xAPI Moodle Library Reference (alpha 1)

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 don't have any kind of data validation so every plugin is responsible for implementing it's own checks.

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 send 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 process is:

1. xAPI checks if the 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
    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

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 will show 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 fake valid IRI but it also uses real IRI identifiers to generate standard verbs and objects 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 helper could also generate Group statements where the subject of the statement is a valid course group. 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"
    }
}
```

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. The item_actor class provides a "get_data()" method to access the raw xAPI data.

get_verb(): item_verb

- Return the statement verb. The item_verb class provides a "get_data()" method to access the raw xAPI data.

get_object(): item_object

- Return the statement object. The item_object class provides a "get_data()" method to access the raw xAPI data.

get_context(): item

- Return the statement context attribute. Note that the item class has no previous validation so the use of "get_data()" method to validate the data is mandatory.

get_result(): item

- Return the statement result attribute. Note that the item class has no previous validation so the use of "get_data()" method to validate the data is mandatory.

get_timestamp(): item

- Return the statement timestamp attribute. Note that the item class has no previous validation so the use of "get_data()" method to validate the data is mandatory.

get_stored(): item

- Return the statement stored attribute. Note that the item class has no previous validation so the use of "get_data()" method to validate the data is mandatory.

get_authority(): item

- Return the statement authority attribute. Note that the item class has no previous validation so the use of "get_data()" method to validate the data is mandatory.

get_version(): item

- Return the statement version attribute. Note that the item class has no previous validation so the use of “get_data()” method to validate the data is mandatory.

get_attachments(): item

- Return the statement attachments attribute. Note that the item class has no previous validation so the use of “get_data()” method to validate the data is mandatory.

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.
    });
});
```