5.5.4 Schedule Task Auditing

The client has a feeling that users might tend to leave the app open in a browser window for days at a time. If the user doesn't refresh the page, then existing tasks won't be audited and a due date can creep up on them without their knowledge.

A better user experience would be to periodically check the due dates so the user doesn't need to remember to refresh the page. To do this, we'll automate the logic in the `auditTask()` function to run every 30 minutes. Luckily, the browser has tools to achieve just this, called **timers.**

JavaScript Timers

Timers allow us to execute blocks of code at a given time. There are two JavaScript methods that accomplish this:

- `setTimeout()`
- `setInterval()`

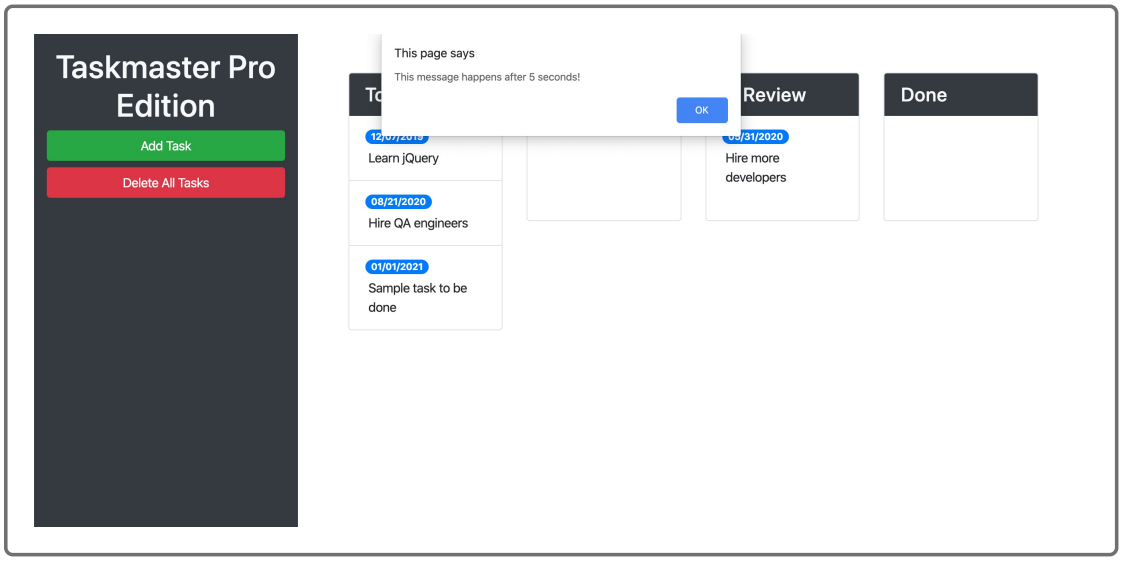
Let's see what `setTimeout()` does first. Type the following code in `script.js` at the bottom of the file:

```
setTimeout(function() {
  alert("This message happens after 5 seconds!");
}, 5000);
```


REWIND

We could also have used `window.setTimeout()` here, but because this method is native to the `window` object, we can simply use `setTimeout()` and the browser will infer it came from the `window`.

Save `script.js` and refresh the page, then wait five seconds. When that five seconds has elapsed, the following alert dialog box should appear:



As we can see, the `setTimeout()` function was given two arguments: a callback function and a number. The callback function is the block of code we want to have executed after an amount of time has passed. That

amount of time comes from the second argument, which is the number of milliseconds we want to wait for.

In the `setTimeout()` function we just implemented, we want the browser to wait five seconds (5,000 milliseconds) before executing a function that puts an alert dialog on the screen. The moment that function is done running, it's done forever and won't execute again. In our case, using `setTimeout()` is not that helpful, as we want to audit our tasks over and over again. So let's try the other timer function, `setInterval()`.

Remove `setTimeout()` from `script.js` and type the following code in its place:

```
setInterval(function() {
  alert("This alert shows up every five seconds!");
}, 5000);
```

Save `script.js` and refresh the page again. This time, after five seconds has elapsed, we'll get an alert just like we did before. But when we dismiss that alert dialog, another one shows up again after five seconds. That's the main difference between `setTimeout()` and `setInterval()`: the former will only run once while the latter will run on a timed schedule based on what is entered in the second argument.

In both cases, the user can still interact with the application while these timers are running because they are **asynchronous** functions. This means they run in the background until their time is up and then execute the callback function, allowing us to still use the other functionality in the app as usual.

PAUSE

Where else have we seen asynchronous behavior in JavaScript?

When we added event listener callbacks to DOM elements.

Hide Answer

Let's update the `setInterval()` function in script.js to audit tasks by changing the code to look like this:

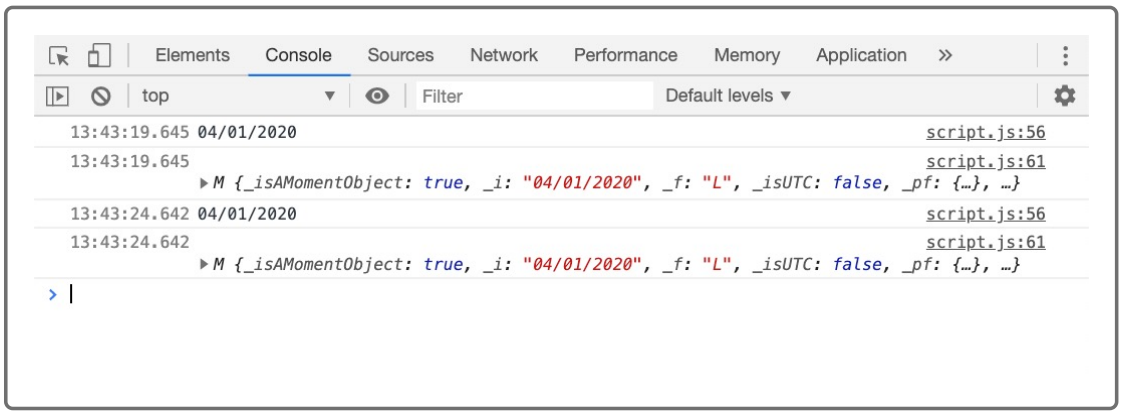
```
setInterval(function () {
  $(".card .list-group-item").each(function (el) {
    auditTask(el);
  });
}, 5000);
```

Here, the jQuery selector passes each element it finds using the selector into the callback function, and that element is expressed in the `el` argument of the function. `auditTask()` then passes the element to its routines using the `el` argument.

In this interval, we loop over every task on the page with a class of `list-group-item` and execute the `auditTask()` function to check the due date of each one.

A true test would be to wait a day to see if the task's background color automatically changes according to its due date, but that's an unrealistic development process. Instead, to verify that the interval is working, we can simply add a `console.log(taskEl)` statement in the `auditTask()` function.

Save `script.js`, refresh the page, and open the console to see if the interval is running every five seconds. After about ten seconds, the console should look something like the image shown below:



Because our `auditTask()` function checks on upcoming due dates by day, it is unnecessary to have this `setInterval()` run every five seconds. Instead, let's update it to run every half hour and change the `5000` millisecond value to `1800000`, which is 30 minutes.

It can be hard to come up with longer time durations in milliseconds, so a good trick to make this easier is to convert the time to something like this:

```
setInterval(function() {
  // code to execute
}, (1000 * 60) * 30);
```

In the previous example, we multiply 1,000 milliseconds by 60 to convert it to 1 minute. Then we multiply that minute by 30 to get a 30-minute timer.

DEEP DIVE

DEEP DIVE

To learn more about how to use `setTimeout()` and `setInterval()`, and how to cancel timer event functions, see the **MDN web documentation on timeouts and**

intervals. **(https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Timeouts_and_intervals)**

Great work—the application now has automated task auditing, which is the last new piece of functionality the client wanted us to implement! Everything from here on out will have to do with styling and UI polishing.