Experience API

Advanced Distributed Learning (ADL) Co-Laboratories

19 October 2012

Version 0.95

Last Revised: Oct 11 10:33:00 UTC 2012

This document was authored by members of the Experience API Working Group (see list on pp. 7-8) in support of the Office of the Deputy Assistant Secretary of Defense (Readiness) Advanced Distributed Learning (ADL) Initiative. Please send all feedback and inquiries to helpdesk@adlnet.gov

Table of Contents

1.0 Revision History: 5
2.0 Role of the Experience API 6
2.1 ADL's Role in the Experience API 6
2.2 Contributors 6
2.2.1 Working Group Participants 6
2.2.2 Requirements Gathering Participants 7
3.0 Definitions 8
Tin Can API (TCAPI) 8
4.0 Statement 9
4.1 Statement Properties: 9
4.1.1 ID: 12
4.1.2 Actor: 12
4.1.3 Verb: 14
4.1.4 Object: 15
4.1.5 Result: 19
4.1.6 Context: 20
4.1.7 Timestamp: 22
4.1.8 Stored: 22
4.1.9 Authority: 22
4.1.10 Voided: 23
4.2 Retrieval of Statements: 24
5.0 Miscellaneous Types 26
5.1 Document: 26
5.2 Language Map 26
5.3 Extensions 26
6.0 Runtime Communication 27
6.1 Encoding: 27
6.2 Version Header: 27
6.3 Concurrency: 27
6.4 Security: 28
6.4.1 Authentication Definitions: 28
6.4.2 OAuth Authorization Scope 29
7.0 Data Transfer (REST) 31
7.1 Error Codes 31
7.2 Statement API: 31
7.3 State API: 33
7.4 Activity Profile API: 35

7.5 Agent Profile API: 36
7.6 Cross Origin Requests: 37
7.7 Validation: 38
Appendix A: Bookmarklet 39
Appendix B: Creating an “IE Mode” Request 41
Appendix C: Example definitions for activities of type “cmi.interaction” 42

1.0 Revision History:

0.8 (Project Tin Can API Deliverable) to 0.9 (March 31, 2012):

Rustici software, whom delivered Project Tin Can API, made modifications to the API prior to the April 2012 Kickoff Meeting. It was voted in this meeting to move those changes into the current spec and revision to 0.9.

0.90 to 0.95 (August 31, 2012):

“Core” verbs and activity types were removed from the specification. References to these verbs in results, context, interactions, and activity definitions have also been removed. It is recommended that implementers prefer community defined verbs to creating their own verbs.

- Verbs, activity types, and extension keys are now URIs
- Restructured and added language around some of the other implementation details and scope.
- Changed from using a person-centric view of agents to a persona-centric view.
- Friend of a Friend (FOAF) agent merging requirement removed.
- Agent objects must now have exactly 1 uniquely identifying property, instead of at least one.

2.0 Role of the Experience API

The Experience API is a service that allows for statements of experience (typically learning experiences, but could be any experience) to be delivered to and stored securely in a Learning Record Store. The Experience API is dependent on Learning Activity Providers to create and track learning experiences; this specification provides a data model and associated components on how to accomplish these tasks.

Specifically, the Experience API provides:

- Structure and definition of statement, state, learner, activity and objects, which are the means by which experiences are conveyed by a Learning Activity Provider.
- Data Transfer methods for the storage and retrieval (but not validation) of these objects to/from a Learning Record Store. Note that the systems storing or retrieving records need not be Learning Activity Providers. LRSs may communicate with other LRSs, or reporting systems.
- Security methods allowing for the trusted exchange of information between the Learning Record Store and trusted sources.

The Experience API is the first of many potential specifications that will merge to create a higher architecture of online learning and training. Authentication services, querying services, visualization services, and personal data services are some examples of additional components that the Experience API is designing to work alongside. While the implementation details of these services are not specified here, the Experience API is designed with these components in mind.

2.1 ADL's Role in the Experience API

ADL has taken a role of steward and facilitator in the development of the Experience API. The Experience API is seen as one piece of the ADL Training and Learning Architecture, which facilitates learning anytime and anywhere. ADL views the Experience API as an evolved version of SCORM that can support similar use cases, but can also support many of the use cases gathered by ADL and submitted by those involved in distributed learning which SCORM could not enable.

2.2 Contributors

My thanks to everyone who contributed to the Experience API project. Many of you have called into the weekly meetings and helped to shape the specification into something that is useful for the entire distributed learning community. Many of you assisted in releasing code samples, products, and documentation to aid those who are creating and adopting the specification. I'd also like to thank all of those who were involved in supplying useful, honest information about your organization's use of SCORM and other learning best practices. Through the use-cases, shared experiences, and knowledge you have shared, ADL and the community clearly identified the first step in creating the Training and Learning Architecture--the Experience API. You are truly the community leaders on which we depend to make our training and education the very best.

Kristy S. Murray, Ed.D.
Director, ADL Initiative|
OSD, Training Readiness & Strategy (TRS)

2.2.1 Working Group Participants

Name:	**Organization:**
Aaron Silvers	ADL
Jonathan Poltrack	ADL
Al Bejcek	NetDimensions
Ali Shahrazad	SaLTBOX
Andrew Downes	
Andy Johnson	ADL

Andy Whitaker	Rustici Software
Anthony Altieri	American Red Cross
Anto Valan	Omnivera Learning Solutions
Avron Barr	Aldo Ventures, Inc.
Ben Clark	Rustici Software
Bill McDonald	Boeing
Brian J. Miller	Rustici Software
Chad Udell	Float Mobile Learning
Dan Allen	Litmos
Dan Kuemmel	Sentry Insurance
Dave Mozealous	Articulate
David Ells	Rustici Software
Eric Johnson	Planning and Learning Technologies, Inc.
Fiona Leteney	Feenix e-learning
Greg Tatka	Menco Social Learning
Ingo Dahn	University Koblenz-Landau
Jason Haag	ADL
Jeff Place	Questionmark
Jennifer Cameron	Sencia Corporate Web Solutions
Jeremy Brockman	
Joe Gorup	CourseAvenue
John Kleeman	Questionmark
Kris Miller	edcetra Training
Kris Rockwell	Hybrid Learning Systems
Lang Holloman	
Lou Wolford	ADL
Luke Hickey	dominKnow
Marcus Birtwhistle	ADL
Mark Davis	Exambuilder
Megan Bowe	Rustici Software
Melanie VanHorn	ADL
Michael Flores	Here Everything's Better
Mike Palmer	OnPoint Digital
Mike Rustici	Rustici Software
Nik Hruska	ADL
Patrick Kedziora	Kedzoh
Paul Roberts	Questionmark
Rich Chetwynd	Litmos
Richard Fouchaux	Ontario Human Rights Commission
Richard Lenz	Organizational Strategies, Inc.
Rick Raymer	Serco
Rob Chadwick	ADL
Robert Lowe	
Russell Duhon	SaLTBOX
Stephen Trevorrow	Problem Solutions, LLC.
Steve Baumgartner	
Steve Flowers	XPConcept
Thomas Ho	
Tim Martin	Rustici Software
Tom Creighton	ADL
Walt Grata	ADL

2.2.2 Requirements Gathering Participants

In collection of requirements for the Experience API, there were many people and organizations that provided invaluable feedback to SCORM, distributed learning efforts, and learning in general. User Voice Site, Rustici Blog, etc.

3.0 Definitions

Experience API (XAPI): The API defined in this document, the product of "Project Tin Can API". A simple, lightweight way for any permitted actor to store and retrieve extensible learning records, learner and learning experience profiles, regardless of the platform used.

Tin Can API (TCAPI): The previous name of the API defined in this document.

Learning Activity Provider (AP): The software object that is communicating with the LRS to record information about a learning experience. May be similar to a SCORM package as it is possible to bundle assets with the software object that does this communication, but may also be separate from the experience it is reporting about.

Learning Activity (activity): Like a SCORM Activity, a unit of instruction, experience, or performance that is to be tracked.

Statement: A simple statement consisting of <Actor (learner)> <verb> <object>, with <result>, in <context> to track an aspect of a learning experience. A set of several statements may be used to track complete details about a learning experience.

Learning Record Store (LRS): A system that stores learning information. Currently, most LRSs are Learning Management Systems (LMSs), however this document uses the term LRS to be clear that a full LMS is not necessary to implement the XAPI. The XAPI is dependent on an LRS to function correctly.

Learning Management System (LMS): Provides the tracking functionality of an LRS, but provides additional administrative and reporting functionality. In this document the term will be used when talking about existing systems that implement learning standards. The XAPI can work independently of an LMS, but is built with knowledge of the suite of services an LMS provides.

Service: A software component responsible for one or more aspects of the distributed learning process. An LMS typically combines many services to design a complete learning experience.

Registration: If the LRS is an LMS, it likely has a concept of registration, an instance of a learner signing up for a particular learning activity. The LMS may also close the registration at some point when it considers the learning experience complete. For Experience API purposes, a registration may be applied more broadly; an LMS could assign a group of students to a group of activities and track all related statements in one registration. Note: activity providers are cautioned against reporting registration other than when assigned by an LRS. An LRS that assigns registrations is likely to reject statements containing unassigned registration IDs.

State: Similar to SCORM suspend data, but allows storage of arbitrary key/document pairs. The LRS does not have to retain state once the learning experience is considered "done" (LRS has closed its "registration").

Profile: A construct where information about the learner or activity is kept, typically in name/document pairs that have meaning to an instructional system component.

Authentication: The concept of verifying the identity of a user or system. This allows interactions between the two "trusted" parties.

Authorization: The affordance of permissions based on a user or system's role: the process of making one user or system "trusted" by another.

Community of Practice: A group, usually connected by a common cause, role or purpose, which operates in a common modality.

4.0 Statement

The statement is the core of the XAPI. All learning events are stored as statements such as: “I did this”.

4.1 Statement Properties:

Actor, verb, and object are required, all other properties are optional. Properties can occur in any order, but are limited to one use each. Each property is discussed below.

Property	Type	Default	Description
id	UUID		UUID assigned by LRS or other trusted source.
actor	Object		Who the statement is about, as an Agent or Group object. 'I'
verb	Object		Action of the Learner or Team object. “Did”.
object	Object		Activity, agent, or another statement that is the object of the statement, “this”. Note that objects which are provided as a value for this field should include a “objectType” field. If not specified, the object is assumed to be an activity.
result	Object		Result object, further details relevant to the specified verb.
context	Object		Context that gives the statement more meaning. Examples: Team actor is working with, altitude in a flight simulator.
timestamp	Date/Time		Timestamp (Formatted according to ISO 8601) of when what this statement describes happened. If not provided, LRS should set this to the value of "stored" time.
stored	Date/Time		Timestamp (Formatted according to ISO 8601) of when this statement was recorded. Set by LRS.
authority	Object		Agent who is asserting this statement is true. Verified by LRS based on authentication, and set by LRS if left blank.
voided	Boolean	false	Indicates that the statement has been voided (see below)

Aside from (potential or required) assignments of properties during initial processing ("id", "authority", "stored", "timestamp"), and the special case of updating the “voided” flag, statements are immutable. Note that the content of activities that are referenced in statements are not considered part of the statement itself. So while the statement is immutable, the activities referenced by that statement are not. This means a deep serialization of a statement into JSON will change if the referenced activities change.

Example of a simple statement:

```
{
        "id":"fd41c918-b88b-4b20-a0a5-a4c32391aaa0",
```

```
        "actor":{
                "objectType": "Agent",
                "name":"Project Tin Can API",
                "mbox":"mailto:user@example.com"
        },
        "verb":{
                "id":"http://adlnet.gov/expapi/verbs/created",
                "display":{ "en-US":"created" }
        },
        "object":{
                "id":"http://example.adlnet.gov/xapi/example/simplestatement",
                "definition":{
                "name":{ "en-US":"simple statement" },
                "description":{ "en-US":"A simple Experience API statement. Note that the LRS
                does not need to have any prior information about the actor (learner), the
                verb, or the activity/object." }
                }
        }
}
```

Simplest possible statement using all properties that MUST or SHOULD be used:

```
{
        "actor":{
                "objectType": "Agent",
                "mbox":"mailto:xapi@adlnet.gov"
        },
        "verb":{
                "id":"http://adlnet.gov/expapi/verbs/created",
                "display":{
                        "en-US":"created"
                }
        },
        "object":{
```

```
                "id":"http://example.adlnet.gov/xapi/example/activity"
        }
}
```

Typical simple completion with verb “attempted”:

```
{
        "actor":{
                "objectType": "Agent",
                "name":"Example Learner",
                "mbox":"mailto:example.learner@adlnet.gov"
        },
        "verb":{
                "id":"http://adlnet.gov/expapi/verbs/attempted",
                "display":{
                "en-US":"attempted"
                }
        },
        "object":{
                "id":"http://example.adlnet.gov/xapi/example/simpleCBT",
                "definition":{
                        "name":{
                                "en-US":"simple CBT course"},
                        "description":{
                                "en-US":"A fictitious example CBT course."
                        }
                }
        },
        "result":{
                "score":{
                        "scaled":0.95
                },
                "success":true,
```

```
                "completion":true

        }

}
```

4.1.1 ID:

The statement ID is a UUID which MAY be generated by the Learning Activity Provider. If a statement is posted without an ID, the LRS MUST assign one.

4.1.2 Actor:

The actor field contains an Agent or Group object, loosely inspired by Friend Of A Friend (FOAF, http://xmlns.com/foaf/spec/#term_Agent), a widely accepted vocabulary for describing identifiable individuals and groups

4.1.2.1 Agent:

An Agent object is identified by an email address (or its hash), OpenID, or account on some system (such as twitter), but only for values where any two Agents that share the same identifying property definitely represent the same identity. The term used for properties with that characteristic is "inverse functional identifiers". In addition to the standard inverse functional properties from FOAF of mbox, mbox_sha1sum, and openid, account is an inverse functional property in XAPI Agents.

For reasons of practicality and privacy, TCAPI Agents MUST be identified by one and only one inverse functional identifier. Agents MUST NOT include more than one inverse functional identifier. If an Activity Provider is concerned about revealing identifying information such as emails, it SHOULD instead use an account with an opaque account name to identify the person.

The table below lists all properties of Agent objects. Inverse functional identifiers are marked with a *."

Property	Description
objectType	"Agent" (Optional, except when used as a statement's object)
name	String (Optional)
mbox*	String in the form "mailto:email address". (Note: Only emails that have only ever been and will ever be assigned to this Agent, but no others, should be used for this property and mbox_sha1sum).
mbox_sha1sum*	String containing the SHA1 hash of a mailto URI (such as goes in an mbox property). An LRS MAY include Agents with a matching hash when a request is based on an mbox.
openid*	The URI of an openid that uniquely identifies this agent.
account*	An account object, see below.

Account

Property	Description
homePage	The URI to the canonical home page for the system the account is on. This is based on FOAF's accountServiceHomePage.
name	The unique ID or name used to log in to this account. This is based on FOAF's accountName.

An example using an opaque account:

```
{
        "objectType": "Agent",
        "account": {
                "homePage": "http://www.example.com",
                "name": "1625378"
        }
}
```

Agents are also important in relation to OAuth. See the section on OAuth for details.

4.1.2.1 Group:

Groups are similar to Agents, represent collections of Agents, and can be used most places Agents can. Groups can either be anonymous or identified. Anonymous Groups MUST include a member property listing constituent Agents. Systems consuming Statements MUST consider all anonymous Groups distinct. Anonymous Groups are useful for describing collections of people where no ready identifier for the group is available, such as ad hoc teams.

Identified Groups MUST, like Agents, include exactly one inverse functional identifier. Identified Groups MAY also include a member property listing constituent Agents. Inverse functional identifiers used for identified Groups SHOULD NOT be used for any Agents.

Systems consuming Statements MUST NOT assume member Agents comprise an exact list of agents in an anonymous or identified Group.

Anonymous Group

Property	Description
objectType	"Group" (Required)
name	String (Optional)

member	(array of) Agent (not Group) objects representing members of this Group.

Identified Group

Property	Description
objectType	"Group" (Required)
name	String (Optional)
mbox*	String in the form "mailto:email address". (Note: Only emails that have only ever been and will ever be assigned to this Agent, but no others, should be used for this property and mbox_sha1sum).
mbox_sha1sum*	String containing the SHA1 hash of a mailto URI (such as goes in an mbox property). An LRS MAY include Agents with a matching hash when a request is based on an mbox.
openid*	The URI of an openid that uniquely identifies this agent.
account*	An account object, see above.
member	(array of) Agent (not Group) objects representing members of this Group.

4.1.3 Verb:

A verb defines what the action is between actors, activities, or most commonly, between an actor and activity. The Experience API does not specify any particular verbs, but rather defines **how** verbs are to be created. It is expected that verb lists exist for various communities of practice. Verbs appear in statements as objects consisting of a URI and a set of display names.

The Verb URI should identify the particular semantics of a word, not the word itself. For example, the English word "fired" could mean different things depending on context, such as "fired a weapon", "fired a kiln", or "fired an employee". In this case, a URI should identify one of these specific meanings, not the word "fired".

The Experience API does not specify any particular verbs (except the reserved ""http://adlnet.gov/expapi/verbs/voided"), but rather defines how verbs are to be used. Communities of practice will develop verbs they find useful and make them available to the general community for use.

A verb in the Experience API is a URI, and denotes a specific meaning not tied to any particular language. For example, a particular verb URI such as http://example.org/firearms#fire or tag:example.com,2012:xQr73H might denote the action of firing a gun, or the verb URI http://example.com/خواند دن/فعل might denote the action of reading a book.

The person who comes up with a new verb should own the URI, or have permission from the owner to use the URI to denote a Experience API verb. The owner of a URI SHOULD make a human-readable description of the intended usage of the verb accessible at the URI.

* NOTE: In some future version, this specification might specify additional machine-readable information about the verb be made available, but the choice to do so is postponed to monitor emerging practices and pain points. ADL plans to release a set of recommended verbs at the same time as this specification. Learning Activity Providers MAY use one these verbs, or

other verb which have wide adoption, if applicable. The verb list to be created by ADL will include verbs corresponding to the verbs previously defined in this specification. If the meaning of one of those verbs is intended, Learning Activity Providers SHOULD use the corresponding ADL verb. Learning Activity Providers MAY create their own verbs instead, as needed.

4.1.3.1 Verb Object:

The verb object is the representation of a verb that is actually included in a statement. In addition to referencing the verb itself via a URI, it includes a display property which provides the human-readable meaning of the verb in one or more languages.

The display property MUST NOT be used to alter the meaning of a statement, rather it MUST be used to illustrate the meaning which is already determined by the verb URI.

All statements SHOULD use the display property. A system reading a statement MUST NOT use the display property to infer any meaning from the statement, rather it MUST use the verb URI to infer meaning, and the display property only for display to a human.

Property	Description	Example
id	A URI that corresponds to a verb definition. Each verb definition corresponds to the meaning of a verb, not the word. A URI should be human-readable and contain the verb meaning.	www.adlnet.gov/XAPIprofile/ran(travelled_a_distance)
display	A language map containing the human readable representation of the verb in at least one language. This does not have any impact on the meaning of the statement, but only serves to give a human-readable display of the meaning already determined by the chosen verb.	display : { "en-US" : "ran"} display : { "en-US" : "ran", "es" : "corrió" }

4.1.4 Object:

The object of a statement is the Activity, Agent, or Statement that is the object of the statement, “this”. Note that objects which are provided as a value for this field should include an “objectType” field. If not specified, the object is assumed to be an Activity.

4.1.4.1 – Activity as “object”

A statement may represent a Learning Activity as an object in the statement.

Property	Description
objectType	Should always be “Activity” when present. Used in cases where type cannot otherwise be determined, such as the value of a statement’s “object” field.
id	URI. If a URL, the URL should refer to metadata for this activity.

definition	Metadata, See below

Activity URI

An activity URI must always refer to a single unique activity. There may be corrections to that activity's definition. Spelling fixes would be appropriate, for example, but changing correct responses would not.

The activity URI is unique, and any reference to it always refers to the same activity. Activity Providers must ensure this is true and the LRS may not attempt to treat multiple references to the same URI as references to different activities, regardless of any information which indicates two authors or organizations may have used the same activity URI.

When defining an activity URI, care must be taken to make sure it will not be re-used. It should use a domain the creator controls or has been authorized to use for this purpose, according to a scheme the domain owner has adopted to make sure activity URIs within that domain remain unique.

Any state or statements stored against an activity URI must be compatible and consistent with any other state or statements that are stored against the same activity URI, even if those statements were stored in the context of a new revision or platform.

NOTE: The prohibition against an LRS treating references to the same activity URI as two different activities, even if the LRS can positively determine that was the intent, is crucial to prevent activity id creators from creating ids that could be easily duplicated, as intent would be indeterminable should a conflict with another system arise.

Activity Definition

Property	**Description**
name	Language Map, The human readable/visual name of the activity
description	Language Map, A description of the activity
type	URI, the type of activity. Note, URI fragments (sometimes called relative URLs) are not valid URIs. Similar to verbs, we recommend that Learning Activity Providers look for and use established, widely adopted, activity types.
interactionType | correctResponsesPattern | choices | scale | source | target | steps	See "Interaction Activities"
extensions	A map of other properties as needed (see: Extensions)

An LRS should update its internal representation of an activity's definition upon receiving a statement with a different definition of the activity from the one stored, but only if it considers the Learning Activity Provider to have the authority to do so.

Activities may be defined in XML according to the schema http://www.adlnet.gov/xapi. LRS's MAY attempt to look up an XML document at the URL given by the activity URI, and check if it conforms to the Experience API schema. If it does, the LRS SHOULD fill in its internal representation of the activities definition based on that document. Note that activity URI's are not required to resolve to such metadata.

Note that multiple activities may be defined in the same metadata document. The LRS MAY choose whether to store information about activities other than those it has received statements for or not.

As part of each group of activities, the activity metadata document may define information about an associated activity provider, which the LRS SHOULD consider the authoritative source for statements about the activity.

Interaction Activities

Traditional e-learning has included structures for interactions or assessments. As a way to allow these practices and structures to extend Experience API's utility, this specification include built in definitions for interactions which borrows from the CMI data model. These definitions are intended to provide a simple and familiar utility for recording interaction data. These definitions are simple to use, and consequently limited. It is expected that communities of practice requiring richer interactions definitions will do so through the use of extensions to an activity's type and definition.

When defining interaction activities, the activity type: "http://www.adlnet.gov/experienceapi/activity-types/cmi.interaction SHOULD be used, and a valid interactionType MUST be specified. If interactionType is specified, an LRS processing MAY validate the remaining properties as specified in the table below, and return HTTP 400 "Bad Request" if the remaining properties are not valid for the interaction type.

Property	Description
interactionType	As in "cmi.interactions.n.type" as defined in the SCORM 2004 4th edition Runtime Environment.
correctResponsesPattern	An array of strings, corresponding to "cmi.interactions.n.correct_responses.n.pattern" as defined in the SCORM 2004 4th edition Runtime Environment, where the final n is the index of the array.
choices | scale | source | target | steps	Array of interaction components specific to the given interaction type (see below).

Interaction Components

Interaction components are defined as follows:

Property	Description

id	As in “cmi.interactions.n.id” as defined in the SCORM 2004 4th edition Runtime Environment
description	Language Map (see below), a description of the interaction component (for example, the text for a given choice in a multiple-choice interaction)

The following table shows the supported lists of CMI interaction components for an interaction activity with the given interactionType.

interactionType	**supported component list(s)**
choice, sequencing	choices
likert	scale
matching	source, target
performance	steps
true-false, fill-in, numeric, other	[No component lists defined]

*See Appendix C for examples of activity definitions for each of the cmi.interaction types.

4.1.4.2 - Agent or Group as “object”

A statement may specify an Agent as an object in the statement. Agents that do this MUST specify an "objectType" property. See section 4.1.2 for details regarding Agents.

4.1.4.3 - Statement as "object"

Another statement may be used as an object in the statement, though some restrictions apply depending on whether the included statement is new, or is simply a reference to an existing statement.

Sub-Statements

When a new statement is included as part of another statement, it is considered a sub-statement, and is subject to certain restrictions. Sub-statements may only be included as parts of other statements, MUST specify an "objectType" property with the value "SubStatement", and MUST NOT have the "id", "stored", "authority”, or “voided” properties. They will be considered part of the parent statement, and MUST NOT contain a sub-statement. Implementations MUST validate the sub-statement as they would other statements, with the addition of these rules.

One interesting use of sub-statements is in creating statements of intention. For example, using sub-statements we can create statements of the form "<I> <planned> (<I> <did> <this>)" to indicate that we've planned to take some action. The concrete example that follows logically states that "I planned to read 'Some Awesome Book'".

```
{
        "actor": {"objectType": "Agent", "mbox":"mailto:test@example.com" },
```

```
        "verb" : { "id":"http://example.com/planned", "display":{"en-US":"planned"} },
        "object": {
                "objectType": "SubStatement",
                "actor" : {"objectType": "Agent", "mbox":"mailto:test@example.com" },
                "verb" : { "id":"http://example.com/read", "display":{"en-US":"read"} },
                "object": {
                        "id":"http://example.com/book",
                        "definition": { "name" : {"en-US":"Some Awesome Book"}}
                }
        }
}
```

Statement References

When an existing statement is included as part of another statement, a statement reference should be used. A statement reference is a simple object consisting an "objectType" property, which MUST be "StatementRef", and an "id" property, which MUST be set to the UUID of a statement which is present on the system.

Statement references are typically used in scenarios such as commenting or grading, and in the special case of voiding (see section 4.1.10 for details on voiding statements). Assuming that some statement has already been stored with the ID 8f87ccde-bb56-4c2e-ab83-44982ef22df0, the following example shows how a comment could be issued on the original statement, using a new statement:

```
{
        "actor" : { "objectType": "Agent", "mbox":"mailto:test@example.com" },
        "verb" : { "id":"http://example.com/commented", "display": {"en-US":"commented"} },
        "object" : {
                "objectType":"StatementRef",
                "id":"8f87ccde-bb56-4c2e-ab83-44982ef22df0"
        },
        "result" : { "response" : "Wow, nice work!" }
}
```

4.1.5 Result:

The result field represents a measured outcome related to the statement, such as completion, success, or score. It is also extendible to allow for arbitrary measurements to be included.

Property	Description
score	Score object (or not specified) - see section 4.1.5.1
success	true, false, or not specified
completion	true, false, or not specified
response	A string response appropriately formatted for the given activity.
duration	Period of time over which the statement occurred. Formatted according to ISO 8601, with a precision of 0.01 seconds.
extensions	A map of other properties as needed (see extensions)

4.1.5.1 Score

Property	Description
scaled	cmi.score.scaled (recommended)
raw	cmi.score.raw
min	cmi.score.min
max	cmi.score.max

4.1.6 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.

Property	Description
registration	UUID of the registration statement is associated with.
instructor	Instructor that the statement relates to, if not included as the Agent or Group of the statement.
team	Team that this statement relates to, if not included as the Agent or Group of the statement.

contextActivities	A map of the types of context to learning activities “activity” this statement is related to.
	
	Valid context types are: “parent”, “grouping”, and “other”.
	
	For example, if I am studying a textbook, for a test, the textbook is the activity the statement is about, but the test is a context activity, and the context type is “other”.
	
	{
	
	"other" : {"id" : "http://example.adlnet.gov/xapi/example/test" }
	
	}
	
	This activity could also be a session, like a section of a specific course, or a particular run through of a scenario. So the statement could be about “Algebra I”, but in the context of “Section 1 of Algebra I”.
	
	There could be an activity hierarchy to keep track of, for example “question 1” on “test 1” for the course “Algebra 1”. When recording results for “question 1”, it we can declare that the question is part of “test 1”, but also that it should be grouped with other statements about “Algebra 1”. This can be done using parent and grouping:
	
	{
	
	"parent" : {"id" : "http://example.adlnet.gov/xapi/example/test 1" },
	
	"grouping" : {"id" : "http://example.adlnet.gov/xapi/example/Algebra1" }
	
	}
	
	This is particularly useful with the object of the statement is an agent, not an activity. “I mentored Ben with context Algebra I”.

revision	Revision of the learning activity associated with this statement. Revisions are to track fixes of minor issues (like a spelling error), if there is any substantive change to the learning objectives, pedagogy, or assets associated with an activity, a new activity ID should be used. Revision format is up to the owner of the associated activity. Not applicable if statement's object is a Person.

platform	Platform used in the experience of this learning activity. Not applicable if statement's object is a Person. Defined vocabulary, TBD.
language	Code representing the language in which the experience being recorded in this statement (mainly) occurred in, if applicable and known. Do not specify any value if not applicable or not known. Format for this value is as defined in RFC 5646 For example, US English would be recorded as: en-US
statement	Another statement (either existing or new), which should be considered as context for this statement. See section 4.1.4.3 for details about including statements within other statements.
extensions	A map of any other domain-specific context relevant to this statement. For example, in a flight simulator altitude, airspeed, wind, attitude, GPS coordinates might all be relevant (see extensions)

4.1.7 Timestamp:

The time at which the statement took place, formatted according to ISO 8601. Note that this can differ from the system time of the event, such as in the case of formal or informal learning that occurs outside of the system.

4.1.8 Stored:

The LRS stored version of timestamp, formatted according to ISO 8601. It is recognized that the content time stamping of a statement (via the "timestamp" field) may differ from the LRS storage time, due to delays in reporting or as a statement is propagated to other systems.

4.1.9 Authority:

The authority property provides information about who or what has asserted that this statement is true. Consequently, the asserting authority may be an Agent (representing the authenticating user or some system or application), or in the case of 3-legged OAuth workflows, a Group of two Agents representing an application and user together. Unless used in the aforementioned 3-legged OAuth workflow, a Group MUST NOT be used to assert authority.

If a statement is stored using a validated OAuth connection, and the LRS creates or modifies the authority property of the statement, the authority MUST contain an agent object that represents the OAuth consumer, either by itself, or as part of a group in the case of 3-legged OAuth. This agent MUST be identified by account, and MUST use the consumer key as the account name field. If the OAuth consumer is a registered application,then the token request endpoint MUST be used as the account homePage, otherwise the temporary credentials endpoint must be used as the account homePage.

Except as specified in the paragraph above, agent objects MUST NOT use any OAuth endpoint as an account homePage.

The behavior of an LRS SHOULD NOT rely on Agents with an account using the temporary credentials endpoint as the homePage and with matching account names coming from the same source, as there is no way to verify that, since multiple unregistered applications could choose the same consumer key. Each unregistered consumer SHOULD pick a unique consumer key.

If a statement is received from another, fully trusted LRS, an LRS MAY choose not to overwrite the authority property

received. Otherwise, an LRS MUST completely ignore any provided authority property, and instead construct its own authority property as described here. If a user connects directly (using HTTP Basic Auth) or is included as part of a 3-legged OAuth workflow, the LRS MUST include the user as an Agent in the authority, and MAY identify the user with any of the legal identifying properties.

OAuth Authorities

In a 3-legged OAuth workflow, authentication involves both an OAuth consumer and a user of the OAuth service provider. For instance, requests made by an authorized Twitter plugin on your Facebook account will include credentials that are specific not only to Twitter as a client application, or you as a user, but the unique combination of both.

To support this concept, an LRS preparing the authority of a statement received via 3-legged OAuth MUST use a pairing of an application and a user. Below is a concrete example which represents a pairing of an OAuth consumer and a user.

```
"authority": {

"objectType" : "group",

        "member": [

                {

                "account": {

                "homePage":"http://example.com/XAPI/OAuth/Token",

                "name":"oauth_consumer_x75db"

                }

                },

                { "mbox":"mailto:bob@example.com" }

        ]

}
```

4.1.10 Voided:

A key factor in enabling the distributed nature of Experience API is through the immutability of statements. Because statements cannot be logically changed or deleted, systems can be assured to have an accurate collection of data based solely off of the stream of statements that are introduced into the LRS.

But, it is clear that statements may not always be valid for all of time once they are made. Mistakes or other factors could require that some previous statement is marked as invalid. For this case, the reserved ""http://adlnet.gov/expapi/verbs/voided" verb can be used, using a statement reference to the invalid statement as the object. See "Statement References" in section 4.1.4.3 for details.

An LRS which has received a statement that voids another statement should mark the target statement as voided using the "voided" field. If the target statement which is referenced cannot be found, the LRS should report an appropriate error indicating as such.

When issuing a voiding statement, the object is required to have its "objectType" field set to "Statement", and must specify the target statement's ID using the "id" field. An example of a voiding statement follows:

```
{
        "actor" : {
        "objectType": "Agent",
        "name" : "Example Admin",
        "mbox" : "mailto:admin@example.adlnet.gov"
        },
        "verb" : {
        "id":"http://adlnet.gov/xapi/verbs#voided",
        "display":{"en-US":"voided"}
        },
        "object" : {
        "objectType":"StatementRef",
        "id" : "e05aa883-acaf-40ad-bf54-02c8ce485fb0"
        }
}
```

The above statement voids a previous statement which is identified with the statement ID “e05aa883-acaf-40ad-bf54-02c8ce485fb0”. The previous statement will now be marked by setting its “voided” flag to true. Any changes to activity or agent definitions which were introduced by the voided statement may be rolled back by the LRS, but this not required.

Any statement that voids another cannot itself be voided. An activity provider that would like to “unvoid” a voided statement should reissue the statement under a new ID. Though voided and voiding statements must be reported as usual through the Experience API, it is recommended that reporting systems do not show voided or voiding statements by default.

4.1.11 Metadata:

The Experience API is extensible to allow any form of metadata, but recommends that existing fields are used to convey information. While metadata is extremely useful for the classification, storage, search, discovery, and retrieval of objects, it is out of the scope of the Experience API. It is recommended that content brokering services enforce metadata on created and distributed content and that querying services create metadata from Experience API statements and other available data.

4.2 Retrieval of Statements:

A collection of statements can be retrieved by performing a query on the “statements” endpoint, see section 7.2 "Statement API" for details.

Property	Type	Description

statements	Array of Statements	List of statements. If the list returned has been limited (due to pagination), and there are more results, they will be located at the “statements” property within the container located at the URL provided by the "more" element of this statement result object.
more	URL	Relative URL that may be used to fetch more results, including the full path and optionally a query string but excluding scheme, host, and port. Empty string if there are no more results to fetch. This URL must be usable for at least 24 hours after it is returned by the LRS. In order to avoid the need to store these URLs and associated query data, an LRS may include all necessary information within the URL to continue the query, but should avoid generating extremely long URLs. The consumer should not attempt to interpret any meaning from the URL returned.

5.0 Miscellaneous Types

5.1 Document:

The Experience API provides a facility for Activity Providers to save arbitrary data in the form of documents, which may be related to an Activity, Agent, or combination of both.

Property	Description
id	String, set by AP, unique within state scope (learner, activity)
updated	Timestamp
contents	Free form.

Note that in the REST binding, State is a document not an object. ID is stored in the URL, updated is HTTP header information, and contents is the HTTP document itself.

5.2 Language Map

A language map is a dictionary where the key is a RFC 5646 Language Tag, and the value is a string in the language specified in the tag. This map should be populated as fully as possible based on the knowledge of the string in question in different languages.

5.3 Extensions

Extensions are defined by a map. The keys of that map MUST be URIs, and the values MAY be any JSON value or data structure. The meaning and structure of extension values under a URI key are defined by the person who coined the URI, who SHOULD be the owner of the URI, or have permission from the owner. The owner of the URI SHOULD make a human-readable description of the intended meaning of the extension supported by the URI accessible at the URI. A learning record store MUST NOT reject a Experience API statement based on the values of the extensions map.

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.

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.

*Note: A statement should not be totally defined by its extensions, and be meaningless otherwise. Experience API 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 Experience API conformant tools.

6.0 Runtime Communication

6.1 Encoding:

All strings must be encoded and interpreted as UTF-8.

6.2 Version Header:

Requests to an LRS MUST include an HTTP header with name "X-Experience-API-Version" to indicate what version of the specification was used to construct the request. For systems written against this version of the specification, the value should always be "0.95". If an LRS cannot fulfill the request due to version incompatibilities, it MUST reject the request with a response of 400 and an error message explaining the problem. Conversely, every response from an LRS MUST include the HTTP header “X-Experience-API-Version” to indicate the version of the specification that was used to process the request.

The versions of the request and response will typically agree, but the version returned from the LRS may be higher in the case of compatible versions (i.e. the client has sent a 0.95 request which can be processed correctly under the 1.0 specification). The LRS MUST NOT attempt to process a request under the rules of a version less than that specified in the request (i.e. an LRS will not attempt to process a 1.0 request using the 0.95 specification). In such cases, the LRS will return a 400 error and an explanation of the problem.

Clients should use the lowest version of the specification which is compatible with the latest released version, and provides all the features the client needs. This will enable those clients to work with LRSs that only support the compatible earlier version specified by the client, as well as those that support the latest version.

An LRS supporting the latest version MUST use that version to process requests from older compatible versions, rather than having a parallel implementation for those versions. However, in the case of incompatible versions, the LRS MAY have a concurrent implementation to process legacy requests. While it is imperative that this specification strives for backwards compatibility, this behavior would allow clients to transition smoothly over time if breaking changes became unavoidable.

6.3 Concurrency:

In order to prevent “lost edits” due to API consumers PUT-ing changes based on old data, XAPI will use HTTP 1.1 entity tags (ETags) to implement optimistic concurrency control in the portions of the API where PUT may overwrite existing data. (State API, Actor and Activity profile APIs). The requirements in the rest of this “Concurrency” section only apply to those APIs.

When responding to a GET request, the LRS will add an ETag HTTP header to the response. The value of this header must be a hexidecimal string of the SHA - 1 digest of the contents, and must be enclosed in quotes.

The reason for specifying the LRS ETag format is to allow API consumers that can’t read the ETag header to calculate it themselves.

When responding to a PUT request, the LRS must handle the If - Match header or If - None - Match header as described in RFC2616, HTTP 1.1, if the If-Match header contains an ETag, or the If-None-Match header contains “*”. In the first case, this is to detect modifications made after the consumer last fetched the document, and in the second case, this is to detect when there is a resource present that the consumer is not aware of.

In either of the above cases, if the header precondition specified fails, the LRS must return HTTP status 412 “Precondition Failed”, and make no modification to the resource.

XAPI consumers should use these headers to avoid concurrency problems. The State API will permit PUT statements

without concurrency headers, since state conflicts are unlikely. For other APIs that use PUT (Actor and Activity Profile), the headers are required. If a PUT request is received without either header for a resource that already exists, the LRS must return HTTP status 409 “Conflict”, and return a plain text body explaining that the consumer must check the current state of the resource and set the “If-Match” header with the current ETag to resolve the conflict. In this case, the LRS must make no modification to the resource.

6.4 Security:

The LRS will support authentication using at least one of the following methods:

- OAuth 1.0 (rfc5849), with signature methods of "HMAC-SHA1", "RSA-SHA1", and "PLAINTEXT"
- HTTP Basic Authentication
- Common Access Cards (implementation details to follow in a later version)

There are a number of expected authentication scenarios to consider for the XAPI. In all cases, the LRS is responsible for making, or delegating, decisions on the validity of statements, and determining what operations may be performed based on the credentials used. It must be possible to configure any LRS to completely support the XAPI using any of the above authentication methods, and any of the workflows describe below. However, LRS may only support favored authentication mechanisms, or limit the known users or registered applications that may authenticate at all or using a specific authentication type. This is to allow administrators to strike the desired balance between interoperability and security.

In particular, the "PLAINTEXT" signature method of OAuth and HTTP Basic Authentication are likely to be turned off by security focused LRS administrators. Therefore LRS administrators are urged to minimally leave OAuth enabled, with at least the signature methods of "HMAC-SHA1" and "RSA-SHA1", and XAPI consumers are urged to use OAuth with one of those signature methods to maximize interoperability.

6.4.1 Authentication Definitions:

A **registered application** is an application that will authenticate to the LRS as an OAuth consumer that has been registered with the LRS. As part of that registration the application's name and a unique consumer key (identifier) shall be recorded by the LRS. Either the application has been assigned a consumer secret, or it has recorded its public key. The LRS must provide a mechanism to complete this registration, or delegate to another system that provides such a mechanism. The means by which this registration is accomplished are not defined by OAuth or the XAPI.

A **known user** is a user account on the LRS, or on a system which the LRS trusts to define users.

The following authentication workflows are anticipated.

1) Registered Application + Known User

This is the standard workflow for OAuth. Use the endpoints described further below to complete the standard OAuth workflow.

If this form of authentication is used to record statements and no authority is specified, the LRS should record the authority as a group consisting of an Agent representing the registered application, and a Person representing the known user.

2) Registered Application + Unknown User

An LRS may choose to trust certain applications to access the XAPI without additional user credentials, that is without invoking the authorize or token steps of the OAuth workflow. In that case, the LRS will consider requests valid that are signed using OAuth with thatapplication’s credentials and with an empty token and token secret. In this case, the application must have been registered with the LRS.

If this form of authentication is used to record statements and no authority is specified, the LRS should record the authority as the Agent representing the registered application.

3) Unregistered Application + Known User

The following must be applied to the standard OAuth workflow:

Since the application is not registered, its representing Agent will not be identified in the same way as a registered Agent, and the LRS must be careful about making assumptions regarding identity. See the section on Authority. A blank consumer secret should be used. The "Temporary Credential" request should then be called. Along with the usual parameters, "consumer_name" should be specified. During the user authentication phase, this name will be displayed to the user, along with a warning that the identity of the application requesting authentication cannot be verified.

Since OAuth is specifying an application, even though it is unverified, the LRS MUST record an authority that includes both that application and the authenticating user, as a group.

4) Known User, no application

This workflow uses HTTPBasicAuthentication. A username/password combination corresponding to an LRS login should be used, and the LRS should record the authority as an Agent identified by the login used, unless another authority is specified and the LRS trusts the known user to specify that authority.

5) No Authentication

Some LRSs may wish to support API access with no authentication, possibly for testing purposes, although there is no requirement to do so. To distinguish an explicitly unauthenticated request from a request that should be given a HTTP Basic Authentication challenge, unauthenticated requests should include headers for HTTP Basic Authentication based on a blank username and password.

6.4.2 OAuth Authorization Scope

The LRS will accept a scope parameter asdefinedinOAuth 2.0. If no scope is specified, a requested scope of “statements/write” and “statements/read/mine” will be assumed. The list of scopes determines the set of permissions that is being requested. An API client should request only the minimal needed scopes, to increase the chances that the request will be granted.

LRSs are not required to support any of these scopes except “all”. These are recommendations for scopes which should enable an LRS and an application communicating using the XAPI to negotiate a level of access which accomplishes what the application needs while minimizing the potential for misuse. The limitations of each scope are in addition to any security limitations placed on the user account associated with the request.

For example, an instructor might grant “statements/read” to a reporting tool, but the LRS would still limit that tool to statements that the instructor could read if querying the LRS with their credentials directly (such as statements relating to their students).

XAPI scope values:

Scope	**Permission**
statements/write	write any statement
statements/read/mine	read statements written by “me”, that is with an authority matching what the LRS would assign if writing a statement with the current token.

statements/read	read any statement
state	read/write state data, limited to activities and actors associated with the current token to the extent it is possible to determine this relationship.
define	(re)Define activities and actors. If storing a statement when this is not granted, IDs will be saved and the LRS may save the original statement for audit purposes, but should not update its internal representation of any actors or activities.
profile	read/write profile data, limited to activities and actors associated with the current token to the extent it is possible to determine this relationship.
all/read	unrestricted read access
all	unrestricted access

OAuth Extended Parameters

Note that the parameters “consumer_name” and “scope” are not part of OAuth 1.0, and therefore if used should be passed as query string or form parameters, not in the OAuth header.

OAuth Endpoints

Temporary Credential Request:
http://example.com/XAPI/OAuth/initiate

Resource Owner Authorization:
http://example.com/XAPI/OAuth/authorize

Token Request:
http://example.com/XAPI/OAuth/token

7.0 Data Transfer (REST)

This section describes The XAPI consists of 4 sub-APIs: statement, state, learner, and activity profile. The four sub-APIs of the Experience API are handled via RESTful HTTP methods. The statement API can be used by itself to track learning records.

7.1 Error Codes

The list below offers some general guidance on HTTP error codes that may be returned from various methods in the API. An LRS MUST return an error code appropriate to the error condition, and SHOULD return a message in the response explaining the cause of the error.

- 400 Bad Request - MAY be returned by any method of the API, and indicates an error condition caused by an invalid or missing argument. The term "invalid arguments" includes malformed JSON or invalid object structures.
- 401 Unauthorized - Indicates that authentication is required, or in the case authentication has been posted in the request, that the given credentials have been refused.
- 403 Forbidden - Indicates that the request is unauthorized for the given credentials. Note this is different than refusing the credentials given. In this case, the credentials have been validated, but the authenticated client is not allowed to perform the given action.
- 404 Not Found - Indicates the requested resource was not found. May be returned by any method that returns a uniquely identified resource, for instance, any state or profile API call targeting a specific document, or the method to retrieve a single statement.
- 409 Conflict - Indicates an error condition due to a conflict with the current state of a resource, in the case of state and profile API calls, or in the statement PUT call. See section 6.3 for more details.
- 412 Precondition Failed - Indicates an error condition due to a failure of a precondition posted with the request, in the case of state and profile API calls. See section 6.3 for more details.
- 500 Internal Server Error - General error condition, typically indicating an unexpected exception in processing on the server.

7.2 Statement API:

The basic communication mechanism of the Experience API.

PUT http://example.com/XAPI/statements

Stores statement with the given ID. This MUST NOT modify an existing statement. If the statement ID already exists, the receiving system SHOULD verify the received statement matches the existing one and return 409 Conflict if they do not match.

An LRS MUST NOT make any modifications to its state based on a receiving a statement with a statementID that it already has a statement for. Whether it responds with "409 Conflict", or "204 No Content", it MUST NOT modify the statement or any other object.

Returns: 204 No Content

Parameter	Type	Default	Description
statementId	String		ID of statement to record

POST http://example.com/XAPI/statements

Stores a statement, or a set of statements. Since the PUT method targets a specific statement ID, POST must be used rather than PUT to save multiple statements, or to save one statement without first generating a statement ID. An alternative for systems that generate a large amount of statements is to provide the LRS side of the API on the AP, and have the LRS query that API for the list of updated (or new) statements periodically. This will likely only be a realistic option for systems that provide a lot of data to the LRS.

Returns: 200 OK, statement ID(s) (UUID).

GET http://example.com/XAPI/statements

This method may be called to fetch a single statement, if the statementId parameter is specified, or a list of statements otherwise, filtered by the given query parameters.

Returns: 200 OK, statement

Parameter	Type	Default	Description
statementId	String		ID of statement to fetch

If statementId not specified, returns: A StatementResult object, a list of statements in reverse chronological order based on "stored" time, subject to permissions and maximum list length. If additional results are available, a URL to retrieve them will be included in the StatementResult object

Returns: 200 OK, Statement Result (See section 4.2 for details)

Parameter	Type	Default	Description
verb	String		Filter, only return statements matching the specified verb.
object	Activity, Agent, or Statement Object (JSON)		Filter, only return statements matching the specified object (activity or actor). Object is an activity: return statements with an object that is an activity with a matching activity ID to the specified activity. Object is an actor: same behavior as "actor" filter, except match against object property of statements.
registration	UUID		Filter, only return statements matching the specified registration ID. Note that although frequently a unique registration ID will be used for one actor assigned to one activity, this should not be assumed. If only statements for a certain actor or activity should be returned, those parameters should also be specified.

context	Boolean	True	When filtering on activities (object), include statements for which any of the context activities match the specified object.
actor	Actor Object (JSON)		Filter, only return statements about the specified agent. Note: at minimum agent objects where every property is identical are considered identical. Additionaly, if the LRS can determine that two actor objects refer to the same agent, they should be treated as identical for filtering purposes. See agent object definition for details.
since	Timestamp		only statements stored since the specified timestamp (exclusive) are returned
until	Timestamp		only statements stored at or before the specified timestamp are returned
limit	Nonnegative Integer	0	Maximum number of statements to return. 0 indicates return the maximum the server will allow.
authoritative	Boolean	True	Only include statements that are asserted by actors authorized to make this assertion (according to the LRS), and are not superseded by later statements.
sparse	Boolean	True	If true, only include minimum information necessary in actor and activity objects to identify them, If false, return populated activity and actor objects. Activity objects contain Language Map objects for name and description. Only one language should be returned in each of these maps. In order to provide these strings in the most relevant language, the LRS will apply the Accept-Language header as described in RFC 2616 (HTTP 1.1), except that this logic will be applied to each language map individually to select which language entry to include, rather than to the resource (list of statements) as a whole.
instructor	Actor Object (JSON)	True	Same behavior as “actor” filter, except match against “context:instructor”.
ascending	Boolean	False	If true, return results in ascending order of stored time

Note: Due to query string limits, this method may be called using POST and form fields if necessary. The LRS will differentiate a POST to add a statement or to list statements based on the parameters passed.

7.3 State API:

Generally, this is a scratch area for activity providers that do not have their own internal storage, or need to persist state across devices. When using the state API, be aware of how the stateId parameter affects the semantics of the call. If it is included, the GET and DELETE methods will act upon a single defined state document identified by "stateId". Otherwise, GET will return the available IDs, and DELETE will delete all state in the context given through the other parameters.

PUT | GET | DELETE http://example.com/XAPI/activities/state

Stores, fetches, or deletes the document specified by the given stateId that exists in the context of the specified activity, agent, and registration (if specified).

Returns: (PUT | DELETE) 204 No Content, (GET) 200 OK - State Content

Parameter	Type	Required	Description
activityId	String	yes	The activity ID associated with this state
agent	(JSON/XML)	yes	The agent associated with this state
registration	UUID	no	The registration ID associated with this state
stateId	String	yes	The id for this state, within the given context

GET http://example.com/XAPI/activities/state

Fetches IDs of all state data for this context (activity + agent [+ registration if specified]). If “since” parameter is specified, this is limited to entries that have been stored or updated since the specified timestamp (exclusive).

Returns: 200 OK, Array of IDs

Parameter	Type	Required	Description
activityId	String	yes	The activity ID associated with these states
agent	(JSON/XML)	yes	The actor associated with these states
registration	UUID	no	The registration ID associated with these states
since	Timestamp	no	Only IDs of states stored since the specified timestamp (exclusive) are returned

DELETE http://example.com/XAPI/activities/state

Deletes all state data for this context (activity + agent [+ registration if specified]).

Returns: 204 No Content

Parameter	Type	Required	Description
activityId	String	yes	The activity ID associated with this state

agent	(JSON/XML)	yes	The actor associated with this state
registration	UUID	no	The registration ID associated with this state

7.4 Activity Profile API:

The Activity Profile API is much like the State API, allowing for arbitrary key / document pairs to be saved which are related to an Activity. When using the profile API for manipulating documents, be aware of how the profileId parameter affects the semantics of the call. If it is included, the GET and DELETE methods will act upon a single defined document identified by "profileId". Otherwise, GET will return the available IDs, and DELETE will delete all state in the context given through the other parameters.

The Activity Profile API also includes a method to retrieve a full description of an activity from the LRS.

GET http://example.com/XAPI/activities

Loads the complete activity object specified.

Returns: 200 OK - Content

Parameter	Type	Required	Description
activityId	String	yes	The ID associated with the activities to load

PUT | GET | DELETE http://example.com/XAPI/activities/profile

Saves/retrieves/deletes the specified profile document in the context of the specified activity

Returns: (PUT | DELETE) 204 No Content, (GET) 200 OK - Profile Content

Parameter	Type	Required	Description
activityId	String	yes	The activity ID associated with this profile
profileId	String	yes	The profile ID associated with this profile

GET http://example.com/XAPI/activities/profile

Loads IDs of all profile entries for an activity. If “since” parameter is specified, this is limited to entries that have been stored or updated since the specified timestamp (exclusive).

Returns: 200 OK - List of IDs

Parameter	Type	Required	Description
activityId	String	yes	The activity ID associated with these profiles
since	Timestamp	no	Only IDs of profiles stored since the specified timestamp (exclusive) are returned

7.5 Agent Profile API:

The Agent Profile API is much like the State API, allowing for arbitrary key / document pairs to be saved which are related to an Agent. When using the profile API for manipulating documents, be aware of how the profileId parameter affects the semantics of the call. If it is included, the GET and DELETE methods will act upon a single defined document identified by "profileId". Otherwise, GET will return the available IDs, and DELETE will delete all state in the context given through the other parameters.

The Agent Profile API also includes a method to retrieve a special object with combined information about an Agent derived from an outside service, such as a directory service.

GET http://example.com/XAPI/agents

Return a special, Person object for a specified agent. The Person object is very similar to an Agent object, but instead of each attribute having a single value, each attribute has an array value, and it is legal to include multiple identifying properties. Note that the argument is still a normal Agent object with a single identifier and no arrays. Note that this is different from the FOAF concept of person, person is being used here to indicate a person-centric view of the LRS agent data, but agents just refer to one persona (a person in one context).

An LRS capable of returning multiple identifying properties for an Person SHOULD require the connecting credentials have increased, explicitly given permissions. An LRS SHOULD reject insufficiently privileged requests with 403 "Forbidden". If an LRS does not have any additional information about an Agent to return, the LRS MUST still return an Person when queried, but that Person object will only include the information associated with the requested Agent.

Person properties. All array properties must be populated with members with the same definition as the similarly named property from Agent objects.

Property	Description
objectType	“Person”. Required.
name	Array of strings. Optional.
mbox*	Array of strings.
mbox_sha1sum*	Array of strings.
openid*	Array of strings.
account*	Array of account objects.

Returns: 200 OK - Expanded Agent Object

Parameter	Type	Required	Description
agent	Object (JSON)	yes	The agent representation to use in fetching expanded agent information

PUT | GET | DELETE http://example.com/XAPI/agents/profile

Saves/retrieves/deletes the specified profile document in the context of the specified agent.

Returns: (PUT | DELETE) 204 No Content, (GET) 200 OK - Profile Content

Parameter	Type	Required	Description
agent	Object (JSON)	yes	The agent associated with this profile
profileId	String	yes	The profile ID associated with this profile

GET http://example.com/XAPI/agents/profile

Loads IDs of all profile entries for an agent. If “since” parameter is specified, this is limited to entries that have been stored or updated since the specified timestamp (exclusive).

Returns: 200 OK - List of IDs

Parameter	Type	Required	Description
agent	Object (JSON)	yes	The agent associated with this profile
since	Timestamp	no	Only IDs of profiles stored since the specified timestamp (exclusive) are returned

7.6 Cross Origin Requests:

One of the goals of the XAPI is to allow cross-domain tracking, and even though XAPI seeks to enable tracking from applications other than browsers, browsers still need to be supported. Internet Explorer 8 and 9 do not implement Cross Origin Resource Sharing, but rather use their own Cross Domain Request API, which can not use all of the XAPI as describe above due to only supporting “GET” and “POST”, and not allowing HTTP headers to be set.

The following describes alternate syntax for consumers to use only when unable to use the usual syntax for specific calls due to the restrictions mentioned above. All LRSs must support this syntax.

Method: All XAPI requests issued must be POST. The intended XAPI method must be included as the only query string parameter on the request. (ex: /XAPI/statements?method=PUT)

Headers: Any required parameters which are expected to appear in the HTTP header must instead be included as a form parameter with the same name

Content: If the XAPI call involved sending content, that content must now be encoded and included as a form parameter called “content”. The LRS will interpret this content as a UTF-8 string, storing binary data is not supported with this syntax.

See Appendix B for an example function written in Javascript which transforms a normal request into one using this alternate syntax.

7.7 Validation:

The function of the LRS within the XAPI is to store and retrieve statements. As long as it has sufficient information to perform these tasks, it is expected that it does them. Validation of statements in the Experience API is focused solely on syntax, not semantics. It is required to enforce rules regarding structure, but not rules regarding meaning. Enforcing the rules that ensure valid meaning among verb definitions, activity types, and extensions is a responsibility carried out by community of practice, in any way they see fit.

Appendix A: Bookmarklet

XAPI enables using an “I learned this” bookmarklet to self-report learning. The following is an example of such a bookmarklet, and the statement that this bookmarklet would send if used on the page: http://scorm.com/xapi.

The bookmarklet would be provided by the LRS to track to, for a specific user. Therefore the LRS URL, authentication, and actor information is hard coded in the bookmarklet. Note that since the authorization token must be included in the bookmarklet, the LRS should provide a token with limited privileges, ideally only enabling the storage of self-reported learning statements.

The UUID generation is only necessary since the PUT method is being used, if a statement is POSTED without an ID the LRS will generate it.

In order to allow cross-domain reporting of statements, a browser that supports the “Access-Control-Allow-Origin” and “Access-Control-Allow-Methods” headers must be used, such as IE 8+, FF 3.5+, Safari 4+, Safari iOS Chrome, or Android browser. Additionally the server must set the required headers.

```
var url = "http://localhost:8080/XAPI/Statements/?statementId="+_ruuid();
var auth = "Basic dGVzdDpwYXNzd29yZA==";
var statement = {actor:{ "objectType": "Agent",
"mbox" : "mailto:learner@example.adlnet.gov"},verb:"",object:{id:"" }};
var definition = statement.object.definition;


statement.verb='http://adlnet.gov/expapi/verbs/experienced';
statement.object.id = window.location.toString();
definition.type="http://adlnet.gov/expapi/activities/link";

var xhr = new XMLHttpRequest();
xhr.open("PUT", url, true);
xhr.setRequestHeader("Content-Type", "application/json");
xhr.setRequestHeader("Authorization", auth);
xhr.onreadystatechange = function() {
if(xhr.readyState == 4 ) {
alert(xhr.status + " : " + xhr.responseText);
}
};
xhr.send(JSON.stringify(statement));

/*!
Modified from: Math.uuid.js (v1.4)
http://www.broofa.com
mailto:robert@broofa.com

Copyright (c) 2010 Robert Kieffer
Dual licensed under the MIT and GPL licenses.
*/
function _ruuid() {
return 'xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx'.replace(/[xy]/g, function(c) {
var r = Math.random()*16|0, v = c == 'x' ? r : (r&0x3|0x8);
return v.toString(16);
});
}
```

Example Statement Using Bookmarklet

Headers:

```
{ 'content-type': 'application/json; charset=UTF-8',
authorization: 'd515309a-044d-4af3-9559-c041e78eb446',
referer: 'http://adlnet.gov/xapi/',
'content-length': '###',
origin: 'http://adlnet.gov' }
```

Method/Path:

```
PUT : /XAPI/Statements/?statementId=ed1d064a-eba6-45ea-a3f6-34cdf6e1dfd9

Body:

{

        "actor": {

                "objectType": "Agent",

                "mbox": "mailto:learner@example.adlnet.gov"

        },

        "verb": "http://adlnet.gov/expapi/verbs/experienced",

        "object": {

                "id": "http://adlnet.gov/xapi/ ",

                "definition": {

                        "type": "http://adlnet.gov/expapi/activities/link"

                }

        }

}
```

Appendix B: Creating an “IE Mode” Request

```
function getIEModeRequest(method, url, headers, data){

var newUrl = url;

Everything that was on query string goes into form vars
var formData = new Array();
var qsIndex = newUrl.indexOf('?');
if(qsIndex > 0){
formData.push(newUrl.substr(qsIndex+1));
newUrl = newUrl.substr(0, qsIndex);
}

Method has to go on querystring, and nothing else
newUrl = newUrl + '?method=' + method;

Headers
if(headers !== null){
for(var headerName in headers){
formData.push(
headerName + "=" +
encodeURIComponent(
headers[headerName]));
}
}

The original data is repackaged as "content" form var
if(data !== null){
formData.push('content=' + encodeURIComponent(data));
}

return {
"method":"POST",
"url":newUrl,
"headers":{},
"data":formData.join("&")
};
}
```

Appendix C: Example definitions for activities of type "cmi.interaction"

true-false

"definition": {

"description": {"en-US": "Does the XAPI include the concept of statements?"},

"type": "http://adlnet.gov/expapi/activities/cmi.interaction",

"interactionType": "true-false",

"correctResponsesPattern": ["true"]

}

choice

"definition": {

"description": {"en-US": "Which of these prototypes are available at the beta site?"},

"type": "http://adlnet.gov/expapi/activities/cmi.interaction",

"interactionType": "multiple-choice",

"correctResponsesPattern": ["golf[,]tetris"],

"choices": [

{"id": "golf", "description": {"en-US": "Golf Example"}},

{"id": "facebook", "description": {"en-US": "Facebook App"}},

{"id": "tetris", "description": {"en-US": "Tetris Example"}},

{"id": "scrabble", "description": {"en-US": "Scrabble Example"}}

]

}

fill-in

"definition": {

"description": {"en-US": "Ben is often heard saying: "},

"type": "http://adlnet.gov/expapi/activities/cmi.interaction",

"interactionType": "fill-in",

"correctResponsesPattern": ["Bob's your uncle"]

}

likert

"definition": {

"description": {"en-US": "How awesome is Experience API?"},

"type": "http://adlnet.gov/expapi/activities/cmi.interaction",

"interactionType": "likert",

"correctResponsesPattern": ["likert_3"],

"scale": [

{"id": "likert_0", "description": {"en-US": "It's OK"}},

{"id": "likert_1", "description": {"en-US": "It's Pretty Cool"}},

{"id": "likert_2", "description": {"en-US": "It's Damn Cool"}},

{"id": "likert_3", "description": {"en-US": "It's Gonna Change the World"}}

]

}

matching

{
"definition":{
"description":{
"en-US":"Match these people to their kickball team:"
},
"type":"http://adlnet.gov/expapi/activities/cmi.interaction",
"interactionType":"matching",
"correctResponsesPattern":[
"ben[.]3[,]chris[.]2[,]troy[.]4[,]freddie[.]1"
],
"source":[
{
"id":"ben",
"description":{
"en-US":"Ben"

```
}
},
{
"id":"chris",
"description":{
"en-US":"Chris"
}
},
{
"id":"troy",
"description":{
"en-US":"Troy"
}
},
{
"id":"freddie",
"description":{
"en-US":"Freddie"
}
}
],
"target":[
{
"id":"1",
"description":{
"en-US":"SCORM Engine"
}
},
{
"id":"2",
"description":{
"en-US":"Pure-sewage"
}
},
{
"id":"3",
"description":{
"en-US":"Project Tin Can API"
}
},
{
"id":"4",
"description":{
"en-US":"SCORM Cloud"
}
}
]
}
}
```


performance


```
"definition": {

"description": {"en-US": "This interaction measures performance over a day of RS sports:"},
```

"type": "http://adlnet.gov/expapi/activities/cmi.interaction",

"interactionType": "performance",

"correctResponsesPattern": ["pong[.]1:[,]dg[.]:10[,]lunch[.]"],

"steps": [

{"id": "pong", "description": {"en-US": "Net pong matches won"}},

{"id": "dg", "description": {"en-US": "Strokes over par in disc golf at Liberty"}},

{"id": "lunch", "description": {"en-US": "Lunch having been eaten"}}

]

}

sequencing

"definition": {

"description": {"en-US": "Order players by their pong ladder position:"},

"type": "http://adlnet.gov/expapi/activities/cmi.interaction",

"interactionType": "sequencing",

"correctResponsesPattern": ["tim[,]mike[,]ells[,]ben"],

"choices": [

{"id": "tim", "description": {"en-US": "Tim"}},

{"id": "ben", "description": {"en-US": "Ben"}},

{"id": "ells", "description": {"en-US": "Ells"}},

{"id": "mike", "description": {"en-US": "Mike"}}

]

}

numeric

"definition": {

"description": {"en-US": "How many jokes is Chris the butt of each day?"},

"type": "http://adlnet.gov/expapi/activities/cmi.interaction",

"interactionType": "numeric",

"correctResponsesPattern": ["4:"]

}

other

"definition": {

"description": {"en-US": "On this map, please mark Franklin, TN"},

"type": "http://adlnet.gov/expapi/activities/cmi.interaction",

"interactionType": "other",

"correctResponsesPattern": ["(35.937432,-86.868896)"]

}