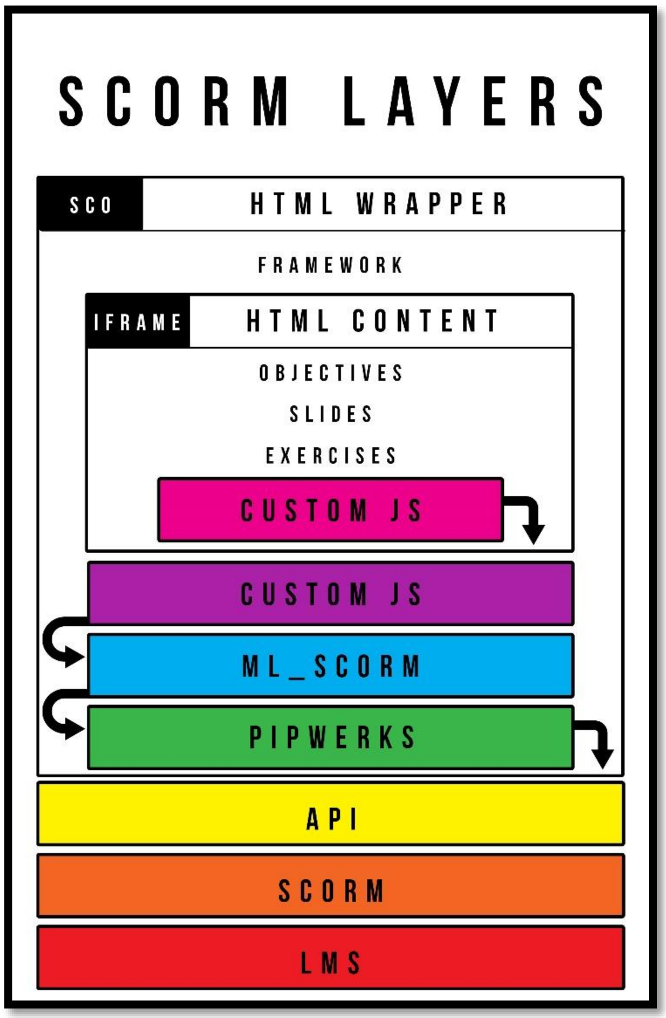
# MOSAIC LEARNING SCORM DOCUMENTATION

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.

# ML_SCORM.JS

*This file is in active development so the following documentation is subject to change. Best attempts will be made to update the documentation as the framework changes. The most recent version of this project can be found at:* [https://github.com/EVMosaic/ml_scorm](https://github.com/EVMosaic/ml_scorm)

The `ml_scorm.js` file is the heart of the Mosaic Learning SCORM development process. It is the middle man between the SCORM API and your custom Javascript which will drive your page. In the following section I will briefly describe the most important parts of it. Once you understand the core principles the rest should be fairly straight forward to understand.

The main engine behind `ml_scorm.js` is the Pipwerks SCORM wrapper (`SCORM_API_wrapper.js`). It is included in the git repository and also available for download at [https://pipwerks.com/laboratory/scorm/](https://pipwerks.com/laboratory/scorm/) This needs to be loaded before `ml_scorm.js`

```
let scorm = pipwerks.SCORM;
```

Pipwerks gives you access to the `pipwerks.SCORM` object. For convenience it is provided under the `scorm` shortcut in `ml_scorm.js`. Any Pipwerks functions can be accessed through dot notation on the `scorm` variable.

```
let lmsConnected = false;
```

Another important global variable is the `lmsConnected` variable. This will be initialized with the rest of the SCORM initialization. If the LMS cannot be connected to this will be set to `false` and prevent any interactions with the LMS. This is not something that needs to be actively managed.

# 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"
}
```

## DEBUG FUNCTIONS

Several debug functions have been included to help with debugging code. These are wrappers of the various `console` functions and included under the `DEBUG` constant. The main difference is the inclusion of the `DEBUG_ENABLED` flag which when set to false will bypass all debug logging. You can toggle this to quickly strip all debug output from your code. Any output that should remain regardless of being in debug or production should be implemented using the standard `console` functions.

```
const DEBUG_ENABLED = true;

const DEBUG = {
  LOG : function(msg) {
    if (DEBUG_ENABLED) {
      console.log(msg);
    }
  },

  ERROR : function(msg) {
    if (DEBUG_ENABLED) {
      console.error(msg);
    }
  },

  WARN : function(msg) {
    if (DEBUG_ENABLED) {
      console.warn(msg);
    }
  },

  INFO : function(msg) {
    if (DEBUG_ENABLED) {
      console.info(msg);
    }
  }
}
```

## CORE FUNCTIONS

The following four functions make up the bulk of the `ml_scorm.js` framework. The first two are necessary for opening and closing the connection to the LMS and the last two are the main workhorses for interacting with the LMS. Nearly all remaining functions are convenience functions that wrap these two in order to make it easier to interact with the LMS.

```
function initSCO() {
  lmsConnected = scorm.init();
}
```

`initSCO()` **must be called before any other SCORM functions can be called.** It should be done at the beginning of your custom Javascript file on every SCO that you create. If it is able to communicate with the LMS it will set `lmsConnected` to `true` and allow you to interact with the LMS with the remaining functions.

Under the hood this calls `pipwerks.SCORM.connection.initialize()` which in turn is responsible for calling `LMSInitalize()`through the LMS API implementation.

```
function closeSCO() {
  scorm.quit();
  lmsConnected = false;
}
```

`closeSCO()` must be called before closing the SCO or the window containing it. It should be called from the custom Javascript file when the SCO is no longer needed. Either `closeSCO()` or a function containing it and any other cleanup tasks should be bound to both the `unload` and `beforeunload` events to ensure it is reliably called. It will only ever be called once. Once `closeSCO()` is called the connection with the LMS is terminated and cannot be reopened again until the page has been reloaded.

Under the hood this calls `pipwerks.SCORM.connection.terminate()` which in turn is responsible for calling `LMSFinish` through the LMS API implementation;

```
function getValue(param) {
  DEBUG.INFO(`retrieving value for ${param}`);
  if (lmsConnected) {
    let value = scorm.get(param);
    DEBUG.LOG(`found value of ${value}`);
    return value;
  } else {
    DEBUG.WARN('LMS NOT CONNECTED');
  }
}
```

After the LMS has been initialized you have the ability to interact with it from your custom javascript file. The first way you can do that is by retrieving values from the LMS using the `getValue(param)` function. If the LMS is connected to it will look up the requested parameter and return the value found.

This is a wrapper of the `pipwerks.SCORM.get(param)` function. The only advantage of calling it this way is that it automatically checks if the LMS is connected before retrieving the value, and allows for custom debug messaging.

```
function setValue(param, value) {
  if (lmsConnected) {
    DEBUG.LOG(`setting ${param} to ${value}`);
    scorm.set(param, value);
    scorm.save();
  } else {
    DEBUG.WARN('LMS NOT CONNECTED');
  }
}
```

The other side of the lookup is setting values which you can do with `setValue(param, value).` Again the function will check if the LMS is connected before updating the value in the LMS. The setValue function will handle saving for you automatically so you do not need to call `scorm.save()` after any values being set in your custom Javascript.

This function is a wrapper of the `pipwerks.SCORM.set(param, value`) function. The added benefits of using this one are the auto checking for the LMS connection, and the auto save feature. It also allows for custom debug messaging. If you are going to call the Pipwerks function independently you need to call `scorm.save()` in order to save your data ta the LMS.


The values available for lookup and setting are listed at [http://scorm.com/scorm-explained/technical-scorm/run-time/run-time-reference/](http://scorm.com/scorm-explained/technical-scorm/run-time/run-time-reference/) Make sure to click SCORM 1.2 to get correct names. You can also read much more in depth about the individual variables in the `SCORM_1_2_pdf` zip file included in the documentation folder. This file contains 4 official pdfs from which the bulk of the knowledge for this documentation came from. If there is ever any doubt about a behavior consult those first and take their word over anything written here.

## CONVENIENCE FUNCTIONS

Several convenience functions have been added to save the hassle of repeated lookups of exact SCORM compliant names. These are all wrappers of `getValue` and `setValue` primed with the correct SCORM variable names. These will most likely be the most frequently added functions as they become needed in the process of development.

An example is provided below, but anywhere they are included in the code there are comments detailing their intended purposes.

```
function completeSCO() {
  setValue('cmi.core.lesson_status', STATUS.COMPLETED);
}
```

This function will set the entire SCO to complete in the LMS. It can be called immediately before `closeSCO` or anywhere else it makes sense to mark the course complete. Note the verbose SCORM name which is avoided as well as the use of the `STATUS` constant.

## Bookmarks

SCORM allows for the storage of a lesson location bookmark which can be updated as a student progresses through a course. Upon returning to the course this bookmark can be retrieved and used to initialize the SCO to the point the student left off at. Two convenience functions have been added for ease of use of this feature.

```
function getBookmark() {
  return getValue('cmi.core.lesson_location');
}
```

```
function setBookmark(location) {
  setValue('cmi.core.lesson_location', location);
}
```

You can store the location as a number or a string, but it is important to note that the values retrieved from the LMS will be returned as strings. If you need to manipulate the returned value as a number or another type you will need to coerce it manually.

# 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.

```
class Objective {
  constructor(index, id="[New Objective]") {
    Object.defineProperty(this, 'index', {
      writable: false,
      configurable: false,
      value: index
    });
    this._id = id;
    this._status = STATUS.NOT_ATTEMPTED;
    this._score = new Score();
```

```
    setValue(`cmi.objectives.${index}.id`, id);
  }

  complete() {
    this._status = STATUS.COMPLETED
    setValue(`cmi.objectives.${this.index}.status`, this._status);
  }

  set id(newId) {
    this._id = newId;
    setValue(`cmi.objectives.${this.index}.id`, newId);
  }
  get id() {
    this._id = getValue(`cmi.objectives.${this.index}.id`);
    return this._id;
  }

  set status(newStatus) {
    this._status = newStatus;
    setValue(`cmi.objectives.${this.index}.status`, newStatus);
  }
  get status() {
    this._status = getValue(`cmi.objectives.${this.index}.status`);
    return this._status;
  }

  set score(rawScore) {
    this._score.raw = rawScore;
    setValue(`cmi.objectives.${this.index}.score.raw`, rawScore);
  }

  get score() {
    return this._score.raw;
  }

  save() {
    scorm.set(`cmi.objectives.${this.index}.id`, this._id);
    scorm.set(`cmi.objectives.${this.index}.score`, this._score.raw);
    scorm.set(`cmi.objectives.${this.index}.status`, this._status);
    scorm.save();
  }
}
```

## 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;
  }
}
```

### THE TRACKED OBJECTIVES CLASS

The TrackedObjectives class is a class to make managing multiple Objective objects even simpler. It has an internal array objectives and two methods on it, the addObjective() method and the InitializeList() method. The objectives array is meant to mirror the LMS cmi.objectives array and will hold all the individual Objectives as objects.

addObjectives will create and push a new Objective onto this list and at creation time register the Objective with the LMS.

initializeList will take in a list of ids as strings and add a new Objective to the objectives array for each id in the list. Objectives will then be registered with the LMS. Note this is a very basic function right now, and in order to avoid conflicts and duplication within the LMS will not complete if there are already objectives present on cmi.objectives. At some point this will be updated with more complex logic to avoid duplication of objectives.

```
class TrackedObjectives {
  constructor() {
    this.objectives = [];
  }

  addObjective(objectiveId) {
    let newObjective = new Objective(this.objectives.length, objectiveId);
    this.objectives.push(newObjective);
  }

  initializeList(listOfIds) {
    let currentObjectivesCount = parseInt(getValue('cmi.object._count'));
    if (currentObjectivesCount) {
      DEBUG.log('objectives already initialized');
      return;
    } else {
      listOfIds.forEach(obj => this.addObjective(obj));
    }
  }
}
```

## 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.*

```
class Interaction {
  constructor(index, config) {
    Object.defineProperty(this, 'index', {
      writable: false,
      configurable: false,
      value: index
    });
    this._id = config.id;
    this._type = config.type;
    this._objectives = config.objectives;
    this._startTime = "00:00:00.0"
    this._finishTime = "00:00:00.0";
    this._correct_responses = config.correct_responses;
    this._weighting = config.weighting;
    this._student_response = '';
    this._result = RESULT.NEUTRAL;
    this._latency = "00:00:00.0";
    this.initialize()
  }

  set id(newId) {
    this._id = newId;
    setValue(`cmi.interactions.${this.index}.id`, newId);
  }
  get id() {
    return this._id;
  }
```

```
set type(newType) {
  this._type = newType;
  setValue(`cmi.objectives.${this.index}.type`, newType);
}
get type() {
  return this._type;
}

set objectives(newObjectives) {
  this._objectives = newObjectives;
  for (i=0; i<this.newObjectives.length; i++) {
    scorm.set(`cmi.interactions.${this.index}.objectives.$[i}.id`,
              this._objectives[i].id);
  }
  scorm.save();
}
get objectives() {
  DEBUG.log(`there are currently ${getValue(`cmi.interactions.${this.index
           }.objectives._count`)} interaction objectives`);
  return this._objectives;
}

set finishTime(t) {
  this._finishTime = t;
  this._latency = formatTime(this.startTime - t);
  scorm.set(`cmi.interactions.${this.index}.time`, t);
  scorm.set(`cmi.interactions.${this.index}.latency`, this._latency);
  scorm.save();
}
get finishTime() {
  return this._finishTime;
}

set correct_responses(newResponses) {
  this.correct_responses = newResponses;
  for (i=0; i<this.correct_responses.length; i++) {
    scorm.set(`cmi.interactions.${this.index}.correct_responses.${i}.pattern`,
             this._correct_responses[i];
  }
  scorm.save();
}
get correct_responses() {
  DEBUG.log(`there are currently ${getValue(`cmi.interactions.${this.index
           }.correct_responses._count`)} correct response patterns`);
  return this._correct_responses;
}

set weighting(newWeight) {
  this._weighting = newWeight;
  setValue(`cmi.interactions.${this.index}.correct_responses`, newWeight);
}
get weighting() {
  return this._weighting;
}

set student_response(newResponse) {
  this._student_response = newResponse;
  setValue(`cmi.interactions.${this.index}.student_response`, newResponse);
}
get student_response() {
```

```
    return this._student_response;
  }

  set result(newResult) {
    this._result = newResult;
    setValue(`cmi.interactions.${this.index}.result`, newResult);
  }
  get result() {
    return this._result;
  }

  initialize() {
    scorm.set(`cmi.interactions.${this.index}.id`, this._id);
    scorm.set(`cmi.interactions.${this.index}.type`, this._type);
    for (i=0; i<this._objectives.length; i++) {
      scorm.set(`cmi.interactions.${this.index}.objectives.$[i}.id`,
this._objectives[i].id);
    }
    for (i=0; i<this.correct_responses.length; i++) {
      scorm.set(`cmi.interactions.${this.index}.correct_responses.${i}.pattern`,
                 this._correct_responses[i];
    }
    scorm.set(`cmi.interactions.${this.index}.weighting`, this._weighting);
    scorm.set(`cmi.interactions.${this.index}.result`, this._result);
    scorm.save();
  }

  formatCurrentTime() {
    let date = new Date();
    return date.toTimeString().slice(0,8);
  }

  formatTime(t) {
    let date = new Date(t);
    let hours = String('0000' + date.getUTCHours()).slice(-4);
    let minutes = date.getUTCMinutes();
    let seconds = date.getUTCSeconds();
    let milliseconds = String(date.getUTCMilliseconds()).substring(0,2);
    let formattedTime = `${hours}:${minutes}:${seconds}.${milliseconds}`;
    return formattedTime;
  }

  begin() {
    this._startTime = formatCurrentTime();
  }

  complete() {
    this._finishTime = formatCurrentTime();
    this._latency = formatTime(this.startTime - this.finishTime);

    scorm.set(`cmi.interactions.${this.index}.time`, this._result);
    scorm.set(`cmi.interactions.${this.index}.latency`, this._latency);
    scorm.save();
  }

  save() {
    scorm.set(`cmi.interactions.${this.index}.id`, this._id);
    scorm.set(`cmi.interactions.${this.index}.type`, this._type);
    for (i=0; i<this._objectives.length; i++) {
      scorm.set(`cmi.interactions.${this.index}.objectives.$[i}.id`,
                this._objectives[i].id);
```

```
    }
    scorm.set(`cmi.interactions.${this.index}.time`, this._finishTime);
    for (i=0; i<this.correct_responses.length; i++) {
      scorm.set(`cmi.interactions.${this.index}.correct_responses.${i}.pattern`,
                this._correct_responses[i];
    }
    scorm.set(`cmi.interactions.${this.index}.weighting`, this._weighting);
    scorm.set(`cmi.interactions.${this.index}.student_response`,
              this._student_response);
    scorm.set(`cmi.interactions.${this.index}.result`, this._result);
    scorm.set(`cmi.interactions.${this.index}.latency`, this._latency);
    scorm.save();
  }
}
```

## THE INTERACTIONCONFIG CLASS

For convenience an `InteractionConfig` class has been created which contains all relevant config options for an Interaction object. You can create a new one with new InteractionConfig`()` which will return a config object with empty strings and arrays for all relevant fields. You can then pass this into the `Interaction` object constructor to receive a fully functional `Interaction` object.

```
class InteractionConfig {
  constructor() {
    this.id = "";
    this.type = "";
    this.objectives = [];
    this.time = "";
    this.correct_responses = [];
    this.weighting = 1;
    this.student_response = "";
    this.relust = "";
    this.latency = "";
  }
}
```

## THE TRACKED INTERACTIONS CLASS

This is similar to the `TrackedObjectives` class. It has one internal array `interactions` that stores the tracked interactions, and one method `addInteraction()` which will add an `interaction` at the next appropriate spot in the `cmi.interactions array`.

There currently is no method for adding multiple interactions at once, but one will be added in the near future.

```
class TrackedInteractions {
  constructor() {
    this.interactions = [];
  }

  addInteraction(config) {
    index = getValue('cmi.interactions._count');
    let newInteraction = new Interaction(index, config);
    this.interactions.push(newInteraction);
  }
}
```

# 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>
```

## RESOURCES

The `resources` tag is used to contain all the individual resources. There's nothing special about it, you just need it.

```
<resources>
...
</resources>
```

## RESOURCE

A `resource` is an individual piece of content. Every single file that is used in your SCORM package needs to be listed here. All html, js, css, images, videos, audio, and any other supporting files. If a file is under the root of the project it needs to be listed in the manifest as a resource.

At its most basic level a `resource` needs an `identifier` which will be used in the organization to link the structure of the SCORM package to the actual content, a `type` which should always be `webcontent` and an `adlcp:scormtype` which can either be `asset` or `sco`. The difference between `asset` and `sco` can get somewhat blurry, but is essentially that assets don't communicate with the LMS while SCOs do.

Inside the `resource` tag a `file` tag lists the actual file location using the `href` attribute. This should be relative to the root of the SCORM package.

```
<resource identifier="lia_resource" type="webcontent" adlcp:scormtype="asset">
    <file href="level_1/level_1_a.html"/>
</resource>
```

It isn't always clear which resources need to be assets and which need to be SCOs, but always remember if you are doing LMS calls it needs to be a SCO. Resources comprised entirely of multimedia will be assets, html pages with a quiz on them will generally be SCOs.

When we get to complicated nesting inside of iframes it gets a little murkier. When using iframes, only the outermost frame needs to be able to communicate with the LMS. The inner pages that get loaded in the iframe can be assets and communicate with `ml_scorm` on the outer page which in turn will communicate with the LMS. I'm still not entirely sure if you can have a SCO loaded inside an iframe inside of another SCO.

```
<resource identifier="lia_resource" type="webcontent" adlcp:scormtype="sco">
    <file href="level_1/level_1_a.html"/>
</resource>
```

Note that you can include multiple `files` per `resource`. If your content requires multiple files such as css, js, and multimedia assets you can list them all under the `resource` that uses them.

```
<resource identifier="lia_resource" type="webcontent" adlcp:scormtype="asset">
    <file href="level_1/level_1_a.html"/>
    <file href="level_1/level_1_a_header.jpg "/>
</resource>
```

In fact at the most basic implementation of the `imsmanifest.xml` document you only need one `resource` with `adlcp:scormtype` of `sco` (so that it can communicate progress and completion with the LMS) with every `file` in the course listed under that `resource`.

This will vastly limit the granularity with which you will be able to to interface with the LMS at, but as a quick and dirty implementation it works, and if the client only needs to track completion of the course as a whole this is a valid implementation.

```
<resource identifier="lia_resource" type="webcontent" adlcp:scormtype="sco">
    <file href="level_1/level_1_a.html"/>
    <file href="level_1/level_1_b.html"/>
    <file href="level_1/level_1_c.html"/>
    <file href="level_1/level_1_style.css"/>
    <file href="level_1/level_1_actions.js"/>
    <file href="level_1/level_1_header.jpg "/>
    <file href="level_1/level_2_a.html"/>
    <file href="level_1/level_2_b.html"/>
    <file href="level_1/level_2_c.html"/>
    <file href="level_1/level_2_style.css"/>
    <file href="level_1/level_2_actions.js"/>
    <file href="level_1/level_2_header.jpg "/>
    <file href="level_1/level_3_a.html"/>
    <file href="level_1/level_3_b.html"/>
    <file href="level_1/level_3_c.html"/>
    <file href="level_1/level_3_style.css"/>
    <file href="level_1/level_3_actions.js"/>
    <file href="level_1/level_3_header.jpg "/>
</resource>
```

An additional tag you can use under `resource` is the `dependency` tag. The `dependency` tag takes an `identifierref` which will link to another `resource`. This can allow you to separate out files that are reused across multiple resources or to simplify your `resource` tags if you have too many files to cleanly understand what is happening in each `resource`.

```
<resource identifier="l1a_resource" type="webcontent" adlcp:scormtype="sco"
href="index.html">
    <file href="index.html"/>
    <dependency identifierref="common_files" />
</resource>
```

The actual `dependency` is just another `resource` with a list of `files`. Make sure the `identifier` is the same as the `identifierref`. It can contain multiple files and multiple resources can link to it.

```
<resource identifier="common_files" type="webcontent" adlcp:scormtype="asset">
    <file href="SCORM_API_wrapper.js"/>
    <file href="ml_scorm.js"/>
    <file href="default_style.css"/>
</resource>
```