
XBlock Documentation

Release 0.1

edX.org

Jun 05, 2018

Contents

1	Concepts	3
1.1	Roles	3
1.2	XBlocks	3
1.3	Runtime	4
2	XBlocks	5
2.1	Python Structure	5
2.2	State	5
2.3	Children	6
2.4	Methods	6
2.5	Views	7
2.6	Handlers	7
2.7	Querying	7
2.8	Tags	7
3	Fragment	9
4	API	11
4.1	Fragment	11
4.2	Runtime	13
4.3	XBlock	13
	Python Module Index	15

Note: This is extremely preliminary documentation. Please get in touch with us if you have any questions or concerns. Do not make any plans based on this document without talking to us first.

The ideas here are our current best guess at how XBlocks will work, but some parts are more settled than others.

To create rich engaging online courses, course authors must be able to combine components from a variety of sources. XBlock is edX's component architecture that makes this possible. Courses are built hierarchically of pieces called **XBlocks**. Like an HTML `<div>`, XBlocks can represent pieces 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.

XBlocks are not limited to just delivering courses. A complete educational ecosystem will make use of a number of web applications, all of which will need access to course content and data. XBlocks provide the structure and APIs needed to build components for use in all of these applications.

Contents:

XBlock is a component architecture for building courseware. They are similar in structure to web applications.

1.1 Roles

The XBlock design recognizes a few different roles people can play:

Block Developer

An XBlock Developer is the author of an XBlock type. This is a Python developer writing Python classes to implement a new kind of XBlock.

Content Author

Original course material is written by a Content Author. This material may be made available to others to use in their own courses.

Course Assembler

A Course Assembler creates or modifies courses by using content created by someone else. Note that the same person can act as a content author and as a course assembler, often within the same authoring session.

Student

The Student (or User) is whoever uses the web applications composed of XBlocks.

1.2 XBlocks

XBlocks are components that combine together to create interactive course content. They need to satisfy two conflicting goals: work together with other blocks to build a complete course; and be independent of other blocks, so they can be combined flexibly.

XBlocks are built similarly to web applications. They maintain state in a storage layer, render themselves through views, and process user actions through handlers.

They differ from web applications, though, because each XBlock renders only a small piece of a complete web page.

1.3 Runtime

XBlocks do not run by themselves, they run within web applications such as Studio, or LMS, known as runtimes. Each runtime provides services to the XBlock, such as storage, URL mapping, and analytics.

Runtimes will differ in the context they provide to XBlocks. For example, while editing content, Studio won't provide user state, because there is no interesting user state. Another runtime might provide user state, but as read-only data.

Runtimes also differ in what views they make use of. Studio might use “edit” to edit the XBlock content, and “read” to preview that content, while the LMS might only use the “read” view to render the XBlock for students. Each runtime is free to define view names it will use for its purposes. XBlock Developers need to understand the runtimes they will be running in to write the proper views.

Runtimes are responsible for performing any authentication needed before executing a view or handler in an XBlock.

Examples of runtimes:

- Studio
- LMS
- XBlock debugging dashboard
- Peer grading workflow app

XBlocks are Python classes that implement a small web application. Like full applications, they have state and methods, and operate on both the server and the client.

2.1 Python Structure

XBlocks are implemented as Python code, and packaged using standard Python techniques. They have Python code, and other file resources, including CSS and Javascript needed to fully render themselves in a browser.

2.2 State

XBlock state (or data) is arbitrary JSON-able data. XBlock state can be scoped on several axes:

- By User: State scoped by user is different for every user in the system.
- By XBlock: State scoped by XBlock can be scoped by various aspects of the XBlock:
 - block usage - the instance of the XBlock in a particular course
 - block definition - the definition of an XBlock created by a content creator (potentially shared across runs of a course)
 - block type - the Python type of the XBlock (shared across all instances of the XBlock in all courses)
 - all - all XBlocks share the same data

For example:

- A user's progress through a particular set of problems would be stored in a User=True, XBlock=Usage scope.
- The content to display in an XBlock would be stored in a User=False, XBlock=Definition scope.
- A user's preferences for a type of XBlock, such as the preferences for a circuit editor, would be stored in a User=True, XBlock=Type scope.

- Information about the user, such as language or timezone, would be stored in a User=True, XBlock=All scope.

XBlocks declare their data with a schema in the XBlock class definition. The schema defines a series of properties, each of which has at least a name, a type, and a scope:

```
upvotes = Int(help="Number of up votes", default=0, scope=Scope(user=False,
↪module=DEFINITION))
downvotes = Int(help="Number of down votes", default=0, scope=Scope(user=False,
↪module=DEFINITION))
voted = Boolean(help="Whether a student has already voted", default=False,
↪scope=Scope(user=True, module=USAGE))
```

For convenience, we also provide predefined scopes: `Scope.content`, `Scope.user_state`, `Scope.preferences`, and `Scope.user_info`.

In XBlock code, state is accessed as attributes on `self`. In our example above, the data is available as `self.upvotes`, `self.downvotes`, and `self.voted`. The data is automatically scoped for the current user and block. Modifications to the attributes are persisted implicitly, there is no `save()` method. The runtime is free to provide these attributes however it likes. For example, it could pre-load the data from a database, or proxy the attributes to load them lazily. It could provide explicitly stored data, or it could provide calculated values as it sees fit.

2.3 Children

In contrast to the conceptual view of XBlocks, an XBlock does not refer directly to its children. Instead, the structure of a tree of XBlocks is maintained by the runtime, and is made available to the XBlock through a runtime service.

This allows the runtime to store, access, and modify the structure of a course without incurring the overhead of the XBlock code itself. The children will not be implicitly available. The runtime will provide a list of child ids, and a child can be loaded with a `get_child()` function call. This means the runtime can defer loading children until they are actually required (if ever).

2.4 Methods

The behavior of an XBlock is determined by its methods, which come in a few categories:

- Views: These are invoked by the runtime to render the XBlock. There can be any number of these, written as ordinary Python methods. Each view has a specific name, such as “edit” or “read”, specified by the runtime that will invoke it.

A typical use of a view is to produce a *fragment* for rendering the block as part of a web page. The user state, settings, and preferences may be used to affect the output in any way the XBlock likes. Views can indicate what data they rely on, to aid in caching their output.

Although views typically produce HTML-based renderings, they can be used for anything the runtime wants. The runtime description of each view should be clear about what return type is expected and how it will be used.

- Handlers: Handlers provide server-side logic invoked by AJAX calls from the browser. There can be any number of these, written as ordinary Python methods. Each handler has a specific name of your choice, such as “submit” or “preview.” The runtime provides a mapping from handler names to actual URLs so that XBlock Javascript code can make requests to its handlers. Handlers can be used with GET requests as well as POST requests.
- Recalculators: (not a great word!) There can be any number of these, written as ordinary Python methods. Each has a specific name, and is invoked by the runtime when a particular kind of recalculation needs to be done. An example is “regrade”, run when a TA needs to adjust a problem, and all the students’ inputs should be checked again, and their grades republished.

- Methods: XBlocks have access to their children and parent, and can invoke methods on them simply by invoking Python methods.

Views and Handlers are both inspired by web applications, but have different uses, and therefore different designs. Views are invoked by the runtime to produce a rendering of some course content. Their results are aggregated together hierarchically, and so are not expressed as an HTTP response, but as a structured Widget. Handlers are invoked by XBlock code in the browser, so they are defined more like traditional web applications: they accept an HTTP request, and produce an HTTP response.

2.5 Views

Views are how XBlocks render themselves. The runtime will invoke a view as part of creating a webpage for part of a course. The XBlock view should return data in the form needed by the runtime. Often, the result will be a *fragment* that the runtime can compose together into a complete page.

Views can specify caching information to let runtimes avoid invoking the view more frequently than needed. TODO: Describe this.

2.6 Handlers

TODO: Describe handlers.

2.7 Querying

Blocks often need access to information from other blocks in a course. An exam page may want to collect information from each problem on the page, for example.

TODO: Describe how that works.

2.8 Tags

TODO: Blocks can have tags and you can use them in querying.

When XBlock views are used to make a web page, each block produces just one piece of the page, by returning a Fragment. A Fragment carries content (usually HTML) and resources (for example, Javascript and CSS) needed by the content. These allow for the composition of HTML, Javascript and CSS elements in a predictable fashion.

Widgets have a number of attributes:

- **Content:** Content is most often HTML, but could also have an arbitrary mimetype. Each widget only has a single content value.
- **Javascript:** Javascript resources can include both external files to link to, and inline Javascript source code. When widgets are composed, external Javascript links will be uniqued, so that any individual page isn't loaded multiple times.
- **CSS:** Like Javascript, CSS can be both external and inline, and the external resources will be uniqued when widgets are composed.
- **Javascript initializer:** The Javascript specified for a widget can also specify a function to be called when that widget is rendered on the page. This function will be passed the DOM element containing all of the content from the widget, and is then expected to execute any code needed to make that widget operational. The Javascript view will also be passed a Javascript runtime object containing a set of functions that generate links back to the XBlocks handlers and views on the runtime server.

Since XBlocks nest hierarchically, a single XBlock view might require collecting renderings from each of its children, and composing them together. The parent will be responsible for deciding how the children's content composes together to create the parent content. The widget system has utilities for composing children's resources together into the parent.

Other information on widgets `[[TODO: write this.]]`:

Caching information specifying how long a widget can be cached for. This can be specified at widget creation, or by using a decorator on the view function (if the cache duration is known ahead of time)

See also:

The *Fragment API* documentation.

4.1 Fragment

class `xblock.fragment.Fragment` (*content=None*)

A fragment of a web page, for XBlock views to return.

A fragment consists of HTML for the body of the page, and a series of resources needed by the body. Resources are specified with a MIME type (such as “application/javascript” or “text/css”) that determines how they are inserted into the page. The resource is provided either as literal text, or as a URL. Text will be included on the page, wrapped appropriately for the MIME type. URLs will be used as-is on the page.

Resources are only inserted into the page once, even if many Fragments in the page ask for them. Determining duplicates is done by simple text matching.

add_content (*content*)

Add content to this fragment.

content is a Unicode string, HTML to append to the body of the fragment. It must not contain a `<body>` tag, or otherwise assume that it is the only content on the page.

add_css (*text*)

Add literal CSS to the Fragment.

add_css_url (*url*)

Add a CSS URL to the Fragment.

add_frag_resources (*frag*)

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

This is used by an XBlock to collect resources from Fragments produced by its children.

frag is a Fragment.

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

add_frags_resources (*frags*)

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

This is used by an XBlock to collect resources from Fragments produced by its children.

frags is a sequence of Fragments.

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

add_javascript (*text*)

Add literal Javascript to the Fragment.

add_javascript_url (*url*)

Add a Javascript URL to the Fragment.

add_resource (*text*, *mimetype*, *placement=None*)

Add a resource needed by this Fragment.

Other helpers, such as `add_css()` or `add_javascript()` are more convenient for those common types of resource.

text: the actual text of this resource, as a unicode string.

mimetype: the MIME type of the resource.

placement: where on the page the resource should be placed:

None: let the Fragment choose based on the MIME type.

“head”: put this resource in the <head> of the page.

“foot”: put this resource at the end of the <body> of the page.

add_resource_url (*url*, *mimetype*, *placement=None*)

Add a resource by URL needed by this Fragment.

Other helpers, such as `add_css_url()` or `add_javascript_url()` are more convenient for those common types of resource.

url: the URL to the resource.

Other parameters are as defined for `add_resource()`.

body_html ()

Get the body HTML for this Fragment.

Returns a Unicode string, the HTML content for the <body> section of the page.

foot_html ()

Get the foot HTML for this Fragment.

Returns a Unicode string, the HTML content for the end of the <body> section of the page.

head_html ()

Get the head HTML for this Fragment.

Returns a Unicode string, the HTML content for the <head> section of the page.

initialize_js (*js_func*)

Register a Javascript function to initialize the Javascript resources.

js_func is the name of a Javascript function defined by one of the Javascript resources. As part of setting up the browser's runtime environment, the function will be invoked, passing a runtime object and a DOM element.

4.2 Runtime

A Runtime object is passed to the XBlock constructor. It has methods for interacting with the XBlock's environment.

class `xblock.runtime.Runtime`

Access to the runtime environment for XBlocks.

A pre-configured instance of this class will be available to XBlocks as *self.runtime*.

get_block (*block_id*)

Get a block by ID.

Returns the block identified by *block_id*, or raises an exception.

handler_url (*url*)

Get the actual URL to invoke a handler.

url is the abstract URL to your handler. It should start with the name you used to register your handler.

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

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, context, view_name*)

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_child (*child, context, view_name=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, context, view_name=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()*.

4.3 XBlock

class `xblock.core.XBlock` (*runtime, model_data*)

Base class for XBlocks.

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

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

runtime is an instance of `xblock.core.Runtime`. Use it to access the environment. It is available in XBlock code as `self.runtime`.

model_data is a dictionary-like interface to runtime storage. XBlock uses it to implement your storage fields.

classmethod `json_handler` (*fn*)

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. Any data returned by the function will be JSON-encoded and returned as the response.

classmethod `postprocess_input` (*node, usage_factory*)

The class can adjust a parsed Usage tree.

classmethod `preprocess_input` (*node, usage_factory*)

The class can adjust a parsed Usage tree.

classmethod `tag` (*tags*)

Add the words in *tags* as class tags to this class.

X

`xblock.core`, [13](#)
`xblock.fragment`, [11](#)
`xblock.runtime`, [13](#)

A

`add_content()` (xblock.fragment.Fragment method), 11
`add_css()` (xblock.fragment.Fragment method), 11
`add_css_url()` (xblock.fragment.Fragment method), 11
`add_frag_resources()` (xblock.fragment.Fragment method), 11
`add_frags_resources()` (xblock.fragment.Fragment method), 11
`add_javascript()` (xblock.fragment.Fragment method), 12
`add_javascript_url()` (xblock.fragment.Fragment method), 12
`add_resource()` (xblock.fragment.Fragment method), 12
`add_resource_url()` (xblock.fragment.Fragment method), 12

B

`body_html()` (xblock.fragment.Fragment method), 12

F

`foot_html()` (xblock.fragment.Fragment method), 12
Fragment (class in xblock.fragment), 11

G

`get_block()` (xblock.runtime.Runtime method), 13

H

`handler_url()` (xblock.runtime.Runtime method), 13
`head_html()` (xblock.fragment.Fragment method), 12

I

`initialize_js()` (xblock.fragment.Fragment method), 12

J

`json_handler()` (xblock.core.XBlock class method), 14

P

`postprocess_input()` (xblock.core.XBlock class method), 14

`preprocess_input()` (xblock.core.XBlock class method), 14

Q

`query()` (xblock.runtime.Runtime method), 13
`querypath()` (xblock.runtime.Runtime method), 13

R

`render()` (xblock.runtime.Runtime method), 13
`render_child()` (xblock.runtime.Runtime method), 13
`render_children()` (xblock.runtime.Runtime method), 13
Runtime (class in xblock.runtime), 13

T

`tag()` (xblock.core.XBlock class method), 14

X

XBlock (class in xblock.core), 13
xblock.core (module), 13
xblock.fragment (module), 11
xblock.runtime (module), 13