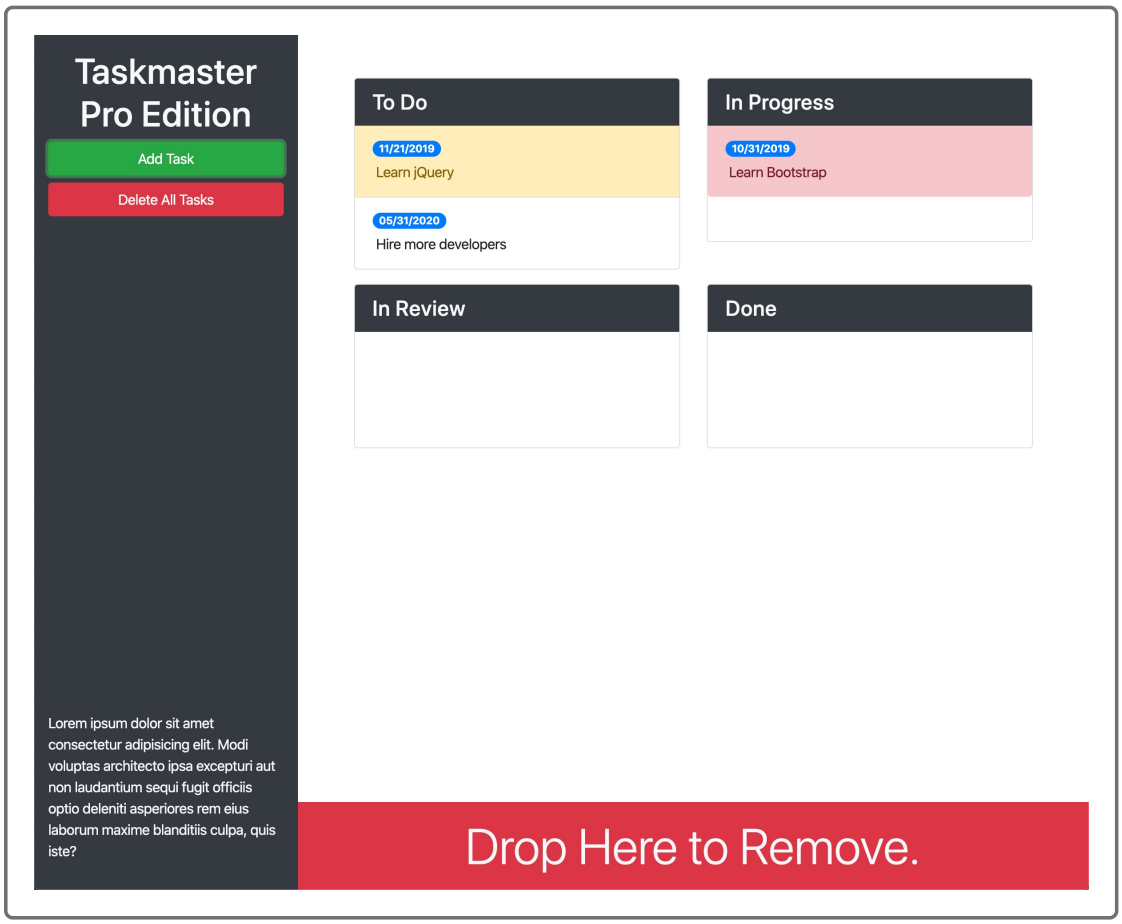
5.4.6 Use Moment.js for Due Date Audits

When we create a task or edit its due date, we want to be able to see if that date is either in the past or within two days from now. If it's in the past, we'll add a red background to the task item to let the user know it's overdue. If it's within the next two days, we'll add a yellow background to it.

Here's a screenshot of what an overdue task and an upcoming task will look like:



Because we need to run this type of functionality in the functions for both creating tasks and editing task due dates, we should create a separate function for it, called `auditTask`, and set it up to accept the task's `<li>` element as a parameter. This way we can add classes to it if need be.

Let's create the following function in `script.js`:

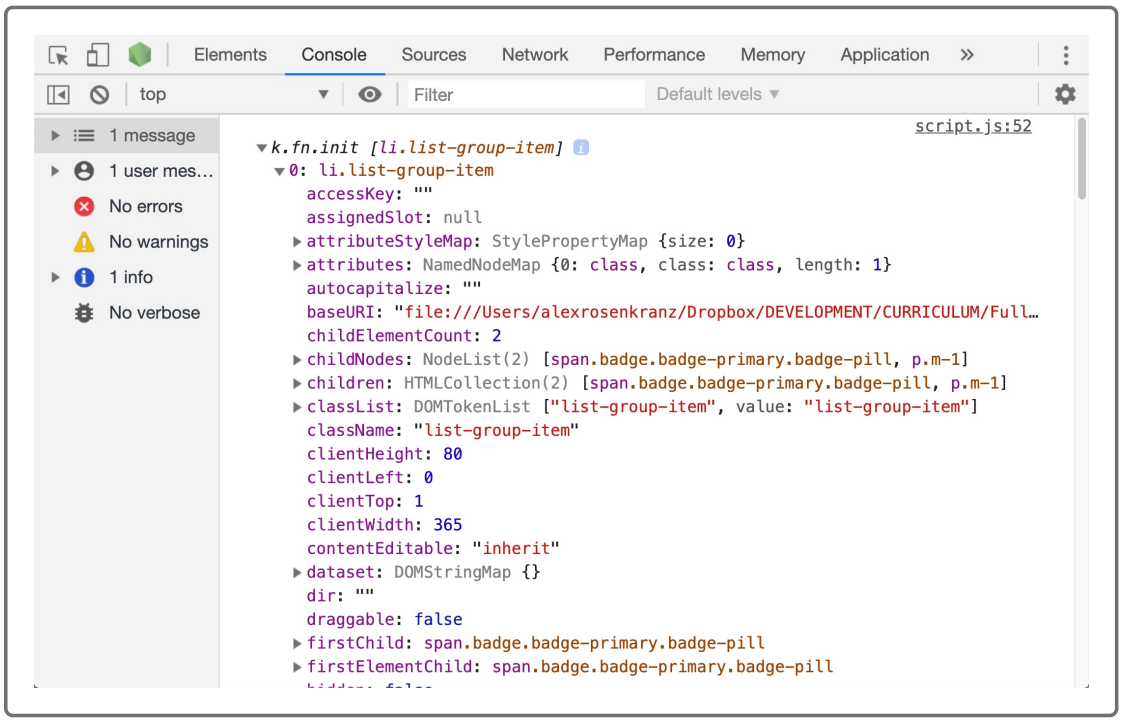
```
var auditTask = function(taskEl) {
  // to ensure element is getting to the function
  console.log(taskEl);
};
```

Now let's update the `createTask()` function to execute it and pass the `taskLi` element to it as an argument. Update `createTask()` to look like this:

```
var createTask = function(taskText, taskDate, taskList) {
  // create elements that make up a task item
  var taskLi = $("<li>").addClass("list-group-item");

  var taskSpan = $("<span>").addClass("badge badge-primary badge-pill"

  var taskP = $("<p>").addClass("m-1").text(taskText);

  // append span and p element to parent li
  taskLi.append(taskSpan, taskP);

  // check due date
  auditTask(taskLi);

  // append to ul list on the page
  $("#list-" + taskList).append(taskLi);
};
```

Note how we're executing `auditTask()` before we add the task item to the page. We want to ensure the element has all of its proper classes before getting to the page, as adding it and then changing how it looks could confuse users.

Let's save `script.js` and refresh the browser. If we have tasks saved, then they should automatically run through the `auditTask()` function and we can check the console to see if it works. If not, create a new task and check the console after submitting it. The console should look like this image:



We should see the elements for each task list item. Now that we know we're getting the element into `auditTask()`, we need to check what date was added to its `<span>` element. This will involve two actions. First, we need to use jQuery to retrieve the date stored in that `<span>` element. Second, we need to use the `moment()` function to convert that date value into a Moment object.

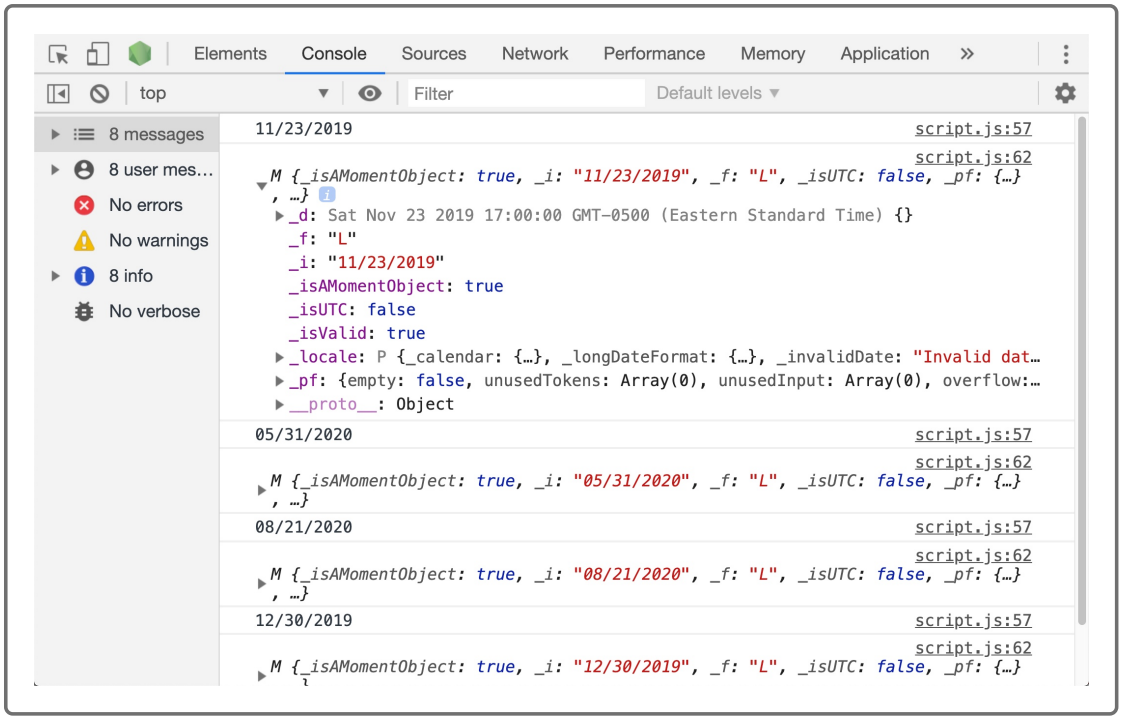
Get and Convert the Due Date

Let's update `auditTask()` to look like this:

```
var auditTask = function(taskEl) {
  // get date from task element
  var date = $(taskEl).find("span").text().trim();
  // ensure it worked
  console.log(date);

  // convert to moment object at 5:00pm
  var time = moment(date, "L").set("hour", 17);
  // this should print out an object for the value of the date variabl
  console.log(time);
```

```
};
```

Save `script.js` and refresh the browser; the due date for each task element should be converted and look like this image in the console:



Now, when a task element is sent into the `auditTask()` function, we can get the date information and parse it into a Moment object using Moment.js. We use jQuery to select the `taskEl` element and find the `<span>` element inside it, then retrieve the text value using `.text()`. We chained on a JavaScript (not jQuery) `.trim()` method to cut off any possible leading or trailing empty spaces.

HIDE PRO TIP

Use the `.trim()` method when reading form field values to ensure no unnecessary empty spaces are in the beginning or end of the string!

Once we get that date information and store it in the `date` variable, we have to pass that value into the `moment()` function to turn it into a Moment object. The nice thing about using JavaScript methods is that we can make sense of what's happening one step at a time.

First, we use the date variable we created from taskEl to make a new Moment object, configured for the user's local time using `moment(date, "L")`. Because the date variable does not specify a time of day (for example, "11/23/2019"), the Moment object will default to 12:00am. Because work usually doesn't end at 12:00am, we convert it to 5:00pm of that day instead, using the `.set("hour", 17)` method. In this case, the value `17` is in 24-hour time, so it's 5:00pm.

Check If the Date Is in the Past

So we have the due date in a format that we can compare to the current date. Now we just need to check if that date is either in the past or imminent, and we'll add a Bootstrap background color class to the task list item if either is true. Let's start by checking if the date has passed.

Update `auditTask()` to look like this:

```
var auditTask = function(taskEl) {
  // get date from task element
  var date = $(taskEl).find("span").text().trim();

  // convert to moment object at 5:00pm
  var time = moment(date, "L").set("hour", 17);

  // remove any old classes from element
  $(taskEl).removeClass("list-group-item-warning list-group-item-dange

  // apply new class if task is near/over due date
  if (moment().isAfter(time)) {
```

```
    $(taskEl).addClass("list-group-item-danger");
  }
};
```

This functionality may be harder to test, as it is checking if the due date has passed today's date and we've explicitly set the jQuery UI Datepicker to not allow past dates to be selected. But we can always test it by removing the `minDate` setting from the `.datepicker()` methods we've used in the application; then we can put that setting back in after we test it.

First we utilize the jQuery `.removeClass()` method to remove any of these classes if they were already in place. This way, if we update the due date from yesterday to a week from now, that red background will be removed, as it will no longer be overdue.

The `moment().isAfter(time)` code inside the `if` statement is known as a **query method** in **Moment.js's documentation. (https://momentjs.com/docs/#/query/is-after/)** This means that we can perform simple true or false checks on the date for more information about it.

We're using `.isAfter()`, which, when we read it left to right, gets the current time from `moment()` and checks if that value comes later than the value of the `time` variable.

Here, we're checking if the current date and time are later than the date and time we pulled from `taskEl`. If so, the date and time from `taskEl` are in the past, and we add the `list-group-item-danger` Bootstrap class to the entire task element. This will give it a red background, to let users know the date has passed.

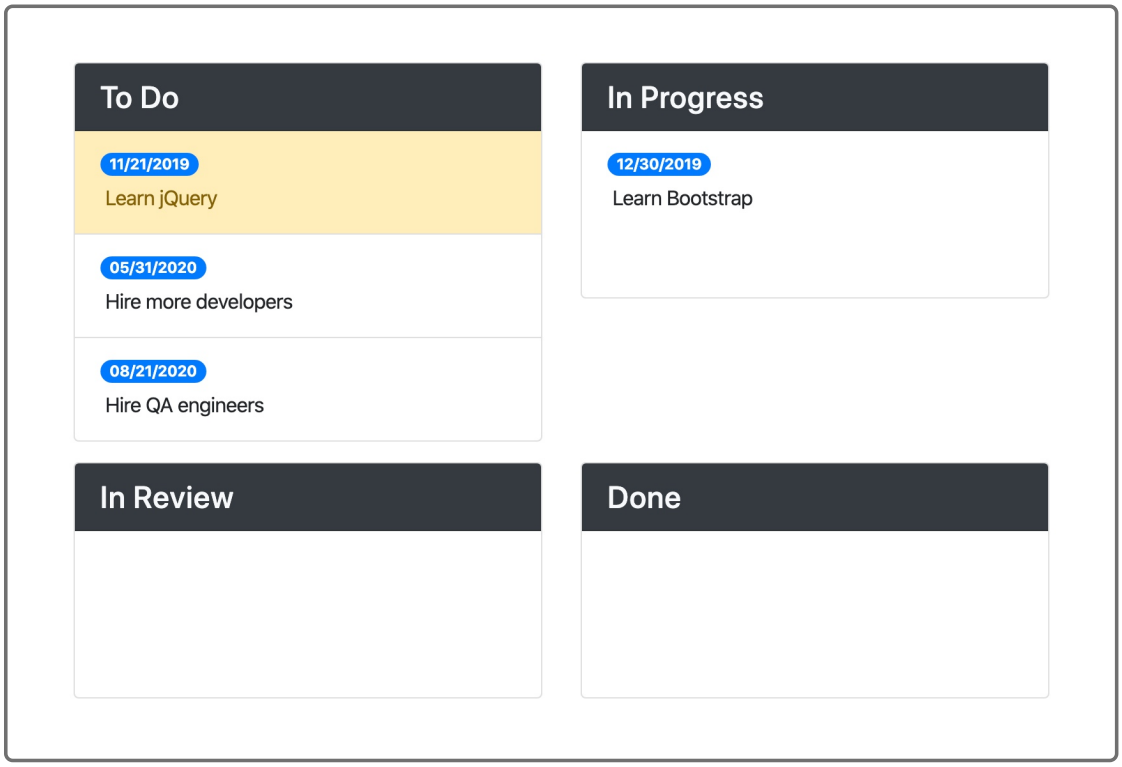
Now that we've handled how to check overdue dates, let's identify upcoming dates.

Check If the Date Is Imminent

To see how many days away the due date is, we'll have to use another Moment.js method called `.diff()`. Let's add an `else if` statement to the `auditTasks()` function so that it looks like this:

```
// apply new class if task is near/over due date
if (moment().isAfter(time)) {
  $(taskEl).addClass("list-group-item-danger");
}
else if (Math.abs(moment().diff(time, "days")) <= 2) {
  $(taskEl).addClass("list-group-item-warning");
}
```

If we save `script.js` and try making a new task due one day from now, we'll see that the task gets a yellow background! It should look something like this image:



Now a user can get a better sense of which tasks need to be prioritized because they are due sooner.

Let's dive into the condition for the `else if` statement. These Moment.js functions literally perform left to right. So when we use `moment()` to get right now and use `.diff()` afterwards to get the difference of right now to a day in the future, we'll get a negative number back. This can be hard to work with, because we have to check if the difference is >= -2, which can be hard to conceptualize.

> **NERD NOTE**
>
> Think about how people say "t minus ten seconds" during a countdown. They are counting down to a target time in the future. If we compare right now to a future time or date, we're counting the time before that target date, so we're in the negatives.

In our code, we're preventing any confusion by testing for a number less than +2, not a number greater than -2. To do this, we've wrapped the returning value of the `moment.diff()` in the JavaScript Math object's `.abs()` method. This ensures we're getting the absolute value of that number.

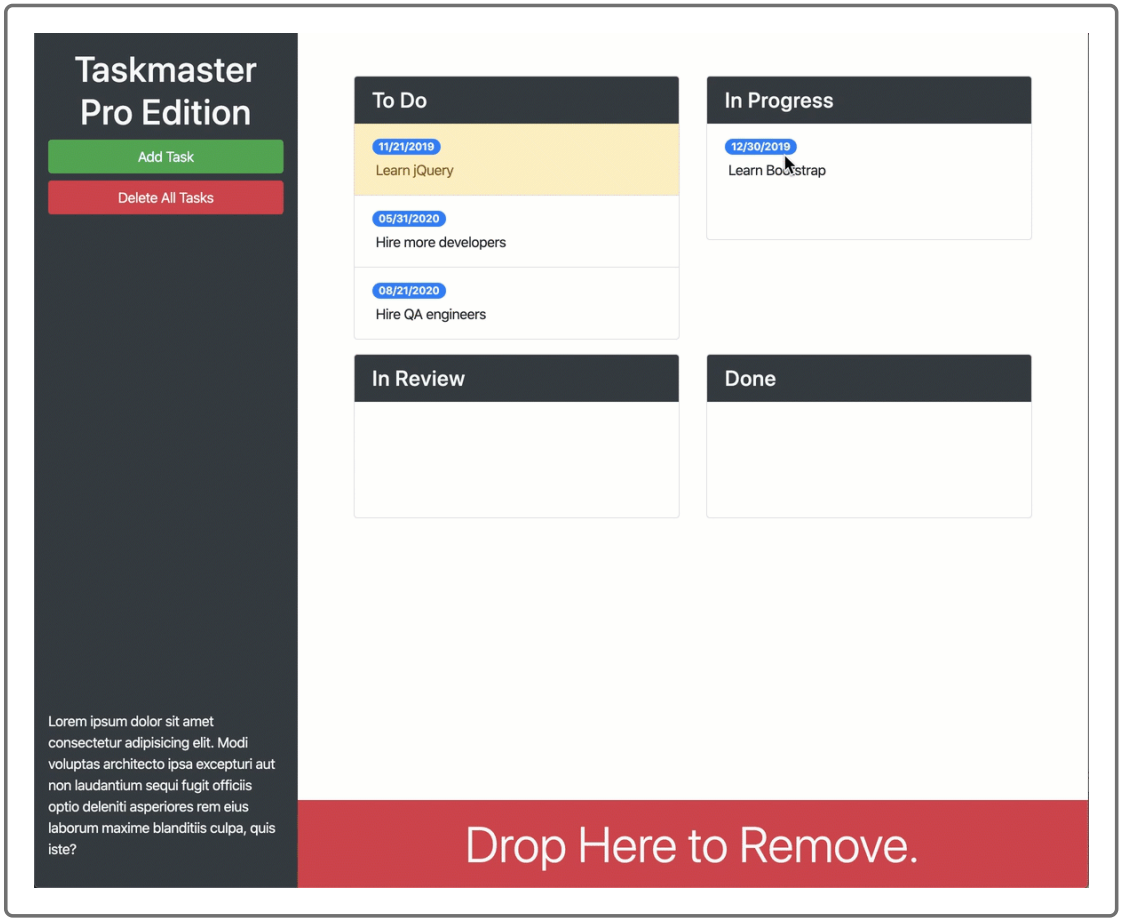
That's it for the `auditTask()` function! Now we just need to execute it when we've finished editing a due date.

Add Audit Functionality to Edit Due Dates

Let's finish up and update the due date `change` event handler function to add one more line of code to the very bottom:

```
$(".list-group").on("change", "input[type='text']", function() {
  var date = $(this).val();

  var status = $(this).closest(".list-group").attr("id").replace("list
  var index = $(this).closest(".list-group-item").index();

  tasks[status][index].date = date;
  saveTasks();

  var taskSpan = $("<span>").addClass("badge badge-primary badge-pill"
  $(this).replaceWith(taskSpan);

  // Pass task's <li> element into auditTask() to check new due date
  auditTask($(taskSpan).closest(".list-group-item"));
});
```

Let's save `script.js` and try editing a task's due date to be a day from now. When we complete updating the date, we should see something like this animation:



So now we can audit tasks when they are both created and edited, giving the user a much stronger sense of urgency for some of these tasks. The best part is that all of these tasks will be audited over again every time we refresh the page, so that we can get some visual feedback when the date draws near.

But what happens when a user just leaves the browser tab open and never refreshes it? This is a common practice for people who work on their computers all day. They leave certain tabs—like email and task lists—open indefinitely. We'll have to come up with a solution that lets the program audit tasks on its own.