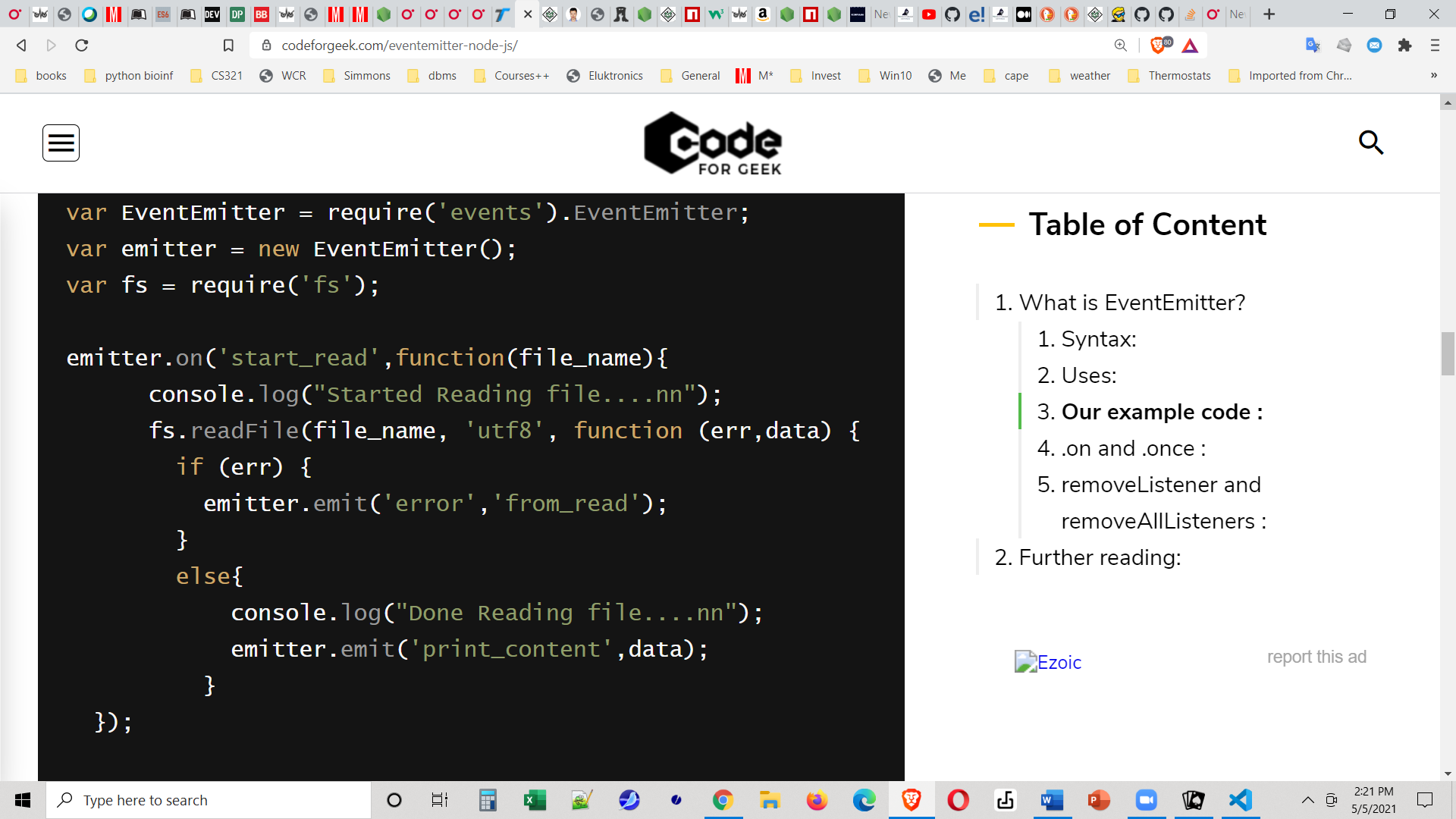
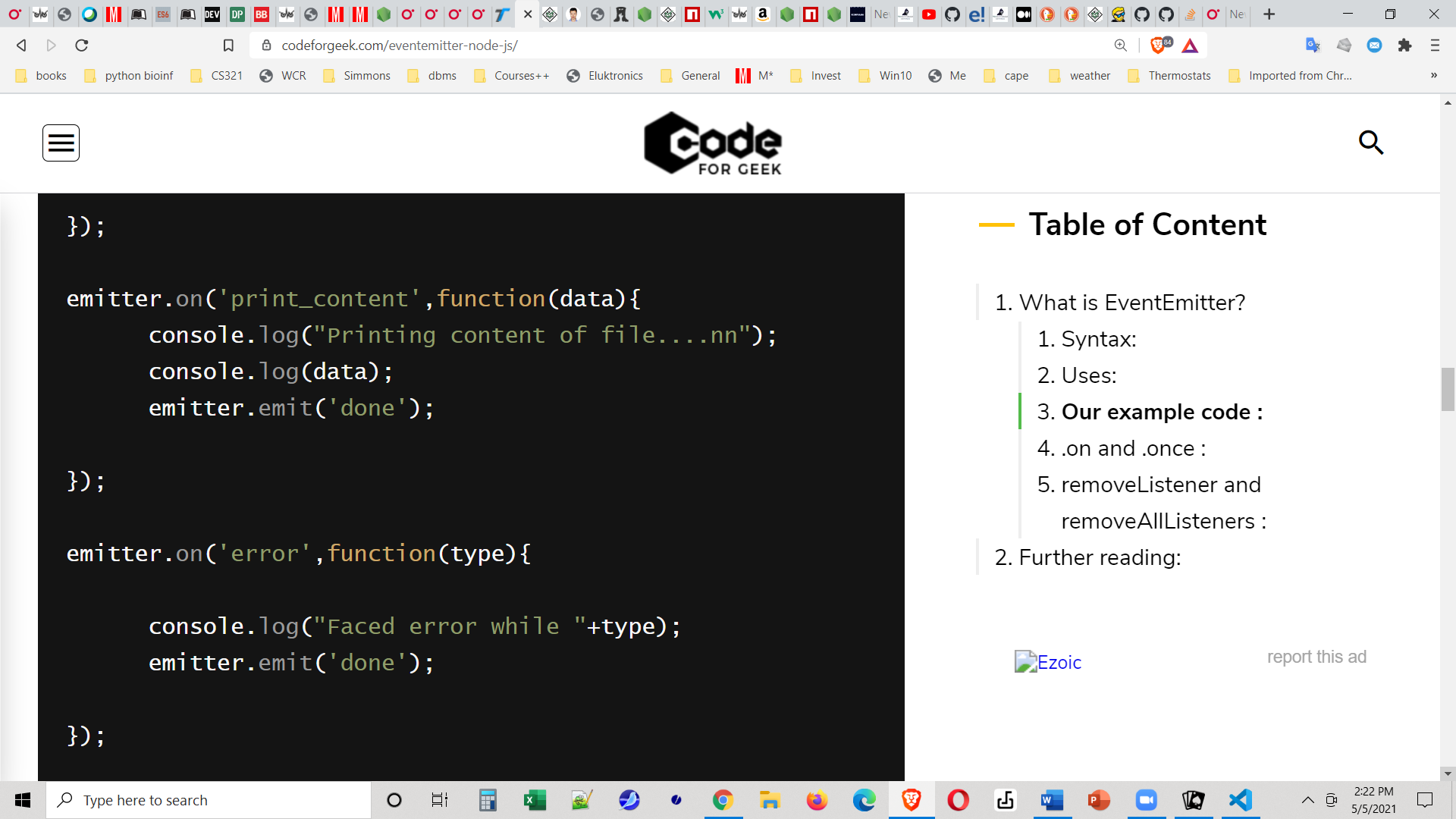
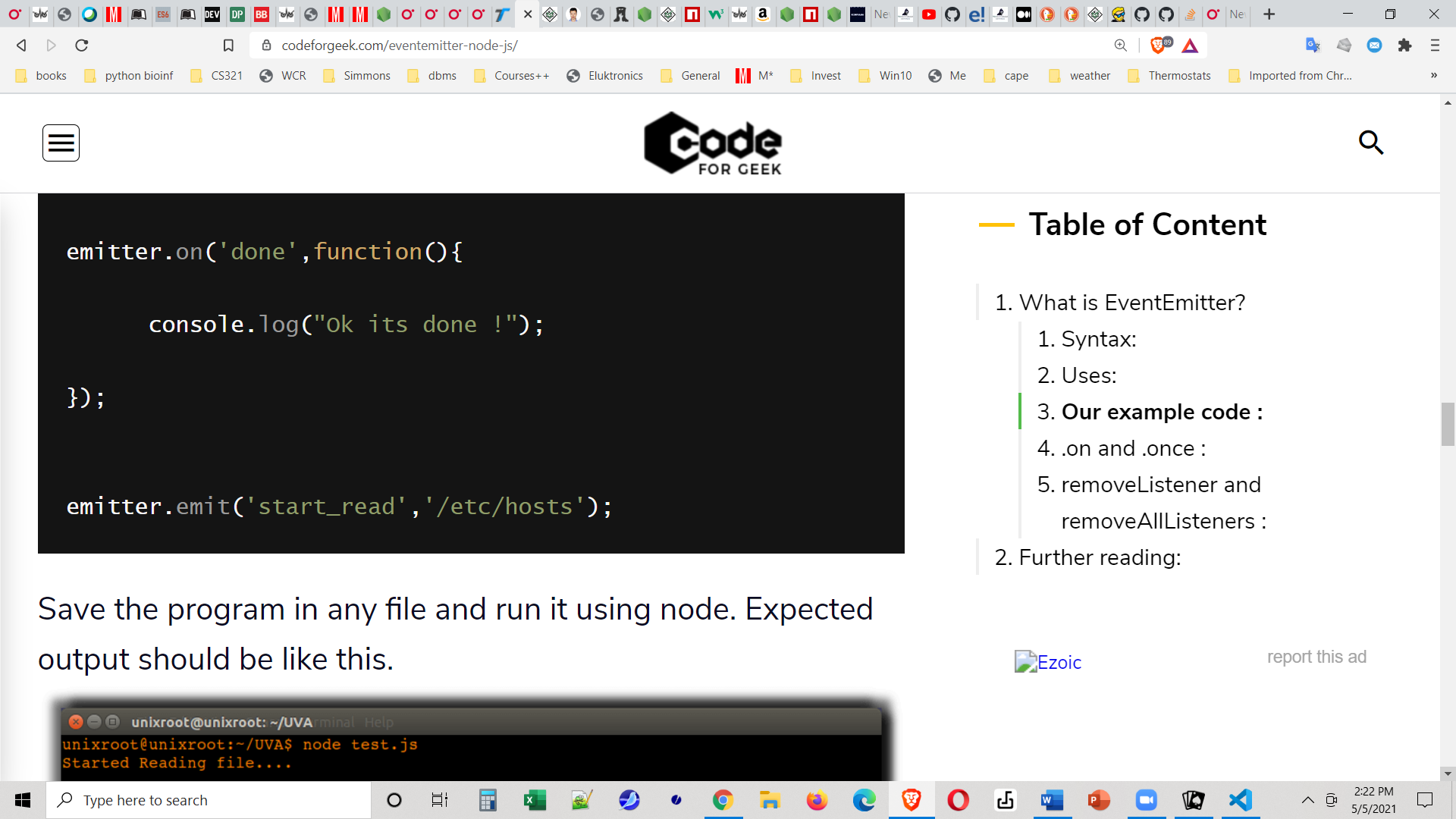
Understanding how an event emitter can control the flow of a program

From <https://codeforgeek.com/eventemitter-node-js/>

The code is:







Let's see how event emitters are used to control this program.

Look at the screen shots.

In the top screen shot we require the ***fs*** (file system) and ***events*** modules and we use the EventEmitter (a constructor function in ***events***) to instantiate an event emitter named emitter**.**

Again, I like to think of emitter as an events manager --- it can *attach event listeners to specific events* ( with emitter.on('someEvent', someFunction), just like in jQuery) and it can *emit specific events, with addition data If desired* (just like the trigger() of jQuery.)

Okay. We have emitter. Now back to our code. In the rest of the top screen shot we attach an event handler to the 'start\_read' event. That code starts at the top of the wavy green line ---- in a minute we'll look at it more carefully – and actually ends with the top line on the middle screen shot.

In the middle screen shot we attach two more event handlers --- at the top of the screen shot the dashed blue line points to the event handler for the 'print\_content' event, and in the middle of the screen shot the red line points to the event handler for the 'error' event.

In the bottom screen shot the heavy blue arrow with the yellow outline points to the event handler for the 'done' event.

So we have event handlers for 'start\_read', 'print\_content', 'done' and 'error'.

Finally, at the bottom of the bottom screen shot, the wavy green line points to the code being 'kicked off' by *emittin*g a 'start\_read' event. Finally!

Well, now, let's see how the 'start\_read' event is handled ---- back on the top screenshot.

We console.log a message that we have started reading, and then we use fs.readfile() to read a file.

If you look at the fs.readfile() it's clear that the first parameter of that function is the file which will be read, the second parameter is the encoding for the file (utf-8), and the third parameter is a callback function. That function has two parameters – err (which is obviously about an error in reading the file) and data (which will hold what was read from the file.

What does that callback function do? It tests to see if an err has been returned (if there was an error).  
If there was an error, the callback emits an 'error' event --- and so control will pass to the handler for the error event. If there is no error then (in the else clause) we console.log a message that we're done and emit a 'print\_content' event --- thereby passing control to the handler or that event.

Please notice that, whether there is an error or not, we pass control of the program to another part of the program by emitting an event and having that event's handler take over.

Let's see what happens next. First suppose that we did have an error --- the code was   
 emitter.emit('error', 'from read');

And the handler which was attached to the error event has been defined. It is in the lower half of the middle screenshot (red arrow). It is a function with one parameter *type*, which is the second parameter from emit('error', 'from read'). That is the value of of type is the string 'from read'. Our function logs a message and then –guess what – it emits a 'done' event.

Of course, our control now passes to the event handler for the done event (heavy blue arrow at the top of the bottom screen shot) which simply logs a message and the processing stops.

Now what happens If, way back when, our fs.readfile() proceded without an error? Well, look back at the top screen shot. We logged a message and then emited a 'print\_contents' event with data (what was read) as the second parameter (blue dashed arrow at the bottom of the top screenshot.) So control now passes to the event handler for the print\_contents event. That handler (blue dashed arrow at the top of the middle screenshot) logs a bunch of stuff, including the data which was passed to it, and then emits a 'done' event. And we saw previously that the 'done' event's handler logs a message and stops.

**So you can see why people refer to Node as "event-driven pro**g**rammin**g**".**