xAPI Deep Dive Actor/Agent

Does a statement get recorded in an LRS if there is no one there to experience it?

The Problem with Abstractions

The Experience API is designed for recording information about experiences, but one of the assumptions is that someone, or a group of someones, has to be the experiencer. Enter the term “actor”, often referred to as the “I” in a xAPI statement, or grammatically, the subject.

There is a lot of abstraction in building xAPI statements, and at first glance, defining the “I” of a statement seems simple and concrete enough that we should start there. Unfortunately, it just isn’t that simple, “Who am I?” is a pretty big question of the ages and that question didn’t get any smaller in xAPI. To compound the issue, not only do you have to define the “I”, or perhaps the “royal we”, you have to tell someone else that it was you. And to further complicate matters, maybe you aren’t interested in just being “I,” but you want to bring along your friend “me” (or friends, “us”). And don’t get me started on “myself”. Okay, enough pronoun soup for a while, back to actor…

To understand the ‘actor’ portion of a statement, it is helpful to take a step back and understand the distinction between a key or property (left hand side of an assignment) versus the value (right hand side of an assignment) in a JSON object. Ultimately a statement is made up of properties that have values assigned to them, ‘actor’ is one such property, and in the case of ‘actor’ its value **must** be an Agent (or Group). This means that ‘actor’ is simply a placeholder (or pointer) and doesn’t have a concrete, standalone representation.

In other words we don’t think of an “Actor” as a noun itself, or type of object, we think of “actor” as pointing to a specific value. Also note that I’m using “actor” (lowercase) versus “Agent” uppercase to distinguish between properties of a statement and the types of values they hold. This is the point where my wife tells me that I am just playing semantics, and if she were a developer I would retort that is *precisely* what I'm doing because semantics are very important to me (us). (By the way, she isn't a developer, so I don't retort at all or I'd be experiencing the doghouse.)

Defining Agents and Groups

Agents

then, are a type of object, and all we can *really* know for sure about that agent is that its representation is consistent because all we have to represent an agent is an inverse functional identifier, which is a fancy way of saying a unique ID that we can trace back to the same entity. That inverse functional identifier can take several forms for Agents, e-mail address (or mbox) being the most easily understood. Along with the raw human readable e-mail address an Agent can be identified by the SHA1 hash of their email address (well, it has to be an IRI so it includes the "mailto:" part).

```
{
    mbox: "mailto:info@xapi.com",
    objectType: "Agent"
}
{
    mbox_sha1sum: "f427d80dc332a166bf5f160ec15f009ce7e68c4c",
    objectType: "Agent"
}
```

Moving beyond e-mail, an agent may be uniquely identified by their OpenID URI. While e-mail is still a pretty universally accepted concept and OpenID has taken off in some circles, the specification also provides for a more system specific variation such that an agent can be identified by combining a unique identifier for a given system, say Twitter.com, and their unique representation on *that* system, for instance their "Twitter handle"; the combination is known simply as an "account." While some systems will come and go, and we may eventually see the end of e-mail addresses, the concept of a unique ID for a system plus that system’s unique ID for an entity (account as a concept) should be flexible enough to last as long as the spec will.

```
{
    account: {
        homePage: "https://twitter.com",
        name: "projecttincan"
    },
    objectType: "Agent"
}
```

One important note, while an Agent may have multiple inverse functional identifiers available for use an Agent object should only include one of them in a given representation to avoid learning record stores from rejecting such requests for privacy reasons related to linking of inverse functional identifiers. And, oh yeah, an Agent can have a ‘name’ so that us humans can more easily associate with it, too.

```
{
    mbox: "mailto:info@xapi.com",
    objectType: "Agent",
    name: "Info at xapi.com"
}
```

In the above examples we also explicitly included the ‘objectType’ property set to “Agent,” that property can be left out whenever an object must be either an Agent or a Group and defaults to an Agent (such as in the ‘actor’ property).

Groups

are similar to Agents in that they are a type of object used to represent an entity, but with the potential of an additional property that allows a group to enumerate all or some of its constituents, specifically the ‘member’ property. Groups must provide the ‘objectType’ property with a value of “Group.” Groups come in two flavors: identified and unidentified (or anonymous). In the former case an identified group has an inverse functional identifier (or unique ID) just as an Agent does, and may or may not include its ‘member’ property. If an identified group includes a ‘member’ property with a list, it should not be assumed to be an exhaustive list, meaning that a statement may call out a specific subset of member Agents for an experience (perhaps the famous ones, or the biggest donors, or the best dressed, or the first to arrive). In the latter case an anonymous group is not associated with any uniquely identifying information, therefore does not have an inverse functional identifier, but must include the “member” property. Although the specification, as of 1.0.0, leaves it open that the member list in this case need not be exhaustive, it is a best practice to make it so, as there is no way to associate other Agents with that part of the statement. And although it is a natural inclination to associate unidentified groups with the exact same set of member Agents as the same group, the specification draws attention to the fact that implementing systems should not make this assumption. Additionally, both kinds of groups’ member lists must only include Agents, therefore it is not possible to nest Groups.

```
{

    mbox: "mailto:info@xapi.com",

    name: "Info at xapi.com",

    objectType: "Group",

    member: [

        {

            mbox_sha1sum:
"48010dcee68e9f9f4af7ff57569550e8b506a88d"

        },

        {
            mbox_sha1sum:
"ca3ffdb44c4727137e29ebf42ee80c2afdd8d328"

        },

        .

        .

        .

    ]

}

{

    objectType: "Group",

    member: [

        {

            mbox_sha1sum:
"90f96ca8c3ae315f0e40df4e16772eb6d05e3937"

        },

        {

            mbox_sha1sum:
"ca3ffdb44c4727137e29ebf42ee80c2afdd8d328"

        }

    ]

}
```

Back to Statements

Now with our Agent/Group objects in hand we just drop them into the “actor” property and away we go; however there are other places in a statement where an Agent can be used as well. The “me” instance mentioned earlier can be accomplished by placing my Agent or Group (“us”) in the ‘object’ property to form a statement similar to “Sam (Agent 1) helped me (Agent 2)”. In that case, the statement uses two Agents, the “actor” property still contains one, Sam in this case, along with the one used in “object,” Brian (or me) in this case. Agents or Groups can also be included in the ‘context’ of a statement as an ‘instructor,’ leading to statements of the form “Brian (actor) learned xAPI from Ben (instructor).” Context can also include a ‘team’ property but it must be a Group.

Last but not least, an Agent is used to populate the ‘authority’ property of a statement, but generally statement creation is done with out this, leaving it to be populated by the LRS (more on “authority” in a future post).

Outside of Statements

Since Agents are set into some of the most valuable parts of a statement’s makeup, they need to be query-able. Agent objects are passed via the “agent” query parameter to the statements API for retrieving statements that have a matching ‘actor’ or ‘object’ property. Send a request with the “related_agents” query flag turned on to find statements where an Agent exists in one of the other possible locations as well.

Agents are cool enough that they get their own API methods, known as the Agent Profile. Agent Profiles really warrant their own post for the future, but for now it is enough to say that we can associate arbitrary data with a particular Agent in an LRS using them. One example use case is storing user preferences. Along with the Agent Profile, Agents are also composed into the State API calls.

Gotchas

Besides being part of innumerable groups in most cases, these days a given person probably has many inverse functional identifiers as well. I personally have three e-mail accounts that I consider separate, one personal, one business, and one for various other things. And each one of those technically has aliases in about ten other domain names. Each of those could be considered distinct inverse functional identifiers, so that means I have about thirty ways to be identified, just by e-mail, not to mention I have at least ten public facing profiles (such as Twitter, Github, Facebook, LinkedIn, Google+, etc.) which all have an “account” concept that could be used in Experience API communications, and those are just the public ones. The key takeaway here is that systems working with Experience API need to account for the fact that a “Person” may have any number of unique identifiers.

Additionally, in all of the inverse functional identifier cases, we can’t know whether that e-mail address or account is a shared one or not, so while an Agent can be loosely associated with a person it should not be assumed to represent a single person. For that matter, we probably shouldn’t assume that it is even a human on the other end. There is also the issue of time and the fact that e-mail addresses or accounts can change hands, for example “info@xapi.com” could be sent to any number of people or “brian@example.com” might change hands from Brian Miller to Brian Smith. Ultimately that is just one of the reasons a ‘timestamp’ property exists, but we’ll get to that in a later post.