XBlock API Guide

Release

Nov 21, 2017

Contents

This document provides reference information on the XBlock API. You use this API to build XBlocks.

EdX also provides the EdX XBlock Tutorial, which describes XBlock concepts in depth and guides developers through the process of creating an XBlock.

CHAPTER 1

Change history for XBlock

These are notable changes in XBlock.

1.1 1.0 - Python 3

- Introduce Python 3 compatibility to the xblock code base. This does not enable Python 2 codebases (like edx-platform) to load xblocks written in Python 3, but it lays the groundwork for future migrations.

1.2 0.5 - ???

No notes provided.

1.3 0.4

- Separate Fragment class out into new web-fragments package
- Make Scope enums (UserScope.* and BlockScope.*) into Sentinels, rather than just ints, so that they can have more meaningful string representations.
- Rename *export_xml* to *add_xml_to_node*, to more accurately capture the semantics.
- Allowed *Runtime* implementations to customize loading from **block_types** to *XBlock* classes.

1.4 0.3 - 2014-01-09

- Added services available through *Runtime.service*, once XBlocks have announced their desires with *@XBlock.needs* and *@XBlock.wants*.

- The "i18n" service provides a *gettext.Translations* object for retrieving localized strings.
- Make *context* an optional parameter for all views.
- Add shortcut method to make rendering an XBlock's view with its own runtime easier.
- Change the user field of scopes to be three valued, rather than two. *False* becomes *UserScope.NONE*, *True* becomes *UserScope.ONE*, and *UserScope.ALL* is new, and represents data that is computed based on input from many users.
- Rename *ModelData* to *FieldData*.
- Rename *ModelType* to *Field*.
- Split xblock.core into a number of smaller modules:
  - xblock.core: Defines XBlock.
  - xblock.fields: Defines ModelType and subclasses, ModelData, and metaclasses for classes with fields.
  - xblock.namespaces: Code for XBlock Namespaces only.
  - xblock.exceptions: exceptions used by all parts of the XBlock project.
- Changed the interface for *Runtime* and *ModelData* so that they function as single objects that manage large numbers of *XBlocks*. Any method that operates on a block now takes that block as the first argument. Blocks, in turn, are responsible for storing the key values used by their field scopes.
- Changed the interface for *model_data* objects passed to *XBlocks* from dict-like to the being cache-like (as was already used by *KeyValueStore*). This removes the need to support methods like iteration and length, which makes it easier to write new *ModelDatas*. Also added an actual *ModelData* base class to serve as the expected interface.

CHAPTER 2

Introduction to XBlocks

As a developer, you build XBlocks that course teams use to create independent course components that work seamlessly with other components in an online course.

For example, you can build XBlocks to represent individual problems or pieces of text or HTML content. Furthermore, like Legos, XBlocks are composable; you can build XBlocks to represent larger structures such as lessons, sections, and entire courses.

A primary advantage to XBlocks is that they are sharable. The code you write can be deployed in any instance of the edX Platform or other XBlock runtime application, then used by any course team using that system.

By combining XBlocks from a wide variety of sources, from text and video, to multiple choice and numerical questions, to sophisticated collaborative and interactive learning laboratories, course teams can create rich and engaging courseware.

2.1 XBlock Independence and Interoperability

You must design your XBlock to meet two goals.

- The XBlock must be independent of other XBlocks. Course teams must be able to use the XBlock without depending on other XBlocks.
- The XBlock must work together with other XBlocks. Course teams must be able to combine different XBlocks in flexible ways.

2.2 XBlocks Compared to Web Applications

XBlocks are like miniature web applications: they maintain state in a storage layer, render themselves through views, and process user actions through handlers.

XBlocks differ from web applications in that they render only a small piece of a complete web page.

Like HTML tags, XBlocks can represent components as small as a paragraph of text, a video, or a multiple choice input field, or as large as a section, a chapter, or an entire course.

CHAPTER 3

XBlock API

class xblock.core.**XBlock**(*runtime*, *field_data=None*, *scope_ids=<object object>*, **args*, ***kwargs*)

Base class for XBlocks.

Derive from this class to create a new kind of XBlock. There are no required methods, but you will probably need at least one view.

Don't provide the __init__ method when deriving from this class.

Construct a new XBlock.

This class should only be instantiated by runtimes.

Parameters

- **runtime** (*Runtime*) – Use it to access the environment. It is available in XBlock code as `self.runtime`.
- **field_data** (`FieldData`) – Interface used by the XBlock fields to access their data from wherever it is persisted. Deprecated.
- **scope_ids** (*ScopeIds*) – Identifiers needed to resolve scopes.

__delattr__

x.__delattr__('name') <==> del x.name

__format__()

default object formatter

__getattribute__

x.__getattribute__('name') <==> x.name

__hash__

__reduce__()

helper for pickle

__reduce_ex__()

helper for pickle

__setattr__
x.__setattr__('name', value) <==> x.name = value

__sizeof__() → int
size of object in memory, in bytes

__str__

add_children_to_node(*node*)
Add children to etree.Element *node*.

add_xml_to_node(*node*)
For exporting, set data on etree.Element *node*.

clear_child_cache()
Reset the cache of children stored on this XBlock.

force_save_fields(*field_names*)
Save all fields that are specified in *field_names*, even if they are not dirty.

get_child(*usage_id*)
Return the child identified by `usage_id`.

get_children(*usage_id_filter=None*)
Return instantiated XBlocks for each of this blocks `children`.

get_parent()
Return the parent block of this block, or None if there isn't one.

get_public_dir()
Gets the public directory for this XBlock.

get_resources_dir()
Gets the resource directory for this XBlock.

handle(*handler_name*, *request*, *suffix=u''*)
Handle *request* with this block's runtime.

handler(*func*)
A decorator to indicate a function is usable as a handler.

The wrapped function must return a *webob.Response* object.

has_cached_parent
Return whether this block has a cached parent block.

has_support(*view*, *functionality*)
Returns whether the given view has support for the given functionality.

An XBlock view declares support for a functionality with the @XBlock.supports decorator. The decorator stores information on the view.

Note: We implement this as an instance method to allow xBlocks to override it, if necessary.

Parameters

- **view** (*object*) – The view of the xBlock.
- **functionality** (*string*) – A functionality of the view. For example: "multi_device".

Returns True or False

index_dictionary()
return key/value fields to feed an index within in a Python dict object values may be numeric / string or dict default implementation is an empty dict

json_handler(*func*)
Wrap a handler to consume and produce JSON.

Rather than a Request object, the method will now be passed the JSON-decoded body of the request. The request should be a POST request in order to use this method. Any data returned by the function will be JSON-encoded and returned as the response.

The wrapped function can raise JsonHandlerError to return an error response with a non-200 status code.

This decorator will return a 405 HTTP status code if the method is not POST. This decorator will return a 400 status code if the body contains invalid JSON.

load_class(*identifier*, *default=None*, *select=None*)
Load a single class specified by identifier.

If *identifier* specifies more than a single class, and *select* is not None, then call *select* on the list of entry_points. Otherwise, choose the first one and log a warning.

If *default* is provided, return it if no entry_point matching *identifier* is found. Otherwise, will raise a PluginMissingError

If *select* is provided, it should be a callable of the form:

```
def select(identifier, all_entry_points):
    # ...
    return an_entry_point
```

The *all_entry_points* argument will be a list of all entry_points matching *identifier* that were found, and *select* should return one of those entry_points to be loaded. *select* should raise *PluginMissingError* if no plugin is found, or *AmbiguousPluginError* if too many plugins are found

load_classes(*fail_silently=True*)
Load all the classes for a plugin.

Produces a sequence containing the identifiers and their corresponding classes for all of the available instances of this plugin.

fail_silently causes the code to simply log warnings if a plugin cannot import. The goal is to be able to use part of libraries from an XBlock (and thus have it installed), even if the overall XBlock cannot be used (e.g. depends on Django in a non-Django application). There is disagreement about whether this is a good idea, or whether we should see failures early (e.g. on startup or first page load), and in what contexts. Hence, the flag.

classmethod load_tagged_classes(*tag*, *fail_silently=True*)
Produce a sequence of all XBlock classes tagged with *tag*.

fail_silently causes the code to simply log warnings if a plugin cannot import. The goal is to be able to use part of libraries from an XBlock (and thus have it installed), even if the overall XBlock cannot be used (e.g. depends on Django in a non-Django application). There is disagreement about whether this is a good idea, or whether we should see failures early (e.g. on startup or first page load), and in what contexts. Hence, the flag.

needs(**service_names*)
A class decorator to indicate that an XBlock class needs particular services.

open_local_resource(*uri*)
Open a local resource.

The container calls this method when it receives a request for a resource on a URL which was generated by Runtime.local_resource_url(). It will pass the URI from the original call to local_resource_url() back to this method. The XBlock must parse this URI and return an open file-like object for the resource.

For security reasons, the default implementation will return only a very restricted set of file types, which must be located in a folder that defaults to "public". The location used for public resources can be changed on a per-XBlock basis. XBlock authors who want to override this behavior will need to take care to ensure that the method only serves legitimate public resources. At the least, the URI should be matched against a whitelist regex to ensure that you do not serve an unauthorized resource.

parse_xml(*node*, *runtime*, *keys*, *id_generator*)
Use *node* to construct a new block.

Parameters

- **node** (*etree.Element*) – The xml node to parse into an xblock.
- **runtime** (*Runtime*) – The runtime to use while parsing.
- **keys** (*ScopeIds*) – The keys identifying where this block will store its data.
- **id_generator** (*IdGenerator*) – An object that will allow the runtime to generate correct definition and usage ids for children of this block.

register_temp_plugin(*class_*, *identifier=None*, *dist=u'xblock'*)
Decorate a function to run with a temporary plugin available.

Use it like this in tests:

```
@register_temp_plugin(MyXBlockClass):
def test_the_thing():
    # Here I can load MyXBlockClass by name.
```

render(*view*, *context=None*)
Render *view* with this block's runtime and the supplied *context*

save()
Save all dirty fields attached to this XBlock.

service_declaration(*service_name*)
Find and return a service declaration.

XBlocks declare their service requirements with *@XBlock.needs* and *@XBlock.wants* decorators. These store information on the class. This function finds those declarations for a block.

Parameters **service_name** (*string*) – the name of the service requested.

Returns One of "need", "want", or None.

supports(**functionalities*)
A view decorator to indicate that an xBlock view has support for the given functionalities.

Parameters **functionalities** – String identifiers for the functionalities of the view. For example: "multi_device".

static tag(*tags*)
Returns a function that adds the words in *tags* as class tags to this class.

ugettext(*text*)
Translates message/text and returns it in a unicode string. Using runtime to get i18n service.

validate()
Ask this xblock to validate itself. Subclasses are expected to override this method, as there is currently

only a no-op implementation. Any overriding method should call super to collect validation results from its superclasses, and then add any additional results as necessary.

wants(**service_names*)
A class decorator to indicate that an XBlock class wants particular services.

xml_element_name()
What XML element name should be used for this block?

xml_text_content()
What is the text content for this block's XML node?

CHAPTER 4

Fields API

Fields declare storage for XBlock data. They use abstract notions of **scopes** to associate each field with particular sets of blocks and users. The hosting runtime application decides what actual storage mechanism to use for each scope.

class xblock.fields.**BlockScope**
Enumeration of block scopes.

The block scope specifies how a field relates to blocks. A *BlockScope* and a *UserScope* are combined to make a *Scope* for a field.

USAGE: The data is related to a particular use of a block in a course.

DEFINITION: The data is related to the definition of the block. Although unusual, one block definition can be used in more than one place in a course.

TYPE: The data is related to all instances of this type of XBlock.

ALL: The data is common to all blocks. This can be useful for storing information that is purely about the student.

classmethod scopes()
Return a list of valid/understood class scopes.

class xblock.fields.**UserScope**
Enumeration of user scopes.

The user scope specifies how a field relates to users. A *BlockScope* and a *UserScope* are combined to make a *Scope* for a field.

NONE: Identifies data agnostic to the user of the *XBlock*. The data is related to no particular user. All users see the same data. For instance, the definition of a problem.

ONE: Identifies data particular to a single user of the *XBlock*. For instance, a student's answer to a problem.

ALL: Identifies data aggregated while the block is used by many users. The data is related to all the users. For instance, a count of how many students have answered a question, or a histogram of the answers submitted by all students.

classmethod scopes()
Return a list of valid/understood class scopes. Why do we need this? I believe it is not used anywhere.

class xblock.fields.**Scope**
Defines six types of scopes to be used: *content*, *settings*, *user_state*, *preferences*, *user_info*, and *user_state_summary*.

The *content* scope is used to save data for all users, for one particular block, across all runs of a course. An example might be an XBlock that wishes to tabulate user "upvotes", or HTML content ti display literally on the page (this example being the reason this scope is named *content*).

The *settings* scope is used to save data for all users, for one particular block, for one specific run of a course. This is like the *content* scope, but scoped to one run of a course. An example might be a due date for a problem.

The *user_state* scope is used to save data for one user, for one block, for one run of a course. An example might be how many points a user scored on one specific problem.

The *preferences* scope is used to save data for one user, for all instances of one specific TYPE of block, across the entire platform. An example might be that a user can set their preferred default speed for the video player. This default would apply to all instances of the video player, across the whole platform, but only for that student.

The *user_info* scope is used to save data for one user, across the entire platform. An example might be a user's time zone or language preference.

The *user_state_summary* scope is used to save data aggregated across many users of a single block. For example, a block might store a histogram of the points scored by all users attempting a problem.

Create a new Scope, with an optional name.

classmethod named_scopes()
Return all named Scopes.

classmethod scopes()
Return all possible Scopes.

class xblock.fields.**ScopeIds**
A simple wrapper to collect all of the ids needed to correctly identify an XBlock (or other classes deriving from ScopedStorageMixin) to a FieldData. These identifiers match up with BlockScope and UserScope attributes, so that, for instance, the *def_id* identifies scopes that use BlockScope.DEFINITION.

Create new instance of ScopeIds(user_id, block_type, def_id, usage_id)

class xblock.fields.**Field**(*help=None*, *default=fields.UNSET*, *scope=ScopeBase(user=UserScope.NONE, block=BlockScope.DEFINITION, name=u'content')*, *display_name=None*, *values=None*, *enforce_type=False*, *xml_node=False*, *force_export=False*, ***kwargs*)
A field class that can be used as a class attribute to define what data the class will want to refer to.

When the class is instantiated, it will be available as an instance attribute of the same name, by proxying through to the field-data service on the containing object.

Parameters

- **help** (*str*) – documentation for the field, suitable for presenting to a user (defaults to None).
- **default** – field's default value. Can be a static value or the special xblock.fields.UNIQUE_ID reference. When set to xblock.fields.UNIQUE_ID, the field defaults to a unique string that is deterministically calculated for the field in the given scope (defaults to None).
- **scope** – this field's scope (defaults to Scope.content).

- **display_name** – the display name for the field, suitable for presenting to a user (defaults to name of the field).
- **values** – a specification of the valid values for this field. This can be specified as either a static specification, or a function that returns the specification. For example specification formats, see the values property definition.
- **enforce_type** – whether the type of the field value should be enforced on set, using self.enforce_type, raising an exception if it's not possible to convert it. This provides a guarantee on the stored value type.
- **xml_node** – if set, the field will be serialized as a separate node instead of an xml attribute (default: False).
- **force_export** – if set, the field value will be exported to XML even if normal export conditions are not met (i.e. the field has no explicit value set)
- **kwargs** – optional runtime-specific options/metadata. Will be stored as runtime_options.

__delete__(*xblock*)
Deletes *xblock* from the underlying data store. Deletes are not cached; they are performed immediately.

__get__(*xblock*, *xblock_class*)
Gets the value of this xblock. Prioritizes the cached value over obtaining the value from the field-data service. Thus if a cached value exists, that is the value that will be returned.

__set__(*xblock*, *value*)
Sets the *xblock* to the given *value*. Setting a value does not update the underlying data store; the new value is kept in the cache and the xblock is marked as dirty until *save* is explicitly called.

Note, if there's already a cached value and it's equal to the value we're trying to cache, we won't do anything.

default
Returns the static value that this defaults to.

delete_from(*xblock*)
Delete the value for this field from the supplied xblock

display_name
Returns the display name for this class, suitable for use in a GUI.

If no display name has been set, returns the name of the class.

enforce_type(*value*)
Coerce the type of the value, if necessary

Called on field sets to ensure that the stored type is consistent if the field was initialized with enforce_type=True

This must not have side effects, since it will be executed to trigger a DeprecationWarning even if enforce_type is disabled

from_json(*value*)
Return value as a native full featured python type (the inverse of to_json)

Called during field reads to convert the stored value into a full featured python object

from_string(*serialized*)
Returns a native value from a YAML serialized string representation. Since YAML is a superset of JSON, this is the inverse of to_string.)

is_set_on(*xblock*)
Return whether this field has a non-default value on the supplied xblock

name
Returns the name of this field.

read_from(*xblock*)
Retrieve the value for this field from the specified xblock

read_json(*xblock*)
Retrieve the serialized value for this field from the specified xblock

to_json(*value*)
Return value in the form of nested lists and dictionaries (suitable for passing to json.dumps).

This is called during field writes to convert the native python type to the value stored in the database

to_string(*value*)
Return a JSON serialized string representation of the value.

values
Returns the valid values for this class. This is useful for representing possible values in a UI.

Example formats:

- A finite set of elements:

```
[1, 2, 3]
```

- A finite set of elements where the display names differ from the values:

```
[
 {"display_name": "Always", "value": "always"},
 {"display_name": "Past Due", "value": "past_due"},
]
```

- A range for floating point numbers with specific increments:

```
{"min": 0 , "max": 10, "step": .1}
```

If this field class does not define a set of valid values, this property will return None.

write_to(*xblock*, *value*)
Set the value for this field to value on the supplied xblock

class xblock.fields.**Boolean**(*help=None*, *default=fields.UNSET*, *scope=ScopeBase(user=UserScope.NONE, block=BlockScope.DEFINITION, name=u'content')*, *display_name=None*, ***kwargs*)
A field class for representing a boolean.

The value, as loaded or enforced, can be either a Python bool, a string, or any value that will then be converted to a bool in the from_json method.

Examples:

```
True -> True
'true' -> True
'TRUE' -> True
'any other string' -> False
[] -> False
['123'] -> True
None - > False
```

class xblock.fields.**Dict**(*help=None*, *default=fields.UNSET*, *scope=ScopeBase(user=UserScope.NONE, block=BlockScope.DEFINITION, name=u'content')*, *display_name=None*, *values=None*, *enforce_type=False*, *xml_node=False*, *force_export=False*, ***kwargs*)

A field class for representing a Python dict.

The value, as loaded or enforced, must be either be None or a dict.

to_string(*value*)

In python3, json.dumps() cannot sort keys of different types, so preconvert None to 'null'.

class xblock.fields.**Float**(*help=None*, *default=fields.UNSET*, *scope=ScopeBase(user=UserScope.NONE, block=BlockScope.DEFINITION, name=u'content')*, *display_name=None*, *values=None*, *enforce_type=False*, *xml_node=False*, *force_export=False*, ***kwargs*)

A field that contains a float.

The value, as loaded or enforced, can be None, '' (which will be treated as None), a Python float, or a value that will parse as an float, ie., something for which float(value) does not throw an error.

class xblock.fields.**Integer**(*help=None*, *default=fields.UNSET*, *scope=ScopeBase(user=UserScope.NONE, block=BlockScope.DEFINITION, name=u'content')*, *display_name=None*, *values=None*, *enforce_type=False*, *xml_node=False*, *force_export=False*, ***kwargs*)

A field that contains an integer.

The value, as loaded or enforced, can be None, '' (which will be treated as None), a Python integer, or a value that will parse as an integer, ie., something for which int(value) does not throw an error.

Note that a floating point value will convert to an integer, but a string containing a floating point number ('3.48') will throw an error.

class xblock.fields.**List**(*help=None*, *default=fields.UNSET*, *scope=ScopeBase(user=UserScope.NONE, block=BlockScope.DEFINITION, name=u'content')*, *display_name=None*, *values=None*, *enforce_type=False*, *xml_node=False*, *force_export=False*, ***kwargs*)

A field class for representing a list.

The value, as loaded or enforced, can either be None or a list.

class xblock.fields.**Set**(**args*, ***kwargs*)

A field class for representing a set.

The stored value can either be None or a set.

Set class constructor.

Redefined in order to convert default values to sets.

class xblock.fields.**String**(*help=None*, *default=fields.UNSET*, *scope=ScopeBase(user=UserScope.NONE, block=BlockScope.DEFINITION, name=u'content')*, *display_name=None*, *values=None*, *enforce_type=False*, *xml_node=False*, *force_export=False*, ***kwargs*)

A field class for representing a string.

The value, as loaded or enforced, can either be None or a basestring instance.

from_string(*value*)

String gets serialized and deserialized without quote marks.

none_to_xml

Returns True to use a XML node for the field and represent None as an attribute.

to_string(*value*)
String gets serialized and deserialized without quote marks.

class xblock.fields.**XMLString**(*help=None*, *default=fields.UNSET*, *scope=ScopeBase(user=UserScope.NONE, block=BlockScope.DEFINITION, name=u'content')*, *display_name=None*, *values=None*, *enforce_type=False*, *xml_node=False*, *force_export=False*, ***kwargs*)
A field class for representing an XML string.

The value, as loaded or enforced, can either be None or a basestring instance. If it is a basestring instance, it must be valid XML. If it is not valid XML, an lxml.etree.XMLSyntaxError will be raised.

to_json(*value*)
Serialize the data, ensuring that it is valid XML (or None).

Raises an lxml.etree.XMLSyntaxError if it is a basestring but not valid XML.

class xblock.fields.**XBlockMixin**(**args*, ***kwargs*)
A wrapper around xblock.core.XBlockMixin that provides backwards compatibility for the old location.

Deprecated.

class xblock.reference.plugins.**Filesystem**(*help=None*, *default=fields.UNSET*, *scope=ScopeBase(user=UserScope.NONE, block=BlockScope.DEFINITION, name=u'content')*, *display_name=None*, *values=None*, *enforce_type=False*, *xml_node=False*, *force_export=False*, ***kwargs*)
An enhanced pyfilesystem.

This returns a file system provided by the runtime. The file system has two additional methods over a normal pyfilesytem:

- *get_url* allows it to return a URL for a file
- *expire* allows it to create files which may be garbage collected after a preset period. *edx-platform* and *xblock-sdk* do not currently garbage collect them, however.

More information can be found at: http://docs.pyfilesystem.org/en/latest/ and https://github.com/pmitros/django-pyfs

The major use cases for this are storage of large binary objects, pregenerating per-student data (e.g. *pylab* plots), and storing data which should be downloadable (for example, serving <img src=...> will typically be faster through this than serving that up through XBlocks views.

__delete__(*xblock*)
We don't support this until we figure out what this means. Files should be deleted through normal pyfilesystem operations.

__get__(*xblock*, *xblock_class*)
Returns a *pyfilesystem* object which may be interacted with.

__set__(*xblock*, *value*)
We interact with a file system by *open*/*close*/*read*/*write*, not *set* and *get*.

We don't support this until we figure out what this means. In the future, this might be used to e.g. store some kind of metadata about the file system in the KVS (perhaps prefix and location or similar?)

CHAPTER 5

Runtime API

Machinery to make the common case easy when building new runtimes

xblock.runtime.**DbModel**
alias of *KvsFieldData*

class xblock.runtime.**DictKeyValueStore**(*storage=None*)
A *KeyValueStore* that stores everything into a Python dictionary.

class xblock.runtime.**IdGenerator**
An abstract object that creates usage and definition ids

create_aside(*definition_id*, *usage_id*, *aside_type*)
Make a new aside definition and usage ids, indicating an XBlockAside of type *aside_type* commenting on an *XBlock* usage *usage_id*

Returns (aside_definition_id, aside_usage_id)

create_definition(*block_type*, *slug=None*)
Make a definition, storing its block type.

If *slug* is provided, it is a suggestion that the definition id incorporate the slug somehow.

Returns the newly-created definition id.

create_usage(*def_id*)
Make a usage, storing its definition id.

Returns the newly-created usage id.

class xblock.runtime.**IdReader**
An abstract object that stores usages and definitions.

get_aside_type_from_definition(*aside_id*)
Retrieve the XBlockAside *aside_type* associated with this aside definition id.

Parameters **aside_id** – The definition id of the XBlockAside.

Returns The *aside_type* of the aside.

get_aside_type_from_usage(*aside_id*)
Retrieve the XBlockAside *aside_type* associated with this aside usage id.

Parameters **aside_id** – The usage id of the XBlockAside.

Returns The *aside_type* of the aside.

get_block_type(*def_id*)
Retrieve the block_type of a particular definition

Parameters **def_id** – The id of the definition to query

Returns The *block_type* of the definition

get_definition_id(*usage_id*)
Retrieve the definition that a usage is derived from.

Parameters **usage_id** – The id of the usage to query

Returns The *definition_id* the usage is derived from

get_definition_id_from_aside(*aside_id*)
Retrieve the XBlock *definition_id* associated with this aside definition id.

Parameters **aside_id** – The definition id of the XBlockAside.

Returns The *definition_id* of the xblock the aside is commenting on.

get_usage_id_from_aside(*aside_id*)
Retrieve the XBlock *usage_id* associated with this aside usage id.

Parameters **aside_id** – The usage id of the XBlockAside.

Returns The *usage_id* of the usage the aside is commenting on.

class xblock.runtime.**KeyValueStore**
The abstract interface for Key Value Stores.

class Key
Keys are structured to retain information about the scope of the data. Stores can use this information however they like to store and retrieve data.

default(*key*)
Returns the context relevant default of the given *key* or raise KeyError which will result in the field's global default.

delete(*key*)
Deletes *key* from storage.

get(*key*)
Reads the value of the given *key* from storage.

has(*key*)
Returns whether or not *key* is present in storage.

set(*key*, *value*)
Sets *key* equal to *value* in storage.

set_many(*update_dict*)
For each (*key, value*) in *update_dict*, set *key* to *value* in storage.

The default implementation brute force updates field by field through set which may be inefficient for any runtimes doing persistence operations on each set. Such implementations will want to override this method.

Update_dict field_name, field_value pairs for all cached changes

class xblock.runtime.**KvsFieldData**(*kvs*, ***kwargs*)

An interface mapping value access that uses field names to one that uses the correct scoped keys for the underlying KeyValueStore

default(*block*, *name*)

Ask the kvs for the default (default implementation which other classes may override).

Parameters

- **block** (*XBlock*) – block containing field to default
- **name** – name of the field to default

delete(*block*, *name*)

Reset the value of the field named *name* to the default

get(*block*, *name*)

Retrieve the value for the field named *name*.

If a value is provided for *default*, then it will be returned if no value is set

has(*block*, *name*)

Return whether or not the field named *name* has a non-default value

set(*block*, *name*, *value*)

Set the value of the field named *name*

set_many(*block*, *update_dict*)

Update the underlying model with the correct values.

class xblock.runtime.**MemoryIdManager**

A simple dict-based implementation of IdReader and IdGenerator.

ASIDE_DEFINITION_ID

alias of MemoryAsideDefinitionId

ASIDE_USAGE_ID

alias of MemoryAsideUsageId

clear()

Remove all entries.

create_aside(*definition_id*, *usage_id*, *aside_type*)

Create the aside.

create_definition(*block_type*, *slug=None*)

Make a definition, storing its block type.

create_usage(*def_id*)

Make a usage, storing its definition id.

get_aside_type_from_definition(*aside_id*)

Get an aside's type from its definition id.

get_aside_type_from_usage(*aside_id*)

Get an aside's type from its usage id.

get_block_type(*def_id*)

Get a block_type by its definition id.

get_definition_id(*usage_id*)

Get a definition_id by its usage id.

get_definition_id_from_aside(*aside_id*)

Extract the original xblock's definition_id from an aside's definition_id.

get_usage_id_from_aside(*aside_id*)
Extract the usage_id from the aside's usage_id.

class xblock.runtime.**Mixologist**(*mixins*)
Provides a facility to dynamically generate classes with additional mixins.

Parameters **mixins** (*iterable* of *class*) – Classes to mixin

mix(*cls*)
Returns a subclass of *cls* mixed with *self.mixins*.

Parameters **cls** (*class*) – The base class to mix into

class xblock.runtime.**NullI18nService**
A simple implementation of the runtime "i18n" service.

strftime(*dtime*, *format*)
Locale-aware strftime, with format short-cuts.

ugettext
Dispatch to the appropriate gettext method to handle text objects.

Note that under python 3, this uses *gettext()*, while under python 2, it uses *ugettext()*. This should not be used with bytestrings.

ungettext
Dispatch to the appropriate ngettext method to handle text objects.

Note that under python 3, this uses *ngettext()*, while under python 2, it uses *ungettext()*. This should not be used with bytestrings.

class xblock.runtime.**ObjectAggregator**(**objects*)
Provides a single object interface that combines many smaller objects.

Attribute access on the aggregate object acts on the first sub-object that has that attribute.

class xblock.runtime.**RegexLexer**(**toks*)
Split text into lexical tokens based on regexes.

lex(*text*)
Iterator that tokenizes *text* and yields up tokens as they are found

class xblock.runtime.**Runtime**(*id_reader*, *field_data=None*, *mixins=()*, *services=None*, *default_class=None*, *select=None*, *id_generator=None*)
Access to the runtime environment for XBlocks.

Parameters

- **id_reader** (IdReader) – An object that allows the *Runtime* to map between *usage_ids*, *definition_ids*, and *block_types*.
- **id_generator** (IdGenerator) – The *IdGenerator* to use for creating ids when importing XML or loading XBlockAsides.
- **field_data** (*FieldData*) – The FieldData to use by default when constructing an *XBlock* from this *Runtime*.
- **mixins** (*tuple*) – Classes that should be mixed in with every *XBlock* created by this *Runtime*.
- **services** (*dictionary*) – Services to make available through the *service()* method. There's no point passing anything here if you are overriding *service()* in your sub-class.

- **default_class** (*class*) – The default class to use if a class can't be found for a particular *block_type* when loading an *XBlock*.
- **select** – A function to select from one or more *XBlock* subtypes found when calling *XBlock.load_class()* to resolve a *block_type*. This is the same *select* as used by `Plugin.load_class()`.

add_block_as_child_node (*block*, *node*)
Export *block* as a child node of *node*.

add_node_as_child (*block*, *node*, *id_generator=None*)
Called by XBlock.parse_xml to treat a child node as a child block.

applicable_aside_types (*block*)
Return the set of applicable aside types for this runtime and block. This method allows the runtime to filter the set of asides it wants to support or to provide even block-level or block_type level filtering. We may extend this in the future to also take the user or user roles.

construct_xblock (*block_type*, *scope_ids*, *field_data=None*, **args*, ***kwargs*)
Construct a new xblock of the type identified by block_type, passing *args and **kwargs into *__init__*.

construct_xblock_from_class (*cls*, *scope_ids*, *field_data=None*, **args*, ***kwargs*)
Construct a new xblock of type cls, mixing in the mixins defined for this application.

create_aside (*block_type*, *keys*)
The aside version of construct_xblock: take a type and key. Return an instance

export_to_xml (*block*, *xmlfile*)
Export the block to XML, writing the XML to *xmlfile*.

field_data
Access the FieldData passed in the constructor.

Deprecated in favor of a 'field-data' service.

get_aside (*aside_usage_id*)
Create an XBlockAside in this runtime.

The *aside_usage_id* is used to find the Aside class and data.

get_aside_of_type (*block*, *aside_type*)
Return the aside of the given aside_type which might be decorating this *block*.

Parameters

- **block** (*XBlock*) – The block to retrieve asides for.
- **aside_type** (*str*) – the type of the aside

get_asides (*block*)
Return all of the asides which might be decorating this *block*.

Parameters **block** (*XBlock*) – The block to render retrieve asides for.

get_block (*usage_id*, *for_parent=None*)
Create an XBlock instance in this runtime.

The *usage_id* is used to find the XBlock class and data.

handle (*block*, *handler_name*, *request*, *suffix=u''*)
Handles any calls to the specified *handler_name*.

Provides a fallback handler if the specified handler isn't found.

Parameters

- **handler_name** – The name of the handler to call
- **request** (*webob.Request*) – The request to handle
- **suffix** – The remainder of the url, after the handler url prefix, if available

handler_url(*block*, *handler_name*, *suffix=u''*, *query=u''*, *thirdparty=False*)
Get the actual URL to invoke a handler.

handler_name is the name of your handler function. Any additional portion of the url will be passed as the *suffix* argument to the handler.

The return value is a complete absolute URL that will route through the runtime to your handler.

Parameters

- **block** – The block to generate the url for
- **handler_name** – The handler on that block that the url should resolve to
- **suffix** – Any path suffix that should be added to the handler url
- **query** – Any query string that should be added to the handler url (which should not include an initial ? or &)
- **thirdparty** – If true, create a URL that can be used without the user being logged in. This is useful for URLs to be used by third-party services.

layout_asides(*block*, *context*, *frag*, *view_name*, *aside_frag_fns*)
Execute and layout the aside_frags wrt the block's frag. Runtimes should feel free to override this method to control execution, place, and style the asides appropriately for their application

This default method appends the aside_frags after frag. If you override this, you must call wrap_aside around each aside as per this function.

Parameters

- **block** (XBlock) – the block being rendered
- **frag** (*html*) – The result from rendering the block

load_aside_type(*aside_type*)
Returns a subclass of XBlockAside that corresponds to the specified *aside_type*.

load_block_type(*block_type*)
Returns a subclass of *XBlock* that corresponds to the specified *block_type*.

local_resource_url(*block*, *uri*)
Get the URL to load a static resource from an XBlock.

block is the XBlock that owns the resource.

***uri* is a relative URI to the resource. The XBlock class's** get_local_resource(uri) method should be able to open the resource identified by this uri.

Typically, this function uses *open_local_resource* defined on the XBlock class, which by default will only allow resources from the "public/" directory of the kit. Resources must be placed in "public/" to be successfully served with this URL.

The return value is a complete absolute URL which will locate the resource on your runtime.

parse_xml_file(*fileobj*, *id_generator=None*)
Parse an open XML file, returning a usage id.

parse_xml_string(*xml*, *id_generator=None*)
Parse a string of XML, returning a usage id.

publish (*block*, *event_type*, *event_data*)
Publish an event.

For example, to participate in the course grade, an XBlock should set has_score to True, and should publish a grade event whenever the grade changes.

In this case the *event_type* would be *grade*, and the *event_data* would be a dictionary of the following form:

{ 'value': <number>, 'max_value': <number>,

}

The grade event represents a grade of value/max_value for the current user.

block is the XBlock from which the event originates.

query (*block*)
Query for data in the tree, starting from *block*.

Returns a Query object with methods for navigating the tree and retrieving information.

querypath (*block*, *path*)
An XPath-like interface to *query*.

render (*block*, *view_name*, *context=None*)
Render a block by invoking its view.

Finds the view named *view_name* on *block*. The default view will be used if a specific view hasn't be registered. If there is no default view, an exception will be raised.

The view is invoked, passing it *context*. The value returned by the view is returned, with possible modifications by the runtime to integrate it into a larger whole.

render_asides (*block*, *view_name*, *frag*, *context*)
Collect all of the asides' add ons and format them into the frag. The frag already has the given block's rendering.

render_child (*child*, *view_name=None*, *context=None*)
A shortcut to render a child block.

Use this method to render your children from your own view function.

If *view_name* is not provided, it will default to the view name you're being rendered with.

Returns the same value as *render()*.

render_childrens (*block*, *view_name=None*, *context=None*)
Render a block's children, returning a list of results.

Each child of *block* will be rendered, just as *render_child()* does.

Returns a list of values, each as provided by *render()*.

resource_url (*resource*)
Get the URL for a static resource file.

resource is the application local path to the resource.

The return value is a complete absolute URL that will locate the resource on your runtime.

service (*block*, *service_name*)
Return a service, or None.

Services are objects implementing arbitrary other interfaces. They are requested by agreed-upon names, see [XXX TODO] for a list of possible services. The object returned depends on the service requested.

XBlocks must announce their intention to request services with the *XBlock.needs* or *XBlock.wants* decorators. Use *needs* if you assume that the service is available, or *wants* if your code is flexible and can accept a None from this method.

Runtimes can override this method if they have different techniques for finding and delivering services.

Parameters

- **block** (*an XBlock*) – this block's class will be examined for service decorators.
- **service_name** (*string*) – the name of the service requested.

Returns An object implementing the requested service, or None.

wrap_aside (*block*, *aside*, *view*, *frag*, *context*)

Creates a div which identifies the aside, points to the original block, and writes out the json_init_args into a script tag.

The default implementation creates a frag to wraps frag w/ a div identifying the xblock. If you have javascript, you'll need to override this impl

wrap_xblock (*block*, *view*, *frag*, *context*)

Creates a div which identifies the xblock and writes out the json_init_args into a script tag.

If there's a *wrap_child* method, it calls that with a deprecation warning.

The default implementation creates a frag to wraps frag w/ a div identifying the xblock. If you have javascript, you'll need to override this impl

CHAPTER 6

Fragment API

Makes the Fragment class available through the old namespace location.

class xblock.fragment.**Fragment**(**args*, ***kwargs*)

A wrapper around web_fragments.fragment.Fragment that provides backwards compatibility for the old location.

Deprecated.

add_frag_resources(*fragment*)

Add all the resources from a single fragment to my resources.

This is used to aggregate resources from another fragment that should be considered part of the current fragment.

The content from the Fragment is ignored. The caller must collect together the content into this Fragment's content.

add_frags_resources(*fragments*)

Add all the resources from *fragments* to my resources.

This is used to aggregate resources from a sequence of fragments that should be considered part of the current fragment.

The content from the Fragments is ignored. The caller must collect together the content into this Fragment's content.

CHAPTER 7

Exceptions API

Module for all xblock exception classes

exception xblock.exceptions.**DisallowedFileError**
Raised by `open_local_resource()` if the requested file is not allowed.

exception xblock.exceptions.**FieldDataDeprecationWarning**
Warning for use of deprecated _field_data accessor

exception xblock.exceptions.**InvalidScopeError**(*invalid_scope*, *valid_scopes=None*)
Raised to indicated that operating on the supplied scope isn't allowed by a KeyValueStore

exception xblock.exceptions.**JsonHandlerError**(*status_code*, *message*)
Raised by a function decorated with XBlock.json_handler to indicate that an error response should be returned.

get_response(***kwargs*)
Returns a Response object containing this object's status code and a JSON object containing the key "error" with the value of this object's error message in the body. Keyword args are passed through to the Response.

exception xblock.exceptions.**KeyValueMultiSaveError**(*saved_field_names*)
Raised to indicated an error in saving multiple fields in a KeyValueStore

Create a new KeyValueMultiSaveError

saved_field_names - an iterable of field names (strings) that were successfully saved before the exception occured

exception xblock.exceptions.**NoSuchDefinition**
Raised by *IdReader.get_block_type()* if the definition doesn't exist.

exception xblock.exceptions.**NoSuchHandlerError**
Raised to indicate that the requested handler was not found.

exception xblock.exceptions.**NoSuchServiceError**
Raised to indicate that a requested service was not found.

exception xblock.exceptions.**NoSuchUsage**
Raised by *IdReader.get_definition_id()* if the usage doesn't exist.

exception xblock.exceptions.**NoSuchViewError**(*block*, *view_name*)
Raised to indicate that the view requested was not found.

Create a new NoSuchViewError

Parameters

- **block** – The XBlock without a view
- **view_name** – The name of the view that couldn't be found

exception xblock.exceptions.**XBlockNotFoundError**(*usage_id*)
Raised to indicate that an XBlock could not be found with the requested usage_id

exception xblock.exceptions.**XBlockSaveError**(*saved_fields*, *dirty_fields*, *message=None*)
Raised to indicate an error in saving an XBlock

Create a new XBlockSaveError

saved_fields - a set of fields that were successfully saved before the error occured *dirty_fields* - a set of fields that were left dirty after the save

Python Module Index

x

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N