Mosaic Learning SCORM Documentation

May 2017 – SCORM 1.2

# Software Stack

When designing SCORM content this is the structure of the software that will be employed. The LMS is at the base layer, which will talk to the SCORM layer, the SCORM layer implements an API for interactions. It is possible to interact with the API directly, but for simplicities sake the use of the Pipwerks wrapper will significantly ease development.

None of the previous layers will ever need to be modified by multimedia developers. The layers that will require modification sit above the Pipwerks layer. The ML\_SCORM layer is a Javascript file that is a further wrapper of the Pipwerks API with many convenience functions, simplicity shortcuts, and local constants available for consistency. This is a file that is currently in development, so while not complete, once it is more stable should need relatively little editing for day to day programming tasks.

Any custom functionality (bookmarking, objective tracking, SCO completion, exercise scoring, etc.) should be placed in a custom Javascript file that will make use of the ML\_SCORM API, and occasionally the Pipwerks API. Ideally all Pipwerks functions will be wrapped in the ML\_SCORM file for simplicities sake.

You should include the Pipwerks file, ML\_SCORM, and your custom Javascript in each HTML file that represents your SCO, and they should be loaded in that order. If the SCO is self-contained all communication between the SCO and the LMS will be done through the custom Javascript file.

If the SCO is a wrapper for smaller lessons/objectives (as in DC Theory) the SCO HTML can also contain an iframe to load more granular content. It is important to note that the iframe will not be responsible for communicating with the LMS. It will communicate with the parent HTML which will in turn communicate with the LMS. It should not have access to the ML\_SCORM or Pipwerks files. It will have its own custom Javascript file however for its internal housekeeping and to communicate with its parent layer.

## Constants

Several constants have been provided to ensure compatibility with the SCORM 1.2 storage options. When setting SCORM variables that fall into one of these categories be sure to use the constants to maintain consistency.

The STATUS constant allows access to the status conditions for objectives and lessons.

const STATUS = {

PASSED : "passed",

FAILED : "failed",

COMPLETED : "completed",

INCOMPLETE : "incomplete",

BROWSED : "browsed",

NOT\_ATTEMPTED : "not attempted"

}

The EXIT constant allows access to the exit conditions for SCORM termination.

const EXIT = {

TIMEOUT : "time-out",

SUSPENED : "suspend",

LOGOUT : "logout",

NORMAL : ""

}

The INTERACTION constant allows access to available interaction types.

const INTERACTION = {

TRUE\_FALSE : "true-false}",

CHOICE : "choice",

FILL : "fill-in",

MATCH : "matching",

PERFORMANCE : "performance",

LIKERT : "likert",

SEQUENCE : "sequencing",

NUMERIC : "numeric"

}

The RESULT constant allows access to available interaction result types

const RESULT = {

CORRECT: "correct";

WRONG: "wrong",

UNANTICIPATED: "unanticipated",

NEUTRAL: "neutral"

}

# Objectives/Interactions

SCORM provides you access to the ambiguously defined Objectives and Interactions. These are intentionally loosely defined to allow developers and designers the ability to use them to cover a wide variety of their needs. They are similar in their implementation and overlap to some extent in their functionality.

The following two sections will provide an overview of the provided functionality and details on how to implement them.

## Objectives

Of the two objectives are simpler to understand and use. An objective can be used to track anything you might want to keep progress on or keep scored. It contains three pieces of data: id, status, and score. The only required portion is id, but you will probably want at least one of the other two in order to track the objective. These can be used to track mastery, hold graded quizzes, track completion of material, or anything that deals with completeness, or numeric scoring.

### Id

id is a string that can be used to identify the objective. It should be human readable and can contain alphanumeric digits.

### status

status is a legal value from the STATUS constant (PASSED, FAILED, COMPLETED, INCOMPLETE, BROWSED, NOT\_ATTEMPTED) and should be used to indicate the current status of the objective.

### score

score is a container object that holds three pieces of data: min, max, and raw

The usage of score is frustratingly poorly documented, but min and max are the minimum and maximum ranges of the score and raw is the actual score the student received. To make your life easier leave min at 0 and max at 100 and treat raw like a percentage if you are going to use scores. In theory you can use other values for max and min, but the documentation is not very clear on what it is expecting, so if you want to use other values proceed at your own peril.

### The objective Class

Objectives are tracked through SCORM in the cmi.objectives array. They should be added into the next available slot which can be found with cmi.objectives.\_count, but their order in the array has no other effects. You can access individual objectives through cmi.objectives.n where n is the index of the objective. From there you can access the individual components through dot notation to their names (id, status, score).

To use objectives with ml\_scorm you have access to the Objective class. To make a new objective you can use the constructor new Objective(index, id). index is the index tracked by SCORM in the cmi.objectives array, and id is human readable name of the objective. score is automatically generated at creation time and initialized to {min: 0, max: 100, raw: 0} and status is initialized to not\_attempted. After initialization the object will register itself with the LMS.

After creating an Objective object, accessing or updating any of its properties through dot notation ie objective.id or objective.id = “Objective 3” will keep your local copy of the object updated with the LMS version of the objective. You do not need to manually manage any of the SCORM updating or saving, it is all handled internally by the object.

You have access to two internal methods to use on Objective objects. The first is the complete() method, which is a shortcut for updating status to STATUS.COMPLETE. Other STATUS shortcuts could be just as easily implemented if so desired. The second is the save() method. This method is largely redundant since updating any values will already save to the LMS, but if you are ever editing the internal data directly (\_id, \_status, \_score) you can use this to force all data stored on the object to save on the LMS.

### The Score Class

The Score class is a simple data class used by the Objective class but could potentially be used in other places. It contains a value for raw, min, and max. It is recommended that min be left at 0, max be left at 100 and raw be used as a percentage value representing the calculated student score.

class Score {

constructor (raw=0, min=0, max=100) {

this.raw = raw;

this.min = min;

this.max = max;

}

}

## Interactions

See page 56 in SCORM\_1\_2\_RunTimeEnv.pdf for more detailed information

Interactions are similar in usage to Objectives, but are more complex in their implementation, and also as vaguely defined. It can be as comprehensive as logging every single click a student makes on a page to as broad reaching as a quiz for the entire SCO. Also if you are making a quiz every question should be its own interaction. An interaction can be any input or response to stimuli you want to track from the student.

Interactions are stored on the LMS in the array cmi.interactions. Each interaction has access to a variety of data points: id, type, objectives, time, correct\_responses, weighting, student\_response, result, and latency. You can access individual interactions through cmi.interactions.n where n is the index of the interaction and the previous options are accessible through dot notation on the n. You are responsible for managing this number manually as the interactions will otherwise be added on to the end of the array which can cause unintended ordering if students are allowed to navigate non-linearly through the course.

Of the available interactions two of these are required: id, type. Type isn’t strictly required but it is a prerequisite for using correct\_responses and student\_response which depend on knowing the type. type requires one of the variables made available in the INTERACTION constant.

### id

The id field is a human readable value to indicate the name of the interaction using alphanumeric digits. This can be a quiz question number, an id number, a code indicating its place in a module or anything that makes sense to you.

### Type

The type field denotes the category of interaction being presented to the user. It is semi-required in that correct\_responses and student\_response both require this to be set to know what kind of data to accept. The type must be one of the seven legal values from the INTERACTION constant: TRUE\_FALSE, CHOICE, FILL, MATCH, PERFORMANCE, LIKERT, SEQUENCE, or NUMERIC.

#### True\_False

True/False interactions are questions with two available options for the student to select. If this type is selected the LMS will expect a one character value of either 0 or 1 or “t” or “f” as a response. True or False can also be used, but the values will be truncated at the first character and submitted as either “T” or “F”.

#### Choice

Choice is for multiple choice questions. Any question that has a set of options for a student to select can be considered multiple choice. Drag and Drop and Hot Spot type questions can be represented by choice or the matching type described below. Multiple answers can be considered correct and some answers can be considered more correct than others. If this type is selected the LMS will expect a single character alphanumeric value as a response, or if multiple answers are allowed an array separated by commas and delimited by curly brackets, i.e. {1,3,4}.

#### Fill

Fill denotes a fill in the blank type question where a student is required to input a simple response in a text format. If this type is selected the LMS will expect an alphanumeric string with significant spaces as a response i.e. “New York”.

#### Match

Match type interactions are interactions that require the pairing of two sets of items. If this type is selected the LMS will expect a set of pairs of identifiers, source and target, separated by a period. I haven’t found an example of this yet, but my best guess is this is also delimited by curly brackets with commas between values i.e. {1.A,2.B,3.C}

#### Performance

Performance is poorly documented, but is for when a student response consists of a series of steps. The expected response is an alphanumeric string consisting of no more than 255 characters.

#### Likert

Likert type interactions are used for when an answer is needed on a scale. Scale in this instance should be considered a range of values that are all considered valid, for instance:

Strongly Agree – Agree – Neutral – Disagree – Strongly Disagree

and not questions where the final result is a singular answer or number, for instance:

Drag the slider to the correct value to demonstrate the correct reading

Likert interactions have no incorrect answers. They can be used in assessing student confidence in an answer, and can be used to affect the score of a separate question, for instance part B of the question

1.a What state is the Statue of Liberty located in?

1.b How confident are you in your answer?

What the LMS is expecting on this type of interaction is poorly documented beyond saying

“This field may be left blank.”

If it is not left blank my best guess is that it is an alphanumeric string consisting of no more than 255 characters i.e. “Extremely Confident”

#### Sequence

Sequence interactions are those which require presented content to be organized in a sequential manner before submission. This could be selecting steps in a procedure, placing values from largest to smallest or similar interactions. If this type is selected the LMS will expect a series of single alphanumeric characters. The exact specifications are not documented, but my best guess is it is expecting curly brackets and commas to delimit the values i.e. {3,5,2,1,4}

#### Numeric

Numeric type interactions are expecting a single integer or floating point decimal number. This can be the result of a calculation or a measurement, a year, or any value that can be represented by a single number.

Numbers retrieved from dragging sliders, spinning wheels, or other UI elements should be recorded here and not in likert types which do not have incorrect answers. The documentation does not mention fractions, but it would probably be safe to assume fractions should be converted to decimal representation before submission. If this type is selected the LMS will expect a single number with or without a decimal portion i.e. “42”

### objectives

The objectives array allow you to create shared relationships between interactions. For instance if you have 5 questions on a quiz, each question represented by an interaction could have an objective of “Quiz 1”. Each interaction is allowed to have multiple objectives so a single interaction may have an objective for a quiz, a mastery, and a completion for a module.

Objectives are rather unfortunately named. The objectives on the interactions have no actual linking to the objectives tracked in cmi.objectives. In the ml\_scorm package however there is a direct link, so each objective is an Objective object, but there is nothing to reflect that on the LMS.

Interaction objectives are stored in the LMS in an array on each interaction at cmi.interactions.n1.objectives.n2 where n1 is the index of the interaction and n2 is the index of objective in the interactions objective array.

The only field on the objectives array is id which is a string consisting of alphanumeric digits.

You also have access to cmi.interactions.n1.objectives.\_count which will return the number of objectives currently in the array.

### Time

The time field allows you to track a timestamp for an interaction. What the timestamp tracks is contradictorily defined as both the following:

“Point in time at which the interaction was first made available to the student for student interaction and response”

“Identification of when the student interaction was completed.”

For the purposes of the ml\_scorm package the latter definition will be assumed. This will record the timestamp when the interaction is submitted for completion by the student. Using the ml\_scorm package you should never have to worry about this value directly as it will be updated internally.

If you ever do need to update the internal value \_finishTime the LMS is expecting a timestamp in the format of HH:MM:SS.SS where hours can be between one and two digits and seconds can have an optional decimal portion with between zero and two digits.

### Correct\_responses

The correct\_responses field is an array available on every interaction that holds the correct response patterns available for an interaction. They are accessible at cmi.interactions.n1.correct\_responses.n2 where n1 is the interaction and n2 is the position of the correct response in the array. Each interaction can have zero, one, or multiple correct responses. Each individual correct pattern is available at cmi.interactions.n1.correct\_responses.n2.pattern.

The format of the patterns are dependent upon the type of interaction denoted in the type field. See the type description above for the expected formats for each type.

### Weighting

Each individual interaction can have a different weight on the final score. Interactions on their own do not contain any score or grade so I’m not 100% sure where this is being used or how this gets calculated. Regardless to use this field simply set this to a number. Higher numbers will be weighted more significantly at whatever stage they are calculated at.

### Student\_response

This is the input received from the student. The format of the response is dependent upon the type of interaction denoted in the type field. See the type description above for the expected formats for each type.

### Result

This field is somewhat confusingly named but can be thought of similar to the status field of the SCO. There are four legal result options available in the RESULT constant: CORRECT, WRONG, UNANTICIPATED, and NEUTRAL.

### Latency

This field is the amount of time that has passed since the presentation of the interaction and the time the student completes the interaction. Using the ml\_scorm package you should never have to worry about this value directly as it will be updated internally.

If you ever do need to update the internal value \_latency the LMS is expecting a timestamp in the format of HHHH:MM:SS.SS where hours can be between one and four digits and seconds can have an optional decimal portion with between zero and two digits.

### The Interactions class

Similar to the Objectives class the Interactions class is a Javascript object used to manage all your manipulation of interactions and handle all the communication with the LMS behind the scenes. To make a new Interaction object you use the constructor new Interaction(index, config) where index is the index in the cmi.interactions array and config is a configuration object that contains any relevant parameters to the interaction you are trying to create. The values passed in through the config object will be used to initialize the Interaction object and register it with the LMS at creation time.

Dev Note: This currently performs a scorm.save() operation on every object creation and could be a potential performance bottleneck. A more complicated solution could involve withholding the save operation until all interactions have been instantiated to cut down on calls to the LMS.

Also similar to the Objectives class accessing and updating properties will keep the interactions synced with the LMS. You do not need to manually manage any of the SCORM updating or saving, it is all handled internally by the object.

You have access to several internal methods to use on Interaction objects. The first two are begin() and complete() which should be called at the start and end of an interaction. At the time of writing they only handle tracking of the start/finish times and latency, but will most likely be updated to also receive an optional callback function so that custom code can be executed on presentation and completion of an interaction.

The third is the save() method. This method is largely redundant since updating any values and proper use of begin() and complete() will already save all relevant data to the LMS, but if you are ever editing the internal data directly (\_id, \_type, \_objectives, \_startTime, \_finishTime, \_correct\_responses, \_weighting, \_student\_response, \_result, \_latency) you can use this to force all data stored on the object to save on the LMS.

The remaining functions deal with formatting time formatCurrentTime() and formatTime() and initialization of the object initialize() and are all called automatically. They do not need to be called externally, unless the time formatting ones are useful to you.

Dev Note: At the moment the formatTime() takes in a time in milliseconds and returns a value formatted for the LMS expectation of latency. If a value is entered that expects a time longer than 24 hours to be returned it will receive unexpected results until this can be updated to a more complex formula. Since it is unlikely we will need latencies of such scale I implemented this the easy way for the moment and can update it if needed.

# Imsmanifest.xml

In order to prepare a SCORM package for deployment to an LMS you will need to include an XML document named imsmanifest.xml which will contain a list of all the files used in the module as well information on how those files should be organized and presented. The imsmanifest.xml file should exist at the root level of your project. Once it is in place, to deploy the SCORM package you simply need to zip the root folder and upload it to an LMS.

There are four main sections that the manifest must contain: the <manifest> <metadata> <organization> and <resources> sections. I will try and document them to the best of my understanding, but for well documented official examples check the official SCORM examples available in the examples folder in the AllGolfExamples.zip folder. Note that the zip file has examples for versions other than 1.2 which is the version being used throughout this document, so only look at the ones using SCORM 1.2.

## Manifest

The manifest is the outermost tag which holds the remaining tags. It references a schema which can be accessed through the links provided below or through a local copy at the root of this project in the scorm12schemadefinition.zip folder.

The only important field in this tag is identifier which should use reverse domain name notation to identify the course and provider.

<?xml version="1.0" standalone="no" ?>

<manifest identifier="com.mosaiclearning.example.eric" version="1"

xmlns="http://www.imsproject.org/xsd/imscp\_rootv1p1p2"

xmlns:adlcp="http://www.adlnet.org/xsd/adlcp\_rootv1p2"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.imsproject.org/xsd/imscp\_rootv1p1p2

imscp\_rootv1p1p2.xsd

http://www.imsglobal.org/xsd/imsmd\_rootv1p2p1

imsmd\_rootv1p2p1.xsd

http://www.adlnet.org/xsd/adlcp\_rootv1p2

adlcp\_rootv1p2.xsd">

</manifest>

## Metadata

In theory there are other things you can put in metadata. Unless you need something really fancy just put this boilerplate immediately after the opening manifest tag. The important thing is that you are declaring the version of SCORM which for our purposes will be 1.2.

<metadata>

<schema>ADL SCORM</schema>

<schemaversion>1.2</schemaversion>

</metadata>

## Organization

SCORM packages are organized in a hierarchical manner. You can control the organization and presentation of this information through the organization tags. The LMS uses this data to determine what order and grouping the files need to be presented to the student in.

The following is an example organization and how it is presented in the LMS. It defines an organization named example\_1 and sets it as the default organization of the project. It then sets up 3 nested levels of content, each containing 3 items per level. Although this example is simple and linear, you can have as many levels of nested content with as many branches and items per level as you need.

<organizations default=”example\_1">

<organization identifier="example\_1 ">

<title>Example Scorm Package</title>

<item identifier="l\_1">

<title> Level 1 </title>

<item identifier="l\_1a\_item " identifierref="l1a\_resource">

<title> Level 1.a </title>

</item>

<item identifier="l\_1b\_item " identifierref="l1b\_resource">

<title> Level 1.b </title>

</item>

<item identifier="l\_1c\_item " identifierref="l1c\_resource">

<title> Level 1.c </title>

</item>

<item identifier="l\_2">

<title> Level 2 </title>

<item identifier="l\_2a\_item " identifierref="l2a\_resource">

<title> Level 2.a </title>

</item>

<item identifier="l\_2b\_item " identifierref="l2b\_resource">

<title> Level 2.b </title>

</item>

<item identifier="l\_2c\_item " identifierref="l1\_resource">

<title> Level 2.c </title>

</item

<item identifier="l\_3">

<title> Level 3 </title>

<item identifier="l\_3a\_item " identifierref="l1\_resource">

<title> Level 3.a </title>

</item>

<item identifier="l\_3b\_item " identifierref="l1\_resource">

<title> Level 3.b </title>

</item>

<item identifier="l\_3c\_item " identifierref="l1\_resource">

<title> Level 3.c </title>

</item>

</item>

</item>

</item>

</organization>

</organizations>



In this diagram you can see how the above organization is presented in the LMS. There is a top level Level 1 section with three nested course items below that with Level 2 and Level 3 following the same pattern. There can be multiple items and nestings on each level.

### Organizations

The highest level of organization tag is the organizations tag. Each SCORM package can hold multiple organizations of the content for instance if you wanted to have multiple levels of difficulty using the same content, or organize different learning tracks to get to the end. In practice you will probably only ever need to use one so you shouldn’t need to do too much with this tag beyond using it to enclose the rest of your hierarchy.

The one thing to make sure of is that the default attribute is set to the identifier of your organization.

<organizations default=”example\_1">

...

</organizations>

### Organization

Each organization of content needs its own organization tag underneath the organizations container. The tag needs an identifier attribute which can be passed to the organizations default attribute so that it loads by default on startup. The identifier should be a unique string within the package or the LMS can get confused about what it needs to serve. Each organization also needs a title tag which needs to contain a string that is the title of the entire package.

<organization identifier=”example\_1">

<title>Example Scorm Package</title>

...

</organization>

### Item

Inside the organization tag the item tag is used to establish the hierarchy of the course. There are two ways to use the item tag: as a section header, which SCORM calls an aggregation item, and as a piece of content. They both have the same basic format, the latter just has more attributes defined. You need at least one content item per organization but can have as many as you need to hold your course.

To use it as a section heading only two things are required: the identifier attribute which should contain a unique string and can follow any format that makes sense to you, and a title tag which should contain a string that will be used as the section heading in the LMS.

<item identifier="l\_1">

<title> Level 1 </title>

</item>

To use it as a piece of content use the same format as above, but also add the identifierref attribute. The identifierref is a reference to a resource that will contain the content to be displayed. Multiple items can reference the same resource in this field. The title tag here will be the name of the actual content and can be displayed as a clickable item in the LMS.

<item identifier="l\_1a\_item" identifierref="l1a\_resource">

<title> Level 1.a </title>

</item>

If your content takes a querystring in the url you can add an optional parameters attribute which takes a string that will be passed through to the url on load.

<item identifier="l\_1b\_item" identifierref="l1b\_resource" parameters="?objective=1b" >

<title> Level 1.b </title>

</item>

Items can be nested inside of each other to create a hierarchy. Both headers and content can be nested inside each other as deeply or as widely as needed.

<item identifier="l\_1">

<title> Level 1 </title>

<item identifier="l\_1a" identifierref="l1a\_resource">

<title> Level 1.a </title>

</item>

</item>