EVENT EMITTERS IN NODE.JS

1.
Introduction To Event Emitters

What Is an Event?

An whole software paradigm focuses around events and how they are used. Event-driven architecture is becoming more widespread, as event-driven systems generate, detect, and respond to many types of events.

An event in event-driven programming is the consequence of a single or numerous actions. This might be a user action or a sensor's periodic output, for example.

Event-driven applications can be thought of as publish-subscribe models, in which a publisher initiates events and subscribers listen to them and respond accordingly.

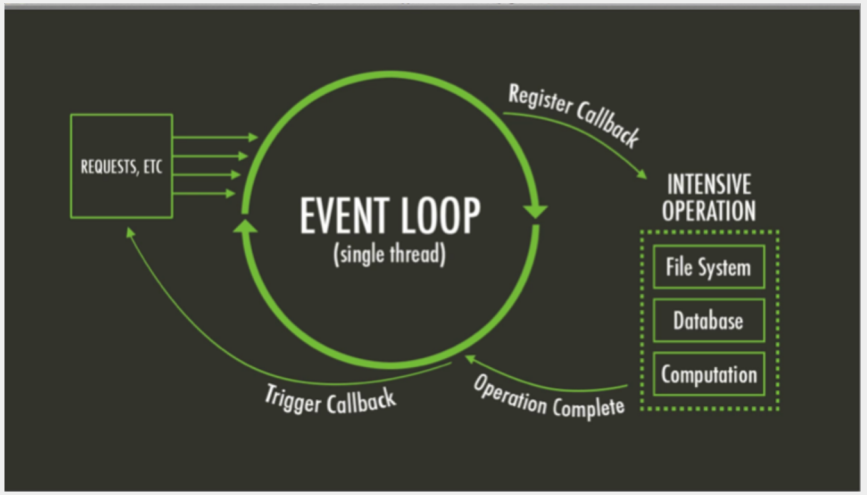
Assume, for example, that we have an image server where users may post images. An action, such as uploading the image, would generate an event in event-driven programming. It would also require 1..n subscribers to that event in order to be used.

Once the upload event is triggered, a subscriber can respond by sending an email to the website's administrator informing them that a user has submitted a photo. Another subscriber may gather data about the activity and store it in a database.

These occurrences are normally independent of one another, yet they may be reliant on one another.

What are Event Emitters?

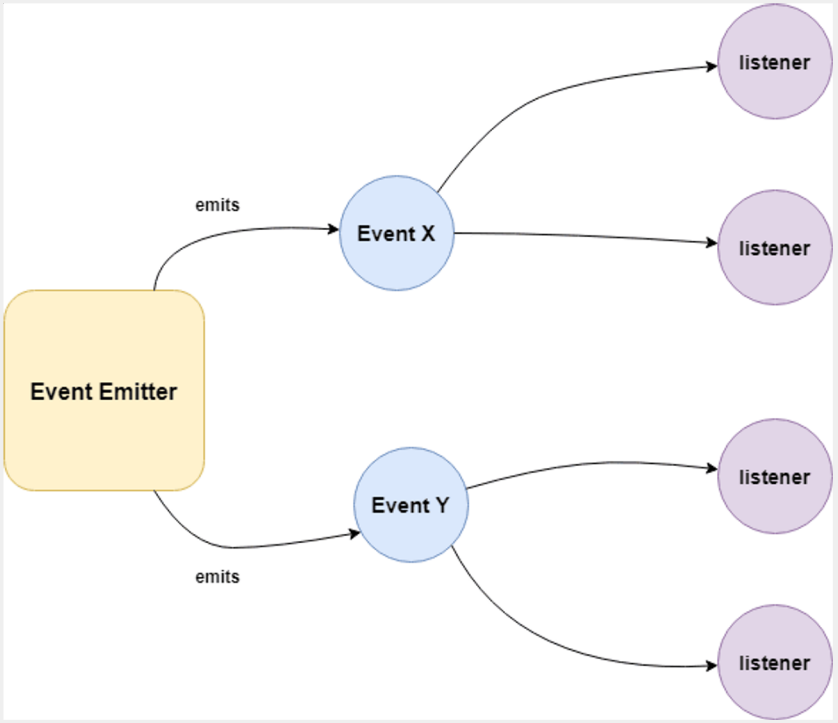
In Node.js an event can be described simply as a string with a corresponding callback. An event can be "emitted" (or in other words, the corresponding callback be called) multiple times or you can choose to only listen for the first time it is emitted.



Event emitters are objects in Node.js that trigger an event by sending a message to signal that an action was completed. JavaScript developers can write code that listens to events from an event emitter, allowing them to execute functions every time those events are triggered. In this context, events are composed of an identifying string and any data that needs to be passed to the listeners

How does it work?

Event Emitter emits the data in an event called message
A Listened is registered on the event message
when the message event emits some data, the listener will get the data.



2.
Methods of an Event Emitter

Methods of the Event Emitter

- **addListener**(event, listener) or **on**(event, listener)
  - Adds a listener for the provided event to the end of the listeners array. There are no checks to detect if the listener has already been added. Numerous calls with the same event and listener combination will result in the listener being added multiple times. Returns the emitter, allowing calls to be chained.
- **once**(event, listener)
  - Adds the event listener only once. Once the event is called for the subsequent event there is no output.
- **removeListener**(event, listener)
  - Removes a listener from the listener array for the specified event
- **removeAllListeners**([event])
  - Removes all listeners, of the specified event

Methods of the Event Emitter

- **setMaxListeners**(n)
  - If more than ten listeners are added to an event, EventEmitters will issue a warning. This is a beneficial default that aids in the detection of memory leaks. Obviously, not all Emitters should be limited to a maximum of ten. This function allows you to boost it. Set to zero to make it infinite.
- **Listeners**(event)
  - Returns an array of all the events attached to the given event.
- **emit**(event, [arg1], [arg2], [...])
  - Execute each of the listeners in order with the supplied arguments. Returns true if the event had listeners, false otherwise.

3.
Steps Involved in Making Event Emitters

Step 1 — Emitting Events

In this step, we'll explore the two most common ways to create an event emitter in Node.js. The first is to use an event emitter object directly, and the second is to create an object that extends the event emitter object.

```
var events   = require("events"),
                    emitter  = new
events.EventEmitter(),
   username = "admin",
                    password = "admin";
// an event listener
emitter.on("userAdded", function (username,
password) {
                    console.log("Added user " +
username);
});
// Emit an event

emitter.emit("userAdded", username,
password);
```

Step 2 — Listening for Events

Node.js allows us to add a listener for an event with the on() function of an event emitter object. This listens for a particular event name and fires a callback when the event is triggered. Adding a listener typically looks like this:

```
eventEmitter.on(event_name, callback_function) {
    action
}
```

```
var events   = require("events"),
                    emitter  = new events.EventEmitter(),
   username = "admin",
                    password = "admin";
// an event listener
emitter.on("userAdded", function (username, password) {
                    console.log("Added user " + username);
});
// Emit an event

emitter.emit("userAdded", username, password);
```

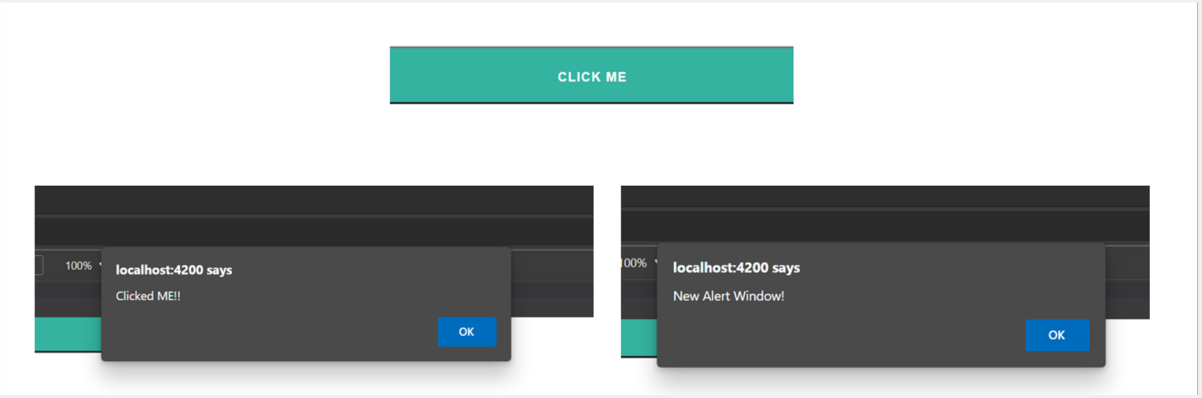
Step 3 — Capturing Event Data

The data is also passed along via the emitted events. Let's look at how we can record the data that comes with an event.

```
myClick = () => {
  alert('Clicked ME!!');
  this.eventEmitter.on('alert', this.anotherAlert);
  this.eventEmitter.emit('alert', 'New Alert Window!');
};

anotherAlert = function (data: string) {
  alert(data);
};
```

```
onClick() {
  this.eventEmitter.on('click', this.myclick);
  this.eventEmitter.emit('click');
}
```



Step 4 — Handling Error Events

If an event emitter is unable to complete its task, it should generate an event to indicate that the task was unsuccessful. An error event is the typical mechanism for an event emitter in Node.js to report failure.

```
buy(email, price) {
    if (this.supply > 0) {
        this.supply-;
        this.emit("buy", email, price, Date.now());
        return;
    }

    this.emit("error", new Error("There are no more tickets left to purchase"));
}
```

```
ticketManager.buy("test@email.com", 10);
ticketManager.buy("test@email.com", 10);
ticketManager.buy("test@email.com", 10);
ticketManager.buy("test@email.com", 10);
```

Continuation....

```
Output
Sending email to test@email.com
Running query: INSERT INTO orders VALUES (email, price, created) VALUES (test@email.com, 10,
Sending email to test@email.com
Running query: INSERT INTO orders VALUES (email, price, created) VALUES (test@email.com, 10,
Sending email to test@email.com
Running query: INSERT INTO orders VALUES (email, price, created) VALUES (test@email.com, 10,
events.js:196
      throw er; // Unhandled 'error' event
      ^

Error: There are no more tickets left to purchase
    at TicketManager.buy (/home/sammy/event-emitters/ticketManager.js:16:28)
    at Object.<anonymous> (/home/sammy/event-emitters/index.js:17:15)
    at Module._compile (internal/modules/cjs/loader.js:1128:30)
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:1167:10)
    at Module.load (internal/modules/cjs/loader.js:983:32)
    at Function.Module._load (internal/modules/cjs/loader.js:891:14)
    at Function.executeUserEntryPoint [as runMain] (internal/modules/run_main.js:71:12)
    at internal/main/run_main_module.js:17:47
Emitted 'error' event on TicketManager instance at:
    at TicketManager.buy (/home/sammy/event-emitters/ticketManager.js:16:14)
    at Object.<anonymous> (/home/sammy/event-emitters/index.js:17:15)
    [... lines matching original stack trace ...]
    at internal/main/run_main_module.js:17:47
```

4.
Using Event Emitters in Angular

Event Emitter

```
var events   = require("events"),
    emitter  = new events.EventEmitter(),
    username = "Eyal",
    password = "Vardi";


// an event listener
emitter.on("userAdded", function (username, password) {
    console.log("Added user " + username);
});


// Emit an event
emitter.emit("userAdded", username, password);
```

One-Time Event Listeners

Sometimes you may be interested in reacting to an event only the first time it occurs.

```
var events = require("events");
var emitter = new events.EventEmitter();
emitter.once("foo", function () {
    console.log("In foo handler");
});

emitter.emit("foo");
emitter.emit("foo");
```

Inspecting Event Listeners

```
var events = require("events");
var EventEmitter = events.EventEmitter;
var emitter = new EventEmitter();

emitter.on("foo", function () { });
emitter.on("foo", function () { });

console.log( EventEmitter.listenerCount(emitter, "foo") );
```

The newListener Event

Each time a new event handler is registered, the event emitter emits a newListener event.

```
var events = require("events");
var EventEmitter = events.EventEmitter;
var emitter = new EventEmitter();

emitter.on("foo", function (){ console.log("In foo handler"); });

emitter.listeners("foo").forEach(function (handler) {
    handler();
});
```

Removing Event Listeners

An event listener can be removed after it’s been attached to an event emitter.

emitter.removeAllListeners([eventName])

emitter.removeListener([eventName],[handler])

```
handler1 = () => {
  console.log('A');
  this.eventEmitter.removeListener('event', this.handler2);
};

handler2 = () => {
  console.log('B');
};

ngOnInit(): void {
  this.myForm = new FormGroup({
    number: new FormControl()
  });
}

onNumberClick() {
  this.eventEmitter.on('event', this.handler1);
  this.eventEmitter.on('event', this.handler2);
  this.eventEmitter.emit('event');
  this.eventEmitter.emit('event');
}
```

```
A                                   fileupload.component.ts:29
B                                   fileupload.component.ts:34
A                                   fileupload.component.ts:29
>
```

Inheriting from Event Emitters

You can create custom objects that inherit from EventEmitter and include additional application-specific logic.

```
var EventEmitter = require("events").EventEmitter;
var util = require("util");

function UserEventEmitter() {
    EventEmitter.call(this);
    this.addUser = function (username, password) {
        // add the user
        // then emit an event
        this.emit("userAdded", username, password);
    };
};

util.inherits(UserEventEmitter, EventEmitter);
```

Full Solution Inheritance

```
function SuperType(name){
    this.name = name;
    this.colors = ['red', 'blue', 'green'];
}
SuperType.prototype.sayName = function(){ return this.name; };


function SubType(name, age){
    SuperType.call(this, name);
    this.age = age;
}
SubType.prototype = Object.create(SuperType.prototype);
SubType.prototype.constructor = SubType;

SubType.prototype.sayAge = function(){
    alert(this.age);
};

SubType.prototype.sayName = function(){
    return SuperType.prototype.sayName.call(this) + "!!";
};
```

Using a Custom Event Emitter

```
var user     = new UserEventEmitter();
var username = "colin";
var password = "password";

user.on("userAdded", function (username, password) {
    console.log("Added user " + username);
});

user.addUser(username, password)

console.log(user instanceof EventEmitter);
```

Using Events to Avoid Callback Hell

```
var fs = require("fs");
var fileName = "foo.txt";
fs.exists(fileName, function (exists) {
    if (exists) { fs.stat(fileName, function (error, stats) {
        if (error) { throw error; }
        if (stats.isFile()) {
            fs.readFile(fileName, "utf8", function (error, data) {
                if (error) { throw error; }
                    console.log(data);
                });
            }
        });
    }
});
```

Continuation

```
var EventEmitter = require("events").EventEmitter;
var util         = require("util");
var fs           = require("fs");

function FileReader(fileName) {
    var _self = this;
    EventEmitter.call(_self);

    _self.on("stats", function() {
        fs.stat(fileName, function(error, stats) {
            if (!error && stats.isFile()) {
                _self.emit("read");
            }
        });
    });
    _self.on("read", function() {
        fs.readFile(fileName, "utf8", function(error, data) {
            if (!error && data) { console.log(data); }
        });
    });

    fs.exists(fileName, function(exists) {
        if (exists) { _self.emit("stats"); }
    });
}

util.inherits(FileReader, EventEmitter);
var reader = new FileReader("foo.txt");
```

Done By

Aakar Mutha, Jason Karter