# Anatomy of a Tin Can API Statement

# About This Book

The Tin Can API can be a little more complicated than it appears on the surface. We challenged Brian Miller to write a series of blog posts on TinCanAPI.com detailing the entire anatomy of a Tin Can statement and all of the possibilities and considerations that go along with the Tin Can specification.

The result was the the 9 part blog series — Deep Dive: Anatomy of a Tin Can Statement.

This is the entire series, bound into one e-book.

Enjoy!

# About The Author

Brian Miller is one of the world's top Tin Can API developers, as well as a contributor to the specification. He's involved in several Tin Can working groups, and he's the curator of the Rustici Software Tin Can Registry. Follow him on Twitter @k95bm01.


# TABLE OF CONTENTS:

# Intended Audience

If you are not familiar with the basic concepts of what the Tin Can API provides, then you might want to start at http://tincanapi.com and return to this book when you are ready to start capturing learning experiences.

Since this is a fairly in-depth look at the structure of a Statement, it's assumed that the reader is already familiar with the basic concept of a Tin Can Statement, has at least seen a Statement, and possibly created one.

This book is for learning designers and developers who will be outfitting systems that will send Statements. We'll take a deep look at the various parts of a Statement's structure and we'll enable the reader to devise a broader set of Statements to capture a more full range of experiences.

If you are familiar with the concepts inherent in the Tin Can API but are new to crafting Statements themselves, you may want to have a look at the "Statements 101" primer available at http://tincanapi.com/statements-101/.

# Introduction

## Anatomy of a Tin Can Statement

Noun verb object.
I did this.

Statements are the vehicle by which experiences are captured when using the Tin Can API specification.

Statements are the vehicle by which experiences are captured when using the Tin Can API specification. Each part of a statement serves a particular purpose, but when used together they form a cohesive unit enabling a flexible, yet powerful system useful for capturing formal learning, informal learning, and virtually any other experience data. In this book I'll dissect the parts of a Statement and examine each individual part in detail, seeing both how it is used as part of a Statement, as well as mentioning when it is useful to the specification in other ways.

Statements are designed both to enable conferring the meaning of an experience, in other words why we'd bother with the data, as well as to facilitate the transport in and out of systems which care more about the shape of data than the content. At the Statement's core is the triple pattern, specifically Actor-Verb-Object, that is common in natural languages. Additional metadata further describes the experience and rounds out the statement's body. Triples have become a common way of capturing streams of data, particularly in the social media realm, and modeling learning experiences in this way has already proven to be quite effective.

But using a triple only gets us halfway to our goal — we still need a way to have systems handle the data effectively. Since the statement stream will be conveyed through the use of web services, JSON is the conventional choice for structuring the data. By combining an easily recognized pattern from natural

language, such as a triple, with a relatively readable yet highly structured data transfer language, in this case JSON, we can achieve both goals.

"JSON" stands for JavaScript Object Notation, and as the name suggests, it's a way of describing the structure of an object in the JavaScript language. Because of its early proliferation amongst web browsers JavaScript has become the language of the web, therefore a subset of the language such as JSON was a natural choice when needing to serialize/deserialize data being sent using web services. This has resulted in JSON either being paired with other structured languages such as XML, or being used exclusively by developers of API services. Wide adoption led to extensive library support in virtually every server side language. Combine library support, minimal size transfers, and relatively easy human readability and JSON becomes a natural choice for building Tin Can Statements.

Objects are the top level construct in JSON and are created using a pair of braces, such as {...}. Within the braces a set of key/value pairs enumerate the object's properties. The keys, or property names, are quoted strings using either double quotes (") or single quotes ('). The property values can be quoted strings, JavaScript primitives (such as 1, true or null), arrays or nested objects. Whitespace outside of quoted values is ignored. Because values of properties can be objects, an arbitrarily deep nesting of objects is possible. Arrays are lists of elements and are wrapped using brackets, such as [...], and elements are

delimited with a comma (,). The elements of an array are the same types as the values of properties and therefore add to the arbitrary nesting ability.

As it is fairly human readable, JSON is often best explained through an example. This example attempts to capture the majority of the possibilities of structuring data with JSON:

```
{
    "simpleProperty": "Some string value",
    "listProperty": [
        "first in list",
        "second in list"
    ],
    "booleanProperty": true,
    "nullProperty": null,
    "nestedObject": {
        "somePropertyOfObject": "I'm inside an object"
    }
}
```

A Statement is a specific type of object that has well defined properties. Drawing from the triple pattern mentioned above, there are three obvious properties, "actor", "verb", "object", all of which are required in every Statement. Along with these data stream staples the Tin Can API adds "result", "context", "id", "timestamp", etc. Each of the properties serving a specific purpose and taking a highly specified set of possible values. Here is an example of a complete, but minimal Statement:

```
{
    "actor": {
        "mbox": "mailto:brian.miller@tincanapi.com"
    },
    "verb": {
        "id": "http://adlnet.gov/expapi/verbs/experienced",
        "display": {
            "en-US": "experienced"
        }
    },
    "object": {
        "id": "http://tincanapi.com/webinar/anatomy-of-a-statement",
        "definition": {
            "type": "http://adlnet.gov/expapi/activities/media",
            "name": {
                "en-US": "Anatomy of a Tin Can Statement"
            }
        }
    }
}
```

Some values are simple strings or primitives, some values are very specific types of strings, such as URIs, and some values are other highly specified objects, such as Agent or Verb. The rest of this book lays out each property that can be contained in either a Statement or a subobject along with its properties and provides examples of the corresponding JSON structures.

| Statement Properties | Specified Subobjects |
|---|---|
| id | Agent |
| actor | Group |
| verb | Verb |
| object | Activity |
| context | Activity Definition |
| result | Context |
| timestamp | Result |
| stored | Score |
| authority | Statement Reference |
| version | Sub-Statement |
| attachments | Language Map |

By convention, object properties in the content will be quoted, usually with single quotes to disambiguate from other types of quoted material. A capital first letter is an indicator of a type of object as provided for in the specification (for example Agent). As the rest of the chapters of this book were originally blog posts, and they should still be available online, please feel free to leave comments or submit errata on the chapter's corresponding post.

# CHAPTER 1:
# Actor/Agent

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

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

**The Problem with Abstractions**

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

There is a lot of abstraction in building Tin Can 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 Tin Can. 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@tincanapi.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: "http://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@tincanapi.com",
      objectType: "Agent",
      name: "Info at TinCanAPI.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@tincanapi.com",
	name: "Info at TinCanAPI.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 Tin Can 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 enumerable 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 Tin Can API communications, and those are just the public ones. The key takeaway here is that systems working with Tin Can 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@tincanapi.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.

Go now, make statements!

# CHAPTER 2:
# Verbs

In continuing with our “Anatomy of a Tin Can Statement” series, here’s the next installment — verbs. In this post, I’ll tell you a *lot* about how verbs work with the Tin Can API. If you have any questions at all, please leave them in the comments below, or email info@tincanapi.com.

## Inclusion in Statements

Verbs are a required part of statements and including them is simple enough. Set a Verb object into the “verb” property of a statement to indicate the action being taken for a given experience. A Verb object can consist only of an “id” property pointing to a URI (well, IRI). Here is an example:

```
{
      id: http://adlnet.gov/expapi/verbs/experienced
}
```

While that’s sufficient it seems people prefer something a little more akin to their own language, therefore Verbs should include a “display” property as well. The ‘display’ property’s value is a language map (a list of language codes with corresponding string values). Language maps are central to giving the Tin Can API  internationalized data interoperability. Here is the same verb, but with a human readable display value:

```
{
      id: "http://adlnet.gov/expapi/verbs/experienced",
      display: {
            "en-US": "experienced"
      }
}
```

Additional language values can be easily added to the language map using the RFC 5646 language tags, “en-US” above is an example of American English.

**History**

Early in the specification process there was a pre-defined set of verbs. In the development of the 0.95 version of the specification that list moved to the object form with full URI for “id” and was moved out of the specification proper in favor of letting new verbs be created at will. ADL still maintains a list of verbs that are specifically designed for the learning community, though there is no reason those verbs can’t be used for other purposes as well. Verbs such as “attempted”, “experienced”, “passed”, “failed”, “answered”, and “completed” (in their URI form of course) match up well with previous standards and have become some of the most common used in Tin Can so far. It is expected that communities of practice will evolve to create their own set of specific verbs known and used within a particular community. And the exception to the rule, since they all have one, there is one predefined verb included in the specification which serves the special purpose of voiding a statement. To void a statement send a new statement with this special verb having id “http://adlnet.gov/expapi/verbs/voided” along with a statement ref (more about these in a future post) to any LRS that may have received the statement.

## Past Tense

Verbs should be past tense. Tin Can is designed to track experiences which by their nature are time based, consequently verbs are past tense because a statement has to be recorded (note past tense) for the experience. No matter how soon after a recorded experience is reported on, that portion of the experience must already be in the past. (This is also one of the reasons why a statement no longer indicates something as “in progress”.) The concept of time passing as relates to streams of activities, potentially within the same experience, provides a significant amount of the complexity required to derive meaning from just a pile of statements, but at the same time provides the flexibility that allows content creators’ imaginations to flourish.

## Resolvability

Verb IDs as URIs mean that many verbs can be resolved to a location (URL), and when they do they can provide additional meta information. The meta information should contain an object with a “name” and “description” properties, at least when requested as JSON, per the specification. These properties are used to provide information specifically about the verb rather than the representation of the verb itself (what the “display” property is for). This is a good start and as verbs and Tin Can evolve we’ll have a way to extend the information surrounding verbs (and other items using URIs). The Internet

purist in me says that because a Verb uses a URI and because I can pick a URI that can be a URL that I should, and that all verbs should resolve, but the specification (rightly so, cause I'm not always a purist) leaves it open that Verbs don't have to resolve.

**To Coin, or Not to Coin**

It really isn't a question! You should avoid coining new verbs except as a last resort. Okay, last resort may be a bit strong as early as Tin Can adoption is, but eventually the set of verbs should move towards a fixed state. Consider the three required parts of a statement–actor, verb, object–only one of these, the verb, can be consistently matched across experiences for different people, or indicate different actions within an experience for the same actor. Those two dimensions are fundamental to reporting and don't work if every new experience comes with a whole new set of verbs. This is where the community of practice comes in, verbs will gain traction through adoption. As verbs gain traction their common use allows system implementers to rely on their semantic meaning which is the foundation of the interoperability that a specification like Tin Can seeks to provide. Statement creators should look for and leverage existing verbs whenever possible.

## Registry

We've taken to calling a list of verbs (and other URI based components) a "registry," and have implemented one, specifically The Registry where you can go to find Verbs that are being used in the wild. Right now it consists of the list of ADL verbs, but we are working on functionality (when not writing blog posts) to allow users to create new verbs through a curated process precisely to prevent the explosion of verb creation that could lead to less interoperability. Certainly we can't prevent people from coining and using new verbs, and new verbs will be necessary over time, but we also want to help those verbs to be ever lasting (as they need to be since statements are) and that those ever lasting verbs will continue to resolve, so we've opened up the "id.tincanapi.com" domain namespace to be used for URI based ids. Verbs (and other items) created in The Registry will have resolvable URLs that will serve the meta data associated with them as defined by the specification.

## Meanings, Not Words

Verbs are tough, and English (other languages do too I'm sure) does its best to make them tougher. So far we've said to reuse verbs when possible to allow interoperability, but we've also said there will be communities of practice that will adopt their own set of verbs. There is a conflict in these two best practices that arises because a single verb may have different meanings

depending on context, a synonym if you will. To say it another way to stress the impact of this, Verb objects have an identifier that maps directly to a singular meaning, not to a specific word. It is the meaning therefore that must match when determining when a Verb object with a given ID can be reused in different cases.

“Fired” is a commonly cited one, probably because it is the one used by the specification. The word “fired” has very different meanings depending on the situation in which it is used. The specification calls out this fact and suggests that verb IDs be used to separate these meanings, the problem with that is how to demarcate the line. For instance are “fired a gun” and “fired a cannon” different verbs? One could argue that both are for the rapid expulsion of a projectile, so the same, similarly arguments can be made that the act of firing the two different instruments require significantly different skill sets, equipment, etc. which might make them different verbs. We are left with a gray (or is that grey?) area that will have to be filled in by systems consuming the statements and ultimately up to us fallible humans to step in and create meaning from difficult semantic relationships. Really only time and adoption can point us in the right direction when it comes to coining verbs and their synonyms.

To conclude, I mean to finish, I mean to sum up, I mean to wrap up… ah, the heck with it! Go now, make statements!

# CHAPTER 3:
# Activity

Anecdotally, the most common question asked when starting down the Tin Can development road is: “How do I get/create an activity ID?”

It seems the simplest place to start with Tin Can is sending a simple statement, and when showing Tin Can to someone new we often start with the Actor-Verb-Object structure and give them the “I did something” example. Satisfying the first two parts of that structure is fairly straightforward, given an example statement or two. Most people can easily identify themselves with an email address which gets them to an Agent for use as the ‘actor’ pretty quickly, and with a list of common verbs already widely known, it is easy enough to copy and paste one of those.

Since Tin Can is intended to be used for tracking experiences, the natural progression then is to include an Activity in the third part of the structure — the “something.” Now we run into a problem, or a bunch of them — nothing to copy and paste, nothing given to us by someone else to use in the statement. We are going to have to actually create something!

So what is an activity? Possibly the only thing we can say about an activity is that it has boundaries. We need a fundamental way to say that an activity is contained to a specific amount of time and/or potentially at a specific location. Those boundaries therefore are physical (or virtually physical, if that can possibly make sense) and/or temporal. Assuming we can identify a set of initial boundaries we may then have the ability to subdivide the area or time encompassed by those boundaries to create smaller and smaller sets of activities. The granularity with which we subdivide the activity space matters very little for the simple

act of recording statements, but is extremely important to be able to later derive meaning from a set of statements for reporting or other types of use.

For instance, if we want to be able to track attendance at a conference, it may be sufficient to create an activity for the conference as a whole. But, if we want to know the most popular speaker at a conference, we will need to have at least session granularity to our data. If we want to determine the most popular page on a website, we will need to have a page-specific activity. But, if we only care about total visitors to a site, we only need an activity for the site itself.

These are pretty big activities, but an activity can be as small or short as a particular instant in a video being seen, or something less physically concrete such as a single question in a quiz.

Beyond defining the boundaries for an activity, it is important to define relationships between activities. In the above examples, a "larger" activity was subdivided into smaller activities, which forms a parent/child relationship. Other relationships can be indicated via the Tin Can API, and while it is possible to create statements using an activity in isolation, it is important to think through how activities can be grouped to realize better reporting and decision making later in the process. Just as relationships between people change over time, relationships between activities are not necessarily fixed. For instance, a

session at a conference could be defined in such a way that the session is given at multiple conferences, so it may be that the session itself relates to multiple conference activities rather than being specific to a conference. This increases the reusability of the activity and can lead to more interesting social reporting possibilities. Alternatively, there could be a specific conference session activity **and** a generic session activity.

Along with the explicit relationships that are defined amongst activities, there is an implicit relationship within a Tin Can statement between the activities and the Verb. We covered that territory in "Statements 101" so I won't repeat it here.

An Activity (note the capital "A") then, as it pertains to the Tin Can API, is a type of object. The Tin Can specification lays out precisely the structure of that object and makes some recommendations on what should be included when creating statements. The structure of an Activity object is quite basic — it includes only three properties. The only requirement is an 'id' property that has a value that is a URI (got me again, it is really an IRI). The other two optional properties are the 'objectType' which has a value of "Activity" when included, and the 'definition' which itself is an object and is where the complexity of an Activity structure lives.

## Identifying

In so far as an Activity identifier is just a URI, constructing one is trivial. In a web-enabled world, URIs are all around us. Unlike with Verbs, as described in “Deep Dive: Verbs”, Activities will be coined liberally and are unique to an Activity, so should only be re-used when specifically talking about the exact same activity, and therefore will generally be coined by an Activity Provider. When selecting identifiers for Activities, the creator should either own or have permission to use a particular domain name space to prevent collisions. Care should also be taken so that the Activity described by a specific identifier is not changed to reference or be reused for what could be considered a different activity, after all it is a unique identifier. While the specification only states that the identifier be a URI, it is considered a best practice to use a scheme that can ultimately be resolvable by a large number of applications, such as “http” and “https”, and to use a fully qualified domain name rather than some shortened representation as are often seen on Intranets. These best practices specifically target the interoperability of systems that the Tin Can API was designed to provide. Following these best practices will also mean that the URI may eventually become a URL with the ability to be resolved to meta data associated with that Activity. Just as with Verbs, there is no requirement to make URIs resolvable, but forward-looking systems will do so and minimally need to allow for it to be done in the future.

Some sample identifiers:

- http://tincanapi.com/TinCanJS/Test/TinCan_getStatement/sync
- http://tincanapi.com/JsTetris_TCAPI
- http://tincanapi.com/GolfExample_TCAPI
- http://tincanapi.com/GolfExample_TCAPI/GolfAssessment.html
- http://tincanapi.com/GolfExample_TCAPI/GolfAssessment/interactions.playing_1

Additionally, our Tin Can bookmarklet will take the URL for any visited webpage and use it as an Activity "id" as the 'object' of a statement automatically.

**Definition**

Along with the "id", an Activity object may contain, and should for statements, a 'definition' property that points to an object itself that contains information about how that Activity is used, can be displayed, etc. It is important to remember that an activity has only one logical definition, even though you can include different definitions in separate statements without error. The LRS and statement consumers will have to pick what they consider to be the "right" definition and are free to do so as they choose.

Two optional (but recommended) properties are straightforward, specifically ‘name’ and ‘description’, with each being assigned a language map value that contains human-readable information about the Activity. A new property that was added in 1.0.0, ‘moreInfo’, provides for including a URL (IRL), a resolvable location, with more human readable information about an Activity. The Activity definition is one of the objects in the Tin Can API that allows for arbitrary extensions via an ‘extensions’ property (extensions are worth a whole post, so plan for one soon).

Finally, the Activity Definition object may contain a “type” property which must have a URI (IRI) as its value. Activity types are very similar to Verbs in a number of ways. Although they do not include a separate ‘display’ property, they should be generically re-usable, may resolve to metadata, and are included in our Registry. When defining a new Activity via a Definition object, the creator should take the time to determine whether there is an existing activity type that matches their activity before creating a new one. There is a nice list of pre-existing activity types that were borne out of the specification process and approved by ADL. We will be adding more to the Registry very soon, as well as accepting submissions from the community in a curated fashion.

```
{
      "id" : "http://tincanapi.com/GolfExample_TCAPI/GolfAssessment.html",
      "definition": {
            "name": {
                  "en-US": "Golf Example Assessment"
            },
            "description": {
                  "en-US": "An Assessment for the Golf Example course."
            },
            "type": "http://adlnet.gov/expapi/activities/assessment"
      },
      "objectType": "Activity"
}
```

Similar to how the specification includes one pre-defined Verb (see "voided"), one activity type in particular is called out by the specification to have special meaning, namely an "Interaction Activity". This activity type is rooted in the e-learning community and carries with it special properties that may be defined in the activity's definition. Interaction activities should have a type designated as 'http://adlnet.gov/expapi/activities/cmi.interaction', and are required to have an 'interactionType' property. For those not familiar with the common "interaction" term in the e-learning community, think of it as a question on a quiz (which is known as an "assessment"). The specification enumerates the list of possible interaction types and the associated properties that are added for each type (which also deserves its own post, man we have a lot to write).

```
{
      "id":
"http://tincanapi.com/GolfExample_TCAPI/GolfAssessment/interactions.ha
ndicap_3",
      "definition": {
            "description": {
                  "en-US": "A 'scratch golfer' has a handicap of ___"
            },
            "type":
http://adlnet.gov/expapi/activities/cmi.interaction,
            "interactionType": "numeric",
            "correctResponsesPattern": [
                  "0"
            ]
      },
      "objectType": "Activity"
}
```

## Parts of a Statement

If the attention to detail paid to the identification and structure of an activity isn't sufficient to express its importance to Tin Can, then the sheer number of places an Activity can be used will. As indicated by the examples above, a common pattern for the creation of statements is to include an Activity as the target, or specifically the 'object,' of a statement. The "Actor-Verb-Activity" is by far the most commonly used statement pattern to date.

```
{
	"actor": {
		"mbox": "mailto:info@tincanapi.com"
	},
	"verb": {
		"id": "http://adlnet.gov/expapi/verbs/attempted"
	},
	"object": {
		"id": "http://tincanapi.com/GolfExample_TCAPI",
		"definition": {
			"name": {
				"en-US": "Golf Example - Tin Can Course"
			},
			"description": {
				"en-US": "An overview of how to play the great
game of golf."
			},
			"type": "http://adlnet.gov/expapi/activities/course"
		},
		"objectType": "Activity"
	}
}
```

Moving beyond the 'object' property, Activities are an essential part of building context for a statement, so much so that "context" is a property of a statement that we haven't gotten to in our Deep Dive series yet, but wherein there is a 'contextActivities' property that itself takes lists of activities. This is where the relationships amongst activities as mentioned above is codified in a statement, and to do so, Activity objects themselves are included. Within the 'contextActivities' object, there is the potential for four lists of activities, specifically 'parent', 'grouping', 'category', and 'other.' In each case, one or more activities are used to provide context for the rest of the statement. The 'parent' list suggests a very direct relationship, one that is potentially recursive through multiple "generations."

The other three types provide for more indirect relationships and are designed to be maximally flexible, but do put more onus on reporting systems to make correct connections amongst activities.

```
{
	"actor": {
		"mbox": "mailto:info@tincanapi.com"
	},
	"verb": {
		"id": "http://adlnet.gov/expapi/verbs/experienced"
	},
	"object": {
		"id":
"http://tincanapi.com/GolfExample_TCAPI/HavingFun/MakeFriends.html",
		 "definition": {
			"name": {
				"en-US": "How to Make Friends on the Golf
Course"
			},
			"description": {
				"en-US": "An overview of how to make friends
on the golf course."
			}
		},
		"objectType": "Activity"
	},
	"context": {
		"contextActivities": {
			"parent": [
				{
					"id":
"http://tincanapi.com/GolfExample_TCAPI",
					"objectType": "Activity"
				}
			]
		}
	}
}
```

## Beyond Statements

Just as we saw with Agents in “Deep Dive: Actor/Agent”, Activities are used outside of statements as well. As a key component of statements, they need to be query-able. Activity objects are

matched through the statements query resource by passing the 'id' property of the object as an "activity" parameter, this matches statements where the Activity is the 'object' of the statement. To retrieve statement results where the activity is the 'object' or in other locations of the statement, set the 'related_activities' query flag to "true," (particularly important when we want to get all statements from a nested activity using one of the 'contextActivities' slots.)

As with Agents, again, Activities get their own API methods as well. Activities have a profile for storing arbitrary data that can be used across Agents for all instances of that Activity. The Tetris game example from the Tin Can Prototypes uses the Activity Profile API to store a list of high scores for the game (which is the base Activity). Each time a player finishes a game, that Activity Profile is accessed to see if their score makes the top ten, and if it does, then it is inserted into the proper rank location and the profile data is saved back to the LRS. Along with the Activity Profile API, an Activity 'id' is a required parameter when accessing the State API. State is then defined as arbitrary data associated with the combination of a unique Agent and a unique Activity (we'll ignore 'registration' for the time being).

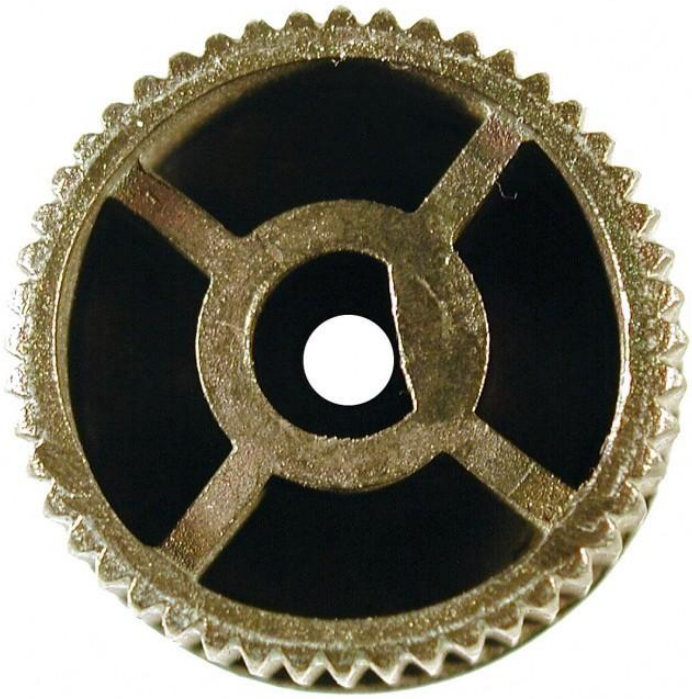
**Be Creative**

The Tin Can ecosystem is in its infancy and everyone has a chance to contribute to how statements will be built, how activities can be related, and the types of things we can track. This is the chance to be influential on the community and decide what kind of data

model is possible, and likely the most malleable part of the specification.

Go now, make statements!

# CHAPTER 4:
# Object



So many objects, so little time…”Guacamole is extra, is that okay?”

Most anyone first encountering Tin Can will find the “I did this” or “I did something” pattern for statements, we’ve used it once or twice ourselves on this site. This refers to the overall basic structure of a Tin Can statement, in other words, the “actor-verb-object” pattern. The “something” then is the object of the statement, and correspondingly the specification includes an ‘object’ property as a required portion of a Tin Can statement.

This seems straightforward; we’ve already seen in this series that the ‘actor’ property requires an Agent/Group object and the ‘verb’ property requires a Verb object, but the tricky part is that we don’t get a specific kind of object for use in the ‘object’ property. Instead we get a choice, and just like at Chipotle, making choices is hard. (Chicken burrito bowl, with a bag of chips by the way.) Our choice, though not as delicious, is amongst an Activity, an Agent, a Statement Reference, and a Sub-statement.

**Activity**

“Activities as objects” is the staple on the menu, the burrito of Tin Can statements, and by far the most used structure. We explored Activity objects in-depth in “Deep Dive: Activity”, covering very quickly that an Activity could be used as the value of the ‘object’ property of statements, and that doing so makes them query-able by that Activity. Consuming these kinds of statements is straightforward and somehow makes them feel self contained. I’ll assume you can find statements with this structure on your own — there are a couple in the Activity post and thousands in our public LRS.

## Agent/Group

Coming in a close second on the menu is my favorite, the burrito bowl. Currently not as common, this statement structure appeals to me because it starts to extend us beyond the common paths of other activity streams and further opens up possibilities related to social networking analysis. Just as with Activities, we covered that Agents/Groups can be the 'object' of statements in "Deep Dive: Actor/Agent" and that they too are query-able, though in this case requiring the "relatedAgents" query parameter. What I like to call Agent-Agent ("Double Agent"? nah, that's just bad) statements don't quite standalone, sometimes they require a fork, and go quite nicely with chips, but either way, you are going to have to dig into them to get the deliciousness out.

Some example usages of Agent-Agent statements:

- Brian contacted Mike.
- Brian was introduced to Tim.
- Mork was tutored by Mindy.
- Dr. Pepper met with Patient Zero.
- Mr. Obama defeated Mr. McCain and Mr. Romney (look a Group!).

In a couple of these statements, we are left without much context that seems very pertinent in the Activity as object world, but if our primary concern is about relationships between persons (or groups of people) then these statements can simplify an activity

provider's work. In the last of these, because of who the Agents are, we can glean a significant amount of context. (The concept of Context I'll talk more about a future deep dive post.)

Here is an example Agent-Agent statement. Note that the 'objectType' property is required when the Agent is in the 'object' of a statement:

```
{
	"actor": {
		"name": "Brian Miller",
		"mbox": "mailto:brian.miller@scorm.com"
	},
	"verb": {
		"id": "http://id.tincanapi.com/verb/contacted",
		"display": {
			"en-US": "contacted"
		}
	},
	"object": {
		"objectType": "Agent",
		"name": "Mike Rustici",
		"mbox": "mailto:mike.rustici@scorm.com"
	}
}
```

**Statement Reference**

For those wanting to cater to a healthier lifestyle, there is always the salad. A Statement Reference is a special kind of object — it is essentially a wrapper around the identifier for an existing statement. But like with the salad, a statement using a Statement Reference as 'object' carries significant weight with it because all of the importance of the referenced statement can reflect on the

referencing statement. (Cause let's face it, a salad is really just a burrito bowl with the lettuce on the bottom and dressing on the side.) A Statement Reference object has two properties and both are required: the 'objectType' property which must have a value of "StatementRef" and an 'id' with a value of a pre-existing statement. For example:

```
{
      "id": "fe2d144f-a5f5-4866-b498-620420fb3a9b",
      "objectType": "StatementRef"
}
```

There are a number of use cases where a Statement Reference makes sense as the object of a statement. Naturally, all of them relate to communications about statements, and some capture social activities common in other stream-based systems. These statements may look like the following:

- Brian favorited statement 'c92dbac8-4a7f-47ac-a508-64136199c568'
- Ben commented on statement '1c580b1b-ab27-4836-9131-9be841139bf9'
- Tim trusted statement 'a6227fe3-2caa-425a-a939-45cc661445cf'
- Grumpy Cat voided statement '89f3403f-92e7-462c-a68c-547633533439'

That last is particularly important because it is a form that is

predefined in the Tin Can specification. The “voided” verb, specifically the Verb with ID “http://adlnet.gov/expapi/verbs/voided”, carries special meaning as covered in “Deep Dive: Verb”. A full voiding statement looks like:

```
{
	"actor": {
		"name": "Auto Test Learner",
		"mbox": "mailto:auto_tests@example.scorm.com",
		"objectType": "Agent"
	},
	"verb": {
		"id": "http://adlnet.gov/expapi/verbs/voided",
		"display": {
			"en-US": "voided"
		}
	},
	"object": {
		"id": "29d943ca-fb8f-4e85-ace4-cec8e157ba78",
		"objectType": "StatementRef"
	}
}
```

After a statement of this form has been issued, the referenced statement is marked as voided and will no longer be included in the statement stream. The statement itself is still available in the LRS, but must be accessed in a direct way, and the voiding statement itself takes the original’s place in the stream.

Taking out the statement IDs and replacing them with more readable versions of a statement shows how these types of statement references can start to be put togther, such as:

- Brian brokered “Tim bought house from Mike”
- Samuel refereed “USA defeated El Salvador”

## Sub-Statement

Really, the only thing left in my analogy is the tacos and the kid's meal, and what better way to describe a Sub-Statement! A Sub-Statement has all of the basic parts of a Statement itself, the "actor-verb-object" pattern is still there, but it can't stand alone. (Who over the age of 10 really eats a single taco?) Some properties of normal Statements are forbidden to exist in a Sub-Statement, and a Sub-Statement has to have an 'objectType' property set to "SubStatement".

```
{
	"actor": {
		"name": "Brian Miller",
		"mbox": "mailto:brian.miller@scorm.com"
	},
	"verb": {
		"id": "http://id.tincanapi.com/verb/planned",
		"display": {
			"en-US": "planned"
		}
	},
	"object": {
		"objectType": "SubStatement",
		"actor": {
			"name": "Brian Miller",
			"mbox": "mailto:brian.miller@scorm.com"
		},
		 "verb": {
			"id": "http://id.tincanapi.com/verb/ran",
			"display": {
				"en-US": "ran"
			}
		},
		"object": {
			"id":
"http://id.tincanapi.com/activity/sample/NashvilleMarathon"
		}
	}
}
```

The English form of Sub-Statements tends to read a little funny, particularly with verbs usually being seen in the past tense. The specification calls out one particular use case — the intention of doing something. In this way, Sub-Statements can be used to indicate that something will happen in the future rather than recording something that has happened.

- Brian planned “Brian ran the Nashville marathon”
- Brian un-planned “Brian ran the Nashville marathon”
- Tim scheduled “Ben attended ADL conference call”
- Jena contracted “Mr. Clean provided Jenafits”

**Get Creative**

In “Deep Dive: Activity”, I talked about the creative possibilities stemming from the malleability of Activities, but the flexibility inherent in the ‘object’ property’s value types takes the creative potential to the next level. Each of these types of objects brings a different dimension to the Tin Can API, some of which are not easily expressed in other types of streams. And while each ‘object’ type is important in its own right, the verb-object combination is the relationship within a statement that allows it to be so expressive. Utilizing all of the options in different pairings allows learning systems and non-learning systems alike to frame a pattern of use that elevates reporting and comprehension beyond what has been possible in the past.

The only remaining question is “if I have a burrito bowl today, Wednesday, do I get a burrito, a salad, or another bowl (!) on

Thursday?”

Go now, make statements!

# CHAPTER 5:
# Extras/Others

In the other posts in this series I've covered some big topics with a post each, but not all properties of Tin Can statements need quite so much attention.

The Actor-Verb-Object pattern commonly pointed out as the basis of Tin Can statements is tremendous for deriving meaning from an experience, but there are several top-level properties that are specifically geared towards the mechanics needed to make the Tin Can API function well. Though they usually get less attention, particularly from non-developers, it is the combination of all of the properties that makes the specification such an achievement. This list of non-elegant, but absolutely critical properties includes 'id', 'timestamp', 'stored', 'authority', and 'version'. All but one have a "simple" value (meaning non-object), and enable specific usages, but also come with their own unique quirks.

**ID**

The 'id' property stores a simple string value, and that string must be a UUID (aka GUID). The short version is that a UUID is a Universally Unique Identifier, and it is the very strict meaning of "universally" that matters in the Tin Can API specification. For Tin Can API statements to provide interoperability they first have to be transferrable from one LRS to another, to make that possible the identifier of a statement can't be unique to only one system; therefore it must have a universally unique identifier, or an identifier that will never collide with one generated by some other system, whether that system is an LRS or an Activity Provider. How to generate UUIDs is beyond the scope of this post, but most programming languages have commonly available libraries, built in types, or at least sample algorithms for generating them efficiently. An example UUID looks like:

```
b1d6c3ea-4345-48a2-95ee-d6aea74cef59
```

All statements stored in an LRS must have an 'id.' The specification purposely leaves it up to the Activity Provider to decide if they want to generate statements with pre-set IDs, though indicates it is a best practice to do so. However, if the Activity Provider does not send an 'id' property as part of a statement one will be created and assigned by the LRS. While the above looks like just a series of dash delimited numbers and letters generating a UUID must follow a particular algorithm, it's critical to generate proper UUIDs to avoid collisions. While creating your own is recommended, it's much better to let the LRS assign a good UUID than to generate them improperly (they should always be computer generated, not manually created).

Statement IDs are useful for retrieving specific statements, for instance by systems that implement favoriting or perhaps badges, via the LRS Statement API which takes the 'id' as a query parameter. Additionally, as mentioned in "Deep Dive: object" the 'id' property is necessary for leveraging Statement References, such as when voiding a statement.

**Timestamp**

The 'timestamp' property's value is an ISO8601 date+time value in a string format that indicates when the statement was created and whose intention is to capture when the experience occurred.

Like with the ‘id’, the ‘timestamp’ value will be set by the LRS if not set in the statement by the Activity Provider. Here is an example timestamp value:

```
2013-08-20T14:22:20.028Z
```

Note that ‘timestamp’ values have sub-second precision and must contain a timezone, so implementing systems should be prepared to deal with these (the ‘Z’ above represents UTC, time zones are well beyond the scope of this post). It is intended that ‘timestamp’ values represent either a past time or the current (soon to be past) time. The specification does specifically call out the case of future timestamp values as being useful in SubStatements where it is expected that a related statement will be sent at or near that future time. Naturally when that occurs, both will then be in the past. The notion of timestamps in the past matches up well with the expectation that verbs use past tense and that statements can only capture what has occurred.

The ‘timestamp’ property is one of the mechanical items that allows Tin Can to function in an offline mode. When offline, statements can be created with an accurate ‘timestamp’ value even though they will not reach the LRS immediately. Reporting tools can then leverage the ‘timestamp’ to properly order what occurred. This also helps facilitate queuing mechanisms that may be online, but want to batch report statements for performance reasons. In a similar sense, it is also possible to capture historical

events that happened well before the specification existed, for instance I could capture a set of learning records for my school years based on offline records that I've kept. The lack of accuracy of the timestamps should be considered, but I could certainly capture experiences close to when they occurred. For instance based on various records I could create a statement for my college graduation at "1999-06-12T14:00:00-04", the date is accurate, the time zone is accurate, the time itself is a little rough but may be meaningless in a timeline spanning decades.

Timestamps can also help provide meaning in statements. In some cases the 'timestamp' value could be sufficient to determine unique context for a given experience. For instance a timestamp in a statement about a conference may imply conference attendance for the given date. When used in conjunction with the 'duration' property of the Result object (a future blog post topic) a reporting system can determine overlapping experiences and have another dimension for comparison.

**Stored**

Along with 'timestamp', the 'stored' property's value is an ISO8601 date+time value, but has a very different meaning. The 'stored' property's value is purely about the mechanics of the API and as such is set by the LRS when that LRS receives the statement. The 'stored' value is then leveraged via the Statement API's query resource for providing the statement stream in one specific order, and optionally including only a range of statements. For instance a system may periodically poll an LRS for

only new statements since a specific point in time using the 'since' query parameter. Alternatively, the 'until' query parameter allows for requesting statements stored before a point in time. It is important to understand that the 'stored' value for a given statement retrieved from two different learning record stores may be different.

**Authority**

The 'authority' property is another that will most often be set by the LRS, but has an object value. Specifically, the 'authority' will contain an Agent or Group object which I covered in the "Deep Dive: Agent/actor" post. When the statement is stored using 3-legged OAuth, the 'authority' will contain a non-identified Group with two members, one for the user and one for the application, otherwise it will hold an Agent representing the user connecting to the LRS.

The authority represents how that statement ended up in the LRS and correspondingly suggests the level of trust of that statement. The level of trust of a statement is directly related to the level of trust of the authority, and the level of trust of the authority is relative; it is the level of trust **you** have in that authority, and therefore the level of trust **you** have in a statement. A statement where the 'actor' and 'authority' match, for instance, has the lowest level of trust as it was self generated. A statement where the 'authority' is a single Agent but different from the 'actor' will often have a higher level of trust, assuming that the authorizing Agent is trusted. The level of trust of a particular Agent can vary

based on how hard it is to assume control of that Agent. For instance, an automated system with significant security controls will likely have a higher degree of trust than a normal user accessing an LRS through a public web interface. The 3-legged OAuth method should provide the most trust because it is based on two parties both agreeing that a statement should be generated, in this case the user has agreed to let the application speak for them and that the application indicates that the experience has occurred.

While the handling of the 'authority' property is fairly mechanical in nature, it is the notion of trust that lends to taking meaning from a statement, and as extension how someone will act on the information conveyed by that statement. For example, a pile of statements created by an accredited university may carry more weight in a job interview than those generated by a bookmarklet, or statements generated by a heart monitor machine in a hospital room will be trusted over those generated by manually entering pulse information in a phone app when diagnosing heart conditions.

**Version**

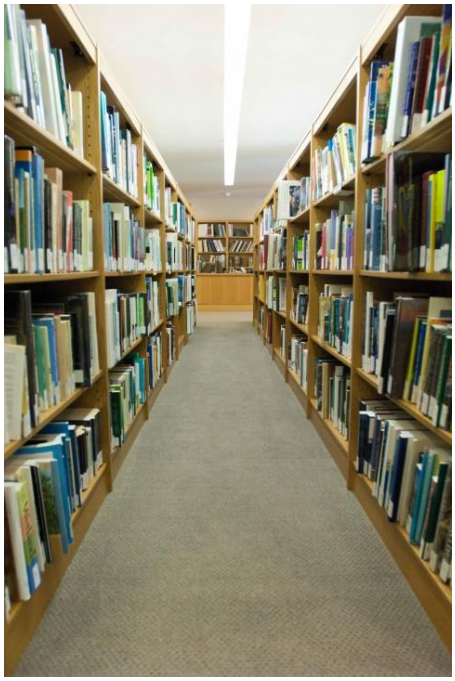
The 'version' property is one of the newest additions to the Tin Can Statement specification. It indicates what version of the API was in use when the statement was recorded, and can be leveraged by systems consuming the statement stream to properly parse and otherwise handle the statement structures without having to make assumptions about the statement version

from its structure. In general this property will be set by the LRS, reserving pre-setting of its value for LRS to LRS transfers. Because this property didn't exist until the 1.0.0 specification, statements retrieved from an LRS using the prior draft specifications will not include this property.

While talking in generalities about the Tin Can API specification, and specifically the Statement structure, it is easy to lose sight of the fact that the meaning we so desperately want to capture has to be codified in actual data elements. But the data elements most often conveyed via "I did X" aren't sufficient to developers building real systems, particularly interoperable ones. The essential components for meaning combined with the more mechanical ones outlined here make a well rounded specification that can work for both implementers and users (Activity Providers) of learning record stores and the associated web service resources.

Go now, make statements!

# CHAPTER 6:
# Context



Although not required, most statements are going to need to include some additional context to convey the extent of their meaning.

With two toddlers at home, I'm fairly used to short, choppy sentences as a manner of communication. I have whole books filled with them, granted they are only about 15 pages long with type even my grandfather can see. And while most of the time I can get my point across, and usually even so can the toddler, and the books are certainly appealing (to at least one of us), I am excited for my daughters to get to explore the richness that is language, particularly one as "colorful" as English. The same can be said of Tin Can API exploration (though not necessarily by my daughters). While the Actor-Verb-Object structure is critical for a bare minimum of understanding, it is the context of the statement that gives it life and, dare I say beauty (okay, that's a bit of a stretch even for me).

Although not required, most statements are going to need to include some additional context to convey the extent of their meaning. One way to capture additional meaning takes the form of a Context object placed in the 'context' property of the statement. All properties of the Context object are optional, and they may be mixed and matched as needed with only a few restrictions. There are nine Context properties ranging from the very specific 'registration' to the completely open 'extensions'.

**contextActivities**

I've already touched on the 'contextActivities' property in the "Deep Dive: Activity" post. As mentioned there, the 'contextActivities' property takes an object as its value as well. This object has four optional properties itself, specifically 'parent',

'grouping', 'category', and 'other.' These properties take the same type of value, either a list (array) of Activity objects or a single Activity object. The Activity objects included in these lists form the relationships amongst Activities and provide structure to what would be otherwise isolated experiences. Placing an Activity in one of the 'contextActivities' properties allows it to be queried via the Statements API using the 'related_activities' parameter.

Using the 'parent' property generally implies that the Statement's object is itself an Activity, specifically one that is a sub-Activity of a larger whole. The other properties have looser relationship qualities, but the specification does call out specific meanings for each. The 'category' property relates to a statement as being part of a "profile" such that it adheres to some known, expected use case. The 'grouping' property allows statements to be associated based on their object's Activity as part of a larger whole but without the direct subset correlation as with 'parent.' Finally, 'other' is for catching any other use cases not defined directly in the specification.

**Registration**

The 'registration' property has its roots in the LMS (Learning Management System) / SCORM world so is related to the concept of a registration there which is tied to when a learner is enrolled or enrolls for a particular experience. This property takes a UUID (or GUID) value as a string just like the 'id' property as covered in "Deep Dive: Extras/Others". Fundamentally, it is in the specification to support the concept of identifying a specific

instance of a person (or persons) having an experience which need not be recorded as a single statement and it is not restricted to be used in statements for a specific Agent or Activity. In the Tin Can space, an experience may be captured with many statements from multiple points of view and be made up of numerous activities, the registration value then can be used to tie all of them together.

For example, in the Tin Can Prototypes JS Tetris game, the top level Activity is consistent and a single Agent may play multiple games which generate numerous statements (one for each level reached, etc.); therefore it is not sufficient for us to look at statements for just the Agent and Activity combination to determine statements unique to a played game. We could try to piece the single game experience together based on a starting and ending statement and a range of timestamps, but this is overly error prone and a little too clever. Instead, each new game is assigned a registration which is included in the Context of the statements generated for that instance of the experience. This makes it possible to discern the set of statements making up a unique game amongst all of those played by a specific Agent using that Activity.

To facilitate the use case intended for 'registration' it is a property that is exposed via the Statements query API. In other words, a client can query the LRS directly for the statements that have a specific registration. Additionally, the registration concept can be applied to the State API as an optional component of what makes a document unique.

## Instructor

The ‘instructor’ property takes a value that is either an Agent or Group as covered in the “Deep Dive: Agent/actor” post. This value is fairly specific to learning experiences, the intended use case for Tin Can API after all, and will likely correlate to certain kinds of verbs. This property is hopefully self explanatory, though need not be confined to a “formal” instructor as informal training experiences occur commonly between two Agents. Statements with an ‘instructor’ property might read like: “Brian learned Tin Can from Ben (instructor).”

## Team

The ‘team’ property requires a Group object as value. I’ll admit, I struggled with the meaning of this one so sought out advice from my team (turns out it was a team of one, but whatever). The key to the ‘team’ property’s meaning is that it is useful when a singular Agent (or subset of a Group) performs an action that is part of an experience where it is important to recognize the team as part of the context. For instance, during a car race a pit crew may constitute a team, but only one member is involved in refueling the vehicle during each stop, so a statement might be created for “Brian refueled Car 33 during pit stop 2 (performed by) team Red” where the “performed by” is inferred from the ‘team’ property having a value. In the training space, you could think of a team of physicians and nurses running a disaster drill where each has an assigned task, but each task contributes to the common objectives of a single team. Without a complete,

cohesive set of tasks performed correctly by each member as the team as a whole the overall result of the exercise could be failure.

**Statement**

In “Deep Dive: object” I talked about Statement References. Statement References can be used as the ‘object’ of the statement, but in the case where another object, perhaps an Activity, makes more sense in that position a Statement Reference can still be used as context within the Context object’s ‘statement’ property. There was also a section of that post that talked about Sub-Statements and how they can be used to indicate a future event. As an example, the ‘statement’ Context property would be a great place to capture a reference to the original “planning” statement in statement(s) generated when the event finally occurs.

**Revision**

The ‘revision’ property takes a string value that has a free form value. Additionally, the specification precludes the use of this property when the ‘object’ of the statement is an Agent or Group. The value of this property is intended to capture small or minor edits made to an experience, where minor edits include typos or spelling errors. More significant edits, where meaning itself may have changed, should be handled through updates to Activity IDs, etc. It is important to remember that this ‘revision’ property is context for the statement rather than part of the definition of a single Activity. It has roots in the “packaged learning world” and is

provided primarily to capture the small revisions where a package may change but an accompanying change to a new Activity ID was not required.

**Platform**

The ‘platform’ value takes a string as well, and equally free form, and again must not be used with an ‘object’ that is an Agent or Group. Because experiences, specifically learning activities, may be delivered in multiple ways this property is intended to capture information about how, or possibly where, the experience occurred. For instance, it might have been delivered via an “online course” or “in person” or perhaps via a “simulator.” It may be the case that these have different Activity objects with unique IDs, but it may also be meaningful to capture all of the ways that someone can experience the same learning objective such that the delivery method is merely context.

**Language**

International interoperability is particularly important to the Tin Can API specification and the ‘language’ Context property provides a way to capture the language of the original experience when known and identifiable. The value for this property should be a string with an RFC 5646 formatted value, the same as the keys that make up the language maps used elsewhere in the specification, such as the ‘display’ property of verbs and the ‘name’ property of Activity Definitions. Combining those language maps with the ‘language’ context can provide a fuller picture of

the actor's experience.

**Extensions**

The 'extensions' property may occur in a couple places in a statement, another place is the Activity Definition as mentioned in the "Deep Dive: Activity" post. Extensions warrant their own post which is coming soon, but essentially it is a catch all for any other context that could possibly be relevant to this specific statement. For some examples of Extensions for use in Context, check out registry.tincanapi.com.

The Context object provides such a varied set of values it is a shame to not include as much information in a statement as is possible. In these early days of Tin Can API adoption, the simple statements win out as it seems we are but mere toddlers exploring a new language, or at least a new way to structure our language. As adoption increases, so will the complexity of the experiences we are able to capture. I feel we've wished to capture them for a long time, now we are empowered to, and the harder task of drawing meaning from all the contextual elements still awaits us.

Go now, make statements!

# CHAPTER 7:
# Result


Statements have an optional ‘result’ property that can be used to capture a “measurable outcome” from an experience.

So far throughout the Deep Dive series that I've been writing, there has been one thing notably lacking, e-learning. It turns out that the framers of the Tin Can API specification hit on something big enough that it need not be boxed in by just the e-learning world, and so far we've seen a lot of adopters thinking outside the typical e-learning sphere. But there is a history there, and the work on the Tin Can API specification was born out of a real desire to advance the specific space around e-learning. The Result object is there to capture some of what is not already catered for and make sure the Tin Can API can first, and maybe foremost, flourish in the e-learning space, though even it doesn't have to be used only for e-learning.

Statements have an optional 'result' property that can be used to capture a "measurable outcome" from an experience. The value of the 'result' property is specifically a Result object which has a number of properties designed to capture the types of results that have historically been generated from learning activities. The properties of the Result object are 'success', 'completion', 'response', 'duration', 'score' and 'extensions.' For readers used to SCORM, these properties have direct analogs to the data you are already capturing, but in Tin Can all properties are optional and the specification doesn't indicate how they must be used together.

**Success**

The 'success' property takes a boolean value (true/false) and provides for a pass/fail categorization of an Activity. It is an

important distinction that the specification says Activity in its description of 'success' because it is possible for an experience to be made up of more than one Activity, particularly when the experience is easily sub-divided. It is entirely possible for an overall experience to be a success or failure with individual sub-activities having the opposite value. For example, the 'success' property of a statement capturing the answering of a question might be 'false' (or failure), but within a statement capturing a summary of the overall quiz the Result object's 'success' property might be 'true' (or pass).

**Completion**

The 'completion' property is a Boolean value (true/false) as well and captures whether the Activity was completed. This is somewhat confusing in light of the very commonly used "completed" (http://adlnet.gov/expapi/verbs/completed) verb. The key here is that the 'completion' property is a part of the specification and indicates that based solely on this statement it can be determined that the actor has done what is required to consider the Activity completed. The commonly used "completed" verb may indicate the same thing for a particular profile, but that is not captured by the specification itself, it is profile specific. The Tin Can specification provides for flexibility in how systems will react to the data stored, the 'completion' property is very much intended to capture the concept of "completion" as it is in the LMS AICC/SCORM world.

## Duration

Continuing the tradition of well named properties, the ‘duration’ property takes a value indicating the length of time taken for the experience captured by the statement. The value is a string but must be formatted using the ISO 8601 standard and is specified to have a maximum precision of 0.01 seconds. Activity Providers need to pay particular attention to the notion of precision with the ‘duration’ property. For example, two values that are both acceptable based on the formatting requirements are “P1M” and “P30D”, but in the first case we can only read this as one month which may be any of 28, 29, 30, or 31 days whereas in the latter we know this to be explicitly 30 days. Because of the requirement to use an ISO 8601 formatted value, arbitrary units can’t be included as a duration easily, that is, the duration can’t be something like “12 slides” or “3 courses” in that case separate statements or an extension is probably the best practice.

As I mentioned in “Deep Dive: Extras”, the ‘duration’ property can work together with the ‘timestamp’ property to give an indication of precisely when and for how long an experience took place allowing reporting systems to do interesting things with time overlaps. One interesting point brought up by Andrew Downes via GitHub is that the specification leaves open when in relation to the life of the experience the ‘timestamp’ occurs. The bottom line seems to be that if the duration is going to be used in relation to the timestamp that there is an agreement between the two and reporting systems need to take this into account. Consensus seems to be favoring that the

'timestamp' represents the ending point of the duration when it is included.

**Response**

The 'response' property takes a string as its value which has a format that is Activity specific which is to say it can store virtually anything. This value could correspond to text entered by a user to answer a question, or a serialized true/false answer, or the identifier for an answer in a multiple choice question, etc. When using the predefined "cmi.interaction" (http://adlnet.gov/expapi/activities/cmi.interaction) Activity Type the value should correspond to one of the entries in the 'correctResponsesPattern' property.

**Score**

Unlike the others in the Result object, the 'score' property takes an object itself, the Score object which has its own set of properties. The Score object's properties are all optional and each is a decimal number. These properties, 'scaled', 'raw', 'min', and 'max,' correspond directly to the CMI properties from the SCORM 2004 specification. The 'raw' value corresponds to a nominal value and when they are provided it must be between the values of 'min' and 'max.' Those two properties, 'min' and 'max,' correspond to nominal values marking the starting and ending point of a range of values. If the 'raw' value can be calculated as a percentage, then it is expected that the 'scaled' value is populated with a value between -1 and 1. For example, on a 25 question quiz

a user may get 20 questions correct, corresponding to a 'raw' value of '20' and a scaled value of '0.8' (or 80%), 'min' could be included as '0' and 'max' as '25.'

One important note, the specification indicates that the Score object and associated properties should not be included for determining progress or completion. Presumably the 'completion' boolean property in the outer Result object should be used for the latter, and an extension is recommended for the former.

**Extensions**

As we saw in "Deep Dive: Context" and "Deep Dive: Activity", the 'extensions' property holds an object of catch all data that is necessary to capture the meaning of the result. An example of an extension that might be useful in a Result object that we've already added to The Registry is "Ending Position," http://id.tincanapi.com/extension/ending-position. For example, it can be used to indicate the final place of a runner in a race. (More about extensions is coming in a future post.)

Leveraging the elements of a Result object in a Tin Can statement facilitates conveying information about what happened during an experience and directly lends itself to tracking objective outcomes. It gives us a way to measure the performance of an Agent when interacting with an Activity, particularly those that are prescribed to the task of assessing capability associated with a learning process. The Result object captures the essence of the Tin Can API being forged from past experience in the learning

standards world and provides an easier path forward for existing content created for the formal training model.

Go now, make statements!

# CHAPTER 8:
# Extensions

Several times throughout the Deep Dive series, I've mentioned "catch all" objects and a future post — here it is.

The framers of the Tin Can API specification knew that the overall structure of a statement, particularly with its oft mentioned Actor-Verb-Object pattern, could capture a great deal of information about learning experiences, but they also realized that there was no way for them to account for all types of experiences that people wished to record. So they left an out in the form of 'extensions' properties.

These 'extensions' properties take a special kind of object where the set of available properties is not known ahead of time, unlike all other objects in the specification. It is this unique structure that leaves it up to the Activity Provider to decide what to capture, and to own how it will be captured.

The 'extensions' property takes as its value an object where the properties of that object are strings in the form of URIs and the values can take any form, including objects. Using URIs as the properties allows for the identifiers to be owned via domain ownership, and therefore prevents the possibility of collisions as long as people respect that ownership. This means that to create (or coin) a new Extension property you should do so in a domain that you own, control, or have been given permission to use.

Here are two example Extension objects, one used to include an ISBN (a book identifier), and the other used to specify a starting point and ending point (perhaps page numbers):

```
{
    "http://id.tincanapi.com/extension/isbn": "978-1449304195"
}
```

```
{
    "http://id.tincanapi.com/extension/ending-point": 36,
    "http://id.tincanapi.com/extension/starting-point": 48
}
```

The ‘extensions’ property can be used in multiple locations within a statement, specifically the Activity Definition, Context, and Result objects. The first example above, the ISBN, would be a perfect fit to include in an Activity Definition for a book Activity. Another key is that specific ‘extensions’ properties can be used in any of these positions. In other words, the starting/ending point properties could make sense in the context of one statement and the result of a different one. The second example above might be used by a teacher to assign a student a set of pages to read which might be included in the context of a statement about the assigning, while the same object might be used in the result of the statement capturing the student’s reading of those pages.

As mentioned, the format of the value for an Extension property is left open which is both maximally flexible for Activity Providers and problematic for reporting systems. In the above example the page numbers are captured as integers (or more specifically numbers), but some other starting point may need to be captured

as a string. There is no right answer as to whether an extension will always have the same formatted value or not.

Using different types of values with the same identifier is problematic for systems that will leverage the data from the statements later. In some cases, distinguishing the format will be straightforward and reporting systems will be able to handle it relatively easily. In other cases, there will be a clear format that an extension value must use. For example, the "http://id.tincanapi.com/extension/geojson" extension has a very precise value format based on the GeoJSON specification. The ISBN extension property used earlier always takes a string but a user must examine the length to properly handle it. Still other times it may be left up to the context in which the property is used as to how to handle its value. This is the trickiest of cases and one which has been coming up in conversations around Tin Can API more and more frequently. Extension coiners should consider including in the description of their extension (more on that below) information about what form the value should take. The example of the ISBN specifically includes:

> Value should be either a 10 digit ISBN or 13 digit ISBN string. Either value is acceptable as implementing systems can easily distinguish the two based on the length of the value.

This allows developers to understand whether an extension will serve their purpose or not, and by conforming to the definition provided, they can expect their usage to interoperate with others' usages.

Unlike other objects in a Statement, because the extension property URIs are the Extension object's properties themselves, there is no place to provide metadata information about the property itself. In other words, there is no "local" way in the statement to provide a human readable name for the extension property or the description needed to explain how that extension property is to be used. This is another key to using URIs as the representation of the properties — many of them are easily convertible to URLs. Specifically using a URL and making that address resolvable enables a way to fetch metadata about the extension property. The format of the metadata is specified in the Tin Can API, when resolving the URL with a request for content-type of "application/json" the host should return a JSON object including a "name" and "description" properties whose values are language map objects as seen elsewhere in the specification.
By example, fetching the resource for an extension property such as "http://id.tincanapi.com/extension/tags" will return:

```
{
    "name": {
       "en-US": "tags"
    },
    "description": {
        "en-US": "A list of arbitrary tags to associate with a
statement.
                  Value of the extension should be an array with each
tag being
                  a string value as an element of the array."
    }
}
```

As described in other posts in this series, The Registry has been created to catalog available extensions to assist with interoperability. New Extension properties can be created in the "id.tincanapi.com" domain using the web interface to ensure that they can always be resolvable. Other extension properties can also be recorded to make them easier to find. There is a nice and ever growing list of extensions already listed, some of which we pre-populated in anticipation of their need by the community. Browsing the list is an excellent way to see the extent of the varied ways extension properties can and will be used. Additionally, 'extensions' are also prescribed for use in the "/about" resource that an LRS must provide, though including the property isn't specifically required at this time. An example is the "powered-by" extension as part of the /about resource result on SCORM Cloud, it returns:

```
{
    "extensions": {
        "http://id.tincanapi.com/extension/powered-by": {
            "name": "Tin Can Engine",
            "homePage": "http://tincanapi.com/lrs-lms/lrs-for-lmss-
home/",
            "version": "2012.1.0.5039b"
        }
    },
    "version" : [ "1.0.0" ]
}
```

“With great power comes great responsibility,” or so it is said, and using “extensions” is no different. With its flexibility it is simple to turn to the easy way out and just shove any data into an Extension when selecting a more complex statement structure or being more specific about an Activity may be more suitable. Even within our own walls, we recently had a conversation about using an Activity with a specific Activity Type rather than using an Extension property with a value that would be a URI itself. It turned out we didn’t need to use “extensions” at all and were better served by using Context activities in its place.

Go now, make statements!

# CHAPTER 9:
# Attachments

As network speeds and the processing power of devices improves, the size of the files we use to capture our experiences, be they photos or other graphics, videos, or complex documents, increases. We need a way to associate these ever-growing files with the metadata capturing the rest of the experience.

As network speeds and the processing power of devices improves, the size of the files we use to capture our experiences, be they photos or other graphics, videos, or complex documents, increases. We need a way to associate these ever-growing files with the metadata capturing the rest of the experience. This data comes in all shapes and sizes, particularly these days, and as flexible and readable as JSON is, it isn't great for capturing large amounts of binary bits, but the Tin Can API specification allows for including attachments with statements for this purpose.

Attachment handling is implemented through a combination of an 'attachments' property of the Statement object itself and optional inclusion of copies of the files themselves. Note the use of plurals here, a single Statement may be associated with multiple files, therefore the 'attachments' property of a Statement takes an array as its value. The elements of this array are objects with properties, some required and some optional, that provide metadata about the included attachment.

**Required Properties**

The required properties of an Attachment object are "usageType", "display", "contentType", "length", and "sha2". Three of these properties, "contentType", "length", and "sha2" describe specifics about the contents of the attachment. The other two indicate how this attachment is to be used related to the Statement's meaning.

The “contentType” is the RFC2046 media type (or MIME type) of the file, such as “application/pdf” or “text/plain” which instructs a system how the data can be parsed, etc. The “length” property’s value is an integer that specifies the size of the attachment in octets. The “sha2” property’s value is a string representing the SHA-2 hash of the contents and is ultimately what is used to uniquely identify an attachment listed within the statement with the file included in a request. The length of the string value can be used to determine which bit size algorithm was used to generate the hash.

The “display” property takes a language map as its value and gives a human readable name for the attachment similar to the same named property included in a verb. Lastly, the “usageType” property’s value must be a URI (IRI) and describes the “why” of the attachment. The “usageType” serves a similar purpose for Attachments as the “activityType” property does for Activities. (More about usage types below.)

**Optional Properties**

Along with the required properties, Attachment objects may also include a “description” property and a “fileUrl” property. The former is similar to the “display” property and takes a language map as a value. The language map provides a longer, human readable description of the purpose of the attachment or other information about it. The latter takes a URL from which the attachment’s data can be retrieved, or at least could have at one time. The “fileUrl” property is what makes it optional to include

the file contents themselves with requests including the Statement, however either the “fileUrl” or the file itself should be provided when storing the statement.

Here is an example of a Statement with an “attachments” property with a single Attachment object:

```
{
    …
    “attachments”: [
        {
            “contentType”: “application/pdf”,
            “usageType”: “http://id.tincanapi.com/attachment/certificate-of-
completion”,
            “display”: {
               “en-US”: “Completion of Tin Can API 101”
            },
            “description”: {
                “en-US”: “Certificate provided as proof of completion of Tin
Can API 101 course.”
            },
            “length”: 63878,
            “sha2”:
“c2a36cbc4db66444d05e134b85a89681f65263cacd93eb4a544f0bef058a5649”
        }
    ]
}
```

This example might be included by a course when sending a statement indicating completion of the training, and includes a printable certificate that the participant can provide for compliance reasons.

## Inclusion of Files on Requests

Just as with the basic parts of the REST interface the Tin Can specification piggybacks on existing, commonly used specifications for inclusion of file attachments in requests, specifically the multipart handling portion of the MIME standard via RFC 1341.

That can be a bit much to take in, so here are the fundamental parts. When including attachments as files in statement requests, the content type of the request becomes “multipart/mixed”. In a multipart/mixed request there will be multiple sections of content separated by block markers, each with a set of headers and a body. The first section (or part) will have the “application/json” content type and the body will contain the normal Statement(s) payload as requests without included files. Each subsequent part will include a special header, specifically the “X-Experience-API-Hash” header, whose value will match the SHA-2 stored in the “sha2” property of the Attachment object of the Statement’s “attachments” property (in other words, what I talked about above). This is how, for a given request, a system can match up the file included in the request with the metadata for that Attachment included in the Statement.

That covers the basics, but there are a lot of rules in the MIME standard about how boundaries between parts are composed, how headers and encodings should be handled, etc. I suggest using a well tested library for MIME handling. Additionally, there are a number of rules about how LRSs and Activity Providers

should act when encountering or sending requests with files. Those topics are really best covered by a deep read of the specifications themselves.

**Use Cases**

The completion certificate example is likely a common use case for attachments for the e-learning industry, but like the rest of the Tin Can API, there is virtually unlimited scope for what could be handled. Signed electronic contracts for real estate transactions or other types of legal exchanges could be attached to statements. As more retail stores switch to fully electronic operations, receipts could be sent attached to statements for a sale. The TCDraw early prototype captured a dynamically generated image showing a handwriting exercise, at the time attachment support was not yet in the Tin Can API specification so it uses Extensions, but could (and should) be updated to use attachments instead. Because Attachments include the hash of the contents as an identifier, the same attachment can be easily associated with more than one Statement. For instance you could send your résumé as an attachment when applying for a job, then the hiring manager may include it along with a signed employment contract notifying HR of a new employee. The contents of the file itself may only need to be sent once, but it could be referenced in multiple statements. These are just a couple of use cases, the possibilities are unlimited.

## Statement Signing

While each of the use cases above are fairly realistic, the specification includes one use case for attachments, specifically statement signing. A statement may be signed to guarantee the ability to verify authenticity, who is asserting the statement, and integrity, that the statement has not been altered. To do so the original content of the statement is serialized and included in the signature such that it can be later decoded and compared with the recorded statement for logical equivalence. The signature then gets its own entry in the “attachments” array and must have a “usageType” of “http://adlnet.gov/expapi/attachments/signature” per the specification. The signature is then included as a file using the normal attachment procedure. Appendix G of the specification contains an excellent example of what the signature, pre-signed and post-signed Statements look like. Though I’ve not personally seen examples of signed statements in the wild nor support in the libraries, I’m hoping that support is added soon (perhaps I’ll even get to it) and that signed statements start showing up.

## Registry for usage types

Attachment usage types are just one more URI that Tin Can users have to deal with as we’ve already seen with verbs, activity types, and extensions. To help facilitate interoperability and to make sure attachment usage types are resolvable, The Registry includes handling of “attachment usages”. It contains an ever growing list of usage types that others have already started using, naturally

including the statement signature one. In the event that one does not exist that fits your use case, you can easily request to coin a new one that will be added to the list. Usage types like the other shareable URIs are curated and if an existing alternative fits the bill, it may be suggested.

**Conclusion**

As we saw with other properties of a Statement in “Deep Dive: Extras/Others” the specification does an excellent job of capturing both sides of the metadata requirements when it comes to Attachments. It handles both the mechanical with properties like “contentType”, “length” and “sha2” as well as the meaningful with properties like “usageType” and “display”. Though many of the tools are still being developed to include support for attachments, as the adoption of Tin Can matures, more and more experiences will include the capturing of binary data. And for a number of reasons, not the least of which legal ones, statement signing has already been defined to take advantage of this feature to ensure a way to trust statements that have been recorded.

Go now, make statements!



# Ready to talk about Tin Can? Well, we want to talk to you!

http://tincanapi.com/talk