

xAPI Statements 101

A breakdown and definition of each component of an xAPI statement.

Actor Verb Object Verbs vs. Activities Context **Result** Extensions

Other Fields

Experience API: Statements 101

In the [technical introduction to xAPI](#), we outlined some basic concepts about statements. Here we'll take a deeper look into the structure of statements, and discover their utility in capturing experiences.

A quick note: when developing your own xAPI statements, the [xAPI Registry](#) will be a valuable resource. It contains resolvable URIs that are the building blocks of xAPI statements. If you don't find the pieces that you're looking for in [The Registry](#), [let us know](#). We'll work with you to get The Registry updated.

At the simplest level, xAPI statement structure can be expressed in the form of "actor verb object". An example of this sort of statement is "Sally experienced 'Solo Hang Gliding'". In the technical introduction, we saw the JSON format of this statement as

```
{
  "actor": {
    "name": "Sally Glider",
    "mbox": "mailto:sally@example.com"
  },
  "verb": {
    "id": "http://adlnet.gov/expapi/verbs/experienced",
    "display": { "en-US": "experienced" }
  },
  "object": {
    "id": "http://example.com/activities/solo-hang-gliding",
```

```

    "definition": {
      "name": { "en-US": "Solo Hang Gliding" }
    }
  }
}

```

Using this example as a base, let's start to explore and expand on the elements of xAPI statements.

Actor

As we conducted the research involved with Project Tin Can, one sentiment that we heard repeatedly in our interviews with industry leaders was this: Learning is becoming, and should become, increasingly person-centric. We discovered that people, not companies, were more interested in the long-term ownership of their experiences and results.

In xAPI, people don't have to be solely identified by one system or ID. That's an important step in making xAPI person-centric. Imagine a consolidated view of your activity across all the systems with which you interact. Though you tweeted using @yourhandle on Twitter, when you see the whole of your activity, you see the experience more in the terms that you tweeted, the same "you" that took 'Solo Hang Gliding' and the same "you" that read Hang Gliding Training Manual. Due to privacy concerns, each statement will contain only one representation of "you", however reporting systems that are aware of these multiple personas may aggregate them.

With that explanation in mind, let's look at our example actor object:

```

{
  "name": "Sally Glider",
  "mbox": "mailto:sally@example.com"
}

```

This actor object has two properties, "name" and "mbox". Only "mbox" uniquely describes this Sally. There may be many people out there with the name Sally, but only one of them owns the email address sally@example.com.

Let's consider another way to represent Sally:

```

{
  "name": "Sally Glider",

```

```

    "account": {
      "homePage": "http://twitter.com",
      "name": "sallyglider434"
    }
  }
}

```

In this case, we are identifying Sally by her twitter account with the handle “sallyglider434”. This flexibility in identifying people is an important feature of xAPI, and you can read more about this in our [Actor/Agent Deep Dive post](#). We’ll also talk about representing systems (not people) and groups as actors as well. For now, those details can be found in the [Experience API specification](#).

▲ [Back to top](#)

Verb

Verbs in xAPI are URIs, and should be paired with a short display string. They are a crucial element of statements, as they describe just what has happened between the actor and object of the statement. The xAPI specification (1.0.0 at the time of this writing) allows any full URI to be used as a verb. We’ve built the [Experience API Registry](#) as a place to store and add verbs, along with a place for your verbs’ URIs to resolve. There is an initial set of verbs published by ADL, and you can find them in the [xAPI Registry](#). It’s certainly a good idea to at least consider these verbs before creating your own. That list includes “experienced”, “attended”, “attempted”, “completed”, “passed”, “failed”, “answered”, “interacted”, “imported”, “created”, “shared”, and “voided”. For more on verbs, [check out our deep-dive post on verbs](#). A basic verb example:

```

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

```

▲ [Back to top](#)

Object

Wrapping up the core statement structure, let’s consider the “object” field. Typically the object will be a xAPI activity, though it might be another actor, and

in the case of voiding, another statement. For now, we'll keep our focus on activities as objects.

While actors in xAPI are usually related to existing people, and verbs tend to have a clear existing definition, activities are more likely to be defined and provided by the systems reporting statements. All activities must be uniquely defined by an URI, and can optionally include descriptive information. Let's take another look at the object from our example statement:

```
{
  "id": "http://example.com/activities/solo-hang-gliding",
  "definition": {
    "name": { "en-US": "Solo Hang Gliding" }
  }
}
```

This is an activity uniquely identified by its "id" field. It's an important principle of xAPI that no two activities are ever referenced by the same ID. By definition an ID corresponds one to one with the logical activity it identifies. It is possible for that logical definition of an activity to become muddled if identifiers are chosen poorly, eg: "hang gliding, swimming, and question #4", but it is still one logical activity. For this reason, creators of activity IDs must be careful to only create activity IDs using domains they control, or have been given a path within that domain to control, and must establish a scheme to ensure uniqueness within that domain. Note that we're using example.com here. Unless you are also making example statements which are never intended to convey any real meaning to anyone, don't do that.

The definition for an activity object can be refined over time (though it shouldn't ever be changed significantly enough to describe what should have been some other new activity). Descriptive fields in the activity definition offer a way for xAPI to natively support internationalization. Let's add some more info to our example object as an example:

```
{
  "id": "http://example.com/activities/solo-hang-gliding",
  "definition": {
    "type": "http://adlnet.gov/expapi/activities/course",
    "name": {
      "en-US": "Solo Hang Gliding",
      "es": "Solo Ala Delta"
    },
    "description": {
      "en-US": "The 'Solo Hang Gliding' course provided by The Hang Glider's Club",
      "es": "El curso de 'Solo Ala Delta' siempre por el Club de
```

```

Planeadores Hang"
    },
    "extensions": {
        "http://example.com/gliderClubId": "course-435"
    }
}
}
}

```

Here we've added a "type" field, indicating the activity is a course on solo hang gliding. We've also added a "description" field, which as expected, contains a short description of the course. In the case of "name" and "description", we've added the Spanish version (care of Google Translate). This way, reporting tools or other display layers could automatically switch the language output based on a user's preferences.

We've also added to the "extensions" field, where we can put arbitrary key (URI) / value pairs that are custom to a particular application (or convention). We'll talk more about extensions below. Full details for activity definitions can be found in the [Experience API specification](#).

▲ [Back to top](#)

Verbs vs. Activities

Sometimes the line between a verb and an object can become blurry. In our example statement about Sally, there's an implication that "Solo Hang Gliding" is a single well defined activity. Perhaps it's a course or test defined by the hang gliding school from example.com. But what if we wanted to say "Sally hang-glided over Mount Magazine"? Is "hang gliding over Mount Magazine" a specific activity that Sally is "experiencing"? Or did Sally "hang glide", with "Mount Magazine" as the object of the statement?

Usually, resolving this ambiguity will rely on the intentions in reporting the data, and the relation of these pieces to other statements. In a system that will track many activities on Mount Magazine, it may improve the data to isolate Mount Magazine as an object. You'll also be able to more easily isolate all of Sally's hang gliding experiences across many different objects, if you can filter on the verb "hang gliding".

In other cases, "hang gliding over Mount Magazine" might make sense as a single activity if it's been defined by some organization or specification, and means something very specific in terms of instruments used, path flown, and so on. In that case, what's important is that "Sally experienced 'Hang Gliding over

Mount Magazine” (which has been defined specifically as The Hang Glider Club’s first hang gliding test).

For a listing of verbs and activities, check out the [xAPI Registry](#). If you don’t find what you’re looking for there, [let us know](#), and we’ll work with you to get the Registry updated.

▲ [Back to top](#)

Context

The “context” field provides a place to add some contextual information to a statement. We can add information such as the instructor for an experience, if this experience happened as part of a team activity, or how an experience fits into some broader activity. Let’s explore some of these elements in our example statement:

```
{
  "actor": {
    "name": "Sally Glider",
    "mbox": "mailto:sally@example.com"
  },
  "verb": {
    "id": "http://adlnet.gov/expapi/verbs/experienced",
    "display": { "en-US": "experienced" }
  },
  "object": {
    "id": "http://example.com/activities/solo-hang-gliding",
    "definition": {
      "type": "http://adlnet.gov/expapi/activities/course",
      "name": { "en-US": "Solo Hang Gliding" }
    }
  },
  "context": {
    "instructor": {
      "name": "Irene Instructor",
      "mbox": "mailto:irene@example.com"
    },
    "contextActivities": {
      "parent": { "id": "http://example.com/activities/hang-gliding-class-a" },
      "grouping": { "id": "http://example.com/activities/hang-gliding-school" }
    }
  }
}
```

In this example, we’ve added a “context” object to the statement. We’re stating that “Sally took the ‘Solo Hang Gliding’ course, under the instruction of Irene, as

part of Hang Gliding Class A, within the context of Hang Gliding School". Whew, now that's a mouthful, but we've added some useful information here. These pieces of context are especially useful in the way statements can be grouped when reporting. The native query API in xAPI allows us to go back and ask for all statements made under Irene's instruction, or to see all statements made from participants of Hang Gliding Class A, or even all statements made within the Hang Gliding School. In a more sophisticated reporting tool, these elements could help us achieve a very logical grouping of the data.

Though not shown here, context objects also have an "extensions" field, allowing arbitrary data to be attached as context for the statement. This data can then be used in specialized reporting tools when the statements are retrieved. We'll talk further about extensions below. The full set of context elements can be found in the [Experience API specification](#).

▲ [Back to top](#)

Result

A statement can also end in some measured outcome. For example, if 'Solo Hang Gliding' is a course or an assessment, we could state that "Sally completed 'Solo Hang Gliding' with a passing score of 95%", or "Sally completed 'Solo Hang Gliding' with a failing score of 2 out of 6", or even "Sally completed 'Solo Hang Gliding' in 4 hours". Let's add a result to our example statement:

```
{
  "actor": {
    "name": "Sally Glider",
    "mbox": "mailto:sally@example.com"
  },
  "verb": {
    "id": "http://adlnet.gov/expapi/verbs/completed",
    "display": {"en-US": "completed"}
  },
  "object": {
    "id": "http://example.com/activities/solo-hang-gliding",
    "definition": {
      "name": { "en-US": "Solo Hang Gliding" }
    }
  },
  "result": {
    "completion": true,
    "success": true,
    "score": {
      "scaled": .95
    }
  }
}
```

```
}  
}
```

Here we've stated that "Sally completed 'Solo Hang Gliding' with a passing score of 95%". The "completion" field tells us that Sally is done, and the "success" field tells us that she passed. The "score" field gives us the 95% figure. In the case of "score", the value is actually a score object, and it could be expressed in terms of a minimum, maximum, and raw value instead. This means we could report a score such as "2 out of 6", using a minimum of 0, a maximum of 6, and a raw value of 2.

We can also record a duration in the result, which tells us how long the activity leading to the result actually took. And in the case of a statement that records an interaction between the actor and the object, the result can have the actor's response. In that way we could say "Sally answered 'Question A' with the response 'Choice B' in 8 seconds, and was correct". Details about interaction activities and their responses can be found in the full xAPI specification. Here again xAPI provides an "extensions" field, this time on the result object, to let applications define their own custom results. Speaking of which...

▲ [Back to top](#)

Extensions

Throughout this tour of xAPI statements, we've seen several occurrences of the "extensions" field. Extensions are available as part of activity definitions, as part of statement context, or as part of some statement result. In each case, they're intended to provide a natural way to extend those elements for some specialized use. The contents of these extensions might be something valuable to just one application, or it might be a convention used by an entire community of practice.

Since "extensions" fields can contain arbitrary data, they unlock a lot of flexibility in what you can express in a statement. But, as the old adage goes, "With great power comes great responsibility". Using extensions responsibly means a few things.

First, though you could use extensions to pack in a lot of application-specific data, but it doesn't always mean you should. A statement shouldn't be totally defined by its extensions, and be meaningless otherwise. xAPI statements should be capturing experiences among actors and objects, and should always strive

to map as much information as possible into the built in elements, in order to leverage interoperability among xAPI conformant tools.

Second, extensions should logically relate to the part of the statement where they are present. Extensions in statement context should provide context to the core experience, while those in the result should provide elements related to some outcome. For activities, they should provide additional information that helps define an activity within some custom application or community.

Finally, the key in extension key/value pairs must be a URI, just like a verb or activity ID. Like a verb or activity ID, it is defined as being unique, which is to say that when you use an extension “key” you are asserting that it has the same meaning as when anyone else uses that key.

With all of that in mind, let’s throw a few extensions onto our example statement:

```
{
  "actor": {
    "name": "Sally",
    "mbox": "mailto:sally@example.com"
  },
  "verb": {
    "id": "http://adlnet.gov/expapi/verbs/completed",
    "display": { "en-US": "completed" }
  },
  "object": {
    "id": "http://example.com/activities/solo-hang-gliding",
    "definition": {
      "type": "http://adlnet.gov/expapi/activities/assessment",
      "name": { "en-US": "Solo Hang Gliding" },
      "extensions": {
        "http://example.com/gliderClubId": "test-435"
      }
    }
  },
  "result": {
    "completion": true,
    "success": true,
    "extensions": {
      "http://example.com/flight/averagePitch": 0.05
    }
  },
  "context": {
    "extensions": {
      "http://example.com/weatherConditions": "rainy"
    }
  }
}
```

In this case, we've added information to define the 'Solo Hang Gliding' assessment within the Hang Glider Club's community using the "gliderClubId" field. We've noted an important piece of context about this experience in the form of the "weatherConditions" field. And finally, we've included a custom metric in the outcome of the statement using the "averagePitch" field. Even without these extra pieces of information, the core statement remains meaningful: "Sally passed 'Solo Hang Gliding'". The extra information just enables more meaning in specialized environments.

We have a list of extensions in our [xAPI Registry](#). Feel free to use the registry to find the extensions you'd like to use, or [request new extensions to be added](#).

▲ [Back to top](#)

Other statement fields

We've pretty much wrapped up our tour of major elements in a xAPI statement. Here we'll take a quick look at a few other small but important pieces of statements.

The "timestamp" and "stored" fields are critical elements of statements. The first records when the experience occurred, while the second records when the experience was added to the xAPI LRS where this statement has been stored. The "stored" field is hence always set by the LRS, while the "timestamp" field can and should be set by the system originating the statement.

The "authority" field captures information about who or what has made this statement. The value for this field is an actor, and could represent a person such as "Sally" or "Irene", or a thing such as "The Glider Field Application". Knowing who made the statement is crucial information when considering statement validity, and is also useful as a form of audit trail. Much more will be said regarding authorization, authentication, and permissions in an upcoming "xAPI Security 101" article. More information can be found today by digging into the [xAPI specification](#).

Finally, the "voided" field on a statement tells us if this statement is no longer valid information. In order to support the distributed nature of xAPI, statements are considered logically immutable. That is, once a statement has been stored in an LRS, there is no way to alter or change that statement. But it's constricting to require that all statement are valid for all of time once made. So, to support invalidating them, as mistakes or otherwise, xAPI provides a mechanism to void

some previous statement using a new statement. Details on voiding statements can be found in the [xAPI specification](#).

That concludes our tour of xAPI statements. If you'd like to easily create some of your own statements, make sure to check out the [Statement Generator](#).

Let's end things off with an information packed version of our example statement, which before this might have seemed quite scary, but should now be discernible:

```
{
  "actor": {
    "name": "Sally Glider",
    "mbox": "mailto:sally@example.com"
  },
  "verb": {
    "id": "http://adlnet.gov/expapi/verbs/completed",
    "display": { "en-US": "completed" }
  },
  "object": {
    "id": "http://example.com/activities/hang-gliding-test",
    "definition": {
      "type": "http://adlnet.gov/expapi/activities/assessment",
      "name": { "en-US": "Hang Gliding Test" },
      "description": {
        "en-US": "The Solo Hang Gliding test, consisting of a
timed flight from the peak of Mount Magazine"
      },
      "extensions": {
        "http://example.com/gliderClubId": "test-435"
      }
    }
  },
  "result": {
    "completion": true,
    "success": true,
    "score": {
      "scaled": 0.95
    },
    "extensions": {
      "http://example.com/flight/averagePitch": 0.05
    }
  },
  "context": {
    "instructor": {
      "name": "Irene Instructor",
      "mbox": "mailto:irene@example.com"
    },
    "contextActivities": {
      "parent": { "id": "http://example.com/activities/hang-
gliding-class-a" }
      "grouping": { "id": "http://example.com/activities/hang-
gliding-school" }
    }
  }
}
```

```
    },
    "extensions": {
      "http://example.com/weatherConditions": "rainy"
    }
  },
  "timestamp": "2012-07-05T18:30:32.360Z",
  "stored": "2012-07-05T18:30:33.540Z",
  "authority": {
    "name": "Irene Instructor",
    "mbox": "mailto:irene@example.com"
  }
}
```

▲ [Back to top](#)

Dive deeper

Next steps

Use the resources below to learn more about Statements.

- [Statement Explorer](#) is an interactive example statement that explains the high level structure of the statement.
- [Statements Deep Dive](#) takes you beyond the basics to help you get the most out of xAPI.

Questions? Ask us anything.

At Rustici Software, we help hundreds of people each month with their xAPI questions. Many aren't sales prospects; they just have questions. We're happy to help. You can ask us anything – really.

- * First name

--

- * Last name

--

* Business email

--


- * Company name

--

Phone number

--	--

* Your message



- * What are you trying to do?

By checking this box you permit Rustici Software to follow up by email or phone in accordance with the [Privacy Policy](#). You will be able to unsubscribe anytime.

Submit

Technical

Overview

Statements 101

Build an LRS



Questions?

If you have a question about xAPI, we're here to help.

[Contact us](#)

Rustici Software

We help eLearning platforms work well together by supporting the standards.

[About our company](#)

eLearning Standards

Learn how Rustici Software supports each

Contact us

Ask us anything. We're here to help.

[Our open job positions](#)
[About our standards' expertise](#)
[Our products](#)
[Technical documentation](#)

[of the following standards:](#)

[SCORM](#) [xAPI](#)
[cmi5](#) [AICC](#)
[LTI](#)

[Contact us](#)
[Subscribe to newsletter](#)
[Get support](#)

[Call us: \(866\) 497-2676](#)
[International: +1 \(615\) 376-9867](#)

[Rustici Software](#)

[Rustici Software](#)
[Rustici Software](#)

[Rustici Software](#)

[xAPI](#)

[xAPI](#)

© Copyright 2025, Rustici Software LLC
[Privacy & Cookies Policy](#) [Corporate Responsibility](#)

Content on this site is licensed under a [Creative Commons Attribution 3.0 License](#)..



Select Language | ▼