Tin Can Client Library Guidelines

This document provides guidelines for implementing a client library that will target the Tin Can specification. The target audience for this document is the proficient software developer who would like to implement a client library for Tin Can, and who has read through the TCAPI specification.

This document is intended to remain language agnostic, and to provide a common set of blueprints that will encourage an instant familiarity for individuals, organizations, or applications that will interact with Tin Can across many languages and platforms. It is also our hope that the design presented will offer a sensible interface that is both flexible and easy to use, and one that addresses common design challenges related generally to Tin Can.

Below, we cover the programmatic binding of the TCAPI, along with some guidance on object structure, exception handling, authentication, and offline storage. It also includes a few examples of how the interface could be used in client code.

Implementers should strive to meet the specifications and guidelines laid out here, but are free to introduce additional methods, constructors, or properties that offer added convenience in their use. But, it is our intention that every client library expose the same common base API as it is described here. The interfaces and examples below are given in Java like syntax, but if the platform or language for which you're writing dictates a certain behavior or style, we expect that this will be respected (for instance, if you're writing a node.js module, we would expect you to respect the asynchronous nature of method calls, and the passing of error objects, and so on).

Standard Types:

The following standard types should be supported, with object properties that match those given in the TCAPI spec.

Statement
Result
Score
Context
Agent
Account
Person
Group
Activity
ActivityDefinition
LanguageMap
StatementsResult
State

Extended Types:

Additionally, the following extended types and enumerations should be implemented:

StatementVerb - Enumeration of core verbs valid for current TCAPI version (experienced, attended, attempted, completed, etc)

ActivityType - Enumeration of activity types given in TCAPI doc (course, module, meeting, media, etc)

LanguageTag - Enumeration corresponding to IETF language tags (http://en.wikipedia.org/wiki/IETF_language_tag), used as key for LanguageMap objects

Interaction, InteractionDefinition, InteractionComponents - Interaction is a subclass of Activity, always uses ActivityType “cmi.interaction”, and uses InteractionDefinition to specify a CMI interaction using the correct InteractionComponents

ActivityProfile, ActorProfile, ActivityState - Subclasses of State, which include fields to identify their context. ActivityProfile should include an ActivityId property, ActorProfile should include an Actor property, and ActivityState should include ActivityId, Actor, and Registration properties.

StatementQueryObject - An object to encapsulate the parameters of a statement query as listed in the TCAPI spec, such as verb, object, registration, etc.

Validation:

Client libraries should attempt to validate the following, before sending a statement to the LRS:
a) Actor objects have at least one uniquely identifying property
b) The verb for a statement is in the core list of verbs from the TCAPI spec
c) The verb for a statement is valid for the given activity to which it refers
d) The results of the statement are valid for the given statement verb
e) Interaction definitions include the correct components, given the interaction type
f) Any other validation tasks that can be completed before sending the statement

The strategies for validation are left open to implementers, but our general guideline is to make validation implicit for the consumer of the library, and to try and make it difficult to code invalid statements in the first place. As an example, one may wish to implement specific classes for each of the interaction types, like so:

ChoiceInteraction
SequencingInteraction
NumericInteraction
etc.

In this way, when the library consumer creates the class, they’ll only have the option of setting

properties that are valid for that type of interaction (in the case of ChoiceInteraction, that property would be called “choices”).

Exceptions:

Client libraries will also be responsible for translating HTTP error codes returned from the TCAPI endpoint into structured exceptions. The following table shows the intended mapping from HTTP error codes to client library exception classes.

HTTP Error code	Exception class
400	InvalidArgumentException
401	UnauthorizedException
404	NotFoundException
409	ConflictException
412	PreconditionFailedException
500	Exception (generic). Client libraries should reattempt the request at least once upon receiving a 500 error.

Generally, client libraries should strive to abstract the TCAPI communication binding (HTTP). This in turn should allow client libraries to present the same interface regardless of the underlying binding (i.e. in the case of a future SOAP binding for the TCAPI).

TCAPI Interface:

For API access, client libraries should present the TCAPI as an object, configured with endpoint and authentication information, and having class methods representing the TCAPI methods outlined in the spec. The object should implement the interface “TCAPI”, and should support the following method signatures (parameters marked with an asterisk are optional, with non-null default values shown in parentheses):

```
void StoreStatement(Statement statement, boolean synchronous* (false));
void StoreStatements(Statement[] statements, boolean synchronous* (false));
void VoidStatements(String[] statementIdsToVoid, boolean synchronous* (false));
void Flush();
Statement GetStatement(String statementId);
StatementResult GetStatements(StatementQueryObject queryObject*);
StatementResult GetStatements(String moreUrl);
```

ActivityState GetActivityState(String activityId, Actor actor, String stateId, String registrationId*);
void SaveActivityState(ActivityState activityState, bool overwrite* (true), ActivityState previousState*);
void DeleteActivityState(String activityId, Actor actor, String stateId, String registrationId*);
void DeleteAllActivityState(String activityId, Actor actor, String registrationId*);
String[] GetActivityStateIds(String activityId, Actor actor, String registrationId*, Date since*);

ActivityProfile GetActivityProfile(String activityId, String profileId);
void SaveActivityProfile(ActivityProfile activityProfile, bool overwrite* (false), ActivityProfile previousProfile*);
void DeleteActivityProfile(String activityId, String profileId);
void DeleteAllActivityProfile(String activityId);
String[] GetActivityProfileIds(String activityId, Date since*);
Activity GetActivity(String activityId);

ActorProfile GetActorProfile(Actor actor, String profileId);
void SaveActorProfile(ActorProfile actorProfile, bool overwrite* (false), ActorProfile previousProfile);
void DeleteActorProfile(Actor actor, String profileId);
void DeleteAllActorProfile(Actor actor);
String[] GetActorProfileIds(Actor actor, Date since*);
Actor GetActor(Actor partialActor);

String GetOAuthAuthrorizationUrl(String redirectUrl*);
OAuthAuthentication UpdateOAuthTokenCredentials(String temporaryCredentialsId, String verifierCode);

Additionally, the TCAPI object should be configurable with the following properties:

Endpoint - The URL for the TCAPI endpoint
AuthenticationConfiguration - Object representing authentication configuration (see below)
TCAPICallback - Interface used in asynchronous statement reporting (see below)
OfflineStorage - Interface used for offline statement queue (see below)
StatementPostInterval - In seconds, how often to post the statement queue to the LRS. Use 0 to disable post interval (and use only the explicit Flush() method).

State and Profile Concurrency:

The TCAPI spec addresses ways to overcome issues related to concurrent updates of state and profile documents. Specifically, clients can issue hashes of the last known content along with a write request to make sure they are not inadvertently overwriting another update.

Client libraries ease the use of these rules through parameters on the various Save calls for

ActivityState, ActivityProfile, and ActorProfile. For each, an optional boolean parameter will allow the application to ignore concurrency issues and overwrite the existing document. In the case of ActivityState, which is less likely to be shared among users or activities, the default behavior is to overwrite.

In any case, the client application can also respect and resolve conflicts by issuing a Set call along with the last known copy of the State or Profile document. The client library will be responsible for calculating the hexidecimal string of the SHA-1 digest of the contents from the given previous copy and sending it in the update request, as outlined in the TCAPI spec. If a conflict occurs, the client library will raise a ConflictException that can then be used by the application to resolve the conflict, typically by getting the most recent copy of the document, merging in the desired changes to it, and saving it (again, providing the most recent copy just fetched, as the previous document).

Authentication:

Authentication support should be provided by the client library, and the TCAPI client object should be configured with an instance of the AuthenticationConfiguration class which configures the authentication behavior. The following authentication configuration subclasses, and the underlying authentication behavior, should be implemented by the client library:

BasicHttpAuthentication
    properties: String Username, String Password, String AuthHeaderValue

OAuthAuthentication
    properties: String ConsumerKey, String ConsumerSecret,
        String TokenId, String TokenSecret

For BasicHttpAuthentication, the configuration should allow the application to either use a username / password combination, in which case the library should perform the steps to create the HTTP Authorization header, or the application may set the HTTP Authorization header value directly.

In the case of OAuth, the configuration should include the consumer key and consumer secret of the client application, and, if applicable, the token credentials that have been obtained via an OAuth handshake with the LRS. The client library will be responsible for using these values to sign outgoing requests as specified in the OAuth 1.0 protocol.

To support the standard OAuth workflow, two convenience methods are available on the TCAPI object.

String GetOAuthAuthorizationUrl(String redirectUrl*).

This method should use the configured consumerKey and consumerSecret to obtain temporary credentials from the Temporary Credential Request endpoint (under the TCAPI endpoint, i.e. /TCAPI/OAuth/initiate), and then use those to generate the URL for a Resource Owner Authorization request (at /TCAPI/OAuth/authorize). Due to timestamp restrictions, this URL should be generated on demand, just before it used. The method also takes an optional parameter, redirectUrl, which is used as the oauth_callback parameter in the OAuth handshake. The LRS will redirect the user to this URL, along with the added OAuth parameters, after they accept (or deny) the authorization request.

Finally, once a verification code has been received from the server using the authorization request, the client application can update the TCAPI object's OAuthAuthentication configuration with Token Credentials from the TCAPI endpoint by using the following method:

OAuthAuthentication UpdateOAuthTokenCredentials(String tempId, String verifierCode).

When this method is called, the client library should retrieve a new set of token credentials (from /TCAPI/OAuth/token), and the OAuthAuthentication configuration referenced by the TCAPI object should be updated with the retrieved token credentials. Further OAuth request signing which is done by the client library should utilize both the consumerKey and consumerSecret along with the retrieved tokenId and tokenSecret. This method also returns the updated OAuthAuthentication configuration object. This is provided as the return value so that client applications may easily obtain and persist the updated values for future use.

Note that if your application is using the OAuth workflow of "Registered Application" as described in the TCAPI specification, you can skip both of the above steps, and simply make signed requests using your consumerKey and consumerSecret, leaving the token credentials blank. This greatly simplifies the authentication process, but also leaves a lot of trust in the client application, which may not be appropriate in many scenarios.

Statement Queueing and Offline Storage:

Client applications should strive to preserve functionality where possible when disconnected from the network / LRS. This may happen in disconnected mobile scenarios, or when an LRS is behind a private network that is inaccessible at the user's current location. Client libraries can provide limited support for disconnected scenarios, using statement queueing.

For client libraries, statement reporting (StoreStatement, StoreStatements, VoidStatements) is intended to be an asynchronous operation. Synchronous calls are provided to aid very simple scenarios when the client application does not need to issue many statements, and will not be supporting offline capability. In this case, the statement queue should be skipped, the TCAPI request should be called immediately, and the method should wait for a result or error before

returning.

When these methods are called asynchronously, client libraries should queue statements and return immediately. At a regular, configurable interval, or when the Flush() method is called, the client library should then attempt to persist the queued statements to the LRS. The queue should be submitted to the LRS in batches with a reasonable maximum size, between 10 and 50. When statements are successfully persisted, the registered object implementing the TCAPICallback interface will have the following method called:

void StatementsStored(Statement[] statements);

When a batch of statements fails, the following method is called on the registered TCAPICallback object:

boolean StatementsFailed(Statement[] failedBatch, Exception exception);

The client application can use this method to notify the user or other parts of the application, or it may attempt to resolve in some other way. This method returns a boolean flag which indicates to the client library if it should continue persisting the rest of the statements in the statement queue. If this method returns false, the statement queue is cleared. This method will not be called as a result of a timeout when connecting to the server (though the synchronous Flush() method should throw an exception as a result of a timeout). Interval posts that fail due to timeout should exercise some backoff algorithm, and reset the backoff when Flush() has been called.

To support disconnected scenarios, statements should be queued into an offline storage facility, provided by a class implementing the OfflineStorage interface, which should provide the following methods:

void AddToStatementQueue(Statement[] statements);
Statement[] GetQueuedStatements(int count);
void RemoveStatementsFromQueue(int count);

Client applications should be sure to limit TCAPI interaction to only statement reporting when disconnected, and should only call Flush() or any of the other TCAPI methods when connection is reestablished between the application and the LRS. Generally, a client application should call Flush() when it has detected a change of state from disconnected to connected.

Client libraries should strive to implement a reasonable default implementation of the OfflineStorage interface if their language or platform implies a standard offline storage system. If an application does not intend to support offline scenarios, it may signal to the client library that no offline storage should be used, in which case the statement queue can be maintained in some data structure that isn’t persisted to permanent storage.

The OfflineStorage component should be segregated to single users or intentionally shared environments. Specifically, a Flush() or interval post of statements should not affect the statement queue of other users, unless this is specifically intended.

Basic Examples:

Example 1 - Identifying Actor by Account, using OAuth authentication provider, storing statements and results:

```
//Create user from system information (account url, account id)
Account userAccount = new Account(system.getHomePageUrl(), system.getUserId());
Actor user = new Actor(system.getCurrentUserName(), userAccount);

//Declare activity
Activity activity = new Activity("example.com/MyActivityId", "Title", "description");

… Previous OAuth handshake resulted in tokenId, tokenSecret …
AuthenticationConfiguration authConfig = new OAuthAuthentication(
                consumerKey, consumerSecret, tokenId, tokenSecret);

TCAPI lrs = new TCAPIRest(getTCAPIEndpoint(), authConfig);

lrs.StoreStatement(new Statement(user, StatementVerb.Experienced, activity));

… User completes activity, with score, success, completion, and duration tracked ...

Statement stmt = new Statement(user, StatementVerb.Completed, activity);
stmt.setResult(new Result(earnedScore, success, completion, duration));
lrs.StoreStatement(stmt);
```

Example 2 - Identifying Actor by Email, using Basic authentication, storing statements with context:

```
//Create user from external system information (name, email)
Actor user = new Actor(system.getUserName(), system.getUserEmail());

//Declare activities
Activity activity1 = new Activity("example.com/MyActivity1", "Test Activity 1", "Testing");
Activity activity2 = new Activity("example.com/MyActivity2", "Test Activity 2", "Testing");
Activity course = new Activity("example.com/MyCourse", "TestCourse", "Test course");

//Setup statement context
Context ctx = new Context();
ctx.setContextActivities = new ContextActivities(null, course, null);

//Create basic authentication provider using username / password recognized by LRS
AuthenticationConfiguration authConfig = new BasicHttpAuthentication(username, password);
```

```
TCAPI lrs = new TCAPIRest(getTCAPIEndpoint(), authConfig);

… User interacts with activity1 ...

lrs.StoreStatement(new Statement(user, StatementVerb.Experienced, activity1, ctx));

… User interacts with activity2 …

lrs.StoreStatement(new Statement(user, StatementVerb.Experienced, activity2, ctx));
```

Example 3 - Fetching statements about current user using OAuth

```
//Create user from external system information (name, email)
Actor user = new Actor(system.getUserName(), system.getUserEmail());

… Previous OAuth handshake resulted in tokenId, tokenSecret …
AuthenticationConfiguration authConfig = new OAuthAuthentication(
                    consumerKey, consumerSecret, tokenId, tokenSecret);

TCAPI lrs = new TCAPIRest(getTCAPIEndpoint(), authConfig);

//Query for all statements made by this user, limiting page size
StatementQueryObject queryObject = new StatementQueryObject();
queryObject.setActor(user);
queryObject.setLimit(50);

//Process all statements, one page at a time, get more pages using moreUrl
StatementsResult statementsResult = lrs.GetStatements(queryObject);
boolean moreToProcess = (statementsResult.getStatements().length() > 0);
while(moreToProcess){
   … process statements from statementsResult.getStatements() …
   String moreUrl = statementsResult.getMoreUrl();
   moreToProcess = (moreUrl != null && moreUrl.length() > 0);
   if(moreToProcess){
      statementsResult = lrs.GetStatements(moreUrl);
   }
}
```

Example 4 - Updating Activity Profile document, respecting concurrency

```
//Create user from external system information (name, email)
Actor user = new Actor(system.getUserName(), system.getUserEmail());

AuthenticationConfiguration authConfig = new BasicHttpAuthentication(username, password);
```

```
TCAPI lrs = new TCAPIRest(getTCAPIEndpoint(), authConfig);
```

… User interaction that generates a new userScore, which will be added to a document with key ‘scoreboard’. Documents are free form, in this case, the scoreboard is a JSON document …

```
String activityId = “example.com/MyActivity1”;
boolean scoreProcessed = false;
int maxAttempts = 3;
int attempts = 0;

while (!scoreProcessed && attempts < maxAttempts){
   try {
      //Get current profile contents, as a string
      ActivityProfile actProfile = lrs.GetActivityProfile(activityId, “scoreboard”);
      String contentStr = new String(actProfile.contents, “UTF-8”);

      //Parse JSON string into ScoreBoard object, apply user’s score to it
      ScoreBoard scoreBoard = JSON.parse(contentStr);
      system.ApplyNewScoreToScoreboard(scoreBoard, userScore);

      //Create new version of this document with contents as new JSON scoreboard
      String newScoreboardStr = JSON.stringify(scoreBoard);
      ActivityProfile newProfile = new ActivityProfile(activityId, “scoreboard”, newScoreBoardStr);

      //Pass in previous profile document to respect concurrency issues
      lrs.SaveActivityProfile(newProfile, false, actProfile);
      scoreProcessed = true;
   }
   catch (ConflictException ce) {
      //Some other process updated the document while we were working on it, retry
      scoreProcessed = false;
      attempts += 1;
   }
}
```

Example 5 - Updating Activity State, ignoring concurrency

```
//Create user from external system information (name, email)
Actor user = new Actor(system.getUserName(), system.getUserEmail());

AuthenticationConfiguration authConfig = new BasicHttpAuthentication(username, password);
TCAPI lrs = new TCAPIRest(getTCAPIEndpoint(), authConfig);
```

… User interaction that generates a new ‘favorite page’ in the favoritePage variable, which will be added to a document with key ‘favorite_pages’. Documents are free form, in this case, the favorite page list is a JSON document …

```
String activityId = “example.com/MyActivity1”;
```

```
//Get current document content as JSON string, parse into favoritePagesList
ActivityState favPageState = lrs.GetActivityState(activityId, user, “favorite_pages”);
String favPagesContent = new String(favPageState.contents, “UTF-8”);
List favoritePagesList = JSON.parse(favPagesContent);

//Append new favorite page generated by user interaction
favoritePagesList.Append(favoritePage);

//Save new list of favorite pages by serializing list to JSON
favPagesContent = JSON.stringify(favoritePagesList);
lrs.SaveActivityState(new ActivityState(activityId, user, “favorite_pages”, favPagesContent));
```

Example 6 - Example of utilizing the TCAPICallback interface

```
public class MyTCAPICallback implements TCAPICallback {
   private Application system = null;  //Representation of external application
   public MyTCAPICallback(Application system){
         this.system = sytem;
   }
   void StatementsStored(Statement[] statements){
      system.logStoredStatements(statements);
   }
   boolean StatementsFailed(Statement[] failedBatch, Exception exception){
      system.logFailedStatements(failedBatch, exception);
      system.signalExitWithDialog(“An error occurred communicating with the LRS”);
      return false;
   }
};
```

… then, in the main file, we pass the callback object to the TCAPI client object …

```
Actor user = new Actor(system.getUserName(), system.getUserEmail());
AuthenticationConfiguration authConfig = new BasicHttpAuthentication(username, password);
TCAPICallback myCallback = new MyTCAPICallback(system);

TCAPI lrs = new TCAPIRest(getTCAPIEndpoint(), authConfig, myCallback);
```

… interaction as usual, client library will call the callback methods appropriately …