ActingWeb Documentation

Release 2.6.1

Greger Teigre Wedel

Apr 22, 2021

Contents:

This is a python library implementing the REST-based ActingWeb distributed micro-services model. It serves as the reference implementation for the ActingWeb REST protocol specification for how such micro-services interact. In order to follow the specification, an application needs to implement a set of API endpoints. These can be implemented using any REST framework (or just plain http requests). See ActingWeb for example/boilerplate project(s).

CHAPTER 1

License

CHAPTER 2

README - actingweb - an ActingWeb Library

This is a python library implementation showcasing the REST-based ActingWeb distributed micro-services model. A typical use case is bot to bot communication on a peer to peer level. It serves as the reference implementation for the ActingWeb REST protocol specification for how such micro-services interact.

2.1 Repository and documentation

The library is available as a PYPI library and installed with *pip install actingweb*. Project home is at https://pypi.org/project/actingweb/.

The git repository for this library can be found at https://github.com/gregertw/actingweb.

The latest documentation for the released version (release branch) of this library can be found at http://actingweb.readthedocs.io/.

The master branch of the library has the latest features and bug fixes and the updated documentation can be found at http://actingweb.readthedocs.io/en/master.

2.2 Why use actingweb?

ActingWeb is well suited for applications where each individual user's data and functionality both needs high degree of security and privacy AND high degree of interactions with the outside world. Typical use cases are Internet of Things where each user's "Thing" becomes a bot that interacts with the outside world, as well as bot to bot communication where each user can get a dedicated, controllable bot talking to other user's bots.

As a developer, you get a set of out of the box functionality from the ActingWeb library:

- an out-of-the-box REST bot representing each user's thing, service, or functionality (your choice)
- a way to store and expose data over REST in a very granular way using properties
- a trust system that allows creation of relationships the user's bot on the user level
- a subscription system that allows one bot (user) to subscribe to another bot's (user's) changes

- **an oauth framework to tie the bot to any other API service and thus allow user to user communication using** individual user's data from the API service

There is a high degree of configurability in what to expose, and although the ActingWeb specification specifies a protocol set to allow bots from different developers to talk to each other, not all functionality needs to be exposed.'

Each user's indvidual bot is called an `actor` and this actor has its own root URL where its data and services are exposed. See below for further details.

2.3 Features of actingweb library

The latest code in master is at all times deployed to https://actingwebdemo.greger.io/ It has implemented a simple sign-up page as a front-end to a REST-based factory URL that will instantiate a new actor with a guid to identify the actor. The guid is then embedded in the actor's root URL, e.g. `https://actingwebdemo.greger.io/9f1c331a3e3b5cf38d4c3600a2ab5d54`.

If you try to create an actor, you will get to a simple web front-end where you can set the actor's data (properties) and delete the actor. You can later access the actor (both /www and REST) by using the Creator you set as username and the passphrase you get when creating the actor and log in.

acting-web-gae-library is a close to complete implementation of the full ActingWeb specification where all functionality can be accessed through the actor's root URL (e.g. `https://actingwebdemo.greger.io/9f1c331a3e3b5cf38d4c3600a2ab5d54`):

- `/properties`: attributed/value pairs as flat or nested json can be set, accessed, and deleted to store this actor's data
- `/meta`: a publicly available json structure allowing actor's to discover each other's capabilities
- `/trust`: access to requesting, approving, and managing trust relationships with other actors of either the same type or any other actor "talking actingweb"
- `/subscriptions`: once a trust relationship is set up, this path allows access to establishing, retrieving, and managing subscriptions that are based on paths and identified with target, sub-target, and resource, e.g. `/resources/folders/12345`
- `/callbacks`: used for verification when establishing trust/subscriptions, to receive callbacks on subscriptions, as well as a programming hook to process webhooks from 3rd party services
- `/resources`: a skeleton to simplify exposure of any type of resource (where /properties is not suited)
- `/oauth`: used to initiate a www-based oauth flow to tie the actor to a specific OAuth user and service. Available if OAuth is turned on and a 3rd party OAuth service has been configured in config.py. /www will also be redirected to /oauth (*OAuth is not enabled in the online actingwebdemo mini-application*)

Sidenote: The **actingweb library** also implements a simple mechanism for protecting the /www path with oauth (not in the specification). On successful OAuth authorisation, it will set a browser cookie to the oauth token. This is not used in the inline demo and requires also that the identity of the user authorising OAuth access is the same user already tied to the instantiated actor. There is a programming hook that allows such verification as part of the OAuth flow, but it is not enabled in the actingwebdemo mini-application.

2.4 Other applications using the actingweb library

There is also another demo application available for Cisco Webex Teams . It uses the actingweb library to implement a Webex Teams bot and integration. If you have signed up as a Cisco Webex Teams user, you can try it out by sending a message to armyknife@webex.bot.

More details about the Army Knife can be found on this blog .

2.5 The ActingWeb Model

The ActingWeb micro-services model and protocol defines a bot-to-bot and micro-service-to-micro-service communication that allows extreme distribution of data and functionality. This makes it very suitable for holding small pieces of sensitive data on behalf of a user or "things" (as in Internet of Things). These sensitive data can then be used and shared in a very granular and controlled way through the secure and distributed ActingWeb REST protocol. This allows you to expose e.g. your location data from your phone directly on the Internet (protected by a security framework) and to be used by other services **on your choosing**. You can at any time revoke access to your data for one particular service without influencing anything else.

2.5.1 The ActingWeb Micro-Services Model

The programming model in ActingWeb is based on an extreme focus on only representing one small set of functionality and for only one user or entity. This is achieved by not allowing any other way of calling the service (in ActingWeb called a "mini-application") than through a user and the mini-app's REST interface (a user's instance of a mini-application is called an *actor* in ActingWeb). From a practical point of view, getting xyz's location through the REST protocol is as simple as doing a GET `http://mini-app-url/xyz/properties/location`.

There is absolutely no way of getting xyz's and yyz's location information in one request, and the security model enforces access based on user (i.e. actor), so even if you have access to `http://mini-app-url/xyz/properties/location`, you may not have access to `http://mini-app-url/yyz/properties/location`.

Any functionality desired across actors, for example xyz sharing location information with yyz **MUST** be done through the ActingWeb REST protocol. However, since the ActingWeb service-to-service REST protocol is standardised, **any** service implementing the protocol can easily share data with other services.

2.5.2 The ActingWeb REST Protocol

The ActingWeb REST protocol specifies a set of default endpoints (like `/properties`, `/trust`, `/subscriptions` etc) that are used to implement the service-to-service communication, as well as a set of suggested endpoints (like `/resources`, `/actions` etc) where the mini-applications can expose their own functionality. All exchanges are based on REST principles and a set of flows are built into the protocol that support exchanging data, establishing trust between actors (per actor, not per mini-application), as well as subscribing to changes.

2.5.3 The ActingWeb Security Model

The security model is based on trust between actors, not mini-applications. This means that each instance of the mini-application holding the sensitive data for one particular person or thing **must** be connected through a trust relationship to another ActingWeb actor, but it doesn't have to be a mini-application of the same type (like location sharing), but could be a location sharing actor establishing a trust relationship with 911 authorities to allow emergency services to always be able to look you up.

There are currently two ways of establishing trust between actors: either through an explicit OAuth flow where an actor is tied to somebody's account somewhere else (like Google, Box.com, etc) or through a flow where one actor requests a trust relationship with another, which then needs to be approved either interactively by a user or programatically through the REST interface.

See http://actingweb.org/ for more information.

2.6 Building and installing

```
# Build source and binary distributions:
python setup.py sdist bdist_wheel --universal

# Upload to test server:
python setup.py sdist upload -r pypitest
twine upload --repository pypitest dist/actingweb-a.b.c.*

# Upload to production server:
twine upload dist/actingweb-a.b.c.*
```

CHAPTER 3

Getting Started

The easiest way to get started is to start out with the actingwebdemo mini-application http://acting-web-demo.readthedocs.io/.

It uses the Flask framework to set up all the REST endpoints that the ActingWeb library exposes, and pretty much the entire specification.

If you want to use another framework, that is easy, the application.py shows how this is done for Flask.

3.1 How it works

An ActingWeb mini-application exposes an endpoint to create a new actor representing one instance on behalf of one person or entity. This could for example be the location of a mobile phone, and the app is thus a location app. The ActingWeb actor representing one mobile phones location can be reached on https://app-url.a-domain.io/actor-id and all the ActingWeb endpoints to get the location, subscribe to location updates and so on can be found below this actor root URL.

Below is an example of how the trust endpoint is exposed in the demo app. The same code here handles all the three different ways the trust endpoint can be invoked (to create a new trust relationship between two actors or change it). The Flask code maps these URL patterns and invokes the app_trust() function. In Handler, the Flask request is parsed and simplified into a dictionary that can be easily passed to the ActingWeb framework. The process() function is then invoked with the right parameters and finally get_response() is used to map the ActingWeb response into a Flask response.

Some endpoints can return template variables to be used in a rendered web output. The variables can be found in h.webobj.response.template_values and can easily be rendered with Jinja2 (Flask's template renderer) this way: return render_template('aw-actor-www-root.html', **h.webobj.response.template_values)

:: @app.route('/<actor_id>/trust', methods=['GET', 'POST', 'DELETE', 'PUT'], strict_slashes=False)
    @app.route('/<actor_id>/trust/<relationship>', methods=['GET', 'POST', 'DELETE', 'PUT'], strict_slashes=False) @app.route('/<actor_id>/trust/<relationship>/<peerid>', methods=['GET', 'POST', 'DELETE', 'PUT'], strict_slashes=False) def app_trust(actor_id, relationship=None, peerid=None):

        h = Handler(request) if peerid:

if not h.process(actor_id=actor_id, relationship=relationship, peerid=peerid): return Response(status=404)

elif relationship:

if not h.process(actor_id=actor_id, relationship=relationship): return Response(status=404)

else:

if not h.process(actor_id=actor_id): return Response(status=404)

return h.get_response()

3.2 Config Object

In order to set the configuration, the config function is used to return a config object that is passed into the actingweb framework.

The ActingWebDemo app shows mostly empty values as placeholders where APP_HOST_FQDN is the only you must make sure matches the domain where the app is hosted.

3.3 All Configuration Variables and Their Defaults

```
# Basic settings for this app
fqdn = "actingwebdemo-dev.appspot.com"  # The host and domain, i.e. FQDN, of the URL
proto = "https://"  # http or https
database = 'dynamodb'                           # 'dynamodb', for future other␣
↪databases supported
ui = True                                       # Turn on the /www path
devtest = True                                  # Enable /devtest path for test␣
↪purposes, MUST be False in production
unique_creator = False                          # Will enforce unique creator field␣
↪across all actors
force_email_prop_as_creator = True              # Use "email" internal property to set␣
↪creator value (after creation and property set)
www_auth = "basic"                              # basic or oauth: basic for creator +␣
↪bearer tokens
logLevel = "DEBUG"                              # Change to WARN for production, DEBUG␣
↪for debugging, and INFO for normal testing

# Configurable ActingWeb settings for this app
type = "urn:actingweb:actingweb.org:gae-demo"   # The app type this actor implements
desc = "GAE Demo actor: "                       # A human-readable description for␣
↪this specific actor
specification = ""                              # URL to a RAML/Swagger etc definition␣
↪if available
version = "1.0"                                 # A version number for this app
info = "http://actingweb.org/"                  # Where can more info be found

# Trust settings for this app
default_relationship = "associate"  # Default relationship if not specified
auto_accept_default_relationship = False  # True if auto-approval

# Known and trusted ActingWeb actors
```

(continues on next page)

(continued from previous page)

```
actors = {
    '<SHORTTYPE>': {
        'type': 'urn:<ACTINGWEB_TYPE>',
        'factory': '<ROOT_URI>',
        'relationship': 'friend',                # associate, friend, partner, admin
        },
}

# OAuth settings for this app, fill in if OAuth is used
oauth = {
    'client_id': "",                                # An empty client_id turns off␣
↪oauth capabilities
    'client_secret': "",
    'redirect_uri': proto + fqdn + "/oauth",
    'scope': "",
    'auth_uri': "",
    'token_uri': "",
    'response_type': "code",
    'grant_type': "authorization_code",
    'refresh_type': "refresh_token",
}
bot = {
    'token': '',
    'email': '',
}

# myself should be an actor if we want actors to have relationships with other actors␣
↪of the same type
actors['myself'] = {
    'type': type,
    'factory': proto + fqdn + '/',
    'relationship': 'friend',  # associate, friend, partner, admin
}
```

3.4 Tailoring behaviour on requests

The on_aw module implements a base class with a set of methods that will be called on certain actions. For example, requests to /bot can and should be handled by the application outside actingweb.

> The /bot path can be used
> to handle requests to the mini-application, for example to create a new actor or create a trust relationship between
> two actors, or just to handle incoming requests that don't use the actor's id in the URL, but where the actor can be
> identified through the POST data.‘‘

To make your own bot handler, make you own instance inheriting the on_aw_base class and override the correct method.

```
from actingweb import on_aw

class my_aw(on_aw.OnAWBase()):
```

(continues on next page)

(continued from previous page)

```
    def bot_post(self, path):
        # Do stuff with posts to the bot
```

CHAPTER 4

ActingWeb Specification - version 1.0

4.1 Introduction

4.1.1 This Document

This specification is written in the format of an ‘IETF <http://www.ietf.org>‘_ draft. However, there is currently no effort to submit this draft to a working group, nor to make it go through the process of becoming an IETF proposed standard or have any other official status. This may or may not be appropriate at some point, but the text is written in this format because people are used to reading specifications in this format and it is a well-proven approach.

4.1.2 Copyright Notice

4.1.3 Terminology

In this document, the key words “**MUST**”, “**MUST NOT**”, “**REQUIRED**”, “**SHALL**”, “**SHALL NOT**”, “**SHOULD**”, “**SHOULD NOT**”, “**RECOMMENDED**”, “**NOT RECOMMENDED**”, “**MAY**”, and “**OPTIONAL**” are to be interpreted as described in BCP 14, RFC 2119 and indicate requirement levels for compliant ActingWeb implementations.

4.1.4 Abstract

The ActingWeb micro-services model and protocol defines a bot-to-bot and micro-service-to-micro-service communication that allows extreme distribution of data and functionality. This makes it very suitable for holding small pieces

of sensitive data on behalf of a user or "things" (as in Internet of Things). These sensitive data can then be used and shared in a very granular and controlled way through the secure and distributed ActingWeb REST protocol. This allows you to expose e.g. your location data from your phone directly on the Internet (protected by the ActingWeb security framework) and to be used by other services *on your choosing*. You can at any time revoke access to your data for one particular service without influencing anything else.

The programming model in ActingWeb is based on an extreme focus on only representing one small set of functionality and for only one user or entity. This is achieved by not allowing any other way of calling the service (in ActingWeb called a "mini-application") than through a user and the mini-app's REST interface (a user's instance of a mini-application is called an *actor* in ActingWeb). From a practical point of view, getting xyz's location through the REST protocol is as simple as doing a GET *http://mini-app-url/xyz/properties/location*.

There is absolutely no way of getting xyz's and yyz's location information in one request, and the security model enforces access based on user (i.e. actor), so even if you have access to *http://mini-app-url/xyz/properties/location*, you will not have access to *http://mini-app-url/yyz/properties/location*.

Any functionality desired across actors, for example xyz sharing location information with yyz *MUST* be done through the ActingWeb REST protocol. However, since the ActingWeb service-to-service REST protocol is standardised, *any* service implementing the protocol can easily share data with other services.

The ActingWeb REST protocol specifies a set of default endpoints (like */properties*, */trust*, */subscriptions* etc) that are used to implement the service-to-service communication, as well as a set of suggested endpoints (like */resources*, */actions* etc) where the mini-applications can expose their own functionality. All exchanges are based on REST principles and a set of flows are built into the protocol that support exchanging data, establishing trust between actors (per actor, not per mini-application), as well as subscribing to changes.

The security model is based on trust between actors, not mini-applications. This means that each instance of the mini-application holding the sensitive data for one particular person or thing *must* be connected through a trust relationship to another ActingWeb actor, but it doesn't have to be a mini-application of the same type (like location sharing), but could be a location sharing actor establishing a trust relationship with 911 authorities to allow emergency services to always be able to look you up.

There are currently two ways of establishing trust between actors: either through an explicit OAuth flow where an actor is tied to somebody's account somewhere else (like Google, Box.com, etc) or through a flow where one actor requests a trust relationship with another, which then needs to be approved either interactively by a user or programatically through the REST interface.

4.1.5 Overview of operations

The ActingWeb is an application-layer communication protocol used between independent software components that can reside on any computer-powered device. These software components or services, referred to as mini-applications, implement the ActingWeb protocol to expose data, resources, and services to other ActingWeb mini-applications or to clients of the ActingWeb services (that not themselves are ActingWeb mini-applications). ActingWeb is targeted towards machine-to-machine communication, and can be used to build Internet of Everything (IoT) type functionality.

ActingWeb assumes a programming model where you can only access data and services through individual instances of the mini-application and where there is one and only one user or entity that is represented. That is, there is no concept of "get all instances" of something, only "get this data for this specific instance."

Thus, in ActingWeb terminology, an *actor* is an instance that is running and exposing the mini-application's services for one specific entity/user/device, i.e. a mini-application may implement location awareness functionality, and a single actor (an instance of the mini-application) may represent one physical mobile device and keep track of that device's current GPS position, as well as allow other actors to access that device's GPS position. This is contrary to typical web applications, where one application normally implements a set of functionality for many entities/users. However, another mini-application could offer location services for users specifically (and not a device) and then multiple location-aware devices and services associated with a single user could serve data to that user's actor. The scope of a mini-application is thus not a rigid concept, but rather a design principle.

In order to access an actor's data, a set of credentials is required. The credentials identifying the requesting actor are unique for the relationship between the two actors. Each such relationship has a trust level, controlling which level of access the requesting actor has to the actor's data. Hence, a mini-application is always contacted through a specific actor, and the credentials give access to the actor's data. The actors establish trust relationships with each other in many-to-many relationships where each relationship is bi-directional (optionally one-way) and has a trust level. The trust relationship is then used as authentication when requesting access. The mini-application itself specifies which access rights are given to which data, resources, and services for each trust level and thus outside the scope of this document, while the method used to create a trust relationship is specified in this document.

The mini-application functionality of the actor is accessed through a set of pre-defined URIs. The ActingWeb protocol for interaction **between** actors is based on RESTful principles, while RPC-style methods like SOAP can also be exposed as part of the mini-applications accessible functionality if desired. The ActingWeb protocol specifies a basic set of URIs (*endpoints*) that must be supported. Mini-applications may (indeed is likely to) offer also other resources and services than those specified in this document.

Each actor implements one and only one mini-application. The mini-application functionality and interface may be described in human or machine-readable form, dependent on the preference of the creator. A description of a mini-application is similar to the interface of a class in object-oriented terminology and exhaustively describes the mini-application so that it can be used by creators of other actors for the purpose of communication between the actors.

Each actor also offers a subscription endpoint, where actors and clients can request to be notified about changes to a given endpoint path (like /resources/target/12345). The notification is either a callback to an URI provided by the requestor or, optionally, as a subscription resource that can be polled.

An actor typically resides on a web server and is thus accessible via a root URI for that specific actor, ex. https://actingweb.net/apps/app1/897JGHGY76HGhgK/

An important aspect of actors is how they relate to traditional user accounts found in interactive (and SOAP/REST-based) web sites and services: In a classic web site/web service implementation, the root URL is the entry point to a login page asking for login credentials or authentication. Credentials, including a username, are thus required as part of the web service request. An ActingWeb actor's root URL will typically not only identify the account in question, but the account's specific functionality, for example the voicemail inbox. The credentials used to access the actor are never an account username and password, but rather the credentials of a trust relationship that has been established between the requesting and requested actor. This trust can for example be established using OAuth or through an approval process as described in this document.

4.1.6 Structure of the Protocol

As a REST-based protocol, the ActingWeb protocol utilises the http protocol specification. All communication between actors is based on usage of the basic principles of an http envelope with a message type, headers and corresponding values, and a payload. Http responses are also re-used.

```
| Actor | ---- http(s) -----| Actor |
```

Figure, http(s) as transport

This specification defines the URI interface each actor must implement, which http methods shall be used to accomplish what, which http response codes shall be used in which situations, as well as definitions of the representations to be used in http requests and responses. There is a set of mandatory URI paths that any actor must respond to, as well as a set of optional paths to be used for specific pre-defined purposes.

The protocol also specifies how trust relationships shall be established, how to use them for authentication and authorisation, as well as how they are terminated.

Finally, the protocol specifies what a subscription is, how a subscription can be established, fulfilled, and cancelled.

4.1.7 Definitions

Endpoint: The ActingWeb protocol specifies a set of endpoints directly under the root URI of the actor that is used to get access to the functionalities offered by an actor

Actor: A software component that exposes the ActingWeb protocol interface and is accessible to other actors or clients

Action: An action is a resource exposed by an actor or an agent (see definition) that results an in external action like turning a lightbulb on or off, to be executed (i.e. external to the actor). The response may even be an OK, as in "action executed"

Agent: A software component that implements partial or in whole a mini-application according to the ActingWeb Specification and which offers not only data, but also resources, actions, or methods that are callable. As opposed to an actor, an agent is typically not publicly available (could be on and off due to battery limitations or connectivity) and often has a Proxy as peer that can receive requests on behalf of the Agent

Client: A software component that uses the ActingWeb protocol to access data, resources, or services from an actor, but which does not implement the ActingWeb protocol interface itself and thus cannot be contacted as an actor

Method: A method is an RPC (Remote Procedure Call) style service that is accessible through a specific URI

Mini-application: A set of functionality that can be instantiated to many actors. Consists of a set of data, resources, methods, and actions that other actors and clients can request and operate on. Defined by the mini-application definition.

Mini-application definition: Either a human- or machine-readable description of the data, resources, methods, and actions a mini-application implements. A human-readable format can be of any type, as long as it is complete enough for somebody to use the actors implementing the mini-application.

Mini-application type: A URN string prefixed with ActingWeb that uniquely identifies the mini-application, ex. urn:actingweb:domain.com:myapp

Resource: A resource is an entity exposed through a URI and other actors or clients can manipulate the resource through commands according to RESTful principles

Root URI: All actors have a root URI where it can be contacted using http type methods, either directly (if the URI is http/https) or encoded in the protocol used (i.e. _method=. . .)

Peer: An actor that has an established trust relationship with another actor. An actor will have many peers

Proxy: A software component that implements a mini-application according to the ActingWeb Specification, but which does not offer anything beyond data and proxying capabilities. A proxy has a peer Agent that implements the same mini-application that it is a proxy for. The agent is using the proxy as a publicly available storage and proxy for communication with other actors. The proxy may store requests for resources, actions, and methods and forward these when the agent comes online

4.2 The Mini-Application Model

4.2.1 What Is a Mini-Application?

The mini-application is the central element in the ActingWeb. The concept is very flexible, practically any tiny functionality, like holding one value, can be wrapped into a mini-application, or you can create one actor implementing all the functionality you need. If you have an existing web service, it may be tempting to just create one ActingWeb mini-application to expose your web service in the ActingWeb. However, you gain a lot more by creating mini-applications that follow the principles of the ActingWeb: user-centric, atomic, 24x7 alive, and isolated. These principles are explained in more detail further below.

The mini-application can in many ways be compared to a class from object-oriented programming languages. The class has private and public data, as well as methods that can be called with certain parameters and return values. The mini-application offers the same (and more) through an interface definition. A class has a name; the mini-application has a type. Both can be instantiated, the class into an object, and the mini-application into an actor.

4.2.2 The Mini-Application's Type

Each mini-application MUST have a type name in the form of a urn prefixed with *actingweb*. A unique namespace MUST be chosen by using either a rightfully owned domain name or email address. All other urns are reserved for use by a coordinating body, currently actingweb.org. Examples of valid mini-application types: *urn:actingweb:payment:creditcard*, *urn:actingweb:mydomain.com:coolapps:app1*, *urn:actingweb:user@domain.com:myapp2*

Any given mini-application type can have zero, one, or many actual implementations (for example in different programming languages, for different server platforms etc). A mini-application type MUST correspond to one and only one human- or machine-readable description.

4.2.3 Option tags

Option tags are textual tags indicating support for a specific functionality as specified in this specification (i.e extensions). The */meta/actingweb/supported* path MUST return a list of comma-separated option tags to indicate which OPTIONAL functionalities that the mini-application has implemented and thus the actor supports.

The below table summarises all the option tags (and thus optional parts) in this specification. Only the basic creation and deletion of an actor and the /properties and /meta paths are mandatory to implement, thus allowing the implementation of a very simple actor. Most actors will also support /callbacks and maybe /www to allow interaction using web pages and getting callback data from third party services. Third party interactions will often be coupled with implementation of /oauth to allow use of OAuth to get access.

Tag	**Description**
trust	The trust endpoint is available to request and establish regular, two-way trust relationships between actors. If trust is available, the actor should also be able to receive callbacks on /callbacks
onewaytrust	The version of trust implemented is more restrictive and although one actor A has a trust relationship with another actor B giving A access to B's functionality, the reverse is not true
subscriptions	The subscriptions endpoint can be used to establish subscriptions on the actor's data, actions, or resources
actions	The actions path is available and offers ways of triggering something to happen. Example:* /actions/turn-lights-off*
resources	The resources path is available and non-actingweb data, but relevant to the actor can be found under the resources path. Example: GET /resources/lights to get all lights available.
methods	The methods path is available and offers non-REST based API access. Example: */methods/soap/sendMessage*
sessions	The sessions path is available and offers access to session-based functionality. Example: */sessions/SIP/2f2ag-2696f-42gga*
www	The www path is available for human web-based interaction with the actor
oauth	The oauth path is available to do an OAuth2 authorisation flow. The /oauth path should give a redirect to the 3rd party authorisation web page that can be presented to the user
proxy	The actor implements capabilities to be a proxy
nested-properties	Announce support for deeper, nested json structures in /properties (beyond the mandatory attribute/value pairs)

4.3 The Actor

4.3.1 What is an Actor?

An actor is a running software component that is ready to respond to external requests, as well as internal events or requests (for example if residing on a mobile phone). The actor implements a small and finite set of functionality with the purpose of offering parts or all of that functionality to the outside world. An actor is not a replacement for a SOAP or REST-based web service, and is not a part of a web site for human interactions. Actors are made for machine-to-machine communication, and is ideal for e.g. bot-to-bot communication.

Each actor is, in the object-oriented terminology, an instantiation of a class, and thus can be seen as an object. The mini-application definition corresponds to the class definition and the mini-application itself to the class declaration. Like objects in object-oriented programming languages, actors can access other actors' data (if they are public) and call methods.

All actors MUST have a globally, unique root URI. The root URI can be in the form of a http or https URL, or any other type of URI that can allow two-way communication and which are specified in this or associated specifications. The root of the URI SHALL uniquely identify the actual actor, i.e. the mini-application instance: http://www.actingweb.net/miniapp1/my_actor_id/. In the case of an email, the full email address SHALL uniquely identify the instance: mailto:my_actor_id@Actingweb.net.

The same holds for SIP URIs, example: sip:my_actor_id@actingweb.net or possibly sip:myapp@actingweb.net;gruu=my_actor_id

4.3.2 Identity

Each actor's root URI MUST be globally routable and, unless the actor is offline, behind NAT or otherwise, a request to this URI MUST reach the actor. Each actor MUST have an id that is created at creation (instantiation) and which is valid throughout the lifetime of the actor.

The id MUST be globally unique. It is RECOMMENDED that a version 5 (SHA-1) UUID (RFC 4122) is used with the base URI of the location of actor as name input to the algorithm. The resulting UUID MUST be added to the base URI using the standard encoding. Example: http://actingweb.net/myapp/f81d4fae7dec11d0a76500a0c91e6bf6

4.3.3 Data Representation

All input and output representation specified throughout this document MUST by in UTF-8 unless otherwise specified. The default data representation is the JSON format and MUST be supported. Alternative data representations, like urlencoded form-data, XML or others, can also be supported and MAY be announced as a capability through the formats element in /meta/actingweb (see /meta endpoint section). Content formats should be negotiated with standard http header mechanisms.

4.3.4 Instantiation of Actors

Actors can be instantiated in several ways dependent on their context and environment. For example, an installed application on a computer or mobile phone that implements a mini-application as an actor, is instantiated the first time it runs. Such instantiation is based on installing the software on a new device, a factory- or manual process.

Actors that reside on a web server MUST implement dynamic instantiation through a mini-application factory present at the level right above the actors' root URIs, ex: http://www.actingweb.net/miniapp1. The actor or client requesting instantiation sends an http POST to the factory URI with the following OPTIONAL application/json data:

```
{
  “creator”: “username”,
  “passphrase”: “secret”,
  “trustee_root”: “uri”
}
```

A special creator user with username “creator” and passphrase “secret” MUST have full access to manage and access the actor through http basic authentication. If “creator” is not supplied, “creator” MUST be the default username.

The ‘trustee_root’ value is a URI pointing to the root URI of an actor that will act as a validator and manager of trust relationships for the new actor. This is typically used when another actor is instantiating a new actor to get access to some new functionality.

If the creation of a new instance was successful, a 201 Created MUST be returned with the *Location* header set to the full root URI of the new actor instance. If the instantiation failed due to problems with the input parameters, a 400 Bad Request MUST be returned. Temporary problems in instantiating a new actor SHOULD result in a 503 Service Temporarily Unavailable. Other server errors SHOULD result in the 500 Internal Server Error return code.

4.3.5 Deleting Actor Instances

An actor instance can be deleted with all its data by sending a DELETE request to the root URI of the actor. The request MUST be authenticated and only the creator or ‘admin’ relationships are allowed to delete an actor (see the section on trust relationships). Upon receiving a valid DELETE, the actor SHOULD clean up its data, any 3rd party webhooks, and stop responding to requests on and below its root URI. If the actor is deleted after being migrated to a newer version/new actor, it MAY choose to respond with 301 Moved Permanently where the Location header is set to the root URI of the new actor.

4.3.6 Endpoints

ActingWeb actors communicate by sending http requests to each other on pre-defined URI paths with pre-defined meanings and representations. These messages can be sent over http/https or any other transport that can support such exchanges.

The available endpoints are the most important structural elements of the protocol specification. Any actor or client will expect another actor implemented according to the ActingWeb specification to respond to a certain set of paths right beneath the root URI of the actor. These so-called endpoints have a defined purpose and use, and the use of http methods and representations are specified in this document.

Below is a summary of the defined endpoints and a summary of their purpose. Details on how to use these endpoints can be found later in this specification in the Endpoints chapter, as well as separate chapters for /trust and /subscriptions. Each of the OPTIONAL access paths has a corresponding options tag (as found in meta/actingweb/supported) with the same name as the path. I.e. if the /www path is implemented, the meta/actingweb/supported options MUST include “www” as an options tag.

Endpoint	Status	Description	httpd methods
/meta	MUST	Meta-information about the actor and it's mini-application.	GET
/properties	MUST | Simple, public data that the actor for simplicity would like to expose in a unified way to simplify read and write. The interface is RESTful where the URI specifies the property or the part of the property tree to operate on. The data are in simple attribute/value pairs.		GET, PUT, DELETE, and POST
/actions	OPTIONAL	Actors may be able to execute actions that are not directly connected to data or resources, but where the action causes an external (maybe physical) event. A GET to an action returns status for the action, while a PUT or POST executes the action.	GET, PUT, POST
/callbacks	OPTIONAL	This URI is used by the actor to receive callbacks for trust and subscription creation, and subscriptions, as well deferred requests sent through a proxy.	PUT, POST
/resources	OPTIONAL	Any resources the actor wants to expose within a RESTful framework may reside here. The in and out representations can be freely defined by the mini-application. The URI path must specify the resource to operate on. GET MUST return information, PUT MUST change state of resource, DELETE MUST delete a resource, and POST MUST create a new resources.	GET, PUT, DELETE, and POST
/methods	OPTIONAL	Any RPC style web service that an actor wants to expose MUST be exposed under this path. There are no restrictions on how to use http methods or on representations. Thus, any RPC-type service (ex. SOAP, XMLRPC) can be exposed here. An http GET can thus give side-effects.	any
/sessions	OPTIONAL	Session-based communication (bi-directional messages or streams) can have their own signalling and media/content protocols. Two actors may want to establish a two-way session over a time-period for communication that is not suited to do over the signalling or media protocols. The /sessions path is made for this purpose. Also, an actor implementing a session-protocol may want to allow http-based access to session data and/or actions. The /sessions path may implement websockets for bi-directional flows.	any
/www	OPTIONAL	If the actor wants to expose a web application for human consumption, this path can be used. A special path /www/init is reserved for the presentation of a web form for humans to initialise a newly instantiated actor.	GET and POST
/oauth	OPTIONAL	The oauth endpoint is used to initiate a binding of the actor to an external OAuth-authorised service. Typically, this is used for actors that represent a service like Google Mail, Box, Dropbox, or any other service with OAuth-based APIs.	GET and POST
/subscriptions	OPTIONAL	Other actors use this endpoint to establish new subscriptions or check state of an existing. A POST to /subscriptions will create a new subscription and return the path to the newly created subscription in the Location header of the http response. A GET on the subscription returns status on the subscription.	GET and POST
/trust	OPTIONAL	Other actors use this endpoint to create or remove relationships. A POST to /trust/trusttype will request the creation of a new relationship of type trusttype. The path to the new trust relationship is returned in the Location header.A DELETE to the trust URI will terminate the relationship. GET on the URI will send status information about the relationship (ex. approved, pending etc).	GET, PUT, DELETE, POST

4.3.7 Agents

Agents are actors that together with a proxy implement a mini-application fully and according to the principles outlined in this specification. Together, an agent/proxy pair MUST implement all mandatory aspects of the mini-application. However, as opposed to the actor, agents MAY be temporarily unavailable or accessible only through an actor acting as a proxy for the agent. For simpler functionality and low-powered devices, a client can instead be used. It can be as simple as just holding credentials that allow updating values, e.g. posting GPS coordinates once in a while through a POST.

4.3.8 Proxies

A proxy is an actor that implements /meta, /properties, /trust, and /subscriptions according to the definition of a mini-application, and that has a special proxy relationship with an agent that is just partially available to other actors. Together, the proxy and the agent MUST implement the mini-application fully. These two actors, the proxy and the agent, together form a pair representing the same entity or data. The proxy MUST be available to other actors on the Internet through a URI (or on a meaningfully defined private network), while the agent can be intermittently unavailable and will typically only communicate with its proxy. They MUST have a two-way trust relationship of the type proxy.

An important characteristics of a proxy is that it MAY be implemented totally generically for any type mini-application and only needs a configuration that defines the /properties endpoint to respond to. In this case, all other endpoints like /actions, /resources etc will just be proxied to its agent. However, a proxy MAY implement more mini-application specific functionality to offload from the agent or if it otherwise makes sense.

A paired proxy and the agent SHOULD have implicit subscriptions to each others /properties access paths in order to synchronise their /properties data.

Being a Proxy For Endpoints Proxied To The Agent

A proxy MAY implement endpoints on behalf of the proxy/agent pair. These MUST be implemented just like any other actor. However, a proxy MAY also implement endpoints that are proxied to the agent that is not available directly. Such a request to a path like /actions/dosumthin MUST give a http 307, Temporarily Moved. This indicates to the requestor that this is a proxy, and that the request should be sent as a subscription creation request to /subscriptions using the path as target (target = actions, sub-target=dosumthin) and with the parameter "proxied" set to true in the payload. If the proxy sees a subscription request with the parameter proxied set to true, it MUST treat the "data" element of the payload in the subscription request as the original payload to the endpoint, and thus forward this payload to the agent when forwarding.

4.4 Endpoints

4.4.1 /meta (MUST)

Each actor has a set of meta-information used to facilitate effective discovery communication and is optionally a part of trust establishment if the requesting actor wants to validate that a given URL for an actor actually points to an actor of a certain and/or version. The meta-information can be found under the /meta path directly below the root URI of the actor. The http GET method is readable without a trust relationship. OPTIONAL paths that are not supported MUST result in 404 Not found.

These are the paths available:

Path	Status	Description
/meta/id	MUST	The id of the actor and is identical to the id embedded in the actor's root path.
/meta/type	MUST	Returns mini-application type in a 200 OK with a one-line text/plain body containing the urn type of the mini-application.
/meta/version	MUST	Returns the version of the mini-application type in a 200 OK with a one-line text/plain body containing the version number in the format a.b.c or a.b where a and b are digits 0-9.
/meta/desc	MUST	Returns a human-readable description of the actor. The description MAY be based on a mini-application template, where information about this actor instance is substituted. Ex. "This actor allows deposits and withdrawals on Bob Bobson's account #1234556 in the Bank of Lutitia".
/meta/info	OPTIONAL	Returns a 302 Refer with Location pointing to a URI with general human-readable web page about the mini-application.
/meta/actingweb/version	MUST	Returns a 200 OK with a text/plain body containing the version number of this specification that the mini-application supports, I.e. 1.0
/meta/actingweb/supported	MUST	Returns a 200 OK with a text/plain body with a comma-separated list of tags identifying the supported OPTIONAL options found in this specification: option1,option2 See the Option tags section earlier in this document. An empty list means that only the mandatory requirements in this specification is supported.
/meta/actingweb/formats	OPTIONAL	Returns a 200 OK with a text/plain body with a comma-separated list of the supported OPTIONAL formats supported by the actor: e.g. xml,txt… The format name MUST be the standardised mime-type file extension. Only formats with standardised mime-types are allowed.
/meta/raml	OPTIONAL	A mini-application MAY choose to represent the mini-application through a RAML file (*http://raml.org*) Returns a URI to where the RAML file is found.

/meta	OPTIONAL	Returns a 200 OK with an application/json body with an json document containing everything defined under the /meta path. Example:
		`{`
		`  "actingweb": {`

4.4.2 /properties (MUST)

Quick interactions between actors are important in ActingWeb. The /properties access path facilitates easy read and write of simple data. If you need to read and write more complex data structures or XML documents, you can use the access paths /resources (for RESTful access to data) or /methods (for RPC-style access like SOAP and XMLRPC).

The /properties path is meant to contain the basic, most important data for the actor's functionality. Most often a newly instantiated actor needs some properties set before proper functioning. However, /properties is not for static configurations only, but also for dynamic data. This implies that the actor MUST use current /properties data in its execution (and not treat them as a configuration file).

The /properties path supports the http methods GET, PUT, DELETE, and POST. The requests can be unauthorised or authorised through a trust relationship by presenting credentials in the Authorization header of the http method.

All attributes under /properties MUST be writable by the admin role.

Attribute/value pairs

The properties that can be stored under /properties are untyped, UTF-8 encoded attribute/value pairs. The semantics of the attribute/value pair (as defined in the mini-application definition) must be used to convert representations into their specific types, for example a string-representation of an integer ("1234") into the integer value 1234.

The below table shows the relationship between the attribute name, it's value, and the URI where the value is stored.

Attribute	Value	URI to attribute
name	Alice	http://www.actingweb.org/app/78hjh76yug/properties/name

Note that to facilitate use of proxies, actors MUST NOT apply any logic or process on semantics when a property is changed using PUT or POST. Syntax SHOULD be checked. This implies that any processing logic in an actor using /properties values must assume that the values can be semantically invalid or even harmful and must do error handling accordingly.

A mini-application MAY choose to support the value for each attribute as either a blob or a json structure and must indicate in the returned MIME-type whether the returned value is text/plain, application/json, or any other content. A GET on /properties MUST return a proper application/json document with all the attribute value pairs. A mini-application MAY also support targeting nested json structs in the path, i.e. /properties/address/street/number, but MUST not assume that such support is present in peer actors. Such support SHOULD be announced as an options tag, *nestedproperties*.

GET

GET methods are used to retrieve properties. A GET can be done for a specific attribute or for the whole set of attributes. The response MUST be a 200 OK with a body of content type application/json.

Example:

When a GET request targets an attribute, the returned representation is the value of that specific attribute only using text/plain as content type: GET /app/78hjh76yug/properties/firstname

A GET for an empty /properties (i.e. no attribute/value pairs set) or a GET for a non-set attribute should result in a 404 Not found from the actor. If the attribute is not accessible without a trust relationship, a 401 Unauthorised MUST be returned. If the request's current trust relationship is not sufficient, a 403 Forbidden MUST be returned.

PUT

The PUT method is used to add or change an attribute/value pair. A PUT to an existing attribute will change the value of that attribute.

All actors MUST accept PUT requests to change a specific element. Successful change of the value MUST result in a 201 Created response.

```
Example:

PUT /app/78hjh76yug/properties/firstname
```

A PUT request to an attribute name not supported by the actor MUST result in a 404 Not Found. If the attribute is not writable without a trust relationship, a 401 Unauthorised MUST be returned. If the request's current trust relationship is not sufficient, a 403 Forbidden MUST be returned.

A PUT body can be of type application/json and MAY be stored as a blob by the mini-app, however, it MAY support nested json and thus MAY also support PUT /app/78hjh76yug/properties/people/person1/firstname

POST

The POST method is used to add or change a collection of attribute/value pairs. Only the /properties root endpoint MUST support POST. Content type application/json MUST be supported and application/x-www-form-urlencoded MAY be supported if the application supports web-based interactions. Successful change of all the values MUST result in a 201 Created response. An error on one or more values MUST result in no values changed and 409 Conflict returned.

```
Example:

POST /app/78hjh76yug/properties
```

Any non-supported attribute names MUST result in a 400 Bad Request. If any of the attributes are not writable without a trust relationship, a 401 Unauthorised MUST be returned. If the request's current trust relationship is not sufficient, a 403 Forbidden MUST be returned.

4.4.3 /actions (OPTIONAL)

Choosing between the /actions or other endpoints like /methods and /resources cannot be done according to clear-cut rules, but each endpoint has some restrictions that may or may not suit what you are trying to accomplish and the one matching what you are trying to do, is the best.

The /actions path is dedicated to operations or actions that not only changes the state of a resource or updates a database, but where triggering the action actually does something outside the actor. An example may be a video recorder where its actor can be requested to record on a specific channel at a specific time.

Any action below the /actions path (ex. /actions/record) MUST respond to POST. This will create or execute a new action. The data representation to be used in the body of the POST is specific to the mini-application. If the actor offers a callback functionality for status updates, the callback URL should be included in the request data representation. The /callbacks endpoint MAY be used by adding an element to the path, e.g. /callbacks/actions.

A successful action MUST return 201 Created. The body of the response MAY contain a mini-application specific data representation detailing the outcome of the action. The response MAY include a Location header pointing to a URL representing the action requested (e.g. /actions/record /3421433). This URL MUST respond to GET requests containing a representation of the action status/progress. The data representation is specified by the mini-application. If allowed, the cancellation of an action SHOULD be available through a DELETE request to the given location, while a PUT to the specific action URI MAY be used to change the action while in progress (for example, temporarily suspend a process).

4.4.4 /callbacks (OPTIONAL)

When an actor is requesting subscriptions, actions, sessions or other functions where a callback is required, the actor MUST create a new leaf node under the /callbacks path. The URI MUST expect POST requests with a data representation according to the requested path the callback was established for. It is up to the mini-application to keep track of the format expected for each callback by establishing sub-paths below /callbacks, e.g. /callbacks/subscriptions/… to handle callbacks on subscriptions and so on.

All requests to /callbacks from other actors or clients MUST be authenticated using the shared secret as bearer token (*Authorization Bearer xxxxxxx*) or be an anonymous POST from a non-ActingWeb application. All /callbacks requests without authentication data MUST return 401 Authentication required, regardless of the callback URI exists or not. Requests for non-existent /callbacks URIs with authentication data SHOULD always return 403 Forbidden.

Differentiating between various types of callbacks and authentication SHOULD be done be adding a path to callbacks/, e.g. callbacks/{callback_type}.

A successfully received POST MUST result in a 204 No Content or 200 Ok (with content). The actor pushing the callback will then clear the callback.

```
Example of a callback:

1. Actor B is interested in actor A's /properties and establishes a
  subscription on actor A's /properties. A callback is established by B
  on : <rooturiB>/callbacks/subscriptions/<actorAid>/afb343f3edfe
2. Actor A's /properties/firstname changes and it uses B's callback
  URI to notify about the change

Actor B thus receives a POST request on its callback URL (Actor B’s root
URI is
http://www.actingweb.net/myapp/f81d4fae-7dec-11d0-a765-00a0c91e6bf.
Actor A’s id is 9f1c331a3e3b5cf38d4c3600a2ab5d54:

POST
  http://www.actingweb.net/myapp/f81d4fae-7dec-11d0-a765-00a0c91e6bf/callbacks/
↪subscriptions/9f1c331a3e3b5cf38d4c3600a2ab5d54/afb343f3edfe

Bob

204 No content

Actor B receives the content in POST (in the case of /properties
changes, the content is application/json) and can immediately identify
this as a callback from actor A, as well as identify the specific
subscription this is a callback for.
```

4.4.5 /resources (OPTIONAL)

The /resources access path is reserved for an actor’s exposure of resources according to RESTful principles (http://en.wikipedia.org/wiki/Representational_State_Transfer). The exposure of resources MUST follow the following rules:

- Resources and sub-parts of a resources MUST be addressable by a URI where the path identifies the part of the resource that the request targets
- GET requests MUST not change state
- Any http method MAY be supported
- Non-GET/POST methods MAY be implemented using POST with the form variable _method set to the real method requested

Data representations and other decisions are up to the mini-application.

4.4.6 /methods (OPTIONAL)

The /methods access path is reserved for RPC (Remote Procedure Call) methods like XML-RPC, SOAP, etc. The paths address methods and a GET request MAY change state as the path and/or GET parameters of the request may include information about the action.

The intention of this path is to allow actors to expose traditional/existing RPC-style methods and isolate such methods to avoid unexpected state change through GET requests on other access paths.

```
Example:

GET /methods/persons/add?firstname=Bob&Lastname=Bobson
```

There are no http response codes, data representations, or other restrictions for this access path.

4.4.7 /sessions (OPTIONAL)

The /sessions access path is reserved for session-type communication between two actors and will always have two parties. The purpose is to enable two actors to create a way to share state and keep track of that state over time. An actor can provide a session type by exposing the session type right below the /sessions path, ex. /sessions/im to identify im, instant messaging sessions. This location MUST respond to POST requests by returning 201 Created with a Location header pointing to a newly created session, ex. /sessions/<requesting-actors-id>/fbe654aacef where fbe654aacef is a session id uniquely identifying this session.

The POST request MAY have an application/json body containing a callback URI that is URL-encoded ("callback": "uri"). If not, the requesting actor MUST respond to requests on the "mirrored" URL, /sessions/<requested-actors-id>/<sessionid>, and the requested actor MUST start sending session-related requests to this URL. If the session creation fails, a 400 Bad request MUST be returned, or if the failure is caused by a server problem, 500 Internal Server Error SHOULD be returned.

Subsequent communication between the two actors SHOULD continue on the returned new session URI and callback URIs using http methods and data representations as specified by the actors' mini-applications. Extensions may specify the http methods and data representations for specific type sessions and it is RECOMMENDED that if such extensions exist, the actors use the extension to facilitate session-type exchanges between different types of mini-applications.

A DELETE request to the session URI MUST terminate the session and return a 200 Ok. If there was a problem terminating the session, a 500 Internal Server Error SHOULD be returned.

Sessions may of course be established outside the ActingWeb actor implementation. The actor MAY choose to expose on-going sessions on other protocols through the /sessions path to allow simple signalling and session updates without the explicit creation of the session as described above. An example could be a SIP-based calling application where an on-going session (for example an instant messaging session) can be exposed through the URI /sessions/*SIP Call-Id*. This can for example allow non-SIP actors to insert messages into the dialog.

4.4.8 /www (OPTIONAL)

The /www access path is a special path as it is not meant for actor to actor communication, but rather allows humans to interact with the actor in a simple way. An actor may choose to expose a full web application below this path.

The /www path does not have any particular restrictions except on /www/init, see next section.

/www/init

If the /www/init path exists, it MUST present a human-readable form with /properties as the defined html form action. The path MUST be authenticated using HTTP DIGEST with username 'creator' (or the username established as the creator) and the passphrase as set when the actor was instantiated.

This form is intended to allow a newly created actor to be initialised by a human being with data in /properties. Thus, when submitting the form, the forms data will be sent to /properties in a POST request. Mini-applications supported this type of initialisation MUST, in addition to application/json, support POST of forms data to /properties.

4.4.9 /oauth (OPTIONAL)

The /oauth endpoint is used if the mini-application supports attaching to a 3rd party service using OAuth for authorisation. This way, an actor can easily expose services to other actors, e.g. a mini-application can offer users to create an actor that represent them towards a text-messaging service and thus easily allow other actors or clients to send text-messages to that user (or on behalf of the user). Such an actor could for example expose /actions/message_me to allow other actors to send text messages to the user who has (OAuth) authorised the actor.

Similar to the /www endpoint, the /oauth endpoint assumes human interaction as the actor should redirect to the 3rd party service's OAuth authorisation web page if a valid oauth token is not found for this actor. Obvisoulty, this page may be embedded in an application.

The /oauth endpoint MUST be able to handle the OAuth2 flow with a redirect back where the code URL parameter is set. It is RECOMMENDED that the mini-application offers a root URL /oauth (i.e. actingweb.net/myapp/oauth) that can be used as the callback URL registered with the 3rd party OAuth service and that the state parameter in the initial OAuth2 request (see the OAuth2 specification) is set to the actor's id). This special root URL can then parse the actor id from the state parameter and redirect to actingweb.net/myapp/<actor-id>/oauth?code=... where processing of the code can be done and the final token request to the 3rd party service can be done.

4.5 /trust - Trust Relationships (OPTIONAL)

4.5.1 Trust Model

Trust relationships form the basis of interaction between actors and is the primary reason for why accounts (with usernames and passwords) are not necessary. Each actor only needs to know the relationships itself has to other actors (with the exception of the creator user credentials). The trust relationship credentials are then used in all communication between the actors using existing methods for authentication.

Each actor (agent or proxy) is responsible for its own set of trust relationships with other actors. Each relationship is bi-directional where one actor initiates a trust relationship that then needs to be approved by the other actor. A mini-application can choose to implement one-way trust levels if necessary for its application, but this is not mandatory to implement. Each actor is responsible for storing and recognising actors it has trust relationships with. Each trust relationship has one out of a set of trust levels as defined in this specification.

A newly formed actor has no trust relationships, and new relationships are formed by requesting a trust relationship. The request is processed by the actor and the request can be approved in real-time or at a later time.

Once a trust relationship has been granted, the actor includes authentication details in all subsequent requests through a bearer token. Access to an actor's resources can thus be granted based on the relationship.

Trust relationships are managed authoritatively by the actor that granted the request, and a relationship can be revoked unilaterally at any time by either the granting or accepting actor.

This specification specifies how to use the http Bearer token method for authentication after exchanging a shared secret as part of the trust relationship creation. An optional verification process is also specified to ensure that both actors can trust the authenticity of the domain hosting the actor. Using https, the actors can mutually assure that their root URIs are correct. However, this specification does not specify how an actor decides on whether a given domain should be trusted. Also, while the methods described here are sufficient for most Internet-based applications, the specification does not provide methods on a security level where each request's integrity can be assured.

4.5.2 Determining Which Relationship to Request

The definition of the mini-application SHOULD contain information about what type of access is given for each type of trust relationship. The actor requesting the access will have some knowledge of the mini-application in order to use it, however, it may be useful to request a human-readable description for each trust relationship directly from the actor. This information can be presented to a user to determine if a given trust relationship is desired or to choose which relationship type to request. A GET to the uri of the trust relationship type + /desc MAY return such a human-readable description. Unlike the /meta/desc description, the text may be generic for the mini-application.

```
Example:

Request to server **\ https://actingweb.net/ GET
/myapp/f81d4fae-7dec-11d0-a765-00a0c91e6bf6/trust/friend/desc

200 OK
```

A friend can deposit and withdraw money as frequently as monthly, but limited up to an amount of $100. A specific friend relationship will also most likely establish an explicit limit at the requested amount.

4.5.3 Relationships and their data

The ‘creator’ user and ‘admin’ relationship MUST allow the retrieval of trust relationships through a GET to /trust and to /trust/’relationship_type’. The content is application/json. A request on a relationship type MAY also be supported and filter on a specific relationship, but give the same output. If no relationships exist, a 404 Not found MUST be returned.

```
Example:

Request to server **\ http://actingweb.net/
GET /myapp/f81d4fae-7dec-11d0-a765-00a0c91e6bf6/trust/friend

200 OK

[
{
  "secret": "ecb8a519288db1498a9b04706fc19e52abd3e0c0",
  "verified": false,
  "peerid": "e41f4aae-4dee-10d0-b725-0af0a413bcf2",
  "relationship": "friend",
  "baseuri":
  "http://actingweb.net/myotherapp/e41f4aae-4dee-10d0-b725-0af0a413bcf2",
  "desc": "Test friend relationship",
  "peer\_approved": true,
  "type": "urn:actingweb:actingweb.org:gae-demo",
  "id": "f81d4fae-7dec-11d0-a765-00a0c91e6bf6",
  "approved": false
},
{
  "secret": "8f4e4e86f249599c4be21aa4445065d4e6905cd4",
  "verified": true,
  "peerid": "testid",
  "relationship": "friend",
  "baseuri": "testurl",
  "desc": "Test friend relationship",
  "peer\_approved": true,
  "type": "urn:actingweb:actingweb.org:gae2-demo",
```

(continues on next page)

(continued from previous page)

```
  "id": "f81d4fae-7dec-11d0-a765-00a0c91e6bf6",
  "approved": true
}
]
```

'creator' and 'admin' MUST also be allowed to do a GET on a specific relationship to retrieve its status. The codes and contentreturned MUST be the same as the ones used when the owner of the relationship makes the request.

The fields are all MANDATORY and are explained below.

Field	Description
id	Actor id of the actor owning the relationship, i.e. MUST be the same as the actor identified in the request
type	The urn: prefixed type of the peer in this relationship
baseuri	The root URI of the peer in the this relationship
relationship	The relationship type
peerid	The id of the peer. This is also embedded in the baseuri
secret	The shared secret to be used as bearer token
verified	Bool that specifies if the remote peer has been verified as reachable on the baseuri
approved	Bool that specifies if this relationship has been approved
peer_approved	Bool that specifies if this relationship has been approved by the peer
desc	Human-readable description of the relationship

4.5.4 Creation of a Trust Relationship

When actor A wants to establish a trust relationship with B, it MUST send a POST request (step 1 in table below) to B's /trust/<trustlevel>. The request does not carry authentication and has an application/json body. Actor B responds (step 2). If actor B through the request can immediately approve the relationship, it MUST respond with 201 Created. If actor B immediately can deny the request, a 403 Forbidden MUST be returned. If the request is well-formed and actor B is ready and willing to process the request, B MUST respond with 202 Accepted. In both cases, the URL of the newly created relationship MUST be returned in the Location header in the response. The URL MUST be formatted the following way: /<baseuri-with-actor-id>/trust/<trustlevel>/<requesting-actor-id>.

If the requested actor wants to do a verification of the requesting actor, it MUST use baseuri, id, secret, and verify attributes to do a GET request to <baseuri>/trust/<trustlevel>/<requested-actor-id> using the secret as the Bearer token in the Authorization header. If the requesting actor supports verification, it MUST accept a GET request to this "reverse" URL, verify the shared secret, and then return the same secret as sent in "verify" as part of the trust relationship request, in the "verificationToken" attribute. If verification is not supported, the regular trust relationship attributes should be returned in application/json content.

B will later try to notify A about an approved trust relationship at the reverse URI /<requestor-id>/trust/<trustlevel/<requested-actor-id>, or a GET poll to the relationship's URL can be used to retrieve status.

The defined attribites to include in the POST request:

Attribute		Description
secret	MUST	Value is the shared secret that actor A will use when authenticating with actor B and the other way around. The requesting actor needs to supply the secret in order for the requested actor to verify the requesting actor.
baseuri	MUST	The base URI of the requesting actor (with id).
id	MUST	The id of the requesting actor.
type	MUST	The urn: type of the mini-app.
desc	OPTIONAL	Human-readable description that explains what the relationship is meant for. Should be phrased to allow a human to evaluate whether to approve or reject the request and later to recognise what the relationship is doing. Ex. "Service subscription of monthly $29.95 for the Geekly Review magazine."
verify	OPTIONAL	The value should be a secret that is used to verify the baseuri and the authenticity of the requestor.

```
Example:

Request to server `*http://actingweb.net/* <http://actingweb.net/>`_

POST /myapp/f81d4fae-7dec-11d0-a765-00a0c91e6bf6/trust/friend

{
  "secret": "8f4e4e86f249599c4be21aa4445065d4e6905cd4",
  "baseuri": "http://myserver.org/app2/e41f4aae-4dee-10d0-b725-0af0a413bcf2",
  "id": "e41f4aae-4dee-10d0-b725-0af0a413bcf2",
  "type": "urn:actingweb:actingweb.net:myapp",
  "desc": "A friend relationship between actors from myapp and myotherapp",
  "verify": "66b6691aae69fb75919e754976a8e2eb6d2719ac"
}

202 Accepted
```

It is RECOMMENDED to use https in this POST exchange as this will protect the shared secret from eavesdropping. Using http SHOULD be reserved for trust requests between actors in a controlled environment like inside an IPsec tunnel where eavesdropping can be ruled out.

After the initial request and response, the URI location of the new trust relationship (i.e. as in the path in the example above) MUST respond to GET requests with response codes as described above (5.). I.e. if the request has not yet been concluded or evaluated, 202 Accepted MUST be returned. If the relationship has been refused, 403 Forbidden MUST be returned. And if the relationship has been approved, 201 Created MUST be returned. Such GET requests MUST be authenticated using the secret as a bearer token in the Authorization header, thus ensuring that only actor A can request an update of the status.

The below table shows an overview of the process where actor A creates a trust relationship with Actor B (i.e. actor B trusts actor A).

Step	**ACTOR A: http://myserver.org/app2/e41f4aae-4dee-10d0-b725-0af0a413bcf2**	**ACTOR B: http://actingweb.net/myapp/f81d4fae-7dec-11d0-a765-00a0c91e6bf6**
1. Request relationship	POST request to initiate request for trust relationship POST*/myapp/f81d4fae-7dec-11d0-a765-00a0c91e6bf6/trust/friend*	
2. Immediate response to request		POST response Returns http response to indicate acceptance or not: 201 Created, 202 Accepted, 403 Forbidden
3. Polling for result (OPTIONAL)	Awaits B's processing, polling can be done (see 3. for return codes). GET */myapp/f81d4fae-7dec-11d0-a765-00a0c91e6bf6/trust/friend/e41f4aae-4dee-10d0-b725-0af0a413bcf2*	

4. Notification of result (OPTIONAL)		POST */trust/friend/f81d4fae-7dec-11d0-a765-00a0c91e6bf6*
		Sends approval or refusal on relationship request in an application/json body:
		{
		"approved": True
		}

Accepting Or Rejecting A Request For Trust Relationship

It is entirely up to the actor receiving the trust request to use whatever methods or processes necessary to evaluate and conclude upon the request. How this is done is outside the scope of this specification. An actor may for example refuse to accept requests with base uris that are not known by the actor and auto-approve requests from other base uris. Criteria for accepting a trust relationship request SHOULD be documented in the mini-application definition.

Use of Callback To Verify

A callback can be used for verification of requesting actor and for notification of the result of the request. The table below shows on overview of the steps involved. Only notification of result is shown as polling was shown in the above section. The below section specifies in detail how the verification is done.

Step	ACTOR A: http://myserver.org/app2/e41f4aae-4dee-10d0-b725-0af0a413bcf2		ACTOR B: http://sctingweb.net/myapp/f81d4fae-7dec-11d0-a765-00a0c91e6bf6
1. Establish trust callback	Creates verification callback URI on /trust/friend/f81d4fae-7dec-11d0-a765-00a0c91e6bf6		
2. Request relationship	POST */myapp/f81d4fae-7dec-11d0-a765-00a0c91e6bf6/trust/friend*	POST request to initiate request for trust relationship	
3.Verification of requesting actor			*GET /trust/friend/f81d4fae-7dec-11d0-a765-00a0c91e6bf6*using secret from step 2 as Bearer token and match "verify" from step 2 with the "verificationToken" returned in this step.
4. Response to request		Response to POST	Returns http response to indicate acceptance or not: 200 Ok, 201 Created, 202 Accepted, 403 Forbidden

Verification

Whether to do verification is up to the requested actor B. Actor A's provided baseuri parameter and Actor B's id are used to create the verification URI used in the above step 4. If Actor A wants to provide verification (because B's mini-application requires or recommends it), it MUST, before sending the initial POST request, make sure that a GET request to its 'baseuri'/'ownid'/trust/'trustlevel'/'actorB-id' will be responded to with a 200 OK (step 1 in table above). The GET request MUST be authenticated using the shared secret that actor A will send in the POST request to B as bearer token, thus ensuring that only actor B can request the URI.

```
Example:

Request to server `*http://myserver.org/* <http://myserver.org/>`__ GET
/app2/e41f4aae-4dee-10d0-b725-0af0a413bcf2/trust/friend/f81d4fae-7dec-11d0-a765-
↪00a0c91e6bf6

200 Ok
```

Note that by using https both in the initial trust creation and for the verification, one can ensure that the root URIs of each of the actors are authentic (through https session setup and certificate validation). With appropriately loaded root certificates on each web server hosting the actors, one can also selectively accept a group of servers and thus actors. An actor may also choose to verify a trust relationship on criteria or methods outside the scope of this specification.

Notification of Result

Once actor B has finalised the request for a relationship and if it returned a 202 Accepted to actor A when receiving the initial POST request, actor B MUST send a POST request to the same URI used for verification with a application/json body containing the attribute approved=True.

Updating a Trust Relationship

The actor owning the trust relationship can update the relationship by sending a PUT request to the relationship URI with a application/json body with attribute/value pairs of the attributes that are to be changed. Normally, only baseuri, desc, and approved are the attributes that can be changed.

Reading Trust Relationship Data

'admin' relationships and the 'creator' user MUST be allowed to read the trust relationship data through an authenticated GET to /myapp/'actorid'/trust/friend/'friendid'. Also the shared secret SHOULD be readable to actors with these two relationships. If the shared secret is not readable, the actor may not be able to support versioning/migration of actors.

```
Example:

Request to server `*http://actingweb.net/* <http://actingweb.net/>`__
GET
/myapp/f81d4fae-7dec-11d0-a765-00a0c91e6bf6/trust/friend/e41f4aae-4dee-10d0-b725-
↪0af0a413bcf2

200 OK

{
  "secret": "66b6691aae69fb75919e754976a8e2eb6d2719ac",
  "verified": true,
  "peerid": "e41f4aae-4dee-10d0-b725-0af0a413bcf2",
  "relationship": "friend",
  "baseuri": "https://actingweb.net/myotherapp/e41f4aae-4dee-10d0-b725-0af0a413bcf2",
  "desc": "Service subscription of monthly $29.95 for the Geekly Review magazine. ",
  "peer_approved": false,
  "type": "urn:actingweb:actingweb.net:myotherapp",
  "id": "f81d4fae-7dec-11d0-a765-00a0c91e6bf6",
  "approved": true
}
```

Verification and Approval

There are two mechanisms for validating a trust requests. As seen in the above example, there are three attributes stored for each trust relationship: *verified*, *peer_approved*, and *approved*. Verification is done as part of the initial trust request where the receiving actor does a request back to the requesting actor on the requesting actor's trust URL. The shared secret and a verification token are used as mutual authentication to verify that the two actors have a clear, trusted communication channel. The requesting actor is assumed to implicitly have verified (or pre-verified, see below) the actor before sending a trust request and it's verified is true as default.

The approval is an explicit mutual approval that can happen automatically based on some criteria (including passed verification) or through an external process either through a trustee or a human. The *approved* attribute is set when the local actor has approved, *peer_approved* is set when the peer actor has approved the relationship.

4.5.5 Initiating a trust request

In some cases it is necessary for an external actor or maybe the creator to request an actor to initiate a trust relationship. This is done through a POST request sent to <actorid>/trust with an application/json body specifying the url of the actor to initiate with and the trust level to request.

```
Example:

{
  "url": "https://actingweb.net/myapp/3973895dbe8457f68cdee59b0810d70a",
  "relationship": "friend"
}
```

If successful, a 201 Created should be returned with an application/json body equal to the body in a GET request to the location of the new relationship. Also, a Location header with the newly created trust relationship MUST be returned.

The actor MAY choose to accept a *type* attribute as a mandatory type that the actor responding to the URL should have. In any case, the actor MAY implement a request to the peer actor to-be's /meta path to verify type, response, and get information about actor capabilities.

4.5.6 Types of Relationships

When a new relationship is requested by an actor, the relationship has a type as specified in the URL path (i.e. /trust/friend/…). The actor receiving the request supports a number of trust relationship types. The mini-application's definition SHOULD define exactly what a 'friend' relationship means in terms of access. For example, one mini-application may allow 'friends' to update most of its properties, while another mini-application may only allow 'friends' to read parts of the properties.

In this section, a small number of relationship types are specified. It is RECOMMENDED that mini-applications use these relationship types (and fill them with own meaning), but only the 'admin' relationship type (besides the 'creator' user) MUST be supported. If the mini-application requires more relationship types, it MAY define new relationship types and specify these in its definition. The guidelines for how to use the specified relationship types should thus be used as a recommendation. It is also conceivable that mini-applications allow actor instances to dynamically change what each trust relationship gives access to. Such dynamic access rights SHOULD be documented in the mini-application definition, but is beyond the scope of this specification.

Below is an overview of the trust relationships. There are only three regular trust relationships meant for access to mini-application functionality/operations: associate, friend, and partner. The proxy and admin relationships have special uses. An actor MAY NOT hold more than one relationship with a given other actor.

Regular relationship name		**Description**
associate	OPTIONAL	The lowest level of trust (apart from no relationship). An actor will likely have the most relationships of this type and it will normally give read access.
friend	OPTIONAL	The friend relationship allows more access than for 'associates'. The friend actor has typically been verified to be friendly and can thus be allowed to write and/or request actions/methods.
partner	OPTIONAL	A partner is more than a friend. A partner actor may be stronger verified than a friend and can be trusted to the most important properties and actions/methods of the actor.
proxy	OPTIONAL	A proxy actor MUST be of the same mini-application type. A subscription from the agent to the proxy on /properties SHOULD be set up to ensure that the the proxy's properties are up to date. If the proxy only is updated through a client and based on client POSTs/PUTs, this is not necessary.
admin	OPTIONAL	The admin relationship has full access to an actor's data and functions similar to the creator user.

The Associate Relationship

The associate relationship is the lowest relationship level. The rights given to this relationship are dependent on the mini-application. Validation of an associate may be limited as the damage an associate can do should be very limited. Normally, an associate should not be able to affect the operations of the actor beyond the scope of its own relationship.

For example, a mini-application made to hold one person's contact information may allow read access to the person's basic contact information to anybody without a trust relationship, while an associate may get access to the full business contact information. The associate relationship can be used for creating networks of actors sharing information. Ex. a two-way associate relationship between two actors with contact information may allow a person to keep a real-time updated address book if each actor subscribes to the others' contact information of interest.

Use of the associate relationship MAY be allowed either unapproved (i.e. no explicit approbal) or auto-approved (possibly in combination with lists of pre-approved base URIs).

The Friend Relationship

The friend relationship is an intermediate step from associate to a partner relationship. The friend actor should be trusted to affect the basic operations of an actor. The validation before approving a new relationship should thus be appropriate for such access. Ex. a friend may be allowed to ask the actor to initiate a new trust relationship with another contact information actor. This would allow a trusted contact to introduce a new friend.

The Partner Relationship

The partner relationship is the most trusted level that gives access to actor operations. The partner may be trusted access to some or most of the inner workings of the actor. Ex. a partner may be allowed to update the actor’s contact information. This relationship can then be used to implement synchronisation between different sources of a person’s contact information. In particular, more dynamic contact information like presence can then be updated. If each presence source (like a mobile phone’s current profile or Messenger/AIM/Skype status) has an actor representing the presence status, the contact information actor can have partner relationships with each presence actor and allow presence information to be consolidated in one place.

The Proxy Relationship

The proxy relationship is a reserved relationship between two actors of the same mini-application type. A proxy relationship MUST always be two-ways such that if one relationship fails, the other MUST be removed. In a proxy relationship, the /properties access path SHOULD always be synchronised through a one-way or two-way subscription (dependent on application). These two subscriptions SHOULD be set up at the time of creation of the proxy relationship. Once set up, the two actors act as one entity to implement the mini-application’s functionality. See also the section on the proxy actor.

The Admin Relationship

The admin relationship has full access to all data and functions of an actor. See also the section on trustees as an alternative to using an admin relationship as this approach allows access to managing relationships without getting full access to the actor.

An admin relationship can also delete an actor (take it completely out of action).

The Creator Special User

The creator user can only be established at the point of instantiating an actor. The creator relationship has two primary objectives:

- other actors can instantiate a new actor and immediately administrate the new actor
- the creator credentials can be either supplied by a human user when instantiating the actor or conveyed to a user and thus allow the user to initialise the actor using the /www/init form or the web front-end supplied by the actor at /www

The creator MUST have all access rights similar to the ‘admin’ relationship. A creator is a user authenticated with username and password, not a trust relationship. The default user name for a creator is ‘creator’ unless another username is provided as part of the instantiation of the actor.

Assigning Individual Rights To An Instance Of A Trust Relationship

A mini-application MAY implement assigning of rights to a specific instance of a trust relationship (i.e. rights per actor). This allows granular access control. For example allowing home contact information to be available to some actors, while others only get access to business information. Whether this granular access control is implemented assigning a new type of trust relationship to a defined access group or on an individual basis is outside the scope of this specification. The same holds for how to determine which relationship to get upgraded or reduced rights.

4.5.7 Using a Trustee To Manage Relationships

The Use of Trustees

Managing trust relationship request can be one of actor's the most complex operations and may require validation of the requesting actor's identity, seeking approval from a human or applying some other logic to evaluate the request. Also, managing trust relationships over time requires maintenance actions, for example deleting a relationship after a certain time period or on a human's request. To simplify this process, another (possibly dedicated) actor called a 'trustee' can take over the relationship management.

A trustee actor can take many forms, two examples may be:

1. A web application that presents a web page to a user and allows the user to approve or reject the request. The user can be authenticated using for example the Google Account API to validate Google credentials for the Google account stored as the trustee's "owner".

2. A desktop application that graphically and dynamically depicts relationships between actors and that allows a human being to create new relationships through connecting and disconnecting icon representations using drag and drop.

For example, a desktop application may allow a user to choose a bank account as a focal point and then visualise drawn lines to actors that represent subscription services that are allowed to withdraw money from the bank account. Hoovering over the relationship line can show details about the relationship, for example restrictions on amount and how often money is allowed to be transferred.

Assigning and Forwarding To a Trustee

A trustee can be assigned either at the instantiation of an actor (by specifying trustee root URI on instantiation in the application/json body using the attribute 'trustee_root') or by changing the trustee's root URI by PUTing a new URI to /trust/trustee (an admin or creator relationship is required) with application/json content and the URL in the attribute 'trustee_root'. In the same PUT, the 'creator' attribute MUST also be available to allow changing creator to 'trustee'. The 'creator' user's passphrase/secret MAY then be used as a bearer token to do trustee REST requests if 'creator' is set to 'trustee'. As a passphrase is supplied at instantiation time for creator, it is RECOMMENDED that the passphrase is checked for security strength if the trustee_root attribute is set.

However, the token MUST NOT give access as bearer token in requests unless the 'creator' user is 'trustee' and the bit strength of the passphrase is > 80 bits.

A trustee MUST have access to the /trust and /subscription endpoints, as well as the root (like creator) using the passphrase as bearer token. It is RECOMMENDED that any other endpoint access is done and accepted only with a regular trust relationship.

It is RECOMMENDED that a trustee also establishes a regular relationship with the actor (of any type). A mini-application MUST make this a requirement before accepting any trustee access with a bearer token as a trust relationship allows verification and possibly explicit approval through an external process to set a trustee. (Note that if trustee_root is set as part of an instantiation, the creator username and password can be used to send requests as creator to the new actor even before trustee access with passphrase as bearer token is allowed.)

A typical use of a trustee is automatic creation of a new actor by another actor to create an ActingWeb network of actors. Let's say a bot instantiating a new text messaging actor before handing over to the user to authenticate the new actor with the 3rd party text messaging service. The actor instantiating a new actor may then initiate a new trust relationship and then use the creator credentials to approve the relationship directly.

4.5.8 Authentication

Authentication of an actor is necessary when the actor is sending a request to another actor where a trust relationship is required to gain the required access. At the time of creating a trust relationship, the actor receiving and approving the request MUST store the actor's id, base URI, and the relationship's shared secret in its authentication store.

The authentication method is based on the http Bearer authentication method. Please refer to https://tools.ietf.org/html/rfc6750 for details on the use of bearer tokens.

Note that the passphrase created at actor instantiation time together with the (default) username 'creator' MUST use http basic authentication.

4.6 /subscriptions - Subscriptions (OPTIONAL)

4.6.1 Establishing a Subscription

The subscription model is based on the concept of creating a subscription on a specific actor path with three elements: target, sub-target, and resource. The most narrow subscription is on a resource. Sub-target and target are then mandatory specified in the subscription request. If a subscription targets all resources within a sub-target, then sub-target and target are specified in the request. Finally, if the entire endpoint is targeted in the subscription, only target is specified in the request.

The /properties endpoint allows easy sharing of simple json data, while the endpoints /actions, /resources, /methods, and /sessions enable sharing of more complex data and execution of procedures. Once an actor has a trust relationship, it can use subscriptions to receive updates about changes and activities. The main use of a subscription is to make sure that one actor gets efficient and near real-time access to changes that occur to another actor.

An actor can establish a new subscription by sending a peer authenticated POST request to /subscriptions/<peerid> where peerid is the actor id of the requesting actor. A json body MUST as a minimum include the attribute "target". The most common targets are properties, resources, actions, methods, and sessions, but a subscription MAY be of any target the mini-app will recognise. Additionally, "subtarget" and "resource" can be specified, thus identifying a sub-path to subscribe to, e.g. /resources/folders/12345.

The "granularity" attribute controls how the subscribing actor wants to be notified. The possible values are "high", "low", and "none", where "high" sends a callback to the subscribing actor with a full application/json body of the update, "low" sends a notification with an application/json body with a single attribute "url" with a URL to where a full json body of the update can be retrieved, and "none" suppresses all notifications and GET polling on the subscription is necessary to retrieve updates. If "granularity" is not present, "none" is assumed.

The requesting actor MUST have a relationship allowing minimum read permissions on the requested path, and the actor receiving the request MUST validate the access rights upon receiving the request. It MUST also validate that the bearer token is valid for the peer specified in the POST URI.

If approved, a 201 Created MUST be returned with the Location header pointing to the relative URI of the newly created subscription in the format /<actorid>/subscriptions/<subscriber-actor-id>/<subscription-id>.

In the below example, a new subscription with id 9d60853cb4915b699f89d7ae13efb382 is created at actor (requested) b373e63030a451b2991c9995438fccf1 from actor (requesting) f08ce818ea515526adcbd157eeaf0ab0.

```
Example:

POST
/b373e63030a451b2991c9995438fccf1/subscriptions/f08ce818ea515526adcbd157eeaf0ab0

{
  "target": "resources",
  "subtarget": "folders",
  "resource": "12345",
  "granularity": "high"
}

201 Created

Location:
/b373e63030a451b2991c9995438fccf1/subscriptions/f08ce818ea515526adcbd157eeaf0ab0/
↪9d60853cb4915b699f89d7ae13efb382
```

(continues on next page)

(continued from previous page)

An actor is RECOMMENDED to support callbacks, but MUST support polling of subscriptions. If a callback is requested by specifying "granularity" other than "none", but it is not supported by the mini-application, a 501 Not Implemented MUST be returned. The requesting actor may then choose to resend the subscription request without a granularity attribute or {"granularity": "none"}.

When a subscription has been registered, the actor MUST record and act upon any changes that are done to the path. The actor MAY reject a subscription request to any path on which it does not support subscriptions. A subscription to /actions and /methods indicates a wish to get updates on any requests to these paths.

4.6.2 Initiating and Enumerating Subscriptions

Sometimes it can be necessary to request an actor to create a subscription with another actor. Actors supporting subscriptions MUST support POST requests to /<actorid>/subscriptions with an application/json body with "peerid" identifying the peer to send a subscription request to, as well as all the attributes to use when requesting a subscription.

It is RECOMMENDED that a mini-application supports GET on /subscriptions and /subscriptions/<peerid> with creator or admin rights and peer rights respectively. The body returned with 200 Ok MUST contain the below elements. See the next section for explanation of the sequence attribute.

The below example is for a GET to subscriptions. A GET to subscriptions/<peerid> does not include the "peerid" attributes for each data element, but rather on the root level.

```
GET /b373e63030a451b2991c9995438fccf1/subscriptions

{
  "id": "b373e63030a451b2991c9995438fccf1",
  "data": [
    {
      "peerid": "f08ce818ea515526adcbd157eeaf0ab0",
      "resource": "",
      "target": "properties",
      "sequence": 1,
      "granularity": "high",
      "subtarget": "data2",
      "subscriptionid": "6f9c496966d35b3b9d3fa2c9efc2934a"
    },
    {
      "peerid": "794edccea3705c0c96defb13857374ae",
      "resource": "45",
      "target": "properties",
      "sequence": 1,
      "granularity": "low",
      "subtarget": "data3",
      "subscriptionid": "016edb81538351b0af3034a6a751b003"
    }
  ]
}
```

A mini-app MAY support GET search parameters like *?target=<value>&subtarget=<value2>&resource=<value3>*.

4.6.3 Getting Subscription Updates

The Diff and Update Model

The hierarchy of subscriptions is important for how the subscription callbacks and diffs work:

```
|... ourtarget |... sub-target1 |... resource1
             |              |... resource2
             |              |... resource3
             |... sub-target2 |... resource1
                            |... resource2
```

Let's start with the simplest case, the subscription targets *resource1* of *sub-target1* in *ourtarget*. The data in the body of an update MUST in this case be the full resource representation (json or other data representation) and not a diff. In fact, the data MAY be anything the mini-app chooses to send as an update. It can be a blob or a data object that has nothing to do with the previous subscription update, as long as it communicates a state change on the resource, and the data representation is documented by the mini-app.

This flexibility allows the subscription mechanism to be used for any resource state change that is desired to be communicated to another actor.

For subscriptions to a target or a sub-target, the expectation is that the changes per update can be found in a json struct following the hierarchy relative to the subscription. I.e. If resource2 in sub-target2 has changed, the diff MUST follow the following example format for a subscription to *ourtarget*:

```
{
  "sub-target2": {
    "resource2": "full_representation"
  }
}
```

and the following for a subscription to *ourtarget*, *sub-target2*:

```
{
  "resource2": "full_representation"
}
```

This means that every time something happens on a resource/endpoint/path, all subscriptions within scope MUST be processed and a specific diff created for each subscription where the json struct follows the hierarchy below the subscription. The diff format is simple: only sub-targets or resources that have changed are included in a diff, and an empty attribute is defined as the same as a non-existent attribute, thus an attribute can be deleted by setting it to "".

Each diff MUST have a sequence number, where the number is 1 for each new subscription and then incremented by 1 for each diff that is created for this subscription.

Further examples with /properties

Initial data POSTed to /properties:

```
{
  "data1": {
    "str1": "initial",
    "str2": "initial"
  },
    "data2": "initial",
    "test": {
      "var1": "initial",
      "var2": "initial",
      "resource": "initial"
  }
}
```

If a PUT to /properties/test has the content { "var1": "hey"} (overwriting test), the resulting /properties will be:

```
{
  "data1": {
    "str1": "initial",
    "str2": "initial"
  },
  "data2": "initial",
  "test": {
    "var1": "hey"
  }
}
```

and the diff to a subscription on /properties MUST be:

```
"test": {
  "var1": "hey"
}
```

And a subscription to /properties/test MUST give the following diff:

```
{
  "var1": "hey"
}
```

If instead a PUT is done to /properties/test/var1 to change it to “change2”, the subscription diff for a /properties subscription will be:

```
"test": {
  "var1": "change2"
}
```

Note here that the resource and var2 attributes in the test struct have not been touched and are thus still present in test if you do a GET on /properties. Compare this to the above example overwriting of the entire test sub-target (with { “var1”: “hey” }), and you can see that there is no way to determine if resources below test were deleted by just looking at the diff. This information can be forced into a diff by doing a DELETE on a /properties path (which MUST send a “resource”: “” as a diff).

A mini-app MAY implement more advanced diff methods, but this diff method MUST be used for the /properties endpoint.

Getting the Updates

There are two ways to get updates on a subscription: polling on the subscription URI or receive callbacks on the subscription callback URI as specified in the previous section.

If the granularity is set to high, the callback is a POST with peer bearer authentication to the subscription URI with the following format:

```
{
  "id": "26597c23469e5ab2b3489b786cf553f5",
  "target": "properties",
  "sequence": 6,
  "timestamp": "2016-11-15T13:47:02.375880Z",
  "granularity": "high",
  "subscriptionid": "552a0d7ec4ab553ea6c912baeb4459eb",
  "data": {
    "test": {
      "resource": "change5"
    }
```

(continues on next page)

(continued from previous page)

```
  }
}
```

The data attribute contains the actual diff relative to the subscription, while the other attributes identifies the actor id originating the update, the target (and subtarget and resource if relevant), the timestamp in UTC, the granularity of the subscription, the subscriptionid, as well as the sequence number of the update. In this case, 6 indicates that this is the 6th diff produced for this specific subscription as identified by subscriptionid.

A peer bearer token authenticated GET on the subscription id MUST offer a similar format, but here an array of data elements MUST represent one or more diff updates. In the below example, the first diff was retrieved earlier, now diff 2 - 6 are collected.

```
{
  "subscriptionid": "26f9bb085fb351c7ba40d395af18a381",
  "id": "26597c23469e5ab2b3489b786cf553f5",
  "resource": "",
  "target": "properties",
  "subtarget": "test",
  "data": [
    {
      "timestamp": "2016-11-15T13:46:51.906980Z",
      "data": "change1",
      "sequence": 2
    },
    {
      "timestamp": "2016-11-15T13:46:53.967370Z",
      "data": "change2",
      "sequence": 3
    },
    {
      "timestamp": "2016-11-15T13:46:57.012820Z",
      "data": "change3",
      "sequence": 4
    },
    {
      "timestamp": "2016-11-15T13:47:00.509530Z",
      "data": {
        "some": "data",
        "resource": "change4"
      },
      "sequence": 5
    },
    {
      "timestamp": "2016-11-15T13:47:03.059410Z",
      "data": {
        "resource": "change5"
      },
      "sequence": 6
    }
  ]
}
```

In the case of callbacks to the subscription callback URI, any 2xx response from the actor will indicate that the update has been received, and it MUST be cleared, i.e. not available anymore when doing a GET on the subscription.

If the callback is not cleared with a 2xx response, the actor SHOULD not retry, and it SHOULD be up to the receiving actor to do a GET poll when it detects a gap in the sequence or triggered by another event. An actor MAY implement a re-try mechanism with an exponential back-off.

Low Granularity and Polling

In the case of granulerity = low, the callback to the subscription callback URI MUST contain application/json content with the following format:

```
{
  "subscriptionid": "26f9bb085fb351c7ba40d395af18a381",
  "id": "26597c23469e5ab2b3489b786cf553f5",
  "target": "properties",
  "subtarget": "test",
  "timestamp": "2016-11-15T13:47:02.915030Z",
  "granularity": "low"
  "sequence": 6,
  "url": "https://actingweb.net/myapp/26597c23469e5ab2b3489b786cf553f5/subscriptions/
↪e3c47b6114ec558dade20d6c45855820/26f9bb085fb351c7ba40d395af18a381/6",
}
```

The data element is here replaced by a url attribute identifying where the subscribing actor can retrieve a specific diff. It is RECOMMENDED that the URL is composed of the subscription URL with the sequence number trailing at the end.

The response content on this URL MUST be similar to a GET on the subscription following this example:

```
{
  "subscriptionid": "26f9bb085fb351c7ba40d395af18a381",
  "id": "26597c23469e5ab2b3489b786cf553f5",
  "target": "properties",
  "subtarget": "test",
  "resource": "",
  "timestamp": "2016-11-15T13:46:48.893650Z",
  "sequence": 6,
  "data": {
    "resource": "initial",
    "var1": "initial",
    "var2": "initial"
  }
}
```

Clearing Callback Status

Subsequent GET requests to the same URL MUST give the same result, both requests to the subscription as well as to the specific diff URL, except that the subscription URL MUST return new diffs.

When a GET is done either to the subscription URL or to a specific diff and the subscribing actor wants to clear all diffs received, a PUT MUST be sent to the subscription URL with an application/json body with one attribute “sequence” and the sequence number that was the last sequence retrieved. All diffs with lower and equal sequence number than this MUST be cleared and no longer appear in GET requests to the subscription URL. GET requests to the diff specific URL (subscription URL + ‘/seqnr’) MUST return 404 Not found after clearing has been done.

```
PUT
/myapp/26597c23469e5ab2b3489b786cf553f5/subscriptions/
↪e3c47b6114ec558dade20d6c45855820/26f9bb085fb351c7ba40d395af18a381

{
  “sequence”: 2
}
```

(continues on next page)

(continued from previous page)

```

204 No content
```

4.7 Security Considerations

There are many security considerations, some are well known for any http-based communication, while some are specific to ActingWeb, in particular the trust model.

The use of https is a MANDATORY requirement for any non-trivial implementation. Some of the validation of peers MAY be based on validation of server-side certificates. This require proper configuration of the web server hosting the mini-app.

Validation and approval of a relationship MAY be based on three elements: 1. Root host of the requesting actor (confirmed by TLS certificate) 2. Callback verification 3. Explicit approval list, static or dynamic, of peer root URL 4. Manual approval

The use of a bearer token in the http Authorization header is subject to the same security concerns as OAuth2 Bearer tokens.

Care must be taken to make sure that the bearer token is sufficiently random and long enough for the security level required.

There is no expiry of ActingWeb peer tokens. However, the relationship can be terminated at any time and thus offers a revocation method. Also, a mini-app MAY enforce automatic expiry of relationships and thus force a new trust establishment flow.

Use of HTTP methods

Some firewalls and proxies may filter http methods like PUT and DELETE and may only allow GET and POST. Also, some application frameworks or browsers may not support other methods. Wherever a method other than GET and POST is specified in this document, the parameter _method='true_method' MUST be supported and the http header X-HTTP-Method-Override: 'true_method' MAY be supported as alternative ways to specify the intended http method.

4.8 IANA Considerations

Currently, ActingWeb.org controls the namespaces that IANA might be responsible for. Dependent on how this specification is moved forward, that may or may not change.

4.9 References and Endnotes

CHAPTER 5

actingweb code

5.1 Subpackages

5.1.1 actingweb.db_dynamodb package

Submodules

actingweb.db_dynamodb.db_actor module

actingweb.db_dynamodb.db_attribute module

actingweb.db_dynamodb.db_peertrustee module

actingweb.db_dynamodb.db_property module

actingweb.db_dynamodb.db_subscription module

actingweb.db_dynamodb.db_subscription_diff module

actingweb.db_dynamodb.db_trust module

5.1.2 actingweb.handlers package

Submodules

actingweb.handlers.base_handler module

actingweb.handlers.bot module

actingweb.handlers.callback_oauth module

actingweb.handlers.callbacks module

actingweb.handlers.devtest module

actingweb.handlers.factory module

actingweb.handlers.meta module

actingweb.handlers.oauth module

actingweb.handlers.properties module

actingweb.handlers.resources module

actingweb.handlers.root module

actingweb.handlers.subscription module

actingweb.handlers.trust module

actingweb.handlers.www module

5.2 Submodules

5.3 actingweb.actor module

class `actingweb.actor.`**`Actor`**(*actor_id=None*, *config=None*)

Bases: `object`

`callback_subscription`(*peerid=None*, *sub_obj=None*, *sub=None*, *diff=None*, *blob=None*)

`create`(*url*, *creator*, *passphrase*, *actor_id=None*, *delete=False*)

“Creates a new actor and persists it.

If delete is True, any existing actors with same creator value will be deleted. If it is False, the one with the correct passphrase will be chosen (if any)

`create_reciprocal_trust`(*url*, *secret=None*, *desc=”*, *relationship=”*, *trust_type=”*)

Creates a new reciprocal trust relationship locally and by requesting a relationship from a peer actor.

`create_remote_subscription`(*peerid=None*, *target=None*, *subtarget=None*, *resource=None*, *granularity=None*)

Creates a new subscription at peerid.

`create_subscription`(*peerid=None*, *target=None*, *subtarget=None*, *resource=None*, *granularity=None*, *subid=None*, *callback=False*)

`create_verified_trust`(*baseuri=”*, *peerid=None*, *approved=False*, *secret=None*, *verification_token=None*, *trust_type=None*, *peer_approved=None*, *relationship=None*, *desc=”*)

Creates a new trust when requested and call backs to initiating actor to verify relationship.

`delete`()

Deletes an actor and cleans up all relevant stored data

`delete_peer_trustee`(*shorttype=None*, *peerid=None*)

delete_properties()
Deletes all properties.

delete_property(*name*)
Deletes a property name. (DEPRECATED, use actor's property store!)

delete_reciprocal_trust(*peerid=None*, *delete_peer=False*)
Deletes a trust relationship and requests deletion of peer's relationship as well.

delete_remote_subscription(*peerid=None*, *subid=None*)

delete_subscription(*peerid=None*, *subid=None*, *callback=False*)
Deletes a specified subscription

get(*actor_id: str = None*) → dict
Retrieves an actor from storage or initialises if it does not exist

get_from_creator(*creator=None*)
Initialise an actor by matching on creator.

If unique_creator config is False, then no actor will be initialised. Likewise, if multiple properties are found with the same value (due to earlier uniqueness off).

get_from_property(*name='oauthId'*, *value=None*)
Initialise an actor by matching on a stored property.

Use with caution as the property's value de-facto becomes a security token. If multiple properties are found with the same value, no actor will be initialised. Also note that this is a costly operation as all properties of this type will be retrieved and proceessed.

get_peer_info(*url: str*) → dict
Contacts an another actor over http/s to retrieve meta information :param url: Root URI of a remote actor :rtype: dict :return: The json response from the /meta path in the data element and last_response_code/last_response_message set to the results of the https request :Example:

>>>{ >>> "last_response_code": 200, >>> "last_response_message": "OK", >>> "data":{} >>>}

get_peer_trustee(*shorttype=None*, *peerid=None*)
Get a peer, either existing or create it as trustee

Will retrieve an existing peer or create a new and establish trust. If no trust exists, a new trust will be established. Use either peerid to target a specific known peer, or shorttype to allow creation of a new peer if none exists

get_properties()
Retrieves properties from db and returns a dict.

get_property(*name*)
Retrieves a property object named name. (DEPRECATED, use actor's property store!)

get_subscription(*peerid=None*, *subid=None*, *callback=False*)
Retrieves a single subscription identified by peerid and subid.

get_subscription_obj(*peerid=None*, *subid=None*, *callback=False*)
Retrieves a single subscription identified by peerid and subid.

get_subscriptions(*peerid=None*, *target=None*, *subtarget=None*, *resource=None*, *callback=False*)
Retrieves subscriptions from db.

get_trust_relationship(*peerid=None*)

get_trust_relationships(*relationship="*, *peerid="*, *trust_type="*)
Retrieves all trust relationships or filtered.

modify(*creator=None*)

modify_trust_and_notify(*relationship=None*, *peerid=None*, *baseuri=”*, *secret=”*, *desc=”*, *approved=None*, *verified=None*, *verification_token=None*, *peer_approved=None*)
Changes a trust relationship and noties the peer if approval is changed.

register_diffs(*target=None*, *subtarget=None*, *resource=None*, *blob=None*)
Registers a blob diff against all subscriptions with the correct target, subtarget, and resource.

If resource is set, the blob is expected to be the FULL resource object, not a diff.

set_property(*name*, *value*)
Sets an actor’s property name to value. (DEPRECATED, use actor’s property store!)

class `actingweb.actor.`**Actors**(*config=None*)
Bases: `object`

Handles all actors

fetch()

class `actingweb.actor.`**DummyPropertyClass**(*v=None*)
Bases: `object`

Only used to deprecate get_property() in 2.4.4

5.4 actingweb.attribute module

class `actingweb.attribute.`**Attributes**(*actor_id=None*, *bucket=None*, *config=None*)
Bases: `object`

Attributes is the main entity keeping an attribute.

It needs to be initalized at object creation time.

delete_attr(*name=None*)

delete_bucket()
Deletes the attribute bucket in the database

get_attr(*name=None*)
Retrieves a single attribute

get_bucket()
Retrieves the attribute bucket from the database

set_attr(*name=None*, *data=None*, *timestamp=None*)
Sets new data for this attribute

class `actingweb.attribute.`**Buckets**(*actor_id=None*, *config=None*)
Bases: `object`

Handles all attribute buckets of a specific actor_id

Access the attributes in .props as a dictionary

delete()

fetch()

fetch_timestamps()

class actingweb.attribute.**InternalStore**(*actor_id=None*, *config=None*, *bucket=None*)

Bases: object

Access to internal attributes using .prop notation

5.5 actingweb.auth module

class actingweb.auth.**Auth**(*actor_id*, *auth_type='basic'*, *config=None*)

Bases: object

The auth class handles authentication and authorisation for the various schemes supported.

The helper function init_actingweb() can be used to give you an auth object and do authentication (or can be called directly). The check_authentication() function checks the various authentication schemes against the path and does proper authentication. There are three types supported: basic (using creator credentials), token (received when trust is created), or oauth (used to bind an actor to an oauth-enabled external service, as well as to log into /www path where interactive web functionality of the actor is available). The check_authorisation() function validates the authenticated user against the config.py access list. check_token_auth() can be called from outside the class to do a simple peer/bearer token verification. The OAuth helper functions are used to: process_oauth_callback() - process an OAuth callback as part of an OAuth flow and exchange code with a valid token validate_oauth_token() - validate and, if necessary, refresh a token set_cookie_on_cookie_redirect() - set a session cookie in the browser to the token value (called AFTER OAuth has been done!)

The response[], acl[], and authn_done variables are useful outside Auth(). authn_done is set when authentication has been done and a final authentication status can be found in response[].

> self.response = {
>
> > "code": 403, # Result code (http) "text": "Forbidden", # Proposed response text "headers": [], # Headers to add to response after authentication has been done

}

self.acl = {

> "authenticated": False, # Has authentication been verified and passed? "authorised": False, # Has authorisation been done and appropriate acls set? "rights": '', # "a", "r" (approve or reject) "relationship": None, # E.g. creator, friend, admin, etc "peerid": '', # Peerid if there is a relationship "approved": False, # True if the peer is approved

}

check_authentication(*appreq*, *path*)

Checks authentication in appreq, redirecting back to path if oauth is done.

check_authorisation(*path=''*, *subpath=''*, *method=''*, *peerid=''*, *approved=True*)

Checks if the authenticated user has acl access rights in config.py.

Takes the path, subpath, method, and peerid of the path (if auth user is different from the peer that owns the path, e.g. creator). If approved is False, then the trust relationship does not need to be approved for access

check_token_auth(*appreq*)

Called with an http request to check the Authorization header and validate if we have a peer with this token.

oauth_delete(*url=None*)

Used to call DELETE from the attached oauth service.

Uses oauth.delete_request((), but refreshes token if necessary. The function fails if token is invalid and no refresh is possible. For web-based flows, validate_oauth_token() needs to be used to validate token and get redirect URI for new authorization flow.

oauth_get (*url=None*, *params=None*)
Used to call GET from the attached oauth service.

Uses oauth.get_request(), but refreshes token if necessary. The function fails if token is invalid and no refresh is possible. For web-based flows, validate_oauth_token() needs to be used to validate token and get redirect URI for new authorization flow.

oauth_head (*url=None*, *params=None*)
Used to call HEAD from the attached oauth service.

Uses oauth.head_request((), but refreshes token if necessary. The function fails if token is invalid and no refresh is possible. For web-based flows, validate_oauth_token() needs to be used to validate token and get redirect URI for new authorization flow.

oauth_post (*url=None*, *params=None*, *urlencode=False*)
Used to call POST from the attached oauth service.

Uses oauth.post_request(), but refreshes token if necessary. The function fails if token is invalid and no refresh is possible. For web-based flows, validate_oauth_token() needs to be used to validate token and get redirect URI for new authorization flow.

oauth_put (*url=None*, *params=None*, *urlencode=False*)
Used to call PUT from the attached oauth service.

Uses oauth.put_request(), but refreshes token if necessary. The function fails if token is invalid and no refresh is possible. For web-based flows, validate_oauth_token() needs to be used to validate token and get redirect URI for new authorization flow.

process_oauth_callback (*code*)
Called when a callback is received as part of an OAuth flow to exchange code for a bearer token.

set_cookie_on_cookie_redirect (*appreq*)
Called after successful auth to set the cookie with the token value.

validate_oauth_token (*lazy=False*)
Called to validate the token as part of a web-based flow.

Returns the redirect URI to send back to the browser or empty string. If lazy is true, refresh_token is used only if < 24h until expiry.

actingweb.auth.**add_auth_response** (*appreq=None*, *auth_obj=None*)
Called after init_actingweb() if add_response was set to False, and now responses should be added.

actingweb.auth.**init_actingweb** (*appreq=None*, *actor_id=None*, *path="*, *subpath="*, *add_response=True*, *config=None*)
Initialises actingweb by loading a config object, an actor object, and authentication object.

More details about the authentication can be found in the auth object. If add_response is True, appreq.response will be changed according to authentication result. If False, appreq will not be touched. authn_done (bool), response['code'], response['text'], and response['headers'] will indicate results of the authentication process and need to be acted upon. 200 is access approved 302 is redirect and auth_obj.redirect contains redirect location where response must be redirected 401 is authentication required, response['headers'] must be added to response 403 is forbidden, text in response['text']

actingweb.auth.**select_auth_type** (*path*, *subpath*, *config=None*)
Selects authentication type based on path and subpath.

Currently are only basic and oauth supported. Peer auth is automatic if an Authorization Bearer <token> header is included in the http request.

5.6 actingweb.aw_proxy module

class `actingweb.aw_proxy.`**`AwProxy`**(*trust_target=None*, *peer_target=None*, *config=None*)

Bases: `object`

Proxy to other trust peers to execute RPC style calls

Initialise with either trust_target to target a specific existing trust or use peer_target for simplicity to use the trust established with the peer.

`change_resource`(*path=None*, *params=None*)

`create_resource`(*path=None*, *params=None*)

`delete_resource`(*path=None*)

`get_resource`(*path=None*, *params=None*)

5.7 actingweb.aw_web_request module

class `actingweb.aw_web_request.`**`AWRequest`**(*url=None*, *params=None*, *body=None*, *headers=None*, *cookies=None*)

Bases: `object`

`arguments`()

`get`(*var=”*)

`get_header`(*header=”*)

class `actingweb.aw_web_request.`**`AWResponse`**

Bases: `object`

`set_cookie`(*name*, *value*, *max_age=1209600*, *path='/'*, *secure=True*)

`set_redirect`(*url*)

`set_status`(*code=200*, *message='Ok'*)

`write`(*body=None*, *encode=False*)

class `actingweb.aw_web_request.`**`AWWebObj`**(*url=None*, *params=None*, *body=None*, *headers=None*, *cookies=None*)

Bases: `object`

5.8 actingweb.config module

class `actingweb.config.`**`Config`**(***kwargs*)

Bases: `object`

static `new_token`(*length=40*)

static `new_uuid`(*seed*)

5.9 actingweb.oauth module

class actingweb.oauth.**OAuth**(*token=None*, *config=None*)

Bases: object

delete_request(*url*)

enabled()

get_request(*url*, *params=None*)

head_request(*url*, *params=None*)

oauth_redirect_uri(*state=''*, *creator=None*)

oauth_refresh_token(*refresh_token*)

oauth_request_token(*code=None*)

post_request(*url*, *params=None*, *urlencode=False*, *basic_auth=False*)

put_request(*url*, *params=None*, *urlencode=False*)

set_token(*token*)

actingweb.oauth.**pagination_links**(*self*)

Links parsed from HTTP Link header

5.10 actingweb.on_aw module

class actingweb.on_aw.**OnAWBase**

Bases: object

Base class to be extended to extend ActingWeb functionality

actions_on_oauth_success()

aw_init(*auth=None*, *webobj=None*)

bot_post(*path*)

Called on POSTs to /bot.

Note, there will not be any actor initialised.

check_on_oauth_success(*token=None*)

delete_actor()

delete_callbacks(*name*)

Customizible function to handle DELETE /callbacks

delete_properties(*path: list*, *old: dict*, *new: dict*) → bool

Called on DELETE to properties

Parameters

- **path** (*list[str]*) – Target path to be deleted
- **old** (*dict*) – Property value that will be deleted (or changed)
- **new** (*dict*) – Property value after path has been deleted

Returns True if DELETE is allowed, False if 403 should be returned

Return type bool

delete_resources (*name*)
Called on DELETE to resources. Return struct for json out.

Returning { } will give a 404 response back to requestor.

get_callbacks (*name*)
Customizible function to handle GET /callbacks

get_properties (*path: list*, *data: dict*) → dict
Called on GET to properties for transformations to be done

Parameters

- **path** (*list[str]*) – Target path requested
- **data** (*dict*) – Data retrieved from data store to be returned

Returns The transformed data to return to requestor or None if 404 should be returned

Return type dict or None

get_resources (*name*)
Called on GET to resources. Return struct for json out.

Returning { } will give a 404 response back to requestor.

post_callbacks (*name*)
Customizible function to handle POST /callbacks

post_properties (*prop: str*, *data: dict*) → dict
Called on POST to properties, once for each property

Parameters

- **prop** (*str*) – Property to be created
- **data** (*dict*) – The data to be stored in prop

Returns The transformed data to store in prop or None if that property should be skipped and not stored

Return type dict or None

post_resources (*name*, *params*)
Called on POST to resources. Return struct for json out.

Returning { } will give a 404 response back to requestor. Returning an error code after setting the response will not change the error code.

post_subscriptions (*sub*, *peerid*, *data*)
Customizible function to process incoming callbacks/subscriptions/ callback with json body, return True if processed, False if not.

put_properties (*path: list*, *old: dict*, *new: dict*) → dict
Called on PUT to properties for transformations to be done before save :param path: Target path requested to be updated :type path: list[str] :param old: Old data from database :type old: dict :param new: :type new: New data from PUT request (after merge) :return: The dict that should be stored or None if 400 should be returned and nothing stored :rtype: dict or None

put_resources (*name*, *params*)
Called on PUT to resources. Return struct for json out.

Returning { } will give a 404 response back to requestor. Returning an error code after setting the response will not change the error code.

www_paths (*path=''*)

5.11 actingweb.peertrustee module

class actingweb.peertrustee.**PeerTrustee**(*actor_id=None*, *peerid=None*, *short_type=None*, *peer_type=None*, *config=None*)

Bases: object

create(*baseuri=None*, *passphrase=None*)

delete()

get()

5.12 actingweb.property module

class actingweb.property.**Properties**(*actor_id=None*, *config=None*)

Bases: object

Handles all properties of a specific actor_id

Access the properties in .props as a dictionary

delete()

fetch()

class actingweb.property.**Property**(*actor_id=None*, *name=None*, *value=None*, *config=None*)

Bases: object

property is the main entity keeping a property.

It needs to be initalised at object creation time.

delete()

Deletes the property in the database

get()

Retrieves the property from the database

get_actor_id()

set(*value*)

Sets a new value for this property

class actingweb.property.**PropertyStore**(*actor_id=None*, *config=None*)

Bases: object

5.13 actingweb.subscription module

class actingweb.subscription.**Subscription**(*actor_id=None*, *peerid=None*, *subid=None*, *callback=False*, *config=None*)

Bases: object

Base class with core subscription methods (storage-related)

add_diff(*blob=None*)

Add a new diff for this subscription

clear_diff(*seqnr*)

Clears one specific diff

clear_diffs(*seqnr=0*)
Clear all diffs up to and including a seqnr

create(*target=None*, *subtarget=None*, *resource=None*, *granularity=None*, *seqnr=1*)
Create new subscription and push it to db

delete()
Delete a subscription in storage

get()
Retrieve subscription from db given pre-initialized variables

get_diff(*seqnr=0*)
Get one specific diff

get_diffs()
Get all the diffs available for this subscription ordered by the timestamp, oldest first

increase_seq()

class actingweb.subscription.**Subscriptions**(*actor_id=None*, *config=None*)
Bases: `object`

Handles all subscriptions of a specific actor_id

Access the indvidual subscriptions in .dbsubscriptions and the subscription data in .subscriptions as a dictionary

delete()

fetch()

5.14 actingweb.trust module

class actingweb.trust.**Trust**(*actor_id=None*, *peerid=None*, *token=None*, *config=None*)
Bases: `object`

create(*baseuri=”*, *peer_type=”*, *relationship=”*, *secret=”*, *approved=False*, *verified=False*, *verification_token=”*, *desc=”*, *peer_approved=False*)
Create a new trust relationship

delete()
Delete the trust relationship

get()
Retrieve a trust relationship with either peerid or token

modify(*baseuri=None*, *secret=None*, *desc=None*, *approved=None*, *verified=None*, *verification_token=None*, *peer_approved=None*)

class actingweb.trust.**Trusts**(*actor_id=None*, *config=None*)
Bases: `object`

Handles all trusts of a specific actor_id

Access the indvidual trusts in .dbtrusts and the trust data in .trusts as a dictionary

delete()

fetch()

CHAPTER 6

CHANGELOG

6.1 v2.6.5: Apr 22, 2021

- Fix bug in subscription_diff handling by replacing query with scan as query requires hash key

6.2 v2.6.4: Apr 11, 2021

- Messed up release versioning, bump up to avoid confusion

6.3 v2.6.3: Apr 11, 2021

- Fix bug in peertrustee handling by replacing dynamodb count() with scan() as count requires a hash key

6.4 v2.6.2: Oct 20, 2020

- Security fix on oauth refresh

6.5 v2.6.1: Aug 30, 2020

- Fix token refresh to also use Basic authorisation

6.6 v2.6.0: Aug 23, 2020

- Add support for optional Basic authorisation in token request (e.g. Fitbit is requiring this)

6.7 v2.5.1: Jan 29, 2019

- Move some annoying info messages to debug in auth/oauth
- Fix bug in set_attr for store where struct is not initialised (attribute.py:70)
- Enforce lower case on creator if @ (i.e. email) in value

6.8 v2.5.0: Nov 17, 2018

- BREAKING: /www/properties template_values now return a dict with { 'key': value} instead of list of { 'name': 'key', 'value': value}
- Add support for scope GET parameter in callback from OAUTH2 provider (useful for e.g. Google)
- Add support for oauth_extras dict in oauth config to set additional oauth paramters forwarded to OAUTH2 provider (Google uses this)
- Add support for dynamic:creator in oauth_extras to preset login hint etc when forwarding to OAuth2 auth endpoints (if creator==email, this allows you to send Google hint on which account to use with 'login_hint': 'dynamic:creator' in oauth_extras in config
- Add support for actor get_from_creator() to initialise an actor from a creator (only usable together with config variable unique_creator)
- Add support for get_properties(), delete_properties(), put_properties(), and post_properties in the on_aw() class. These allows on_aw overriding functions to process any old and new properties and return the resulting properties to be stored, deleted, or returned
- Move all internal (oauth_token, oauth_token_expiry, oauth_refresh_token, oauth_token_refresh_token_expiry, cookie_redirect, and trustee_root) data from properties (where they are exposed on GET /<actor_id>/properties) to internal variable store (attributes). Introduce config variable migrate_2_5_0 (default True) that will look for properties with oauth variable names if not found in internal store and move them over to internal store (should be turned off when all actors have migrated their oauth properties over to store)
- Add new interface InternalStore() (attribute.py) for storing and retrieving internal variables on an actor (i.e. attributes). All actors now have .store that can be used either as a dict or dot-notation. actor.store.var = 'this' or actor.store['var'] = 'this'. Set the variable to None to delete it. All variables are immediately stored to the database. Note that variable values must be json serializable
- Add new interface PropertyStore() (property.py) for storing and retrieving properties. Used just like InternalStore() and access through actor.property.my_var or actor.property['my_var']
- InternalStore(actor_id=None, config=None, bucket=None) can be used independently and the optional bucket parameter allows you to create an internal store that stores a set of variables in a specific bucket. A bucket is retrieved all at once and variables are written to database immediately
- Fix issue where downstream (trusts) server processing errors resulted in 405 instead of 500 error code
- Fix bug in oauth.put_request() where post was used instead of put
- Fix issue where 200 had Forbidden text

6.9 v2.4.3: Sep 27, 2018

- Don't do relative import with import_module, AWS Lambda gets a hiccup

6.10 v2.4.2: Sep 27, 2018

- Get rid of future requirement, just a pain

6.11 v2.4.1: Sep 26, 2018

- Fix bad relative imports
- Use extras_require for future (python2 support)

6.12 v2.4.0: Sep 22 2018

- Support python3

6.13 v2.3.0: Dec 27, 2017

- Entire API for handlers and Actor() as well as other objects changed to be PEP8 compliant
- Add support for head_request(() in oauth and oauth_head() in auth
- Change all uses of now() to utcnow()
- db_gae for Google AppEngine is not kept updated, so folder deprecated and just kept for later reference
- Full linting/PEP8 review
- Add support for actor_id (set id) on Actor.create()

6.14 v2.2.2: Dec 3, 2017

- Fix bug in region for properties and attributes resulting in using us-east-1 for these (and not us-west-1 as default)

6.15 v2.2.1: Dec 3, 2017

- Add support for environment variable AWS_DB_PREFIX to support multiple actingweb tables in same DynamoDB region

6.16 v2.2.0: Nov 25, 2017

- Add support for attribute.Attributes() and attribute.Buckets() (to be used for internal properties not exposed)
- Various bug fixes to make the oauth flows work

6.17 v2.1.2: Nov 12, 2017

- Split out actingweb module as a separate pypi library and repository
- Python2 support, not python3
- Support AWS DynamoDB and Google Datastore in sub-modules
- Refactor out a set of handlers to allow easy integration into any web framework
- actingwebdemo as a full-functioning demo app to show how the library is used

6.18 Jul 9, 2017

- Fix bug with unique actor setting and actor already exists
- Improve handling of enforce use of email property as creator
- Fix auth bug for callbacks (401 when no auth is expected)
- Add support for "lazy refresh" of oauth token, i.e. refresh if expired or refresh token has <24h to expiry
- Add support for Actors() class in actor.py to get a list of all actors with id and creator (ONLY for admin usage)
- Fix various bugs when subscriptions don't exist
- Improve logging when actor cannot be created

6.19 Apr 2, 2017

- Changed license to BSD after approval from Cisco Systems
- Fix bug in deletion of trust relationship that would not delete subscription
- Add support for GET param ?refresh=true for web-based sessions to ignore set cookie and do oauth
- Fix bug in oauth.oauth_delete() returning success when >299 is returned from upstream

6.20 Mar 11, 2017

- Fix bug in aw_actor_callbacks.py on does exist test after db refactoring
- Fix bug in handling of www/init form to set properties
- Add support to enforce that creator (in actor) is unique (Config.unique_creator bool)
- Add support to enforce that a creator field set to "creator" is overwritten if property "email" is set (Config.force_email_prop_as_creator bool, default True). Note that username for basic login then changes from creator to the value of email property. This functionality can be useful if actor is created by trustee and email is set later
- Add new DbActor.py function get_by_creator() to allow retrieving an actor based on the creator value

6.21 Feb 25, 2016

- Major refactoring of all database code
  - All db entities are now accessible only from the actingweb/* libraries
  - Each entity can be accessed one by one (e.g. trust.py exposes trust class) and as a list (e.g. trust.py exposes trusts class)
  - actor_id and any parameters that identify the entity must be set when the class is instantiated
  - get() must be called on the object to retrieve it from the database and the object is returned as a dictionary
  - Subsequent calls to get() will return the dictionary without database access, but any changes will be synced to database immediately
  - The actingweb/* libraries do not contain any database-specific code, but imports a db library that exposes the barebone db operations per object
  - The google datastore code can be found in actingweb/db_gae
  - Each database entity has its own .py file exposing get(), modify(), create(), delete() and some additional search/utility functions where needed
  - These db classes do not do anything at init, and get() and create() must include all parameters
  - The database handles are kept in the object, so modify() and delete() require a get() or create() before they can be called
- Currently, Google Datastore is the only supported db backend, but the db_* code can now fairly easily be adapted to new databases

6.22 Nov 19, 2016

- Create a better README in rst
- Add readthedocs.org support with conf.py and index.rst files
- Add the actingweb spec as an rst file
- Add a getting-started rst file
- Correct diff timestamps to UTC standard with T and Z notation
- Fix json issue where diff sub-structures are escaped
- Add 20 sec timeout on all urlfethc (inter-actor) communication
- Support using creator passphrase as bearer token IF creator username == trustee and passphrase has bitstrength > 80
- Added id, peerid, and subscriptionid in subscriptions to align with spec
- Add modiify() for actor to allow change of creator username
- Add support for /trust/trustee operations to align with spec
- Add /devtest path and config.devtest bool to allow test scripts
- Add /devtest testing of all aw_proxy functionality

6.23 Nov 17, 2016

- Renaming of getPeer() and deletePeer() to get_peer_trustee() and delete_peer_trustee() to avoid confusion
- Support for oauth_put() (and corresponding put_request()) and fix to accept 404 without refreshing token
- aw_proxy support for get_resource(), change_resource((), and delete_resource(()
- Support PUT on /resources

6.24 Nov 5, 2016

- Add support for getResources in aw_proxy.py
- Renamed peer to peerTrustee in peer.py to better reflect that it is created by actor as trustee

6.25 Nov 1, 2016

- Add support for change_resource(() and delete_resource(() in aw_proxy.py
- Add support for PUT to /resources and on_put_resources() in on_aw_resources.py

6.26 Oct 28, 2016

- Add support for establishment and tear-down of peer actors as trustee, actor.getPeer() and actor.deletePeer()
  - Add new db storage for peers created as trustee
  - Add new config.actor section in config.py to define known possible peers
- Add new actor support function: getTrustRelationshipByType()
- Add new AwProxy() class with helper functions to do RPCish peer operations on trust relationships
  - Either use trust_target or peer_target to send commands to a specific trust or to the trust associated with a peer (i.e. peer created by this app as a trustee)
  - Support for create_resource() (POST on remote actor path like /resources or /properties)
- Fix bug where clean up of actor did not delete remote subscription (actor.delete())
  - Add remoteSubscription deletion in aw-actor-subscription.py
  - Fix auth issue in aw-actor-callbacks.py revealed by ths bug

6.27 Oct 26, 2016

- Add support for trustee by adding trustee_root to actor factory
- Add debug logging in auth process
- Fix bug where actors created within the same second got the same id

6.28 Oct 15, 2016

- Added support for requests to /bot and a bot (permanent) token in config.py to do API requests without going through the /<actorid>/ paths. Used to support scenarios where users can communicate with a bot to initiate creation of an actor (or to do commands that don't need personal oauth authorization.

6.29 Oct 12, 2016

- Support for actor.get_from_property(property-name, value) to initialse an actor from db by looking up a property value (it must be unique)

6.30 Oct 9, 2016

- Added support for GET, PUT, and DELETE for any sub-level of /properties, also below resource, i.e. /properties/<subtarget>/<resource>/something/andmore/…
- Fixed bug where blob=", i.e. deletion, would not be registered

6.31 Oct 7, 2016

- Added support for resource (in addition to target and subtarget) in subscriptions, thus allowing subscriptions to e.g. /resources/files/<fileid> (where <fileid> is the resource to subscribe to. /properties/subtarget/resource subscriptions are also allowed.

6.32 Oct 6, 2016

- Added support for /resources with on_aw_resources.py in on_aw/ to hook into GET, DELETE, and POST requests to /resources
- Added fixes for box.com specific OAUTH implementation
- Added new function oauth_get(), oauth_post(), and oauth_delete() to Auth() class. These will refresh a token if necessary and can be used insted of oauth.get_request(), post_request(), and delete_request(()
- Minor refactoring of inner workings of auth.py and oauth.py wrt return values and error codes

6.33 Sep 25, 2016

- Added use_cache=False to all db operations to avoid cache issue when there are multiple instances of same app in gae

6.34 Sep 4, 2016

- Refactoring of creation of trust: - ensure that secret is generated by initiating peer - ensure that a peer cannot have more than one relationship - ensure that a secret can only be used for one relationship

6.35 Aug 28, 2016

- Major refactoring of auth.py. Only affects how init_actingweb() is used, see function docs

6.36 Aug 21, 2016: New features

- Removed the possibility of setting a secret when initiating a new relationship, as well as ability to change secret. This is to avoid the possibility of detecting existing secrets (from other peers) by testing secrets

6.37 Aug 15, 2016: Bug fixes

- Added new acl[“approved”] flag to auth.py indicating whether an authenticated peer has been approved
- Added new parameter to the authorise() function to turn off the requirement that peer has been approved to allow access
- Changed default relationship to the lowest level (associate) and turned off default approval of the default relationship
- Added a new authorisation check to subscriptions to make sure that only peers with access to a path are allowed to subscribe to those paths
- Added a new approval in trust to allow non-approved peers to delete their relationship (in case they want to “withdraw” their relationship request)
- Fixed uncaught json exception in create_remote_subscription()
- Fixed possibility of subpath being None instead of ‘’ in auth.py
- Fixed handling of both bool json type and string bool value for approved parameter for trust relationships

6.38 Aug 6, 2016: New features

- Support for deleting remote subscription (i.e. callback and subscription, dependent on direction) when an actor is deleted
  - New delete_remote_subscription() in actor.py
  - Added deletion to actor.delete()
  - New handler for DELETE of /callbacks in aw-actor-callbacks.py
  - New on_delete_callbacks() in on_aw_callbacks.py

6.39 Aug 6, 2016: Bug fixes

- Fixed bug where /meta/nonexistent resulted in 500

6.40 Aug 3, 2016: New features

- Support for doing callbacks when registering diffs
  - New function in actor.py: callback_subscription()
  - Added defer of callbacks to avoid stalling responses when adding diffs
  - Added new function get_trust_relationship() to get one specific relationship based on peerid (instead of searching using get_trust_relationships())
- Improved diff registration
  - Totally rewrote register_diffs() to register diffs for subscriptions that are not exact matches (i.e. broader/higher-level and more specific)
  - Added debug logging to trace how diffs are registered
- Owner-based access only to /callbacks/subscriptions
- Support for handling callbacks for subscriptions
  - New function in on_aw_callbacks.py: on_post_subscriptions() for handling callbacks on subscriptions
  - Changed aw-actor-callbacks.py to handle POSTs to /callbacks/subscriptions and forward those to on_post_subscriptions()

6.41 Aug 3, 2016: Bug fixes

- Added no cache to the rest of subscriptionDiffs DB operations to make sure that deferred subscription callbacks don't mess up sequencing
- Changed meta/raml to meta/specification to allow any type of specification language

6.42 Aug 1, 2016: New features

- Added support for GET on subscriptions as peer, generic register diffs function, as well as adding diffs when changing /properties. Also added support for creator initiating creation of a subscription by distingushing on POST to /subscriptions (as creator to inititate a subscription with another peer) and to /subscriptions/<peerid> (as peer to create subscription)
- Subscription is also created when initiating a remote subscription (using callback bool to set flag to identify a subscription where callback is expected). Still missing support for sending callbacks (high/low/none), as well as processing callbacks
- Added support for sequence number in subscription, so that missing diffs can be detected. Specific diffs can be retrieved by doing GET as peer on /subscriptions/<peerid>/<subid>/<seqnr> (and the diff will be cleared)

6.43 Jul 27, 2016: New features

- Started adding log statements to classes and methods
- Added this file to track changes

- Added support for requesting creation of subscriptions, GETing (with search) all subscriptions as creator (not peer), as well as deletion of subscriptions when an actor is deleted (still remaining GET all relationship as peer, GET on relationship to get diffs, DELETE subscription as peer, as well as mechanism to store diffs)

6.44 Jul 27, 2016: Bug fixes

- Changed all ndb.fetch() calls to not include a max item number
- Cleaned up actor delete() to go directly on database to delete all relevant items
- Fixed a bug where the requested peer would not store the requesting actor's mini-app type in db (in trust)
- Added use_cache=False in all trust.py ndb calls to get rid of the cache issues experienced when two different threads communicate to set up a trust
- Added a new check and return message when secret is not included in an "establish trust" request (requestor must always include secret)

6.45 July 12, 2016: New features

- config.py cleaned up a bit

6.46 July 12, 2016: Bug fixes

- Fix in on_aw_oauth_success where token can optionally supplied (first time oauth was done the token has not been flushed to db)
- Fix in on_aw_oauth_success where login attempt with wrong Spark user did not clear the cookie_redirect variable
- Fixed issue with wrong Content-Type header for GET and DELETE messages without json body

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index