Return type list

`find_document_for_id`(*document_id*)
Find the document with the given ID in the database. On success this returns a single document.

Parameters **`document_id`** (*`string`*) –

Returns The document with the given _id

Return type dict

`find_documents`(*query*, *skip=0*, *limit=0*, *sorting=None*, *sort_direction='asc'*)
Use this function to read from the database. This method returns complete documents with _id fields. Do not use this to construct json responses!

Parameters

- **`query`** (*`dict`*) –
- **`skip`** (*`int`*) – Count of documents which should be skipped in the query. This is useful for pagination.
- **`limit`** (*`int`*) – Number of documents which should be returned. This number is of course the maximum.
- **`sorting`** (*`str`*) – String identifying the key which is used for sorting.
- **`sort_direction`** (*`str`*) – 'asc' or 'desc'

Returns The list of found documents. If no document is found the list is empty.

Return type list

`insert`(*document*)
Inserts a document. If the given document has a ID the ID is removed and a new ID will be generated. Time will be set to now.

Parameters **`document`** (*`dict`*) –

Returns The ObjectId of the insrted document

Return type str

`remove`(*document*)
Deletes the document. The document must have a valid _id

Parameters **`document`** (*`dict`*) –

`save`(*document*)
Saves the document. The document must have a valid _id

Parameters **`document`** (*`dict`*) –

Returns The ObjectId of the insrted document

Return type str

For coders the helpers used in this service may be of interest:

`service.database.`**`encode_underscore_fields`**(*data*)
Removes _id

Parameters **`data`** (*`dict`*) –

Return type dict

`service.database.`**`encode_underscore_fields_list`**(*data_list*)

Removes _id for every dict in the list

Parameters **`data_list`** (*`list`*) –

Return type list

`service.database.`**`inject_object_id`**(*query*)

Traverses all fields of the query dict and converts all '_id' to ObjectId instances.

Parameters **`query`** (*`dict`*) –

Return type dict

4.2 Default-API

Some API functions are used in nearly every assembly. To prevent users from writing the same code over and over some default functions for assemblies are implemented and can be called by every assembly.

Warning: Default API-functions hide explicitly defined functions in the assembly with the same name at the moment. This may change in future versions where assemblies can overwrite default functionality.

4.3 Database statistics

The database app also exposes an API-function for loading database statistics of the user. The statistics are:

- 'time_budget': Sum of all the CPU time (in seconds) used by the user.
- 'storage_budget': Integral over the storage the user user over time. The value is in bytes*seconds.
- 'current_storage': The amount of of storage (in bytes) the user uses at the moment.
- 'last_calculation': Timestamp of the last time the budgets were calculated. This is needed if the frontend tries to calculate the storage budget up to the current time.
- 'assembly_count': Number of assemblies the user owns.
- 'installed_assemblies_count': Number of assemblies the user has installed.
- 'all_assemblies_count': Number of assemblies available for the user.

Assemblies

Assemblies are the key element of Pinyto as they empower the user to do with her data as she likes. Users can write their own assemblies but they also can install assemblies from other users. If those other users update their assembly the user automatically uses the new version without having to change anything.

Warning: It is crucial that users can trust the authors of the assemblies they have installed. They can check the sourcecode of the all assemblies as they install them but if an update changes the assembly the user gets no notification and the changed assembly may leak or delete data.
Users can prevent that by forking assemblies of other users which makes the assembly in the state as it is one of their own. By doing that their assembly does not change if the original author changes her assembly and the user can be sure that no harmful code is executed. This method has the downside that updates with bugfixes or new features do not get installed automatically. The user may delete the forked assembly in case of a good update and fork the original again. The usability of this procedure could be improved in future versions.

5.1 Installing assemblies

If you run your own Pinyto server there are two ways to install new assemblies. The first possibility is to store them in your database. You may want to create user accounts to have telling names for the assemblies. For example for the assembly `pinyto/Todo` a user with the name “pinyto” must be present.

If you check out a new version of Pinyto there might be new default assemblies in the `api` module. With this version come three bundled assemblies:

- **pinyto/DocumentsAdmin** is used for the backoffice to let you browse all your documents there.
- **pinyto/Todo** is the assembly for the example app for Pinyto which is a simple TODO-list.
- **bborsalino/Librarian** is an assembly for a app used to manage the books in your flat. This assembly uses a job which completes incomplete data for books by asking a publicly available database. This could be a good example for your next assembly because the TODO-app does not need a job.

The assemblies in the `api` module provide data migrations which insert the user and the assembly itself into the database.

Normally the code of every assembly gets executed in a seccomp secured sandbox. Using the sandbox might decrease the performance of the assembly execution. So Pinyto gives administrators the opportunity to install assemblies as Django apps inside the `api` module. If an assembly is called and a directly installed assembly in the `api` module is found the code from there gets executed without a sandbox. In order to have this working an assembly with the same name has to exist in the database.

Warning: A bad admin could trick users in thinking that an assembly is not harmful by having different code saved in the database than in the Django app in `api`. We considered this a minor threat because if your admin wants to harm you she could do all sorts of bad things to your data. It is certainly best to have an admin you can trust. If you do not trust your admin become your own admin on your own server.
If you write an assembly inside of `api` make sure to insert the correct code of the assembly in the data migration.

For users there is no visible difference between assemblies executed in the sandbox and assemblies executed directly. If you have benchmarks showing how big the difference is please share them.

5.2 Calling Assemblies

Assemblies are called with the `api_call` view in the `api_prototype` module.

`api_call` checks in the `api` module if there is a directly executable version of the assembly. If there is none `load_api` is called.

5.3 Executing Jobs

API calls can save documents of different type. The “type”: “job” is special as it is the type of a document describing a scheduled job. A job scheduled this way gets executed immediately after the request saving the document is finished. The scheduling document must have the following structure:

- “type”: “job”
- “data”: A dictionary containing the following attributes and data:
  - “assembly_user”: The username of the author of the assembly.
  - “assembly_name”: The name of the assembly.
  - “job_name”: The name of the job.

After each finished request Django calls `check_for_jobs`.

5.4 The Sandbox

The Pinyto sandbox is used if `safely_exec` is called.

`safely_exec` starts a process and executes `sandbox` there.

The process executing `sandbox` creates a new instance of `SecureHost`. The initialization of this class forks the sandbox process into two parts:

1. The host process which has access to the database the request and the services.
2. The child which has only a pipe to communicate to the host process. All open file descriptors in the child process get closed and the database, request and service objects get replaced by sandbox versions. Those sandbox versions of the services have the same signatures but communicate only to the host process which asks the real services for answers which get returned into the child process. This is done because the sandbox is secured using *seccomp* and seccomp only allows reading an writing to already open file descriptors. The access to any other function of the kernel is blocked by the kernel. If the exec call in the child process tries to do anything other than calculations with the data in its memory or communication to the services over the pipe to the host process it will get terminated by the kernel.

There are some helper functions and classes which may be relevant for understanding how the sandbox works:

`api_prototype.sandbox_helpers.`**`libc_exit`**(*n=1*)
Invoke _exit(2) system call.

Parameters **`n`** (`int`) –

`api_prototype.sandbox_helpers.`**`read_exact`**(*fp*, *n*)
Read only the specified number of bytes

Parameters

- **`fp`** (`file`) – file pointer
- **`n`** (`int`) – number of bytes to read

Return type bytes

`api_prototype.sandbox_helpers.`**`write_exact`**(*fp*, *s*)
Write only the specified number of bytes

Parameters

- **`fp`** (`file`) – file pointer
- **`s`** (`bytes`) – string to write and not a byte more than that

`api_prototype.sandbox_helpers.`**`write_to_pipe`**(*pipe*, *data_dict*)
Writes the data_dict to the give pipe.

Parameters

- **`pipe`** (`socket.Socket`) – one part of socket.socketpair()
- **`data_dict`** (`dict`) –

`api_prototype.sandbox_helpers.`**`read_from_pipe`**(*pipe*)
Reads a json string from the pipe and decodes the json of that string.

Parameters **`pipe`** (`socket.Socket`) – one part of socket.socketpair()

Return type dict

`api_prototype.sandbox_helpers.`**`escape_all_objectids_and_datetime`**(*conv_dict*)
This function escapes all ObjectId objects to make the dict json serializable.

Parameters **`conv_dict`** (`dict`) –

`api_prototype.sandbox_helpers.`**`unescape_all_objectids_and_datetime`**(*conv_dict*)
This function reverses the escape of all ObjectId objects done by escape_all_objectids_and_datetime.

Parameters **`conv_dict`** (`dict`) –

`api_prototype.sandbox_helpers.`**`piped_command`**(*pipe*, *command_dict*)
Writes the command_dict to the pipe end reads the answer.

Parameters

- **`pipe`** (`socket.Socket`) – one part of socket.socketpair()
- **`command_dict`** (`dict`) –

class `api_prototype.sandbox_helpers.`**`NoResponseFromHostException`**
This is a custom exception which gets returned if no valid response is returned.

class `api_prototype.models.`**`SandboxCollectionWrapper`**(*child_pipe*)
This wrapper is user to expose the db to the users assemblies. This is the class with the same methods to be used in the sandbox.

count (*query*)

Use this function to get a count from the database.

Parameters **query** (*dict*) –

Returns The number of documents matching the query

Return type int

find (*query*, *skip=0*, *limit=0*, *sorting=None*, *sort_direction='asc'*)

Use this function to read from the database. This method encodes all fields beginning with _ for returning a valid json response.

Parameters

- **query** (*dict*) –
- **skip** (*int*) – Count of documents which should be skipped in the query. This is useful for pagination.
- **limit** (*int*) – Number of documents which should be returned. This number is of course the maximum.
- **sorting** (*str*) – String identifying the key which is used for sorting.
- **sort_direction** (*str*) – 'asc' or 'desc'

Returns The list of found documents. If no document is found the list is empty.

Return type list

find_distinct (*query*, *attribute*)

Return a list representing the diversity of a given attribute in the documents matched by the query.

Parameters

- **query** (*str*) – json
- **attribute** (*str*) – String describing the attribute

Returns A list of values the attribute can have in the set of documents described by the query

Return type list

find_document_for_id (*document_id*)

Find the document with the given ID in the database. On success this returns a single document.

Parameters **document_id** (*string*) –

Returns The document with the given _id

Return type dict

find_documents (*query*, *skip=0*, *limit=0*, *sorting=None*, *sort_direction='asc'*)

Use this function to read from the database. This method returns complete documents with _id fields. Do not use this to construct json responses!

Parameters

- **query** (*dict*) –
- **skip** (*int*) – Count of documents which should be skipped in the query. This is useful for pagination.
- **limit** (*int*) – Number of documents which should be returned. This number is of course the maximum.
- **sorting** (*str*) – String identifying the key which is used for sorting.

- **sort_direction** (*str*) – 'asc' or 'desc'

Returns The list of found documents. If no document is found the list is empty.

Return type list

insert (*document*)

Inserts a document. If the given document has a ID the ID is removed and a new ID will be generated. Time will be set to now.

Parameters **document** (*dict*) –

Returns The ObjectId of the insrted document

Return type str

remove (*document*)

Deletes the document. The document must have a valid _id

Parameters **document** (*dict*) –

save (*document*)

Saves the document. The document must have a valid _id

Parameters **document** (*dict*) –

Returns The ObjectId of the insrted document

Return type str

class api_prototype.models.**SandboxRequestPost** (*child_pipe*)

This wrapper is used to expose Django's request object to the users assemblies. This class implements the most used methods of the request object.

This class is used to emulate request.POST

get (*param*)

Returns the specified param

Parameters **param** (*str*) –

class api_prototype.models.**SandboxRequest** (*child_pipe*)

This wrapper is user to expose Django's request object to the users assemblies. This class implements the most used methods of the request object.

init_body ()

This needs to be called after the seccomp process is initialized to fill in valid body data for the request.

class api_prototype.models.**CanNotCreateNewInstanceInTheSandbox** (*class_name*)

This Exception is thrown if a script wants to create an object of a class that can not be created in the sandbox.

class api_prototype.models.**Factory** (*pipe_child_end*)

Use this factory to create objects in the sandboxed process. Just pass the class name to the create method.

create (*class_name*, **args*)

This method will create an object of the class of classname with the arguments supplied after that. If the class can not be created in the sandbox it throws an Exception.

Parameters

- **class_name** (*str*) –
- **args** – additional arguments

Returns Objects of the type specified in class_name

Return type Object

class `api_prototype.models.`**`SandboxParseHtml`**(*pipe_child_end*, *html*)

This wrapper is user to expose html parsing functionality to the sandbox. This is the ParseHtml class with the same methods to be used in the sandbox.

`contains`(*descriptions*)

Use this function to check if the html contains the described tag. The descriptions must be a list of python dictionaries with `{'tag': 'tagname', 'attrs': dict}`

Parameters **`descriptions`** (*dict*) –

Return type boolean

`find_element_and_collect_table_like_information`(*descriptions*, *searched_information*)

If you are retrieving data from websites you might need to get the contents of a table or a similar structure. This is the function to get that information. The descriptions must be a list of python dictionaries with `{'tag': 'tag name', 'attrs': dict}`. The last description in this list will be used for a findAll of that element. This should select all the rows of the table you want to read. specify all the information you are searching for in searched_information in the following format: `{'name': {'search tag': 'td', 'search attrs': dict, 'captions': ['list', 'of', 'captions'], 'content tag': 'td', 'content attrs': dict},'next name': ...}`

Parameters

- **`descriptions`** (*dict*) –
- **`searched_information`** (*dict*) –

Return type dict

`find_element_and_get_attribute_value`(*descriptions*, *attribute*)

Use this function to find the described tag and return the value from attribute if the tag is found. Returns empty string if the tag or the attribute is not found. The descriptions must be a list of python dictionaries with `{'tag': 'tag name', 'attrs': dict}`

Parameters

- **`descriptions`** (*dict*) –
- **`attribute`** (*str*) –

Returns string or list if attribute is class

class `api_prototype.models.`**`SandboxHttp`**(*pipe_child_end*)

This wrapper is user to expose http requests to the sandbox. This is the Http class with the same methods to be used in the sandbox.

`get`(*url*)

This issues a http request to the supplied url and returns the response as a string. If the request fails an empty string is returned.

Parameters **`url`** – Url with http:// or https:// at the beginning

Type str

Return type str

`post`(*url*, *data*)

This issues a http request to the supplied url and returns the response as a string. If the request fails an empty string is returned.

Parameters

- **`url`** (*str*) – Url with http:// or https:// at the beginning

- **`data`** (*`dict`*) – payload data

Return type str

class `api_prototype.sandbox_helpers.`**`EmptyRequest`**

This class is used for processing jobs. They need request.body but it can be empty.

CHAPTER 6

Services

Assemblies can only access some services explicitly exposed to them because the sandbox can not allow arbitrary calls to any library. The services integrated in Pinyto also try to be easy and pleasurable to use.

The database wrapper is already explained in the Database section.

6.1 Response Helper

Most requests in pinyto expect a response in the form of a HttpResponse object with type "application/json" and a JSON encoded string as payload. Because this is so common an easy to use helper function is used:

`service.response.`**`json_response`**`(`*data*`)`

Returns the json as string with correct mimetype.

@param data: dict @return: HttpResponse

6.2 Factory

In assemblies it is not allowed to include python modules or classes. Service classes can be instantiated using the `Factory` class which is accessible.

class `service.models.`**`Factory`**

Use this factory to create objects in outside the sandboxed process. Just pass the class name to the create method.

static **`create`**`(`*class_name*, **args*`)`

This method will create an object of the class of classname with the arguments supplied after that. If the class can not be created in the sandbox it throws an Exception. The Exception gets thrown even if this is not executed inside the sandbox because every code should be executable in the sandbox.

Parameters

- **`class_name`** (*`str`*) –
- **`args`** – additional arguments

Returns Objects of the type specified in class_name

Return type Object

The factory can create the following classes:

- Http

- ParseHtml

6.3 Http

`Http` uses internally the `requests` library from Apache.

class `service.http.`**`Http`**

Objects of this class can be used to connect to remote websites.

static **`get`**(*url*)

This issues a http request to the supplied url and returns the response as a string. If the request fails an empty string is returned.

Parameters **`url`** – Url with http:// or https:// at the beginning

Type str

Return type str

static **`post`**(*url='‘*, *data=None*)

This issues a http request to the supplied url and returns the response as a string. If the request fails an empty string is returned.

For the params do not forget to add an @ to the beginning of each param name.

Parameters

- **`url`** (*`str`*) – Url with http:// or https:// at the beginning
- **`data`** (*`dict`*) – payload data

Return type str

6.4 ParseHtml

`ParseHtml` is based on BeautifulSoup version 4. The interface is quite different as the wrapper does not use a fluid interface.

class `service.parsehtml.`**`ParseHtml`**(*html*)

Use this service to get information from html documents.

`contains`(*descriptions*)

Use this function to check if the html contains the described tag. The descriptions must be a list of python dictionaries with `{'tag': 'tagname', 'attrs': dict}`

Parameters **`descriptions`** (*`dict`*) –

Return type boolean

`find_element_and_collect_table_like_information`(*descriptions*, *searched_information*)

If you are retrieving data from websites you might need to get the contents of a table or a similar structure. This is the function to get that information. The descriptions must be a list of python dictionaries with `{'tag': 'tag name', 'attrs': dict}`. The last description in this list will be used for a findAll of that element. This should select all the rows of the table you want to read. specify all the information you are searching for in searched_information in the following format: `{'name': {'search tag': 'td', 'search attrs': dict, 'captions': ['list', 'of', 'captions'], 'content tag': 'td', 'content attrs': dict},'next name': ...}`

Parameters

- **`descriptions`** (*`dict`*) –
- **`searched_information`** (*`dict`*) –

Return type dict

`find_element_and_get_attribute_value`(*descriptions*, *attribute*)

Use this function to find the described tag and return the value from attribute if the tag is found. Returns empty string if the tag or the attribute is not found. The descriptions must be a list of python dictionaries with `{'tag': 'tag name', 'attrs': dict}`

Parameters

- **`descriptions`** (*`dict`*) –
- **`attribute`** (*`str`*) –

Returns string or list if attribute is class

For this class the `extract_content` function from `service.xml` might become handy.

`service.parsehtml.`**`extract_content`**(*tag*)

Takes a tag and returns the string content without markup.

Parameters **`tag`** – BeautifulSoup Tag

Return type str

CHAPTER 7

Indices and tables

- genindex
- modindex
- search