Bukkit Metadata API Usage Guide

Overview:

The Bukkit Metadata API privides a new framework for exchanging information between plugins without requiring plugins to be dependant on each other. Metadata providers attach metadata values to existing Bukkit entities such as players, blocks, and worlds. Metadata consumers then read these values as needed. Example uses of Metadata are:

- Attaching chat prefixes to Players
- Marking areas of the world as “owned” by a player
- Publishing a player’s bank account balance, factional allegiance, or NPC emnity

The following architectural goals were used when developing the Metadata framework:

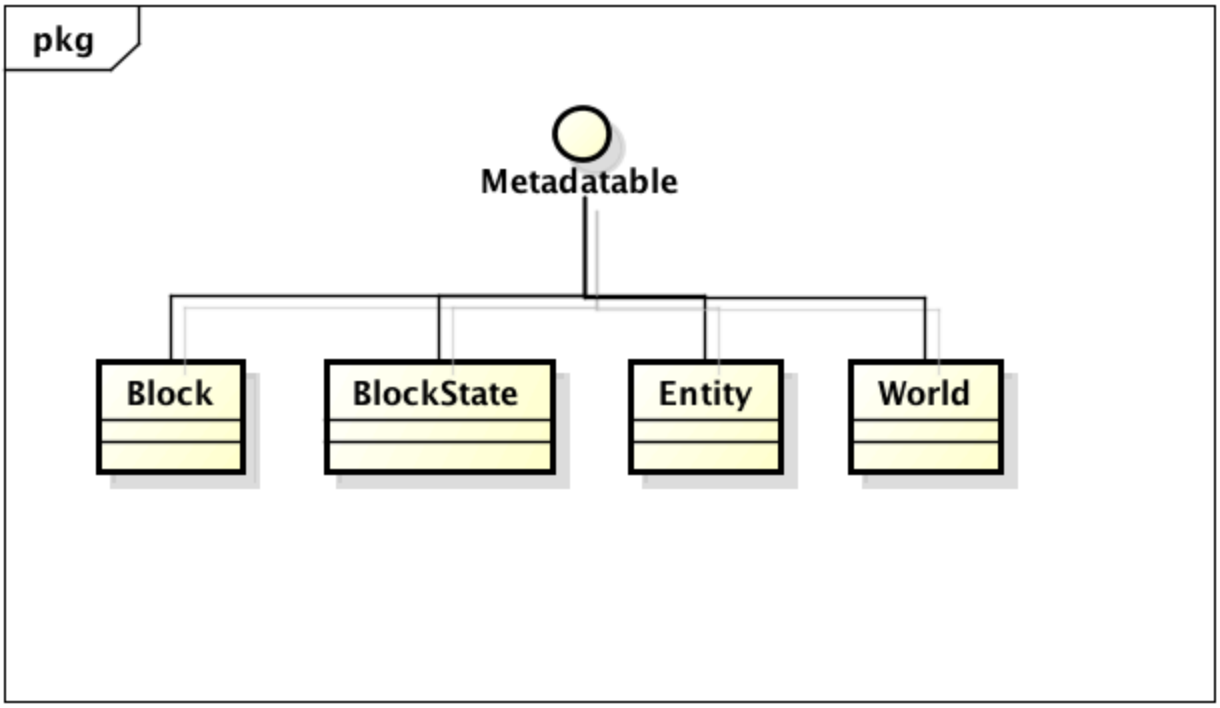
- All metadata is lazy. Metadata values are not actually computed until another plugin requests them. Memory and CPU are conserved by not computing and storing unnecessary metadata values.
- All metadata is cached. Once a metadata value is computed its value is cached in the metadata store to prevent further unnecessary computation. An invalidation mechanism is provided to flush the cache and force recompilation of metadata values.
- All metadata is softly referenced. The metadata framework only keeps soft references to the values in metadata as a way to facilitate garbage collection. Since all metadata is both lazy and cached, metadata values will be transparently regenerated should the garbage collector purge the values due to memory constraints.
- Metadata access is thread safe. Care has been taken to protect the internal data structures and access them in a thread safe manner.
- Metadata is exposed for all objects that descend from Entity, Block, and World. All Entity and World metadata is stored at the Server level and all Block metadata is stored at the World level.
- Metadata is NOT keyed on references to original objects - instead metadata is keyed off of unique fields within those objects. Doing this allows metadata to exist for blocks that are in chunks not currently in memory. Additionally, Player objects are keyed off of player name so that Player metadata remains consistent between logins.
- Metadata convenience methods have been added to all Entities, Players, Blocks, BlockStates, and World allowing direct access to an individual instance's metadata.
- Players and OfflinePlayers share a single metadata store, allowing player metadata to

be manipulated regardless of the player's current online status.

Metadata Keys

All metadata values are referenced by a unique string called the metadataKey. Most metadata keys are plugin specific, but common ones like “ChatPrefix” can be shared between many plugins. To get or set a metadata value you only need to know the metadataKey. You don’t need to know anything about any other plugin that will read or write to that metadata. Multiple plugins can publish values for the same metadataKey without conflicting. If two plugins publish a metadata value to the same object with the same metadataKey, both values will be available for metadata consumers to use.

Metadatable Objects and Getting/Setting Metadata

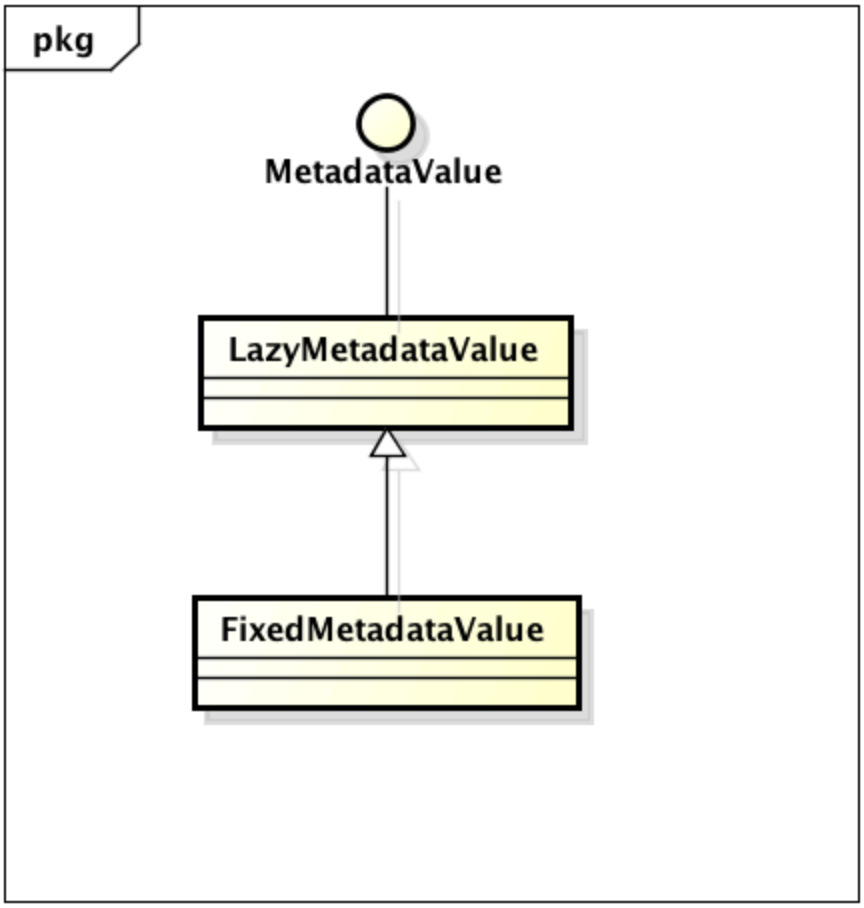


Game objects that implement Metadatable are able to have metadata set on them. All Metadatable objects have four key metadata methods:

1. setMetadata(String metadataKey, MetadataValue newMetadataValue);
2. getMetadata(String metadataKey);
3. hasMetadata(String metadataKey);

4. removeMetadata(String metadataKey, Plugin owningPlugin);

Types of Metadata



There are two standard types of metadata provided by the Metadata framework.

- LazyMetadataValue - these metadata values are computed only as needed. When you create a LazyMetadataValue, you must pass a Callable<Object> implementation to the constructor so that the LazyMetadataValue knows how to ask your plugin for the latest value. By default, a lazy metadata value will remember the last value it computed until its invalidate() method. LazyMetadataValues also have an optional cache strategy which is discussed below.
- FixedMetadataValue - a specialization of LazyMetadataValue that always returns the same result when asked for its value. Once constructed with an initial value, these objects never change.

Caching Strategies

If you choose to publish a LazyMetadataValue instead of a FixedMetadataValue, you have the option of using one of three possible cache strategies. A cache strategy is a way of telling the metadata value how frequently it should ask your plugin for a new value.

- CACHE_AFTER_FIRST_EVAL - this is the default caching strategy. When used, a metadata value will compute and cache its value the first time it is accessed and only recompute if the invalidate() method is called or the garbage collector collects the internally cached value.
- NEVER_CACHE - when this caching strategy is used, the metadata value is recomputed every time it is asked for.
- CACHE_ETERNALLY - when this caching strategy is used, the metadata value will only be computed the first time it is accessed. Future calls to invalidate() will be ignored. This strategy is normally only used by FixedMetadataValue objects.

Sample Plugin

A sample plugin that uses the metadata framework can be found at https://github.com/rmichela/MetadataDemo/blob/master/src/com/ryanmichela/metadatademo/MetadataDemo.java