XBlock API Guide

The Axim Collaborative

May 09, 2024

CONTENTS

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

This document also contains the Open edX XBlock Tutorial, which describes XBlock concepts in depth and guides developers through the process of creating an XBlock.

CHAPTER

ONE

CHANGE HISTORY FOR XBLOCK

1.1 Unreleased

1.2 4.0.1 - 2024-04-24

- unpin lxml constraint.

1.3 4.0.0 - 2024-04-18

- xblock.fragment has returned as a pass-though component to web_fragments.fragment

1.4 3.0.0 - 2024-03-18

Various extraneous classes have been removed from the XBlock API, simplifying its implementation and making debugging of XBlock instances easier. We believe that most, if not all, XBlock API users will be unaffected by this change. Some improvements have also been made to the reference documentation.

Specific changes:

- **Removed:**
  - `xblock.XBlockMixin` (still available as `xblock.core.XBlockMixin`)
  - `xblock.core.SharedBlockBase` (replaced with `xblock.core.Blocklike`)
  - `xblock.internal.Nameable`
  - `xblock.internal.NamedAttributesMetaclass`
  - `xblock.django.request.HeadersDict`
  - `xblock.fields.XBlockMixin` (still available as `xblock.core.XBlockMixin`)
  - `xblock.mixins.RuntimeServicesMixin`
  - `xblock.mixins.ScopedStorageMixin`
  - `xblock.mixins.IndexInfoMixin`
  - `xblock.mixins.XmlSerializationMixin`
  - `xblock.mixins.HandlersMixin`
  - `xblock.mixins.ChildrenModelMetaclass`

  - `xblock.mixins.HierarchyMixin`
  - `xblock.mixins.ViewsMixin`
- **Added:**
  - `xblock.core.Blocklike`, the new common ancestor of `XBlock` and `XBlockAside`, and `XBlockMixin`, replacing `xblock.core.SharedBlockBase`.
  - New attributes on `xblock.core.XBlockAside`, each behaving the same as their `XBlock` counterpart:
    * `usage_key`
    * `context_key`
    * `index_dictionary`
  - Various new attributes on `xblock.core.XBlockMixin`, encompassing the functionality of these former classes:
    * `xblock.mixins.IndexInfoMixin`
    * `xblock.mixins.XmlSerializationMixin`
    * `xblock.mixins.HandlersMixin`
- **Docs**
  - Various docstrings have been improved, some of which are published in the docs.
  - XBlockAside will now be represented in the API docs, right below XBlock on the "XBlock API" page.
  - XBlockMixin has been removed from the docs. It was only ever documented under the "Fields API" page (which didn't make any sense), and it was barely even documented there. We considered adding it back to the "XBlock API" page, but as noted in the class's new docstring, we do not want to encourage any new use of XBlockMixin.

1.5 2.0.0 - 2024-02-26

- Removed deprecations
- xblock.fragment (removed completely)
- xblock.runtime.Runtime._aside_from_xml (just the id_generator argument)
- xblock.runtime.Runtime._usage_id_from_node (just the id_generator argument)
- xblock.runtime.Runtime.add_node_as_child (just the id_generator argument)
- xblock.runtime.Runtime.parse_xml_string (just the id_generator argument)
- xblock.runtime.Runtime.parse_xml_file (just the id_generator argument)

1.6 1.10.0 - 2024-01-12

- Add two new properties to `XBlock` objects: `.usage_key` and `.context_key`. These simply expose the values of `.scope_ids.usage_id` and `.scope_ids.usage_id.context_key`, providing a convenient replacement to the deprecated, edx-platform-specific block properties `.location` and `.course_id`, respectively.

1.7 1.9.1 - 2023-12-22

- Fix: add `get_javascript_i18n_catalog_url` missing `xblock` parameter to match the Open edX LMS XBlockI18nService.

1.8 1.9.0 - 2023-11-20

- Support for OEP-58 JavaScript translations:
  - Introduced abstract JavaScript translations support by adding the `i18n_js_namespace` property and `get_i18n_js_namespace` method to the `SharedBlockBase`. This allows XBlocks to define a JavaScript namespace so the XBlock i18n runtime service can manage and load JavaScript translations for XBlocks.
  - Added the stub `get_javascript_i18n_catalog_url` method to the `NullI18nService` class to be implemented by runtime services.
  - See the edx-platform atlas translations proposal

1.9 1.8.1 - 2023-10-07

- Python Requirements Update
- Update setup.py, adds required packages

1.10 1.8.0 - 2023-09-25

- Added xblock-utils repository code into this repository along with docs.
  - Docs moved into the docs/ directory.
  - See https://github.com/openedx/xblock-utils/issues/197 for more details.

1.11 1.7.0 - 2023-08-03

- Switch from `edx-sphinx-theme` to `sphinx-book-theme` since the former is deprecated. See https://github.com/openedx/edx-sphinx-theme/issues/184 for more details.
- Added support for Django 4.2

1.12 1.6.1 - 2022-01-28

- Fix Release Issue with PyPi release workflow

1.13 1.6.0 - 2022-01-25

- Dropped Django22, 30 and 31 support
- Added Django40 Support in CI

1.14 1.5.1 - 2021-08-26

- Deprecated the Runtime.user_id property in favor of the user service.

1.15 1.5.0 - 2021-07-27

- Added Django 3.0, 3.1 & 3.2 support

1.16 1.4.2 - 2021-05-24

- Upgraded all Python dependencies.

1.17 1.4.1 - 2021-03-20

- Added XBlockParseException exception.

1.18 1.3.1 - 2020-05-06

- Fixed import error of mock.

1.19 1.3.0 - 2020-05-04

- Drop support to python 2.7 and add support to python 3.8. typing package failing on py3.8 so add constraint.

1.20 1.2.8 - 2019-10-24

- Ensure the version file is closed after reading its content.

1.21 1.2.7 - 2019-10-15

- Changed how illegal XML characters are sanitized, to speed the operation. The old way was removing more characters than are required by the XML specification.

1.22 1.2.6 - 2019-09-24

- Add support for relative dates to DateTime fields.

1.23 1.2.5 - 2019-09-19

- Changes for Python 2/3 compatibility.

1.24 1.2.4 - 2019-08-27

- Added an API for notifying the Runtime when an XBlock's `save()` method is called.
- Added a mechanism for Runtime subclasses to more easily add extra CSS classes to the that wraps rendered XBlocks

1.25 1.2.3 - 2019-07-24

Allow Mixologist class to consume both class objects and string paths to classes as a part of initialization.

1.26 1.2.1 - 2018-09-05

Add a method to get completion mode for a block.

1.27 1.2.1 - 2018-06-25

Suppress a spurious warning when using lazily-translated text as the default value of a String field.

1.28 1.2.0 - Aside filtering

- Add capability for XBlockAsides to apply only to XBlocks that match certain conditions

1.29 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.

1.30 0.5 - ???

No notes provided.

1.31 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.

1.32 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

TWO

INTRODUCTION TO XBLOCKS

This section introduces XBlocks.

2.1 Overview

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 Open edX Platform or other XBlock runtime application, then used by any course team using that system.

In educational applications, XBlocks can be used to represent individual problems, web-formatted text and videos, interactive simulations and labs, or collaborative learning experiences. Furthermore, XBlocks are composable, allowing an XBlock developer to control the display of other XBlocks to compose lessons, sections, and entire courses.

2.2 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.

2.3 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 `<div>` 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

THREE

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 new type of XBlock.

Subclasses of XBlocks can:

- Name one or more **views**, i.e. methods which render the block to HTML.
- Access the **parents** of their instances.
- Access and manage the **children** of their instances.
- Request **services** from the runtime, for their instances to use.
- Define scoped **fields**, which instances will use to store content, settings, and data.
- Define how instances are serialized to and deserialized from **OLX** (Open Learning XML).
- Mark methods as **handlers** for AJAX requests.
- Be installed into a platform as an entry-point **plugin**.

Note: Don't override the `__init__` method when deriving from this class.

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.

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.

property context_key

A key identifying the learning context (course, library, etc.) that contains this XBlock-like usage.

Equivalent to *.scope_ids.usage_id.context_key*.

Returns: * *LearningContextKey*, if *.scope_ids.usage_id* is a *UsageKey* instance. * *None*, otherwise.

After https://github.com/openedx/XBlock/issues/708 is complete, we can assume that *.scope_ids.usage_id* is always a *UsageKey*, and that this method will always return a *LearningContextKey*.

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`.

classmethod get_i18n_js_namespace()

Gets the JavaScript translations namespace for this XBlock-like class.

Returns

The JavaScript namespace for this XBlock-like class. None: If this doesn't have JavaScript translations configured.

Return type

str

get_parent()

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

classmethod get_public_dir()

Gets the public directory for this XBlock-like class.

classmethod get_resources_dir()

Gets the resource directory for this XBlock-like class.

handle(*handler_name*, *request*, *suffix=''*)

Handle *request* with this block's runtime.

classmethod handler(*func*)

A decorator to indicate a function is usable as a handler.

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

property 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** (*str*) – A functionality of the view. For example: "multi_device".

Returns

True or False

index_dictionary()

Return a dict containing information that could be used to feed a search index.

Values may be numeric, string, or dict.

classmethod 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.

classmethod 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

classmethod 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.

classmethod needs(**service_names*)

A class decorator to indicate that an XBlock-like class needs particular services.

classmethod 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-like 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-like basis. XBlock-like 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.

classmethod parse_xml(*node*, *runtime*, *keys*)

Use *node* to construct a new block.

Parameters

- **node** (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.

classmethod register_temp_plugin(*class_*, *identifier=None*, *dist='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.

classmethod service_declaration(*service_name*)

Find and return a service declaration.

XBlock-like classes declare their service requirements with *@XBlock{Aside}.needs* and *@XBlock{Aside}.wants* decorators. These store information on the class. This function finds those declarations for a block.

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

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

classmethod 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.

property usage_key

A key identifying this particular usage of the XBlock-like, unique across all learning contexts in the system.

Equivalent to to *.scope_ids.usage_id*.

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.

classmethod wants(**service_names*)

A class decorator to indicate that a XBlock-like 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?

class xblock.core.**XBlockAside**(*scope_ids*, *field_data=None*, ***, *runtime*, ***kwargs*)

Base class for XBlock-like objects that are rendered alongside *XBlock* views.

Subclasses of XBlockAside can:

- Specify which XBlock views they are to be **injected** into.
- Request **services** from the runtime, for their instances to use.
- Define scoped **fields**, which instances will use to store content, settings, and data.
- Define how instances are serialized to and deserialized from **OLX** (Open Learning XML).
- Mark methods as **handlers** for AJAX requests.
- Be installed into a platform as an entry-point **plugin**.

Parameters

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

add_xml_to_node(*node*)

For exporting, set data on *node* from ourselves.

classmethod aside_for(*view_name*)

A decorator to indicate a function is the aside view for the given view_name.

Aside views should have a signature like:

```
@XBlockAside.aside_for('student_view')
def student_aside(self, block, context=None):
    ...
    return Fragment(...)
```

`aside_view_declaration`(*view_name*)

Find and return a function object if one is an aside_view for the given view_name

Aside methods declare their view provision via @XBlockAside.aside_for(view_name) This function finds those declarations for a block.

Parameters
view_name (*str*) – the name of the view requested.

Returns
either the function or None

`property context_key`

A key identifying the learning context (course, library, etc.) that contains this XBlock-like usage.

Equivalent to *.scope_ids.usage_id.context_key*.

Returns: * *LearningContextKey*, if *.scope_ids.usage_id* is a *UsageKey* instance. * *None*, otherwise.

After https://github.com/openedx/XBlock/issues/708 is complete, we can assume that *.scope_ids.usage_id* is always a *UsageKey*, and that this method will always return a *LearningContextKey*.

`force_save_fields`(*field_names*)

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

`classmethod get_i18n_js_namespace`()

Gets the JavaScript translations namespace for this XBlock-like class.

Returns
The JavaScript namespace for this XBlock-like class. None: If this doesn't have JavaScript translations configured.

Return type
str

`classmethod get_public_dir`()

Gets the public directory for this XBlock-like class.

`classmethod get_resources_dir`()

Gets the resource directory for this XBlock-like class.

`handle`(*handler_name*, *request*, *suffix=''*)

Handle *request* with this block's runtime.

`classmethod handler`(*func*)

A decorator to indicate a function is usable as a handler.

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

`index_dictionary`()

Return a dict containing information that could be used to feed a search index.

Values may be numeric, string, or dict.

`classmethod 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.

classmethod 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

classmethod 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 needs(**service_names*)

A class decorator to indicate that an XBlock-like class needs particular services.

needs_serialization()

Return True if the aside has any data to serialize to XML.

If all of the aside's data is empty or a default value, then the aside shouldn't be serialized as XML at all.

classmethod 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-like 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-like basis. XBlock-like 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.

classmethod parse_xml(*node*, *runtime*, *keys*)

Use *node* to construct a new block.

Parameters

- **node** (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.

classmethod register_temp_plugin(*class_*, *identifier=None*, *dist='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.
```

save()

Save all dirty fields attached to this XBlock.

classmethod service_declaration(*service_name*)

Find and return a service declaration.

XBlock-like classes declare their service requirements with *@XBlock{Aside}.needs* and *@XBlock{Aside}.wants* decorators. These store information on the class. This function finds those declarations for a block.

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

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

classmethod should_apply_to_block(*block*)

Return True if the aside should be applied to a given block. This can be overridden if some aside should only wrap blocks with certain properties.

property usage_key

A key identifying this particular usage of the XBlock-like, unique across all learning contexts in the system.

Equivalent to to *.scope_ids.usage_id*.

classmethod wants(**service_names*)

A class decorator to indicate that a XBlock-like 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

FOUR

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.**Boolean**(*help=None*, *default=fields.UNSET*, *scope=ScopeBase(user=UserScope.NONE, block=BlockScope.DEFINITION, name='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
```

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

class xblock.fields.**Dict**(*help=None*, *default=fields.UNSET*, *scope=ScopeBase(user=UserScope.NONE, block=BlockScope.DEFINITION, name='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.

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

to_string(*value*)

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

class xblock.fields.**Field**(*help=None*, *default=fields.UNSET*, *scope=ScopeBase(user=UserScope.NONE, block=BlockScope.DEFINITION, name='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.

property default

Returns the static value that this defaults to.

delete_from(*xblock*)

Delete the value for this field from the supplied xblock

property 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

property name

Returns the name of this field.

static needs_name(*field*)

Returns whether the given) is yet to be named.

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.

property 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.**Float**(*help=None*, *default=fields.UNSET*, *scope=ScopeBase(user=UserScope.NONE, block=BlockScope.DEFINITION, name='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.

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

class xblock.fields.**Integer**(*help=None*, *default=fields.UNSET*, *scope=ScopeBase(user=UserScope.NONE, block=BlockScope.DEFINITION, name='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.

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

class xblock.fields.**List**(*help=None*, *default=fields.UNSET*, *scope=ScopeBase(user=UserScope.NONE, block=BlockScope.DEFINITION, name='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.

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

class xblock.fields.**Scope**(*user*, *block*, *name=None*)

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**(*user_id*, *block_type*, *def_id*, *usage_id*)

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.**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.

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

class xblock.fields.**String**(*help=None*, *default=fields.UNSET*, *scope=ScopeBase(user=UserScope.NONE, block=BlockScope.DEFINITION, name='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.

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*)

String gets serialized and deserialized without quote marks.

property 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.**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.**XMLString**(*help=None*, *default=fields.UNSET*, *scope=ScopeBase(user=UserScope.NONE, block=BlockScope.DEFINITION, name='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.

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

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.field_data.**FieldData**

An interface allowing access to an XBlock's field values indexed by field names.

default(*block*, *name*)

Get the default value for this field which may depend on context or may just be the field's global default. The default behavior is to raise KeyError which will cause the caller to return the field's global default.

Parameters

- **block** (*XBlock*) – the block containing the field being defaulted
- **name** (*str*) – the field's name

abstract delete(*block*, *name*)

Reset the value of the field named *name* to the default for XBlock *block*.

Parameters

- **block** (*XBlock*) – block to modify
- **name** (*str*) – field name to delete

abstract get(*block*, *name*)

Retrieve the value for the field named *name* for the XBlock *block*.

If no value is set, raise a *KeyError*.

The value returned may be mutated without modifying the backing store.

Parameters

- **block** (*XBlock*) – block to inspect
- **name** (*str*) – field name to look up

has(*block*, *name*)

Return whether or not the field named *name* has a non-default value for the XBlock *block*.

Parameters

- **block** (*XBlock*) – block to check
- **name** (*str*) – field name

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

Set the value of the field named *name* for XBlock *block*.

value may be mutated after this call without affecting the backing store.

Parameters

- **block** (*XBlock*) – block to modify
- **name** (*str*) – field name to set
- **value** – value to set

set_many(*block*, *update_dict*)

Update many fields on an XBlock simultaneously.

Parameters

- **block** (*XBlock*) – the block to update
- **update_dict** (*dict*) – A map of field names to their new values

CHAPTER

FIVE

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.

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.**IdGenerator**

An abstract object that creates usage and definition ids

abstract 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)

abstract 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.

abstract 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.

abstract 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.

abstract 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.

abstract 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

abstract 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

abstract 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.

abstract 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(*scope*, *user_id*, *block_scope_id*, *field_name*, *block_family='xblock.v1'*)

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.

Create new instance of Key(scope, user_id, block_scope_id, field_name, block_family)

default(*key*)

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

abstract delete(*key*)

Deletes *key* from storage.

abstract get(*key*)

Reads the value of the given *key* from storage.

abstract has(*key*)

Returns whether or not *key* is present in storage.

abstract 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 or names of 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.

get_javascript_i18n_catalog_url(*block*)

Return the URL to the JavaScript i18n catalog file.

Parameters
block (XBlock) – The block that is requesting the URL.

This method returns None in NullI18nService. When implemented in a runtime, it should return the URL to the JavaScript i18n catalog so it can be loaded in frontends.

strftime(*dtime*, *format*)

Locale-aware strftime, with format short-cuts.

property 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.

property 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*, *id_generator*, *field_data=None*, *mixins=()*, *services=None*, *default_class=None*, *select=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** (*dict*) – 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-like subtypes found when calling *XBlock.load_class()* or *XBlockAside.load_class()* to resolve a *block_type*.

add_block_as_child_node(*block*, *node*)

Export *block* as a child node of *node*.

add_node_as_child(*block*, *node*)

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*.

property 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 instances for all of the asides that will decorate this *block*.

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

Returns
List of XBlockAside instances

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=''*)

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

abstract handler_url(*block*, *handler_name*, *suffix=''*, *query=''*, *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** (*str*) – The HTML result from rendering the block
- **aside_frag_fns** (*list((aside, aside_fn))*) – The asides and closures for rendering to call

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*.

abstract 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*)

Parse an open XML file, returning a usage id.

parse_xml_string(*xml*)

Parse a string of XML, returning a usage id.

abstract 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_children(*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()*.

abstract 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.

save_block(*block*)

Finalize/commit changes for the field data from the specified block. Called at the end of an XBlock's save() method. Runtimes may ignore this as generally the field data implementation is responsible for persisting changes.

(The main use case here is a runtime and field data implementation that want to store field data in XML format - the only way to correctly serialize a block to XML is to ask the block to serialize itself all at once, so such implementations cannot persist changes on a field-by-field basis.)

Parameters
block (*XBlock*) – the block being saved

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

Returns
An object implementing the requested service, or None.

property user_id

Access the current user ID.

Deprecated in favor of a 'user' service.

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

SIX

PLUGINS API

class xblock.plugin.**Plugin**

Base class for a system that uses entry_points to load plugins.

Implementing classes are expected to have the following attributes:

entry_point: The name of the entry point to load plugins from.

classmethod 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

classmethod 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 register_temp_plugin(*class_*, *identifier=None*, *dist='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.
```

class xblock.reference.plugins.**Filesystem**(*help=None*, *default=fields.UNSET*, *scope=ScopeBase(user=UserScope.NONE, block=BlockScope.DEFINITION, name='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: https://github.com/openedx/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.

CHAPTER

SEVEN

EXCEPTIONS API

Module for all xblock exception classes

exception xblock.exceptions.**DisallowedFileError**

Raised by *XBlock.open_local_resource()* or *XBlockAside.open_local_resource()*.

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 occurred

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.**UserIdDeprecationWarning**

Warning for use of deprecated user_id accessor

exception xblock.exceptions.**XBlockNotFoundError**(*usage_id*)

Raised to indicate that an XBlock could not be found with the requested usage_id

exception xblock.exceptions.**XBlockParseException**

Raised if parsing the XBlock olx fails.

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 occurred *dirty_fields* - a set of fields that were left dirty after the save

CHAPTER

EIGHT

OPEN EDX XBLOCK TUTORIAL

8.1 Introduction

The *Open edX XBlock Tutorial* is created using RST files and Sphinx. You, the user community, can help update and revise this documentation project on GitHub.

https://github.com/openedx/XBlock/tree/master/docs/xblock-tutorial/

The Open edX community welcomes contributions from other Open edX community members. You can find guidelines for how to contribute to Open edX documentation in the GitHub `openedx/docs.openedx.org` repository - although note that these specific docs are authored in the `openedx/XBlock` repository.

8.1.1 Other Open edX Resources

The docs.openedx.org site has numerous resources for learning about the Open edX platform. Specifically, there are pages of information that are targeted at the following audiences:

- Users of named releases
- Educators (those using the Open edX platform for teaching)
- Course Operators (those engaged in the mechanics of running an Open edX course)
- Site Operators
- Developers
- Documentors
- Translators

8.2 XBlock Overview

8.2.1 Introduction to XBlocks

This section introduces XBlocks.

Overview

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 Open edX Platform or other XBlock runtime application, then used by any course team using that system.

In educational applications, XBlocks can be used to represent individual problems, web-formatted text and videos, interactive simulations and labs, or collaborative learning experiences. Furthermore, XBlocks are composable, allowing an XBlock developer to control the display of other XBlocks to compose lessons, sections, and entire courses.

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.

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 `<div>` 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.

XBlock API and Runtimes

Any web application can be an *XBlock runtime* by implementing the XBlock API. Note that the XBlock API is not a RESTful API. XBlock runtimes can compose web pages out of XBlocks that were developed by programmers who do not need to know anything about the other components that a web page might be using or displaying.

XBlocks and the Open edX Platform

The Open edX Platform is an XBlock runtime and the Open edX community currently provides most of the support for the development of the XBlock library and specification. Programmers who use Tutor or the edx-platform devstack instead of the xblock-sdk to develop an XBlock should make sure that their XBlock is fully compliant with the XBlock specification before deploying to other XBlock runtimes. More specifically, XBlocks should package any services provided by edx-platform that a different XBlock compliant runtime might not provide.

The Open edX Platform currently has a large suite of XBlocks built into its primary repository that are available to course developers. Those XBlocks include HTML content, videos, and interactive problems. The Open edX Platform also includes many specialized XBlocks such as the Google Drive file tool and Open Response Assessments. For more information, see *XBlocks and the edX Platform*.

XBlocks for Developers

Developers can select from functionality developed by the Open edX community by installing an XBlock on their Open edX instance. Developers can integrate new or propriety functionality for use in XBlock runtimes by developing a new XBlock using the supported XBlock API.

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 `<div>` 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.

Prerequisites

This tutorial is for developers who are new to XBlock. To use this tutorial, you should have basic knowledge of the following technologies.

- Python
- JavaScript
- HTML and CSS
- Python VirtualEnv
- Git

XBlock Resources

This tutorial is meant to guide developers through the process of creating an XBlock, and to explain the *concepts* and *anatomy* of XBlocks.

The XBlock SDK supports the creation of new XBlocks. Developers should also see the *Open edX XBlock API Guide*.

XBlock Independence and Interoperability

You must design your XBlock to meet two criteria.

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

8.2.2 XBlock Examples

This section shows example XBlocks. These examples are meant to demonstrate simple XBlocks and are not meant to showcase the range of capabilities.

Google Drive & Calendar XBlock

Course teams can use the Google Drive and Calendar XBlock to embed Google documents and calendars in their courseware.

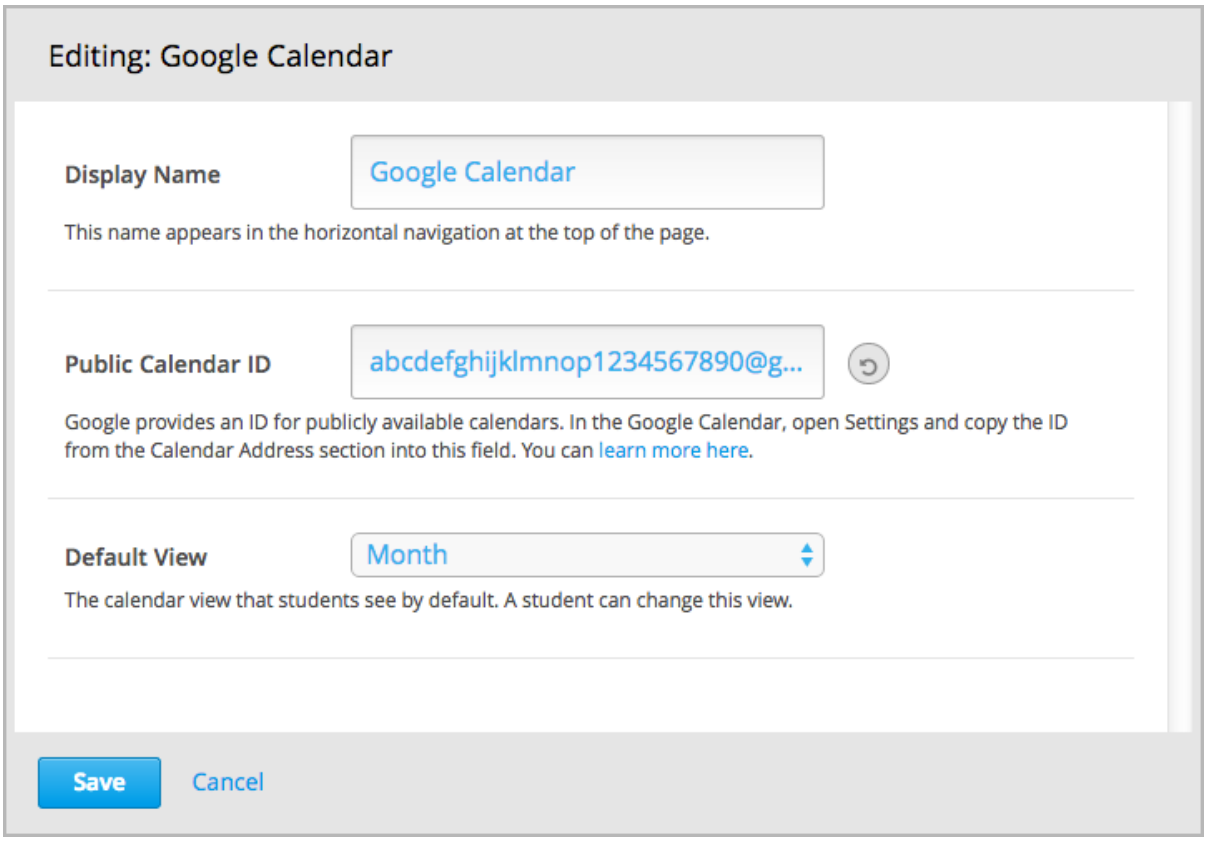
The Google Drive and Calendar XBlock is created and stored in a separate GitHub repository. You can explore the contents of this XBlock repository to learn how it is structured and developed.

Instructions are provided so that you can install the XBlock on your Open edX system. For more information, see *XBlocks and the edX Platform*.

Adding the XBlock to Courseware

When the Google Drive and Calendar XBlock is installed on an Open edX instance, course teams can add Google documents and calendars to courseware.

For example, in Studio, course teams can add and configure a Google calendar component.



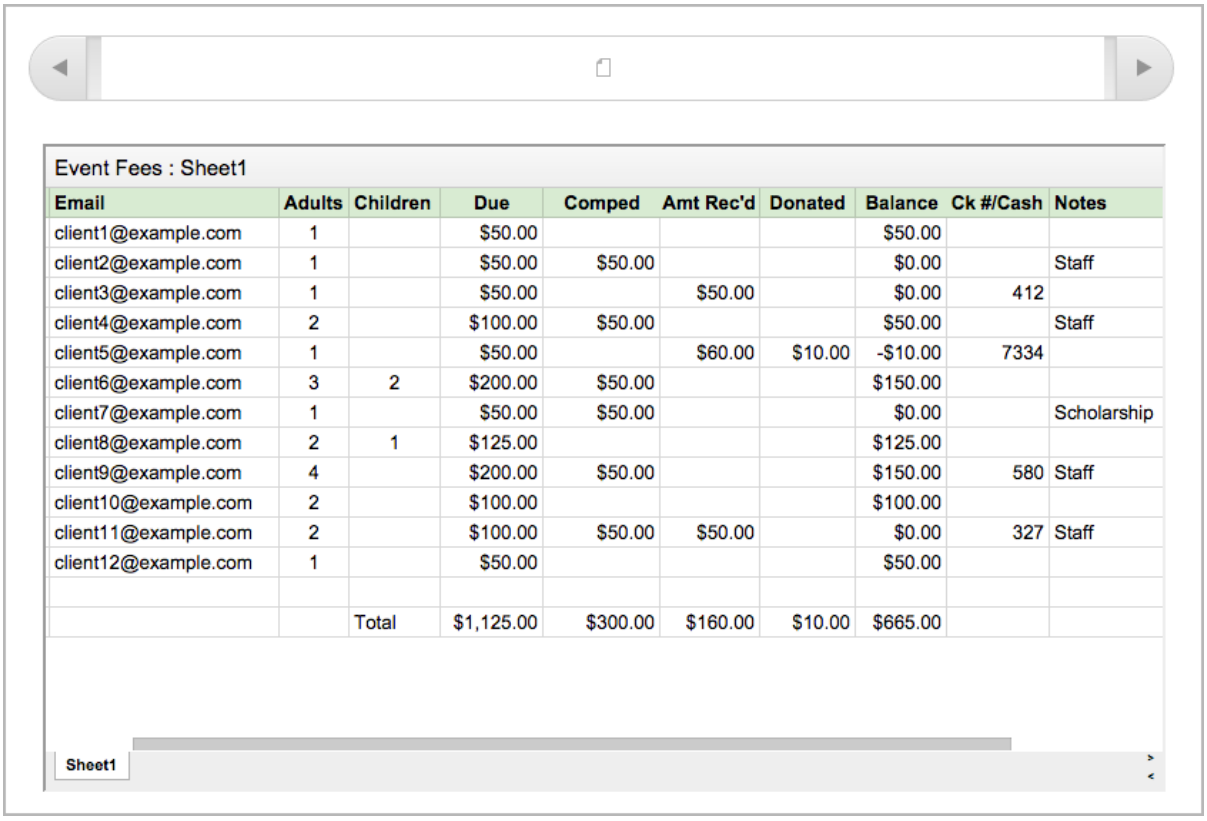
Course teams or developers can also add a Google calendar using OLX (open learning XML).

```
<google-calendar
  url_name="4115e717366045eaae7764b2e1f25e4c"
  calendar_id="abcdefghijklmnop1234567890@group.calendar.google.com"
  default_view="1"
  display_name="Class Schedule"
/>
```

For more information, see Google calendar tool and Google Drive file tool in *Building and Running an Open edX Course*.

Viewing the XBlock

When course teams use the Google Drive and Calendar XBlock, learners can view the referenced Google documents and calendars directly in their the courseware.



Event Fees : Sheet1

Email	Adults	Children	Due	Comped	Amt Rec'd	Donated	Balance	Ck #/Cash	Notes
client1@example.com	1		$50.00				$50.00		
client2@example.com	1		$50.00	$50.00			$0.00		Staff
client3@example.com	1		$50.00		$50.00		$0.00	412	
client4@example.com	2		$100.00	$50.00			$50.00		Staff
client5@example.com	1		$50.00		$60.00	$10.00	-$10.00	7334	
client6@example.com	3	2	$200.00	$50.00			$150.00		
client7@example.com	1		$50.00	$50.00			$0.00		Scholarship
client8@example.com	2	1	$125.00				$125.00		
client9@example.com	4		$200.00	$50.00			$150.00	580	Staff
client10@example.com	2		$100.00				$100.00		
client11@example.com	2		$100.00	$50.00	$50.00		$0.00	327	Staff
client12@example.com	1		$50.00				$50.00		
		Total	$1,125.00	$300.00	$160.00	$10.00	$665.00		

Sheet1

Examples in the XBlock SDK

The XBlock SDK that you use in this tutorial also contains several example XBlocks.

We will use the Thumbs XBlock in the sections *Customize Your XBlock* and *Anatomy of an XBlock*.

You can explore the other example XBlocks in the XBlock SDK.

- View Counter XBlock
- Problem XBlock
- Slider XBlock
- Several Content XBlocks
- Several Structure XBlocks

8.3 Build an XBlock: Quick Start

This part of the tutorial guides you through building an XBlock. At the end, you will have the skeleton of an XBlock that you can then *customize*.

To continue, see the following sections.

8.3.1 Install XBlock Prerequisites

To build an XBlock, you must have the following tools on your computer.

- *Python 3.8*
- *Git*
- *A Virtual Environment*

Python 3.8

To run the a virtual environment and the XBlock SDK, and to build an XBlock, you must have Python 3.8 installed on your computer.

Download Python for your operating system and follow the installation instructions.

Git

Open edX repositories, including XBlock and the XBlock SDK, are stored on GitHub.

To build your own XBlock, and to deploy it later, you must use Git for source control.

If you do not have Git installed, or you are are unfamiliar with the tool, see the GitHub Help.

A Virtual Environment

It is recommended that you develop your XBlock using a Python virtual environment. A virtual environment is a tool to keep the dependencies required by different projects in separate places.

With a virtual environment you can manage the requirements of your XBlock in a separate location so they do not conflict with requirements of other Python applications you might need.

The instructions and examples in this tutorial use VirtualEnv and VirtualEnvWrapper to build XBlocks. You can also use PyEnv.

After you have installed Python 3.8, follow the VirtualEnv Installation instructions.

For information on creating the virtual environment for your XBlock, see *Create and Activate the Virtual Environment*.

8.3.2 Set Up the XBlock Software Development Kit

Before you continue, make sure that you are familiar with the subjects in the *Install XBlock Prerequisites* section.

When you have installed all prerequisites, you are ready to set up the XBlock SDK in a virtual environment. To do this, complete the following steps.

Create a Directory for XBlock Work

It is recommended that you create a directory in which to store all your XBlock work, including a virtual environment, the XBlock SDK, and the XBlocks you develop.

1. At the command prompt, run the following command to create the directory.

```
$ mkdir xblock_development
```

2. Change directories to the `xblock_development` directory.

```
$ cd xblock_development
```

   The rest of your work will be from this directory.

Create and Activate the Virtual Environment

You must have a virtual environment tool installed on your computer. For more information, see *Install XBlock Prerequisites*. If you have multiple Python versions on your machine, see managing different Python versions with virtualenv.

Then create the virtual environment in your `xblock_development` directory.

1. At the command prompt in `xblock_development`, run the following command to create the virtual environment.

```
$ virtualenv xblock-env
```

2. Run the following command to activate the virtual environment.

```
$ source xblock-env/bin/activate
```

   When the virtual environment is activated, the command prompt shows the name of the virtual directory in parentheses.

```
(xblock-env) $
```

Clone the XBlock Software Development Kit

The XBlock SDK is a Python application you use to help you build new XBlocks. The XBlock SDK contains three main components:

- An XBlock creation tool that builds the skeleton of a new XBlock.
- An XBlock runtime for viewing and testing your XBlocks during development.
- Sample XBlocks that you can use as the starting point for new XBlocks, and for your own learning.

After you *create and activate the virtual environment*, you clone the XBlock SDK and install its requirements. To do this, complete the following steps at a command prompt.

1. In the `xblock_development` directory, run the following command to clone the XBlock SDK repository from GitHub.

```
(xblock-env) $ git clone https://github.com/openedx/xblock-sdk.git
```

2. In the same directory, create an empty directory called *var*.

```
(xblock-env) $ mkdir var
```

3. Run the following command to change to the `xblock-sdk` directory.

```
(xblock-env) $ cd xblock-sdk
```

4. Run the following commands to install the XBlock SDK requirements.

```
(xblock-env) $ make install
```

5. Run the following command to return to the `xblock_development` directory, where you will perform the rest of your work.

```
(xblock-env) $ cd ..
```

When the requirements are installed, you are in the `xblock_development` directory, which contains the `var`, `xblock-env`, and `xblock-sdk` subdirectories. You can now *create your first XBlock*.

8.3.3 Create Your First XBlock

Before you continue, make sure that you have *set up the XBlock SDK*. You then create the XBlock and deploy it in the XBlock SDK.

Create an XBlock

You use the XBlock SDK to create skeleton files for an XBlock. To do this, follow these steps at a command prompt.

1. Change to the `xblock_development` directory, which contains the `var`, `xblock-env`, and `xblock-sdk` subdirectories.
2. Run the following command to create the skeleton files for the XBlock.

```
(xblock-env) $ xblock-sdk/bin/workbench-make-xblock
```

   Instructions in the command window instruct you to determine a short name and a class name. Follow the guidelines in the command window to determine the names that you want to use.

```
You will be prompted for two pieces of information:

* Short name: a single word, all lower-case, for directory and file
  names. For a hologram 3-D XBlock, you might choose "holo3d".

* Class name: a valid Python class name.  It's best if this ends with
  "XBlock", so for our hologram XBlock, you might choose
  "Hologram3dXBlock".

Once you specify those two names, a directory is created in the
``xblock_development`` directory containing the new project.

If you don't want to create the project here, or you enter a name
incorrectly, type Ctrl-C to stop the creation script.  If you don't want
the resulting project, delete the directory it created.
```

3. At the command prompt, enter the Short Name you selected for your XBlock.

```
$ Short name: myxblock
```

4. At the command prompt, enter the Class name you selected for your XBlock.

```
$ Class name: MyXBlock
```

The skeleton files for the XBlock are created in the `myxblock` directory. For more information about the XBlock files, see *Anatomy of an XBlock*.

Install the XBlock

After you create the XBlock, you install it in the XBlock SDK.

In the `xblock_development` directory, use `pip` to install your XBlock.

```
(xblock-env) $ pip install -e myxblock
```

You can then test your XBlock in the XBlock SDK.

Create the SQLite Database

Before running the XBlock SDK the first time, you must create the SQLite database.

1. In the `xblock_development` directory, run the following command to create the database and the tables.

```
(xblock-env) $ python xblock-sdk/manage.py migrate
```

Run the XBlock SDK Server

To see the web interface of the XBlock SDK, you must run the SDK server.

In the `xblock_development` directory, run the following command to start the server.

```
(xblock-env) $ python xblock-sdk/manage.py runserver
```

Note: If you do not specify a port, the XBlock SDK server uses port 8000. To use a different port, specify it in the `runserver` command.

Then test that the XBlock SDK is running. In a browser, go to `http://localhost:8000`. You should see the following page.

The page shows the XBlocks installed automatically with the XBlock SDK. Note that the page also shows the **MyXBlock** XBlock that you created in *Create Your First XBlock*.

Get Help for the XBlock SDK Server

To get help for the XBlock SDK `runserver` command, run the following command.

```
(xblock-env) $ python xblock-sdk/manage.py help
```

The command window lists and describes the available commands.

Next Steps

You have now completed the Getting Started section of the XBlock tutorial. In the next sections, you will learn *how to use the XBlock SDK*, about the *anatomy of an XBlock*, and *how to customize your new XBlock*.

8.3.4 What Browsers Do I Need to Support?

For the latest information on browser support for the Open edX platform, see Open edX Browser Support.

8.4 Anatomy of an XBlock

This part of the tutorial explains the XBlock skeleton, and uses examples from the Thumbs XBlock that is installed with the XBlock SDK.

The Thumbs XBlock enables learners to vote up or down. The Thumbs XBlock keeps track of vote totals.

For information about making the XBlock that you created function like the example Thumbs XBlock, see *Customize Your XBlock*.

8.4.1 The XBlock Python File

This section of the tutorial walks through the Python file, thumbs.py, for the Thumbs XBlock example in the XBlock SDK.

If you completed the steps in *Build an XBlock: Quick Start*, you can find this file locally at `xblock_development/xblock-sdk/sample_xblocks/thumbs/thumbs.py`.

In the XBlock Python file, you define *fields*, *views*, *handlers*, and workbench scenarios.

- *Thumb XBlock Fields*
- *Thumb XBlock Student View*
- *Thumb XBlock Vote Handler*

Thumb XBlock Fields

The `thumbs.py` file defines the following fields for the XBlock in the `ThumbsBlockBase` class.

```
class ThumbsBlockBase(object):
    upvotes = Integer(
        help="Number of up votes",
        default=0,
        scope=Scope.user_state_summary
    )
    downvotes = Integer(
        help="Number of down votes",
        default=0,
        scope=Scope.user_state_summary
    )
    voted = Boolean(
```

(continues on next page)

(continued from previous page)

```
        help="Has this student voted?",
        default=False,
        scope=Scope.user_state
    )
```

Note the following details about the fields in the Thumbs XBlock.

- `upvotes` and `downvotes` store the cumulative up and down votes of users.

  These fields have the scope `Scope.user_state_summary`. This indicates that the data in these fields are specific to the XBlock and the same for all users.

- `voted` stores whether the user has voted. This field has the scope `Scope.user_state`. This indicates that the data in this field applies to the XBlock and to the specific user.

For more information, see *XBlock Fields*.

Thumb XBlock Student View

The `thumbs.py` file defines the student view for the XBlock in the `ThumbsBlockBase` class.

```
def student_view(self, context=None):  # pylint: disable=W0613
    """
    Create a fragment used to display the XBlock to a student.
    `context` is a dictionary used to configure the display (unused)

    Returns a `Fragment` object specifying the HTML, CSS, and JavaScript
    to display.
    """

    # Load the HTML fragment from within the package and fill in the template

    html_str = pkg_resources.resource_string(
        __name__,
        "static/html/thumbs.html".decode('utf-8')
    )
    frag = Fragment(str(html_str).format(block=self))

    # Load the CSS and JavaScript fragments from within the package
    css_str = pkg_resources.resource_string(
        __name__,
        "static/css/thumbs.css".decode('utf-8')
    )
    frag.add_css(str(css_str))

    js_str = pkg_resources.resource_string(
        __name__,
        "static/js/src/thumbs.js".decode('utf-8')
    )
    frag.add_javascript(str(js_str))

    frag.initialize_js('ThumbsBlock')
    return frag
```

The student view composes and returns the fragment from static HTML, JavaScript, and CSS files. A web page displays the fragment to learners.

Note the following details about student view.

- The static HTML content is added when the fragment is initialized.

```
html_str = pkg_resources.resource_string(
     __name__,
     "static/html/thumbs.html".decode('utf-8')
 )
frag = Fragment(str(html_str).format(block=self))
```

- The JavaScript and CSS file contents are added to the fragment with the `add_javascript()` and `add_css()` methods.
- The JavaScript in the fragment must be initialized using the name of the XBlock class. The name also maps to the function that initializes the XBlock in the *JavaScript file*.

```
frag.initialize_js('ThumbsBlock')
```

For more information, see *View Methods*.

Thumb XBlock Vote Handler

The `thumbs.py` file defines a handler that adds a user's vote to the XBlock.

```
@XBlock.json_handler
def vote(self, data, suffix=''):  # pylint: disable=unused-argument
    """
    Update the vote count in response to a user action.
    """
    # Here is where we would prevent a student from voting twice, but then
    # we couldn't click more than once in the demo!
    #
    #     if self.voted:
    #         log.error("cheater!")
    #         return

    if data['voteType'] not in ('up', 'down'):
        log.error('error!')
        return

    if data['voteType'] == 'up':
        self.upvotes += 1
    else:
        self.downvotes += 1

    self.voted = True

    return {'up': self.upvotes, 'down': self.downvotes}
```

Note the following details about the vote handler.

- The `upvotes` or `downvotes` fields are updated based on the user's vote.

- The `voted` field is set to `True` for the user.
- The updated `upvotes` and `downvotes` fields are returned.

For more information, see *Handler Methods*.

8.4.2 The XBlock HTML File

This section of the tutorial walks through the HTML file, thumbs.html, that is part of the Thumbs XBlock in the XBlock SDK.

If you completed the steps in *Build an XBlock: Quick Start*, you can find this file locally at `xblock_development/xblock-sdk/sample_xblocks/thumbs/static/html/thumbs.html`.

In the XBlock HTML file, you define the HTML content that is added to a *fragment*. The HTML content can reference the XBlock *fields*. The fragment is returned by the *view method*, to be displayed by the *runtime* application.

```
<p>
  <span class='upvote'><span class='count'>{self.upvotes}</span>&uarr;</span>
  <span class='downvote'><span class='count'>{self.downvotes}</span>&darr;</span>
</p>
```

Note the following details about the HTML file.

- The `class` values reference styles in the `thumbs.css` file. For more information, see *The XBlock Stylesheets*.
- The values `self.upvotes` and `self.downvotes` reference the fields in the XBlock Python class.

8.4.3 The XBlock JavaScript File

This section of the tutorial walks through the JavaScript file, thumbs.js, that is part of the Thumbs XBlock in the XBlock SDK.

If you completed the steps in *Build an XBlock: Quick Start*, you can find this file locally at `xblock_development/xblock-sdk/sample_xblocks/thumbs/static/js/src/thumbs.js`.

In the XBlock JavaScript file, you define code that manages user interaction with the XBlock. The code is added to a *fragment*.

The XBlock's JavaScript uses the runtime handler, and can use the `children` and `childMap` functions as needed.

The JavaScript references the XBlock *fields* and *methods*. The fragment is returned by the *view method*, to be displayed by the *runtime* application.

```
function ThumbsAside(runtime, element, block_element, init_args) {
  return new ThumbsBlock(runtime, element, init_args);
}

function ThumbsBlock(runtime, element, init_args) {
  function updateVotes(votes) {
    $('.upvote .count', element).text(votes.up);
    $('.downvote .count', element).text(votes.down);
  }

  var handlerUrl = runtime.handlerUrl(element, 'vote');

  $('.upvote', element).click(function(eventObject) {
```

(continues on next page)

(continued from previous page)

```
    $.ajax({
        type: "POST",
        url: handlerUrl,
        data: JSON.stringify({voteType: 'up'}),
        success: updateVotes
    });
  });

  $('.downvote', element).click(function(eventObject) {
    $.ajax({
        type: "POST",
        url: handlerUrl,
        data: JSON.stringify({voteType: 'down'}),
        success: updateVotes
    });
  });
  return {};
};
```

Note the following details about the JavaScript file.

- The function `ThumbsBlock` initializes the XBlock. A JavaScript function to initialize the XBlock is required.
- The `ThumbsBlock` function maps to the constructor in the *XBlock Python file* and provides access to its methods and fields.
- The `ThumbsBlock` function uses the runtime handler.

  ```
  var handlerUrl = runtime.handlerUrl(element, 'vote');
  ```

- The `ThumbsBlock` function includes the `POST` commands to increase the up and down votes in the XBlock.

The XBlock JavaScript code can also use the `children` and `childMap` functions as needed. For more information, see *XBlock Children*.

8.4.4 The XBlock Stylesheets

This section of the tutorial walks through the CSS file, thumbs.css, that is part of the Thumbs XBlock in the XBlock SDK.

If you completed the steps in *Build an XBlock: Quick Start*, you can find this file locally at `xblock_development/xblock-sdk/sample_xblocks/thumbs/static/css/thumbs.css`.

In the XBlock CSS file, you define the styles that are added to the fragment that is returned by the view method to be displayed by the runtime application.

```
.upvote, .downvote {
  cursor: pointer;
  border: 1px solid #888;
  padding: 0 .5em;
}
.upvote { color: green; }
.downvote { color: red; }
```

The styles in `thumbs.css` are referenced in the *XBlock HTML file*.

8.5 Customize Your XBlock

Now that you have created your XBlock skeleton, `myxblock`, you need to make it do something. This part of the tutorial explains modifying `myxblock`; for practical purposes, we will update it to match the Thumbs XBlock that is installed with the XBlock SDK.

For more information about the Thumbs XBlock, see *Anatomy of an XBlock*

For more information about the different components of an XBlock, see *XBlock Concepts*.

8.5.1 Customize myxblock.py

This section describes how to modify the Python file of the XBlock you created, `myxblock.py`, to provide the functionality in the Thumbs XBlock example in the XBlock SDK.

In `myxblock.py`, you will define *fields*, *views*, *handlers*, and workbench scenarios.

The Default XBlock Python File

When you *create a new XBlock*, the default Python file is created automatically, with skeletal functionality defined. In the `xblock_development/myxblock/myxblock/` directory, see the file `myxblock.py`.

```
"""TO-DO: Write a description of what this XBlock is."""

import pkg_resources

from web_fragments.fragment import Fragment
from xblock.core import XBlock
from xblock.fields import Integer, Scope


class MyXBlock(XBlock):
    """
    TO-DO: document what your XBlock does.
    """

    # Fields are defined on the class.  You can access them in your code as
    # self.<fieldname>.

    # TO-DO: delete count, and define your own fields.
    count = Integer(
        default=0, scope=Scope.user_state,
```

(continues on next page)

(continued from previous page)

```
        help="A simple counter, to show something happening",
    )

    def resource_string(self, path):
        """Handy helper for getting resources from our kit."""
        data = pkg_resources.resource_string(__name__, path)
        return data.decode("utf8")

    # TO-DO: change this view to display your data your own way.
    def student_view(self, context=None):
        """
        The primary view of the MyXBlock, shown to students
        when viewing courses.
        """
        html = self.resource_string("static/html/myxblock.html")
        frag = Fragment(html.format(self=self))
        frag.add_css(self.resource_string("static/css/myxblock.css"))
        frag.add_javascript(self.resource_string("static/js/src/myxblock.js"))
        frag.initialize_js('MyXBlock')
        return frag

    # TO-DO: change this handler to perform your own actions.  You may need more
    # than one handler, or you may not need any handlers at all.
    @XBlock.json_handler
    def increment_count(self, data, suffix=''):
        """
        An example handler, which increments the data.
        """
        # Just to show data coming in...
        assert data['hello'] == 'world'

        self.count += 1
        return {"count": self.count}

    # TO-DO: change this to create the scenarios you'd like to see in the
    # workbench while developing your XBlock.
    @staticmethod
    def workbench_scenarios():
        """A canned scenario for display in the workbench."""
        return [
            ("MyXBlock",
            """<myxblock/>
            """),
            ("Multiple MyXBlock",
            """<vertical_demo>
                <myxblock/>
                <myxblock/>
                <myxblock/>
                </vertical_demo>
            """),
        ]
```

Add Comments

As a best practice and because XBlocks can be shared, you should add comments to the `myxblock.py` file. Replace the "TO DO" indicators with a description of what the XBlock does and any details future developers or users would want to know.

Add XBlock Fields

You determine the data your XBlock stores through *fields*. Fields store user and XBlock state as JSON data.

To customize your `myxblock.py` file so that it has the same functionality as the `thumbs.py` file, you need to add three fields to the XBlock, each with the right *scope*.

- `upvotes`, to store the number of times users up-vote the XBlock. The value applies to the XBlock and all users collectively.
- `downvotes`, to store the number of times users down-vote the XBlock. The value applies to the XBlock and all users collectively.
- `voted`, to record whether or not the user has voted. The value applies to the XBlock and each user individually.

Review the *XBlock Fields* section, then add the required fields to `myxblock.py`. You can remove the `count` field, which was defined automatically when you created the XBlock.

Check Fields Against the Thumbs XBlock

After you have defined the fields, check your work against the fields in the Thumbs XBlock, in the file `xblock_development/xblock-sdk/sample_xblocks/thumbs/thumbs.py`.

```
class ThumbsBlockBase(object):
    upvotes = Integer(
        help="Number of up votes",
        default=0,
        scope=Scope.user_state_summary
    )
    downvotes = Integer(
        help="Number of down votes",
        default=0,
        scope=Scope.user_state_summary
    )
    voted = Boolean(
        help="Has this student voted?",
        default=False,
        scope=Scope.user_state
    )
```

If necessary, make corrections to the fields in your XBlock so that they match the fields in the Thumbs XBlock.

Note the following details.

- `upvotes` and `downvotes` have the scope `Scope.user_state_summary`. This indicates that the data in these fields are specific to the XBlock and the same for all users.
- `voted` has the scope `Scope.user_state`. This indicates that the data in this field applies to the XBlock and to the specific user.

Define the Student View

The XBlock Python file must contain one or more *view methods*.

To run the XBlock in the Open edX Platform Learning Management System, there must be a method named `student_view`. If you intend the XBlock to run in a different *runtime application*, you might need to define a different name. For more information, see *Open edX Learning Management System as an XBlock Runtime*.

In `myxblock.py`, examine the `student_view` method that was defined automatically when you created the XBlock.

The student view composes and returns the *fragment* from static HTML, JavaScript, and CSS files. A web page displays the fragment to learners.

Note the following details about student view.

- The static HTML is added when the fragment is initialized.

```
html = self.resource_string("static/html/myxblock.html")
frag = Fragment(unicode(html_str).format(self=self))
```

- The JavaScript and CSS files are added to the fragment with the `add_javascript()` and `add_css()` methods.
- The JavaScript in the fragment must be initialized using the name of the XBlock class. The name also maps to the function that initializes the XBlock in the *JavaScript file*.

```
frag.initialize_js('MyXBlock')
```

As you can see, the necessary functions of the view were added automatically. Check the student view in `myxblock.py` against the student view in thumbs.py. Note that the only differences are the file names of the HTML, CSS, and JavaScript files added to the fragment. As the file names are correct for MyXBlock, you do not need to edit the student view at all.

Define the Vote Handler

Handlers process input events from the XBlock JavaScript code. You use handlers to add interactivity to your block. In your XBlock, you use a handler to process votes from users.

The vote handler in your XBlock must perform the following functions.

1. Update `upvotes` or `downvotes` fields based on the user's vote.
2. Set the `voted` field to `True` for the user.
3. Return the updated `upvotes` and `downvotes` fields.

Review the *XBlock Methods* section, then implement the vote handler in `myxblock.py`.

You can use any name for the vote handler, and you will use the same name in the JavaScript code to connect browser events to the vote handler running in the server. To match the Thumbs XBlock, use the name `vote`.

Check the Handler Against the Thumbs XBlock

After you have defined the vote handler, check your work against the handler in the Thumbs XBlock.

```
@XBlock.json_handler
def vote(self, data, suffix=''):  # pylint: disable=unused-argument
    """
    Update the vote count in response to a user action.
    """
    # Here is where we would prevent a student from voting twice, but then
    # we couldn't click more than once in the demo!
    #
    #     if self.voted:
    #         log.error("cheater!")
    #         return

    if data['voteType'] not in ('up', 'down'):
        log.error('error!')
        return

    if data['voteType'] == 'up':
        self.upvotes += 1
    else:
        self.downvotes += 1

    self.voted = True

    return {'up': self.upvotes, 'down': self.downvotes}
```

If necessary, make corrections to the handler in your XBlock so that it matches the handler in the Thumbs XBlock.

Next Step

After you complete your customizations to the Python file, you can continue on and *customize the XBlock HTML file*.

8.5.2 Customize myxblock.html

This section describes how to modify the static HTML file of the XBlock you created, `myxblock.html`, to provide the functionality in the Thumbs XBlock example in the XBlock SDK.

In `myxblock.html`, you will define the HTML content that is added to a *fragment*. The HTML content can reference the XBlock *fields*. The fragment is returned by the *view method*.

The Default XBlock HTML File

When you *create a new XBlock*, the default static HTML file is created automatically, with skeletal functionality defined. In the `xblock_development/myxblock/myxblock/static/html` directory, see the file `myxblock.html`.

```
<div class="myxblock_block">
  <p>MyXBlock: count is now
    <span class='count'>{self.count}</span> (click me to increment).
  </p>
</div>
```

The file contains HTML to display the `count` field that was added by default to the XBlock. Delete the HTML between the `div` elements.

Add HTML Content

You can create HTML as needed to display the state of your XBlock. The Thumbs XBlock displays the up and down votes. Create a single paragraph and follow the guidelines below.

- Create two `span` elements, to display up-votes and down-votes.
- Use `upvote` and `downvote` as class values for the `span` elements. You will define these classes in `myxblock.css`. For more information, see *Customize myxblock.css*.
- Within each `span` element, create another `span` element, each with the class value `count`. For the value of each embedded `span` element, reference the `upvotes` and `downvotes` fields you defined in the *Python file* for the XBlock.
- For the value of each of the outer `span` elements, use the HTML unicode characters `&uarr;` and `&darr` to show thumbs up and thumbs down symbols next to the number of votes.

Check HTML Against the Thumbs XBlock

After you have defined the HTML, check your work against the HTML in the Thumbs XBlock, in the file `xblock_development/xblock-sdk/sample_xblocks/thumbs/static/html/thumbs.html`.

```
<p>
  <span class='upvote'><span class='count'>{self.upvotes}</span>&uarr;</span>
  <span class='downvote'><span class='count'>{self.downvotes}</span>&darr;</span>
</p>
```

If necessary, make corrections to the HTML in your XBlock so that it matches the HTML in the Thumbs XBlock.

Next Step

After you complete your customizations to the HTML file, you can continue on and *customize the XBlock JavaScript file*.

8.5.3 Customize myxblock.js

This section describes how to modify the JavaScript file of the XBlock you created, `myxblock.js`, to provide the functionality in the Thumbs XBlock example in the XBlock SDK.

In `myxblock.js`, you will define code that manages user interaction with the XBlock. The code is added to a *fragment*.

The Default XBlock JavaScript File

When you *create a new XBlock*, the default JavaScript file is created automatically, with skeletal functionality defined. In the `xblock_development/myxblock/myxblock/static/js/snc` directory, see the file `myxblock.js`.

```
/* Javascript for MyXBlock. */
function MyXBlock(runtime, element) {

    function updateCount(result) {
        $('.count', element).text(result.count);
    }

    var handlerUrl = runtime.handlerUrl(element, 'increment_count');

    $('p', element).click(function(eventObject) {
        $.ajax({
            type: "POST",
            url: handlerUrl,
            data: JSON.stringify({"hello": "world"}),
            success: updateCount
        });
    });

    $(function ($) {
        /* Here's where you'd do things on page load. */
    });
}
```

The file contains JavaScript code to increment the `count` field that was added by default to the XBlock. Delete this code.

Add JavaScript Code

JavaScript code implements the browser-side functionality you need for your XBlock. The Thumbs XBlock uses clicks on the up and down vote buttons to call the handler to record votes.

Follow the guidelines below to implement JavaScript code.

- Add the function `MyXBlock` to initialize the XBlock.

  The `MyXBlock` function maps to the constructor in the *XBlock Python file* and provides access to its methods and fields.

- Add the URL to the vote handler to the `MyXBlock` function.

```
var handlerUrl = runtime.handlerUrl(element, 'vote');
```

- Add `POST` commands in the `MyXBlock` function to increase the up and down votes in the XBlock.

Note: Do not change the main function name, `MyXBlock`.

Check JavaScript Against the Thumbs XBlock

After you have defined the JavaScript code, check your work against the code in the Thumbs XBlock, in the file `xblock_development/xblock-sdk/sample_xblocks/thumbs/static/js/src/thumbs.js`.

```
function ThumbsAside(runtime, element, block_element, init_args) {
  return new ThumbsBlock(runtime, element, init_args);
}

function ThumbsBlock(runtime, element, init_args) {
  function updateVotes(votes) {
    $('.upvote .count', element).text(votes.up);
    $('.downvote .count', element).text(votes.down);
  }

  var handlerUrl = runtime.handlerUrl(element, 'vote');

  $('.upvote', element).click(function(eventObject) {
    $.ajax({
        type: "POST",
        url: handlerUrl,
        data: JSON.stringify({voteType: 'up'}),
        success: updateVotes
    });
  });

  $('.downvote', element).click(function(eventObject) {
    $.ajax({
        type: "POST",
        url: handlerUrl,
        data: JSON.stringify({voteType: 'down'}),
        success: updateVotes
    });
```

(continues on next page)

(continued from previous page)

```
  });
  return {};
};
```

If necessary, make corrections to the code in your XBlock so that it matches the code in the Thumbs XBlock.

Next Step

After you complete your customizations to the JavaScript file, you can continue on and *customize the XBlock CSS file*.

8.5.4 Customize myxblock.css

This section describes how to modify the static CSS file of the XBlock you created, `myxblock.css`, to provide the functionality in the Thumbs XBlock example in the XBlock SDK.

In `myxblock.css`, you define the styles that are added to the fragment that is returned by the view method to be displayed by the runtime application.

The Default XBlock CSS File

When you *create a new XBlock*, the default static CSS file is created automatically, with skeletal functionality defined. In the `xblock_development/myxblock/myxblock/static/css` directory, see the file `myxblock.css`.

```
/* CSS for MyXBlock */

.myxblock_block .count {
   font-weight: bold;
}

.myxblock_block p {
   cursor: pointer;
}
```

The file contains CSS code to format the `count` field that was added by default to the XBlock. Delete this code.

Add CSS Code

You must add CSS code to format the XBlock content. Follow the guidelines below.

- Create a single class that defines formatting for `.upvote` and `.downvote`.
- The cursor type is pointer.
- The border is 1px, solid, and with the color #888.
- The padding is `0.5em`;
- The color for `.upvote` is green and for `downvote` is red.

Check CSS Against the Thumbs XBlock

After you have defined the CSS code, check your work against the CSS in the Thumbs XBlock, in the file `xblock_development/xblock-sdk/sample_xblocks/thumbs/static/css/thumbs.css`.

```
.upvote, .downvote {
  cursor: pointer;
  border: 1px solid #888;
  padding: 0 .5em;
}
.upvote { color: green; }
.downvote { color: red; }
```

If necessary, make corrections to the CSS code in your XBlock so that it matches the code in the Thumbs XBlock.

The styles in `thumbs.css` are referenced in the *XBlock HTML file*.

Congrats!

You've completed customizing MyXBlock to have up and down voting functionality. Read on for more about XBlocks - and have fun making your next XBlock!

8.6 XBlock Concepts

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, lessons, or course sections. For more information, see *Introduction to XBlocks*.

This part of the tutorial provides conceptual information about XBlocks that all XBlock developers must understand.

8.6.1 XBlock Fields

You use XBlock fields to store state data for your XBlock.

- *XBlock Fields and State*
- *Field Scope*
- *Fields and Data Storage*

XBlock Fields and State

XBlock fields are Python attributes that store user and XBlock state as JSON data.

You define the fields in the XBlock Python file. For example, the `thumbs.py` file in the XBlock SDK includes three fields.

```
class ThumbsBlockBase(object):
    upvotes = Integer(
        help="Number of up votes",
        default=0,
        scope=Scope.user_state_summary
    )
    downvotes = Integer(
        help="Number of down votes",
        default=0,
        scope=Scope.user_state_summary
    )
    voted = Boolean(
        help="Has this student voted?",
        default=False,
        scope=Scope.user_state
    )
```

Field Names

The field names you define in the Python file are also used in the XBlock *JavaScript* and *HTML* code.

Field Parameters

When you initialize an XBlock field, you define three parameters.

- `help`: A help string for the field that can be used in an application such as edX Studio.
- `default`: The default value for the field.
- `scope`: The scope of the field. For more information, see the next section.

Field Scope

Field scope is the relationship of the field to users and the XBlock.

You define the field scope when initializing the field in the XBlock Python file. For example, in `thumbs.py`, the `voted` field is initialized to have the scope `user_state`.

```
voted = Boolean(help="Has this student voted?", default=False,
    scope=Scope.user_state)
```

User Scope

Fields can relate to users in the following ways.

- **No user**: the field data is not related to any users. No learner activity created modified the field value, and all learners see the same value.For example, a field that contains course content is independent of users.

Note: The XBlock cannot modify the value of a field that is not related to any users.

- **One user**: the field data is specific to a single user. For example, the answer to a problem is specific to the user who submitted it.
- **All users**: the field data is common for all users. Learner activity can change the field value, and all learners see the same value. For example, the total number of learners who answer a question is the same for all users.

Note: Field data related to all users is not the same as aggregate or query data. The same value is shared for all users, and you cannot associate specific actions to specific users.

XBlock Scope

Fields can relate to XBlocks in the following ways.

- **Block usage**: the field data is related to an instance, or usage, of the XBlock in a particular course. In most cases, you use the **Block usage** scope. For example, for an XBlock that polls learners and shows totals for each response, you would need the question and available answers to be specific to that instance of the XBlock in your course.
- **Block definition**: the field data is related to the definition of the XBlock. The definition is specified by the content creator. A definition can be shared across one or more uses. For example, you could create a single XBlock definition with many uses, and those uses can appear across courses or within the same course.
- **Block type**: The field data is related to the Python type of the XBlock, and is shared across all instances of the XBlock in all courses.
- **All**: The field data is related to all XBlocks, of all types. Any XBlock can access the field data.

Note: When you use the **All** scope, there is potential for name conflicts. If you have two fields of the same name with the scope **All** in different XBlock types, both fields point to the same data. Therefore you should use caution when using **All**.

User and Block Scope Independence

The user and block scope of a field are independent of each other. The field scope you define specifies both. The following examples show different ways you can combine user and block scope.

- A user's progress through a particular set of problems is stored in a field with the scope **One user** and **XBlock usage**.
- The content to display in an XBlock is stored in a field with the scope **No user** and **Block definition**.
- A user's preferences for a type of XBlock are stored in a field with the scope with **One user** and **XBlock type**.
- Information about the user, such as language or timezone, is stored in a field with the scope with **One user** and **All**.

Scope combinations that are used together frequently are available is a set of predefined scopes, as described below.

Predefined Scopes

XBlock includes the following predefined scopes that you can use when configuring fields. Each of these scopes includes the indicated user and block scope settings.

- `Scope.content`
  - Block definition
  - No user
- `Scope.settings`
  - Block usage
  - No user
- `Scope.user_state`
  - Block usage
  - One user
- `Scope.preferences`
  - Block type
  - One user
- `Scope.user_info`
  - All blocks
  - One user
- `Scope.user_state_summary`
  - Block usage
  - All users

Fields and Data Storage

Because XBlock fields are written and retrieved as single entities, you cannot store a large amount of data in a single field.

To store very large amounts of data, you should split the data across many smaller fields.

Initializing Fields

You do not use the `__init__` method with XBlocks.

XBlocks can be used in many contexts, and the `__init__` method might not work in those contexts.

To initialize field values, use one of the following alternatives.

- Use `xblock.fields.UNIQUE_ID` to set a default string value for the field.
- Use a lazy property decorator, so that when a field is first accessed, a function is called to set the value.
- Run the logic to set the default field value in the view instead of the `__init__` method.

Fields and OLX

XBlock fields map to attributes in the OLX (open learning XML) definition.

For example, you might include the fields `href`, `maxwidth`, and `maxheight` in a `SimpleVideoBlock` XBlock. You configure the fields as in the following example.

```
class SimpleVideoBlock(XBlock):
    """
    An XBlock providing Embed capabilities for video
    """

    href = String(help="URL of the video page at the provider",
        default=None, scope=Scope.content)
    maxwidth = Integer(help="Maximum width of the video", default=800,
        scope=Scope.content)
    maxheight = Integer(help="Maximum height of the video", default=450,
        scope=Scope.content)
```

By default, the `SimpleVideoBlock` XBlock is represented in OLX as in the following example:

```
<simplevideo
    href="https://vimeo.com/46100581"
    maxwidth="800"
    maxheight="450"
/>
```

You can customize the OLX representation of the XBlock by using the `xblock.parse_xml()` and `xblock.add_xml_to_node()` methods.

Field Requirements in the edX Platform

For information about field requirements in the edX Platform, see *Open edX LMS* and *Open edX Studio*.

Default Fields in a New XBlock

When you create a new XBlock, the `count` field is added to the Python file by default. This field is for demonstration purposes and you should replace it with your own field definitions.

8.6.2 XBlock Methods

You use XBlock methods in the XBlock Python file to define the behavior of your XBlock.

View Methods

XBlock view methods are Python methods invoked by the XBlock runtime to render the XBlock.

An XBlock can have multiple view methods. For example, an XBlock might have a student view for rendering the XBlock for learners, and an editing view for rendering the XBlock to course staff.

Note: The XBlock view names are specified by runtime applications; you cannot use arbitrary view names.

For information about the view requirements in the edX Platform, see *Open edX LMS* and *Open edX Studio*.

Typically, you define a view to produce a fragment that is used to render the XBlock as part of a web page. Fragments are aggregated hierarchically. You can use any field to affect the rendering of the XBlock as needed.

In the following example, the Thumbs sample XBlock in the XBlock SDK defines a student view.

```
def student_view(self, context=None):  # pylint: disable=W0613
    """
    Create a fragment used to display the XBlock to a student.
    `context` is a dictionary used to configure the display (unused)

    Returns a `Fragment` object specifying the HTML, CSS, and JavaScript
    to display.
    """

    # Load the HTML fragment from within the package and fill in the template

    html_str = pkg_resources.resource_string(
        __name__,
        "static/html/thumbs.html".decode('utf-8')
    )
    frag = Fragment(str(html_str).format(block=self))
```

(continues on next page)

(continued from previous page)

```
    # Load the CSS and JavaScript fragments from within the package
    css_str = pkg_resources.resource_string(
        __name__,
        "static/css/thumbs.css".decode('utf-8')
    )
    frag.add_css(str(css_str))

    js_str = pkg_resources.resource_string(
        __name__,
        "static/js/src/thumbs.js".decode('utf-8')
    )
    frag.add_javascript(str(js_str))

    frag.initialize_js('ThumbsBlock')
    return frag
```

Although view methods typically produce HTML-based renderings, they can be used for other purposes. See the documentation for your runtime application to verify the type of data the view must return and how it will be used.

Handler Methods

You write handlers to implement the server side of your XBlock's interactive features.

XBlock handlers are Python methods invoked by AJAX calls from the user's browser. Handlers accept an HTTP request and return an HTTP response.

An XBlock can have any number of handlers. For example, a problem XBlock might contain `submit` and `show_answer` handlers.

Each handler has a specific name of your choosing that is mapped to from specific URLs by the runtime. The runtime provides a mapping from handler names to specific URLs so that the XBlock JavaScript code can make requests to its handlers. Handlers can be used with `GET` and `POST` requests.

Handler methods also emit events for learner interactions and grades. For more information, see *When an XBlock Should Emit Events*.

In the following example, the Thumbs sample XBlock in the XBlock SDK defines a handler for voting.

```
def vote(self, data, suffix=''):  # pylint: disable=unused-argument
    """
    Update the vote count in response to a user action.
    """
    # Here is where we would prevent a student from voting twice, but then
    # we couldn't click more than once in the demo!
    #
    #     if self.voted:
    #         log.error("cheater!")
    #         return

    if data['voteType'] not in ('up', 'down'):
        log.error('error!')
        return
```

(continues on next page)

(continued from previous page)

```
    if data['voteType'] == 'up':
        self.upvotes += 1
    else:
        self.downvotes += 1

    self.voted = True

    return {'up': self.upvotes, 'down': self.downvotes}
```

Default Methods in a New XBlock

When you create a new XBlock, two methods are added automatically.

- The view method `student_view`.

  You can modify the contents of this view, but to use your XBlock with the edX Platform, you must keep the method name `student_view`.

- The handler method `increment_count`.

  This method is for demonstration purposes and you can remove it.

8.6.3 XBlock Fragments

A fragment is a part of a web page returned by an XBlock view method.

Fragment Contents

A fragment typically contains all the resources needed to display the XBlock in a web page, including HTML content, JavaScript, and CSS resources.

HTML Content

Content in a fragment is typically HTML, though some runtimes might require views that return other mime-types. Each fragment has only a single content value.

JavaScript

A fragment contains the JavaScript resources necessary to run the XBlock. JavaScript resources can include both external files to link to, and inline source code.

When fragments are composed, external JavaScript links are made unique, so that files are not loaded multiple times.

JavaScript Initializer

The JavaScript specified for a fragment can also specify a function to be called when that fragment is rendered on the page.

The DOM element containing all of the content in the fragment is passed to this function, which then executes any code needed to make that fragment operational.

The JavaScript view is also passed a JavaScript runtime object that contains a set of functions to generate links back to the XBlock's handlers and views on the runtime server.

For example, see the code in the Thumbs XBlock, in the file `xblock_development/xblock-sdk/sample_xblocks/thumbs/static/js/source/thumbs.js`.

```
function ThumbsAside(runtime, element, block_element, init_args) {
  return new ThumbsBlock(runtime, element, init_args);
}

function ThumbsBlock(runtime, element, init_args) {
  function updateVotes(votes) {
    $('.upvote .count', element).text(votes.up);
    $('.downvote .count', element).text(votes.down);
  }

  var handlerUrl = runtime.handlerUrl(element, 'vote');

  $('.upvote', element).click(function(eventObject) {
    $.ajax({
        type: "POST",
        url: handlerUrl,
        data: JSON.stringify({voteType: 'up'}),
        success: updateVotes
    });
  });

  $('.downvote', element).click(function(eventObject) {
    $.ajax({
        type: "POST",
        url: handlerUrl,
        data: JSON.stringify({voteType: 'down'}),
        success: updateVotes
    });
  });
  return {};
};
```

CSS

A fragment contains CSS resources to control how the XBlock is displayed. CSS resources can include both external files to link to and inline source code.

When fragments are composed, external JavaScript links will are made unique, so that files are not loaded multiple times.

Fragments and XBlock Children

Because XBlocks are nested hierarchically, a single XBlock view might require collecting renderings from each of its children, then composing them together. The parent XBlock view must handle composing its children's content together to create the parent content.

The fragment system has utilities for composing children's resources together into the parent.

Fragments and Views

You configure fragments in XBlock view methods.

In the following example, the Thumbs sample XBlock in the XBlock SDK defines a student view that composes and returns a fragment with HTML, JavaScript, and CSS strings generated from the XBlock's static files.

```
def student_view(self, context=None):  # pylint: disable=W0613
    """
    Create a fragment used to display the XBlock to a student.
    `context` is a dictionary used to configure the display (unused)

    Returns a `Fragment` object specifying the HTML, CSS, and JavaScript
    to display.
    """

    # Load the HTML fragment from within the package and fill in the template

    html_str = pkg_resources.resource_string(
        __name__,
        "static/html/thumbs.html".decode('utf-8')
    )
    frag = Fragment(str(html_str).format(block=self))

    # Load the CSS and JavaScript fragments from within the package
    css_str = pkg_resources.resource_string(
        __name__,
        "static/css/thumbs.css".decode('utf-8')
    )
    frag.add_css(str(css_str))

    js_str = pkg_resources.resource_string(
        __name__,
        "static/js/src/thumbs.js".decode('utf-8')
    )
    frag.add_javascript(str(js_str))
```

(continues on next page)

(continued from previous page)

```
    frag.initialize_js('ThumbsBlock')
    return frag
```

8.6.4 XBlock Children

An XBlock can have child XBlocks.

XBlock Tree Structure

An XBlock does not refer directly to its children. Instead, the structure of a tree of XBlocks is maintained by the runtime application, and is made available to the XBlock through a runtime service. For more information, see *XBlock Runtimes*.

This allows the runtime to store, access, and modify the structure of a course without incurring the overhead of the XBlock code itself.

XBlock children are not implicitly available to their parents. The runtime provides the parent XBlock with a list of child XBlock IDs. The child XBlock can then be loaded with the `get_child()` function. Therefore the runtime can defer loading child XBlocks until they are actually required.

Accessing Children (Server-Side)

To access XBlock children through the server, use the following methods.

- To iterate over the XBlock's children, use `self.get_children` which returns the IDs for each child XBlock.
- Then, to access a child XBlock, use `self.get_child(usage_id)` for your desired ID. You can then modify the child XBlock using its `.save()` method.
- To render a given child XBlock, use `self.runtime.render_child(usage_id)`.
- To render all children for a given XBlock, use `self.runtime.render_children()`.
- To ensure the XBlock children are rendered correctly, add the `fragment.content` into the parent XBlock's HTML file, then use `fragment.add_frag_resources()` (or `.add_frags_resources()`, to render all children). This ensures that the JavaScript and CSS of child elements are included.

Accessing Children (Client-Side)

To access XBlock children through the client, with JavaScript, use the following methods.

- Use `runtime.children(element)`, where `element` is the DOM node that contains the HTML representation of your XBlock's server-side view. (`runtime` is automatically provided by the XBlock runtime.)
- Similarly, you can use `runtime.childMap(element, name)` to get a child element that has a specific name.

8.6.5 XBlock Runtimes

An XBlock runtime is the application that hosts XBlock. For example, the XBlock SDK, the *Open edX LMS*, and *Open edX Studio* are all XBlock runtime applications. You can also render an individual XBlock in HTML with the XBlock URL.

Runtime Functions

An XBlock runtime application performs the following functions.

- Instantiate XBlocks with the correct data access.
- Display the HTML returned by XBlock views.

> **Note:** Runtime applications document the view names they require of XBlocks.

- Bind the front-end JavaScript code to the correct DOM elements.
- Route handler requests from the client-side XBlock to the server-side handlers.

Extending XBlocks

A runtime application can have mixin classes that it combines with your XBlock class. Therefore, your instances of your XBlock might be subclasses of your original XBlock class.

By using mixins, a runtime application can add field data and methods to all XBlocks that it hosts, without requiring that XBlocks themselves are aware of the runtime they are being hosted in.

JavaScript Runtimes

The application that runs XBlocks uses a JavaScript runtime to load XBlocks. Specifically, the JavaScript runtime provides the following functions to XBlocks.

- The Runtime Handler
- XBlock Children
- A map of the XBlock children

The XBlock SDK JavaScript Runtime

The file 1.js in the XBlock SDK provides the JavaScript runtime for the workbench.

```
// XBlock runtime implementation.

var RuntimeProvider = (function() {

var getRuntime = function(version) {
  if (! this.versions.hasOwnProperty(version)) {
    throw 'Unsupported XBlock version: ' + version;
  }
  return this.versions[version];
};

var versions = {
  1: {
    handlerUrl: function(block, handlerName, suffix, query) {
      suffix = typeof suffix !== 'undefined' ? suffix : '';
      query = typeof query !== 'undefined' ? query : '';
      var usage = $(block).data('usage');
      var url_selector = $(block).data('url_selector');
      if (url_selector !== undefined) {
        baseUrl = window[url_selector];
      }
      else {baseUrl = handlerBaseUrl;}

      // studentId and handlerBaseUrl are both defined in block.html
      return (baseUrl + usage +
                      "/" + handlerName +
                      "/" + suffix +
              "?student=" + studentId +
                      "&" + query);
    },
    children: function(block) {
      return $(block).prop('xblock_children');
    },
    childMap: function(block, childName) {
      var children = this.children(block);
      for (var i = 0; i < children.length; i++) {
        var child = children[i];
        if (child.name == childName) {
          return child
```

(continues on next page)

(continued from previous page)

```
          }
        }
      }
    }
  };

  return {
    getRuntime: getRuntime,
    versions: versions
  };
}());

var XBlock = (function () {

  var initializeBlock = function (element) {
      $(element).prop('xblock_children', initializeBlocks($(element)));

      var version = $(element).data('runtime-version');
      if (version === undefined) {
          return null;
      }

      var runtime = RuntimeProvider.getRuntime(version);
      var initFn = window[$(element).data('init')];
      var jsBlock;
      if(initFn.length == 2) {
          jsBlock = new initFn(runtime, element) || {};
      } else if (initFn.length == 3) {
          var data = $(".xblock_json_init_args", element).text();
          if (data) data = JSON.parse(data); else data = {};
          jsBlock = new initFn(runtime, element, data) || {};
      }

      jsBlock.element = element;
      jsBlock.name = $(element).data('name');
      return jsBlock;
  };

  var initializeBlocks = function (element) {
      return $(element).immediateDescendents('.xblock-v1').map(function(idx, elem) {
          return initializeBlock(elem);
      }).toArray();
  };

  return {
      initializeBlocks: initializeBlocks
  };
}());

var XBlockAsides = (function () {

  var initializeAside = function (element) {
```

(continues on next page)

(continued from previous page)

```
        var version = $(element).data('runtime-version');
        if (version === undefined) {
            return null;
        }

        var runtime = RuntimeProvider.getRuntime(version);
        var initFn = window[$(element).data('init')];
        var jsBlock;
        // $(element).siblings('div.xblock-v1')[0]
        var block_element = $(element).siblings('[data-usage="'+$(element).data('block_id
↪')+'"]')
        var data = $(".xblock_json_init_args", element).text();
        if (data) data = JSON.parse(data); else data = {};
        jsBlock = new initFn(runtime, element, block_element, data) || {};

        jsBlock.element = element;
        return jsBlock;
    };

    var initializeAsides = function (elements) {
        return elements.map(function(idx, elem) {
            return initializeAside(elem);
        }).toArray();
    };

    return {
        initializeAsides: initializeAsides
    };
}());

$(function() {
    // Find all the children of an element that match the selector, but only
    // the first instance found down any path.  For example, we'll find all
    // the ".xblock" elements below us, but not the ones that are themselves
    // contained somewhere inside ".xblock" elements.
    $.fn.immediateDescendents = function(selector) {
        return this.children().map(function(idx, element) {
            if ($(element).is(selector)) {
                return element;
            } else {
                return $(element).immediateDescendents(selector).toArray();
            }
        });
    };

    $('body').on('ajaxSend', function(elm, xhr, s) {
        // Pass along the Django-specific CSRF token.
        xhr.setRequestHeader('X-CSRFToken', $.cookie('csrftoken'));
    });

    XBlock.initializeBlocks($('body'));
    XBlockAsides.initializeAsides($('.xblock_asides-v1'))
```

(continues on next page)

(continued from previous page)

```
});
```

The JavaScript Runtime Handler

The JavaScript runtime initializes the XBlock each time it is loaded by a user and returns the handler so the XBlock can communicate with the server.

From the example above, the following part of the runtime generates and returns the handler to the XBlock.

```
var versions = {
  1: {
    handlerUrl: function(block, handlerName, suffix, query) {
      suffix = typeof suffix !== 'undefined' ? suffix : '';
      query = typeof query !== 'undefined' ? query : '';
      var usage = $(block).data('usage');
      var url_selector = $(block).data('url_selector');
      if (url_selector !== undefined) {
        baseUrl = window[url_selector];
      }
      else {baseUrl = handlerBaseUrl;}

      // studentId and handlerBaseUrl are both defined in block.html
      return (baseUrl + usage +
                        "/" + handlerName +
                        "/" + suffix +
                "?student=" + studentId +
                        "&" + query);
    . . .
```

The runtime handler code is called by the XBlock's JavaScript code to get the XBlock URL.

For example, the Thumbs XBlock in the XBlock SDK, the thumbs.js file gets the handler from the XBlock runtime.

```
var handlerUrl = runtime.handlerUrl(element, 'vote');
```

XBlock Children

The JavaScript runtime also returns the list of child XBlocks to the XBlock.

From the example above, the following part of the runtime returns the list of children to the XBlock.

```
. . .

children: function(block) {
  return $(block).prop('xblock_children');
},
. . .
```

An XBlock uses the `children` method when it needs to iterate over an ordered list of its child XBlocks.

XBlock Child Map

The JavaScript runtime also returns the a map of child XBlocks to the running XBlock.

From the example above, the following part of the runtime generates and returns the list of children to the XBlock.

```
. . .

childMap: function(block, childName) {
    var children = this.children(block);
    for (var i = 0; i < children.length; i++) {
        var child = children[i];
        if (child.name == childName) {
            return child
        }
    }
}
. . .
```

An XBlock uses the `childMap` function when it needs to access different child XBlocks to perform different actions on them.

For example, the Problem XBlock in the XBlock SDK loads JavaScript code that gets the map of child XBlocks.

```
function handleCheckResults(results) {
$.each(results.submitResults || {}, function(input, result) {
     callIfExists(runtime.childMap(element, input), 'handleSubmit', result);
 });
$.each(results.checkResults || {}, function(checker, result) {
    callIfExists(runtime.childMap(element, checker), 'handleCheck', result);
```

XBlock Runtime API

For more information, see XBlock Runtime API in the *XBlock API Guide*.

Rendering XBlocks with the XBlock URL

The XBlock URL supports HTML rendering of an individual XBlock without the user interface of the LMS.

To use the XBlock URL and return the HTML rendering of an individual XBlock, you use the following URL path for an XBlock on an edX site.

`https://{host}/xblock/{usage_id}`

Finding the `usage_id`

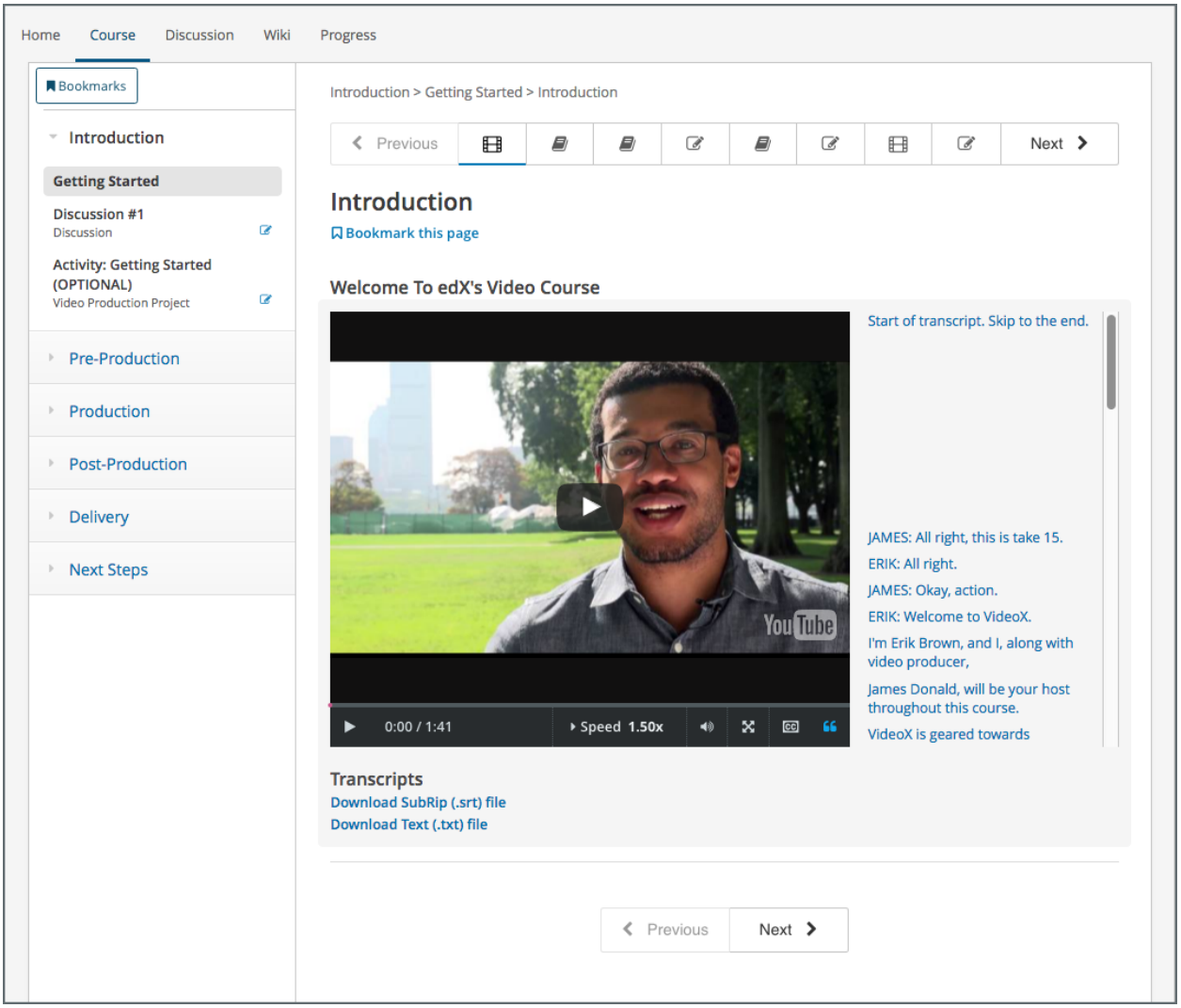
The `usage_id` is the unique identifier for the problem, video, text, or other course content component, or for sequential or vertical course container component. There are several ways to find the `usage_id` for an XBlock in the LMS, including viewing either the staff debug info or the page source. For more information, see Finding the Usage ID for Course Content.

Example XBlock URLs

For example, a video component in the "Creating Video for the edX Platform" course on the edx.org site has the following URL.

`https://courses.edx.org/courses/course-v1:edX+VideoX+1T2016/courseware/ccc7c32c65d342618ac76409254ac238/1a52e689bcec4a9eb9b7da0bf16f682d/`

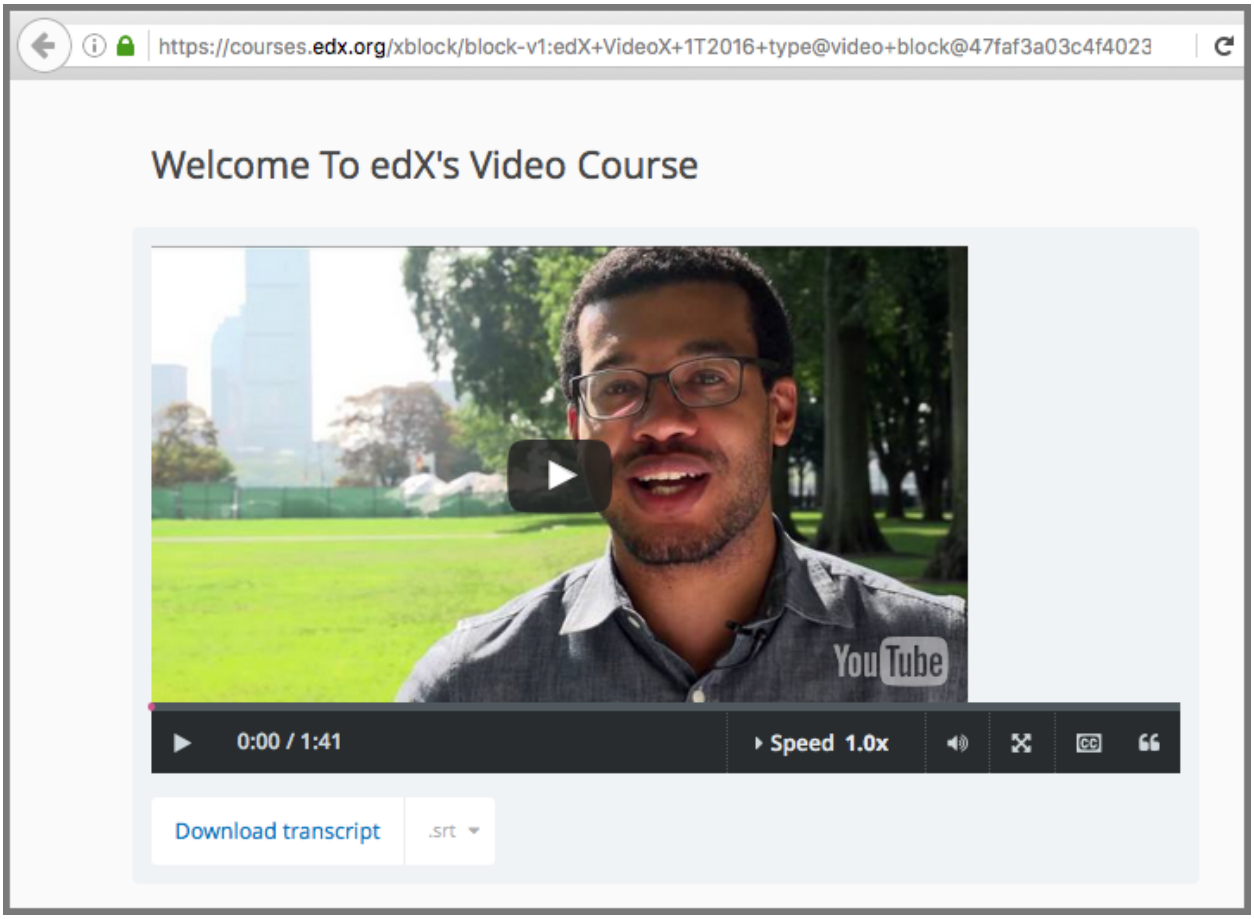
This video component appears as follows in the LMS.



To construct the XBlock URL for the same video component, you obtain its `usage_id` and then use the following URL format.

`https://courses.edx.org/xblock/block-v1:edX+VideoX+1T2016+type@video+block@47faf3a03c4f4023b187528c2593`

When you use this URL, the video component appears in your browser as follows.



For courses created prior to October 2014, the `usage_id` begins with `i4x://`, as in the following example.

`https://courses.edx.org/xblock/i4x://edX/DemoX.1/problem/47bf6dbce8374b789e3ebdefd74db332`

8.6.6 XBlocks, Events, and Grading

Events are emitted by the server or the browser to capture information about interactions with the courseware.

In most cases, your XBlock must emit events.

For example, *assigning a grade* is a common event.

- *When an XBlock Should Emit Events*
- *Publish Events in Handler Methods*
- *Publish Grade Events*

When an XBlock Should Emit Events

Analysis of events can provide insight about how learners use the XBlock. Using event data, analysts should be able to reconstruct the state of the XBlock at any point in time.

Your XBlock should emit an event whenever a significant state change occurs, and when a grade for the learner's interaction is assigned. For example, when a learner submits an answer or otherwise interacts with your XBlock, an event should record that action.

To assign grades from your XBlock, it must emit a *grade event*.

Publish Events in Handler Methods

You define *handler methods* to emit events.

In the handler, you use the XBlock runtime interface `publish` method to emit the event. The `runtime.publish` method causes the runtime application to save the event data in the application event stream.

The following code shows the `runtime.publish` method syntax in an XBlock handler.

```
self.runtime.publish(self, "event_type",
                     { event_dictionary })
```

Note the following information about the `runtime.publish` method.

- The `event_type` uniquely identifies the event in log files.
- The event dictionary contains key-value pairs that define the event.

Publish Grade Events

To assign a grade for a learner's interaction with the XBlock, the XBlock handler method must publish a grade event.

A grade event uses the `runtime.publish` method with specific arguments.

- The event type is `grade`.
- The event dictionary must contain two entries.
  - `value`: The learner's score
  - `max_value`: The maximum possible score

The current user's `user_id` is implicit in the event dictionary.

..The event dictionary can also contain the `user_id` entry. If `user_id` is not specified, the current user's ID is used.

For example, the following handler code emits a grade for the learner that is stored in the `submission_result` variable in an XBlock with the maximum grade of `1.0`.

```
self.runtime.publish(self, "grade",
                     { value: submission_result
                       max_value: 1.0 })
```

Typically, the handler method also returns the calculated grade, so that it can be displayed to the learner.

has_score Variable

To be graded, in addition to publishing the grade event, the XBlock must also have a `has_score` variable set to `True`.

```
has_score = True
```

8.7 XBlocks and the edX Platform

8.7.1 Open edX Studio as an XBlock Runtime

Open edX Studio is the application in the Open edX Platform that instructors use to build courseware.

Because instructors use Studio to add and configure XBlocks, Studio is also an *XBlock runtime* application.

As an XBlock developer, you must understand what XBlock properties Studio requires.

Studio Requirements for XBlocks

Studio requires XBlocks to have the following properties.

- A *view method* named `studio_view`. This is the view that renders the XBlock in the Studio editor, allowing the instructor to configure it. In Studio, the instructor accesses this view by selecting **Edit** in the component.
- A view method named `author_view`. This view is used to display the XBlock in the Studio preview mode.

  The `author_view` method should be as close as possible to the LMS `student_view`, but may contain inline editing capabilities.

  If you do not define an `author_view`, the preview mode uses the `student_view`. For more information, see *Open edX Learning Management System as an XBlock Runtime*.
- A class property named `non_editable_metadata_fields`. This variable contains a list of the XBlock fields that should not be displayed in the Studio editor.

8.7.2 Open edX Learning Management System as an XBlock Runtime

The Open edX Learning Management System (LMS) is the application in the Open edX Platform that learners use to view and interact with courseware.

Because it presents XBlocks to learners and records their interactions, the LMS is also an *XBlock runtime* application.

As an XBlock developer, you must understand what XBlock properties the LMS requires.

LMS Requirements for XBlocks

The LMS requires XBlocks to have the following properties.

- A *view method* named `student_view`. This is the view that renders the XBlock in the LMS for learners to see and interact with.

  In addition, the `student_view` method is used to render the XBlock in the Studio preview mode, unless the XBlock also has an `author_view` method. For more information, see *Open edX Studio as an XBlock Runtime*.

- A class property named `has_score` with a value of `True` if the XBlock is to be graded.
- A class property named `icon_class`, which controls the icon that displays to learners in the unit navigation bar on the **Course** page when the XBlock is in that unit. The variable must have one of the following values.

Value	Icon
`problem`	
`video`	
`other`	

Internationalization Support

The LMS is currently capable of supporting internationalization (i18n) and localization (l10n) of static UI text included in your XBlock – also known as "chrome" or "labels". Translation of user-generated content stored as XBlock state is not currently supported.

To present XBlock language translations in the LMS you must include the translated strings for your chosen "locale" in the GNU Gettext Portable Object file format. Translated strings must be stored in a "domain" file named "text.po".

- locale: A set of parameters that defines the user's language, region and any special variant preferences that the user wants to see in their user interface
- domain: A Gettext application representing the set of translated strings corresponding to a particular locale.

Each "text.po" domain file consists of one or more string/translation pairs for the language/locale. Further, each translation pair consists of two fields: "msgid" for the base string, and "msgstr" for its corresponding translation.

There is no limit on the number of locales/domains that can be included with your XBlock. However, your specific Open edX installation may not be configured to support every locale that you provide.

You can learn more about the GNU Gettext Portable Object file format and download the GNU Gettext software using the following resources:

- https://www.gnu.org/software/gettext/
- https://en.wikipedia.org/wiki/Gettext
- https://www.drupal.org/node/1814954

In addition to GNU Gettext, it is also possible to utilize the Open edX "i18n-tools" GNU Gettext wrapper to work with your XBlock locales and domains. You will need to modify the i18n-tools YAML configuration file to work with your XBlock project. More information about the i18n-tools project and its configuration file can be found at:

- https://github.com/openedx/i18n-tools

- https://github.com/openedx/i18n-tools/blob/master/conf/locale/config.yaml

Adding Translated Strings to your XBlock

1. Create a directory within your XBlock code project named "translations". This directory should be located at the same level in your code project as your XBlock implementation file. For example:
   - `http://github.com/my_org/my_xblock/my_xblock/my_xblock.py`
   - `http://github.com/my_org/my_xblock/my_xblock/translations/`
2. Create a set of language directories for each of your locales within this new "translations" directory. You may specify either a general language code or a specific language locale code for the name of each directory. Include an "LC_MESSAGES" directory with each language directory.
   - `../my_xblock/translations/ar/LC_MESSAGES/`
   - `../my_xblock/translations/es-es/LC_MESSAGES/`
3. Create a domain file named "text.po". You can use the Gettext `xgettext` command directly, or another tool of your choosing, such as Django's `makemessages` utility, or i18n-tools. For more information on how to use these tools, see the following resources.
   - Gettext: https://www.gnu.org/software/gettext/manual/gettext.html
   - Gettext: http://phptal.org/manual/en/split/gettext.html
   - Django: https://docs.djangoproject.com/en/dev/topics/i18n/translation/#localization-how-to-create-language-files
   - i18n-tools: https://github.com/openedx/i18n-tools
4. Repeat the domain file creation process for each language/locale you support.

   In the following example, we will use the i18n-tools utilites to generate a "text.po" file.

   1. Create an alternative configuration file containing the details for your particular XBlock project
   2. Run `i18n_tool extract` to automatically find strings and populate the PO file.
   3. Run `i18n_tool generate` to compile your human-readable PO file to a machine-readable "MO" binary file
   4. Repeat the extraction/generation process for as many languages/locales as you require for your XBlock
   5. Add all of your translation directories and PO/MO files to your XBlock code project for distribution
5. Open each "text.po" domain file and, for each "msgid" string, add a corresponding "msgstr" translation. PO files can be edited by hand, with a tool such as Pedit or Emacs, or through a third party service such as Transifex.
6. Place each locale's "text.po" domain file within the corresponding "LC_MESSAGES" directory.
   - `../my_xblock/translations/ar/LC_MESSAGES/text.po`
   - `../my_xblock/translations/es-es/LC_MESSAGES/text.po`
7. Compile your "text.po" files into binary "text.mo" files using the Gettext `msgfmt` command (or via the tool of your choice), and include these "text.mo" files alongside your "text.po" files in your code project.
   - `../my_xblock/translations/ar/LC_MESSAGES/text.mo`
   - `../my_xblock/translations/ar/LC_MESSAGES/text.po`

The resulting directory/file structure should look like this.

```
/my_xblock
├── my_xblock.py
└── translations
    ├── ar
    |   └── LC_MESSAGES
    |       ├── text.mo
    |       └── text.po
    ├── es-es
    |   └── LC_MESSAGES
    |       ├── text.mo
    |       └── text.po
    ├── ru
    |   └── LC_MESSAGES
    |       ├── text.mo
    |       └── text.po
    └── zh-cn
        └── LC_MESSAGES
            ├── text.mo
            └── text.po
```

You can now run the LMS and update your preferred language via Account Settings in order to observe the translated strings for your chosen locale.

Note: In the absence of an available language locale and domain file, the LMS XBlock runtime will attempt to match strings marked for translation within your XBlock using its own set of language locales and domains. However, it is not recommended that you rely on the LMS mechanism for internationalization support. There is no guarantee your strings will be matched by the LMS, and even if matches are found, the translations may be incorrect in the context of your specific XBlock.

8.7.3 Deploy Your XBlock in Devstack

This section provides instructions for deploying your XBlock in devstack.

For more information about devstack, see the Installing, Configuring, and Running the Open edX Platform.

Prerequisites

Before proceeding with the steps to deploy your XBlock, ensure the following requirements are met.

- Devstack is running. For instructions, see the devstack repository.
- Ensure you have the XBlock directory in a location you can access from the devstack containers (e.g. *edx-platform/src/*`).

Installing the XBlock

The following instructions will help you install a XBlock on your OpenEdX devstack. Since LMS and Studio run on separate Docker containers, you will need to install the XBlock to the virtual environments of both containers.

Note: These steps consider you're running the Docker based Devstack provisioned at `~/devstack_workspace/`.

---

1. From your devstack folder (`~/devstack_workspace/devstack`), enter the LMS container shell:

   ```
   $ make lms-shell
   ```

2. Install the XBlock on `edx-platform` virtual enviroment:

   ```
   root@7beb9df53150:/edx/app/edxapp/edx-platform# pip install path/to/xblock
   ```

3. Use `C-d` to exit the LMS shell and enter Studio shell with:

   ```
   $ make studio-shell
   ```

4. Install the XBlock in the same way you've installed it on LMS:

   ```
   root@7beb9df53150:/edx/app/edxapp/edx-platform# pip install path/to/xblock
   ```

5. To make sure the XBlock is available, you will need to restart both LMS and Studio:

   ```
   $ make lms-restart && make studio-restart
   ```

After this, you'll be able to enable and add the XBlock to your course.

Enable the XBlock in Your Course

To use a XBlock, you must enable it in each course in which you intend to use it.

1. Log in to Studio.
2. Open the course.
3. From the **Settings** menu, select **Advanced Settings**.
4. In the **Advanced Module List** field, place your cursor between the braces, and then type the exact name of the XBlock.

   Note: The name you enter must match exactly the name specified in your XBlock's `setup.py` file.

   If you see other values in the **Advanced Module List** field, add a comma after the closing quotation mark for the last value, and then type the name of your XBlock.

5. At the bottom of the page, select **Save Changes**.

Add an Instance of the XBlock to a Unit

You can add instances of the XBlock in any unit in the course.

On the unit page, under **Add New Component**, select **Advanced**.

Your XBlock is listed as one of the types you can add.

Select the name of your XBlock to add an instance to the unit.

You can then edit the properties of the instance as needed by selecting the **Edit** button.

For more information about working with components in Studio, see Developing Course Components in the *Building and Running an Open edX* guide.

8.7.4 Submit Your XBlock to edX

Many developers and institutions submit the XBlocks they develop to edX, to benefit course teams and learners who create and take classes on edx.org.

Note that you are not required to submit your XBlock to edX. You and other edX service providers can run your XBlock without involving edX.

To submit your XBlock to edx.org, complete the following steps.

1. Upload the XBlock to a repository on GitHub.
2. Create a new branch in the edx-platform GitHub repository.
3. In your branch, add a line to the requirements/edx/github.txt file that indicates the version of your XBlock to use.

   Note: The requirements file addition is the only change you should make in your branch. Do not include the code for your XBlock in the pull request.

4. Create a pull request for your branch in the edx-platform GitHub repository.
5. Add a thorough description of your XBlock and its intended use to the pull request. You must include instructions to manually test that the XBlock is working properly.
6. Add a link to your XBlock repository in the pull request.

After you submit the pull request, edX will review your XBlock to ensure that it is appropriate for use on edx.org. Specifically, edX will review your XBlock for security, scalability, accessibility, and fitness of purpose. You should be prepared to respond to questions and comments from edX in your pull request.

8.8 Open edX Glossary

Glossary

8.9 Appendices

8.9.1 Using XBlock Software Development Kit

The XBlock SDK is a Python application you use to help you build new XBlocks. The XBlock SDK contains three main components:

- An XBlock creation tool that builds the skeleton of a new XBlock.
- An XBlock runtime for viewing and testing your XBlocks during development.
- Sample XBlocks that you can use as the starting point for new XBlocks, and for your own learning.

In *Build an XBlock: Quick Start*, you *set up the XBlock Software Development Kit (SDK)*. You had to do this to *create your first XBlock*.

While covering some of the same topics, this part of the tutorial is included as a later reference for using the XBlock SDK.

Getting Started with the XBlock SDK

This section describes how to get started with the XBlock SDK.

Clone the XBlock Software Development Kit

The XBlock SDK is a Python application you use to help you build new XBlocks. The XBlock SDK contains three main components:

- An XBlock creation tool that builds the skeleton of a new XBlock.
- An XBlock runtime for viewing and testing your XBlocks during development.
- Sample XBlocks that you can use as the starting point for new XBlocks, and for your own learning.

After you *create and activate the virtual environment*, you clone the XBlock SDK and install its requirements. To do this, complete the following steps at a command prompt.

1. In the `xblock_development` directory, run the following command to clone the XBlock SDK repository from GitHub.

```
(xblock-env) $ git clone https://github.com/openedx/xblock-sdk.git
```

2. In the same directory, create an empty directory called *var*.

```
(xblock-env) $ mkdir var
```

3. Run the following command to change to the `xblock-sdk` directory.

```
(xblock-env) $ cd xblock-sdk
```

4. Run the following commands to install the XBlock SDK requirements.

```
(xblock-env) $ make install
```

5. Run the following command to return to the `xblock_development` directory, where you will perform the rest of your work.

```
(xblock-env) $ cd ..
```

Create an XBlock

You use the XBlock SDK to create skeleton files for an XBlock. To do this, follow these steps at a command prompt.

1. Change to the `xblock_development` directory, which contains the `var`, `xblock-env`, and `xblock-sdk` subdirectories.
2. Run the following command to create the skeleton files for the XBlock.

```
(xblock-env) $ xblock-sdk/bin/workbench-make-xblock
```

   Instructions in the command window instruct you to determine a short name and a class name. Follow the guidelines in the command window to determine the names that you want to use.

```
You will be prompted for two pieces of information:

* Short name: a single word, all lower-case, for directory and file
  names. For a hologram 3-D XBlock, you might choose "holo3d".

* Class name: a valid Python class name.  It's best if this ends with
  "XBlock", so for our hologram XBlock, you might choose
  "Hologram3dXBlock".

Once you specify those two names, a directory is created in the
``xblock_development`` directory containing the new project.

If you don't want to create the project here, or you enter a name
incorrectly, type Ctrl-C to stop the creation script.  If you don't want
the resulting project, delete the directory it created.
```

3. At the command prompt, enter the Short Name you selected for your XBlock.

```
$ Short name: myxblock
```

4. At the command prompt, enter the Class name you selected for your XBlock.

```
$ Class name: MyXBlock
```

The skeleton files for the XBlock are created in the `myxblock` directory. For more information about the XBlock files, see *Anatomy of an XBlock*.

Install the XBlock

After you create the XBlock, you install it in the XBlock SDK.

In the `xblock_development` directory, use `pip` to install your XBlock.

```
(xblock-env) $ pip install -e myxblock
```

You can then test your XBlock in the XBlock SDK.

Create the SQLite Database

Before running the XBlock SDK the first time, you must create the SQLite database.

1. In the `xblock_development` directory, run the following command to create the database and the tables.

```
(xblock-env) $ python xblock-sdk/manage.py migrate
```

Run the XBlock SDK Server

To see the web interface of the XBlock SDK, you must run the SDK server.

In the `xblock_development` directory, run the following command to start the server.

```
(xblock-env) $ python xblock-sdk/manage.py runserver
```

Note: If you do not specify a port, the XBlock SDK server uses port 8000. To use a different port, specify it in the `runserver` command.

Then test that the XBlock SDK is running. In a browser, go to `http://localhost:8000`. You should see the following page.

The page shows the XBlocks installed automatically with the XBlock SDK. Note that the page also shows the **MyXBlock** XBlock that you created in *Create Your First XBlock*.

Get Help for the XBlock SDK Server

To get help for the XBlock SDK `runserver` command, run the following command.

```
(xblock-env) $ python xblock-sdk/manage.py help
```

The command window lists and describes the available commands.

CHAPTER

NINE

XBLOCK.UTILS

9.1 Package having various utilities for XBlocks

9.1.1 Purpose

`xblock/utils` package contains a collection of utility functions and base test classes that are useful for any XBlock.

9.1.2 Documentation

StudioEditableXBlockMixin

```
from xblock.utils.studio_editable import StudioEditableXBlockMixin
```

This mixin will automatically generate a working `studio_view` form that allows content authors to edit the fields of your XBlock. To use, simply add the class to your base class list, and add a new class field called `editable_fields`, set to a tuple of the names of the fields you want your user to be able to edit.

```
@XBlock.needs("i18n")
class ExampleBlock(StudioEditableXBlockMixin, XBlock):
    ...
    mode = String(
        display_name="Mode",
        help="Determines the behaviour of this component. Standard is recommended.",
        default='standard',
        scope=Scope.content,
        values=('standard', 'crazy')
    )
    editable_fields = ('mode', 'display_name')
```

That's all you need to do. The mixin will read the optional `display_name`, `help`, `default`, and `values` settings from the fields you mention and build the editor form as well as an AJAX save handler.

If you want to validate the data, you can override `validate_field_data(self, validation, data)` and/or `clean_studio_edits(self, data)` - see the source code for details.

Supported field types:

- Boolean: `field_name = Boolean(display_name="Field Name")`
- Float: `field_name = Float(display_name="Field Name")`
- Integer: `field_name = Integer(display_name="Field Name")`

- String: `field_name = String(display_name="Field Name")`
- String (multiline): `field_name = String(multiline_editor=True, resettable_editor=False)`
- String (html): `field_name = String(multiline_editor='html', resettable_editor=False)`

Any of the above will use a dropdown menu if they have a pre-defined list of possible values.

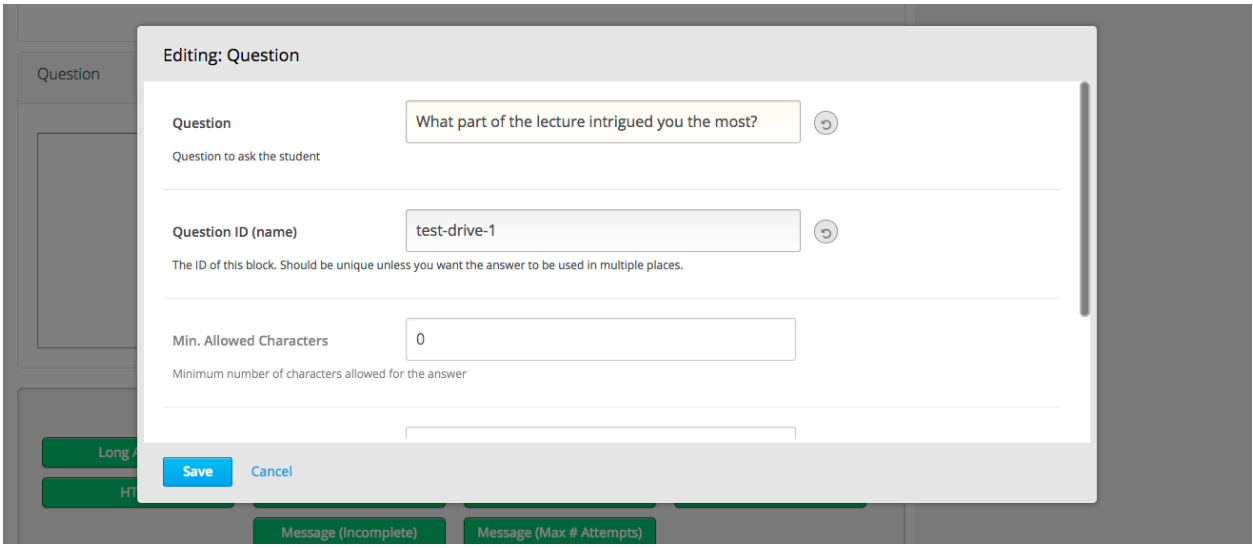
- List of unordered unique values (i.e. sets) drawn from a small set of possible values: `field_name = List(list_style='set', list_values_provider=some_method)`
  - The `List` declaration must include the property `list_style='set'` to indicate that the `List` field is being used with set semantics.
  - The `List` declaration must also define a `list_values_provider` method which will be called with the block as its only parameter and which must return a list of possible values.
- Rudimentary support for Dict, ordered List, and any other JSONField-derived field types
  - `list_field = List(display_name="Ordered List", default=[])`
  - `dict_field = Dict(display_name="Normal Dict", default={})`

Supported field options (all field types):

- `values` can define a list of possible options, changing the UI element to a select box. Values can be set to any of the formats defined in the XBlock source code:
  - 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}`
  - A callable that returns one of the above. (Note: the callable does *not* get passed the XBlock instance or runtime, so it cannot be a normal member function)
- `values_provider` can define a callable that accepts the XBlock instance as an argument, and returns a list of possible values in one of the formats listed above.
- `resettable_editor` - defaults to `True`. Set `False` to hide the "Reset" button used to return a field to its default value by removing the field's value from the XBlock instance.

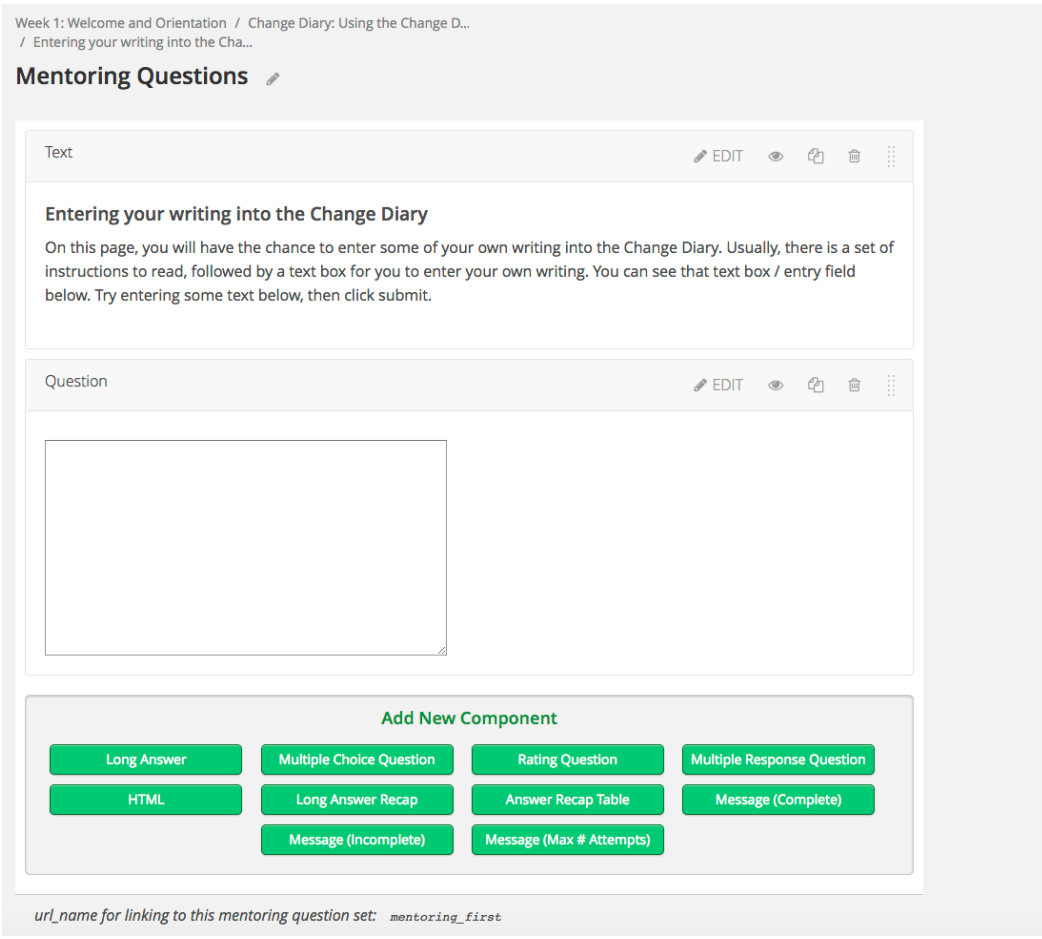
Basic screenshot: 

StudioContainerXBlockMixin

```
from xblock.utils.studio_editable import StudioContainerXBlockMixin
```

This mixin helps to create XBlocks that allow content authors to add, remove, or reorder child blocks. By removing any existing `author_view` and adding this mixin, you'll get editable, re-orderable, and deletable child support in Studio. To enable authors to add arbitrary blocks as children, simply override `author_edit_view` and set `can_add=True` when calling `render_children` - see the source code. To restrict authors so they can add only specific types of child blocks or a limited number of children requires custom HTML.

An example is the mentoring XBlock:



child_isinstance

```
from xblock.utils.helpers import child_isinstance
```

If your XBlock needs to find children/descendants of a particular class/mixin, you should use

```
child_isinstance(self, child_usage_id, SomeXBlockClassOrMixin)
```

rather than calling

```
isinstance(self.runtime.get_block(child_usage_id), SomeXBlockClassOrMixin)
```

On runtimes such as those in edx-platform, `child_isinstance` is orders of magnitude faster.

XBlockWithSettingsMixin

This mixin provides access to instance-wide XBlock-specific configuration settings. See *Accessing XBlock specific settings* for details.

ThemableXBlockMixin

This mixin provides XBlock theming capabilities built on top of XBlock-specific settings. See *Theming support* for details.

To learn more, refer to the page.

Settings and theme support

Accessing XBlock specific settings

XBlock utils provide a mixin to simplify accessing instance-wide XBlock-specific configuration settings: `XBlockWithSettingsMixin`. This mixin aims to provide a common interface for pulling XBlock settings from the LMS SettingsService.

`SettingsService` allows individual XBlocks to access environment and django settings in an isolated manner:

- XBlock settings are represented as dictionary stored in django settings and populated from environment *.json files (cms.env.json and lms.env.json)
- Each XBlock is associated with a particular key in that dictionary: by default an XBlock's class name is used, but XBlocks can override it using the `block_settings_key` attribute/property.

Please note that at the time of writing the implementation of `SettingsService` assumed "good citizenship" behavior on the part of XBlocks, i.e. it does not check for key collisions and allows modifying mutable settings. Both `SettingsService` and `XBlockWithSettingsMixin` are not concerned with contents of settings bucket and return them as is. Refer to the `SettingsService` docstring and implementation for more details.

Using XBlockWithSettingsMixin

In order to use `SettingsService` and `XBlockWithSettingsMixin`, a client XBlock *must* require it via standard `XBlock.wants('settings')` or `XBlock.needs('settings')` decorators. The mixins themselves are not decorated as this would not result in all descendant XBlocks to also be decorated.

With `XBlockWithSettingsMixin` and `wants` decorator applied, obtaining XBlock settings is as simple as

```
self.get_xblock_settings()  # returns settings bucket or None
self.get_xblock_settings(default=something)  # returns settings bucket or "something"
```

In case of missing or inaccessible XBlock settings (i.e. no settings service in runtime, no `XBLOCK_SETTINGS` in settings, or XBlock settings key is not found) `default` value is used.

Theming support

XBlock theming support is built on top of XBlock-specific settings. XBlock utils provide `ThemableXBlockMixin` to streamline using XBlock themes.

XBlock theme support is designed with two major design goals:

- Allow for a different look and feel of an XBlock in different environments.
- Use a pluggable approach to hosting themes, so that adding a new theme will not require forking an XBlock.

The first goal made using `SettingsService` and `XBlockWithSettingsMixin` an obvious choice to store and obtain theme configuration. The second goal dictated the configuration format - it is a dictionary (or dictionary-like object) with the following keys:

- `package` - "top-level" selector specifying package which hosts theme files
- `locations` - a list of locations within that package

Examples:

```
# will search for files red.css and small.css in my_xblock package
{
    'package': 'my_xblock',
    'locations': ['red.css', 'small.css']
}

# will search for files public/themes/red.css in my_other_xblock.assets package
default_theme_config = {
    'package': 'my_other_xblock.assets',
    'locations': ['public/themes/red.css']
}
```

Theme files must be included into package (see python docs for details). At the time of writing it is not possible to fetch theme files from multiple packages.

Note: XBlock themes are *not* LMS themes - they are just additional CSS files included into an XBlock fragment when the corresponding XBlock is rendered. However, it is possible to misuse this feature to change look and feel of the entire LMS, as contents of CSS files are not checked and might contain selectors that apply to elements outside of the XBlock in question. Hence, it is advised to scope all CSS rules belonging to a theme with a global CSS selector `.themed-xblock.<root xblock element class>`, e.g. `.themed-xblock.poll-block`. Note that the `themed-xblock` class is not automatically added by `ThemableXBlockMixin`, so one needs to add it manually.

Using ThemableXBlockMixin

In order to use `ThemableXBlockMixin`, a descendant XBlock must also be a descendant of `XBlockWithSettingsMixin` (`XBlock.wants` decorator requirement applies) or provide a similar interface for obtaining the XBlock settings bucket.

There are three configuration parameters that govern `ThemableXBlockMixin` behavior:

- `default_theme_config` - default theme configuration in case no theme configuration can be obtained
- `theme_key` - a key in XBlock settings bucket that stores theme configuration
- `block_settings_key` - inherited from `XBlockWithSettingsMixin` if used in conjunction with it

It is safe to omit `default_theme_config` or set it to `None` in case no default theme is available. In this case, `ThemableXBlockMixin` will skip including theme files if no theme is specified via settings.

`ThemableXBlockMixin` exposes two methods:

- `get_theme()` - this is used to get theme configuration. Default implementation uses `get_xblock_settings` and `theme_key`, descendants are free to override it. Normally, it should not be called directly.
- `include_theme_files(fragment)` - this method is an entry point to `ThemableXBlockMixin` functionality. It calls `get_theme` to obtain theme configuration, fetches theme files and includes them into fragment. `fragment` must be a web_fragments.fragment instance.

So, having met usage requirements and set up theme configuration parameters, including theme into XBlock fragment is a one liner:

```
self.include_theme_files(fragment)
```

PYTHON MODULE INDEX

X

INDEX

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X