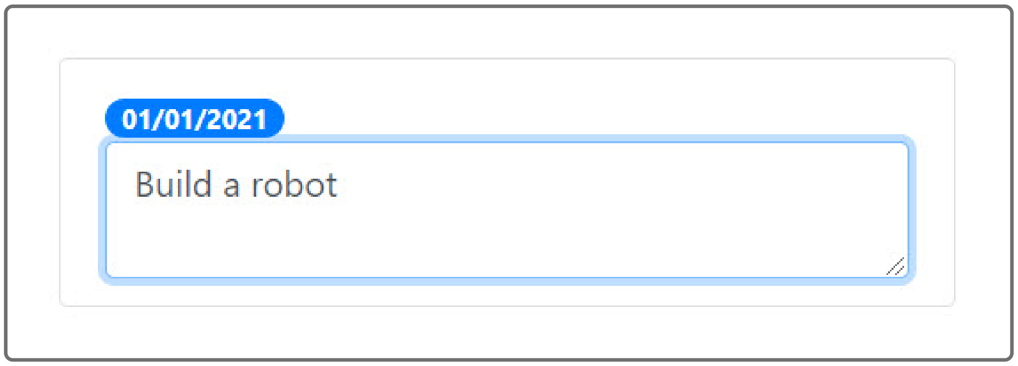
5.1.6 Add Ability to Edit Task Description

The client wants to click on a task's description to edit it inline. To accomplish this, we can swap out the `<p>` element with a form element when the task is being edited. The following screenshot demonstrates as much:



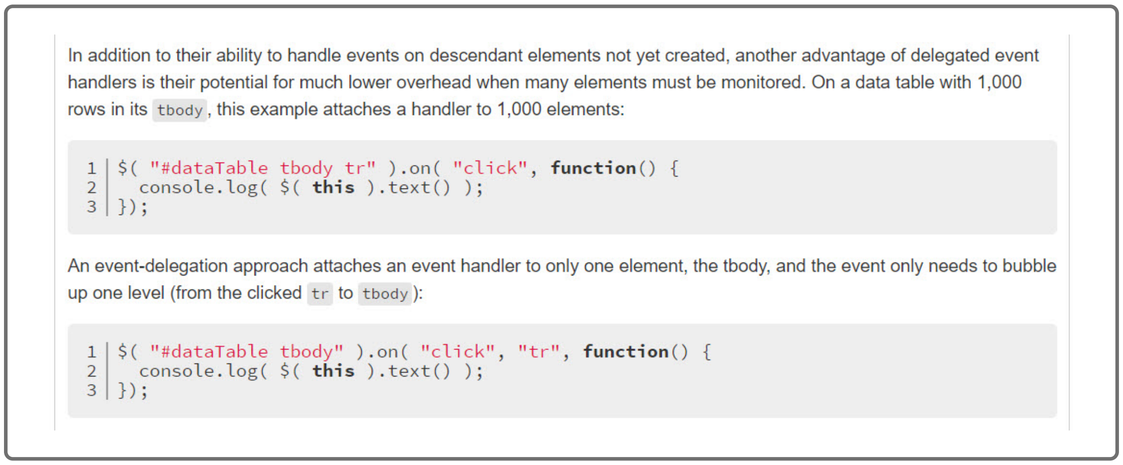
The jQuery `click()` method would only work for elements that exist on the page at the time `click()` is used. Tasks are dynamically created elements, meaning we'd need to use event delegation to account for them.

REWIND

Event delegation refers to offsetting the click event to a parent that will always exist, then checking which child element triggered the event. A plain JavaScript solution would look something like this:

```
document.querySelector("#wrapper").addEventListener("click"
 if (event.target.matches(".task")) {
   console.log("dynamic task was clicked");
 }
});
```

There's a shorthand jQuery method for event delegation using the `on()` method. One section in the **documentation for on() (https://api.jquery.com/on/#direct-and-delegated-events)** addresses handling delegated clicks versus direct clicks. Note the examples they include in the following image:


In addition to their ability to handle events on descendant elements not yet created, another advantage of delegated event handlers is their potential for much lower overhead when many elements must be monitored. On a data table with 1,000 rows in its `tbody`, this example attaches a handler to 1,000 elements:

```
$( "#dataTable tbody tr" ).on( "click", function() {
  console.log( $( this ).text() );
});
```

An event-delegation approach attaches an event handler to only one element, the tbody, and the event only needs to bubble up one level (from the clicked `tr` to `tbody`):

```
$( "#dataTable tbody" ).on( "click", "tr", function() {
  console.log( $( this ).text() );
});
```

In jQuery's example, click event listeners on all `<tr>` elements are delegated to a parent `<tbody>` element—hence the extra argument in the `on()` method: `$("#dataTable tbody").on("click", "tr")`. We'll do something similar with the `<p>` elements, delegating clicks to the parent `<ul>` with class `list-group`.

Add the following event and callback function to `script.js` right after the `saveTasks()` function:

```
$(".list-group").on("click", "p", function() {
  console.log("<p> was clicked");
});
```

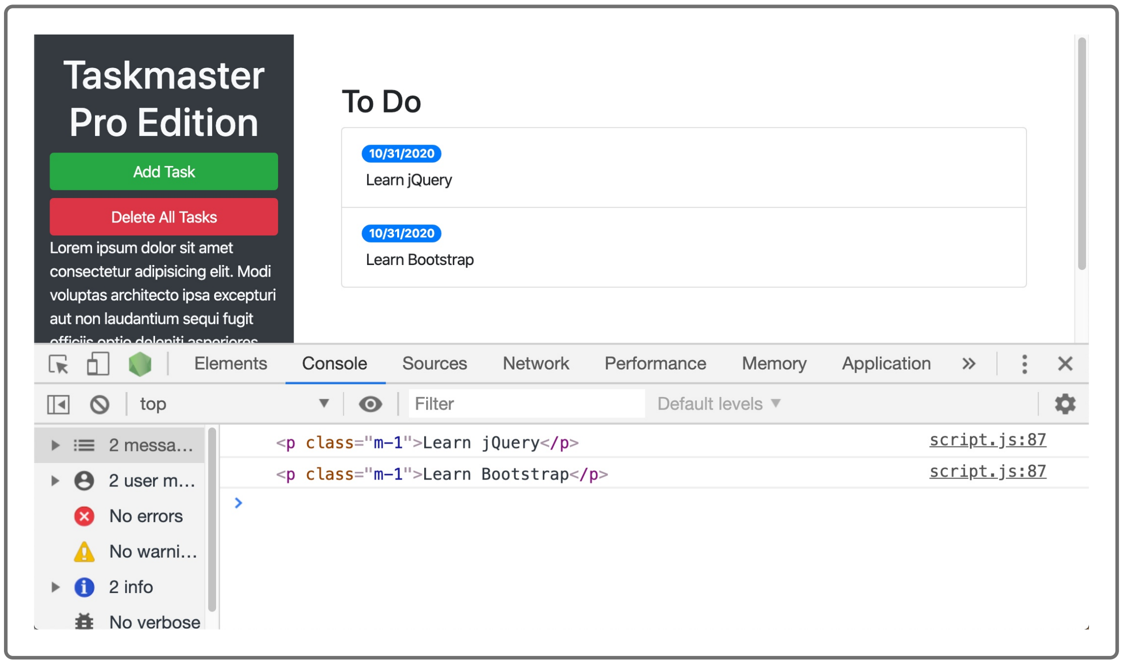
Save the file and test the app in the browser by creating a few tasks and then clicking on the task text in the browser. The task text will be inside the `<p>` elements we're targeting in our click listener.

In previous projects, we used `event.target` to get the affected element, and that'd certainly still work here. Another common trick (particularly when using jQuery) is to use the `this` keyword. We've used `this` in relation to objects to refer to themselves; DOM elements are objects too, so there's no reason why we can't use `this` in event callbacks.

Update the `on()` callback to look like this instead:

```
$(".list-group").on("click", "p", function() {
  console.log(this);
});
```

Save and test again, noting that `this` refers to the actual elements now, as seen in the image below:



```
<p class="m-1">Learn jQuery</p>        script.js:87
<p class="m-1">Learn Bootstrap</p>     script.js:87
```

Keep in mind that `this` is native to JavaScript; no jQuery magic here. With the `$` character, however, we can convert `this` into a jQuery object.

Try the following code:

```
$(".list-group").on("click", "p", function() {
  var text = $(this).text();
  console.log(text);
});
```

The `text()` method will get the inner text content of the current element, represented by `$(this)`. The `text()` method often works with the `trim()` method to remove any extra white space before or after. Many jQuery methods can be chained together, such as `text().trim()`.

Thus, we can update the function to look like this:

```
$(".list-group").on("click", "p", function() {
  var text = $(this)
    .text()
    .trim();
});
```

HIDE PRO TIP

JavaScript can recognize chained methods whether or not they're on the same line. The following examples both work:

```
var text = $(this).text().trim();

var text = $(this)
 .text()
 .trim();
```

However, putting methods on a new line can improve readability.

Like plain JavaScript, jQuery allows us to create dynamic elements. Add the following code after the `text` declaration:

```
var textInput = $("<textarea>")
  .addClass("form-control")
  .val(text);
```

While `$("textarea")` tells jQuery to find all existing `<textarea>` elements, as we've seen before, `$("<textarea>")` tells jQuery to create a new `<textarea>` element. The former uses the element name as a selector, the latter uses the HTML syntax for an opening tag to indicate the element to be created.

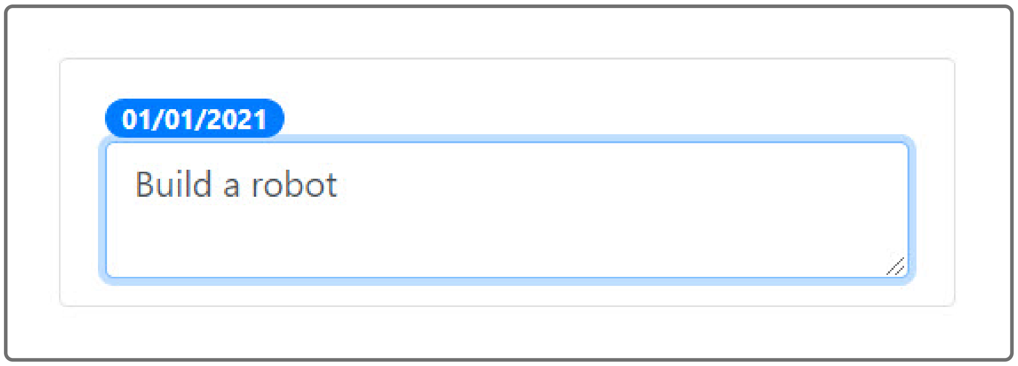
This `<textarea>` element, saved in the variable `textInput`, only exists in memory, though. We still need to append it somewhere on the page.

Specifically, we want to swap out the existing `<p>` element with the new `<textarea>`. Fortunately, there's a method for that.

Add the following line after the `textInput` declaration:

```
$(this).replaceWith(textInput);
```

Save and test in the browser. Clicking on a `<p>` element should turn it into a `<textarea>`, as demonstrated in the image below:



Note that the user still has to click on the `<textarea>` to begin editing. A better experience would be to automatically highlight the input box for them. A highlighted element is considered in **focus**, an event that can also be triggered programmatically.

After the `replaceWith()` call, add the following line:

```
textInput.trigger("focus");
```

At this point, we have the first half of the editing process complete. Now we need to update and save the task. But without a Save button, how do we know when a user is done editing? The client has requested that the `<textarea>` revert back when it goes out of focus, so we can use that event in lieu of a "Save" button.

Below the delegated `<p>` click, add a new event listener:

```
$(".list-group").on("blur", "textarea", function() {

});
```

This blur event will trigger as soon as the user interacts with anything other than the `<textarea>` element. When that happens, we need to collect a few pieces of data: the current value of the element, the parent element's ID, and the element's position in the list. These data points will help us update the correct task in the `tasks` object.

In the blur callback, add the following variable declarations:

```
// get the textarea's current value/text
var text = $(this)
  .val()
  .trim();

// get the parent ul's id attribute
var status = $(this)
  .closest(".list-group")
  .attr("id")
  .replace("list-", "");

// get the task's position in the list of other li elements
var index = $(this)
  .closest(".list-group-item")
  .index();
```

DEEP DIVE ▲

DEEP DIVE

Remember, the jQuery documentation will clarify any methods that don't immediately make sense. For instance, the **documentation for the index() method (https://api.jquery.com/index/)** points out that child elements are indexed starting at zero. Hey, that's just like arrays!

Note that `.replace()`, in the status variable initialization, isn't actually a jQuery method. It's a regular JavaScript operator to find and replace text in a string. We can, in fact, chain jQuery and JavaScript operators together in our operations, which is great.

Here, we're chaining it to `attr()`, which is returning the ID, which will be "list-" followed by the category. We're then chaining that to `.replace()` to remove "list-" from the text, which will give us the category name (e.g., "toDo") that will match one of the arrays on the tasks object (e.g., `tasks.toDo`).

If the user edited the first to-do in the list, these three variables could look something like this:

```
text = "Walk the dog";
status = "toDo";
index = 0;
```

We'd want to update the overarching `tasks` object with the new data. If we knew the values ahead of time, we could write the following:

```
tasks.toDo[0].text = "Walk the dog";
```

Because we don't know the values, we'll have to use the variable names as placeholders. Underneath the three variables, add the following lines:

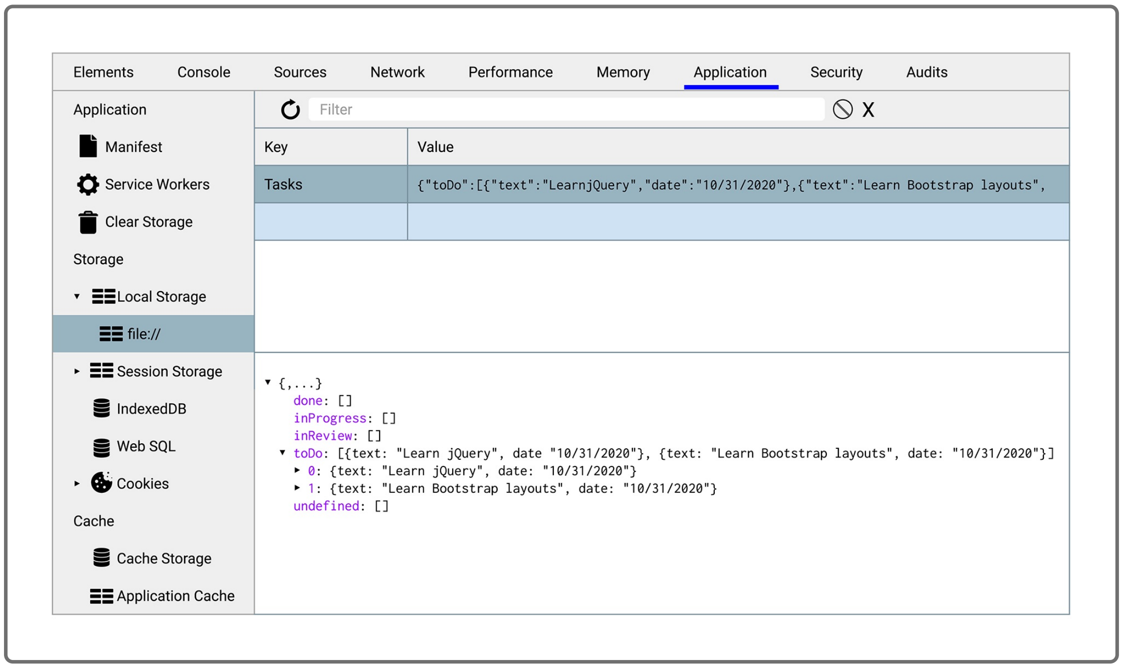
```
tasks[status][index].text = text;
saveTasks();
```

Let's digest this one step at a time:

- `tasks` is an object.
- `tasks[status]` returns an array (e.g., `toDo`).
- `tasks[status][index]` returns the object at the given index in the array.
- `tasks[status][index].text` returns the text property of the object at the given index.

Updating this `tasks` object was necessary for `localStorage`, so we call `saveTasks()` immediately afterwards.

Save and test the app again, checking the DevTools Application tab for correct updates to `localStorage`, as seen in the following image:



Key	Value
Tasks	{"toDo":[{"text":"LearnjQuery","date":"10/31/2020"},{"text":"Learn Bootstrap layouts",

```
▾ {,...}
    done: []
    inProgress: []
    inReview: []
  ▾ toDo: [{text: "Learn jQuery", date "10/31/2020"}, {text: "Learn Bootstrap layouts", date: "10/31/2020"}]
    ▸ 0: {text: "Learn jQuery", date: "10/31/2020"}
    ▸ 1: {text: "Learn Bootstrap layouts", date: "10/31/2020"}
    undefined: []
```

Now we just need to convert the `<textarea>` back into a `<p>` element. At the bottom of the blur callback function, add the following block of code:

```
// recreate p element
var taskP = $("<p>")
  .addClass("m-1")
  .text(text);

// replace textarea with p element
$(this).replaceWith(taskP);
```

Congratulations, you've just finished your first major feature using a brand-new library! With this knowledge, you can tackle the next step: editing the due dates.