**Asynchronous JavaScript - Part 4… -** [Marc Kirk](https://medium.com/@byteslovesbits?source=post_page-----4fd493c0a994--------------------------------)Feb 26, 2022

In [part 3](https://medium.com/@byteslovesbits/asynchronous-javascript-part-3-864df36177f3), we discussed asynchronous callbacks. To recap, we use the help of the browser to give the illusion that JavaScript is asynchronous. **setTimeout()**is part of the browser’s global object and it allows us to give the illusion that synchronous JavaScript code is asynchronous. Figure 1.

A screenshot of a computer

Description automatically generated with medium confidence

Figure 1 — setimeout is supplied by the browser

**Blocking Code**

Let’s say we have a fancy pants web-page with a single button. We never know when a user will click that button. It could be now or it could be in several hours. We just don’t know. Because JavaScript is synchronous, this means the page just sits there waiting for the button to be clicked. It hangs. Unable to proceed to the next line of code — until the button is clicked. This is what is known as **blocking code**. Everything is blocked until the button has been pressed.

*Blocking code: Code cannot execute until some other code finishes*

To observer similar blocking behaviour, type **alert()** into the console. Figure 2 — the **alert()**method is a synchronous browser method and it completely blocks everything until the dialogue box is dismissed by the user.

From a user’s perspective, blocking code is horrendous and is usually a worst case scenario. The user cannot do anything with the page. It is blocked! They cannot scroll, zoom in, click a button, send a message, click like. The page is effectively dead!

Graphical user interface, text

Description automatically generated

Figure 2 — blocking code using the alert() method

So how does the browser prevent blocking code? In other words, how does it improve JavaScript’s synchronous model of execution? The browser provides us with asynchronous methods as part of its web-apis. We use these asynchronous methods to add **event listeners** to buttons or elements. The event listener waits asynchronously until the button is clicked. It then executes some code in response to the click. This way, the web page is free to do other useful stuff. There are many types of event listeners, such as events that fire when the mouse is clicked or the page loads.

Let’s look at an example of an event listener. Enter the code from Figure 3 into your web-browser’s console, followed by enter. Now click anywhere inside the white document to the left of the console. Observe how the document colour changes from white to red. Figure 4. This my friends, is asynchronous JavaScript in action. What we are doing is listening for a click event on the document's body and when that event happens we change the document’s CSS background color property to red. Also, note how the browser asynchronously changed the body style to **<body style="background-color: red;"></body>**.

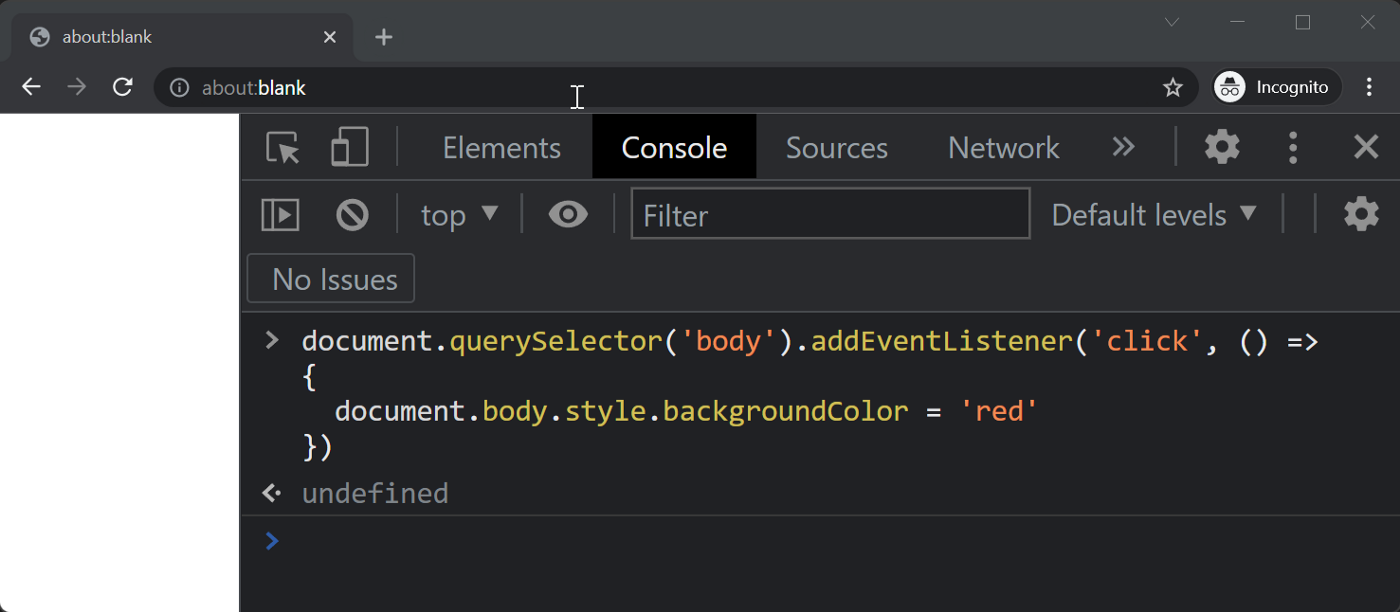


Figure 3 — adding an event listener to the dom body object

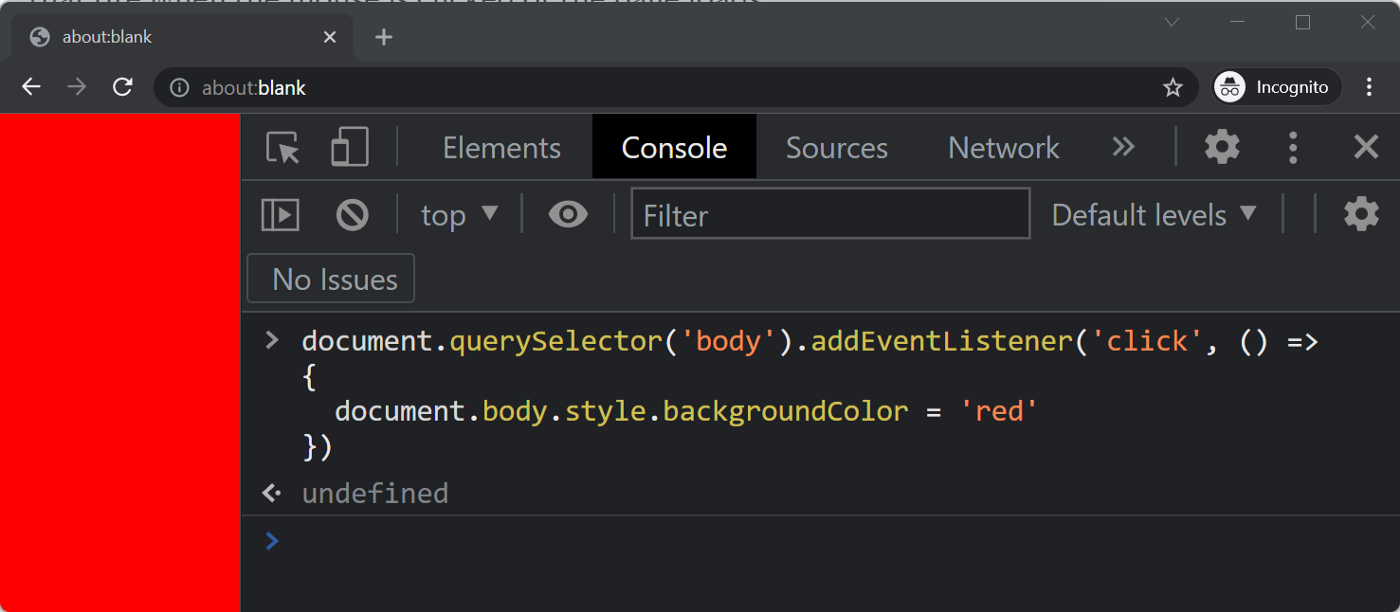


Figure 4- the event fires and the body changes colour

A screenshot of a computer

Description automatically generated with medium confidence

Figure 5 — the browser asynchronously updating the DOM

Now that we have a better understanding of why blocking code is bad and asynchronous code is good, let’s give you a little taster of what to expect in part 5. The event loop. Understanding the event loop will drive these concepts home. Once you fully understand the event loop, you will lay the foundations to being a world class asynchronous programmer.

Figure 6 shows a high level overview of the JavaScript Event loop.

Diagram

Description automatically generated