Events Guide

Events allow developers to execute own piece of code when something important happens (fe. player joins). WaterdogPE comes with powerful API which allows developers to create and call own events or handle default events.

Event class

A base event is represented by extending the `dev.waterdog.event.Event` class.

CancellableEvent

If base event class implements `CancellableEvent` interface, the event will be considered as cancellable. We use cancellable events to signalize that some task should be canceled or ignored. CancellableEvent implements these methods:

- `isCancelled() : bool` returns true if the event is cancelled.
- `setCancelled(bool) : void` sets whether the event will be cancelled.

AsyncEvent

Our goal is to use the multi threading feature provided by Java as much as possible. Therefore we have created async events. To mark an event as async, use the `AsyncEvent` annotation. This event will have all the event handlers called asynchronously using a thread pool executor. Async events are supposed to not block the original thread where the event is called from. Using async events is recommended especially for events which don't change any values.
Example of async event:

```
@AsyncEvent
public class TransferCompleteEvent extends Events {
  // Your code here...
}
```

Event handling

Events are handled using runnables with the event as an argument (consumer). Subscribing to an event is done through the `EventManager` class using the `subscribe(Event, Handler, EventPriority)` method. This will add your method to list of handers which will be executed once event is called. Example of handling an event:

```
public void onEnable() {
  // PlayerChatEvent - the event to be subscribed
  // this::onChat - reference to the method which will be executed
  this.getProxy().getEventManager().subscribe(PlayerChatEvent.class, this::onChat);
}

public void onChat(PlayerChatEvent event) {
  // Getting value provided by event
  ProxiedPlayer player = event.getPlayer();
  // Cancelling event
  event.setCancelled(true);
}
```

Event priority

If the event has more handlers we might want to create, prioritize the order of events. using the `EventPriority` enum we can define priorities per handler. Events with lower priority will be executed first and its values may be changed by handlers with higher priority. By default `EventPriority.NORMAL` is used.
Subscribing with defined priority: `subscribe(PlayerChatEvent.class, this::onChat, EventPriority.HIGHEST)`

Event calling

To call specific event instances we use the `callEvent(Event)` method. This method will schedule every task from event handlers in prioritized order. If the event has `AsyncEvent` annotation present the method will return `CompletableFuture<Event>` which will be completed once all handlers will be executed. If the event has not annotation preset, `null` will be returned.
Calling non-async event:

```
```

```
ProtocolCodecRegisterEvent event = new ProtocolCodecRegisterEvent(protocol, builder);
proxy.getEventManager().callEvent(event);
if (event.isCancelled()){
  // Your code here...
}
```

Calling async event:

```
PlayerLoginEvent event = new PlayerLoginEvent(this);
this.proxy.getEventManager().callEvent(event).whenComplete((futureEvent, exception) -> {
  if (futureEvent.isCancelled()) {
    // Your code here...
  }
});
```

Revision #8
Created Tue, Nov 24, 2020 8:58 AM by Alemiz
Updated Sun, Oct 30, 2022 7:52 AM by TobiasDev