xAPI Moodle Library Reference (Moodle 3.9)

Introduction

This document contains a technical explanation on how the xAPI Moodle integration library works. The main purpose of this document is to keep the dev documentation while the oficial Dev Wiki page is not yet created.

The **Experience API (xAPI)** is an e-learning software specification that allows learning content and learning systems to speak to each other in a manner that records and tracks all types of learning experiences.

xAPI defines web services that can be used by any plugin or APP to connect to the LMS in a standard way. The main actors in xAPI communication are:

- **Client**: any plugin or APP that wants to use xAPI webservices
- **xAPI LRS (server site)**: responsible for storing and serve the data emitted by the clients

One good aspect about xAPI is that the amount of messages that can be shared between the clients and the LRS is quite limited. The main message type is called "**Statement**" and all statements can be summarized as "*An actor XX executes action YY on object ZZ*".

The typical statement object is a JSON element containing:

- **Actor**: the person or group that do something
- **Verb**: the action the Actor do
- **Object**: the object on the Verb is executed. There are more than one type of objects that can be defined here but, for now, you can think about it as a specific "Quiz Attempt" for example.
- Other fields: for now, the rest of xAPI fields have a basic data validation so every plugin is responsible for implementing it's own checks if they need them.

The xAPI specification can be consulted in the xAPI specs page: https://github.com/adlnet/xAPI-Spec

Moodle xAPI functionalities

The final objective of Moodle xAPI library is **NOT** to implement a full LRS inside Moodle but to provide an easy way to handle xAPI statements within any plugin that needs it.

For now Moodle will focus on supporting the main xAPI request which is called "**statement**". A statement must be seen as a tracking message that can be used by any plugin to store user activity directly to the Moodle without programming any additional web service.

The current library implements:

- A new webservice called **core_xapi_statement_post** to process xAPI statements and generate standard Moodle events.
- A class **\core_xapi\handler** that any plugin can extend in order to user xAPI and generate specific plugin contexts. This way any plugin could use xAPI without implementing any new web service.
- A class **\core_xapi\iri** to easily translate random information into valid IRI values (needed for xAPI objects and verbs)

- A class **\core_xapi\local\statement** to create and extract information from statement data.
- A bunch of predefined classes in **\core_xapi\local\statement\item_XXX** to generate xAPI statement items from the plugins (exemple provided in “Generating statements in PHP”).

Statement processing

The **core_xapi_statement_post** webservice receives a JSON encoded statement (could be an array or a single one) and a component frankenstyle name.

The processing sequence is:

1. The core xAPI library checks if the specified component has a xAPI handling class implementation. Otherwise it returns an error.
2. xAPI checks the statement structure, if any check fails, it returns an error. The main validations for now are:
    a. Check that only supported xAPI attributes are used. If the statement array has some format error all statements will be rejected (see “Statement data validation” for more information).
    b. Check all users and groups are created in the LMS
    c. Check the current user ($USER) is an actor of the statement (actors can be a single agent or a group)
3. For every statement:
    a. **xAPI asks the component handler to convert the statement into a standard event (function statement_to_event).** In case the plugin could not convert some statement into an event, this statement will be marked as “not processed” and will continue to the next statement.
    b. xAPI triggers the generated events
4. Once all statements are processed:
    a. If ALL statements are marked as “not processed” it returns an error.
    b. If some statement could be processed, returns an array containing “true” if the statement is processed and “false” otherwise.

Note that the plugin “statement_to_event” is responsible for:

- Grant that the specified user has permission to execute that statement.
- Provide a valid Moodle event to trigger
- Process any result attached to the statement

Statement data validation

The core_xapi_statement_post webservice will reject and statements batch that does not comply with the above xAPI data validation:

- **actor**: this is a mandatory field. Actor objectType attribute must be “Agent” or “Group”

    - **agent**: agent can be only real moodle users and must be identified by account (user id as name and wwwroot as homepage) or mbox (user email). The other agent identifiers are not supported (mbox_sha1sum and openid).
    - **group**: all groups must be real moodle groups identified by account (group id as name and wwwroot as homepage). Anonymous groups are not supported.
- **verb**: this is a mandatory field. Verb id must be provided in a valid IRI format.
- **object**: this is a mandatory field. The objectType attribute can be “Agent”, “Group” ar “Activity”:
    - **agent** or **group**: both have the same restrictions as when they act as actors.
    - **activity**: activity id must be provided as a valid IRI format.
        - **definition**: in case a definition is provided, the only accepted interaction types are: choice, fill-in, long-fill-in, true-false, matching, performance, sequencing, likert, numeric, compound and other.
- **result**:in case it is provided the duration must be provided in a valid ISO 8601 compatible with the PHP DateInterval class (More info: https://bugs.php.net/bug.php?id=53831).
    - **score**: all score fields are optional and do not have any particular data validation.
- **context**: it has no specific validation.
- **attachments**: if present must be an array of elements and every element must have: usageType, display, contentType, a valid usageType IRI, a numeric length and a sha2.
- **authority**: must be a valid xAPI actor.
- **timestamp**, **stored** and **version**: those optional fields are just strings with no particular validation or usage for now.

Generating statements in PHP

The xAPI library provides classes to translate Moodle elements into xAPI structures. Those statements could be sent to JS via the “$PAGE->requires->data_for_js” method.

The next example shows how to generate a basic statement:

```
use core_xapi\local\statement;
use core_xapi\local\statement\item_actor;
use core_xapi\local\statement\item_verb;
use core_xapi\local\statement\item_activity;

(...)
// Generate statement.
$statement = new statement();
$statement->set_actor(item_agent::create_from_user($USER));
$statement->set_verb(item_verb::create_from_id('bake'));
$statement->set_object(item_activity::create_from_id('cake'));
```

As can be seen in the example, the function item_agent has a static method to generate a valid xAPI actor from a user record. The result object of executing that code will be something like:

```
{
    "actor": {
        "objectType": "Agent",
        "account": {
```

```
            "homePage": "http:\/\/localhost\/m\/H5P",
            "name": "2"
        }
    },
    "verb": {
        "id": "http:\/\/localhost\/m\/H5P\/xapi\/verb\/bake"
    },
    "object": {
        "objectType": "Activity",
        "id": "http:\/\/localhost\/m\/H5P\/xapi\/object\/cake"
    }
}
```

For the object and verbs, xAPI uses a standard IRI format. The xAPI library will convert any string into a valid IRI but it can also use real IRI identifiers to generate standard verbs and object structures.

The next example is more a less the same but using valid IRI values for verbs and objects:

```
use core_xapi\local\statement;
use core_xapi\local\statement\item_actor;
use core_xapi\local\statement\item_verb;
use core_xapi\local\statement\item_activity;

(...)
// Generate statement.
$statement = new statement();
$statement->set_actor(item_agent::create_from_user($USER));
$statement->set_verb(item_verb::create_from_id('http://adlnet.gov/xapi/verbs/bake'));
$statement->set_object(item_activity::create_from_id('http://adlnet.gov/xapi/activities/cake')
);
```

The resulting structure will be:

```
{
    "actor": {
        "objectType": "Agent",
        "account": {
            "homePage": "http:\/\/localhost\/m\/H5P",
            "name": "2"
        }
    },
    "verb": {
        "id": "http:\/\/adlnet.gov\/xapi\/verbs\/bake"
    },
    "object": {
        "objectType": "Activity",
        "id": "http:\/\/adlnet.gov\/xapi\/activities\/cake"
    }
}
```

The xAPI library also provides classes to convert moodle groups into valid xAPI group actors. To generate one the code is as following:

```
use core_xapi\local\statement;
use core_xapi\local\statement\item_actor;
use core_xapi\local\statement\item_verb;
```

```
use core_xapi\local\statement\item_activity;

(...)
// Generate statement.
$group = groups_get_group($groupid);
$statement = new statement();
$statement->set_actor(item_group::create_from_group($group));
$statement->set_verb(item_verb::create_from_id('bake'));
$statement->set_object(item_activity::create_from_id('cake'));
```

The result will be:

```
{
    "actor": {
        "objectType": "Group",
        "name": "Group A",
        "account": {
            "homePage": "http:\/\/localhost\/m\/H5P",
            "name": "1"
        }
    },
    "verb": {
        "id": "http:\/\/localhost\/m\/H5P\/xapi\/verb\/bake"
    },
    "object": {
        "objectType": "Activity",
        "id": "http:\/\/localhost\/m\/H5P\/xapi\/object\/cake"
    }
}
```

xAPI statements attributes implementation

All xAPI statement attributes have it's own implementation in case any plugin needs them. All classes are implemented in the namespace "core_xapi\local\statement", Those are the classes implemented:

- **Item:** all xAPI attributes derived from this class. This class provides a method to generate a valid item from a xAPI data structure (create_from_data), implements de json serialization and provides the basic "get_data" method to access the raw element data structure.
    - **Item_actor**: this is an abstract class extended by both two xAPI valid actors:
        - **Item_agent**: an individual user. The class has a static method called "create_from_user" to generate a valid xAPI actor from a moodle user record.
        - **Item_group**: a group statement. Moodle xAPI implementation only accepts named groups (it will reject any anonymous group). To generate a valid xAPI group from a moodle group the class has a static method called "create_from_group". It also provides methods to get all moodle users from that group (get_all_users) and the group record itself (get_group).
    - **Item_verb**: represents the statement verb. The class has a static method "create_from_id" which accepts both valid IRI values or any random string to create a valid xAPI verb structure. IT algo has a method to get back the verb value (get_id).
    - **Item_object**: this is an abstract class extended by all possible xAPI valid objects:
        - **Item_agent** or **Item_group**: both can be statements actor or objects as well
        - **Item_activity**: activity can be any kind of string value which the plugin decides. It has the same methods as item_verb to create and get this value ("create_from_id" and "get_id"). The xAPI activity could have also an atribute

called “definition” that contains a SCORM like data defining the user interaction.
        - **Item_definition**: contains all the specific interaction details such as the interaction type (in SCORM format), the user answer or the correct answer pattern.
    - **Item_result**: this optional attribute represents any kind of results and score derived from the user interaction. IT has no strict data validation and it has to be created directly from a valid xAPI data structure using the method “create_from_data”. It provides two methods to get the result score if present (get_score) and one to convert the result duration into seconds (get_duration). It uses also one subclass to handle score:
        - **Item_score**: the class representing the xAPI score data. It has no validation and no specific methods.
    - **Item_context**: the class representing the xAPI context data. It has no validation and no specific methods.
    - **Item_attachment**: validate the xAPI attachment structure

Adapting a plugin to use xAPI functions

All the plugins could use the new xAPI functionalities to:

- Send xAPI statements, interact with them, trigger events and generate log store entries.

Step 1: program a xapi handle class

Implement a class “\PLUGINNAME\classes\xapi\handler” which extends “\core_xapi\handler” and override this methods:

- **statement_to_event (statement $statement): core\event\base**: to convert a statement into a valid Moodle event
- In case that the plugin wants to accept also Group statements (by default any Group statement will be rejected) it can override the method **supports_group_actors(): bool**

Step 2: code events

All xAPI statements MUST be converted into a valid Moodle event. To do so the plugin needs to **code all the necessary events** for this.

Useful methods to handle xAPI statements inside plugins

The **core_xapi\local\statement** provides several methods to extract information from a statement object.

get_user (): stdClass

- Return the Moodle user represented by the statement actor. In the case of a group actor, this function will throw an xapi_exception.

get_all_users(): array

- Will return an array with all Moodle users represented in the xAPI actor object. This can be used in both single agent and group statements.

get_group(): stdClass

- Return the Moodle group record represented in the xAPI actor. In the case of a single agent actor, this function will throw an xapi_exception.

get_verb_id (): string

- Return the current verb ID value from a statement.

get_activity_id(): string

- Return the current activity ID. If the statement object is not an activity (could be also an agent or a group) the method will throw an xapi_exception.

minify (): ?array

- Return a minified version of the statement suitable for using as a “other” field of a Moodle event.

get_actor(): item_actor

- Return the statement actor.

get_verb(): item_verb

- Return the statement verb.

get_object(): item_object

- Return the statement object.

get_context(): item_context

- Return the statement context attribute.

get_result(): item_result

- Return the statement result attribute.

get_timestamp(): string

- Return the statement timestamp attribute.

get_stored(): string

- Return the statement stored attribute.

get_authority(): item_actor

- Return the statement authority attribute.

get_version(): string

- Return the statement version attribute.

get_attachments(): item_attachment

- Return the statement attachments attribute.

Using Ajax lib to send a statement to Moodle

The webservices have a different behaviour depending if it is processing a single statement or an array of them.

This code could be an example of a single xAPI statement post JS code:

```
require(['core/ajax'], function(ajax) {
    var data = {
        component: 'mod_h5p',
        requestjson: JSON.stringify(statement)
    };
    var promises = ajax.call([
       {
           methodname: 'core_xapi_statement_post',
           args: data
       }
    ]);
    promises[0].done(function(response) {
       // Put some success messages...
    }).fail(function(ex) {
       // Do some failure message handling...
    });
});
```

On the other hand, if you send more than one statement, the webservice will return an array of booleans indicating which statements are successfully stored and which not. Only in the case of none of the statements could be processed (usually because of an invalid statement structure) the webservice will return an error.

This could be an example of sending more than one statement to Moodle:

```
require(['core/ajax'], function(ajax) {
    var data = {
        component: 'mod_h5p',
        requestjson: JSON.stringify(statements)
    };
    var promises = ajax.call([
        {
            methodname: 'core_xapi_statement_post',
            args: data
        }
    ]);
    promises[0].done(function(response) {
        // Check if it's a success or not.
        for (var i = 0; i < response.length; i++) {
            if (response[i]) {
               // Do some success handling.
            } else {
               // Do some failure handling.
            }
        }
    }).fail(function(ex) {
        // None of the statements are processed.
    });
});
```