To learn more about Project Tin Can, visit TinCanAPI.com.

Tin Can API (REST binding — version 0.9)

Definitions

Tin Can API (TCAPI): The API defined in this document is the product of "Project Tin Can". It's a simple, lightweight way for any permitted actor to store and retrieve extensible learning records as well as learner and learning experience profiles, regardless of the platform used.

Learning Activity Provider (AP): Like a SCORM package, the AP is the software object that is communicating with the LRS to record information about a learning experience.

Learning Activity (activity): Like a SCORM activity, the Learning Activity is 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): An LRS is 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 TCAPI.

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.

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 to be complete. For Tin Can 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 registrations 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 TCAPI State 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: Learners and activities can both have arbitrary key/document pairs of profile data

stored about them. This could be used for leader boards, to note learner preferences, learner strengths & weaknesses, etc.

Objects

Statement

The statement is the core of the TCAPI. All learning events are stored as statements in the form of "I did this". Verb and object are required, all other properties are optional.

Property	Type	Default	Description
id	UUID		may be assigned by statement creator or LRS.
actor	JSON/XML object		Learner or Team object the statement is about. "I". If not specified, LRS will infer based on authentication, and populate the actor.
verb	String		String. See table below.
inProgress	Boolean	false	Should the LRS wait for further information abo this statement, this statement is just a mention o learning experience in progress, but not yet to b submitted.
object			Activity, person, or another statement that is the object of the statement, "this". Note that Agent objects which are provided as a value for this field should include an "objectType" field. If no specified, the object is assumed to be an activity
result			Result object, further details relevant to the specified verb.
context	JSON/XML object		Context that gives the statement more meaning. Examples: the team an actor is working with, altitude in a flight simulator.
timestamp			Timestamp of when the thing that this statement describes happened.
stored			Timestamp of when this statement was recorded Set by LRS

authority			Actor who is asserting this statement is true. Verified by LRS based on authentication, and se by LRS if left blank.
voided	Boolean	false	Indicates that the statement has been voided (se below)

Aside from the possible initial assignment of "ID" and "Authority" by the LRS, the assignment of "Stored" whenever a statement is passed from system to system, and the "voided" flag, statements are immutable. Note that objects are referenced in statements (actors, activities); the contents of the object are not considered part of the statement itself. So while the statement is immutable, the actors and activities referenced by that statement are not. This means a deep serialization of a statement into JSON will change if the referenced activities and actors change.

Example of a simple statement:

```
{
    "id" : "fd41c918-b88b-4b20-a0a5-a4c32391aaa0",
    "actor" : {
        "name" : ["Project Tin Can"],
        "mbox" : ["mailto:tincan@scorm.com"]
    },
    "verb" : "created",
    "object" : {
        "id" :
"http://example.scorm.com/tincan/example/simplestatement"
,
        "definition" : {
            "name" : { "en-US" : "simple statement" } ,
            "description" : { "en-US" : "A simple Tin Can
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:

```
{
   "verb" : "created",
   "object" : { "id" :
"http://example.scorm.com/tincan/example/simplestatement"
   }
}
```

Typical simple completion with score:

```
{
    "actor" : {
        "name" : "Example Learner",
        "mbox" : "mailto:learner@example.scorm.com"
    },
    "verb" : "attempted",
    "object" : {
        "id" :
"http://example.scorm.com/tincan/example/simpleCBT",
        "definition" : {
            "name" : { "en-US" : "simple CBT course" } ,
            "description" : { "en-US" : "A fictitious
example CBT course." }
        }
    },
    "result" : {
          "score" : { "scaled" : .95},
          "success" : true,
          "completion" : true
    }
}
```

Statement Verbs:

There are two major competing goals in choosing a set of defined verbs for use in statements:

1. Provide sufficient verbs to clearly express any foreseeable learning event
2. Ensure that different verbs are not used to express the same concept.

The first goal suggests a large set of verbs, but a small set is better for the second goal.

The list of verbs below focuses on the second goal, while providing enough verbs to express concepts currently used for SCORM-based tracking.

The table below shows valid object types and results for use with defined verbs. The results listed are valid, not required results for that verb — so a statement with the verb "read" may report completion, but doesn't have to, and it should not report a score.

The attempt column indicates if a statement with the listed verb indicates the start of a new attempt. In other words, if prior statements exist for this learner and object, does the new

statement represent a distinct interaction with that object (new attempt), or does it add detail to existing statements (no new attempt)? No specific behavior is triggered by attempt vs. non-attempt verbs. However, clients should choose the appropriate verb type, and reporting tools may use this information to logically group results.

The list below is sorted into groups of related verbs. Related verbs have similar meanings, the same applicable object types, results, context, and attempt behavior, and can be handled as a group for reporting purposes. The specific verb used should always be displayed on reports except when aggregating statements.

Verb	**Object Types**	**Results**	**Attempt?**
experienced, attended	Content (video, book, article, blog), Event	Completion	Yes
attempted	Any	Completion, Success, Score, (interaction details)	Yes
completed, passed, failed	Any	Completion, Success, Score, (interaction details)	Yes
answered	question	Completion, Success, Score, interaction details	Yes
interacted	control, thing, interaction	(interaction details)	No
imported, created, shared	Any		No
voided	Statement		No

Completed, passed, mastered, and failed are special in that they indicate a specific result value. All statements using these verbs shall be read by an LRS as having completion = true, and passed, mastered, and failed will have success set to true, true, or false respectively. They are provided as a convenience to allow these common concepts to be compactly and clearly expressed. It is okay to explicitly set these values as described above in a statement with one of these verbs, or leave the results blank. A statement using the completed, passed, or failed verb and containing contradictory results is invalid.

When queried, an LRS must expand statements using completed, passed, or failed to include

the appropriate results block as implied by the verb used.

The imported verb exists as a way to get activity or actor definitions into an LRS without requiring a separate import API or misuse of another verb. Since activities or actors can be used in statements without previously importing them, and an LRS is required to save the information provided in such statements, it is possible to import an activity definition just by using it in a statement. Although a consumer may repeat details about activities and actors on each statement, for efficiency it is best to do so only once, particularly in cases where there is a lot of detail (such as when strings are defined in multiple languages).

Verbs are not case-sensitive. TCAPI consumers may specify verbs in any case, but the canonical representation is all lowercase as show above, and the LRS must return this representation of the verb when queried.

Voiding Statements:

A key factor in enabling the distributed nature of Tin Can 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 “voided” verb can be used, using the invalid statement as the object.

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" : {
        "name" : ["Example Admin"],
        "mbox" : ["mailto:admin@example.scorm.com"]
    },
    "verb" : "voided",
    "object" : {
        "objectType":"Statement",
        "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 actor 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 Tin Can API, it is recommended that reporting systems do not show voided or voiding statements by default.

Result

If the result of a statement is logically a simple string, eg: I commented *"This question is a little vague"* on "question1", then that string may be used as the result object. Otherwise, the result object is as follows:

Property	**Description**
score	Score object (or not specified)
success	true, false, or not specified
completion	Completed, or not specified
response	A response appropriately formatted for the given activity. Only valid for an interaction activity. In the case of an activity of type "cmi.interaction", this field is formatted according to the "`cmi.interactions.n.learner_response`" element defined in the SCORM 2004 4th edition Runtime Environment.
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.

Context

Context information relevant to the current statement.

Property	**Description**
registration	UUID of the registration statement is associated with.
instructor	Instructor that the statement relates to, if not included as the acto or object of the statement.
team	Team that this statement relates to, if not included as the actor or object 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 the activity the statement is about, but the test is a context activit and the context type is "other".
	
	`{ "other" : {"id" : "http://example.scorm.com/tincan/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 statemen 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 exampl "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 wit other statements about "Algebra 1". This can be done using pare and grouping:
	
	`{ "parent" : {"id" : "http://example.scorm.com/tincan/example/test 1" }, "grouping" : {"id" : "http://example.scorm.com/tincan/example/Algebra1" }`

	} This is particularly useful with the object of the statement is an actor, 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) 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 RFC3066. 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. This could be used to ac context to a comment, or when grading.
extensions	A map of any other domain-specific context relevant to this statement. For example, in a flight simulator altitude, airspeed, w attitude, GPS coordinates might all be relevant.

Score

Property	**Description**
scaled	cmi.score.scaled (recommended)

raw	cmi.score.raw
min	cmi.score.min
max	cmi.score.max

State

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.

Agent (learner or team)

These will be agent objects, based on FOAF agent objects http://xmlns.com/foaf/spec/#term_Agent.

A key design goal for the TCAPI is to enable an LRS to receive learning records about a learner that has not yet been defined in the LRS, or from a LP that does not have access to the identifier used by the LRS for that learner. FOAF, or "Friend of a Friend" agent objects provide some capabilities that will help in achieving these goals.

1. FOAF provides a vocabulary with a variety of options for uniquely defining an agent (or person) such as email address, weblog address, or account on a system (the LRS for example).
2. OWL (Web Ontology Language), which FOAF uses, provides a way to declare what properties can be used to uniquely identify an entity — they have the "Inverse Functional Property". Using this information, it is possible to merge two different sets of statements about the same entity, provided they have a match in one of these properties. For example, email is an inverse functional property for Person, if we have two sets of statements (FOAF objects) about someone who has the email address person1@example.com then we know that those statements are about the same person.

When defining a Learner, Team, or Agent, at least one field that has the Inverse Functional Property (http://www.w3.org/wiki/InverseFunctionalProperty) must be defined. Note that the "account" field in FOAF is not defined as having the Inverse Functional Property, but in the context of the TCAPI it will be considered to have this property.

TCAPI Agent objects have some differences from the FOAF version, so they will be defined here. Inverse functional properties are marked with a *.

NOTE: All properties (except type) of Agent, Person, and Group are arrays. The most correct, or most recent data should be listed first in each array.

Where the property names in tables below are links, please follow the links to details about that property in the FOAF documentation, which is the authoritative documentation for that property (a brief description is included here for convenience). Where no link is given, this document is authoritative on the usage of the property.

Agent

property	**description**
objectType	"Agent", "Person", or "Group". Will always be specified by the LRS. When not specified by an LRS consumer, the LRS will attempt to infer the type based on the properties present, howeve if the type could be either "Agent" or "Person" then "Person" wi be assumed.
name	(array of) Name of this agent.
mbox*	(array of) Email address that has only ever been assigned to this agent.
mbox_sha1sum*	(array of) SHA1 of an Email address that has only ever been assigned to this agent. The LRS will compare this value with mb values by applying SHA1(mbox).
openid*	(array of) The URI associated with an openID for this agent.
account*	(array of) Account objects. See below

Account

property	**description**

accountServiceHomePage	The URI to the canonical home page for the system the account is on.
accountName	The unique ID or name used to log in to this account.

Note: If the agent is authenticated via the OAuth Registered application workflow (with no known user), then the consumer_key should be used as the accountName, and the token request endpoint (eg: http://example.com/TCAPI/OAuth/token) should be used as the accountServiceHomePage. **It is crucial not to do this for any agent object where a known user is involved, as this would lead to logically different agents being considered identical by the LRS.**

Person (subclass of Agent)

If an agent is a person, some systems may need to know the components of their name. Therefore, this information should be provided if available. Using givenName & familyName is preferred if known, otherwise firstName and lastName should be used, if known. See the FOAF documentation for each property linked below for a discussion of why both options are provided.

property	description
givenName	(array of) a person's given name
familyName	(array of) a person's family name
firstName	(array of) a person's first name
lastName	(array of) a person's last Name

Group (subclass of Agent)

property	description
member	(array of) Agent objects representing members of this grou

NOTE: Groups are currently only supported in the Asserter field of the statement, and only for a group with 2 members as needed for OAuth authentication of a consumer and a user.

The LRS should consider agents with matching fields having the inverse functional property (such as email) to be the same agent. This equivalence should be applied both when filtering

statements based on agent, and when reporting. However, the LRS should still be able to report on the original agent that was associated with any statement, before applying any merge operation.

Activity

Property	**Description**
objectType	Should always be “Activity” when present. Used in cases where typ cannot otherwise be determined, such as the value of a statement’s “object” field.
id	URI , may be a URL. If a URL, the URL should refer to metadata f this activity. If it does not resolve to activity metadata, then it must resolve. This URI is unique. Any reference to it always refers to the same activity, the AP must ensure this is true and the LRS may not attem to treat multiple reference to the same URI as references to differen activities, regardless of any information which indicates two author organizations may have picked the same activity ID. When defining activity ID, care must be taken to make sure it will not be re-used. I should use a domain the creator controls or has been authorized to u creating IDs, according to a scheme the domain owner has adopted make sure IDs within that domain remain unique. The prohibition against an LRS treating references to the same activ ID as two different activities, even if the LRS can positively determ that was the intent, is crucial to prevent activity ID creators from getting by with sloppy IDs and then later encountering a system tha unable to determine the original intent. Any state or statements stored against an activity id must be compatible and consistent with any other state or statements that are stored against the same activity ID, even if those statements were stored in the context of a new revision or platform. Notice that there no way to specify revision or platform on State! If this consistency not be maintained, it is time to create a new activity id.
definition	Metadata, See below

An activity id URI must always refer to a single unique activity. There may be corrections to that

activities definition. Spelling fixes would be appropriate, for example, but changing correct responses would not.

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 would have considered the statement authoritative if the activity was previously unknown to the LRS.

Activity Definition

name	Language Map (see below), what the activit is called
description	Language Map (see below), a description of the activity (question text if a question)
type	course, module, meeting, media, performance, simulation, assessment, interaction, cmi.interaction, question, objective, link
interactionType	Identifies a specific interaction type, in the case of an interaction activity (see below)
extensions	A map of other properties as needed

Activity Metadata:

Activities may be defined in XML according to the schema http://projecttincan.com/tincan.xsd. Upon first encountering an activity ID in a statement, if the ID is a URL, and the activity definition type is not "link", the LRS should retrieve the document at that URL and check if it conforms to the TinCan schema. If it does, the LRS should fill in its internal representation of the activities definition based on that document.

Note that multiple activities may be defined in the same metadata document, even one served from an activity ID URL. 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 activity provider that the LRS should expect will report statements for that activity. The LRS should incorporate that data to the extent it works with the registration workflow the LRS has adopted for OAuth.

Link: If the activity definition type is “link”, then the URI of the activity must be a URL to a resource. Activities of type “link” are defined by the resource they link to. For all other activities, their ID should either not resolve, or resolve to metadata as described above. The purpose of the “link” type is to simplify the process of making statements about existing resources for which a URL is known, and to remove the need to generate metadata for that resource. For example, the statement “I experienced this”, where this is a web page is best represented with an activity type of “link”.

Interaction Activities:

By design, Tin Can is intended to be a communication structure, a vocabulary for conveying what a person or people did in many contexts. It would be impossible for early versions of the specification to provide detailed data structures for every community of practice.

Traditional e-learning has included structures for interactions or assessments. As an example for how those practices and structures might be used to extend Tin Can's definition and utility, we've included a definition for interactions in the Tin Can specification which borrows from the CMI data model. (This carries the further benefit of preventing any loss of capability as we move from SCORM 1.2 and SCORM 2004 to something new and better.)

The definitions that accompany the “cmi.interaction” activity type are intended to accommodate that which was possible in SCORM and something slightly more. They are not intended to encompass *everything which is possible* in the world of assessments going forward. That's a task to be left to people more expert in that area who will further extend Tin Can in the future.

When using the “cmi.interaction” activity type, the following activity definition fields are processed as follows:

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 CMI interaction components specific to the given interaction type (see below).

CMI Interaction Components:

CMI interaction components are defined as follows:

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 of type "cmi.interaction" 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.

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.

Activity / Learner profile

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

Note that in the REST binding, activity and learner profiles are documents, not objects. ID is stored in the URL, updated is HTTP header information, and contents is the HTTP document itself.

StatementsResult

Container for a list of statements and a continuation link for fetching additional results for that list.

Property	Type	Description
statements	Array of Statements	List of statements
more	URL	Relative URL that may be used to fetch more results, including the full path and optionally a query string b excluding scheme, host, and port. Empty string if ther are no more results to fetch. This URL must be usable for at least 24 hours after it 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 URI continue the query, but should avoid generating extremely long URLs. The consumer should not atten to interpret any meaning from the URL returned.

Runtime Communication

The TCAPI consists of 4 sub-APIs: statement, state, learner, and activity profile. The statement API can be used by itself to track learning records.

Encoding

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

Security

The LRS will support authentication using the following methods:

- OAuth 1.0 (rfc5849), with signature methods of "HMAC-SHA1", "RSA-SHA1", and "PLAINTEXT"
- HTTP Basic Authentication

There are a number of expected authentication scenarios to consider for the TCAPI. 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 TCAPI using either of the above authentication methods, and any of the workflows describe below. However, an LRS may only be configured to support favored authentication mechanisms, or limit the known users or registered applications that may authenticate at all or using a specific authentication type. In summary, the LRS must be capable of supporting any authentication scenario, but may be configured with any desired restrictions. 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 TCAPI consumers are urged to use OAuth with one of those signature methods to maximize interoperability.

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) have been recorded. 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 TCAPI.

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 asserter is specified, the LRS should record the asserter as a group consisting of an Agent representing the registered application, and a Person representing the known user.

2) Registered Application

An LRS may choose to trust certain applications to access the TCAPI 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 that application'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 asserter is specified, the LRS should record the asserter 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, it's consumer key will not be stored by the LRS. A consumer key should be picked that will be the same every time that application (or installation of the application) communicates, but will not be used by another application. 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 can not be verified.

Since OAuth is specifying an application here, even though it is unverified, the LRS may want to record an asserter that includes both that application and the authenticating user, as a group.

4) Known User, no application

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

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.

OAuth Authorization Scope

The LRS will accept a scope parameter as defined in OAuth 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 TCAPI 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).

TCAPI scope values:

scope	**permission**
statements/write	write any statement
statements/read/mine	read statements written by "me", that is with an asserter matchi what the LRS would assign if writing a statement with the curre 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 th relationship.
define	(re)Define activities and actors. If storing a statement when this

	not granted, IDs will be saved and the LRS may save the origin 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 associate with the current token to the extent it is possible to determine th 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/TCAPI/OAuth/initiate

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

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

Concurrency

In order to prevent “lost edits” due to API consumers PUT-ing changes based on old data, TCAPI 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.

TCAPI 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.

Statements

Store (Statement)

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

Stores a statement, or a set of statements. Returns: 200 OK, statement ID(s) (UUID). 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.

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

Returns: 200 OK, statement

Parameter	Type	Default	Description
statementId	String		ID of statement to fetch

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

Returns: 204 No Content
Errors: 409 Conflict

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 will never 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 will not modify the statement or any other object.

Parameter	Type	Default	Description
statementId	String		ID of statement to record

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

Returns: A StatementList 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 StatementList object.

Parameter	Type	Default	Description
verb	String		Filter, only return statements matching the specified verb.
object	Actor Object (JSON/XML)		Filter, only return statements matching the specified object (activity or actor). Object is an activity: return statements with 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 o 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 statemen 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/XML)		Filter, only return statements about the specified agent. Note: at minimum agent objects where every property is identical are considered identical. Additionally, if the LR 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 wil 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 activ 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 t include, rather than to the resource (list of statements) as a whole.
instructor	Actor Object (JSON/XML)	True	Same behavior as "actor" filter, except matc against "context:instructor".

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.

State

Generally, this is a scratch area for activity providers that do not have their own internal storage, or need to persist state across devices.

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

Stores, fetches, or deletes the specified state document in the context of the specified activity, actor, and registration (if specified).

Parameter	Type	Required	Description
activityId	String	yes	The activity ID associated with this state
actor	(JSON/XML)	yes	The actor associated

			with this state
registrationId	String	no	The registration ID associated with this state
stateId	String	yes	The id for this state, within the given context

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

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

Parameter	**Type**	**Required**	**Description**
activityId	String	yes	The activity ID associated with this state
actor	(JSON/XML)	yes	The actor associated with this state
registrationId	String	no	The registration ID associated with this state

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

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

Parameter	**Type**	**Required**	**Description**
activityId	String	yes	The activity ID associated with these states

actor	(JSON/XML)	yes	The actor associated with these states
registrationId	String	no	The registration ID associated with these states
since	Timestamp	no	Only IDs of states stored since the specified timestamp (exclusive) are returned

Activity Profile

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

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

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/TCAPI/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).

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

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

Loads the complete activity object specified.

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

Actor Profile

PUT | GET | DELETE http://example.com/TCAPI/actors/profile

Saves/retrieves/deletes the specified profile document in the context of the specified learner (learner may be an individual or a team)

Parameter	Type	Required	Description
actor	(JSON/XML)	yes	The actor associated with this profile
profileId	String	yes	The profile ID associated with this profile

GET http://example.com/TCAPI/actors/profile

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

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

GET http://example.com/TCAPI/actors

Loads full actor object for the specified actor. Even though an actor object is specified in the get call, the LRS may have more information about that actor that can be returned. For example, the actor object passed in could include only an email address, but the LRS could return an actor object populated with name, department, and role.

Parameter	Type	Required	Description
actor	(JSON/XML)	yes	The (partial) actor representation to use in fetching actor information

Cross Origin Requests in Internet Explorer

One of the goals of the TCAPI is to allow cross-domain tracking, and even though the TCAPI 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 TCAPI 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 TCAPI requests issued must be POST. The intended TCAPI method

must be included as the only query string parameter on the request. (ex: /TCAPI/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 TCAPI 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.

Problems

Registration: LMS concept, but may need to be included in launch information, tracking.

Solution: Registration ID (UUID) becomes part of the statement stream, may be specified by clients when storing statements. If the LRS provides a launch link, it would provide the registration to track against in that link, if the launch is based on a registration. If the LRS provides a registration ID, then the AP must use it when reporting statements, and should use it when querying statements unless specifically intending to retrieve previous registration information as well. This method allows for multiple simultaneous registrations.

Rejected Solution: "Registered for" verb causes all prior statements to be "non-authoritative", and starts new registration. **Only one registration at a time is valid**. Clients don't really have to know about registrations. This does not work because different assignments may be made for both an activity and a child of that activity. In that case, using the registered verb on the child activity would reset progress from both the assignment of that particular activity and its parent. Also, only one registration at a time being valid is an unreasonable restriction due to this same sort of overlap. A registration for an activity should not preclude a registration on another activity that has it as a child.

The concepts of "attempt" and "submitted" are similar. Furthermore, specifying some verbs as signifying a new attempt, and others not limits verb choice and therefore statement expressiveness. We should consider merging these concepts somehow. The problem with doing that is that "submitted" would be associated with ending an attempt, whereas the attempt vs. non-attempt verbs determine whether or not to start a new attempt. Both seem to be needed, submitted when it is known whether or not more details will be sent, and attempt vs non-attempt to determine whether a new statement is a revision of an existing attempt (an instructor grading the attempt for example), or the start of a new attempt.

Result: Determining “authoritative” results will be difficult unless each result field is constrained to only be enabled on one verb. If two statements have different verbs, and different results, particularly different partial results, how does that get summarized? Could potentially be solved with a “scored” verb, “passed” verb, etc.

Statement visibility (privacy) is a concern. There is nothing in the API to prevent any actor from viewing any statements written by any other actor, though an LRS may chose to limit this. To avoid an interoperability mess, at minimum, best practices on what actor types (admin, instructor, etc) can view which statements should be established.

FOAF account is not defined as having the inverse functional property, however we need a way to uniquely identify agents (people) based on their LRS account as an option. Consider adding an extension property rather than changing the definition of FOAF account.

Should the FOAF agent object have a type added so the LRS can differentiate between a person and an agent? Does it matter?

Results/Score section should be updated to use CMI5

Possibilities

UUID of statement could be a hash of other statement fields (except store time). This could potentially allow two systems to generate the same statement, with the same ID, if describing the same event (at the same time).

Statements could be signed by the “Authority”. This would require canonicalization of statements first.

Appendix A: Bookmarklet

The TCAPI 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://tincanapi.com.

The bookmarklet would be provided by and tracked to the LRS 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/TCAPI/Statements/?statementId="+_r
uuid();
var auth = "Basic dGVzdDpwYXNzd29yZA==";
var statement = {actor:{"mbox" :
["mailto:learner@example.scorm.com"]},verb:"",object:{id:
"" }};
var definition = statement.object.definition;

statement.verb='experienced';
statement.object.id = window.location.toString();
definition.type="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://tincanapi.com/',
  'content-length': '###',
  origin: 'http://tincanapi.com' }
```

Method/Path:

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

Body:

```
{
     "actor":{
          "mbox":["mailto:learner@example.scorm.com"]
     },
     "verb":"experienced",
     "object":{
          "id":"http://tincanapi.com ",
          "definition":{
               "type":"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 TCAPI include the concept of statements?"},
        "type": "cmi.interaction",
        "interactionType": "true-false",
        "correctResponsesPattern": ["true"]
}
```

choice

```
"definition": {
        "description": {"en-US": "Which of these prototypes are available at the beta site?"},
        "type": "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": "cmi.interaction",
        "interactionType": "fill-in",
        "correctResponsesPattern": ["Bob’s your uncle"]
}
```

likert

```
"definition": {
        "description": {"en-US": "How awesome is Tin Can?"},
        "type": "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": "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"}},
               {"id": "4", "description": {"en-US": "SCORM Cloud"}}
            ]
}
```

performance

```
"definition": {
            "description": {"en-US": "This interaction measures performance over a day of RS
sports:"},
            "type": "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": "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": "cmi.interaction",
        "interactionType": "numeric",
        "correctResponsesPattern": ["4:"]
}
```

other

```
"definition": {
        "description": {"en-US": "On this map, please mark Franklin, TN"},
        "type": "cmi.interaction",
        "interactionType": "other",
        "correctResponsesPattern": ["(35.937432,-86.868896)"]
}
```